Aula 12 - Padrões de Projeto com JavaScript

aTip Learn - Lógica de Programação com JavaScript

Resumo da Aula

Nesta aula, exploraremos padrões de projeto, soluções reutilizáveis para problemas comuns no desenvolvimento de software. Aprenderemos seus benefícios e como podem tornar o código mais eficiente, flexível e fácil de manter. Abordaremos categorias como padrões de criação, estruturais e comportamentais, com exemplos práticos em JavaScript. Você ganhará uma compreensão sólida dos conceitos básicos de padrões de projeto e estará pronto para aplicá-los em seus projetos, melhorando a qualidade e eficiência do seu código.

Objetivos

- Compreender o que são padrões de projeto e qual sua importância no desenvolvimento de software.
- Conhecer as categorias de padrões de projeto: criação, estruturais e comportamentais.
- Aprender a aplicar padrões de projeto em JavaScript.
- Entender os benefícios dos padrões de projeto e como podem melhorar a qualidade do código.

Conteúdo

Introdução aos Padrões de Projeto

O que são padrões de projeto? Padrões de projeto são soluções reutilizáveis para problemas comuns no desenvolvimento de software. Eles representam boas práticas para resolver problemas de forma eficiente, flexível e fácil de manter. O uso de padrões de projeto pode tornar o código mais organizado, legível e reutilizável, além de facilitar a colaboração entre desenvolvedores, já que eles fornecem uma linguagem comum para discutir e compartilhar soluções.

Por exemplo, uma aplicação Web pode ser composta de diversos componentes, como botões, formulários, menus, etc. Cada componente pode ser exibido seguindo dois estilos diferentes, um estilo claro e um estilo escuro. Como podemos armazenar a informação sobre qual estilo deve ser exibido? Solucionar esse problema envolve dois requisitos básicos, a informação deve ser armazenada de forma única e sempre que essa informação for atualizada, todos os componentes que dependem dela devem ser atualizados. Essa é uma situação comum em desenvolvimento de software e pode ser resolvida com o uso dos padrões de projeto `Singleton` e `Observer`, sobre os quais falaremos mais adiante.

Por que usar padrões de projeto? Como mencionado anteriormente, utilizar padrões de projeto pode melhorar a qualidade do nosso código, tornando-o mais flexível e fácil de manter. Isso é especialmente verdadeiro quando estamos atuando em uma equipe de desenvolvedores, pois os padrões fornecem uma linguagem comum para discutir a solução de problemas recorrentes. Além disso, os padrões de projeto são soluções testadas e comprovadas em diversos cenários, o que nos permite economizar tempo e esforço na implementação de soluções.

O exemplo que mencionamos anteriormente pode ser resolvido de diversas formas diferentes, como armazenar a informação em uma variável global e passar a informação como argumento para cada componente. Entretanto, essas soluções podem ser difíceis de manter, especialmente com o aumento no número de componentes, rapidamente problemas como a duplicação e desatualização da informação do layout podem surgir. Utilizando padrões de projeto, estamos comunicando a todos os desenvolvedores de que forma a informação deve ser armazenada e atualizada, evitando problemas que poderiam surgir caso cada desenvolvedor implementasse sua própria solução.

Benefícios dos padrões de projeto.

- **Reutilização de código**: Os padrões de projeto são soluções gerais para problemas comuns, dessa forma, o código utilizado na solução de um problema muitas vezes pode ser reutilizado em outros problemas semelhantes.

- **Flexibilidade**: Utilizar padrões de projeto torna o código mais flexível, permitindo que ele seja adaptado a novos requisitos ou cenários sem a necessidade de grandes alterações.
- **Legibilidade**: Por fornecerem uma linguagem comum para discutir e compartilhar soluções, os padrões de projeto tornam o código mais legível e fácil de entender. Ao ver uma classe nomeada como `Singleton`, por exemplo, já sabemos que ela implementa esse padrão e podemos inferir como ela funciona.
- **Eficiência**: Os padrões de projeto são soluções para problemas recorrentes em projetos de software, o que significa que eles foram criados e refinados a partir da necessidade dos desenvolvedores desses projetos durante anos. Utilizar padrões de projeto pode economizar tempo e esforço na implementação de soluções, já que eles fornecem uma solução testada e comprovada para o problema.

Categorias de Padrões de Projeto

Padrões de Criação Os padrões de criação são padrões que lidam com a criação de objetos, abstraindo o processo de criação e tornando o sistema independente da forma como os objetos são criados, compostos e representados.

