Primeiros Passos com Padrões de Projeto

Marcos Brizeno

Primeiros passos com Padrões de Projeto

Marcos Brizeno

This book is for sale at http://leanpub.com/primeiros-passos-com-padroes-de-projeto

This version was published on 2016-02-20

Olá, esse livro foi escrito pensando em pessoas que possuem um bom conhecimento de Orientação a Objetos e querem dar os primeiros passos com Padrões de Projeto.

Não iremos entrar em muitos detalhes sobre os padrões, mas o meu livro Refatorando com Padrões de Projeto é um excelente maneira de aprofundar os estudos.

Se você gostar do que vai ler aqui, compartilhe o material com outras pessoas!

Se tiver qualquer feedback me avise em @marcosbrizeno.

Esse livro foi distribuído de graça e pode ser encontrado em leanpub.com.

Boa leitura!

Conteúdo

INTRODUÇÃO 1

O QUE É UM PADRÃO DE PROJETO? 2
O Surgimento dos Padrões de Projeto 2
Os padrões da Gangue dos Quatro 3

REVISITANDO A ORIENTAÇÃO A OBJETOS 4
Alguns Princípios de Design Orientado a Objetos 4

EXEMPLOS DE PADRÕES DE PROJETO 6

O PADRÃO SIMPLE FACTORY 7
Exemplo de Aplicação 7
Um Pouco de Teoria 10
Quando Não Usar 11
Evoluindo o Simple Factory para Factory Method 11
Evoluindo o Simple Factory para Builder 12

O PADRÃO STRATEGY 13
Exemplo de Uso 13
Um Pouco de Teoria 16
Quando Não Usar 17
Evoluindo o Strategy para Template Method 17
Evoluindo o Strategy para State 18

A ESTRADA A FRENTE 19

APLICANDO PADRÕES DE PROJETO 20
Refatorando com Padrões de Projeto 20

REFERÊNCIAS EXTERNAS 22

SOBRE O AUTOR 23

INTRODUÇÃO

Como desenvolvedores, nós enfrentamos vários problemas para lidar com a informação nos sistemas: estruturar e armazenar as informações, transformar dados para que possam ser lidos pelos usuário, agrupar dados de diferentes sistemas etc. Assim, desenhar um sistema e sua arquitetura nem sempre é uma tarefa simples e direta. Além disso, conforme aprendemos mais sobre o domínio e a maneira como o sistema é utilizado, percebemos que o desenho inicial não será suficiente e sentimos a necessidade de modificá-lo.

No entanto, a maioria dos desafios que encontramos não é único. Certamente o domínio do sistema pode ser diferente, mas a essência dos problemas continua a mesma: como lidar com informação. Assim surgiu a ideia dos Padrões de Projeto, como uma base coletiva de conhecimento com soluções para esses "problemas comuns" que encontramos todos os dias.

Ao longo deste livro vamos ver como se deu o surgimento dos Padrões de Projeto e como eles foram adaptados para o desenvolvimento de software. Em seguida vamos analisar características do paradigma Orientado a Objetos e como eles buscam simplificar o desenvolvimento de aplicações. Por fim, serão explorados dois exemplos de padrões voltados para o desenvolvimento Orientado a Objetos, para exemplificar como eles podem ajudar na manutenção de sistemas.

O QUE É UM PADRÃO DE PROJETO?

Desenvolvimento de software é uma área relativamente nova comparada com outras áreas de conhecimento. Contando a partir da Ada Lovelace como a desenvolvedora do primeiro programa de computador e o surgimento da Máquina de Turing, ainda não temos 100 anos de história.

Por isso, muitas das nossas práticas são "emprestadas" de outras áreas. O maior exemplo é a Engenharia de Software onde tentamos aplicar muitas das ideias de outras Engenharia nos projetos de desenvolvimento de software.

O Surgimento dos Padrões de Projeto

O surgimento dos Padrões de Projeto, por exemplo, vem da arquitetura, através do livro "A Patter Language" escrito por Christopher Alexander, Sara Ishikawa e Murray Silverstein, do Center for Environmental Structure of Berkeley, California, publicado em 1977.

O livro define uma linguagem chamada pelos autores de "linguagem de padrões", que se origina a partir dos 253 padrões apresentados no livro. O objetivo é que, com essa linguagem em comum, pessoas ao redor do mundo possam compartilhar ideias sobre como organizar bairros, cidades, projetar casas, escritórios e quaisquer outros espaços.

Quando levamos esse conceito para o mundo do desenvolvimento, temos os Padrões de Projeto como soluções de implementação para um problema em um determinado contexto. O objetivo ao documentar as soluções comumente encontradas é montar um "banco de conhecimento" onde pessoas com o mesmo problema podem consultar qual seria uma boa solução.

Com o ritmo que cresce a tecnologia da informação, essa base de conhecimento se torna ainda mais valiosa pois nos permite aprender com as experiências de outras pessoas. Mas, antes de explorar os padrões em si, é importante entender como essa base de conhecimento surgiu.

