Design Patterns

> Design Patterns são soluções já consolidadas para problemas comuns em software design, especificamente design de código.

Assim Design Patterns são esqueletos de ideias que já foram utilizadas diversas vezes em outros projetos e que especificam uma solução para problemas comuns quando estamos desenvolvendo nosso código. Dessa forma podemos utilizar esses padrões para melhorar a qualidade do nosso código e focar especificamente nos detalhes do problemas que estamos querendo resolver.

Design Patterns são muitas vezes confundidos com algoritmos prontos que adicionamos a nosso código. Na real eles são principalmente conceitos arquiteturais que podem definir toda a comunicação do seu código.

Justamente por não serem algoritmos prontos, os Design Patterns tem diferentes implementações dependendo da linguagem escolhida.

Design Patterns são ferramentas incríveis que todo desenvolvedor deve revisitar de tempos em tempos na sua carreiras. A combinação de Design Pattern é uma habilidade poderossa na hora de criar código com mais qualidade e eficiência, ao mesmo tempo que flexibiliza futuras decisões no projeto.

Quando não aplicar Design Patterns

Temos que ter muito cuidado ao implementar uma solução utilizando algum Design Pattern. Atualmente várias linguagens já apresentam esses padrões implementados, então adicionar uma implementação conforme exemplo de outra linguagem, pode na real, aumentar a complexidade do código sem benefício algum ao projeto.

Pegue por exemplo o `Observer Pattern` no **c#**. O próprio c# já implementa um estrutura chamada **event** que define um tipo de Observer. Implementar o padrão nesse caso não trás benefício nenhum no projeto e aumenta a complexidade do mesmo, além de gerar confusão a programadores que estão acostumados com a especificação da linguagem.

Outro exemplo é o `Strategy Pattern` que apresenta como intensão definir uma família de algoritmos, em muitos casos pode ser simplesmente substituido como uma expressão lambda.

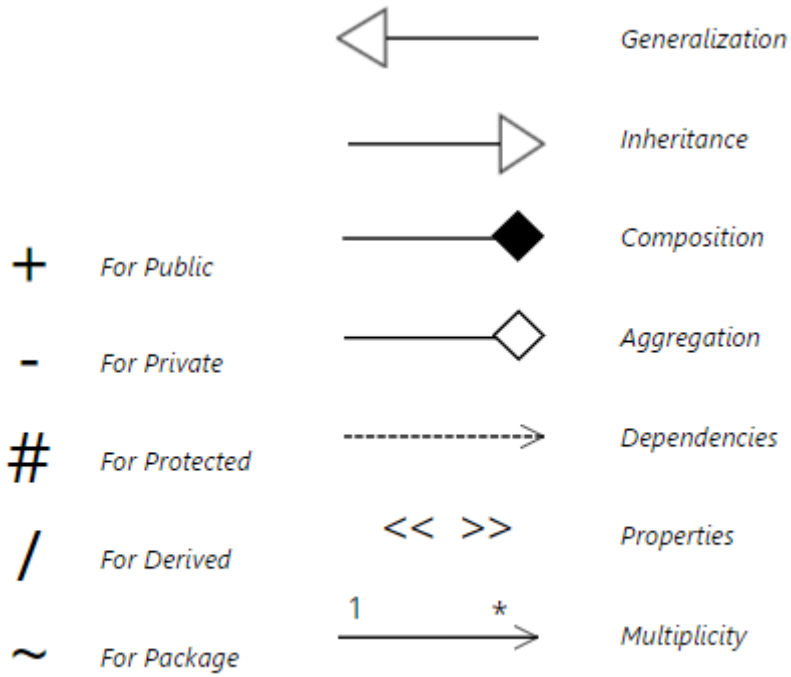
Diagrama de classes UML

> UML é uma estrutura padrão para desenvolvimento de software, bastante utilizada na diagramação de estruturas do código, bancos de dados e fluxo de código.

