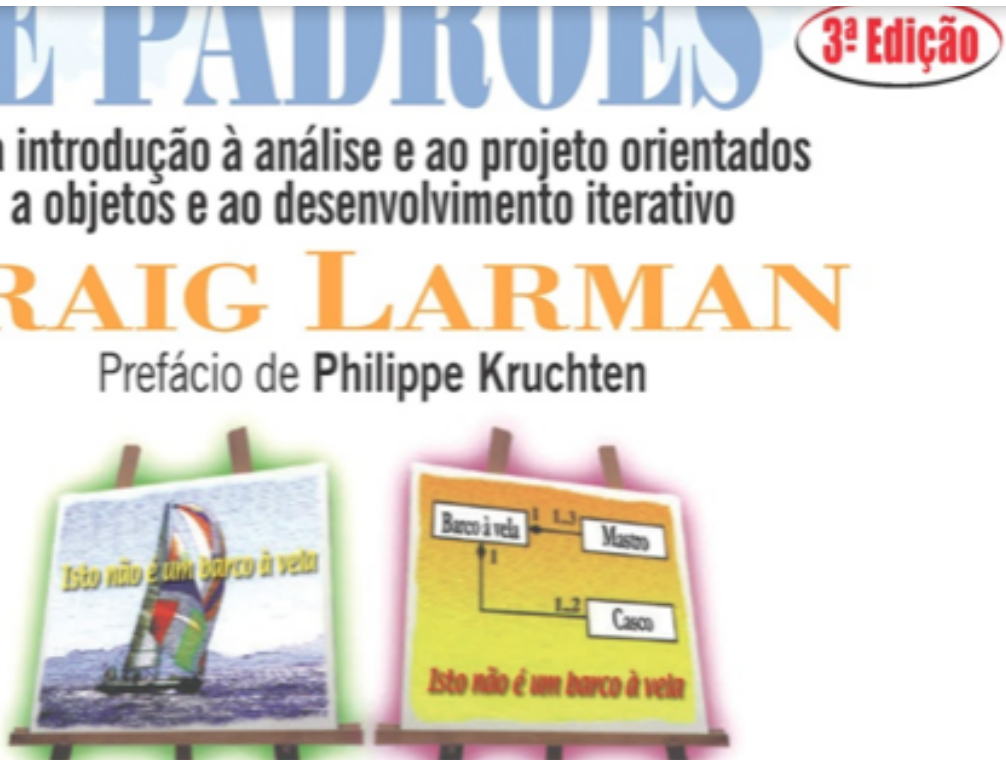

Capa do livro onde é explicado em detalhes cada um dos padrões GRASP

Padrões esquecidos - GRASP

🛠️ **Rodrigo Vieira Pinto, MSc**
Chapter Lead - Backend @ VR Benefícios |...
Publicado em 28 de fev. de 2022

Recentemente, conversando com o **Reginaldo Tostes**, verificamos algo que nos pareceu meio óbvio depois de um tempo, mas que, durante a conversa, nos chamou muito a atenção: boa parte dos conceitos relacionados à programação, como padrões ou boas práticas, que nos aparecem como novos são, na verdade, conceitos antigos, mas com uma nova roupagem.

Isso não é ruim. É possível que, com uma nova maneira de enxergarmos a mesma coisa, possamos entender melhor essa coisa, e finalmente utilizá-la, com mais informação e conhecimento de causa.

Martin Fowler. Logo na sua capa em português (pelo menos na edição que tenho) é possível ler a seguinte frase:

> *"Com frequência me perguntam qual é o melhor livro para conhecer o projeto orientado a objeto. Desde que o conheci, Utilizando UML e Padrões é a minha sugestão"*

No livro, é possível ver a modelagem de um sistema inteiro e suas diversas camadas e requisitos utilizando padrões de projeto do GoF, bem como o Processo Unificado (se você já ouviu falar do RUP, Rational Unified Process, é exatamente isso), diversos diagramas UML e a aplicação dos tais padrões GRASP.

Para quem tem dificuldades em modelar sistemas orientados a objeto, é um livro que dá diversas dicas em como ler requisitos e convertê-los em classes.

Lembro de ter conversado bastante sobre eles em diversos projetos. De ter citado eles na minha monografia de pós-graduação e até no **mestrado**. E, claro, de usá-los em alguns projetos. Mas de uns anos pra cá, dificilmente escuto falar deles. E acredito que existam bons motivos. Tentarei encontrá-los ao longo do artigo.

Também acredito que eles ainda sejam usados, mas nos foram apresentados com uma outra roupagem, o que faz que nós conheçamos eles hoje em dia com outros nomes.

Vou citá-los numa ordem um pouco diferente do que eles normalmente são listados, e no final do artigo explico o motivo.

O que são padrões GRASP?

GRASP é uma sigla para **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns (ou

A ideia neste artigo é apenas entender os motivos de não falarmos tanto neste conjunto de padrões, assim como falamos dos padrões GoF, ou dos princípios SOLID, ou dos padrões de microsserviços. Acredito que a maioria dos padrões GRASP são usados sim, mas que estão com uma nova roupagem, de certa forma ocultos em outros padrões.

