DESIGN PATTERNS: UMA EXPLICAÇÃO GERAL DO CONCEITO E UM APROFUNDAMENTO NO PADRÃO BUILDER

Você já percebeu padrões de implementação ou formas de resolver problemas se repetirem no projeto e codificação de diversos softwares no qual você foi um dos programadores? Caso sua resposta seja sim, então há uma grande chance desses padrões se tratarem de um dos diversos padrões descritos nos *Design Patterns* (Padrões de Projeto, em uma tradução adaptada). Caso sua resposta seja não, após ler o conteúdo abaixo e se aprofundar mais nos estudos de *Design Patterns*, você provavelmente perceberá esses padrões nos seus sistemas e projetos, embora não tenha percebido eles antes apenas por não ter em mente que esses padrões existem.

Explicando em termos simples, os *Design Patterns* tratam-se de padrões que se repetem frequentemente no projeto e codificação de quase todos os softwares, mesmo que cada um desses softwares busque resolver problemas completamente diferentes. Os *Design Patterns* tratam-se disso: soluções generalistas para problemas que surgem de forma recorrente ao longo do desenvolvimento de softwares. Eles não foram inventados por alguém, mas sim percebidos por alguém, afinal são padrões que aparecem naturalmente em projetos de software que possuam um bom planejamento.

Para entendermos melhor o significado de Padrões de Projeto (*Design Patterns*), podemos recorrer um pouco à linguística. Na língua portuguesa, o termo "padrão" assume diversos significados diferentes, e é necessário especificar qual sentido significado essa palavra assume em Padrões de Projeto. Na língua inglesa, *pattern* trata-se de algo que se repete, que mantém um padrão. Exemplo: o piso de porcelanato decorado de uma casa seguirá um mesmo *pattern*, um mesmo modelo, ou seja, um mesmo padrão, pois ele se repetirá diversas e diversas vezes (observe a imagem abaixo). Portanto, o significado que devemos assumir no termo "padrão" em Padrões de Projeto é justamente esse: algo que se repete seguindo um modelo.

Fonte da imagem: https://www.leroymerlin.com.br

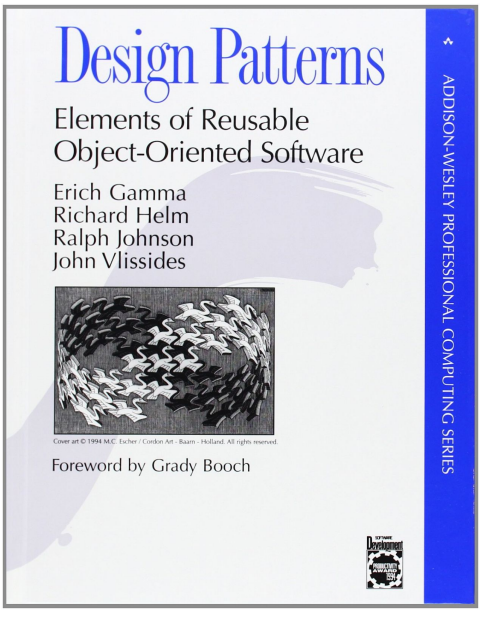
Benefícios dos Design Patterns

Os *Design Patterns* são muito importantes para o processo de desenvolvimento de softwares. Conheça os principais benefícios que eles proporcionam:

- Eles focam na reutilização de soluções, o que traz um ganho de produtividade e dispensa a necessidade de sempre "reinventar a roda". Mesmo em problemas que são completamente diferentes, é possível fragmentá-los e achar locais onde é possível aplicar os *Design Patterns* e obter soluções.
- Como os *Design Patterns* incluem mais de 20 diferentes padrões que podem ser utilizados, é bem fácil achar algum que se aplique ao problema ou parte do problema que você como programador precisará resolver.
- Por serem mundialmente conhecidos, qualquer programador, que também conheça sobre o assunto, conseguirá identificá-los no código de um software no qual você participou do desenvolvimento. Logo, os *Design Patterns* facilitam na leitura de código entre programadores diferentes.
- E, por fim, um dos maiores benefícios que os *Design Patterns* trazem é o de proporcionar uma melhor organização e manutenção de projetos, através de padrões já testados e aprovados pelo mercado. Como os softwares mudam/evoluem e precisam de manutenção constante, é necessário que os programadores projetem softwares de qualidade, reutilizáveis e de fácil manutenabilidade. E adivinhem só: os *Design Patterns*, se corretamente aplicados, proporcionam tudo isso.

