

PROF. MSC. RODRIGO AYRES

Clean Code e Padrões de Projeto

Clean Code

- ▶ Surgiu em 2008, quando Robert C.I Martins lançou o livro *Clean Code: A Handbook of Agile Software Craftsmanship*, que é referência no assunto.
- ▶ O autor, aliás, está entre as 17 pessoas que assinaram o Manifesto Ágil, que é a base para diversas metodologias ágeis, como o Scrum.
- ▶ Essa técnicas proporciona benefícios aos programadores, pois facilita o entendimento, a manutenção do código fonte e a realização dos testes necessários para verificar se a lógica está correta, bem como se o programa cumpre com a sua função.

Clean Code – O que é?

- ▶ Uma das atividades das pessoas que desenvolvem software é realizar manutenções em sistemas.
- ▶ Para isso, é preciso fazer a leitura e o entendimento de inúmeros códigos fontes de programas desenvolvidos por outras pessoas
- ▶ Essa atividade pode ser muito mais fácil de se executar caso o desenvolvimento seja feito com base em boas práticas de programação.

Clean Code – O que é?

- ▶ Técnica que estabelece um conjunto de boas práticas e orientações sobre **como desenvolver códigos que sejam facilmente entendidos, escritos e mantidos pelos devs.**
- ▶ O objetivo é garantir implementações com códigos de qualidade e que possam ser facilmente reutilizados.

Para que serve e por que usar?

- ▶ Manutenção de software que contém código mal escrito:
 - ▶ Uma pequena alteração pode impactar no funcionamento de todo o sistema, o que torna necessário realizar inúmeras outras alterações para conseguir implementar a mudança desejada.
 - ▶ Isso acontece porque códigos escritos dessa forma não foram pensados para facilitar a manutenção, o entendimento e o seu reaproveitamento.
- ▶ Em muitos cenários, quando analisamos a quantidade de esforços que terão de ser aplicados para realizar a alteração necessária ou até mesmo para outras alterações que poderão surgir no futuro, chegamos à conclusão que é mais fácil desenvolver toda a aplicação novamente.

As Sete Principais regras do Clean Code

- ▶ 1. Utilizar nomenclatura clara e intuitiva
 - ▶ **É importante utilizar nomes que tenham relação com a finalidade do código.** Uma função para exibir um alerta na tela, por exemplo, pode se chamar `exibirMensagem()`.
- ▶ **KISS**
 - ▶ Mantenha as coisas simples! Particularmente acho que é a base de uma boa solução. Normalmente tendemos a complicar as coisas que poderiam ser muito mais simples.
 - ▶ Então, Keep It Stupid Simple (Mantenha isto estupidamente simples - KISS)!

As Sete Principais regras do Clean Code

- ▶ 2. Seguir os padrões utilizados no código
 - ▶ Se você começou agora em um projeto ou acabaram de definir suas convenções, siga-as! Se utilizam por exemplo constantes em maiúsculo, enumeradores com **E** como prefixo, não importa! Siga sempre os padrões do projeto
- ▶ 3. Manter os dados de configuração separados do código fonte
 - ▶ Os dados de configuração, como strings de conexão com o banco de dados, devem ser adicionados em um arquivo separado do código fonte.

As Sete Principais regras do Clean Code

- ▶ Algo que toda aplicação tem são suas configurações, como as conhecidas **ConnectionStrings**.
- ▶ Tente sempre deixar estas configurações ou o *parse* delas em um nível mais alto possível.
- ▶ Evite sobrescrever configurações em métodos dentro de **Controllers** ou algo do tipo. Se possível, mantenha esta passagem no método principal, no início da aplicação e não mexa mais nisto!

As Sete Principais regras do Clean Code

▶ Regra do escoteiro

- ▶ "Deixe sempre o acampamento mais limpo do que você encontrou!" O mesmo vale para nosso código. Devolva (Check in) sempre o código melhor do que você o obteve. Se todo desenvolvedor no time tiver esta visão, e devolver um pedacinho de código melhor do que estava antes, em pouco tempo teremos uma grande mudança.