O livro "Padrões de Projeto - Soluções Reutilizáveis de Software Orientado a Objetos" da famosa Gangue dos Quatro (Gang of Four em inglês) é talvez a referência mais utilizada sobre o assunto. Esse livro popularizou os conceitos de Padrões de Projeto apresentando as soluções em C++ e Smalltalk, duas linguagens com características bem diferentes e que faziam sucesso nos anos 90.

Um ponto importante sobre padrões é que eles não são "inventados", mas sim extraídos de códigos reais. A solução proposta por um padrão surge do conhecimento comum de várias pessoas que passaram pelo mesmo problema e aplicaram soluções semelhantes. A maior prova disso é que os livros que documentam Padrões de Projeto são geralmente escritos por várias pessoas.

Outro bom exemplo é o livro "Padrões de Arquitetura de Aplicações Corporativas" escrito por Martin Fowler com a colaboração de David Rice, Mathew Foemmel, Edward Hieatt, Robert Mee e Randy

Stafford. A diferença deste livro é que os padrões apresentados não são específicos para linguagens Orientada a Objetos, mas para projetos corporativos, que precisam ser mais robustos e lidar com problemas não triviais.

Existe também Padrões de Projeto que propões soluções para problemas específicos de uma linguagem ou plataforma, como por exemplo o livro "Core J2EE Patterns", que apresenta soluções específica para a linguagem Java na plataforma Java 2 Enterprise Edition. Eles continuam sendo considerados padrões, a única diferença é que o contexto de sua aplicação é mais restrito.

Os padrões da Gangue dos Quatro

Sendo a referência mais conhecida, é bem provável que ao ouvir falar de padrões de projeto você escute sobre um dos 23 padrões documentados pela Gangue do Quatro. Para linguagens Orientadas a Objetos esses padrões são tão úteis que são encontrados em muitas das bibliotecas mais utilizadas, e alguns são aplicados até mesmo na biblioteca padrão da linguagem.

Ao longo do livro os autores constroem uma aplicação para exemplificar as várias situações de onde os Padrões de Projeto foram extraídos. Cada um deles é documentando com: nome, objetivo, motivação, contexto, solução e o exemplo. Além disso os padrões são classificados de acordo com a natureza do problema que tenta resolver: criar objetos (padrões de criação), estruturar os objetos (padrões estruturais) ou dividir comportamento (padrões comportamentais).

Padrões de Criação

Factory Method, Abstract Factory, Builder, Prototype e Singleton

Os Padrões de Criação tem como intenção principal abstrair o processo de criação de objetos, ou seja, a sua instanciação. Desta maneira o sistema não precisa se preocupar com a lógica de criação de objetos, permitindo que ela evolua independente do resto.

Padrões Estruturais

Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy

Os Padrões Estruturais se preocupam em como as classes e objetos são compostos, ou seja, sua estrutura. O objetivo destes padrões e facilitar o design do sistema, melhorando as maneiras de relacionamento entre as entidades.

Padrões Comportamentais

Interpreter, Template Method, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Stratefy, Visitor

Os Padrões Comportamentais atuam na distribuição das responsabilidades entre os objetos, ou seja, como eles se comportam. Estes padrões facilitam a comunicação entre os objetos, distribuindo as responsabilidades.

REVISITANDO A ORIENTAÇÃO A OBJETOS

A programação Orientada a Objetos é um paradigma bastante disseminado entre os desenvolvedores, assim cada pessoa acaba tendo seu próprio entendimento. Então, antes de falar sobre os padrões desse paradigma, vamos analisar as ideias do pensamento orientado a objetos e partir de um ponto comum.

Alan Kay cunhou o termo "Orientação a Objetos" em 1967 e é tido como o criador do paradigma. Em um email, que é a fonte de informação mais importante sobre o tema http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en, Dr. Kay explica quais foram suas motivações iniciais ao criar a Programação Orientada a Objetos.

Segundo sua definição, podemos entender o paradigma Orientado a Objetos em duas partes: 1) objetos seriam como células biológicas, que se comunicam apenas através de mensagens e 2) esconder os dados, pois cada objeto possui sua própria lógica para lidar com a informação sem precisar expô-las.

Ao definir uma classe é comum falar na "interface do objeto" como sendo a maneira como os objetos desta classe trocam mensagens, ou seja os seus métodos públicos. Essa interface esconde o estado interno do objeto e permite que o desenvolvedor foque nas interações, simplificando o entendimento do programa.

A primeira linguagem a aplicar esses conceitos foi Smalltalk, uma linguagem de tipo dinâmico onde não existem tipos primitivos e tudo é implementado como uma classe. Ao longo do desenvolvimento de novas linguagens, outras funcionalidades foram adicionadas, como por exemplo a herança e o polimorfismo.

As linguagens Orientada a Objetos mais recentes tendem a misturar conceitos e funcionalidades de diversos paradigmas, como por exemplo representar métodos como objetos, utilizar lambdas para passar blocos de código como parâmetros para outros métodos etc. Mas por trás de todas essas novas funcionalidades continua o pensamento de esconder os dados para facilitar a comunicação entre objetos.

