

Mergulho nos **PADRÕES** *DE PROJETO*



Alexander Shvets

Mergulho nos **PADRÕES** DE **PROJETO**

v2021-1.14

VERSÃO DEMONSTRATIVA

Compre o livro completo:

<https://refactoring.guru/pt-br/design-patterns/book>

Algumas poucas palavras sobre direitos autorais

Oi! Meu nome é Alexander Shvets. Sou o autor do livro **Mergulho nos Padrões de Projeto** e o curso online **Mergulho na Refatoração**.



Este livro é apenas para seu uso pessoal. Por favor, não compartilhe-o com terceiros exceto seus membros da família. Se você gostaria de compartilhar o livro com um amigo ou colega, compre e envie uma nova cópia para eles. Você também pode comprar uma licença de instalação para toda a sua equipe ou para toda a empresa.

Toda a renda das vendas de meus livros e cursos é gasta no desenvolvimento do **Refactoring.Guru**. Cada cópia vendida ajuda o projeto imensamente e faz o momento do lançamento de um novo livro ficar cada vez mais perto.

© Alexander Shvets, Refactoring.Guru, 2021

✉ **support@refactoring.guru**

🖼 Ilustrações: Dmitry Zhart

🇧🇷 Tradução: Fernando Schumann

✎ Edição: Gabriel Chaves

*Eu dedico este livro para minha esposa, Maria.
Se não fosse por ela, eu provavelmente teria
terminado o livro só daqui a uns 30 anos.*

Índice

| | |
|---|-----------|
| Índice | 4 |
| Como ler este livro | 6 |
| INTRODUÇÃO À PROGRAMAÇÃO ORIENTADA A OBJETOS | 7 |
| Básico da POO | 8 |
| Pilares da POO | 14 |
| Relações entre objetos..... | 22 |
| INTRODUÇÃO AOS PADRÕES DE PROJETO | 28 |
| O que é um padrão de projeto? | 29 |
| Por que devo aprender padrões?..... | 34 |
| PRINCÍPIOS DE PROJETO DE SOFTWARE..... | 35 |
| Características de um bom projeto..... | 36 |
| Princípios de projeto..... | 41 |
| § Encapsule o que varia | 42 |
| § Programe para uma interface, não uma implementação | 47 |
| § Prefira composição sobre herança | 52 |
| Princípios SOLID | 56 |
| § S: Princípio de responsabilidade única..... | 57 |
| § O: Princípio aberto/fechado | 59 |
| § L: Princípio de substituição de Liskov | 63 |
| § I: Princípio de segregação de interface | 70 |
| § D: Princípio de inversão de dependência | 73 |

| | |
|---|------------|
| CATÁLOGO DOS PADRÕES DE PROJETO | 77 |
| Padrões de projeto criacionais..... | 78 |
| § Factory Method | 80 |
| § Abstract Factory | 97 |
| § Builder | 113 |
| § Prototype | 134 |
| § Singleton | 150 |
| Padrões de projeto estruturais | 160 |
| § Adapter | 163 |
| § Bridge | 177 |
| § Composite | 194 |
| § Decorator | 208 |
| § Facade | 228 |
| § Flyweight | 239 |
| § Proxy | 254 |
| Padrões de projeto comportamentais | 268 |
| § Chain of Responsibility | 272 |
| § Command | 292 |
| § Iterator | 313 |
| § Mediator | 329 |
| § Memento | 345 |
| § Observer | 362 |
| § State | 378 |
| § Strategy | 395 |
| § Template Method | 410 |
| § Visitor | 424 |
| Conclusão | 441 |

Como ler este livro

Este livro contém as descrições de 22 padrões de projeto clássicos formulados pela “Gangue dos Quatro” (ing. “Gang of Four”, ou simplesmente GoF) em 1994.

Cada capítulo explora um padrão em particular. Portanto, você pode ler de cabo a rabo ou escolher aqueles padrões que você está interessado.

Muitos padrões são relacionados, então você pode facilmente pular de tópico para tópico usando numerosos links. O fim de cada capítulo tem uma lista de links entre o padrão atual e outros. Se você ver o nome de um padrão que você não viu ainda, apenas continue lendo—este item irá aparecer em um dos próximos capítulos.

Os padrões de projeto são universais. Portanto, todos os exemplos de código neste livro são em pseudocódigo que não prendem o material a uma linguagem de programação em particular.

Antes de estudar os padrões, você pode refrescar sua memória indo até **os termos chave da programação orientada a objetos**. Aquele capítulo também explica o básico sobre diagramas UML, que são úteis porque o livro tem um monte deles. É claro, se você já sabe de tudo isso, você pode seguir direto para **aprender os padrões patterns**.

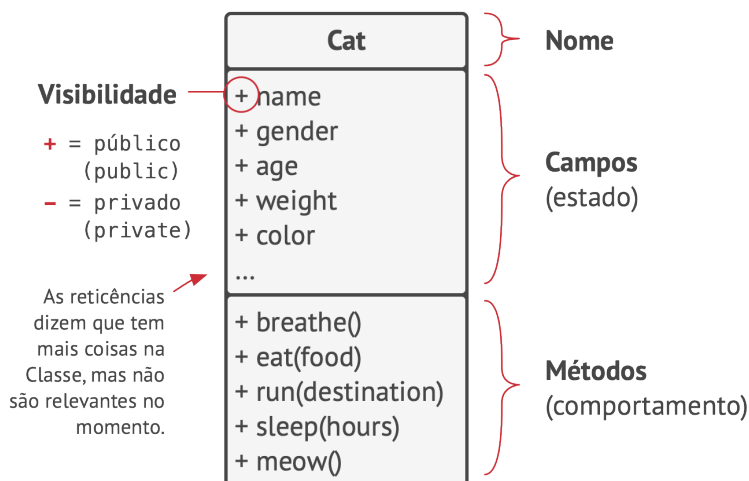
INTRODUÇÃO À PROGRAMAÇÃO ORIENTADA A OBJETOS

Básico da POO

A **Programação Orientada à Objetos** (POO) é um paradigma baseado no conceito de envolver pedaços de dados, e comportamentos relacionados aqueles dados, em uma coleção chamada **objetos**, que são construídos de um conjunto de “planos de construção”, definidos por um programador, chamados de **classes**.