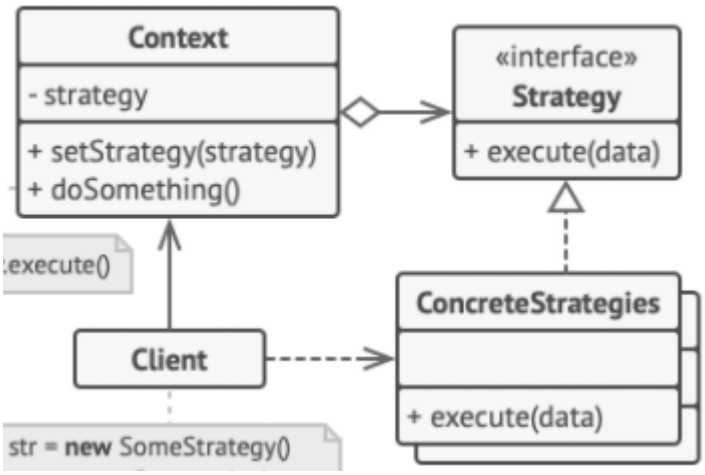
Quando falamos de Design Patters utilizamos muito UML para representar visualmente a especificação das estruturas no código. Dessa forma temos uma visão simples e direta de como podemos implementar o Design Pattern.

Ao mesmo tempo que o UML ajuda a entender as responsabilidades de cada agente no código, cada linguagem tem uma forma de implementação diferente para o Design Pattern e não deve ser levado literalmente.

Símbolos do UML



Exemplo



Classificação dos Design Patterns

Como Design Patterns são soluções para problemas comuns, eles podem ser classificados nos mais diversos tipos e formatos de especificações. As principais categorias que vemos são:

- Creational Patterns
- Structural Patterns
- Behavioral Patterns

Dependendo do domínio de problemas que estamos querendo resolver também temos vários outras categorias. Como Design Patterns focados em Performance, Design Patterns para computação gráfica e acesso a GPU, Design Patterns para computação distribuida e muitos outros.

