**PRÁTICAS DE ENGENHARIA DE SOFTWARE FAPA-N1 - 202120.124440.05**

**Alunos:**

Lucas Alexsander Garcia de Souza 2020210620,
Douglas Silva da Silva 201621543,
Matheus Larentis Mallmann 201815113,
Guilherme Fleck Oliveira 2019111111,
Leonardo Oliveira de Farias 2019111904

# Padrões GoF
# (Estruturais e Comportamentais)

PORTO ALEGRE
2021

- **Abstract Factory**
    - Descrição do padrão:

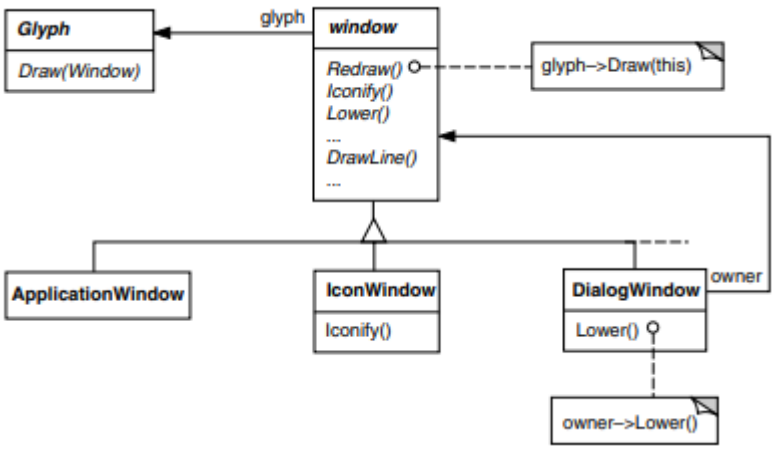
O padrão **Abstract Factory** tem por objetivo único a criação de novos objetos fornecendo uma interface para a criação de um conjunto de objetos, que estarão relacionados ou dependentes um dos outros, mas sem especificar quais serão as suas classes concretas.

    - Uso do mesmo:

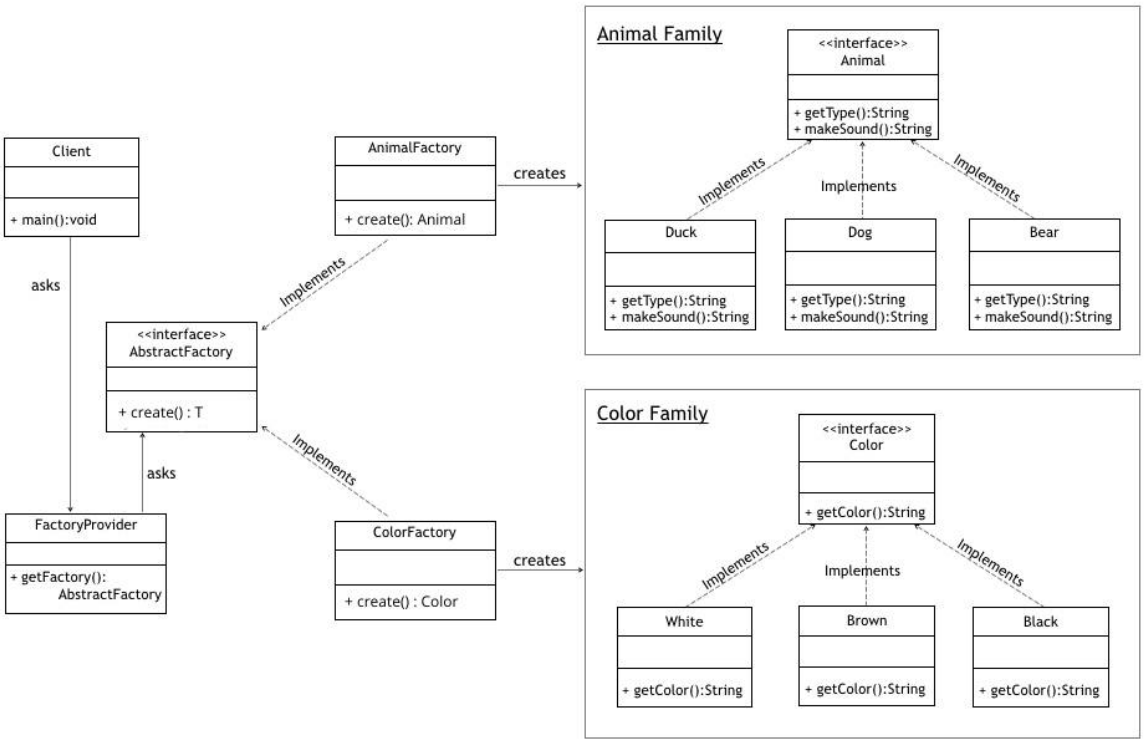
O padrão Abstract Factory pode ser utilizado quando:

1. Precisa ser ocultado informações como representação, implementação, armazenamento ou localização, evitando propagação de mudanças em cadeia;
2. Quando as interfaces externas dos sistemas operacionais e as de programação de aplicação são diferentes em diferentes plataformas, limitando as dependências de cada uma das plataformas;
3. Resolver problemas de acoplamento forte, não deixando as classes difíceis de reutilizar isoladamente, por dependerem uma das outras, tornando o acoplamento fraco aumenta-se a facilidade de uma classe ser aprendida, modificada e estendida por um sistema.

    - Modelo de Implementação:




    - Exemplos de uso:




- **Adapter (object)**

  - Descrição do padrão:

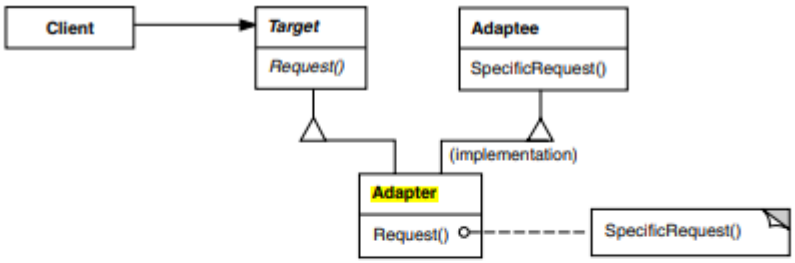
    O padrão **Adapter** pode ser visto como um padrão de interface para um objeto. Ela tem por função permitir que uma classe com interfaces incompatíveis trabalhem em conjunto e também é conhecida como Wrapper.

  - Uso do mesmo:

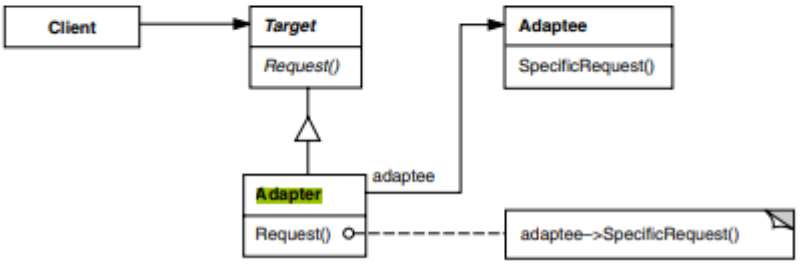
    O padrão Adapter pode ser utilizado quando:

    1. Quando precisa ser usada uma classe existente mas ela não é compatível com a interface necessária;
    2. Existem muitas subclasses e não é possível adaptar subclasses para cada uma delas, sendo mais prático a implementação de um objeto adaptador baseado na classe base;

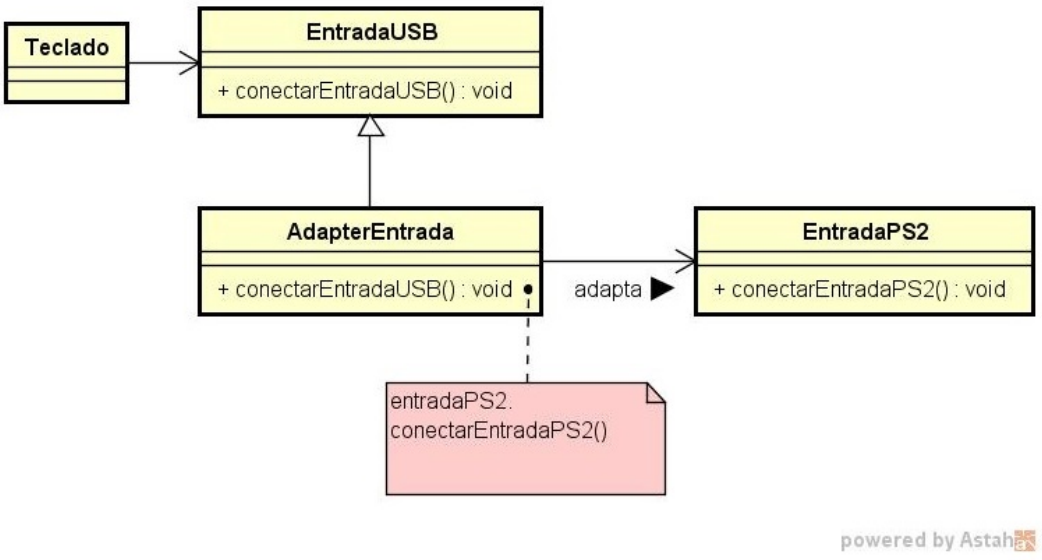
  - Modelo de Implementação:




Um adaptador de objeto depende da composição de objetos:




- Exemplos de uso:



- **Bridge**
    - Descrição do padrão:

O padrão **Bridge** tem por objetivo separar ou desacoplar a implementação de sua abstração, delegando operações da abstração para sua implementação visto ambas serem muito parecidas. Essa delegação exemplo extremo de composição de objetos mostrando como pode ser substituída a herança pela composição de objetos como mecanismo de reutilização de códigos.