Objetos, classes

Você gosta de gatos? Espero que sim, porque vou tentar explicar os conceitos da POO usando vários exemplos com gatos.



Este é um diagrama UML da classe. UML é a sigla do inglês Unified Modeling Language e significa Linguagem de Modelagem Unificada.

Você verá muitos desses diagramas no livro.

É uma prática comum deixar os nomes dos membros e da classe nos diagramas em inglês, como faríamos em um código real. No entanto, comentários e notas também podem ser escritos em português.

Neste livro, posso citar nomes de classes em português, mesmo que apareçam em diagramas ou em código em inglês (assim como fiz com a classe `Gato`). Quero que você leia o livro como se estivéssemos conversando entre amigos. Não quero que você encontre palavras estranhas toda vez que eu precisar fazer referência a alguma aula.

Digamos que você tenha um gato chamado Tom. Tom é um objeto, uma instância da classe `Gato`. Cada gato tem uma porção de atributos padrão: nome, gênero, idade, peso, cor, etc. Estes são chamados os *campos* de uma classe.

Todos os gatos também se comportam de forma semelhante: eles respiram, comem, correm, dormem, e miam. Estes são os *métodos* da classe. Coletivamente, os campos e os métodos podem ser referenciados como *membros* de suas classes.

Dados armazenados dentro dos campos do objeto são referenciados como *estados*, e todos os métodos de um objeto definem seu *comportamento*.

**Tom: Cat**

```

name   = "Tom"
sex    = "macho"
age    = 3
weight = 7
color  = marrom
texture = listrada
  
```

**Nina: Cat**

```

name   = "Nina"
sex    = "fêmea"
age    = 2
weight = 5
color  = cinza
texture = lisa
  
```

Objetos são instâncias de classes.

Nina, a gata do seu amigo também é uma instância da classe **Gato**. Ela tem o mesmo conjunto de atributos que Tom. A diferença está nos valores destes seus atributos: seu gênero é fêmea, ela tem uma cor diferente, e pesa menos.

Então uma *classe* é como uma planta de construção que define a estrutura para *objetos*, que são instâncias concretas daquela classe.

Hierarquias de classe

Tudo é uma beleza quando se trata de apenas uma classe. Naturalmente, um programa real contém mais que apenas uma

classe. Algumas dessas classes podem ser organizadas em **hierarquias de classes**. Vamos descobrir o que isso significa.

Digamos que seu vizinho tem um cão chamado Fido. Acontece que cães e gatos têm muito em comum: nome, gênero, idade, e cor são atributos de ambos cães e gatos. Cães podem respirar, dormir, e correr da mesma forma que os gatos. Então parece que podemos definir a classe base `Animal` que listaria os atributos e comportamentos em comum.

Uma classe mãe, como a que recém definimos, é chamada de uma **superclasse**. Suas filhas são as **subclasses**. Subclasses herdam estado e comportamento de sua mãe, definindo apenas atributos e comportamentos que diferem. Portanto, a classe `Gato` teria o método `miado` e a classe `Cão` o método `latido`.

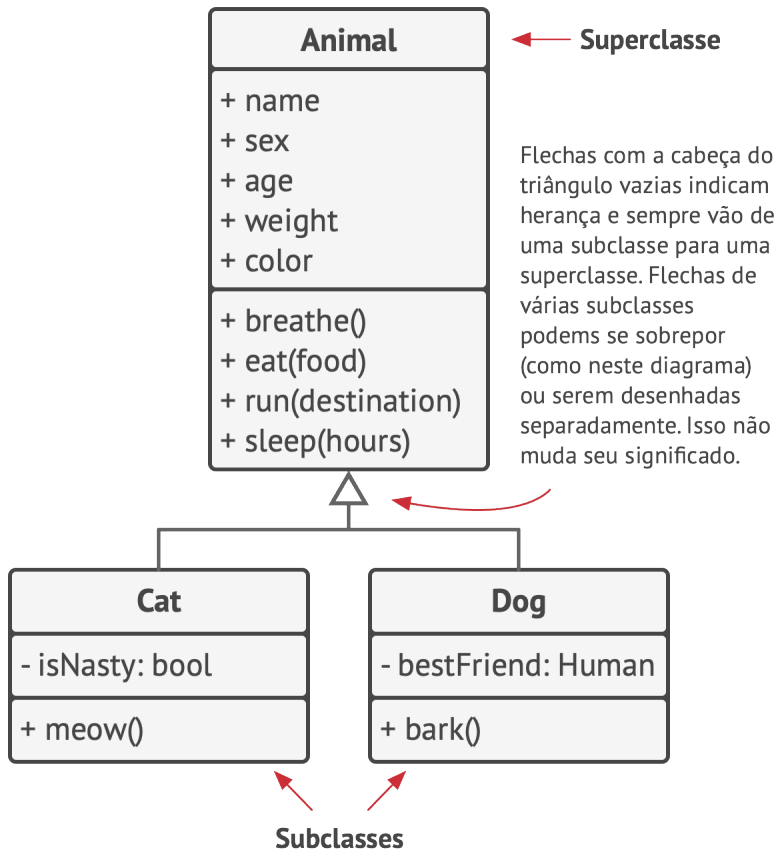
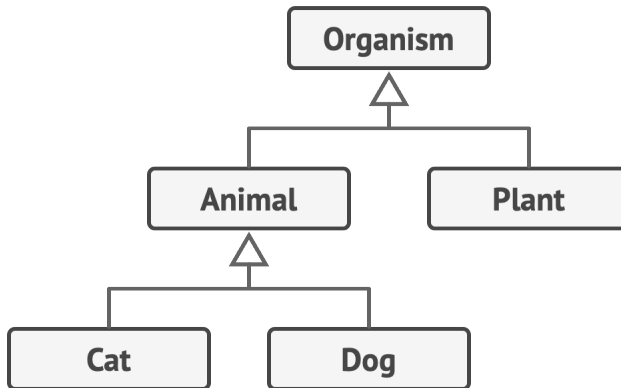


Diagrama UML de uma hierarquia de classe. Todas as classes neste diagrama são parte da hierarquia de classe `Animal`.

