

# Creational Patterns

## Padrões aplicados

### Singleton

No sistema, aplicamos o padrão Singleton ao componente responsável por gerar códigos de acesso. A principal motivação foi garantir que essa funcionalidade crítica tivesse uma única instância controlada e consistente durante todo o ciclo de vida da aplicação. Como o gerador de código de acesso representa um componente único dentro do domínio da aplicação, o Singleton se mostrou a melhor forma de assegurar que apenas uma instância exista no sistema, garantindo assim previsibilidade e integridade no seu funcionamento.

### Benefícios

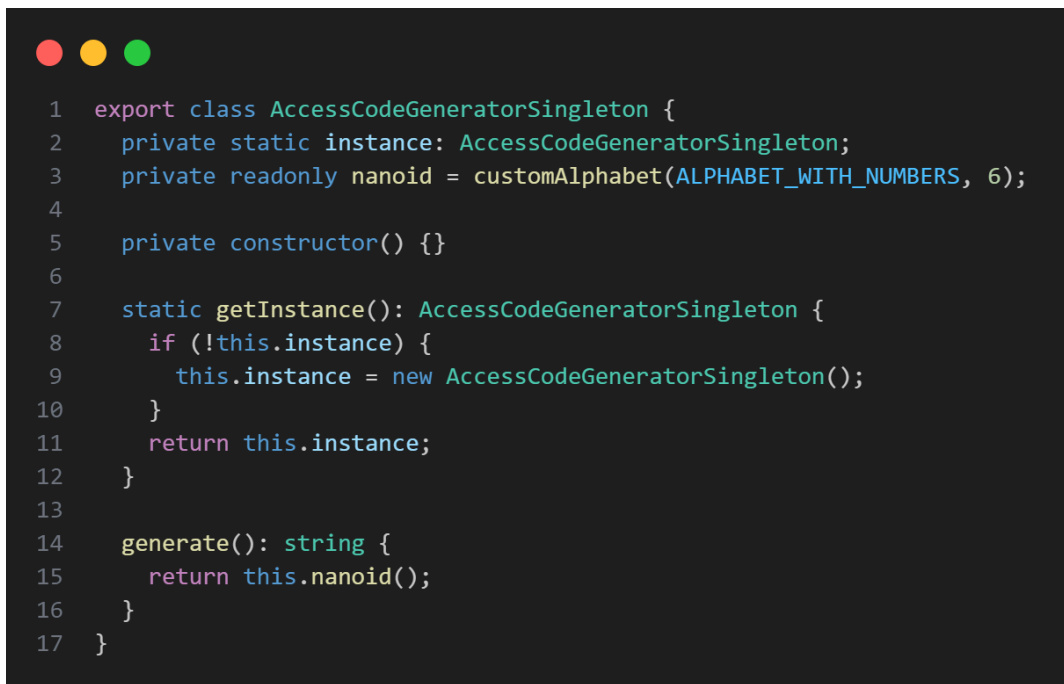
- **Centralização:** Garante que todos os códigos de acesso sejam gerados por uma única instância.
- **Consistência:** Facilita o controle de regras (por exemplo, histórico de códigos já gerados), pois tudo está centralizado em um único objeto.
- **Eficiência:** Evita criar múltiplas instâncias desnecessárias do gerador, economizando recursos e simplificando o gerenciamento de dependências.

### Comparação do código



```
1  export class AccessCodeGenerator {
2      private static readonly nanoid = customAlphabet(ALPHABET_WITH_NUMBERS, 6);
3
4      static generate(): string {
5          return this.nanoid();
6      }
7  }
8
```

Figura 01: Antes



```
1  export class AccessCodeGeneratorSingleton {
2      private static instance: AccessCodeGeneratorSingleton;
3      private readonly nanoid = customAlphabet(ALPHABET_WITH_NUMBERS, 6);
4
5      private constructor() {}
6
7      static getInstance(): AccessCodeGeneratorSingleton {
8          if (!this.instance) {
9              this.instance = new AccessCodeGeneratorSingleton();
10         }
11         return this.instance;
12     }
13
14     generate(): string {
15         return this.nanoid();
16     }
17 }
```

Figura 02: Depois

## Padrões não aplicados

### Factory Method e Abstract Factory

Optamos por não aplicar os padrões Factory ou Abstract Factory neste sistema, pois não foram identificados cenários com variações significativas de objetos ou hierarquias de subclasses que justificassem seu uso. Futuramente, o Factory pode ser usado para diversos tipos de conteúdo do clipboard, como texto, arquivos ou imagens.

### Builder

O padrão Builder também foi considerado desnecessário no estágio atual do projeto. Pois nosso projeto não tem a construção de objetos envolvendo múltiplos passos, regras opcionais e combinações de atributos. Mas ele pode vir a ser útil caso optemos por implementar clipboards com atributos configuráveis, como tipo, estratégia de expiração, permissões de acesso, etc.

