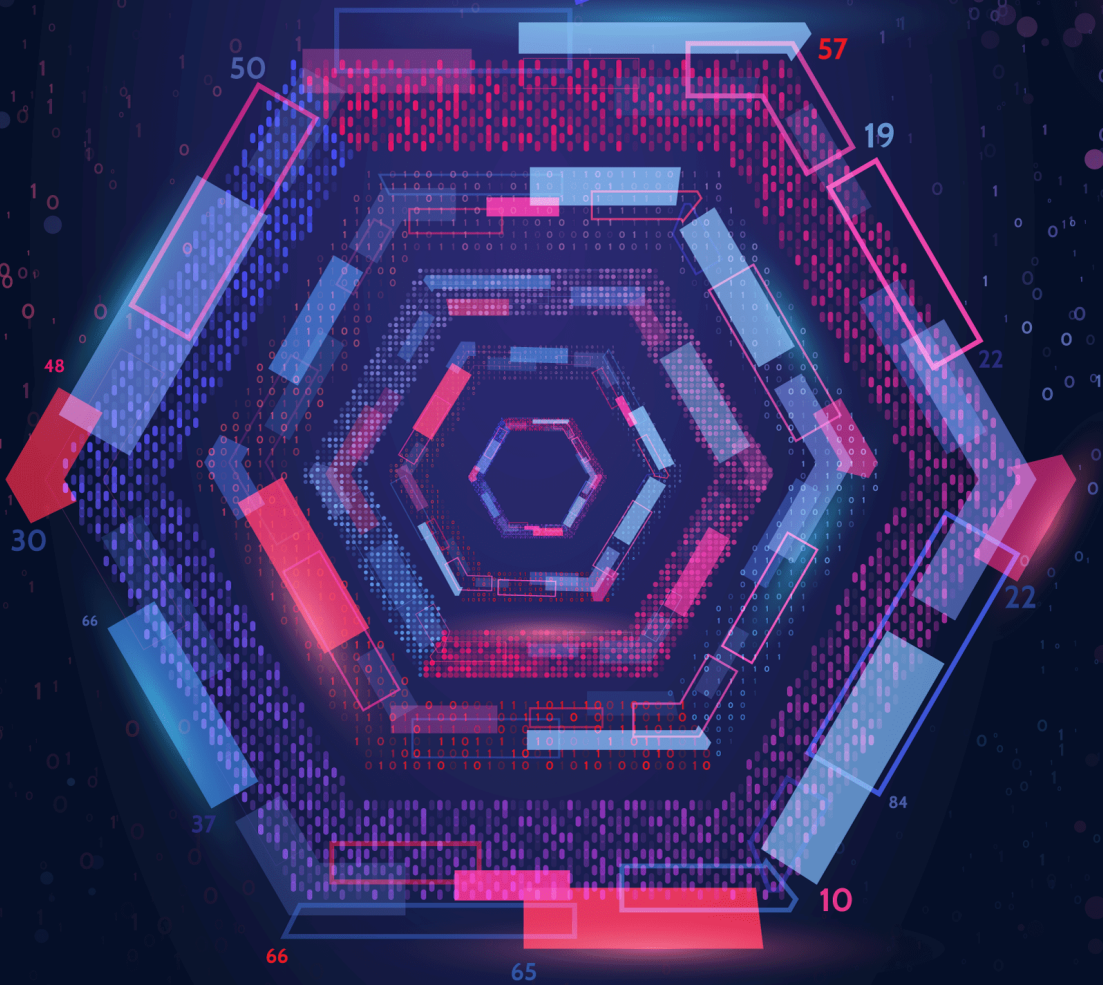
Mergulho nos PADRÕES DE PROJETO



Alexander Shvets

Mergulho nos PADRÕES DE PROJETO

v2021-1.15

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.

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

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**.

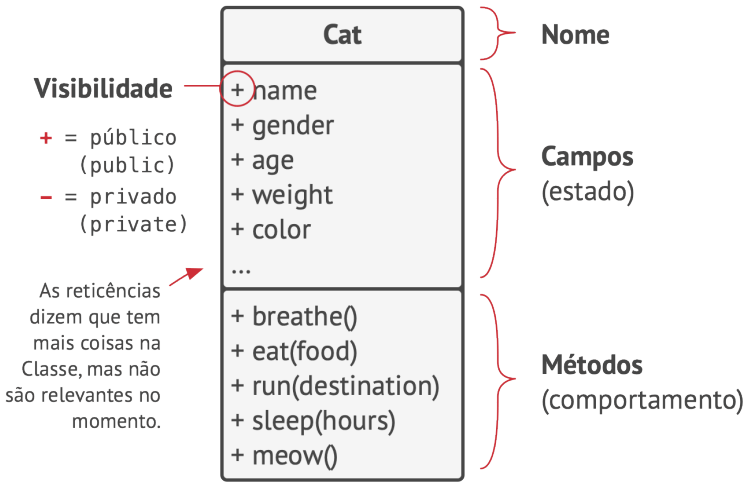
INTRODUÇÃO À PROGRAMAÇÃO ORIENTADA A OBJETOS

Básico da POO

A ***P**rogramação **O**rienteda à **O**bjetos* (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.



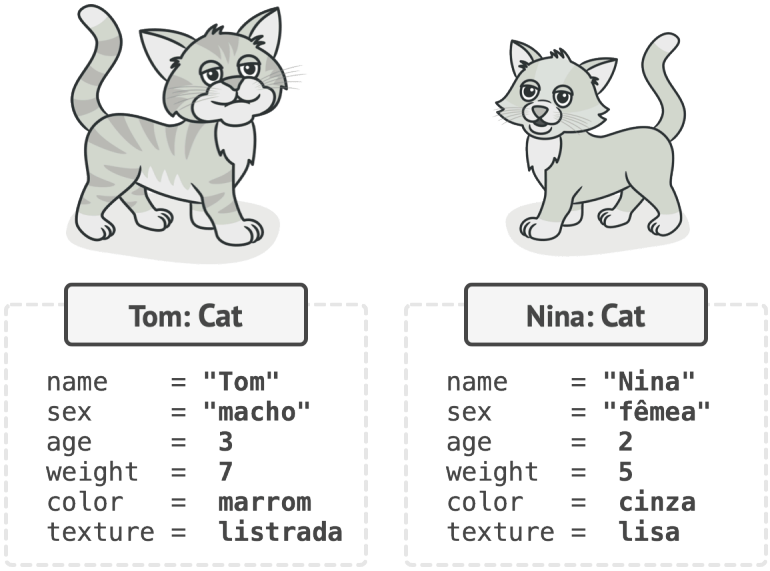
Este é um diagrama UML da classe. UML é a sigla do inglês Unified Modeling Language e significa Linguagem de Modelagem Unificada. Você verá muitos desses diagramas no livro.

> 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 sãos 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*.



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`.

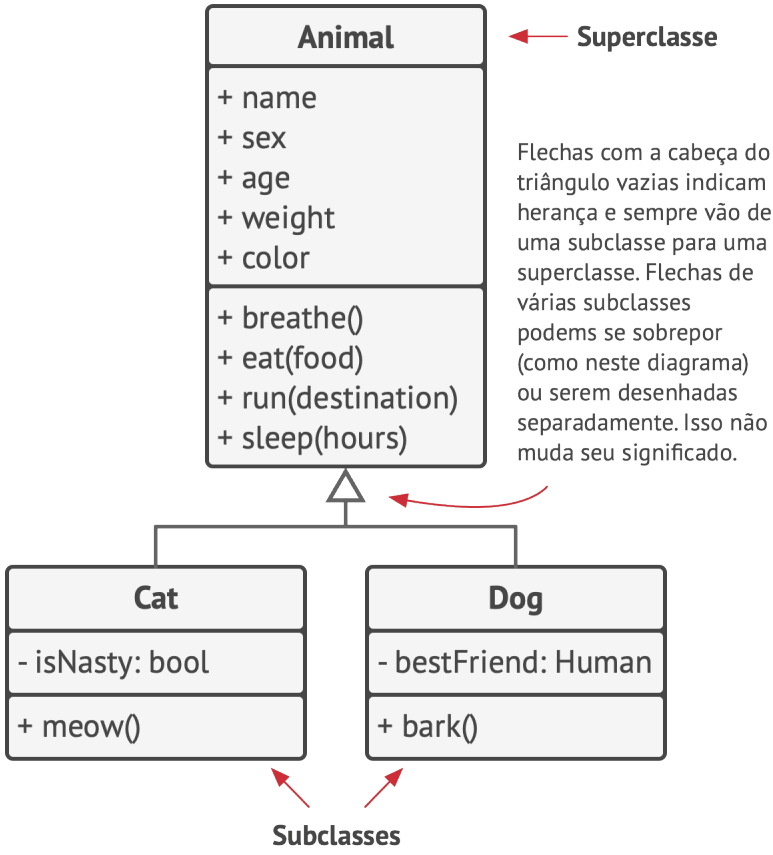
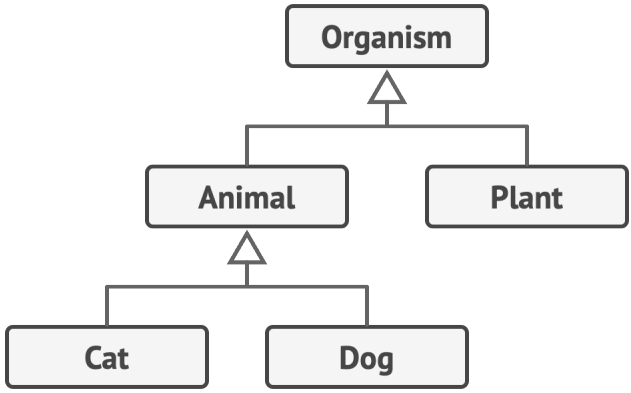


Diagrama UML de uma hierarquia de classe. Todas as classes neste diagrama são parte da hierarquia de classe `Animal`.

Assumindo que temos um requesito 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.

Pilares da POO

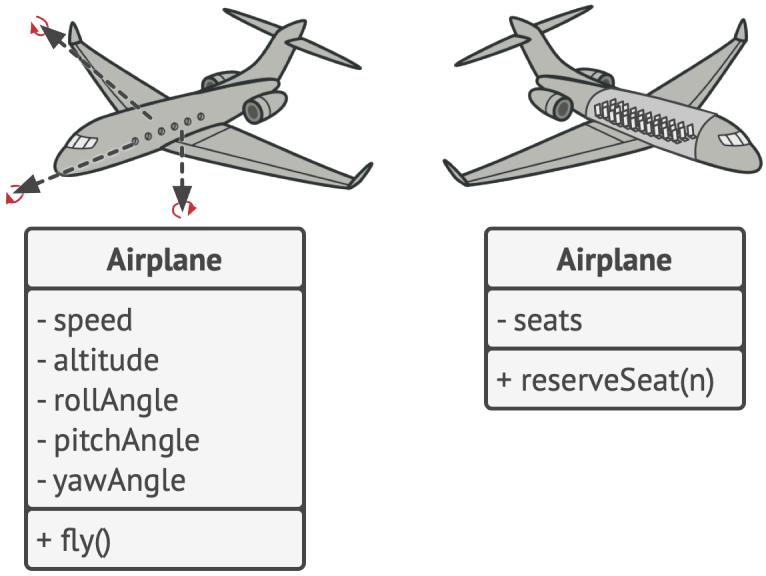
A programação orientada por objetos é baseada em quatro pilares, conceitos que diferenciam ele de outros paradigmas de programação.

Abstração

Na maioria das vezes quando você está criando um programa com a POO, você molda os objetos do programa baseado em objetos do mundo real. Contudo, objetos do programa não representam os originais com 100% de precisão (e raramente é necessário que eles cumpram isso). Ao invés disso, seus objetos apenas *modelam* atributos e comportamentos de objetos reais em um contexto específico, ignorando o resto.

Por exemplo, uma classe `Avião` poderia provavelmente existir em um simulador de voo e em uma aplicação de compra de passagens aéreas. Mas no primeiro caso, ele guardaria de-

talhes relacionados ao próprio voo, enquanto que no segundo caso você se importaria apenas com as poltronas disponíveis e os locais delas.



Diferentes modelos de um mesmo objeto real.

A *Abstração* é um modelo de um objeto ou fenômeno do mundo real, limitado a um contexto específico, que representa todos os detalhes relevantes para este contexto com grande precisão e omite o resto.

Encapsulamento

Para ligar um motor de carro, você precisa apenas girar a chave ou apertar um botão. Você não precisa conectar os fios debaixo do capô, rotacionar o eixo das manivelas e cilindros, e iniciar o

ciclo de força do motor. Estes detalhes estão embaixo do capô do carro. Você tem apenas uma interface simples: um interruptor de ignição, um volante, e alguns pedais. Isso ilustra como cada objeto tem um **interface**—uma parte pública do objeto, aberto a interações com outros objetos.

O *Encapsulamento* é a habilidade de um objeto de esconder parte de seu estado e comportamentos de outros objetos, expondo apenas uma interface limitada para o resto do programa.

Encapsular alguma coisa significa torná-la `privada`, e portanto acessível apenas por dentro dos métodos da sua própria classe. Há um modo um pouco menos restritivo chamado `protegido` que torna um membro da classe disponível para subclasses também.

Interfaces e classes/métodos abstratos da maioria das linguagens de programação são baseados em conceitos de abstração e encapsulamento. Em linguagens de programação modernas orientadas a objetos, o mecanismo de interface (geralmente declarado com a palavra chave `interface` ou `protocol`) permite que você defina contratos de interação entre objetos. Esse é um dos motivos pelos quais as interfaces somente se importam com os comportamentos de objetos, e porque você não pode declarar um campo em uma interface.

> O fato da palavra *interface* aparecer como a parte pública de um objeto enquanto que também temos o tipo `interface` na maioria das linguagens de programação pode ser bem confuso, estou com você nessa.

Imagine que você tenha uma interface `TransporteAéreo` com um método `voar(origem, destino, passageiros)`.

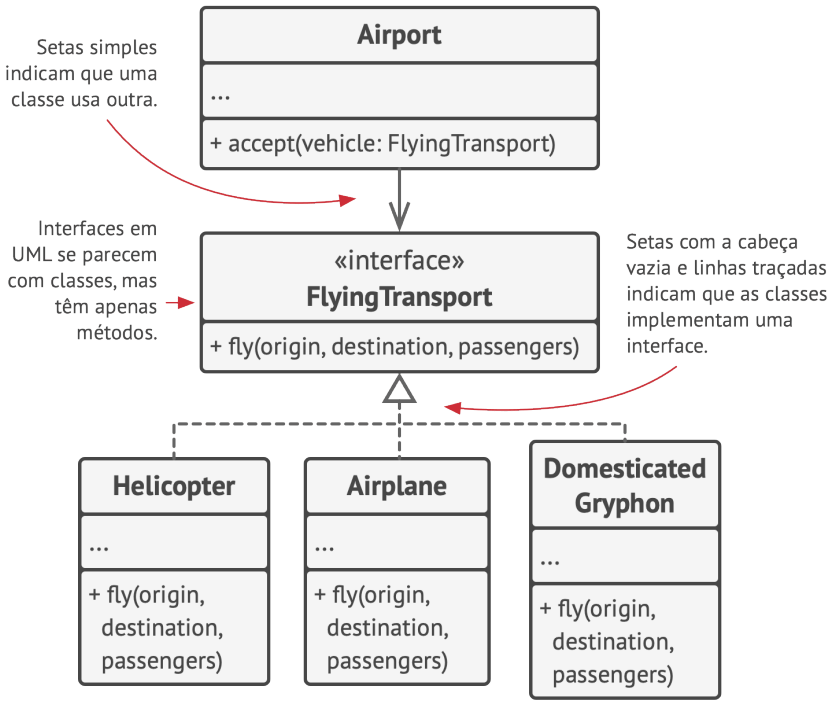


Diagrama UML de diversas clases implementando uma interface.

Quando desenvolvendo um simulador de transporte aéreo você restringiu a classe `Aeroporto` para trabalhar apenas com

objetos que implementam a interface `TransporteAéreo`. Após isso, você pode ter certeza que qualquer objeto passado para um objeto aeroporto, seja ela um `Avião`, um `Helicóptero`, ou um inesperado `GrifoDomesticado`, todos serão capazes de aterrissar ou decolar deste tipo de aeroporto.

Você pode mudar a implementação do método `voar` nessas classes de qualquer maneira que deseje. Desde que a assinatura do método permaneça a mesma que a declarada na interface, todas as instâncias da classe `Aeroporto` podem trabalhar com seus objetos voadores sem problemas.

Herança

A *Herança* é a habilidade de construir novas classes em cima de classes já existentes. O maior benefício da herança é a reutilização de código. Se você quer criar uma classe que é apenas um pouco diferente de uma já existente, não há necessidade de duplicar o código. Ao invés disso, você estende a classe existente e coloca a funcionalidade adicional dentro de uma subclasse resultante, que herdará todos os campos de métodos da superclasse.

A consequência de usar a herança é que as subclasses têm a mesma interface que sua classe mãe. Você não pode esconder um método em uma subclasse se ele foi declarado na superclasse. Você deve também implementar todos os métodos abstratos, mesmo que eles não façam sentido em sua subclasse.

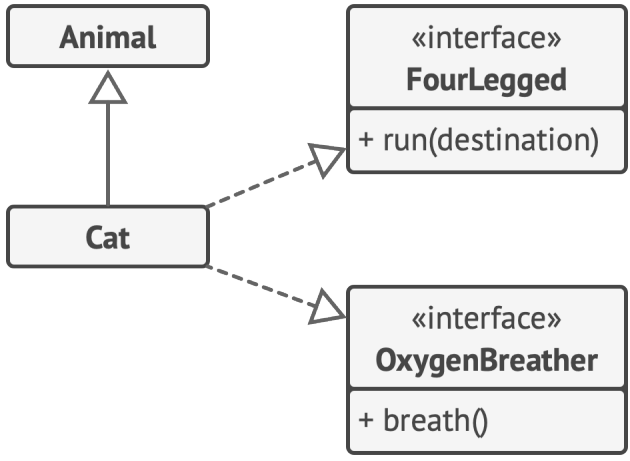


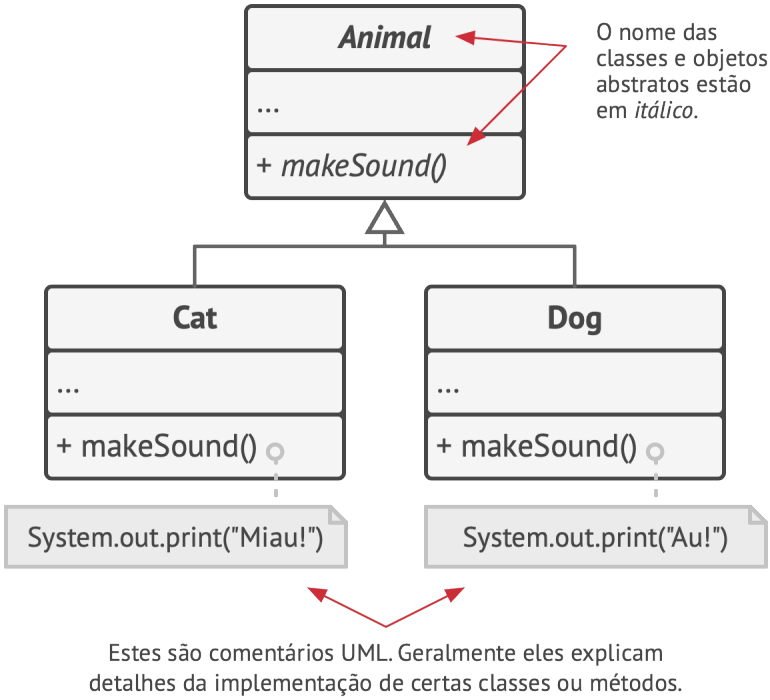
Diagrama UML de estender uma classe única versus implementar múltiplas interfaces ao mesmo tempo.

Na maioria das linguagens de programação uma subclasse pode estender apenas uma superclasse. Por outro lado, qualquer classe pode implementar várias interfaces ao mesmo tempo. Mas como mencionado anteriormente, se uma superclasse implementa uma interface, todas as suas subclasses também devem implementá-la.

Polimorfismo

Vamos ver alguns exemplos de animais. A maioria dos `Animais` podem produzir sons. Nós podemos antecipar que todas as subclasses terão que sobrescrever o método base `produzirSom` para que cada subclasse possa emitir o som correto; portanto nós podemos declará-lo *abstrato* agora mesmo. Isso permite omitir qualquer implementação padrão do mé-

todo na superclasse, mas força todas as subclasses a se virarem com o que têm.



Imagine que colocamos vários gatos e cães em uma bolsa grande. Então, com os olhos fechados, nós tiramos os animais um a um para fora da bolsa. Após tirarmos um animal da bolsa, nós não sabemos com certeza o que ele é. Contudo, se fizermos carícias no animal o suficiente, ele vai emitir um som de alegria específico, dependendo de sua classe concreta.

```
bolsa = [new Gato(), new Cão()];

foreach (Animal a : bolsa)
        a.produzirSom()

// Miau!
// Au!
```

O programa não sabe o tipo concreto do objeto contido dentro da variável `a`; mas, graças ao mecanismo especial chamado *polimorfismo*, o programa pode rastrear a subclasse do objeto cujo método está sendo executado e executar o comportamento apropriado.

O *Polimorfismo* é a habilidade de um programa detectar a classe real de um objeto e chamar sua implementação mesmo quando seu tipo real é desconhecido no contexto atual.

Você também pode pensar no polimorfismo como a habilidade de um objeto "fingir" que é outra coisa, geralmente uma classe que ele estende ou uma interface que ele implementa. No nosso exemplo, os cães e gatos na bolsa estavam fingindo ser animais genéricos.

Relações entre objetos

Além da *herança* e *implementação* que já vimos, há outros tipos de relações entre objetos que ainda não falamos.

Dependência

UML da Dependência. O professor depende dos materiais do curso.

A *dependência* é o mais básico e o mais fraco tipo de relações entre classes. Existe uma dependência entre duas classes se algumas mudanças na definição de uma das classes pode resultar em modificações em outra classe. A dependência tipicamente ocorre quando você usa nomes de classes concretas em seu código. Por exemplo, quando especificando tipos em assinaturas de métodos, quando instanciando objetos através de chamadas do construtor, etc. Você pode tornar a dependência mais fraca se você fazer seu código ser dependente de interfaces ou classes abstratas ao invés de classes concretas.

Geralmente, um diagrama UML não mostra todas as dependências—há muitas delas em um código de verdade. Ao invés de poluir o diagrama com dependências, você pode ser seletivo e mostrar apenas aquelas que são importantes para o que quer que seja que você está comunicando.

Associação

UML da Associação. O professor se comunica com os alunos.

A *associação* é um relacionamento no qual um objeto usa ou interage com outro. Em diagramas UML, o relacionamento de associação é mostrado por uma seta simples desenhada de um objeto e apontada para outro que ele utiliza. A propósito, ter uma associação bi-direcional é uma coisa completamente normal. Neste caso, a flecha precisa apontar para ambos. A associação pode ser vista como um tipo especializado de dependência, onde um objeto sempre tem acesso aos objetos os quais ele interage, enquanto que a dependência simples não estabelece uma ligação permanente entre os objetos.

Em geral, você usa uma associação para representar algo como um campo em uma classe. A ligação está sempre ali, você sempre pode fazer um pedido para o cliente dela. Mas nem sempre precisa ser um campo. Se você está modelando suas classes de uma perspectiva de interface, ele pode apenas indicar a presença de um método que irá retornar o pedido do cliente.

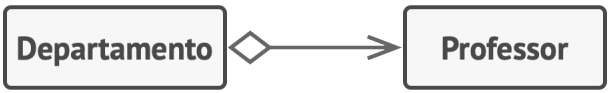
Para solidificar seu entendimento da diferença entre associação e dependência, vamos ver o exemplo combinado. Imagine que você tem uma classe `Professor`

```
class Professor is
  field Student student
  // ...
  method teach(Course c) is
    // ...
    this.student.remember(c.getKnowledge())
```

Observe o método `ensinar`. Ele precisa de um argumento da classe `Curso`, que então é usado no corpo do método. Se alguém muda o método `obterConhecimento` da classe `Curso` (altera seu nome, ou adiciona alguns parâmetros necessários, etc) nosso código irá quebrar. Isso é chamado de dependência.

Agora olha para o campo `aluno` e como ele é usado no método `ensinar`. Nós podemos dizer com certeza que a classe `Aluno` é também uma dependência para a classe `Professor`: se o método `lembrar` mudar, o código da `Professor` irá quebrar. Contudo, uma vez que o campo `aluno` está sempre acessível para qualquer método do `Professor`, a classe `Aluno` não é apenas uma dependência, mas também uma associação.

Agregação



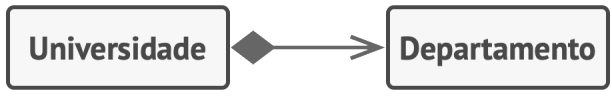
UML da Agregação. O departamento contém professores.

A *agregação* é um tipo especializado de associação que representa relações individuais (one-to-many), múltiplas (many-to-many), e totais (whole-part) entre múltiplos objetos, enquanto que uma associação simples descreve relações entre pares de objetos.

Geralmente, sob agregação, um objeto "tem" um conjunto de outros objetos e serve como um contêiner ou coleção. O componente pode existir sem o contêiner e pode ser ligado através de vários contêineres ao mesmo tempo. No UML a relação de agregação é mostrada como uma linha e um diamante vazio na ponta do contêiner e uma flecha apontando para o componente.

Embora estejamos falando sobre relações entre objetos, lembre-se que o UML representa relações entre *classes*. Isso significa que um objeto universidade pode consistir de múltiplos departamentos mesmo que você veja apenas um "bloco" para cada entidade no diagrama. A notação do UML pode representar quantidades em ambos os lados da relação, mas tudo bem omití-las se as quantidades não participam do contexto.

Composição



UML da Composição. A universidade consiste de departamentos.

A *composição* é um tipo específico de agregação, onde um objeto é composto de um ou mais instâncias de outro. A distinção entre esta relação e as outras é que o componente só pode existir como parte de um contêiner. No UML a relação de composição é desenhada do mesmo modo que para a agregação, mas com um diamante preenchido na base da flecha.

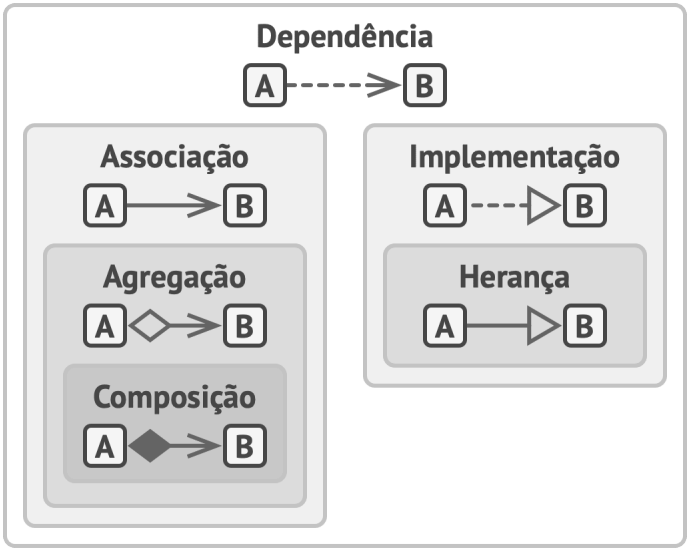
Observe que, na língua Inglesa, muitas pessoas usam com frequência o termo "composition" (composição) quando realmente querem dizer tanto a agregação como a composição. O exemplo mais notório para isso é o famoso princípio "choose composition over inheritance" (escolha composição sobre herança). Não é porque as pessoas desconhecem a diferença, mas porque a palavra "composition" (por exemplo em "object composition") soa mais natural para elas.

Panorama geral

Agora que conhecemos todos os tipos de relações entre os objetos, vamos ver como eles se conectam. Espero que isso o guie em dúvidas como "qual a diferença entre agregação e composição" ou "a herança é um tipo de dependência?"

- **Dependência:** Classe A pode ser afetada por mudanças na classe B.
- **Associação:** Objeto A sabe sobre objeto B. Classe A depende de B.
- **Agregação:** Objeto A sabe sobre objeto B, e consiste de B. Classe A depende de B.

- **Composição:** Objeto A sabe sobre objeto B, consiste de B, e gerencia o ciclo de vida de B. Classe A depende de B.
- **Implementação:** Classe A define métodos declarados na interface B. objetos de A podem ser tratados como B. Classe A depende de B.
- **Herança:** Classe A herda a interface e implementação da classe B mas pode estendê-la. Objets de A podem ser tratados como B. Classe A depende de B.



Relações entre objetos e classes: da mais fraca a mais forte.

INTRODUÇÃO AOS PADRÕES

O que é um padrão de projeto?

Padrões de projeto são soluções típicas para problemas comuns em projeto de software. Eles são como plantas de obra pré fabricadas que você pode customizar para resolver um problema de projeto recorrente em seu código.

Você não pode apenas encontrar um padrão e copiá-lo para dentro do seu programa, como você faz com funções e bibliotecas que encontra por aí. O padrão não é um pedaço de código específico, mas um conceito geral para resolver um problema em particular. Você pode seguir os detalhes do padrão e implementar uma solução que se adeque às realidades do seu próprio programa.

Os padrões são frequentemente confundidos com algoritmos, porque ambos os conceitos descrevem soluções típicas para alguns problemas conhecidos. Enquanto um algoritmo sempre define um conjunto claro de ações para atingir uma meta, um padrão é mais uma descrição de alto nível de uma solução. O código do mesmo padrão aplicado para dois programas distintos pode ser bem diferente.

Uma analogia a um algoritmo é que ele seria uma receita de comida: ambos têm etapas claras para chegar a um objetivo. Por outro lado, um padrão é mais como uma planta de obras:

você pode ver o resultado e suas funcionalidades, mas a ordem exata de implementação depende de você.

Do que consiste um padrão?

A maioria dos padrões são descritos formalmente para que as pessoas possam reproduzi-los em diferentes contextos. Aqui estão algumas seções que são geralmente presentes em uma descrição de um padrão:

- O **Propósito** do padrão descreve brevemente o problema e a solução.
- A **Motivação** explica a fundo o problema e a solução que o padrão torna possível.
- As **Estruturas** de classes mostram cada parte do padrão e como se relacionam.
- **Exemplos de código** em uma das linguagens de programação populares tornam mais fácil compreender a ideia por trás do padrão.

Alguns catálogos de padrão listam outros detalhes úteis, tais como a aplicabilidade do padrão, etapas de implementação, e relações com outros padrões.

Classificação dos padrões

Padrões de projeto diferem por sua complexidade, nível de detalhes, e escala de aplicabilidade ao sistema inteiro sendo de-

senvolvido. Eu gosto da analogia com a construção de uma rodovia: você sempre pode fazer uma intersecção mais segura instalando algumas sinaleiras ou construindo intercomunicações de vários níveis com passagens subterrâneas para pedestres.

Os padrões mais básicos e de baixo nível são comumente chamados *idiomáticos*. Eles geralmente se aplicam apenas à uma única linguagem de programação.

Os padrões mais universais e de alto nível são os *padrões arquitetônicos*; desenvolvedores podem implementar esses padrões em praticamente qualquer linguagem. Ao contrário de outros padrões, eles podem ser usados para fazer o projeto da arquitetura de toda uma aplicação.

Além disso, todos os padrões podem ser categorizados por seu *propósito*, ou intenção. Esse livro trata de três grupos principais de padrões:

- Os **padrões criacionais** fornecem mecanismos de criação de objetos que aumentam a flexibilidade e a reutilização de código.

- Os **padrões estruturais** explicam como montar objetos e classes em estruturas maiores, enquanto ainda mantém as estruturas flexíveis e eficientes.

- Os **padrões comportamentais** cuidam da comunicação eficiente e da assinalação de responsabilidades entre objetos.

História dos padrões

Quem inventou os padrões de projeto? Essa é uma boa pergunta, mas não muito precisa. Os padrões de projeto não são conceitos obscuros e sofisticados—bem o contrário. Os padrões são soluções típicas para problemas comuns em projetos orientados a objetos. Quando uma solução é repetida de novo e de novo em vários projetos, alguém vai eventualmente colocar um nome para ela e descrever a solução em detalhe. É basicamente assim que um padrão é descoberto.

O conceito de padrões foi primeiramente descrito por Christopher Alexander em *Uma Linguagem de Padrões*[1]. O livro descreve uma "linguagem" para o projeto de um ambiente urbano. As unidades dessa linguagem são os padrões. Eles podem descrever quão alto as janelas devem estar, quantos andares um prédio deve ter, quão largas as áreas verdes de um bairro devem ser, e assim em diante.

A ideia foi seguida por quatro autores: Erich Gamma, John Vlissides, Ralph Johnson, e Richard Helm. Em 1994, eles publicaram *Padrões de Projeto — Soluções Reutilizáveis de Software Orientado a Objetos*[2], no qual eles aplicaram o conceito de padrões de projeto para programação. O livro mostrava 23 padrões que resolviam vários problemas de projeto orientado a

1. *Uma Linguagem de Padrões*: **https://refactoring.guru/pt-br/pattern-language-book**
2. *Padrões de Projeto — Soluções Reutilizáveis de Software Orientado a Objetos*: **https://refactoring.guru/pt-br/gof-book**

objetos e se tornou um best-seller rapidamente. Devido a seu longo título, as pessoas começaram a chamá-lo simplesmente de “o livro da Gangue dos Quatro (Gang of Four)” que logo foi simplificado para o “livro GoF”.

Desde então, dúzias de outros padrões orientados a objetos foram descobertos. A “abordagem por padrões” se tornou muito popular em outros campos de programação, então muitos desses padrões agora existem fora do projeto orientado a objetos também.

Por que devo aprender padrões?

A verdade é que você pode conseguir trabalhar como um programador por muitos anos sem saber sobre um único padrão. Muitas pessoas fazem exatamente isso. Ainda assim, contudo, você estará implementando alguns padrões mesmo sem saber. Então, por que gastar tempo para aprender sobre eles?

- Os padrões de projeto são um kit de ferramentas para **soluções tentadas e testadas** para problemas comuns em projeto de software. Mesmo que você nunca tenha encontrado esses problemas, saber sobre os padrões é ainda muito útil porque eles ensinam como resolver vários problemas usando princípios de projeto orientado a objetos.

- Os padrões de projeto definem uma linguagem comum que você e seus colegas podem usar para se comunicar mais eficientemente. Você pode dizer, "Oh, é só usar um Singleton para isso," e todo mundo vai entender a ideia por trás da sua sugestão. Não é preciso explicar o que um singleton é se você conhece o padrão e seu nome.

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 Gamma[1], 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 nivel 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 importanta *como* os impostos são calculados.

```
method getOrderTotal(order) is
  total = 0
  foreach item in order.lineItems
    total += item.price * item.quantity

  if (order.country == "US")
    // Imposto das vendas nos EUA.
    total += total * 0.07
  else if (order.country == "EU"):
    // Imposto sobre o valor acrescentado.
    total += total * 0.20

  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.

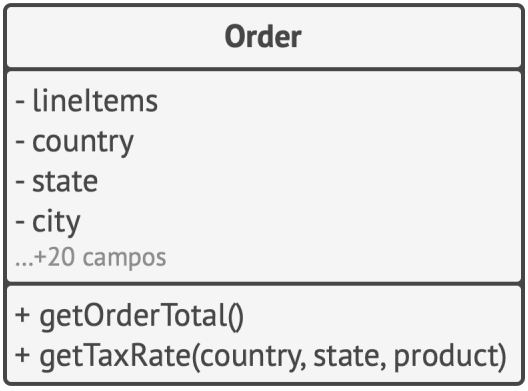
```
method getOrderTotal(order) is
  total = 0
  foreach item in order.lineItems
    total += item.price * item.quantity

  total += total * getTaxRate(order.country)

  return total

method getTaxRate(country) is
  if (country == "US")
    // Imposto das vendas nos EUA.
    return 0.07
  else if (country == "EU")
    // Imposto sobre o valor acrescentado.
    return 0.20
  else
    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.

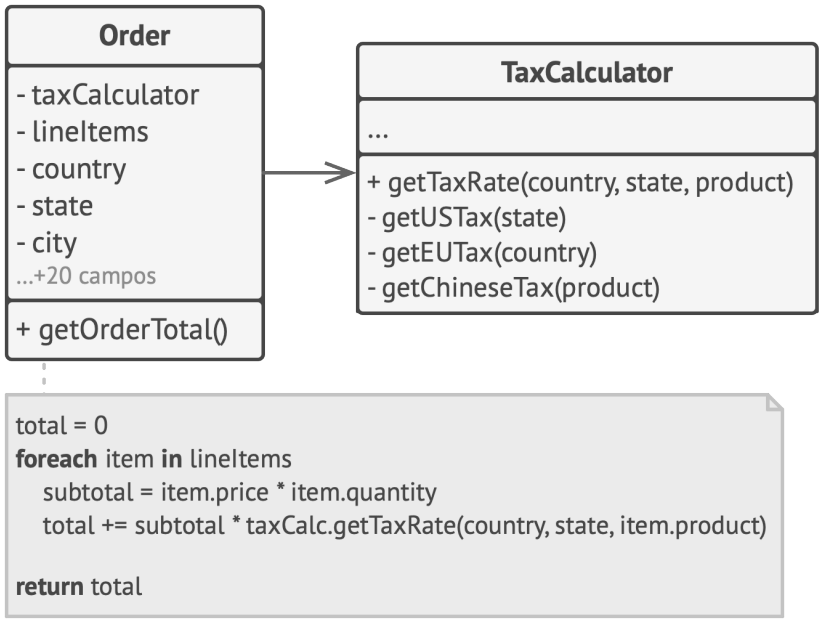
Encapsulamento a nível de classe

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. Extraindo 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`.

Programe para uma interface, não uma implementação

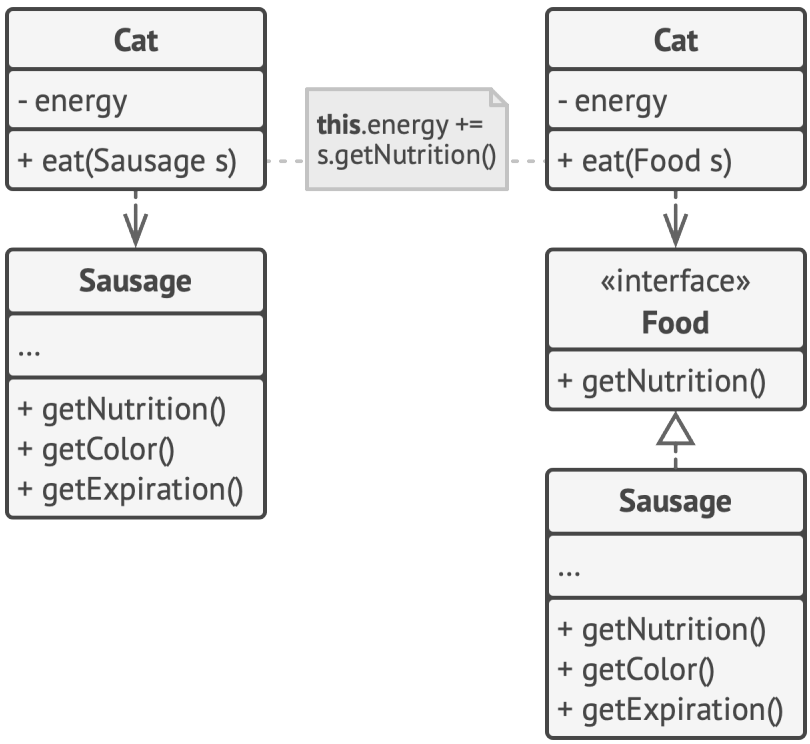
> Programe para uma interface, não uma implementação.
> Dependa de abstrações, não classes concretas.

Você pode perceber se o projeto é flexível o bastante se você pode estendê-lo facilmente sem quebrar o código existente. Vamos garantir que esta afirmação é correta ao olhar para mais um exemplo com gatos. Um `Gato` que pode comer qualquer comida é mais flexível que um gato que come apenas salsichas. Você ainda pode alimentar o primeiro gato com salsichas porque elas são um subconjunto de "qualquer comida"; contudo, você pode estender o cardápio do gato com qualquer outra comida.

Quando você quer fazer duas classes colaborarem, você pode começar fazendo uma delas ser dependente da outra. Caramba, eu sempre começo fazendo isso eu mesmo. Contudo tem um meio mais flexível de configurar uma colaboração entre objetos:

1. Determinar o que exatamente um objeto precisa do outro: quais métodos ele executa?
2. Descreva estes métodos em uma nova interface ou classe abstrata.

3. Faça a classe que é uma dependência implementar essa interface.
4. Agora faça a segunda classe ser dependente dessa interface ao invés de fazer isso na classe concreta. Você ainda pode fazê-la funcionar com objetos da classe original, mas a conexão é agora muito mais flexível.



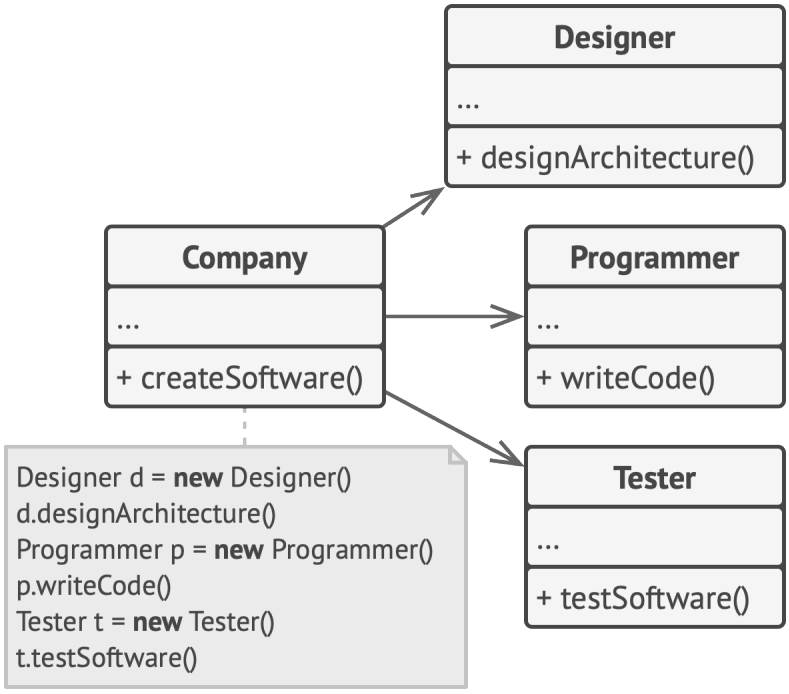
Antes e depois de extrair a interface. O código à direita é mais flexível que o código à esquerda, mas também é mais complicado.

Após fazer esta mudança, você provavelmente não sentirá qualquer benefício imediato. Pelo contrário, o código parece mais complicado que estava antes. Contudo, se você se você

sentir que aqui é um bom ponto de extensão para uma funcionalidade adicional, ou que outras pessoas que usam seu código podem querer estendê-lo aqui, então use isto.

Exemplo

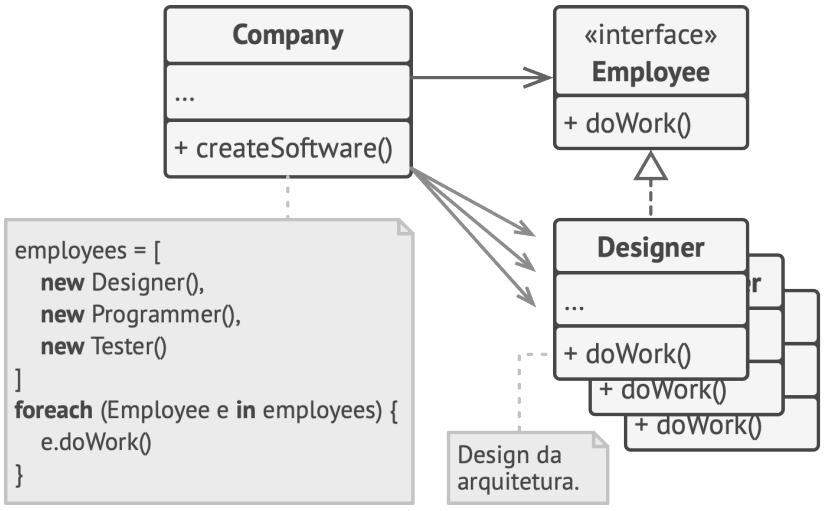
Vamos ver outro exemplo que ilustra que trabalhar com objetos através de interfaces pode ser mais benéfico que depender de suas classes concretas. Imagine que você está criando um simulador de empresa desenvolvedora de software. Você tem classes diferentes que representam vários tipos de funcionários.



ANTES: todas as classes eram firmemente acopladas.

No começo, a classe `Empresa` era firmemente acoplada as classes concretas dos empregados. Contudo, apesar da diferença em suas implementações, nós podemos generalizar vários métodos relacionados ao trabalho e então extrair uma interface comum para todas as classes de empregados.

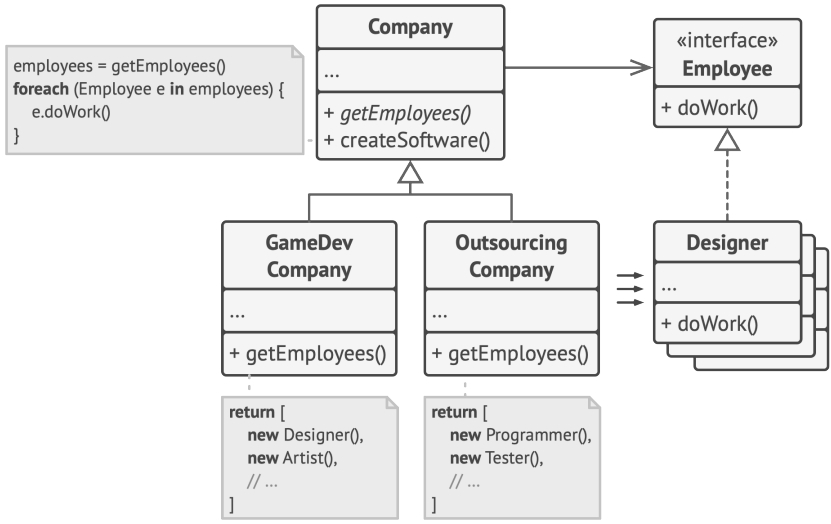
Após fazer isso, nós podemos aplicar o polimorfismo dentro da classe `Empresa`, tratando vários objetos de empregados através da interface `Empregado`.



MELHOR: o polimorfismo nos ajudou a simplificar o código, mas o resto da classe `Empresa` ainda depende das classes concretas dos empregados.

A classe `Empresa` permanece acoplada as classes dos empregados. Isso é ruim porque se nós introduzirmos novos tipos de empresas que funcionam com outros tipos de empregados, nós teríamos que sobrescrever a maior parte da classe `Empresa` ao invés de reutilizar aquele código.

Para resolver este problema, nós podemos declarar o método para obter os empregados como *abstratos*. Cada empresa concreta irá implementar este método de forma diferente, criando apenas aqueles empregados que precisam.



DEPOIS: o método primário da classe `Empresa` é independente de classes concretas de empregados. Os objetos empregados são criados em subclasses concretas de empresas.

Após essa mudança, a classe `Empresa` se tornou independente de várias classes de empregados. Agora você pode estender esta classe e introduzir novos tipos de empresas e empregados enquanto ainda reutiliza uma porção da classe empresa base. Estendendo a classe empresa base não irá quebrar o código existente que já se baseia nela. A propósito, você acabou de ver um padrão de projeto ser colocado em ação! Este foi um exemplo do padrão *Factory Method*. Não se preocupe: vamos discuti-lo mais tarde com detalhes.

Prefira composição sobre herança

A herança é provavelmente a maneira mais óbvia de reutilizar código entre classes. Você tem duas classes com o mesmo código. Crie uma classe base comum para ambas essas duas classes e mova o código similar para dentro dela. Barbadinha!

Infelizmente, a herança vem com um lado ruim que se torna aparente apenas quando seu programa já tem um monte de classes e mudar tudo fica muito difícil. Aqui temos uma lista desses problemas.

- **Uma subclasse não pode reduzir a interface da superclasse**. Você tem que implementar todos os métodos abstratos da classe mãe mesmo que você não os utilize.

- **Quando sobrescrevendo métodos você precisa se certificar que o novo comportamento é compatível com o comportamento base**. Isso é importante porque objetos da subclasses podem ser passados para qualquer código que espera objetos da superclasse e você não quer que aquele código quebre.

- **A herança quebra o encapsulamento da superclasse** porque os detalhes internos da classe mãe se tornam disponíveis para a subclasse. Pode ocorrer uma situação oposta onde um programador faz a superclasse ficar ciente de alguns detalhe das subclasses para deixar qualquer futura extensão mais fácil.

- **As subclasses estão firmemente acopladas as superclasses**. Quaisquer mudanças em uma superclasse podem quebrar a funcionalidade das subclasses.

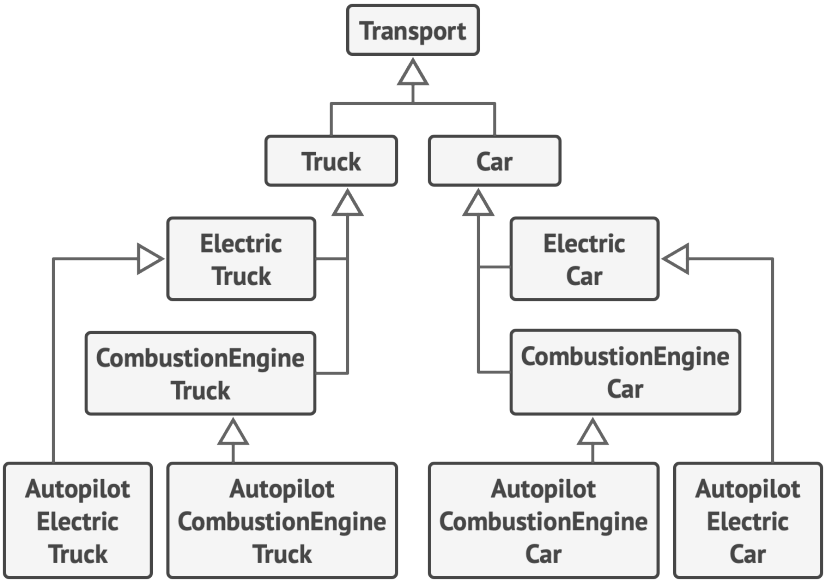
- **Tentar reutilizar código através da herança pode levar a criação de hierarquias de heranças paralelas**. A herança geralmente acontece em apenas uma dimensão. Mas sempre que há duas ou mais dimensões, você tem que criar várias combinações de classe, inchando a hierarquia de classe para um tamanho ridículo.

  Há uma alternativa para a herança chamada *composição*. Enquanto a herança representa uma relação "é um(a)" entre classes (um carro *é um* transporte), a composição representa a relação "tem um(a)" (um carro *tem um* motor).

  Eu deveria mencionar que este princípio também se aplica à agregação—uma versão mais relaxada da composição onde um objeto pode ter uma referência para um outro, mas não gerencia seu ciclo de vida. Aqui temos um exemplo: um carro *tem um* motorista, mas ele ou ela podem usar outro carro ou apenas caminhar *sem o carro*.

Exemplo

Imagine que você precisa criar uma aplicação catálogo para um fabricante de carros. A empresa fabrica tanto carros como caminhões; eles podem ser elétricos ou a gasolina; todos os modelos tem controles manuais ou autopiloto.

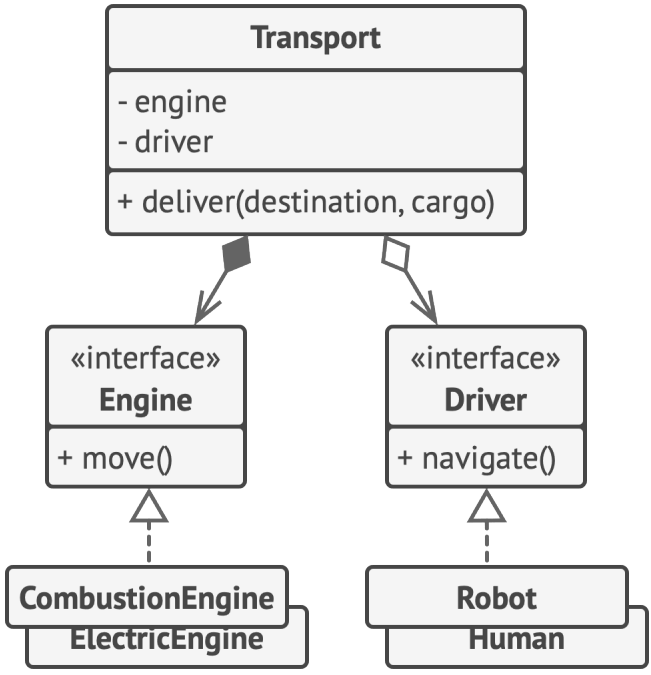


HERANÇA: estendendo uma classe em várias dimensões (tipo de carga × tipo de motor × tipo de navegação) pode levar a uma explosão combinada de subclasses.

Como você pode ver, cada parâmetro adicional resulta em uma multiplicação do número de subclasses. Tem muita duplicação de código entre as subclasses porque a subclasses não pode estender duas classes ao mesmo tempo.

Você pode resolver este problema com a composição. Ao invés de objetos de carro implementarem um comportamento por conta própria, eles podem delegar isso para outros objetos.

O benefício acumulado é que você pode substituir um comportamento no tempo de execução. Por exemplo, você pode substituir um objeto motor ligado a um objeto carro apenas assinalando um objeto de motor diferente para o carro.



COMPOSIÇÃO: diferentes "dimensões" de funcionalidade extraídas para suas próprias hierarquias de classe.

Esta estrutura de classes lembra o padrão *Strategy*, que vamos tratar mais tarde neste livro.

Princípios SOLID

Agora que você conhece os princípios básicos de projeto, vamos dar uma olhada em cinco deles que são comumente conhecidos como os princípios SOLID. Robert Martin os introduziu no livro *Agile Software Development, Principles, Patterns, and Practices*[1].

O *SOLID* é uma sigla mnemônica em inglês para cinco princípios de projeto destinados a fazer dos projetos de software algo mais compreensivo, flexível, e sustentável.

Como tudo na vida, usar estes princípios sem cuidado pode causar mais males que bem. O custo de aplicar estes princípios na arquitetura de um programa pode ser torná-lo mais complicado que deveria ser. Eu duvido que haja um produto de software de sucesso na qual todos estes princípios foram aplicados ao mesmo tempo. Empenhar-se para seguir estes princípios é bom, mas sempre tente ser pragmático e não tome tudo que está escrito aqui como um dogma.

1. *Agile Software Development, Principles, Patterns, and Practices*: **https://refactoring.guru/pt-br/principles-book**

S Princípio de responsabilidade única
ingle Responsibility Principle

> Uma classe deve ter apenas uma razão para mudar.

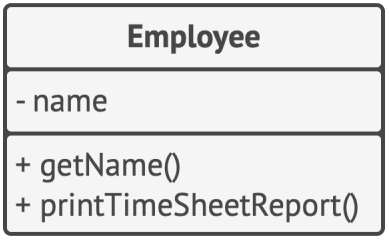
Tente fazer com que cada classe seja responsável por uma única parte da funcionalidade fornecida pelo software, e faça aquela responsabilidade ser inteiramente encapsulada pela (podemos também dizer *escondida dentro da*) classe.

O objetivo principal deste princípio é reduzir a complexidade. Você não precisa inventar um projeto sofisticado para um programa que tem apenas 200 linhas de código. Faça uma dúzia de métodos bonitos, e você ficará bem. Os verdadeiros problemas emergem quando um programa cresce e se modifica constantemente. Em algum ponto as classes se tornam tão grandes que você não lembra mais seus detalhes. A navegação pelo código se torna um rastejar, e você tem que escanear todas as classes ou um programa inteiro para encontrar coisas específicas. O número de entidades em um programa faz seu cérebro transbordar, e você sente que você está perdendo controle sobre o código.

Tem mais: se uma classe faz muitas coisas, você terá que mudá-la cada vez que uma dessas coisas muda. Enquanto faz isso, você está arriscando quebrar outras partes da classe que você nem pretendia mexer. Se em determinado momento você sente que está se tornando difícil focar em aspectos específicos de um programa, lembre-se do princípio da responsabilidade única e verifique se já não é hora de dividir algumas classes em partes.

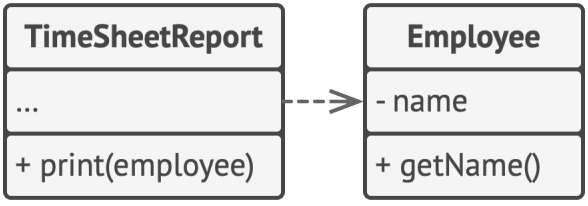
Exemplo

A classe `Empregado` tem vários motivos para mudar. O primeiro motivo pode ser relacionado a função principal da classe: gerenciar os dados dos empregados. Contudo, há outro motivo: o formato do relatório da tabela de tempos pode mudar com o tempo, fazendo com que você mude o código dentro da classe.



ANTES: a classe contém vários comportamentos diferentes.

Resolva o problema movendo o comportamento relacionado aos relatórios para uma classe em separado. Essa mudança permite que você mova outras coisas relacionadas ao relatório para a nova classe.



DEPOIS: o comportamento adicional está em sua própria classe.

O Princípio aberto/fechado
pen/Closed Principle

> As classes devem ser abertas para extensão mas fechadas para modificação.

A ideia principal deste princípio é prevenir que o código existente quebre quando você implementa novas funcionalidades.

Uma classe é *aberta* se você pode estendê-la, produzir uma subclasse e fazer o que quiser com ela—adicionar novos métodos ou campos, sobrescrever o comportamento base, etc. Algumas linguagens de programação permitem que você restrinja a futura extensão de uma classe com palavras chave como `final`. Após isso, a classe não é mais aberta. Ao mesmo tempo, a classe é *fechada* (você também pode chamá-la de *completa*) se ela estiver 100% pronta para ser usada por outras classes—sua interface está claramente definida e não será mudada no futuro.

Quando eu fiquei sabendo sobre este princípio pela primeira vez, fiquei confuso porque as palavras *aberta* e *fechada* parecem mutualmente exclusivas. Mas em se tratando deste princípio, uma classe pode ser tanto aberta (para extensão) e fechada (para modificação) ao mesmo tempo.

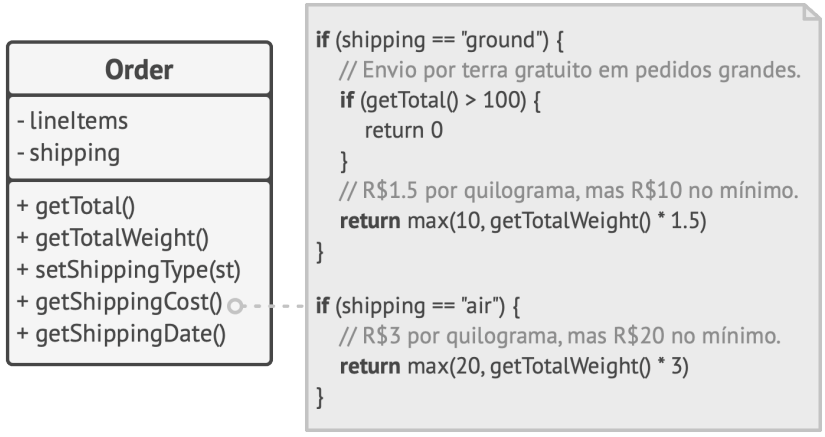
Se a classe já foi desenvolvida, testada, revisada, e incluída em algum framework ou é de alguma forma já usada na aplicação, tentar mexer com seu código é arriscado. Ao invés de mudar o código da classe diretamente, você pode criar subclasses e sobrescrever

partes da classe original que você quer que se comporte de forma diferente. Você vai cumprir seu objetivo mas também não quebrará os clientes existentes da classe original.

Este princípio não foi feito para ser aplicado para todas as mudanças de uma classe. Se você sabe que há um bug na classe, apenas vá e corrija-o; não crie uma subclasse para ele. Uma classe filha não deveria ser responsável pelos problemas da classe mãe.

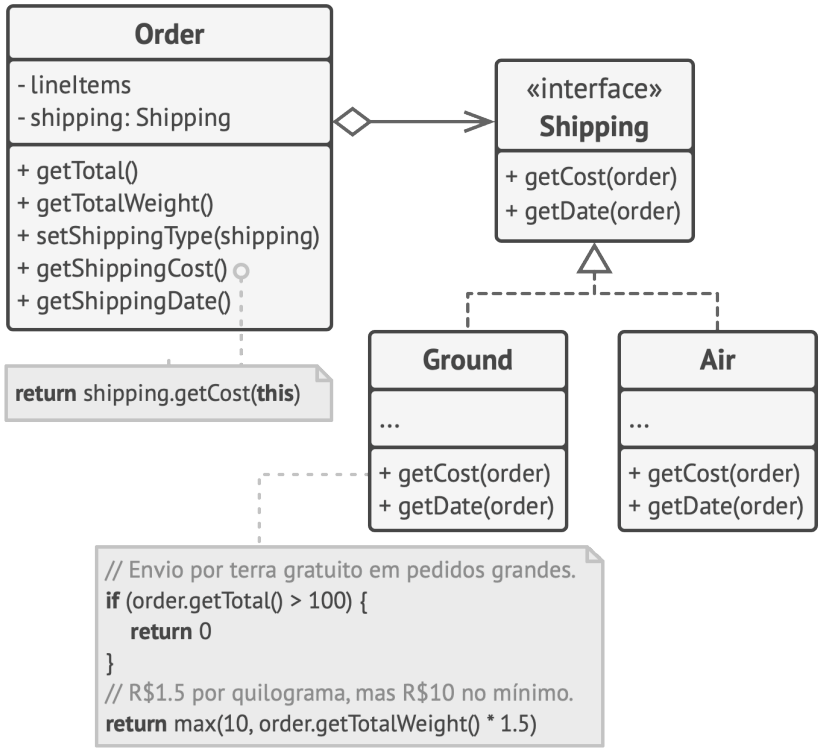
Exemplo

Você tem uma aplicação e-commerce com uma classe `Pedido` que calcula os custos de envio e todos os métodos de envio estão codificados dentro da classe. Se você precisa adicionar um novo método de envio, você terá que mudar o código da classe `Pedido`, arriscando quebrá-la.



ANTES: você tem que mudar a classe `Pedido` sempre que adicionar um novo método de envio na aplicação.

Você pode resolver o problema aplicando o padrão *Strategy*. Comece extraindo os métodos de envio para classes separadas com uma interface comum.



DEPOIS: adicionando um novo método de envio não necessita mudanças em classes existentes.

Agora quando você precisa implementar um novo método de envio, você pode derivar uma nova classe da interface `Envio` sem tocar em qualquer código da classe `Pedido`. O código cliente da classe `Pedido` irá ligar os pedidos com o objeto do envio com a nova classe sempre que o usuário selecionar estes métodos de envio na interface gráfica.

Como um bônus, essa solução permite que você mova o cálculo de tempo de envio para classes mais relevantes, de acordo com o *princípio de responsabilidade única*.

L
Princípio de substituição de Liskov[1]
iskov Substitution Principle

> Quando estendendo uma classe, lembre-se que você deve ser capaz de passar objetos da subclasse em lugar de objetos da classe mãe sem quebrar o código cliente.

Isso significa que a subclasse deve permanecer compatível com o comportamento da superclasse. Quando sobrescrevendo um método, estenda o comportamento base ao invés de substituí-lo com algo completamente diferente.

O princípio de substituição é um conjunto de checagens que ajudam a prever se um subclasse permanece compatível com o código que foi capaz de trabalhar com objetos da superclasse. Este conceito é crítico quando desenvolvendo bibliotecas e frameworks, porque suas classes serão usadas por outras pessoas cujo código você não pode diretamente acessar ou mudar.

Ao contrário de outros princípios de projeto que são amplamente abertos à interpretação, o princípio de substituição tem um conjunto de requerimentos formais para subclasses, e especificamente para seus métodos. Vamos ver esta lista em detalhes.

---

1. Este princípio foi nomeado por Barbara Liskov, que o definiu em 1987 em seu trabalho *Data abstraction and hierarchy*: https://refactoring.guru/liskov/dah

- **Os tipos de parâmetros em um método de uma subclasse devem *coincidir* ou serem *mais abstratos* que os tipos de parâmetros nos métodos da superclasse.** Soa confuso? Vamos ver um exemplo.

    - Digamos que temos uma classe com um método que deve alimentar gatos: `alimentar(Gato g)`. O código cliente sempre passa objetos gato nesse método.

    - **Bom:** Digamos que você criou uma subclasse que sobrescreveu o método para que ele possa alimentar qualquer animal (uma superclasse dos gatos): `alimentar(Animal g)`. Agora se você passar um objeto desta subclasse ao invés de um objeto da superclasse para o código cliente, tudo ainda vai funcionar bem. O método pode alimentar quaisquer animais, então ele ainda pode alimentar qualquer gato passado pelo cliente.

    - **Ruim:** Você criou outra subclasse e restringiu o método alimentar para aceitar apenas gatos de Bengala (uma subclasse dos gatos): `alimentar(GatoBengala c)`. O que vai acontecer com o código cliente se você ligá-lo com um objeto como este ao invés da classe original? Já que o método só pode alimentar uma raça de gatos, ele não vai servir para gatos genéricos mandados pelo cliente, quebrando toda a funcionalidade relacionada.

- **O tipo de retorno de um método de uma subclasse deve *coincidir* ou ser um *subtipo* do tipo de retorno no método da super-**

classe. Como pode ver, os requerimentos para o tipo de retorno são inversos aos requerimentos para tipos de parâmetros.

  - Digamos que você tem uma classe com um método `comprarGato(): Gato`. O código cliente espera receber qualquer gato como resultado da execução desse método.

  - **Bom:** Uma subclasse sobrescreve o método dessa forma: `comprarGato(): GatoBengala`. O cliente recebe um gato de Bengala, que ainda é um gato, então está tudo bem.

  - **Ruim:** Uma subclasse sobrescreve o método dessa forma: `comprarGato(): Animal`. Agora o código cliente quebra já que ele recebe um tipo desconhecido de animal genérico (um jacaré? um urso?) que não se adequa à estrutura desenhada para um gato.

  Outro anti-exemplo vem do mundo das linguagens de programação com tipagem dinâmica: o método base retorna uma string, mas o método sobrescrito retorna um número.

- **Um método em uma subclasse não deve lançar tipos de exceções que não são esperados que o método base lançaria.** Em outras palavras, tipos de exceções devem *coincidir* ou serem *subtipos* daquelas que o método base já é capaz de lançar. Esta regra vem do fato que os blocos `try–catch` no código cliente têm como alvo tipos de exceções específicas que o método base provavelmente lançará. Portanto, uma exceção inespe-

rada pode escapar atravessar as linhas de defesa do código cliente e quebrar toda a aplicação.

