Padrões de Projeto: Mediator, Observer, State, Template Method e Interpreter

Elias B. C. Carneiro[1], **Matteus Colins**[1], **Rayanne S. de Oliveira** [1], **Regivaldo Carvalho**[1]

[1]Universidade Federal do Maranhão (UFMA)
65080-805 – São Luís – MA – Brasil

elias.ccarneiro@gmail.com, matteusc.moreira@gmail.com, rayanneo390@gmail.com, regivaldojr@gmail.com

Abstract. *We summary in this article 5 of the 23 design patterns proposed in 1994 by the authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. The five patterns are state, feedback, observer, mediator and interpreter. And, for each of them is presented the purpose of the standard, the class diagram, a code example and a description of its functionality. Finally, we conclude the article with our observations on each of the standards addressed.*

Resumo. *São apresentados de forma breve neste artigo 5 dos 23 padrões de projeto propostos, em 1994, pelos autores Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. Os cinco padrões abordados são: estado, gabarito, observador, mediador e interpretador. E, para cada um deles é apresentado o propósito do padrão, o diagrama de classe, um exemplo de código e uma descrição de sua funcionalidade. Por fim, encerramos o artigo com nossas observações sobre cada um dos padrões abordados.*

1. Introdução

Design patterns, padrão de projeto criado por: Erich Gamma, John Vlissides, Ralph Jonhson e Richard Helm,conhecidos como Gang of Four(GoF). Objetivo do Gof, foi documentar soluções obtidas por experiências vivenciadas nos projetos de c++ e SmallTalk [1], que deu origem ao livro chamado Design Patterns: Elements of Reusable Object-Oriented Software (Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos) em 1994 [2].

O livro descreve 23 padrões de design para orientação a objeto. Esses padrões são necessários para solucionar dificuldade como: falha na manutenção dos códigos com grande número de ifs no mesmo método, sobrecarga de classes e problemas em engenharia de software [3].

Os padrões de projeto são classificados em 3 categorias: Criação, responsável, pela criação de novos objetos do qual pode ser desacoplado do sistema de implementação, destacando-se(abstract factory, builder, factory method, prototype e singleton). Estrutura, responsável pela forma de como as classes e os objetos lidam com conceitos de herança, polimorfismo usado para compor a interface, para obter novas funcionalidades, destacando-se(adapter, bridge,composite, decorator, facade, flyweight e proxy).

E por último comportamentais, que são responsáveis por lidarem com o processo de comunicação, gerenciamento de relações, e responsabilidades entre objetos, destacando-se (chain of responsibility, command, interpreter,iterator, mediator, memento, observer, state, strategy, template method e visitor)[4].

Os padrões destacados são os que 5 dos comportamentais, interpreter, responsável pela criação de código, onde o próprio será responsável por processar e interpretar o parâmetro,que foi lhe passado. Mediator, utilizado na abstração de comunicação, evitando que objetos se comuniquem diretamente, passando a responsabilidade para o mediador. Observer, permite a notificação de estados conforme a mudança de estado. State, permite que o objeto mude o comportamento, quando o estado muda internamente. Por último Template Method , as subclasses determinam como o irão implementar os passos de um algoritmo.

2. Padrões de Projeto

Padrões de projeto são descrições de objetos e classes comunicantes que precisam ser personalizadas para resolver um problema geral de projeto num contexto particular [1]. Qualquer definição dada a padrões de projeto relaciona o padrão como uma solução para um determinado problema em um contexto específico [2].

2.1. Mediator

O padrão Mediator consiste na criação de um objeto capaz de gerenciar a comunicação de um grupo de objetos. Defini um objeto que encapsula a forma como um conjunto de objetos interage. O Mediator promove o acoplamento fraco ao evitar que os objetos se refiram uns aos outros explicitamente e permite variar suas interações independentemente [1].

O Mediator possui três componentes principais:

- **Mediator**: uma interface para a comunicação entre o grupo de objetos;
- **ConcreteMediator**: implementa as ações da interface Mediator e coordena o comportamento do conjunto de objetos;
- **Colleague**: uma classe Colleague representa um objeto do conjunto de objetos que a classe ConcreteMediator gerencia.