As Sete Principais regras do Clean Code

- ▶ 4. Evitar repetições excessivas
 - ▶ Portanto, evite estruturas de repetições aninhadas, como diversos "ifs" seguidos, pois eles aumentam a complexidade do código.

```
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] == $_POST['user_password_old']) {
                    if (strlen($_POST['user_password_new']) < 6) {
                        if (strlen($_POST['user_name']) < 6) {
                            if (preg_match('/^[a-z\d]{2,}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 32) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_session($_POST['user_email']);
                                                $_SESSION['msg'] = 'User registered successfully';
                                                header('Location: /index.php');
                                                exit();
                                            } else $msg = 'Invalid email';
                                        } else $msg = 'Email must be at least 32 characters';
                                    } else $msg = 'Username must be at least 6 characters';
                                } else $msg = 'Username must be unique';
                            } else $msg = 'Username must be alphanumeric';
                        } else $msg = 'Password must be at least 6 characters';
                    } else $msg = 'Passwords do not match';
                } else $msg = 'Empty Password';
            } else $msg = 'Empty Username';
        }
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```



As Sete Principais regras do Clean Code

- ▶ 5. Ter cuidado com o uso de comentários no código
 - ▶ Além disso, há riscos em utilizar comentários em excesso, pois dificilmente eles serão alterados quando houver uma manutenção do código.
 - ▶ Isso significa que um comentário pode conter informações sobre determinado trecho do algoritmo que não corresponde mais ao que a função executa.