> Na maioria das linguagens de programação modernas, especialmente as de tipo estático (Java, C#, e outras), estas regras eram construídas dentro da linguagem. Você não será capaz de compilar um programa que viola estas regras.

- **Uma subclasse não deve fortalecer pré-condições.** Por exemplo, o método base tem um parâmetro com tipo `int`. Se uma subclasse sobrescreve este método e precisa que o valor de um argumento passado para o método deve ser positivo (lançando uma exceção se o valor é negativo), isso torna as pré-condições mais fortes. O código cliente, que funcionava bem até agora quando números negativos eram passados no método, agora quebra se ele começa a trabalhar com um objeto desta subclasse.

- **Uma subclasse não deveria enfraquecer pós-condições.** Digamos que você tem uma classe com um método que trabalha com uma base de dados. Um método da classe deve sempre fechar todas as conexões abertas com a base de dados quando elas retornarem um valor.

  Você criou uma subclasse e mudou ela para que as conexões de base de dados continuem abertas para que você possa reutilizá-las. Mas o cliente pode não saber nada sobre suas in-

tenções. Como ele espera que os métodos fechem todas as conexões, ele pode simplesmente terminar o programa logo após chamar o método, poluindo o sistema com conexões fantasmas com a base de dados.

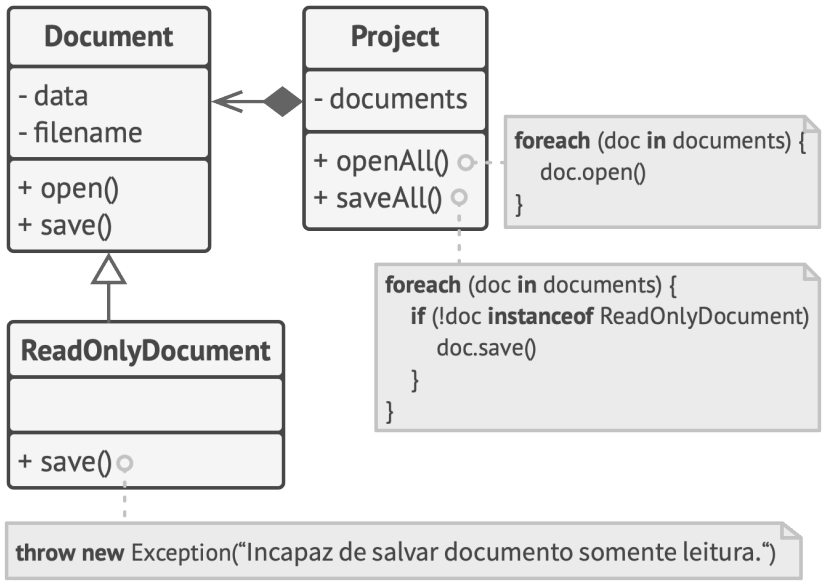
- **Invariantes de uma superclasse devem ser preservadas.** Esta é provavelmente a regra menos formal de todas. As *invariantes* são condições nas quais um objeto faz sentido. Por exemplo, invariantes de um gato são ter quatro patas, uma cauda, a habilidade de miar, etc. A parte confusa das invariantes é que, enquanto elas podem ser definidas explicitamente na forma de contratos de interface ou um conjunto de asserções com métodos, elas podem também serem implícitas por certos testes de unidade e expectativas do código cliente.

  A regra nas invariantes é a mais fácil de violar porque você pode não entender ou não perceber todas as invariantes de uma classe complexa. Portanto, a modo mais seguro de estender uma classe é introduzir novos campos e métodos, e não mexer com qualquer membro já existente da superclasse. É claro, isso nem sempre é viável no mundo real.

- **Uma subclasse não deve mudar valores de campos privados da superclasse.** *O que? E como isso é possível?* Acontece que algumas linguagens de programação permitem que você acesse membros privados de uma classe através de mecanismos reflexivos. Outras linguagens (Python, JavaScript) não têm quaisquer proteção para seus membros privados de forma alguma.

Exemplo

Vamos ver um exemplo de hierarquia de classes de documento que viola o princípio de substituição.

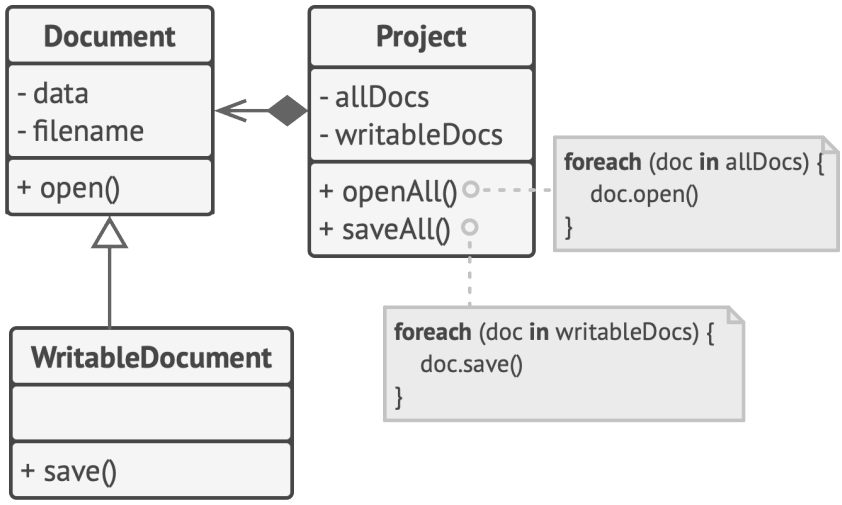


ANTES: salvar não faz sentido em um documento somente leitura, então a subclasse tenta resolver isso ao resetar o comportamento base no método sobrescrito.

O método `salvar` na subclasse `DocSomenteLeitura` lança uma exceção se alguém tenta chamá-lo. O método base não tem essa restrição. Isso significa que o código cliente pode quebrar se nós não checarmos o tipo de documento antes de salvá-lo.

O código resultante também viola o princípio aberto/fechado, já que o código cliente se torna dependente das classes con-

cretas de documentos. Se você introduzir uma nova subclasse de documento, você vai precisar mudar o código cliente para suportá-la.



DEPOIS: o problema foi resolvido após tornar a classe documento somente leitura como a classe base da hierarquia.

Você pode resolver o problema redesenhando a hierarquia de classe: uma subclasse deve estender o comportamento de uma superclasse, portanto, o documento somente leitura se torna a classe base da hierarquia. O documento gravável é agora uma subclasse que estende a classe base e adiciona o comportamento de salvar.

I Princípio de segregação de interface
nterface Segregation Principle

> Clientes não devem ser forçados a depender de métodos que não usam.

Tente fazer com que suas interfaces sejam reduzidas o suficiente para que as classes cliente não tenham que implementar comportamentos que não precisam.

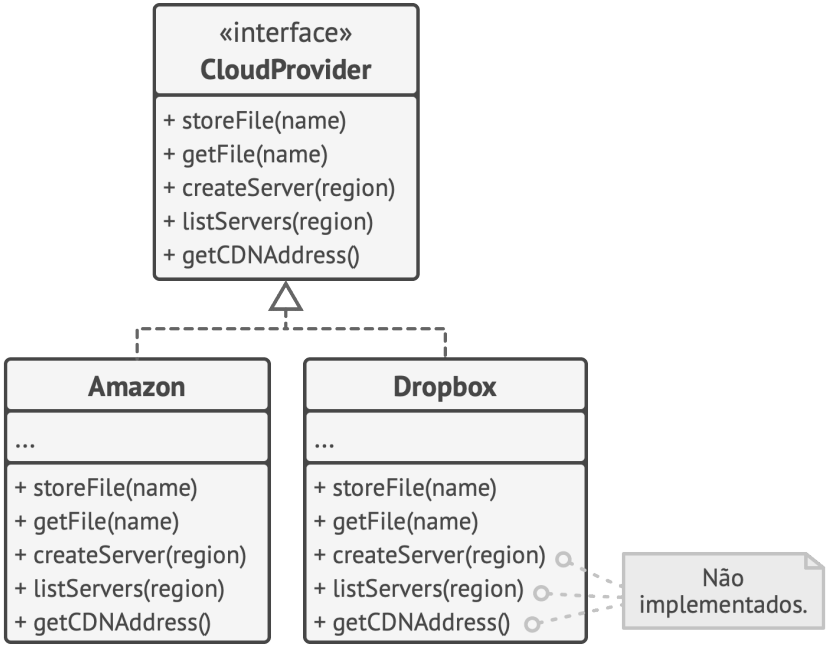
De acordo com o princípio de segregação de interface, você deve quebrar interfaces "gordas" em classes mais granulares e específicas. Os clientes devem implementar somente aqueles métodos que realmente precisam. Do contrário, uma mudança em uma interface "gorda" irá quebrar clientes que nem sequer usam os métodos modificados.

A herança de classe permite que uma classe tenha apenas uma superclasse, mas ela não limita o número de interfaces que aquela classe pode implementar ao mesmo tempo. Portanto não há necessidade de empilhar toneladas de métodos sem relação nenhuma em uma única interface. Quebre-as em algumas interfaces refinadas—você pode implementar todas em uma única classe se necessário. Contudo, algumas classes podem ficar bem implementando apenas uma delas.

Exemplo

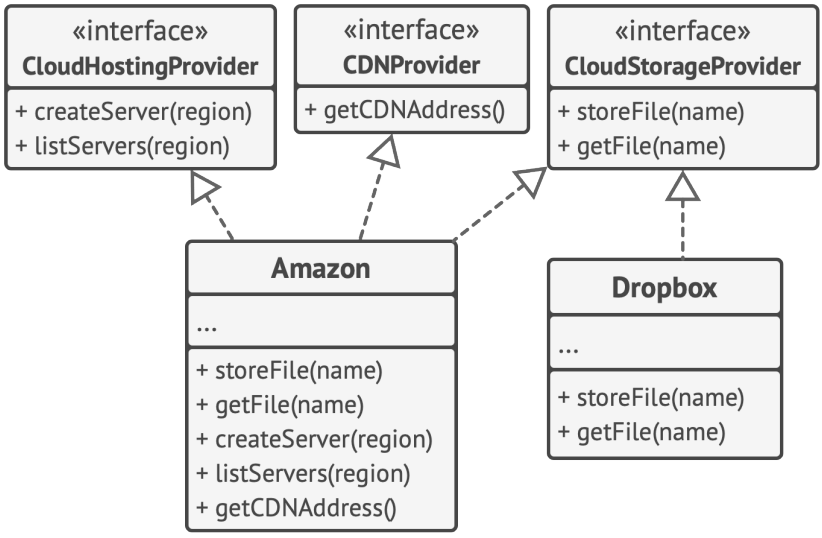
Imagine que você criou uma biblioteca que torna mais fácil integrar aplicações com vários fornecedores de computação da nuvem. Embora a versão inicial suportava apenas a Amazon Cloud, ela continha o conjunto completo de serviços e funcionalidades de nuvem.

Naquele momento você assumiu que todos os fornecedores de nuvem teriam a mesma gama de funcionalidade que a Amazon. Mas quando chegou a hora de implementar o suporte a outro fornecedor, aconteceu que a maioria das interfaces da biblioteca ficaram muito largas. Alguns métodos descreviam funcionalidades que outros servidores de nuvens não tinham.



ANTES: nem todos os clientes podem satisfazer os requerimentos de uma interface inchada.

Embora você ainda possa implementar esses métodos e colocar alguns tocos ali, não seria uma solução bonita. A melhor abordagem é quebrar a interface em partes. Classes que são capazes de implementar o que a interface original podem agora implementar apenas diversas interfaces refinadas. Outras classes podem implementar somente àquelas interfaces as quais têm métodos que façam sentidos para elas.



DEPOIS: uma interface inchada foi quebrada em um conjunto de interfaces mais granulares.

Como com os outros princípios, você pode exagerar com este aqui. Não divida mais uma interface que já está bastante específica. Lembre-se que, quanto mais interfaces você cria, mais complexo seu código se torna. Mantenha o equilíbrio.

D Princípio de inversão de dependência
ependency Inversion Principle

> Classes de alto nível não deveriam depender de classes de baixo nível. Ambas devem depender de abstrações. As abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.

Geralmente quando fazendo projeto the software, você pode fazer uma distinção entre dois níveis de classes.

- **Classes de baixo nível** implementam operações básicas tais como trabalhar com um disco, transferindo dados pela rede, conectar-se a uma base de dados, etc.
- **Classes de alto nível** contém lógica de negócio complexa que direcionam classes de baixo nível para fazerem algo.

Algumas pessoas fazem o projeto de classes de baixo nível primeiro e só depois trabalham nas de alto nível. Isso é muito comum quando você começa a desenvolver um protótipo de um novo sistema, e você não tem certeza do que será possível em alto nível porque as coisas de baixo nível não foram implementadas ainda ou não são claras. Com tal abordagem, classes da lógica de negócio tendem a ficar dependentes de classes primitivas de baixo nível.

O princípio de inversão de dependência sugere trocar a direção desta dependência.

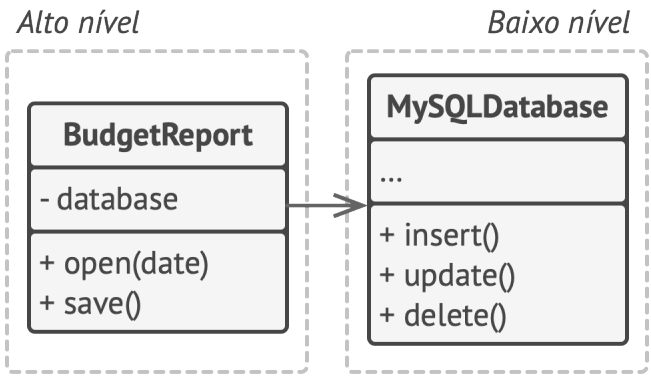
1. Para começar, você precisa descrever as interfaces para as operações de baixo nível que as classes de alto nível dependem, preferivelmente em termos de negócio. Por exemplo, a lógica do negócio deve chamar um método `abrirRelatório(arquivo)` ao invés de uma série de métodos `abrirArquivo(x)`, `lerBytes(n)`, `fecharArquivo(x)`. Estas interfaces contam como de alto nível.
2. Agora você pode fazer classes de alto nível dependentes daquelas interfaces, ao invés de classes concretas de baixo nível. Essa dependência será muito mais suave que a original.
3. Uma vez que as classes de baixo nível implementam essas interfaces, elas se tornam dependentes do nível da lógica do negócio, revertendo a direção da dependência original.

O princípio de inversão de dependência quase sempre anda junto com o *princípio aberto/fechado*: você pode estender classes de baixo nível para usar diferentes classes de lógica do negócio sem quebrar classes existentes.

Exemplo

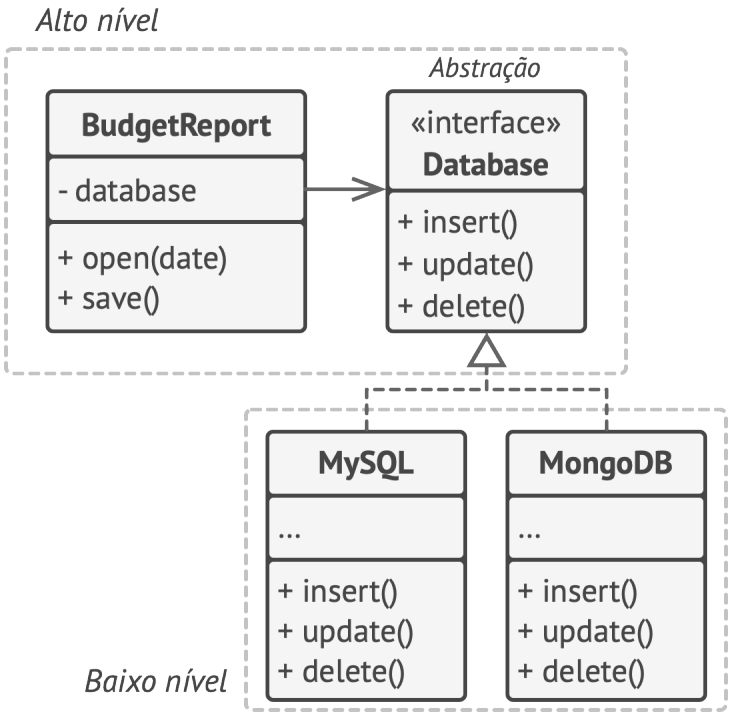
Neste exemplo, a classe de alto nível de relatório de orçamento usa uma classe de baixo nível de base de dados para ler e manter seus dados. Isso significa que quaisquer mudanças na classe de baixo nível, tais como quando uma nova versão da base de dados for lançada, podem afetar a classe de alto

nível, que não deveria se importar com os detalhes do armazenamento de dados.



ANTES: uma classe de alto nível depende de uma classe de baixo nível.

Você pode corrigir esse problema criando uma interface de alto nível que descreve as operações de ler/escrever e fazer a classe de relatório usar aquela interface ao invés da classe de baixo nível. Então você pode mudar ou estender a classe de baixo nível original para implementar a nova interface de ler/escrever declarada pela lógica do negócio.



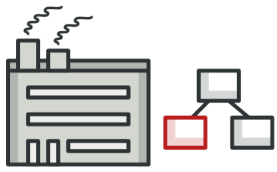
DEPOIS: classes de baixo nível dependem de uma abstração de alto nível.

Como resultado, a direção da dependência original foi invertida: classes de baixo nível agora dependem das abstrações de alto nível.

CATÁLOGO DOS PADRÕES DE PROJETO

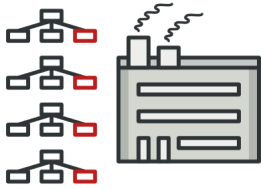
Padrões de projeto criacionais

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.

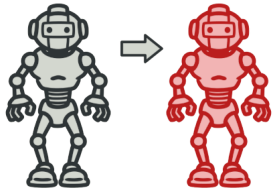


Abstract Factory

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

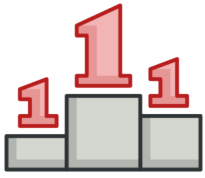
Builder

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



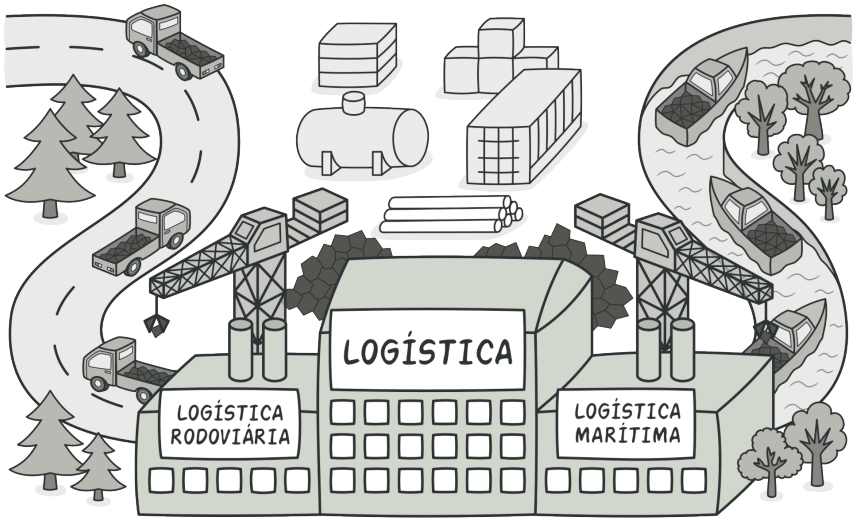
Prototype

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

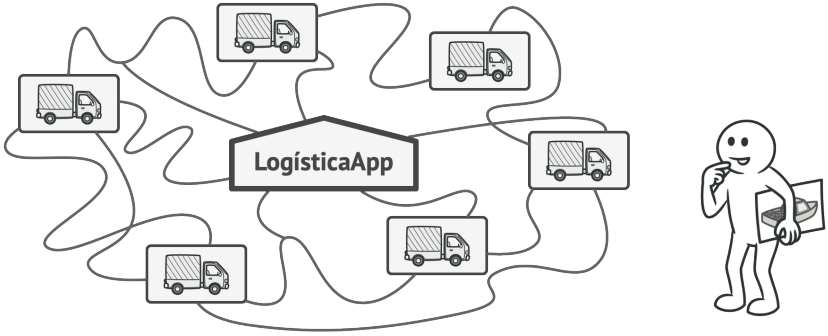
Também conhecido como: Método fábrica, Construtor virtual

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.



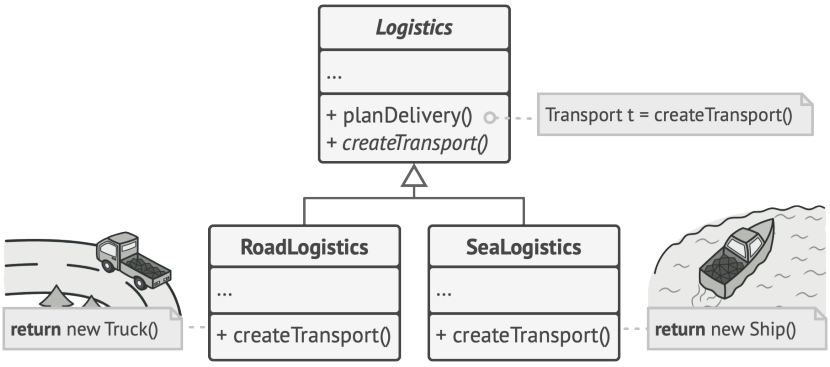
Adicionar uma nova classe ao programa não é tão simples se o restante do código já estiver acoplado às classes existentes.

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*.

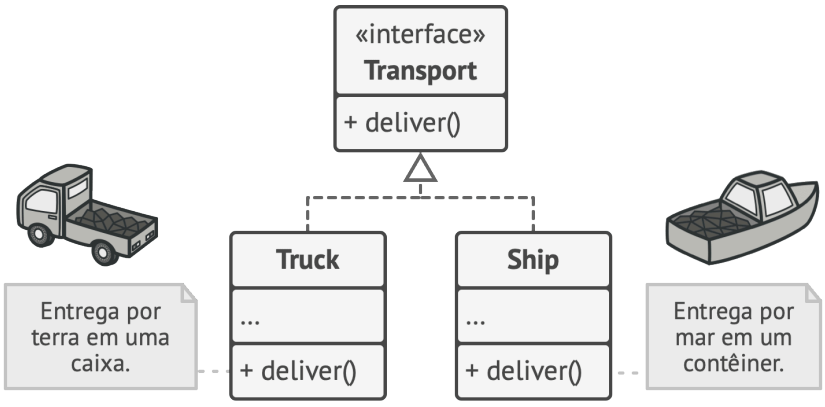


As subclasses podem alterar a classe de objetos retornados pelo método fábrica.

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

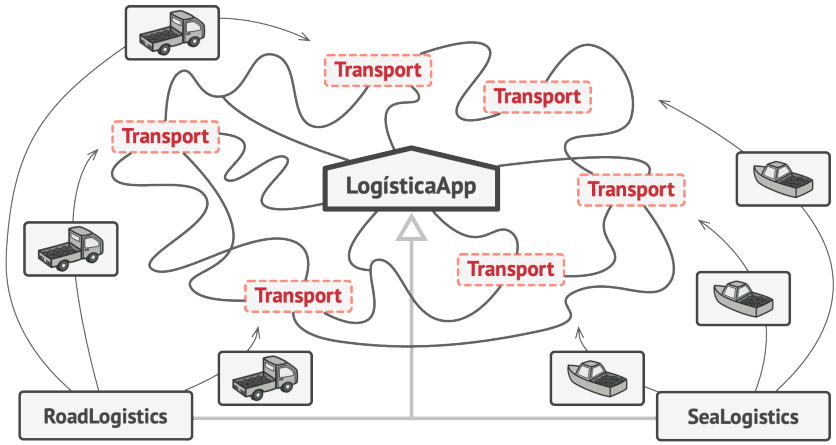
e alterar a classe de produtos que estão sendo criados pelo método.

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.

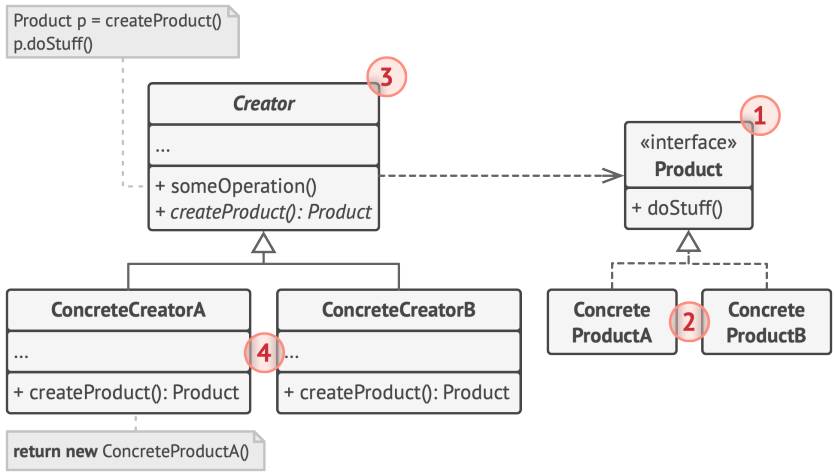
Por exemplo, ambas as classes `Caminhão` e `Navio` devem implementar a interface `Transporte`, que declara um método chamado `entregar`. Cada classe implementa esse método de maneira diferente: caminhões entregam carga por terra, navios entregam carga por mar. O método fábrica na classe `LogísticaViária`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

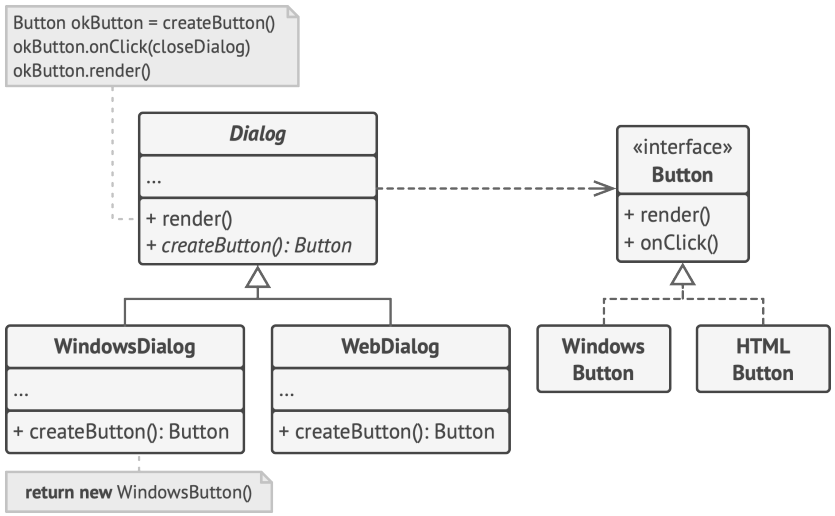
Observe que, apesar do nome, a criação de produtos **não** é a principal responsabilidade do criador. Normalmente, a classe criadora já possui alguma lógica de negócio relacionada aos produtos. O método fábrica ajuda a dissociar essa lógica das classes concretas de produtos. Aqui está uma analogia: uma 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.

```
// A classe criadora declara o método fábrica que deve retornar
// um objeto de uma classe produto. As subclasses da criadora
// geralmente fornecem a implementação desse método.
class Dialog is
  // A criadora também pode fornecer alguma implementação
  // padrão do Factory Method.
  abstract method createButton():Button

  // Observe que, apesar do seu nome, a principal
  // responsabilidade da criadora não é criar produtos. Ela
  // geralmente contém alguma lógica de negócio central que
  // depende dos objetos produto retornados pelo método
  // fábrica. As subclasses pode mudar indiretamente essa
  // lógica de negócio ao sobrescreverem o método fábrica e
  // retornarem um tipo diferente de produto dele.
  method render() is
    // Chame o método fábrica para criar um objeto produto.
    Button okButton = createButton()
```

```
    // Agora use o produto.
    okButton.onClick(closeDialog)
    okButton.render()


// Criadores concretos sobrescrevem o método fábrica para mudar
// o tipo de produto resultante.
class WindowsDialog extends Dialog is
  method createButton():Button is
    return new WindowsButton()

class WebDialog extends Dialog is
  method createButton():Button is
    return new HTMLButton()


// A interface do produto declara as operações que todos os
// produtos concretos devem implementar.
interface Button is
  method render()
  method onClick(f)

// Produtos concretos fornecem várias implementações da
// interface do produto.
class WindowsButton implements Button is
  method render(a, b) is
    // Renderiza um botão no estilo Windows.
  method onClick(f) is
    // Vincula um evento de clique do SO nativo.

class HTMLButton implements Button is
  method render(a, b) is
```

```
    // Retorna uma representação HTML de um botão.
  method onClick(f) is
    // Vincula um evento de clique no navegador web.


class Application is
  field dialog: Dialog

  // A aplicação seleciona um tipo de criador dependendo da
  // configuração atual ou definições de ambiente.
  method initialize() is
    config = readApplicationConfigFile()

    if (config.OS == "Windows") then
      dialog = new WindowsDialog()
    else if (config.OS == "Web") then
      dialog = new WebDialog()
    else
      throw new Exception("Error! Unknown operating system.")

  // O código cliente trabalha com uma instância de um criador
  // concreto, ainda que com sua interface base. Desde que o
  // cliente continue trabalhando com a criadora através da
  // interface base, você pode passar qualquer subclasse da
  // criadora.
  method main() is
    this.initialize()
    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`. E 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

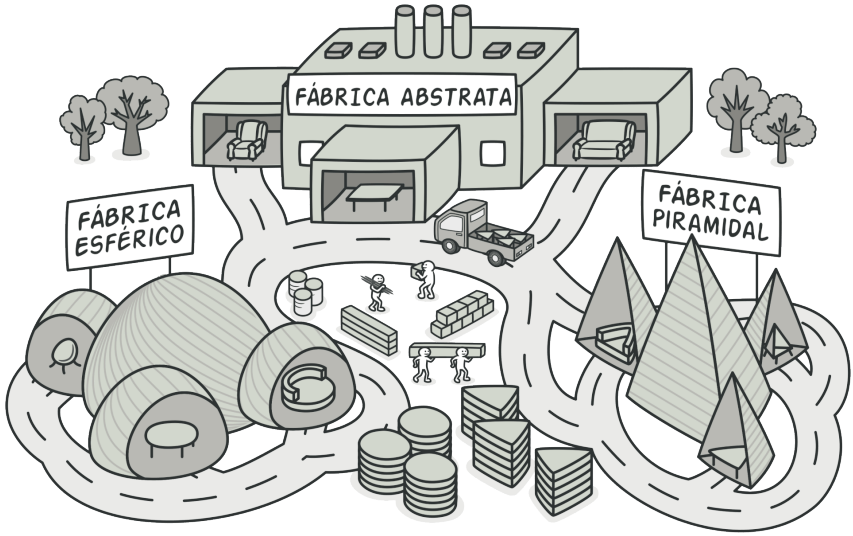
- Muitos projetos começam usando o **Factory Method** (menos complicado e mais customizável através de subclasses) e evoluem para o **Abstract Factory**, **Prototype**, ou **Builder** (mais flexíveis, mas mais complicados).

- 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.



ABSTRACT FACTORY

Também conhecido como: Fábrica abstrata

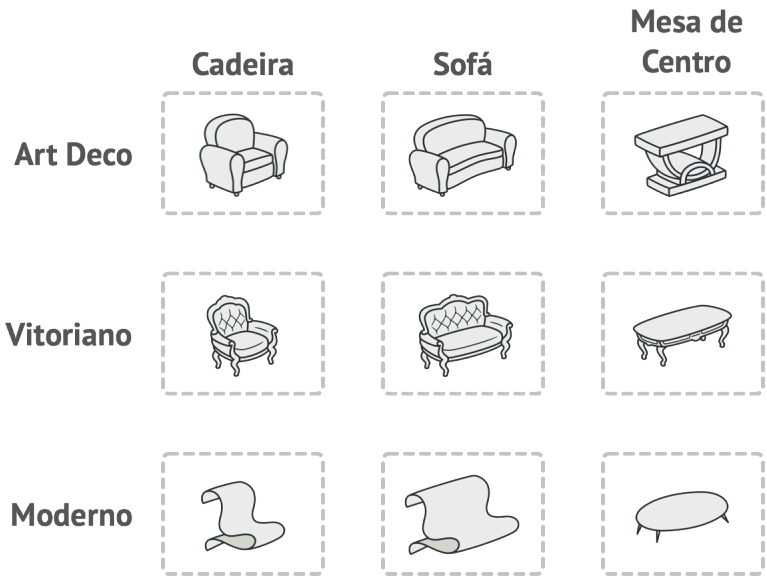
O **Abstract Factory** é um padrão de projeto criacional que permite que você produza famílias de objetos relacionados sem ter que especificar suas classes concretas.

☹ Problema

Imagine que você está criando um simulador de loja de mobílias. Seu código consiste de classes que representam:

1. Uma família de produtos relacionados, como: `Cadeira` + `Sofá` + `MesaDeCentro`.

2. Várias variantes dessa família. Por exemplo, produtos `Cadeira` + `Sofá` + `MesaDeCentro` estão disponíveis nessas variantes: `Moderno`, `Vitoriano`, `ArtDeco`.



Famílias de produtos e suas variantes.

Você precisa de um jeito de criar objetos de mobília individuais para que eles combinem com outros objetos da mesma fa-

mília. Os clientes ficam muito bravos quando recebem mobília que não combina.

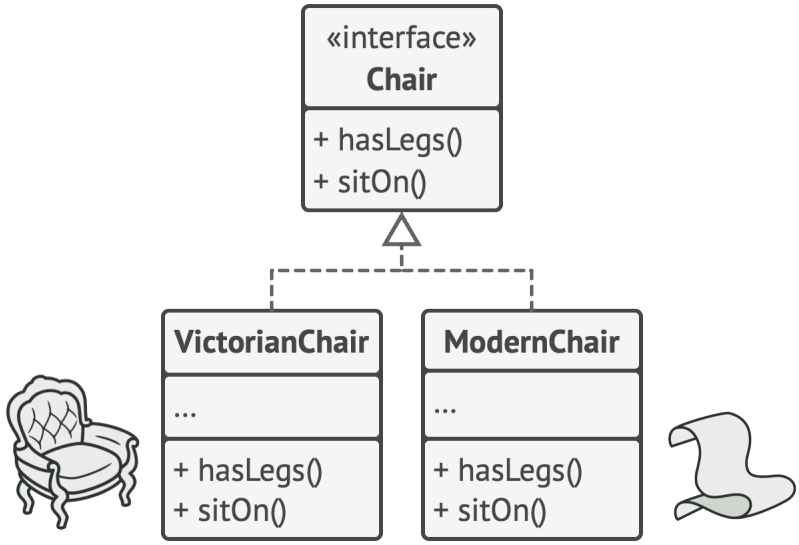
Um sofá no estilo Moderno não combina com cadeiras de estilo Vitoriano

E ainda, você não quer mudar o código existente quando adiciona novos produtos ou famílias de produtos ao programa. Os vendedores de mobílias atualizam seus catálogos com frequência e você não vai querer mudar o código base cada vez que isso acontece.

😊 Solução

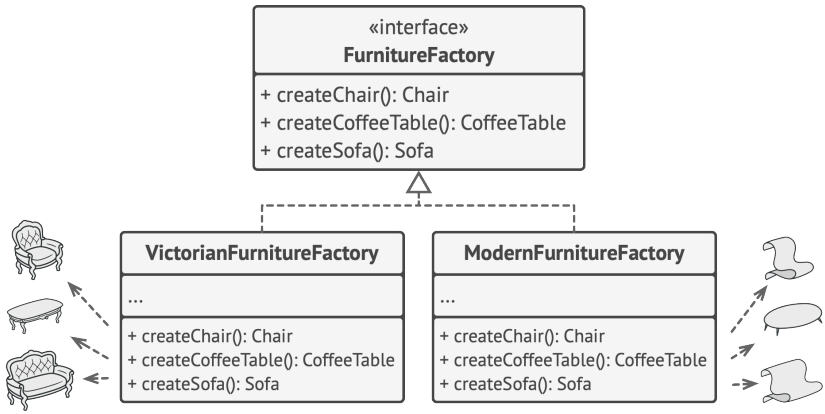
A primeira coisa que o padrão Abstract Factory sugere é declarar explicitamente interfaces para cada produto distinto da família de produtos (ex: cadeira, sofá ou mesa de centro). Então você pode fazer todas as variantes dos produtos seguirem essas interfaces. Por exemplo, todas as variantes de cadeira podem implementar a interface `Cadeira`; todas as

variantes de mesa de centro podem implementar a interface `MesaDeCentro`, e assim por diante.



Todas as variantes do mesmo objeto podem ser movidas para uma mesma hierarquia de classe.

O próximo passo é declarar a *fábrica abstrata*—uma interface com uma lista de métodos de criação para todos os produtos que fazem parte da família de produtos (por exemplo, `criarCadeira`, `criarSofá` e `criarMesaDeCentro`). Esses métodos devem retornar tipos **abstratos** de produtos representados pelas interfaces que extraímos previamente: `Cadeira`, `Sofá`, `MesaDeCentro` e assim por diante.



Cada fábrica concreta corresponde a uma variante de produto específica.

Agora, e o que fazer sobre as variantes de produtos? Para cada variante de uma família de produtos nós criamos uma classe fábrica separada baseada na interface `FábricaAbstrata`. Uma fábrica é uma classe que retorna produtos de um tipo em particular. Por exemplo, a classe `FábricaMobíliaModerna` só pode criar objetos `CadeiraModerna`, `SofáModerno`, e `MesaDeCentroModerna`.

O código cliente tem que funcionar com ambos as fábricas e produtos via suas respectivas interfaces abstratas. Isso permite que você mude o tipo de uma fábrica que passou para o código cliente, bem como a variante do produto que o código cliente recebeu, sem quebrar o código cliente atual.

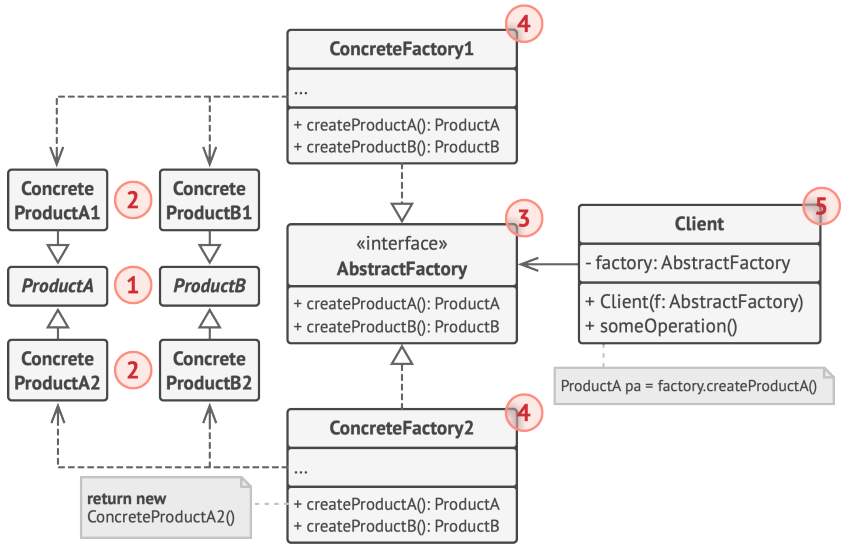
O cliente não deveria se importar com a classe concreta da fábrica com a qual está trabalhando.

Digamos que o cliente quer que uma fábrica produza uma cadeira. O cliente não precisa estar ciente da classe fábrica, e nem se importa que tipo de cadeira ele receberá. Seja ela um modelo Moderno ou no estilo Vitoriano, o cliente precisa tratar todas as cadeiras da mesma maneira, usando a interface abstrata `Cadeira`. Com essa abordagem, a única coisa que o cliente sabe sobre a cadeira é que ela implementa o método `sentar` de alguma maneira. E também, seja qual for a variante da cadeira retornada, ela sempre irá combinar com o tipo de sofá ou mesa de centro produzido pelo mesmo objeto fábrica.

Há mais uma coisa a se clarificar: se o cliente está exposto apenas às interfaces abstratas, o que realmente cria os objetos fábrica então? Geralmente, o programa cria um objeto fábrica concreto no estágio de inicialização. Antes disso, o programa

deve selecionar o tipo de fábrica dependendo da configuração ou definições de ambiente.

Estrutura



1. **Produtos Abstratos** declaram interfaces para um conjunto de produtos distintos mas relacionados que fazem parte de uma família de produtos.

2. **Produtos Concretos** são várias implementações de produtos abstratos, agrupados por variantes. Cada produto abstrato (cadeira/sofá) deve ser implementado em todas as variantes dadas (Vitoriano/Moderno).

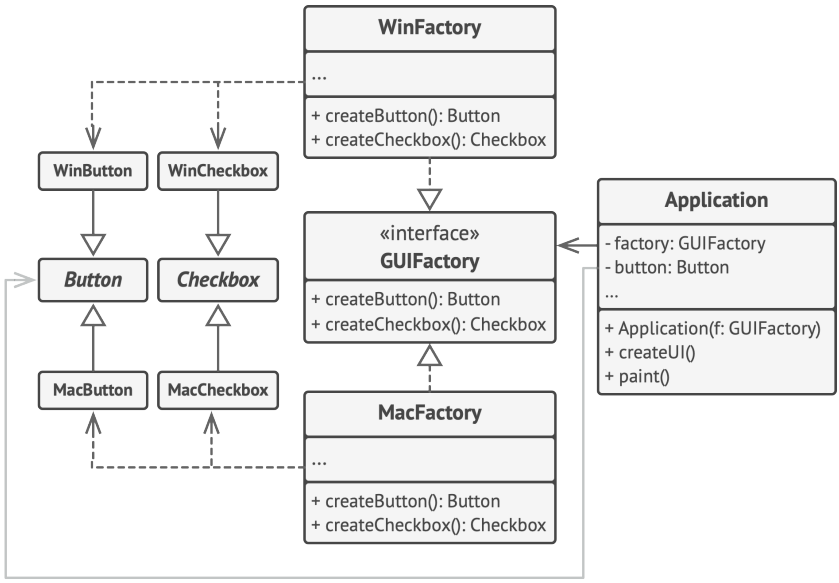
3. A interface **Fábrica Abstrata** declara um conjunto de métodos para criação de cada um dos produtos abstratos.

4. **Fábricas Concretas** implementam métodos de criação fábrica abstratos. Cada fábrica concreta corresponde a uma variante específica de produtos e cria apenas aquelas variantes de produto.

5. Embora fábricas concretas instanciam produtos concretos, assinaturas dos seus métodos de criação devem retornar produtos *abstratos* correspondentes. Dessa forma o código cliente que usa uma fábrica não fica ligada a variante específica do produto que ele pegou de uma fábrica. O **Cliente** pode trabalhar com qualquer variante de produto/fábrica concreto, desde que ele se comunique com seus objetos via interfaces abstratas.

Pseudocódigo

Este exemplo ilustra como o padrão **Abstract Factory** pode ser usado para criar elementos UI multiplataforma sem ter que ligar o código do cliente às classes UI concretas, enquanto mantém todos os elementos criados consistentes com um sistema operacional escolhido.



Exemplo das classes UI multiplataforma.

É esperado que os mesmos elementos UI de um aplicativo multiplataforma se comportem de forma semelhante, mas que se pareçam um pouco diferentes nos diferentes sistemas operacionais. Além disso, é seu trabalho garantir que os elementos UI coincidam com o estilo do sistema operacional atual. Você não vai querer que seu programa renderize controles macOS quando é executado no Windows.

A interface da fábrica abstrata declara um conjunto de métodos de criação que o código cliente pode usar para produzir diferentes tipos de elementos de UI que coincidam com o SO particular. Fábricas concretas correspondem a sistemas operacionais específicos e criam os elementos de UI que corresponde com aquele SO em particular.

Funciona assim: quando a aplicação inicia, ela checa o tipo de sistema operacional que está sendo utilizado. A aplicação usa essa informação para criar um objeto fábrica de uma classe que corresponde com o sistema operacional. O resto do código usa essa fábrica para criar elementos UI. Isso previne que elementos errados sejam criados.

Com essa abordagem, o código cliente não depende de classes concretas de fábricas e elementos UI desde que ele trabalhe com esses objetos através de suas interfaces abstratas. Isso também permite que o código do cliente suporte outras fábricas ou elementos UI que você possa adicionar no futuro.

Como resultado, você não precisa modificar o código do cliente cada vez que adicionar uma variação de elementos de UI em sua aplicação. Você só precisa criar uma nova classe fábrica que produza esses elementos e modificar de forma sutil o código de inicialização da aplicação de forma que ele selecione aquela classe quando apropriado.

```
// A interface fábrica abstrata declara um conjunto de métodos
// que retorna diferentes produtos abstratos. Estes produtos são
// chamados uma família e estão relacionados por um tema ou
// conceito de alto nível. Produtos de uma família são
// geralmente capazes de colaborar entre si. Uma família de
// produtos pode ter várias variantes, mas os produtos de uma
// variante são incompatíveis com os produtos de outro variante.
interface GUIFactory is
  method createButton():Button
```

```
    method createCheckbox():Checkbox


// As fábricas concretas produzem uma família de produtos que
// pertencem a uma única variante. A fábrica garante que os
// produtos resultantes sejam compatíveis. Assinaturas dos
// métodos fabrica retornam um produto abstrato, enquanto que
// dentro do método um produto concreto é instanciado.
class WinFactory implements GUIFactory is
  method createButton():Button is
    return new WinButton()
  method createCheckbox():Checkbox is
    return new WinCheckbox()

// Cada fábrica concreta tem uma variante de produto
// correspondente.
class MacFactory implements GUIFactory is
  method createButton():Button is
    return new MacButton()
  method createCheckbox():Checkbox is
    return new MacCheckbox()


// Cada produto distinto de uma família de produtos deve ter uma
// interface base. Todas as variantes do produto devem
// implementar essa interface.
interface Button is
  method paint()

// Produtos concretos são criados por fábricas concretas
// correspondentes.
class WinButton implements Button is
```

```
    method paint() is
      // Renderiza um botão no estilo Windows.

class MacButton implements Button is
    method paint() is
      // Renderiza um botão no estilo macOS.

// Aqui está a interface base de outro produto. Todos os
// produtos podem interagir entre si, mas a interação apropriada
// só é possível entre produtos da mesma variante concreta.
interface Checkbox is
    method paint()

class WinCheckbox implements Checkbox is
    method paint() is
      // Renderiza uma caixa de seleção estilo Windows.

class MacCheckbox implements Checkbox is
    method paint() is
      // Renderiza uma caixa de seleção no estilo macOS.


// O código cliente trabalha com fábricas e produtos apenas
// através de tipos abstratos: GUIFactory, Button e Checkbox.
// Isso permite que você passe qualquer subclasse fábrica ou de
// produto para o código cliente sem quebrá-lo.
class Application is
    private field factory: GUIFactory
    private field button: Button
    constructor Application(factory: GUIFactory) is
      this.factory = factory
    method createUI() is
```

```
    this.button = factory.createButton()
  method paint() is
    button.paint()


// A aplicação seleciona o tipo de fábrica dependendo da atual
// configuração do ambiente e cria o widget no tempo de execução
// (geralmente no estágio de inicialização).
class ApplicationConfigurator is
  method main() is
    config = readApplicationConfigFile()

    if (config.OS == "Windows") then
      factory = new WinFactory()
    else if (config.OS == "Mac") then
      factory = new MacFactory()
    else
      throw new Exception("Error! Unknown operating system.")

    Application app = new Application(factory)
```

Aplicabilidade

Use o Abstract Factory quando seu código precisa trabalhar com diversas famílias de produtos relacionados, mas que você não quer depender de classes concretas daqueles produtos-eles podem ser desconhecidos de antemão ou você simplesmente quer permitir uma futura escalabilidade.

O Abstract Factory fornece a você uma interface para a criação de objetos de cada classe das famílias de produtos. Desde que seu código crie objetos a partir dessa interface, você não precisará se preocupar em criar uma variante errada de um produto que não coincida com produtos já criados por sua aplicação.

- Considere implementar o Abstract Factory quando você tem uma classe com um conjunto de **métodos fábrica** que desfoquem sua responsabilidade principal.

- Em um programa bem desenvolvido *cada classe é responsável por apenas uma coisa*. Quando uma classe lida com múltiplos tipos de produto, pode valer a pena extrair seus métodos fábrica em uma classe fábrica solitária ou uma implementação plena do Abstract Factory.

Como implementar

1. Mapeie uma matriz de tipos de produtos distintos versus as variantes desses produtos.

2. Declare interfaces de produto abstratas para todos os tipos de produto. Então, faça todas as classes concretas de produtos implementar essas interfaces.

3. Declare a interface da fábrica abstrata com um conjuntos de métodos de criação para todos os produtos abstratos.

4. Implemente um conjunto de classes fábricas concretas, uma para cada variante de produto.

5. Crie um código de inicialização da fábrica em algum lugar da aplicação. Ele deve instanciar uma das classes fábrica concretas, dependendo da configuração da aplicação ou do ambiente atual. Passe esse objeto fábrica para todas as classes que constroem produtos.

6. Escaneie o código e encontre todas as chamadas diretas para construtores de produtos. Substitua-as por chamadas para o método de criação apropriado no objeto fábrica.

Prós e contras

✓ Você pode ter certeza que os produtos que você obtém de uma fábrica são compatíveis entre si.

✓ Você evita um vínculo forte entre produtos concretos e o código cliente.

✓ *Princípio de responsabilidade única*. Você pode extrair o código de criação do produto para um lugar, fazendo o código ser de fácil manutenção.

✓ *Princípio aberto/fechado*. Você pode introduzir novas variantes de produtos sem quebrar o código cliente existente.

✕ O código pode tornar-se mais complicado do que deveria ser, uma vez que muitas novas interfaces e classes são introduzidas junto com o padrão.

⇄ 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).

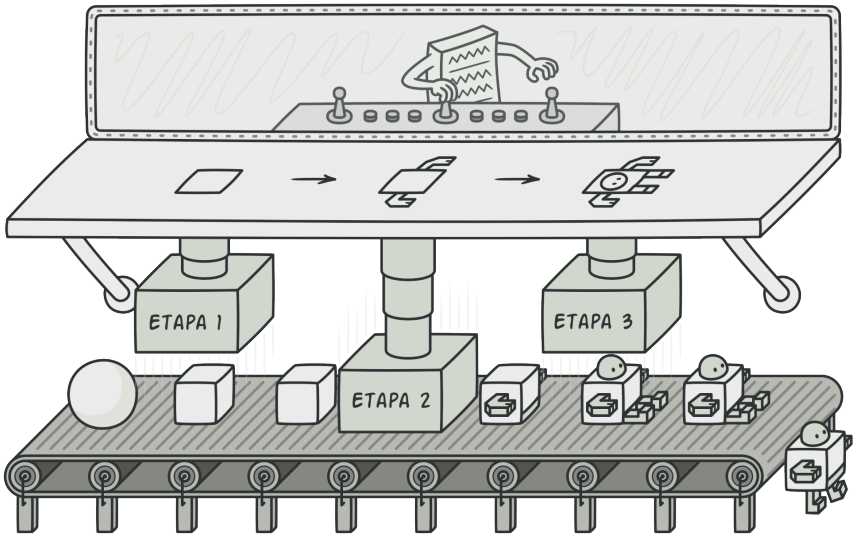
- O **Builder** foca em construir objetos complexos passo a passo. O **Abstract Factory** se especializa em criar famílias de objetos relacionados. O *Abstract Factory* retorna o produto imediatamente, enquanto que o *Builder* permite que você execute algumas etapas de construção antes de buscar o produto.

- 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.

- O **Abstract Factory** pode servir como uma alternativa para o **Facade** quando você precisa apenas esconder do código cliente a forma com que são criados os objetos do subsistema.

- Você pode usar o **Abstract Factory** junto com o **Bridge**. Esse pareamento é útil quando algumas abstrações definidas pelo *Bridge* só podem trabalhar com implementações específicas. Neste caso, o *Abstract Factory* pode encapsular essas relações e esconder a complexidade do código cliente.

- As **Fábricas Abstratas**, **Construtores**, e **Protótipos** podem todos ser implementados como **Singletons**.



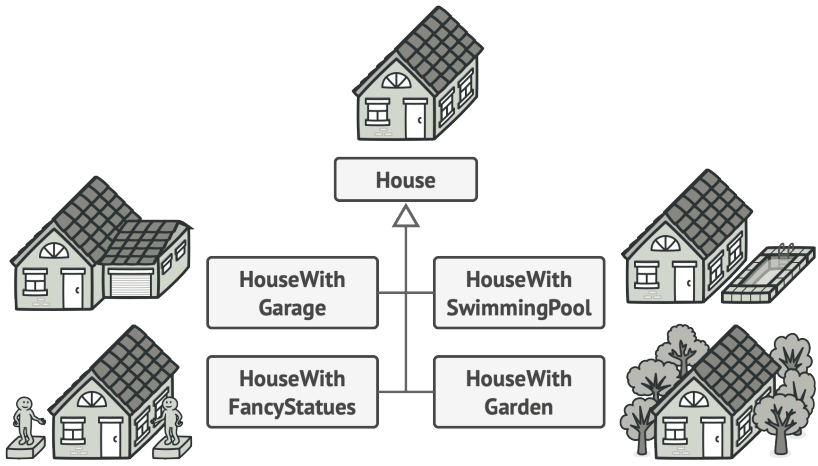
BUILDER

Também conhecido como: Construtor

O **Builder** é um padrão de projeto criacional que permite a você construir objetos complexos passo a passo. O padrão permite que você produza diferentes tipos e representações de um objeto usando o mesmo código de construção.

☹ Problema

Imagine um objeto complexo que necessite de uma inicialização passo a passo trabalhosa de muitos campos e objetos agrupados. Tal código de inicialização fica geralmente enterrado dentro de um construtor monstruoso com vários parâmetros. Ou pior: espalhado por todo o código cliente.

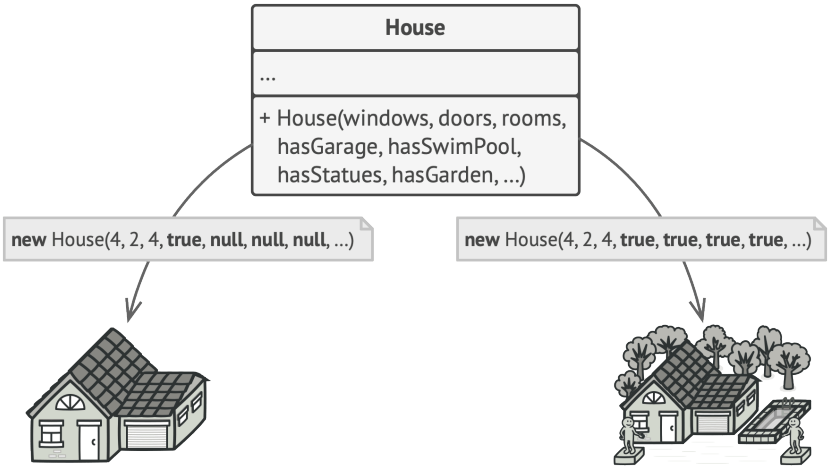


Você pode tornar o programa muito complexo ao criar subclasses para cada possível configuração de um objeto.

Por exemplo, vamos pensar sobre como criar um objeto `Casa`. Para construir uma casa simples, você precisa construir quatro paredes e um piso, instalar uma porta, encaixar um par de janelas, e construir um teto. Mas e se você quiser uma casa maior e mais iluminada, com um jardim e outras miudezas (como um sistema de aquecimento, encanamento, e fiação elétrica)?

A solução mais simples é estender a classe base `Casa` e criar um conjunto de subclasses para cobrir todas as combinações de parâmetros. Mas eventualmente você acabará com um número considerável de subclasses. Qualquer novo parâmetro, tal como o estilo do pórtico, irá forçá-lo a aumentar essa hierarquia cada vez mais.

Há outra abordagem que não envolve a propagação de subclasses. Você pode criar um construtor gigante diretamente na classe `Casa` base com todos os possíveis parâmetros que controlam o objeto casa. Embora essa abordagem realmente elimine a necessidade de subclasses, ela cria outro problema.



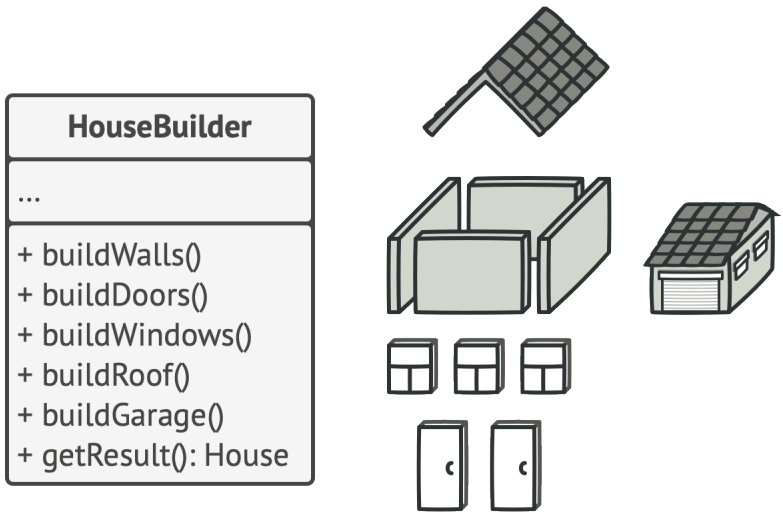
O construtor com vários parâmetros tem um lado ruim: nem todos os parâmetros são necessários todas as vezes.

Na maioria dos casos a maioria dos parâmetros não será usada, tornando **as chamadas do construtor em algo feio de se ver**. Por exemplo, apenas algumas casas têm piscinas, então os pa-

râmetros relacionados a piscinas serão inúteis nove em cada dez vezes.

😊 Solução

O padrão Builder sugere que você extraia o código de construção do objeto para fora de sua própria classe e mova ele para objetos separados chamados *builders*. "Builder" significa "construtor", mas não usaremos essa palavra para evitar confusão com os construtores de classe.



O padrão Builder permite que você construa objetos complexos passo a passo. O Builder não permite que outros objetos acessem o produto enquanto ele está sendo construído.

O padrão organiza a construção de objetos em uma série de etapas (`construirParedes`, `construirPorta`, etc.). Para criar um objeto você executa uma série de etapas em um objeto

builder. A parte importante é que você não precisa chamar todas as etapas. Você chama apenas aquelas etapas que são necessárias para a produção de uma configuração específica de um objeto.

Algumas das etapas de construção podem necessitar de implementações diferentes quando você precisa construir várias representações do produto. Por exemplo, paredes de uma cabana podem ser construídas com madeira, mas paredes de um castelo devem ser construídas com pedra.

Nesse caso, você pode criar diferentes classes construturas que implementam as mesmas etapas de construção, mas de maneira diferente. Então você pode usar esses builders no processo de construção (i.e, um pedido ordenado de chamadas para as etapas de construção) para produzir diferentes tipos de objetos.

Builders diferentes executam a mesma tarefa de várias maneiras.

Por exemplo, imagine um builder que constrói tudo de madeira e vidro, um segundo builder que constrói tudo com pedra e ferro, e um terceiro que usa ouro e diamantes. Ao chamar o mesmo conjunto de etapas, você obtém uma casa normal do primeiro builder, um pequeno castelo do segundo, e um palácio do terceiro. Contudo, isso só vai funcionar se o código cliente que chama as etapas de construção é capaz de interagir com os builders usando uma interface comum.

Diretor

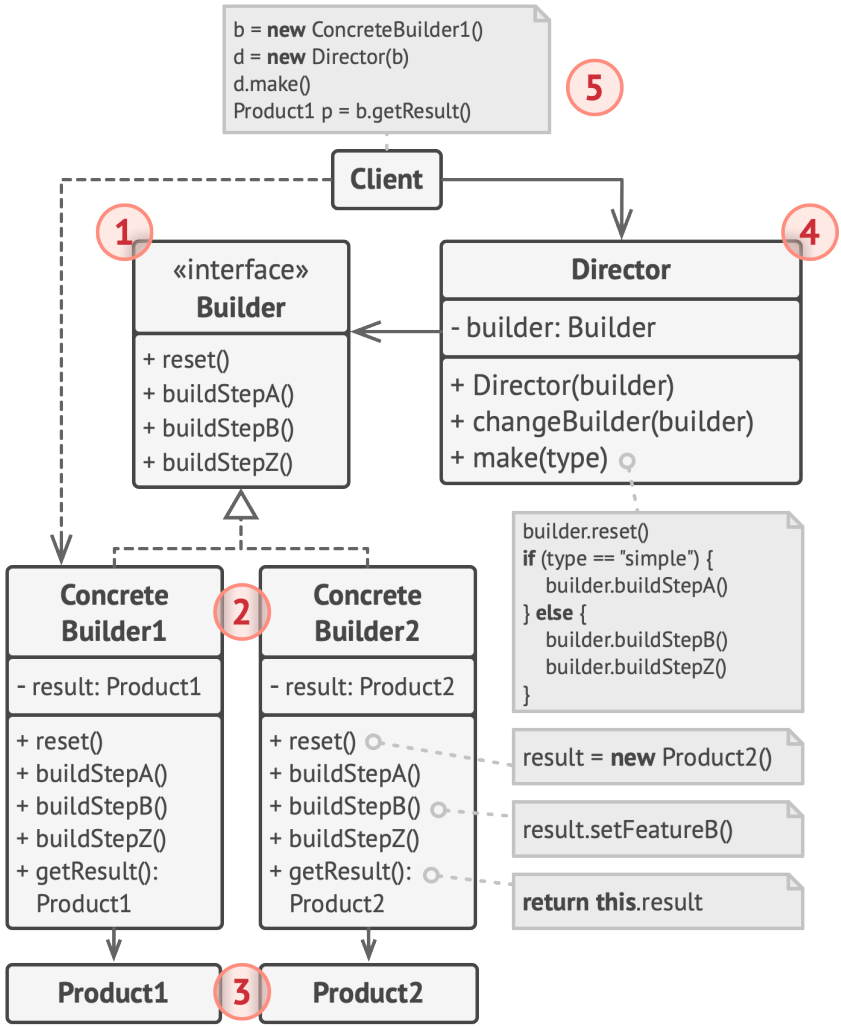
Você pode ir além e extrair uma série de chamadas para as etapas do builder que você usa para construir um produto em uma classe separada chamada *diretor*. A classe diretor define a ordem na qual executar as etapas de construção, enquanto que o builder provê a implementação dessas etapas.

O diretor sabe quais etapas de construção executar para obter um produto que funciona.

Ter uma classe diretor em seu programa não é estritamente necessário. Você sempre pode chamar as etapas de construção em uma ordem específica diretamente do código cliente. Contudo, a classe diretor pode ser um bom lugar para colocar várias rotinas de construção para que você possa reutilizá-las em qualquer lugar do seu programa.

Além disso, a classe diretor esconde completamente os detalhes da construção do produto do código cliente. O cliente só precisa associar um builder com um diretor, inicializar a construção com o diretor, e então obter o resultado do builder.

Estrutura



1. A interface **Builder** declara etapas de construção do produto que são comuns a todos os tipos de builders.

2. **Builders Concretos** provém diferentes implementações das etapas de construção. Builders concretos podem produzir produtos que não seguem a interface comum.

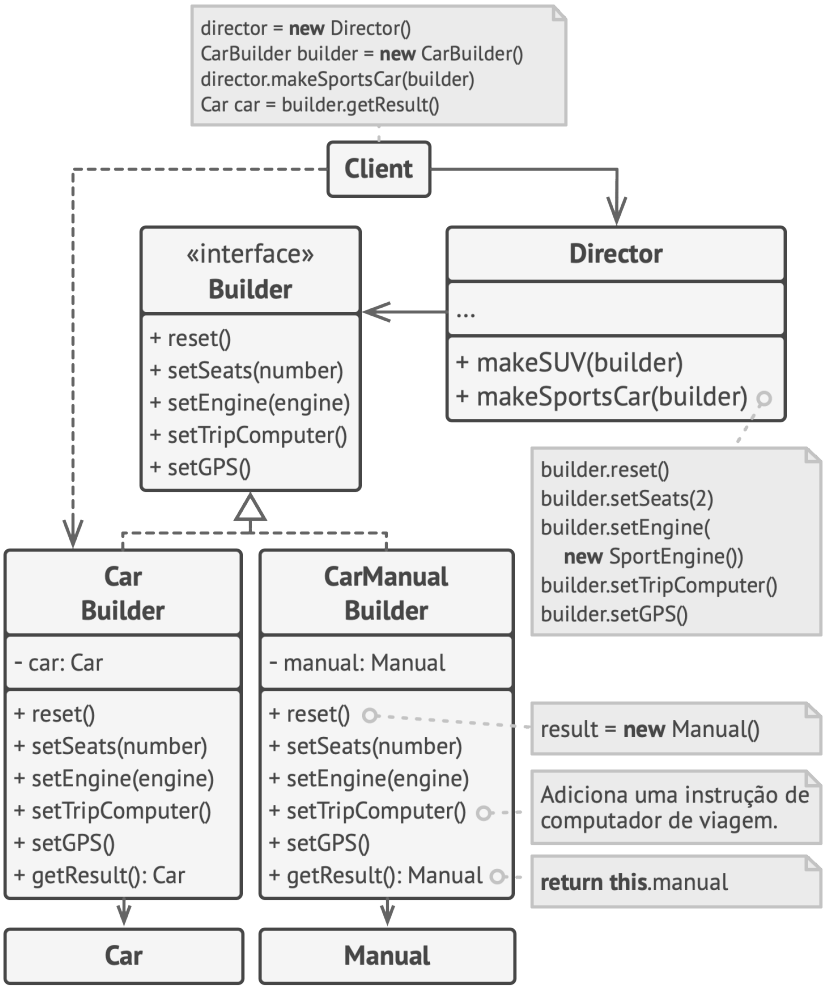
3. **Produtos** são os objetos resultantes. Produtos construídos por diferentes builders não precisam pertencer a mesma interface ou hierarquia da classe.

4. A classe **Diretor** define a ordem na qual as etapas de construção são chamadas, então você pode criar e reutilizar configurações específicas de produtos.

5. O **Cliente** deve associar um dos objetos builders com o diretor. Usualmente isso é feito apenas uma vez, através de parâmetros do construtor do diretor. O diretor então usa aquele objeto builder para todas as futuras construções. Contudo, há uma abordagem alternativa para quando o cliente passa o objeto builder ao método de produção do diretor. Nesse caso, você pode usar um builder diferente a cada vez que você produzir alguma coisa com o diretor.

Pseudocódigo

Este exemplo do padrão **Builder** ilustra como você pode reutilizar o mesmo código de construção de objeto quando construindo diferentes tipos de produtos, tais como carros, e como criar manuais correspondentes para eles.



O exemplo do passo a passo da construção de carros e do manual do usuário que se adapta a aqueles modelos de carros.

Um carro é um objeto complexo que pode ser construído em centenas de maneiras diferentes. Ao invés de inchar a classe `Carro` com um construtor enorme, nós extraímos o código de montagem do carro em uma classe de construção de carro se-

parada. Essa classe tem um conjunto de métodos para configurar as várias partes de um carro.

Se o código cliente precisa montar um modelo de carro especial e ajustado, ele pode trabalhar com o builder diretamente. Por outro lado, o cliente pode delegar a montagem à classe diretor, que sabe como usar um builder para construir diversos modelos de carros populares.

Você pode ficar chocado, mas cada carro precisa de um manual (sério, quem lê aquilo?). O manual descreve cada funcionalidade do carro, então os detalhes dos manuais variam de acordo com os diferentes modelos. É por isso que faz sentido reutilizar um processo de construção existente para ambos os carros e seus respectivos manuais. É claro, construir um manual não é o mesmo que construir um carro, e é por isso que devemos providenciar outra classe builder que se especializa em compor manuais. Essa classe implementa os mesmos métodos de construção que seus parentes builder de carros, mas ao invés de construir partes de carros, ela as descreve. Ao passar esses builders ao mesmo objeto diretor, podemos tanto construir um carro como um manual.

A parte final é obter o objeto resultante. Um carro de metal e um manual de papel, embora relacionados, são coisas muito diferentes. Não podemos colocar um método para obter resultados no diretor sem ligar o diretor às classes de produto concretas. Portanto, nós obtemos o resultado da construção a partir do builder que fez o trabalho.

