**A close-up of a screen

AI-generated content may be incorrect.**

**A close-up of a chart

AI-generated content may be incorrect.**

**Dívida Técnica (Technical Debt)**

**Definição:**  
É como um "empréstimo" que se faz para entregar algo rápido hoje, mas que terá de ser "pago" (corrigido) no futuro.

**Exemplo Prático:**  
Usar *hardcoded values* (valores fixos no código) em vez de configurar variáveis de ambiente, sabendo que isso trará problemas futuros.

**Code Smells ("Maus Cheiros" de Código)**

**Definição:**  
Indicadores superficiais de **problemas estruturais no código**, que sugerem a necessidade de refatoração (*refactoring*).  
Não são erros funcionais, mas sinais de má prática ou complexidade excessiva.

**Como Identificar:**

* Ferramentas de análise estática (ex: SonarQube).
* Revisões de código (*code reviews*).
* Sensibilidade da equipa a práticas de *clean code*.

**A screenshot of a computer

AI-generated content may be incorrect.**

**A close-up of a computer screen

AI-generated content may be incorrect.**

Adapter

O Adapter age como um 'tradutor' entre duas classes que não falam a mesma 'língua'."

* **Casos de uso:** Integração com APIs externas ou bibliotecas antigas.
* **Cuidado:** "Não confundir com o *Facade*, que simplifica interfaces complexas, mas não as converte."
* Structural Pattern

Criar a classe do ADAPTER

public class XmlToJsonAdapter<T> : IJsonParser<T>

{

public T Parse(string data)

{

IXmlParser<T> xmlParser = new XmlParser<T>();

return xmlParser.Parse(data);

}

public string ConvertToJson(T obj)

{

IJsonParser<T> jsonParser = new JsonParser<T>();

return jsonParser.ConvertToJson(obj);

}

}

Alterar o MAIN

//INterface que o cliente espera /

IJsonParser<Note> parser = new XmlToJsonAdapter<Note>();

Note note = parser.Parse(someXml);

Console.WriteLine(parser.ConvertToJson(note));

A screenshot of a computer

AI-generated content may be incorrect.

* **Vantagem:** Elimina construtores "telescópicos" (ex: new Computador(CPU, RAM, GPU, ...)).
* Builder para objetos com muitos parâmetros.
* **Cuidado:** Pode adicionar complexidade se o objeto for simples.

*Ao escrever o Codigo*

*"Imaginem que estamos a construir um sistema para uma concessionária de carros. Cada carro tem múltiplas propriedades: cor, motor, número de portas, marca, etc. Se usarmos um construtor tradicional, rapidamente o código fica confuso, especialmente se alguns valores forem opcionais ou tivermos combinações repetidas."*

**2. Padrão Builder: Construção Passo a Passo**

*"O****Builder****resolve isto dividindo a construção em etapas. Em vez de termos um construtor gigante, temos métodos pequenos e claros, cada um responsável por uma parte do objeto."*

**Pontos-chave a destacar:**

* **Encadeamento de Métodos** (*Method Chaining*):  
  *"Reparem que cada setX() retorna this. Isto permite escrever código fluído, como uma frase:"*
* **Controle sobre o processo**:  
  *"O Builder permite definir apenas o que interessa. Se não especificarmos a cor, usará um valor padrão (ex: 'vermelho')."*
* **Imutabilidade**:  
  *"O objeto Car só é criado no final, com build(). Antes disso, podemos validar ou ajustar os dados sem riscos."*

**3. Padrão Director: Configurações Pré-Definidas**

*"Mas e se houver combinações que usamos frequentemente? Por exemplo, 'carro desportivo' ou 'carro familiar'? Criar manualmente cada um é repetitivo. Aqui entra o****Director****."*

**Como funciona?**

* *"O Director é um 'chefe de obra' que sabe montar objetos complexos usando o Builder."*
* *"Ele encapsula receitas prontas, como buildLambo() ou buildMazda()."*

**Exemplo prático:**

typescript

Copy

const director = new Director();

const meuLambo = director.buildLambo(); // Lamborghini pré-configurado

*"Internamente, o Director usa o Builder, mas o cliente (quem chama o código) não precisa saber os detalhes. É como pedir um 'café com leite' em vez de explicar ao barista a receita!"*

**Vantagens:**

* **Consistência**: *"Garante que todos os Lamborghinis saem iguais."*
* **Simplicidade**: *"O código cliente fica mais limpo."*
* **Facilidade de manutenção**: *"Se a receita do 'Lambo' mudar, alteramos apenas o Director."*

A close-up of a message

AI-generated content may be incorrect.

