O que são Padrões de Projeto?

DESIGN PATTERNS

PROF. ME JEFFERSON PASSERINI

O que é um padrão de projeto?

Design Patterns

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.

Design Patterns

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 desenvolvido.**

- Uma analogia interessante é 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.

Classificação dos Padrões

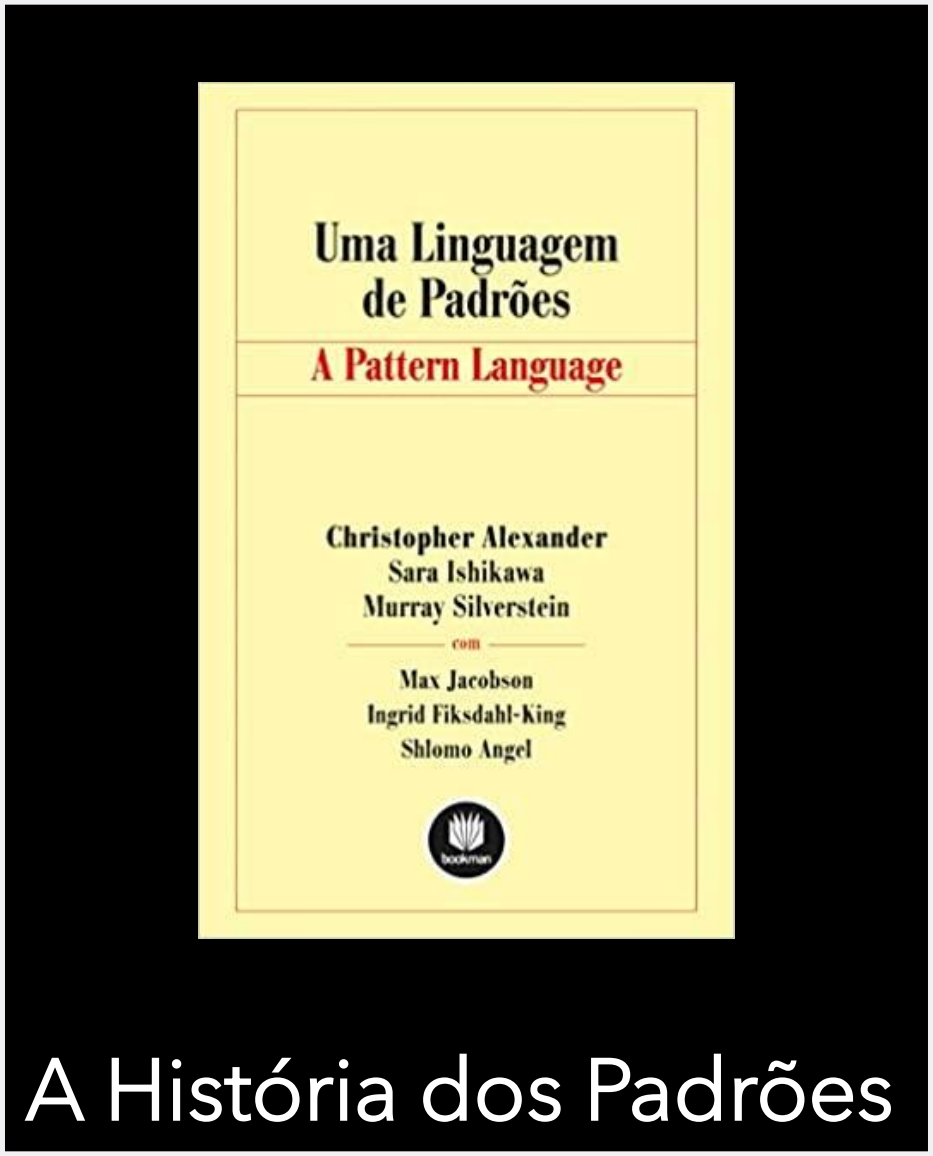
- Além disso, todos os padrões podem ser categorizados por seu propósito, ou intenção:
  - Os **padrões criacionais** fornecem mecanismos de criação de objetos que aumentam a flexibilidade e a reutilização de código.
  - Os **padrões estruturais** explicam como montar objetos e classes em estruturas maiores, enquanto ainda mantém as estruturas flexíveis e eficientes.
  - Os **padrões comportamentais** cuidam da comunicação eficiente e da assinalação de responsabilidades entre objetos.

A 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.