```
// Usar o padrão Builder só faz sentido quando seus produtos são
// bem complexos e requerem configuração extensiva. Os dois
// produtos a seguir são relacionados, embora eles não tenham
// uma interface em comum.
class Car is
  // Um carro pode ter um GPS, computador de bordo, e alguns
  // assentos. Diferentes modelos de carros (esportivo, SUV,
  // conversível) podem ter diferentes funcionalidades
  // instaladas ou equipadas.

class Manual is
  // Cada carro deve ter um manual do usuário que corresponda
  // a configuração do carro e descreva todas suas
  // funcionalidades.


// A interface builder especifica métodos para criar as
// diferentes partes de objetos produto.
interface Builder is
  method reset()
  method setSeats(...)
  method setEngine(...)
  method setTripComputer(...)
  method setGPS(...)

// As classes builder concretas seguem a interface do
// builder e fornecem implementações específicas das etapas
// de construção. Seu programa pode ter algumas variações de
// builders, cada uma implementada de forma diferente.
class CarBuilder implements Builder is
  private field car:Car

```

```
    // Uma instância fresca do builder deve conter um objeto
    // produto em branco na qual ela usa para montagem futura.
    constructor CarBuilder() is
      this.reset()

    // O método reset limpa o objeto sendo construído.
    method reset() is
      this.car = new Car()

    // Todas as etapas de produção trabalham com a mesma
    // instância de produto.
    method setSeats(...) is
      // Define o número de assentos no carro.

    method setEngine(...) is
      // Instala um tipo de motor.

    method setTripComputer(...) is
      // Instala um computador de bordo.

    method setGPS(...) is
      // Instala um sistema de posicionamento global.

    // Builders concretos devem fornecer seus próprios
    // métodos para recuperar os resultados. Isso é porque
    // vários tipos de builders podem criar produtos
    // inteiramente diferentes que nem sempre seguem a mesma
    // interface. Portanto, tais métodos não podem ser
    // declarados na interface do builder (ao menos não em
    // uma linguagem de programação de tipo estático).
    //
    // Geralmente, após retornar o resultado final para o
```

```
    // cliente, espera-se que uma instância de builder comece
    // a produzir outro produto. É por isso que é uma prática
    // comum chamar o método reset no final do corpo do método
    // `getProduct`. Contudo este comportamento não é
    // obrigatório, e você pode fazer seu builder esperar por
    // uma chamada explícita do reset a partir do código cliente
    // antes de se livrar de seu resultado anterior.
    method getProduct():Car is
      product = this.car
      this.reset()
      return product

// Ao contrário dos outros padrões criacionais, o Builder
// permite que você construa produtos que não seguem uma
// interface comum.
class CarManualBuilder implements Builder is
    private field manual:Manual

    constructor CarManualBuilder() is
      this.reset()

    method reset() is
      this.manual = new Manual()

    method setSeats(...) is
      // Documenta as funcionalidades do assento do carro.

    method setEngine(...) is
      // Adiciona instruções do motor.

    method setTripComputer(...) is
      // Adiciona instruções do computador de bordo.
```

```
97
98    method setGPS(...) is
99      // Adiciona instruções do GPS.
100
101   method getProduct():Manual is
102     // Retorna o manual e reseta o builder.
103
104
105 // O diretor é apenas responsável por executar as etapas de
106 // construção em uma sequência em particular. Isso ajuda quando
107 // produzindo produtos de acordo com uma ordem específica ou
108 // configuração. A rigor, a classe diretor é opcional, já que o
109 // cliente pode controlar os builders diretamente.
110 class Director is
111   private field builder:Builder
112
113   // O diretor trabalha com qualquer instância builder que
114   // o código cliente passar a ele. Dessa forma, o código
115   // cliente pode alterar o tipo final do produto recém
116   // montado.
117   method setBuilder(builder:Builder)
118     this.builder = builder
119
120   // O diretor pode construir diversas variações do produto
121   // usando as mesmas etapas de construção.
122   method constructSportsCar(builder: Builder) is
123     builder.reset()
124     builder.setSeats(2)
125     builder.setEngine(new SportEngine())
126     builder.setTripComputer(true)
127     builder.setGPS(true)
128
```

```
  method constructSUV(builder: Builder) is
    // ...


// O código cliente cria um objeto builder, passa ele para o
// diretor e então inicia o processo de construção. O resultado
// final é recuperado do objeto builder.
class Application is

  method makeCar() is
    director = new Director()

    CarBuilder builder = new CarBuilder()
    director.constructSportsCar(builder)
    Car car = builder.getProduct()

    CarManualBuilder builder = new CarManualBuilder()
    director.constructSportsCar(builder)

    // O produto final é frequentemente retornado de um
    // objeto builder uma vez que o diretor não está
    // ciente e não é dependente de builders e produtos
    // concretos.
    Manual manual = builder.getProduct()
```

Aplicabilidade

Use o padrão Builder para se livrar de um "construtor telescópico".

Digamos que você tenha um construtor com dez parâmetros opcionais. Chamar um monstro desses é muito inconveniente; portanto, você sobrecarrega o construtor e cria diversas versões curtas com menos parâmetros. Esses construtores ainda se referem ao principal, passando alguns valores padrão para qualquer parâmetro omitido.

```
class Pizza {
  Pizza(int size) { ... }
  Pizza(int size, boolean cheese) { ... }
  Pizza(int size, boolean cheese, boolean pepperoni) { ... }
  // ...
```

Criar tal monstro só é possível em linguagens que suportam sobrecarregamento de método, tais como C# ou Java.

O padrão Builder permite que você construa objetos passo a passo, usando apenas aquelas etapas que você realmente precisa. Após implementar o padrão, você não vai mais precisar amontoar dúzias de parâmetros em seus construtores.

Use o padrão Builder quando você quer que seu código seja capaz de criar diferentes representações do mesmo produto (por exemplo, casas de pedra e madeira).

O padrão Builder pode ser aplicado quando a construção de várias representações do produto envolvem etapas similares que diferem apenas nos detalhes.

A interface base do builder define todas as etapas de construção possíveis, e os buildrs concretos implementam essas etapas para construir representações particulares do produto. Enquanto isso, a classe diretor guia a ordem de construção.

Use o Builder para construir árvores Composite ou outros objetos complexos.

O padrão Builder permite que você construa produtos passo a passo. Você pode adiar a execução de algumas etapas sem quebrar o produto final. Você pode até chamar etapas recursivamente, o que é bem útil quando você precisa construir uma árvore de objetos.

Um builder não expõe o produto não finalizado enquanto o processo de construção estiver executando etapas. Isso previne o código cliente de obter um resultado incompleto.

Como implementar

1. Certifique-se que você pode definir claramente as etapas comuns de construção para construir todas as representações do produto disponíveis. Do contrário, você não será capaz de implementar o padrão.

2. Declare essas etapas na interface builder base.

3. Crie uma classe builder concreta para cada representação do produto e implemente suas etapas de construção.

Não se esqueça de implementar um método para recuperar os resultados da construção. O motivo pelo qual esse método não pode ser declarado dentro da interface do builder é porque vários builders podem construir produtos que não tem uma interface comum. Portanto, você não sabe qual será o tipo de retorno para tal método. Contudo, se você está lidando com produtos de uma única hierarquia, o método de obtenção pode ser adicionado com segurança para a interface base.

4. Pense em criar uma classe diretor. Ela pode encapsular várias maneiras de construir um produto usando o mesmo objeto builder.

5. O código cliente cria tanto os objetos do builder como do diretor. Antes da construção começar, o cliente deve passar um objeto builder para o diretor. Geralmente o cliente faz isso apenas uma vez, através de parâmetros do construtor do diretor. O diretor usa o objeto builder em todas as construções futuras. Existe uma alternativa onde o builder é passado diretamente ao método de construção do diretor.

6. O resultado da construção pode ser obtido diretamente do diretor apenas se todos os produtos seguirem a mesma interface. Do contrário o cliente deve obter o resultado do builder.

Prós e contras

✓ Você pode construir objetos passo a passo, adiar as etapas de construção ou rodar etapas recursivamente.

✓ Você pode reutilizar o mesmo código de construção quando construindo várias representações de produtos.

✓ *Princípio de responsabilidade única*. Você pode isolar um código de construção complexo da lógica de negócio do produto.

✕ A complexidade geral do código aumenta uma vez que o padrão exige criar múltiplas classes novas.

⇄ Relações com outros padrões

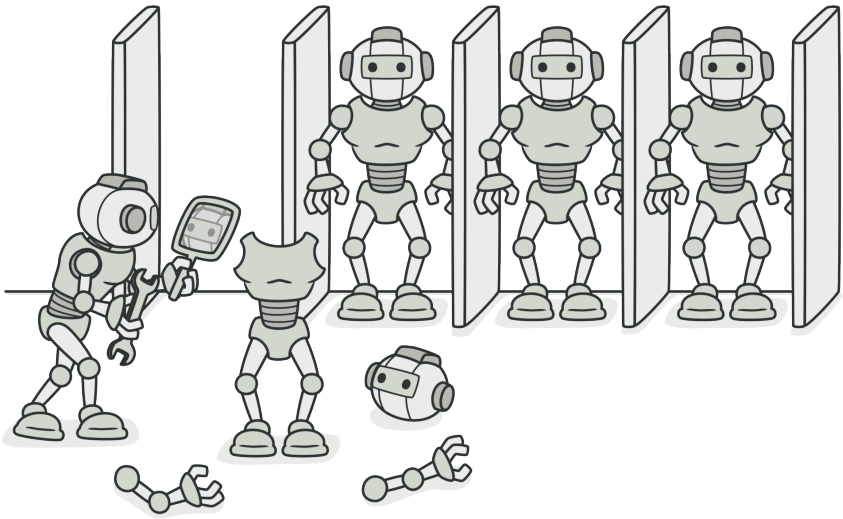
- Muitos projetos começam usando o **Factory Method** (menos complicado e mais customizável através de subclasses) e evoluem para o **Abstract Factory**, **Prototype**, ou **Builder** (mais flexíveis, mas mais complicados).

- O **Builder** foca em construir objetos complexos passo a passo. O **Abstract Factory** se especializa em criar famílias de objetos relacionados. O *Abstract Factory* retorna o produto imediatamente, enquanto que o *Builder* permite que você execute algumas etapas de construção antes de buscar o produto.

- Você pode usar o **Builder** quando criar árvores **Composite** complexas porque você pode programar suas etapas de construção para trabalhar recursivamente.

- Você pode combinar o **Builder** com o **Bridge**: a classe *diretor* tem um papel de abstração, enquanto que diferentes *construtores* agem como *implementações*.

- As **Fábricas Abstratas**, **Construtores**, e **Protótipos** podem todos ser implementados como **Singletons**.



PROTOTYPE

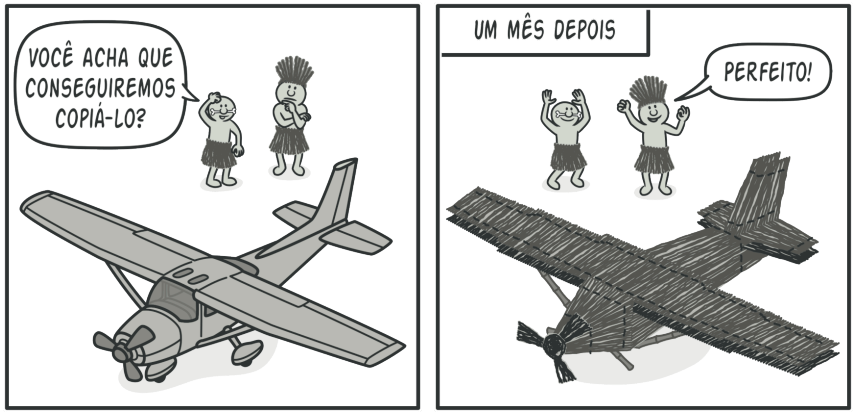
Também conhecido como: Protótipo, Clone

O **Prototype** é um padrão de projeto criacional que permite copiar objetos existentes sem fazer seu código ficar dependente de suas classes.

☹ Problema

Digamos que você tenha um objeto, e você quer criar uma cópia exata dele. Como você o faria? Primeiro, você tem que criar um novo objeto da mesma classe. Então você terá que ir por todos os campos do objeto original e copiar seus valores para o novo objeto.

Legal! Mas tem uma pegadinha. Nem todos os objetos podem ser copiados dessa forma porque alguns campos de objeto podem ser privados e não serão visíveis fora do próprio objeto.



*Copiando um objeto "do lado de fora" **nem sempre** é possível.*

Há ainda mais um problema com a abordagem direta. Uma vez que você precisa saber a classe do objeto para criar uma cópia, seu código se torna dependente daquela classe. Se a dependência adicional não te assusta, tem ainda outra pegadinha. Algumas vezes você só sabe a interface que o objeto segue, mas não sua classe concreta, quando, por exemplo, um parâ-

metro em um método aceita quaisquer objetos que seguem uma interface.

😊 Solução

O padrão Prototype delega o processo de clonagem para o próprio objeto que está sendo clonado. O padrão declara um interface comum para todos os objetos que suportam clonagem. Essa interface permite que você clone um objeto sem acoplar seu código à classe daquele objeto. Geralmente, tal interface contém apenas um único método `clonar`.

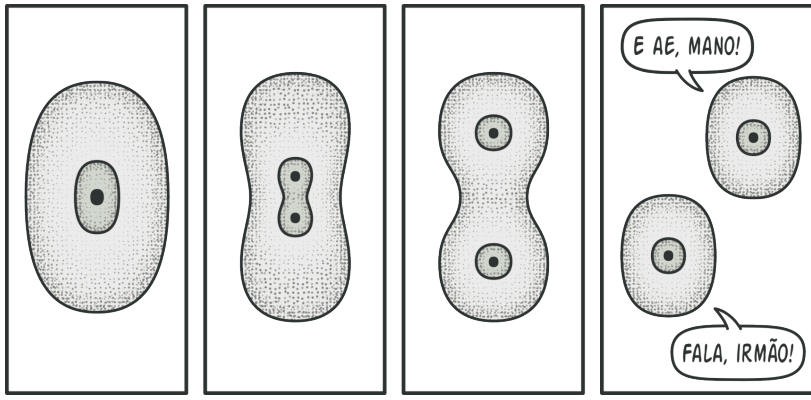
A implementação do método `clonar` é muito parecida em todas as classes. O método cria um objeto da classe atual e carrega todos os valores de campo para do antigo objeto para o novo. Você pode até mesmo copiar campos privados porque a maioria das linguagens de programação permite objetos acessar campos privados de outros objetos que pertençam a mesma classe.

Um objeto que suporta clonagem é chamado de um *protótipo*. Quando seus objetos têm dúzias de campos e centenas de possíveis configurações, cloná-los pode servir como uma alternativa à subclasses.

Pré construir protótipos pode ser uma alternativa às subclasses.

Funciona assim: você cria um conjunto de objetos, configurados de diversas formas. Quando você precisa um objeto parecido com o que você configurou, você apenas clona um protótipo ao invés de construir um novo objeto a partir do nada.

Analogia com o mundo real



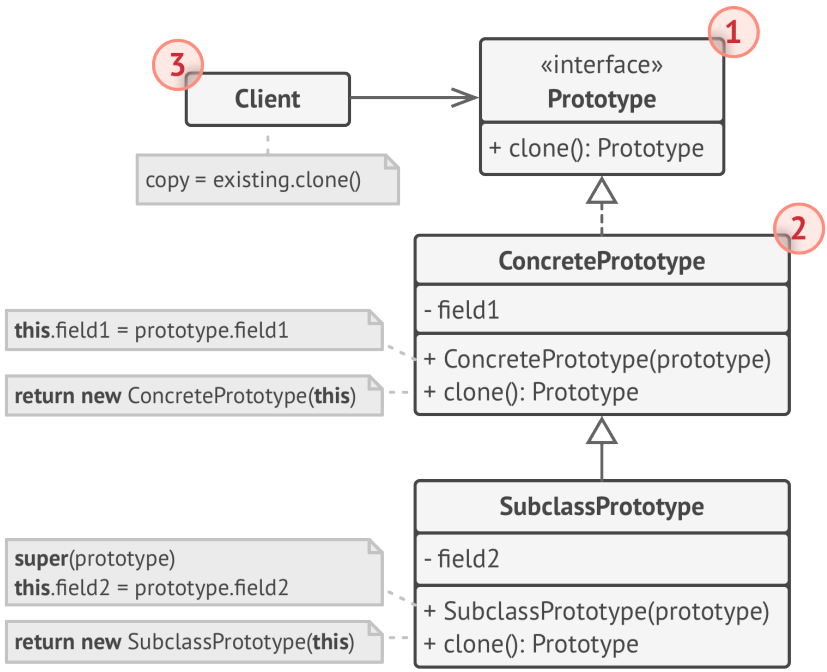
A divisão de uma célula.

Na vida real, os protótipos são usados para fazer diversos testes antes de se começar uma produção em massa de um produto. Contudo, nesse caso, os protótipos não participam de qualquer produção, ao invés disso fazem um papel passivo.

Já que protótipos industriais não se copiam por conta própria, uma analogia ao padrão é o processo de divisão celular chamado mitose (biologia, lembra?). Após a divisão mitótica, um par de células idênticas são formadas. A célula original age como um protótipo e tem um papel ativo na criação da cópia.

Estrutura

Implementação básica

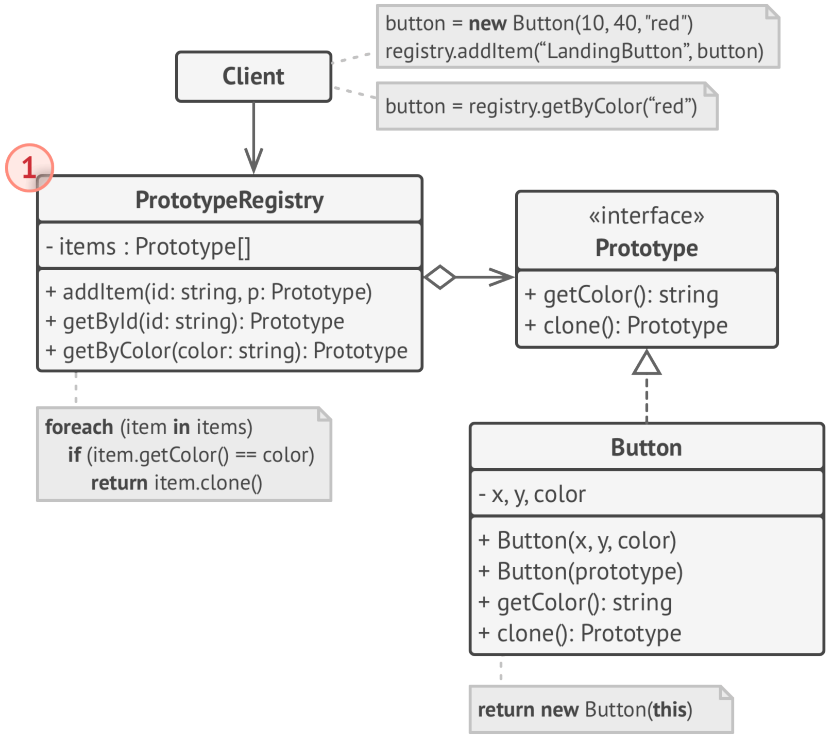


1. A interface **Protótipo** declara os métodos de clonagem. Na maioria dos casos é apenas um método `clonar`.

2. A classe **Protótipo Concreta** implementa o método de clonagem. Além de copiar os dados do objeto original para o clone, esse método também pode lidar com alguns casos específicos do processo de clonagem relacionados a clonar objetos ligados, desfazendo dependências recursivas, etc.

3. O **Cliente** pode produzir uma cópia de qualquer objeto que segue a interface do protótipo.

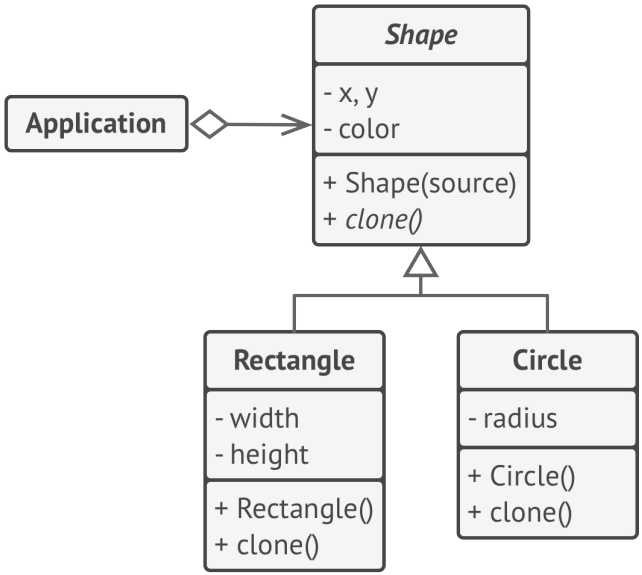
Implementação do registro do protótipo



1. O **Registro do Protótipo** fornece uma maneira fácil de acessar protótipos de uso frequente. Ele salva um conjunto de objetos pré construídos que estão prontos para serem copiados. O registro de protótipo mais simples é um hashmap `nome → protótipo`. Contudo, se você precisa de um melhor critério de busca que apenas um nome simples, você pode construir uma versão muito mais robusta do registro.

Pseudocódigo

Neste exemplo, o padrão **Prototype** permite que você produza cópias exatas de objetos geométricos, sem acoplamento com o código das classes deles.



Clonando um conjunto de objetos que pertencem à uma hierarquia de classe.

Todas as classes de formas seguem a mesma interface, que fornece um método de clonagem. Uma subclasse pode chamar o método de clonagem superior antes de copiar seus próprios valores de campo para o objeto resultante.

```
1  // Protótipo base.
2  abstract class Shape is
```

```
    field X: int
    field Y: int
    field color: string

    // Um construtor normal.
    constructor Shape() is
      // ...

    // O construtor do protótipo. Um objeto novo é inicializado
    // com valores do objeto existente.
    constructor Shape(source: Shape) is
      this()
      this.X = source.X
      this.Y = source.Y
      this.color = source.color

    // A operação de clonagem retorna uma das subclasses Shape.
    abstract method clone():Shape


// Protótipo concreto. O método de clonagem cria um novo objeto
// e passa ele ao construtor. Até o construtor terminar, ele tem
// uma referência ao clone fresco. Portanto, ninguém tem acesso
// ao clone parcialmente construído. Isso faz com que o clone
// resultante seja consistente.
class Rectangle extends Shape is
    field width: int
    field height: int

    constructor Rectangle(source: Rectangle) is
      // Uma chamada para o construtor pai é necessária para
      // copiar campos privados definidos na classe pai.
```

```
    super(source)
    this.width = source.width
    this.height = source.height

  method clone():Shape is
    return new Rectangle(this)


class Circle extends Shape is
  field radius: int

  constructor Circle(source: Circle) is
    super(source)
    this.radius = source.radius

  method clone():Shape is
    return new Circle(this)


// Em algum lugar dentro do código cliente.
class Application is
  field shapes: array of Shape

  constructor Application() is
    Circle circle = new Circle()
    circle.X = 10
    circle.Y = 10
    circle.radius = 20
    shapes.add(circle)

    Circle anotherCircle = circle.clone()
    shapes.add(anotherCircle)
```

```
    // A variável `anotherCircle` contém uma cópia exata do
    // objeto `circle`.

    Rectangle rectangle = new Rectangle()
    rectangle.width = 10
    rectangle.height = 20
    shapes.add(rectangle)

  method businessLogic() is
    // O protótipo arrasa porque permite que você produza
    // uma cópia de um objeto sem saber coisa alguma sobre
    // seu tipo.
    Array shapesCopy = new Array of Shapes.

    // Por exemplo, nós não sabemos os elementos exatos no
    // vetor shapes. Tudo que sabemos é que eles são todos
    // shapes. Mas graças ao polimorfismo, quando nós
    // chamamos o método `clone` em um shape, o programa
    // checa sua classe real e executa o método de clonagem
    // apropriado definido naquela classe. É por isso que
    // obtemos clones apropriados ao invés de um conjunto de
    // objetos Shape simples.
    foreach (s in shapes) do
      shapesCopy.add(s.clone())

    // O vetor `shapesCopy` contém cópias exatas dos filhos
    // do vetor `shape`.
```

Aplicabilidade

Utilize o padrão Prototype quando seu código não deve depender de classes concretas de objetos que você precisa copiar.

Isso acontece muito quando seu código funciona com objetos passados para você de um código de terceiros através de alguma interface. As classes concretas desses objetos são desconhecidas, e você não pode depender delas mesmo que quisesse.

O padrão Prototype fornece o código cliente com uma interface geral para trabalhar com todos os objetos que suportam clonagem. Essa interface faz o código do cliente ser independente das classes concretas dos objetos que ele clona.

Utilize o padrão quando você precisa reduzir o número de subclasses que somente diferem na forma que inicializam seus respectivos objetos. Alguém pode ter criado essas subclasses para ser capaz de criar objetos com uma configuração específica.

O padrão Prototype permite que você use um conjunto de objetos pré construídos, configurados de diversas formas, como protótipos.

Ao invés de instanciar uma subclasse que coincide com alguma configuração, o cliente pode simplesmente procurar por um protótipo apropriado e cloná-lo.

Como implementar

1. Crie uma interface protótipo e declare o método `clonar` nela. Ou apenas adicione o método para todas as classes de uma hierarquia de classes existente, se você tiver uma.

2. Uma classe protótipo deve definir o construtor alternativo que aceita um objeto daquela classe como um argumento. O construtor deve copiar os valores de todos os campos definidos na classe do objeto passado para a nova instância recém criada. Se você está mudando uma subclasse, você deve chamar o construtor da classe pai para permitir que a superclasse lide com a clonagem de seus campos privados.

   Se a sua linguagem de programação não suporta sobrecarregamento de métodos, você pode definir um método especial para copiar os dados do objeto. O construtor é um local mais conveniente para se fazer isso porque ele entrega o objeto resultante logo depois que você chamar o operador `new`.

3. O método de clonagem geralmente consiste em apenas uma linha: executando um operador `new` com a versão protótipo do construtor. Observe que toda classe deve explicitamente sobrescrever o método de clonagem and usar sua própria classe junto com o operador `new`. Do contrário, o método de clonagem pode produzir um objeto da classe superior.

4. Opcionalmente, crie um registro protótipo centralizado para armazenar um catálogo de protótipos usados com frequência.

Você pode implementar o registro como uma nova classe factory ou colocá-lo na classe protótipo base com um método estático para recuperar o protótipo. Esse método deve procurar por um protótipo baseado em critérios de busca que o código cliente passou para o método. O critério pode ser tanto uma string ou um complexo conjunto de parâmetros de busca. Após o protótipo apropriado ser encontrado, o registro deve clonálo e retornar a cópia para o cliente.

Por fim, substitua as chamadas diretas para os construtores das subclasses com chamadas para o método factory do registro do protótipo.

⚖ Prós e contras

- ✓ Você pode clonar objetos sem acoplá-los a suas classes concretas.
- ✓ Você pode se livrar de códigos de inicialização repetidos em troca de clonar protótipos pré-construídos.
- ✓ Você pode produzir objetos complexos mais convenientemente.
- ✓ Você tem uma alternativa para herança quando lidar com configurações pré determinadas para objetos complexos.

- ✕ Clonar objetos complexos que têm referências circulares pode ser bem complicado.

⇄ 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.

- O **Prototype** pode ajudar quando você precisa salvar cópias de **comandos** no histórico.

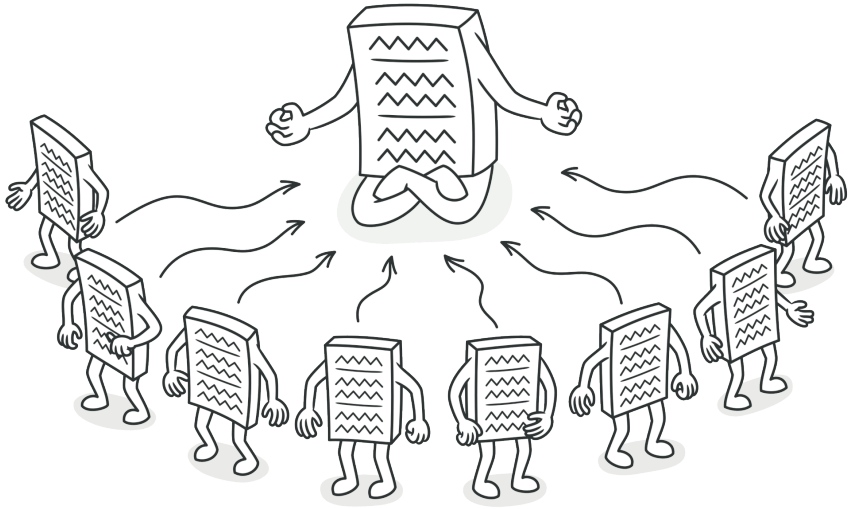
- Projetos que fazem um uso pesado de **Composite** e do **Decorator** podem se beneficiar com frequência do uso do **Prototype**. Aplicando o padrão permite que você clone estruturas complexas ao invés de reconstruí-las do zero.

- 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.

- Algumas vezes o **Prototype** pode ser uma alternativa mais simples a um **Memento**. Isso funciona se o objeto, o estado no qual você quer armazenar na história, é razoavelmente intui-

tivo e não tem ligações para recursos externos, ou as ligações são fáceis de se restabelecer.

- As **Fábricas Abstratas**, **Construtores**, e **Protótipos** podem todos ser implementados como **Singletons**.



SINGLETON

Também conhecido como: Carta única

O **Singleton** é um padrão de projeto criacional que permite a você garantir que uma classe tenha apenas uma instância, enquanto provê um ponto de acesso global para essa instância.

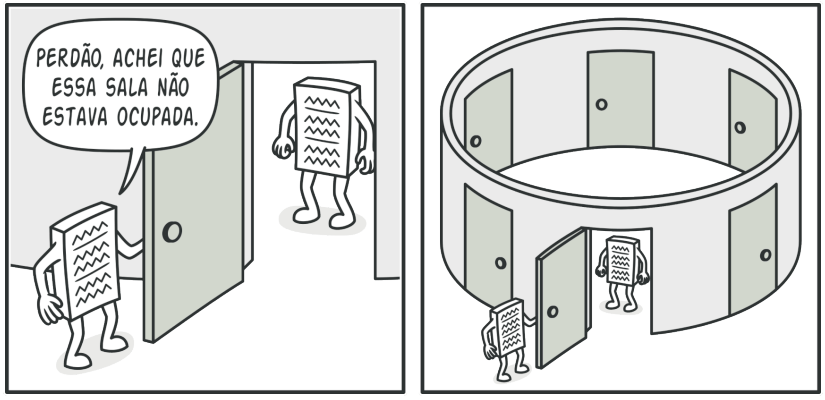
☹ Problema

O padrão Singleton resolve dois problemas de uma só vez, violando o *princípio de responsabilidade única*:

1. **Garantir que uma classe tenha apenas uma única instância**. Por que alguém iria querer controlar quantas instâncias uma classe tem? A razão mais comum para isso é para controlar o acesso a algum recurso compartilhado—por exemplo, uma base de dados ou um arquivo.

   Funciona assim: imagine que você criou um objeto, mas depois de um tempo você decidiu criar um novo. Ao invés de receber um objeto fresco, você obterá um que já foi criado.

   Observe que esse comportamento é impossível implementar com um construtor regular uma vez que uma chamada do construtor **deve** sempre retornar um novo objeto por design.



Clientes podem não se dar conta que estão lidando com o mesmo objeto a todo momento.

2. **Fornece um ponto de acesso global para aquela instância**. Se lembra daquelas variáveis globais que você (tá bom, eu) usou para guardar alguns objetos essenciais? Embora sejam muito úteis, elas também são muito inseguras uma vez que qualquer código pode potencialmente sobrescrever os conteúdos daquelas variáveis e quebrar a aplicação.

   Assim como uma variável global, o padrão Singleton permite que você acesse algum objeto de qualquer lugar no programa. Contudo, ele também protege aquela instância de ser sobrescrita por outro código.

   Há outro lado para esse problema: você não quer que o código que resolve o problema #1 fique espalhado por todo seu programa. É muito melhor tê-lo dentro de uma classe, especialmente se o resto do seu código já depende dela.

   Hoje em dia, o padrão Singleton se tornou tão popular que as pessoas podem chamar algo de *singleton* mesmo se ele resolve apenas um dos problemas listados.

😊 Solução

Todas as implementações do Singleton tem esses dois passos em comum:

- Fazer o construtor padrão privado, para prevenir que outros objetos usem o operador `new` com a classe singleton.

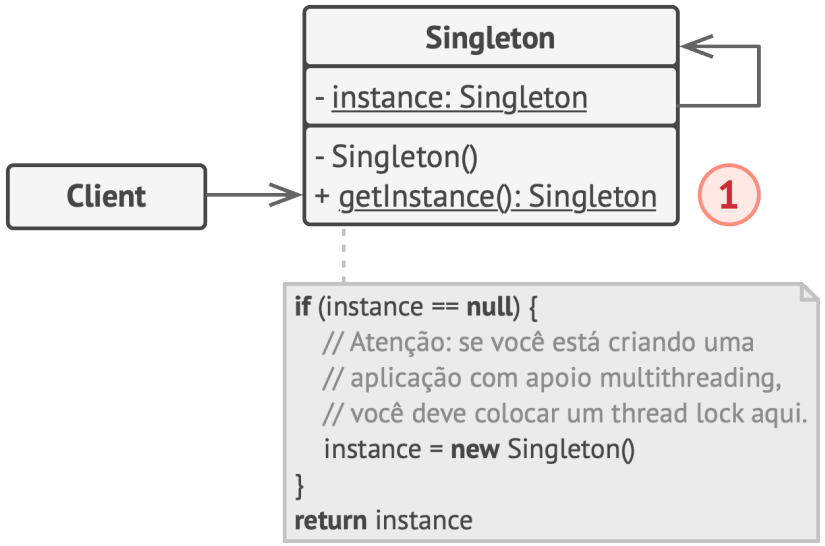
- Criar um método estático de criação que age como um construtor. Esse método chama o construtor privado por debaixo dos panos para criar um objeto e o salva em um campo estático. Todas as chamadas seguintes para esse método retornam o objeto em cache.

  Se o seu código tem acesso à classe singleton, então ele será capaz de chamar o método estático da singleton. Então sempre que aquele método é chamado, o mesmo objeto é retornado.

Analogia com o mundo real

O governo é um excelente exemplo de um padrão Singleton. Um país pode ter apenas um governo oficial. Independentemente das identidades pessoais dos indivíduos que formam governos, o título, "O Governo de X", é um ponto de acesso global que identifica o grupo de pessoas no command.

Estrutura



1. A classe **Singleton** declara o método estático `getInstance` que retorna a mesma instância de sua própria classe.

   O construtor da singleton deve ser escondido do código cliente. Chamando o método `getInstance` deve ser o único modo de obter o objeto singleton.

Pseudocódigo

Neste exemplo, a classe de conexão com a base de dados age como um **Singleton**. Essa classe não tem um construtor público, então a única maneira de obter seu objeto é chamando o método `getInstance`. Esse método coloca o

primeiro objeto criado em cache e o retorna em todas as chamadas subsequentes.

```
// A classe Database define o método `getInstance` que permite
// clientes acessar a mesma instância de uma conexão a base de
// dados através do programa.
class Database is
  // O campo para armazenar a instância singleton deve ser
  // declarado como estático.
  private static field instance: Database

  // O construtor do singleton devem sempre ser privado para
  // prevenir chamadas diretas de construção com o operador
  // `new`.
  private constructor Database() is
    // Algum código de inicialização, tal como uma conexão
    // com um servidor de base de dados.
    // ...

  // O método estático que controla acesso à instância do
  // singleton
  public static method getInstance() is
    if (Database.instance == null) then
      acquireThreadLock() and then
        // Certifique que a instância ainda não foi
        // inicializada por outra thread enquanto está
        // estiver esperando pela liberação do `lock`.
        if (Database.instance == null) then
          Database.instance = new Database()
    return Database.instance

```

```
  // Finalmente, qualquer singleton deve definir alguma lógica
  // de negócio que deve ser executada em sua instância.
  public method query(sql) is
    // Por exemplo, todas as solicitações à base de dados de
    // uma aplicação passam por esse método. Portanto, você
    // pode colocar a lógica de throttling ou cache aqui.
    // ...

class Application is
  method main() is
    Database foo = Database.getInstance()
    foo.query("SELECT ...")
    // ...
    Database bar = Database.getInstance()
    bar.query("SELECT ...")
    // A variável `bar` vai conter o mesmo objeto que a
    // variável `foo`.
```

Aplicabilidade

Utilize o padrão Singleton quando uma classe em seu programa deve ter apenas uma instância disponível para todos seus clientes; por exemplo, um objeto de base de dados único compartilhado por diferentes partes do programa.

O padrão Singleton desabilita todos os outros meios de criar objetos de uma classe exceto pelo método especial de criação. Esse método tanto cria um novo objeto ou retorna um objeto existente se ele já tenha sido criado.

Utilize o padrão Singleton quando você precisa de um controle mais estrito sobre as variáveis globais.

Ao contrário das variáveis globais, o padrão Singleton garante que há apenas uma instância de uma classe. Nada, a não ser a própria classe singleton, pode substituir a instância salva em cache.

Observe que você sempre pode ajustar essa limitação e permitir a criação de qualquer número de instâncias singleton. O único pedaço de código que requer mudanças é o corpo do método `getInstance`.

Como implementar

1. Adicione um campo privado estático na classe para o armazenamento da instância singleton.

2. Declare um método de criação público estático para obter a instância singleton.

3. Implemente a "inicialização preguiçosa" dentro do método estático. Ela deve criar um novo objeto na sua primeira chamada e colocá-lo no campo estático. O método deve sempre retornar aquela instância em todas as chamadas subsequentes.

4. Faça o construtor da classe ser privado. O método estático da classe vai ainda ser capaz de chamar o construtor, mas não os demais objetos.

5. Vá para o código cliente e substitua todas as chamadas diretas para o construtor do singleton com chamadas para seu método de criação estático.

Prós e contras

✓ Você pode ter certeza que uma classe só terá uma única instância.

✓ Você ganha um ponto de acesso global para aquela instância.

✓ O objeto singleton é inicializado somente quando for pedido pela primeira vez.

✗ Viola o *princípio de responsabilidade única*. O padrão resolve dois problemas de uma só vez.

✗ O padrão Singleton pode mascarar um design ruim, por exemplo, quando os componentes do programa sabem muito sobre cada um.

✗ O padrão requer tratamento especial em um ambiente multithreaded para que múltiplas threads não possam criar um objeto singleton várias vezes.

✗ Pode ser difícil realizar testes unitários do código cliente do Singleton porque muitos frameworks de teste dependem de herança quando produzem objetos simulados. Já que o construtor da classe singleton é privado e sobrescrever métodos estáticos é impossível na maioria das linguagem, você terá que pensar em uma maneira criativa de simular o singleton. Ou apenas não escreva os testes. Ou não use o padrão Singleton.

⇄ Relações com outros padrões

- Uma classe **fachada** pode frequentemente ser transformada em uma **singleton** já que um único objeto fachada é suficiente na maioria dos casos.

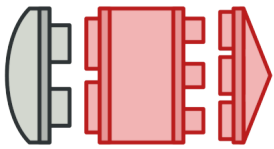
- O **Flyweight** seria parecido com o **Singleton** se você, de algum modo, reduzisse todos os estados de objetos compartilhados para apenas um objeto flyweight. Mas há duas mudanças fundamentais entre esses padrões:

    1. Deve haver apenas uma única instância singleton, enquanto que uma classe *flyweight* pode ter múltiplas instâncias com diferentes estados intrínsecos.
    2. O objeto *singleton* pode ser mutável. Objetos flyweight são imutáveis.

- As **Fábricas Abstratas**, **Construtores**, e **Protótipos** podem todos ser implementados como **Singletons**.

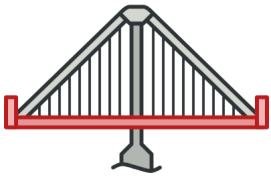
Padrões de projeto estruturais

Os padrões estruturais explicam como montar objetos e classes em estruturas maiores mas ainda mantendo essas estruturas flexíveis e eficientes.



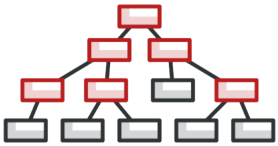
Adapter

Permite a colaboração de objetos de interfaces incompatíveis.



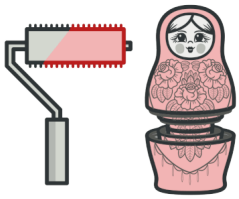
Bridge

Permite que você divida uma classe grande ou um conjunto de classes intimamente ligadas em duas hierarquias separadas—abstração e implementação—que podem ser desenvolvidas independentemente umas das outras.



Composite

Permite que você componha objetos em estrutura de árvores e então trabalhe com essas estruturas como se fossem objetos individuais.

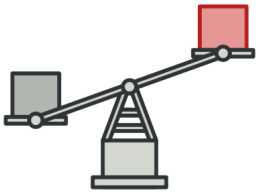


Decorator

Permite que você adicione novos comportamentos a objetos colocando eles dentro de um envoltório (wrapper) de objetos que contém os comportamentos.

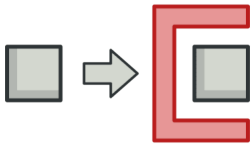
Facade

Fornece uma interface simplificada para uma biblioteca, um framework, ou qualquer outro conjunto complexo de classes.



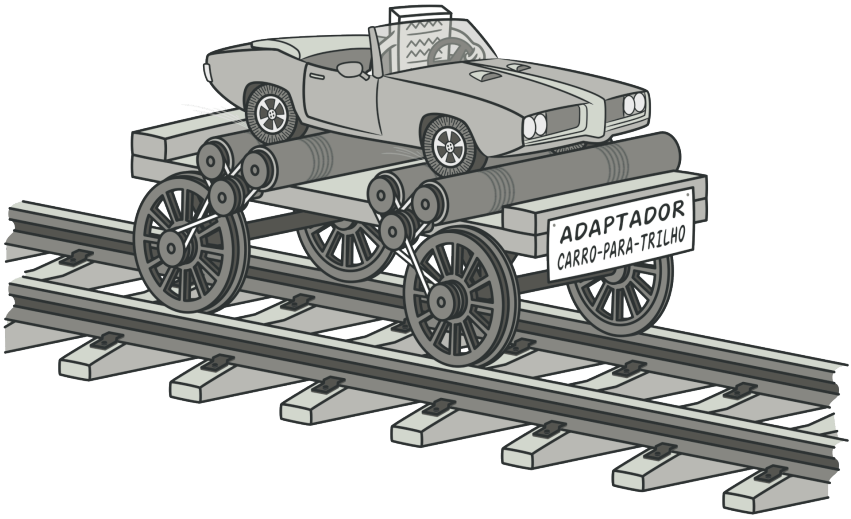
Flyweight

Permite que você coloque mais objetos na quantidade disponível de RAM ao compartilhar partes do estado entre múltiplos objetos ao invés de manter todos os dados em cada objeto.



Proxy

Permite que você forneça um substituto ou um espaço reservado para outro objeto. Um proxy controla o acesso ao objeto original, permitindo que você faça algo ou antes ou depois do pedido chegar ao objeto original.



ADAPTER

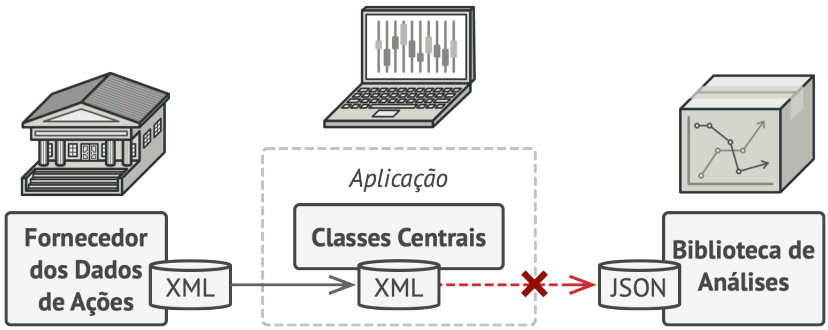
Também conhecido como: Adaptador, Wrapper

O **Adapter** é um padrão de projeto estrutural que permite objetos com interfaces incompatíveis colaborarem entre si.

☹ Problema

Imagine que você está criando uma aplicação de monitoramento do mercado de ações da bolsa. A aplicação baixa os dados as ações de múltiplas fontes em formato XML e então mostra gráficos e diagramas maneiros para o usuário.

Em algum ponto, você decide melhorar a aplicação ao integrar uma biblioteca de análise de terceiros. Mas aqui está a pegadinha: a biblioteca só trabalha com dados em formato JSON.



Você não pode usar a biblioteca "como ela está" porque ela espera os dados em um formato que é incompatível com sua aplicação.

Você poderia mudar a biblioteca para que ela funcione com XML. Contudo, isso pode quebrar algum código existente que depende da biblioteca. E pior, você pode não ter acesso ao código fonte da biblioteca para começo de conversa, fazendo dessa abordagem uma tarefa impossível.

😊 Solução

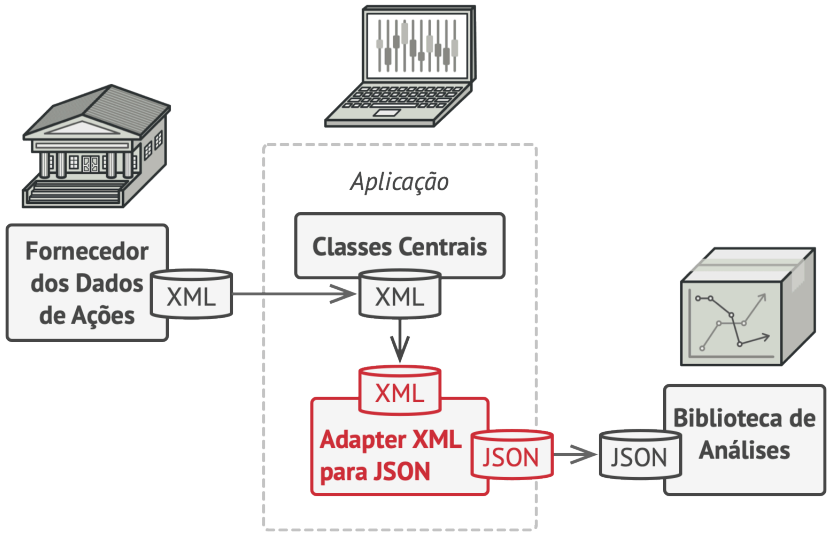
Você pode criar um *adaptador*. Ele é um objeto especial que converte a interface de um objeto para que outro objeto possa entendê-lo.

Um adaptador encobre um dos objetos para esconder a complexidade da conversão acontecendo nos bastidores. O objeto encobrido nem fica ciente do adaptador. Por exemplo, você pode encobrir um objeto que opera em metros e quilômetros com um adaptador que converte todos os dados para unidades imperiais tais como pés e milhas.

Adaptadores podem não só converter dados em vários formatos, mas também podem ajudar objetos com diferentes interfaces a colaborar. Veja aqui como funciona:

1. O adaptador obtém uma interface, compatível com um dos objetos existentes.
2. Usando essa interface, o objeto existente pode chamar os métodos do adaptador com segurança.
3. Ao receber a chamada, o adaptador passa o pedido para o segundo objeto, mas em um formato e ordem que o segundo objeto espera.

Algumas vezes é possível criar um adaptador de duas vias que pode converter as chamadas em ambas as direções.



Vamos voltar à nossa aplicação da bolsa de valores. Para resolver o dilema dos formatos incompatíveis, você pode criar adaptadores XML-para-JSON para cada classe da biblioteca de análise que seu código trabalha diretamente. Então você ajusta seu código para comunicar-se com a biblioteca através desses adaptadores. Quando um adaptador recebe uma chamada, ele traduz os dados entrantes XML em uma estrutura JSON e passa a chamada para os métodos apropriados de um objeto de análise encoberto.

Analogia com o mundo real

Quando você viaja do Brasil para a Europa pela primeira vez, você pode ter uma pequena surpresa quando tenta carregar seu laptop. O plugue e os padrões de tomadas são diferentes em diferentes países.

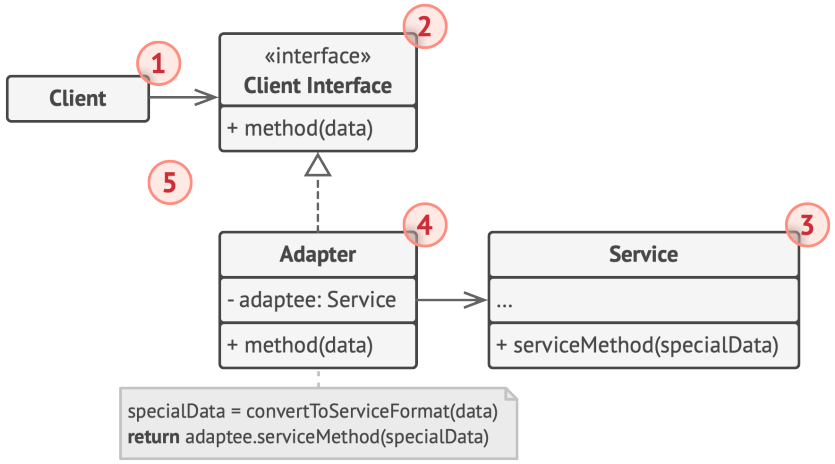
Uma mala antes e depois de uma viagem ao exterior.

É por isso que seu plugue do Brasil não vai caber em uma tomada da Alemanha. O problema pode ser resolvido usando um adaptador de tomada que tenha o estilo de tomada Brasileira e o plugue no estilo Europeu.

Estrutura

Adaptador de objeto

Essa implementação usa o princípio de composição do objeto: o adaptador implementa a interface de um objeto e encobre o outro. Ele pode ser implementado em todas as linguagens de programação populares.



1. O **Cliente** é uma classe que contém a lógica de negócio do programa existente.

2. A **Interface do Cliente** descreve um protocolo que outras classes devem seguir para ser capaz de colaborar com o código cliente.

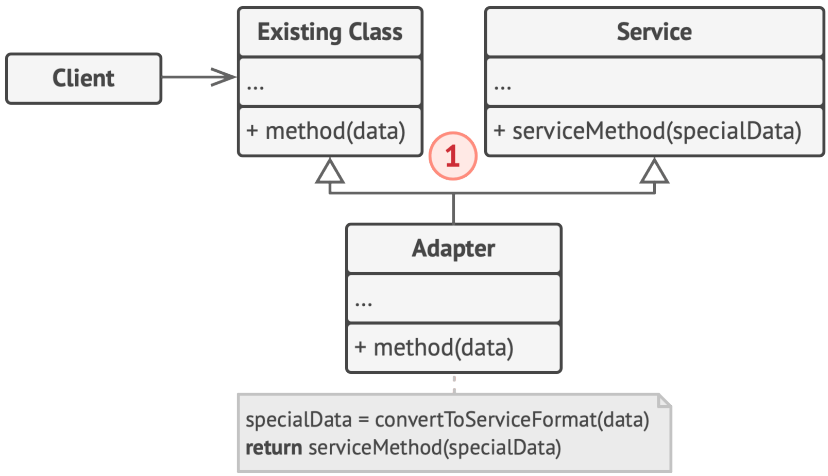
3. O **Serviço** é alguma classe útil (geralmente de terceiros ou código legado). O cliente não pode usar essa classe diretamente porque ela tem uma interface incompatível.

4. O **Adaptador** é uma classe que é capaz de trabalhar tanto com o cliente quanto o serviço: ela implementa a interface do cliente enquanto encobre o objeto do serviço. O adaptador recebe chamadas do cliente através da interface do adaptador e as traduz em chamadas para o objeto encobrido do serviço em um formato que ele possa entender.

5. O código cliente não é acoplado à classe concreta do adaptador desde que ele trabalhe com o adaptador através da interface do cliente. Graças a isso, você pode introduzir novos tipos de adaptadores no programa sem quebrar o código cliente existente. Isso pode ser útil quando a interface de uma classe de serviço é mudada ou substituída: você pode apenas criar uma nova classe adaptador sem mudar o código cliente.

Adaptador de classe

Essa implementação utiliza herança: o adaptador herda interfaces de ambos os objetos ao mesmo tempo. Observe que essa abordagem só pode ser implementada em linguagens de programação que suportam herança múltipla, tais como C++.

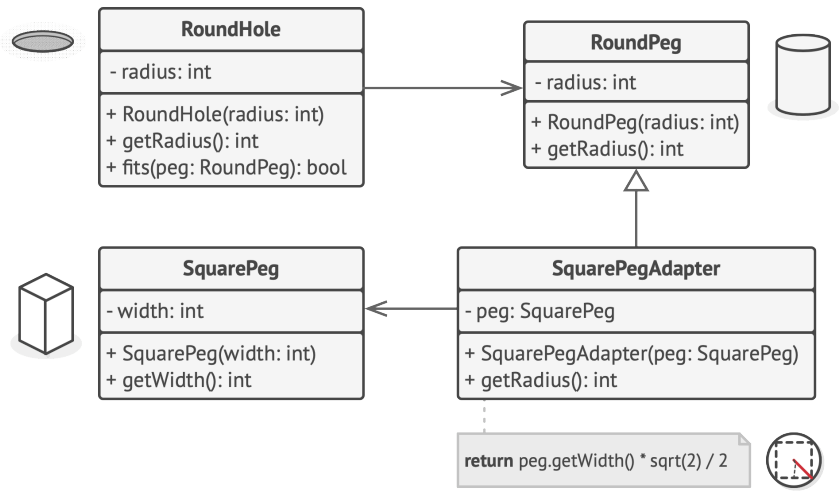


1. A **Classe Adaptador** não precisa encobrir quaisquer objetos porque ela herda os comportamentos tanto do cliente como do serviço. A adaptação acontece dentro dos métodos sobrescri-

tos. O adaptador resultante pode ser usado em lugar de uma classe cliente existente.

Pseudocódigo

Esse exemplo do padrão **Adapter** é baseado no conflito clássico entre pinos quadrados e buracos redondos.



Adaptando pinos quadrados para buracos redondos.

O adaptador finge ser um pino redondo, com um raio igual a metade do diâmetro do quadrado (em outras palavras, o raio do menor círculo que pode acomodar o pino quadrado).

```
1  // Digamos que você tenha duas classes com interfaces
2  // compatíveis: RoundHole (Buraco Redondo) e RoundPeg (Pino
3  // Redondo).
4  class RoundHole is
```

```
  constructor RoundHole(radius) { ... }

  method getRadius() is
    // Retorna o raio do buraco.

  method fits(peg: RoundPeg) is
    return this.getRadius() >= peg.getRadius()

class RoundPeg is
  constructor RoundPeg(radius) { ... }

  method getRadius() is
    // Retorna o raio do pino.


// Mas tem uma classe incompatível: SquarePeg (Pino Quadrado).
class SquarePeg is
  constructor SquarePeg(width) { ... }

  method getWidth() is
    // Retorna a largura do pino quadrado.


// Uma classe adaptadora permite que você encaixe pinos
// quadrados em buracos redondos. Ela estende a classe RoundPeg
// para permitir que objetos do adaptador ajam como pinos
// redondos.
class SquarePegAdapter extends RoundPeg is
  // Na verdade, o adaptador contém uma instância da classe
  // SquarePeg.
  private field peg: SquarePeg

```

```
  constructor SquarePegAdapter(peg: SquarePeg) is
    this.peg = peg

  method getRadius() is
    // O adaptador finge que é um pino redondo com um raio
    // que encaixaria o pino quadrado que o adaptador está
    // envolvendo.
    return peg.getWidth() * Math.sqrt(2) / 2