    - Uso do mesmo:

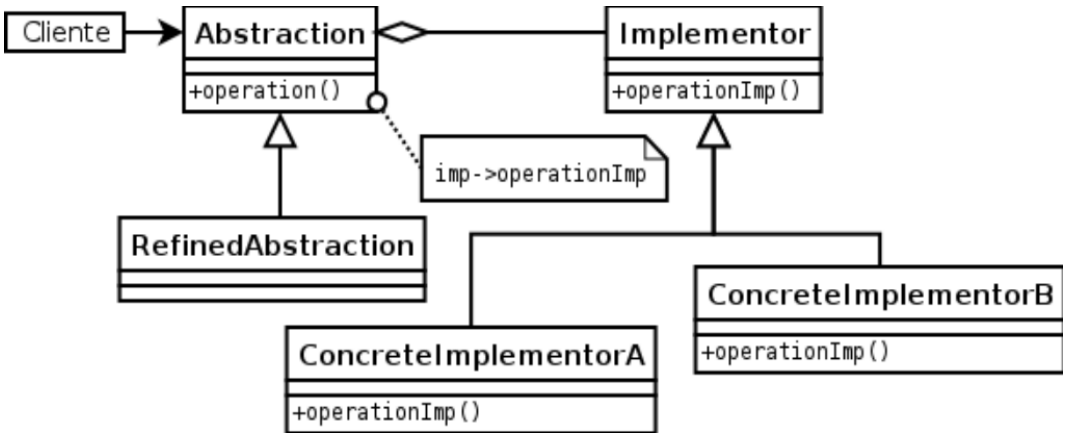
O padrão Bridge pode ser utilizado quando:

Quando uma interface possa variar independente de sua implementação;

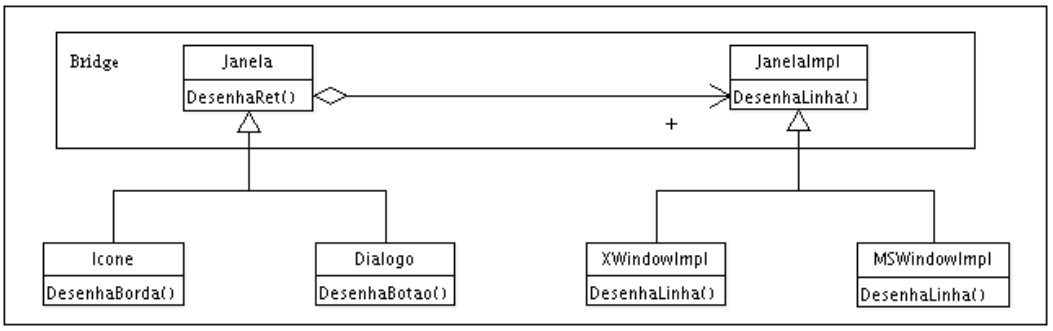
Quando deseja-se evitar conexão permanente entre implementação e abstração;

Para evitar que clientes sofram com mudanças de novas implementações ou mudanças;

    - Modelo de Implementação:



- Exemplos de uso:




- **Composite**
  - Descrição do padrão:

O padrão **Composite** permite que clientes tratem  composições de objetos e objetos individuais de maneira uniforme, estruturando objetos em forma de árvore para representar hierarquias do tipo partes-todo.

Usado para representar a estrutura física de documentos mas também a composição recursiva em termos de orientação a objetos.

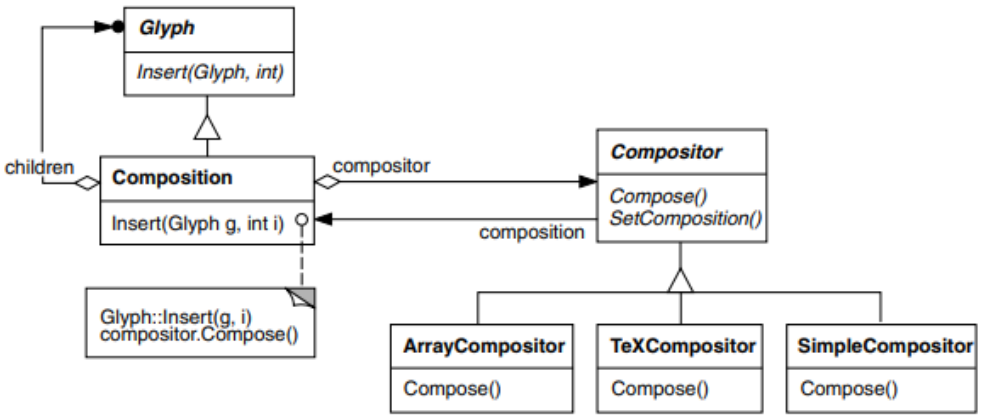
  - Uso do mesmo:

O padrão Composite pode ser utilizado quando:

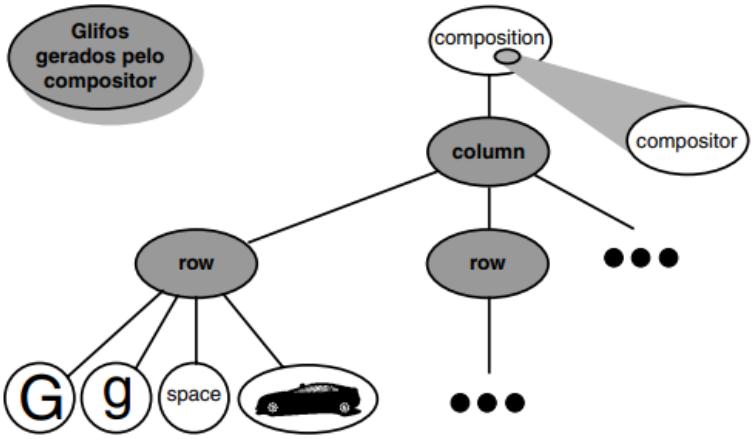
1. Usado para representar documentos;

2. Usando para representar qualquer estrutura hierárquica potencialmente complexa;

- Modelo de Implementação:




- Exemplos de uso:



- **Command**: o padrão Command possui, como definição, o objetivo de encapsular uma solicitação como um objeto, permitindo que parametrize outros objetos com diferentes solicitações, enfileirando ou registrando solicitações, e para implementar recursos de cancelamento de operações. Isto inclui informações como o nome do método, o objeto que o método pertence e os valores dos parâmetros do método.

No exemplo acima, a aplicação cria um comando concreto e configura o recebedor (Receiver) para que ele possa executá-lo. A classe Receiver, que pode ser qualquer classe na aplicação, sabe como executar o trabalho necessário. ConcreteCommand é a responsável por manter um vínculo entre uma ação e um objeto da classe Receiver. Com isso, um objeto invoca um método de execução, e o ConcreteCommand realiza ações no objeto recebedor. O papel do Command, que pode ser uma interface ou classe abstrata, é de disponibilizar uma interface comum entre todas as classes concretas que a implementam.

Aplicações: botões e itens de menu, gravações de macros, códigos mobile, networking, processamento paralelo, undo multinível, etc.

Abaixo, um exemplo de interruptor de luz em Java:

```
import java.util.List;
import java.util.ArrayList;

/** Interface do Command */
public interface Command {
  void execute();
}

/** Classe invocadora */
public class Switch {
  private List<Command> history = new ArrayList<Command>();

  public void storeAndExecute(Command cmd) {
        this.history.add(cmd); // optional
    cmd.execute();
  }
}

/** Classe recebedora */
public class Light {

  public void turnOn() {
        System.out.println("A luz está ligada");
  }

  public void turnOff() {
        System.out.println("A luz está desligada");
  }
}
```

```java
/** O Command para ligar a luz - ConcreteCommand #1 */
public class FlipUpCommand implements Command {
  private Light theLight;

  public FlipUpCommand(Light light) {
        this.theLight = light;
  }

  @Override    // Command
  public void execute() {
        theLight.turnOn();
  }
}

/** O Command para desligar a luz - ConcreteCommand #2 */
public class FlipDownCommand implements Command {
  private Light theLight;

  public FlipDownCommand(Light light) {
        this.theLight = light;
  }

  @Override    // Command
  public void execute() {
        theLight.turnOff();
  }
}

/* Classe teste */
public class PressSwitch {
  public static void main(String[] args){
        // Checa o número de argumentos
        if (args.length != 1) {
         System.err.println("Argumento \"ON\" or \"OFF\" é necessário.");
        System.exit(-1);
        }

        Light lamp = new Light();
        Command switchUp = new FlipUpCommand(lamp);
        Command switchDown = new FlipDownCommand(lamp);

        Switch mySwitch = new Switch();

        switch(args[0]) {
        case "ON":
        mySwitch.storeAndExecute(switchUp);
        break;
        case "OFF":
        mySwitch.storeAndExecute(switchDown);
        break;
```

```
        default:
          System.err.println("Argumento \"ON\" or \"OFF\" é necessário.");
        System.exit(-1);
        }
  }
}
```

- **Memento**: assim como Command, é um padrão comportamental. Ele permite armazenar o estado interno de um objeto em um determinado momento, para possibilidade de retorná-lo ao estado original, sem que isso ocasione problemas de encapsulamento.

Ele funciona de maneira que uma classe é responsável por salvar o estado do objeto desejado, enquanto uma segunda classe torna-se responsável por armazenar todas essas cópias (que são conhecidos como mementos).

Este padrão é se utiliza de três elementos na sua implantação: Originador (que é o objeto cujo estado se deseja capturar), Armazenador (responsável por armazenar o estado interno do Originador) e o Memento (que armazena todos os Mementos, e que acessará o Originador para desfazer qualquer mudança efetuada, se necessário).

O Armazenador deve requisitar um objeto memento, antes de se valer do Originador. Após efetuar as operações desejadas no Originador, o cliente devolve a este objeto Memento, caso deseje fazer qualquer alteração. O objeto Memento não permite o acesso de qualquer classe além da classe Originador. Assim, o padrão mostra-se útil por não violar o conceito de encapsulamento.

O defeito do Memento é que por ele sempre estar guardando o estado do objeto, ele pode guardar objetos de maneira desnecessária, e assim utilizando muita memória da máquina.

Abaixo, em Java, a utilização do memento para desfazer ações de mudança de estado:

```
import java.util.List;
```

```
import java.util.ArrayList;

class Originator {

      private String state;

      public void setState(String state) {

      System.out.println("Originator: Mudando estado para " + state);

        this.state = state;

      }

      public Memento saveState() {

        System.out.println("Originator: Salvando o Memento.");

        return new Memento(this.state);

      }

      public void restoreState(Memento memento) {

        this.state = memento.getSavedState();

        System.out.println("Originator: Estado após restaurar o
Memento: " + state);

      }

      public static class Memento {

        private final String state;

        public Memento(String stateToSave) {

            this.state = stateToSave;

      }

        private String getSavedState() {

            return this.state;

      }

      }

}

class Armazenador {

      public static void main(String[] args) {
```

```
        List<Originator.Memento> savedStates = new ArrayList<>();

        Originator originator = new Originator();

        originator.setState("Estado1");

        originator.setState("Estado2");

        savedStates.add(originador.saveState());

        originator.setState("Estado3");

      // Podemos ter múltiplos mementos e escolher qual queremos
restaurar.

        savedStates.add(originator.saveState());

        originator.set("Estado4");

        originador.restoreState(savedStates.get(1));

      }

}
```

Segue retorno do código citado:

```
Originador: Mudando estado para Estado1
Originador: Mudando estado para Estado2
Originador: Salvando o Memento.
Originador: Mudando estado para Estado3
Originador: Salvando o Memento.
Originador: Mudando estado para Estado4
Originador: Estado apos restaurar o Memento: Estado3
```

- **Iterator**: o Iterator tem como objetivo encapsular uma iteração, dependendo de uma interface chamada Iterator como padrão do projeto. Por encapsular as implementações de iterações, não precisamos mais ver que tipo de coleção está sendo utilizada pelos objetos, como um ArrayList ou um HashTable. Com a utilização do Padrão, precisamos apenas de um loop para lidarmos via polimorfismo com qualquer coleção de itens desde que ela apenas implemente o Iterator.