* **Uso em Bibliotecas:** Base de sistemas de eventos em React (hooks) e Angular (RxJS).
* **Citação de Fowler:** *"O Observer é essencial para arquiteturas reativas e desacopladas."*
* **Desvantagem:** Observadores não controlados podem causar *memory leaks*.

**Introdução ao Problema**

*"Imaginem que estamos a desenvolver um sistema de notificações, como um blog ou uma plataforma de newsletters. Precisamos de avisar os utilizadores quando um novo conteúdo é publicado. Mas como fazê-lo de forma eficiente, sem acoplar o código do 'publicador' ao código de cada 'assinante'?"*

**2. Padrão Observer: Desacoplamento entre Publicador e Assinantes**

*"O****Observer****(ou****Pub-Sub****) resolve isto definindo dois papéis claros:"*

1. **Publisher (Publicador)**: Mantém uma lista de "ouvintes" e notifica-os quando algo acontece.
2. **Subscribers (Assinantes)**: Reagem às notificações do publicador.

**Analogia:**  
*"Pensem numa rádio (Publisher) e nos seus ouvintes (Subscribers). A rádio emite o sinal, mas não sabe quem está a ouvir. Cada ouvinte decide se quer sintonizar ou desligar o rádio."*

**3. Componentes do Código**

**🔹 Interface**EventListener**(Contrato dos Observers)**

*"É o contrato que garante que todos os assinantes sabem receber notificações. Funciona como um 'acordo' entre o publicador e os ouvintes."*

**🔹 Classe**Subscriber**(Assinante Concreto)**

*"Cada assinante implementa update() à sua maneira. Aqui, simulamos um e-mail, mas poderia ser um SMS, push notification, etc."*

**🔹 Classe**Publisher**(Publicador)**

*"O publicador não sabe quem são os assinantes, apenas chama update() neles. Isto é o****desacoplamento****em ação!"*

**4. Vantagens do Padrão Observer**

✅ **Extensibilidade**: *"Podemos adicionar novos tipos de assinantes (ex: SMSSubscriber) sem modificar o Publisher."*  
✅ **Baixo acoplamento**: *"O publicador não depende de classes concretas, apenas da interface EventListener."*  
✅ **Dinamismo**: *"Assinantes podem entrar/sair da lista em tempo de execução (ex: addSubscriber())."*

**A close-up of a text

AI-generated content may be incorrect.**

**Imaginemos que estamos a fazer uma compra online. No fim, precisamos de escolher como queremos pagar: cartão de crédito, PayPal . O valor é o mesmo, o que muda é a forma como o pagamento é feito.**

**Este exemplo implementa o padrão Strategy, que nos permite mudar o comportamento de um sistema em tempo de execução, sem alterar o seu funcionamento principal.**

**No nosso exemplo:**

* **IPaymentStrategy é a interface – define a estratégia base para qualquer tipo de pagamento. Representa o papel de Strategy no padrão.**
* **CreditCardStrategy, PayPalStrategy e são as implementações concretas dessa estratégia – cada uma sabe como processar o pagamento da sua forma. São os Concrete Strategies.**
* **PaymentService é o Context – representa o sistema de pagamentos. Não sabe como o pagamento é feito, mas sabe que tem uma estratégia associada e que a pode mudar dinamicamente.**
* **As classes CreditCard e PayPal são modelos de dados auxiliares – contêm a informação e o estado dos métodos de pagamento.**

**O que este padrão nos permite fazer?**

* **Escolher a forma de pagamento em tempo de execução, sem alterar o funcionamento interno do sistema.**
* **Adicionar novas formas de pagamento no futuro sem mexer no código já existente.**

**Resumo Final**

**O padrão Strategy é como um terminal de pagamento onde escolhemos se queremos pagar com cartão, PayPal ou cripto. O sistema sabe que tem de cobrar, mas não se preocupa com os detalhes — esses ficam a cargo da estratégia escolhida.**

**Assim, conseguimos trocar a forma de pagamento sem trocar o sistema inteiro — só trocamos a estratégia.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Explicação Simples**

**Imaginemos um restaurante de hambúrgueres. Nem todos os hambúrgueres são iguais – podemos ter hambúrgueres vegetarianos ou de carne. Mas o processo de encomendar um hambúrguer é sempre o mesmo: pedimos, preparam-no e entregam-no.**

**No nosso exemplo:**

**Burger é uma interface – representa qualquer tipo de hambúrguer. Aqui temos o papel de Product, ou seja, o contrato base para todos os hambúrgueres.**