 - ▶ Por isso, **eles devem ser utilizados com moderação e alterados sempre que houver necessidade.**

As Sete Principais regras do Clean Code

- ▶ 6. Realizar o tratamento de erros
 - ▶ Sempre procure a causa raiz do problema, nunca resolva as coisas superficialmente. No dia-a-dia, na correria, tendemos a corrigir os problemas superficialmente e não adentrar neles, o que muitas vezes causa o re-trabalho!
 - ▶ Tente sempre procurar a causa raiz e resolver assim o problema de uma vez por todas!

As Sete Principais regras do Clean Code

- ▶ Executar testes limpos
 - ▶ Testes unitários, eles devem ser realizados com pouco código, ou seja, **nada de testar toda a aplicação de uma vez.**
 - ▶ É preciso depurar pequenos blocos, que devem ser independentes.
 - ▶ Dessa forma, haverá a evolução dos testes sem que a parte já testada apresente um novo erro em função de novos trechos testados. A

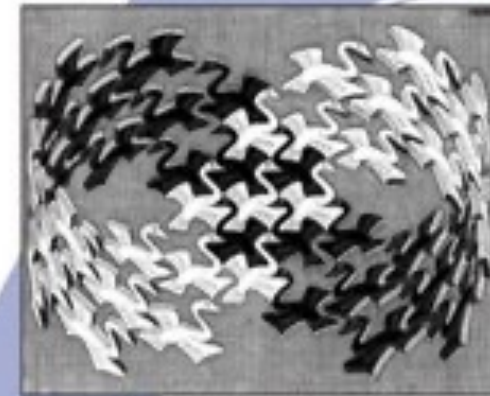
Design Patterns

- ▶ Ganharam popularidade no ano de 1995, quando o livro *Design Patterns: Elements of Reusable Object-Oriented Software* foi publicado.
- ▶ Os autores ficaram conhecidos como “Gangue dos Quatro” e, após esse incentivo, muitas outras obras foram lançadas a respeito.
- ▶ “Gangue dos Quatro”:
 - ▶ Erich Gamma
 - ▶ Richard Helm
 - ▶ Ralph Johnson
 - ▶ John Vlissides

Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch

Design Patterns - A Inspiração

- ▶ **Padrões de projeto** (design patterns) são soluções típicas para problemas comuns em projeto de software.
- ▶ Cada padrão é como uma planta de construção que você pode customizar para resolver um problema de projeto particular em seu código.
- ▶ Eles descrevem a solução para um problema que ocorre repetidamente em nosso ambiente
- ▶ Caracterizam a parte central da solução para aquele problema de uma forma que você possa usar esta solução muitas vezes.