### Prototype

Considerando que o projeto não possui muitas classes com múltiplas instâncias, não vimos necessidade de utilizar o padrão Prototype. As instâncias atuais são, em geral, únicas ou facilmente reconstruídas, sem impacto significativo em desempenho ou reutilização de configuração.

# Structural Patterns

## Padrões aplicados

### Decorator

#### Benefícios

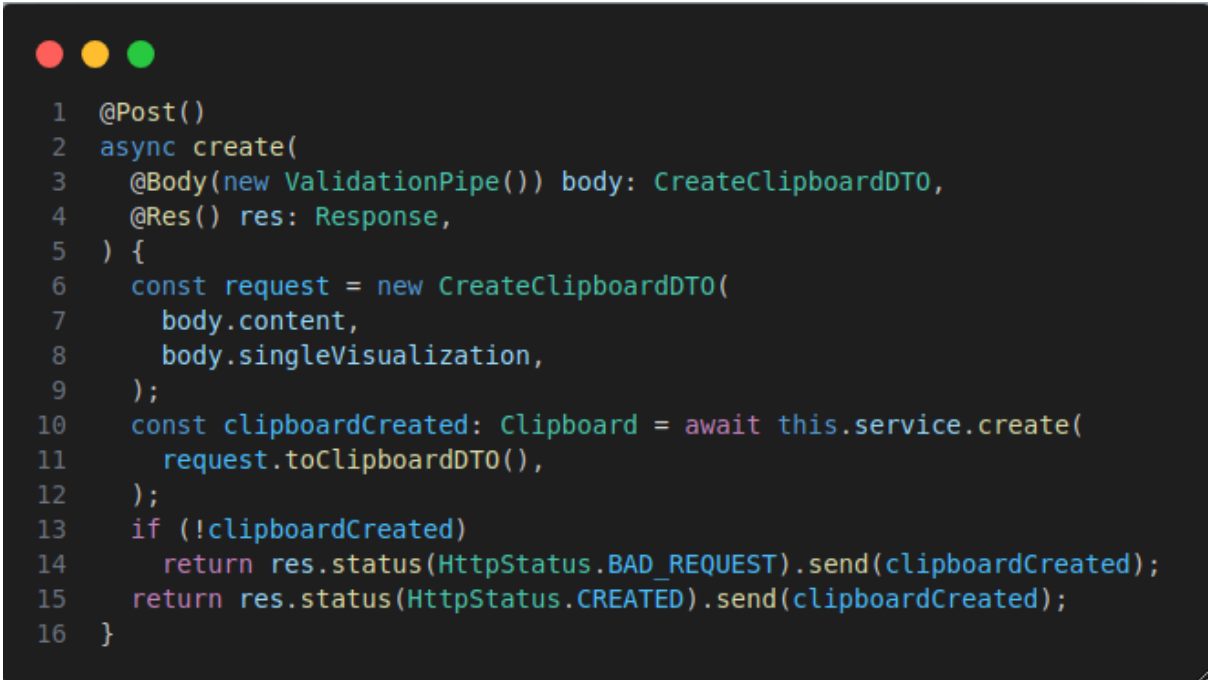
- Você pode estender o comportamento de um objeto sem fazer uma nova subclasse.
- Você pode combinar diversos comportamentos ao envolver o objeto com múltiplos decoradores.
- Deixa o alvo mais simples

### Adapter

#### Benefícios


- Maior separação do código
- Mais simples

### Comparação de código



```
1  @Post()
2  async create(
3    @Body(new ValidationPipe()) body: CreateClipboardDTO,
4    @Res() res: Response,
5  ) {
6    const request = new CreateClipboardDTO(
7      body.content,
8      body.singleVisualization,
9    );
10   const clipboardCreated: Clipboard = await this.service.create(
11     request.toClipboardDTO(),
12   );
13   if (!clipboardCreated)
14     return res.status(HttpStatus.BAD_REQUEST).send(clipboardCreated);
15   return res.status(HttpStatus.CREATED).send(clipboardCreated);
16 }
```

Figura 03: Antes



```
1  @Post()
2  async create(
3      @Body(new ValidationPipe()) body: CreateClipboardDTO,
4      @Res() res: Response,
5  ) {
6      const clipboardDTO = ClipboardAdapter.fromCreateDto(body);
7      const clipboardCreated: Clipboard = await this.service.create(clipboardDTO);
8
9      if (!clipboardCreated) {
10         return res.status(HttpStatus.BAD_REQUEST).send(clipboardCreated);
11     }
12     return res.status(HttpStatus.CREATED).send(clipboardCreated);
13 }
```

Figura 04: Depois

## Padrões não aplicados

### Bridge

Não utilizamos o padrão Bridge pois o projeto não apresenta abstrações que demandem múltiplas implementações intercambiáveis ou desacoplamento entre níveis. Atualmente, os serviços possuem ligações bem definidas e diretas, não sendo necessário esse tipo de separação.