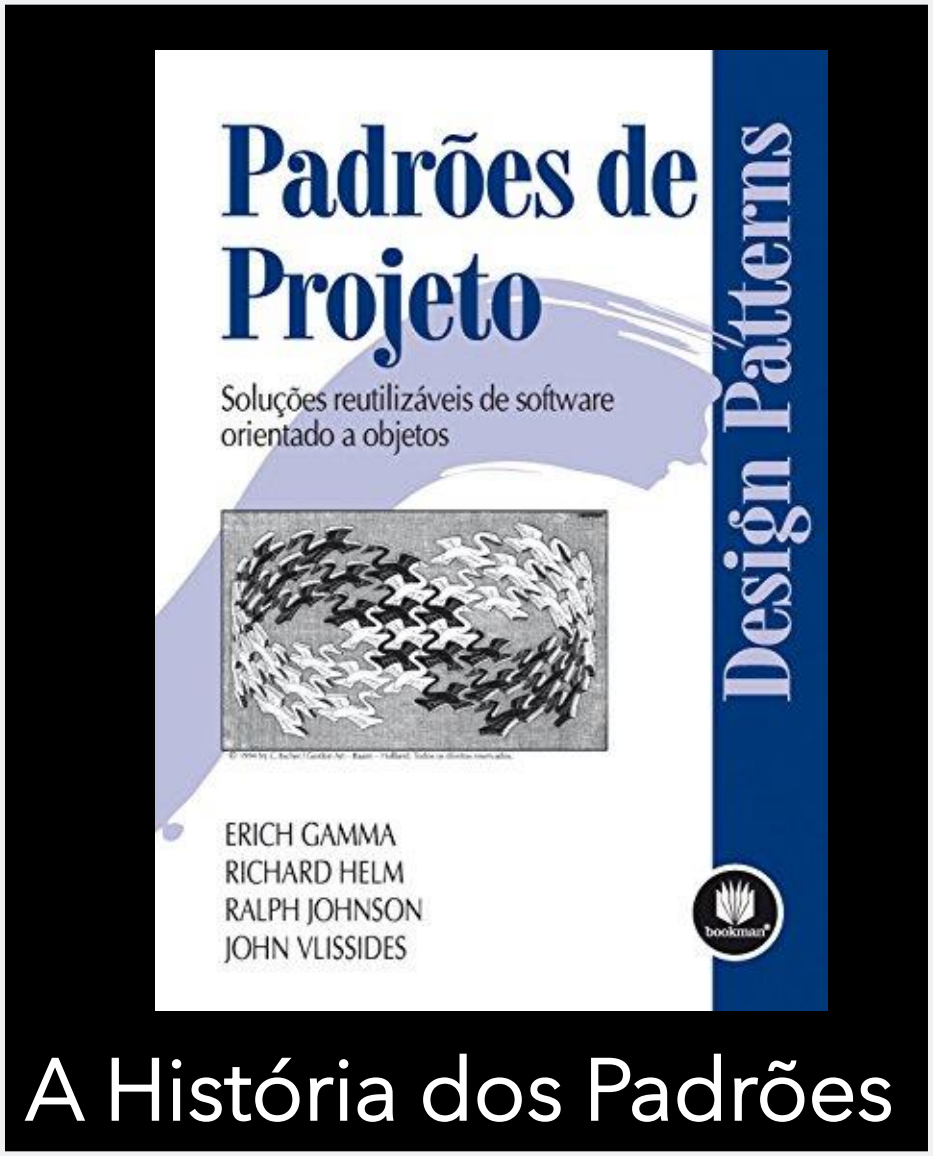


A História dos Padrões

- O conceito de padrões foi primeiramente descrito por ***Christopher Alexander*** em ***Uma Linguagem de Padrões***.

- 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 História dos Padrões

- 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***, 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 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"***.

Por que devo aprender design pattern?

- 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 Padrões de Projetos

Características de um bom projeto de software

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.

Reutilização de código

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

Reutilização de código

- 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 componentes do software e torná-los de mais fácil reutilização. Contudo, isso às vezes vem com um preço de tornar os componentes mais complicados.

Reutilização de código

- Exemplos de 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. Identificam abstrações importantes para a solução do problema, representam eles por classes e definem suas relações. O Junit é um exemplo.
- Existem um ***nível médio*** onde se encaixam os ***padrões de projeto***, são menores e mais abstratos que frameworks, sendo a verdadeira descrição de como um par de classes pode se relacionar e interagir entre si.

Reutilização de código

- O nível de reutilização aumenta quando você move classes para padrões e, finalmente, para frameworks.

Extensibilidade

- A mudança é uma constante na vida de programador:
  - Você já publicou um game para windows e os usuários pedem para mac/os;
  - Você criou um framework de interface gráfica com botões quadrados, meses mais tarde é solicitado com botões redondos;
  - Você desenhou uma arquitetura brilhante para um site de e-commerce, mas apenas um mês depois os clientes pedem por uma funcionalidade que permita que eles aceitem pedidos por telefone.
- Cada desenvolver vai ter um dúzia de histórias e situações parecidas.

Extensibilidade

- Quando você acaba de desenvolver e liberar uma versão do seu software você passa a conhecer melhor o problema. Você cresceu profissionalmente e acha o seu código antigo horrível.

- Algo além do seu controle mudou, alguma tecnologia ou lei.

- As regras do jogo mudam: seu cliente estava satisfeito com seu produto mas vê “onze” alterações (pequenas mudanças) que ele gostaria de ver implementadas.

Extensibilidade

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

Esse é um dos motivos de fazermos um projeto da arquitetura de uma aplicação.

- **Princípios de projeto**

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

Princípios de projeto

Infelizmente as respostas para essas perguntas são diferentes dependendo do tipo de aplicação que você está construindo. De qualquer modo existem vários princípios universais de projeto de software.

Encapsule o que varia

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

O objetivo principal é minimizar o efeito causado por mudanças.

- Situação:
  - Imagine que seu projeto é um navio, e as mudanças são terríveis minas que se escondem sob as águas. Atinja uma 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 for atingido e vai continuar flutuando.

- Da mesma forma, ***você pode isolar as partes de um programa em módulos independentes***, protegendo o resto do código de efeitos adversos. Desta forma você gasta menos tempo fazendo o programa voltar a funcionar, implementado e testando mudanças.

- *Quanto menos tempo se gasta com mudanças, mais tempo temos para implementar novas funcionalidades.*

Encapsule o que varia

Encapsulamento à nível de método.

- Digamos que temos um e-commerce.
- Em algum lugar do código há um método **obterTotalPedido()** que calcula o total final de um pedido, incluindo os impostos.
- Situação:
  - Podemos antecipar que o código relacionado aos impostos vai precisar mudar no futuro, o rateio das taxas depende de legislação vigente. Como resultado será necessário mudar o método com certa frequência.
  - Mas o nome do método utilizado não demonstra a importância dos impostos no processo.

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

Encapsule o que varia

Encapsulamento à nível de método.

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

- As mudanças relacionadas a impostos se tornaram isoladas em um único método.

- Se o cálculo da lógica de impostos se tornar muito complexa, fica agora mais fácil movê-lo para uma classe separada.

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

Encapsule o que varia

Encapsulamento à nível de classe.

.Com o tempo percebe-se que existe a ***necessidade de incorporar novas responsabilidades a um método***.

.Esses comportamentos adicionais geralmente vem com novos atributos auxiliares e métodos o que ***desfoca o objetivo da classe original***.

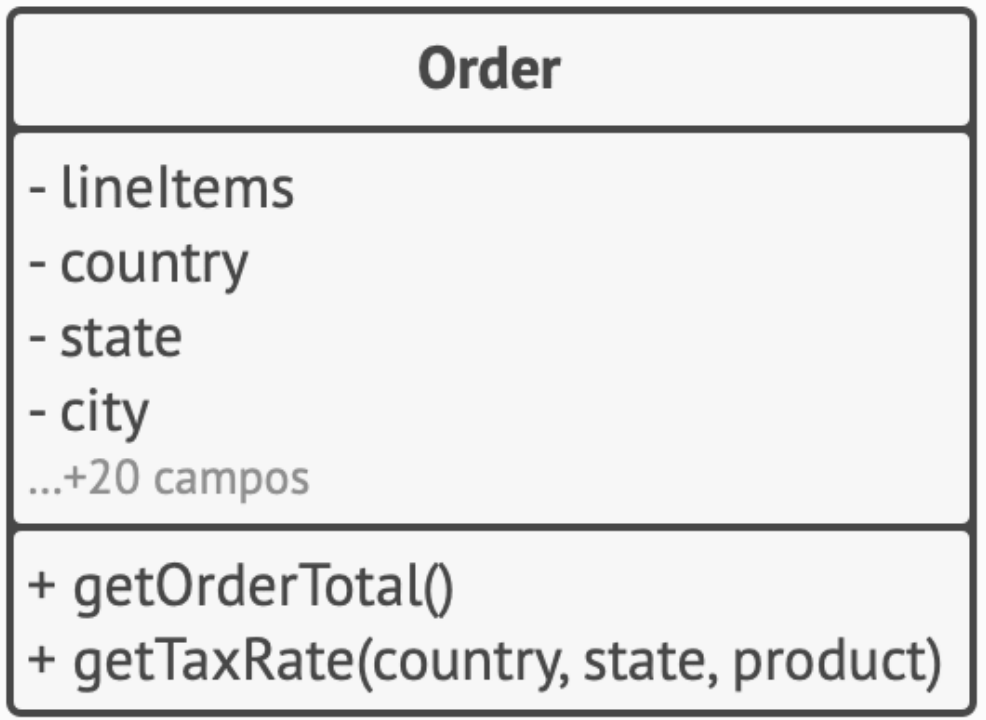
.Extraindo tudo para uma **nova classe pode-se tornar as coisas mais claras**.