Qual a relação entre os padrões?

- ▶ Para criar um sistema, podem ser necessários vários padrões.
- ▶ Diferentes designers podem usar diferentes padrões para resolver o mesmo problema.
- ▶ Geralmente:
 - ▶ Alguns padrões se 'encaixam' naturalmente
 - ▶ Um padrão pode levar a outro
 - ▶ Alguns padrões são similares e alternativos
 - ▶ Padrões podem ser descobertos e documentados
 - ▶ Padrões não são métodos ou frameworks
 - ▶ Padrões dão uma dica para resolver um problema de forma efetiva

Qual a relação entre os padrões?

- ▶ Passamos a ter um vocabulário comum para conversar sobre projetos de software.
- ▶ Soluções que não tinham nome passam a ter nome.
- ▶ Ao invés de discutirmos um sistema em termos de pilhas, filas, árvores e listas ligadas, passamos a falar de coisas de muito mais alto nível como Fábricas, Fachadas, Observador, Estratégia, etc.

Padrões de Projeto

- ▶ Desde 1995, o desenvolvimento de software passou a ter o seu primeiro catálogo de soluções para projeto de software: o livro GoF
- ▶ Categorias de Padrões de acordo com o livro:
 - ▶ Padrões de Criação
 - ▶ Padrões Estruturais
 - ▶ Padrões Comportamentais

Quais são?

▶ Criacionais:

- ▶ Abstract Factory
- ▶ Builder
- ▶ Factory Method
- ▶ Prototype
- ▶ Singleton

▶ Estruturais

- ▶ Adapter
- ▶ Bridge
- ▶ Composite
- ▶ Decorator
- ▶ Façade
- ▶ Flyweight
- ▶ Proxy

▶ Comportamentais

- ▶ Chain of Responsibility
- ▶ Command
- ▶ Interpreter
- ▶ Iterator
- ▶ Mediator
- ▶ Memento
- ▶ Observer
- ▶ State
- ▶ Strategy
- ▶ Template Method
- ▶ Visitor

Complexidade

▶ Fácil

- ▶ Facade
- ▶ Singleton
- ▶ Mediator
- ▶ Iterator
- ▶ Strategy
- ▶ Command
- ▶ Builder
- ▶ State
- ▶ Template Method
- ▶ Factory Method
- ▶ Memento
- ▶ Prototype

▶ Intermédiaria

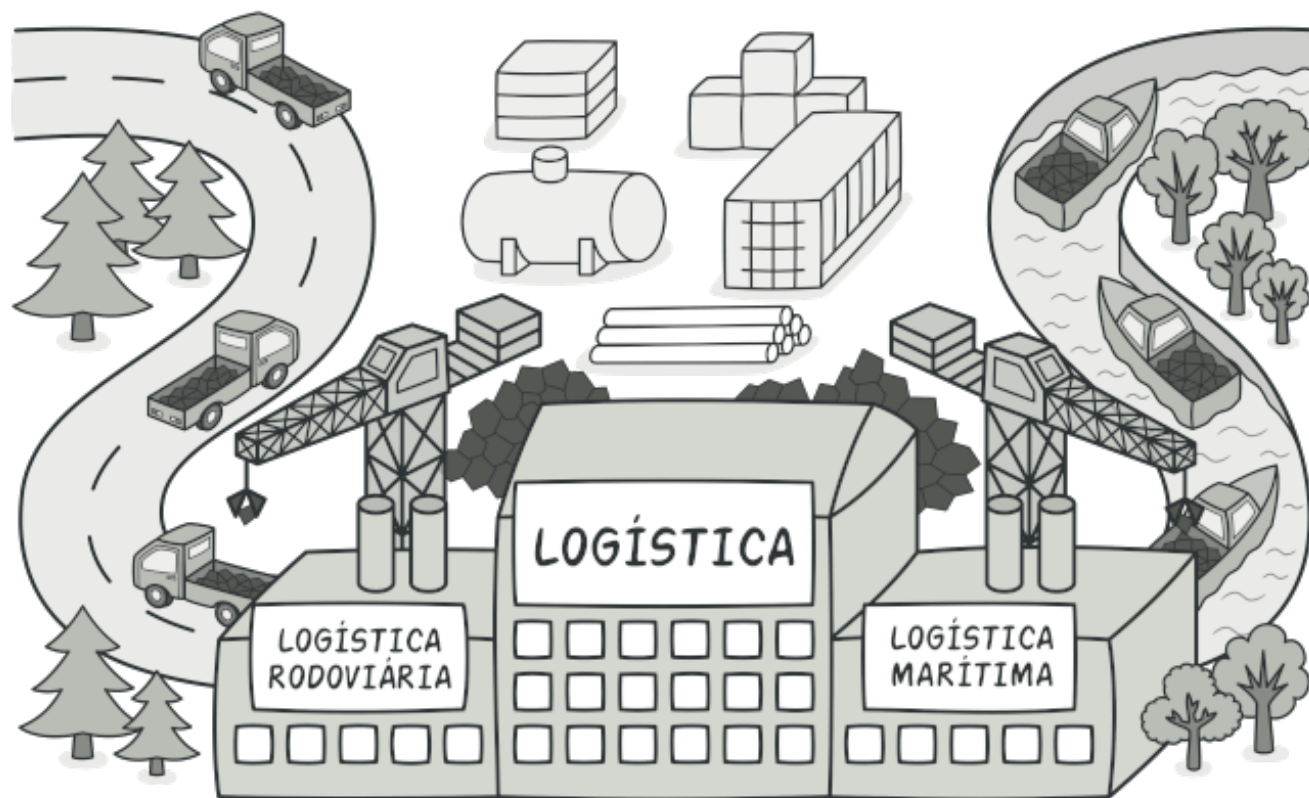
- ▶ Proxy
- ▶ Decorator
- ▶ Adapter
- ▶ Bridge
- ▶ Observer

▶ **Avançados:**

- ▶ Composite
- ▶ Interpreter
- ▶ Chain Of Responsibility
- ▶ Abstract Factory