Alguns Princípios de Design Orientado a Objetos

Com a grande adoção de linguagens Orientada a Objetos, surgiram vários princípios para guiar o design de sistemas. Esses princípios ajudam a organizar o código e facilitam a manutenção através de boas práticas sobre como escrever as interações entre os objetos.

O conjunto de princípios que se tornou mais famoso é o SOLID, criado por Robert C. Martin (também conhecido na comunidade como "Uncle Bob"). O problema que levou a criação deles foi a dificuldade de gerenciar dependências entre objetos e classes, especialmente quando a aplicação começa a crescer.

Segundo Robert, um código que não consegue fazer um bom gerenciamento de dependência se torna difícil de manter, frágil e não reutilizável. Em resumo, esses são os princípios e o que eles pregam:

- Single Responsibility Principle (Princípio da Responsabilidade Única): Cada classe deve ter um, e apenas um, motivo para mudar;
- Open Closed Principle (Princípio Aberto Fechado): Deve ser possível estender o comportamento de uma classe sem modificá-la;
- Liskov Substitution Principle (Princípio da Substituição de Liskov): Classes derivadas devem ser compatíveis com sua classe base;
- Interface Segregation Principle (Princípio da Segregação de Interface): Crie interfaces mínimas e específicas para o cliente;
- Dependency Inversion Principle (Princípio da Inversão de Dependência): Dependa de abstrações ao invés de classes concretas.

Em seu site, Robert apresenta artigos dedicados a cada um dos princípio, com motivação, exemplos e aplicações. Além disso, outros princípios também podem ser encontrados lá e a leitura é recomendada para aprofundar o conhecimento no assunto: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod. No meu blog, tenho uma série de posts dedicada aos princípios solid que também pode ajudar a entender melhor a importância deles: https://brizeno.wordpress.com/solid/.

Além do SOLID, também vale a pena explorar outros dois princípios de design Orientado a Objetos que estão fortemente presentes nos Padrões de Projeto:

- Prefira Composição ao invés de Herança: ao herdar de uma classe estamos aumentando o nível de acoplamento entre elas, portanto é preferível compor objetos e criar interfaces para expor sua lógica;
- Programe voltado a Interface e não a Implementação: ao desenvolver uma classe não pense em como ela vai funcionar, mas sim em como ela será utilizada. Exponha interfaces que façam sentido e simplificam seu uso.

Esses princípios foram especialmente citados aqui pois, ao refatorar o código e aplicar um padrão, estamos também seguindo os princípios de design Orientado a Objetos.

EXEMPLOS DE PADRÕES DE PROJETO

Para exemplificar as principais vantagens de utilizar Padrões de Projetos, foram escolhidos os padrões Simple Factory e Strategy, devido a simplicidade de utilização e por resolverem problemas comuns do dia-a-dia. Nos capítulos seguinte vamos mostrar um exemplo de código que pode se beneficiar dos padrões, mostrando como ficaria o resultado final e quais suas vantagens.

Ao estudar aplicações de padrões, o mais importante não é saber como aplicar o padrão ou qual implementação utilizar. Refatorar um código para aplciar um padrão não é tão difícil quanto decidir quando aplicá-lo. Ao ver exemplos, atente para as melhorias de design que precisam acontecer e em como indentificá-las.

O PADRÃO SIMPLE FACTORY

Criar objetos geralmente é uma tarefa simples, passamos valores para o construtor e pronto, temos um objeto instanciado e pronto para uso. No entanto, algumas vezes a criação não é tão direta assim e alguma manipulação com os dados pode ser necessária. Em outros casos é preciso configurar o objeto, mesmo depois da execução do construtor.

Quando instanciar objetos não for uma tarefa tão simples, os padrões de criação oferecem boas soluções. Como vimos antes, os padrões documentados pela Gangue dos Quatro foram classificados em grupos de acordo com o tipo de problema que resolvem. Padrões que facilitam criar novas instâncias de objetos são classificados como padrões de criação.

Uma versão simples e que pode ser utilizada mais facilmente, é o Simple Factory. Nele vamos extrair a lógica que cria objeto para uma classe especializada, deixando que a manipulação dos dados não atrapalhe o restante da lógica de negócio. Alguns autores nem consideram o Simple Factory como um padrão, devido a sua simplicidade.

Exemplo de Aplicação

Na Listagem 1 temos um exemplo de código que pode se beneficiar do uso do Simple Factory. Nele, o método buscarProdutosPreferenciais recebe um usuário consulta um serviço externo para obter os produtos que o usuário costuma comprar. Ao final, os produtos são validados e filtrados para que apenas aqueles com estoque sejam retornados.

No entanto, antes de chegar na lógica que realmente faz a busca dos produtos preferenciais do usuário, é preciso configurar a chamada ao serviço externo. Essa responsabilidade extra aumenta o tamanho e a complexidade do método.