**VeggieBurger e BeefBurger são implementações diferentes – um é vegetariano, o outro é de carne. Estas classes concretas representam os Concrete Products, ou seja, os hambúrgueres reais que serão preparados.**

**BurgerFactory é uma classe abstrata – define como se encomenda um hambúrguer (com o método orderBurger()), mas não diz qual hambúrguer criar. Esta é a função do Creator, a classe que sabe como tratar o pedido, mas delega a criação específica.**

**As classes VeggieFactory e BeefFactory são fábricas concretas – cada uma sabe que tipo de hambúrguer deve criar. Estas são os Concrete Creators, as implementações reais que decidem qual hambúrguer concreto produzir ao implementar o método createBurger().**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**O que este padrão nos permite fazer?**

**Separar o processo de criação do objeto do uso do objeto.**

**Adicionar novos tipos de hambúrgueres no futuro sem alterar o código existente (seguindo o princípio aberto/fechado da programação orientada a objetos).**

**Tornar o código mais organizado e fácil de manter, especialmente em sistemas com muitos tipos de objetos parecidos.**

A close-up of a screen

AI-generated content may be incorrect.

**Explicação Simples**

Imaginemos que queremos construir uma casa. Para isso, precisamos de ferramentas: um martelo e uma serra. Mas a casa em si **não devia preocupar-se com que tipo de ferramentas está a usar**, apenas que as tem e que pode usá-las.

**No nosso exemplo:**

* **Hammer** e **Saw** são as ferramentas – classes com comportamento próprio (ambas têm o método Use()). Estas são as **dependências** da classe House.
* **LaserSaw** é uma versão especializada da serra, que herda de Saw – um exemplo de **polimorfismo**, onde podemos trocar a implementação sem mudar o código que a usa.
* **IToolUser** é uma interface que define métodos para receber as ferramentas – representa o **contrato** de injeção, útil para aplicar boas práticas como a inversão de controlo.
* **House** é a classe que usa as ferramentas para construir algo. Em vez de criar as ferramentas diretamente dentro de si (o que a tornaria dependente delas), permite que lhe sejam fornecidas através dos métodos SetHammer() e SetSaw(). Esta é a **classe dependente**, mas as dependências são injetadas de fora.
* **Program (Main)** é o ponto de entrada onde as ferramentas são criadas e fornecidas à casa – ou seja, aqui é feita a **injeção de dependências manualmente**.

**O que esta abordagem nos permite fazer?**

* Substituir ferramentas (por exemplo, Saw por LaserSaw) **sem mudar o código da classe House**.
* Testar o comportamento da House com ferramentas diferentes, **sem alterar a lógica da construção**.
* Evitar **acoplamento forte**, o que facilita a manutenção, testes e evolução do sistema.

**Resumo Final**

A **Injeção de Dependência** é como entregar as ferramentas certas a quem vai construir, em vez de obrigar essa pessoa a fabricá-las antes de começar a trabalhar.

A House só quer construir. Se lhe dermos um martelo e uma serra, ela trata do resto — **sem saber de onde vieram** ou como funcionam por dentro. Isto torna o código mais limpo, reutilizável e preparado para mudanças.

A close-up of a circle

AI-generated content may be incorrect.

**Explicação Simples**

Imaginemos que temos diferentes tipos de pizzas (como Pepperoni ou Vegetariana) e diferentes tipos de restaurantes (como um americano ou italiano). Cada restaurante prepara e entrega a pizza à sua maneira, mas a pizza em si é independente do restaurante.

**No nosso exemplo:**

* **Pizza** é uma classe abstrata – representa o conceito geral de pizza, com as propriedades sauce e toppings e o método assemble(). É a **implementação** que pode variar.
* **PepperoniPizza** e **VeggiePizza** são pizzas concretas – cada uma implementa a forma como é montada. Representam as **concrete implementations**.
* **Restaurant** é uma classe abstrata – define como o restaurante adiciona molho, ingredientes e faz a entrega. É a **abstração**, e depende de uma instância de Pizza (a implementação).
* **AmericanRestaurant** e **ItalianRestaurant** são restaurantes concretos – cada um define **como** monta a pizza (que molho usar, que toppings adicionar). São as **refinadas abstrações**.
* O método deliver() chama os passos necessários: escolhe o molho, os ingredientes, monta a pizza e entrega.

**O que este padrão nos permite fazer?**

* Separar a lógica do restaurante da lógica da pizza – cada um **pode mudar sem afetar o outro**.
* Criar novas pizzas ou novos restaurantes **sem alterar o código existente**.
* Tornar o sistema **mais flexível**, permitindo combinações entre várias pizzas e vários tipos de restaurantes.