- ▶ Visitor

Factory Method

- 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.



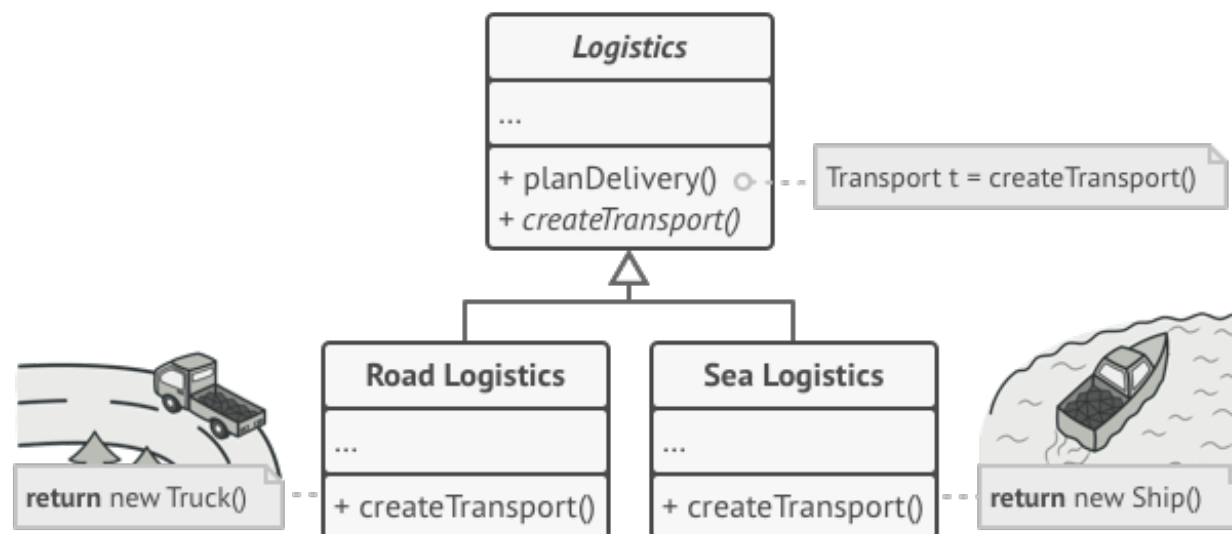
Adicionar uma nova classe ao programa não é tão simples se o restante do código já estiver acoplado às classes existentes.

Problema

- ▶ 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

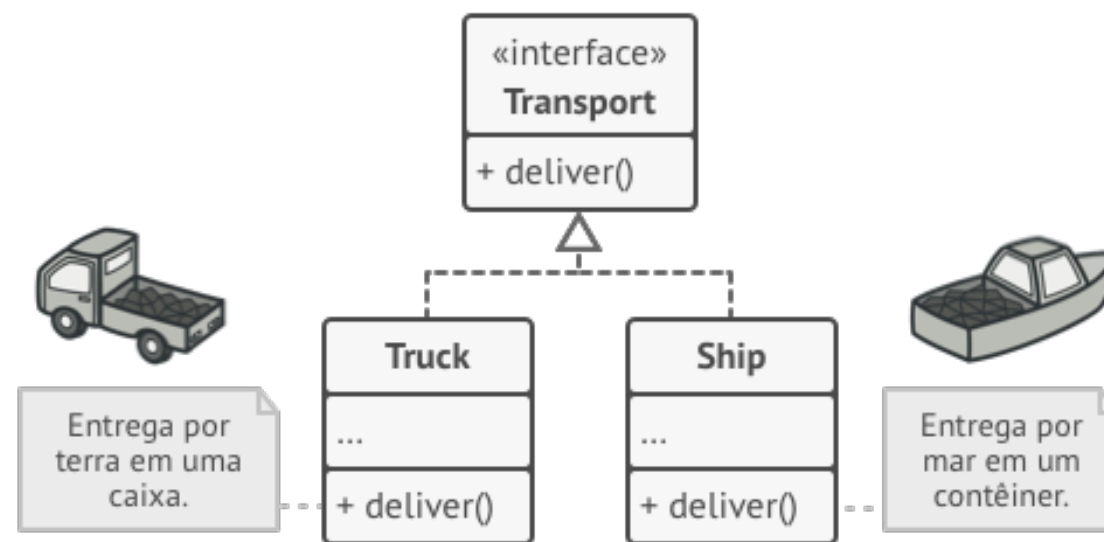
- ▶ O padrão Factory Method sugere que você substitua chamadas diretas de construção de objetos (usando o operador new) por chamadas para um método fábrica especial.
- ▶ Os objetos ainda são criados através do operador new, mas esse está sendo chamado de dentro do método fábrica."



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

Solução

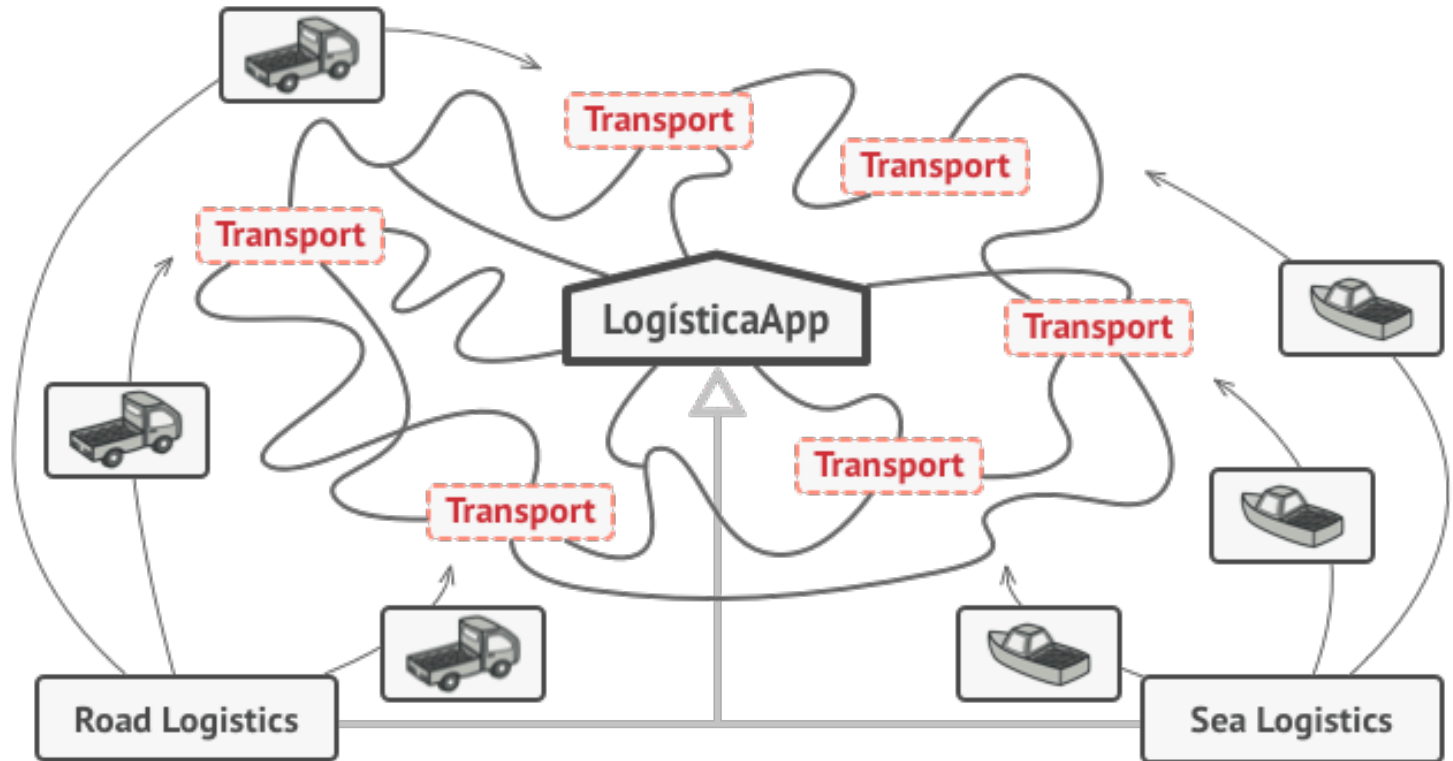
- ▶ À primeira vista, essa mudança pode parecer sem sentido: apenas mudamos a chamada do construtor de uma parte do programa para outra. No entanto, considere o seguinte: agora você pode sobrescrever o método fábrica em uma subclasse e alterar a classe de produtos que estão sendo criados pelo método.
- ▶ 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.

Solução

- 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.

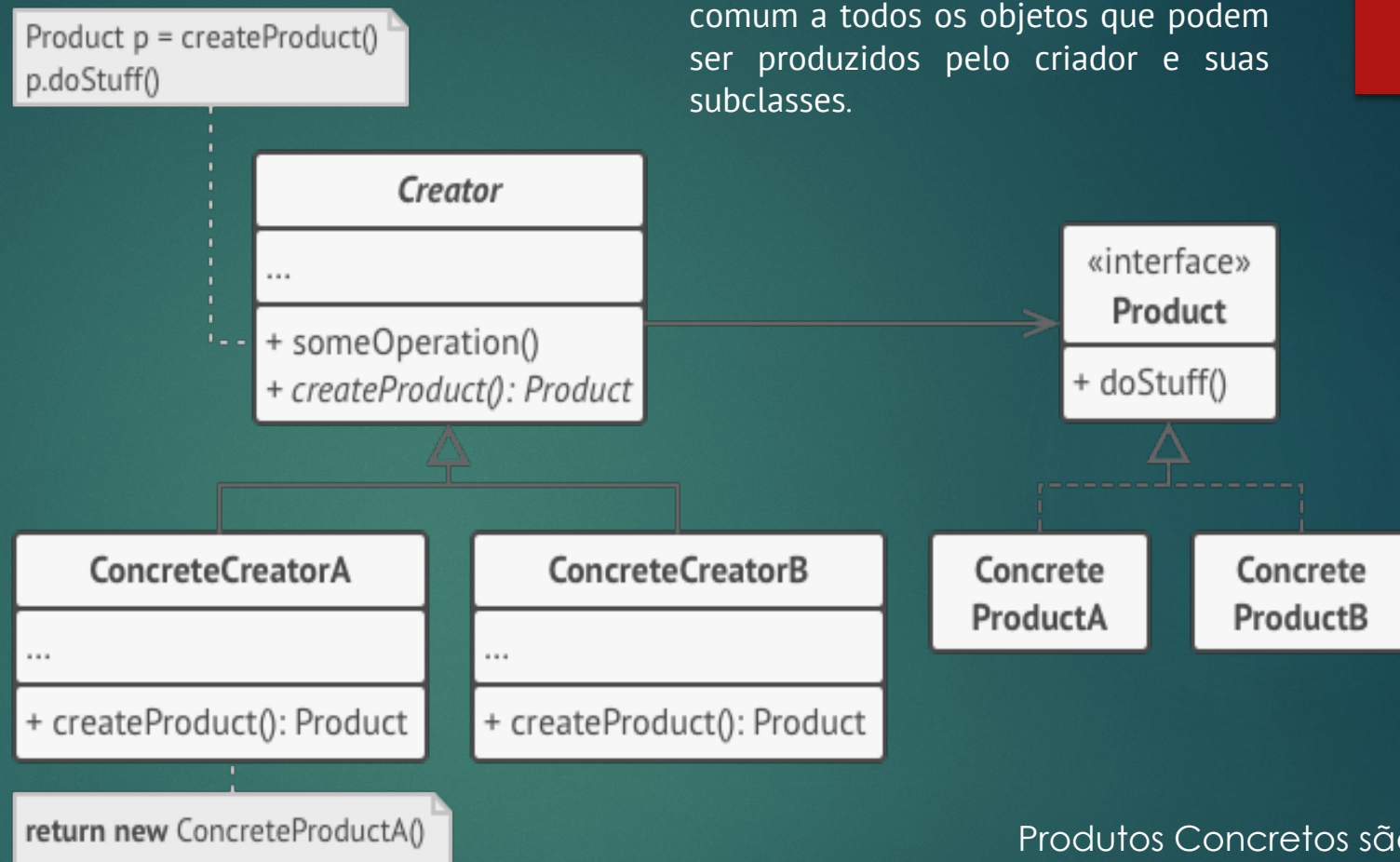
Solução

- ▶ 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.

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.



O **Produto** declara a interface, que é comum a todos os objetos que podem ser produzidos pelo criador e suas subclasses.

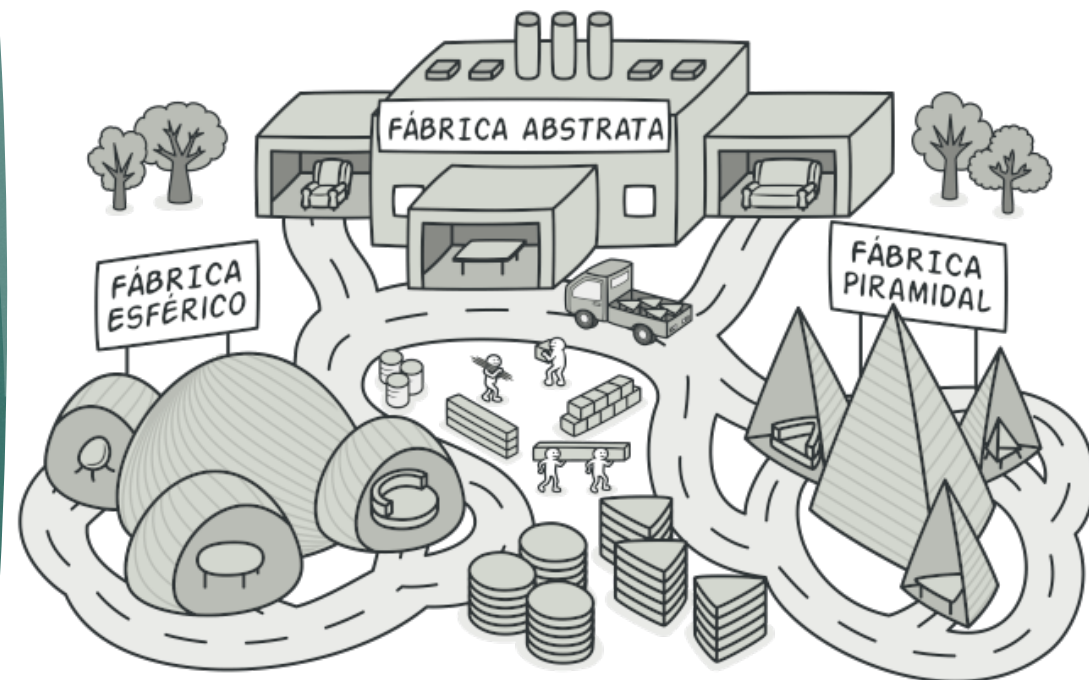
Criadores Concretos sobrescrevem o método fábrica base para retornar um tipo diferente de produto.

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.

Produtos Concretos são implementações diferentes da interface do produto

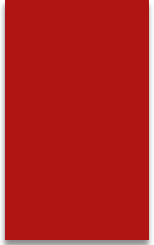
Abstract Factory










- 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.



O Problema