Alguns exemplos de padrões de criação são:

- **Singleton**: Garante que uma classe tenha apenas uma instância e fornece um ponto de acesso global para essa instância. É bastante útil quando queremos armazenar informações globais que devem ser acessadas por diversos componentes da aplicação, ou quando temos um recurso, como uma conexão de banco de dados, que deve ser compartilhado por todos os componentes da aplicação.
- **Factory**: Permite criar objetos sem que seja necessário especificar a classe exata do objeto que será criado. Isso é útil quando desejamos desacoplar a criação de objetos da sua utilização. Ou seja, o padrão Factory permite que a criação de objetos seja feita de forma genérica, sem a necessidade de conhecer a implementação específica de cada objeto. Uma analogia seria uma fábrica de veículos, onde você pode dizer para a fábrica criar um carro com determinadas características, sem precisar saber qual modelo de carro será criado.
- **Builder**: Permite a construção de objetos complexos passo a passo, separando a construção de um objeto de sua representação. Isso é útil quando queremos criar objetos com muitos atributos, mas não queremos passar todos esses atributos como argumentos para o construtor. O padrão Builder permite que a construção do objeto seja feita de forma incremental, adicionando os atributos necessários um a um.

Padrões Estruturais Os padrões estruturais são padrões que lidam com a composição de classes e objetos para formar estruturas maiores. Eles ajudam a garantir que, ao mudar a estrutura de um objeto, o impacto nas classes que dependem dele seja o menor possível.

Alguns exemplos de padrões estruturais são:

- **Adapter**: Permite que objetos com interfaces incompatíveis possam trabalhar juntos. Isso é útil quando queremos utilizar um objeto que possui uma interface diferente da que precisamos, sem precisar modificar o objeto original. Imagine que sua aplicação pode ser configurada para utilizar diferentes sistemas de banco de dados, cada banco de dados possui seus próprios métodos para realizar operações de leitura e escrita. Utilizando o padrão Adapter, podemos criar um adaptador para cada sistema de banco de dados, de forma que a aplicação possa chamar os métodos de leitura e escrita de forma genérica, sem precisar saber qual sistema de banco de dados está sendo utilizado.
- **Decorator**: Permite adicionar responsabilidades a um objeto de forma dinâmica. Isso é útil quando queremos adicionar funcionalidades a um objeto sem modificar sua estrutura. Por exemplo, imagine que você possui um método que realiza requisições HTTP e você deseja adicionar um log de cada requisição realizada. Utilizando o padrão Decorator, você pode criar um decorador que adiciona a funcionalidade de log ao método de requisição HTTP, sem modificar o método original.
- **Composite**: Permite tratar objetos individuais e composições de objetos de forma uniforme. Imagine por exemplo que você possui um componente da interface do usuário que é composto por diversos outros componentes (como um formulário composto por diversos campos). Cada campo pode ter seu próprio

método `validar` para verificar as informações inseridas pelo usuário. Utilizando o padrão Composite, você pode tratar o formulário como um único objeto, chamando o método `validar` do formulário, que por sua vez chama o método `validar` de cada campo individualmente.

Padrões Comportamentais Os padrões comportamentais são padrões que lidam com a comunicação entre objetos e a distribuição de responsabilidades entre eles. Eles ajudam a definir como os objetos interagem entre si e como as responsabilidades são distribuídas de forma eficiente.

Alguns exemplos de padrões comportamentais são:

- **Observer**: Permite que um objeto observe as mudanças em outro objeto e seja notificado quando essas mudanças ocorrem. Isso é útil quando queremos que um objeto seja notificado quando o estado de outro objeto muda. Por exemplo, imagine que você possui um componente da interface do usuário que exibe uma lista de itens, e essa lista de itens é armazenada em um objeto separado. Utilizando o padrão Observer, você pode fazer com que o componente da interface do usuário seja notificado sempre que um item é adicionado ou removido da lista, mantendo a interface sempre atualizada.
- **Strategy**: Permite definir uma família de algoritmos, encapsulá-los e torná-los intercambiáveis. Isso é útil quando queremos que um objeto possa utilizar diferentes algoritmos de forma dinâmica. Por exemplo, imagine que você possui um componente da interface do usuário que realiza uma operação de ordenação em uma lista de itens. Utilizando o padrão Strategy, você pode definir diferentes algoritmos de ordenação (como ordenação por nome, por data, por valor, etc.) e permitir que o componente da interface do usuário escolha qual algoritmo utilizar em tempo de execução.