O padrão Mediator possui algumas vantagens, como: aumento no reuso de código, eliminação do relacionamento de muitos para muitos, além de que as regras de comunicação entre os objetos ficam centralizada no objeto mediador. Mas uma das desvantagens é que a complexidade fica com o objeto mediador, assim se torna mais difícil a manutenção do código.

O padrão mediator deve ser usado quando [1]:

- Um conjunto de objetos se comunica de maneiras bem-definidas, porém complexas.
- As interdependências resultantes são desestruturadas e difíceis de entender.
- A reutilização de um objeto é difícil porque ele referencia e se comunica com muitos outros objetos.

2.2. Observer

A intenção do padrão Observer é notificar e atualizar automaticamente os dependentes de um determinado objeto sempre que ocorrerem alterações neste objeto. Em sistemas em que existam objetos dependentes entre si, é preciso manter a consistência entre esses objetos, ou seja, é necessário notificar qualquer mudança sofrida por um determinado objeto aos seus dependentes, sendo que, para facilitar a reutilização desses objetos, deve-se reduzir o acoplamento entre eles [1].

Deve-se utilizar o padrão Observer quando mudanças em um objeto acarretam mudanças em outros objetos ou quando for desejável diminuir o acoplamento entre objetos. A aplicação do Observer nessas situações permite a reutilização e variação de objetos de forma independente [1].

Uma das vantagens do padrão Observer é que a reutilização e alteração da implementação dos objetos observados e observadores podem ser realizadas sem afetar toda a aplicação [3].

2.3. State

O padrão State permite a um objeto alterar seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado de classe.

O padrão State de ser usado quando [1]:

- O comportamento de um objeto depende do seu estado e ele pode mudar seu comportamento em tempo de execução, dependendo desse estado;

- Operações têm comandos condicionais grandes, de várias alternativas, que dependem do estado do objeto. Este estado é normalmente representado por uma ou mais constantes enumeradas. Freqüentemente, várias operações conterão esta mesma estrutura condicionada. O padrão State coloca cada ramo do comando adicional em uma classe separada. Isto lhe permite tratar o estado do objeto como um objeto propriamente dito, que pode variar independentemente de outros objetos.

2.4. Template Method

O propósito do padrão Template Method consiste em determinar o esqueleto de um algoritmo, de forma que alguns passos desse algoritmo possam ser redefinidos na subclasse sem ter a sua estrutura alterada [1]. O Template Method define um algoritmo que utiliza operações abstratas que são sobrepostas nas subclasses, oferecendo um comportamento concreto [3].

O padrão Template Method deve ser utilizado quando for desejável delegar a implementação dos comportamentos variantes de uma classe às suas subclasses [3]. O Template Method pode ser utilizado quando subclasses diferentes possuem comportamentos comuns. Nesse caso eles devem ser concentrados em uma única classe evitando a duplicação de código [1].

Uma das vantagens do Template Method é que ele oferece uma base fundamental para reuso de código, já que permite a realizar refatoração de ações comuns entre classes.

2.5. Interpreter

Este padrão descreve como definir uma gramática para linguagens simples, representar e interpretar sentenças. O propósito deste padrão é que dada uma linguagem, define uma representação para a sua gramática juntamente com um interpretador que usa representação para interpretar as sentenças da linguagem [1].

No Interpreter a idéia de que há vários tipos de composição é essencial [1] e pode ser utilizado para representar e resolver problemas recorrentes que possam ser expressos sob a forma de uma linguagem formal simples. O padrão usa classes para representar cada regra de uma gramática (expressão regular) [3].

Esse padrão é melhor aplicado em programas de sintaxes simples, pois ele fornece uma flexibilidade de adicionar e remover regras de sintaxes e múltiplas implementações de uma declaração.

3. Metodologia

Este trabalho refere-se a uma pesquisa sobre padrões de projeto Gof. A seguir, descrevem-se as etapas para o desenvolvimento do artigo.

- Primeiro foram realizadas pesquisas bibliográficas para coletar e organizar informações sobre design patterns e seus padrões de projeto:interpreter, mediator, observer, state,e template method;
- Na segunda etapa, foi realizado o desenvolvimento da modelagem dos padrões de projeto utilizando-se os padrões da UML 2.3, que permitiu visualizar das funções nos diagramas de sequência e classe ;
- Na terceira etapa foi realizada a codificação utilizando os padrões de projeto, na linguagem java 8.0.