- ▶ Imagine que você está criando um simulador de loja de móveis. Seu código consiste de classes que representam:
- ▶ Uma família de produtos relacionados, como: Cadeira + Sofá + MesaDeCentro.
- ▶ Várias variantes dessa família. Por exemplo, produtos Cadeira + Sofá + MesaDeCentro estão disponíveis nessas variantes: Moderno, Vitoriano, ArtDeco.

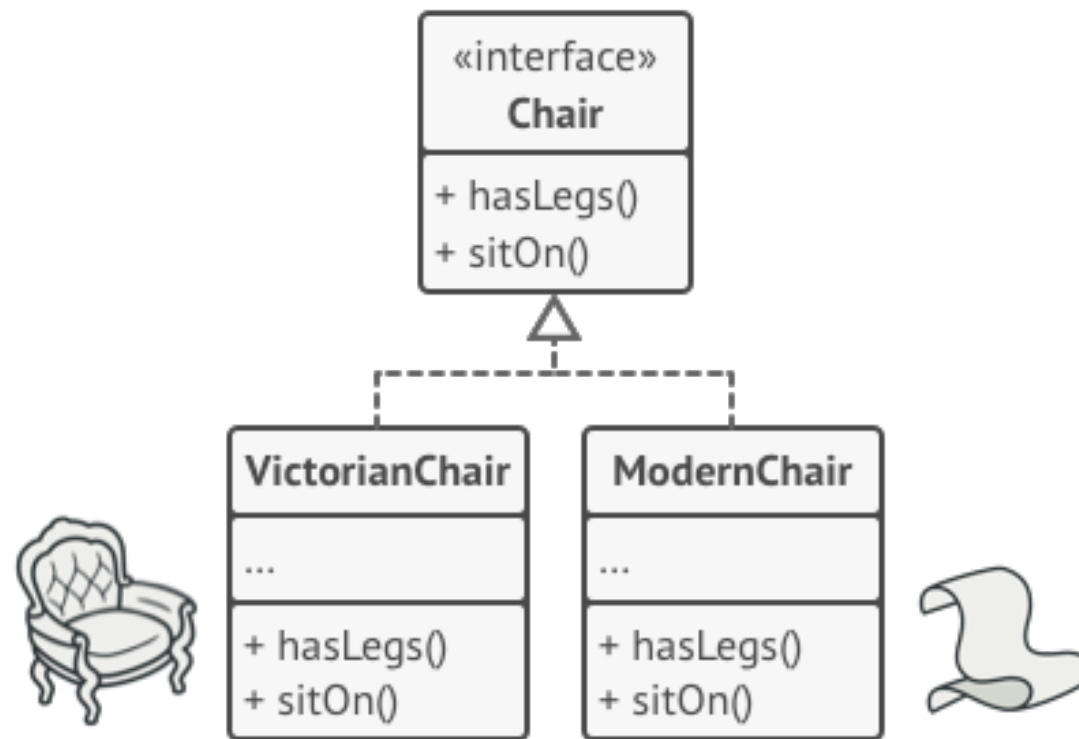


	Cadeira	Sofá	Mesa de Centro
Art Deco			
Vitoriano			
Moderno			

Famílias de produtos e suas variantes.

Solução

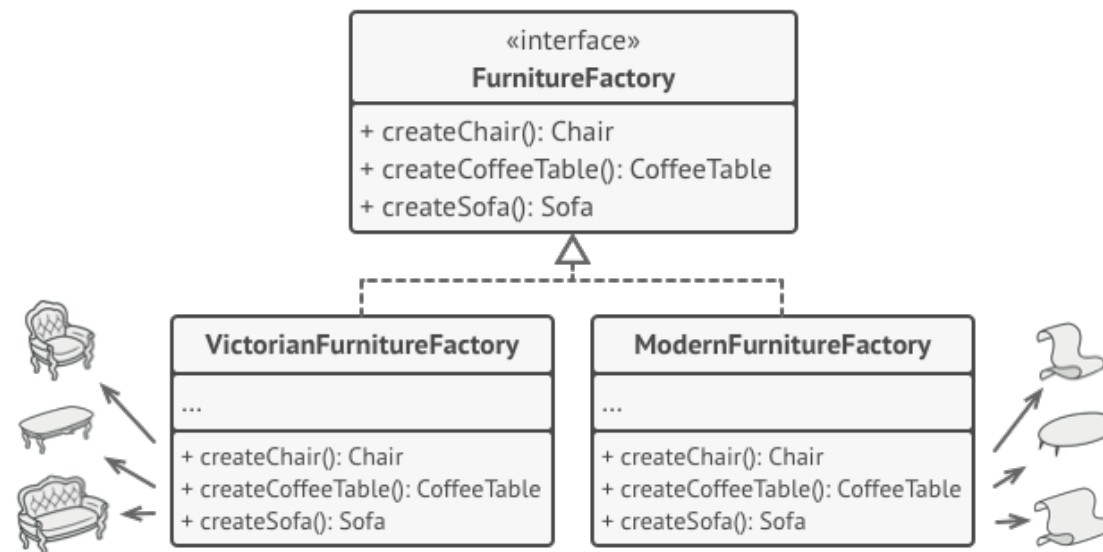
- ▶ A primeira coisa que o padrão Abstract Factory sugere é declarar explicitamente interfaces para cada produto distinto da família de produtos (ex: cadeira, sofá ou mesa de centro).
- ▶ Então você pode fazer todas as variantes dos produtos seguirem essas interfaces.
- ▶ Por exemplo, todas as variantes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

Solução

- ▶ 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.



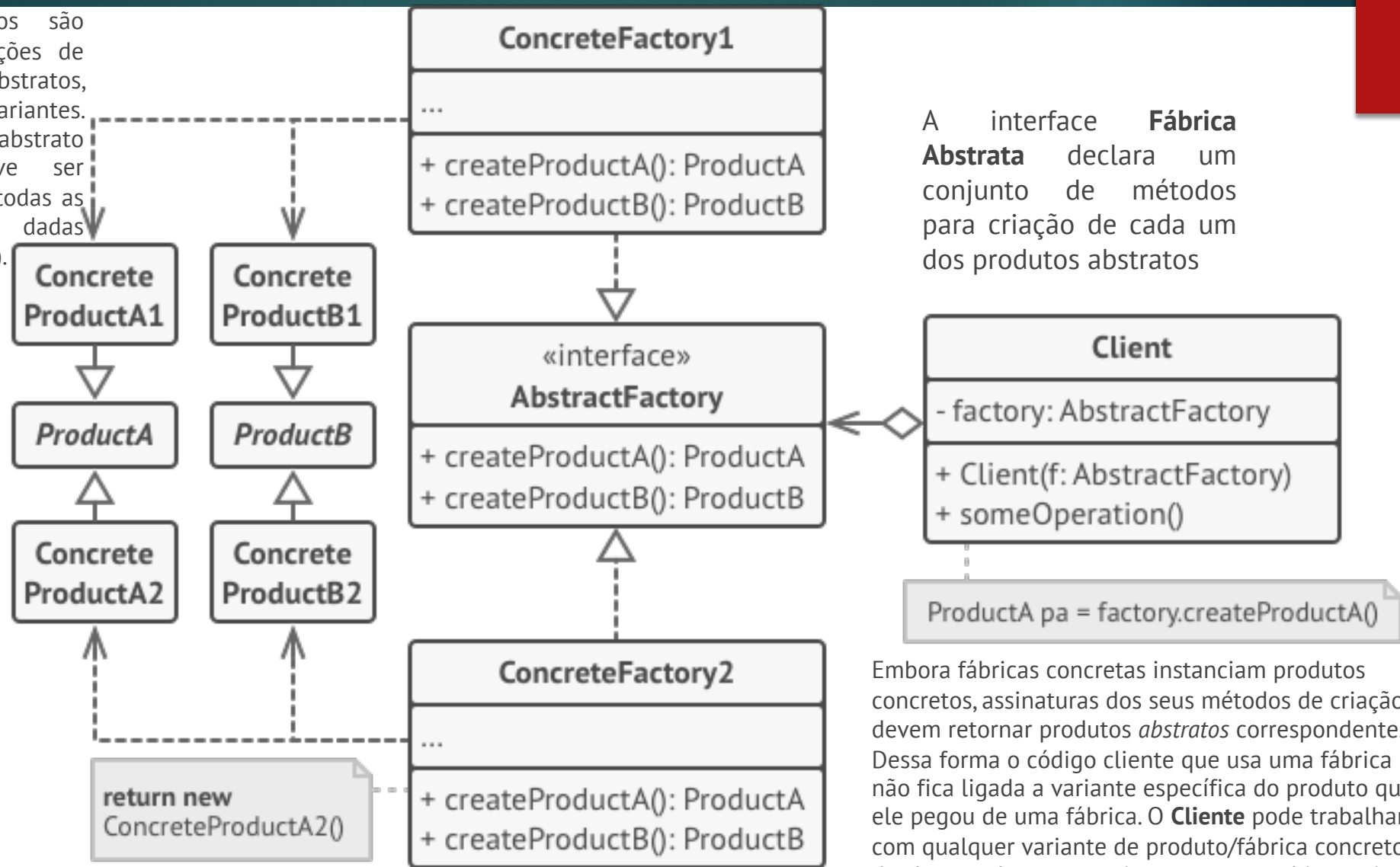
Cada fábrica concreta corresponde a uma variante de produto específica.

Solução

- ▶ 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.

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).

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



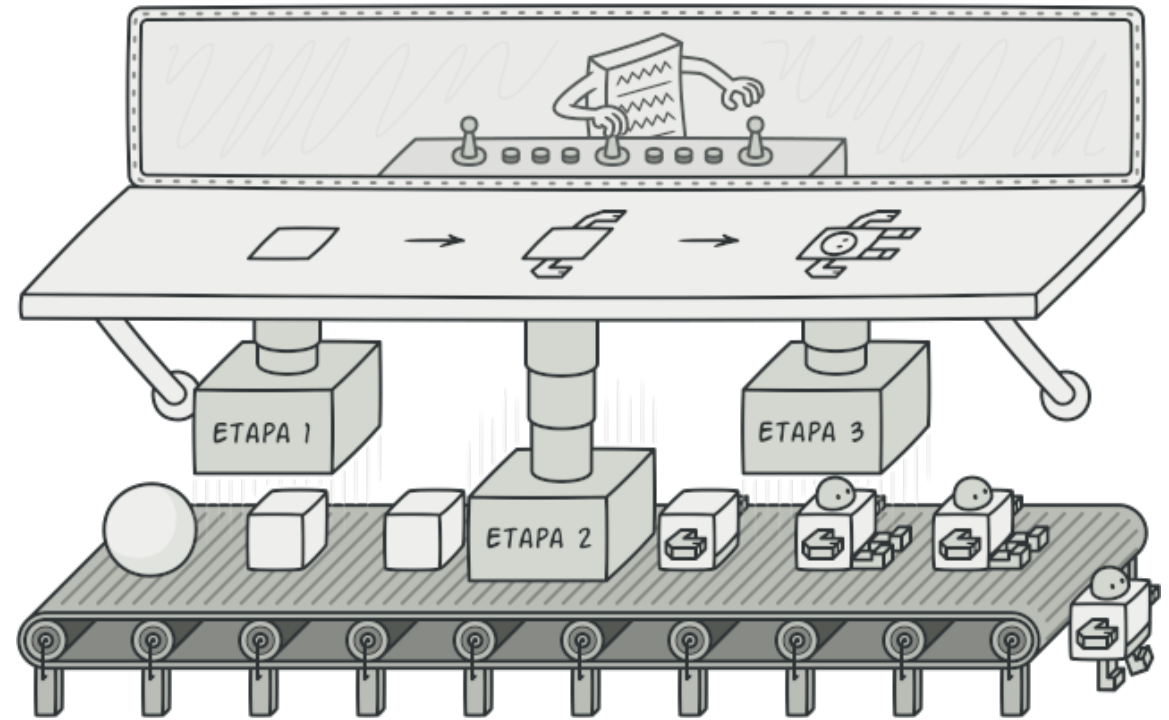
A interface **Fábrica Abstrata** declara um conjunto de métodos para criação de cada um dos produtos abstratos

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.

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.

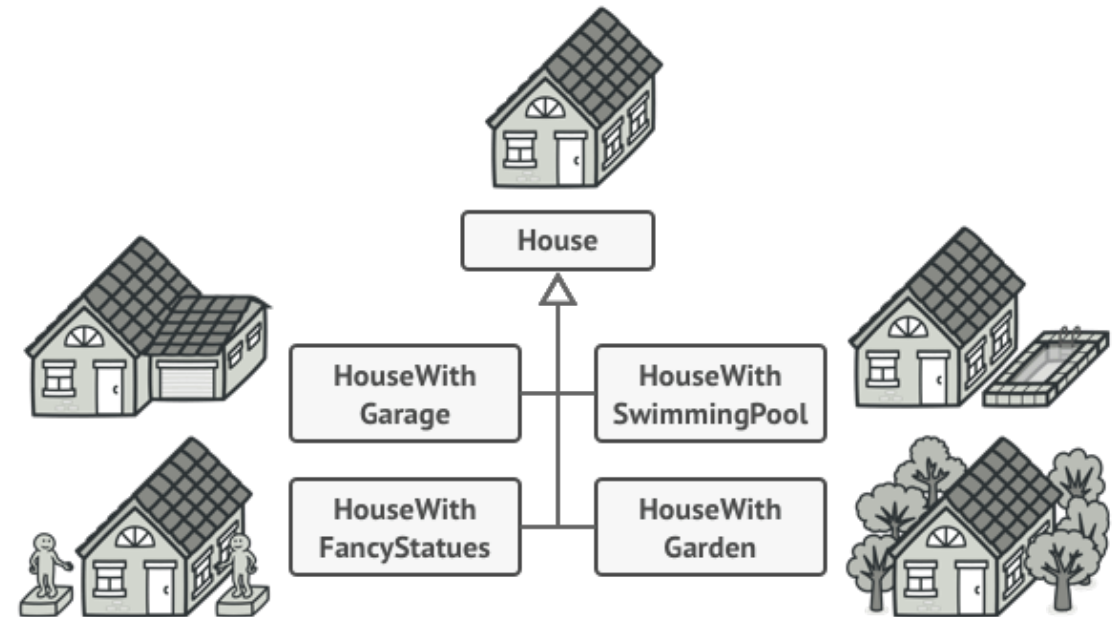
Builder

- 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.



O 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.



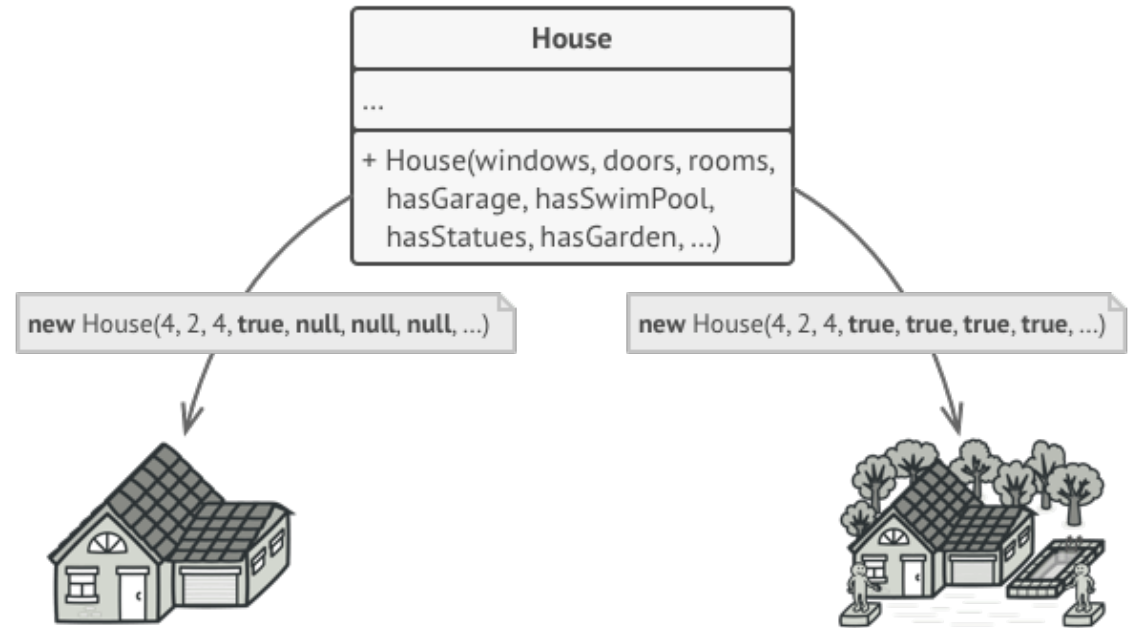
Você pode tornar o programa muito complexo ao criar subclasses para cada possível configuração de um objeto.

○ Problema

- ▶ 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 elimina a necessidade de subclasses, ela cria outro problema.