Anteriormente, também estávamos com o código vinculado a classes como ArrayList, agora usamos apenas uma interface (que é o Iterator).

Abaixo, um exemplo de implementação do Iterator em Java, onde a classe principal MenuItem é simplemente um item de menu que possui nome, que poderia ser um menu que apareceria na seção de menu de um site, por exemplo:

```
class MenuItem {

	String nome;

	MenuItem(String nome) {
		this.nome = nome;
	}

}

interface Iterator {
	boolean hasNext();
	Object next();
}

public class MenuIterator implements Iterator {

	MenuItem[] itens;
	int posicao = 0;

	public MenuIterator(MenuItem[] itens) {
		this.itens = itens;
	}

	public Object next() {
```

```
                MenuItem menuItem = itens[posicao];
        posicao++;
                return menuItem;
        }

        public boolean hasNext() {
                if (posicao >= itens.length || itens[posicao] == null) {
                return false;
                } else {
                return true;
                }
        }

}
```

- **Mediator**: este padrão promove um baixo acoplamento, evitando que os objetos façam referência uns aos outros de forma explícita, permitindo a variação no uso da interação de forma independente. Pensando em casos em que é necessário permitir que um grupo de objetos se comuniquem sem que haja acoplamento, ou para remover o forte acoplamento presente em relacionamentos muitos para muitos, ele pode ser utilizado.

Ao introduzir um Mediator, os objetos podem se comunicar sem se conhecer, já que um objeto Mediador deve encapsular toda a comunicação entre um grupo de objetos, além de sua interface ser utilizada para iniciar a comunicação e receber notificações.

Abaixo, um exemplo em UML de uma implementação Mediator:

Vantagens: o desacoplamento entre os diversos participantes da rede de comunicação, onde os participantes não se conhecem, a eliminação de relacionamentos muitos para muitos, onde são todos substituídos por relacionamentos um para muitos, e a política de comunicações estar centralizada no mediador e poder ser alterada sem mexer nos colaboradores.

Desvantagens: a centralização pode ser uma fonte de gargalos de desempenho e de risco para o sistema em caso de falhas, além de na prática os mediadores tenderem a se tornar mais complexos.

Abaixo, um exemplo em VB .NET, utilizando 5 classes, sendo elas:

**Participante** - representa a classe abstrata Colleague -Mantém uma referência ao seu objeto Mediator - se comunicam com o Mediator sempre que necessário; de outra forma se comunica com um participante;

**Membro** - representa a classe ConcreteColleague e herda de Participante;

**NaoMembro** - representa a classe ConcreteColleague e herda de Participante;

**AbstractChatSala** - representa a classe Mediator - Define uma interface para comunicação com os objetos Participante;

**ChatSala** - representa a classe concreta ConcreteMediator - Conhece as classes Participante e mantêm uma referência aos objetos Participante e implementa a comunicação e transferência de mensagens entre os objetos da classe Participante;

1- Participante.vb

```
''' <summary>
'''  A classe 'AbstractColleague'
''' </summary>
MustInherit Class Participante


      Private _chatsala As Chatsala
      Private _nome As String


    ' Construtor
      Public Sub New(ByVal nome As String)
      Me._nome = nome
      End Sub


    ' Pega o nome do participante
      Public ReadOnly Property Nome() As String
      Get
            Return _nome
      End Get
    End Property


    ' Pega a sala de chat
      Public Property Chatsala() As Chatsala
      Get
            Return _chatsala
      End Get
      Set(ByVal value As Chatsala)
            _chatsala = value
```

```
      End Set

      End Property



    ' Envia mensagem para um dado participante

      Public Sub Enviar(ByVal [para] As String, ByVal mensagem As
String)

      _chatsala.Enviar(_nome, [para], mensagem)

      End Sub



    ' Recebe mensagem de um participante

      Public Overridable Sub Receber(ByVal [de] As String, ByVal
mensagem As String)

        Console.WriteLine("{0} para {1}: '{2}'", [de], Nome,
mensagem)

      End Sub



End Class
```

2- Membro.vb

```
''' <summary>

''' A classe 'ConcreteColleague'

''' </summary>

Class Membro

      Inherits Participante



    ' Construtor

      Public Sub New(ByVal nome As String)

      MyBase.New(nome)

      End Sub

    'sobrescreve o método Receber

      Public Overrides Sub Receber(ByVal [de] As String, ByVal
mensagem As String)

      Console.Write("para Membro : ")

      MyBase.Receber([de], mensagem)

      End Sub

End Class
```

3- NaoMembro.vb

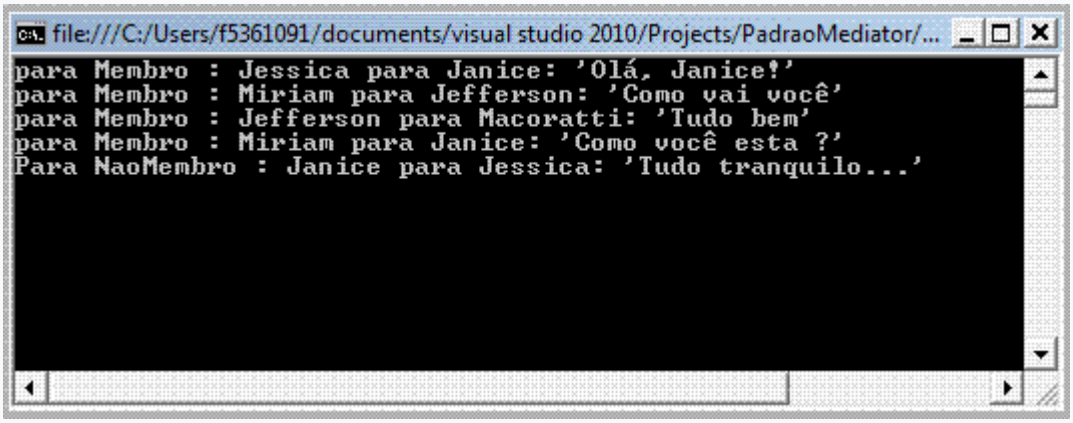
```
''' <summary>

''' A classe 'ConcreteColleague'

''' </summary>

Class NaoMembro

      Inherits Participante



    ' Construtor

      Public Sub New(ByVal nome As String)

      MyBase.New(nome)

      End Sub

      'sobrescreve o método Receber

      Public Overrides Sub Receber(ByVal [de] As String, ByVal
mensagem As String)

      Console.Write("Para NaoMembro : ")

      MyBase.Receber([de], mensagem)

      End Sub

End Class
```

4- AbstractChatSala

```
''' <summary>

''' A classe abstrata 'Mediator'

''' </summary>

MustInherit Class AbstractChatSala

      Public MustOverride Sub Registro(ByVal participante As
Participante)

      Public MustOverride Sub Enviar(ByVal [de] As String, ByVal
[para] As String, ByVal message As String)

End Class
```

5- ChatSala

```
''' <summary>

''' A classe concreta  'ConcreteMediator'

''' </summary>

Class Chatsala

    Inherits AbstractChatSala



     Private _participantes As New Dictionary(Of String,
Participante)()



     Public Overrides Sub Registro(ByVal _participante As
Participante)

     If Not _participantes.ContainsValue(_participante) Then

           _participantes(_participante.Nome) = _participante

     End If



     _participante.Chatsala = Me

     End Sub



     Public Overrides Sub Enviar(ByVal [de] As String, ByVal [para]
As String, ByVal mensagem As String)

     Dim _participante As Participante = _participantes([para])

     If _participante IsNot Nothing Then

           _participante.Receber([de], mensagem)

     End If

     End Sub

End Class
```




## Decorator

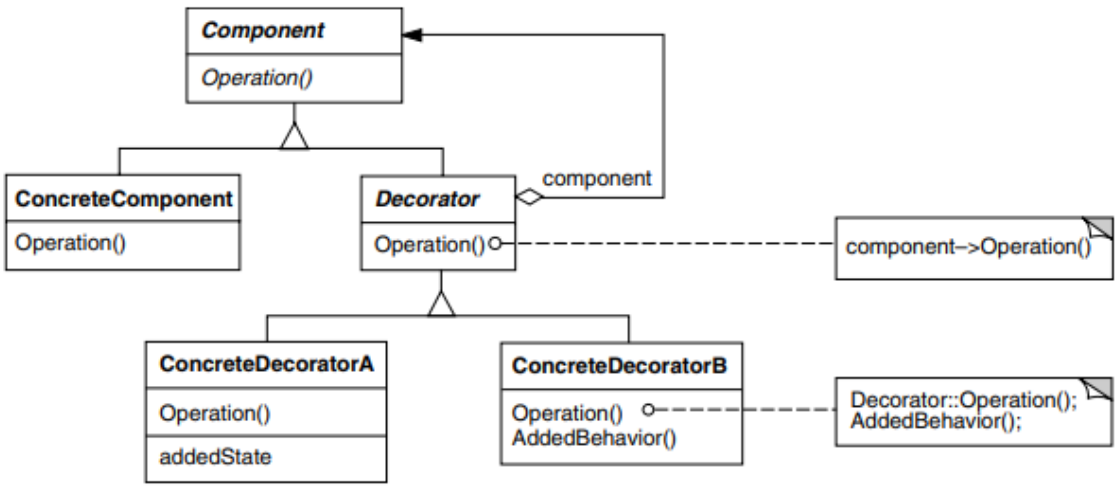
- Descrição do Padrão:

São uma alternativa flexível ao uso de subclasses pois agregam responsabilidade adicionais a um objeto de forma dinâmica.