Sem mais delongas, vamos aos padrões. Vou descrever os padrões usando palavras do próprio Larman, mescladas com as minhas em alguns casos:

Especialista da informação (Information Expert): trata da atribuição de responsabilidades dos objetos

Problema: dada uma determinada responsabilidade, qual objeto deve tê-la?

Solução: um objeto deve possuir determinadas responsabilidades caso possua informação suficiente para satisfazê-las, ou seja, se ele for o especialista na informação.

Exemplo (extraído do livro do Larman):

Considere um sistema de vendas, onde já temos alguns objetos de domínio, como ItemDeVenda, Venda e Produto. Se precisarmos saber qual é o total geral de uma venda, para qual classe iremos perguntar?

Embora esse exemplo possa parecer simples à primeira vista (espero que você tenha entendido que a classe Venda é uma boa candidata para responder nossa pergunta), o autor cita um problema comum nesses casos: a classe que pode ser a responsável pode não existir ainda, e é necessário criá-la. Nesse caso, é importante pesquisar nas histórias que escrevemos ou nas informações que conseguimos com os *stakeholders*.

Ao meu ver, esse padrão parece bastante com o princípio da responsabilidade única, presente nos princípios SOLID, embora o SRP busque resolver o problema de uma forma diferente. Escrevi recentemente sobre esse princípio **aqui**.

Solução: atribuindo responsabilidades de modo a manter o acoplamento entre os objetos baixo.

Por que não "falamos" dele hoje em dia: Esse me parece um padrão bem abstrato. Falamos dele até hoje. É uma dos padrões que, entendo, continua extremamente válido. Diversos autores concordam, entretanto, que esse padrão pode ser alcançado aplicando-se o **Princípio de Inversão de Dependências**, visto que esse princípio nos ensina que nossas classes devem depender de classes mais estáveis que as nossas. Entendo que o **Princípio de Segregação de Interfaces** também pode nos ajudar, pois ele também ajuda no desacoplamento, por meio da ideia de depender, quando necessário, de classes não concretas.

Controlador (Controller): oferece diretrizes para a criação de objetos que receberão informações da interface com o mundo externo (seja ela gráfica ou mesmo REST) para serem usados pelo domínio. Notar que esse objeto pode ser tanto um controlador do padrão MVC ou mesmo um objeto de domínio

Por que não "falamos" dele hoje em dia: esse padrão já era uma velho conhecido de quem estudou e aplicou o padrão MVC. Embora ele não se aplique somente a objetos onde o "C" do MVC se aplica, é onde mais o aplicamos. Portanto, na verdade, falamos bastante dele sim, mas não em conjunto com outros padrões do GRASP.

Coesão alta (High Cohesion): outro padrão que deve ser utilizado em conjunto com o padrão Especialista da informação.

Problema: como não atribuir muitas responsabilidades a determinados objetos, em detrimentos dos demais?

Solução: Fazer a atribuição de modo a deixar a coesão entre os objetos alta, ou seja, delegar responsabilidades a objetos de forma que não fiquem sobrecarregados, e que, se for o caso, delegar responsabilidades para outros objetos mais especialistas.

possível que as responsabilidades sejam executadas de forma semelhante, mas não exatamente igual, por objetos semelhantes.

Problema: como atribuir responsabilidades de acordo com o tipo do objeto?

Solução: para isso, é adequado utilizar-se de mecanismos polimórficos, sem que, para isso, seja necessária a utilização de lógica condicional para teste de tipo.

Por que não "falamos" dele hoje em dia: embora acredite que esse padrão tenha certa relação com o SRP (de novo!), o **Princípio do Aberto-Fechado** cai como uma luva para atende-lo. Ao invés de fazermos testes de tipo com "ifs" (e executar lógica de acordo com esses testes), podemos criar classes que encapsulam a lógica dos "ifs".

Invenção Pura (Pure Fabrication): padrão utilizado em casos nos quais padrões citados anteriormente (e o Creator, citado no final dessa lista) não sejam suficientes, ou conflitem entre si.

Problema: Qual objeto satisfaz uma responsabilidade (ou um conjunto de responsabilidades) sem que a aplicação dos padrões anteriormente citados provoque um conflito entre eles?

Solução: Neste caso, é necessário criar um objeto artificial que possua somente responsabilidades altamente coesas. Se for necessário, criar mais objetos com essas diretrizes.

Por que não "falamos" dele hoje em dia: Esse padrão parece o famigerado reservatório de gasolina para carros a álcool :P . Entretanto, ele era e é útil no caso de, mesmo aplicando todo o nosso conhecimento de padrões, não conseguimos encontrar um objeto que seja responsável por um determinado comportamento.

Indireção (Indirection): utilizado quando não é desejável que haja acoplamento entre dois ou mais objetos, sendo que esses precisam se comunicar de alguma forma.