4. Resultados

4.1. Mediator

¡¡ mediator ¿¿

Diagrama de Classes - Mediator:

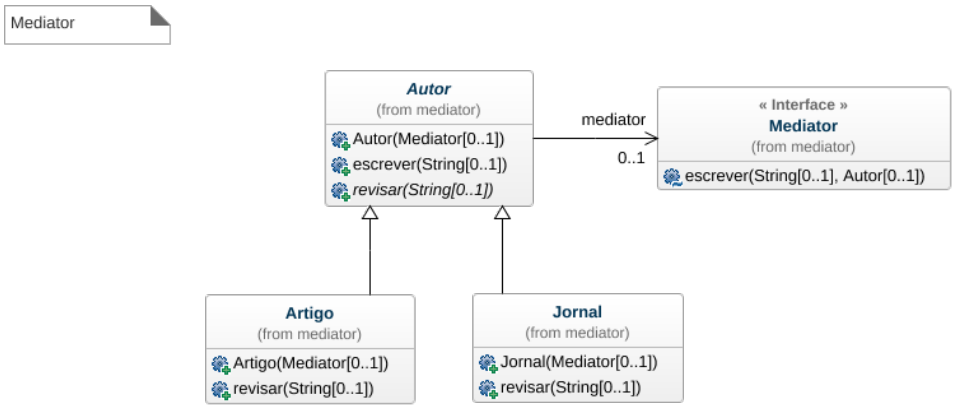

Figure 1. Diagrama de Classes - Mediator (Fonte: os autores)

Diagrama de Sequência - Mediator:

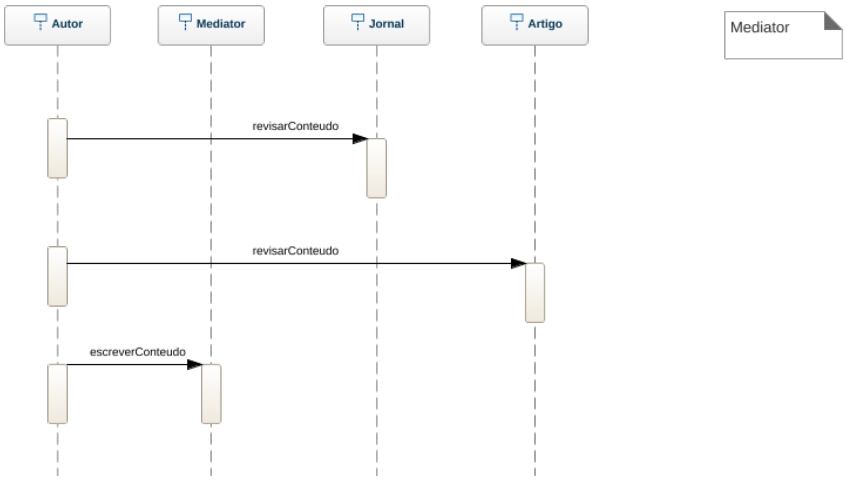

Figure 2. Diagrama de Classes - Mediator (Fonte: os autores)

Exemplo de Código - Mediator:

```
1 package mediator;
2
3 public interface Mediator {
4   void escrever(String conteudo, Autor autor);
5 }
```

Listing 1. Interface Mediator

```java
package mediator;

public class Jornal extends Autor{
  public Jornal(Mediator m) {
    super(m);
  }

  @Override
  public void revisar(String conteudo) {
    System.out.println("Conteudo Revisado" + conteudo);
  }
}
```

Listing 2. Classe Jornal

```java
package mediator;

public abstract class Autor {
  protected Mediator mediator;

  public Autor(Mediator m) {
    mediator = m;
  }

  public void escrever(String conteudo) {
    mediator.escrever(conteudo, this);
  }

  public abstract void revisar(String conteudo);
}
```

Listing 3. Classe Autor

```java
package mediator;

public class Artigo extends Autor{

  public Artigo(Mediator m) {
    super(m);
  }

  @Override
  public void revisar(String conteudo) {
    System.out.println("Conteudo Revisado" + conteudo);
  }
}
```

Listing 4. Classe Artigo

4.2. Observer

Diagrama de Classes - Observer:

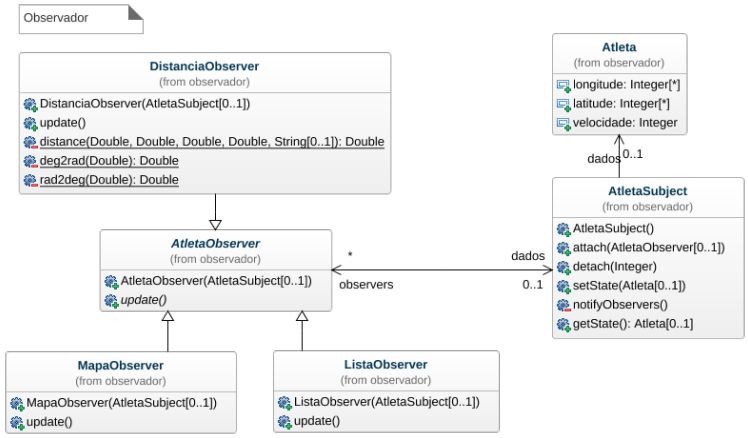

Figure 3. Diagrama de Classes - Observer (Fonte: os autores)

Diagrama de Sequência - Observer:

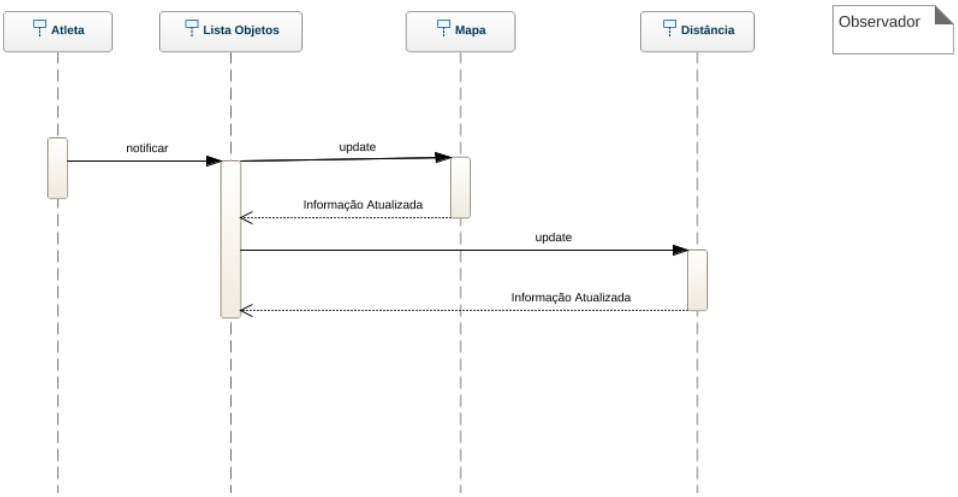

Figure 4. Diagrama de Classes - Observer (Fonte: os autores)

Exemplo de Código - Observer:

```
1  package observador;
2
3  import java.util.ArrayList;
4
5  public class Atleta {
6    public ArrayList<Integer> longitude;
7    public ArrayList<Integer> latitude;
8    public int velocidade;
```

```
}
```

Listing 5. Classe Atleta

```
package observador;

public abstract class AtletaObserver {
  protected AtletaSubject dados;

  public AtletaObserver(AtletaSubject dados) {
        this.dados = dados;
    }

    public abstract void update();
}
```

Listing 6. Classe AtletaObserver

```
package observador;

import java.util.ArrayList;

public class AtletaSubject {
  protected ArrayList<AtletaObserver> observers;
    protected Atleta dados;

  public AtletaSubject() {
        observers = new ArrayList<AtletaObserver>();
    }

    public void attach(AtletaObserver observer) {
        observers.add(observer);
    }

    public void detach(int indice) {
        observers.remove(indice);
    }

    public void setState(Atleta dados) {
        this.dados = dados;
        notifyObservers();
    }

    private void notifyObservers() {
        for (AtletaObserver observer : observers) {
            observer.update();
        }
    }

    public Atleta getState() {
        return dados;
```

```
    }
}
```