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

Encapsule o que varia

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

Situação atual do projeto

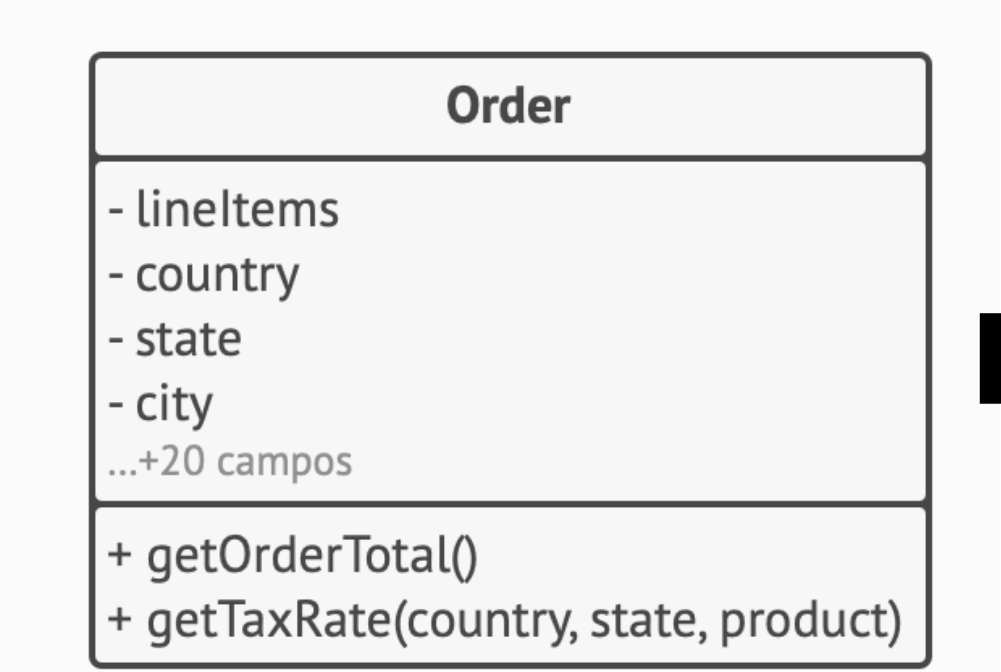


ANTES: calculando impostos em uma classe `Pedido`.

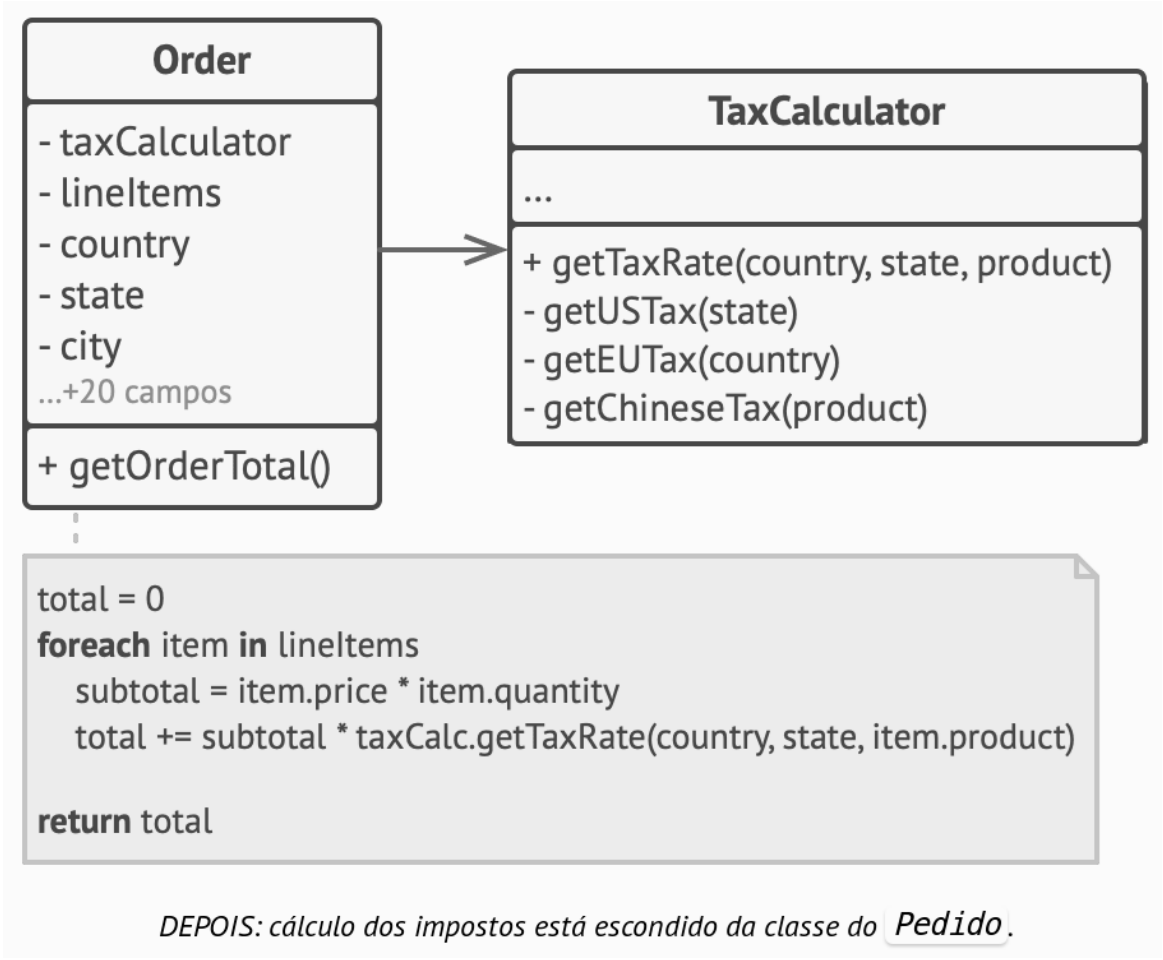
Encapsule o que varia

Encapsulamento à nível de classe.