Exemplos de Padrões de Projeto em JavaScript

Singleton O padrão **Singleton** serve para garantir que uma determinada classe possua uma única instância e fornece uma forma única de acessar essa instância. Existem diversos casos onde esse padrão é útil, como por exemplo, quando temos uma configuração global da nossa aplicação que deve ser acessada de diversas partes do nosso código, nesse caso nós queremos garantir que haja apenas uma cópia dessa configuração. Outro exemplo é quando temos um recurso que precisa ser compartilhado, como uma conexão de banco de dados, nesse caso queremos garantir que todos os componentes da aplicação utilizem a mesma conexão.

```
class ConfiguracaoSingleton {
  static #instancia = null;

  static obterInstancia() {
    if (ConfiguracaoSingleton.#instancia === null) {
      ConfiguracaoSingleton.#instancia = new ConfiguracaoSingleton();
    }
    return ConfiguracaoSingleton.#instancia;
  }

  constructor() {
    this.configuracao = {};
  }

  definirConfiguracao(chave, valor) {
    this.configuracao[chave] = valor;
  }

  lerConfiguracao(chave) {
    return this.configuracao[chave];
  }
}

const configuracao = ConfiguracaoSingleton.obterInstancia();
```

```
configuracao.definirConfiguracao("idioma", "pt-BR");

const outraConfiguracao = ConfiguracaoSingleton.obterInstancia();
console.log(outraConfiguracao.lerConfiguracao("idioma")); // 'pt-BR'
```

Nesse exemplo, criamos uma classe `ConfiguracaoSingleton` que implementa o padrão Singleton. Para garantir que uma única instância da classe seja utilizada nós implementamos o método estático `obterInstancia`, que verifica se a instância já foi criada e a retorna, ou cria uma nova instância caso ela ainda não exista. Dessa forma, podemos garantir que a configuração da nossa aplicação seja armazenada de forma única e acessível de qualquer parte do código.

Quando inicializamos as variáveis `configuracao` e `outraConfiguracao`, ambas estão acessando a mesma instância da classe `ConfiguracaoSingleton`, garantindo que a configuração seja compartilhada entre elas.

Factory O padrão Factory é um padrão de projeto que facilita a criação de objetos. Em vez de criar objetos diretamente usando o operador `new`, você utiliza uma função ou método ("fábrica") que decide qual tipo de objeto criar e retorna uma instância desse objeto.

Esse padrão é útil quando você precisa criar objetos de diferentes tipos, mas quer manter a lógica de criação de objetos separada do código que utiliza esses objetos. Isso permite que você adicione novos tipos de objetos sem precisar modificar o código que os utiliza.

```
class Carro {
    acelerar() {
        console.log("Acelerando um carro...");
    }
}

class Bicicleta {
    pedalar() {
        console.log("Pedalando uma bicicleta...");
    }
}

// Função Factory
function criarVeiculo(propulsao) {
    if (propulsao === "pedal") {
        return new Bicicleta();
    } else if (propulsao === "motor") {
        return new Carro();
    } else {
        throw new Error("Tipo de veículo não suportado.");
    }
}

// Usando a Factory
const veiculo1 = criarVeiculo("motor");
veiculo1.acelerar(); // "Dirigindo um carro..."

const veiculo2 = criarVeiculo("pedal");
veiculo2.pedalar(); // "Pedalando uma bicicleta..."
```

Nesse exemplo, criamos duas classes, `Carro` e `Bicicleta`, que representam diferentes tipos de veículos. Em seguida, criamos uma função `criarVeiculo` que atua como uma fábrica de veículos, recebendo um parâmetro `propulsao` que determina qual tipo de veículo deve ser criado. Dessa forma, podemos criar diferentes tipos de veículos sem precisar modificar o código que os utiliza.