Listagem 1 - Método que busca os produtos preferenciais de um usuário

```
public List<Produto> buscarProdutosPreferenciais(Usuario usuario) {
  ConfiguracoesServicos config = new ConfiguracoesServicoProdutosPreferenciais();
  config.setRecurso("/produtos/preferenciais/");
  config.setIdUsuario(usuario.getId());

  if (getAmbienteAtual() == Ambiente.LOCAL) {
    config.setEndpoint("localhost:1234");
    config.setXid(gerarUuid());
    config.setVersaoApi("2.1");
  } else {
```

```
    if (toggleApiNova.habilitado()) {
      config.setVersaoApi("2.1");
      config.setXid(gerarUuid());
    } else {
      config.setVersaoApi("1.9");
      config.setXid("");
    }
    config.setEndpoint("https://intra." + getAmbienteAtual().toString() + ".mega\
corp.com/");
  }

  ServicoRest servico = new ServicoRest(config);
  List<Produto> produtos = criarProdutos(servico.executarGet())

  List<Produto> produtosValidos = new ArrayList<Produto>();
  for (Produto produto : produtos) {
    if (produto.temEstoque()) {
      produtosValidos.add(produto);
    }
  }
  return produtosValidos;
}
```

Note quantas linhas apenas a criação das configurações toma. Isso acaba desviando a atenção da responsabilidade principal do método, que é buscar os produtos preferenciais do usuário.

No exemplo anterior existem dois tipos de configurações, uma para o ambiente local e outra para os demais ambientes. A configuração dos ambientes remotos precisa lidar ainda com um feature toggle, para determinar qual versão da API deve ser utilizada.

A criação de configurações locais pode ser extraída como mostrado na Listagem 2. A identificação do recurso é comum a todos as configurações, portanto pode ser definida já no construtor da fábrica, assim como o id do usuário. As demais configurações são específicas ao ambiente local, ficando portanto dentro do método criarConfiguracaoLocal.

Listagem 2 - Classe fábrica para criar configurações dos serviços

```
class FabricaConfiguracaoServicos {

  private ConfiguracoesServicos config;

  public FabricaConfiguracaoServicoProdutosPreferenciais(String idUsuario) {
    config = new ConfiguracoesServicoProdutosPreferenciais();
    config.setRecurso("/produtos/preferenciais/");
    config.setIdUsuario(idUsuario);
  }

  public ConfiguracoesServicos criarConfiguracaoLocal() {
    config.setEndpoint("localhost:1234");
    config.setXid(gerarUuid());
    config.setVersaoApi("2.1");
    return config;
  }
}
```

A lógica para criar as configurações de serviços para outros ambientes depende do feature toggle. Nesse caso vamos passar a informação se o toggle está ativo e o ambiente atual como parâmetros. A implementação pode ficar como na Listagem 3.

Listagem 3 - Lógica específica para configuração de serviços remotos

```
class FabricaConfiguracaoServicos {

  public ConfiguracoesServicos criarConfiguracaoRemota(boolean usarApiNova, Ambi\
ente ambiente) {
    if (usarApiNova) {
      config.setVersaoApi("2.1");
      config.setXid(gerarUuid());
    } else {
      config.setVersaoApi("1.9");
      config.setXid("");
    }
    config.setEndpoint("https://intra." + ambiente.toString() + ".megacorp.com");
  }
}
```

Para utilizar o Simple Factory vamos simplesmente substituir o código dentro do if pela chamada ao método fábrica apropriado. Vamos também extrair essa parte em um novo método, para facilitar a leitura, como mostrado na Listagem 4:

Listagem 4 - Método para buscar produtos aplicando o padrão Simple Strategy

```
public List<Produto> buscarProdutosPreferenciais(Usuario usuario) {
  ConfiguracoesServicos config = criarConfiguracaoDoAmbiente(usuario)

  ServicoRest servico = new ServicoRest(config);
  List<Produto> produtos = criarProdutos(servico.executarGet())

  List<Produto> produtosValidos = new ArrayList<Produto>();
  for (Produto produto : produtos) {
    if (produto.temEstoque()) {
      produtosValidos.add(produto);
    }
  }
  return produtosValidos;
}

public ConfiguracoesServicos criarConfiguracaoDoAmbiente(Usuario usuario) {
  FabricaConfiguracaoServicos fabrica = new FabricaConfiguracaoServicos(usuario.\
getId());

  if (getAmbienteAtual() == Ambiente.LOCAL) {
    return fabrica.criarConfiguracaoLocal();
  } else {
    return fabrica.criarConfiguracaoRemota(toggleApiNova.habilitado(), getAmbien\
teAtual());
  }
}
```

Apesar de simples, o ganho com a separação das responsabilidades é bem grande, principalmente na legibilidade do código.

Um Pouco de Teoria

Como mencionado antes, todos os padrões ajudam o seu código a seguir os princípios de design Orientado a Objetos. No caso do Simple Factory, o maior benefício é a divisão de responsabilidades seguindo o Princípio da Responsabilidade Única.

Quem lê o código do método buscarProdutosPreferenciais pode focar apenas em entender sua lógica, sem ter que se preocupar em como as configurações são criadas. Como passamos muito mais tempo lendo código do que escrevendo, essa é uma grande vantagem.