Listing 7. Classe AtletaSubject

```
package observador;

public class DistanciaObserver extends AtletaObserver {

    public DistanciaObserver(AtletaSubject dados) {
        super(dados);
    }

    @Override
    public void update() {
      double distance = this.distance(dados.getState().longitude
         .get(0), dados.getState().longitude.get(1),
          dados.getState().latitude.get(0), dados.getState().
             longitude.get(1), "K");

        System.out.println("Distancia Percorrida: " + distance )
           ;
    }

    private static double distance(double lat1, double lon1,
       double lat2, double lon2, String unit) {
        double theta = lon1 - lon2;
        double dist = Math.sin(deg2rad(lat1)) * Math.sin(deg2rad
           (lat2)) + Math.cos(deg2rad(lat1)) * Math.cos(deg2rad(
           lat2)) * Math.cos(deg2rad(theta));
        dist = Math.acos(dist);
        dist = rad2deg(dist);
        dist = dist * 60 * 1.1515;
        if (unit == "K") {
            dist = dist * 1.609344;
        } else if (unit == "N") {
            dist = dist * 0.8684;
        }

        return (dist);
    }

    //This function converts decimal degrees to radians
    private static double deg2rad(double deg) {
        return (deg * Math.PI / 180.0);
    }

    //This function converts radians to decimal degrees
                                :*/
```

```
39      private static double rad2deg(double rad) {
40          return (rad * 180 / Math.PI);
41      }
42
43  }
```

Listing 8. Classe DistanciaObserver

```
1
2   package observador;
3
4   public class ListaObserver extends AtletaObserver{
5
6     public ListaObserver(AtletaSubject dados) {
7       super(dados);
8     }
9
10    @Override
11      public void update() {
12          System.out.println("Lista:\n Latitude: " + dados.
               getState().latitude
13                  + "\nLongitude: " + dados.getState().longitude +
                        "\nVelocidade: "
14                  + dados.getState().velocidade);
15      }
16  }
```

Listing 9. Classe ListaObserver

```
1
2   package observador;
3
4   public class MapaObserver extends AtletaObserver {
5
6       public MapaObserver(AtletaSubject dados) {
7           super(dados);
8       }
9
10      @Override
11      public void update() {
12        System.out.println("Pontos para o mapa:\nLatitude: " +
             dados.getState().latitude
13                  + "%\nLongitude: " + dados.getState().latitude);
14      }
15  }
```

Listing 10. Classe MapaObserver

4.3. State

¡¡ state ¿¿

Diagrama de Classes - State:

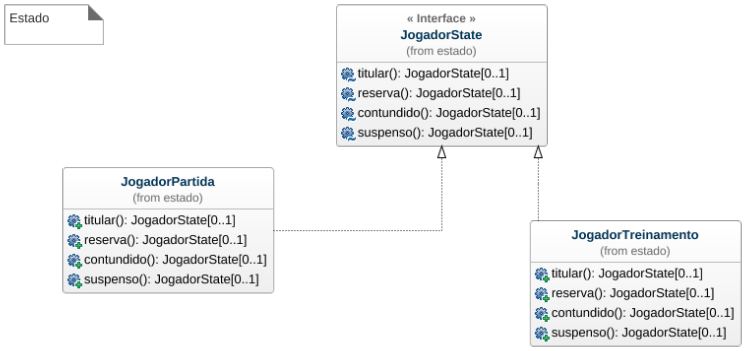

Figure 5. Diagrama de Classes - State (Fonte: os autores)

Diagrama de Sequência - State:

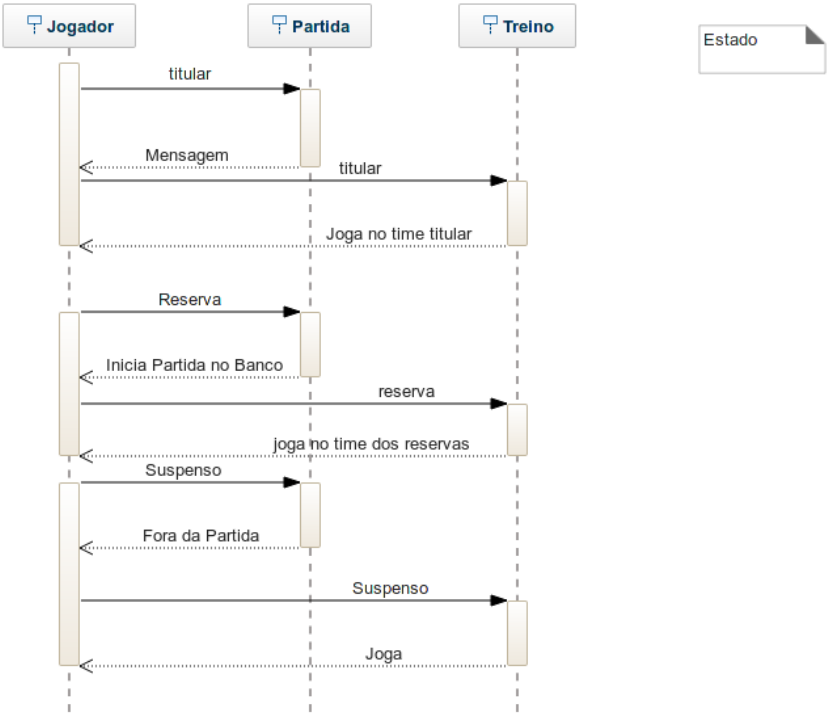

Figure 6. Diagrama de Sequência - State (Fonte: os autores)

Exemplo de Código - State:

```
1
2  package state;
3
4  public class JogadorPartida implements JogadorState {
5
6    @Override
7    public JogadorState titular() {
8          System.out.println("Jogador inicia partida jogando");
9          return this;
10   }
11
12   @Override
13   public JogadorState reserva() {
14     System.out.println("Jogador inicia partida no banco");
15     return this;
16   }
17
18   @Override
19   public JogadorState contundido() {
20     System.out.println("Jogador permanece no departamento medico
         ");
21     return this;
22   }
23
24   @Override
25   public JogadorState suspenso() {
26     System.out.println("Jogador nao participa da partida");
27
28     return this;
29   }
30
31 }
```

Listing 11. Classe JogadorPartida

```
1
2  package state;
3
4  public interface JogadorState {
5    JogadorState titular();
6    JogadorState reserva();
7    JogadorState contundido();
8    JogadorState suspenso();
9  }
```

Listing 12. Interface JogadorState

```
1
2  package state;
3
```

```
4  public class JogadorTreinamento implements JogadorState{
5
6    @Override
7    public JogadorState titular() {
8          System.out.println("Jogador participa do treino jogando
               do lado dos titulares.");
9          return this;
10   }
11
12   @Override
13   public JogadorState reserva() {
14         System.out.println("Jogador participa do treino jogando
               do lado dos reservas.");
15         return this;
16   }
17
18   @Override
19   public JogadorState contundido() {
20         System.out.println("Jogador faz treinos isolados de
               recuperacao.");
21         return this;
22   }
23
24   @Override
25   public JogadorState suspenso() {
26         System.out.println("Jogador participa do treinamento no
               time dos reservas.");
27         return this;
28   }
29
30 }
```

Listing 13. Interface JogadorTreinamento

4.4. Template Method

¡¡ template method ¿¿

Diagrama de Classes - Template Method:

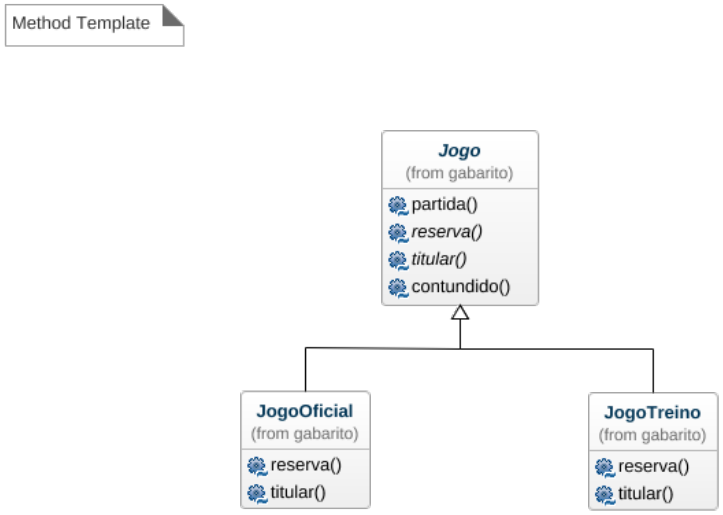

Figure 7. Diagrama de Classes - Template Method (Fonte: os autores)

Diagrama de Sequência - Template Method:

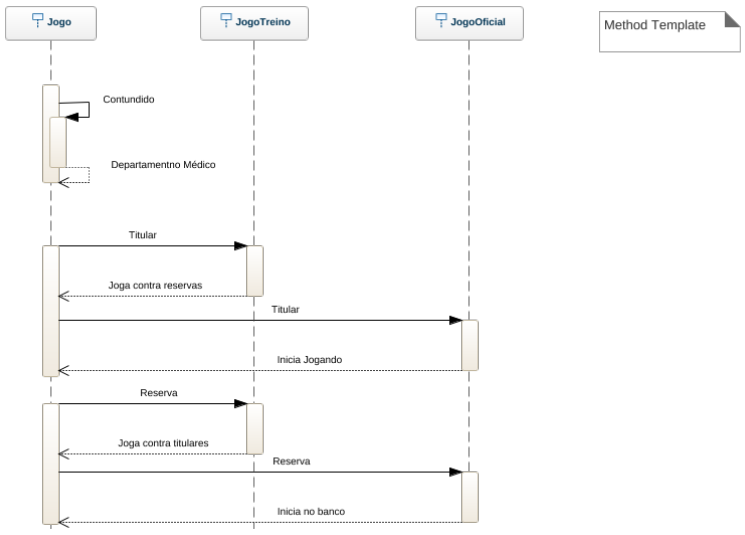

Figure 8. Diagrama de Classes - Template Method (Fonte: os autores)

Exemplo de Código - Template Method:

```java
package template;

public abstract class Jogo {
  final void partida() {
    reserva();
    titular();
  }
  abstract void reserva();
  abstract void titular();

  final void contundido() {
    System.out.println("Departamentno Medico");
  }
}
```

Listing 14. Classe Jogo

```java
package template;

public class JogoOficial extends Jogo {

  @Override
  void reserva() {
    // TODO Auto-generated method stub
    System.out.println("Iniciam a partida no banco de reservas."
        );
  }

  @Override
  void titular() {
    // TODO Auto-generated method stub
    System.out.println("Jogo intenso.");
  }

}
```

Listing 15. Classe JogoOficial

```java
package template;

public class JogoTreino extends Jogo {

  @Override
  void reserva() {
    // TODO Auto-generated method stub
    System.out.println("Jogam contra os times titulares e
       ajustes tecnicos.");
  }

  @Override
```

```
12    void titular() {
13      // TODO Auto-generated method stub
14      System.out.println("Jogo leve e ajustes tecnicos.");
15    }
16
17  }
```

Listing 16. Classe JogoTreino

4.5. Interpreter

¡¡ interpreter ¿¿

Diagrama de Classes - Interpreter:

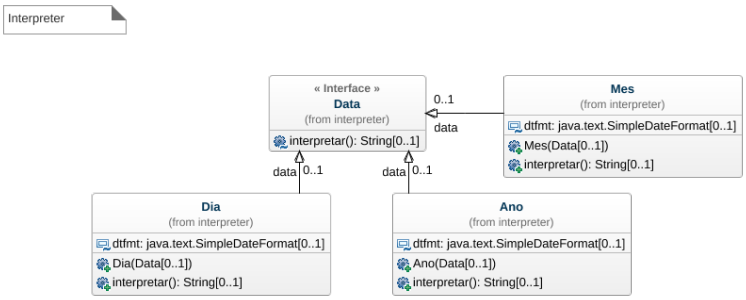

Figure 9. Diagrama de Classes - Interpreter (Fonte: os autores)

Diagrama de Sequência - Interpreter:

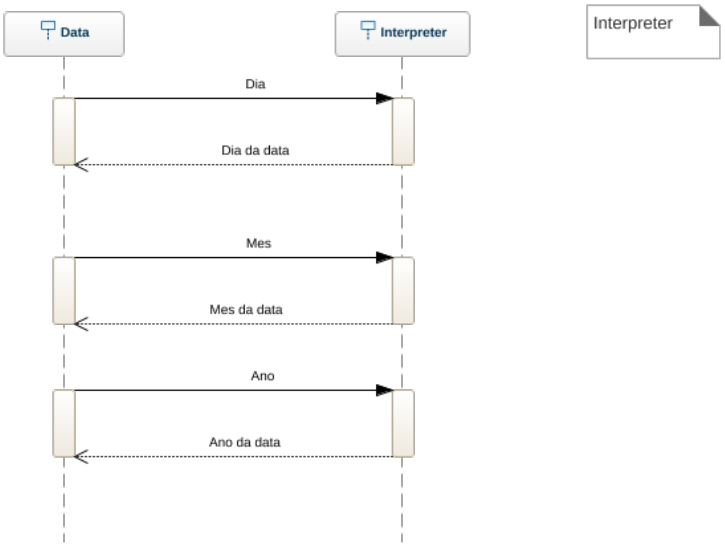

Figure 10. Diagrama de Classes - Interpreter (Fonte: os autores)