// Em algum lugar no código cliente.
hole = new RoundHole(5)
rpeg = new RoundPeg(5)
hole.fits(rpeg) // true

small_sqpeg = new SquarePeg(5)
large_sqpeg = new SquarePeg(10)
// Isso não vai compilar (tipos incompatíveis).
hole.fits(small_sqpeg)

small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
hole.fits(small_sqpeg_adapter) // true
hole.fits(large_sqpeg_adapter) // false
```

Aplicabilidade

Utilize a classe Adaptador quando você quer usar uma classe existente, mas sua interface não for compatível com o resto do seu código.

O padrão Adapter permite que você crie uma classe de meio termo que serve como um tradutor entre seu código e a classe antiga, uma classe de terceiros, ou qualquer outra classe com uma interface estranha.

Utilize o padrão quando você quer reutilizar diversas subclasses existentes que não possuam alguma funcionalidade comum que não pode ser adicionada a superclasse.

Você pode estender cada subclasse e colocar a funcionalidade faltante nas novas classes filhas. Contudo, você terá que duplicar o código em todas as novas classes, o que **cheira muito mal**.

Uma solução muito mais elegante seria colocar a funcionalidade faltante dentro da classe adaptadora. Então você encobriria os objetos com as funcionalidades faltantes dentro do adaptador, ganhando tais funcionalidades de forma dinâmica. Para isso funcionar, as classes alvo devem ter uma interface em comum, e o campo do adaptador deve seguir aquela interface. Essa abordagem se parece muito com o padrão **Decorator**.

Como implementar

1. Certifique-se que você tem ao menos duas classes com interfaces incompatíveis:

- Uma classe *serviço* útil, que você não pode modificar (quase sempre de terceiros, antiga, ou com muitas dependências existentes).
- Uma ou mais classes *cliente* que seriam beneficiadas com o uso da classe serviço.

2. Declare a interface cliente e descreva como o cliente se comunica com o serviço.

3. Cria a classe adaptadora e faça-a seguir a interface cliente. Deixe todos os métodos vazios por enquanto.

4. Adicione um campo para a classe do adaptador armazenar uma referência ao objeto do serviço. A prática comum é inicializar esse campo via o construtor, mas algumas vezes é mais conveniente passá-lo para o adaptador ao chamar seus métodos.

5. Um por um, implemente todos os métodos da interface cliente na classe adaptadora. O adaptador deve delegar a maioria do trabalho real para o objeto serviço, lidando apenas com a conversão da interface ou formato dos dados.

6. Os Clientes devem usar o adaptador através da interface cliente. Isso irá permitir que você mude ou estenda o adaptador sem afetar o código cliente.

Prós e contras

✓ *Princípio de responsabilidade única*. Você pode separar a conversão de interface ou de dados da lógica primária do negócio do programa.

✓ *Princípio aberto/fechado*. Você pode introduzir novos tipos de adaptadores no programa sem quebrar o código cliente existente, desde que eles trabalhem com os adaptadores através da interface cliente.

✕ A complexidade geral do código aumenta porque você precisa introduzir um conjunto de novas interfaces e classes. Algumas vezes é mais simples mudar a classe serviço para que ela se adeque com o resto do seu código.

Relações com outros padrões

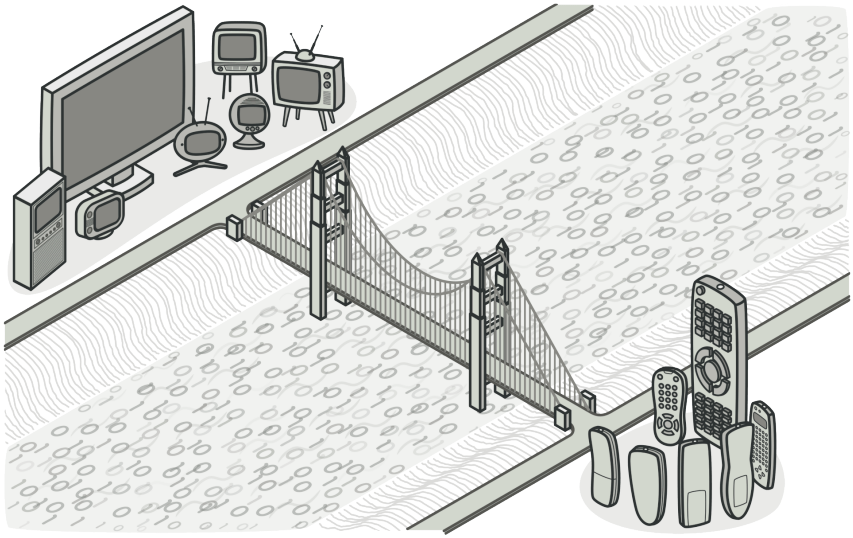
- O **Bridge** é geralmente definido com antecedência, permitindo que você desenvolva partes de uma aplicação independentemente umas das outras. Por outro lado, o **Adapter** é comumente usado em aplicações existentes para fazer com que classes incompatíveis trabalhem bem juntas.

- O **Adapter** muda a interface de um objeto existente, enquanto que o **Decorator** melhora um objeto sem mudar sua interface. Além disso, o *Decorator* suporta composição recursiva, o que não seria possível quando você usa o *Adapter*.

- O **Adapter** fornece uma interface diferente para um objeto encapsulado, o **Proxy** fornece a ele a mesma interface, e o **Decorator** fornece a ele com uma interface melhorada.

- O **Facade** define uma nova interface para objetos existentes, enquanto que o **Adapter** tenta fazer uma interface existente ser utilizável. O *Adapter* geralmente envolve apenas um objeto, enquanto que o *Facade* trabalha com um inteiro subsistema de objetos.

- O **Bridge**, **State**, **Strategy** (e de certa forma o **Adapter**) têm estruturas muito parecidas. De fato, todos esses padrões estão baseados em composição, o que é delegar o trabalho para outros objetos. Contudo, eles todos resolvem problemas diferentes. Um padrão não é apenas uma receita para estruturar seu código de uma maneira específica. Ele também pode comunicar a outros desenvolvedores o problema que o padrão resolve.



BRIDGE

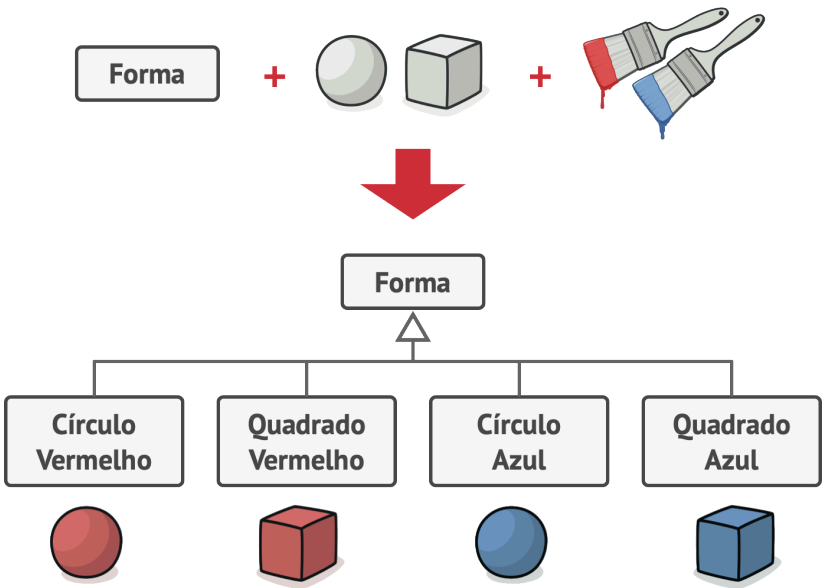
Também conhecido como: Ponte

O **Bridge** é um padrão de projeto estrutural que permite que você divida uma classe grande ou um conjunto de classes intimamente ligadas em duas hierarquias separadas—abstração e implementação—que podem ser desenvolvidas independentemente umas das outras.

☹ Problema

Abstração? Implementação? Soam um pouco assustadoras? Fique calmo que vamos considerar um exemplo simples.

Digamos que você tem uma classe `Forma` geométrica com um par de subclasses: `Círculo` e `Quadrado`. Você quer estender essa hierarquia de classe para incorporar cores, então você planeja criar as subclasses de forma `Vermelho` e `Azul`. Contudo, já que você já tem duas subclasses, você precisa criar quatro combinações de classe tais como `CírculoAzul` e `QuadradoVermelho`.



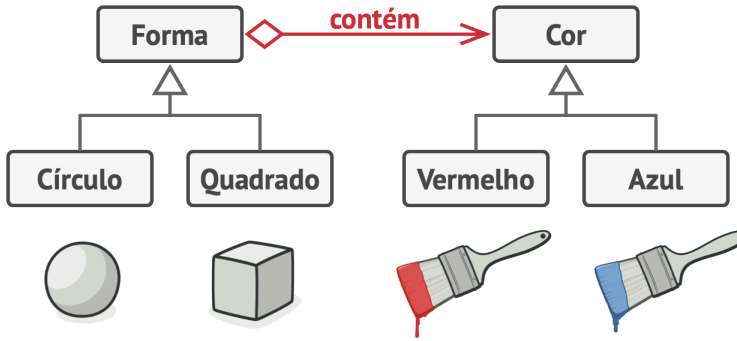
O número de combinações de classes cresce em progressão geométrica.

Adicionar novos tipos de forma e cores à hierarquia irá fazê-la crescer exponencialmente. Por exemplo, para adicionar uma forma de triângulo você vai precisar introduzir duas subclasses, uma para cada cor. E depois disso, adicionando uma nova cor será necessário três subclasses, uma para cada tipo de forma. Quanto mais longe vamos, pior isso fica.

😊 Solução

Esse problema ocorre porque estamos tentando estender as classes de forma em duas dimensões diferentes: por forma e por cor. Isso é um problema muito comum com herança de classe.

O padrão Bridge tenta resolver esse problema ao trocar de herança para composição do objeto. Isso significa que você extrai uma das dimensões em uma hierarquia de classe separada, para que as classes originais referenciem um objeto da nova hierarquia, ao invés de ter todos os seus estados e comportamentos dentro de uma classe.



Você pode prevenir a explosão de uma hierarquia de classe ao transformá-la em diversas hierarquias relacionadas.

Seguindo essa abordagem nós podemos extrair o código relacionado à cor em sua própria classe com duas subclasses: `Vermelho` e `Azul`. A classe `Forma` então ganha um campo de referência apontando para um dos objetos de cor. Agora a forma pode delegar qualquer trabalho referente a cor para o objeto ligado a cor. Aquela referência vai agir como uma ponte entre as classes `Forma` e `Cor`. De agora em diante, para adicionar novas cores não será necessário mudar a hierarquia da forma e vice versa.

Abstração e implementação

O livro GoF [1] introduz os termos *Abstração* e *Implementação* como parte da definição do Bridge. Na minha opinião, os termos soam muito acadêmicos e fazem o padrão parecer algo

1. "Gang of Four" (Gangue dos Quatro) é um apelido dado aos quatro autores do livro original sobre padrões de projeto: *Padrões de Projeto: Elementos de Software Reutilizáveis Orientados a Objetos* **https://refactoring.guru/pt-br/gof-book**.

mais complicado do que realmente é. Tendo lido o exemplo simples com formas e cores, vamos decifrar o significado por trás das palavras assustadoras do livro GoF.

Abstração (também chamado de *interface*) é uma camada de controle de alto nível para alguma entidade. Essa camada não deve fazer nenhum tipo de trabalho por conta própria. Ela deve delegar o trabalho para a camada de *implementação* (também chamada de *plataforma*).

Observe que não estamos falando sobre *interfaces* ou *classes abstratas* da sua linguagem de programação. São coisas diferentes.

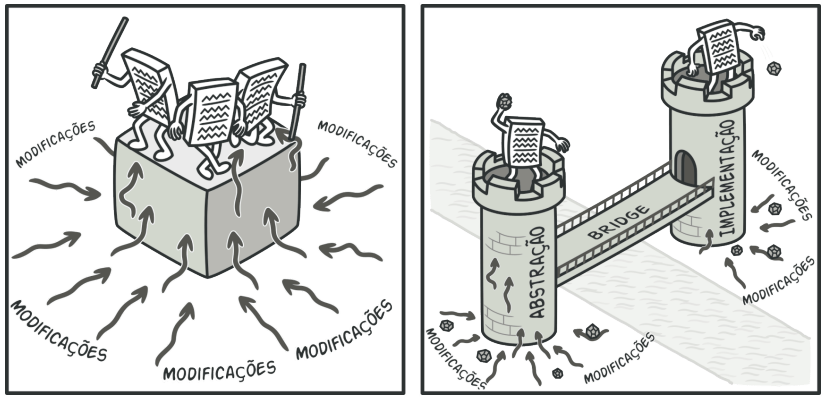
Quando falamos sobre aplicações reais, a abstração pode ser representada por uma interface gráfica de usuário (GUI), e a implementação pode ser o código subjacente do sistema operacional (API) a qual a camada GUI chama em resposta às interações do usuário.

Geralmente falando, você pode estender tal aplicação em duas direções independentes:

- Ter diversas GUIs diferentes (por exemplo, feitas para clientes regulares ou administradores).
- Suportar diversas APIs diferente (por exemplo, para ser capaz de lançar a aplicação no Windows, Linux e macOS).

No pior dos cenários, essa aplicação pode se parecer como uma enorme tigela de espaguete onde centenas de condicionais conectam diferentes tipos de GUI com vários APIs por todo o código.

Você pode trazer ordem para esse caos extraindo o código relacionado a específicas combinações interface-plataforma para classes separadas. Contudo, logo você vai descobrir que existirão *muitas* dessas classes.

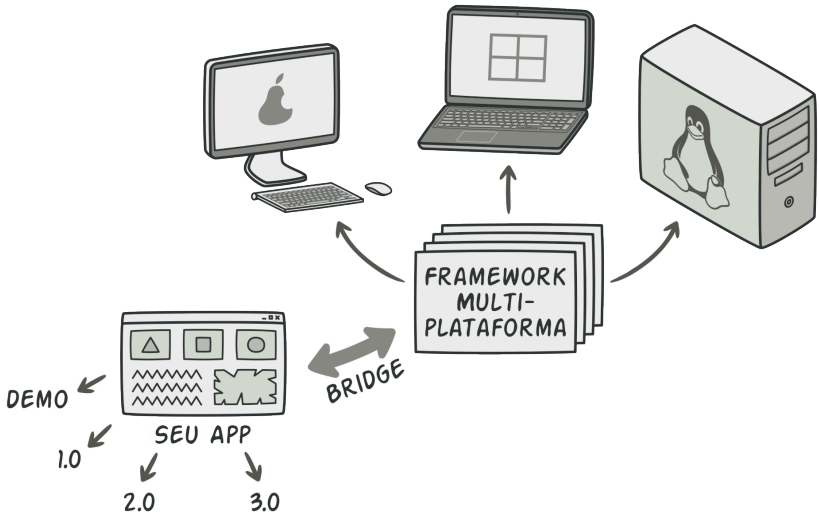


Fazer até mesmo uma única mudança em uma base de códigos monolítica é bastante difícil porque você deve entender a coisa toda muito bem. Fazer mudanças em módulos menores e bem definidos é muito mais fácil.

A hierarquia de classes irá crescer exponencialmente porque adicionando um novo GUI ou suportando um API diferente irá ser necessário criar mais e mais classes.

Vamos tentar resolver esse problema com o padrão Bridge. Ele sugere que as classes sejam divididas em duas hierarquias:

- Abstração: a camada GUI da aplicação.
- Implementação: As APIs do sistema operacional.

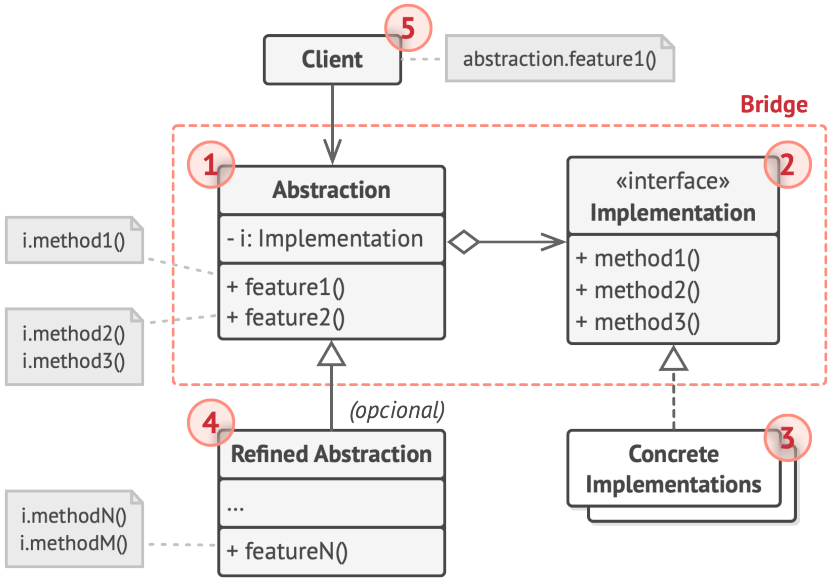


Uma das maneiras de se estruturar uma aplicação multiplataforma.

O objeto da abstração controla a aparência da aplicação, delegando o verdadeiro trabalho para o objeto de implementação ligado. Implementações diferentes são intercambiáveis desde que sigam uma interface comum, permitindo que a mesma GUI trabalhe no Windows e Linux.

Como resultado você pode mudar as classes GUI sem tocar nas classes ligadas a API. Além disso, adicionar suporte para outro sistema operacional só requer a criação de uma subclasse na hierarquia de implementação.

Estrutura



1. A **Abstração** fornece a lógica de controle de alto nível. Ela depende do objeto de implementação para fazer o verdadeiro trabalho de baixo nível.

2. A **Implementação** declara a interface que é comum para todas as implementações concretas. Um abstração só pode se comunicar com um objeto de implementação através de métodos que são declarados aqui.

   A abstração pode listar os mesmos métodos que a implementação, mas geralmente a abstração declara alguns comportamentos complexos que dependem de uma ampla variedade de operações primitivas declaradas pela implementação.

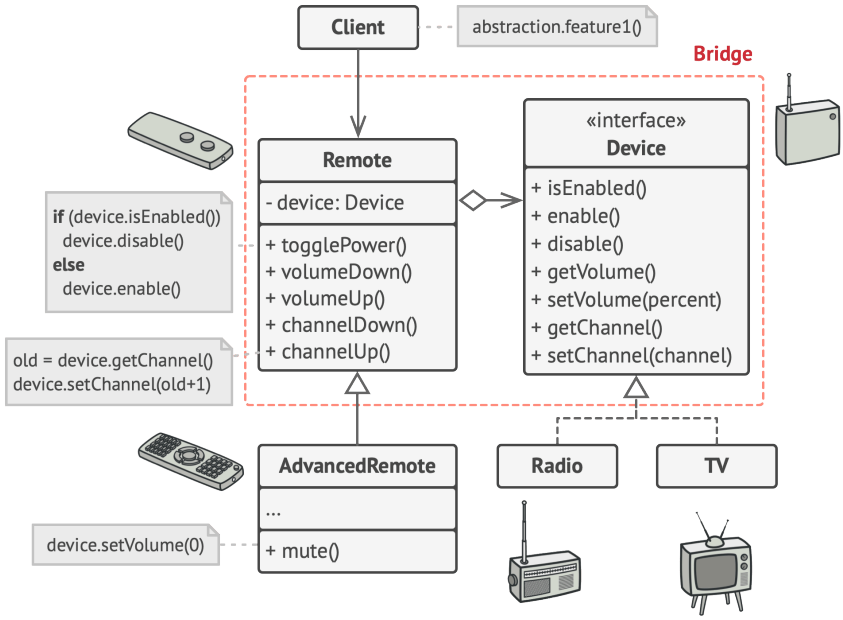
3. **Implementações Concretas** contém código plataforma-específicos.

4. **Abstrações Refinadas** fornecem variantes para controle da lógica. Como seu superior, trabalham com diferentes implementações através da interface geral de implementação.

5. Geralmente o **Cliente** está apenas interessado em trabalhar com a abstração. Contudo, é trabalho do cliente ligar o objeto de abstração com um dos objetos de implementação.

Pseudocódigo

Este exemplo ilustra como o padrão **Bridge** pode ajudar a dividir o código monolítico de uma aplicação que gerencia dispositivos e seus controles remotos. As classes `Dispositivo` agem como a implementação, enquanto que as classes `Controle` agem como abstração.



A hierarquia de classe original é dividida em duas partes: dispositivos e controles remotos.

A classe de controle remoto base declara um campo de referência que liga ela com um objeto de dispositivo. Todos os controles trabalham com dispositivos através da interface geral de dispositivo, que permite que o mesmo controle suporte múltiplos tipos de dispositivo.

Você pode desenvolver as classes de controle remoto independentemente das classes de dispositivo. Tudo que é necessário é criar uma nova subclasse de controle. Por exemplo, um controle remoto básico pode ter apenas dois botões, mas você pode estendê-lo com funcionalidades adicionais, tais como uma bateria adicional ou touchscreen.

O código cliente liga o tipo de controle remoto desejado com um objeto dispositivo específico através do construtor do controle.

```
// A "abstração" define a interface para a parte "controle" das
// duas hierarquias de classe. Ela mantém uma referência a um
// objeto da hierarquia de "implementação" e delega todo o
// trabalho real para esse objeto.
class RemoteControl is
  protected field device: Device
  constructor RemoteControl(device: Device) is
    this.device = device
  method togglePower() is
    if (device.isEnabled()) then
      device.disable()
    else
      device.enable()
  method volumeDown() is
    device.setVolume(device.getVolume() - 10)
  method volumeUp() is
    device.setVolume(device.getVolume() + 10)
  method channelDown() is
    device.setChannel(device.getChannel() - 1)
  method channelUp() is
    device.setChannel(device.getChannel() + 1)


// Você pode estender classes a partir dessa hierarquia de
// abstração independentemente das classes de dispositivo.
class AdvancedRemoteControl extends RemoteControl is
  method mute() is
```

```
    device.setVolume(0)


// A interface "implementação" declara métodos comuns a todas as
// classes concretas de implementação. Ela não precisa coincidir
// com a interface de abstração. Na verdade, as duas interfaces
// podem ser inteiramente diferentes. Tipicamente a interface de
// implementação fornece apenas operações primitivas, enquanto
// que a abstração define operações de alto nível baseada
// naquelas primitivas.
interface Device is
  method isEnabled()
  method enable()
  method disable()
  method getVolume()
  method setVolume(percent)
  method getChannel()
  method setChannel(channel)


// Todos os dispositivos seguem a mesma interface.
class Tv implements Device is
  // ...

class Radio implements Device is
  // ...


// Em algum lugar no código cliente.
tv = new Tv()
remote = new RemoteControl(tv)
remote.togglePower()
```

```
60
61  radio = new Radio()
62  remote = new AdvancedRemoteControl(radio)
```

Aplicabilidade

Utilize o padrão Bridge quando você quer dividir e organizar uma classe monolítica que tem diversas variantes da mesma funcionalidade (por exemplo, se a classe pode trabalhar com diversos servidores de base de dados).

Quanto maior a classe se torna, mais difícil é de entender como ela funciona, e mais tempo se leva para fazer mudanças. As mudanças feitas para uma das variações de funcionalidade podem precisar de mudanças feitas em toda a classe, o que quase sempre resulta em erros ou falha em lidar com efeitos colaterais.

O padrão Bridge permite que você divida uma classe monolítica em diversas hierarquias de classe. Após isso, você pode modificar as classes em cada hierarquia independentemente das classes nas outras. Essa abordagem simplifica a manutenção do código e minimiza o risco de quebrar o código existente.

Utilize o padrão quando você precisa estender uma classe em diversas dimensões ortogonais (independentes).

O Bridge sugere que você extraia uma hierarquia de classe separada para cada uma das dimensões. A classe original delega o trabalho relacionado para os objetos pertencentes àquelas hierarquias ao invés de fazer tudo por conta própria.

Utilize o Bridge se você precisar ser capaz de trocar implementações durante o momento de execução.

Embora seja opcional, o padrão Bridge permite que você substitua o objeto de implementação dentro da abstração. É tão fácil quanto designar um novo valor para um campo.

*A propósito, este último item é o maior motivo pelo qual muitas pessoas confundem o Bridge com o padrão **Strategy**. Lembre-se que um padrão é mais que apenas uma maneira de estruturar suas classes. Ele também pode comunicar intenções e resolver um problema.*

Como implementar

1. Identifique as dimensões ortogonais em suas classes. Esses conceitos independentes podem ser: abstração/plataforma, domínio/infraestrutura, front-end/back-end, ou interface/implementação.

2. Veja quais operações o cliente precisa e defina-as na classe abstração base.

3. Determine as operações disponíveis em todas as plataformas. Declare aquelas que a abstração precisa na interface geral de implementação.

4. Crie classes concretas de implementação para todas as plataformas de seu domínio, mas certifique-se que todas elas sigam a interface de implementação.

5. Dentro da classe de abstração, adicione um campo de referência para o tipo de implementação. A abstração delega a maior parte do trabalho para o objeto de implementação que foi referenciado neste campo.

6. Se você tem diversas variantes de lógica de alto nível, crie abstrações refinadas para cada variante estendendo a classe de abstração básica.

7. O código cliente deve passar um objeto de implementação para o construtor da abstração para associar um com ou outro. Após isso, o cliente pode esquecer sobre a implementação e trabalhar apenas com o objeto de abstração.

Prós e contras

✓ Você pode criar classes e aplicações independentes de plataforma.

✓ O código cliente trabalha com abstrações em alto nível. Ele não fica exposto a detalhes de plataforma.

✓ *Princípio aberto/fechado*. Você pode introduzir novas abstrações e implementações independentemente uma das outras.

✓ *Princípio de responsabilidade única*. Você pode focar na lógica de alto nível na abstração e em detalhes de plataforma na implementação.

✕ Você pode tornar o código mais complicado ao aplicar o padrão em uma classe altamente coesa.

⇄ Relações com outros padrões

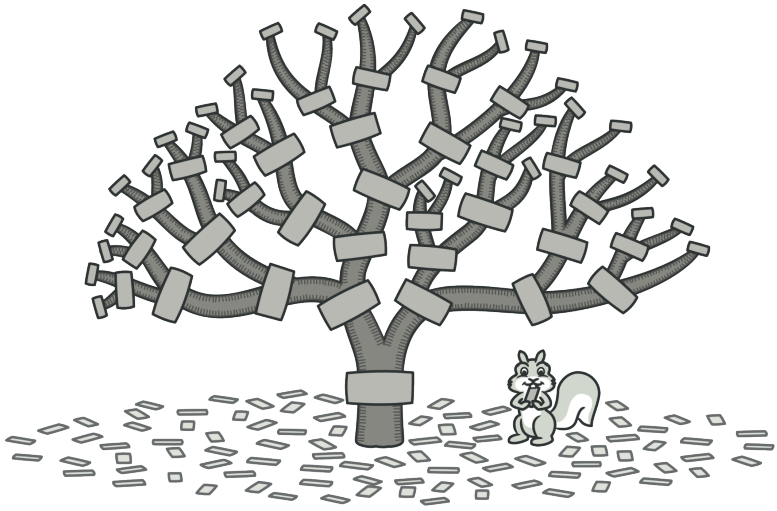
- O **Bridge** é geralmente definido com antecedência, permitindo que você desenvolva partes de uma aplicação independentemente umas das outras. Por outro lado, o **Adapter** é comumente usado em aplicações existentes para fazer com que classes incompatíveis trabalhem bem juntas.

- O **Bridge**, **State**, **Strategy** (e de certa forma o **Adapter**) têm estruturas muito parecidas. De fato, todos esses padrões estão baseados em composição, o que é delegar o trabalho para outros objetos. Contudo, eles todos resolvem problemas diferentes. Um padrão não é apenas uma receita para estruturar seu código de uma maneira específica. Ele também pode comunicar a outros desenvolvedores o problema que o padrão resolve.

- Você pode usar o **Abstract Factory** junto com o **Bridge**. Esse pareamento é útil quando algumas abstrações definidas pelo *Bridge* só podem trabalhar com implementações específicas.

Neste caso, o *Abstract Factory* pode encapsular essas relações e esconder a complexidade do código cliente.

- Você pode combinar o **Builder** com o **Bridge**: a classe *diretor* tem um papel de abstração, enquanto que diferentes *construtores* agem como *implementações*.



COMPOSITE

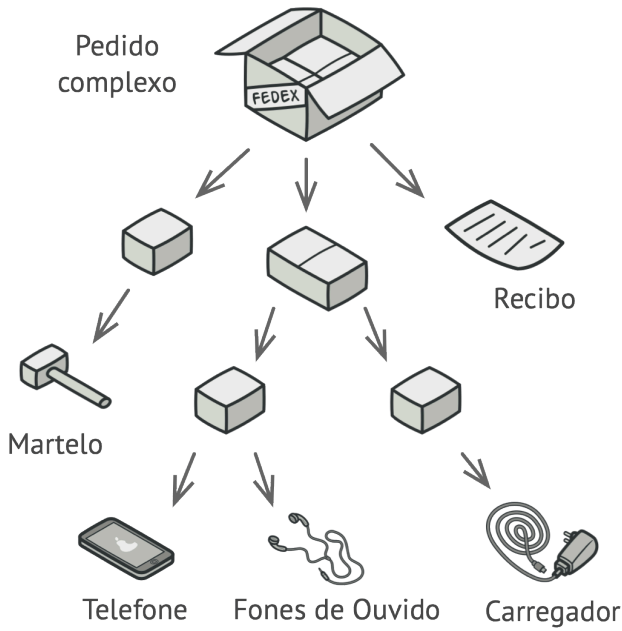
Também conhecido como: Árvore de objetos, Object tree

O **Composite** é um padrão de projeto estrutural que permite que você componha objetos em estruturas de árvores e então trabalhe com essas estruturas como se elas fossem objetos individuais.

☹ Problema

Usar o padrão Composite faz sentido apenas quando o modelo central de sua aplicação pode ser representada como uma árvore.

Por exemplo, imagine que você tem dois tipos de objetos: `Produtos` e `Caixas`. Uma `Caixa` pode conter diversos `Produtos` bem como um número de `Caixas` menores. Essas `Caixas` menores também podem ter alguns `Produtos` ou até mesmo `Caixas` menores que elas, e assim em diante.



Um pedido pode envolver vários produtos, embalados em caixas, que são embalados em caixas maiores e assim em diante. Toda a estrutura se parece com uma árvore de cabeça para baixo.

Digamos que você decida criar um sistema de pedidos que usa essas classes. Os pedidos podem conter produtos simples sem qualquer compartimento, bem como caixas recheadas com produtos... e outras caixas. Como você faria para determinar o preço total desse pedido?

Você pode tentar uma solução direta: desempacotar todas as caixas, verificar cada produto e então calcular o total. Isso pode ser viável no mundo real; mas em um programa, não é tão simples como executar uma iteração. Você tem que conhecer as classes dos `Produtos` e `Caixas` que você está examinando, o nível de aninhamento das caixas e outros detalhes cabeludos de antemão. Tudo isso torna uma solução direta muito confusa ou até impossível.

😊 Solução

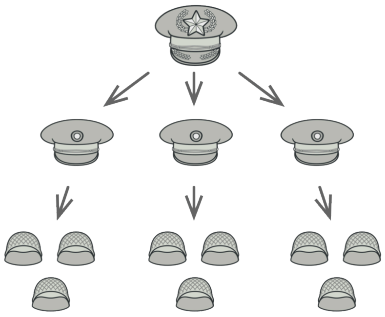
O padrão Composite sugere que você trabalhe com `Produtos` e `Caixas` através de uma interface comum que declara um método para a contagem do preço total.

Como esse método funcionaria? Para um produto, ele simplesmente retornaria o preço dele. Para uma caixa, ele teria que ver cada item que ela contém, perguntar seu preço e então retornar o total para essa caixa. Se um desses itens for uma caixa menor, aquela caixa também deve verificar seu conteúdo e assim em diante, até que o preço de todos os componentes internos sejam calculados. Uma caixa pode até adicionar um custo extra para o preço final, como um preço de embalagem.

O padrão Composite permite que você rode um comportamento recursivamente sobre todos os componentes de uma árvore de objetos.

O maior benefício dessa abordagem é que você não precisa se preocupar sobre as classes concretas dos objetos que compõem essa árvore. Você não precisa saber se um objeto é um produto simples ou uma caixa sofisticada. Você pode tratar todos eles com a mesma interface. Quando você chama um método os próprios objetos passam o pedido pela árvore.

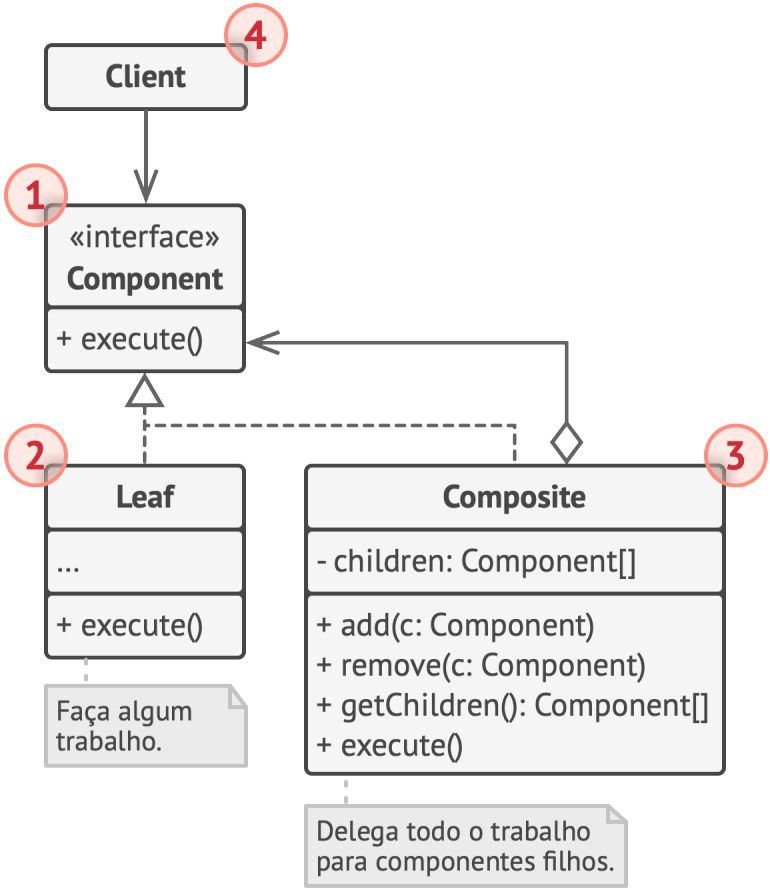
Analogia com o mundo real



Um exemplo de uma estrutura militar.

Exércitos da maioria dos países estão estruturados como hierarquias. Um exército consiste de diversas divisões; uma divisão é um conjunto de brigadas, e uma brigada consiste de pelotões, que podem ser divididos em esquadrões. Finalmente, um esquadrão é um pequeno grupo de soldados reais. Ordens são dadas do topo da hierarquia e são passadas abaixo para cada nível até cada soldado saber o que precisa ser feito.

Estrutura



1. A interface **Componente** descreve operações que são comuns tanto para elementos simples como para elementos complexos da árvore.

2. A **Folha** é um elemento básico de uma árvore que não tem sub-elementos.

   Geralmente, componentes folha acabam fazendo boa parte do verdadeiro trabalho, uma vez que não tem mais ninguém para delegá-lo.

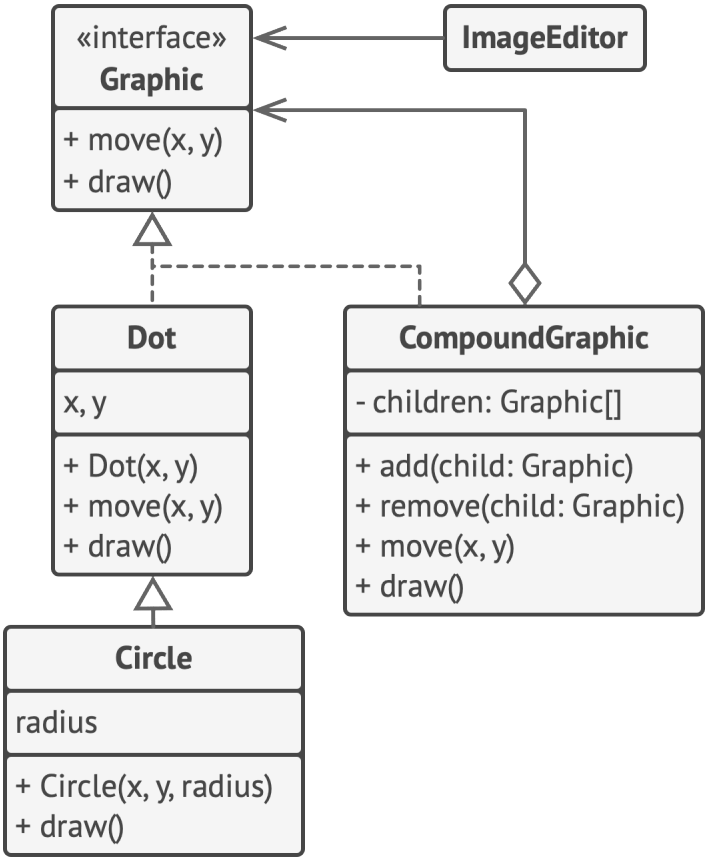
3. O **Contêiner** (ou *composite*) é o elemento que tem sub-elementos: folhas ou outros contêineres. Um contêiner não sabe a classe concreta de seus filhos. Ele trabalha com todos os sub-elementos apenas através da interface componente.

   Ao receber um pedido, um contêiner delega o trabalho para seus sub-elementos, processa os resultados intermediários, e então retorna o resultado final para o cliente.

4. O **Cliente** trabalha com todos os elementos através da interface componente. Como resultado, o cliente pode trabalhar da mesma forma tanto com elementos simples como elementos complexos da árvore.

Pseudocódigo

Nesse exemplo, o padrão **Composite** deixa que você implemente pilhas de formas geométricas em um editor gráfico.



Exemplo do editor de formas geométricas.

A classe `GráficoComposto` é um contêiner que pode ter qualquer número de sub-formas, incluindo outras formas compostas. Uma forma composta tem os mesmos métodos que uma forma simples. Contudo, ao invés de fazer algo próprio, uma forma composta passa o pedido recursivamente para todas as suas filhas e "soma" o resultado.

O código cliente trabalha com todas as formas através da interface única comum à todas as classes de forma. Portanto, o

cliente não sabe se está trabalhando com uma forma simples ou composta. O cliente pode trabalhar com estruturas de objeto muito complexas sem ficar acoplado à classe concreta que formou aquela estrutura.

```
// A interface componente declara operações comuns para ambos os
// objetos simples e complexos de uma composição.
interface Graphic is
  method move(x, y)
  method draw()

// A classe folha representa objetos finais de uma composição.
// Um objeto folha não pode ter quaisquer sub-objetos.
// Geralmente, são os objetos folha que fazem o verdadeiro
// trabalho, enquanto que os objetos composite somente delegam
// para seus sub componentes.
class Dot implements Graphic is
  field x, y

  constructor Dot(x, y) { ... }

  method move(x, y) is
    this.x += x, this.y += y

  method draw() is
    // Desenhar um ponto em X e Y.

// Todas as classes componente estendem outros componentes.
class Circle extends Dot is
  field radius

```

```
    constructor Circle(x, y, radius) { ... }

    method draw() is
      // Desenhar um círculo em X e Y com raio R.

// A classe composite representa componentes complexos que podem
// ter filhos. Objetos composite geralmente delegam o verdadeiro
// trabalho para seus filhos e então "somam" o resultado.
class CompoundGraphic implements Graphic is
    field children: array of Graphic

    // Um objeto composite pode adicionar ou remover outros
    // componentes (tanto simples como complexos) para ou de sua
    // lista de filhos.
    method add(child: Graphic) is
      // Adiciona um filho para o vetor de filhos.

    method remove(child: Graphic) is
      // Remove um filho do vetor de filhos.

    method move(x, y) is
      foreach (child in children) do
        child.move(x, y)

    // Um composite executa sua lógica primária em uma forma
    // particular. Ele percorre recursivamente através de todos
    // seus filhos, coletando e somando seus resultados. Já que
    // os filhos do composite passam essas chamadas para seus
    // próprios filhos e assim em diante, toda a árvore de
    // objetos é percorrida como resultado.
    method draw() is
      // 1. Para cada componente filho:
```

```
    //     - Desenhe o componente.
    //     - Atualize o retângulo limitante.
    // 2. Desenhe um retângulo tracejado usando as
    // limitantes.

// O código cliente trabalha com todos os componentes através de
// suas interfaces base. Dessa forma o código cliente pode
// suportar componentes folha simples e composites complexos.
class ImageEditor is
  field all: CompoundGraphic

  method load() is
    all = new CompoundGraphic()
    all.add(new Dot(1, 2))
    all.add(new Circle(5, 3, 10))
    // ...

  // Combina componentes selecionados em um componente
  // composite complexo.
  method groupSelected(components: array of Graphic) is
    group = new CompoundGraphic()
    foreach (component in components) do
      group.add(component)
      all.remove(component)
    all.add(group)
    // Todos os componentes serão desenhados.
    all.draw()
```

Aplicabilidade

Utilize o padrão Composite quando você tem que implementar uma estrutura de objetos tipo árvore.

O padrão Composite fornece a você com dois tipos básicos de elementos que compartilham uma interface comum: folhas simples e contêineres complexos. Um contêiner pode ser composto tanto de folhas como por outros contêineres. Isso permite a você construir uma estrutura de objetos recursiva aninhada que se assemelha a uma árvore.

Utilize o padrão quando você quer que o código cliente trate tanto os objetos simples como os compostos de forma uniforme.

Todos os elementos definidos pelo padrão Composite compartilham uma interface comum. Usando essa interface o cliente não precisa se preocupar com a classe concreta dos objetos com os quais está trabalhando.

Como implementar

1. Certifique-se que o modelo central de sua aplicação possa ser representada como uma estrutura de árvore. Tente quebrá-lo em elementos simples e contêineres. Lembre-se que contêineres devem ser capazes de conter tanto elementos simples como outros contêineres.

2. Declare a interface componente com uma lista de métodos que façam sentido para componentes complexos e simples.

3. Crie uma classe folha que represente elementos simples. Um programa pode ter múltiplas classes folha diferentes.

4. Crie uma classe contêiner para representar elementos complexos. Nessa classe crie um vetor para armazenar referências aos sub-elementos. O vetor deve ser capaz de armazenar tanto folhas como contêineres, então certifique-se que ele foi declarado com um tipo de interface componente.

   Quando implementar os métodos para a interface componente, lembre-se que um contêiner deve ser capaz de delegar a maior parte do trabalho para os sub-elementos.

5. Por fim, defina os métodos para adicionar e remover os elementos filhos no contêiner.

   Tenha em mente que essas operações podem ser declaradas dentro da interface componente. Isso violaria o *princípio de segregação de interface* porque os métodos estarão vazios na classe folha. Contudo, o cliente será capaz de tratar de todos os elementos de forma igual, mesmo ao montar a árvore.

Prós e contras

✓ Você pode trabalhar com estruturas de árvore complexas mais convenientemente: utilize o polimorfismo e a recursão a seu favor.

✓ *Princípio aberto/fechado*. Você pode introduzir novos tipos de elemento na aplicação sem quebrar o código existente, o que agora funciona com a árvore de objetos.

✕ Pode ser difícil providenciar uma interface comum para classes cuja funcionalidade difere muito. Em certos cenários, você precisaria generalizar muito a interface componente, fazendo dela uma interface de difícil compreensão.

Relações com outros padrões

- Você pode usar o **Builder** quando criar árvores **Composite** complexas porque você pode programar suas etapas de construção para trabalhar recursivamente.

- O **Chain of Responsibility** é frequentemente usado em conjunto com o **Composite**. Neste caso, quando um componente folha recebe um pedido, ele pode passá-lo através de uma corrente de todos os componentes pai até a raiz do objeto árvore.

- Você pode usar **Iteradores** para percorrer árvores **Composite**.

- Você pode usar o **Visitor** para executar uma operação sobre uma árvore **Composite** inteira.

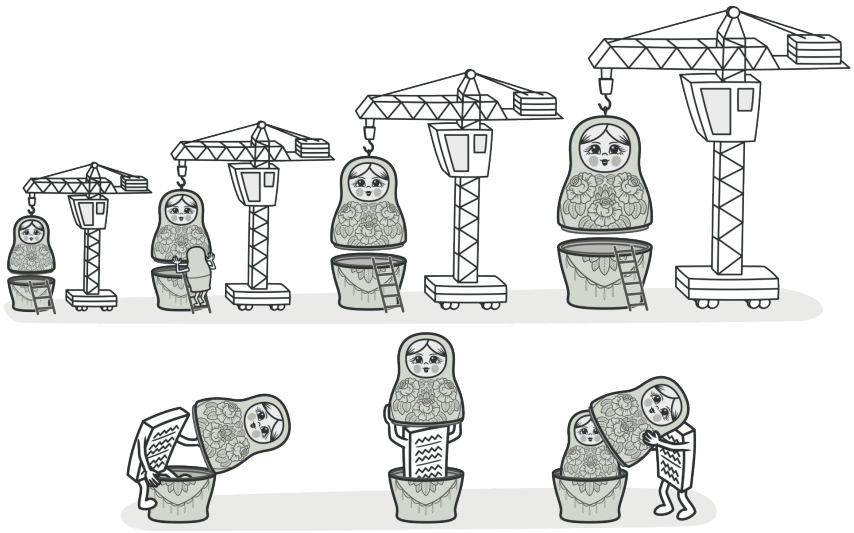
- Você pode implementar nós folha compartilhados da árvore **Composite** como **Flyweights** para salvar RAM.

- O **Composite** e o **Decorator** tem diagramas estruturais parecidos já que ambos dependem de composição recursiva para organizar um número indefinido de objetos.

  Um *Decorador* é como um *Composite* mas tem apenas um componente filho. Há outra diferença significativa: o *Decorador* adiciona responsabilidades adicionais ao objeto envolvido, enquanto que o *Composite* apenas "soma" o resultado de seus filhos.

  Contudo, os padrões também podem cooperar: você pode usar o *Decorador* para estender o comportamento de um objeto específico na árvore **Composite**

- Projetos que fazem um uso pesado de **Composite** e do **Decorator** podem se beneficiar com frequência do uso do **Prototype**. Aplicando o padrão permite que você clone estruturas complexas ao invés de reconstruí-las do zero.



DECORATOR

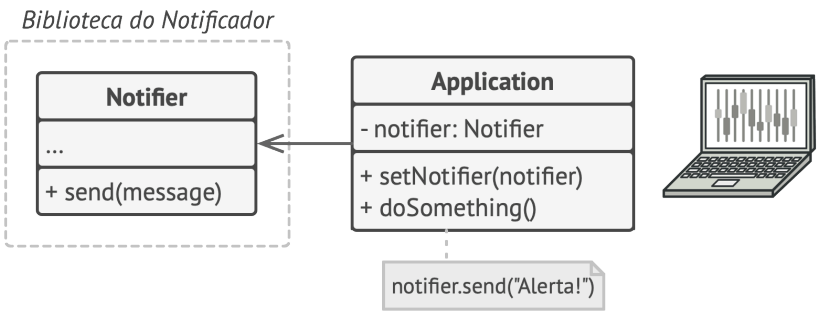
Também conhecido como: Decorador, Envoltório, Wrapper

O **Decorator** é um padrão de projeto estrutural que permite que você acople novos comportamentos para objetos ao colocá-los dentro de invólucros de objetos que contém os comportamentos.

☹ Problema

Imagine que você está trabalhando em um biblioteca de notificação que permite que outros programas notifiquem seus usuários sobre eventos importantes.

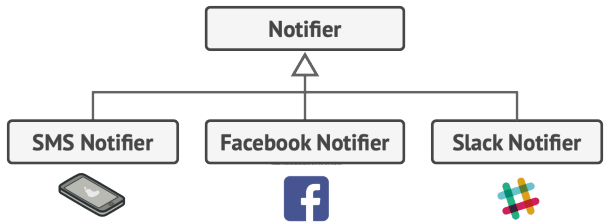
A versão inicial da biblioteca foi baseada na classe `Notificador` que tinha apenas alguns poucos campos, um construtor, e um único método `enviar`. O método podia aceitar um argumento de mensagem de um cliente e enviar a mensagem para uma lista de emails que eram passadas para o notificador através de seu construtor. Uma aplicação de terceiros que agia como cliente deveria criar e configurar o objeto notificador uma vez, e então usá-lo a cada vez que algo importante acontecesse.



Um programa poderia usar a classe notificador para enviar notificações sobre eventos importantes para um conjunto predefinido de emails.

Em algum momento você se dá conta que os usuários da biblioteca esperam mais que apenas notificações por email. Muitos deles gostariam de receber um SMS acerca de problemas

críticos. Outros gostariam de ser notificados no Facebook, e, é claro, os usuários corporativos adorariam receber notificações do Slack.

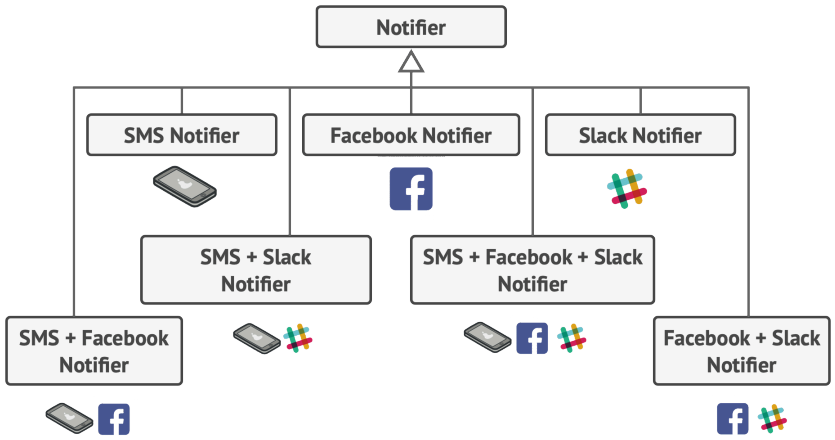


Cada tipo de notificação é implementada em uma subclasse do notificador.

Quão difícil isso seria? Você estende a classe `Notificador` e coloca os métodos de notificação adicionais nas novas subclasses. Agora o cliente deve ser instanciado à classe de notificação que deseja e usar ela para todas as futura notificações.

Mas então alguém, com razão, pergunta a você, "Por que você não usa diversos tipos de notificação de uma só vez? Se a sua casa pegar fogo, você provavelmente vai querer ser notificado por todos os canais."

Você tenta resolver esse problema criando subclasses especiais que combinam diversos tipos de métodos de notificação dentro de uma classe. Contudo, rapidamente você nota que isso irá inflar o código imensamente, e não só da biblioteca, o código cliente também.



Combinação explosiva de subclasses.

Você precisa encontrar outra maneira de estruturar classes de notificação para que o número delas não quebre um recorde do Guinness acidentalmente.

😊 Solução

Estender uma classe é a primeira coisa que vem à mente quando você precisa alterar o comportamento de um objeto. Contudo, a herança vem com algumas ressalvas sérias que você precisa estar ciente.

- A herança é estática. Você não pode alterar o comportamento de um objeto existente durante o tempo de execução. Você só pode substituir todo o objeto por outro que foi criado de uma subclasse diferente.

- As subclasses só podem ter uma classe pai. Na maioria das linguagens, a herança não permite que uma classe herde comportamentos de múltiplas classes ao mesmo tempo.

  Uma das maneiras de superar essas ressalvas é usando *Agregação* ou *Composição*[1] ao invés de *Herança*. Ambas alternativas funcionam quase da mesma maneira: um objeto *tem uma* referência com outro e delega alguma funcionalidade, enquanto que na herança, o próprio objeto *é* capaz de fazer a função, herdando o comportamento da sua superclasse.

  Com essa nova abordagem você pode facilmente substituir o objeto "auxiliador" por outros, mudando o comportamento do contêiner durante o tempo de execução. Um objeto pode usar o comportamento de várias classes, ter referências a múltiplos objetos, e delegar qualquer tipo de trabalho a eles. A agregação/composição é o princípio chave por trás de muitos padrões de projeto, incluindo o Decorator. Falando nisso, vamos voltar à discussão desse padrão.

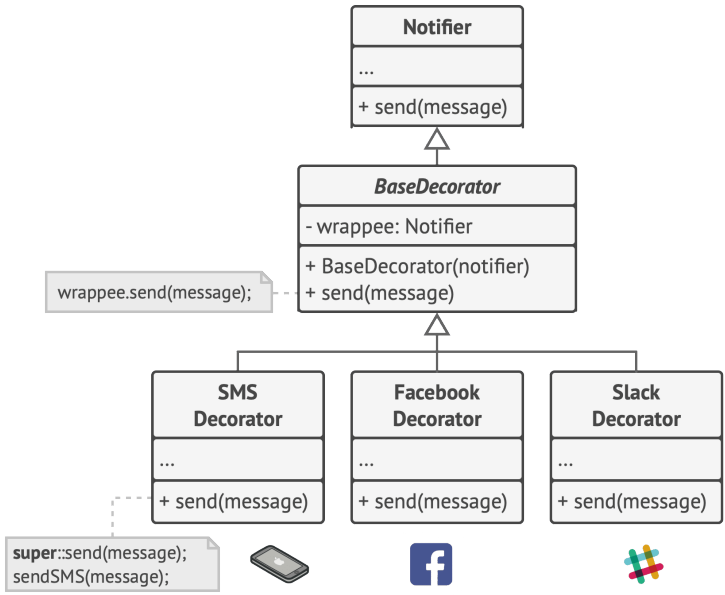
Herança vs. Agregação

1. *Agregação*: objeto A contém objetos B; B pode viver sem A.
   Composição: objeto A consiste de objetos B; A gerencia o ciclo de vida de B; B não pode viver sem A.

"Envoltório" (ing. "wrapper") é o apelido alternativo para o padrão Decorator que expressa claramente a ideia principal dele. Um *envoltório* é um objeto que pode ser ligado com outro objeto *alvo*. O envoltório contém o mesmo conjunto de métodos que o alvo e delega a ele todos os pedidos que recebe. Contudo, o envoltório pode alterar o resultado fazendo alguma coisa ou antes ou depois de passar o pedido para o alvo.

Quando um simples envoltório se torna um verdadeiro decorador? Como mencionei, o envoltório implementa a mesma interface que o objeto envolvido. É por isso que da perspectiva do cliente esses objetos são idênticos. Faça o campo de referência do envoltório aceitar qualquer objeto que segue aquela interface. Isso lhe permitirá cobrir um objeto em múltiplos envoltórios, adicionando o comportamento combinado de todos os envoltórios a ele.

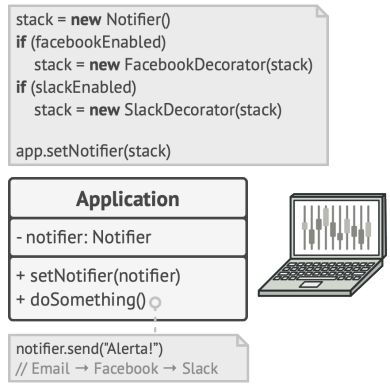
No nosso exemplo de notificações vamos deixar o simples comportamento de notificação por email dentro da classe `Notificador` base, mas transformar todos os métodos de notificação em decoradores.



Vários métodos de notificação se tornam decoradores.

O código cliente vai precisar envolver um objeto notificador básico em um conjunto de decoradores que coincidem com as preferências do cliente. Os objetos resultantes serão estruturados como uma pilha.

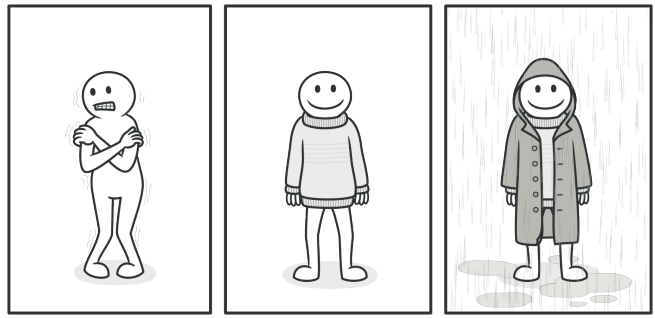
O último decorador na pilha seria o objeto que o cliente realmente trabalha. Como todos os decoradores implementam a mesma interface que o notificador base, o resto do código cliente não quer saber se ele funciona com o objeto “puro” do notificador ou do decorador.



As aplicações pode configurar pilhas complexas de notificações decoradores

Podemos utilizar a mesma abordagem para vários comportamentos tais como formatação de mensagens ou compor uma lista de recipientes. O cliente pode decorar o objeto com quaisquer decoradores customizados, desde que sigam a mesma interface que os demais.

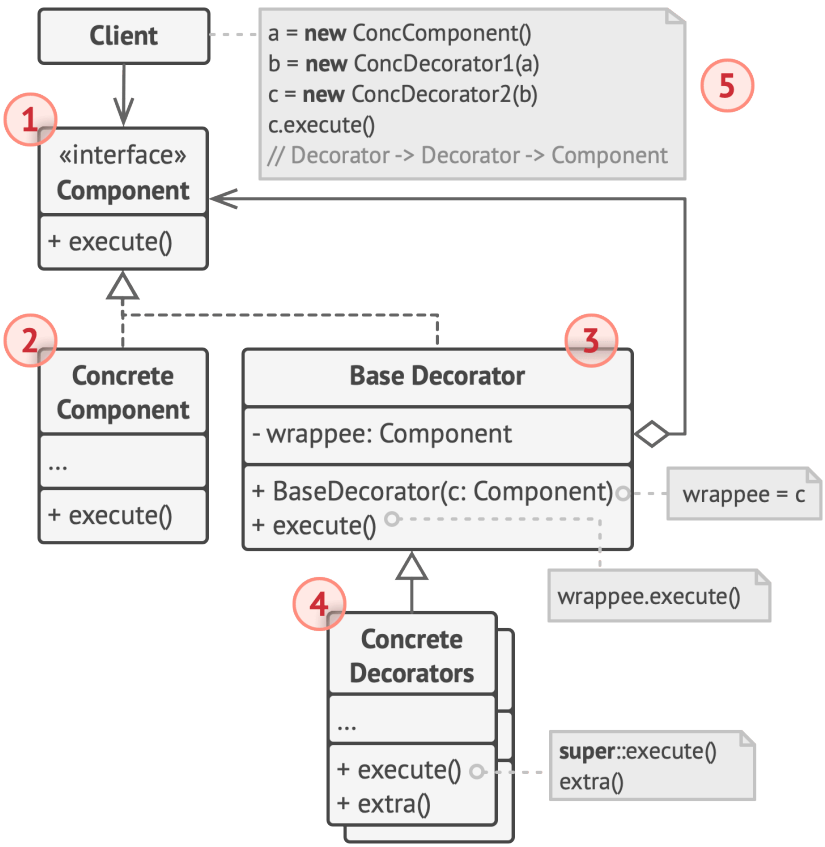
Analogia com o mundo real



Você tem um efeito combinado de usar múltiplas peças de roupa.

Vestir roupas é um exemplo de usar decoradores. Quando você está com frio, você se envolve com um suéter. Se você ainda sente frio com um suéter, você pode vestir um casaco por cima. Se está chovendo, você pode colocar uma capa de chuva. Todas essas vestimentas "estendem" seu comportamento básico mas não são parte de você, e você pode facilmente remover uma peça de roupa sempre que não precisar mais dela.

Estrutura



1. O **Componente** declara a interface comum tanto para os envoltórios como para os objetos envolvidos.

2. O **Componente Concreto** é uma classe de objetos sendo envolvidos. Ela define o comportamento básico, que pode ser alterado por decoradores.

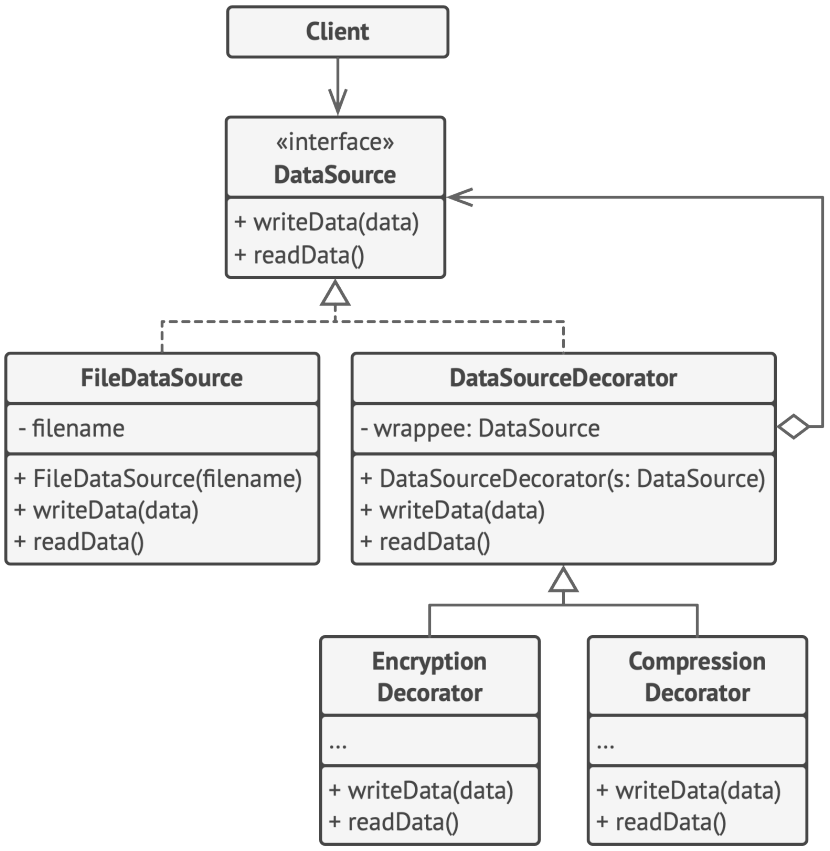
3. A classe **Decorador Base** tem um campo para referenciar um objeto envolvido. O tipo do campo deve ser declarado assim como a interface do componente para que possa conter ambos os componentes concretos e os decoradores. O decorador base delega todas as operações para o objeto envolvido.

4. Os **Decoradores Concretos** definem os comportamentos adicionais que podem ser adicionados aos componentes dinamicamente. Os decoradores concretos sobrescrevem métodos do decorador base e executam seus comportamentos tanto antes como depois de chamarem o método pai.

5. O **Cliente** pode envolver componentes em múltiplas camadas de decoradors, desde que trabalhe com todos os objetos através da interface do componente.

Pseudocódigo

Neste exemplo, o padrão **Decorator** lhe permite comprimir e encriptar dados sensíveis independentemente do código que verdadeiramente usa esses dados.



Exemplo da encriptação e compressão com decoradores.

A aplicação envolve o objeto da fonte de dados com um par de decoradores. Ambos invólucros mudam a maneira que os dados são escritos e lidos no disco:

- Antes dos dados serem **escritos no disco**, os decoradores encriptam e comprimem eles. A classe original escreve os dados protegidos e encriptados para o arquivo sem saber da mudança.

- Logo antes dos dados serem **lidos do disco**, ele passa pelos mesmos decoradores que descomprimem e decodificam eles.

  Os decoradores e a classe da fonte de dados implementam a mesma interface, que os torna intercomunicáveis dentro do código cliente.

```
// A interface componente define operações que podem ser
// alteradas por decoradores.
interface DataSource is
  method writeData(data)
  method readData():data

// Componentes concretos fornecem uma implementação padrão para
// as operações. Pode haver diversas variações dessas classes em
// um programa.
class FileDataSource implements DataSource is
  constructor FileDataSource(filename) { ... }

  method writeData(data) is
    // Escreve dados no arquivo.

  method readData():data is
    // Lê dados de um arquivo.

// A classe decorador base segue a mesma interface que os outros
// componentes. O propósito primário dessa classe é definir a
// interface que envolve todos os decoradores concretos. A
// implementação padrão do código de envolvimento pode também
// incluir um campo para armazenar um componente envolvido e os
// meios para inicializá-lo.
```

```
class DataSourceDecorator implements DataSource is
  protected field wrappee: DataSource

  constructor DataSourceDecorator(source: DataSource) is
    wrappee = source

  // O decorador base simplesmente delega todo o trabalho para
  // a o componente envolvido. Comportamentos extra podem ser
  // adicionados em decoradores concretos.
  method writeData(data) is
    wrappee.writeData(data)

  // Decoradores concretos podem chamar a implementação pai da
  // operação ao invés de chamar o objeto envolvido
  // diretamente. Essa abordagem simplifica a extensão de
  // classes decorador.
  method readData():data is
    return wrappee.readData()

// Decoradores concretos devem chamar métodos no objeto
// envolvido, mas podem adicionar algo próprio para o resultado.
// Os decoradores podem executar o comportamento adicional tanto
// antes como depois da chamada ao objeto envolvido.
class EncryptionDecorator extends DataSourceDecorator is
  method writeData(data) is
    // 1. Encriptar os dados passados.
    // 2. Passar dados encriptados para o método writeData
    // do objeto envolvido.

  method readData():data is
    // 1. Obter os dados do método readData do objeto
    // envolvido.
```

```
    // 2. Tentar decifrá-lo se for encriptado.
    // 3. Retornar o resultado.

// Você pode envolver objetos em diversas camadas de
// decoradores.
class CompressionDecorator extends DataSourceDecorator is
  method writeData(data) is
    // 1. Comprimir os dados passados.
    // 2. Passar os dados comprimidos para o método
    // writeData do objeto envolvido.

  method readData():data is
    // 1. Obter dados do método readData do objeto
    // envolvido.
    // 2. Tentar descomprimi-lo se for comprimido.
    // 3. Retornar o resultado.

// Opção 1. Um exemplo simples de uma montagem decorador.
class Application is
  method dumbUsageExample() is
    source = new FileDataSource("somefile.dat")
    source.writeData(salaryRecords)
    // O arquivo alvo foi escrito com dados simples.

    source = new CompressionDecorator(source)
    source.writeData(salaryRecords)
    // O arquivo alvo foi escrito com dados comprimidos.

    source = new EncryptionDecorator(source)
    // A variável fonte agora contém isso:
    // Encryption > Compression > FileDataSource
    source.writeData(salaryRecords)
```

```
    // O arquivo foi escrito com dados comprimidos e
    // encriptados.


// Opção 2. Código cliente que usa uma fonte de dados externa.
// Objetos SalaryManager não sabem e nem se importam sobre as
// especificações de armazenamento de dados. Eles trabalham com
// uma fonte de dados pré configurada recebida pelo configurador
// da aplicação.
class SalaryManager is
  field source: DataSource

  constructor SalaryManager(source: DataSource) { ... }

  method load() is
    return source.readData()

  method save() is
    source.writeData(salaryRecords)
  // ...Outros métodos úteis...


// A aplicação pode montar diferentes pilhas de decoradores no
// tempo de execução, dependendo da configuração ou ambiente.
class ApplicationConfigurator is
  method configurationExample() is
    source = new FileDataSource("salary.dat")
    if (enabledEncryption)
      source = new EncryptionDecorator(source)
    if (enabledCompression)
      source = new CompressionDecorator(source)

```

```
    logger = new SalaryManager(source)
    salary = logger.load()
  // ...
```

Aplicabilidade

Utilize o padrão Decorator quando você precisa ser capaz de projetar comportamentos adicionais para objetos em tempo de execução sem quebrar o código que usa esses objetos.

O Decorator lhe permite estruturar sua lógica de negócio em camadas, criar um decorador para cada camada, e compor objetos com várias combinações dessa lógica durante a execução. O código cliente pode tratar de todos esses objetos da mesma forma, como todos seguem a mesma interface comum.

Utilize o padrão quando é complicado ou impossível estender o comportamento de um objeto usando herança.

Muitas linguagens de programação tem a palavra chave `final` que pode ser usada para prevenir a extensão de uma classe. Para uma classe final, a única maneira de reutilizar seu comportamento existente seria envolver a classe com seu próprio invólucro usando o padrão Decorator.

Como implementar

1. Certifique-se que seu domínio de negócio pode ser representado como um componente primário com múltiplas camadas opcionais sobre ele.

2. Descubra quais métodos são comuns tanto para o componente primário e para as camadas opcionais. Crie uma interface componente e declare aqueles métodos ali.

3. Crie uma classe componente concreta e defina o comportamento base nela.

4. Crie uma classe decorador base. Ela deve ter um campo para armazenar uma referência ao objeto envolvido. O campo deve ser declarado com o tipo da interface componente para permitir uma ligação entre os componentes concretos e decoradores. O decorador base deve delegar todo o trabalho para o objeto envolvido.

5. Certifique-se que todas as classes implementam a interface componente.

6. Crie decoradores concretos estendendo-os a partir do decorador base. Um decorador concreto deve executar seu comportamento antes ou depois da chamada para o método pai (que sempre delega para o objeto envolvido).

7. O código cliente deve ser responsável por criar decoradores e compô-los do jeito que o cliente precisa.

Prós e contras

✓ Você pode estender o comportamento de um objeto sem fazer um nova subclasse.

✓ Você pode adicionar ou remover responsabilidades de um objeto no momento da execução.

✓ Você pode combinar diversos comportamentos ao envolver o objeto com múltiplos decoradores.

✓ *Princípio de responsabilidade única*. Você pode dividir uma classe monolítica que implementa muitas possíveis variantes de um comportamento em diversas classes menores.

✕ É difícil remover um invólucro de uma pilha de invólucros.

✕ É difícil implementar um decorador de tal maneira que seu comportamento não dependa da ordem do pilha de decoradores.

✕ A configuração inicial do código de camadas pode ficar bastante feia.

Relações com outros padrões

- O **Adapter** muda a interface de um objeto existente, enquanto que o **Decorator** melhora um objeto sem mudar sua interface. Além disso, o *Decorator* suporta composição recursiva, o que não seria possível quando você usa o *Adapter*.

- O **Adapter** fornece uma interface diferente para um objeto encapsulado, o **Proxy** fornece a ele a mesma interface, e o **Decorator** fornece a ele com uma interface melhorada.

- O **Chain of Responsibility** e o **Decorator** têm estruturas de classe muito parecidas. Ambos padrões dependem de composição recursiva para passar a execução através de uma série de objetos. Contudo, há algumas diferenças cruciais.

  Os handlers do *CoR* podem executar operações arbitrárias independentemente uma das outras. Eles também podem parar o pedido de ser passado adiante em qualquer ponto. Por outro lado, vários *decoradores* podem estender o comportamento do objeto enquanto mantém ele consistente com a interface base. Além disso, os decoradores não tem permissão para quebrar o fluxo do pedido.

- O **Composite** e o **Decorator** tem diagramas estruturais parecidos já que ambos dependem de composição recursiva para organizar um número indefinido de objetos.

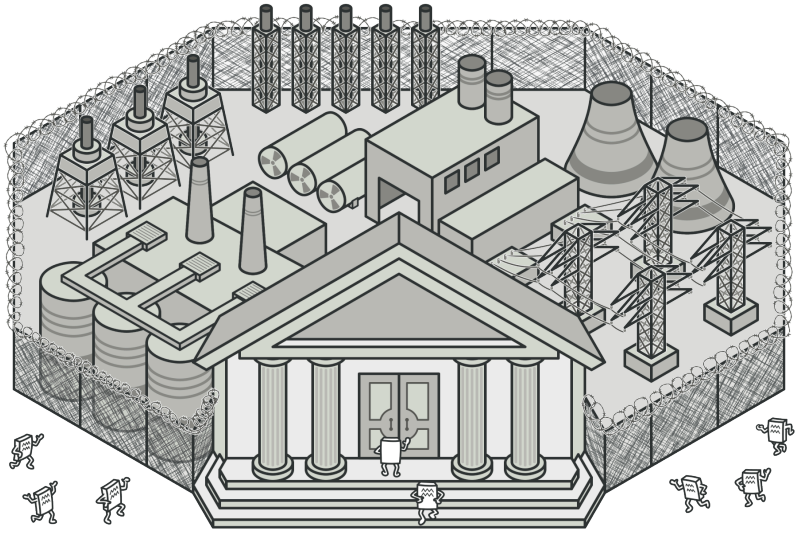
  Um *Decorador* é como um *Composite* mas tem apenas um componente filho. Há outra diferença significativa: o *Decorador* adiciona responsabilidades adicionais ao objeto envolvido, enquanto que o *Composite* apenas "soma" o resultado de seus filhos.

Contudo, os padrões também podem cooperar: você pode usar o *Decorador* para estender o comportamento de um objeto específico na árvore **Composite**

- Projetos que fazem um uso pesado de **Composite** e do **Decorator** podem se beneficiar com frequência do uso do **Prototype**. Aplicando o padrão permite que você clone estruturas complexas ao invés de reconstruí-las do zero.

- O **Decorator** permite que você mude a pele de um objeto, enquanto o **Strategy** permite que você mude suas entranhas.

- O **Decorator** e o **Proxy** têm estruturas semelhantes, mas propósitos muito diferentes. Alguns padrões são construídos no princípio de composição, onde um objeto deve delegar parte do trabalho para outro. A diferença é que o *Proxy* geralmente gerencia o ciclo de vida de seu objeto serviço por conta própria, enquanto que a composição do *decoradores* é sempre controlada pelo cliente.



FACADE

Também conhecido como: Fachada

O **Facade** é um padrão de projeto estrutural que fornece uma interface simplificada para uma biblioteca, um framework, ou qualquer conjunto complexo de classes.

☹ Problema

Imagine que você precisa fazer seu código funcionar com um amplo conjunto de objetos que pertencem a uma sofisticada biblioteca ou framework. Normalmente, você precisaria inicializar todos aqueles objetos, rastrear as dependências, executar métodos na ordem correta, e assim por diante.

Como resultado, a lógica de negócio de suas classes vai ficar firmemente acoplada aos detalhes de implementação das classes de terceiros, tornando difícil compreendê-lo e mantê-lo.

☺ Solução

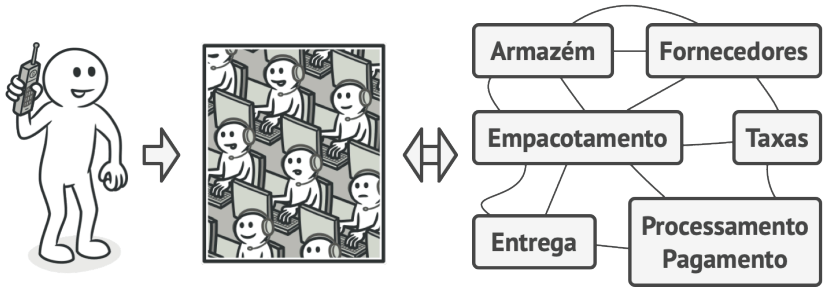
Uma fachada é uma classe que fornece uma interface simples para um subsistema complexo que contém muitas partes que se movem. Uma fachada pode fornecer funcionalidades limitadas em comparação com trabalhar com os subsistemas diretamente. Contudo, ela inclui apenas aquelas funcionalidades que o cliente se importa.

Ter uma fachada é útil quando você precisa integrar sua aplicação com uma biblioteca sofisticada que tem dúzias de funcionalidades, mas você precisa de apenas um pouquinho delas.

Por exemplo, uma aplicação que carrega vídeos curtos engraçados com gatos para redes sociais poderia potencialmente usar uma biblioteca de conversão de vídeo profissional. Contudo, tudo que ela realmente precisa é uma classe com um

único método `codificar(nomeDoArquivo, formato)`. Após criar tal classe e conectá-la com a biblioteca de conversão de vídeo, você terá sua primeira fachada.

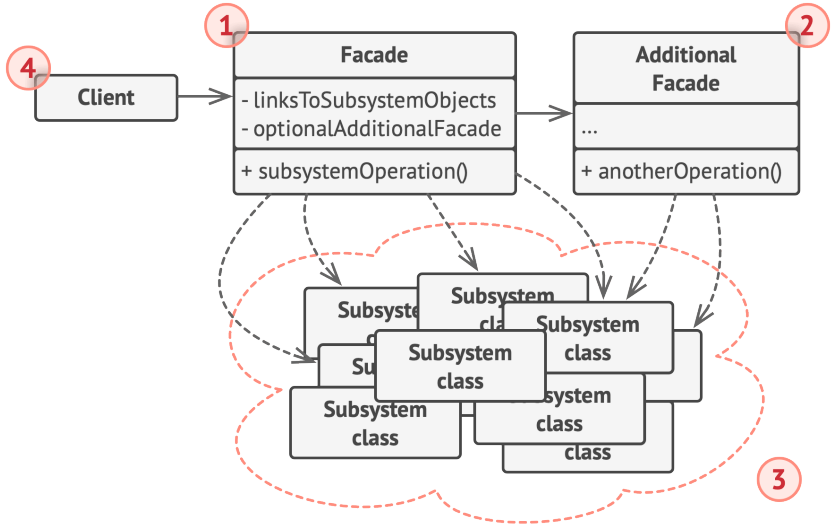
Analogia com o mundo real



Fazer pedidos por telefone.

Quando você liga para uma loja para fazer um pedido, um operador é sua fachada para todos os serviços e departamentos da loja. O operador fornece a você uma simples interface de voz para o sistema de pedido, pagamentos, e vários sistemas de entrega.

Estrutura



1. A **Fachada** fornece um acesso conveniente para uma parte particular da funcionalidade do subsistema. Ela sabe onde direcionar o pedido do cliente e como operar todas as partes móveis.

2. Uma classe **Fachada Adicional** pode ser criada para prevenir a poluição de uma única fachada com funcionalidades não relevantes que podem torná-lo mais uma estrutura complexa. Fachadas adicionais podem ser usadas tanto por clientes como por outras fachadas.

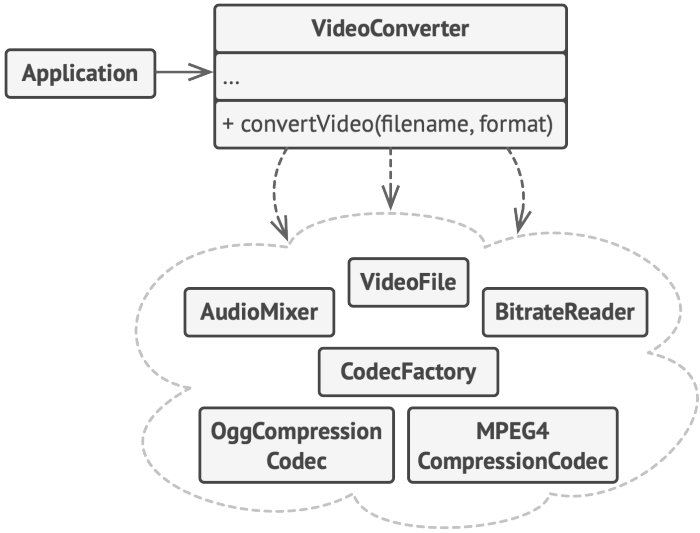
3. O **Subsistema Complexo** consiste em dúzias de objetos variados. Para tornar todos eles em algo que signifique alguma coisa, você tem que mergulhar fundo nos detalhes de implementação do subsistema, tais como objetos de inicialização na ordem correta e suprí-los com dados no formato correto.

As classes do subsistema não estão cientes da existência da fachada. Elas operam dentro do sistema e trabalham entre si diretamente.

4. O **Cliente** usa a fachada ao invés de chamar os objetos do subsistema diretamente.

Pseudocódigo

Neste exemplo, o padrão **Facade** simplifica a interação com um framework complexo de conversão de vídeo.



Um exemplo de isolamento de múltiplas dependências dentro de uma única classe fachada.

Ao invés de fazer seu código funcionar com dúzias de classes framework diretamente, você cria a classe fachada que

encapsula aquela funcionalidade e a esconde do resto do código. Essa estrutura também ajuda você a minimizar o esforço usando para atualizar para futuras versões do framework ou substituí-lo por outro. A única coisa que você precisaria mudar em sua aplicação seria a implementação dos métodos da fachada.