Outro benefício da separação dessas responsabilidades é que a maneira como as configurações do serviço são criadas vai mudar menos do que lógica definida em buscarProdutosPreferenciais. Ao lidar com dependências entre classes é sempre bom que uma classe dependa de outras menos prováveis de mudar.

Os testes do método buscarProdutosPreferencias também estão validando a criação das configurações de serviços, mesmo que indiretamente. Criando uma classe especializada para isso, podemos limitar o escopo dos testes e focar na lógica de negócio, testando a criação de configurações em outros testes isolados.

O Simple Factory é um bom ponto de início para separar a criação de objetos do seu uso. Como vimos no exemplo anterior, poucas classes são criadas e a estrutura do padrão é bem simples. Se o seu contexto permite isolar a maneira como objetos são criados e você tiver que lidar apenas com um tipo de objeto, o Simple Factory é uma excelente maneira de resolver o problema.

Quando Não Usar

Apesar de simples, existem situações onde utilizar o padrão Simple Factory não ajuda muito. Um sinal bem claro de que o padrão não está sendo efetivo é quando a classe fábrica começa a crescer e ter vários métodos para criar os mesmos produtos de maneiras diferentes. Essa talvez seja uma boa hora para aplicar outros padrões fábrica.

Nas seções seguintes vamos comparar como evoluir do Simple Factory para o Factory Method ou para o Builder, dependendo de qual a necessidade do seu contexto. Não vamos entrar em detalhes sobre estes padrões, mas outros recursos com mais detalhes serão indicados ao final do artigo (veja a seção de Referências Externas).

Evoluindo o Simple Factory para Factory Method

Se existem vários métodos que podem ser agrupados com suas próprias lógicas para criar objetos, provavelmente nomeados com um sufixo ou prefixo em comum, talvez seja melhor utilizar o Factory Method e separar esses grupos em outras classes fábrica.

Voltando ao exemplo anterior, imagine que será necessário criar configurações para outros serviços. Ao invés de colocar tudo em um única classe, é melhor criar fábricas especializadas em construir as configuração de cada serviço. Refatorar o Simple Factory para o Factory Method vai ajudar a evitar que a classe fábrica cresça.

Com o Factory Method, continuaremos criando objetos do tipo ConfiguracoesServicos, mas as regras deles serão divididas em classes separadas. Essas classes fábricas seguem uma mesma interface, garantindo que elas possam ser trocadas facilmente. Ao fim, evitamos ter uma classe fábrica com muitas responsabilidades e, como elas são intercambiáveis, conseguimos ter flexibilidade no código que as utiliza.

Evoluindo o Simple Factory para Builder

Se ao rever os métodos você perceber que existe pouca variação na lógica entre eles, apenas mudam-se os valores que são atribuídos ou quais atributos são utilizados, será necessário criar muitos métodos fábrica, com muita duplicação entre si.

No exemplo anterior, imagine que várias configurações de serviço devem ser criadas, mas a diferença entre os métodos é o valor de alguns atributos. Nesse caso podemos utilizar o Builder para expor a configuração dos atributos mas esconder a criação dos objetos.

Com o Builder, ao invés de definirmos o processo de como o objeto será construído, oferecemos métodos para que a classe cliente consiga configurar, de maneira simples, o produto final. Assim evitamos criar vários métodos fábrica para cada possível cenário, sem perder a separação da responsabilidade de criação.

O PADRÃO STRATEGY

Ao codificar um programa, seguimos um conjunto de regras de negócio e, muitas vezes, o fluxo precisa ser dividido em vários caminhos. Quando o código cresce com várias regras diferentes, podemos recorrer aos padrões comportamentais para melhorar a flexibilidade e facilitar a manutenção.

Padrões comportamentais vão dividir as responsabilidades para resolver problemas de complexidade do código. O Strategy é um exemplo de padrão comportamental que busca isolar os vários caminhos que o algoritmo pode seguir, facilitando escolher um fluxo específico sem precisar se preocupar com os outros.

Exemplo de Uso

Na Listagem 5 temos a implementação do código que deve calcular os pontos que um passageiro ganhou em voos pela companhia aérea. As regras variam de acordo com a distância voada, o status do passageiro e o tipo de bilhete comprado.

Dado um passageiro, o algoritmo busca a lista de voos que ainda não foram computados e, para cada um deles aplica as regras de pontuação. Caso o bilhete seja de primeira classe e o passageiro tenha a categoria diamante, a distância total do voo será convertida em pontos. Caso o passageiro não seja diamante, ele receberá metade da distância em voos de primeira classe, um quarto em voos econômicos e um décimo em voos promocionais.