- Uso do mesmo:

    - Acrescentar responsabilidades a objetos individuais de forma dinâmica e transparente, sem afetar os demais objetos;
    - Quando determinadas responsabilidades podem acabar sendo removidas no decorrer do projeto;

- Modelo de Implementação:




- Exemplos de uso (se possível ligado ao projeto do semestre):

```java
public interface DataSource {
    void writeData(String data);

    String readData();
}
```

(Uma interface de dados comum, que define as operações ler e escrever)

```java
public class FileDataSource implements DataSource {
    private String name;

    public FileDataSource(String name) {
        this.name = name;
    }

    @Override
    public void writeData(String data) {
        File file = new File(name);
        try (OutputStream fos = new FileOutputStream(file)) {
            fos.write(data.getBytes(), 0, data.length());
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
    }
```

```java
    @Override
    public String readData() {
        char[] buffer = null;
        File file = new File(name);
        try (FileReader reader = new FileReader(file)) {
            buffer = new char[(int) file.length()];
            reader.read(buffer);
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
        return new String(buffer);
    }
}
```

(Leitor-Escritor de dados simples)

```java
public class DataSourceDecorator implements DataSource {
    private DataSource wrappee;

    DataSourceDecorator(DataSource source) {
        this.wrappee = source;
    }

    @Override
    public void writeData(String data) {
        wrappee.writeData(data);
    }

    @Override
    public String readData() {
        return wrappee.readData();
    }
}
```

(Decorador base abstrato)

```java
public class EncryptionDecorator extends DataSourceDecorator {

    public EncryptionDecorator(DataSource source) {
        super(source);
    }

    @Override
    public void writeData(String data) {
        super.writeData(encode(data));
    }

    @Override
    public String readData() {
        return decode(super.readData());
    }

    private String encode(String data) {
        byte[] result = data.getBytes();
        for (int i = 0; i < result.length; i++) {
            result[i] += (byte) 1;
        }
        return Base64.getEncoder().encodeToString(result);
    }
```

```
    private String decode(String data) {
        byte[] result = Base64.getDecoder().decode(data);
        for (int i = 0; i < result.length; i++) {
            result[i] -= (byte) 1;
        }
        return new String(result);
    }
}
```

(Decorador de encriptação)

```
public class CompressionDecorator extends DataSourceDecorator {
    private int compLevel = 6;

    public CompressionDecorator(DataSource source) {
        super(source);
    }

    public int getCompressionLevel() {
        return compLevel;
    }

    public void setCompressionLevel(int value) {
        compLevel = value;
    }

    @Override
    public void writeData(String data) {
        super.writeData(compress(data));
    }

    @Override
    public String readData() {
        return decompress(super.readData());
    }
```

```
private String compress(String stringData) {
    byte[] data = stringData.getBytes();
    try {
        ByteArrayOutputStream bout = new ByteArrayOutputStream(512);
        DeflaterOutputStream dos = new DeflaterOutputStream(bout, new Deflater(compLevel))
        dos.write(data);
        dos.close();
        bout.close();
        return Base64.getEncoder().encodeToString(bout.toByteArray());
    } catch (IOException ex) {
        return null;
    }
}

private String decompress(String stringData) {
    byte[] data = Base64.getDecoder().decode(stringData);
    try {
        InputStream in = new ByteArrayInputStream(data);
        InflaterInputStream iin = new InflaterInputStream(in);
        ByteArrayOutputStream bout = new ByteArrayOutputStream(512);
        int b;
        while ((b = iin.read()) != -1) {
            bout.write(b);
        }
        in.close();
        iin.close();
        bout.close();
        return new String(bout.toByteArray());
    } catch (IOException ex) {
        return null;
    }
}
```

(Decorador de compressão)

```
public class Demo {
    public static void main(String[] args) {
        String salaryRecords = "Name,Salary\nJohn Smith,100000\nSteven Jobs,912000";
        DataSourceDecorator encoded = new CompressionDecorator(
                                        new EncryptionDecorator(
                                            new FileDataSource("out/OutputDemo.txt")));
        encoded.writeData(salaryRecords);
        DataSource plain = new FileDataSource("out/OutputDemo.txt");

        System.out.println("- Input ----------------");
        System.out.println(salaryRecords);
        System.out.println("- Encoded --------------");
        System.out.println(plain.readData());
        System.out.println("- Decoded --------------");
        System.out.println(encoded.readData());
    }
}
```

(Código Cliente)

- Correlações com outros padrões, se existirem:
    - Adapter: um padrão Decorator muda as responsabilidades de um objeto, não a sua interface. Enquanto um Adapter dará ao objeto uma interface completamente nova.
    - Composite: Um padrão Decorator pode ser visto como um padrão Composite degenerado com somente um componente. Contudo, um Decorator acrescenta responsabilidades adicionais e ele não se destina a agregação de objetos.
    - Strategy: Esses dois padrões são duas maneiras distintas de como mudar um objeto. O padrão Decorator permite mudar a superfície do objeto, enquanto o padrão Strategy permite mudar o interior do objeto.

## Observer

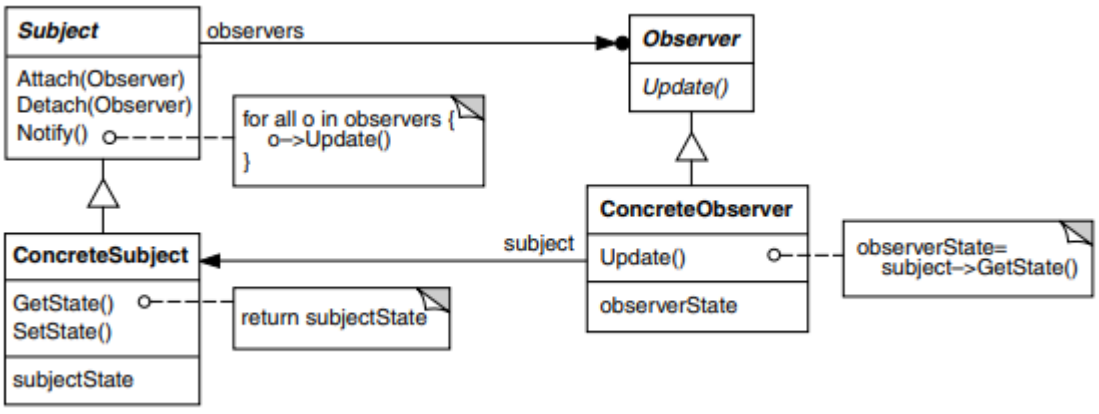
- Descrição do Padrão:

Define a dependência um-para-muitos entre objetos, com o objetivo de atualizar automaticamente os demais objetos quando um objeto sofrer alguma alteração.

- Uso do mesmo:

    - Quando uma mudança em um objeto exige mudanças em outros, e você não sabe quantos objetos necessitam ser alterados;
    - Quando um objeto seja capaz de notificar outros objetos, de forma que não haja um acoplamento forte entre eles.

- Modelo de Implementação:




- Exemplos de uso:

```java
public class EventManager {
    Map<String, List<EventListener>> listeners = new HashMap<>();

    public EventManager(String... operations) {
        for (String operation : operations) {
            this.listeners.put(operation, new ArrayList<>());
        }
    }

    public void subscribe(String eventType, EventListener listener) {
        List<EventListener> users = listeners.get(eventType);
        users.add(listener);
    }

    public void unsubscribe(String eventType, EventListener listener) {
        List<EventListener> users = listeners.get(eventType);
        users.remove(listener);
    }

    public void notify(String eventType, File file) {
        List<EventListener> users = listeners.get(eventType);
        for (EventListener listener : users) {
            listener.update(eventType, file);
        }
    }
}
```

(Publicador básico)

```java
public class Editor {
    public EventManager events;
    private File file;

    public Editor() {
        this.events = new EventManager("open", "save");
    }

    public void openFile(String filePath) {
        this.file = new File(filePath);
        events.notify("open", file);
    }

    public void saveFile() throws Exception {
        if (this.file != null) {
            events.notify("save", file);
        } else {
            throw new Exception("Please open a file first.");
        }
    }
}
```

(Publicador concreto, rastreado por outros objetos)

```java
public interface EventListener {
    void update(String eventType, File file);
}
```

(Interface comum do observer)

```
public class EmailNotificationListener implements EventListener {
    private String email;

    public EmailNotificationListener(String email) {
        this.email = email;
    }

    @Override
    public void update(String eventType, File file) {
        System.out.println("Email to " + email + ": Someone has performed " + eventType
    }
}
```

(Envia emails ao receber notificações)

```
public class LogOpenListener implements EventListener {
    private File log;

    public LogOpenListener(String fileName) {
        this.log = new File(fileName);
    }

    @Override
    public void update(String eventType, File file) {
        System.out.println("Save to log " + log + ": Someone has performed " + eventType
    }
}
```

(Escreve uma mensagem em um registro ao receber uma notificação)

```
public class Demo {
    public static void main(String[] args) {
        Editor editor = new Editor();
        editor.events.subscribe("open", new LogOpenListener("/path/to/log/file.txt"));
        editor.events.subscribe("save", new EmailNotificationListener("admin@example.com

        try {
            editor.openFile("test.txt");
            editor.saveFile();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

(Inicialização)

- Correlações com outros padrões, se existirem:
    - Mediator: quando o relacionamento de dependência entre subjects (fornece uma interface para acrescentar e remover objetos, permitindo associar e desassociar objetos observer) e observers (define uma interface de atualização para objetos que deveria ser notificados sobre mudanças em um Subject) se torna complexo, torna-se necessário a implementação de um objeto para controlar esses relacionamentos.

- Singleton: O objeto criado para controlar os subjects e os observers podem se tornar únicos e globalmente acessíveis como a implementação do Singleton.

**State**

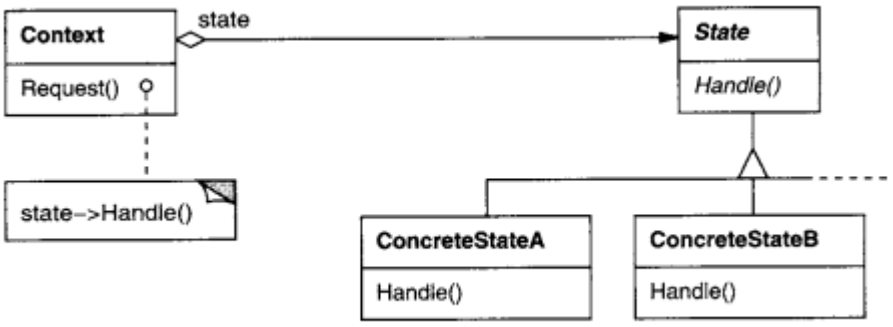
- Descrição do Padrão:

Permite que um objeto altere o seu estado interno, fazendo assim com que se pareça como outra classe, pois terá um comportamento diferente do anterior.

- Uso do mesmo:

  - Quando o objeto depende do seu estado para executar tais operações;
  - Desacopla o estado desse objeto e seus métodos em um objeto específico para esses estados
  - Facilita a adição de novos estados sem a necessidade de alterar estados anteriores

- Modelo de Implementação:




- Exemplos de uso:

```
public abstract class State {
    Player player;

    /**
     * Context passes itself through the state constructor. This may help a
     * state to fetch some useful context data if needed.
     */
    State(Player player) {
        this.player = player;
    }

    public abstract String onLock();
    public abstract String onPlay();
    public abstract String onNext();
    public abstract String onPrevious();
}
```

(Interface comum de state)

```java
public class LockedState extends State {

    LockedState(Player player) {
        super(player);
        player.setPlaying(false);
    }

    @Override
    public String onLock() {
        if (player.isPlaying()) {
            player.changeState(new ReadyState(player));
            return "Stop playing";
        } else {
            return "Locked...";
        }
    }

