Command Design Pattern

em JavaScript

VICTOR SERPA DO CARMO

Neste artigo será apresentado um escopo sobre design patterns, que são soluções de reutilização de código para problemas comuns que ocorrem no desenvolvimento de um projeto, tendo como foco principal o estilo Command Pattern.

Este é um assunto muito explorado nas comunidades de desenvolvimento por se tratar de uma convenção de técnicas que ajudam na construção de um código mais otimizado, provendo também um conhecimento de melhores técnicas para resolver as necessidades dos problemas de uma aplicação.

Introdução

Design Patterns começou de fato na arquitetura há décadas atrás com o arquiteto Christopher Alexander. Ele constantemente se deparava em resolver problemas repetitivos de design relacionado a construções e cidades, percebendo então que o uso gerava certos padrões que poderiam ser otimizados. Em 1977 então, junto com Sara Ishikawa e Murray Silverstein, o arquiteto produziu uma linguagem de padrões entitulado “A Pattern Language” que ajudaria qualquer um que desejasse projetar e construir em qualquer escala.

Em torno de 30 anos atrás, alguns engenheiros de software começaram a incorporar os princípios que Christopher escreveu, o que deveria ser um guia para desenvolvedores novatos que procuravam melhorar suas habilidades de código. Pode-se então notar que os conceitos por trás de design patterns estiveram envolvidos na programação desde o principio, porém de uma maneira menos formal.

Mas afinal, o que de fato são Design Patterns na programação?

Pattern por si só é uma solução reutilizável que pode ser aplicada para problemas que ocorrem frequentemente ao escrever uma aplicação. Podemos ver patterns como

modelos de como resolver determinados problemas, tendo diversos tipos diferentes para cada situação. Assim, design patterns são melhores praticas formalizadas, desenvolvidas e testadas por vários programadores com a finalidade de resolver os diferentes problemas enfrentados no dia a dia na programação e também em outras áreas como engenharia e arquitetura.

O uso de design patterns é extremamente vantajoso e empodera muito o programador na hora de buscar as melhores soluções para o código. Uma de suas vantagens é o fato de patterns serem soluções aprovadas, provendo assim conhecimentos sólidos para resolver questões no desenvolvimento de software envolvendo técnicas já testadas que incorporam a experiência e conhecimento dos desenvolvedores que ajudaram na sua definição. Outra vantagem é o fato dessas técnicas serem extremamente reutilizáveis e robustas, prontas para poder ser adaptadas as necessidades do projeto. Outro fato que reflete a importância do uso de DP é pela sua clareza ao se expressar com o código, normalmente os design patterns possuem uma estrutura genérica pronta para uma determinada solução, o que ajuda na elegância de grandes aplicações, influenciando drasticamente na hora de sua manutenção ou colaboração com outros desenvolvedores.

Porém, como nem tudo são apenas vantagens, patterns não conseguem solucionar todos as questões dos projetos de software por não ser uma solução exata, mas podem ser um bom auxiliador na hora de seu desenvolvimento. É importante também que se tenha o conhecimento dos diferentes design para fazer um bom uso de suas principais características, fazendo assim com que o programador dedique tempo a conhecer as regras e esquemas, bem como qual o problema que cada pattern soluciona.

Categorias de Design Pattern

Design patterns podem ser divididos em categorias, onde cada design se foca em um resolver um determinado problema com base em modelos da orientação a objetos. Em JavaScript, os patterns podem ser distribuídos em 3 grandes categorias que são *Creational Design Pattern*, *Structural Design Pattern* e *Behavioral Design Patterns.*

Creational patterns são focados em lidar com mecanismos de criação de objeto de modo adequado para a situação em que se está trabalhando. Essa categoria de patterns são responsáveis por resolver a complexibilidade na criação de objetos em um projeto,

controlando assim este processo. Alguns patterns que se encaixam nessa categoria são: Constructor, Factory, Abstract, Prototype, Singleton e Builder.

Structural patterns se preocupam com a composição do objeto e normalmente identificam maneiras simples de relacionar objetos diferentes. Eles garantem que o sistema inteiro não precisa ser mudado quando se altera apenas uma parte dele. Alguns patterns que se encaixam nessa categoria são: Decorator, Facade, Flyweight, Adapter e Proxy.

Behavioral patterns basicamente se concentram na melhoria e simplificação da comunicação entre diferentes objetos em um sistema. Alguns patterns que se encaixam nessa categoria são: Command, Iterator, Mediator, Observer, e Visitor.

Anti-Patterns

Se consideramos pattern uma representação de boas praticas, anti-pattern é exatamente aquilo que temos que evitar em um código. Por isso deve-se estar atento na ocorrência deles principalmente em projetos que são mantidos por vários programadores e que estão constantemente em desenvolvimento. Em JavaScript, essas más praticas devem ser evitadas sempre, como a poluição de muitas variáveis no namespace global, usar JavaScript inline, usar *document.write* entre outras, para uma melhor aplicação e legibilidade do projeto como um todo.