Assumindo que temos um requisito de negócio parecido, podemos ir além e extrair uma classe ainda mais geral de todos os `Organismos` que será a superclasse para `Animais` e `Plantas`. Tal pirâmide de classes é uma **hierarquia**. Em tal hierarquia, a classe `Gato` herda tudo que veio das classes `Animal` e `Organismos`.



Classes em um diagrama UML podem ser simplificadas se é mais importante mostrar suas relações que seus conteúdos.

Subclasses podem sobrescrever o comportamento de métodos que herdaram de suas classes parentes. Uma subclasse pode tanto substituir completamente o comportamento padrão ou apenas melhorá-lo com coisas adicionais.

21 páginas

do livro completo são omitidas na versão demonstrativa

PRINCÍPIOS DE PROJETO DE SOFTWARE

Características de um bom projeto

Antes de prosseguirmos para os próprios padrões, vamos discutir o processo de arquitetura do projeto de software: coisas que devemos almejar e coisas que devemos evitar.

Reutilização de código

Custo e tempo são duas das mais valiosas métricas quando desenvolvendo qualquer produto de software. Menos tempo de desenvolvimento significa entrar no mercado mais cedo que os competidores. Baixo custo de desenvolvimento significa que mais dinheiro pode ser usado para marketing e uma busca maior para clientes em potencial.

A **reutilização de código** é um dos modos mais comuns para se reduzir custos de desenvolvimento. O propósito é simples: ao invés de desenvolver algo novamente e do zero, por que não reutilizar código já existente em novos projetos?

A ideia parece boa no papel, mas fazer um código já existente funcionar em um novo contexto geralmente exige esforço adicional. O firme acoplamento entre os componentes, dependências de classes concretas ao invés de interfaces, operações codificadas (hardcoded)—tudo isso reduz a flexibilidade do código e torna mais difícil reutilizá-lo.

Utilizar padrões de projeto é uma maneira de aumentar a flexibilidade dos componente do software e torná-los de mais fácil reutilização. Contudo, isso às vezes vem com um preço de tornar os componentes mais complicados.

Eis aqui um fragmento de sabedoria de Erich Gamma¹, um dos fundadores dos padrões de projeto, sobre o papel dos padrões de projeto na reutilização de código:

“

Eu vejo três níveis de reutilização.

No nível mais baixo, você reutiliza classes: classes de bibliotecas, contêineres, talvez “times” de classes como o contêiner/iterator.

Os frameworks são o mais alto nível. Eles realmente tentam destilar suas decisões de projeto. Eles identificam abstrações importantes para a solução de um problema, representam eles por classes, e definem relações entre eles. O JUnit é um pequeno framework, por exemplo. Ele é o “Olá, mundo” dos frameworks. Ele tem o `Test`, `TestCase`, `TestSuite` e relações definidas.

Um framework é tipicamente mais bruto que apenas uma única classe. Também, você estará lidando com frameworks se fizer subclasses em algum lugar. Eles usam o chamado princípio Hollywood de “não nos chamem, nós chamaremos você”. O framework permite que você defina seu comportamento cus-

1. Erich Gamma e a Flexibilidade e Reutilização: <https://refactoring.guru/gamma-interview>

tomizado, e ele irá chamar você quando for sua vez de fazer alguma coisa. É a mesma coisa com o JUnit, não é? Ele te chama quando quer executar um teste para você, mas o resto acontece dentro do framework.

Há também um nível médio. É aqui que eu vejo os padrões. Os padrões de projeto são menores e mais abstratos que os frameworks. Eles são uma verdadeira descrição de como um par de classes pode se relacionar e interagir entre si. O nível de reutilização aumenta quando você move de classes para padrões e, finalmente, para frameworks.

O que é legal sobre essa camada média é que os padrões oferecem reutilização em uma maneira que é menos arriscada que a dos frameworks. Construir um framework é algo de alto risco e investimento. Os padrões podem reutilizar ideias e conceitos de projeto independentemente do código concreto.

”



Extensibilidade

Mudança é a única constante na vida de um programador.

- Você publicou um vídeo game para Windows, mas as pessoas pedem uma versão para macOS.
- Você criou um framework de interface gráfica com botões quadrados, mas, meses mais tarde, botões redondos é que estão na moda.
- Você desenhou uma arquitetura brilhante para um site de e-commerce, mas apenas um mês mais tarde os clientes pedem

por uma funcionalidade que permita que eles aceitem pedidos pelo telefone.

Cada desenvolvedor de software tem dúzias de histórias parecidas. Há vários motivos porque isso ocorre.

Primeiro, nós entendemos o problema melhor quando começamos a resolvê-lo. Muitas vezes, quando você termina a primeira versão da aplicação, você já está pronto para reescrevê-la do nada porque agora você entende muito bem dos vários aspectos do problema. Você também cresceu profissionalmente, e seu código antigo é horrível.

Algo além do seu controle mudou. Isso é porque muitos das equipes de desenvolvimento saem do eixo de suas ideias originais para algo novo. Todos que confiaram no Flash para uma aplicação online estão reformulando ou migrando seu código para um navegador após o fim do suporte ao Flash pelos navegadores.

O terceiro motivo é que as regras do jogo mudam. Seu cliente estava muito satisfeito com a versão atual da aplicação, mas agora vê onze “pequenas” mudanças que ele gostaria para que ele possa fazer outras coisas que ele nunca mencionou nas sessões de planejamento inicial. Essas não são modificações frívolas: sua excelente primeira versão mostrou para ele que algo mais era possível.

Há um lado bom: se alguém pede que você mude algo em sua aplicação, isso significa que alguém ainda se importa com ela.

Isso é porque todos os desenvolvedores veteranos tentam se precaver para futuras mudanças quando fazendo o projeto da arquitetura de uma aplicação.