Builder O padrão Builder é um padrão de projeto que permite a construção de objetos complexos passo a passo. Ele é especialmente útil quando temos objetos com muitas propriedades, ao invés de passarmos todas as propriedades como parâmetros para o construtor, podemos utilizar uma classe Builder para construir o objeto passo a passo.

```
class Carro {
    constructor(builder) {
        this.modelo = builder.modelo;
        this.cor = builder.cor;
        this.ano = builder.ano;
        this.cilindradas = builder.cilindradas;
    }
}

class CarroBuilder {
    constructor() {
        this.modelo = "";
        this.cor = "";
        this.ano = "";
        this.cilindradas = 1600;
    }

    setModelo(modelo) {
        this.modelo = modelo;
        return this;
    }

    setCor(cor) {
        this.cor = cor;
        return this;
    }

    setAno(ano) {
        this.ano = ano;
        return this;
    }

    setCilindradas(cilindradas) {
        this.cilindradas = cilindradas;
        return this;
    }

    build() {
        return new Carro(this);
    }
}

const carro = new CarroBuilder()
    .setModelo("Fusca")
    .setCor("Vermelho")
    .setAno(1970)
    .build();

console.log(carro); // Carro { modelo: 'Fusca', cor: 'Vermelho', ano: 1970, cilindradas: 1600 }
```

Nesse exemplo temos uma classe `Carro` que possui diversas propriedades, e uma classe `CarroBuilder` que permite a construção de um objeto `Carro` passo a passo. A classe `CarroBuilder` possui métodos para definir cada propriedade do carro, e um método `build` que retorna o objeto `Carro` construído. Dessa forma, podemos criar um objeto `Carro` passo a passo, definindo apenas as propriedades que desejamos.

Perceba que os métodos da classe `CarroBuilder` utilizados para definir as propriedades do carro tem como valor de retorno a própria instância da classe (`this`). Isso permite que possamos encadear a chamada dos métodos, tornando a construção do objeto mais fluída e legível.

Podemos ainda definir valores padrão para as propriedades do carro na classe `CarroBuilder`, como no caso da propriedade `cilindradas`, que é definida como 1600 caso não seja especificada.

Observer O padrão Observer é um padrão de projeto que permite que um objeto (o "observador") observe as mudanças em outro objeto (o "sujeito") e seja notificado quando essas mudanças ocorrem. Isso é útil quando precisamos notificar diversos objetos sobre mudanças em um objeto, sem que esses objetos precisem conhecer a implementação do objeto observado.

```
class ConexaoInternet { // Sujeito
  #conectado = false;
  #observadores = [];

  adicionarObservador(observador) {
    this.#observadores.push(observador);
  }

  notificarObservadores() {
    this.#observadores.forEach(observador => observador.atualizarConexao(this.#conectado));
  }

  conectar() {
    this.#conectado = true;
    this.notificarObservadores();
  }

  desconectar() {
    this.#conectado = false;
    this.notificarObservadores();
  }
}

class Logger { // Observador
  atualizarConexao(conectado) {
    if (conectado) {
      console.log("Conexão estabelecida.");
    } else {
      console.log("Conexão perdida.");
    }
  }
}

const conexao = new ConexaoInternet();
const logger = new Logger();

conexao.adicionarObservador(logger);
```

```
conexao.conectar(); // "Conexão estabelecida."
conexao.desconectar(); // "Conexão perdida."
```

Nesse exemplo, temos uma classe `ConexaoInternet` que representa um objeto que pode estar conectado ou desconectado. A classe possui um método `adicionarObservador` que permite adicionar observadores que serão notificados sobre mudanças na conexão. Quando a conexão é estabelecida ou perdida, o método `notificarObservadores` é chamado para notificar todos os observadores sobre a mudança.

A classe `Logger` é um exemplo de observador que implementa o método `atualizarConexao`, que é chamado quando a conexão é estabelecida ou perdida. Dessa forma, o observador pode reagir às mudanças na conexão de acordo com sua lógica específica.

Conclusão

Nessa aula exploramos o que são os padrões de projeto e para que são utilizados. Vimos as categorias de padrões de projeto, como os padrões de criação, estruturais e comportamentais, e exemplos práticos em JavaScript dos padrões mais comumente utilizados.

É importante ressaltar que o que vimos nessa aula é apenas uma pequena amostra dos padrões de projeto existentes, e que existem muitos outros padrões que podem ser aplicados em diferentes situações. A escolha do padrão de projeto correto depende do problema que estamos tentando resolver e das características específicas do nosso sistema.

Para explorar mais a fundo o mundo dos padrões de projeto veja os links e referências abaixo:

- Site Um Guia Extensivo de Padrões de Design JavaScript
- Site Refactoring Guru (em Inglês)
- Site JavaScript Design Patterns (em Inglês)