Listagem 5 - Método para calcular a pontuação de um passageiro

```
public int calcularPontuacao(Passageiro passageiro) {
  int totalDePontos = 0;
  List<Voo> voosPendentes = passageiro.getVoosComPontuacaoPendente();

  for (Voo voo : voosPendentes) {
    Bilhete bilhete = passageiro.getBilhete(voo.getNumero());

    if (bilhete.isPrimeiraClasse() && passageiro.isDiamante()) {
      totalDePontos += voo.getDistancia();
    } else if (bilhete.isPrimeiraClasse()) {
      totalDePontos += voo.getDistancia()/2;
    } else if (bilhete.isClasseEconomica()) {
      totalDePontos += voo.getDistancia()/4;
    } else {
      // bilhete promocional
```

```
      totalDePontos += voo.getDistancia()/10;
    }
  }

  passageiro.atualizarPontuacao(totalDePontos);
  passageiro.marcarVoosComoProcessados();

  return totalDePontos;
}
```

Separando os fluxos do algoritmo, conseguimos extrair 4 estratégias: 1) voo de um passageiro diamante na primeira classe, 2) voo na primeira classe, 3) voo na classe econômica e 4) voo promocional.

A interface comum das estratégias é bem simples e precisa apenas oferecer um método que retorna a pontuação do passageiro, dado a distância do voo. Uma Interface poderia ser implementada conforme a Listagem a seguir:

Listagem 6 - Interface simples para estratégias de cálculo de pontuação

```
interface EstrategiaDePontuacao {
  public int calcularPontuacao(int distanciaDeVoo);
}
```

A implementação das estratégias também é bem simples e pequena, pois as regras separadas são simples. Veja como ficaria o código na Listagem 7:

Listagem 7 - Classes com implementação de estratégias seguindo a interface

```
class EstrategiaPrimeiraClasseDiamante implements EstrategiaDePontuacao {
  public int calcularPontuacao(int distanciaDeVoo){
    return distanciaDeVoo;
  }
}

class EstrategiaPrimeiraClasse implements EstrategiaDePontuacao {
  public int calcularPontuacao(int distanciaDeVoo){
    return distanciaDeVoo/2;
  }
}

class EstrategiaClasseEconomica implements EstrategiaDePontuacao {
  public int calcularPontuacao(int distanciaDeVoo){
```

```
    return distanciaDeVoo/4;
  }
}

class EstrategiaPromocional implements EstrategiaDePontuacao {
  public int calcularPontuacao(int distanciaDeVoo){
    return distanciaDeVoo/10;
  }
}
```

Um ponto de atenção ao aplicar o padrão Strategy é encontrar onde deve ser decidido qual estratégia utilizar. Para determinar a regra de pontuação precisamos de informações de um passageiro e do bilhete. Vamos então criar um método na classe Bilhete que retorna qual a estratégia de pontuação deve ser utilizada. Veja a implementação na Listagem seguinte:

Listagem 8 - Método que decide qual estratégia deve ser utilizada

```
class Bilhete {
  public EstrategiaDePontuacao getEstrategiaDePontuacao(boolean isPassageiroDiam\
ante){
    if (bilhete.isPrimeiraClasse() && isPassageiroDiamante) {
      return new EstrategiaPrimeiraClasseDiamante();
    } else if (bilhete.isPrimeiraClasse()) {
      return new EstrategiaPrimeiraClasse();
    } else if (bilhete.isClasseEconomica()) {
      return new EstrategiaClasseEconomica();
    } else {
      return new EstrategiaPromocional();
    }
  }
}
```

Agora basta que o método calcularPontuacao pegue a estratégia do bilhete para definir quantos pontos devem ser dados ao passageiro. A utilização ficaria como mostrado na Listagem 8.

Listagem 9 - Cálculo de pontuação utilizando as estratégias implementadas

```
public int calcularPontuacao(Passageiro passageiro) {
  int totalDePontos = 0;
  List<Voo> voosPendentes = passageiro.getVoosComPontuacaoPendente();

  for (Voo voo : voosPendentes) {
    Bilhete bilhete = passageiro.getBilhete(voo.getNumero());
    EstrategiaDePontuacao estrategia = bilhete.getEstrategiaDePontuacao(passagei\
ro.isDiamante());
    totalDePontos += estrategia.calcularPontuacao(voo.getDistancia());
  }

  passageiro.atualizarPontuacao(totalDePontos);
  passageiro.marcarVoosComoProcessados();

  return totalDePontos;
}
```

Ao quebrar o fluxo do algoritmo em estratégias conseguimos simplificar o método calcularPontuacao e dividir a responsabilidade de saber qual regra utilizar com a classe Bilhete. Por sua vez, a lógica de cálculo da pontuação fica definida dentro de cada uma das estratégias.

No final ganhamos maior coesão ao separar as responsabilidades, facilitando modificar e até mesmo adicionar novas regras de pontuação.

Um Pouco de Teoria

Assim como no caso do padrão Simple Factory, ao aplicar o Strategy fica bem claro a divisão das responsabilidades, seguindo o Princípio da Responsabilidade Única. A lógica para calcular a pontuação fica mais simples uma vez que está separada.

Após distribuir os fluxos do algoritmo nas classes estratégias, podemos também mover os testes e fazê-los validar apenas uma parte do código. Considerando que testes são a melhor forma de documentação, uma nova pessoa na equipe conseguirá ver o funcionamento de um fluxo de cada vez, ao invés de todo o algoritmo.

