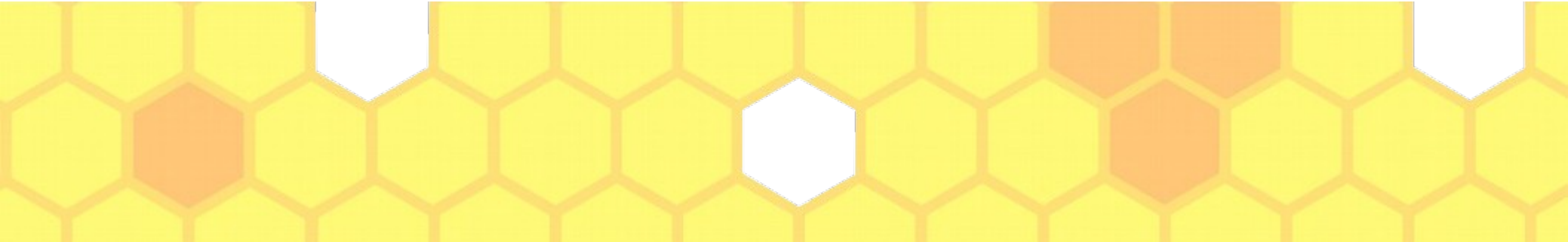




Intenção oficial

Separar a construção de um objeto complexo da sua representação de modo que o mesmo processo de construção possa criar diferentes representações.



Problema

- Imagine um objeto complexo que necessite de uma inicialização passo a passo trabalhosa de muitos campos e objetos agrupados. Tal código de inicialização fica geralmente enterrado dentro de um construtor monstruoso com vários parâmetros.

```
1  <?php
2
3  class Casa {
4
5      public function __construct($janelas, $portas, $quartos, $banheiro, $garagem, $piscina...)
6
7  }
8
9
10 $casa = new Casa(4, 2, 3, 1, TRUE, NULL, NULL, NULL, TRUE...);
11
12
```

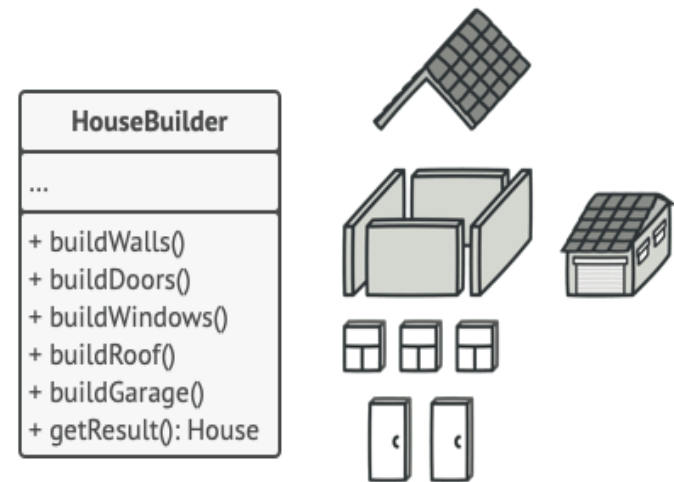
Solução

- O padrão Builder sugere que você extraia o código de construção do objeto para fora de sua classe e mova ele para objetos separados chamados builders.