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



ANTES: calculando impostos em uma classe `Pedido`.



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

Programe uma Interface, não uma implementação

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

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

Programe uma Interface, não uma implementação

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

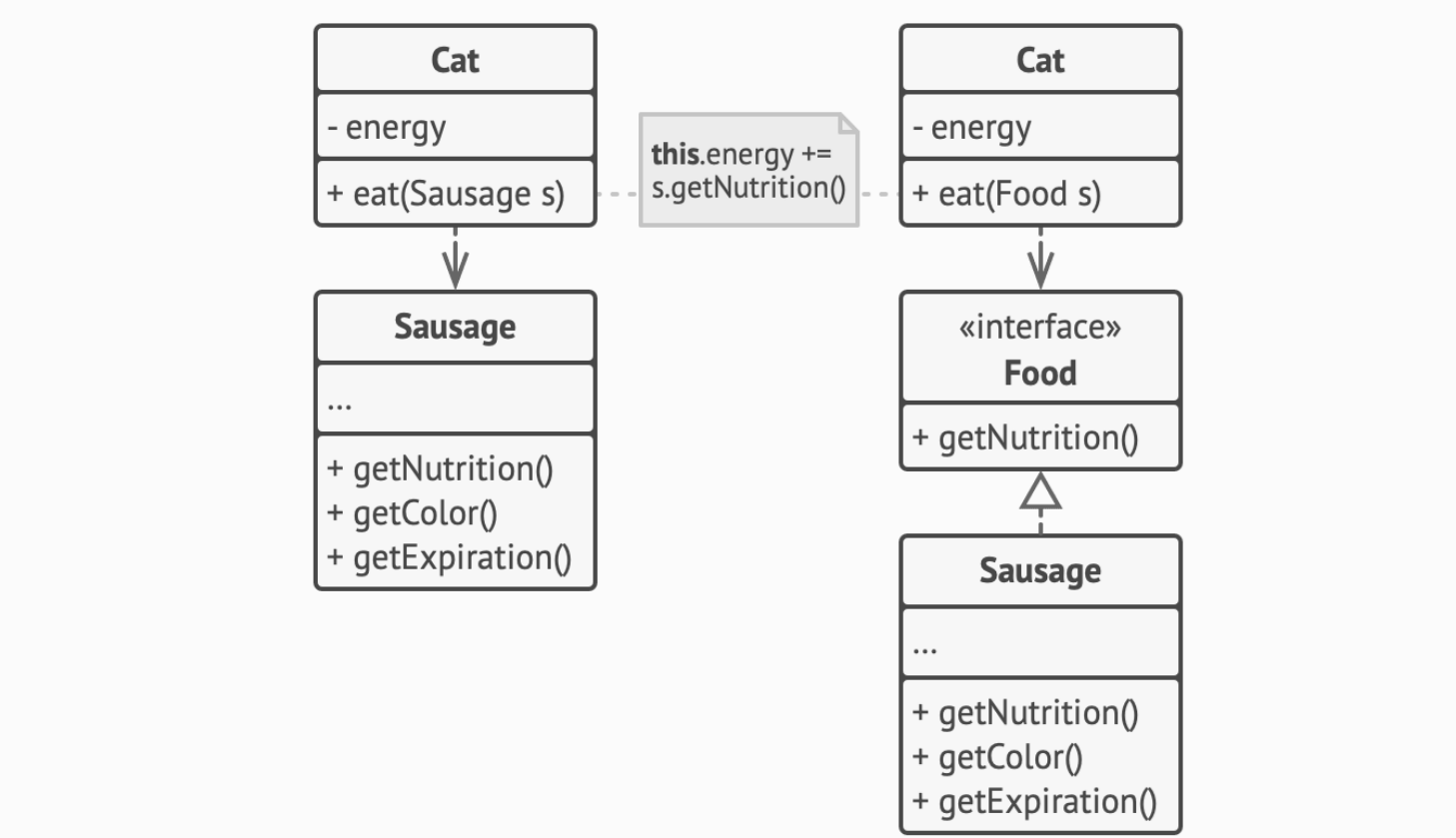
- Determinar o que exatamente um objeto precisa do outro: quais métodos ele executa?
- Descreva estes métodos em uma nova interface ou classe abstrata.
- Faça a classe que é uma dependência implementar essa interface.
- Agora faça a segunda classe ser dependente dessa interface ao invés de fazer isso na classe concreta. Você ainda pode fazê-la funcionar com objetos da classe original, mas a conexão é agora mais flexível.

Programe uma Interface, não uma implementação

Após fazer esta mudança, você provavelmente não sentirá qualquer benefício imediato.

Pelo contrário, o código parece mais complicado que estava antes.

Contudo, se você sentir que é um bom ponto de extensão para um funcionalidade adicional, ou que outros desenvolvedores que usam seu código podem querer estendê-lo, use esta regra.