Princípios de projeto

O que é um bom projeto de software? Como você pode medi-lo? Que práticas você deveria seguir para conseguir isso? Como você pode fazer sua arquitetura flexível, estável, e fácil de se entender?

Estas são boas perguntas; mas, infelizmente, as respostas são diferentes dependendo do tipo de aplicação que você está construindo. De qualquer forma, há vários princípios universais de projeto de software que podem ajudar você a responder essa perguntas para seu próprio projeto. A maioria dos padrões de projeto listados neste livro são baseados nestes princípios.

Encapsule o que varia

Identifique os aspectos da sua aplicação que variam e separe-os dos que permanecem os mesmos.

O objetivo principal deste princípio é minimizar o efeito causado por mudanças.

Imagine que seu programa é um navio, e as mudanças são terríveis minas que se escondem sob as águas. Atinja a mina e o navio afunda.

Sabendo disso, você pode dividir o casco do navio em compartimentos independentes que podem ser facilmente selados para limitar os danos a um único compartimento. Agora, se o navio atingir uma mina, o navio como um todo vai continuar à tona.

Da mesma forma, você pode isolar as partes de um programa que variam em módulos independentes, protegendo o resto do código de efeitos adversos. Dessa forma você gasta menos tempo fazendo o programa voltar a funcionar, implementando e testando as mudanças. Quanto menos tempo você gasta fazendo mudanças, mais tempo você tem para implementar novas funcionalidades.

Encapsulamento à nível de método

Digamos que você está fazendo um website de e-commerce. Em algum lugar do seu código há um método `obterTotalPedido` que calcula o total final de um pedido, incluindo impostos. Nós podemos antecipar que o código relacionado aos impostos precisa mudar no futuro. O rateio da taxa depende do país, estado, ou até mesmo cidade onde o cliente reside, e a fórmula atual pode mudar com o tempo devido a novas leis ou regulamentações. Como resultado, você irá precisar mudar o método `obterTotalPedido` com certa frequência. Mas até mesmo o nome do método sugere que ele não se importa *como* os impostos são calculados.

```
1 method getOrderTotal(order) is
2     total = 0
3     foreach item in order.lineItems
4         total += item.price * item.quantity
5
6     if (order.country == "US")
7         // Imposto das vendas nos EUA.
8         total += total * 0.07
9     else if (order.country == "EU"):
10        // Imposto sobre o valor acrescentado.
11        total += total * 0.20
12
13    return total
```

ANTES: código de cálculo de impostos misturado com o resto do código do método.

Você pode extrair a lógica do cálculo do imposto em um método separado, escondendo-o do método original.

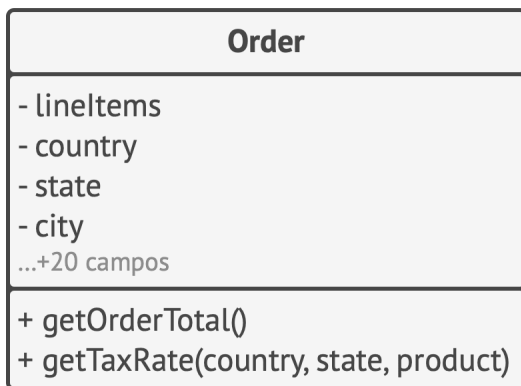
```
1  method getOrderTotal(order) is
2      total = 0
3      foreach item in order.lineItems
4          total += item.price * item.quantity
5
6      total += total * getTaxRate(order.country)
7
8      return total
9
10 method getTaxRate(country) is
11     if (country == "US")
12         // Imposto das vendas nos EUA.
13         return 0.07
14     else if (country == "EU")
15         // Imposto sobre o valor acrescentado.
16         return 0.20
17     else
18         return 0
```

DEPOIS: você pode obter o rateio de impostos chamando o método designado para isso.

As mudanças relacionadas aos impostos se tornaram isoladas em um único método. E mais, se o cálculo da lógica de impostos se tornar muito complexo, fica agora mais fácil movê-lo para uma classe separada.

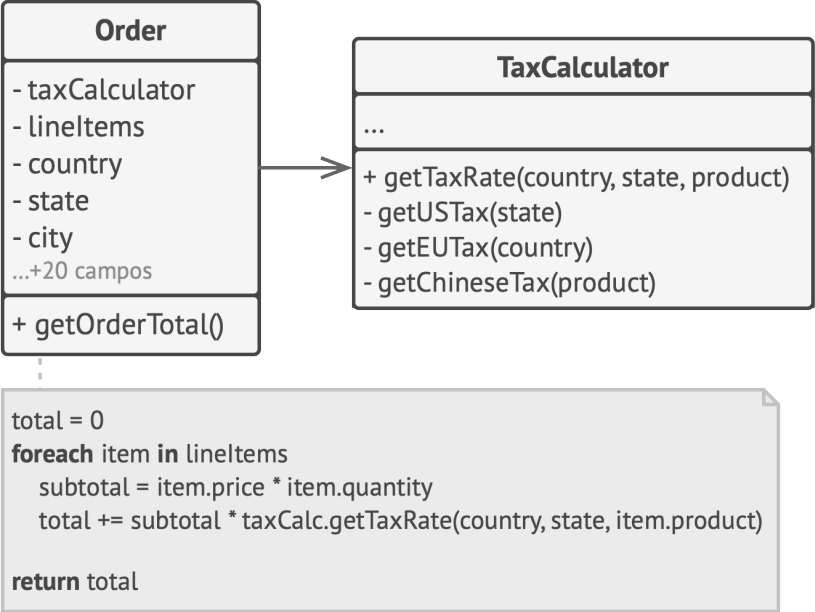
Encapsulamento a nível de classe

Com o tempo você pode querer adicionar mais e mais responsabilidades para um método que é usado para fazer uma coisa simples. Esses comportamentos adicionais quase sempre vem com seus próprios campos de ajuda e métodos que eventualmente desfocam a responsabilidade primária da classe que o contém. Extraíndo tudo para uma nova classe pode tornar as coisas mais claras e simples.



ANTES: calculando impostos em uma classe *Pedido* .