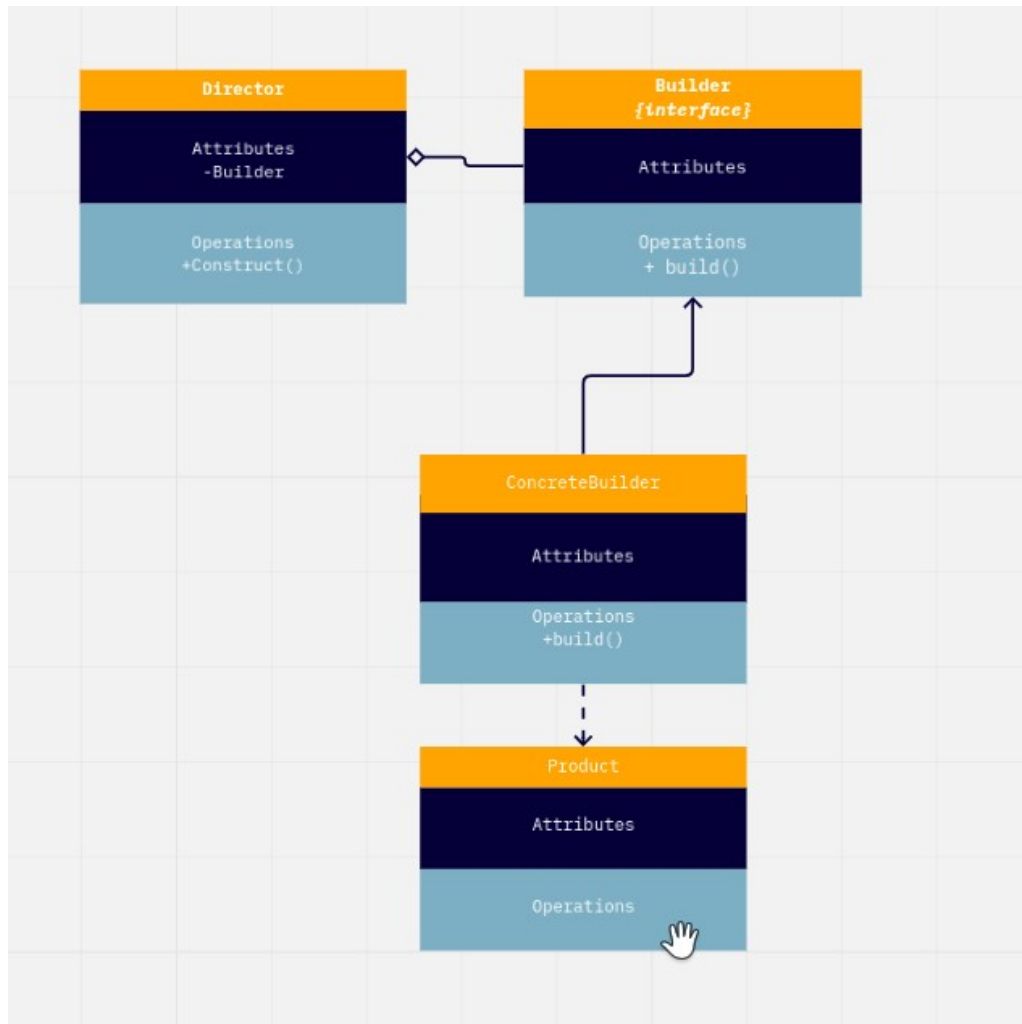


Solução

- O padrão organiza a construção de objetos em uma série de etapas (construirParedes, construirPorta, etc.)

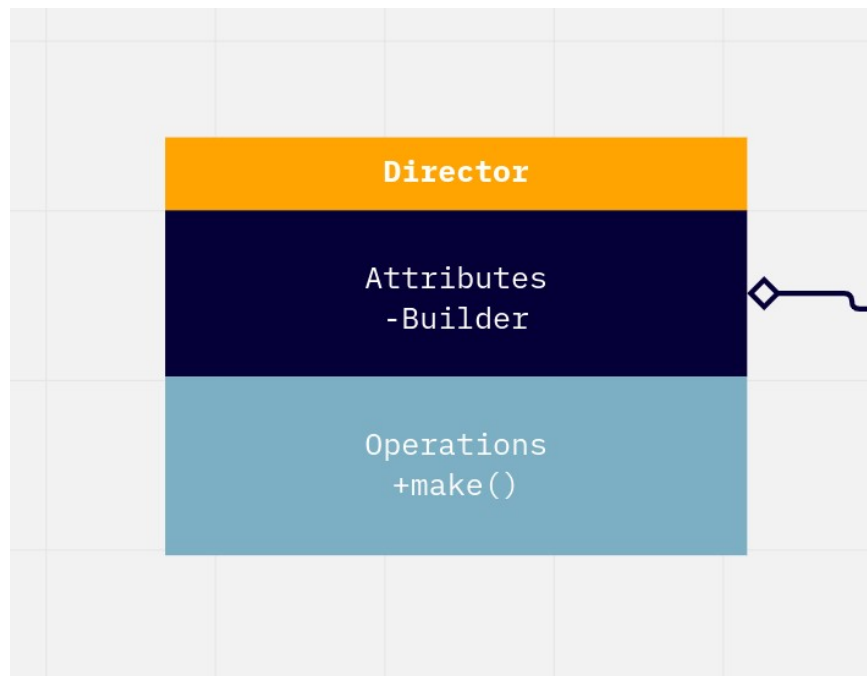


Estrutura



Director

- A classe "Director" é opcional. Ela pode definir a ordem em que as etapas de construção dos objetos são executadas.