Como a estrutura de estratégias define uma interface comum para todos, também é fácil notar o Princípio da Substituição de Liskov. Trocar as estratégias, ou até mesmo adicionar novas, não vai ter nenhum impacto pois o código que as utiliza continuará lidando com a mesma interface.

O Princípio da Inversão de Dependência também fica claro pois o cliente não usa as estratégias concretas diretamente, apenas uma interface. Assim, cada implementação pode ter suas próprias regras sem interferir na estrutura do código.

Outro ponto importante é que devido ao baixo acoplamento entre a classe, fica mais fácil evoluir os códigos separadamente. Como as estratégias tendem a mudar menos do que o código do cliente, também seguimos a ideia de que uma classe deve depender de outras menos prováveis de mudar.

Quando Não Usar

Semelhante ao padrão Simple Factory, o Strategy é uma excelente maneira de começar a refatorar seu código. Mas, devido a sua simplicidade, eventualmente pode ser necessário partir para soluções mais robustas e evitar que as estratégias cresçam sem limite.

Como em qualquer padrão, é importante entender o contexto do problema para identificar a melhor solução. Se o contexto muda, a solução provavelmente mudará. Existem duas grandes necessidades de contexto para aplicar o Strategy de maneira efetiva: 1) os fluxos do algoritmo podem ser separados de maneira independente e 2) uma vez que sabemos qual caminho seguir, ele não muda até o final da execução do algoritmo.

Nas seções a seguir vamos comparar o Strategy com os padrões Template Method e State, que podem ajudar quando um dos contextos detalhados anteriormente não puder ser cumprido. Não vamos entrar em detalhes sobre a implementação destes padrões, mas recursos com mais detalhes serão indicados ao final do artigo (veja a seção de Referências Externas).

Evoluindo o Strategy para Template Method

Garantir que o fluxo do algoritmo possa ser separado nem sempre é possível. A vezes as regras que você precisa aplicar não vão permitir que o algoritmo seja quebrado em fluxos diferentes. Uma ideia para começar e validar essa possibilidade é tentar duplicar o código entre as estratégias.

Se o algoritmo separado precisar de muita duplicação, talvez a separação em estratégias não seja a melhor opção. Os ganhos obtidos com a flexibilidade da solução não vão compensar os gastos com a manutenção de código repetido. Nesses casos, outros padrões podem ajudar a resolver o problema, como o Template Method.

Ao aplicar o Template Method definimos uma estrutura base, que será comum a todas as variações de fluxos, além de vários pontos gancho onde podemos variar a implementação. Dessa forma os pontos comuns ficam centralizados na classe mãe e as classes filhas podem implementar sua própria lógica se beneficiando do algoritmo base.

Caso fosse necessário aplicar uma mesma regra em todas as diferentes estratégias (por exemplo, incluir um bônus de 10% para primeiras compras), essa regra precisaria ser duplicada. Poderíamos definir um template que calcula a pontuação baseado na distância, regra que seria definida pelas classes filhas, e sempre adiciona essa bonificação extra ao final.

Evoluindo o Strategy para State

Com relação a segunda necessidade de contexto, precisamos garantir que uma vez que a estratégia é definida ela não mudará, já que a principal vantagem de utilizar o Strategy é diminuir a quantidade de decisões que precisam ser tomadas. Se for preciso fazer várias verificações em pontos diferentes para descobrir qual estratégia utilizar, ou modificar a estratégia atual, a aplicação do padrão não ajudará muito.

Na maioria das vezes é preciso revisitar todo o algoritmo para tentar fazer com que essa decisão só aconteça uma vez. No entanto, caso as regras do algoritmo realmente precisem que o fluxo mude no meio do caminho, pode ser melhor aplicar o padrão State.

Assim como no padrão Strategy, o State também sugere dividir os fluxos do algoritmo mas, ao invés de escolher uma estratégia a ser seguida, criamos estados contendo suas informações e regras.

Esses estados serão facilmente trocados, conforme necessário, pois cada um deles saberá qual deve ser o próximo estado a ser chamado, bem semelhante a uma Máquina de Estados Finita. Assim não precisamos nos preocupar em escolher qual estado utilizar, basta configurar o ponto de partida.

Poderíamos evoluir o código do exemplo anterior para o State caso os status do passageiro tivessem uma maior influência (por exemplo, se para cada status um multiplicador de pontuação diferente fosse aplicado). Ao invés de fazer com que cada estratégia adicione vários ifs para saber o que fazer dependendo do passageiro, cada status seria mapeado para um estado que saberia aplicar suas regras bem como quando o passageiro evoluiu para um novo estado.

A ESTRADA A FRENTE

Com os padrões apresentados aqui, espero que você tenha uma ideia melhor das vantagens de utilizá-los e dos benefícios que eles proporcionam.

Padrões de Projeto são uma excelente ferramenta pois, além dos benefícios no código, eles também facilitam as conversas dentro do time. Se uma pessoa disser que refatorou aquele código que estavam trabalhando ontem pra utilizar um Simple Factory, você já sabe o que foi feito sem precisar entrar em muitos detalhes de como foi implementado.