    @Override
    public String onPlay() {
        player.changeState(new ReadyState(player));
        return "Ready";
    }

    @Override
    public String onNext() {
        return "Locked...";
    }

    @Override
    public String onPrevious() {
        return "Locked...";
    }
}
```

```java
public class ReadyState extends State {

    public ReadyState(Player player) {
        super(player);
    }

    @Override
    public String onLock() {
        player.changeState(new LockedState(player));
        return "Locked...";
    }

    @Override
    public String onPlay() {
        String action = player.startPlayback();
        player.changeState(new PlayingState(player));
        return action;
    }

    @Override
    public String onNext() {
        return "Locked...";
    }

    @Override
    public String onPrevious() {
        return "Locked...";
    }
}
```

```java
public class PlayingState extends State {

    PlayingState(Player player) {
        super(player);
    }

    @Override
    public String onLock() {
        player.changeState(new LockedState(player));
        player.setCurrentTrackAfterStop();
        return "Stop playing";
    }

    @Override
    public String onPlay() {
        player.changeState(new ReadyState(player));
        return "Paused...";
    }

    @Override
    public String onNext() {
        return player.nextTrack();
    }

    @Override
    public String onPrevious() {
        return player.previousTrack();
    }
}
```

```java
public class Player {
    private State state;
    private boolean playing = false;
    private List<String> playlist = new ArrayList<>();
    private int currentTrack = 0;

    public Player() {
        this.state = new ReadyState(this);
        setPlaying(true);
        for (int i = 1; i <= 12; i++) {
            playlist.add("Track " + i);
        }
    }

    public void changeState(State state) {
        this.state = state;
    }

    public State getState() {
        return state;
    }

    public void setPlaying(boolean playing) {
        this.playing = playing;
    }
```

```
    public boolean isPlaying() {
        return playing;
    }

    public String startPlayback() {
        return "Playing " + playlist.get(currentTrack);
    }

    public String nextTrack() {
        currentTrack++;
        if (currentTrack > playlist.size() - 1) {
            currentTrack = 0;
        }
        return "Playing " + playlist.get(currentTrack);
    }

    public String previousTrack() {
        currentTrack--;
        if (currentTrack < 0) {
            currentTrack = playlist.size() - 1;
        }
        return "Playing " + playlist.get(currentTrack);
    }

    public void setCurrentTrackAfterStop() {
        this.currentTrack = 0;
    }
}
```

```
public class UI {
    private Player player;
    private static JTextField textField = new JTextField();

    public UI(Player player) {
        this.player = player;
    }

    public void init() {
        JFrame frame = new JFrame("Test player");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel context = new JPanel();
        context.setLayout(new BoxLayout(context, BoxLayout.Y_AXIS));
        frame.getContentPane().add(context);
        JPanel buttons = new JPanel(new FlowLayout(FlowLayout.CENTER));
        context.add(textField);
        context.add(buttons);

        // Context delegates handling user's input to a state object. Naturally,
        // the outcome will depend on what state is currently active, since all
        // states can handle the input differently.
        JButton play = new JButton("Play");
        play.addActionListener(e -> textField.setText(player.getState().onPlay()));
        JButton stop = new JButton("Stop");
        stop.addActionListener(e -> textField.setText(player.getState().onLock()));
        JButton next = new JButton("Next");
        next.addActionListener(e -> textField.setText(player.getState().onNext()));
        JButton prev = new JButton("Prev");
        prev.addActionListener(e -> textField.setText(player.getState().onPrevious()));
        frame.setVisible(true);
        frame.setSize(300, 100);
        buttons.add(play);
        buttons.add(stop);
        buttons.add(next);
        buttons.add(prev);
    }
}
```

```
public class Demo {
    public static void main(String[] args) {
        Player player = new Player();
        UI ui = new UI(player);
        ui.init();
    }
}
```

- Correlações com outros padrões, se existirem:
    - Flyweight: explica quando e como objetos State podem ser compartilhados
    - Singletons: objetos State são frequentemente Singletons. Padrão Singleton garante a existência de apenas uma instância dessa classe, mantendo um ponto global de acesso ao seu objeto.

## Strategy

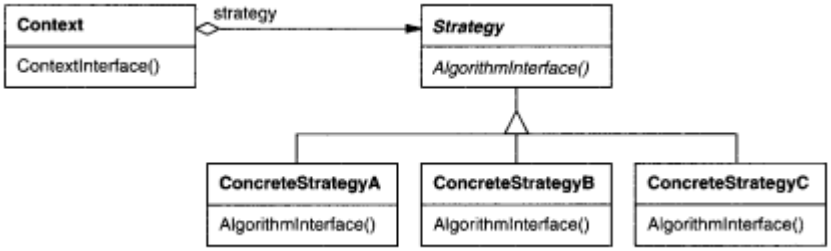
- Descrição do Padrão:

Visa dividir as regras de negócio do projeto dos algoritmos destas regras de negócio.

- Uso do mesmo:

    - Quando você quer usar diferentes variantes de um algoritmo dentro de um objeto e ser capaz de trocar de um algoritmo para outro durante a execução;
    - Quando você tem muitas classes parecidas que somente diferem na forma que elas executam algum comportamento;
    - Para isolar a lógica do negócio de uma classe dos detalhes de implementação de algoritmos que podem ser tão importantes no contexto da lógica;

- Modelo de Implementação:




- Exemplos de uso:

```
public interface PayStrategy {
    boolean pay(int paymentAmount);
    void collectPaymentDetails();
}
```

```
public class PayByPayPal implements PayStrategy {
    private static final Map<String, String> DATA_BASE = new HashMap<>();
    private final BufferedReader READER = new BufferedReader(new InputStreamReader(Syste
    private String email;
    private String password;
    private boolean signedIn;

    static {
        DATA_BASE.put("amanda1985", "amanda@ya.com");
        DATA_BASE.put("qwerty", "john@amazon.eu");
    }

    /**
     * Collect customer's data.
     */
    @Override
    public void collectPaymentDetails() {
        try {
            while (!signedIn) {
                System.out.print("Enter the user's email: ");
                email = READER.readLine();
                System.out.print("Enter the password: ");
                password = READER.readLine();
                if (verify()) {
                    System.out.println("Data verification has been successful.");
                } else {
                    System.out.println("Wrong email or password!");
                }
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
```

```
    private boolean verify() {
        setSignedIn(email.equals(DATA_BASE.get(password)));
        return signedIn;
    }

    /**
     * Save customer data for future shopping attempts.
     */
    @Override
    public boolean pay(int paymentAmount) {
        if (signedIn) {
            System.out.println("Paying " + paymentAmount + " using PayPal.");
            return true;
        } else {
            return false;
        }
    }

    private void setSignedIn(boolean signedIn) {
        this.signedIn = signedIn;
    }
}
```

```java
public class PayByCreditCard implements PayStrategy {
    private final BufferedReader READER = new BufferedReader(new InputStreamReader(Syste
    private CreditCard card;

    /**
     * Collect credit card data.
     */
    @Override
    public void collectPaymentDetails() {
        try {
            System.out.print("Enter the card number: ");
            String number = READER.readLine();
            System.out.print("Enter the card expiration date 'mm/yy': ");
            String date = READER.readLine();
            System.out.print("Enter the CVV code: ");
            String cvv = READER.readLine();
            card = new CreditCard(number, date, cvv);

            // Validate credit card number...

        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
```

```java
    @Override
    public boolean pay(int paymentAmount) {
        if (cardIsPresent()) {
            System.out.println("Paying " + paymentAmount + " using Credit Card.");
            card.setAmount(card.getAmount() - paymentAmount);
            return true;
        } else {
            return false;
        }
    }

    private boolean cardIsPresent() {
        return card != null;
    }
}
```

```java
public class CreditCard {
    private int amount;
    private String number;
    private String date;
    private String cvv;

    CreditCard(String number, String date, String cvv) {
        this.amount = 100_000;
        this.number = number;
        this.date = date;
        this.cvv = cvv;
    }

    public void setAmount(int amount) {
        this.amount = amount;
    }

    public int getAmount() {
        return amount;
    }
}
```

```java
public class Order {
    private int totalCost = 0;
    private boolean isClosed = false;

    public void processOrder(PayStrategy strategy) {
        strategy.collectPaymentDetails();
        // Here we could collect and store payment data from the strategy.
    }

    public void setTotalCost(int cost) {
        this.totalCost += cost;
    }

    public int getTotalCost() {
        return totalCost;
    }

    public boolean isClosed() {
        return isClosed;
    }

    public void setClosed() {
        isClosed = true;
    }
}
```

```
public class Demo {
    private static Map<Integer, Integer> priceOnProducts = new HashMap<>();
    private static BufferedReader reader = new BufferedReader(new InputStreamReader(Syst
    private static Order order = new Order();
    private static PayStrategy strategy;

    static {
        priceOnProducts.put(1, 2200);
        priceOnProducts.put(2, 1850);
        priceOnProducts.put(3, 1100);
        priceOnProducts.put(4, 890);
    }

    public static void main(String[] args) throws IOException {
        while (!order.isClosed()) {
            int cost;
```

```
    public static void main(String[] args) throws IOException {
        while (!order.isClosed()) {
            int cost;

            String continueChoice;
            do {
                System.out.print("Please, select a product:" + "\n" +
                        "1 - Mother board" + "\n" +
                        "2 - CPU" + "\n" +
                        "3 - HDD" + "\n" +
                        "4 - Memory" + "\n");
                int choice = Integer.parseInt(reader.readLine());
                cost = priceOnProducts.get(choice);
                System.out.print("Count: ");
                int count = Integer.parseInt(reader.readLine());
                order.setTotalCost(cost * count);
                System.out.print("Do you wish to continue selecting products? Y/N: ");
                continueChoice = reader.readLine();
            } while (continueChoice.equalsIgnoreCase("Y"));

            if (strategy == null) {
                System.out.println("Please, select a payment method:" + "\n" +
                        "1 - PalPay" + "\n" +
                        "2 - Credit Card");
                String paymentMethod = reader.readLine();

                // Client creates different strategies based on input from user,
                // application configuration, etc.
                if (paymentMethod.equals("1")) {
                    strategy = new PayByPayPal();
                } else {
                    strategy = new PayByCreditCard();
                }
            }
```

```
            order.processOrder(strategy);