Creational Patterns

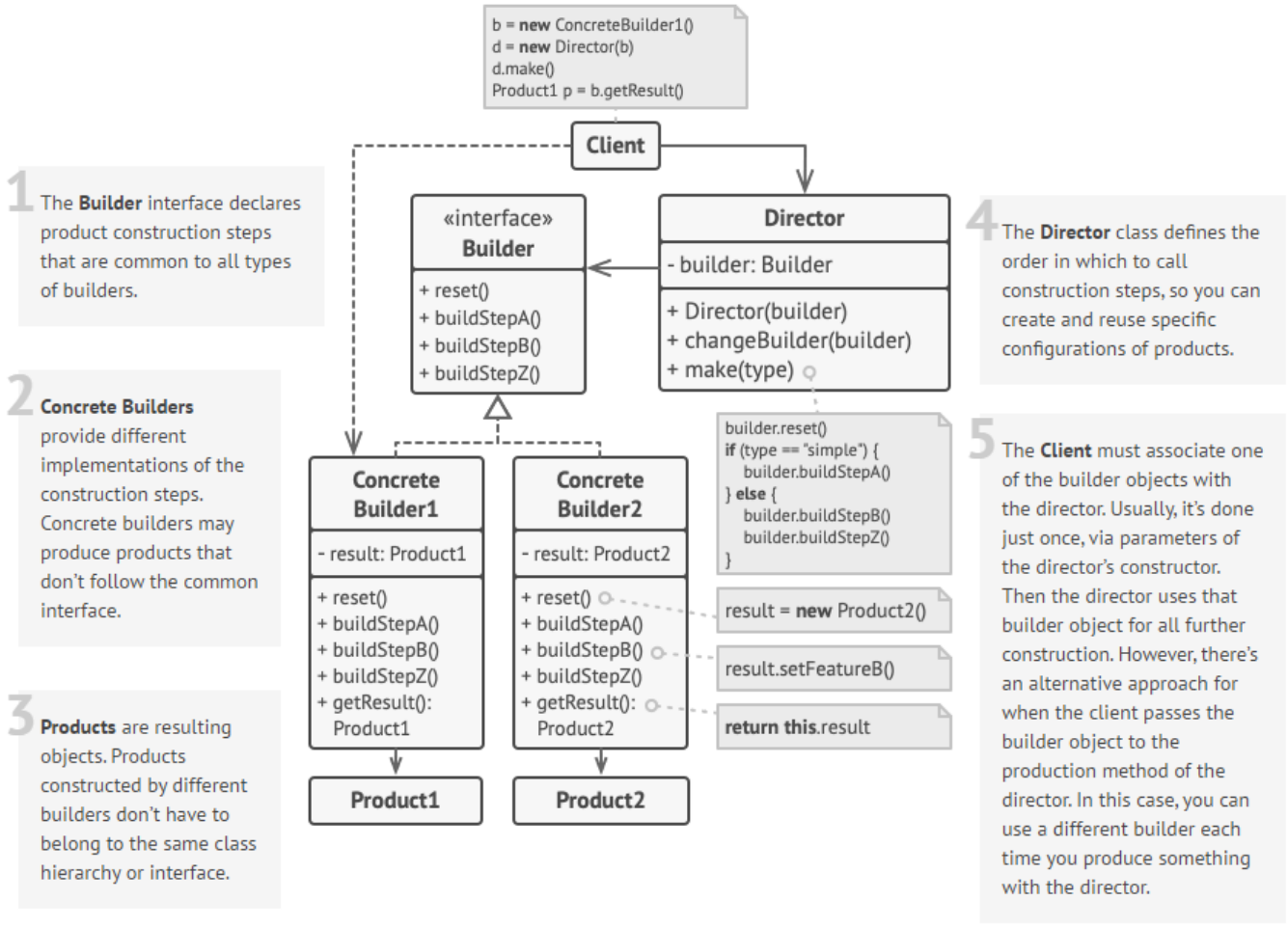
Provê mecanismos de criação de objetos que aumentam a flexibilidade e o reuso de código existem.

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

Builder

Builder Pattern é destinado a resolver a construção de objetos complexos permitindo produzir diferentes resultados utilizando o mesmo código de construção.

Builder é fortemente implementado utilizando **Fluent API** facilitando a legibilidade do código.



Builder Pattern pode ser utilizado nos mais diversos tipos de problemas como:

- Criação de objetos para testes
- Criação de objetos complexos que são montados através de várias interações
- Validação na criaçaõ dos objetos