```
// Essas são algumas das classes de um framework complexo de um
// conversor de vídeo de terceiros. Nós não controlamos aquele
// código, portanto não podemos simplificá-lo.

class VideoFile
// ...

class OggCompressionCodec
// ...

class MPEG4CompressionCodec
// ...

class CodecFactory
// ...

class BitrateReader
// ...

class AudioMixer
// ...


// Nós criamos uma classe fachada para esconder a complexidade
```

```
// do framework atrás de uma interface simples. É uma troca
// entre funcionalidade e simplicidade.
class VideoConverter is
  method convert(filename, format):File is
    file = new VideoFile(filename)
    sourceCodec = new CodecFactory.extract(file)
    if (format == "mp4")
      destinationCodec = new MPEG4CompressionCodec()
    else
      destinationCodec = new OggCompressionCodec()
    buffer = BitrateReader.read(filename, sourceCodec)
    result = BitrateReader.convert(buffer, destinationCodec)
    result = (new AudioMixer()).fix(result)
    return new File(result)

// As classes da aplicação não dependem de um bilhão de classes
// fornecidas por um framework complexo. Também, se você decidir
// trocar de frameworks, você só precisa reescrever a classe
// fachada.
class Application is
  method main() is
    convertor = new VideoConverter()
    mp4 = convertor.convert("funny-cats-video.ogg", "mp4")
    mp4.save()
```

Aplicabilidade

Utilize o padrão Facade quando você precisa ter uma interface limitada mas simples para um subsistema complexo.

Com o passar do tempo, subsistemas ficam mais complexos. Até mesmo aplicar padrões de projeto tipicamente leva a criação de mais classes. Um subsistema pode tornar-se mais flexível e mais fácil de se reutilizar em vários contextos, mas a quantidade de códigos padrão e de configuração que ele necessita de um cliente cresce cada vez mais. O Facade tenta consertar esse problema fornecendo um atalho para as funcionalidades mais usadas do subsistema que corresponde aos requerimentos do cliente.

Utilize o Facade quando você quer estruturar um subsistema em camadas.

Crie fachadas para definir pontos de entrada para cada nível de um subsistema. Você pode reduzir o acoplamento entre múltiplos subsistemas fazendo com que eles se comuniquem apenas através de fachadas.

Por exemplo, vamos retornar ao nosso framework de conversão de vídeo. Ele pode ser quebrado em duas camadas: relacionados a vídeo e áudio. Para cada camada, você cria uma fachada e então faz as classes de cada camada se comunicarem entre si através daquelas fachadas. Essa abordagem se parece muito com o padrão **Mediator**.

Como implementar

1. Verifique se é possível providenciar uma interface mais simples que a que o subsistema já fornece. Você está no caminho

certo se essa interface faz o código cliente independente de muitas classes do subsistema.

2. Declare e implemente essa interface em uma nova classe fachada. A fachada deve redirecionar as chamadas do código cliente para os objetos apropriados do subsistema. A fachada deve ser responsável por inicializar o subsistema e gerenciar seu ciclo de vida a menos que o código cliente já faça isso.

3. Para obter o benefício pleno do padrão, faça todo o código cliente se comunicar com o subsistema apenas através da fachada. Agora o código cliente fica protegido de qualquer mudança no código do subsistema. Por exemplo, quando um subsistema recebe um upgrade para uma nova versão, você só precisa modificar o código na fachada.

4. Se a fachada ficar **grande demais**, considere extrair parte de seu comportamento para uma nova e refinada classe fachada.

Prós e contras

✓ Você pode isolar seu código da complexidade de um subsistema.

✕ Uma fachada pode se tornar **um objeto deus** acoplado a todas as classes de uma aplicação.

⇄ Relações com outros padrões

- O **Facade** define uma nova interface para objetos existentes, enquanto que o **Adapter** tenta fazer uma interface existente ser utilizável. O *Adapter* geralmente envolve apenas um objeto, enquanto que o *Facade* trabalha com um inteiro subsistema de objetos.

- O **Abstract Factory** pode servir como uma alternativa para o **Facade** quando você precisa apenas esconder do código cliente a forma com que são criados os objetos do subsistema.

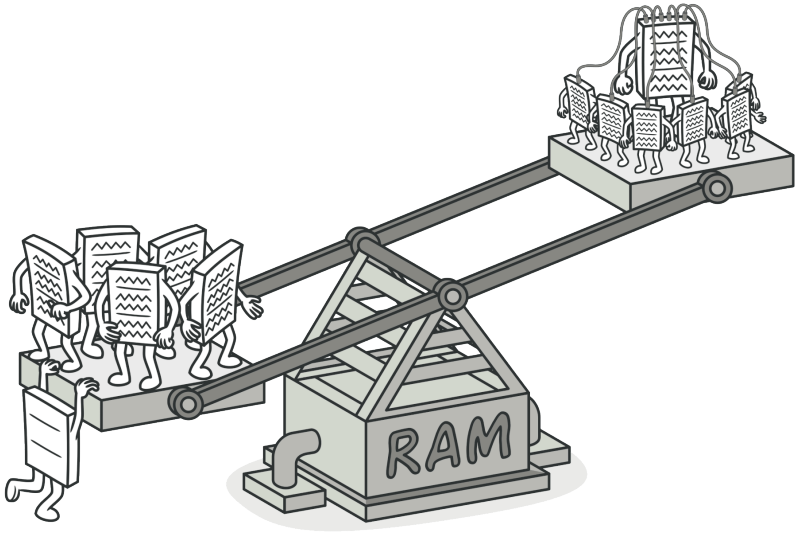
- O **Flyweight** mostra como fazer vários pequenos objetos, enquanto o **Facade** mostra como fazer um único objeto que represente um subsistema inteiro.

- O **Facade** e o **Mediator** têm trabalhos parecidos: eles tentam organizar uma colaboração entre classes firmemente acopladas.

    - O *Facade* define uma interface simplificada para um subsistema de objetos, mas ele não introduz qualquer nova funcionalidade. O próprio subsistema não está ciente da fachada. Objetos dentro do subsistema podem se comunicar diretamente.
    - O *Mediator* centraliza a comunicação entre componentes do sistema. Os componentes só sabem do objeto mediador e não se comunicam diretamente.

- Uma classe **fachada** pode frequentemente ser transformada em uma **singleton** já que um único objeto fachada é suficiente na maioria dos casos.

- O **Facade** é parecido como o **Proxy** no quesito que ambos colocam em buffer uma entidade complexa e inicializam ela sozinhos. Ao contrário do *Facade*, o *Proxy* tem a mesma interface que seu objeto de serviço, o que os torna intermutáveis.



FLYWEIGHT

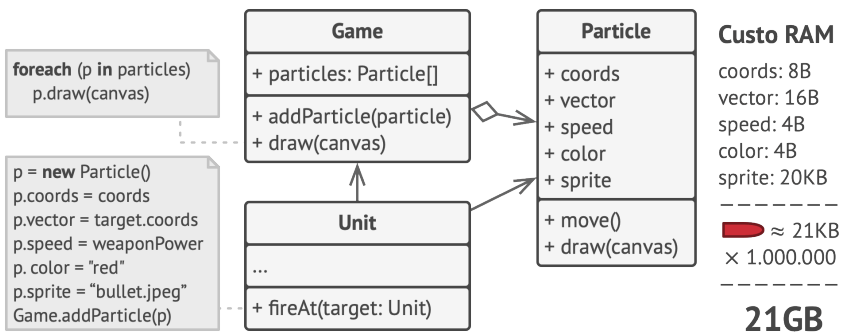
Também conhecido como: Peso mosca, Cache

O **Flyweight** é um padrão de projeto estrutural que permite a você colocar mais objetos na quantidade de RAM disponível ao compartilhar partes comuns de estado entre os múltiplos objetos ao invés de manter todos os dados em cada objeto.

☹ Problema

Para se divertir após longas horas de trabalho você decide criar um jogo simples: os jogadores estarão se movendo em um mapa e atirado uns aos outros. Você escolhe implementar um sistema de partículas realístico e faz dele uma funcionalidade distinta do jogo. Uma grande quantidades de balas, mísseis, e estilhaços de explosões devem voar por todo o mapa e entregar adrenalina para o jogador.

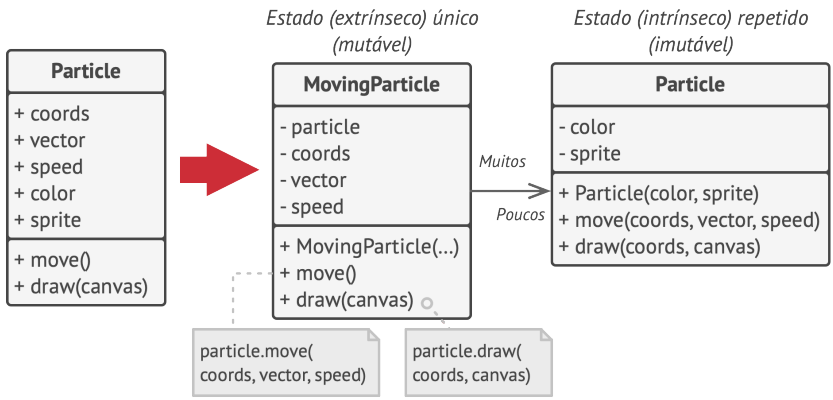
Ao completar, você sobe suas últimas mudanças, constrói o jogo e manda ele para um amigo para um test drive. Embora o jogo tenha rodado impecavelmente na sua máquina, seu amigo não foi capaz de jogar por muito tempo. No computador dele, o jogo continuava quebrando após alguns minutos de gameplay. Após algumas horas pesquisando nos registros do jogo você descobre que ele quebrou devido a uma quantidade insuficiente de RAM. Acontece que a máquina do seu amigo é muito menos poderosa que o seu computador, e é por isso que o problema apareceu facilmente na máquina dele.



O verdadeiro problema está relacionado ao seu sistema de partículas. Cada partícula, tais como uma bala, um míssil, ou um estilhaço era representado por um objeto separado contendo muita informação. Em algum momento, quando a destruição na tela do jogadora era tanta, as novas partículas criadas não cabiam mais no RAM restante, então o programa quebrava.

😊 Solução

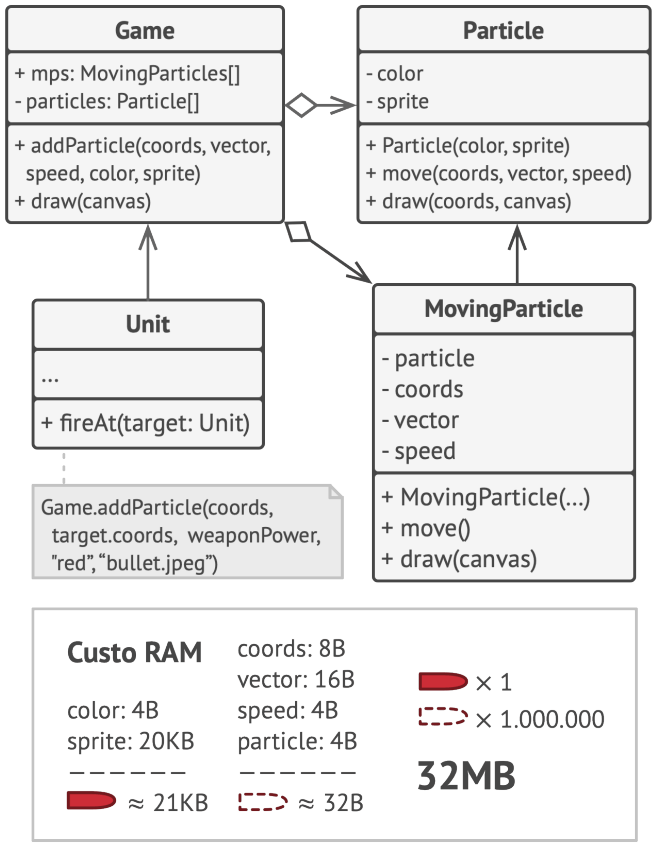
Olhando de perto a classe `Partícula`, você pode notar que a cor e o campo sprite consomem muita memória se comparado aos demais campos. E o pior é que esses dois campos armazenam dados quase idênticos para todas as partículas. Por exemplo, todas as balas têm a mesma cor e sprite.



Outras partes do estado de uma partícula, tais como coordenadas, vetor de movimento e velocidade, são únicos para cada partícula. Afinal de contas, todos os valores desses campos mudam com o tempo. Esses dados representam todo o con-

texto de mudança na qual a partícula existe, enquanto que a cor e o sprite permanecem constante para cada partícula.

Esse dado constante de um objeto é usualmente chamado de *estado intrínseco*. Ele vive dentro do objeto; outros objetos só podem lê-lo, não mudá-lo. O resto do estado do objeto, quase sempre alterado "pelo lado de fora" por outros objetos é chamado *estado extrínseco*.



O padrão Flyweight sugere que você pare de armazenar o estado extrínseco dentro do objeto. Ao invés disso, você deve passar esse estado para métodos específicos que dependem dele. Somente o estado intrínseco fica dentro do objeto, permitindo que você o reutilize em diferentes contextos. Como resultado, você vai precisar de menos desses objetos uma vez que eles diferem apenas em seu estado intrínseco, que tem menos variações que o extrínseco.

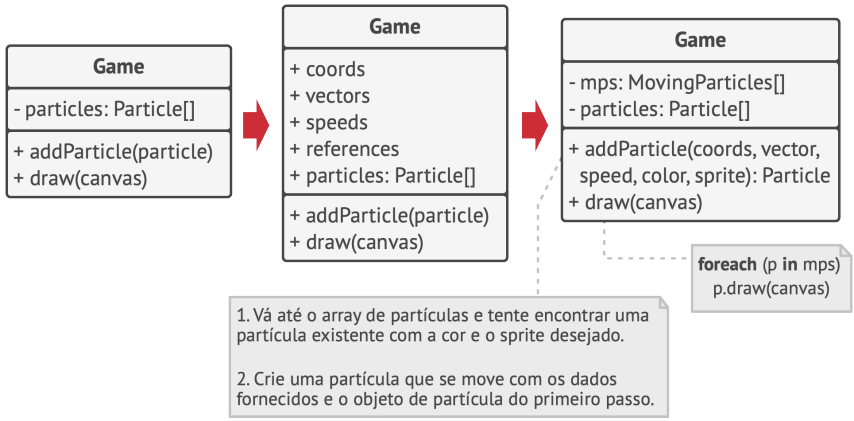
Vamos voltar ao nosso jogo. Assumindo que extraímos o estado extrínseco de nossa classe de partículas, somente três diferentes objetos serão suficientes para representar todas as partículas no jogo: uma bala, um míssil, e um pedaço de estilhaço. Como você provavelmente já adivinhou, um objeto que apenas armazena o estado intrínseco é chamado de um flyweight.

Armazenamento do estado extrínseco

Para onde vai o estado extrínseco então? Algumas classes ainda devem ser capazes de armazená-lo, certo? Na maioria dos casos, ele é movido para o objeto contêiner, que agrega os objetos antes de aplicarmos o padrão.

No nosso caso, esse seria o objeto principal `Jogo` que armazena todas as partículas no campo `partículas`. Para mover o estado extrínseco para essa classe você precisa criar diversos campos array para armazenar coordenadas, vetores, e a velocidade de cada partícula individual. Mas isso não é tudo.

Você vai precisar de outra array para armazenar referências ao flyweight específico que representa a partícula. Essas arrays devem estar em sincronia para que você possa acessar todos os dados de uma partícula usando o mesmo índice.



Uma solução mais elegante é criar uma classe de contexto separada que armazenaria o estado extrínseco junto com a referência para o objeto flyweight. Essa abordagem precisaria apenas de uma única array na classe contêiner.

Calma aí! Não vamos precisar de tantos objetos contextuais como tínhamos no começo? Tecnicamente, sim, mas nesse caso, esses objetos são muito menores que antes. Os campos mais pesados foram movidos para alguns poucos objetos flyweight. Agora, milhares de objetos contextuais podem reutilizar um único objeto flyweight pesado ao invés de armazenar milhares de cópias de seus dados.

Flyweight e a imutabilidade

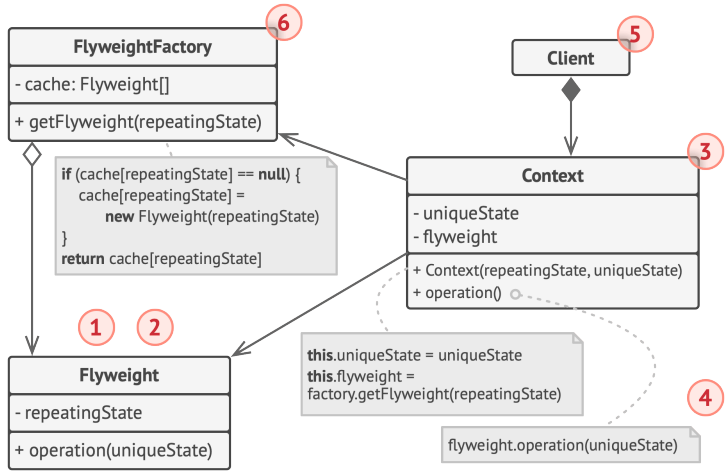
Já que o mesmo objeto flyweight pode ser usado em diferentes contextos, você tem que certificar-se que seu estado não pode ser modificado. Um flyweight deve inicializar seu estado apenas uma vez, através dos parâmetros do construtor. Ele não deve expor qualquer setter ou campos públicos para outros objetos.

Fábrica Flyweight

Para um acesso mais conveniente para vários flyweights, você pode criar um método fábrica que gerencia um conjunto de objetos flyweight existentes. O método aceita o estado intrínseco do flyweight desejado por um cliente, procura por um objeto flyweight existente que coincide com esse estado, e retorna ele se for encontrado. Se não for, ele cria um novo flyweight e o adiciona ao conjunto.

Há várias opções onde esse método pode ser colocado. O lugar mais óbvio é um contêiner de flyweights. Alternativamente você pode criar uma nova classe fábrica. Ou você pode fazer o método fábrica ser estático e colocá-lo dentro da própria classe do flyweight.

Estrutura



1. O padrão Flyweight é somente uma optimização. Antes de aplicá-lo, certifique-se que seu programa tem mesmo um problema de consumo de RAM relacionado a existência de múltiplos objetos similares na memória ao mesmo tempo. Certifique-se que o problema não possa ser resolvido por outra forma relevante.

2. A classe **Flyweight** contém a porção do estado do objeto original que pode ser compartilhada entre múltiplos objetos. O mesmo objeto flyweight pode ser usado em muitos contextos diferentes. O estado armazenado dentro de um flyweight é chamado "intrínseco". O estado passado pelos métodos flyweight é chamado "extrínseco".

3. A classe **Contexto** contém o estado extrínseco, único para todos os objetos originais. Quando um contexto é pareado com

um dos objetos flyweight, ele representa o estado completo do objeto original.

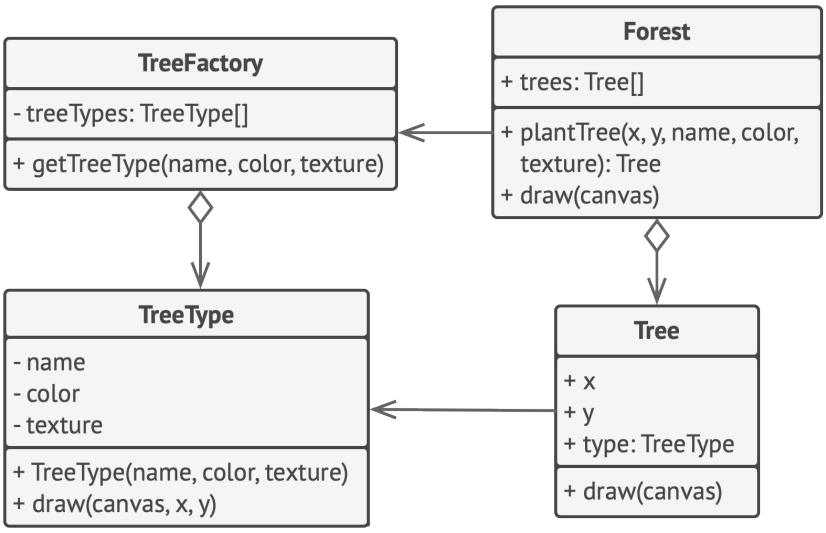
4. Geralmente, o comportamento do objeto original permanece na classe flyweight. Nesse caso, quem chamar o método do flyweight deve também passar os dados apropriados do estado extrínseco nos parâmetros do método. Por outro lado, o comportamento pode ser movido para a classe contexto, que usaria o flyweight meramente como um objeto de dados.

5. O **Cliente** calcula ou armazena o estado extrínseco dos flyweights. Da perspectiva do cliente, um flyweight é um objeto modelo que pode ser configurado no momento da execução ao passar alguns dados de contexto nos parâmetros de seus métodos.

6. A **Fábrica Flyweight** gerencia um conjunto de flyweights existentes. Com a fábrica os clientes não precisam criar flyweights diretamente. Ao invés disso, eles chamam a fábrica, passam os dados de estado intrínseco para o flyweight desejado. A fábrica procura por flyweights já criados e então retorna um existe que coincide o critério de busca ou cria um novo se nada for encontrado.

Pseudocódigo

Neste exemplo o padrão **Flyweight** ajuda a reduzir o uso de memória quando renderizando milhões de objetos árvore em uma tela.



O padrão extrai o estado intrínseco repetido para um classe `Árvore` principal e o move para dentro da classe flyweight `TipoÁrvore`.

Agora ao invés de armazenar os mesmos dados em múltiplos objetos, ele armazena apenas alguns objetos flyweight e os liga aos objetos `Árvore` apropriados que agem como contexto. O código cliente cria novos objetos árvore usando a fábrica flyweight, que encapsula a complexidade de busca pelo objeto correto e o reutiliza se necessário.

```
// A classe flyweight contém uma parte do estado de uma árvore
// Esses campos armazenam valores que são únicos para cada
// árvore em particular. Por exemplo, você não vai encontrar
// coordenadas da árvore aqui. Já que esses dados geralmente são
// GRANDES, você gastaria muita memória mantendo-os em cada
```

```
// objeto árvore. Ao invés disso, nós podemos extrair a textura,
// cor e outros dados repetitivos em um objeto separado os quais
// muitas árvores individuais podem referenciar.
class TreeType is
  field name
  field color
  field texture
  constructor TreeType(name, color, texture) { ... }
  method draw(canvas, x, y) is
    // 1. Cria um bitmap de certo tipo, cor e textura.
    // 2. Desenha o bitmap em uma tela nas coordenadas X e
    // Y.

// A fábrica flyweight decide se reutiliza o flyweight existente
// ou cria um novo objeto.
class TreeFactory is
  static field treeTypes: collection of tree types
  static method getTreeType(name, color, texture) is
    type = treeTypes.find(name, color, texture)
    if (type == null)
      type = new TreeType(name, color, texture)
      treeTypes.add(type)
    return type

// O objeto contextual contém a parte extrínseca do estado da
// árvore. Uma aplicação pode criar bilhões desses estados, já
// que são muito pequenos:
// apenas dois números inteiros para coordenadas e um campo de
// referência.
class Tree is
  field x,y
  field type: TreeType
```

```
  constructor Tree(x, y, type) { ... }
  method draw(canvas) is
    type.draw(canvas, this.x, this.y)

// As classes Tree (Árvore) e Forest (Floresta) são os clientes
// flyweight. Você pode uni-las se não planeja desenvolver mais
// a classe Tree.
class Forest is
  field trees: collection of Trees

  method plantTree(x, y, name, color, texture) is
    type = TreeFactory.getTreeType(name, color, texture)
    tree = new Tree(x, y, type)
    trees.add(tree)

  method draw(canvas) is
    foreach (tree in trees) do
      tree.draw(canvas)
```

Aplicabilidade

Utilize o padrão Flyweight apenas quando seu programa deve suportar um grande número de objetos que mal cabem na RAM disponível.

O benefício de aplicar o padrão depende muito de como e onde ele é usado. Ele é mais útil quando:

- uma aplicação precisa gerar um grande número de objetos similares

- isso drena a RAM disponível no dispositivo alvo
- os objetos contém estados duplicados que podem ser extraídos e compartilhados entre múltiplos objetos

Como implementar

1. Divida os campos de uma classe que irá se tornar um flyweight em duas partes:

    - o estado intrínseco: os campos que contém dados imutáveis e duplicados para muitos objetos
    - o estado extrínseco: os campos que contém dados contextuais únicos para cada objeto

2. Deixe os campos que representam o estado intrínseco dentro da classe, mas certifique-se que eles sejam imutáveis. Eles só podem obter seus valores iniciais dentro do construtor.

3. Examine os métodos que usam os campos do estado extrínseco. Para cada campo usado no método, introduza um novo parâmetro e use-o ao invés do campo.

4. Opcionalmente, crie uma classe fábrica para gerenciar o conjunto de flyweights. Ela deve checar por um flyweight existente antes de criar um novo. Uma vez que a fábrica está rodando, os clientes devem pedir flyweights apenas através dela. Eles devem descrever o flyweight desejado ao passar o estado intrínseco para a fábrica.

5. O cliente deve armazenar ou calcular valores para o estado extrínseco (contexto) para ser capaz de chamar métodos de objetos flyweight. Por conveniência, o estado extrínseco junto com o campo de referência flyweight podem ser movidos para uma classe de contexto separada.

Prós e contras

✓ Você pode economizar muita RAM, desde que seu programa tenha muitos objetos similares.

✕ Você pode estar trocando RAM por ciclos de CPU quando parte dos dados de contexto precisa ser recalculado cada vez que alguém chama um método flyweight.

✕ O código fica muito mais complicado. Novos membros de equipe sempre se perguntarão por que o estado de uma entidade foi separado de tal forma.

Relações com outros padrões

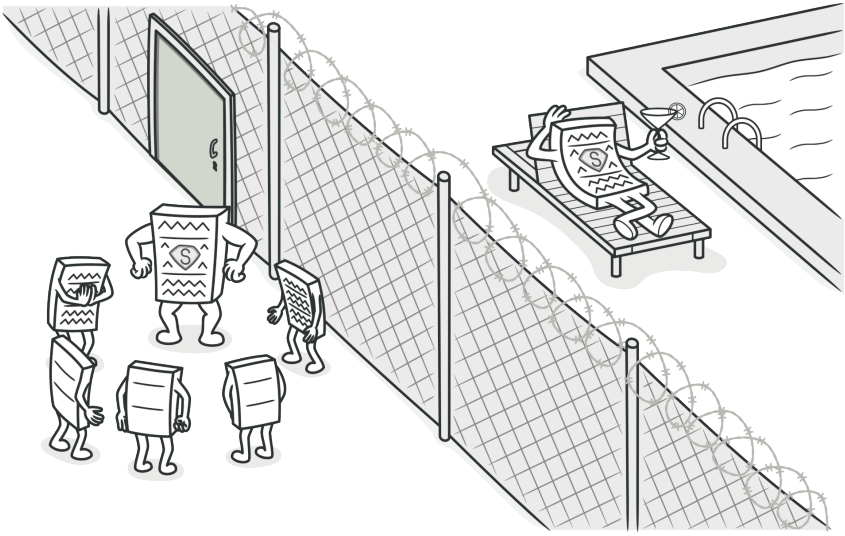
- Você pode implementar nós folha compartilhados da árvore **Composite** como **Flyweights** para salvar RAM.

- O **Flyweight** mostra como fazer vários pequenos objetos, enquanto o **Facade** mostra como fazer um único objeto que represente um subsistema inteiro.

- O **Flyweight** seria parecido com o **Singleton** se você, de algum modo, reduzisse todos os estados de objetos compartilhados

para apenas um objeto flyweight. Mas há duas mudanças fundamentais entre esses padrões:

1. Deve haver apenas uma única instância singleton, enquanto que uma classe *flyweight* pode ter múltiplas instâncias com diferentes estados intrínsecos.
2. O objeto *singleton* pode ser mutável. Objetos flyweight são imutáveis.

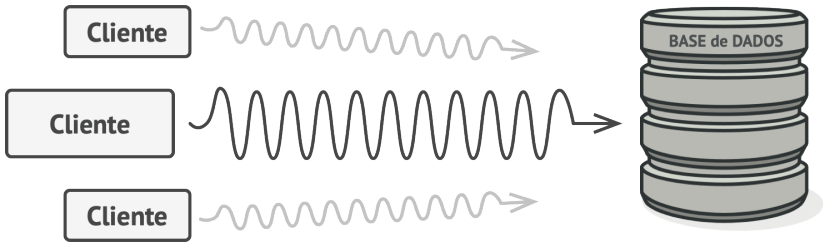


PROXY

O **Proxy** é um padrão de projeto estrutural que permite que você forneça um substituto ou um espaço reservado para outro objeto. Um proxy controla o acesso ao objeto original, permitindo que você faça algo ou antes ou depois do pedido chegar ao objeto original.

☹ Problema

Por que eu iria querer controlar o acesso a um objeto? Aqui está um exemplo: você tem um objeto grande que consome muitos recursos do sistema. Você precisa dele de tempos em tempos, mas não sempre.



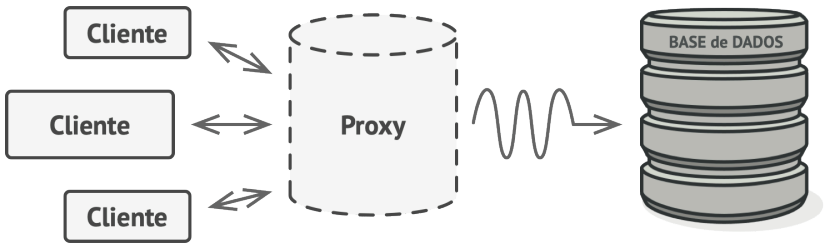
Solicitações para bases de dados podem ser bem lentas.

Você poderia implementar uma inicialização preguiçosa: criar esse objeto apenas quando ele é realmente necessário. Todos os clientes do objeto teriam que executar algum código adiado de inicialização. Infelizmente, isso provavelmente resultaria em muito código duplicado.

Em um mundo ideal, gostaríamos que você colocasse esse código diretamente dentro da classe do nosso objeto, mas isso nem sempre é possível. Por exemplo, a classe pode fazer parte de uma biblioteca fechada de terceiros.

😊 Solução

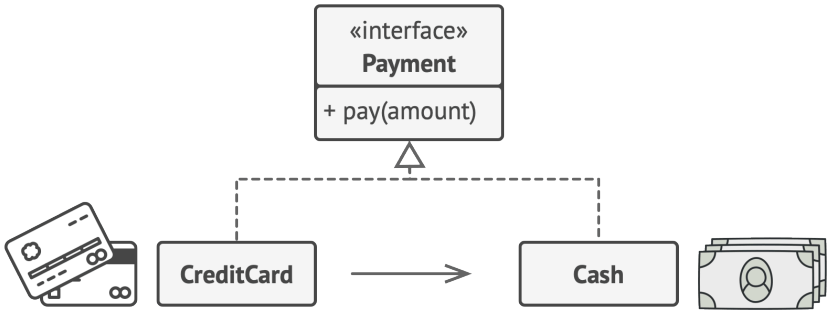
O padrão Proxy sugere que você crie uma nova classe proxy com a mesma interface do objeto do serviço original. Então você atualiza sua aplicação para que ela passe o objeto proxy para todos os clientes do objeto original. Ao receber uma solicitação de um cliente, o proxy cria um objeto do serviço real e delega a ele todo o trabalho.



O proxy se disfarça de objeto de base de dados. Ele pode lidar com inicializações preguiçosas e caches de resultados sem que o cliente ou a base de dados fiquem sabendo.

Mas qual é o benefício? Se você precisa executar alguma coisa tanto antes como depois da lógica primária da classe, o proxy permite que você faça isso sem mudar aquela classe. Uma vez que o proxy implementa a mesma interface que a classe original, ele pode ser passado para qualquer cliente que espera um objeto do serviço real.

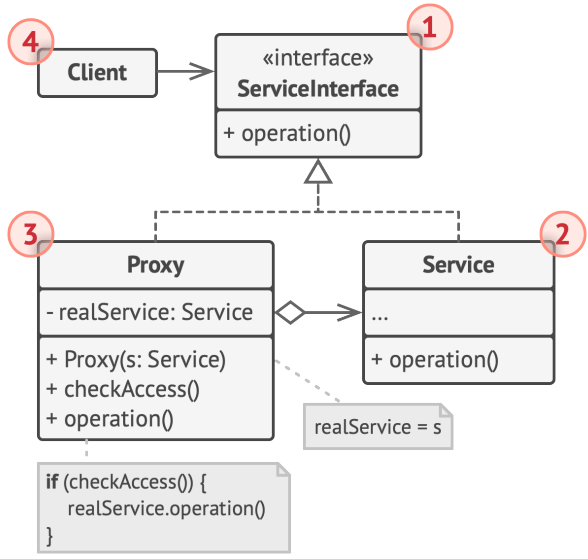
Analogia com o mundo real



Cartões de crédito podem ser usados como pagamentos assim como o dinheiro.

Um cartão de crédito é um proxy para uma conta bancária, que é um proxy para uma porção de dinheiro. Ambos implementam a mesma interface porque não há necessidade de carregar uma porção de dinheiro por aí. Um cliente se sente bem porque não precisa ficar carregando montanhas de dinheiro por aí. Um dono de loja também fica feliz uma vez que a renda da transação é adicionada eletronicamente para sua conta sem o risco de perdê-la no depósito ou de ser roubado quando estiver indo ao banco.

Estrutura



1. A **Interface do Serviço** declara a interface do Serviço. O proxy deve seguir essa interface para ser capaz de se disfarçar como um objeto do serviço.

2. O **Serviço** é uma classe que fornece alguma lógica de negócio útil.

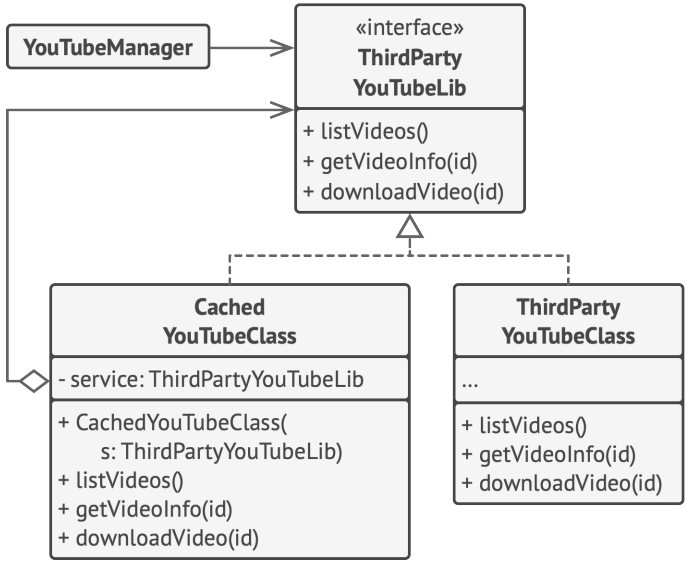
3. A classe **Proxy** tem um campo de referência que aponta para um objeto do serviço. Após o proxy finalizar seu processamento (por exemplo: inicialização preguiçosa, acesso, acessar controle, colocar em cache, etc.), ele passa o pedido para o objeto do serviço.

Geralmente os proxies gerenciam todo o ciclo de vida dos seus objetos de serviço.

4. O **Cliente** deve trabalhar tanto com os serviços e proxies através da mesma interface. Dessa forma você pode passar uma proxy para qualquer código que espera um objeto do serviço.

Pseudocódigo

Este exemplo ilustra como o padrão **Proxy** pode ajudar a introduzir uma inicialização preguiçosa e cache para um biblioteca de integração terceirizada do YouTube.



Colocando em cache os resultados de um serviço com um proxy.

A biblioteca fornece a nós com uma classe de download de vídeo. Contudo, ela é muito ineficiente. Se a aplicação cli-

ente pedir o mesmo vídeo múltiplas vezes, a biblioteca apenas baixa de novo e de novo, ao invés de colocar ele em cache e reutilizar o primeiro arquivo de download.

A classe proxy implementa a mesma interface que a classe baixadora original e delega-a todo o trabalho. Contudo, ela mantém um registro dos arquivos baixados e retorna o resultado em cache quando a aplicação pede o mesmo vídeo múltiplas vezes.

```
// A interface de um serviço remoto.
interface ThirdPartyYouTubeLib is
  method listVideos()
  method getVideoInfo(id)
  method downloadVideo(id)

// A implementação concreta de um serviço conector. Métodos
// dessa classe podem pedir informações do YouTube. A velocidade
// do pedido depende da conexão do usuário com a internet, bem
// como do YouTube. A aplicação irá ficar lenta se muitos
// pedidos forem feitos ao mesmo tempo, mesmo que todos peçam a
// mesma informação.
class ThirdPartyYouTubeClass implements ThirdPartyYouTubeLib is
  method listVideos() is
    // Envia um pedido API para o YouTube.

  method getVideoInfo(id) is
    // Obtém metadados sobre algum vídeo.

  method downloadVideo(id) is
```

```
    // Baixa um arquivo de vídeo do YouTube.

// Para salvar largura de banda, nós podemos colocar os
// resultados do pedido em cache e mantê-los por determinado
// tempo. Mas pode ser impossível colocar tal código diretamente
// na classe de serviço. Por exemplo, ele pode ter sido
// fornecido como parte de uma biblioteca de terceiros e/ou
// definida como `final`. É por isso que nós colocamos o código
// do cache em uma nova classe proxy que implementa a mesma
// interface que a classe de serviço. Ela delega ao objeto do
// serviço somente quando os pedidos reais foram enviados.
class CachedYouTubeClass implements ThirdPartyYouTubeLib is
  private field service: ThirdPartyYouTubeLib
  private field listCache, videoCache
  field needReset

  constructor CachedYouTubeClass(service: ThirdPartyYouTubeLib) is
    this.service = service

  method listVideos() is
    if (listCache == null || needReset)
      listCache = service.listVideos()
    return listCache

  method getVideoInfo(id) is
    if (videoCache == null || needReset)
      videoCache = service.getVideoInfo(id)
    return videoCache

  method downloadVideo(id) is
    if (!downloadExists(id) || needReset)
      service.downloadVideo(id)
```

```
53
54  // A classe GUI, que é usada para trabalhar diretamente com um
55  // objeto de serviço, permanece imutável desde que trabalhe com
56  // o objeto de serviço através de uma interface. Nós podemos
57  // passar um objeto proxy com segurança ao invés de um objeto
58  // real de serviço uma vez que ambos implementam a mesma
59  // interface.
60  class YouTubeManager is
61    protected field service: ThirdPartyYouTubeLib
62
63    constructor YouTubeManager(service: ThirdPartyYouTubeLib) is
64      this.service = service
65
66    method renderVideoPage(id) is
67      info = service.getVideoInfo(id)
68      // Renderiza a página do vídeo.
69
70    method renderListPanel() is
71      list = service.listVideos()
72      // Renderiza a lista de miniaturas do vídeo.
73
74    method reactOnUserInput() is
75      renderVideoPage()
76      renderListPanel()
77
78  // A aplicação pode configurar proxies de forma fácil e rápida.
79  class Application is
80    method init() is
81      aYouTubeService = new ThirdPartyYouTubeClass()
82      aYouTubeProxy = new CachedYouTubeClass(aYouTubeService)
83      manager = new YouTubeManager(aYouTubeProxy)
84      manager.reactOnUserInput()
```

Aplicabilidade

Há dúzias de maneiras de utilizar o padrão Proxy. Vamos ver os usos mais populares.

Inicialização preguiçosa (proxy virtual). Este é quando você tem um objeto do serviço peso-pesado que gasta recursos do sistema por estar sempre rodando, mesmo quando você precisa dele de tempos em tempos.

Ao invés de criar um objeto quando a aplicação inicializa, você pode atrasar a inicialização do objeto para um momento que ele é realmente necessário.

Controle de acesso (proxy de proteção). Este é quando você quer que apenas clientes específicos usem o objeto do serviço; por exemplo, quando seus objetos são partes cruciais de um sistema operacional e os clientes são várias aplicações iniciadas (incluindo algumas maliciosas).

O proxy pode passar o pedido para o objeto de serviço somente se as credenciais do cliente coincidem com certos critérios.

Execução local de um serviço remoto (proxy remoto). Este é quando o objeto do serviço está localizado em um servidor remoto.

- Neste caso, o proxy passa o pedido do cliente pela rede, lidando com todos os detalhes sujos pertinentes a se trabalhar com a rede.

- **Registros de pedidos (proxy de registro). Este é quando você quer manter um histórico de pedidos ao objeto do serviço.**

- O proxy pode fazer o registro de cada pedido antes de passar ao serviço.

- **Cache de resultados de pedidos (proxy de cache). Este é quando você precisa colocar em cache os resultados de pedidos do cliente e gerenciar o ciclo de vida deste cache, especialmente se os resultados são muito grandes.**

- O proxy pode implementar o armazenamento em cache para pedidos recorrentes que sempre acabam nos mesmos resultados. O proxy pode usar como parâmetros dos pedidos as chaves de cache.

- **Referência inteligente. Este é para quando você precisa ser capaz de se livrar de um objeto peso-pesado assim que não há mais clientes que o usam.**

- O proxy pode manter um registro de clientes que obtiveram uma referência ao objeto serviço ou seus resultados. De tempos em tempos, o proxy pode verificar com os clientes se eles ainda estão ativos. Se a lista cliente ficar vazia, o proxy pode

remover o objeto serviço e liberar os recursos de sistema que ficaram empatados.

O proxy pode também fiscalizar se o cliente modificou o objeto do serviço. Então os objetos sem mudança podem ser reutilizados por outros clientes.

Como implementar

1. Se não há uma interface do serviço pré existente, crie uma para fazer os objetos proxy e serviço intercomunicáveis. Extrair a interface da classe serviço nem sempre é possível, porque você precisaria mudar todos os clientes do serviço para usar aquela interface. O plano B é fazer do proxy uma subclasse da classe serviço e, dessa forma, ele herdará a interface do serviço.

2. Crie a classe proxy. Ela deve ter um campo para armazenar uma referência ao serviço. Geralmente proxies criam e gerenciam todo o ciclo de vida de seus serviços. Em raras ocasiões, um serviço é passado ao proxy através do construtor pelo cliente.

3. Implemente os métodos proxy de acordo com o propósito deles. Na maioria dos casos, após realizar algum trabalho, o proxy deve delegar o trabalho para o objeto do serviço.

4. Considere introduzir um método de criação que decide se o cliente obtém um proxy ou serviço real. Isso pode ser um simples

método estático na classe do proxy ou um método factory todo implementado.

5. Considere implementar uma inicialização preguiçosa para o objeto do serviço.

Prós e contras

✓ Você pode controlar o objeto do serviço sem os clientes ficarem sabendo.

✓ Você pode gerenciar o ciclo de vida de um objeto do serviço quando os clientes não se importam mais com ele.

✓ O proxy trabalha até mesmo se o objeto do serviço ainda não está pronto ou disponível.

✓ *Princípio aberto/fechado*. Você pode introduzir novos proxies sem mudar o serviço ou clientes.

✕ O código pode ficar mais complicado uma vez que você precisa introduzir uma série de novas classes.

✕ A resposta de um serviço pode ter atrasos.

Relações com outros padrões

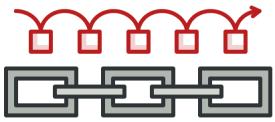
- O **Adapter** fornece uma interface diferente para um objeto encapsulado, o **Proxy** fornece a ele a mesma interface, e o **Decorator** fornece a ele com uma interface melhorada.

- O **Facade** é parecido como o **Proxy** no quesito que ambos colocam em buffer uma entidade complexa e inicializam ela sozinhos. Ao contrário do *Facade*, o *Proxy* tem a mesma interface que seu objeto de serviço, o que os torna intermutáveis.

- O **Decorator** e o **Proxy** têm estruturas semelhantes, mas propósitos muito diferentes. Alguns padrões são construídos no princípio de composição, onde um objeto deve delegar parte do trabalho para outro. A diferença é que o *Proxy* geralmente gerencia o ciclo de vida de seu objeto serviço por conta própria, enquanto que a composição do *decoradores* é sempre controlada pelo cliente.

Padrões de projeto comportamentais

Padrões comportamentais são voltados aos algoritmos e a designação de responsabilidades entre objetos.



Chain of Responsibility

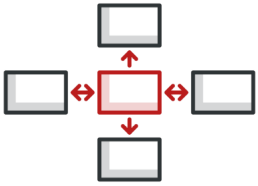
Permite que você passe pedidos por uma corrente de handlers. Ao receber um pedido, cada handler decide se processa o pedido ou passa para o próximo handler da corrente.

Command

Transforma o pedido em um objeto independente que contém toda a informação sobre o pedido. Essa Transformação permite que você parametrize métodos com diferentes pedidos, atrase ou coloque a execução do pedido em uma fila, e suporte operações que não podem ser feitas.

Iterator

Permite que você percorra elementos de uma coleção sem expor as representações estruturais deles (lista, pilha, árvore, etc.)

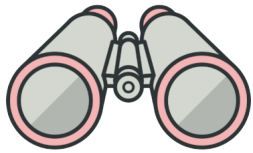


Mediator

Permite que você reduza as dependências caóticas entre objetos. O padrão restringe comunicações diretas entre objetos e os força a colaborar apenas através do objeto mediador.

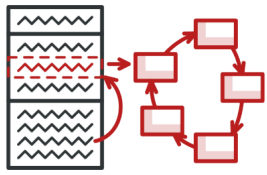
Memento

Permite que você salve e restaure o estado anterior de um objeto sem revelar os detalhes de sua implementação.



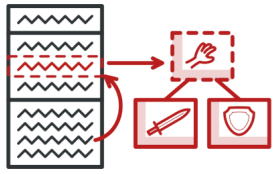
Observer

Permite que você defina um mecanismo de assinatura para notificar múltiplos objetos sobre quaisquer eventos que aconteçam com o objeto que eles estão observando.



State

Permite que um objeto altere seu comportamento quando seu estado interno muda. Parece como se o objeto mudasse de classe.

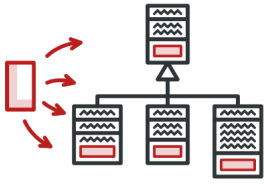


Strategy

Permite que você defina uma família de algoritmos, coloque-os em classes separadas, e faça os objetos deles intercambiáveis.

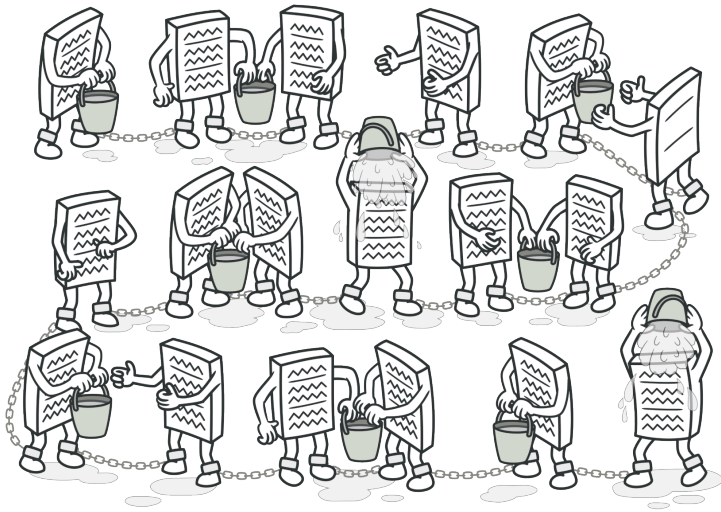
Template Method

Define o esqueleto de um algoritmo na superclasse mas deixa as subclasses sobrescreverem etapas específicas do algoritmo sem modificar sua estrutura.



Visitor

Permite que você separe algoritmos dos objetos nos quais eles operam.



CHAIN OF RESPONSIBILITY

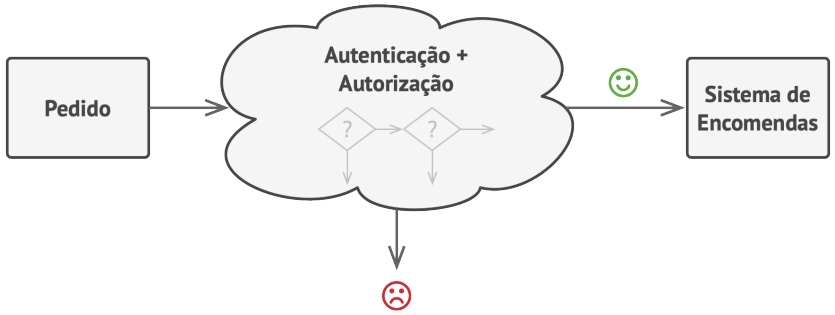
Também conhecido como: CoR, Corrente de responsabilidade, Corrente de comando, Chain of command

O **Chain of Responsibility** é um padrão de projeto comportamental que permite que você passe pedidos por uma corrente de handlers. Ao receber um pedido, cada handler decide se processa o pedido ou o passa adiante para o próximo handler na corrente.

☹ Problema

Imagine que você está trabalhando em um sistema de encomendas online. Você quer restringir o acesso ao sistema para que apenas usuários autenticados possam criar pedidos. E também somente usuários que tem permissões administrativas devem ter acesso total a todos os pedidos.

Após um pouco de planejamento, você se dá conta que essas checagens devem ser feitas sequencialmente. A aplicação pode tentar autenticar um usuário ao sistema sempre que receber um pedido que contém as credenciais do usuário. Contudo, se essas credenciais não estão corretas e a autenticação falha, não há razão para continuar com outras checagens.



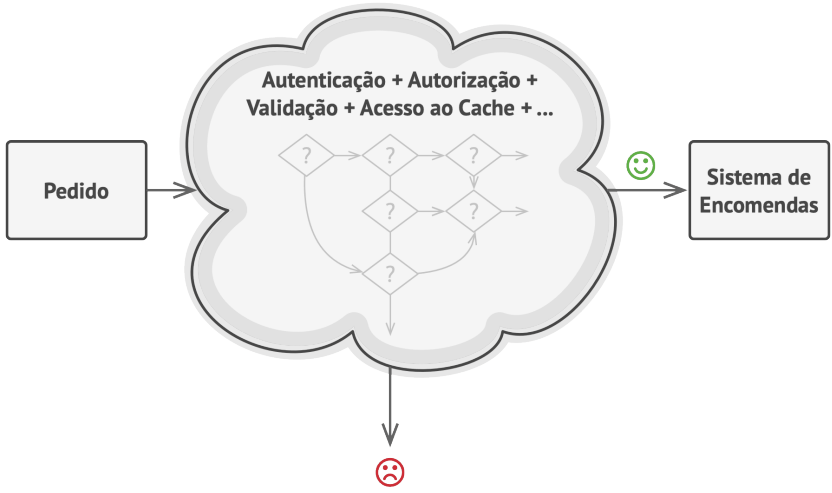
O pedido deve passar por uma série de checagens antes do sistema de encomendas possa lidar ele mesmo com o pedido.

Durante os próximos meses você implementou diversas mais daquelas checagens sequenciais.

- Um de seus colegas sugeriu que não é seguro passar dados brutos diretamente para o sistema de encomendas. Então você adicionou uma etapa adicional de validação para limpar os dados no pedido.

- Mais tarde, alguém notou que o sistema é vulnerável à ataques de força bruta. Para evitar isso, você prontamente adicionou uma checagem que filtra repetidas falhas vindas do mesmo endereço de IP.

- Outra pessoa sugeriu que você poderia agilizar o sistema se retornasse resultados de cache em pedidos repetidos contendo os mesmos dados. Portanto, você adicionou outra checagem que permite que o pedido passe através do sistema apenas se não há uma resposta adequada armazenada em cache.



Quanto mais o código cresce, mais bagunçado ele fica.

O código das checagens, que já parecia uma bagunça, ficou mais e mais inchado a medida que você foi adicionando novas funcionalidades. Mudar uma checagem às vezes afetava outras. E o pior de tudo, quando você tentou reutilizar as checagens para proteger os componentes do sistema, você teve que duplicar parte do código uma vez que esses componentes precisavam de algumas dessas checagens, mas nem todos eles.

O sistema ficou muito difícil de compreender e caro de se manter. Você lutou com o código por um tempo, até que um dia você decidiu refatorar a coisa toda.

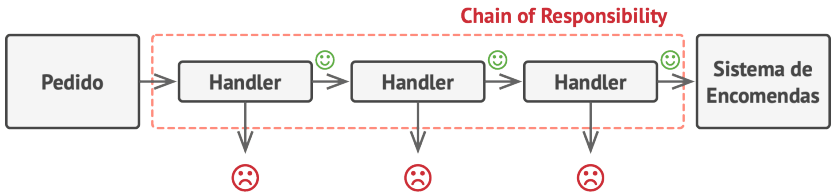
😊 Solução

Como muitos outros padrões de projeto comportamental, o **Chain of Responsibility** se baseia em transformar certos comportamentos em objetos solitários chamados *handlers*. No nosso caso, cada checagem devem ser extraída para sua própria classe com um único método que faz a checagem. O pedido, junto com seus dados, é passado para esse método como um argumento.

O padrão sugere que você ligue esses handlers em uma corrente. Cada handler ligado tem um campo para armazenar uma referência ao próximo handler da corrente. Além de processar o pedido, handlers o passam adiante na corrente. O pedido viaja através da corrente até que todos os handlers tiveram uma chance de processá-lo.

E aqui está a melhor parte: um handler pode decidir não passar o pedido adiante na corrente e efetivamente parar qualquer futuro processamento.

Em nosso exemplo com sistema de encomendas, um handler realiza o processamento e então decide se passa o pedido adiante na corrente ou não. Assumindo que o pedido contenha os dados adequados, todos os handlers podem executar seu comportamento principal, seja ele uma checagem de autenticação ou armazenamento em cache.

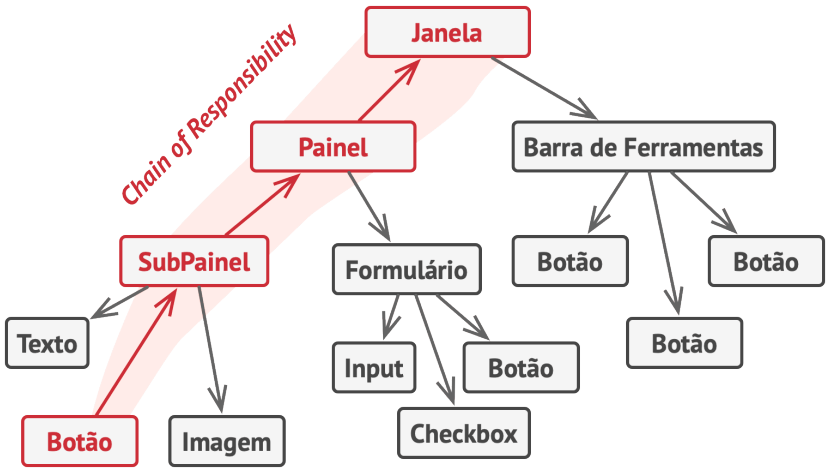


Handlers estão alinhados um a um, formando uma corrente.

Contudo, há uma abordagem ligeiramente diferente (e um tanto quanto canônica) na qual, ao receber o pedido, um handler decide se ele pode processá-lo ou não. Se ele pode, ele não passa o pedido adiante. Então é um handler que processa o pedido ou mais ninguém. Essa abordagem é muito comum quando lidando com eventos em pilha de elementos dentro de uma interface gráfica de usuário.

Por exemplo, quando um usuário clica um botão, o evento se propaga através da corrente de elementos GUI que começam com aquele botão, prossegue para seus contêineres (como planilhas ou painéis), e termina com a janela principal da

aplicação. O evento é processado pelo primeiro elemento na corrente que é capaz de lidar com ele. Esse exemplo também é notável porque ele mostra que uma corrente pode sempre ser extraída de um objeto árvore.



Uma corrente pode ser formada por uma secção de um objeto.

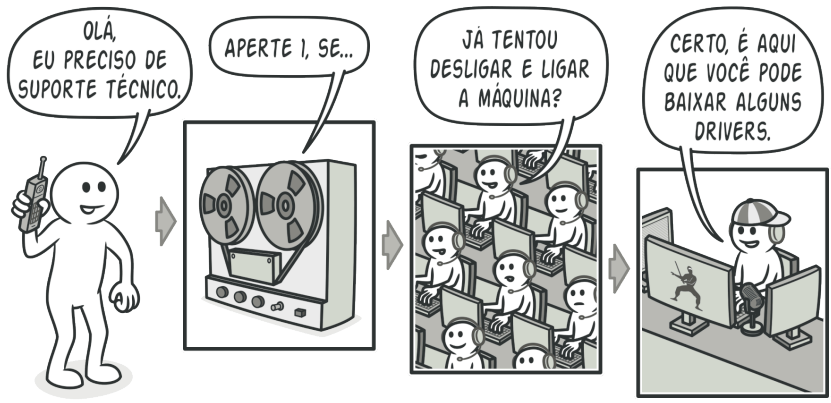
É crucial que todas as classes handler implementem a mesma interface. Cada handler concreto deve se importar apenas se o seguinte tem o método `executar`. Dessa maneira você pode compor correntes durante a execução, usando vários handlers sem acoplar seu código com suas classes concretas.

Analogia com o mundo real

Você acabou de comprar e instalar um novo hardware em seu computador. Como você é um geek, o computador tem diversos sistemas operacionais instalados. Você tenta ligar todos eles para ver se o hardware é suportado. O Windows detecta e ativa

o hardware automaticamente. Contudo, seu amado Linux se recusa a trabalhar com o novo hardware. Com uma pequena ponta de esperança, você decide ligar para o número do suporte técnico escrito na caixa.

A primeira coisa que você ouve é uma voz robótica do outro lado. Ela sugere nove soluções populares para vários problemas, nenhum dos quais é relevante para seu caso. Após um tempo, a voz robótica conecta você com um operador de carne e osso.

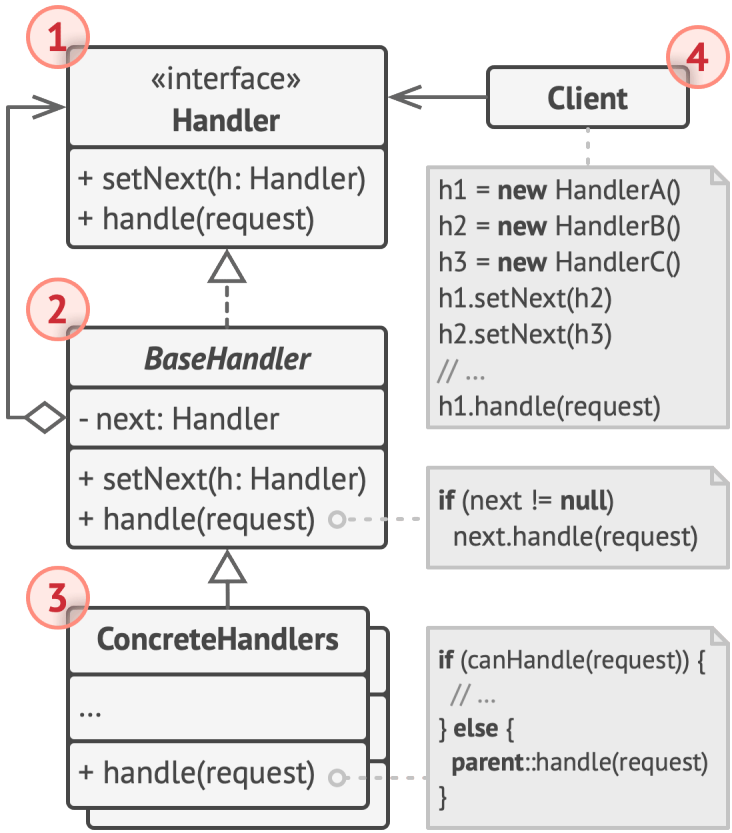


Uma chamada para o suporte técnico pode atravessar diversos operadores.

Infelizmente, o operador não foi capaz de sugerir algo específico também. Ele continuava recitando longos protocolos do manual, se recusando a escutar seus comentários. Após escutar a frase “você tentou desligar e ligar o computador” pela décima vez, você exige ser conectado a um engenheiro.

Eventualmente o operador passa sua chamada para um dos engenheiros, que estava ansioso por contato humano já que estava sentado por horas em sua escura sala do servidor no subsolo de algum prédio. O engenheiro lhe diz onde baixar os drivers apropriados para seu novo hardware e como instalá-los no Linux. Finalmente, a solução! Você termina sua chamada, transbordando de alegria.

Estrutura



1. O **Handler** declara a interface, comum a todos os handlers concretos. Ele geralmente contém apenas um único método para lidar com pedidos, mas algumas vezes ele pode conter outro método para configurar o próximo handler da corrente.

2. O **Handler Base** é uma classe opcional onde você pode colocar o código padrão que é comum a todas as classes handler.

   Geralmente, essa classe define um campo para armazenar uma referência para o próximo handler. Os clientes podem construir uma corrente passando um handler para o construtor ou setter do handler anterior. A classe pode também implementar o comportamento padrão do handler: pode passar a execução para o próximo handler após checar por sua existência.

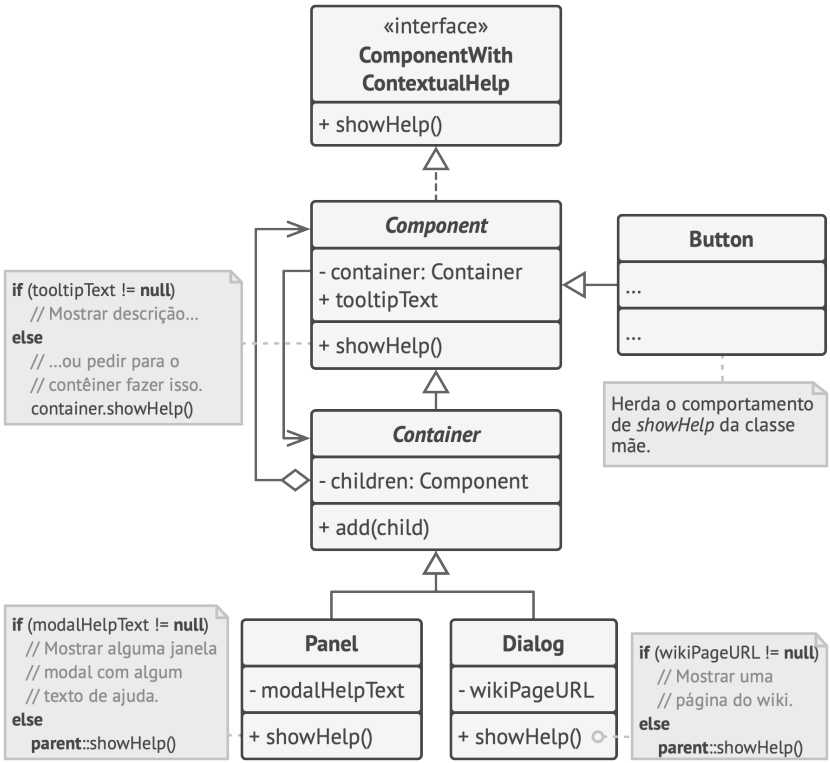
3. **Handlers Concretos** contém o código real para processar pedidos. Ao receber um pedido, cada handler deve decidir se processa ele e, adicionalmente, se passa ele adiante na corrente.

   Os handlers são geralmente auto contidos e imutáveis, aceitando todos os dados necessários apenas uma vez através do construtor.

4. O **Cliente** pode compor correntes apenas uma vez ou compô-las dinamicamente, dependendo da lógica da aplicação. Note que um pedido pode ser enviado para qualquer handler na corrente—não precisa ser ao primeiro.

Pseudocódigo

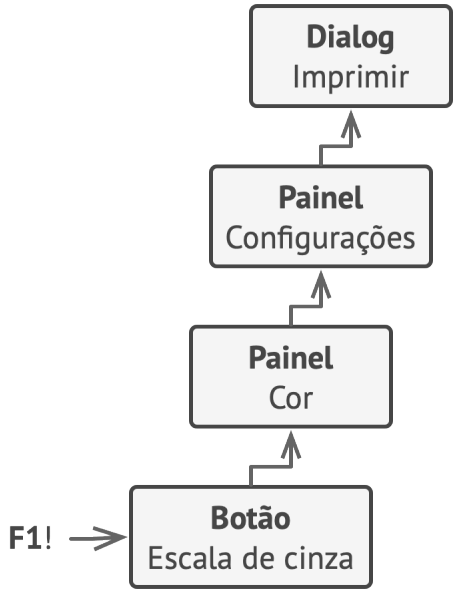
Neste exemplo, o padrão **Chain of Responsibility** é responsável por mostrar informação de ajuda contextual para elementos de GUI ativos.



As classes de interface do usuário são construídas com o padrão Composite. Cada elemento é ligado com seu elemento contêiner. A qualquer momento, você pode construir uma corrente de elementos que começa com o elemento em si e vai através de todos os elementos do seu contêiner.

O GUI da aplicação é geralmente estruturado como uma árvore de objetos. Por exemplo, a classe `Dialog`, que renderiza a janela principal da aplicação, seria a raíz do objeto árvore. O dialog contém `Painéis`, que podem conter outros painéis ou simplesmente elementos de baixo nível como `Botões` e `CamposDeTexto`.

Um componente simples pode mostrar descrições contextuais breves, desde que o componente tenha um texto de ajuda assinalado. Mas componentes mais complexos definem seu próprio modo de mostrar ajuda contextual, tais como mostrar um pedaço do manual ou abrir uma página de navegador.



Assim é como um pedido de ajuda atravessa os objetos GUI.

Quando um usuário aponta o cursor do mouse para um elemento e aperta a tecla `F1`, a aplicação detecta o componente abaixo do cursor e manda um pedido de ajuda. O pedido atravessa todos os elementos do contêiner até chegar no elemento capaz de mostrar a informação de ajuda.