            System.out.print("Pay " + order.getTotalCost() + " units or Continue shoppin
            String proceed = reader.readLine();
            if (proceed.equalsIgnoreCase("P")) {
                // Finally, strategy handles the payment.
                if (strategy.pay(order.getTotalCost())) {
                    System.out.println("Payment has been successful.");
                } else {
                    System.out.println("FAIL! Please, check your data.");
                }
                order.setClosed();
            }
        }
    }
}
```

- Correlações com outros padrões, se existirem:

    - Command: ambos padrões parametrizam um objeto com alguma ação, porém com propósitos diferentes;
    - Decorator: permite que você mude o contorno do objeto, enquanto o Strategy permite que você altere o interior do objeto;
    - State: considerado como uma extensão de Strategy. Ambos mudam o comportamento do contexto ao delegar algum trabalho para objetos auxiliares.

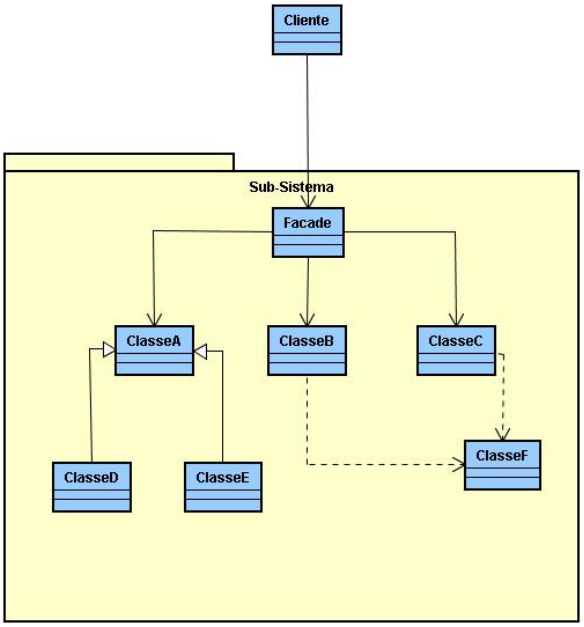
**Façade**

O Façade é um padrão estrutural, seu propósito é simplificar o acesso a códigos complexos, como bibliotecas ou sistemas grandes, escondendo a complexidade do código em uma interface simplificada.

Usos:

- Fornecer uma interface simplificada para um sistema complexo.
- Simplificar a inclusão de dependências.
- Dividir o sistema em camadas e limitar a responsabilidade de cada camada, fornecendo acesso a determinadas funcionalidades através da façade.

Modelo:




Exemplo:

```
class CPU {
  public void freeze() { ... }
  public void jump(long position) { ... }
  public void execute() { ... }
}

class Memory {
  public void load(long position, byte[] data) { ... }
}

class HardDrive {
  public byte[] read(long lba, int size) { ... }
}

class Computer {
  private CPU cpu;
  private Memory memory;
  private HardDrive hardDrive;

  public Computer() {
    this.cpu = new CPU();
    this.memory = new Memory();
    this.hardDrive = new HardDrive();
  }

  public void startComputer() {
    cpu.freeze();
    memory.load(BOOT_ADDRESS, hardDrive.read(BOOT_SECTOR, SECTOR_SIZE));
    cpu.jump(BOOT_ADDRESS);
    cpu.execute();
  }
}

class You {
  public static void main(String[] args) {
    Computer facade = new Computer();
    facade.startComputer();
  }
}
```

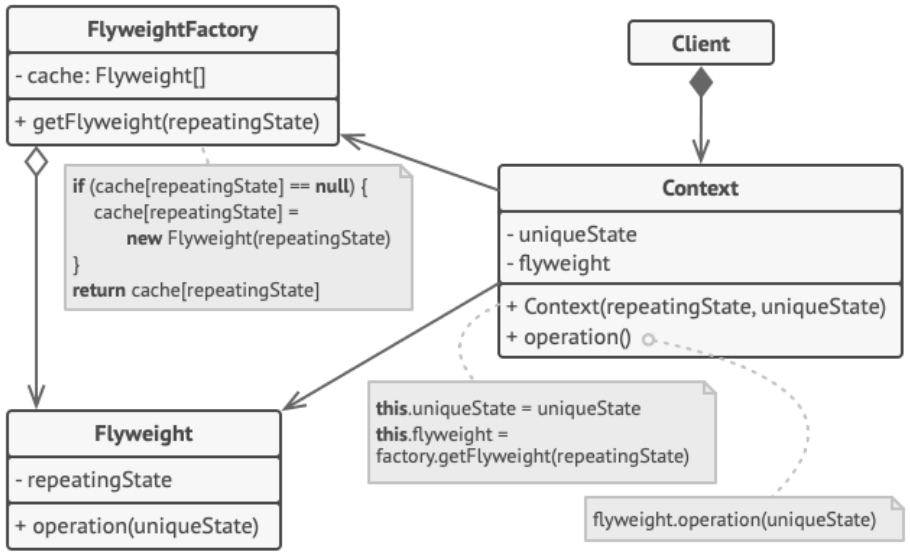
**Flyweight**

Flyweight é um padrão utilizado para reduzir o uso de memória de uma funcionalidade muito pesada, onde informações são muito repetidas.

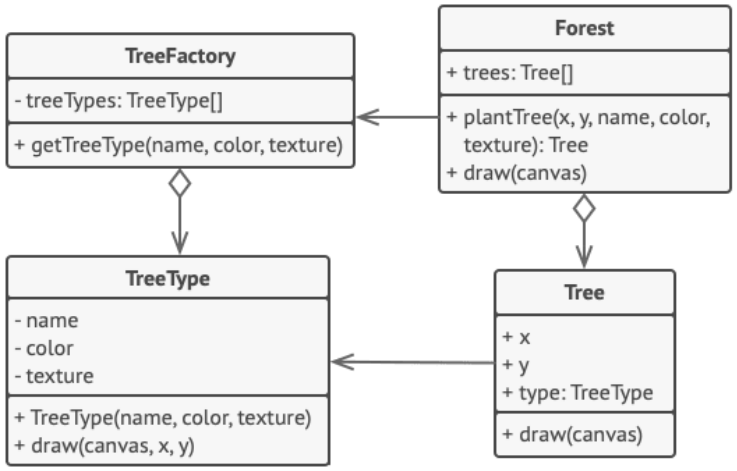
Usos:

- Quando o sistema gera vários objetos similares
- Quando é necessário reduzir o uso de memória

Modelo:




Exemplo:




Implementação:

```
class TreeType is
    field name
    field color
    field texture
    constructor TreeType(name, color, texture) { ... }
```

```
    method draw(canvas, x, y) is
        // 1. Cria um bitmap de certo tipo, cor e textura.
        // 2. Desenha o bitmap em uma tela nas coordenadas X e Y.

class TreeFactory is
    static field treeTypes: collection of tree types
    static method getTreeType(name, color, texture) is
        type = treeTypes.find(name, color, texture)
        if (type == null)
            type = new TreeType(name, color, texture)
            treeTypes.add(type)
        return type

class Tree is
    field x,y
    field type: TreeType
    constructor Tree(x, y, type) { ... }
    method draw(canvas) is
        type.draw(canvas, this.x, this.y)

class Forest is
    field trees: collection of Trees

    method plantTree(x, y, name, color, texture) is
        type = TreeFactory.getTreeType(name, color, texture)
        tree = new Tree(x, y, type)
        trees.add(tree)

    method draw(canvas) is
        foreach (tree in trees) do
            tree.draw(canvas)
```

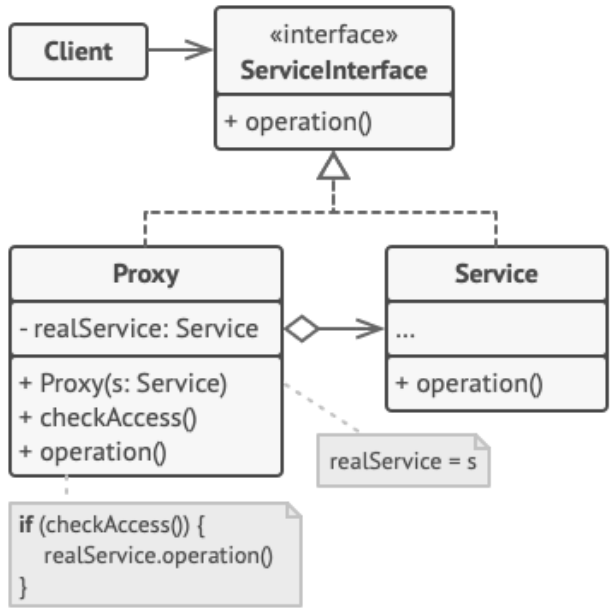
**Proxy**

O padrão Proxy serve como um intermediário entre a aplicação e algum serviço, fornecendo uma interface idêntica ao serviço, e possibilitando manipular a entrada e a saída de dados, tanto antes quanto após a interação com o serviço real.

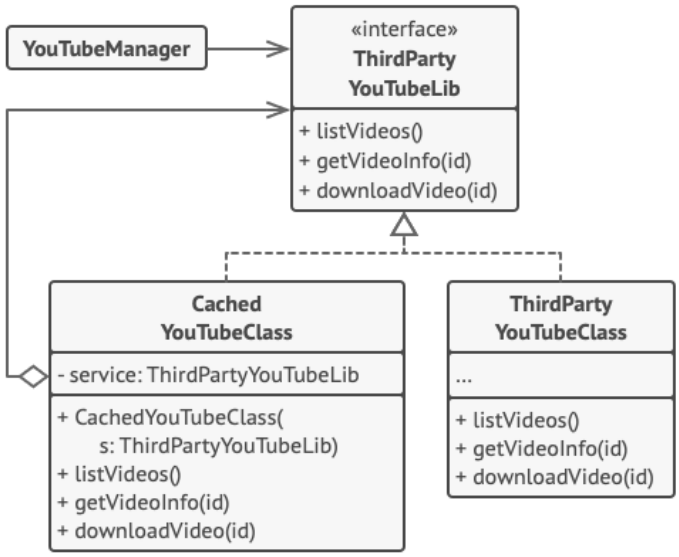
Usos:

- Quando é necessário manipular alguma funcionalidade de uma biblioteca externa.
- Para evitar repetição de códigos necessários para preparar ou mapear um dado que interage com um serviço.
- Para implementar um cache.

Modelo:




Exemplo:




Exemplo de Implementação:

```
// A interface de um serviço remoto.
interface ThirdPartyYouTubeLib is
    method listVideos()
    method getVideoInfo(id)
    method downloadVideo(id)

class ThirdPartyYouTubeClass implements ThirdPartyYouTubeLib is
    method listVideos() is
        // Envia um pedido API para o YouTube.

    method getVideoInfo(id) is
        // Obtém metadados sobre algum vídeo.
```

```
    method downloadVideo(id) is
        // Baixa um arquivo de vídeo do YouTube.