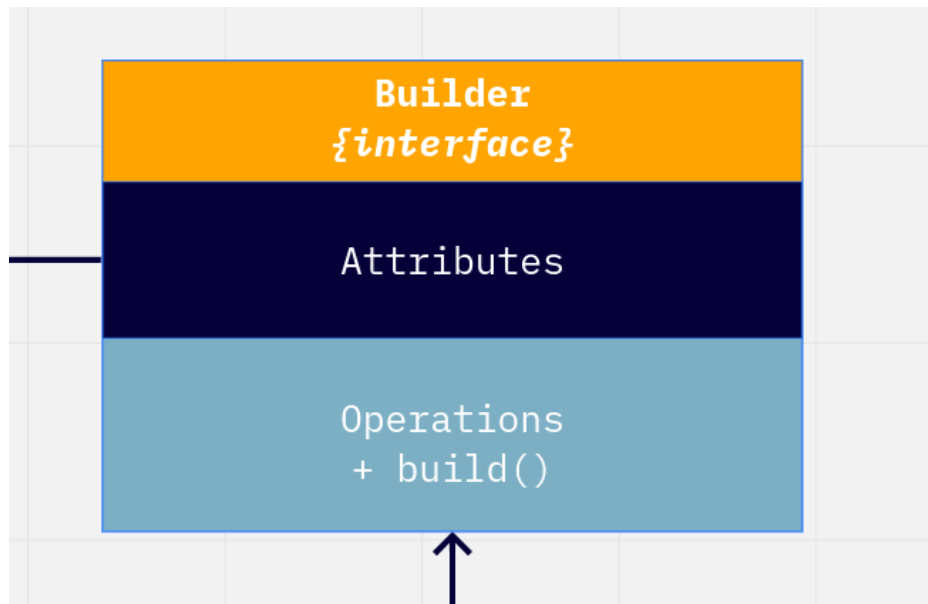
Director

- Esta é a classe que controla o construtor responsável por gerar o objeto do produto final. Um objeto Director é instanciado e seus métodos construtores são chamados. O método inclui um parâmetro para capturar objetos específicos do tipo Concrete Builder que serão então utilizados para gerar o produto (product). Dessa forma, o director, chama os métodos do concrete builder na ordem correta para gerar o objeto produto.



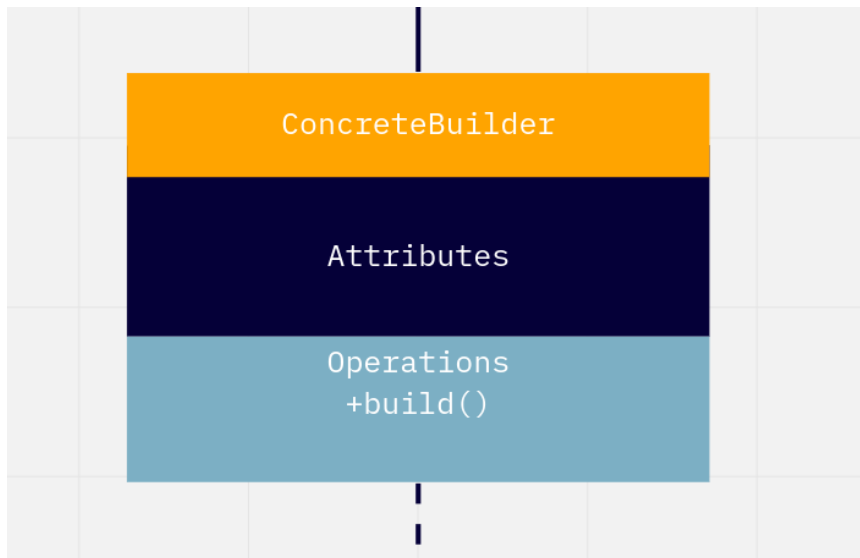
Interface Builder

- Define a interface de todos os objetos "Builder". As etapas de construção em comum são definidas aqui.



Concrete Builder

- Esta classe é responsável pela construção e pela montagem das partes do produto por meio da implementação da classe builder. Elas podem produzir produtos de tipos diferentes.



Concrete Builder

- Concrete Builder

Ela define e mantém o controle da representação que a classe cria, além de fornecer uma interface para recuperação do produto.