```
// A interface do handler declara um método para a construção da
// corrente de handlers. Ela também declara um método para
// executar um pedido.
interface ComponentWithContextualHelp is
  method showHelp()


// A classe base para componentes simples.
abstract class Component implements ComponentWithContextualHelp is
  field tooltipText: string

  // O contêiner do componente age como o próximo elo na
  // corrente de handlers.
  protected field container: Container

  // O componente mostra um tooltip (dica de contexto) se há
  // algum texto de ajuda assinalado a ele. Do contrário ele
  // passa a chamada adiante ao contêiner, se ele existir.
  method showHelp() is
    if (tooltipText != null)
      // Mostrar dica de contexto.
    else
      container.showHelp()


// Contêineres podem conter tanto componentes simples como
// outros contêineres como filhos. As relações da corrente são
// definidas aqui. A classe herda o comportamento showHelp de
// sua mãe.
abstract class Container extends Component is
  protected field children: array of Component

```

```
    method add(child) is
      children.add(child)
      child.container = this


// Componentes primitivos estão de bom tamanho com a
// implementação de ajuda padrão.
class Button extends Component is
  // ...

// Mas componentes complexos podem sobrescrever a implementação
// padrão. Se o texto de ajuda não pode ser fornecido de uma
// nova maneira, o componente pode sempre chamar a implementação
// base (veja a classe Component).
class Panel extends Container is
  field modalHelpText: string

  method showHelp() is
    if (modalHelpText != null)
      // Mostra uma janela modal com texto de ajuda.
    else
      super.showHelp()

// ...o mesmo que acima...
class Dialog extends Container is
  field wikiPageURL: string

  method showHelp() is
    if (wikiPageURL != null)
      // Abre a página de ajuda do wiki.
    else
      super.showHelp()
```

```
65
66
67  // Código cliente.
68  class Application is
69    // Cada aplicação configura a corrente de forma diferente.
70    method createUI() is
71      dialog = new Dialog("Budget Reports")
72      dialog.wikiPageURL = "http://..."
73      panel = new Panel(0, 0, 400, 800)
74      panel.modalHelpText = "This panel does..."
75      ok = new Button(250, 760, 50, 20, "OK")
76      ok.tooltipText = "This is an OK button that..."
77      cancel = new Button(320, 760, 50, 20, "Cancel")
78      // ...
79      panel.add(ok)
80      panel.add(cancel)
81      dialog.add(panel)
82
83    // Imagine o que acontece aqui.
84    method onF1KeyPress() is
85      component = this.getComponentAtMouseCoords()
86      component.showHelp()
```

Aplicabilidade

Utilize o padrão Chain of Responsibility quando é esperado que seu programa processe diferentes tipos de pedidos em várias maneiras, mas os exatos tipos de pedidos e suas sequências são desconhecidos de antemão.

O padrão permite que você ligue vários handlers em uma corrente e, ao receber um pedido, perguntar para cada handler se ele pode ou não processá-lo. Dessa forma todos os handlers tem a chance de processar o pedido.

Utilize o padrão quando é essencial executar diversos handlers em uma ordem específica.

Já que você pode ligar os handlers em uma corrente em qualquer ordem, todos os pedidos irão atravessar a corrente exatamente como você planejou.

Utilize o padrão CoR quando o conjunto de handlers e suas encomendas devem mudar no momento de execução.

Se você providenciar setters para um campo de referência dentro das classes handler, você será capaz de inserir, remover, ou reordenar os handlers de forma dinâmica.

Como implementar

1. Declare a interface do handler e descreva a assinatura de um método para lidar com pedidos.

   Decida como o cliente irá passar os dados do pedido para o método. A maneira mais flexível é converter o pedido em um objeto e passá-lo para o método handler como um argumento.

2. Para eliminar código padrão duplicado nos handlers concretos, pode valer a pena criar uma classe handler base abstrata, derivada da interface do handler.

   Essa classe deve ter um campo para armazenar uma referência ao próximo handler na corrente. Considere tornar a classe imutável. Contudo, se você planeja modificar correntes no tempo de execução, você precisa definir um setter para alterar o valor do campo de referência.

   Você também pode implementar o comportamento padrão conveniente para o método handler, que vai passar adiante o pedido para o próximo objeto a não ser que não haja mais objetos. Handlers concretos irão ser capazes de usar esse comportamento ao chamar o método pai.

3. Um por um crie subclasses handler concretas e implemente seus métodos handler. Cada handler deve fazer duas decisões ao receber um pedido:

   - Se ele vai processar o pedido.
   - Se ele vai passar o pedido adiante na corrente.

4. O cliente pode tanto montar correntes sozinho ou receber correntes pré construídas de outros objetos. Neste último caso, você deve implementar algumas classes fábrica para construir correntes de acordo com a configuração ou definições de ambiente.

5. O cliente pode ativar qualquer handler da corrente, não apenas o primeiro. O pedido será passado ao longo da corrente até que algum handler se recuse a passá-lo adiante ou até ele chegar ao fim da corrente.

6. Devido a natureza dinâmica da corrente, o cliente deve estar pronto para lidar com os seguintes cenários:

   - A corrente pode consistir de um único elo.
   - Alguns pedidos podem não chegar ao fim da corrente.
   - Outros podem chegar ao fim da corrente sem terem sido tratados.

Prós e contras

✓ Você pode controlar a ordem de tratamento dos pedidos.

✓ *Princípio de responsabilidade única*. Você pode desacoplar classes que invocam operações de classes que realizam operações.

✓ *Princípio aberto/fechado*. Você pode introduzir novos handlers na aplicação sem quebrar o código cliente existente.

✕ Alguns pedidos podem acabar sem tratamento.

Relações com outros padrões

- O **Chain of Responsibility**, **Command**, **Mediator** e **Observer** abrangem várias maneiras de se conectar remetentes e destinatários de pedidos:

  - O *Chain of Responsibility* passa um pedido sequencialmente ao longo de um corrente dinâmica de potenciais destinatários até que um deles atua no pedido.
  - O *Command* estabelece conexões unidirecionais entre remetentes e destinatários.
  - O *Mediator* elimina as conexões diretas entre remetentes e destinatários, forçando-os a se comunicar indiretamente através de um objeto mediador.
  - O *Observer* permite que destinatários inscrevam-se ou cancelem sua inscrição dinamicamente para receber pedidos.

- O **Chain of Responsibility** é frequentemente usado em conjunto com o **Composite**. Neste caso, quando um componente folha recebe um pedido, ele pode passá-lo através de uma corrente de todos os componentes pai até a raiz do objeto árvore.

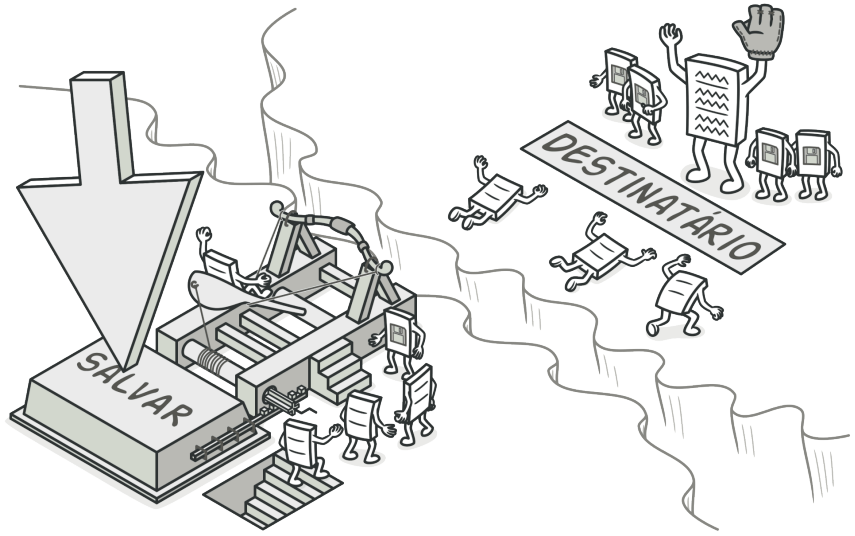
- Handlers em uma **Chain of Responsibility** podem ser implementados como **comandos**. Neste caso, você pode executar várias operações diferentes sobre o mesmo objeto contexto, representado por um pedido.

  Contudo, há outra abordagem, onde o próprio pedido é um objeto *comando*. Neste caso, você pode executar a mesma operação em uma série de diferentes contextos ligados em uma corrente.

- O **Chain of Responsibility** e o **Decorator** têm estruturas de classe muito parecidas. Ambos padrões dependem de compo-

sição recursiva para passar a execução através de uma série de objetos. Contudo, há algumas diferenças cruciais.

Os handlers do *CoR* podem executar operações arbitrárias independentemente uma das outras. Eles também podem parar o pedido de ser passado adiante em qualquer ponto. Por outro lado, vários *decoradores* podem estender o comportamento do objeto enquanto mantém ele consistente com a interface base. Além disso, os decoradores não tem permissão para quebrar o fluxo do pedido.



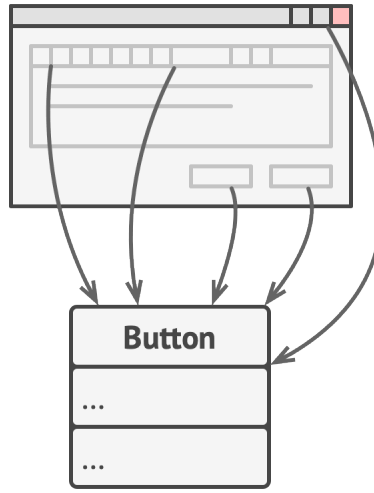
COMMAND

Também conhecido como: Comando, Ação, Action, Transação, Transaction

O **Command** é um padrão de projeto comportamental que transforma um pedido em um objeto independente que contém toda a informação sobre o pedido. Essa transformação permite que você parametrize métodos com diferentes pedidos, atrase ou coloque a execução do pedido em uma fila, e suporte operações que não podem ser feitas.

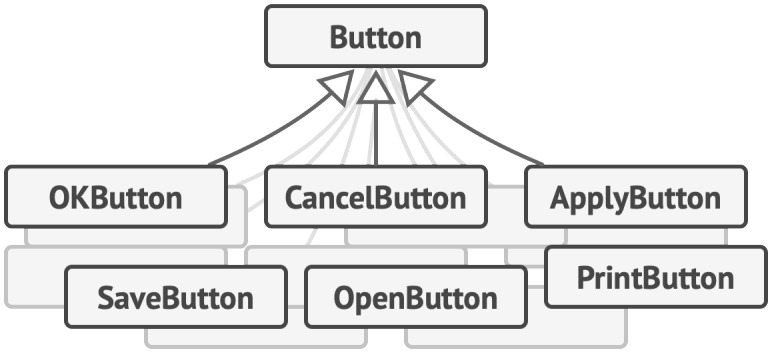
☹ Problema

Imagine que você está trabalhando em uma nova aplicação de editor de texto. Sua tarefa atual é criar uma barra de tarefas com vários botões para várias operações do editor. Você criou uma classe `Botão` muito bacana que pode ser usada para botões na barra de tarefas, bem como para botões genéricos de diversas caixas de diálogo.



Todos os botões de uma aplicação são derivadas de uma mesma classe.

Embora todos esses botões pareçam similares, eles todos devem fazer coisas diferentes. Aonde você deveria colocar o código para os vários handlers de cliques desses botões? A solução mais simples é criar um monte de subclasses para cada local que o botão for usado. Essas subclasses conteriam o código que teria que ser executado em um clique de botão.



Várias subclasses de botões. O que pode dar errado?

Não demora muito e você percebe que essa abordagem é falha. Primeiro você tem um enorme número de subclasses, e isso seria okay se você não arriscasse quebrar o código dentro dessas subclasses cada vez que você modificar a classe base `Botão`. Colocando em miúdos: seu código GUI se torna absurdamente dependente de um código volátil da lógica do negócio.

Várias classes implementam a mesma funcionalidade.

E aqui está a parte mais feia. Algumas operações, tais como copiar/colar texto, precisariam ser invocadas de diversos lugares. Por exemplo, um usuário poderia criar um pequeno botão “Copiar” na barra de ferramentas, ou copiar alguma coisa atra-

vés do menu de contexto, ou apenas apertando `Crtl+C` no teclado.

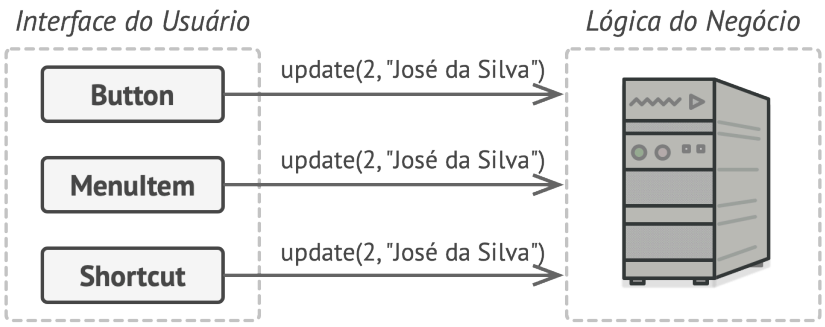
Inicialmente, quando sua aplicação só tinha a barra de ferramentas, tudo bem colocar a implementação de várias operações dentro das subclasses do botão. Em outras palavras, ter o código de cópia de texto dentro da subclasse `BotãoCópia` parecia certo. Mas então, quando você implementou menus de contexto, atalhos, e outras coisas, você teve que ou duplicar o código da operação em muitas classes ou fazer menus dependentes de botões, o que é uma opção ainda pior.

😊 Solução

Um bom projeto de software quase sempre se baseia no princípio da separação de interesses, o que geralmente resulta em dividir a aplicação em camadas. O exemplo mais comum: uma camada para a interface gráfica do usuário e outra camada para a lógica do negócio. A camada GUI é responsável por renderizar uma bonita imagem na tela, capturando quaisquer dados e mostrando resultados do que o usuário e a aplicação estão fazendo. Contudo, quando se trata de fazer algo importante, como calcular a trajetória da lua ou compor um relatório anual, a camada GUI delega o trabalho para a camada inferior da lógica do negócio.

Dentro do código pode parecer assim: um objeto GUI chama um método da lógica do negócio, passando alguns argumen-

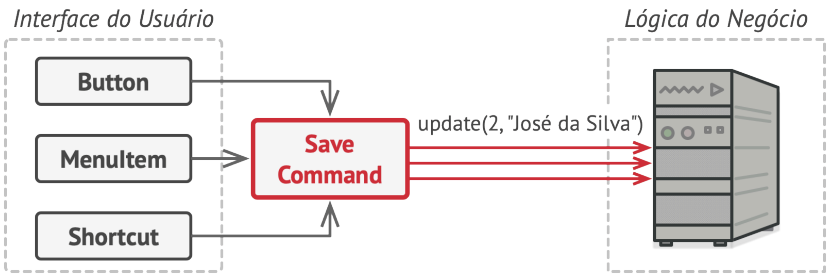
tos. Este processo é geralmente descrito como um objeto mandando um *pedido* para outro.



Os objetos GUI podem acessar os objetos da lógica do negócio diretamente.

O padrão Command sugere que os objetos GUI não enviem esses pedidos diretamente. Ao invés disso, você deve extrair todos os detalhes do pedido, tais como o objeto a ser chamado, o nome do método, e a lista de argumentos em uma classe *comando* separada que tem apenas um método que aciona esse pedido.

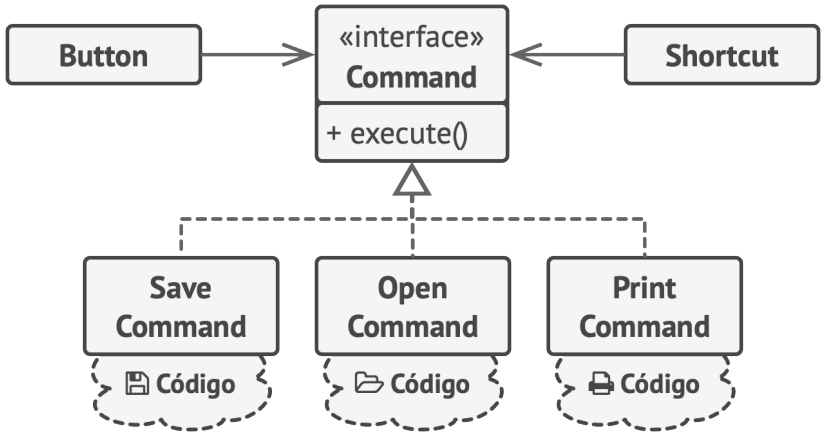
Objetos comando servem como links entre vários objetos GUI e de lógica de negócio. De agora em diante, o objeto GUI não precisa saber qual objeto de lógica do negócio irá receber o pedido e como ele vai ser processado. O objeto GUI deve acionar o comando, que irá lidar com todos os detalhes.



Acessando a lógica do negócio através do comando.

O próximo passo é fazer seus comandos implementarem a mesma interface. Geralmente é apenas um método de execução que não pega parâmetros. Essa interface permite que você use vários comandos com o mesmo remetente do pedido, sem acoplá-lo com as classes concretas dos comandos. Como um bônus, agora você pode trocar os objetos comando ligados ao remetente, efetivamente mudando o comportamento do remetente no momento da execução.

Você pode ter notado uma peça faltante nesse quebra cabeças, que são os parâmetros do pedido. Um objeto GUI pode ter fornecido ao objeto da camada de negócio com alguns parâmetros, como deveremos passar os detalhes do pedido para o destinatário? Parece que o comando deve ser ou pré configurado com esses dados, ou ser capaz de obtê-los por conta própria.



Os objetos GUI delegam o trabalho aos comandos.

Vamos voltar ao nosso editor de texto. Após aplicarmos o padrão Command, nós não mais precisamos de todas aquelas subclasses de botões para implementar vários comportamentos de cliques. É suficiente colocar apenas um campo na classe `Botão` base que armazena a referência para um objeto comando e faz o botão executar aquele comando com um clique.

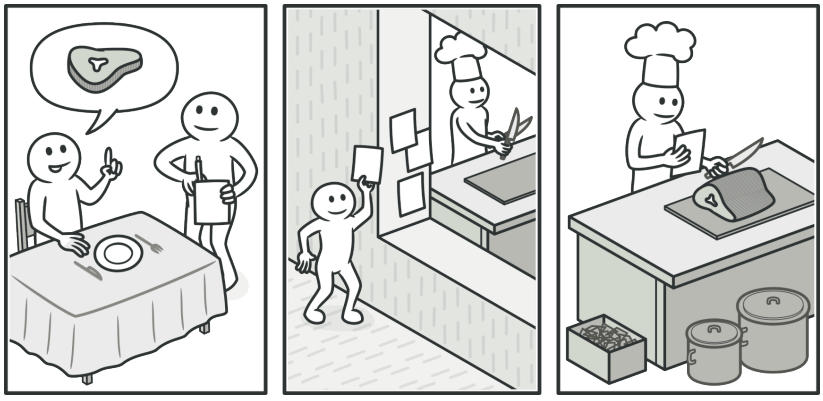
Você vai implementar um monte de classes comando para cada possível operação e ligá-los aos seus botões em particular, dependendo do comportamento desejado para os botões.

Outros elementos GUI, tais como menus, atalhos, ou caixas de diálogo inteiras, podem ser implementados da mesma maneira. Eles serão ligados a um comando que será executado quando um usuário interage com um elemento GUI. Como você provavelmente adivinhou, os elementos relacionados a mesma

operação serão ligados aos mesmos comandos, prevenindo a duplicação de código.

Como resultado, os comandos se tornam uma camada intermédia conveniente que reduz o acoplamento entre as camadas GUI e de lógica do negócio. E isso é apenas uma fração dos benefícios que o padrão Command pode oferecer.

Analogia com o mundo real



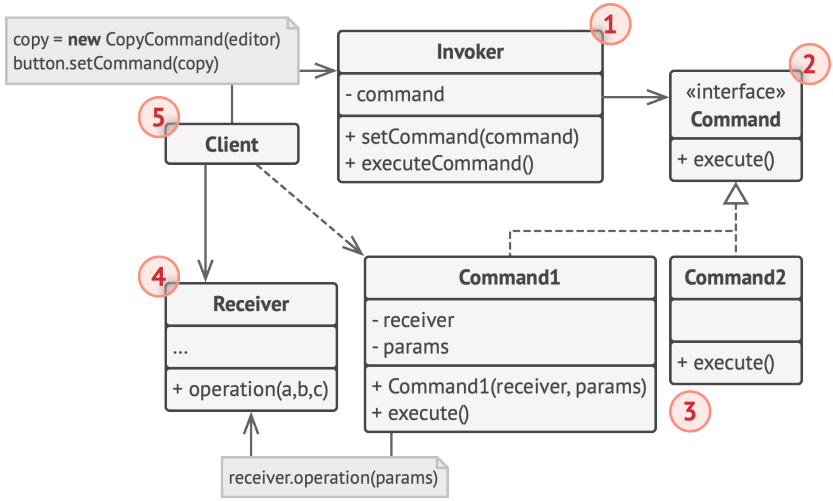
Fazendo um pedido em um restaurante.

Após uma longa caminhada pela cidade, você chega a um restaurante bacana e senta numa mesa perto da janela. Um garçom amigável se aproxima e rapidamente recebe seu pedido, escrevendo-o em um pedaço de papel. O garçom vai até a cozinha e prende o pedido em uma parede. Após algum tempo, o pedido chega até o chef, que o lê e cozinha a refeição de acordo. O cozinheiro coloca a refeição em uma bandeja junto

com o pedido. O garçom acha a bandeja, verifica o pedido para garantir que é aquilo que você queria, e o traz para sua mesa.

O pedido no papel serve como um comando. Ele permanece em uma fila até que o chef esteja pronto para serví-lo. O pedido contém todas as informações relevantes necessárias para cozinhar a refeição. Ele permite ao chef começar a cozinhar imediatamente ao invés de ter que ir até você para clarificar os detalhes do pedido pessoalmente.

Estrutura



1. A classe **Remetente** (também conhecida como *invocadora*) é responsável por iniciar os pedidos. Essa classe deve ter um campo para armazenar a referência para um objeto comando. O remetente aciona aquele comando ao invés de enviar o pedido diretamente para o destinatário. Observe que o remetente

não é responsável por criar o objeto comando. Geralmente ele é pré criado através de um construtor do cliente.

2. A interface **Comando** geralmente declara apenas um único método para executar o comando.

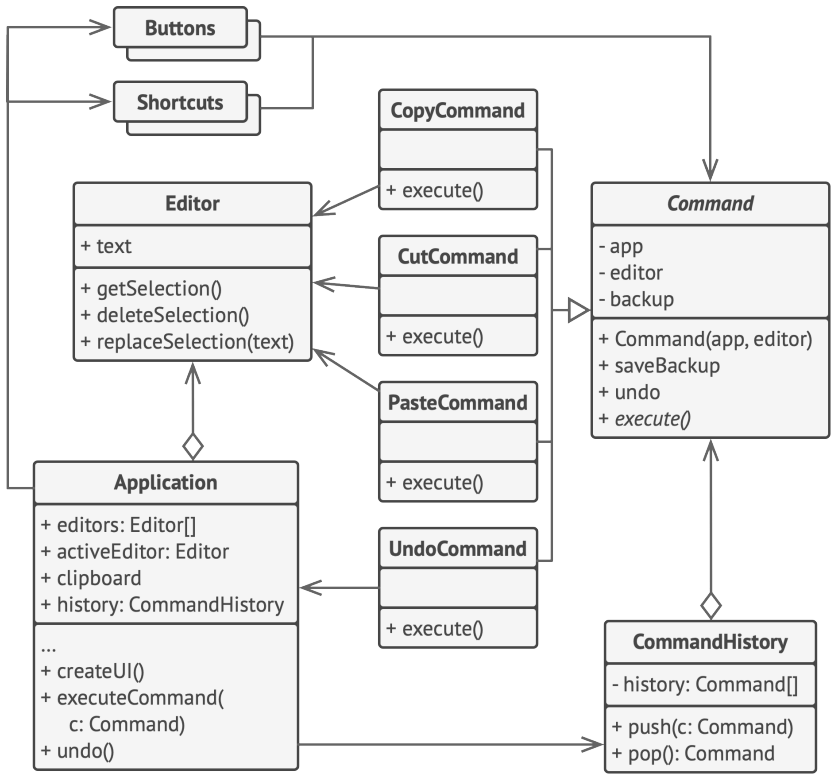
3. **Comandos Concretos** implementam vários tipos de pedidos. Um comando concreto não deve realizar o trabalho por conta própria, mas passar a chamada para um dos objetos da lógica do negócio. Contudo, para simplificar o código, essas classes podem ser fundidas. Os parâmetros necessários para executar um método em um objeto destinatário podem ser declarados como campos no comando concreto. Você pode tornar os objetos comando imutáveis ao permitir que apenas inicializem esses campos através do construtor.

4. A classe **Destinatária** contém a lógica do negócio. Quase qualquer objeto pode servir como um destinatário. A maioria dos comandos apenas lida com os detalhes de como um pedido é passado para o destinatário, enquanto que o destinatário em si executa o verdadeiro trabalho.

5. O **Cliente** cria e configura objetos comando concretos. O cliente deve passar todos os parâmetros do pedido, incluindo uma instância do destinatário, para o construtor do comando. Após isso, o comando resultante pode ser associado com um ou múltiplos destinatários.

Pseudocódigo

Neste exemplo, o padrão **Command** ajuda a manter um registro da história de operações executadas e torna possível reverter uma operação se necessário.



Operações não executáveis em um editor de texto.

Os comandos que resultam de mudanças de estado do editor (por exemplo, cortando e colando) fazem uma cópia de backup do estado do editor antes de executarem uma operação associada com o comando. Após o comando ser executado, ele é

colocado em um histórico de comando (uma pilha de objetos comando) junto com a cópia de backup do estado do editor naquele momento. Mais tarde, se o usuário precisa reverter uma operação, a aplicação pode pegar o comando mais recente do histórico, ler o backup associado ao estado do editor, e restaurá-lo.

O código cliente (elementos GUI, histórico de comandos, etc.) não é acoplado às classes comando concretas porque ele trabalha com comandos através da interface comando. Essa abordagem permite que você introduza novos comandos na aplicação sem quebrar o código existente.

```
// A classe comando base define a interface comum para todos
// comandos concretos.
abstract class Command is
  protected field app: Application
  protected field editor: Editor
  protected field backup: text

  constructor Command(app: Application, editor: Editor) is
    this.app = app
    this.editor = editor

  // Faz um backup do estado do editor.
  method saveBackup() is
    backup = editor.text

  // Restaura o estado do editor.
  method undo() is
```

```
      editor.text = backup

    // O método de execução é declarado abstrato para forçar
    // todos os comandos concretos a fornecer suas próprias
    // implementações. O método deve retornar verdadeiro ou
    // falso dependendo se o comando muda o estado do editor.
    abstract method execute()


// Comandos concretos vêm aqui.
class CopyCommand extends Command is
    // O comando copy (copiar) não é salvo no histórico já que
    // não muda o estado do editor.
    method execute() is
      app.clipboard = editor.getSelection()
      return false

class CutCommand extends Command is
    // O comando cut (cortar) muda o estado do editor, portanto
    // deve ser salvo no histórico. E ele será salvo desde que o
    // método retorne verdadeiro.
    method execute() is
      saveBackup()
      app.clipboard = editor.getSelection()
      editor.deleteSelection()
      return true

class PasteCommand extends Command is
    method execute() is
      saveBackup()
      editor.replaceSelection(app.clipboard)
      return true
```

```
50
51  // A operação undo (desfazer) também é um comando.
52  class UndoCommand extends Command is
53    method execute() is
54      app.undo()
55      return false
56
57
58  // O comando global history (histórico) é apenas uma pilha.
59  class CommandHistory is
60    private field history: array of Command
61
62    // Último a entrar...
63    method push(c: Command) is
64      // Empurra o comando para o fim do vetor do histórico.
65
66    // ...primeiro a sair.
67    method pop():Command is
68      // Obter o comando mais recente do histórico.
69
70
71  // A classe do editor tem verdadeiras operações de edição de
72  // texto. Ela faz o papel de destinatária: todos os comandos
73  // acabam delegando a execução para os métodos do editor.
74  class Editor is
75    field text: string
76
77    method getSelection() is
78      // Retorna o texto selecionado.
79
80    method deleteSelection() is
81      // Deleta o texto selecionado.
```

```
82
83    method replaceSelection(text) is
84      // Insere os conteúdos da área de transferência na
85      // posição atual.
86
87
88  // A classe da aplicação define as relações de objeto. Ela age
89  // como uma remetente: quando alguma coisa precisa ser feita,
90  // ela cria um objeto comando e executa ele.
91  class Application is
92    field clipboard: string
93    field editors: array of Editors
94    field activeEditor: Editor
95    field history: CommandHistory
96
97    // O código que assinala comandos para objetos UI pode se
98    // parecer como este.
99    method createUI() is
100     // ...
101     copy = function() { executeCommand(
102       new CopyCommand(this, activeEditor)) }
103     copyButton.setCommand(copy)
104     shortcuts.onKeyPress("Ctrl+C", copy)
105
106     cut = function() { executeCommand(
107       new CutCommand(this, activeEditor)) }
108     cutButton.setCommand(cut)
109     shortcuts.onKeyPress("Ctrl+X", cut)
110
111     paste = function() { executeCommand(
112       new PasteCommand(this, activeEditor)) }
113     pasteButton.setCommand(paste)
```

```
    shortcuts.onKeyPress("Ctrl+V", paste)

    undo = function() { executeCommand(
      new UndoCommand(this, activeEditor)) }
    undoButton.setCommand(undo)
    shortcuts.onKeyPress("Ctrl+Z", undo)

// Executa um comando e verifica se ele foi adicionado ao
// histórico.
method executeCommand(command) is
  if (command.execute)
    history.push(command)

// Pega o comando mais recente do histórico e executa seu
// método undo(desfazer). Observe que nós não sabemos a
// classe desse comando. Mas nós não precisamos saber, já
// que o comando sabe como desfazer sua própria ação.
method undo() is
  command = history.pop()
  if (command != null)
    command.undo()
```

Aplicabilidade

Utilize o padrão Command quando você quer parametrizar objetos com operações.

O padrão Command podem tornar uma chamada específica para um método em um objeto separado. Essa mudança abre várias possibilidades de usos interessantes: você pode passar

comandos como argumentos do método, armazená-los dentro de outros objetos, trocar comandos ligados no momento de execução, etc.

Aqui está um exemplo: você está desenvolvendo um componente GUI como um menu de contexto, e você quer que os usuários sejam capazes de configurar os items do menu que aciona as operações quando um usuário clica em um item.

Utilize o padrão Command quando você quer colocar operações em fila, agendar sua execução, ou executá-las remotamente.

Como qualquer outro objeto, um comando pode ser serializado, o que significa convertê-lo em uma string que pode ser facilmente escrita em um arquivo ou base de dados. Mais tarde a string pode ser restaurada no objeto comando inicial. Dessa forma você pode adiar e agendar execuções do comando. Mas isso não é tudo! Da mesma forma, você pode colocar em fila, fazer registro de log ou enviar comandos por uma rede.

Utilize o padrão Command quando você quer implementar operações reversíveis.

Embora haja muitas formas de implementar o desfazer/refazer, o padrão Command é talvez a mais popular de todas.

Para ser capaz de reverter operações, você precisa implementar o histórico de operações realizadas. O histórico do co-

mando é uma pilha que contém todos os objetos comando executados junto com seus backups do estado da aplicação relacionados.

Esse método tem duas desvantagens. Primeiro, se não for fácil salvar o estado da aplicação por parte dela ser privada. Esse problema pode ser facilmente mitigado com o padrão **Memento**.

Segundo, os backups de estado podem consumir uma considerável quantidade de RAM. Portanto, algumas vezes você pode recorrer a uma implementação alternativa: ao invés de restaurar a um estado passado, o comando faz a operação inversa. A operação reversa também cobra um preço: ela pode ter sua implementação difícil ou até impossível.

Como implementar

1. Declare a interface comando com um único método de execução.

2. Comece extraindo pedidos para dentro de classes concretas comando que implementam a interface comando. Cada classe deve ter um conjunto de campos para armazenar os argumentos dos pedidos junto com uma referência ao objeto destinatário real. Todos esses valores devem ser inicializados através do construtor do comando.

3. Identifique classes que vão agir como *remetentes*. Adicione os campos para armazenar comandos nessas classes. Remeten-

tes devem sempre comunicar-se com seus comandos através da interface comando. Remetentes geralmente não criam objetos comando por conta própria, mas devem obtê-los do código cliente.

4. Mude os remetentes para que executem o comando ao invés de enviar o pedido para o destinatário diretamente.

5. O cliente deve inicializar objetos na seguinte ordem:

    - Crie os destinatários.
    - Crie os comandos, e os associe com os destinatários se necessário.
    - Crie os remetentes, e os associe com os comandos específicos.

Prós e contras

- ✓ *Princípio de responsabilidade única*. Você pode desacoplar classes que invocam operações de classes que fazem essas operações.
- ✓ *Princípio aberto/fechado*. Você pode introduzir novos comandos na aplicação sem quebrar o código cliente existente.
- ✓ Você pode implementar desfazer/refazer.
- ✓ Você pode implementar a execução adiada de operações.
- ✓ Você pode montar um conjunto de comandos simples em um complexo.

✕ O código pode ficar mais complicado uma vez que você está introduzindo uma nova camada entre remetentes e destinatários.

⇄ Relações com outros padrões

- O **Chain of Responsibility**, **Command**, **Mediator** e **Observer** abrangem várias maneiras de se conectar remetentes e destinatários de pedidos:
  - O *Chain of Responsibility* passa um pedido sequencialmente ao longo de um corrente dinâmica de potenciais destinatários até que um deles atua no pedido.
  - O *Command* estabelece conexões unidirecionais entre remetentes e destinatários.
  - O *Mediator* elimina as conexões diretas entre remetentes e destinatários, forçando-os a se comunicar indiretamente através de um objeto mediador.
  - O *Observer* permite que destinatários inscrevam-se ou cancelem sua inscrição dinamicamente para receber pedidos.
- Handlers em uma **Chain of Responsibility** podem ser implementados como **comandos**. Neste caso, você pode executar várias operações diferentes sobre o mesmo objeto contexto, representado por um pedido.

  Contudo, há outra abordagem, onde o próprio pedido é um objeto *comando*. Neste caso, você pode executar a mesma ope-

ração em uma série de diferentes contextos ligados em uma corrente.

- Você pode usar o **Command** e o **Memento** juntos quando implementando um "desfazer". Neste caso, os comandos são responsáveis pela realização de várias operações sobre um objeto alvo, enquanto que os mementos salvam o estado daquele objeto momentos antes de um comando ser executado.

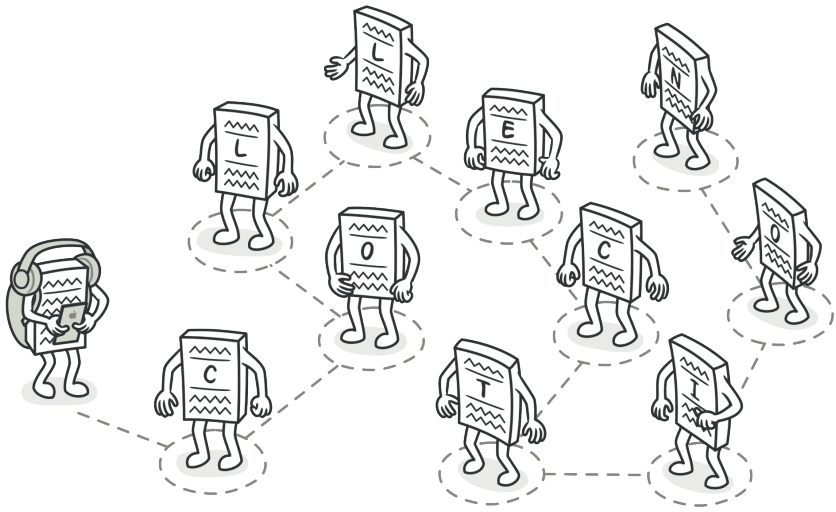
- O **Command** e o **Strategy** podem ser parecidos porque você pode usar ambos para parametrizar um objeto com alguma ação. Contudo, eles têm propósitos bem diferentes.

  - Você pode usar o *Command* para converter qualquer operação em um objeto. Os parâmetros da operação se transformam em campos daquele objeto. A conversão permite que você atrase a execução de uma operação, transforme-a em uma fila, armazene o histórico de comandos, envie comandos para serviços remotos, etc.

  - Por outro lado, o *Strategy* geralmente descreve diferentes maneiras de fazer a mesma coisa, permitindo que você troque esses algoritmos dentro de uma única classe contexto.

- O **Prototype** pode ajudar quando você precisa salvar cópias de **comandos** no histórico.

- Você pode tratar um **Visitor** como uma poderosa versão do padrão **Command**. Seus objetos podem executar operações sobre vários objetos de diferentes classes.



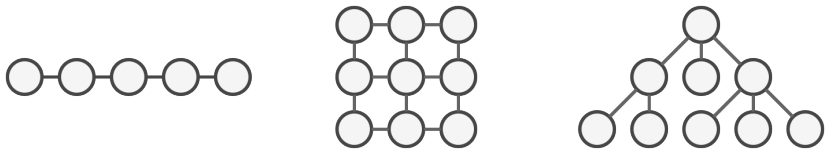
ITERATOR

Também conhecido como: Iterador

O **Iterator** é um padrão de projeto comportamental que permite a você percorrer elementos de uma coleção sem expor as representações dele (lista, pilha, árvore, etc.)

☹ Problema

Coleções são um dos tipos de dados mais usados em programação. Não obstante, uma coleção é apenas um contêiner para um grupo de objetos.



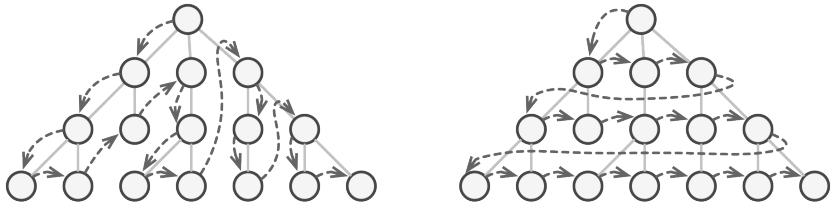
Vários tipos de coleções.

A maioria das coleções armazena seus elementos em listas simples. Contudo, alguns deles são baseados em pilhas, árvores, grafos, e outras estruturas complexas de dados.

Mas independente de quão complexa uma coleção é estruturada, ela deve fornecer uma maneira de acessar seus elementos para que outro código possa usá-los. Deve haver uma maneira de ir de elemento em elemento na coleção sem ter que acessar os mesmos elementos repetidamente.

Isso parece uma tarefa fácil se você tem uma coleção baseada numa lista. Você faz um loop em todos os elementos. Mas como você faz a travessia dos elementos de uma estrutura de dados complexas sequencialmente, tais como uma árvore. Por exemplo, um dia você pode apenas precisar percorrer em profundidade em uma árvore. No dia seguinte você pode precisar percorrer na amplitude. E na semana seguinte, você pode pre-

cisar algo diferente, como um acesso aleatório aos elementos da árvore.



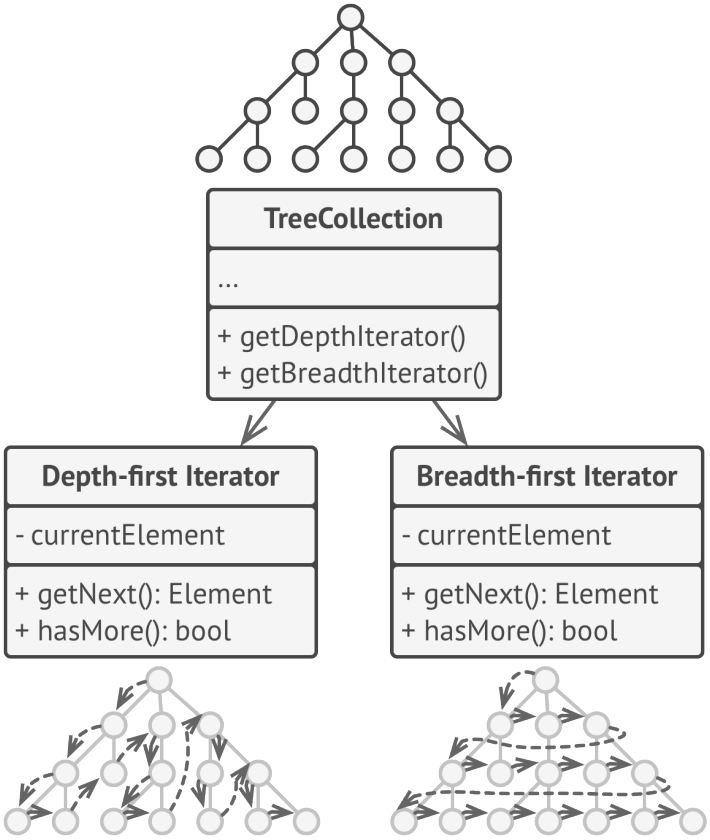
A mesma coleção pode ser atravessada de diferentes formas.

Adicionando mais e mais algoritmos de travessia para uma coleção gradualmente desfoca sua responsabilidade primária, que é um armazenamento de dados eficiente. Além disso, alguns algoritmos podem ser feitos para aplicações específicas, então incluí-los em uma coleção genérica pode ser estranho.

Por outro lado, o código cliente que deveria trabalhar com várias coleções pode não se importar com a maneira que elas armazenam seus elementos. Contudo, uma vez que todas as coleções fornecem diferentes maneiras de acessar seus elementos, você não tem outra opção além de acoplar seu código com as classes de coleções específicas.

😊 Solução

A ideia principal do padrão Iterator é extrair o comportamento de travessia de uma coleção para um objeto separado chamado um *iterador*.



Iteradores implementam vários algoritmos de travessia. Alguns objetos iterador podem fazer a travessia da mesma coleção ao mesmo tempo.

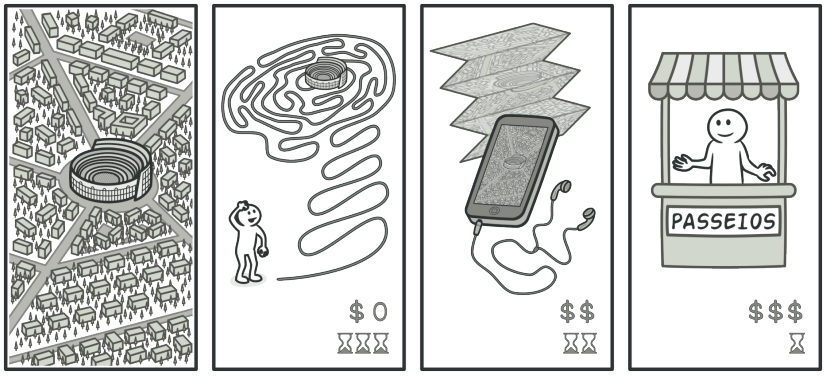
Além de implementar o algoritmo em si, um objeto iterador encapsula todos os detalhes da travessia, tais como a posição atual e quantos elementos faltam para chegar ao fim. Por causa disso, alguns iteradores podem averiguar a mesma coleção ao mesmo tempo, independentemente um do outro.

Geralmente, os iteradores fornecem um método primário para pegar elementos de uma coleção. O cliente pode manter esse

método funcionando até que ele não retorne mais nada, o que significa que o iterador atravessou todos os elementos.

Todos os iteradores devem implementar a mesma interface. Isso faz que o código cliente seja compatível com qualquer tipo de coleção ou qualquer algoritmo de travessia desde que haja um iterador apropriado. Se você precisar de uma maneira especial para a travessia de uma coleção, você só precisa criar uma nova classe iterador, sem ter que mudar a coleção ou o cliente.

Analogia com o mundo real



Várias maneiras de se caminhar por Roma.

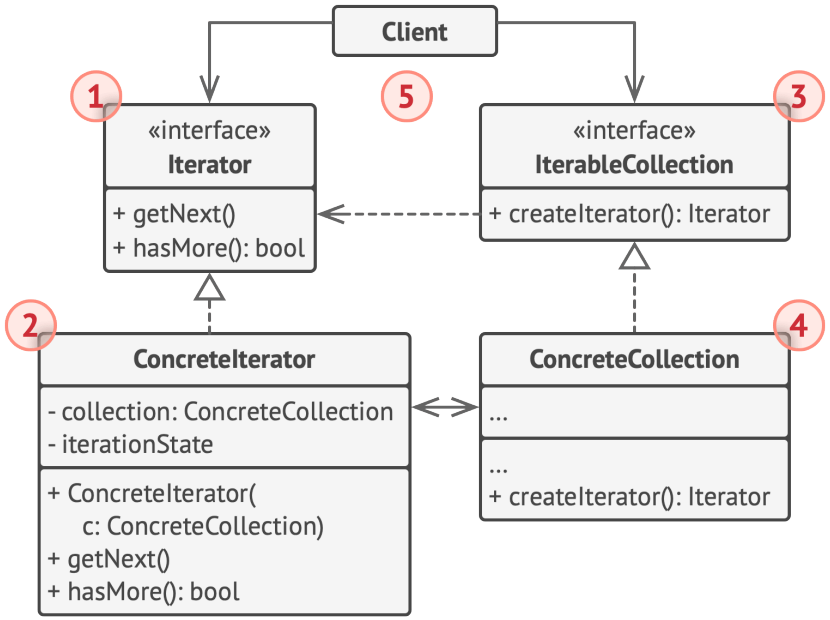
Você planeja visitar Roma por alguns dias e visitar todas suas principais atrações e pontos de interesse. Mas uma vez lá, você poderia gastar muito tempo andando em círculos, incapaz de achar até mesmo o Coliseu.

Por outro lado, você pode comprar um guia virtual na forma de app para seu smartphone e usá-lo como navegação. É inteligente e barato, e você pode ficar em lugares interessantes por quanto tempo quiser.

Uma terceira alternativa é gastar um pouco da verba da viagem e contratar um guia local que conhece a cidade como a palma de sua mão. O guia poderia ser capaz de criar um passeio que se adeque a seus gostos, mostrar todas as atrações, e contar um monte de histórias interessantes. Isso seria mais divertido, mas, infelizmente, mais caro também.

Todas essas opções—direções aleatórias criadas em sua cabeça, o navegador do smartphone, ou o guia humano—agem como iteradores sobre a vasta coleção de locais e atrações de Roma.

Estrutura



1. A interface **Iterador** declara as operações necessárias para percorrer uma coleção: buscar o próximo elemento, pegar a posição atual, recomeçar a iteração, etc.

2. **Iteradores Concretos** implementam algoritmos específicos para percorrer uma coleção. O objeto iterador deve monitorar o progresso da travessia por conta própria. Isso permite que diversos iteradores percorram a mesma coleção independentemente de cada um.

3. A interface **Coleção** declara um ou mais métodos para obter os iteradores compatíveis com a coleção. Observe que o tipo do retorno dos métodos deve ser declarado como a interface do

iterador para que as coleções concretas possam retornar vários tipos de iteradores.

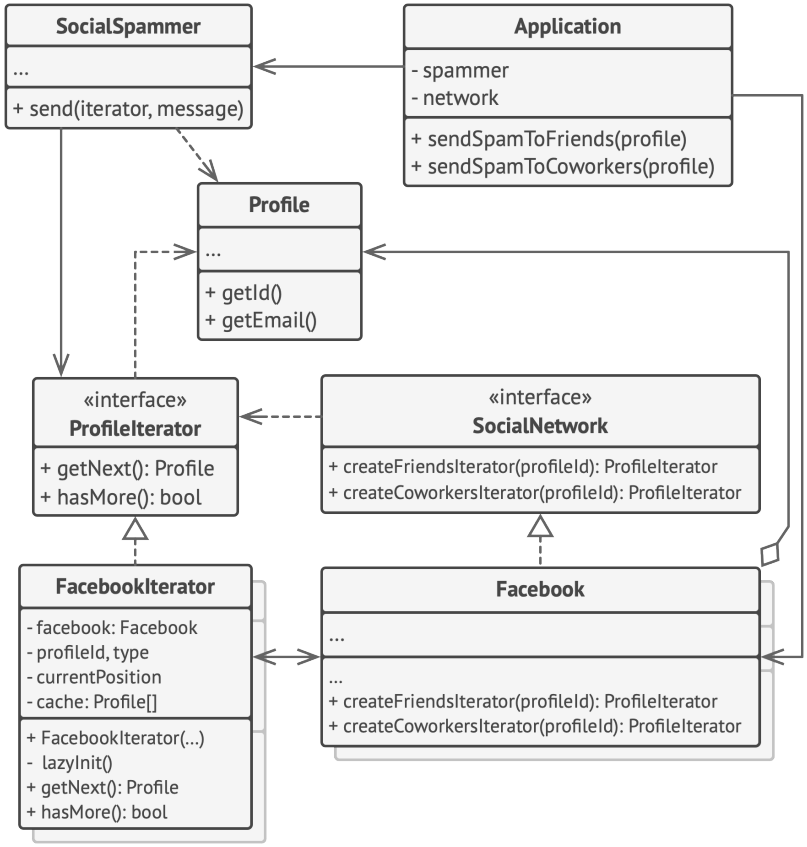
4. **Coleções Concretas** retornam novas instâncias de uma classe iterador concreta em particular cada vez que o cliente pede por uma. Você pode estar se perguntando, onde está o resto do código da coleção? Não se preocupe, ele deve ficar na mesma classe. É que esses detalhes não são cruciais para o padrão atual, então optamos por omiti-los.

5. O **Cliente** trabalha tanto com as coleções como os iteradores através de suas interfaces. Dessa forma o cliente não fica acoplado a suas classes concretas, permitindo que você use várias coleções e iteradores com o mesmo código cliente.

   Tipicamente, os clientes não criam iteradores por conta própria, mas ao invés disso os obtêm das coleções. Ainda assim, em certos casos, o cliente pode criar um diretamente; por exemplo, quando o cliente define seu próprio iterador especial.

Pseudocódigo

Neste exemplo, o padrão **Iterator** é usado para percorrer uma coleção especial que encapsula acesso ao grafo social do Facebook. A coleção fornece vários iteradores que podem percorrer perfis de várias maneiras.



Exemplo de uma iteração sobre perfis sociais.

O iterador 'amigos' pode ser usado para verificar os amigos de um dado perfil. O iterador 'colegas' faz a mesma coisa, exceto por omitir amigos que não trabalham na mesma companhia como uma pessoa alvo. Ambos iteradores implementam uma interface comum que permite os clientes recuperar os perfis sem mergulhar nos detalhes de implementação como autenticação e pedidos REST.

O código cliente não está acoplado à classes concretas porque funciona com coleções e iteradores somente através de interfaces. Se você decidir conectar sua aplicação com uma nova rede social, você simplesmente precisa fornecer as novas classes de iteração e coleção sem mudar o código existente.

```
// A interface da coleção deve declarar um método fábrica para
// produzir iteradores. Você pode declarar vários métodos se há
// diferentes tipos de iteração disponíveis em seu programa.
interface SocialNetwork is
  method createFriendsIterator(profileId):ProfileIterator
  method createCoworkersIterator(profileId):ProfileIterator


// Cada coleção concreta é acoplada a um conjunto de classes
// iterador concretas que ela retorna. Mas o cliente não é, uma
// vez que a assinatura desses métodos retorna interfaces de
// iterador.
class Facebook implements SocialNetwork is
  // ...o grosso do código da coleção deve vir aqui...

  // Código de criação do iterador.
  method createFriendsIterator(profileId) is
    return new FacebookIterator(this, profileId, "friends")
  method createCoworkersIterator(profileId) is
    return new FacebookIterator(this, profileId, "coworkers")


// A interface comum a todos os iteradores.
interface ProfileIterator is
  method getNext():Profile
```

```
    method hasMore():bool


// A classe iterador concreta.
class FacebookIterator implements ProfileIterator is
    // O iterador precisa de uma referência para a coleção que
    // ele percorre.
    private field facebook: Facebook
    private field profileId, type: string

    // Um objeto iterador percorre a coleção independentemente
    // de outros iteradores. Portanto ele tem que armazenar o
    // estado de iteração.
    private field currentPosition
    private field cache: array of Profile

    constructor FacebookIterator(facebook, profileId, type) is
        this.facebook = facebook
        this.profileId = profileId
        this.type = type

    private method lazyInit() is
        if (cache == null)
            cache = facebook.socialGraphRequest(profileId, type)

    // Cada classe iterador concreta tem sua própria
    // implementação da interface comum do iterador.
    method getNext() is
        if (hasMore())
            currentPosition++
            return cache[currentPosition]

```

```
    method hasMore() is
      lazyInit()
      return currentPosition < cache.length


// Aqui temos outro truque útil: você pode passar um iterador
// para uma classe cliente ao invés de dar acesso a ela à uma
// coleção completa. Dessa forma, você não expõe a coleção ao
// cliente.
//
// E tem outro benefício: você pode mudar o modo que o cliente
// trabalha com a coleção no tempo de execução ao passar a ele
// um iterador diferente. Isso é possível porque o código
// cliente não é acoplado às classes iterador concretas.
class SocialSpammer is
  method send(iterator: ProfileIterator, message: string) is
    while (iterator.hasMore())
      profile = iterator.getNext()
      System.sendEmail(profile.getEmail(), message)


// A classe da aplicação configura coleções e iteradores e então
// os passa ao código cliente.
class Application is
  field network: SocialNetwork
  field spammer: SocialSpammer

  method config() is
    if working with Facebook
      this.network = new Facebook()
    if working with LinkedIn
      this.network = new LinkedIn()
```

```
90      this.spammer = new SocialSpammer()
91
92    method sendSpamToFriends(profile) is
93      iterator = network.createFriendsIterator(profile.getId())
94      spammer.send(iterator, "Very important message")
95
96    method sendSpamToCoworkers(profile) is
97      iterator = network.createCoworkersIterator(profile.getId())
98      spammer.send(iterator, "Very important message")
```

Aplicabilidade

Utilize o padrão Iterator quando sua coleção tiver uma estrutura de dados complexa por debaixo dos panos, mas você quer esconder a complexidade dela de seus clientes (seja por motivos de conveniência ou segurança).

O iterador encapsula os detalhes de se trabalhar com uma estrutura de dados complexa, fornecendo ao cliente vários métodos simples para acessar os elementos da coleção. Embora essa abordagem seja muito conveniente para o cliente, ela também protege a coleção de ações descuidadas ou maliciosas que o cliente poderia fazer se estivesse trabalhando com as coleções diretamente.

Utilize o padrão para reduzir a duplicação de código de travessia em sua aplicação.

O código de algoritmos de iteração não triviais tendem a ser muito pesados. Quando colocados dentro da lógica de negócio da aplicação, ele pode desfocar a responsabilidade do codigo original e torná-lo um código de difícil manutenção. Mover o código de travessia para os iteradores designados pode ajudar você a tornar o código da aplicação mais enxuto e limpo.

Utilize o Iterator quando você quer que seu código seja capaz de percorrer diferentes estruturas de dados ou quando os tipos dessas estruturas são desconhecidos de antemão.

O padrão fornece um par de interfaces genérica tanto para coleções como para iteradores. Já que seu código agora usa essas interfaces, ele ainda vai funcionar se você passar diversos tipos de coleções e iteradores que implementam essas interfaces.

Como implementar

1. Declare a interface do iterador. Ao mínimo, ela deve ter um método para buscar o próximo elemento de uma coleção. Mas por motivos de conveniência você pode adicionar alguns outros métodos, tais como recuperar o elemento anterior, saber a posição atual, e checar o fim da iteração.

2. Declare a interface da coleção e descreva um método para buscar iteradores. O tipo de retorno deve ser igual à interface do iterador. Você pode declarar métodos parecidos se você planeja ter grupos distintos de iteradores.

3. Implemente classes iterador concretas para as coleções que você quer percorrer com iteradores. Um objeto iterador deve ser ligado com uma única instância de coleção. Geralmente esse link é estabelecido através do construtor do iterador.

4. Implemente a interface da coleção na suas classes de coleção. A ideia principal é fornecer ao cliente com um atalho para criar iteradores, customizados para uma classe coleção em particular. O objeto da coleção deve passar a si mesmo para o construtor do iterador para estabelecer um link entre eles.

5. Vá até o código cliente e substitua todo o código de travessia de coleção com pelo uso de iteradores. O cliente busca um novo objeto iterador a cada vez que precisa iterar sobre os elementos de uma coleção.

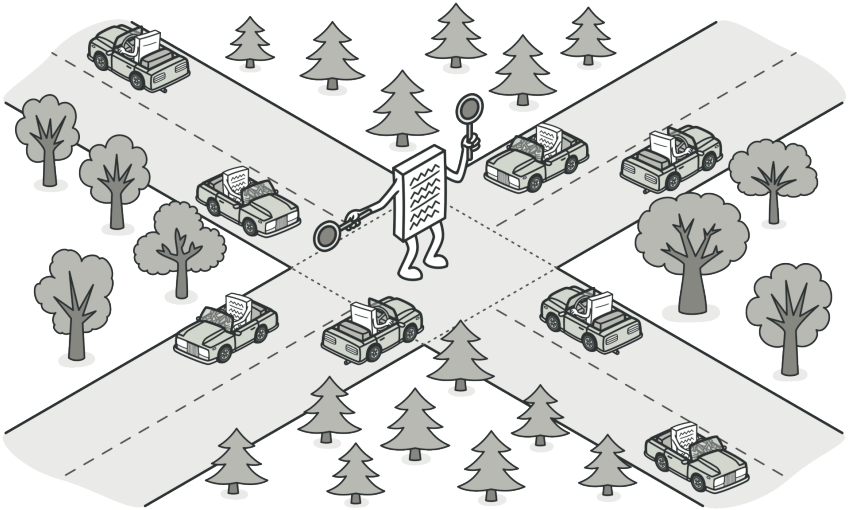
Prós e contras

- ✓ *Princípio de responsabilidade única*. Você pode limpar o código cliente e as coleções ao extrair os pesados algoritmos de travessia para classes separadas.
- ✓ *Princípio aberto/fechado*. Você pode implementar novos tipos de coleções e iteradores e passá-los para o código existente sem quebrar coisa alguma.
- ✓ Você pode iterar sobre a mesma coleção em paralelo porque cada objeto iterador contém seu próprio estado de iteração.
- ✓ Pelas mesmas razões, você pode atrasar uma iteração e continuá-la quando necessário.

- ✕ Aplicar o padrão pode ser um preciosismo se sua aplicação só trabalha com coleções simples.
- ✕ Usar um iterador pode ser menos eficiente que percorrer elementos de algumas coleções especializadas diretamente.

⇄ Relações com outros padrões

- Você pode usar **Iteradores** para percorrer árvores **Composite**.
- 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.
- Você pode usar o **Memento** junto com o **Iterator** para capturar o estado de iteração atual e revertê-lo se necessário.
- Você pode usar o **Visitor** junto com o **Iterator** para percorrer uma estrutura de dados complexas e executar alguma operação sobre seus elementos, mesmo se eles todos tenham classes diferentes.



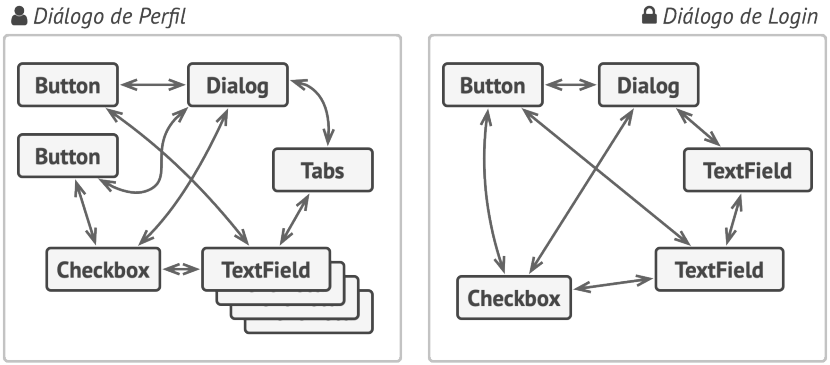
MEDIATOR

Também conhecido como: Mediador, Intermediário, Intermediary, Controlador, Controller

O **Mediator** é um padrão de projeto comportamental que permite que você reduza as dependências caóticas entre objetos. O padrão restringe comunicações diretas entre objetos e os força a colaborar apenas através do objeto mediador.

☹ Problema

Digamos que você tem uma caixa de diálogo para criar e editar perfis de clientes. Ela consiste em vários controles de formulário tais como campos de texto, caixas de seleção, botões, etc.



As relações entre os elementos da interface de usuário podem se tornar caóticas a medida que a aplicação evolui.

Alguns dos elementos do formulário podem interagir com outros. Por exemplo, selecionando a caixa de “Eu tenho um cão” pode revelar uma caixa de texto escondida para inserir o nome do cão. Outro exemplo é o botão enviar que tem que validar todos os campos antes de salvar os dados.

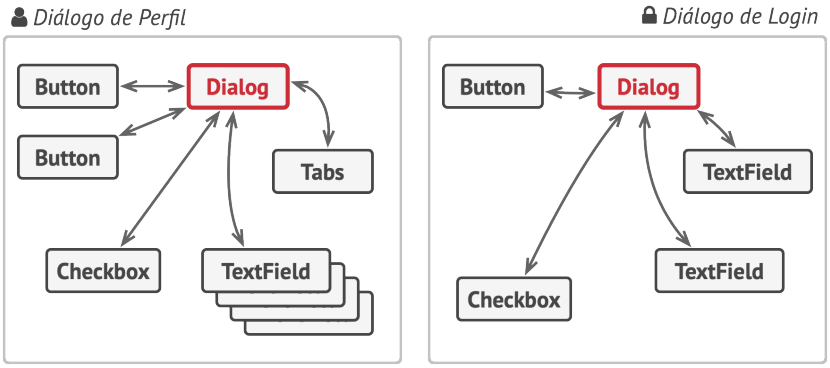
Os elementos podem ter várias relações com outros elementos. Portanto, mudanças a alguns elementos podem afetar os outros.

Ao ter essa lógica implementada diretamente dentro do código dos elementos de formulários você torna as classes dos elementos muito difíceis de se reutilizar em outros formulários da aplicação. Por exemplo, você não será capaz de usar aquela classe de caixa de seleção dentro de outro formulário porque ela está acoplado com o campo de texto do nome do cão. Você pode ter ou todas as classes envolvidas na renderização do formulário de perfil, ou nenhuma.

😊 Solução

O padrão Mediator sugere que você deveria cessar toda comunicação direta entre componentes que você quer tornar independentes um do outro. Ao invés disso, esses componentes devem colaborar indiretamente, chamando um objeto mediador especial que redireciona as chamadas para os componentes apropriados. Como resultado, os componentes dependem apenas de uma única classe mediadora ao invés de serem acoplados a dúzias de outros colegas.

No nosso exemplo com o formulário de edição de perfil, a classe diálogo em si pode agir como mediadora. O mais provável é que a classe diálogo já esteja ciente de todos seus sub-elementos, então você não precisa introduzir novas dependências nessa classe.



Os elementos UI devem se comunicar indiretamente, através do objeto mediador.

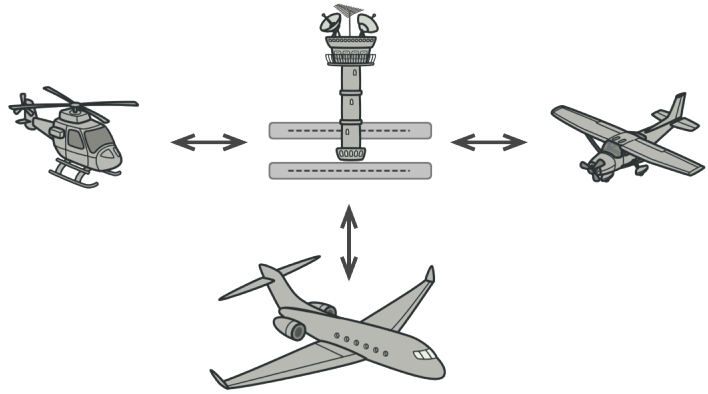
A mudança mais significativa acontece com os próprios elementos do formulário. Vamos considerar o botão de enviar. Antes, cada vez que um usuário clicava no botão, ele teria que validar os valores de todos os elementos de formulário. Agora seu único trabalho é notificar a caixa de diálogo sobre o clique. Ao receber essa notificação, a própria caixa de diálogo realiza as validações ou passa a tarefa para os elementos individuais. Portanto, ao invés de estar amarrado a uma dúzia de elementos de formulário, o botão está dependente apenas da classe diálogo.

Você pode ir além e fazer a dependência ainda mais frouxa extraindo a interface comum de todos os tipos de caixas de diálogo. A interface deve declarar o método de notificação que todos os elementos do formulário podem usar para notificar a caixa de diálogo sobre eventos acontecendo a aqueles elementos. Portanto, nosso botão enviar deve agora ser capaz

de trabalhar com qualquer caixa de diálogo que implemente aquela interface.

Dessa forma, o padrão Mediator permite que você encapsule uma complexa rede de relações entre vários objetos em apenas um objeto mediador. Quanto menos dependências uma classe tenha, mais fácil essa classe se torna para se modificar, estender, ou reutilizar.

Analogia com o mundo real



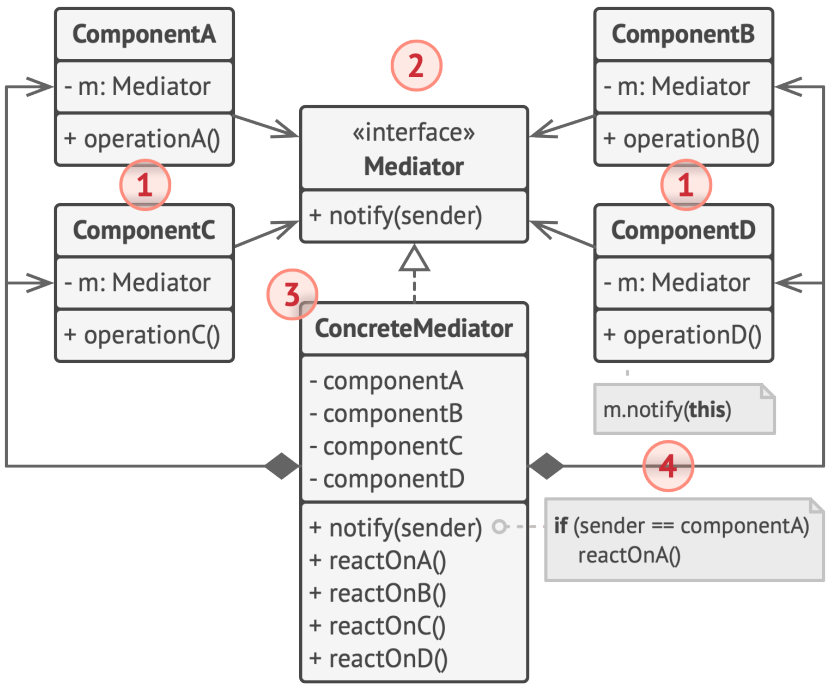
Pilotos de aeronaves não falam entre si diretamente na hora de decidir quem é o próximo a aterrissar seu avião. Toda a comunicação passa pela torre de controle.

Os pilotos de aeronaves que se aproximam ou partem da área de controle do aeroporto não se comunicam diretamente entre si. Ao invés disso falam com um controlador de tráfego aéreo, que está sentando em uma torre alta perto da pista de aterrissagem. Sem o controlador do tráfego aéreo os pilotos precisariam estar cientes de cada avião nas redondezas do aeroporto,

discutindo as prioridades de aterrissagem com um comitê de dúzias de outros pilotos. Isso provavelmente aumentaria em muito as estatísticas de acidentes aéreos.

A torre não precisa fazer o controle de todo o voo. Ela existe apenas para garantir o condicionamento da área do terminal devido ao número de pessoas envolvidas ali, o que poderia ser demais para um piloto.

Estrutura



1. Os **Componentes** são várias classes que contém alguma lógica de negócio. Cada componente tem uma referência a um mediador, declarada com o tipo de interface do mediador. O compo-

nente não está ciente da classe atual do mediador, então você pode reutilizar o componente em outros programas ao ligá-lo com um mediador diferente.

2. A interface do **Mediador** declara métodos de comunicação com os componentes, os quais geralmente incluem apenas um método de notificação. Os componentes podem passar qualquer contexto como argumentos desse método, incluindo seus próprios objetos, mas apenas de tal forma que nenhum acoplamento ocorra entre um componente destinatário e a classe remetente.

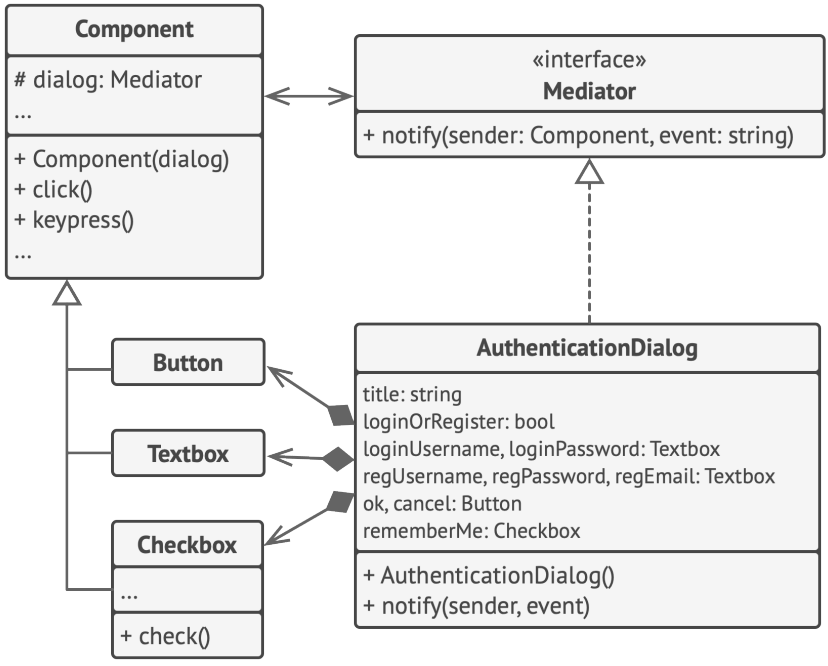
3. Os **Mediadores Concretos** encapsulam as relações entre vários componentes. Os mediadores concretos quase sempre mantém referências de todos os componentes os quais gerenciam e, algumas vezes, até gerenciam o ciclo de vida deles.

4. Componentes não devem estar cientes de outros componentes. Se algo importante acontece dentro ou para um componente, ele deve apenas notificar o mediador. Quando o mediador recebe a notificação, ele pode facilmente identificar o remetente, o que é suficiente para decidir que componente deve ser acionado em retorno.

   Da perspectiva de um componente, tudo parece como uma caixa preta. O remetente não sabe quem vai acabar lidando com o seu pedido, e o destinatário não sabe quem enviou o pedido em primeiro lugar.

Pseudocódigo

Neste exemplo, o padrão **Mediator** ajuda você a eliminar dependências mútuas entre várias classes UI: botões, caixas de seleção, e textos de rótulos.



Estrutura das classes UI caixa de diálogo.

Um elemento, acionado por um usuário, não se comunica com outros elementos diretamente, mesmo que pareça que ele deva fazer isso. Ao invés disso, o elemento apenas precisa fazer o mediador saber do evento, passando qualquer informação de contexto junto com a notificação.

Neste exemplo, todas caixas de diálogo de autenticação agem como o mediador. Elas sabem quais os elementos concretos devem colaborar e facilita sua comunicação indireta. Ao receber a notificação de um evento, a caixa de diálogo decide que elemento deve lidar com o evento e redireciona a chamada de acordo.

