

# Template Method E Flyweight

---

Daniel Gandolfi e Felipe Romio



01


# Template Method



# Template Method



O Template Method é um padrão de projeto comportamental que define o esqueleto de um algoritmo na superclasse mas deixa as subclasses sobrescrever em etapas específicas do algoritmo sem modificar sua estrutura.



# Quando usar?

- Quando várias classes compartilham um algoritmo, mas com etapas específicas que variam.
- Em situações onde a lógica básica precisa ser fixa, mas permite extensões.
- Quando a manutenção de código é uma preocupação importante.

# Vantagens

- Promove a reutilização de código.
- Facilita a manutenção, pois a lógica básica é centralizada.
- Permite a extensão de funcionalidades através de subclasses.

# Desvantagens

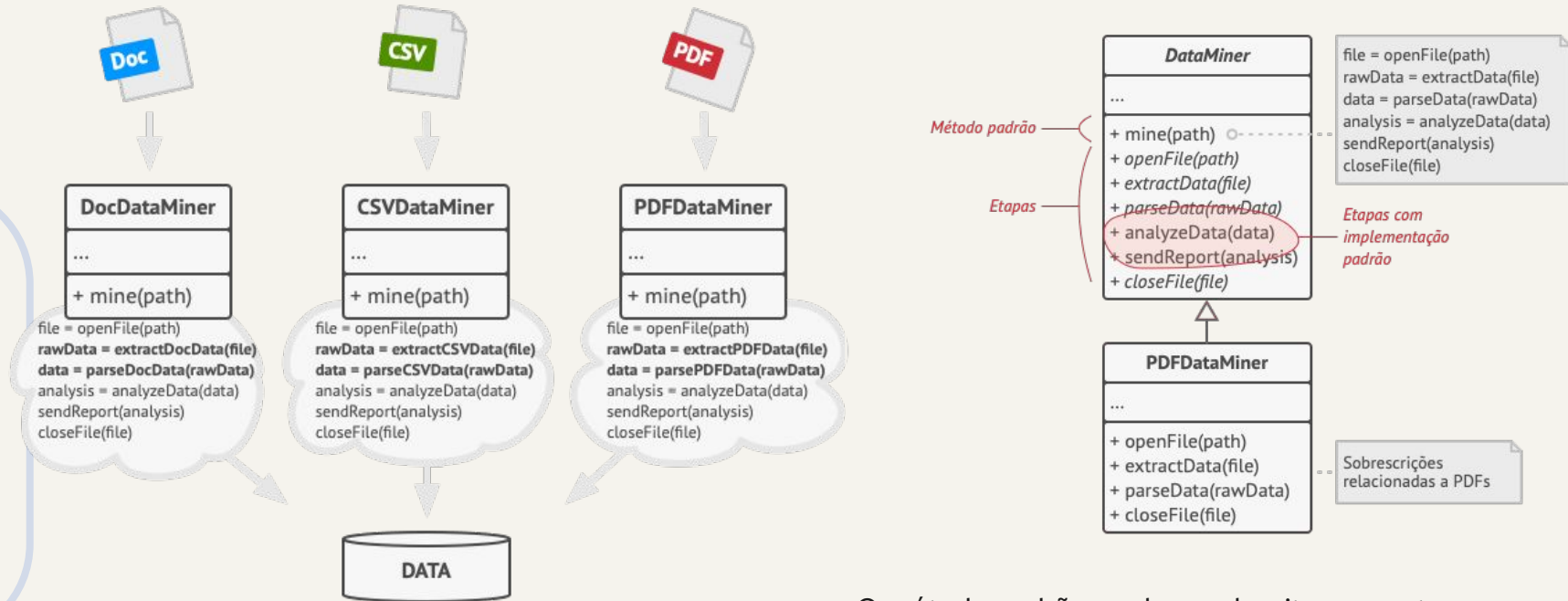
- Pode aumentar a complexidade do código se não for bem implementado.
- Dependência de uma estrutura de classe que pode ser restritiva.
- Difícil de alterar se o algoritmo precisar mudar significativamente.

# Problema x Solução

Imagine que você está criando uma aplicação de mineração de dados que analisa documentos corporativos. Os usuários alimentam a aplicação com documentos em vários formatos (PDF, DOC, CSV), e ela tenta extrair dados significativos desses documentos para um formato uniforme.

O padrão do Template Method sugere que você quebre um algoritmo em uma série de etapas, transforme essas etapas em métodos, e coloque uma série de chamadas para esses métodos dentro de um único método padrão. As etapas podem ser tanto abstratas, ou ter alguma implementação padrão.

# Problema x Solução



Código contém muita duplicação de comandos

O método padrão quebra o algoritmo em etapas, permitindo que subclasses sobrescrevam essas etapas mas não o método atual.



# Exemplo de código e explicação

```
public abstract class ProcessaPedido
{
    public void ProcessaPedido(ProcessaPedido pPedido)
    {
        CalculaFrete(pPedido);
        CalculaPesoPacote(pPedido);
        CalculaDesconto(pPedido);
    }

    protected virtual void CalculaDesconto(Pedido pPedido){}
    protected abstract void CalculaPesoPacote(Pedido pPedido);
    protected abstract void CalculaFrete(Pedido pPedido);
}
```



Template Method


Template Method define a estrutura de um processo (neste caso, o processamento do pedido) e permite que subclasses customizem partes desse processo, mantendo a ordem e estrutura geral do algoritmo.