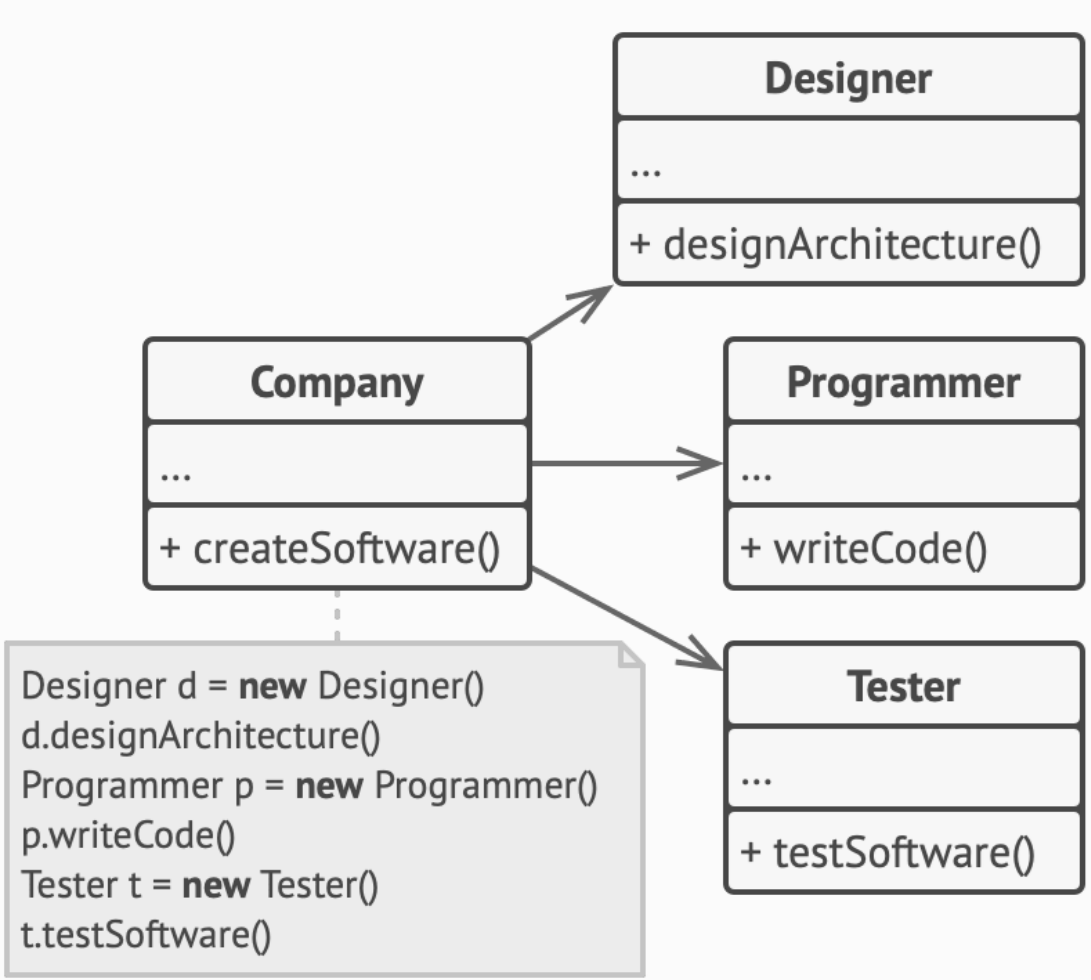
Antes e depois de extrair a interface. O código à direita é mais flexível que o código à esquerda, mas também é mais complicado.

Programe uma Interface, não uma implementação

Vamos ver outro exemplo que ilustra que trabalhar com objetos através de interfaces pode ser mais benéfico que depender de classes concretas.

Imagine que você está criando um simulador de empresa desenvolvedora de software.

Você tem diferentes classes que representam vários tipos de funcionários.



ANTES: todas as classes eram firmemente acopladas.

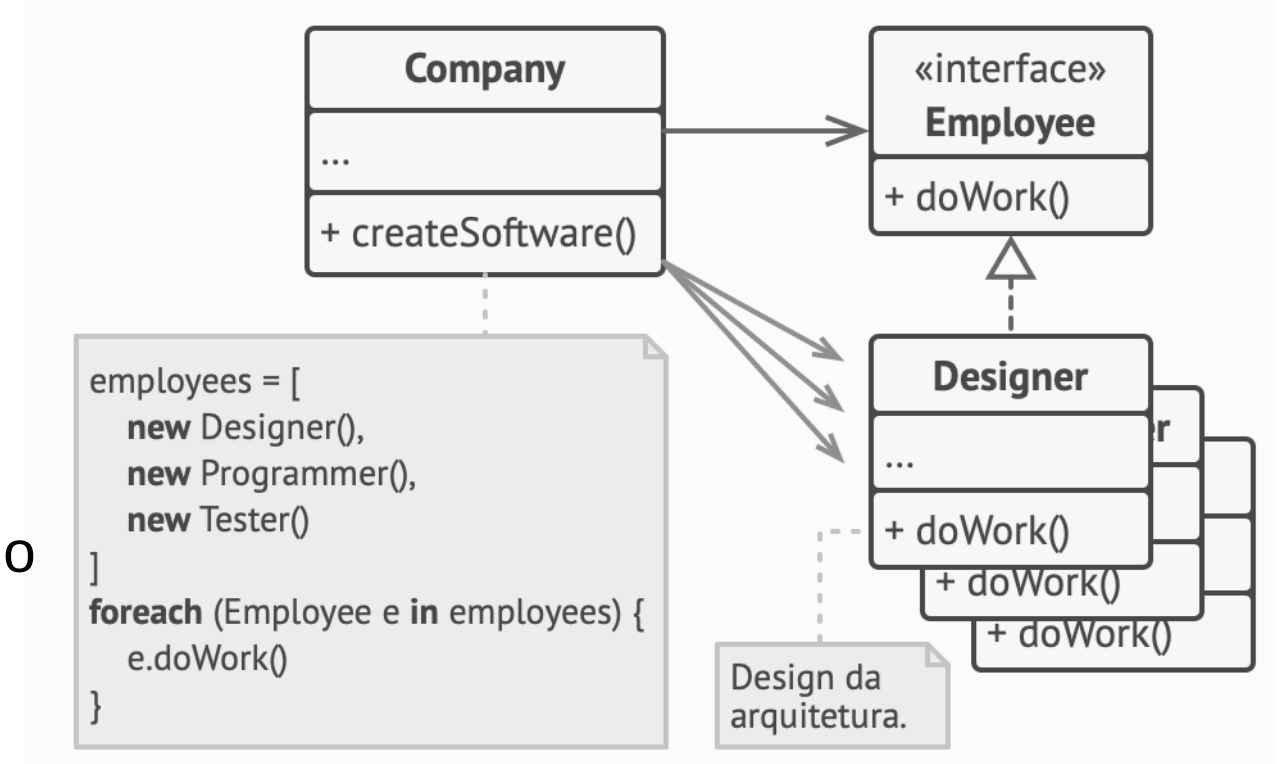
Programe uma Interface, não uma implementação

No começo a classe Empresa era firmemente acoplada as classes concretas dos empregados.

Contudo, apesar da diferença em suas implementações, pode-se generalizar vários métodos relacionados ao trabalho e então extrair uma interface comum para todas as classes de empregados.

Após fazer isso, pode-se aplicar o polimorfismo dentro da classe Empresa, tratando vários objetos de empregados através da interface Empregado.

Melhor: o polimorfismo ajudou a simplificar o código, mas o resto da classe Empresa ainda depende de classes concretas.



