Introdução

Para introdução ao ***Padrão de Projeto Criacional Builder*** será fornecido 4 links da internet para executarem as tarefas sugeridas na linguagem java. Com base no exposto em cada link, faça um resumo da proposta apresentando o padrão *builder*, e o seu entendimento sobre o assunto. A entrega deverá conter o arquivo doc preenchido referente cada aplicação, código fonte com a implementação sugerida em cada site.

> **Tip**
>
> Todos os códigos, informações de build e informações da tarefa podem ser encontradas aqui, para saber como rodar via CLI do java, leia o arquivo mise.toml

Link 1: Refactor.guru

1. ***Definição do padrão builder:***
   O *Builder* é um padrão de projeto criacional que permite a você construir objetos complexos passo a passo. O padrão permite que você produza diferentes tipos e representações de um objeto usando o mesmo código de construção.
2. ***Modelagem padrão:***
   - A interface **Builder** declara etapas de construção do produto que são comuns a todos os tipos de *builders*.
   - **Builders Concretos** provém diferentes implementações das etapas de construção. *Builders* concretos podem produzir produtos que não seguem a interface comum.
   - **Produtos** são os objetos resultantes. Produtos construídos por diferentes *builders* não precisam pertencer a mesma interface ou hierarquia da classe.
   - A classe **Diretor** define a ordem na qual as etapas de construção são chamadas, então você pode criar e reutilizar configurações específicas de produtos.
   - O **Cliente** deve associar um dos objetos *builders* com o diretor. O diretor então usa aquele objeto *builder* para todas as futuras construções.
3. ***Como implementar o padrão:***
   1. 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.
   2. Declare essas etapas na interface *builder* base.
   3. Crie uma classe *builder* concreta para cada representação do produto e implemente suas etapas de construção.

    4. Pense em criar uma classe diretor. Ela pode encapsular várias maneiras de construir um produto usando o mesmo objeto *builder*.
    5. 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.
    6. 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*.
4. ***Problema apresentado:***
   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. Ou pior: espalhado por todo o código cliente.
   Por exemplo, vamos pensar sobre como criar um objeto `Casa`. Para construir uma casa simples, você precisa construir quatro paredes e um piso, instalar uma porta, encaixar um par de janelas, e construir um teto. Mas e se você quiser uma casa maior e mais iluminada, com um jardim e outras miudezas (como um sistema de aquecimento, encanamento, e fiação elétrica)?
5. ***Solução apresentada:***
   O padrão *Builder* sugere que você extraia o código de construção do objeto para fora de sua própria classe e mova ele para objetos separados chamados *builders*.
   O padrão organiza a construção de objetos em uma série de etapas (`construirParedes`, `construirPorta`, etc.). Para criar um objeto você executa uma série de etapas em um objeto *builder*. A parte importante é que você não precisa chamar todas as etapas. Você chama apenas aquelas etapas que são necessárias para a produção de uma configuração específica de um objeto.
   Algumas das etapas de construção podem necessitar de implementações diferentes quando você precisa construir várias representações do produto. Por exemplo, paredes de uma cabana podem ser construídas com madeira, mas paredes de um castelo devem ser construídas com pedra.
   Nesse caso, você pode criar diferentes classes construturas que implementam as mesmas etapas de construção, mas de maneira diferente. Então você pode usar esses *builders* no processo de construção (i.e, um pedido ordenado de chamadas para as etapas de construção) para produzir diferentes tipos de objetos.
   Você pode ir além e extrair uma série de chamadas para as etapas do builder que você usa para construir um produto em uma classe separada chamada *diretor*. A classe diretor define a ordem na qual executar as etapas de construção, enquanto que o *builder* provê a implementação dessas etapas.
   A classe diretor não é estritamente necessária, mas implementa-lá pode poupar muito tempo com rotinas de construções repetitivas, para que você possa reutilizá-las em qualquer lugar do seu programa, além disso, a classe diretor esconde completamente os detalhes da construção do produto do código cliente.
6. ***Conclusão:***
   Com os exemplos do refactor.guru, ficou fácil de conseguir enxergar as estruturas

e aplicações do *builder*, podemos concluir pelos exemplos dados que é um excelente padrão quando a necessidade de passar diversas configurações ou coisas parecidas para um objeto, "encapsulando" todas estas opções em métodos ou construtores, para facilitar a criação do objeto.

Link 2: Programando em Java

1. ***Definição do padrão builder:***
   O *Builder* tem como objetivo eliminar a complexidade na criação de objetos e também deixar mais intuitivo este processo. Neste tutorial será apresentada a forma mais básica de como podemos utilizar o padrão *Builder*.
2. ***Modelagem padrão:***
   O link não apresenta uma *Modelagem Padrão*
3. ***Como implementar o padrão:***
   O link não apresenta um *Como implementar o padrão*
4. ***Problema apresentado:***
   O problema apresentado pelo link, é um tanto quanto vago, por não realmente especificar o problema e só demonstrar o "código resolvendo", mas basicamente, é sobre o cadastro de `Pessoas`, estas pessoas, possuem `Endereços` e `Telefones` à serem cadastrados também, cada um com sua respectiva classe.
5. ***Solução apresentada:***
   É definido então, um `Pessoa Builder`, usada como a classe *builder*. No exemplo é usado uma abordagem diferente da do refactor guru, a abordagem usada, é por meio de um método estático chamado "builder", uma abordagem diferente que só muda a aparência do código, pois a funcionalidade é a mesma.
6. ***Conclusão:***
   O site Programando em Java apresenta exemplos mais práticos, mas ainda sim didáticos, sem focar muito na teoria de como estruturar o padrão, de acordo com eles é dividir a criação de um objeto em vários nomes, por exemplo, ao invés de definir um `setter` para cada parâmetro, definir um `setter` que abrangem um grupo de parâmetros, por exemplo, nome e sobrenome, seriam inseridos através de um `setNomeCompleto()`.