Objetos da classe `Pedido` delegam todo o trabalho relacionado a impostos para um objeto especial que fará isso.



DEPOIS: cálculo dos impostos está escondido da classe do `Pedido`.

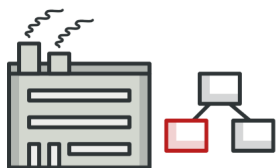
30 páginas

do livro completo são omitidas na versão demonstrativa

CATÁLOGO DOS PADRÕES DE PROJETO

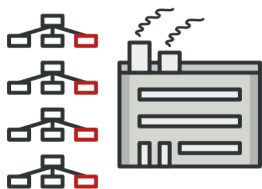
Padrões de projeto criacionais

Os padrões criacionais fornecem vários mecanismos de criação de objetos, que aumentam a flexibilidade e reutilização de código já existente.



Factory Method

Fornece uma interface para criar objetos em uma superclasse, mas permite que as subclasses alterem o tipo de objetos que serão criados.



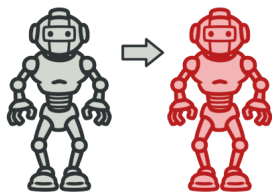
Abstract Factory

Permite que você produza famílias de objetos relacionados sem ter que especificar suas classes concretas.



Builder

Permite construir objetos complexos passo a passo. O padrão permite produzir diferentes tipos e representações de um objeto usando o mesmo código de construção.



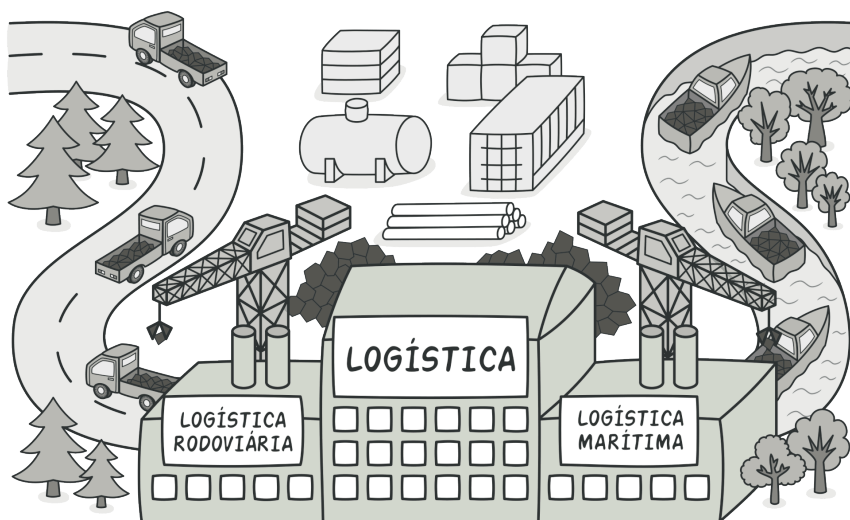
Prototype

Permite que você copie objetos existentes sem fazer seu código ficar dependente de suas classes.



Singleton

Permite a você garantir que uma classe tem apenas uma instância, enquanto provê um ponto de acesso global para esta instância.



FACTORY METHOD

Também conhecido como: Método fábrica, Construtor virtual

O **Factory Method** é um padrão criacional de projeto que fornece uma interface para criar objetos em uma superclasse, mas permite que as subclasses alterem o tipo de objetos que serão criados.

☹ Problema

Imagine que você está criando uma aplicação de gerenciamento de logística. A primeira versão da sua aplicação pode lidar apenas com o transporte de caminhões, portanto a maior parte do seu código fica dentro da classe `Caminhão`.

Depois de um tempo, sua aplicação se torna bastante popular. Todos os dias você recebe dezenas de solicitações de empresas de transporte marítimo para incorporar a logística marítima na aplicação.



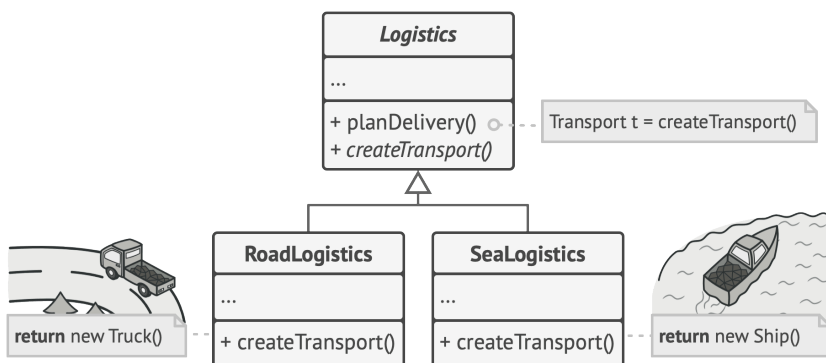
Adicionar uma nova classe ao programa não é tão simples se o restante do código já estiver acoplado às classes existentes.

Boa notícia, certo? Mas e o código? Atualmente, a maior parte do seu código é acoplada à classe `Caminhão`. Adicionar `Navio` à aplicação exigiria alterações em toda a base de código. Além disso, se mais tarde você decidir adicionar outro tipo de transporte à aplicação, provavelmente precisará fazer todas essas alterações novamente.

Como resultado, você terá um código bastante sujo, repleto de condicionais que alteram o comportamento da aplicação, dependendo da classe de objetos de transporte.

😊 Solução

O padrão Factory Method sugere que você substitua chamadas diretas de construção de objetos (usando o operador `new`) por chamadas para um método *fábrica* especial. Não se preocupe: os objetos ainda são criados através do operador `new`, mas esse está sendo chamado de dentro do método fábrica. Objetos retornados por um método fábrica geralmente são chamados de *produtos*.