Command Pattern

Foi abordado até agora o que são os design patterns, quais os seus diferentes tipos bem como suas vantagens e desvantagens, mas agora iremos abordar o foco principal desse artigo que é de fato o padrão Command Pattern. Esse padrão é responsável por encapsular a invocação, requisição ou operações em um único objeto, o que nos da a habilidade de passar por parâmetro a chamada de métodos a serem executados ao critério do objeto.

Neste pattern, um objeto é na real a representação de um verbo, diferente da maioria dos outros designs que representam substantivos, e isso é visto de forma um pouco contrariada pelos desenvolvedores, o que de fato não interfere na popularidade deste DP. O command pattern nasceu da necessidade de se emitir requisições para

objetos sem necessariamente ter conhecimento sobre a operação que esta sendo requisitada ou o ate mesmo do objeto que recebe essa requisição.

Em resumo, temos um objeto que encapsula a requisição, podendo então receber a solicitação de diferentes requisições por parâmetro, tendo também a característica de enfileirar comandos bem como a possibilidade de desfazer as operações. A ideia geral por traz do Command pattern é o fato de ele nos prover meios de separar as responsabilidades do objeto que emite uma requisição dos objetos que realmente executam e processam essa solicitação, que no caso são os métodos, delegando assim a responsabilidade para objetos diferentes, sendo basicamente uma abstração do objeto que implementa o método para o objeto que o invoca. Essas requisições são chamadas de *events (eventos)* e o código que processa a requisição é chamado de *event handlers* (manipuladores de eventos).

Para a implementação de um comando simples, é necessário uma ação junto com o objeto que invoca essa ação. Este comando irá necessitar também de uma operação que executa a ação, normalmente chamada como *run()* ou *execute()*. Dessa forma, todos os comandos que possuem a mesma interface (ou seja, possuem os mesmos parâmetros) podem ser executados através do operador *execute()* e facilmente trocados conforme necessidade, fazendo dessa, uma das grandes vantagens ao usar esse pattern.

Exemplo de uso no dia a dia

Supõe-se que estamos construindo uma aplicação que suporta as ações de recortar, copiar e colar. Estas ações podem ser disparadas de diferentes maneiras em nosso app, seja ela por um menu do sistema, ou então clicando com botão direito do mouse e copiando ou até mesmo com o tão conhecido atalho do teclado.

Command pattern nos auxilia nessa atividade ao centralizar o processamento dessas ações, tendo um objeto para cada operação. Assim, será necessário apenas um comando para processar as requisições de *recortar*, um para as requisições de *copiar*, e uma para as requisições de *colar.*

Como é feita a implementação?

Muito já se foi falado sobre os aspectos conceituais do Command pattern, mas agora será de fato apresentado o código de como se implementa esse design para podermos ter uma visão melhor de tudo que já foi abordado.

```
(function(){

    var Cusco = {

        // latir
        latir: function( nome ){
            return 'O cusco ' + nome + ' latiu';
        },

        // andar
        andar: function( nome ){
            return 'O cusco ' + nome + ' andou!';
        },

        // correr
        correr: function( nome ){
            return 'O cusco ' + nome + ' correu!';
        },

        // sentar
        sentar: function( nome ){
            return 'O cusco ' + nome + ' sentou!';
        }

    };

})();
```

Olhando para o código acima, podemos ver que temos um objeto (Cusco) comum com seus métodos, nada de diferente já visto em JavaScript, e poderíamos muito bem fazer a chamada das funções diretamente do objeto com o código *Cusco.latir()* por exemplo, como estamos trivialmente acostumados. Porém, dependendo da aplicação, teríamos um problema caso o nome dos métodos se alterassem, tendo assim um problema em ter que modificar o nome da chamada dos métodos em todo o nosso código.

Para evitar esse tipo de inconsistência, o Command pattern traz o modelo de ter uma função *execute()* que recebe o nome do método a ser executado bem como as informações por parâmetro do devido comando. Para isso então iremos implementar no objeto acima a função que ira invocar os métodos do objeto.

```

    Cusco.execute = function ( metodo ) {
        return Cusco[metodo] && Cusco[metodo].apply( Cusco, [].slice.call(arguments, 1) );
    };

```

Dessa forma podemos fazer a chamada dos métodos do nosso objeto através da função que invoca de fato os métodos. Para isso podemos fazer o nosso objeto Cusco latir, andar, correr, sentar através das seguintes chamadas.

```
30
31      Cusco.execute( 'latir', 'Tob' );
32      Cusco.execute( 'andar', 'Tob' );
33      Cusco.execute( 'correr', 'Tob' );
34      Cusco.execute( 'Sentar', 'Tob' );
35
```

Esse exemplo simples de implementação mostra então a criação de uma abstração e um *gap* (lacuna) entre a implementação acima e a execução do método que basicamente é o método *execute().* Podemos assim, trabalhar melhor com esse método em uma aplicação maior para que se possa por exemplo executar o mesmo método entre correr e andar, porem com velocidades diferentes.