○ Problema

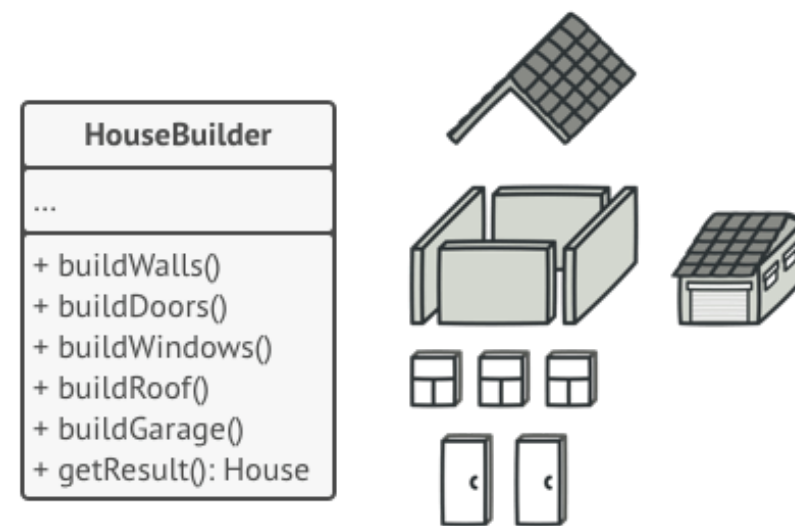
- ▶ Na maioria dos casos a maioria dos parâmetros não será usada, tornando as chamadas do construtor em algo feio de se ver. Por exemplo, apenas algumas casas têm piscinas, então os parâmetros relacionados a piscinas serão inúteis nove em cada dez vezes.



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



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.

Solução

- ▶ 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 construtor. 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.

Solução

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



Construtores diferentes executam a mesma tarefa de várias maneiras.

Solução

- ▶ Por exemplo, imagine um construtor que constrói tudo de madeira e vidro, um segundo construtor 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 construtor, 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 construtores usando uma interface comum.

Diretor

- ▶ Você pode ir além e extrair uma série de chamadas para as etapas do construtor 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 construtor provê a implementação dessas etapas.



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

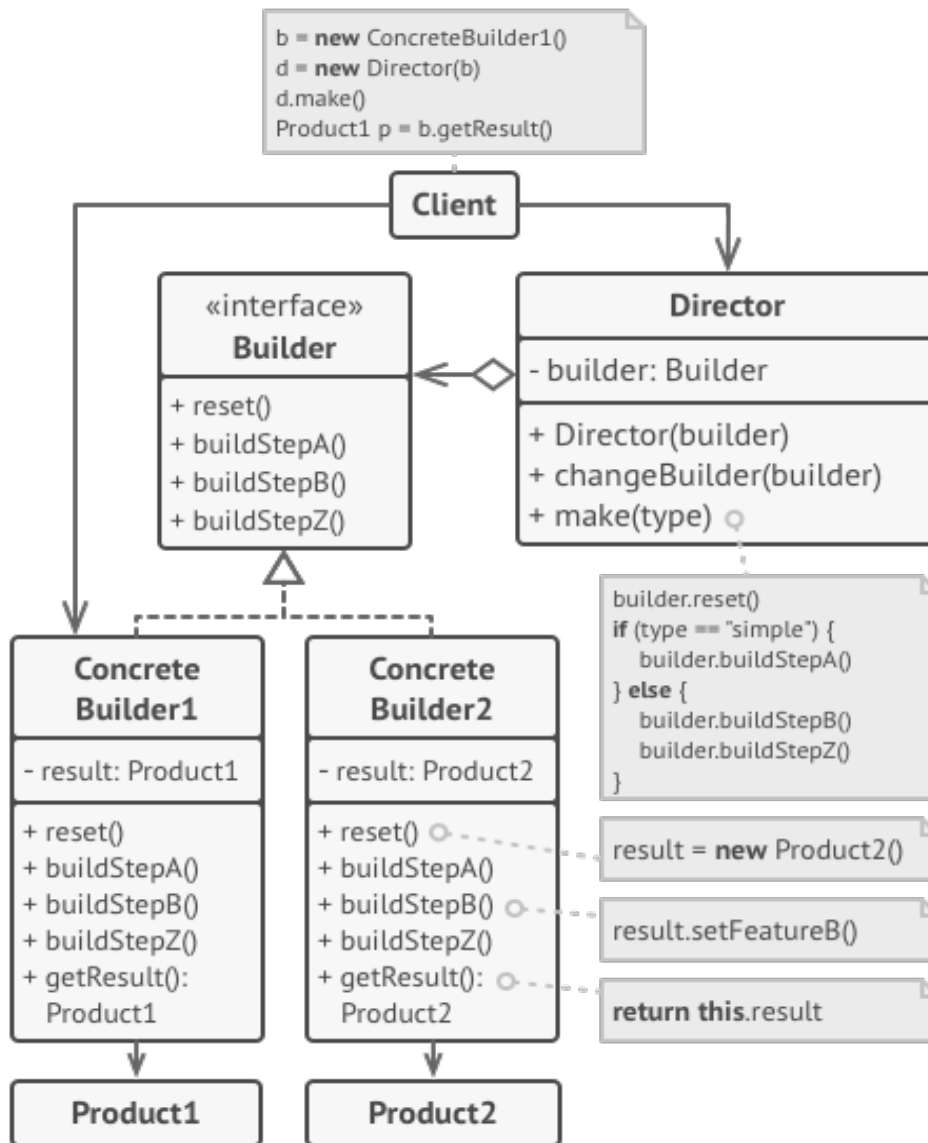
Diretor

- ▶ 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 construtor com um diretor, inicializar a construção com o diretor, e então obter o resultado do construtor.

A interface **Construtor** declara etapas de construção do produto que são comuns a todos os tipos de construtoras.

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

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

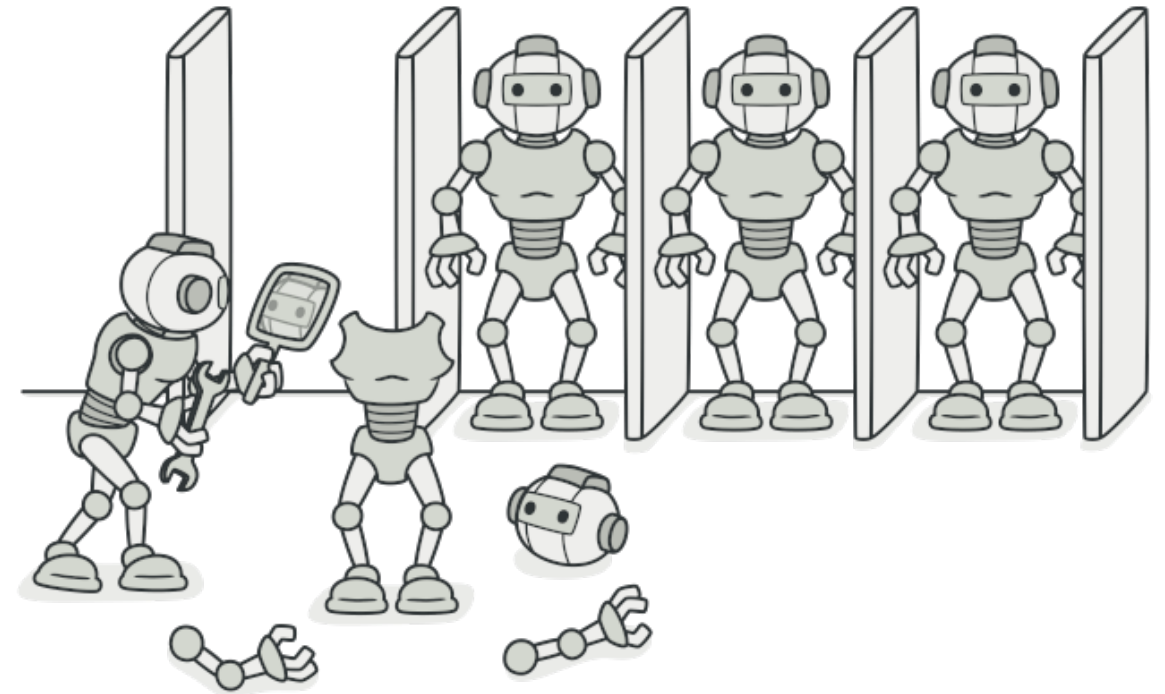


A classe **Diretor** define a ordem na qual as etapas de construção são chamadas, então você pode criar e reutilizar configurações específicas de produtos.

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