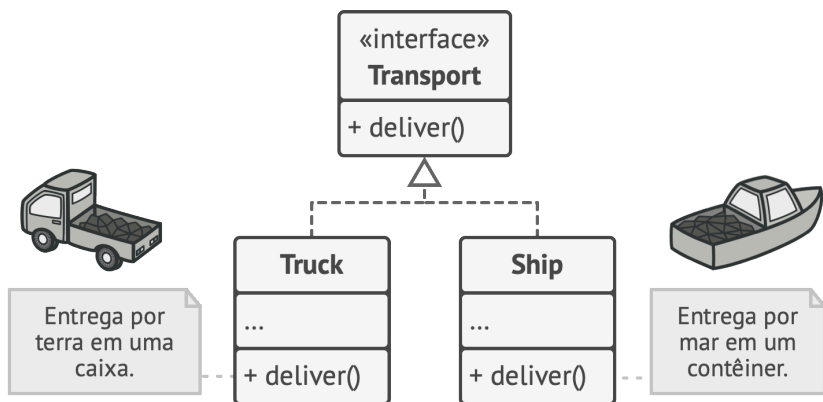


As subclasses podem alterar a classe de objetos retornados pelo método fábrica.

À primeira vista, essa mudança pode parecer sem sentido: apenas mudamos a chamada do construtor de uma parte do programa para outra. No entanto, considere o seguinte: agora você pode sobrescrever o método fábrica em uma subclasse

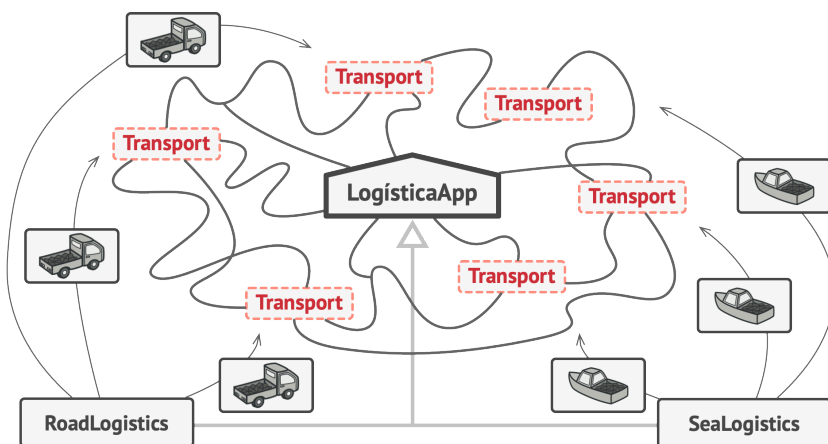
e alterar a classe de produtos que estão sendo criados pelo método.

Porém, há uma pequena limitação: as subclasses só podem retornar tipos diferentes de produtos se esses produtos tiverem uma classe ou interface base em comum. Além disso, o método fábrica na classe base deve ter seu tipo de retorno declarado como essa interface.



Todos os produtos devem seguir a mesma interface.

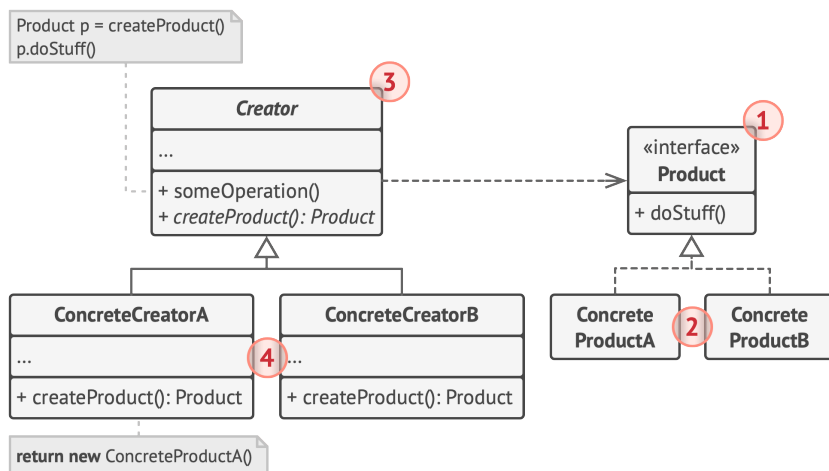
Por exemplo, ambas as classes `Caminhão` e `Navio` devem implementar a interface `Transporte`, que declara um método chamado `entregar`. Cada classe implementa esse método de maneira diferente: caminhões entregam carga por terra, navios entregam carga por mar. O método fábrica na classe `LogísticaViária` retorna objetos de caminhão, enquanto o método fábrica na classe `LogísticaMarítima` retorna navios.



Desde que todas as classes de produtos implementem uma interface comum, você pode passar seus objetos para o código cliente sem quebrá-lo.

O código que usa o método fábrica (geralmente chamado de código *cliente*) não vê diferença entre os produtos reais retornados por várias subclasses. O cliente trata todos os produtos como um `Transporte` abstrato. O cliente sabe que todos os objetos de transporte devem ter o método `entregar`, mas como exatamente ele funciona não é importante para o cliente.

Estrutura



1. O **Produto** declara a interface, que é comum a todos os objetos que podem ser produzidos pelo criador e suas subclasses.
2. **Produtos Concretos** são implementações diferentes da interface do produto.
3. A classe **Criador** declara o método fábrica que retorna novos objetos produto. É importante que o tipo de retorno desse método corresponda à interface do produto.

Você pode declarar o método fábrica como abstrato para forçar todas as subclasses a implementar suas próprias versões do método. Como alternativa, o método fábrica base pode retornar algum tipo de produto padrão.