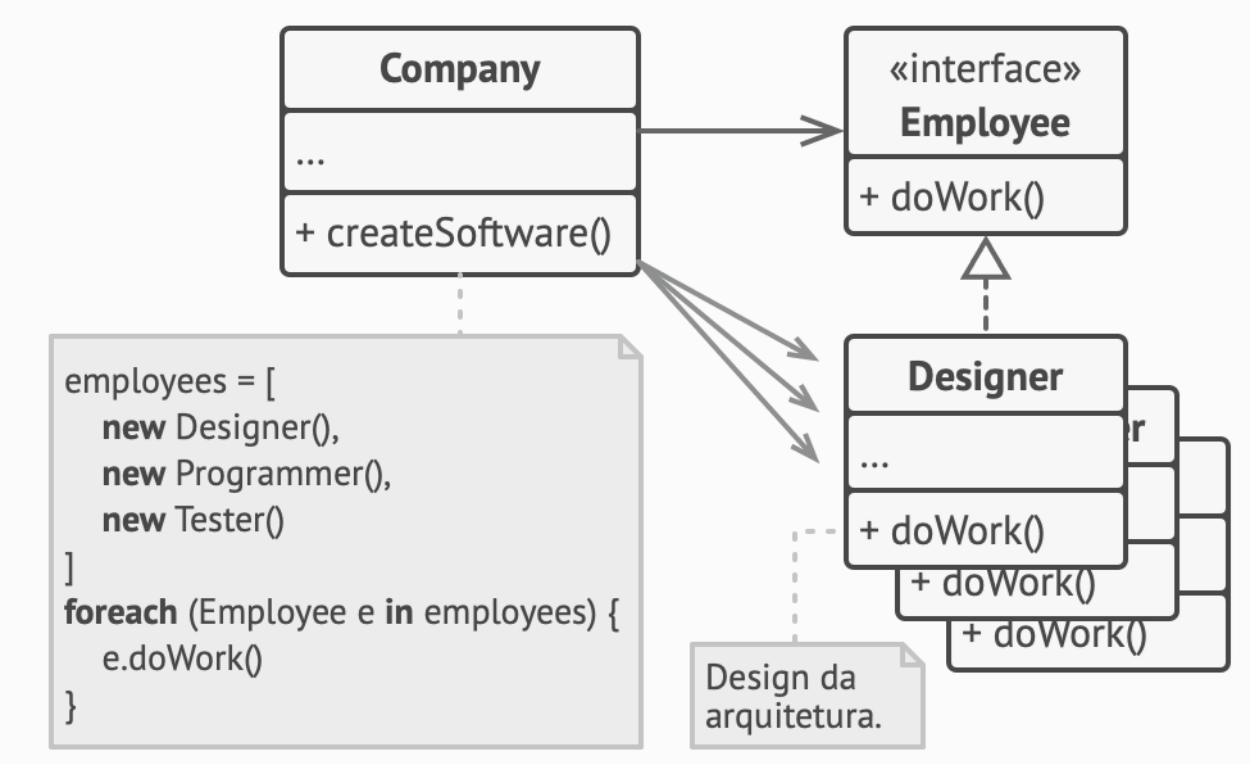
Programe uma Interface, não uma implementação

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, teríamos que sobrescrever a maior parte da classe Empresa ou invés de reutilizar aquele código.

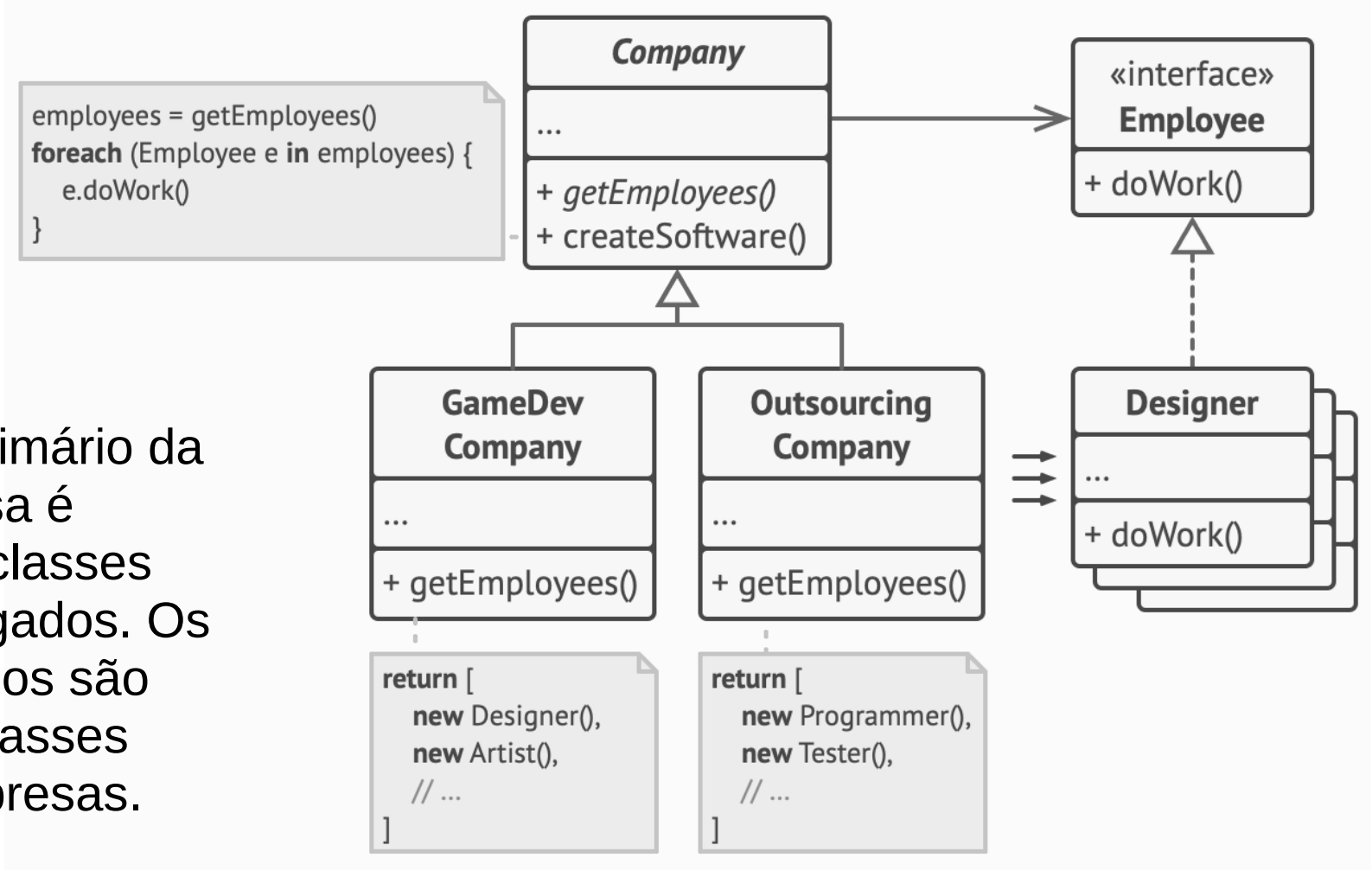
Para resolver este problema, nós podemos declarar o método para obter empregados como abstrato.

Cada empresa concreta irá implementar este método de forma diferente, criando apenas aqueles empregas que precisam.