História dos Design Patterns

Como relatado acima, os Design Patterns surgem a partir de padrões que se repetem em diversos projetos e codificação de softwares, ou seja através da experimentação ao longo do desenvolvimento de softwares, principalmente os que envolvam Programação Orientada a Objetos. Entretanto, os *Design Patterns* realmente se tornaram de conhecimento público quando quatro programadores decidiram, em um livro, estabelecer e catalogar os problemas mais comuns e as formas de resolvê-los. Esses programadores eram Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides (que passaram a serem conhecidos por *Gang of Four*, ou "Gangue dos Quatro") e o livro se tratava do *Design Patterns: Elements of Reusable Object – Oriented Software* (Padrões de Projetos: Elementos de Software Orientado a Objetos Reutilizáveis), publicado pela primeira vez em 1994. Esse livro se tornou então a maior referência mundial para *Design Patterns*, e até hoje é atual e muito utilizado.



Categorias de Design Patterns

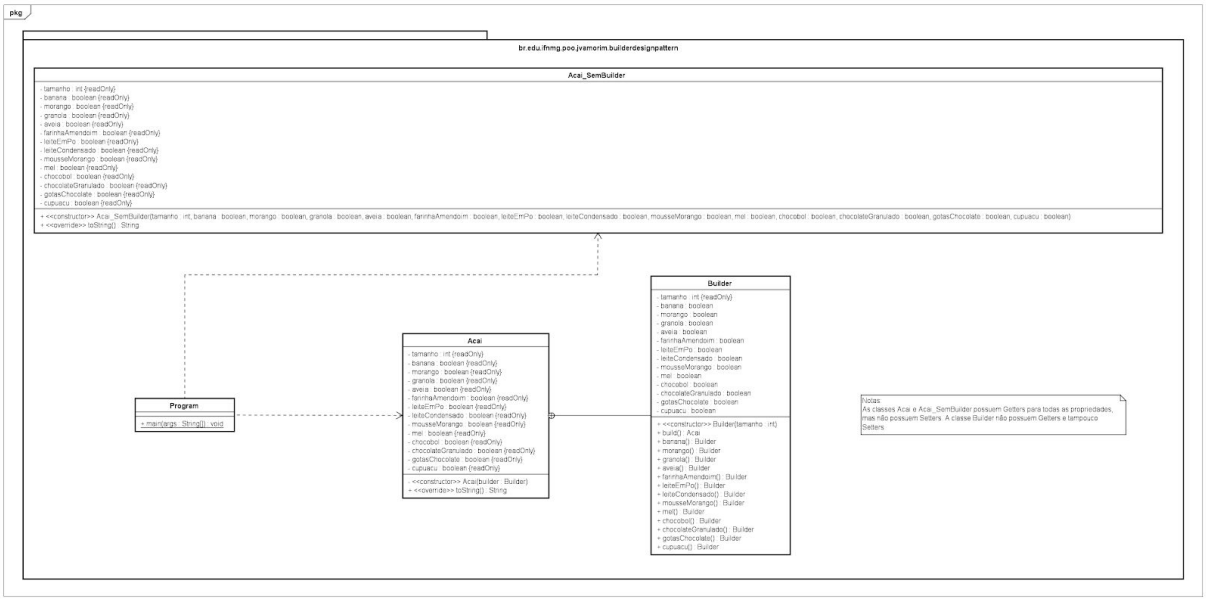
1. **Creational Patterns**: são os padrões de criação e tratam da construção do objeto e de referência. Nele, constam: *Abstract Factory, Builder, Factory Method, Prototype, Singleton.*
2. **Structural Patterns**: são os padrões estruturais que tratam da relação entre os objetos e como eles interagem entre si para formarem objetos grandes e complexos. Nele, constam: *Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy.*
3. **Behavioral Patterns**: são os padrões comportamentais que tratam da comunicação entre os objetos, especialmente em termos de responsabilidade e de algoritmo. Nele, constam: *Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.*

Builder: o Design Pattern que melhora a instanciação de objetos complexos

O *Builder* trata-se de um padrão de projeto que faz parte da categoria de *Creational Patterns*, ou seja, ele tratará da construção do objeto. O objetivo do Builder é melhorar e facilitar a construção de objetos cuja construção seja complexa, e ele fará isso através de uma construção passo-a-passo do objeto.