Problema: Como permitir que objetos troquem informações entre si sem que haja

parecidos com esse, dificilmente se comenta sobre eles quando o assunto é fazer com que dois objetos se conversem sem se conhecerem. De qualquer forma, podemos dizer que existem outros padrões que buscam resolver esse problema, em "detrimento" do padrão Indireção.

Variações Protegidas (Protected Variations): oferece uma solução para as variações que um sistema sofre ao longo do tempo, de modo que uma variação num determinado ponto não comprometa o restante do sistema.

Problema: Como projetar um sistema de modo que suas variações não o afetem por completo, mas apenas no trecho em que a variação ocorra?

Solução: É preciso encapsular o(s) elemento(s) que varia(m), identificando pontos de instabilidade previsíveis e atribuindo responsabilidades para criação de uma interface estável em torno deles.

Por que não "falamos" dele hoje em dia: Esse padrão apenas fornece mais justificativas para que se **encapsule o que varia**.

E por último, o princípio GRASP mais importante de todos (na minha humilde e modesta opinião), visto que é difícil encontrar um padrão ou princípio equivalente em outras listas de padrões e princípios:

Criador (Creator): trabalha com a responsabilidade de criação de objetos.

Problema: Dado um objeto A, quem deve criá-lo?

Solução: Atribuir à outra classe (por exemplo, B) essa responsabilidade, desde que B satisfaça os seguintes critérios:

- B “contém“ A ou agrega A de forma composta;
- B registra A;
- B usa A de maneira muito próxima;

```
class UsuarioDTO {

   private String nome;

   private String dataNascimento;

}
```

Esse DTO é utilizado no controller de um serviço REST qualquer. Então, em algum momento, os dados presentes nesse DTO deverão estar num objeto de domínio.

Digamos que esse objeto de domínio é uma classe Usuario:

```
class Usuario {

   private String nome;

   private LocalDate dataNascimento;

}
```

Para realizar essa conversão, alguns programadores criam uma outra classe, normalmente chamada de "UsuarioFabrica", "UsuarioFactory" ou até mesmo "UsuarioAssembler":

```
        DateTimeFormatter formatador = DateTimeFormatter
          .ofPattern("dd/MM/yyyy");

        Usuario usuario = new Usuario();
        usuario.setNome(usuarioDTO.getNome());

        usuario.setDataNascimento(LocalDate
          .parse(usuarioDTO.getDataNascimento(), formatador));

        return usuario;

    }

}
```

Nessa abordagem, precisamos, criar getters e setters, expondo o estado dos objetos (mesmo que estejamos usando lombok).

Também precisamos criar setters para todos os atributos de Usuario, para que a fábrica consiga atribuir os valores ao objeto. Embora o padrão Creator não dizer nada à respeito, normalmente os desenvolvedores ignoram a possibilidade de se usar o construtor do objeto como momento onde podemos passar os atributos mínimos para o objeto funcionar, sem setters.

Mas, se aplicarmos o padrão Creator, tudo que precisamos fazer é escrever o método abaixo, na classe UsuarioDTO:

```
Usuario toUsuario() {

    DateTimeFormatter formatador = DateTimeFormatter
```

com o método acima.

- eliminamos a classe UsuarioAssembler
- removemos todos os getters e setters das classes Usuario e UsuarioDTO (mesmo que você tenha usado anotações do Lombok, como @Getter, @Setter ou a famigerada @Data)

Para não dizer que sou contra o Lombok, podemos fazer um bom uso de suas anotações, anotando o que pode ser útil, com as anotações necessárias. Nem mais, nem menos.

Por exemplo, em Usuario podemos deixar simplesmente assim:

```
@AllArgsConstructor
class Usuario {
   private String nome;
   private LocalDate dataNascimento;
}
```

E UsuarioDTO ficaria assim:

```
class UsuarioDTO {

   private String nome;
```

```
            .ofPattern("dd/MM/yyyy");

        return new Usuario(nome, LocalDate
            .parse(dataNascimento, formatador));

    }

}
```

É possível ver que diversos desses padrões foram absorvidos por outros princípios, ou mesmo outros padrões. Entendo que isso, no entanto, não desmerece o conjunto de princípios reunidos no GRASP. Os padrões podem ser organizados em listas, de acordo com o que se quer destacar, e quando se deseja entender como atribuir responsabilidades aos objetos, a organização sugerida pelo GRASP pode ser uma boa pedida.

Desculpe-me por exemplificar apenas o padrão Criador. Entendo que ele ainda se faz muito necessário, tamanha a quantidade de vezes que ele não é aplicado. De qualquer forma, as fontes foram passadas.

E você? Conhecia os padrões GRASP? Entende que eles, da forma como foram apresentados pelo Larman, podem ajudar o desenvolvimento de software OO ou eles podem ser considerados "ultrapassados"? Se não conhecia, vale a pena dar uma olhada. E não deixe de comentar!

Até a próxima!

Fernando Boaglio 1 a ···
Especialista de tecnologia CitiBank 💻 | JUG leader Java Meetup SP ☕ | Open Source 🐧 | Autor dos livros: {Spri...