Como explorado no começo do artigo, existem vários livros que apresentam padrões para linguagens Orientada a Objetos e outros mais específicos para plataformas e linguagens. Procure o material que melhor se adequa ao seu contexto e continue aprendendo sobre essas soluções.

APLICANDO PADRÕES DE PROJETO

Utilizar padrões de projeto também tem suas desvantagens, a principal é que eles criam uma estrutura um tanto quanto complexa de classes, utilizando-se de herança e delegando chamadas para dar ao código final mais flexibilidade na sua utilização e manutenção.

Criar muitas classes não é necessariamente algo ruim, lembre-se da discussão no começo do artigo: a ideia do paradigma Orientado a Objetos é esconder os dados e facilitar a interação entre objetos. Se você está utilizando uma linguagem Orientada a Objetos, não deve ter medo de criar novas classes ou extrair partes do comportamento, mas é importante saber quando vale a pena fazê-lo.

Apesar do código ficar mais simples, separado em pequenos métodos, quem não conhece os padrões vai precisar entender as várias classes que são criadas e seguir o fluxo do código para entender o que está acontecendo.

A principal dica para utilizar Padrões de Projeto de maneira eficiente é: evite escrever código aplicando um Padrão de Projeto, prefira refatorar para um padrão. Se o código não precisa da flexibilidade proporcionada pelos padrões, evite complicá-lo.

Deixe que seu código evolua e que as regras de negócio lhe digam o que deve ser melhorado. Quem desenvolve utilizando TDD (Test-Driven Development) pode pensar em aplicar um padrão durante a etapa de refatoração, pois terá mais entendimento dos requisitos e da implementação necessária, além da segurança dos testes automatizados.

Antes de decidir refatorar o código para aplicar um Padrão de Projeto, veja quais princípios ele está ferindo e tente fazer correções mais simples. Não se torne um dicionário de padrões, forçando situações para aplicá-los. Pense no contexto do seu problema primeiro e depois na solução que irá aplicar.

Refatorando com Padrões de Projeto

Quando o seu contexto realmente permitir aplicar um padrão, vá em frente e refatore o código. Dê pequenos passos, sempre atualizando testes e garantindo que tudo continua funcionando. Encare os Padrões de Projeto como um destino final de uma caminhada que começa com pequenos passos de refatoração.

Outra grande vantagem dos Padrões de Projeto é que eles são soluções que foram aplicadas por várias pessoas e, como ainda são conhecidos até hoje, resolvem bem o problema. Então ao aplicar um padrão você está utilizando um código que já foi validado por várias pessoas em vários projetos diferentes.

No artigo Is Design Dead, (veja a seção de Referências Externas) Martin Fowler sugere alguns pontos para aproveitar ao máximo os padrões de projeto:

- Invista tempo aprendendo sobre padrões;
- Concentre-se em aprender quando aplicá-los (não muito cedo);
- Concentre-se em como implementar padrões na sua forma mais simples, e adicionar complexidade depois;
- Se você adicionar um padrão, e depois perceber que ele não está ajudando, não tenha medo de removê-lo.

No livro "Refatorando com Padrões de Projeto" são apresentados vários padrões, tendo como plano de fundo um código que será refatorado para aplicar um padrão. Se quiser se aprofundar mais no tema, leia mais em: http://www.casadocodigo.com.br/products/livro-refatoracao-ruby

Livro Refatorando com Padrões de Projeto

Espero que o conteúdo apresentado possa ajudar nos primeiros passos e que tenha lhe motivado a ir mais longe. Não deixe de conferir os recursos indicados no final do texto para ter uma visão mais aprofundada do conteúdo e saber quais são os próximos passos.

REFERÊNCIAS EXTERNAS

http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en

Email do Dr. Alan Kay sobre a origiem da Programação Orientada a Objetos

http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Artigos sobre Princípios de Design Orientado a Objetos no site de Robert C. Martin

http://martinfowler.com/articles/designDead.html

Artigo “Is Design Dead” no site de Martin Fowler

http://casadocodigo.com.br/products/livro-refatoracao-ruby

Livro Refatorando com padrões de projeto de Marcos Brizeno

https://brizeno.wordpress.com/padroes

Exemplos com outros padrões da Gangue dos Quatros no site de Marcos Brizeno

SOBRE O AUTOR

Cientista da Computação pela Universidade Estadual do Ceará e Consultor de Desenvolvimento na ThoughtWorks Brazil. Apaixonado por Engenharia de Software e Metodologias Ágeis.

Autor do livro Refatorando com Padrões de Projetos[1] e com contribuições no Thoughtworks Antologia Brasil[2] e Práticas e Tendências em Testes[3].

Publica regularmente no blog brizeno.wordpress.com[4] e em @marcosbrizeno[5].

[1] https://www.casadocodigo.com.br/products/livro-refatoracao-ruby
[2] https://www.casadocodigo.com.br/pages/sumario-thoughtworks-antologia
[3] https://info.thoughtworks.com/praticas-e-tendencias-em-teste-ebook.html
[4] https://brizeno.wordpress.com
[5] https://twitter.com/marcosbrizeno