Exemplo de Código - Interpreter:

```java
package interpreter;

import java.text.SimpleDateFormat;

public class Ano implements Data{

  private Data data;
  SimpleDateFormat dtfmt = new SimpleDateFormat("YYYY");

  public Ano(Data data) {
    this.data = data;
  }

  @Override
  public String interpretar() {
    // TODO Auto-generated method stub
    return dtfmt.format(data);
  }

}
```

Listing 17. Classe Ano

```java
package interpreter;

public interface Data {
  String interpretar();
}
```

Listing 18. Interface Data

```java
package interpreter;

import java.text.SimpleDateFormat;

public class Dia implements Data {
  private Data data;
  SimpleDateFormat dtfmt = new SimpleDateFormat("dd");

  public Dia(Data data) {
    this.data = data;
  }

  @Override
  public String interpretar() {
    // TODO Auto-generated method stub
    return dtfmt.format(data);
  }
}
```

Listing 19. Classe Dia

```java
package interpreter;

import java.text.SimpleDateFormat;

public class Mes implements Data{
  private Data data;
  SimpleDateFormat dtfmt = new SimpleDateFormat("m");

  public Mes(Data data) {
    this.data = data;
  }

  @Override
  public String interpretar() {
    // TODO Auto-generated method stub
    return dtfmt.format(data);
  }
}
```

Listing 20. Classe Mes

5. Considerações Finais

As mudanças de software, podem acontecer,e possivelmente irão por vários motivos, mudança na regra de negócio, novas funcionalidades adicionadas, manutenção de sistema e etc. O problema na leitura dos códigos e a falta de padronização pode gerar um alto custo para o cliente e dificuldade dos programadores em solucionar tais problemas.

Por esse motivo, durante o desenvolvimento de um projeto de software, é necessário a utilização de meios que facilitem o processo de desenvolvimento e manutenção. O design pattern é uma ótima opção pois os padrões descrevem como utilizá los, um dos pontos a se destacar é se preocupar e entender qual dos padrões se encaixam para que os padrões não dificultem ainda mais os problemas recorrentes.

A intenção foi esclarecer quando usar os métodos e exemplificar por meio de códigos as possibilidades que os padrões podem possibilitar, assim reduzir problemas recorrente no desenvolvimento e manutenção de software.

References

[1] GAMMA, Erich et al. (2000). *Padrões de Projeto: soluções reutilizáveis de software orientado a objetos*, Porto Alegre: Bookman.

[2] ALUR, Deepak; CRUPI, John; MALKS, Dan (2004). *Core J2EE patterns: as melhores práticas e estratégias de design.*, Rio de Janeiro: Elsevier.

[3] AGORNAVIS (S/D). *Padrões Design de com aplicações em Java*. Disponível em: `http://www.inf.ufpr.br/andrey/ci163/Design_Patterns.pdf`. Acessado em: 23 de junho de 2019.

[4] DEVMEDIA (S/D). *Introdução: Design Pattern*. Disponível em: `https://www.devmedia.com.br/introducao-design-pattern/18838`. Acessado em: 23 de junho de 2019.

[5] DEVMEDIA (S/D). *Design Patterns: Padrões "GoF"*. Disponível em: https://www.devmedia.com.br/design-patterns-padroes-gof/16781. Acessado em: 23 de junho de 2019.

[6] FARIAS, Kleinner. *Padrão de Projeto Interpreter*. Disponível em: http://www.wiki.les.inf.puc-rio.br/uploads/c/cc/Interpreter_20082.ppt. Acessado em: 25 de junho de 2019.

[7] NETTO, Manoel Teixeira de Abreu. *Padrão de Projeto Interpreter: Projeto de Sistemas de Software* . Disponível em: wiki.les.inf.puc-rio.br/uploads/3/36/Interpreter.pdf. Acessado em: 25 de junho de 2019.