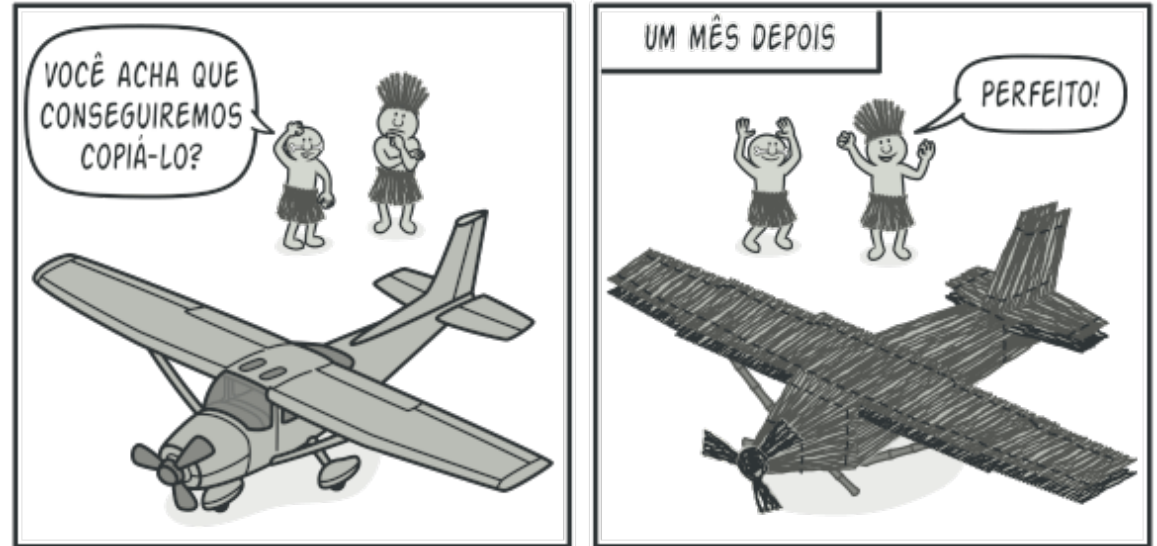
Prototype

- ▶ 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.

Problema

- ▶ 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 parâmetro em um método aceita quaisquer objetos que seguem uma interface.

Solução

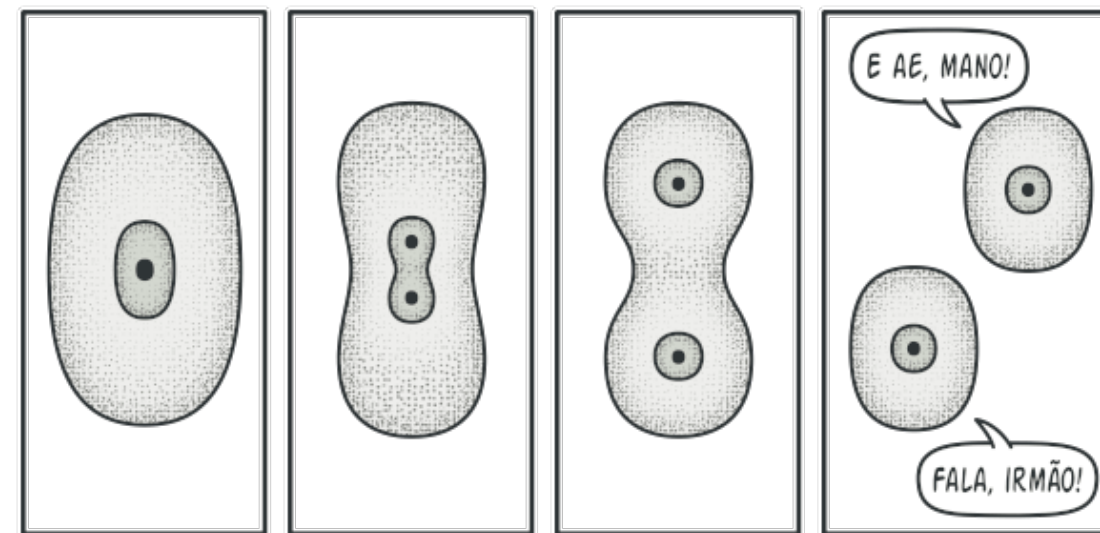
- ▶ O padrão Prototype delega o processo de clonagem para o próprio objeto que está sendo clonado. O padrão declara um interface comum para todos os objetos que suportam clonagem. Essa interface permite que você clone um objeto sem acoplar seu código à classe daquele objeto. Geralmente, tal interface 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 às subclasses.



Pré construir protótipos pode ser uma alternativa às subclasses.

Solução

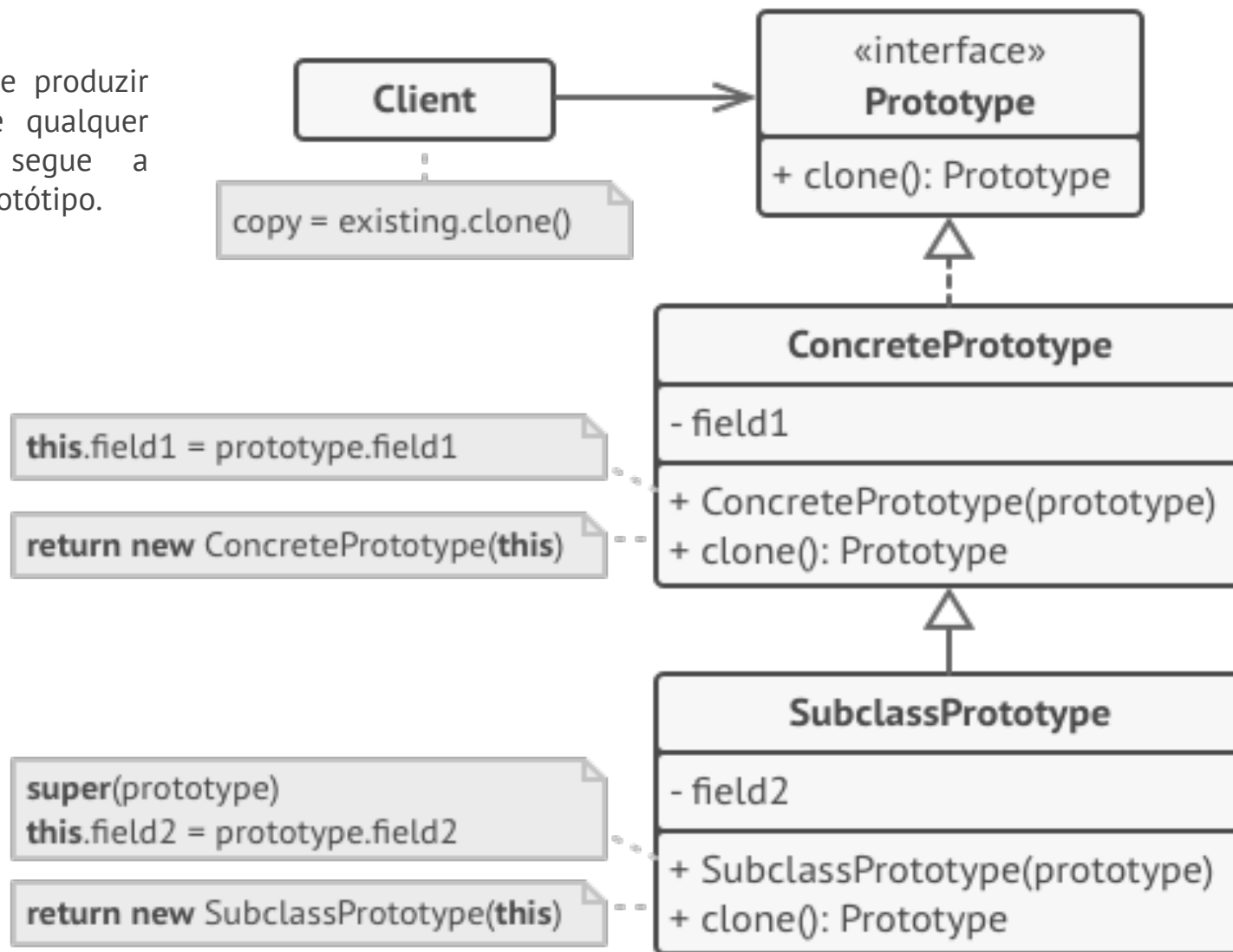
- ▶ Funciona assim: você cria um conjunto de objetos, configurados de diversas formas. Quando você precisa um objeto parecido com o que você configurou, você apenas clona um protótipo ao invés de construir um novo objeto a partir do nada.
- ▶ Na vida real, os protótipos são usados para fazer diversos 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.



A divisão de uma célula.

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.

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

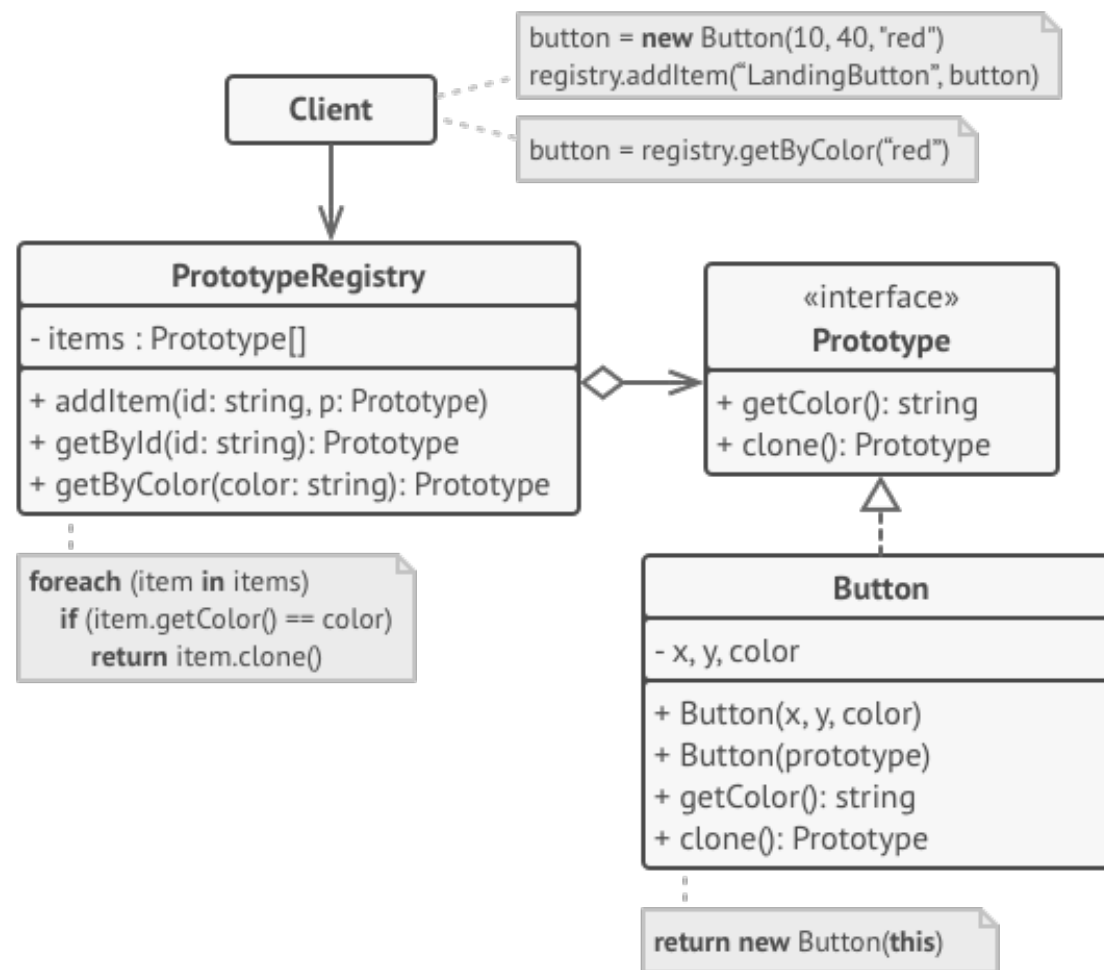


A interface **Protótipo** declara os métodos de clonagem. Na maioria dos casos é apenas um método clonar

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.

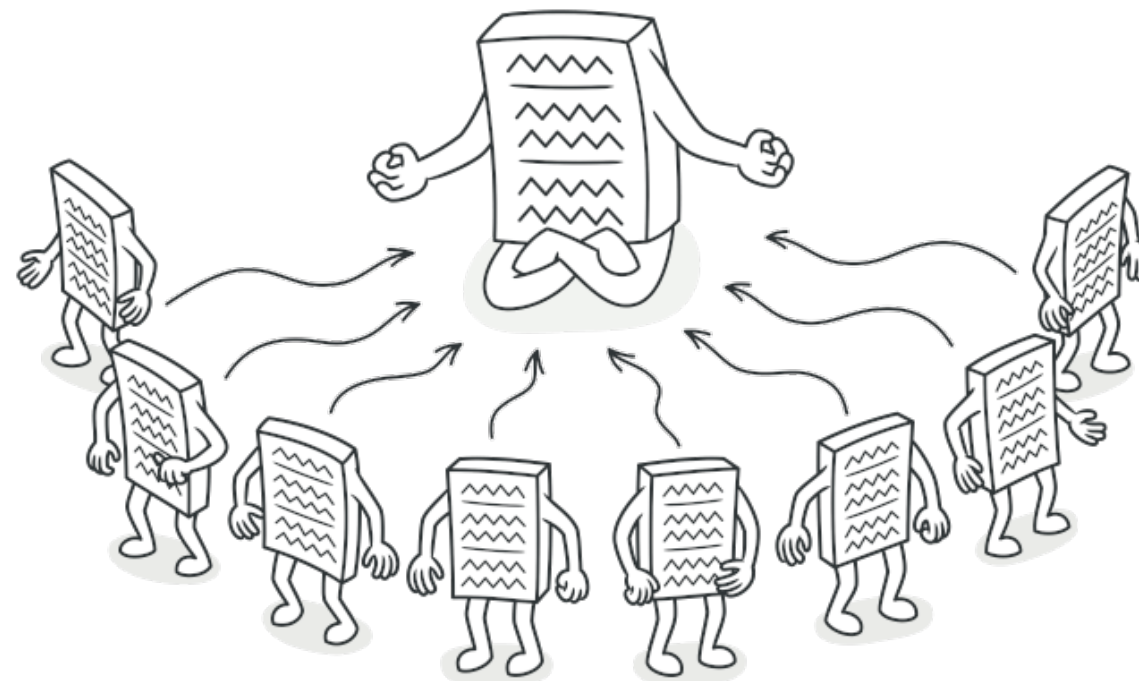
Registro de protótipo

- ▶ 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.



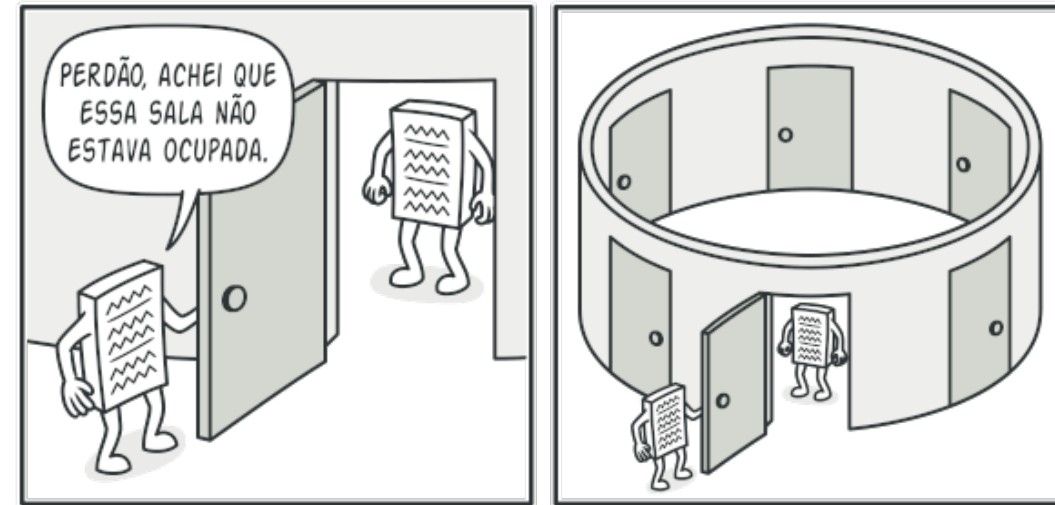
Singleton

- ▶ 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.



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

Problema

- ▶ 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.

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.