Programe uma Interface, não uma implementação

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



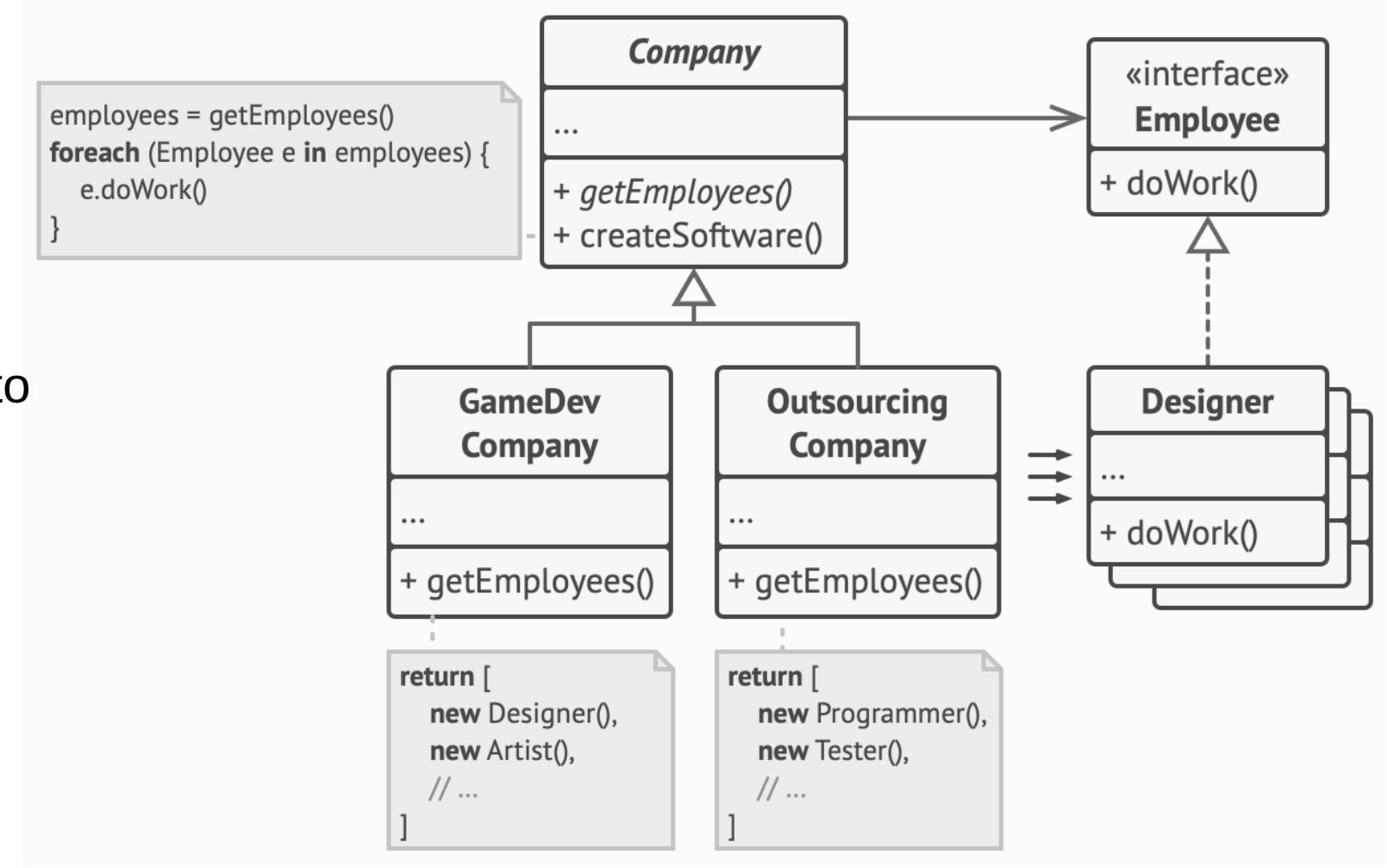
Programe uma Interface, não uma implementação

- Após a mudança, a classe Empresa se tornou independente de várias classes 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.

- Observação: *Factory method*.



Prefira Composição sobre Herança

- A herança é o meio mais obvio de reutilizar código entre classes. Você tem duas classes com o mesmo código. Cria uma classe base comum para ambas as duas classes e mova o código similar para dentro dela.

- Infelizmente, a herança vem com um lado ruim que se torna aparente apenas quanto seu programa já tem um monte de classes e mudar tudo fica muito difícil.

Prefira Composição sobre Herança

- Problemas com o uso de herança:
  - ***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 como comportamento base:*** isso é importante porque objetos da subclasse podem ser passados para qualquer código que espera objetos da superclasse e você não quer que aquele código quebre.

Prefira Composição sobre Herança

- Problemas com o uso de herança:
  - ***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 o programador faz a superclasse ficar ciente de alguns detalhes da subclasse 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.