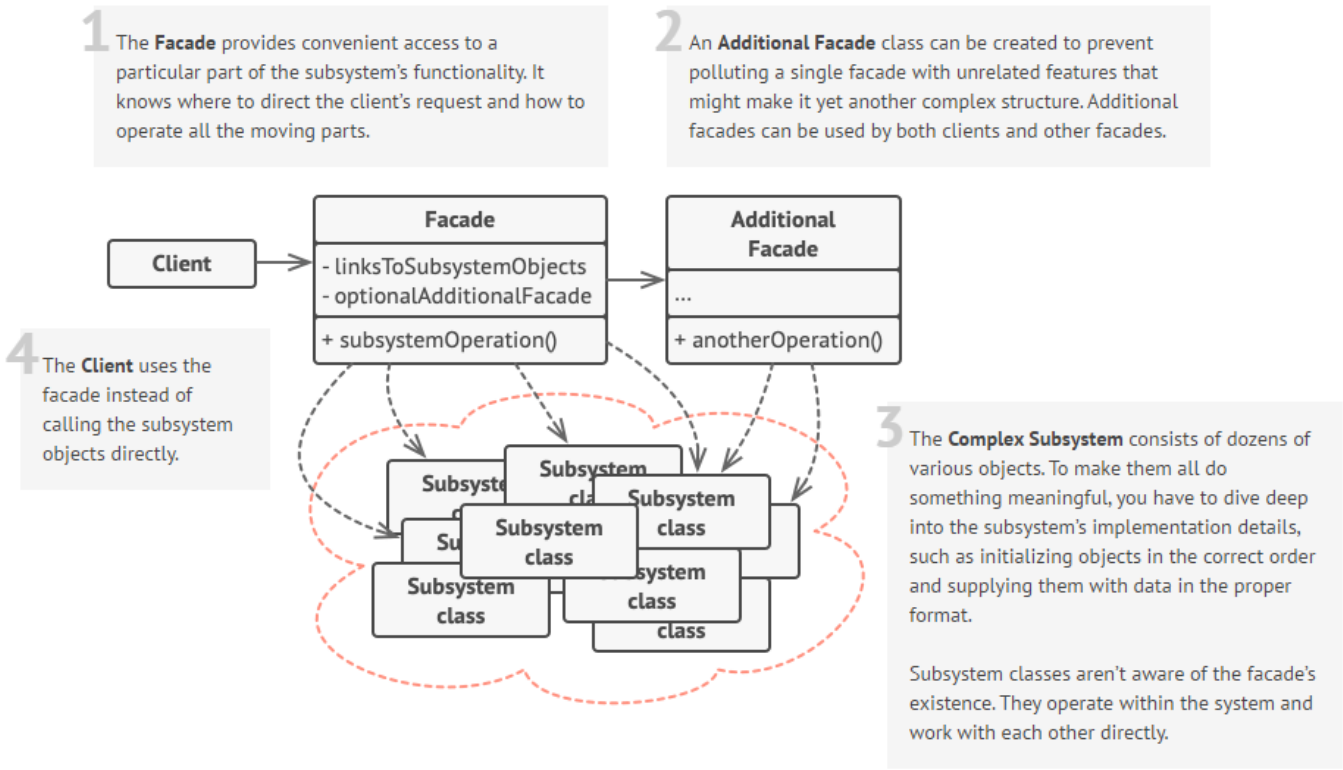
Structural Patterns

Provê mecanismos para montar classes e objetos em grandes estruturas, enquanto ainda mantem flexibilidade e eficiência.

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Facade Pattern

Facade Pattern provê uma interface simplificada para uma biblioteca, framework ou qualquer outro tipo de conjunto de classes complexo.