Esclarecendo o exemplo

Acima temos um diagrama que exemplifica melhor a comunicação da estrutura do exemplo apresentado. O Command design pattern possui 4 partes fundamentais. A primeira é o *Client*, que basicamente é onde fazemos a requisição dos “serviços” do objeto invocando virtualmente os métodos através da método *execute()*. No exemplo acima fizemos isso solicitando que o nosso objeto Cusco corra, ande, lata e sente. A segunda parte basicamente é o *Receiver*, no nosso caso o objeto “Cusco”, que basicamente conheçe como lidar com a operação relacionada com o comando a ser executado, em alguns ambientes mais avançados pode-se também manter um histórico do comandos executados. A outra parte é o *Command* que, como o nome prevê, mantém informações sobre as ações a serem tomadas, ou seja, a nossa função *execute()* do objeto. Por último o *Invoker*, que lida com a requisição da função *execute()*, que basicamente são os métodos do objeto.

Uma implementação idiomática de Command Pattern

A implementação simples desse design usando o exemplo do Cusco dá para ter uma noção de como o pattern trabalha, mas de fato não exemplifica todo o poder que o Command possui. Abaixo apresento uma implementação desenvolvida por Derick Bailey que pode ser encontrada em https://jsfiddle.net/derickbailey/HsDNG/. Devido a alta qualidade do exemplo prefiro manter a implementação tal qual foi desenvolvida e me focar em abordar as ideias aplicadas por ele.

```
Executor = (function(){
    var executor = {};

    var commands = {};

    executor.handle = function(commandName, callback){
        var commandHandler = {
            ref: this,
            callback: callback
        };
        commands[commandName] = commandHandler;
    };

    executor.execute = function(commandName, data){
        var cmd = commands[commandName];
        if (cmd){
            cmd.callback.call(cmd.ref, data);
        }
    };

    return executor;
})();
```

Primeiramente temos o nosso *Receiver (objeto* Executor), e é interessante notarmos que temos não apenas o comando *execute*, mas também o comando *handle*, que basicamente possibilita a criação de novos comandos que ficam armazenados no objeto *commands,* dentro do nosso *receiver*. Outro fato interessante é a modularização e encapsulamento usado para se criar o objeto *executor*.

```
// a module to represent a view, which responds to a command
(function(Executor, $){
    // a command handler ... it's a function! cause, hey...
    // JavaScript is all about functions.
    var showTheText = function(data){
        var el = $("#showit");
        el.text("You told me to say: '" + data + "', so I did.");
    };

    // handle the command
    Executor.handle("doit", showTheText);
})(Executor, jQuery);
```

Logo adiante em sua implementação, Derick cria o primeiro *handler* "doit", fazendo com que a função "showTheText" se transforme no comando que pode ser invocado com o dado nome.

```
27 // a module to represent the "form"
28 (function(Executor){
29     var buttonEl = $("#sayit");
30     var textEl = $("#saythis");
31
32     buttonEl.click(function(e){
33         e.preventDefault();
34
35         Executor.execute("doit", textEl.val());
36     });
37 })(Executor);
```

Focando-se apenas na linha 35, podemos perceber então a chamada do comando que emite as devidas informações por parâmetro. Nada diferente do exemplo simples anteriormente apresentado, porem a implementação como um todo foi melhor elaborada por possibilitar a criação de novos comandos e por um apresentar um encapsulamento melhor dos métodos, o que serviu como motivação para apresentar esta outra opção de implementação.

Conclusão

Design patterns é de fato um assunto muito rico e extremamente interessante para o desenvolvimento seja qual for a linguagem. Em JavaScript, esse estudo torna-se quase uma necessidade por ser uma linguagem muito versátil e que possibilita diversas proezas e gambiarras no seu código. Os Patterns também são ótimas estruturas a serem seguidas na hora de começar o desenvolvimento de uma aplicação, o que possibilita o ganho de tempo no planejamento de toda a estrutura do código.

Com relação ao Command Pattern, me agradou muito a maneira como ele abstrai a execução de métodos na aplicação, possibilitando uma maior versatilidade e até mesmo uma ferramenta poderosa em aplicações de grande escala, principalmente pelo poder de controlar a chamada de métodos.

Acredito que para um desenvolvimento completo, seja necessário um conhecimento em pelo menos grande parte dos design patterns mais populares para julgar a melhor aplicabilidade no projeto que se está desenvolvendo. Meus estudos aqui

apenas começaram, mas ja pude ter uma boa base do poder dessa ferramenta, por isso preciso ainda testar os demais patterns e chegar na conclusão do que mais me agradará e usarei na maioria dos projetos, mas sempre é claro, com a possibilidade de se adaptar de projeto para projeto.

Referências

1. Learning JavaScript Design Patterns - Addy Osmani https://addyosmani.com/resources/essentialjsdesignpatterns/book/
2. JavaScript Design Patterns - dofactory http://www.dofactory.com/javascript/design-patterns
3. Design Patterns - SourceMaking https://sourcemaking.com/design_patterns
4. JavaScript Design Patterns: Command - Joe Zimmerman https://www.joezimjs.com/javascript/javascript-design-patterns-command/
5. A Idiomatic JavaScript Command Design Pattern - Derick Bailey https://jsfiddle.net/derickbailey/HsDNG/