### Composite

O padrão de projeto Composite não foi adotado neste sistema, uma vez que a modelagem das entidades não demanda uma estrutura hierárquica ou em árvore, típica de cenários onde objetos compostos e individuais devem ser tratados de forma uniforme.

### Facade

O padrão de projeto Facade não foi utilizado, pois não identificamos a necessidade de sua aplicação no contexto atual do sistema. A complexidade das interações entre os subsistemas é baixa, e não há cenários em que a criação de uma interface unificada traria benefícios significativos em termos de simplificação ou encapsulamento das operações.

## Flyweight

Não utilizamos o padrão Flyweight porque o sistema não utiliza uma grande quantidade de objetos repetidos que justifique o compartilhamento de instâncias para economia de memória.

## Proxy

O padrão proxy não foi utilizado porque não há necessidade de controlar o acesso a objetos e serviços de forma individualizada.

# Funcionamento do sistema

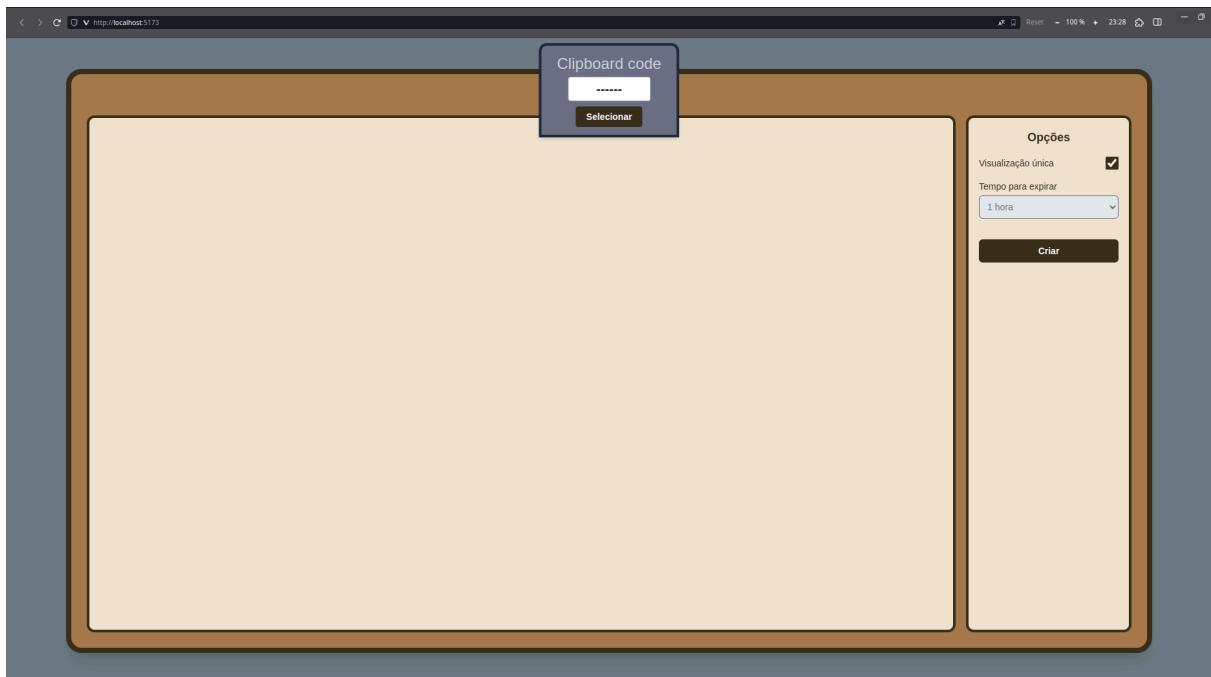


Figura 05: Primeiro acesso à página



Figura 06: Escrita do conteúdo

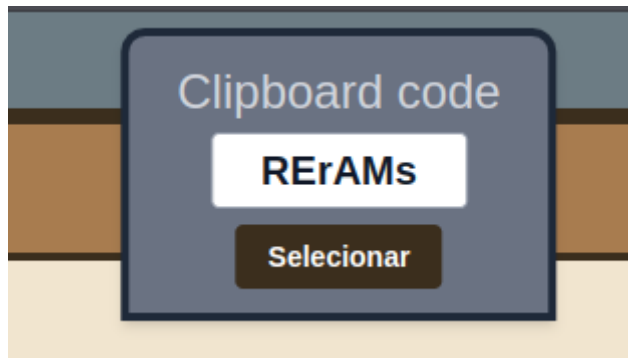


Figura 07: Geração do código



Figura 08: Código colado em outra aba, recuperando conteúdo