Behavioral Patterns

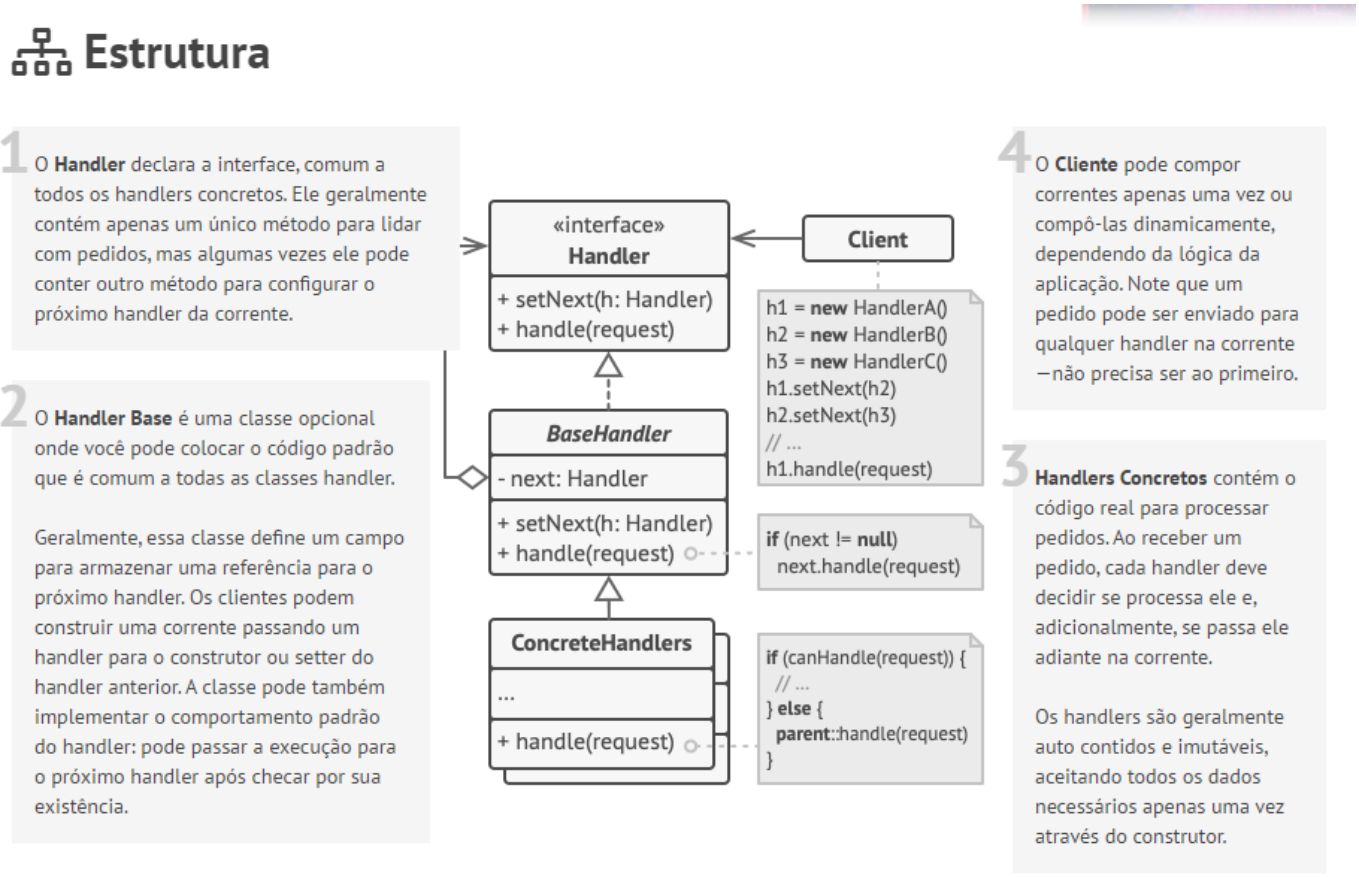
Define uma comunicação efetiva e atribui responsabilidades entre os objetos e classes.

- Chain of Responsibility
- Monad
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy

- Template
- Visitor

Chain of Responsability Pattern

Chain of Responsability Pattern é uma especificaçaõ estrutural que garante a passagem de requisições em uma cadeia de validações. Cada **Handler** (etapa da cadeia) é responsável por processar a requisição de acordo com o parâmetros de entrada de cada etapa.



Exemplos

No folder **ex_1** temos o exemplo da implementação do **Builder Pattern**.

No folder **ex_2** temos um exemplo de implementação do **Facade Pattern** na solução da criação de um cliente HTTP para usuário. Podemos perceber como vários subsystems são utilizados e temos apenas uma chamada de método pela API pública da classe `UserClient`.

No folder **ex_3** temos um exemplo da implementaçaõ do **Chain of Responsability Pattern** na solução de um sistema de rotas em uma aplicação.

Glossário

Família de algoritmos

Conjuntos de algoritmos que tem por finalidade implementar uma solução para o mesmo problema.

Exemplo: Tracar uma rota no mapa, podemos ter rotas considerando que o usuário estará a pé, de carro, de bicicleta e assim por diante.

Bibliografia

- Refactoring Guru