```
// A interface mediadora declara um método usado pelos
// componentes para notificar o mediador sobre vários eventos. O
// mediador pode reagir a esses eventos e passar a execução para
// outros componentes.
interface Mediator is
  method notify(sender: Component, event: string)


// A classe mediadora concreta. A rede entrelaçada de conexões
// entre componentes individuais foi desentrelaçada e movida
// para dentro do mediador.
class AuthenticationDialog implements Mediator is
  private field title: string
  private field loginOrRegisterChkBx: Checkbox
  private field loginUsername, loginPassword: Textbox
  private field registrationUsername, registrationPassword,
          registrationEmail: Textbox
  private field okBtn, cancelBtn: Button

  constructor AuthenticationDialog() is
    // Cria todos os objetos componentes e passa o atual
    // mediador em seus construtores para estabelecer links.

  // Quando algo acontece com um componente, ele notifica o
```

```
    // mediador. Ao receber a notificação, o mediador pode fazer
    // alguma coisa por conta própria ou passar o pedido para
    // outro componente.
    method notify(sender, event) is
      if (sender == loginOrRegisterChkBx and event == "check")
        if (loginOrRegisterChkBx.checked)
          title = "Log in"
          // 1. Mostra componentes de formulário de login.
          // 2. Esconde componentes de formulário de
          // registro.
        else
          title = "Register"
          // 1. Mostra componentes de formulário de
          // registro.
          // 2. Esconde componentes de formulário de
          // login.
      if (sender == okBtn && event == "click")
        if (loginOrRegister.checked)
          // Tenta encontrar um usuário usando as
          // credenciais de login.
          if (!found)
            // Mostra uma mensagem de erro acima do
            // campo login.
        else
          // 1. Cria uma conta de usuário usando dados dos
          // campos de registro.
          // 2. Loga aquele usuário.
          // ...

// Os componentes se comunicam com o mediador usando a interface
// do mediador. Graças a isso, você pode usar os mesmos
// componentes em outros contextos ao ligá-los com diferentes
```

```
// objetos mediadores.
class Component is
  field dialog: Mediator

  constructor Component(dialog) is
    this.dialog = dialog

  method click() is
    dialog.notify(this, "click")

  method keypress() is
    dialog.notify(this, "keypress")

// Componentes concretos não falam entre si. Eles têm apenas um
// canal de comunicação, que é enviar notificações para o
// mediador.
class Button extends Component is
  // ...

class Textbox extends Component is
  // ...

class Checkbox extends Component is
  method check() is
    dialog.notify(this, "check")
  // ...
```

Aplicabilidade

Utilize o padrão Mediator quando é difícil mudar algumas das classes porque elas estão firmemente acopladas a várias outras classes.

O padrão lhe permite extrair todas as relações entre classes para uma classe separada, isolando quaisquer mudanças para um componente específico do resto dos componentes.

Utilize o padrão quando você não pode reutilizar um componente em um programa diferente porque ele é muito dependente de outros componentes.

Após você aplicar o Mediator, componentes individuais se tornam alheios aos outros componentes. Eles ainda podem se comunicar entre si, mas de forma indireta, através do objeto mediador. Para reutilizar um componente em uma aplicação diferente, você precisa fornecer a ele uma nova classe mediadora.

Utilize o Mediator quando você se encontrar criando um monte de subclasses para componentes apenas para reutilizar algum comportamento básico em vários contextos.

Como todas as relações entre componentes estão contidas dentro do mediador, é fácil definir novas maneiras para esses componentes colaborarem introduzindo novas classes mediadoras, sem ter que mudar os próprios componentes.

Como implementar

1. Identifique um grupo de classes firmemente acopladas que se beneficiariam de estar mais independentes (por exemplo, para uma manutenção ou reutilização mais fácil dessas classes).

2. Declare a interface do mediador e descreva o protocolo de comunicação desejado entre os mediadores e os diversos componentes. Na maioria dos casos, um único método para receber notificações de componentes é suficiente.

   Essa interface é crucial quando você quer reutilizar classes componente em diferentes contextos. Desde que o componente trabalhe com seu mediador através da interface genérica, você pode ligar o componente com diferentes implementações do mediador.

3. Implemente a classe concreta do mediador. Essa classe se beneficia por armazenar referências a todos os componentes que gerencia.

4. Você pode ainda ir além e fazer que o mediador fique responsável pela criação e destruição de objetos componente. Após isso, o mediador pode montar uma **fábrica** ou uma **fachada**.

5. Componentes devem armazenar uma referência ao objeto do mediador. A conexão é geralmente estabelecida no construtor do componente, onde o objeto mediador é passado como um argumento.

6. Mude o código dos componentes para que eles chamem o método de notificação do mediador ao invés de métodos de outros componentes. Extraia o código que envolve chamar os outros componentes para a classe do mediador. Execute esse código sempre que o mediador receba notificações daquele componente.

Prós e contras

✓ *Princípio de responsabilidade única*. Você pode extrair as comunicações entre vários componentes para um único lugar, tornando as de mais fácil entendimento e manutenção.

✓ *Princípio aberto/fechado*. Você pode introduzir novos mediadores sem ter que mudar os próprios componentes.

✓ Você pode reduzir o acoplamento entre os vários componentes de um programa.

✓ Você pode reutilizar componentes individuais mais facilmente.

✕ Com o tempo um mediador pode evoluir para um **Objeto Deus**.

Relações com outros padrões

- O **Chain of Responsibility**, **Command**, **Mediator** e **Observer** abrangem várias maneiras de se conectar remetentes e destinatários de pedidos:

  - O *Chain of Responsibility* passa um pedido sequencialmente ao longo de um corrente dinâmica de potenciais destinatários até que um deles atua no pedido.
  - O *Command* estabelece conexões unidirecionais entre remetentes e destinatários.
  - O *Mediator* elimina as conexões diretas entre remetentes e destinatários, forçando-os a se comunicar indiretamente através de um objeto mediador.
  - O *Observer* permite que destinatários inscrevam-se ou cancelem sua inscrição dinamicamente para receber pedidos.

- O **Facade** e o **Mediator** têm trabalhos parecidos: eles tentam organizar uma colaboração entre classes firmemente acopladas.

  - O *Facade* define uma interface simplificada para um subsistema de objetos, mas ele não introduz qualquer nova funcionalidade. O próprio subsistema não está ciente da fachada. Objetos dentro do subsistema podem se comunicar diretamente.
  - O *Mediator* centraliza a comunicação entre componentes do sistema. Os componentes só sabem do objeto mediador e não se comunicam diretamente.

- A diferença entre o **Mediator** e o **Observer** é bem obscura. Na maioria dos casos, você pode implementar qualquer um desses padrões; mas às vezes você pode aplicar ambos simultaneamente. Vamos ver como podemos fazer isso.

O objetivo primário do *Mediator* é eliminar dependências múltiplas entre um conjunto de componentes do sistema. Ao invés disso, esses componentes se tornam dependentes de um único objeto mediador. O objetivo do *Observer* é estabelecer comunicações de uma via dinâmicas entre objetos, onde alguns deles agem como subordinados de outros.

Existe uma implementação popular do padrão Mediator que depende do *Observer*. O objeto mediador faz o papel de um publicador, e os componentes agem como assinantes que inscrevem-se ou removem a inscrição aos eventos do mediador. Quando o *Mediator* é implementado dessa forma, ele pode parecer muito similar ao *Observer*.

Quando você está confuso, lembre-se que você pode implementar o padrão Mediator de outras maneiras. Por exemplo, você pode ligar permanentemente todos os componentes ao mesmo objeto mediador. Essa implementação não se parece com o *Observer* mas ainda irá ser uma instância do padrão Mediator.

Agora imagine um programa onde todos os componentes se tornaram publicadores permitindo conexões dinâmicas entre si. Não haverá um objeto mediador centralizado, somente um conjunto distribuído de observadores.

MEMENTO

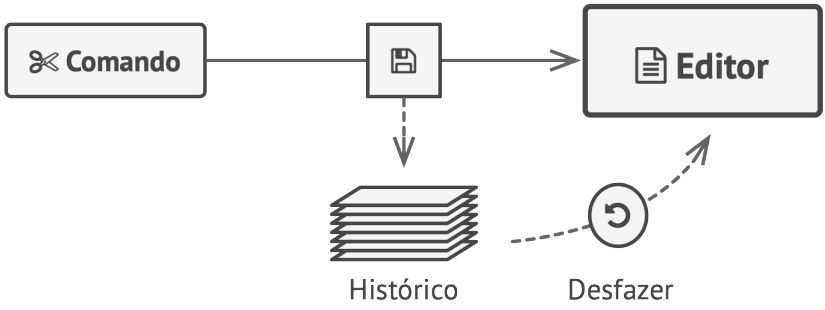
Também conhecido como: Lembrança, Retrato, Snapshot

O **Memento** é um padrão de projeto comportamental que permite que você salve e restaure o estado anterior de um objeto sem revelar os detalhes de sua implementação.

☹ Problema

Imagine que você está criando uma aplicação de editor de texto. Além da simples edição de texto, seu editor pode formatar o texto, inserir imagens em linha, etc.

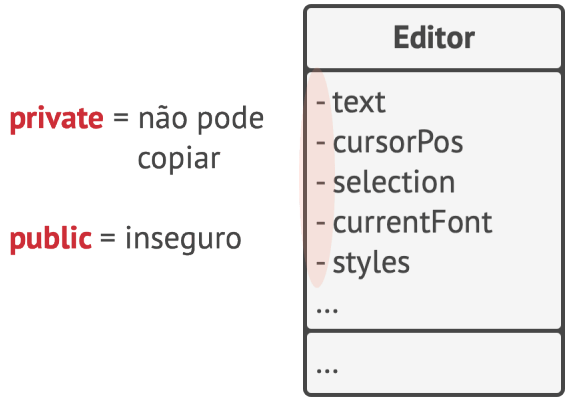
Em determinado momento você decide permitir que os usuários desfaçam quaisquer operações realizadas no texto. Essa funcionalidade tem se tornado tão comum nos últimos anos que, hoje em dia, as pessoas esperam que toda aplicação a tenha. Para a implementação você decide usar a abordagem direta. Antes de realizar qualquer operação, a aplicação grava o estado de todos os objetos e salva eles em algum armazenamento. Mais tarde, quando um usuário decide reverter a ação, a aplicação busca o último retrato do histórico e usa ele para restaurar o estado de todos os objetos.



Antes de executar uma operação, a aplicação salva um retrato do estado dos objetos, que pode mais tarde ser usada para restaurá-los a seu estado anterior.

Vamos pensar sobre esses retratos de estado. Como exatamente você produziria um? Você provavelmente teria que percorrer todos os campos de um objeto e copiar seus valores para o armazenamento. Contudo, isso só funcionaria se o objeto tiver poucas restrições de acesso a seu conteúdo. Infelizmente, a maioria dos objetos reais não deixa os outros espiarem dentro deles assim tão facilmente, escondendo todos os dados significativos em campos privados.

Ignore esse problema por enquanto e vamos assumir que nossos objetos comportam-se como hippies: preferindo relações abertas e mantendo seus estado público. Embora essa abordagem resolveria o problema imediato e permitiria que você produzisse retratos dos estados de seus objetos à vontade, ela ainda tem vários problemas graves. No futuro, você pode decidir refatorar algumas das classes do editor, ou adicionar e remover alguns campos. Parece fácil, mas isso também exigiria mudar as classes responsáveis por copiar o estado dos objetos afetados.



Como fazer uma cópia do estado privado de um objeto?

E tem mais. Vamos considerar os próprios “retratos” do estado do editor. Que dados ele contém? No mínimo dos mínimos ele terá o próprio texto, coordenadas do cursor, posição atual do scroll, etc. Para fazer um retrato você teria que coletar todos esses valores e colocá-los em algum tipo de contêiner.

O mais provável é que você vai armazenar muitos desses objetos contêineres dentro de alguma lista que representaria o histórico. Portanto os contêineres terminariam sendo objetos de uma classe. A classe não teria muitos métodos, mas vários campos que espelhariam o estado do editor. Para permitir que objetos sejam escritos e lidos no e do retrato, você teria que provavelmente tornar seus campos públicos. Isso iria expor todos os estados do editor, privados ou não. Outras classes se tornariam dependentes de cada pequena mudança na classe do retrato, o que poderia acontecer dentro dos campos privados e métodos sem afetar as classes externas.

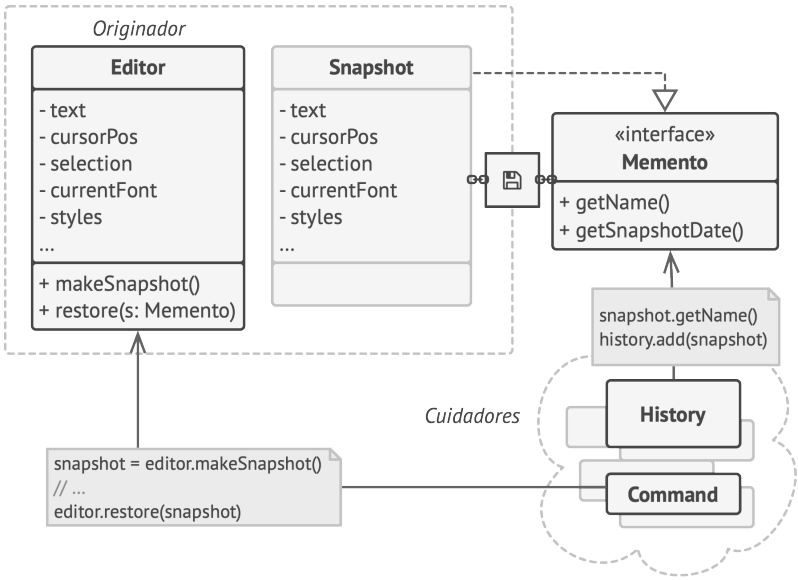
Parece que chegamos em um beco sem saída: você ou expõe todos os detalhes internos das classes tornando-as frágeis, ou restringe o acesso ao estado delas, tornando impossível produzir retratos. Existe alguma outra maneira de implementar o "desfazer"?

😊 Solução

Todos os problemas que vivenciamos foram causados por um encapsulamento quebrado. Alguns objetos tentaram fazer mais do que podiam. Para coletar os dados necessários para fazer uma ação, eles invadiram o espaço privado de outros objetos ao invés de deixar esses objetos fazer a verdadeira ação.

O padrão Memento delega a criação dos retratos do estado para o próprio dono do estado, o objeto *originador*. Portanto, ao invés de outros objetos tentarem copiar o estado do editor "a partir do lado de fora", a própria classe do editor pode fazer o retrato já que tem acesso total a seu próprio estado.

O padrão sugere armazenar a cópia do estado de um objeto em um objeto especial chamado *memento*. Os conteúdos de um memento não são acessíveis para qualquer outro objeto exceto aquele que o produziu. Outros objetos podem se comunicar com mementos usando uma interface limitada que pode permitir a recuperação dos metadados do retrato (data de criação, nome a operação efetuada, etc.), mas não ao estado do objeto original contido no retrato.



O originador tem acesso total ao memento, enquanto que o cuidador pode acessar somente os metadados

Tal regra restritiva permite que você armazene mementos dentro de outros objetos, geralmente chamados de *cuidadores*. Uma vez que o cuidador trabalha com o memento apenas por meio de uma interface limitada, ele não será capaz de mexer com o estado armazenado dentro do memento. Ao mesmo tempo, o originador tem acesso total a todos os campos dentro do memento, permitindo que ele o restaure ao seu estado anterior à vontade.

Em nosso exemplo de editor de texto, nós podemos criar uma classe histórico separada para agir como a cuidadora. Uma pilha de mementos armazenada dentro da cuidadora irá crescer a cada vez que o editor estiver prestes a executar uma operação. Você pode até mesmo renderizar essa pilha dentro do

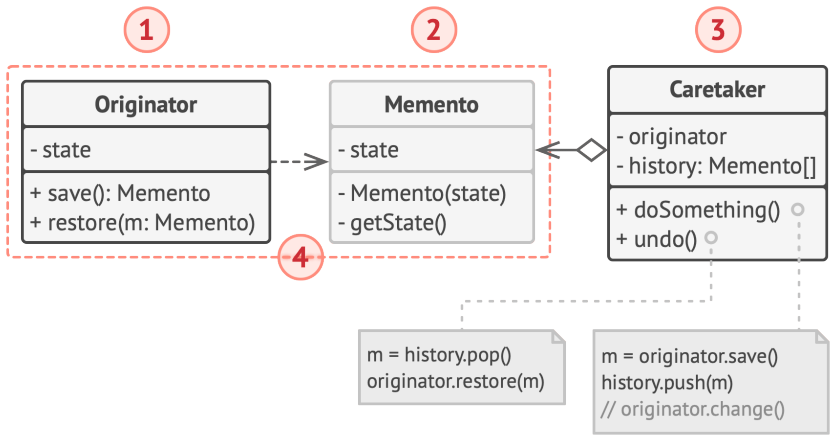
UI da aplicação, mostrando o histórico de operações realizadas anteriormente para um usuário.

Quando um usuário aciona o desfazer, o histórico pega o memento mais recente da pilha e o passa de volta para o editor, pedindo uma reversão. Já que o editor tem acesso total ao memento, ele muda seu próprio estado com os valores obtidos do memento.

Estrutura

Implementação baseada em classes aninhadas

A implementação clássica do padrão dependente do apoio para classes aninhadas, disponível em muitas linguagens de programação populares (tais como C++, C#, e Java).



1. A classe **Originadora** pode produzir retratos de seu próprio estado, bem como restaurar seu estado de retratos quando necessário.

2. O **Memento** é um objeto de valor que age como um retrato do estado da originadora. É uma prática comum fazer o memento imutável e passar os dados para ele apenas uma vez, através do construtor.

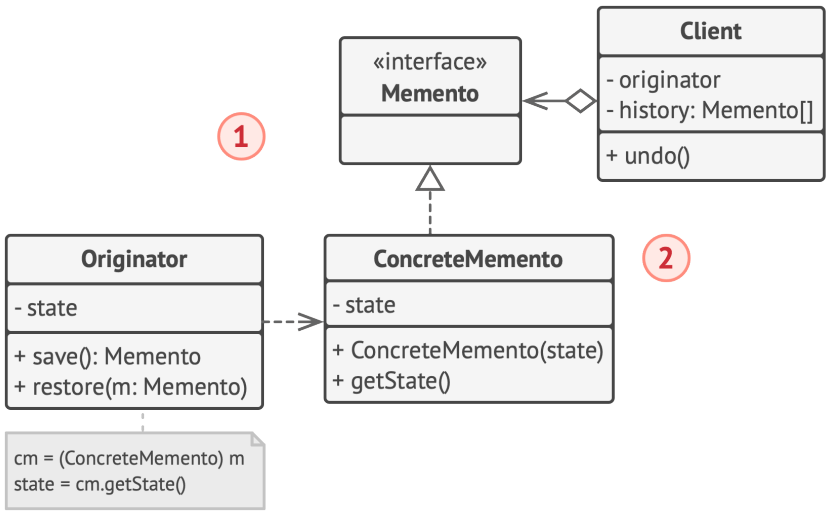
3. A **Cuidadora** sabe não só "quando" e "por quê" capturar o estado da originadora, mas também quando o estado deve ser restaurado.

   Uma cuidadora pode manter registros do histórico da originadora armazenando os mementos em um pilha. Quando a originadora precisa voltar atrás no histórico, a cuidadora busca o memento mais do topo da pilha e o passa para o método de restauração da originadora.

4. Nessa implementação, a classe memento está aninhada dentro da originadora. Isso permite que a originadora acesse os campos e métodos do memento, mesmo que eles tenham sido declarados privados. Por outro lado, a cuidadora tem um acesso muito limitado aos campos do memento, que permite ela armazenar os mementos em uma pilha, mas não permite mexer com seu estado.

Implementação baseada em uma interface intermediária

Há uma implementação alternativa, adequada para linguagens de programação que não suportam classes aninhadas (sim, PHP, estou falando de você).



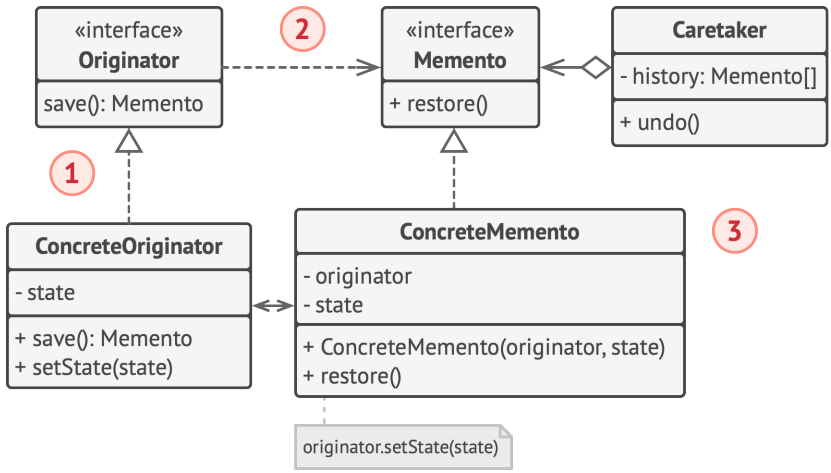
1. Na ausência de classes aninhadas, você pode restringir o acesso aos campos do memento ao estabelecer uma convenção para que cuidadoras possam trabalhar com um memento através apenas de uma interface intermediária explicitamente declarada, que só declararia os métodos relacionados aos metadados do memento.

2. Por outro lado, as originadoras podem trabalhar com um objeto memento diretamente, acessando campos e métodos declarados na classe memento. O lado ruim dessa abordagem

é que você precisa declarar todos os membros do memento como públicos.

Implementação com um encapsulamento ainda mais estrito

Há ainda outra implementação que é útil quando você não quer deixar a mínima chance de outras classes acessarem o estado da originadora através do memento.



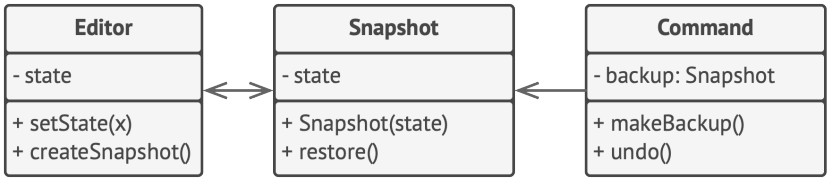
1. Essa implementação permite ter múltiplos tipos de originadoras e mementos. Cada originadora trabalha com uma classe memento correspondente. Nem as originadoras nem os mementos expõem seu estado para ninguém.

2. Cuidadoras são agora explicitamente restritas de mudar o estado armazenado nos mementos. Além disso, a classe cuidadora se torna independente da originadora porque o método de restauração agora está definido na classe memento.

3. Cada memento se torna ligado à originadora que o produziu. A originadora passa a si mesmo para o construtor do memento, junto com os valores de seu estado. Graças a relação íntima entre essas classes, um memento pode restaurar o estado da sua originadora, desde que esta última tenha definido os setters apropriados.

Pseudocódigo

Este exemplo usa o padrão Memento junto com o padrão **Command** para armazenar retratos do estado de um editor de texto complexo e restaurá-lo para um estado anterior desses retratos quando necessário



Salvando retratos do estado de um editor de texto.

Os objetos comando agem como cuidadores. Eles buscam o memento do editor antes de executar operações relacionadas aos comandos. Quando um usuário tenta desfazer o comando mais recente, o editor pode usar o memento armazenando naquele comando para reverter a si mesmo para o estado anterior.

A classe memento não declara qualquer campo público, getters, ou setters. Portanto nenhum objeto pode alterar seus

conteúdos. Os mementos estão ligados ao objeto do editor que os criou. Isso permite que um memento restaure o estado do editor ligado a ele passando os dados via setters no objeto do editor. Já que mementos são ligados com objetos do editor específicos, você pode fazer sua aplicação suportar diversas janelas independentes do editor com uma pilha centralizada de desfazer.

```
// O originador tem alguns dados importantes que podem mudar com
// o tempo. Ele também define um método para salvar seu estado
// dentro de um memento e outro método para restaurar o estado
// dele.
class Editor is
  private field text, curX, curY, selectionWidth

  method setText(text) is
    this.text = text

  method setCursor(x, y) is
    this.curX = curX
    this.curY = curY

  method setSelectionWidth(width) is
    this.selectionWidth = width

  // Salva o estado atual dentro de um memento.
  method createSnapshot():Snapshot is
    // O memento é um objeto imutável; é por isso que o
    // originador passa seu estado para os parâmetros do
    // construtor do memento.
    return new Snapshot(this, text, curX, curY, selectionWidth)
```

```
24
25  // A classe memento armazena o estado anterior do editor.
26  class Snapshot is
27    private field editor: Editor
28    private field text, curX, curY, selectionWidth
29
30    constructor Snapshot(editor, text, curX, curY, selectionWidth) is
31      this.editor = editor
32      this.text = text
33      this.curX = curX
34      this.curY = curY
35      this.selectionWidth = selectionWidth
36
37    // Em algum momento, um estado anterior do editor pode ser
38    // restaurado usando um objeto memento.
39    method restore() is
40      editor.setText(text)
41      editor.setCursor(curX, curY)
42      editor.setSelectionWidth(selectionWidth)
43
44  // Um objeto comando pode agir como cuidador. Neste caso, o
45  // comando obtém o memento antes que ele mude o estado do
46  // originador. Quando o undo(desfazer) é solicitado, ele
47  // restaura o estado do originador a partir de um memento.
48  class Command is
49    private field backup: Snapshot
50
51    method makeBackup() is
52      backup = editor.createSnapshot()
53
54    method undo() is
55      if (backup != null)
```

```
56          backup.restore()
57      // ...
```

Aplicabilidade

Utilize o padrão Memento quando você quer produzir retratos do estado de um objeto para ser capaz de restaurar um estado anterior do objeto.

O padrão Memento permite que você faça cópias completas do estado de um objeto, incluindo campos privados, e armazená-los separadamente do objeto. Embora a maioria das pessoas vão lembrar desse padrão graças ao caso "desfazer", ele também é indispensável quando se está lidando com transações (isto é, se você precisa reverter uma operação quando se depara com um erro).

Utilize o padrão quando o acesso direto para os campos/getters/setters de um objeto viola seu encapsulamento.

O Memento faz o próprio objeto ser responsável por criar um retrato de seu estado. Nenhum outro objeto pode ler o retrato, fazendo do estado original do objeto algo seguro e confiável.

Como implementar

1. Determine qual classe vai fazer o papel de originadora. É importante saber se o programa usa um objeto central deste tipo ou múltiplos objetos pequenos.

2. Crie a classe memento. Um por um, declare o conjunto dos campos que espelham os campos declarados dentro da classe originadora.

3. Faça a classe memento ser imutável. Um memento deve aceitar os dados apenas uma vez, através do construtor. A classe não deve ter setters.

4. Se a sua linguagem de programação suporta classes aninhadas, aninhe o memento dentro da originadora. Se não, extraia uma interface em branco da classe memento e faça todos os outros objetos usá-la para se referir ao memento. Você pode adicionar algumas operações de metadados para a interface, mas nada que exponha o estado da originadora.

5. Adicione um método para produção de mementos na classe originadora. A originadora deve passar seu estado para o memento através de um ou múltiplos argumentos do construtor do memento.

   O tipo de retorno do método deve ser o da interface que você extraiu na etapa anterior (assumindo que você extraiu alguma

coisa). Por debaixo dos panos, o método de produção de memento deve funcionar diretamente com a classe memento.

6. Adicione um método para restaurar o estado da classe originadora para sua classe. Ele deve aceitar o objeto memento como um argumento. Se você extraiu uma interface na etapa anterior, faça-a do tipo do parâmetro. Neste caso, você precisa converter o tipo do objeto que está vindo para a classe memento, uma vez que a originadora precisa de acesso total a aquele objeto.

7. A cuidadora, estando ela representando um objeto comando, um histórico, ou algo completamente diferente, deve saber quando pedir novos mementos da originadora, como armazená-los, e quando restaurar a originadora com um memento em particular.

8. O elo entre cuidadoras e originadoras deve ser movido para dentro da classe memento. Neste caso, cada memento deve se conectar com a originadora que criou ele. O método de restauração também deve ser movido para a classe memento. Contudo, isso tudo faria sentido somente se a classe memento estiver aninhada dentro da originadora ou a classe originadora fornece setters suficientes para sobrescrever seu estado.

Prós e contras

✓ Você pode produzir retratos do estado de um objeto sem violar seu encapsulamento.

✓ Você pode simplificar o código da originadora permitindo que a cuidadora mantenha o histórico do estado da originadora.

✕ A aplicação pode consumir muita RAM se os clientes criarem mementos com muita frequência.

✕ Cuidadoras devem acompanhar o ciclo de vida da originadora para serem capazes de destruir mementos obsoletos.

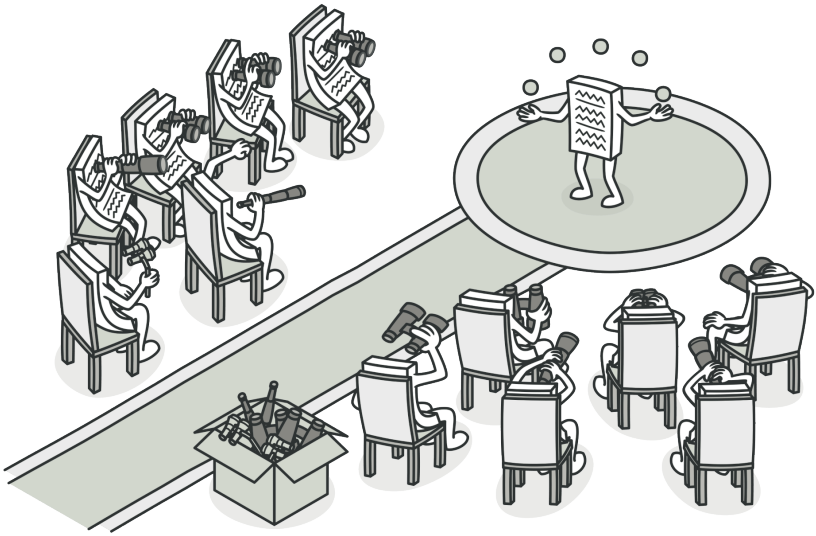
✕ A maioria das linguagens de programação dinâmicas, tais como PHP, Python, e JavaScript, não conseguem garantir que o estado dentro do memento permaneça intacto.

⇄ Relações com outros padrões

- Você pode usar o **Command** e o **Memento** juntos quando implementando um "desfazer". Neste caso, os comandos são responsáveis pela realização de várias operações sobre um objeto alvo, enquanto que os mementos salvam o estado daquele objeto momentos antes de um comando ser executado.

- Você pode usar o **Memento** junto com o **Iterator** para capturar o estado de iteração atual e revertê-lo se necessário.

- Algumas vezes o **Prototype** pode ser uma alternativa mais simples a um **Memento**. Isso funciona se o objeto, o estado no qual você quer armazenar na história, é razoavelmente intuitivo e não tem ligações para recursos externos, ou as ligações são fáceis de se restabelecer.



OBSERVER

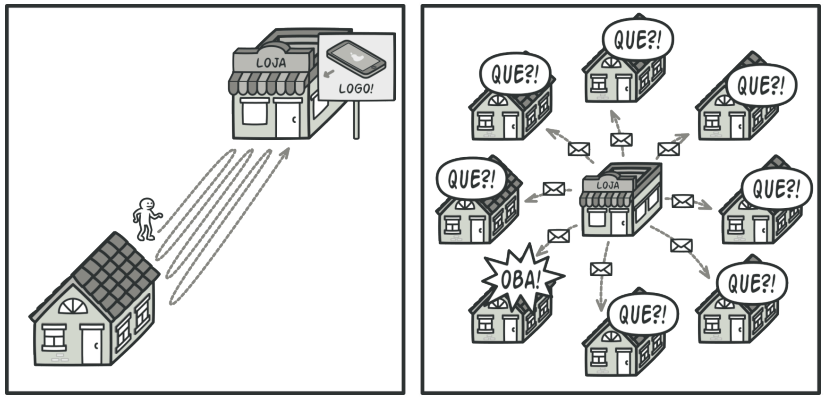
Também conhecido como: Observador, Assinante do evento, Event-Subscriber, Escutador, Listener

O **Observer** é um padrão de projeto comportamental que permite que você defina um mecanismo de assinatura para notificar múltiplos objetos sobre quaisquer eventos que aconteçam com o objeto que eles estão observando.

☹ Problema

Imagine que você tem dois tipos de objetos: um `Cliente` e uma `Loja`. O cliente está muito interessado em uma marca particular de um produto (digamos que seja um novo modelo de iPhone) que logo deverá estar disponível na loja.

O cliente pode visitar a loja todos os dias e checar a disponibilidade do produto. Mas enquanto o produto ainda está a caminho, a maioria desses visitas serão em vão.



Visitando a loja vs. enviando spam

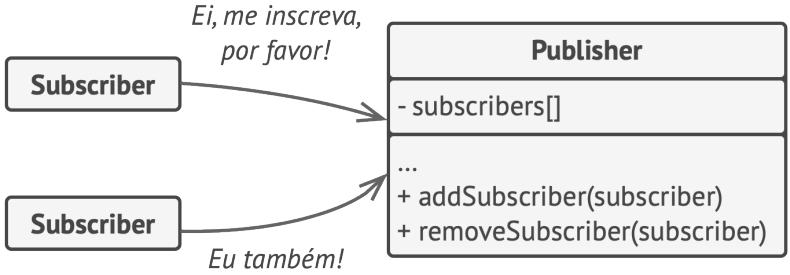
Por outro lado, a loja poderia mandar milhares de emails (que poderiam ser considerados como spam) para todos os clientes cada vez que um novo produto se torna disponível. Isso salvaria alguns clientes de incontáveis viagens até a loja. Porém, ao mesmo tempo, irritaria outros clientes que não estão interessados em novos produtos.

Parece que temos um conflito. Ou o cliente gasta tempo verificando a disponibilidade do produto ou a loja gasta recursos notificando os clientes errados.

😊 Solução

O objeto que tem um estado interessante é quase sempre chamado de *sujeito*, mas já que ele também vai notificar outros objetos sobre as mudanças em seu estado, nós vamos chamá-lo de *publicador*. Todos os outros objetos que querem saber das mudanças do estado do publicador são chamados de *assinantes*.

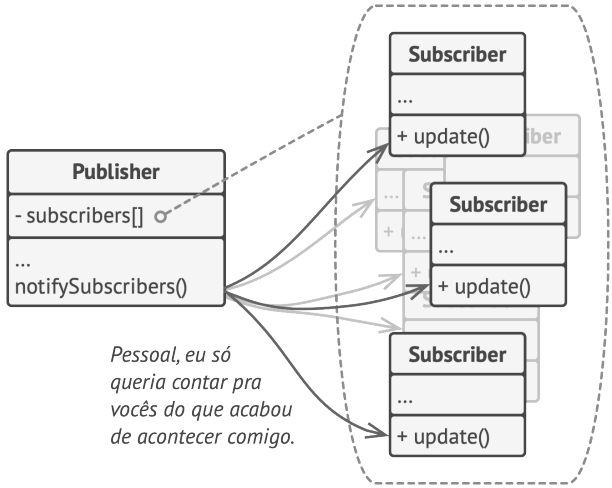
O padrão Observer sugere que você adicione um mecanismo de assinatura para a classe publicadora para que objetos individuais possam assinar ou desassinar uma corrente de eventos vindo daquela publicadora. Nada tema! Nada é complicado como parece. Na verdade, esse mecanismo consiste em 1) um vetor para armazenar uma lista de referências aos objetos do assinante e 2) alguns métodos públicos que permitem adicionar assinantes e removê-los da lista.



Um mecanismo de assinatura permite que objetos individuais inscrevam-se a notificações de eventos.

Agora, sempre que um evento importante acontece com a publicadora, ele passa para seus assinantes e chama um método específico de notificação em seus objetos.

Aplicações reais podem ter dúzias de diferentes classes assinantes que estão interessadas em acompanhar eventos da mesma classe publicadora. Você não iria querer acoplar a publicadora a todas essas classes. Além disso, você pode nem estar ciente de algumas delas de antemão se a sua classe publicadora deve ser usada por outras pessoas.



A publicadora notifica os assinantes chamando um método específico de notificação em seus objetos.

É por isso que é crucial que todos os assinantes implementem a mesma interface e que a publicadora comunique-se com eles apenas através daquela interface. Essa interface deve declarar o método de notificação junto com um conjunto de parâmetros

que a publicadora pode usar para passar alguns dados contextuais junto com a notificação.

Se a sua aplicação tem diferentes tipos de publicadoras e você quer garantir que seus assinantes são compatíveis com todas elas, você pode ir além e fazer todas as publicadoras seguirem a mesma interface. Essa interface precisa apenas descrever alguns métodos de inscrição. A interface permitirá assinantes observar o estado das publicadoras sem se acoplar a suas classes concretas.

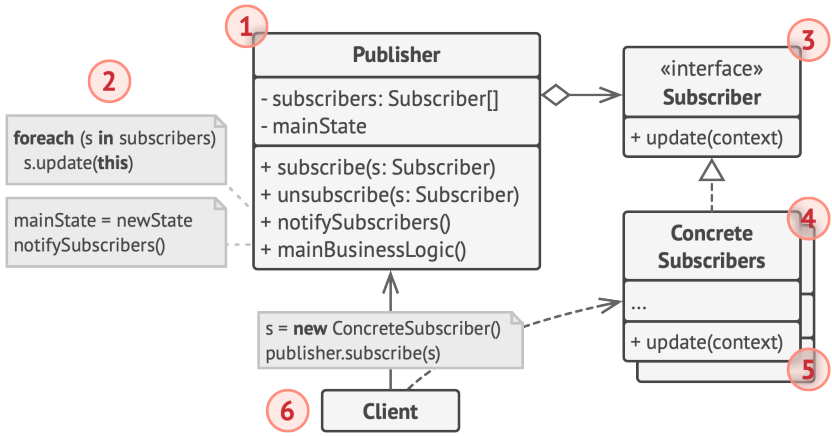
Analogia com o mundo real

Assinaturas de revistas e jornais.

Se você assinar um jornal ou uma revista, você não vai mais precisar ir até a banca e ver se a próxima edição está disponível. Ao invés disso a publicadora manda novas edições diretamente para sua caixa de correio após a publicação ou até mesmo com antecedência.

A publicadora mantém uma lista de assinantes e sabe em quais revistas eles estão interessados. Os assinantes podem deixar essa lista a qualquer momento quando desejarem que a publicadora pare de enviar novas revistas para eles.

Estrutura



1. A **Publicadora** manda eventos de interesse para outros objetos. Esses eventos ocorrem quando a publicadora muda seu estado ou executa algum comportamento. As publicadoras contêm uma infraestrutura de inscrição que permite novos assinantes se juntar aos atuais assinantes ou deixar a lista.

2. Quando um novo evento acontece, a publicadora percorre a lista de assinantes e chama o método de notificação declarado na interface do assinante em cada objeto assinante.

3. A interface do **Assinante** declara a interface de notificação. Na maioria dos casos ela consiste de um único método

`atualizar`. O método pode ter vários parâmetros que permite que a publicadora passe alguns detalhes do evento junto com a atualização.

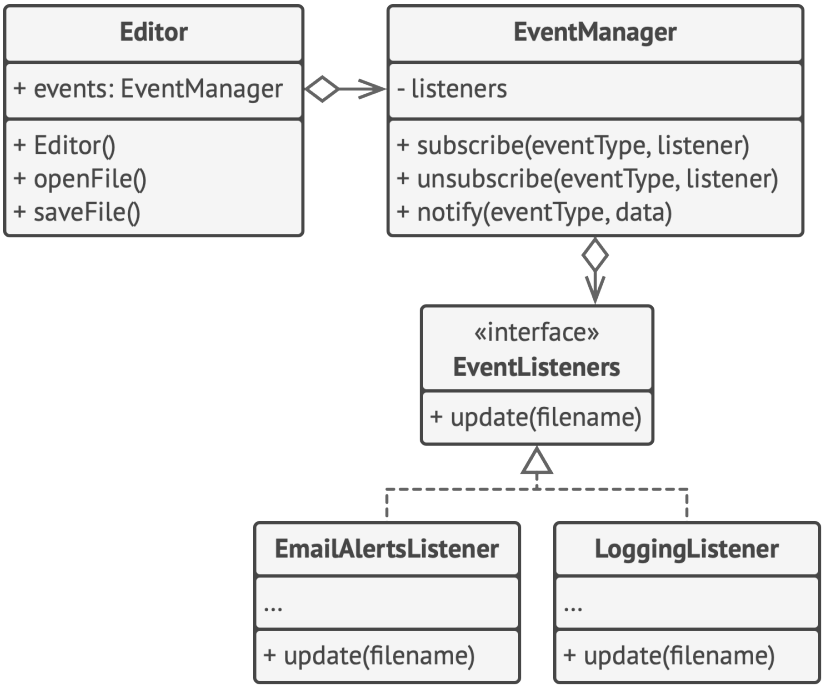
4. **Assinantes Concretos** realizam algumas ações em resposta às notificações enviadas pela publicadora. Todas essas classes devem implementar a mesma interface para que a publicadora não fique acoplada à classes concretas.

5. Geralmente, assinantes precisam de alguma informação contextual para lidar com a atualização corretamente. Por esse motivo, as publicadoras quase sempre passam algum dado de contexto como argumentos do método de notificação. A publicadora pode passar a si mesmo como um argumento, permitindo que o assinante recupere quaisquer dados diretamente.

6. O **Cliente** cria a publicadora e os objetos assinantes separadamente e então registra os assinantes para as atualizações da publicadora.

Pseudocódigo

Neste exemplo o padrão **Observer** permite que um objeto editor de texto notifique outros objetos de serviço sobre mudanças em seu estado.



Notificando objetos sobre eventos que aconteceram com outros objetos.

A lista de assinantes é compilada dinamicamente: objetos podem começar ou parar de ouvir às notificações durante a execução do programa, dependendo do comportamento desejado pela sua aplicação.

Nesta implementação, a classe do editor não mantém a lista de assinatura por si mesmo. Ele delega este trabalho para um objeto ajudante especial devotado a fazer apenas isso. Você pode melhorar aquele objeto para servir como um enviador de eventos centralizado, permitindo que qualquer objeto aja como uma publicadora.

Adicionar novos assinantes ao programa não exige mudança nas classes publicadoras existentes, desde que elas trabalhem com todos os assinantes através da mesma interface.

```
// A classe publicadora base inclui o código de gerenciamento de
// inscrições e os métodos de notificação.
class EventManager is
  private field listeners: hash map of event types and listeners

  method subscribe(eventType, listener) is
    listeners.add(eventType, listener)

  method unsubscribe(eventType, listener) is
    listeners.remove(eventType, listener)

  method notify(eventType, data) is
    foreach (listener in listeners.of(eventType)) do
      listener.update(data)

// O publicador concreto contém a verdadeira lógica de negócio
// que é de interesse para alguns assinantes. Nós podemos
// derivar essa classe a partir do publicador base, mas isso nem
// sempre é possível na vida real devido a possibilidade do
// publicador concreto já ser uma subclasse. Neste caso, você
// pode remendar a lógica de inscrição com a composição, como
// fizemos aqui.
class Editor is
  public field events: EventManager
  private field file: File

  constructor Editor() is
```

```
    events = new EventManager()

  // Métodos da lógica de negócio podem notificar assinantes
  // acerca de mudanças.
  method openFile(path) is
    this.file = new File(path)
    events.notify("open", file.name)

  method saveFile() is
    file.write()
    events.notify("save", file.name)

  // ...


// Aqui é a interface do assinante. Se sua linguagem de
// programação suporta tipos funcionais, você pode substituir
// toda a hierarquia do assinante por um conjunto de funções.
interface EventListener is
  method update(filename)

// Assinantes concretos reagem a atualizações emitidas pelo
// publicador a qual elas estão conectadas.
class LoggingListener implements EventListener is
  private field log: File
  private field message

  constructor LoggingListener(log_filename, message) is
    this.log = new File(log_filename)
    this.message = message

  method update(filename) is
```

```
60      log.write(replace('%s',filename,message))
61
62  class EmailAlertsListener implements EventListener is
63    private field email: string
64
65    constructor EmailAlertsListener(email, message) is
66      this.email = email
67      this.message = message
68
69    method update(filename) is
70      system.email(email, replace('%s',filename,message))
71
72
73  // Uma aplicação pode configurar publicadores e assinantes
74  // durante o tempo de execução.
75  class Application is
76    method config() is
77      editor = new Editor()
78
79      logger = new LoggingListener(
80        "/path/to/log.txt",
81        "Someone has opened the file: %s")
82      editor.events.subscribe("open", logger)
83
84      emailAlerts = new EmailAlertsListener(
85        "admin@example.com",
86        "Someone has changed the file: %s")
87      editor.events.subscribe("save", emailAlerts)
```

Aplicabilidade

Utilize o padrão Observer quando mudanças no estado de um objeto podem precisar mudar outros objetos, e o atual conjunto de objetos é desconhecido de antemão ou muda dinamicamente.

Você pode vivenciar esse problema quando trabalhando com classes de interface gráfica do usuário. Por exemplo, você criou classes de botões customizados, e você quer deixar os clientes colocar algum código customizado para seus botões para que ele ative sempre que usuário aperta um botão.

O padrão Observer permite que qualquer objeto que implemente a interface do assinante possa se inscrever para notificações de eventos em objetos da publicadora. Você pode adicionar o mecanismo de inscrição em seus botões, permitindo que o cliente coloque seu próprio código através de classes assinantes customizadas.

Utilize o padrão quando alguns objetos em sua aplicação devem observar outros, mas apenas por um tempo limitado ou em casos específicos.

A lista de inscrição é dinâmica, então assinantes podem entrar e sair da lista sempre que quiserem.

Como implementar

1. Olhe para sua lógica do negócio e tente quebrá-la em duas partes: a funcionalidade principal, independente de outros códigos, irá agir como publicadora; o resto será transformado em um conjunto de classes assinantes.

2. Declare a interface do assinante. No mínimo, ela deve declarar um único método `atualizar`.

3. Declare a interface da publicadora e descreva um par de métodos para adicionar um objeto assinante e removê-lo da lista. Lembre-se que publicadoras somente devem trabalhar com assinantes através da interface do assinante.

4. Decida onde colocar a lista atual de assinantes e a implementação dos métodos de inscrição. Geralmente este código se parece o mesmo para todos os tipos de publicadoras, então o lugar óbvio para colocá-lo é dentro de uma classe abstrata derivada diretamente da interface da publicadora. Publicadoras concretas estendem aquela classe, herdando o comportamento de inscrição.

   Contudo, se você está aplicando o padrão para uma hierarquia de classe já existente, considere uma abordagem baseada na composição: coloque a lógica da inscrição dentro de um objeto separado, e faça todos as publicadoras reais usá-la.

5. Crie as classes publicadoras concretas. A cada vez que algo importante acontece dentro de uma publicadora, ela deve notificar seus assinantes.

6. Implemente os métodos de notificação de atualização nas classes assinantes concretas. A maioria dos assinantes precisarão de dados contextuais sobre o evento. Eles podem ser passados como argumentos do método de notificação.

   Mas há outra opção. Ao receber uma notificação, o assinante pode recuperar os dados diretamente da notificação. Neste caso, a publicadora deve passar a si mesma através do método de atualização. A opção menos flexível é ligar uma publicadora ao assinante permanentemente através do construtor.

7. O cliente deve criar todas os assinantes necessários e registrá-los com suas publicadoras apropriadas.

Prós e contras

✓ *Princípio aberto/fechado*. Você pode introduzir novas classes assinantes sem ter que mudar o código da publicadora (e vice versa se existe uma interface publicadora).

✓ Você pode estabelecer relações entre objetos durante a execução.

✕ Assinantes são notificados em ordem aleatória

⇄ Relações com outros padrões

- O **Chain of Responsibility**, **Command**, **Mediator** e **Observer** abrangem várias maneiras de se conectar remetentes e destinatários de pedidos:

  - O *Chain of Responsibility* passa um pedido sequencialmente ao longo de um corrente dinâmica de potenciais destinatários até que um deles atua no pedido.
  - O *Command* estabelece conexões unidirecionais entre remetentes e destinatários.
  - O *Mediator* elimina as conexões diretas entre remetentes e destinatários, forçando-os a se comunicar indiretamente através de um objeto mediador.
  - O *Observer* permite que destinatários inscrevam-se ou cancelem sua inscrição dinamicamente para receber pedidos.

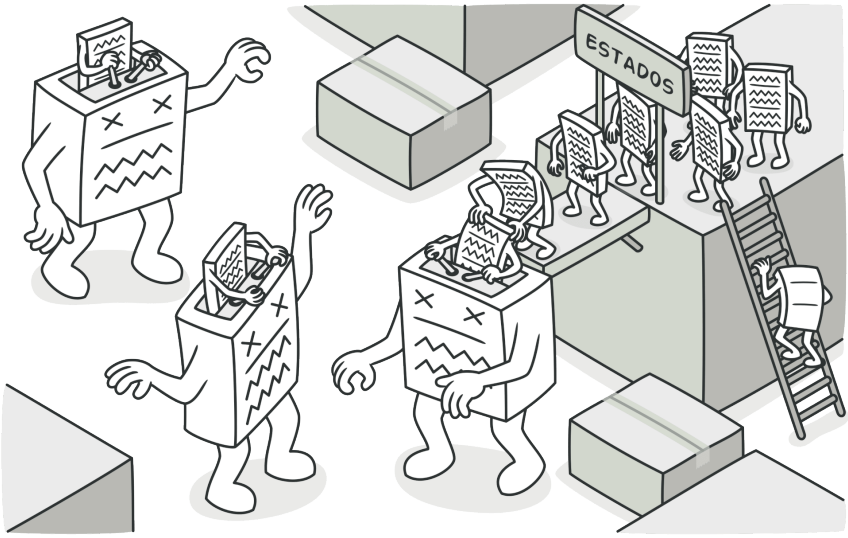
- A diferença entre o **Mediator** e o **Observer** é bem obscura. Na maioria dos casos, você pode implementar qualquer um desses padrões; mas às vezes você pode aplicar ambos simultaneamente. Vamos ver como podemos fazer isso.

  O objetivo primário do *Mediator* é eliminar dependências múltiplas entre um conjunto de componentes do sistema. Ao invés disso, esses componentes se tornam dependentes de um único objeto mediador. O objetivo do *Observer* é estabelecer comunicações de uma via dinâmicas entre objetos, onde alguns deles agem como subordinados de outros.

Existe uma implementação popular do padrão Mediator que depende do *Observer*. O objeto mediador faz o papel de um publicador, e os componentes agem como assinantes que inscrevem-se ou removem a inscrição aos eventos do mediador. Quando o *Mediator* é implementado dessa forma, ele pode parecer muito similar ao *Observer*.

Quando você está confuso, lembre-se que você pode implementar o padrão Mediator de outras maneiras. Por exemplo, você pode ligar permanentemente todos os componentes ao mesmo objeto mediador. Essa implementação não se parece com o *Observer* mas ainda irá ser uma instância do padrão Mediator.

Agora imagine um programa onde todos os componentes se tornaram publicadores permitindo conexões dinâmicas entre si. Não haverá um objeto mediador centralizado, somente um conjunto distribuído de observadores.



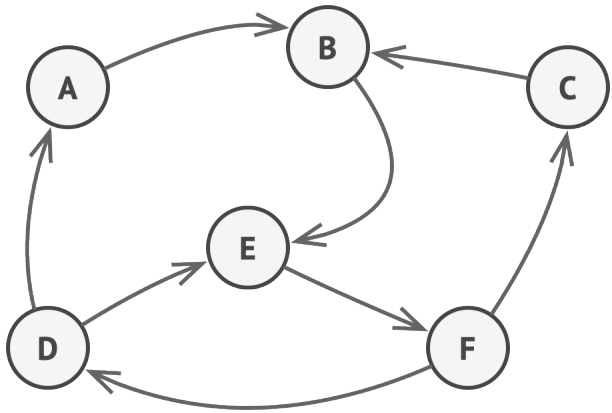
STATE

Também conhecido como: Estado

O **State** é um padrão de projeto comportamental que permite que um objeto altere seu comportamento quando seu estado interno muda. Parece como se o objeto mudasse de classe.

☹ Problema

O padrão State é intimamente relacionado com o conceito de uma **Máquina de Estado Finito**.



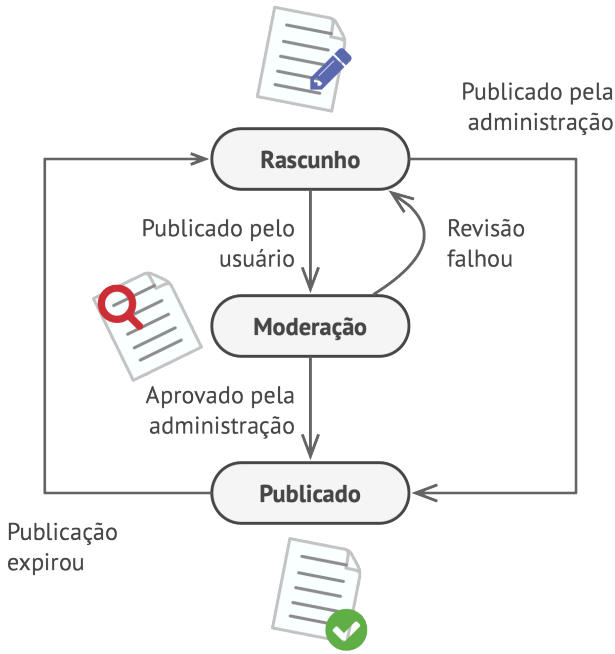
Máquina de Estado Finito.

A ideia principal é que, em qualquer dado momento, há um número *finito* de *estados* que um programa possa estar. Dentro de qualquer estado único, o programa se comporta de forma diferente, e o programa pode ser trocado de um estado para outro instantaneamente. Contudo, dependendo do estado atual, o programa pode ou não trocar para outros estados. Essas regras de troca, chamadas *transições*, também são finitas e pré determinadas.

Você também pode aplicar essa abordagem para objetos. Imagine que nós temos uma classe `Documento`. Um documento pode estar em um de três estados: `Rascunho`, `Moderação` e

`Publicado`. O método `publicar` do documento funciona um pouco diferente em cada estado:

- No `Rascunho`, ele move o documento para a moderação.
- Na `Moderação` ele torna o documento público, mas apenas se o usuário atual é um administrador.
- No `Publicado` ele não faz nada.



Possíveis estados e transições de um objeto documento.

Máquinas de estado são geralmente implementadas com muitos operadores de condicionais (`if` ou `switch`) que selecionam o comportamento apropriado dependendo do estado atual do objeto. Geralmente esse "estado" é apenas um con-

junto de valores dos campos do objeto. Mesmo se você nunca ouviu falar sobre máquinas de estado finito antes, você provavelmente já implementou um estado ao menos uma vez. A seguinte estrutura de código lembra alguma coisa para você?

```
class Document is
  field state: string
  // ...
  method publish() is
    switch (state)
      "draft":
        state = "moderation"
        break
      "moderation":
        if (currentUser.role == 'admin')
          state = "published"
        break
      "published":
        // Não fazer nada.
        break
  // ...
```

A maior fraqueza de uma máquina de estados baseada em condicionais se revela quando começamos a adicionar mais e mais estados e comportamentos baseados em estados para a classe `Documento`. A maioria dos métodos irá conter condicionais monstruosas que selecionam o comportamento apropriado de um método de acordo com o estado atual. Um código como esse é muito difícil de se fazer manutenção porque qualquer

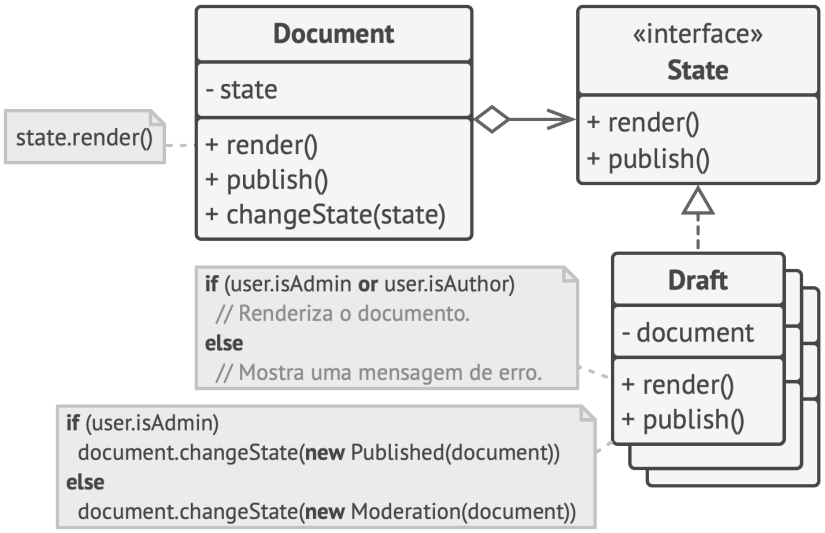
mudança na lógica de transição pode necessitar de mudanças de condicionais de estado em todos os métodos.

O problema tende a ficar maior a medida que o projeto evolui. É muito difícil prever todos os possíveis estados e transições no estágio inicial de projeto. Portanto, uma máquina de estados enxuta, construída com um número limitado de condicionais pode se tornar uma massa inchada e disforme com o tempo.

😊 Solução

O padrão State sugere que você crie novas classes para todos os estados possíveis de um objeto e extraia todos os comportamentos específicos de estados para dentro dessas classes.

Ao invés de implementar todos os comportamentos por conta própria, o objeto original, chamado *contexto*, armazena uma referência para um dos objetos de estado que representa seu estado atual, e delega todo o trabalho relacionado aos estados para aquele objeto.



O documento delega o trabalho para um objeto de estado.

Para fazer a transição do contexto para outro estado, substitua o objeto do estado ativo por outro objeto que represente o novo estado. Isso é possível somente se todas as classes de estado seguirem a mesma interface e o próprio contexto funcione com esses objetos através daquela interface.

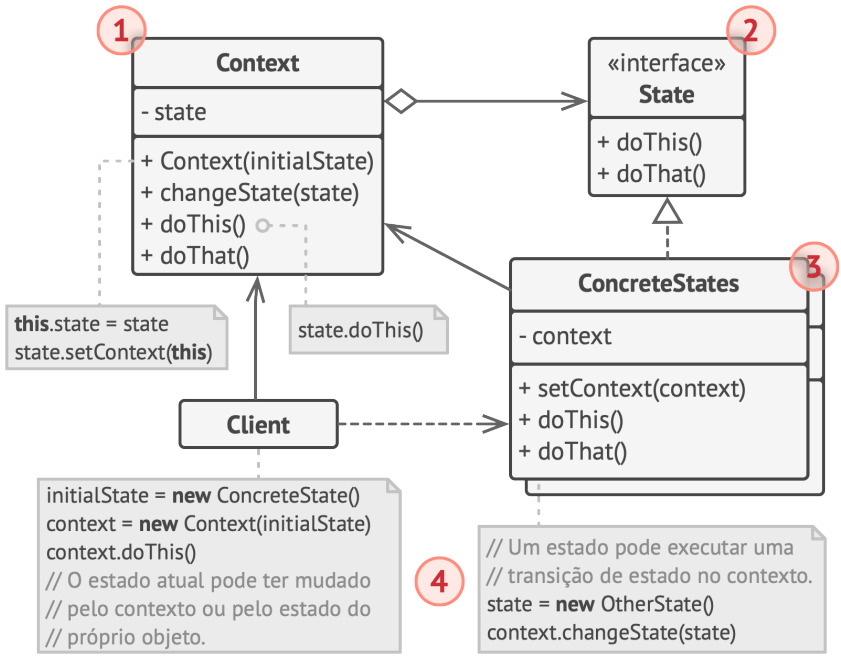
Essa estrutura pode ser parecida com o padrão **Strategy**, mas há uma diferença chave. No padrão State, os estados em particular podem estar cientes de cada um e iniciar transições de um estado para outro, enquanto que estratégias quase nunca sabem sobre as outras estratégias.

Analogia com o mundo real

Os botões e interruptores de seu smartphone comportam-se de forma diferente dependendo do estado atual do dispositivo:

- Quando o telefone está desbloqueado, apertar os botões leva eles a executar várias funções.
- Quando o telefone está bloqueado, apertar qualquer botão leva a desbloquear a tela.
- Quando a carga da bateria está baixa, apertar qualquer botão mostra a tela de carregamento.

Estrutura



1. O **Contexto** armazena uma referência a um dos objetos concretos de estado e delega a eles todos os trabalhos específicos de estado. O contexto se comunica com o objeto estado através da interface do estado. O contexto expõe um setter para passar a ele um novo objeto de estado.

2. A interface do **Estado** declara métodos específicos a estados. Esses métodos devem fazer sentido para todos os estados concretos porque você não quer alguns dos seus estados tendo métodos inúteis que nunca irão ser chamados.

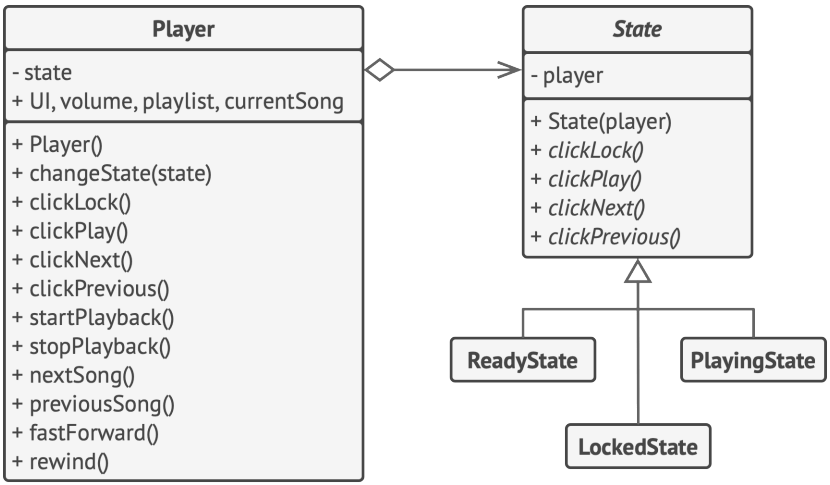
3. Os **Estados Concretos** fornecem suas próprias implementações para os métodos específicos de estados. Para evitar duplicação ou código parecido em múltiplos estados, você pode fornecer classes abstratas intermediárias que encapsulam alguns dos comportamentos comuns.

   Objetos de estado podem armazenar referências retroativas para o objeto de contexto. Através dessa referência o estado pode buscar qualquer informação desejada do objeto contexto, assim como iniciar transições de estado.

4. Ambos os estados de contexto e concretos podem configurar o próximo estado do contexto e realizar a atual transição de estado ao substituir o objeto estado ligado ao contexto.

Pseudocódigo

Neste exemplo, o padrão **State** permite que os mesmos controles de tocador de mídia se comportem diferentemente, dependendo do atual estado do tocador.



Exemplo da troca do comportamento de um objeto com objetos de estado.

O objeto principal do tocador está sempre ligado ao objeto estado que realiza a maior parte do trabalho para o tocador. Algumas ações substituem o objeto do estado atual do tocador por outro, que muda a maneira do tocador reagir às interações do usuário.