Prefira Composição sobre Herança

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

Prefira Composição sobre Herança

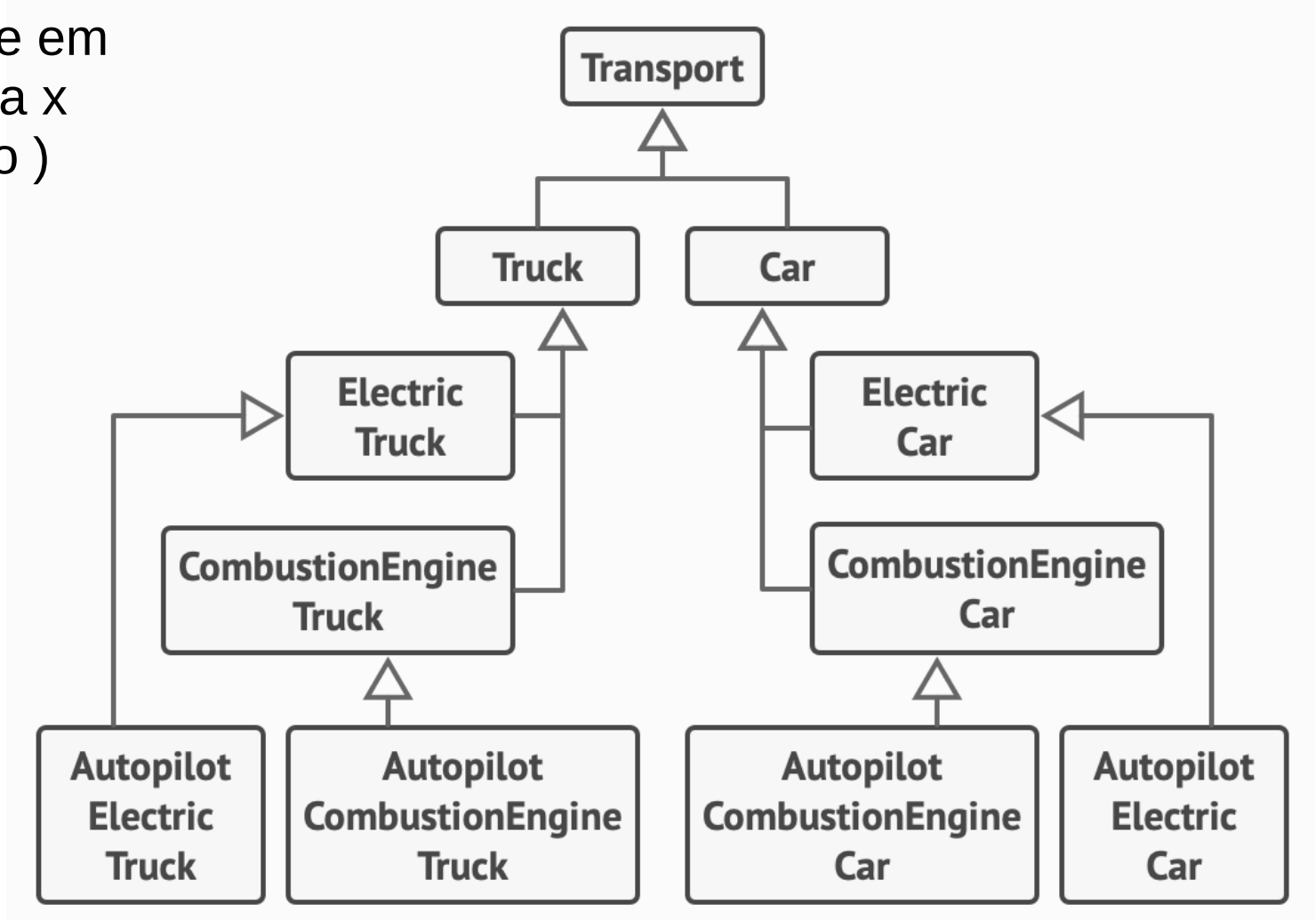
- Existe alternativa para a herança a composição.

- 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 que é uma versão mais relaxada a composição, onde um objeto pode ter uma referência para um outro, mas não gerencia seu ciclo de vida.

Prefira Composição sobre Herança

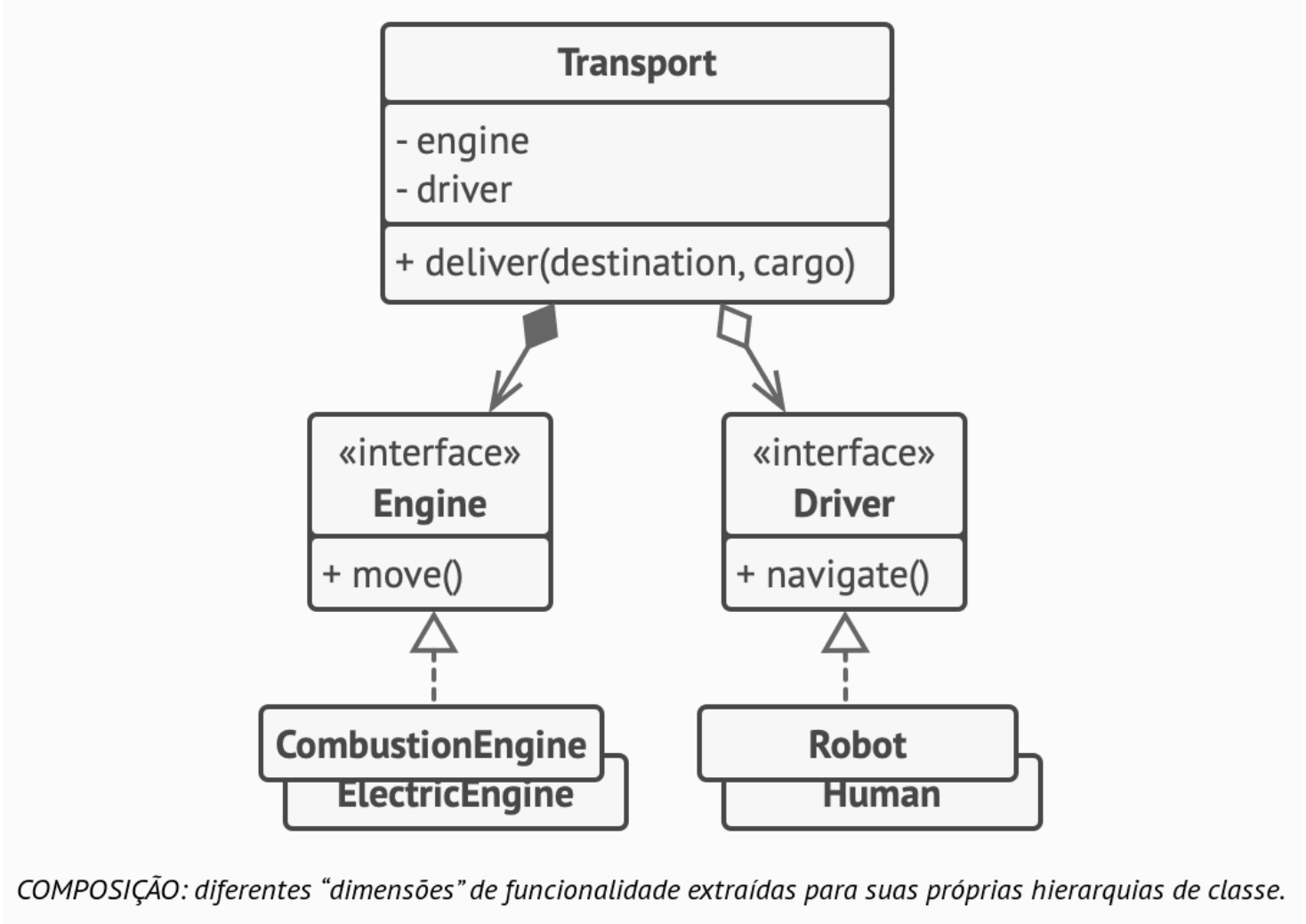
Herança: estendendo uma classe em várias dimensões (tipo de carga x tipo motor x tipo de navegação) pode levar a uma explosão combinada de subclasses



Prefira Composição sobre Herança

- Como pode-se observar, cada parâmetro adicional resulta em uma multiplicação do número de subclasses.
- Tem muita duplicação de código entre as subclasses porque as subclasses não pode estender duas classes ao mesmo tempo.
- Uma solução é o uso de 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, pode-se substituir um objeto motor ligado a um objeto carro apenas assinalando um objeto de motor diferente para o carro.

Prefira Composição sobre Herança



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