**Resumo Final**

O **padrão Bridge** é como permitir que qualquer tipo de restaurante prepare qualquer tipo de pizza, sem os dois estarem presos um ao outro. Podemos ter uma pizza de pepperoni feita num restaurante italiano, ou uma vegetariana num restaurante americano — **misturar livremente** sem reescrever o código.

Separar a “abstração” (o restaurante) da “implementação” (a pizza) dá-nos **liberdade para expandir** o sistema em qualquer direção.

A close-up of a text

AI-generated content may be incorrect.

**Explicação Simples**

Imaginemos que estás a pedir um café. Começas com um simples espresso. Depois queres adicionar leite... depois açúcar. Mas cada cliente pode querer uma combinação diferente, e não queremos criar uma nova classe para cada variação de café possível.

Este exemplo aplica o **padrão Decorator**, que nos permite **adicionar funcionalidades extra a um objeto dinamicamente**, sem modificar a sua estrutura original.

**No nosso exemplo:**

* **Coffee** é a classe abstrata base – representa um café genérico. Tem um método getDescription() e um método abstrato cost(). É o nosso **componente base**.
* **Espresso** é uma implementação concreta de Coffee – representa um café simples. É o **componente concreto**.
* **CoffeeDecorator** é uma classe abstrata que **herda de Coffee** – serve como base para os ingredientes extra. É o **decorador base**.
* **WithMilk** e **WithSugar** são decoradores concretos – recebem um Coffee e adicionam comportamento: aumentam o custo e a descrição. São os **decoradores reais** que adicionam novas funcionalidades.
* **Program (Main)** mostra como podemos encadear decoradores: começamos com um espresso, depois adicionamos leite, e depois açúcar — tudo isso **sem alterar o objeto original**, apenas o “embrulhando” em novas camadas.

**O que este padrão nos permite fazer?**

* Adicionar funcionalidades de forma **modular** e **dinâmica**, sem criar subclasses para cada combinação.
* Seguir o princípio **aberto/fechado** – o código está aberto para extensão, mas fechado para modificação.
* Compor objetos complexos de forma simples, através de **camadas de responsabilidade**.

**Resumo Final**

Este exemplo aplica o **padrão Decorator**, que nos permite **adicionar funcionalidades extra a um objeto dinamicamente**, sem modificar a sua estrutura original.

Cada ingrediente é um “decorador” que embrulha o anterior e adiciona algo novo – sem nunca alterar o que já existia. O resultado é um sistema flexível, reutilizável e fácil de expandir, sem criar centenas de subclasses para cada possível combinação.

A close-up of a text

AI-generated content may be incorrect.

**Explicação Simples**

Imaginemos que temos uma luz e queremos controlá-la com um comando remoto. A ideia é que o comando remoto **não saiba exatamente o que está a controlar** – apenas sabe que tem de "executar um comando".

**No nosso exemplo:**

* **ICommand** é a interface base – define o método Execute(), que representa uma ação a ser executada. É o **comando genérico**.
* **Light** é o **receiver** – o objeto que sabe fazer as ações reais, como TurnOn() e TurnOff().
* **TurnOnLightCommand** e **TurnOffLightCommand** são **comandos concretos** – cada um sabe exatamente o que fazer com a Light. Guardam uma referência à luz e implementam Execute() com a ação específica.
* **RemoteControl** é o **invoker** – o botão do comando remoto. Não sabe o que o comando faz, só sabe que tem um botão para o executar.
* **Program (Main)** é o cliente – cria a luz, os comandos e diz ao comando remoto qual comando deve executar.

**O que este padrão nos permite fazer?**

* **Desacoplar quem executa a ação (remote)** de **quem sabe como a ação é feita (luz)**.
* Encapsular ações como objetos, tornando possível: **guardar, desfazer, repetir ou agendar** comandos.
* Adicionar novas ações (comandos) **sem mudar o código do comando remoto**.

**Resumo Final**

**O que o padrão Command faz de forma técnica:**  
**O Padrão Command encapsula uma solicitação (request) como um objeto, permitindo parametrizar objetos com diferentes pedidos, enfileirar ou registar pedidos, e suportar operações que possam ser desfeitas (undoable operations).**  
Isto facilita a implementação de ações flexíveis, agendáveis, reversíveis e independentes do invocador.

O **padrão Command** é como ter um comando remoto universal: o botão do comando não sabe se vai ligar uma luz, abrir uma porta ou ligar uma máquina de café. Só sabe que, quando carregas, **executa um comando**.

Quem define o que acontece realmente é o **comando** – que sabe qual é o aparelho (o receiver) e qual é a ação. Isto torna o sistema muito flexível e extensível.