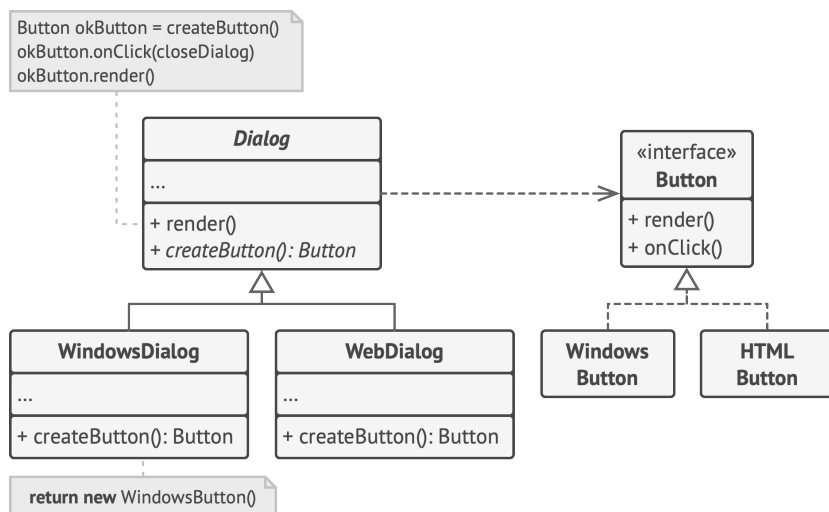
Observe que, apesar do nome, a criação de produtos **não** é a principal responsabilidade do criador. Normalmente, a classe criadora já possui alguma lógica de negócio relacionada aos produtos. O método fábrica ajuda a dissociar essa lógica das classes concretas de produtos. Aqui está uma analogia: uma grande empresa de desenvolvimento de software pode ter um departamento de treinamento para programadores. No entanto, a principal função da empresa como um todo ainda é escrever código, não produzir programadores.

4. **Criadores Concretos** sobrescrevem o método fábrica base para retornar um tipo diferente de produto.

Observe que o método fábrica não precisa **criar** novas instâncias o tempo todo. Ele também pode retornar objetos existentes de um cache, um conjunto de objetos, ou outra fonte.

Pseudocódigo

Este exemplo ilustra como o **Factory Method** pode ser usado para criar elementos de interface do usuário multiplataforma sem acoplar o código do cliente às classes de UI concretas.



Exemplo de diálogo de plataforma cruzada.

A classe base diálogo usa diferentes elementos da UI do usuário para renderizar sua janela. Em diferentes sistemas operacionais, esses elementos podem parecer um pouco diferentes, mas ainda devem se comportar de forma consistente. Um botão no Windows ainda é um botão no Linux.

Quando o método fábrica entra em ação, você não precisa reescrever a lógica da caixa de diálogo para cada sistema operacional. Se declararmos um método fábrica que produz botões dentro da classe base da caixa de diálogo, mais tarde podemos criar uma subclasse de caixa de diálogo que retorna botões no estilo Windows do método fábrica. A subclasse herda a maior parte do código da caixa de diálogo da classe base, mas, graças ao método fábrica, pode renderizar botões estilo Windows na tela.

Para que esse padrão funcione, a classe base da caixa de diálogo deve funcionar com botões abstratos: uma classe base ou uma interface que todos os botões concretos seguem. Dessa forma, o código da caixa de diálogo permanece funcional, independentemente do tipo de botão com o qual ela trabalha.

Obviamente, você também pode aplicar essa abordagem a outros elementos da UI. No entanto, com cada novo método fábrica adicionado à caixa de diálogo, você se aproxima do padrão **Abstract Factory**. Não se preocupe, falaremos sobre esse padrão mais tarde.

```

1  // A classe criadora declara o método fábrica que deve retornar
2  // um objeto de uma classe produto. As subclasses da criadora
3  // geralmente fornecem a implementação desse método.
4  class Dialog is
5      // A criadora também pode fornecer alguma implementação
6      // padrão do Factory Method.
7      abstract method createButton():Button
8
9      // Observe que, apesar do seu nome, a principal
10     // responsabilidade da criadora não é criar produtos. Ela
11     // geralmente contém alguma lógica de negócio central que
12     // depende dos objetos produto retornados pelo método
13     // fábrica. As subclasses pode mudar indiretamente essa
14     // lógica de negócio ao sobrescreverem o método fábrica e
15     // retornarem um tipo diferente de produto dele.
16     method render() is
17         // Chame o método fábrica para criar um objeto produto.
18         Button okButton = createButton()


```




```
19     // Agora use o produto.
20     okButton.onClick(closeDialog)
21     okButton.render()
22
23
24     // Criadores concretos sobrescrevem o método fábrica para mudar
25     // o tipo de produto resultante.
26     class WindowsDialog extends Dialog is
27         method createButton():Button is
28             return new WindowsButton()
29
30     class WebDialog extends Dialog is
31         method createButton():Button is
32             return new HTMLButton()
33
34
35     // A interface do produto declara as operações que todos os
36     // produtos concretos devem implementar.
37     interface Button is
38         method render()
39         method onClick(f)
40
41     // Produtos concretos fornecem várias implementações da
42     // interface do produto.
43     class WindowsButton implements Button is
44         method render(a, b) is
45             // Renderiza um botão no estilo Windows.
46         method onClick(f) is
47             // Vincula um evento de clique do SO nativo.
48
49     class HTMLButton implements Button is
50         method render(a, b) is
```


```
51     // Retorna uma representação HTML de um botão.
52     method onClick(f) is
53         // Vincula um evento de clique no navegador web.
54
55
56     class Application is
57         field dialog: Dialog
58
59         // A aplicação seleciona um tipo de criador dependendo da
60         // configuração atual ou definições de ambiente.
61         method initialize() is
62             config = readApplicationConfigFile()
63
64             if (config.OS == "Windows") then
65                 dialog = new WindowsDialog()
66             else if (config.OS == "Web") then
67                 dialog = new WebDialog()
68             else
69                 throw new Exception("Error! Unknown operating system.")
70
71         // O código cliente trabalha com uma instância de um criador
72         // concreto, ainda que com sua interface base. Desde que o
73         // cliente continue trabalhando com a criadora através da
74         // interface base, você pode passar qualquer subclasse da
75         // criadora.
76         method main() is
77             this.initialize()
78             dialog.render()
```


Aplicabilidade

 **Use o Factory Method quando não souber de antemão os tipos e dependências exatas dos objetos com os quais seu código deve funcionar.**

 O Factory Method separa o código de construção do produto do código que realmente usa o produto. Portanto, é mais fácil estender o código de construção do produto independentemente do restante do código.

Por exemplo, para adicionar um novo tipo de produto à aplicação, só será necessário criar uma nova subclasse criadora e substituir o método fábrica nela.