# Exemplo de código e explicação

```
public class ProcessaPedidoOnLine : ProcessaPedido
{
    protected override void CalculaPesoPacote(Pedido pPedido)
    {
        //aqui entraria o código para calcular o peso do pacote
    }

    protected override void CalculaFrete(Pedido pPedido)
    {
        //aqui entraria o código para calcular o frete
    }
}
```

Mostra uma classe concreta em que nela serão definidos todos os detalhes para implementação do algoritmo definido na classe base.



02

Flyweight

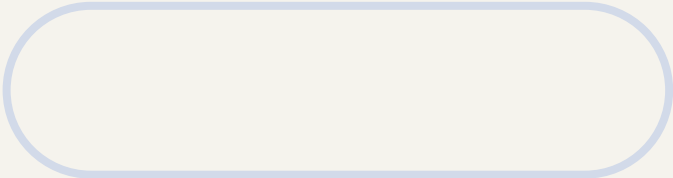


# Flyweight



---

O Flyweight é um padrão de projeto estrutural que otimiza o uso da memória ao compartilhar partes comuns de estado entre múltiplos objetos, em vez de armazenar todos os dados em cada instância individual. Isso permite criar mais objetos sem aumentar significativamente o consumo de memória.



# Quando usar?

- Usa-se o padrão Flyweight quando o sistema exige a criação de um grande número de objetos semelhantes, como elementos gráficos em um jogo ou caracteres em um editor de texto, e esses objetos possuem partes que podem ser compartilhadas.
- Esse padrão é particularmente vantajoso quando o custo de criação e armazenamento de objetos é elevado, pois ajuda a economizar memória e melhorar o desempenho.

# Os objetos são divididos em dois tipos de estado:

- **Estado Intrínseco:**
  - Dados imutáveis e compartilhados entre instâncias.
  - Armazenado dentro do objeto flyweight.
- **Estado Extrínseco:**
  - Dados variáveis que mudam de acordo com o contexto.
  - Passados como parâmetros quando o objeto flyweight é utilizado.

# Vantagens

- **Economia de Memória:** Reduz a quantidade de objetos necessários, evitando criações desnecessárias.
- **Melhor Performance:** Diminui o tempo de criação e o uso de memória, beneficiando sistemas com muitos objetos semelhantes.
- **Facilidade de Compartilhamento:** Permite que objetos flyweight sejam facilmente compartilhados entre diferentes partes do sistema.

# Desvantagens

- **Troca de RAM por ciclos de CPU:** Pode haver aumento no uso de processamento quando os dados de contexto precisam ser recalculados a cada chamada de método flyweight.
- **Complexidade de código:** A estrutura do código se torna mais complicada, dificultando o entendimento.
- **Curva de aprendizado:** Novos membros da equipe podem ter dificuldade para entender por que o estado de uma entidade foi separado dessa forma.



# Problema x Solução

Em uma cafeteria que atende um grande número de clientes, a equipe frequentemente recebe pedidos de cafés com sabores semelhantes. Cada pedido cria uma nova instância do objeto de café, o que resulta em um alto consumo de memória e impacta negativamente o desempenho do sistema. Isso se torna um problema especialmente crítico à medida que o número de mesas e pedidos aumenta, tornando a gestão dos objetos duplicados cada vez mais complexa.

O padrão Flyweight permite que a aplicação mantenha um número limitado de objetos de café em memória, utilizando a reutilização para diferentes pedidos. O resultado é uma aplicação mais eficiente, que pode escalar melhor e responder rapidamente a um volume maior de pedidos, melhorando a experiência do cliente na cafeteria.

# Exemplo Prático

```
class Cafe {
  constructor(sabor, preco) {
    this.sabor = sabor;
    this.preco = preco;
  }

  getSabor() {
    return this.sabor;
  }
}

class Cafeteria {
  constructor() {
    this.cafes = [];
  }

  fazerPedido(sabor, mesa) {
    this.cafes[mesa] = FábricaDeSaboresDeCafe.getSaborDeCafe(sabor);
    console.log("Sabor do café na mesa " + mesa + " é " + this.cafes[mesa].getSabor());
  }
}
```

# Exemplo Prático

```
class FábricaDeSaboresDeCafe {
  constructor() {
    this.sabores = [];
  }

  static getSaborDeCafe(sabor) {
    if (!this.sabores[sabor]) {
      this.sabores[sabor] = new Cafe(sabor, Math.floor(Math.random() * 10));
    }
    return this.sabores[sabor];
  }

  static getTotalSaboresDeCafeCriados() {
    return this.sabores.length;
  }
}

let cafeteria = new Cafeteria();

cafeteria.fazerPedido("Cappuccino", 1);
cafeteria.fazerPedido("Cappuccino", 2);
cafeteria.fazerPedido("Espresso", 3);
cafeteria.fazerPedido("Cappuccino", 4);
cafeteria.fazerPedido("Espresso", 5);

console.log("Total de Sabores de Café Criados: " +
  FábricaDeSaboresDeCafe.getTotalSaboresDeCafeCriados());
```

# Referencias

<https://refactoring.guru/pt-br/design-patterns/template-method>

<https://www.devmedia.com.br/patterns-template-method/18953>

<https://refactoring.guru/pt-br/design-patterns/flyweight>

<https://medium.com/@jonesroberto/design-patterns-parte-13-flyweight-9f96433bce05>