class CachedYouTubeClass implements ThirdPartyYouTubeLib is
    private field service: ThirdPartyYouTubeLib
    private field listCache, videoCache
    field needReset

    constructor CachedYouTubeClass(service: ThirdPartyYouTubeLib) is
        this.service = service

    method listVideos() is
        if (listCache == null || needReset)
            listCache = service.listVideos()
        return listCache

    method getVideoInfo(id) is
        if (videoCache == null || needReset)
            videoCache = service.getVideoInfo(id)
        return videoCache

    method downloadVideo(id) is
        if (!downloadExists(id) || needReset)
            service.downloadVideo(id)

class YouTubeManager is
    protected field service: ThirdPartyYouTubeLib

    constructor YouTubeManager(service: ThirdPartyYouTubeLib) is
        this.service = service

    method renderVideoPage(id) is
        info = service.getVideoInfo(id)
        // Renderiza a página do vídeo.

    method renderListPanel() is
        list = service.listVideos()
        // Renderiza a lista de miniaturas do vídeo.

    method reactOnUserInput() is
        renderVideoPage()
        renderListPanel()

// A aplicação pode configurar proxies de forma fácil e rápida.
class Application is
    method init() is
```

```
        aYouTubeService = new ThirdPartyYouTubeClass()
        aYouTubeProxy = new CachedYouTubeClass(aYouTubeService)
        manager = new YouTubeManager(aYouTubeProxy)
        manager.reactOnUserInput()
```

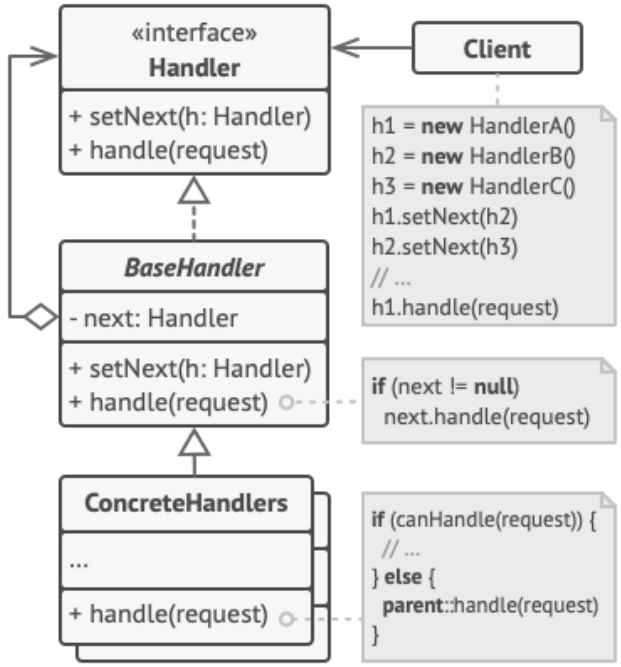
**Chain of Responsibility**

Chain of Responsibility é um padrão comportamental, estruturado de maneira similar a uma lista encadeada, ele serve para dividir diversos procedimentos que precisam ocorrer em sequência em classes separadas, que lidarão com a requisição e determinarão se o fluxo deve seguir ou ser finalizado.

Usos:

- Quando diversas validações precisam ser feitas antes de uma requisição ser processada.
- Na construção de uma árvore de elementos, para lidar com eventos.
- Quando múltiplos objetos podem lidar com uma requisição mas o objeto específico não é pré-determinado

Modelo:




Exemplo:

```
abstract class PurchasePower
{
    protected static final double BASE = 500;
    protected PurchasePower successor;

    abstract protected double getAllowable();

    abstract protected String getRole();

    public void setSuccessor(PurchasePower successor)
    {
        this.successor = successor;

    }

    public void processRequest(PurchaseRequest request)
    {
        if (request.getAmount() < this.getAllowable())
        {
            System.out.println(this.getRole() + " will approve $" + request.getAmount());
        }
        else if (successor != null)
        {
            successor.processRequest(request);
        }
    }
}
```

```
class ManagerPPower extends PurchasePower
{
    protected double getAllowable()
    {
        return BASE*10;
    }

    protected String getRole()
    {
        return "Manager";
    }
}

class DirectorPPower extends PurchasePower
{
    protected double getAllowable()
    {
        return BASE*20;
    }

    protected String getRole()
    {
        return "Director";
    }
}

class VicePresidentPPower extends PurchasePower
{
    protected double getAllowable()
    {
        return BASE*40;
    }

    protected String getRole()
    {
        return "Vice President";
    }
}

class PresidentPPower extends PurchasePower
{
    protected double getAllowable()
    {
        return BASE*60;
    }

    protected String getRole()
    {
        return "President";
    }
}
```

```java
class PurchaseRequest
{

    private double amount;
    private String purpose;

    public PurchaseRequest(double amount, String purpose)
    {
        this.amount = amount;
        this.purpose = purpose;
    }

    public double getAmount()
    {
        return amount;
    }

    public void setAmount(double amt)
    {
        amount = amt;
    }

    public String getPurpose()
    {
        return purpose;
    }

    public void setPurpose(String reason)
    {
        purpose = reason;
    }
}
```

```java
class CheckAuthority
{
    public static void main(String[] args)
    {
        ManagerPPower manager = new ManagerPPower();
        DirectorPPower director = new DirectorPPower();
        VicePresidentPPower vp = new VicePresidentPPower();
        PresidentPPower president = new PresidentPPower();
        manager.setSuccessor(director);
        director.setSuccessor(vp);
        vp.setSuccessor(president);

        // Press Ctrl+C to end.
        try
        {
            while (true)
            {
                System.out.println("Enter the amount to check who should approve your expenditure.");
                System.out.print(">");
                double d = Double.parseDouble(new BufferedReader(new InputStreamReader(System.in)).readLine());
                manager.processRequest(new PurchaseRequest(d, "General"));
            }
        } catch (Exception e)
        {
            System.exit(1);
        }
    }
}
```

- ## Adapter (Class)

  - ### Descrição do padrão:

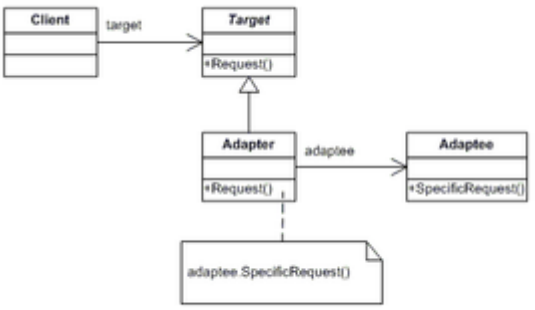
    O padrão **Adapter** converte a interface de uma classe para outra interface que o cliente espera encontrar, "traduzindo" solicitações do formato requerido pelo usuário para o formato compatível com a classe *adaptee* e as redirecionando. Dessa forma, o Adaptador permite que classes com interfaces incompatíveis trabalhem juntas.

  - ### Uso do mesmo:

    O padrão Adapter pode ser utilizado quando:

    1. se deseja utilizar uma classe existente, porém sua interface não corresponde à interface que se necessita;
    2. o desenvolvedor quiser criar classes reutilizáveis que cooperem com classes não-relacionadas ou não-previstas, ou seja, classes que não possuem necessariamente interfaces compatíveis;
    3. (exclusivamente para adaptadores de objetos) é necessário utilizar muitas subclasses existentes, porém, impossível de adaptar essas interfaces criando subclasses para cada uma. Um adaptador de objeto pode adaptar a interface de sua classe mãe.

  - ### Modelo de Implementação:

  - ### Exemplos de uso:

```java
//Classe adaptada (Adaptee)
class SensorXbox2 {

    //Solicitação Especifica
    public void conectarXbox2() {
        System.out.println("Um novo controle foi conectado ao sensor do Xbox.");
    }
}

//Classe alvo (Target)
class SensorPS5 {

    //Solicitação
    public void conectarPS5() {
        System.out.println("Um novo controle foi conectado ao sensor do PS5.");
    }
}

//Classe adaptador (Adapter)
class AdaptadorPS5ParaXbox2 extends SensorPS5 {

    private SensorXbox2 adaptee;

    public AdaptadorPS5ParaXbox2(SensorXbox2 adaptee) {
        this.adaptee = adaptee;
    }

    //Override da solicitação
    public void conectarPS5() {

adaptee.conectarXbox2();

System.out.println("But stadia wins!");
    }
}
```

```java
//Classe Cliente(Client)
public class ControlePS5 {

    private SensorPS5 sensorAQueSeConecta;

    public void Conectar(SensorPS5 sensor){

this.sensorAQueSeConecta = sensor;

sensorAQueSeConecta.conectarPS5();
    }

//}
//public class Teste{
    public static void main(String[] args) {

        //Tem-se um Xbox2 e mas deseja-se usar um controle ps5:
        SensorXbox2 adaptee = new SensorXbox2();
        ControlePS5 target = new ControlePS5();

        //Cria-se um falso sensor de PS5 que, na verdade, traduz e repassa os comandos para o Xbox em questão:
        AdaptadorPS5ParaXbox2 adapter = new AdaptadorPS5ParaXbox2(adaptee);

        //Conecta-se o controle ao adapter:

target.Conectar(adapter);
    }
}

//Saída:
//Um novo controle foi conectado ao sensor do Xbox.
//But stadia wins!
```

- Correlações com outros padrões:

**Bridge**: estrutura similar a um adaptador de objeto;

**Decorator**: aumenta outro objeto sem mudar sua interface;

**Proxy**: padrão estrutural que permite a definição de um representante ou “procurador” para outro objeto e não muda sua interface.

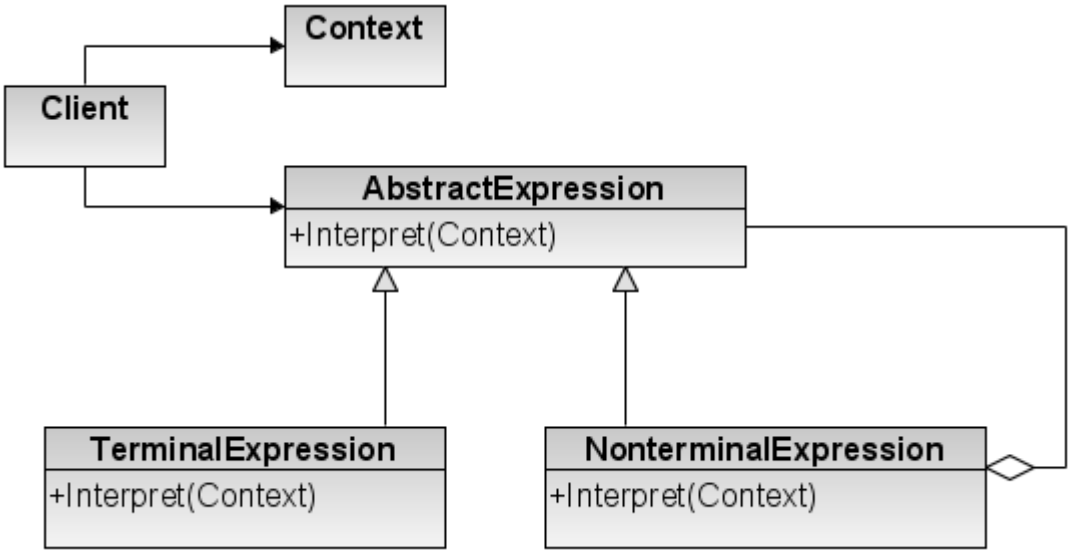
- **Interpreter**
  - Descrição do padrão:

    É um padrão muito utilizado para a resolução de problemas quando a modelagem de sistemas ou softwares. Esse padrão está incluso na categoria de Padrão Comportamental, ou seja, ele busca solucionar problemas de modelagem que tratam o comportamento de classes.

  - Uso do mesmo:

    1. Formato das consultas em banco de dados especializados como em SQL.
    2. Formato de mensagens usado pelos protocolos de comunicação.
    3. Tradução/conversão de linguagens ou símbolos para outra linguagem como números romanos para números decimais.
    4. Uso em expressões regulares ou XML.
    5. Uso de interpretação de formato em datas como DD-MM-AAAA ou MM-DD-AAAA.

- Modelo de Implementação:




- Exemplos de uso:

```
public class Client
    {
        public void BuildAndInterpretCommands()
        {
            Context context = new Context("Dot Net context");
            NonterminalExpression root = new NonterminalExpression
            {
                Expression1 = new TerminalExpression(),
                Expression2 = new TerminalExpression()
            };

root.Interpret(context);
        }
    }
```

- Correlações com outros padrões: -

- **Template Method**

- Descrição do padrão:

  Um **Template Method** auxilia na definição de um algoritmo com partes do mesmo definidas por métodos abstratos.

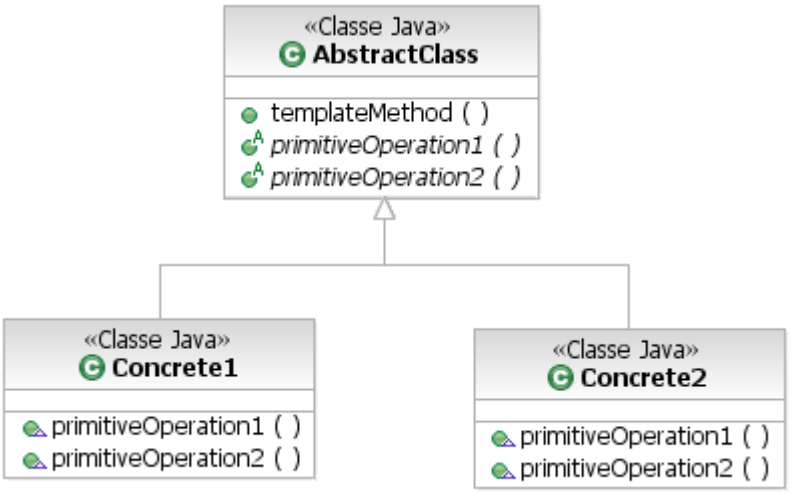
## ○ Uso do mesmo:

1. O padrão Template Method pode ser usado para implementar as partes invariantes de um algoritmo uma só vez e deixar para as subclasses a implementação do comportamento que pode variar.

2. Quando o comportamento comum entre subclasses deve ser fatorado e concentrado numa classe comum para evitar duplicação de código.

3. Para controlar extensões de subclasses. Pode-se definir um método-template que chama operações “gancho” em pontos específicos, desta forma permitindo extensões somente nesses pontos.

## ○ Modelo de Implementação:



- Exemplos de uso:

```java
package br.com.nc.architect.templatemethod;

public abstract class AbstractClass {

    public final void templateMethod() {
        System.out.println("AbstractClass.templateMethod() called");
        primitiveOperation1();
        primitiveOperation2();

    }

    public abstract void primitiveOperation1();
    public abstract void primitiveOperation2();
}
```

```java
package br.com.nc.architect.templatemethod;

public class Concrete1 extends AbstractClass {

    public void primitiveOperation1() {
        System.out.println("Concrete1.primitiveOperation1() called");
    }

    public void primitiveOperation2() {
        System.out.println("Concrete1.primitiveOperation2() called");
    }
}
```

```java
package br.com.nc.architect.templatemethod;

public class Concrete2 extends AbstractClass {

    public void primitiveOperation1() {
        System.out.println("Concrete2.primitiveOperation1() called");
    }

    public void primitiveOperation2() {
        System.out.println("Concrete2.primitiveOperation2() called");
    }
}
```

```java
package br.com.nc.architect.templatemethod;

public class TestTemplateMethod {

    public static void main(String[] args) {
        System.out.println("Test TemplateMethod");
        System.out.println("-------------------------");

        AbstractClass class1 = new Concrete1();
        AbstractClass class2 = new Concrete2();

        class1.templateMethod();
        class2.templateMethod();
    }
}
```

- Correlações com outros padrões:

**Factory Method:** é uma especialização do **Template Method**. Ao mesmo tempo, o *Factory Method* pode servir como uma etapa em um *Template Method* grande.

**Strategy:** baseado em composição: você pode alterar partes do comportamento de um objeto ao suprir ele como diferentes estratégias que correspondem a aquele comportamento. O *Template Method* funciona a nível de classe, então é estático. O *Strategy* trabalha a nível de objeto, permitindo que você troque os comportamentos durante a execução.

- **Visitor**
  - Descrição do padrão:

    O ***visitor pattern*** é um padrão de projeto comportamental. Representa uma operação a ser realizada sobre elementos da estrutura de um objeto. O Visitor permite que se crie uma nova operação sem que se mude a classe dos elementos sobre os quais ela opera. É uma maneira de separar um algoritmo da estrutura de um objeto. Um resultado prático é a habilidade de adicionar novas funcionalidades a estruturas de um objeto pré-existente sem a necessidade de modificá-las.

  - Uso do mesmo:

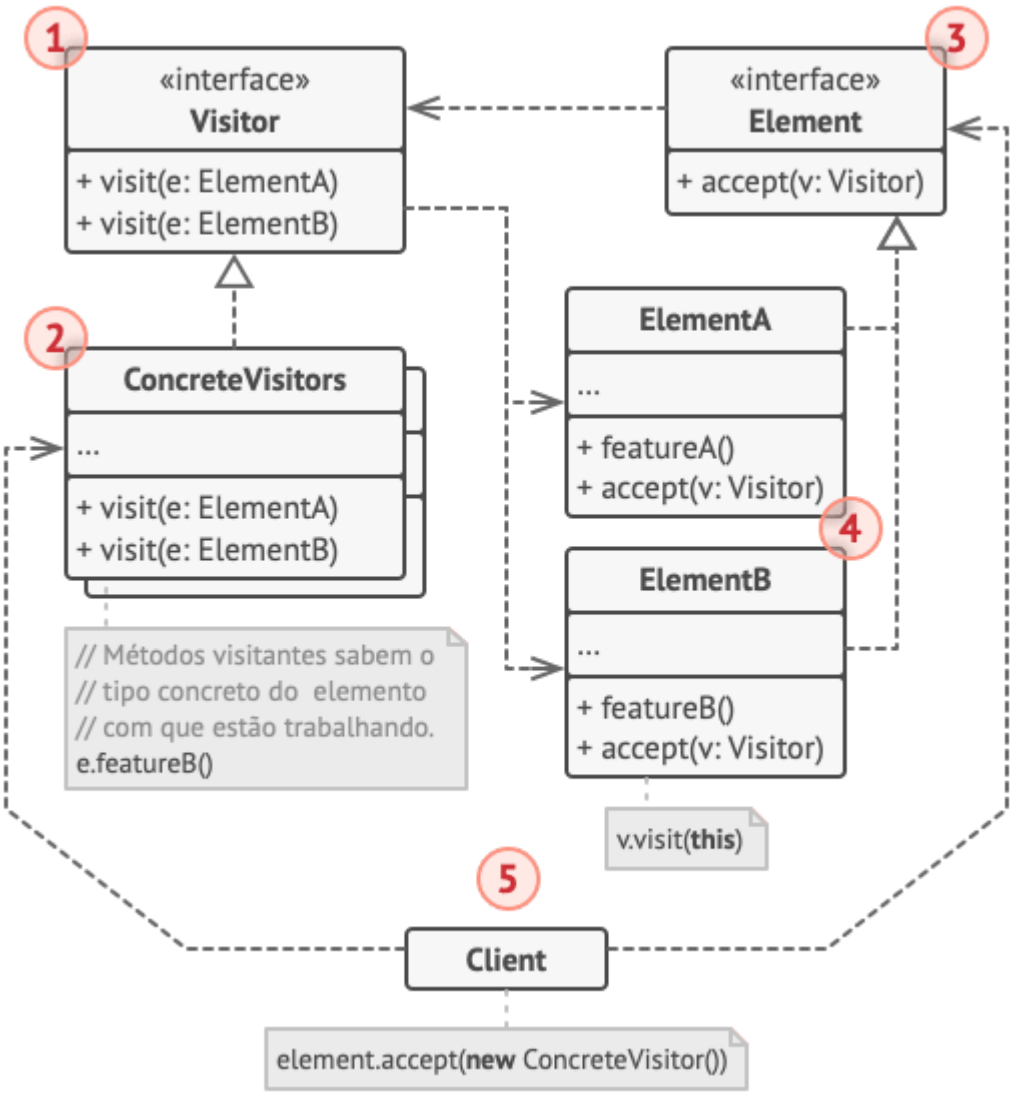
    O padrão Visitor é uma solução para separar o algoritmo da estrutura. Uma das vantagens desse padrão é a habilidade de adicionar novas operações a uma estrutura já existente.

    O padrão Visitor permite que você execute uma operação sobre um conjunto de objetos com diferentes classes ao ter o objeto visitante implementando diversas variantes da mesma operação, que correspondem a todas as classes alvo.

    O padrão permite que você torne as classes primárias de sua aplicação mais focadas em seu trabalho principal ao extrair todos os comportamentos em um conjunto de classes visitantes.

    Você pode extrair esse comportamento para uma classe visitante separada e implementar somente aqueles métodos visitantes que aceitam objetos de classes relevantes, deixando o resto vazio.

  - Modelo de Implementação:




- Exemplos de uso:

```
Visitor gravidade = new Gravidade(); //esse é o nosso visitor, responsável pelo
comportamento de queda.
Solido solido = new Solido("caixa"); //solido que recebera o comportamento
solido.accept(gravidade); //recebe o comportamento Gravidade
```

- Correlações com outros padrões:

Você pode tratar um Visitor como uma poderosa versão do padrão **Command**. Seus objetos podem executar operações sobre vários objetos de diferentes classes, pode usar o Visitor para executar uma operação sobre uma árvore **Composite** inteira e pode usar junto com o **Iterator** para percorrer uma estrutura de dados complexos e executar alguma operação sobre seus elementos, mesmo se eles todos tenham classes diferentes.