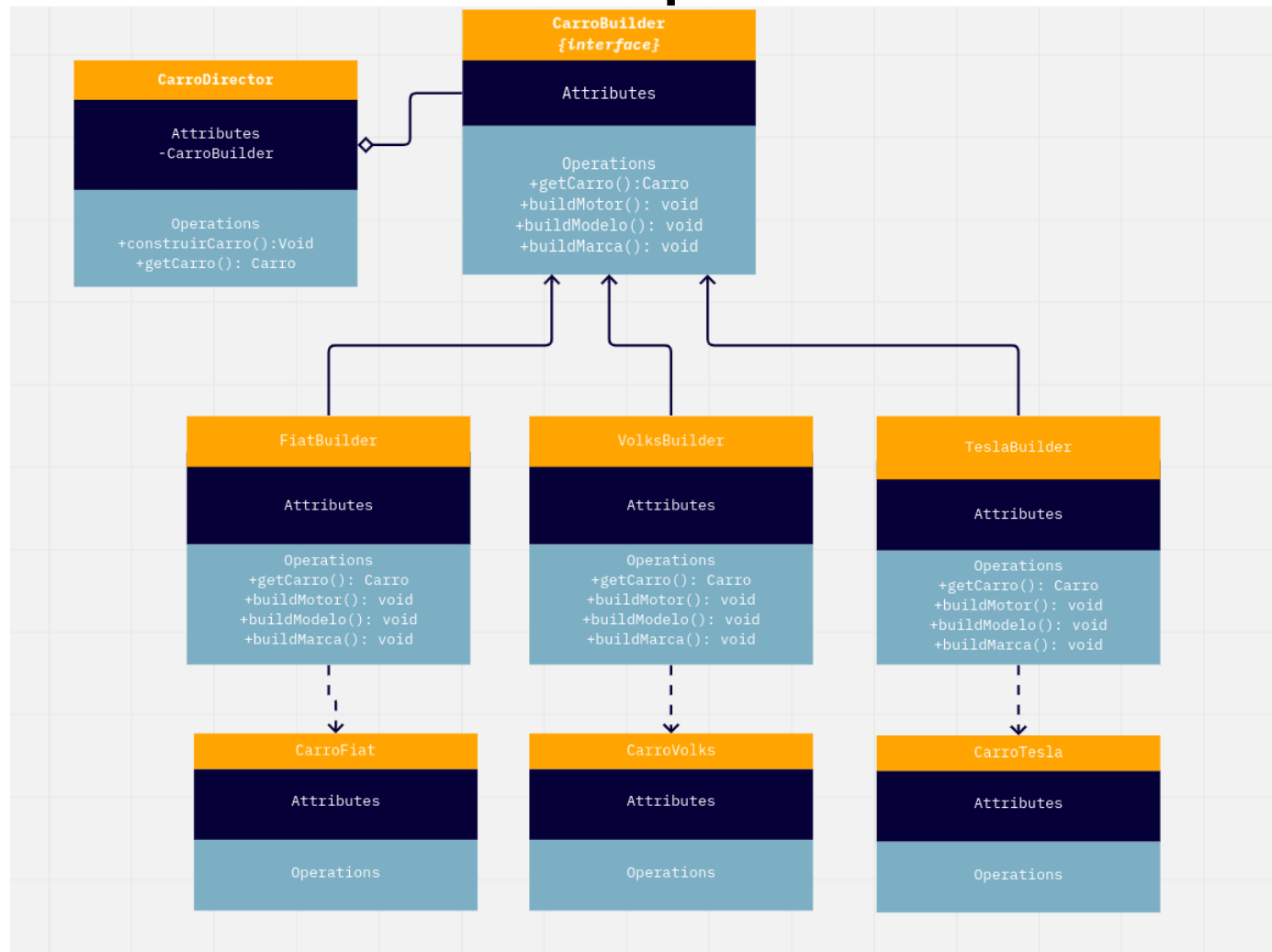


Product

- Os produtos finais são os objetos que o cliente deseja consumir. O objeto complexo construído.



Exemplo



Como implementar

- Certifique-se que você pode definir claramente as etapas comuns de construção para construir todas as representações do produto disponíveis. Do contrário, você não será capaz de implementar o padrão.