```
1  // A classe AudioPlayer age como um contexto. Ela também mantém
2  // uma referência para uma instância de uma das classes de
3  // estado que representa o atual estado do tocador de áudio.
```

```
class AudioPlayer is
  field state: State
  field UI, volume, playlist, currentSong

  constructor AudioPlayer() is
    this.state = new ReadyState(this)

    // O contexto delega o manuseio das entradas do usuário
    // para um objeto de estado. Naturalmente, o resultado
    // depende de qual estado está ativo, uma vez que cada
    // estado pode lidar com as entradas de forma diferente.
    UI = new UserInterface()
    UI.lockButton.onClick(this.clickLock)
    UI.playButton.onClick(this.clickPlay)
    UI.nextButton.onClick(this.clickNext)
    UI.prevButton.onClick(this.clickPrevious)

  // Outros objetos devem ser capazes de trocar o estado ativo
  // do tocador.
  method changeState(state: State) is
    this.state = state

  // Métodos de UI delegam a execução para o estado ativo.
  method clickLock() is
    state.clickLock()
  method clickPlay() is
    state.clickPlay()
  method clickNext() is
    state.clickNext()
  method clickPrevious() is
    state.clickPrevious()

```

```
    // Um estado pode chamar alguns métodos de serviço no
    // contexto.
    method startPlayback() is
      // ...
    method stopPlayback() is
      // ...
    method nextSong() is
      // ...
    method previousSong() is
      // ...
    method fastForward(time) is
      // ...
    method rewind(time) is
      // ...

// A classe de estado base declara métodos que todos os estados
// concretos devem implementar e também fornece uma referência
// anterior ao objeto de contexto associado com o estado.
// Estados podem usar a referência anterior para realizar a
// transição contexto para outro estado.
abstract class State is
    protected field player: AudioPlayer

    // O contexto passa a si mesmo através do construtor do
    // estado. Isso pode ajudar o estado a recuperar alguns
    // dados de contexto úteis se for necessário.
    constructor State(player) is
      this.player = player

    abstract method clickLock()
    abstract method clickPlay()
    abstract method clickNext()
```

```
68    abstract method clickPrevious()
69
70
71  // Estados concretos implementam vários comportamentos
72  // associados com um estado do contexto.
73  class LockedState extends State is
74
75    // Quando você desbloqueia um tocador bloqueado, ele vai
76    // assumir um dos dois estados.
77    method clickLock() is
78      if (player.playing)
79        player.changeState(new PlayingState(player))
80      else
81        player.changeState(new ReadyState(player))
82
83    method clickPlay() is
84      // Bloqueado, então não faz nada.
85
86    method clickNext() is
87      // Bloqueado, então não faz nada.
88
89    method clickPrevious() is
90      // Bloqueado, então não faz nada.
91
92
93  // Eles também podem ativar transições de estado no contexto.
94  class ReadyState extends State is
95    method clickLock() is
96      player.changeState(new LockedState(player))
97
98    method clickPlay() is
99      player.startPlayback()
```

```
100       player.changeState(new PlayingState(player))
101
102     method clickNext() is
103       player.nextSong()
104
105     method clickPrevious() is
106       player.previousSong()
107
108
109   class PlayingState extends State is
110     method clickLock() is
111       player.changeState(new LockedState(player))
112
113     method clickPlay() is
114       player.stopPlayback()
115       player.changeState(new ReadyState(player))
116
117     method clickNext() is
118       if (event.doubleclick)
119         player.nextSong()
120       else
121         player.fastForward(5)
122
123     method clickPrevious() is
124       if (event.doubleclick)
125         player.previous()
126       else
127         player.rewind(5)
```

Aplicabilidade

Utilize o padrão State quando você tem um objeto que se comporta de maneira diferente dependendo do seu estado atual, quando o número de estados é enorme, e quando o código estado específico muda com frequência.

O padrão sugere que você extraia todo o código estado específico para um conjunto de classes distintas. Como resultado, você pode adicionar novos estados ou mudar os existentes independentemente uns dos outros, reduzindo o custo da manutenção.

Utilize o padrão quando você tem uma classe populada com condicionais gigantes que alteram como a classe se comporta de acordo com os valores atuais dos campos da classe.

O padrão State permite que você extraia ramificações dessas condicionais para dentro de métodos de classes correspondentes. Ao fazer isso, você também limpa para fora da classe principal os campos temporários e os métodos auxiliares envolvidos no código estado específico.

Utilize o State quando você tem muito código duplicado em muitos estados parecidos e transições de uma máquina de estado baseada em condições.

O padrão State permite que você componha hierarquias de classes estado e reduza a duplicação ao extrair código comum para dentro de classes abstratas base.

Como implementar

1. Decida qual classe vai agir como contexto. Poderia ser uma classe existente que já tenha código dependente do estado; ou uma nova classe, se o código específico ao estado estiver distribuído em múltiplas classes.

2. Declare a interface do estado. Embora ela vai espelhar todos os métodos declarados no contexto, mire apenas para aqueles que possam conter comportamento específico ao estado.

3. Para cada estado real, crie uma classe que deriva da interface do estado. Então vá para os métodos do contexto e extraia todo o código relacionado a aquele estado para dentro de sua nova classe.

   Ao mover o código para a classe estado, você pode descobrir que ela depende de membros privados do contexto. Há várias maneiras de contornar isso:

   - Torne esses campos ou métodos públicos.
   - Transforme o comportamento que você está extraindo para um método público dentro do contexto e chame-o na classe estado. Essa maneira é feia mas rápida, e você pode sempre consertá-la mais tarde.

- Aninhe as classes estado dentro da classe contexto, mas apenas se sua linguagem de programação suporta classes aninhadas.

4. Na classe contexto, adicione um campo de referência do tipo de interface do estado e um setter público que permite sobrescrever o valor daquele campo.

5. Vá até o método do contexto novamente e substitua as condicionais de estado vazias por chamadas aos métodos correspondentes do objeto estado.

6. Para trocar o estado do contexto, crie uma instância de uma das classes estado e a passe para o contexto. Você pode fazer isso dentro do próprio contexto, ou em vários estados, ou no cliente. Aonde quer que isso seja feito, a classe se torna dependente da classe estado concreta que ela instanciou.

Prós e contras

✓ *Princípio de responsabilidade única*. Organiza o código relacionado a estados particulares em classes separadas.

✓ *Princípio aberto/fechado*. Introduz novos estados sem mudar classes de estado ou contexto existentes.

✓ Simplifica o código de contexto ao eliminar condicionais de máquinas de estado pesadas.

✕ Aplicar o padrão pode ser um exagero se a máquina de estado só tem alguns estados ou raramente muda eles.

⇄ Relações com outros padrões

- O **Bridge**, **State**, **Strategy** (e de certa forma o **Adapter**) têm estruturas muito parecidas. De fato, todos esses padrões estão baseados em composição, o que é delegar o trabalho para outros objetos. Contudo, eles todos resolvem problemas diferentes. Um padrão não é apenas uma receita para estruturar seu código de uma maneira específica. Ele também pode comunicar a outros desenvolvedores o problema que o padrão resolve.

- O **State** pode ser considerado como uma extensão do **Strategy**. Ambos padrões são baseados em composição: eles mudam o comportamento do contexto ao delegar algum trabalho para objetos auxiliares. O *Strategy* faz esses objetos serem completamente independentes e alheios entre si. Contudo, o *State* não restringe dependências entre estados concretos, permitindo que eles alterem o estado do contexto à vontade.

STRATEGY

Também conhecido como: Estratégia

O **Strategy** é um padrão de projeto comportamental que permite que você defina uma família de algoritmos, coloque-os em classes separadas, e faça os objetos deles intercambiáveis.

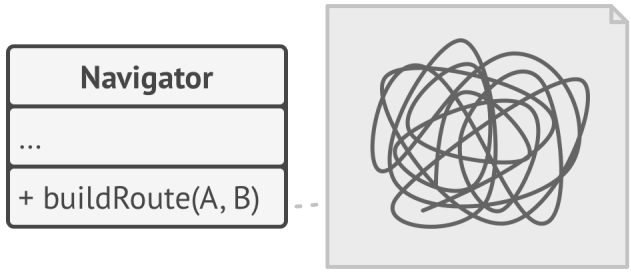
☹ Problema

Um dia você decide criar uma aplicação de navegação para viajantes casuais. A aplicação estava centrada em um mapa bonito que ajudava os usuários a se orientarem rapidamente em uma cidade.

Uma das funcionalidades mais pedidas para a aplicação era o planejamento automático de rotas. Um usuário deveria ser capaz de entrar com um endereço e ver a rota mais rápida no mapa.

A primeira versão da aplicação podia apenas construir rotas sobre rodovias, e isso agradou muito quem viaja de carro. Porém aparentemente, nem todo mundo dirige em suas férias. Então com a próxima atualização você adicionou uma opção de construir rotas de caminhada. Logo após isso você adicionou outra opção para permitir que as pessoas usem o transporte público.

Contudo, isso foi apenas o começo. Mais tarde você planejou adicionar um construtor de rotas para ciclistas. E mais tarde, outra opção para construir rotas até todas as atrações turísticas de uma cidade.



O código do navegador ficou muito inchado.

Embora da perspectiva de negócio a aplicação tenha sido um sucesso, a parte técnica causou a você muitas dores de cabeça. Cada vez que você adicionava um novo algoritmo de roteamento, a classe principal do navegador dobrava de tamanho. Em determinado momento, o monstro se tornou algo muito difícil de se manter.

Qualquer mudança a um dos algoritmos, seja uma simples correção de bug ou um pequeno ajuste no valor das ruas, afetava toda a classe, aumentando a chance de criar um erro no código já existente.

Além disso, o trabalho em equipe se tornou ineficiente. Seus companheiros de equipe, que foram contratados após ao bem sucedido lançamento do produto, se queixavam que gastavam muito tempo resolvendo conflitos de fusão. Implementar novas funcionalidades necessitava mudanças na classe gigantesca, conflitando com os códigos criados por outras pessoas.

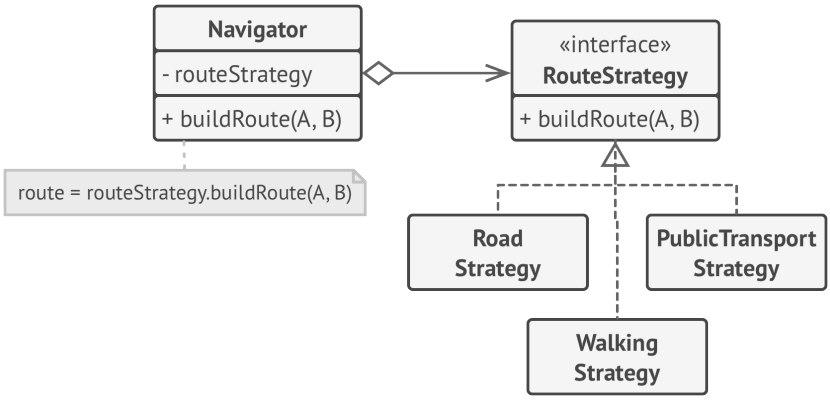
😊 Solução

O padrão Strategy sugere que você pegue uma classe que faz algo específico em diversas maneiras diferentes e extraia todos esses algoritmos para classes separadas chamadas *estratégias*.

A classe original, chamada *contexto*, deve ter um campo para armazenar uma referência para um dessas estratégias. O contexto delega o trabalho para um objeto estratégia ao invés de executá-lo por conta própria.

O contexto não é responsável por selecionar um algoritmo apropriado para o trabalho. Ao invés disso, o cliente passa a estratégia desejada para o contexto. Na verdade, o contexto não sabe muito sobre as estratégias. Ele trabalha com todas elas através de uma interface genérica, que somente expõe um único método para acionar o algoritmo encapsulado dentro da estratégia selecionada.

Desta forma o contexto se torna independente das estratégias concretas, então você pode adicionar novos algoritmos ou modificar os existentes sem modificar o código do contexto ou outras estratégias.

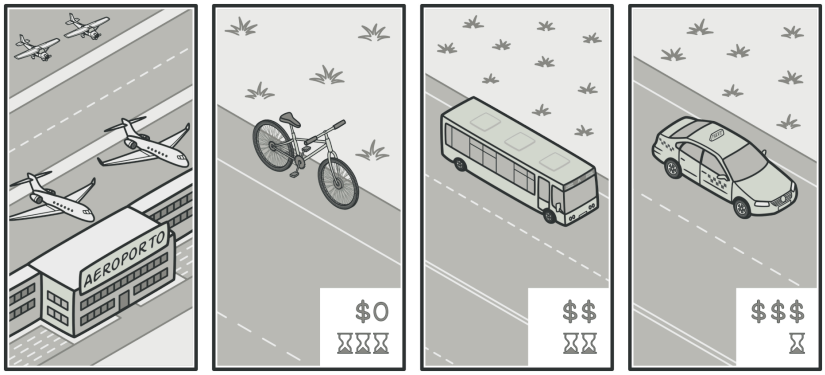


Estratégias de planejamento de rotas.

Em nossa aplicação de navegação, cada algoritmo de roteamento pode ser extraído para sua própria classe com um único método `construirRota`. O método aceita uma origem e um destino e retorna uma coleção de pontos da rota.

Mesmo dando os mesmos argumentos, cada classe de roteamento pode construir uma rota diferente, a classe navegadora principal não se importa qual algoritmo está selecionado uma vez que seu trabalho primário é renderizar um conjunto de pontos num mapa. A classe tem um método para trocar a estratégia ativa de rotas, então seus clientes, bem como os botões na interface de usuário, podem substituir o comportamento de rotas selecionado por um outro.

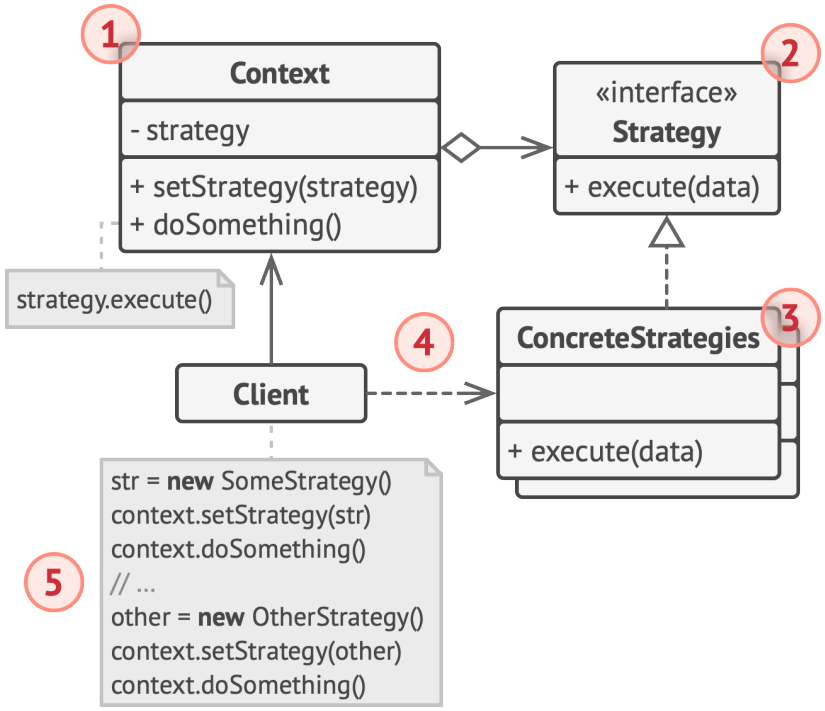
Analogia com o mundo real



Várias estratégias para se chegar ao aeroporto.

Imagine que você tem que chegar ao aeroporto. Você pode pegar um ônibus, pedir um táxi, ou subir em sua bicicleta. Essas são suas estratégias de transporte. Você pode escolher uma das estratégias dependendo de fatores como orçamento ou restrições de tempo.

Estrutura



1. O **Contexto** mantém uma referência para uma das estratégias concretas e se comunica com esse objeto através da interface da estratégia.

2. A interface **Estratégia** é comum à todas as estratégias concretas. Ela declara um método que o contexto usa para executar uma estratégia.

3. **Estratégias Concretas** implementam diferentes variações de um algoritmo que o contexto usa.

4. O contexto chama o método de execução no objeto estratégia ligado cada vez que ele precisa rodar um algoritmo. O contexto não sabe qual tipo de estratégia ele está trabalhando ou como o algoritmo é executado.

5. O **Cliente** cria um objeto estratégia específico e passa ele para o contexto. O contexto expõe um setter que permite o cliente mudar a estratégia associada com contexto durante a execução.

Pseudocódigo

Neste exemplo, o contexto usa múltiplas **estratégias** para executar várias operações aritméticas.

```
// A interface estratégia declara operações comuns a todas as
// versões suportadas de algum algoritmo. O contexto usa essa
// interface para chamar o algoritmo definido pelas estratégias
// concretas.
interface Strategy is
  method execute(a, b)

// Estratégias concretas implementam o algoritmo enquanto seguem
// a interface estratégia base. A interface faz delas
// intercomunicáveis no contexto.
class ConcreteStrategyAdd implements Strategy is
  method execute(a, b) is
    return a + b

class ConcreteStrategySubtract implements Strategy is
```

```
    method execute(a, b) is
      return a - b

class ConcreteStrategyMultiply implements Strategy is
    method execute(a, b) is
      return a * b

// O contexto define a interface de interesse para clientes.
class Context is
    // O contexto mantém uma referência para um dos objetos
    // estratégia. O contexto não sabe a classe concreta de uma
    // estratégia. Ele deve trabalhar com todas as estratégias
    // através da interface estratégia.
    private strategy: Strategy

    // Geralmente o contexto aceita uma estratégia através do
    // construtor, e também fornece um setter para que a
    // estratégia possa ser trocado durante o tempo de execução.
    method setStrategy(Strategy strategy) is
      this.strategy = strategy

    // O contexto delega algum trabalho para o objeto estratégia
    // ao invés de implementar múltiplas versões do algoritmo
    // por conta própria.
    method executeStrategy(int a, int b) is
      return strategy.execute(a, b)


// O código cliente escolhe uma estratégia concreta e passa ela
// para o contexto. O cliente deve estar ciente das diferenças
// entre as estratégia para que faça a escolha certa.
class ExampleApplication is
```

```
48    method main() is
49      Cria um objeto contexto.
50
51      Lê o primeiro número.
52      Lê o último número.
53      Lê a ação desejada da entrada do usuário
54
55      if (action == addition) then
56        context.setStrategy(new ConcreteStrategyAdd())
57
58      if (action == subtraction) then
59        context.setStrategy(new ConcreteStrategySubtract())
60
61      if (action == multiplication) then
62        context.setStrategy(new ConcreteStrategyMultiply())
63
64      result = context.executeStrategy(First number, Second number)
65
66      Imprimir resultado.
```

Aplicabilidade

Utilize o padrão Strategy quando você quer usar diferentes variantes de um algoritmo dentro de um objeto e ser capaz de trocar de um algoritmo para outro durante a execução.

O padrão Strategy permite que você altere indiretamente o comportamento de um objeto durante a execução ao associá-lo com diferentes sub-objetos que pode fazer sub-tarefas específicas em diferentes formas.

Utilize o Strategy quando você tem muitas classes parecidas que somente diferem na forma que elas executam algum comportamento.

O padrão Strategy permite que você extraia o comportamento variante para uma hierarquia de classe separada e combine as classes originais em uma, portando reduzindo código duplicado.

Utilize o padrão para isolar a lógica do negócio de uma classe dos detalhes de implementação de algoritmos que podem não ser tão importantes no contexto da lógica.

O padrão Strategy permite que você isole o código, dados internos, e dependências de vários algoritmos do restante do código. Vários clientes podem obter uma simples interface para executar os algoritmos e trocá-los durante a execução do programa.

Utilize o padrão quando sua classe tem um operador condicional muito grande que troca entre diferentes variantes do mesmo algoritmo.

O padrão Strategy permite que você se livre dessa condicional ao extrair todos os algoritmos para classes separadas, todos eles implementando a mesma interface. O objeto original delega a execução de um desses objetos, ao invés de implementar todas as variantes do algoritmo.

Como implementar

1. Na classe contexto, identifique um algoritmo que é sujeito a frequentes mudanças. Pode ser também uma condicional enorme que seleciona e executa uma variante do mesmo algoritmo durante a execução do programa.

2. Declare a interface da estratégia comum para todas as variantes do algoritmo.

3. Um por um, extraia todos os algoritmos para suas próprias classes. Elas devem todas implementar a interface estratégia.

4. Na classe contexto, adicione um campo para armazenar uma referência a um objeto estratégia. Forneça um setter para substituir valores daquele campo. O contexto deve trabalhar com o objeto estratégia somente através da interface estratégia. O contexto pode definir uma interface que deixa a estratégia acessar seus dados.

5. Os Clientes do contexto devem associá-lo com uma estratégia apropriada que coincide com a maneira que esperam que o contexto atue em seu trabalho primário.

Prós e contras

✓ Você pode trocar algoritmos usados dentro de um objeto durante a execução.

- ✓ Você pode isolar os detalhes de implementação de um algoritmo do código que usa ele.
- ✓ Você pode substituir a herança por composição.
- ✓ *Princípio aberto/fechado*. Você pode introduzir novas estratégias sem mudar o contexto.

- ✕ Se você só tem um par de algoritmos e eles raramente mudam, não há motivo real para deixar o programa mais complicado com novas classes e interfaces que vêm junto com o padrão.
- ✕ Os Clientes devem estar cientes das diferenças entre as estratégias para serem capazes de selecionar a adequada.
- ✕ Muitas linguagens de programação modernas tem suporte do tipo funcional que permite que você implemente diferentes versões de um algoritmo dentro de um conjunto de funções anônimas. Então você poderia usar essas funções exatamente como se estivesse usando objetos estratégia, mas sem inchar seu código com classes e interfaces adicionais.

⇄ Relações com outros padrões

- O **Bridge**, **State**, **Strategy** (e de certa forma o **Adapter**) têm estruturas muito parecidas. De fato, todos esses padrões estão baseados em composição, o que é delegar o trabalho para outros objetos. Contudo, eles todos resolvem problemas diferentes. Um padrão não é apenas uma receita para estruturar seu código de uma maneira específica. Ele também pode comunicar a outros desenvolvedores o problema que o padrão resolve.

- O **Command** e o **Strategy** podem ser parecidos porque você pode usar ambos para parametrizar um objeto com alguma ação. Contudo, eles têm propósitos bem diferentes.

    - Você pode usar o *Command* para converter qualquer operação em um objeto. Os parâmetros da operação se transformam em campos daquele objeto. A conversão permite que você atrase a execução de uma operação, transforme-a em uma fila, armazene o histórico de comandos, envie comandos para serviços remotos, etc.

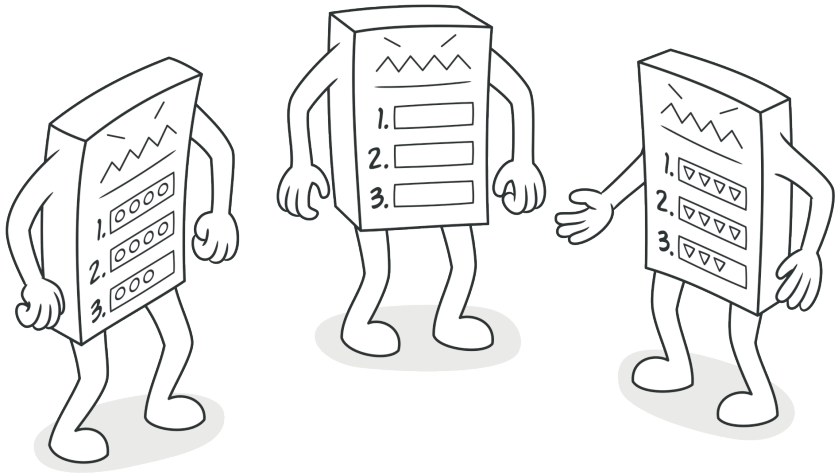
    - Por outro lado, o *Strategy* geralmente descreve diferentes maneiras de fazer a mesma coisa, permitindo que você troque esses algoritmos dentro de uma única classe contexto.

- O **Decorator** permite que você mude a pele de um objeto, enquanto o **Strategy** permite que você mude suas entranhas.

- O **Template Method** é baseado em herança: ele permite que você altere partes de um algoritmo ao estender essas partes em subclasses. O **Strategy** é baseado em composição: você pode alterar partes do comportamento de um objeto ao suprir ele como diferentes estratégias que correspondem a aquele comportamento. O *Template Method* funciona a nível de classe, então é estático. O *Strategy* trabalha a nível de objeto, permitindo que você troque os comportamentos durante a execução.

- O **State** pode ser considerado como uma extensão do **Strategy**. Ambos padrões são baseados em composição: eles mudam o

comportamento do contexto ao delegar algum trabalho para objetos auxiliares. O *Strategy* faz esses objetos serem completamente independentes e alheios entre si. Contudo, o *State* não restringe dependências entre estados concretos, permitindo que eles alterem o estado do contexto à vontade.



TEMPLATE METHOD

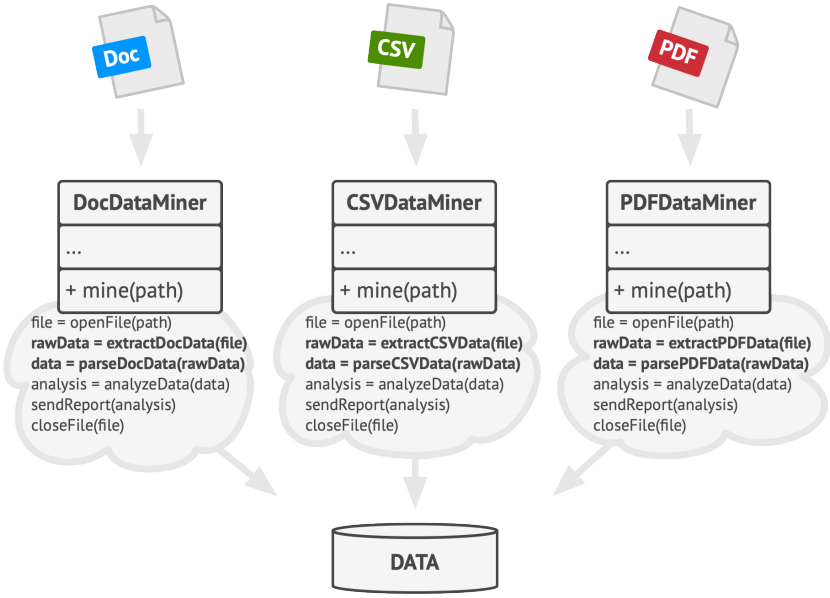
Também conhecido como: Método padrão

O **Template Method** é um padrão de projeto comportamental que define o esqueleto de um algoritmo na superclasse mas deixa as subclasses sobrescreverem etapas específicas do algoritmo sem modificar sua estrutura.

☹ Problema

Imagine que você está criando uma aplicação de mineração de dados que analisa documentos corporativos. Os usuários alimentam a aplicação com documentos em vários formatos (PDF, DOC, CSV), e ela tenta extrair dados significativos desses documentos para um formato uniforme.

A primeira versão da aplicação podia funcionar somente com arquivos DOC. Na versão seguinte, ela era capaz de suportar arquivos CSV. Um mês depois, você a "ensinou" a extrair dados de arquivos PDF.



Classes de mineração de dados continham muito código duplicado.

Em algum momento você percebeu que todas as três classes tem muito código parecido. Embora o código para lidar com vários formatos seja inteiramente diferente em todas as classes, o código para processamento de dados e análise é quase idêntico. Não seria bacana se livrar da duplicação de código, deixando a estrutura do algoritmo intacta?

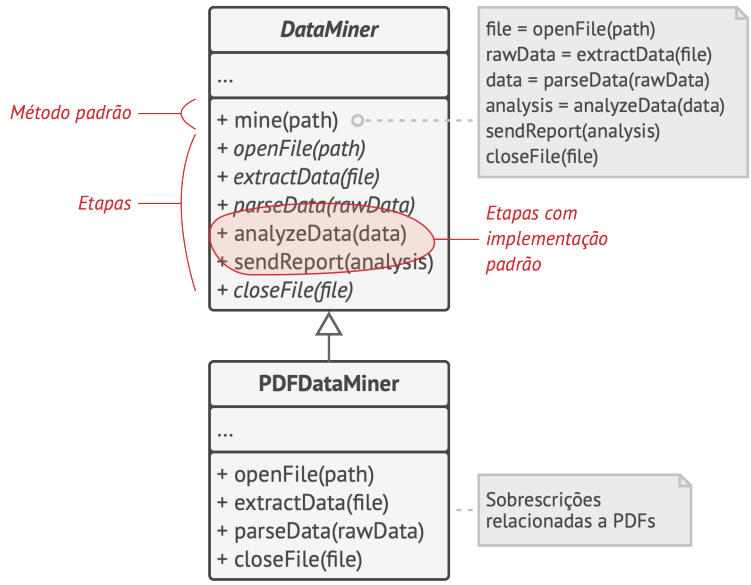
Havia outro problema relacionado com o código cliente que usou essas classes. Ele tinha muitas condicionais que pegavam um curso de ação apropriado dependendo da classe do objeto processador. Se todas as três classes processantes tiverem uma interface comum ou uma classe base, você poderia eliminar as condicionais no código cliente e usar polimorfismo quando chamar métodos em um objeto sendo processado.

😊 Solução

O padrão do Template Method sugere que você quebre um algoritmo em uma série de etapas, transforme essas etapas em métodos, e coloque uma série de chamadas para esses métodos dentro de um único *método padrão*. As etapas podem ser tanto `abstratas`, ou ter alguma implementação padrão. Para usar o algoritmo, o cliente deve fornecer sua própria subclasse, implementar todas as etapas abstratas, e sobrescrever algumas das opcionais se necessário (mas não o próprio método padrão).

Vamos ver como isso vai funcionar com nossa aplicação de mineração de dados. Nós podemos criar uma classe base para

todos os três algoritmos de processamento. Essa classe define um método padrão que consiste de uma série de chamadas para várias etapas de processamento de documentos.



O método padrão quebra o algoritmo em etapas, permitindo que subclasses sobrescrevam essas etapas mas não o método atual.

A princípio nós podemos declarar todos os passos como `abstratos`, forçando as subclasses a fornecer suas próprias implementações para esses métodos. No nosso caso, as subclasses já tem todas as implementações necessárias, então a única coisa que precisamos fazer é ajustar as assinaturas dos métodos para coincidirem com os da superclasse.

Agora vamos ver o que podemos fazer para nos livrarmos do código duplicado. Parece que o código para abrir/fechar arquivos e extrair/analisar os dados são diferentes para vários for-

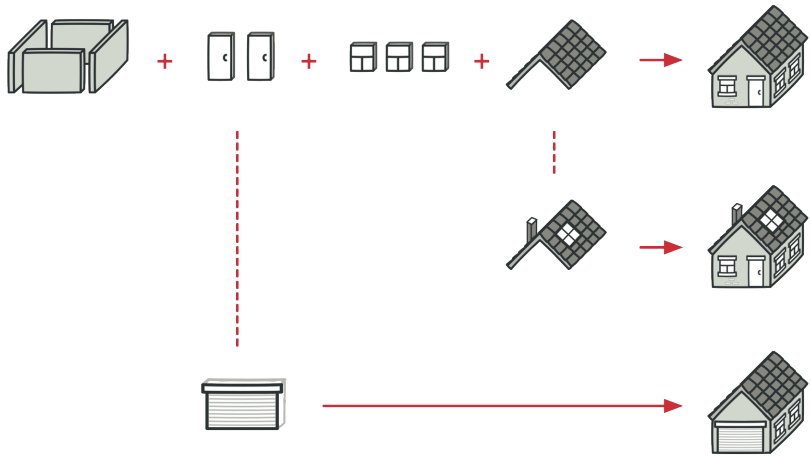
matos de dados, então não tem porque tocar nesses métodos. Contudo, a implementação dessas etapas, tais como analisar os dados brutos e compor relatórios, é muito parecida, então eles podem ser erguidos para a classe base, onde as subclasses podem compartilhar o código.

Como você pode ver, nós temos dois tipos de etapas:

- *etapas abstratas* devem ser implementadas por cada subclasse
- *etapas opcionais* já tem alguma implementação padrão, mas ainda podem ser sobrescritas se necessário.

Existe outro tipo de etapa chamado *ganchos*(hooks). Um gancho é uma etapa opcional com um corpo vazio. Um método padrão poderia funcionar até mesmo se um hook não for sobrescrito. Geralmente os hooks são colocados antes e depois de etapas cruciais de algoritmos, fornecendo às subclasses com pontos de extensão adicionais para um algoritmo.

Analogia com o mundo real

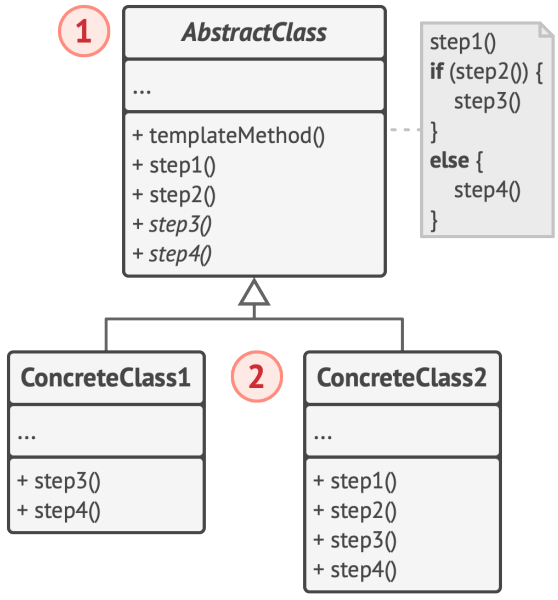


Um típico plano de arquitetura pode ser levemente alterado para melhor servir às necessidades do cliente.

A abordagem do Template Method pode ser usada na construção em massa de moradias. O plano arquitetônico para construir uma casa padrão pode conter diversos pontos de extensões que permitiriam um dono em potencial ajustar alguns detalhes na casa resultante.

Cada etapa de construção, tais como estabelecer as fundações, enquadramento, construindo paredes, instalando encanamento e fiação elétrica para água e eletricidade, etc. pode ser levemente mudado para se ter uma casa resultante um pouco diferente das outras.

Estrutura

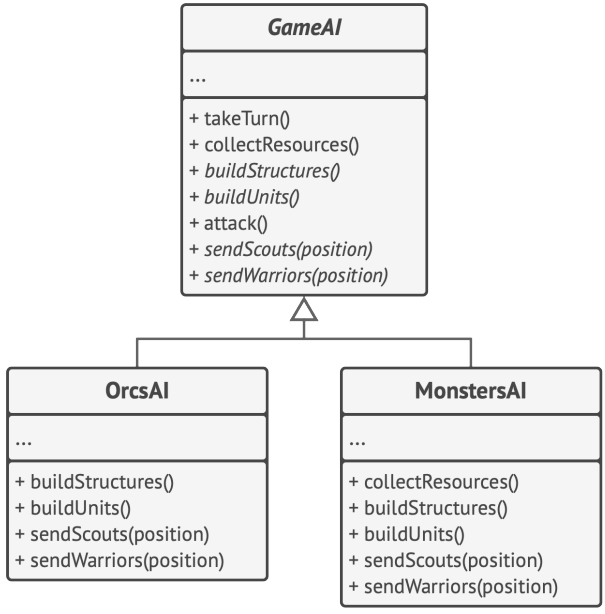


1. A **Classe Abstrata** declara métodos que agem como etapas de um algoritmo, bem como o próprio método padrão que chama esses métodos em uma ordem específica. Os passos podem ser declarados como `abstratos` ou ter alguma implementação padrão.

2. As **Classes Concretas** podem sobrescrever todas as etapas, mas não o próprio método padrão.

Pseudocódigo

Neste exemplo, o padrão **Template Method** fornece um "esqueleto" para várias ramificações de inteligência artificial de um jogo de estratégia simples.



Classes IA de um jogo simples.

Todas as raças do jogo tem quase o mesmo tipo de unidades e construções. Portanto você pode reutilizar a mesma estrutura de IA para várias raças, enquanto é capaz de sobrescrever alguns dos detalhes. Com essa abordagem, você pode sobrescrever a IA dos Orcs para torná-la mais agressiva, fazer os humanos mais orientados a defesa e fazer os monstros serem incapazes de construir qualquer coisa. Adicionando uma nova raça ao jogo irá necessitar a criação de uma nova subclasse

IA e a sobrescrição dos métodos padrão declarados na classe IA base.

```
// A classe abstrata define um método padrão que contém um
// esqueleto de algum algoritmo composto de chamadas, geralmente
// para operações abstratas primitivas. Subclasses concretas
// implementam essas operações, mas deixam o método padrão em si
// intacto.
class GameAI is
  // O método padrão define o esqueleto de um algoritmo.
  method turn() is
    collectResources()
    buildStructures()
    buildUnits()
    attack()

  // Algumas das etapas serão implementadas diretamente na
  // classe base.
  method collectResources() is
    foreach (s in this.builtStructures) do
      s.collect()

  // E algumas delas podem ser definidas como abstratas.
  abstract method buildStructures()
  abstract method buildUnits()

  // Uma classe pode ter vários métodos padrão.
  method attack() is
    enemy = closestEnemy()
    if (enemy == null)
      sendScouts(map.center)
```

```
      else
        sendWarriors(enemy.position)

    abstract method sendScouts(position)
    abstract method sendWarriors(position)

// Classes concretas têm que implementar todas as operações
// abstratas da classe base, mas não podem sobrescrever o método
// padrão em si.
class OrcsAI extends GameAI is
  method buildStructures() is
    if (there are some resources) then
      // Construir fazendas, depois quartéis, e então uma
      // fortaleza.

  method buildUnits() is
    if (there are plenty of resources) then
      if (there are no scouts)
        // Construir peão, adicionar ele ao grupo de
        // scouts (batedores).
      else
        // Construir um bruto, adicionar ele ao grupo
        // dos guerreiros.

  // ...

  method sendScouts(position) is
    if (scouts.length > 0) then
      // Enviar batedores para posição.


  method sendWarriors(position) is
```

```
    if (warriors.length > 5) then
      // Enviar guerreiros para posição.

// As subclasses também podem sobrescrever algumas operações com
// uma implementação padrão.
class MonstersAI extends GameAI is
  method collectResources() is
    // Monstros não coletam recursos.

  method buildStructures() is
    // Monstros não constroem estruturas.

  method buildUnits() is
    // Monstros não constroem unidades.
```

Aplicabilidade

Utilize o padrão Template Method quando você quer deixar os clientes estender apenas etapas particulares de um algoritmo, mas não todo o algoritmo e sua estrutura.

O Template Method permite que você transforme um algoritmo monolítico em uma série de etapas individuais que podem facilmente ser estendidas por subclasses enquanto ainda mantém intacta a estrutura definida em uma superclasse.

Utilize o padrão quando você tem várias classes que contém algoritmos quase idênticos com algumas diferenças menores.

Como resultado, você pode querer modificar todas as classes quando o algoritmo muda.

Quando você transforma tal algoritmo em um Template Method, você também pode erguer as etapas com implementações similares para dentro de uma superclasse, eliminando duplicação de código. Códigos que variam entre subclasses podem permanecer dentro das subclasses.

Como implementar

1. Analise o algoritmo alvo para ver se você quer quebrá-lo em etapas. Considere quais etapas são comuns a todas as subclasses e quais permanecerão únicas.

2. Crie a classe abstrata base e declare o método padrão e o conjunto de métodos abstratos representando as etapas do algoritmo. Contorne a estrutura do algoritmo no método padrão ao executar as etapas correspondentes. Considere tornar o método padrão como `final` para prevenir subclasses de sobrescrevê-lo.

3. Tudo bem se todas as etapas terminarem sendo abstratas. Contudo, alguns passos podem se beneficiar de ter uma implementação padrão. Subclasses não tem que implementar esses métodos.

4. Pense em adicionar ganchos entre as etapas cruciais do algoritmo.

5. Para cada variação do algoritmo, crie uma nova subclasse concreta. Ela *deve* implementar todas as etapas abstratas, mas *pode* também sobrescrever algumas das opcionais.

Prós e contras

✓ Você pode deixar clientes sobrescrever apenas certas partes de um algoritmo grande, tornando-os menos afetados por mudanças que acontece por outras partes do algoritmo.

✓ Você pode elevar o código duplicado para uma superclasse.

✕ Alguns clientes podem ser limitados ao fornecer o esqueleto de um algoritmo.

✕ Você pode violar o *princípio de substituição de Liskov* ao suprimir uma etapa padrão de implementação através da subclasse.

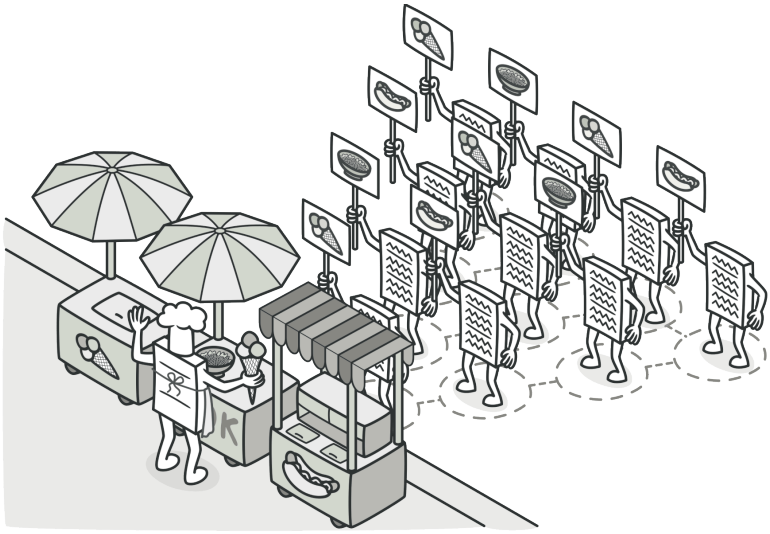
✕ Implementações do padrão Template Method tendem a ser mais difíceis de se manter quanto mais etapas eles tiverem.

Relações com outros padrões

- 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.

- O **Template Method** é baseado em herança: ele permite que você altere partes de um algoritmo ao estender essas partes em subclasses. O **Strategy** é baseado em composição: você pode alterar partes do comportamento de um objeto ao suprir

ele como diferentes estratégias que correspondem a aquele comportamento. O *Template Method* funciona a nível de classe, então é estático. O *Strategy* trabalha a nível de objeto, permitindo que você troque os comportamentos durante a execução.



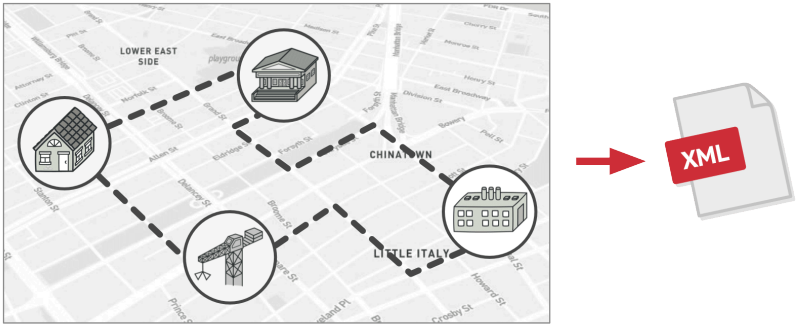
VISITOR

Também conhecido como: Visitante

O **Visitor** é um padrão de projeto comportamental que permite que você separe algoritmos dos objetos nos quais eles operam.

☹ Problema

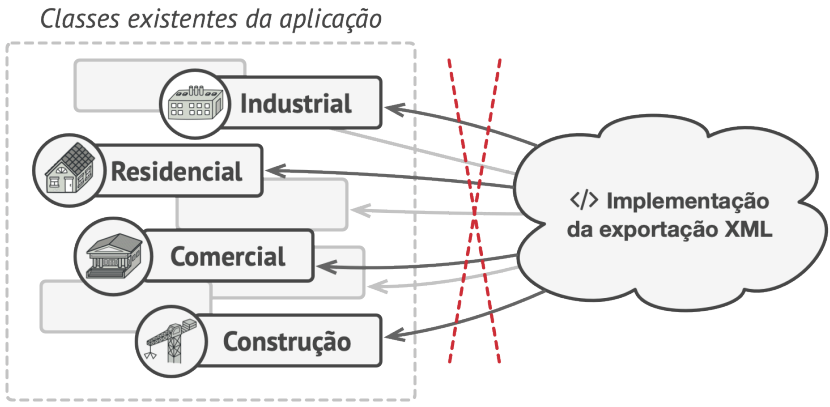
Imagine que sua equipe desenvolve uma aplicação que funciona com informações geográficas estruturadas em um grafo colossal. Cada vértice do gráfico pode representar uma entidade complexa como uma cidade, mas também coisas mais granulares como indústrias, lugares turísticos, etc. Os vértices estão conectados entre si se há uma estrada entre os objetos reais que eles representam. Por debaixo dos panos, cada tipo de vértice é representado por sua própria classe, enquanto que cada vértice específico é um objeto.



Exportando o grafo para XML.

Em algum momento você tem uma tarefa de implementar a exportação do grafo para o formato XML. No começo, o trabalho parecia muito simples. Você planejou adicionar um método de exportação para cada classe nó e então uma alavancagem recursiva para ir a cada nó do grafo, executando o método de exportação. A solução foi simples e elegante: graças ao polimorfismo, você não estava acoplando o código que chamava o método de exportação com as classes concretas dos nós.

Infelizmente, o arquiteto do sistema se recusou a permitir que você alterasse as classes nó existentes. Ele disse que o código já estava em produção e ele não queria arriscar quebrá-lo por causa de um possível bug devido às suas mudanças.



O método de exportação XML teve que ser adicionado a todas as classes nodo, o que trouxe o risco de quebrar toda a aplicação se quaisquer bugs passarem junto com a mudança.

Além disso, ele questionou se faria sentido ter um código de exportação XML dentro das classes nó. O trabalho primário dessas classes era trabalhar com dados geográficos. O comportamento de exportação XML ficaria estranho ali.

Houve outra razão para a recusa. Era bem provável que após essa funcionalidade ser implementada, alguém do departamento de marketing pediria que você fornecesse a habilidade para exportar para um formato diferente, ou pediria alguma outra coisa estranha. Isso forçaria você a mudar aquelas frágeis e preciosas classes novamente.

😊 Solução

O padrão Visitor sugere que você coloque o novo comportamento em uma classe separada chamada *visitante*, ao invés de tentar integrá-lo em classes já existentes. O objeto original que teve que fazer o comportamento é agora passado para um dos métodos da visitante como um argumento, desde que o método acesse todos os dados necessários contidos dentro do objeto.

Agora, e se o comportamento puder ser executado sobre objetos de classes diferentes? Por exemplo, em nosso caso com a exportação XML, a verdadeira implementação vai provavelmente ser um pouco diferente nas variadas classes nó. Portanto, a classe visitante deve definir não um, mas um conjunto de métodos, cada um capaz de receber argumentos de diferentes tipos, como este:

```
class ExportVisitor implements Visitor is
  method doForCity(City c) { ... }
  method doForIndustry(Industry f) { ... }
  method doForSightSeeing(SightSeeing ss) { ... }
  // ...
```

Mas como exatamente nós chamaríamos esses métodos, especialmente quando lidando com o grafo inteiro? Esses métodos têm diferentes assinaturas, então não podemos usar o polimorfismo. Para escolher um método visitante apropriado que

seja capaz de processar um dado objeto, precisaríamos checar a classe dele. Isso não parece um pesadelo?

```
foreach (Node node in graph)
  if (node instanceof City)
    exportVisitor.doForCity((City) node)
  if (node instanceof Industry)
    exportVisitor.doForIndustry((Industry) node)
  // ...
}
```

Você pode perguntar, por que não usamos o sobrecarregamento de método? Isso é quando você dá a todos os métodos o mesmo nome, mesmo se eles suportam diferentes conjuntos de parâmetros. Infelizmente, mesmo assumindo que nossa linguagem de programação suporta o sobrecarregamento (como Java e C#), isso não nos ajudaria. Já que a classe exata de um objeto nó é desconhecida de antemão, o mecanismo de sobrecarregamento não será capaz de determinar o método correto para executar. Ele irá usar como padrão o método que usa um objeto da classe `Nó` base.

Contudo, o padrão Visitor resolve esse problema. Ele usa uma técnica chamada **Double Dispatch**, que ajuda a executar o método apropriado de um objeto sem precisarmos de condicionais pesadas. Ao invés de deixar o cliente escolher uma versão adequada do método para chamar, que tal delegarmos essa escolha para os objetos que estamos passando para a visitante como argumentos? Já que os objetos sabem suas próprias

classes, eles serão capazes de escolher um método adequado na visitante de forma simples. Eles “aceitam” uma visitante e dizem a ela qual método visitante deve ser executado.

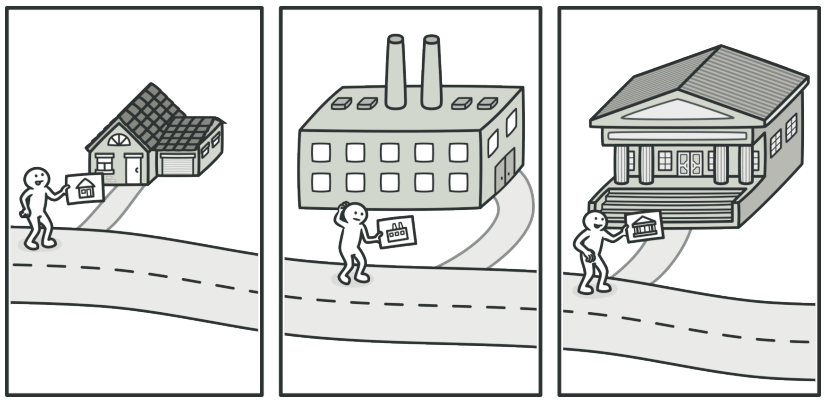
```
// Código cliente
foreach (Node node in graph)
  node.accept(exportVisitor)

// Cidade
class City is
  method accept(Visitor v) is
    v.doForCity(this)
  // ...

// Indústria
class Industry is
  method accept(Visitor v) is
    v.doForIndustry(this)
  // ...
```

Eu confesso. Tivemos que mudar as classes nó de qualquer jeito. Mas ao menos a mudança foi trivial e ela permite que nós adicionemos novos comportamentos sem alterar o código novamente.

Agora, se extrairmos uma interface comum para todas as visitantes, todos os nós existentes podem trabalhar com uma visitante que você introduzir na aplicação. Se você se deparar mais tarde adicionando um novo comportamento relacionado

aos nós, tudo que você precisa fazer é implementar uma nova classe visitante.

Analogia com o mundo real

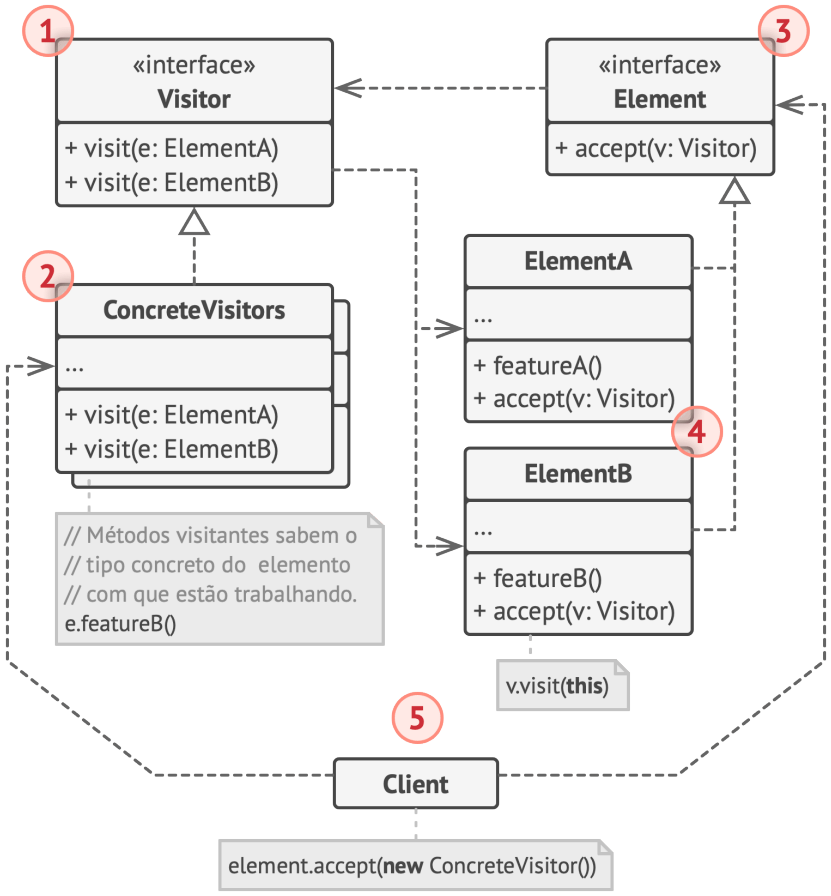


Um bom agente de seguros está sempre pronto para oferecer diferentes apólices para vários tipos de organizações.

Imagine um agente de seguros experiente que está ansioso para obter novos clientes. Ele pode visitar cada prédio de uma vizinhança, tentando vender apólices para todos que encontra. Dependendo do tipo de organização que ocupa o prédio, ele pode oferecer apólices de seguro especializadas:

- Se for um prédio residencial, ele vende seguros médicos.
- Se for um banco, ele vende seguro contra roubo.
- Se for uma cafeteria, ele vende seguro contra incêndios e enchentes.

Estrutura



1. A interface **Visitante** declara um conjunto de métodos visitantes que podem receber elementos concretos de uma estrutura de objetos como argumentos. Esses métodos podem ter os mesmos nomes se o programa é escrito em uma linguagem que suporta sobrecarregamento, mas o tipo dos parâmetros devem ser diferentes.

2. Cada **Visitante Concreto** implementa diversas versões do mesmo comportamento, feitos sob medida para diferentes elementos concretos de classes.

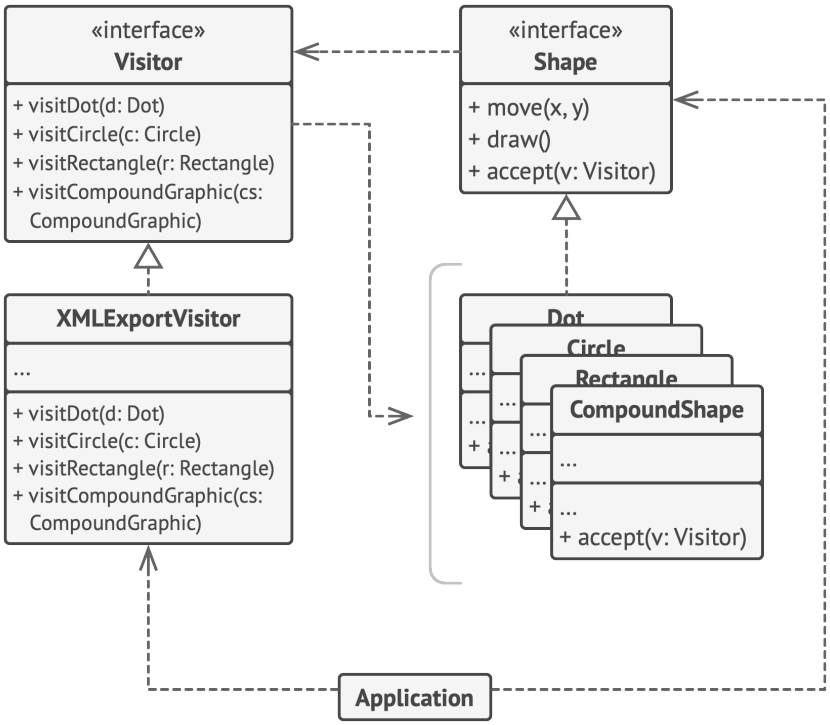
3. A interface **Elemento** declara um método para "aceitar" visitantes. Esse método deve ter um parâmetro declarado com o tipo da interface do visitante.

4. Cada **Elemento Concreto** deve implementar o método de aceitação. O propósito desse método é redirecionar a chamada para o método visitante apropriado que corresponde com a atual classe elemento. Esteja atento que mesmo se uma classe elemento base implemente esse método, todas as subclasses deve ainda sobrescrever esse método em suas próprias classes e chamar o método apropriado no objeto visitante.

5. O **Cliente** geralmente representa uma coleção de outros objetos complexos (por exemplo, uma árvore **Composite**). Geralmente, os clientes não estão cientes de todos as classes elemento concretas porque eles trabalham com objetos daquele coleção através de uma interface abstrata.

Pseudocódigo

Neste exemplo, o padrão **Visitor** adiciona suporte a exportação XML para a hierarquia de classe de formas geométricas.



Exportando vários tipos de objetos para o formato XML atráves do objeto visitante.

```
// O elemento interface declara um método `accept` que toma a
// interface do visitante base como um argumento.
interface Shape is
  method move(x, y)
  method draw()
  method accept(v: Visitor)

// Cada classe concreta de elemento deve implementar o método
// `accept` de tal maneira que ele chama o método visitante que
// corresponde com a classe do elemento.
class Dot implements Shape is
```

```
  // ...

  // Observe que nós estamos chamando `visitDot`, que coincide
  // com o nome da classe atual. Dessa forma nós permitimos
  // que o visitante saiba a classe do elemento com o qual ele
  // trabalha.
  method accept(v: Visitor) is
    v.visitDot(this)

class Circle implements Shape is
  // ...
  method accept(v: Visitor) is
    v.visitCircle(this)

class Rectangle implements Shape is
  // ...
  method accept(v: Visitor) is
    v.visitRectangle(this)

class CompoundShape implements Shape is
  // ...
  method accept(v: Visitor) is
    v.visitCompoundShape(this)


// A interface visitante declara um conjunto de métodos
// visitantes que correspondem com as classes elemento. A
// assinatura de um método visitante permite que o visitante
// identifique a classe exata do elemento com o qual ele está
// lidando.
interface Visitor is
  method visitDot(d: Dot)
```

```
    method visitCircle(c: Circle)
    method visitRectangle(r: Rectangle)
    method visitCompoundShape(cs: CompoundShape)

// Visitantes concretos implementam várias versões do mesmo
// algoritmo, que pode trabalhar com todas as classes elemento
// concretas.
//
// Você pode usufruir do maior benefício do padrão Visitor
// quando estiver usando ele com uma estrutura de objeto
// complexa, tal como uma árvore composite. Neste caso, pode ser
// útil armazenar algum estado intermediário do algoritmo
// enquanto executa os métodos visitantes sobre vários objetos
// da estrutura.
class XMLExportVisitor implements Visitor is
    method visitDot(d: Dot) is
        // Exporta a ID do dot (ponto) e suas coordenadas de
        // centro.

    method visitCircle(c: Circle) is
        // Exporta a ID do circle (círculo), coordenadas do
        // centro, e raio.


    method visitRectangle(r: Rectangle) is
        // Exporta a ID do retângulo, coordenadas do topo à
        // esquerda, largura e altura.

    method visitCompoundShape(cs: CompoundShape) is
        // Exporta a ID da forma bem como a lista de ID dos seus
        // filhos.

```

```
76
77  // O código cliente pode executar operações visitantes sobre
78  // quaisquer conjuntos de elementos sem saber suas classes
79  // concretas. A operação accept (aceitar) direciona a chamada
80  // para a operação apropriada no objeto visitante.
81  class Application is
82    field allShapes: array of Shapes
83
84    method export() is
85      exportVisitor = new XMLExportVisitor()
86
87      foreach (shape in allShapes) do
88        shape.accept(exportVisitor)
```

Se você está se perguntando por que precisamos do método `aceitar` neste exemplo, meu artigo **Visitor e Double Dispatch** responde essa dúvida em detalhes.

Aplicabilidade

Utilize o Visitor quando você precisa fazer uma operação em todos os elementos de uma estrutura de objetos complexa (por exemplo, uma árvore de objetos).

O padrão Visitor permite que você execute uma operação sobre um conjunto de objetos com diferentes classes ao ter o objeto visitante implementando diversas variantes da mesma operação, que correspondem a todas as classes alvo.

Utilize o Visitor para limpar a lógica de negócio de comportamentos auxiliares.

O padrão permite que você torne classes primárias de sua aplicação mais focadas em seu trabalho principal ao extrair todos os comportamentos em um conjunto de classes visitantes.

Utilize o padrão quando um comportamento faz sentido apenas dentro de algumas classes de uma uma hierarquia de classe, mas não em outras.

Você pode extrair esse comportamento para uma classe visitante separada e implementar somente aqueles métodos visitantes que aceitam objetos de classes relevantes, deixando o resto vazio.

Como implementar

1. Declare a interface da visitante com um conjunto de métodos "visitando", um para cada classe elemento concreta que existe no programa.

2. Declare a interface elemento. Se você está trabalhando com uma hierarquia de classes elemento existente, adicione o método de "aceitação" para a classe base da hierarquia. Esse método deve aceitar um objeto visitante como um argumento.

3. Implemente os métodos de aceitação em todas as classes elemento concretas. Esses métodos devem simplesmente re-

direcionar a chamada para um método visitante no objeto visitante que está vindo e que coincide com a classe do elemento atual.

4. As classes elemento devem trabalhar apenas com visitantes através da interface do visitante. Os visitantes, contudo, devem estar cientes de todas as classes elemento concretas referenciadas como tipos de parâmetros dos métodos visitantes.

5. Para cada comportamento que não possa ser implementado dentro da hierarquia do elemento, crie uma nova classe visitante concreta e implemente todos os métodos visitantes.

   Você pode encontrar uma situação onde o visitante irá necessitar acesso para alguns membros privados da classe elemento. Neste caso, você pode ou fazer desses campos ou métodos públicos, violando o encapsulamento do elemento, ou aninhando a classe visitante na classe elemento. Está última só é possível se você tiver sorte e estiver trabalhando com uma linguagem de programação que suporta classes aninhadas.

6. O cliente deve criar objetos visitantes e passá-los para os elementos através dos métodos de "aceitação".

Prós e contras

✓ *Princípio aberto/fechado*. Você pode introduzir um novo comportamento que pode funcionar com objetos de diferentes classes sem mudar essas classes.

✓ *Princípio de responsabilidade única*. Você pode mover múltiplas versões do mesmo comportamento para dentro da mesma classe.

✓ Um objeto visitante pode acumular algumas informações úteis enquanto trabalha com vários objetos. Isso pode ser interessante quando você quer percorrer algum objeto de estrutura complexa, tais como um objeto árvore, e aplicar o visitante para cada objeto da estrutura.

✕ Você precisa atualizar todos os visitantes a cada vez que a classe é adicionada ou removida da hierarquia de elementos.

✕ Visitantes podem não ter seu acesso permitido para campos e métodos privados dos elementos que eles deveriam estar trabalhando.

⇄ Relações com outros padrões

- Você pode tratar um **Visitor** como uma poderosa versão do padrão **Command**. Seus objetos podem executar operações sobre vários objetos de diferentes classes.

- Você pode usar o **Visitor** para executar uma operação sobre uma árvore **Composite** inteira.

- Você pode usar o **Visitor** junto com o **Iterator** para percorrer uma estrutura de dados complexas e executar alguma operação sobre seus elementos, mesmo se eles todos tenham classes diferentes.

Conteúdo Adicional

Confuso com o por que de não podermos simplesmente substituir o padrão Visitor com o sobrecarregamento de método? Leia meu artigo **Visitor e Double Dispatch** para aprender mais sobre os detalhes sórdidos a respeito.

Conclusão

Parabéns! Você chegou ao fim do livro!

Contudo, há muitos outros padrões no mundo. Espero que o livro tenha sido sua largada para aprender padrões e desenvolver habilidades super heróicas de padrões de projeto.

Aqui estão algumas ideias que irão ajudá-lo em decidir o que fazer agora.

- </> Não se esqueça que você também tem **acesso ao arquivo** com amostra de códigos que podem ser baixadas em diferentes linguagens de programação.

- Leia o livro de Joshua Kerievsky "**Refatoração para Padrões**".

- Não sabe nada sobre refatoração? **Eu tenho um curso para você**.

- Imprima estas **folhas de consultas sobre padrões** e coloque-as em algum lugar onde você é capaz de vê-las a todo momento.

- **Deixe sua opinião** sobre este livro. Ficarei muito feliz em saber sua opinião, mesmo sendo uma altamente crítica ☺