Link 3: Medium

1. ***Definição do padrão builder:***
   Separar a construção de um objeto complexo de sua representação para que os mesmos processos de construção possam criar representações diferentes.
   Em outras palavras, o padrão *Builder* é útil para criar objetos complexos que possuem várias partes.
2. ***Modelagem padrão:***
   - Director — Gera um objeto utilizando a interface `Builder`.

- **Builder** — Determina uma interface para criar as partes do objeto **Product**, definindo todos os passos que devem ser executados para criá-lo.
- **ConcreteBuilder** — Realiza a montagem das partes do produto através do `Builder`.
- **Product** — Representa o objeto completo a ser construído pelo `Builder`, gerando assim o resultado final.

3. ***Como implementar o padrão:***
   O link não apresenta um *Como implementar o padrão*
4. ***Problema apresentado:***
   O problema é a dificuldade de criar novos sanduíches no sistema.
5. ***Solução apresentada:***
   A solução foi criar uma *builder* para os sanduíche, para construir os sanduíches de forma mais fácil. Temos a Classe `Sandwich` que seria a representação de **Product**, temos o `SandwichBuilder` que seria o **ConcreteBuilder**, também temos o `Director` e `IBuilder` que pelos nomes acredito que dispensam apresentações (**Director** e **Builder**).
6. ***Conclusão:***
   A conclusão que consegui extrair do Medium é bem simples, pois ele simplifica bastante os conceitos, usando os `Directors` ao invés de métodos ou construtores, foi criado um `Director` para criar os diversos tipos de sanduíches, este `Directos` possui diversos métodos que utilizam o `Builder` para gerar um objeto, como por exemplo `CreateBigMac()` ou `CreateBigTasty()`.

Link 4: GeeksForge

1. ***Definição do padrão builder:***
   O *Builder Design Pattern* é um padrão criacional usado em design de software para construir um objeto complexo passo a passo. Ele permite a construção de um produto de forma passo a passo, onde o processo de construção pode mudar com base no tipo de produto que está sendo construído. Este padrão separa a construção de um objeto complexo de sua representação, permitindo que o mesmo processo de construção crie representações diferentes.
2. ***Modelagem padrão:***
   1. **Product** - O Produto é o objeto complexo que o padrão *Builder* é responsável por construir.
      - Ele pode consistir em vários componentes ou partes, e sua estrutura pode variar de acordo com a implementação.
      - O Produto normalmente é uma classe com atributos que representam as diferentes partes que o Construtor constrói.
   2. **Builder** - O `Builder` é uma interface ou uma classe abstrata que declara as etapas de construção para criar um objeto complexo.
      - Normalmente inclui métodos para construir partes individuais do produto.

    - Ao definir uma interface, o `Builder` permite a criação de diferentes construtores de concreto que podem produzir variações do produto.
3. **Concrete Builder** - As classes `ConcreteBuilder` implementam a interface `Builder`, fornecendo implementações específicas para construir cada parte do produto.
    - Cada `ConcreteBuilder` é personalizado para criar uma variação específica do produto.
    - Ele monitora o produto que está sendo construído e fornece métodos para definir ou construir cada parte.
4. **Director** - O `Diretor` é responsável por gerenciar o processo de construção do objeto complexo.
    - Ele colabora com um `Construtor`, mas não conhece os detalhes específicos sobre como cada parte do objeto é construída.
    - Ele fornece uma interface de alto nível para construir o produto e gerenciar as etapas necessárias para criar o objeto complexo.
5. **Client** - O `Cliente` é o código que inicia a construção do objeto complexo.
    - Ele cria um objeto `Builder` e o passa ao `Director` para iniciar o processo de construção.
    - O `Cliente` pode retirar o produto final do Construtor após a conclusão da construção.

3. ***Como implementar o padrão:***
    1. **Crie a Classe de** `Product`: Defina o objeto (produto) que será construído. Esta classe contém todos os campos que compõem o objeto.
    2. **Crie a Classe** `Builder`: Esta classe terá métodos para definir as diferentes partes do produto. Cada método retorna o próprio *builder* para permitir o encadeamento de métodos.
    3. **Adicione um método de Construção**: na classe *builder*, adicione um método chamado `build()` (ou similar) que monta o produto e retorna o objeto final.
    4. **Usar o** `Director` **(Opcional)**: Se necessário, você pode criar uma classe diretora para controlar o processo de construção e decidir a ordem em que as peças serão construídas.
    5. **O cliente usa o construtor**: o cliente usará o construtor para definir as peças desejadas passo a passo e chamar o `build()` método para obter o produto final.
4. ***Problema apresentado:***

    Você tem a tarefa de implementar um sistema para construir computadores personalizados. Cada computador pode ter configurações diferentes com base nas preferências do usuário. O objetivo é fornecer flexibilidade na criação de computadores com CPUs, RAM e opções de armazenamento variadas.
5. ***Solução apresentada:***

    O site demonstra uma solução em código java, demonstrando o padrão de design ***Builder***, onde a classe **Computer** é o produto, Builder é a interface,

GamingComputerBuilder é o **Builder**, ComputerDirector é o **Director** e o Client monta o produto usando o construtor e o diretor.

6. ***Conclusão:***

   Neste site, é dado um passo a passo de como se implementar totalmente o *builder*, porém de uma forma genérica, basicamente implementando basicamente a "montagem de um computador", o computador possui ram, cpu e storage, e ao invés de criar um construtor passando todos os parâmetros, é criado um método para passar cada parâmetro ao objeto, e depois um método para construí-lo