Como implementar

- Declare as etapas na interface builder.

```
<?php
namespace App\Interfaces;
use App\Carro;

Interface CarroBuilder
{
    public function getCarro(): Carro;
    public function buildMotor();
    public function buildModelo();
    public function buildAnoFabricacao();
}
```



Como implementar

- Crie uma classe builder concreta para cada representação do produto e implemente suas etapas de construção.

```
<?php
namespace App\Builders;
use App\CarroBuilder;

class TeslaBuilder extends CarroBuilder
{
    public function buildMotor()
    {
        // Operação complexa.
        $this->carro->motor = 'Elétrico';
        return $this;
    }

    public function buildFabricante()
    {
        // Operação complexa.
        $this->carro->fabricante = 'Tesla';
        return $this;
    }

    public function buildModelo()
    {
        // Operação complexa.
        $this->carro->modelo = 'Model S';
        return $this;
    }
}
```

Como implementar

- Pense em criar uma classe diretor. Ela pode encapsular várias maneiras de construir um produto usando o mesmo objeto builder.

```
<?php
namespace App\DirectorBuilder;
use App\Interfaces\CarroBuilder;
class CarroDirector {
    protected $builder;
    protected $carro;

    public function __construct(CarroBuilder $builder)
    {
        $this->builder = $builder;
    }

    public function construirCarro()
    {
        $this->builder
            ->buildMotor()
            ->buildFabricante()
            ->buildModelo()
            ->buildAnoFabricacao();
        return $this;
    }

    public function getCarro()
    {
        return $this->builder->getCarro();
    }
}
```


Como implementar

- Pense em criar uma classe diretor. Ela pode encapsular várias maneiras de construir um produto usando o mesmo objeto builder.

```
<?php
namespace App\DirectorBuilder;
use App\Interfaces\CarroBuilder;

class CarroDirector {
    protected $builder;
    protected $carro;

    public function __construct(CarroBuilder $builder)
    {
        $this->builder = $builder;
    }

    public function construirCarro()
    {
        $this->builder
            ->buildMotor()
            ->buildFabricante()
            ->buildModelo()
            ->buildAnoFabricacao();
        return $this;
    }

    public function getCarro()
    {
        return $this->builder->getCarro();
    }
}
```

Como implementar

- O código cliente cria tanto os objetos do builder como do diretor. Antes da construção começar, o cliente deve passar um objeto builder para o diretor. Geralmente o cliente faz isso apenas uma vez, através de parâmetros do construtor do diretor. O diretor usa o objeto builder em todas as construções futuras. Existe uma alternativa onde o builder é passado diretamente ao método de construção do diretor.

```
* @return mixed
*/
public function handle()
{
    $fiatBuilder = new FiatBuilder();
    $teslaBuilder = new TeslaBuilder();

    $directorCarro = new CarroDirector($fiatBuilder);
    $carroFiat = $directorCarro
        ->construirCarro()
        ->getCarro();

    $directorCarro = new CarroDirector($teslaBuilder);
    $carroTesla = $directorCarro
        ->construirCarro()
        ->getCarro();
}
```

Como implementar

- O resultado da construção pode ser obtido diretamente do diretor apenas se todos os produtos seguirem a mesma interface. Do contrário o cliente deve obter o resultado do builder.

```
* @return mixed
*/
public function handle()
{
    $fiatBuilder = new FiatBuilder();
    $teslaBuilder = new TeslaBuilder();

    $directorCarro = new CarroDirector($fiatBuilder);
    $carroFiat = $directorCarro
        ->construirCarro()
        ->getCarro();

    $directorCarro = new CarroDirector($teslaBuilder);
    $carroTesla = $directorCarro
        ->construirCarro()
        ->getCarro();
}
```



Prós e contras

- Você pode construir objetos passo a passo, adiar as etapas de construção ou rodar etapas recursivamente.
- Você pode reutilizar o mesmo código de construção quando construindo várias representações de produtos.
- Princípio de responsabilidade única. Você pode isolar um código de construção complexo da lógica de negócio do produto.
- A complexidade geral do código aumenta uma vez que o padrão exige criar muitas classes novas.



Referências

- <https://refactoring.guru/pt-br/design-patterns/builder>
- <https://brizeno.wordpress.com/category/padroles-de-projeto/builder/>
- <https://www.devmedia.com.br/design-patterns-aplicando-os-padroles-builder-singleton-e-prototype/31023>
- <https://gbbigardi.medium.com/arquitetura-e-desenvolvimento-de-software-parte-4-builder-848867107f71>
- https://sourcemaking.com/design_patterns/builder