 **Use o Factory Method quando desejar fornecer aos usuários da sua biblioteca ou framework uma maneira de estender seus componentes internos.**

 Herança é provavelmente a maneira mais fácil de estender o comportamento padrão de uma biblioteca ou framework. Mas como o framework reconheceria que sua subclasse deve ser usada em vez de um componente padrão?

A solução é reduzir o código que constrói componentes no framework em um único método fábrica e permitir que qualquer pessoa sobrescreva esse método, além de estender o próprio componente.

Vamos ver como isso funcionaria. Imagine que você escreva uma aplicação usando um framework de UI de código aberto. Sua aplicação deve ter botões redondos, mas o framework fornece apenas botões quadrados. Você estende a classe padrão `Botão` com uma gloriosa subclasse `BotãoRedondo`. Mas agora você precisa informar à classe principal `UIFramework` para usar a nova subclasse no lugar do botão padrão.

Para conseguir isso, você cria uma subclasse `UIComBotõesRedondos` a partir de uma classe base do framework e sobrescreve seu método `criarBotão`. Enquanto este método retorna objetos `Botão` na classe base, você faz sua subclasse retornar objetos `BotãoRedondo`. Agora use a classe `UIComBotõesRedondos` no lugar de `UIFramework`. É isso!



Use o Factory Method quando deseja economizar recursos do sistema reutilizando objetos existentes em vez de recriá-los sempre.



Você irá enfrentar essa necessidade ao lidar com objetos grandes e pesados, como conexões com bancos de dados, sistemas de arquivos e recursos de rede.

Vamos pensar no que deve ser feito para reutilizar um objeto existente:

1. Primeiro, você precisa criar algum armazenamento para manter o controle de todos os objetos criados.

2. Quando alguém solicita um objeto, o programa deve procurar um objeto livre dentro desse conjunto.
3. ...e retorná-lo ao código cliente.
4. Se não houver objetos livres, o programa deve criar um novo (e adicioná-lo ao conjunto de objetos).

Isso é muito código! E tudo deve ser colocado em um único local para que você não polua o programa com código duplicado.

Provavelmente, o lugar mais óbvio e conveniente onde esse código deve ficar é no construtor da classe cujos objetos estamos tentando reutilizar. No entanto, um construtor deve sempre retornar **novos objetos** por definição. Não pode retornar instâncias existentes.

Portanto, você precisa ter um método regular capaz de criar novos objetos e reutilizar os existentes. Isso parece muito com um método fábrica.



Como implementar

1. Faça todos os produtos implementarem a mesma interface. Essa interface deve declarar métodos que fazem sentido em todos os produtos.

2. Adicione um método fábrica vazio dentro da classe criadora. O tipo de retorno do método deve corresponder à interface comum do produto.
3. No código da classe criadora, encontre todas as referências aos construtores de produtos. Um por um, substitua-os por chamadas ao método fábrica, enquanto extrai o código de criação do produto para o método fábrica.

Pode ser necessário adicionar um parâmetro temporário ao método fábrica para controlar o tipo de produto retornado.

Neste ponto, o código do método fábrica pode parecer bastante feio. Pode ter um grande operador `switch` que escolhe qual classe de produto instanciar. Mas não se preocupe, resolveremos isso em breve.

4. Agora, crie um conjunto de subclasses criadoras para cada tipo de produto listado no método fábrica. Sobrescreva o método fábrica nas subclasses e extraia os pedaços apropriados do código de construção do método base.
5. Se houver muitos tipos de produtos e não fizer sentido criar subclasses para todos eles, você poderá reutilizar o parâmetro de controle da classe base nas subclasses.

Por exemplo, imagine que você tenha a seguinte hierarquia de classes: a classe base `Correio` com algumas subclasses: `CorreioAéreo` e `CorreioTerrestre`; as classes `Transporte`

são `Avião`, `Caminhão` e `Trem`. Enquanto a classe `CorreioAéreo` usa apenas objetos `Avião`, o `CorreioTerrestre` pode funcionar com os objetos `Caminhão` e `Trem`. Você pode criar uma nova subclasse (por exemplo, `CorreioFerroviário`) para lidar com os dois casos, mas há outra opção. O código do cliente pode passar um argumento para o método fábrica da classe `CorreioTerrestre` para controlar qual produto ele deseja receber.

6. Se, após todas as extrações, o método fábrica base ficar vazio, você poderá torná-lo abstrato. Se sobrar algo, você pode tornar isso em um comportamento padrão do método.

Prós e contras

- ✓ Você evita acoplamentos firmes entre o criador e os produtos concretos.
- ✓ *Princípio de responsabilidade única.* Você pode mover o código de criação do produto para um único local do programa, facilitando a manutenção do código.
- ✓ *Princípio aberto/fechado.* Você pode introduzir novos tipos de produtos no programa sem quebrar o código cliente existente.
- ✗ O código pode se tornar mais complicado, pois você precisa introduzir muitas subclasses novas para implementar o padrão. O melhor cenário é quando você está introduzindo o padrão em uma hierarquia existente de classes criadoras.

⇔ Relações com outros padrões

- Muitos projetos começam usando o **Factory Method** (menos complicado e mais customizável através de subclasses) e evoluem para o **Abstract Factory**, **Prototype**, ou **Builder** (mais flexíveis, mas mais complicados).
- Classes **Abstract Factory** são quase sempre baseadas em um conjunto de **métodos fábrica**, mas você também pode usar o **Prototype** para compor métodos dessas classes.
- Você pode usar o **Factory Method** junto com o **Iterator** para permitir que uma coleção de subclasses retornem diferentes tipos de iteradores que são compatíveis com as coleções.
- O **Prototype** não é baseado em heranças, então ele não tem os inconvenientes dela. Por outro lado, o *Prototype* precisa de uma inicialização complicada do objeto clonado. O **Factory Method** é baseado em herança mas não precisa de uma etapa de inicialização.
- O **Factory Method** é uma especialização do **Template Method**. Ao mesmo tempo, o *Factory Method* pode servir como uma etapa em um *Template Method* grande.

345 páginas

do livro completo são omitidas na versão demonstrativa