Para uma explicação mais detalhada a respeito do Builder e do código que é mostrado abaixo, incluindo detalhes de implementação de cada classe, acesse o seguinte vídeo que é um complemento a esse presente texto: https://youtu.be/1d_lZ8eLDlU .

Observe o seguinte diagrama de classes:



Caso tenha dificuldades na visualização, a imagem também está disponível no seguinte endereço: https://github.com/JV-Amorim/BuilderDesignPattern/blob/master/diagrama-de-classes.png

A classe *Acai_SemBuilder* possui as mesmas funcionalidades da classe *Acai*, porém

sem utilizar o *Builder*. Esse fato torna a instanciação de um objeto dessa classe muito mais complicada em relação à classe *Acai*, que possui o *Builder*. Para perceber isso, basta tentar fazer a leitura de um objeto abaixo instanciado sem utilizar o *Builder* e um instanciado utilizando o *Builder*. É muito fácil se perder e não entender o que está acontecendo na instanciação do objeto sem *Builder*, pois a quantidade de argumentos passados no construtor é muito grande. Para saber o que cada argumento representa é necessário recorrer ao código do construtor, abrindo a classe e a documentação, o que é bem inviável e contraprodutivo.

Além disso, teremos de definir valores para propriedades que nem sequer serão usadas, já que neste contexto, um açaí ter como valor *false* uma propriedade que representa um adicional, simplesmente significa que esse adicional/propriedade será ignorado durante as etapas que envolvem a produção desse açaí no mundo real.

Observe abaixo a construção de dois pedidos de açaí, sem a utilização do *Builder* e com a utilização do *Builder*. No final o resultado é de fato o mesmo, mas a complexidade para criar sem o *Builder* é muitíssima maior. Qualquer programador que fosse ler o código abaixo, teria muito mais dificuldade para ler os trechos onde o *Builder* não é utilizado.

```
// Pedido 1: um açaí de 300 mL, com banana e leite em pó.
// Pedido 2: um açaí de 700 mL, com cupuaçu.


System.out.println("SEM UTILIZAR O BUILDER:\n");

Acai_SemBuilder acaiPedido1_SemBuilder = new Acai_SemBuilder(300, true, false,
false, false, false, true, false, false, false, false, false, false, false);

Acai_SemBuilder acaiPedido2_SemBuilder = new Acai_SemBuilder(700, false, false,
false, false, false, false, false, false, false, false, false, false, true);

System.out.println(acaiPedido1_SemBuilder);
System.out.println(acaiPedido2_SemBuilder);


System.out.println("UTILIZANDO O BUILDER:\n");

Acai acaiPedido1_ComBuilder = new Acai.Builder(300).banana().leiteEmPo().build();
Acai acaiPedido2_ComBuilder = new Acai.Builder(700).cupuacu().build();

System.out.println(acaiPedido1_ComBuilder);
System.out.println(acaiPedido2_ComBuilder);
```

Output:

```
SEM UTILIZAR O BUILDER:

AÇAÍ - 300 mL
Adicionais:
- Banana
- Leite em pó

AÇAÍ - 700 mL
Adicionais:
- Cupuaçu

UTILIZANDO O BUILDER:

AÇAÍ - 300 mL
Adicionais:
- Banana
- Leite em pó

AÇAÍ - 700 mL
Adicionais:
- Cupuaçu
```

Não há diferenças no resultado ao utilizar ou não o *Builder*, como explicado mais acima e no vídeo). A diferença se dará realmente na legibilidade, facilidade de manutenção e alteração do código, entre outros benefícios.

Link do repositório no GitHub: https://github.com/JV-Amorim/BuilderDesignPattern **.**

Fontes utilizadas:

- https://www.hostgator.com.br/blog/design-patterns-e-seus-beneficios/
- https://refactoring.guru/pt-br/design-patterns/builder
- http://luizricardo.org/2013/08/construindo-objetos-de-forma-inteligente-builder-pattern-e-fluent-interfaces/