Este padrão é muito útil quando queremos **encapsular ações como objetos**, o que nos permite gravar, desfazer ou agendar essas ações no futuro.

A close-up of a text

AI-generated content may be incorrect.

**Explicação Simples**

Imaginemos que estamos a escrever num editor de texto e queremos ser capazes de fazer "undo" (desfazer) para voltar a estados anteriores do texto. A ideia é capturar e restaurar o estado sem que o próprio editor tenha de se preocupar com os detalhes de como isso é feito.

No nosso exemplo: • **TextEditor** é o originador – é o objeto que cria momentos (mementos) que guardam o seu estado atual (o texto escrito).

• **Memento** é o memento – é o objeto que guarda o estado interno do editor (o texto naquela altura).

• **History** é o caretaker – é quem guarda os vários estados (mementos) e sabe como restaurar o estado anterior, mas não precisa de saber o que está dentro dos mementos.

• **Program (Main)** é o cliente – cria o editor, guarda estados no histórico, escreve texto e pede para desfazer alterações.

O que este padrão nos permite fazer? • **Capturar e restaurar o estado de um objeto** sem violar o encapsulamento dos seus dados.

• **Permitir "undo"** (desfazer alterações) de forma limpa e organizada.

• **Guardar vários momentos** de forma segura e independente, sem misturar lógica de histórico no objeto principal.

• **Separar responsabilidades**: o editor continua focado em escrever texto; o histórico preocupa-se em guardar e restaurar estados.

Resumo Final O padrão Memento é como ter um botão de "Guardar" e "Desfazer" automático no nosso programa. O editor (originador) apenas cria e restaura os seus próprios estados sem saber como eles são guardados.

O caretaker (histórico) é que se encarrega de manter esses estados.  
Desta forma, conseguimos voltar atrás no tempo facilmente, sem que o editor tenha de gerir sozinho o seu próprio histórico.

Este padrão é muito útil sempre que precisamos de **gravar estados**, **desfazer operações** ou até **reverter mudanças** em sistemas complexos, mantendo tudo organizado e encapsulado.

**O Padrão Memento captura e externaliza o estado interno de um objeto sem violar o encapsulamento, permitindo que esse estado seja restaurado mais tarde.**  
Isto facilita a implementação de funcionalidades como undo/redo, checkpoints e históricos de alterações, de forma segura e organizada.

A close-up of a text

AI-generated content may be incorrect.

**Explicação Simples**

Imaginemos que estamos num restaurante e queremos obter informações nutricionais sobre os diferentes menus, pratos e ingredientes. Para fazer isso, utilizamos um padrão de design que nos permite **adicionar funcionalidades** (como calcular as calorias, proteínas ou hidratos de carbono) a uma estrutura composta de objetos, **sem modificar a própria estrutura**.

No nosso exemplo:

* **IVisitor** é a interface do visitante – define os métodos necessários para visitar os diferentes tipos de elementos (menu, prato, ingrediente).
* **IVisitable** é a interface que torna os elementos visitáveis – define o método Accept(), que permite que um visitante execute ações sobre os elementos.
* **Menu**, **MenuItem** e **Ingredient** são os elementos visitáveis – representam os objetos que compõem o menu do restaurante e que podem ser "visitados" para recolher informações nutricionais.
* **NutritionVisitor** é o visitante concreto – é o objeto responsável por calcular a informação nutricional, como calorias, proteínas e hidratos de carbono.
* **Program (Main)** é o cliente – cria os menus, os pratos e os ingredientes, aplica o visitante e imprime as informações nutricionais calculadas.

O que este padrão nos permite fazer?

* **Adicionar funcionalidades** como cálculos ou análise de dados a uma estrutura composta de objetos (neste caso, o menu do restaurante) sem modificar o código dos próprios elementos.
* **Centralizar as operações** feitas sobre a estrutura composta, mantendo o código organizado e fácil de entender.
* **Facilitar a adição de novos tipos de operações** no futuro (por exemplo, se quisermos adicionar um visitante para calcular o valor calórico total de todas as refeições, basta criar um novo visitante sem alterar o código dos menus ou ingredientes).
* **Desacoplar as operações dos objetos visitados**, permitindo que a estrutura do menu permaneça inalterada enquanto o comportamento é modificado por visitantes.

Resumo Final O padrão Visitor é como um consultor nutricional que pode ser enviado para cada elemento de um menu (menu, prato e ingrediente) sem que o menu precise saber como calcular a informação. O **menu** não tem de saber nada sobre como calcular as calorias ou proteínas dos seus ingredientes; quem faz isso é o **Visitor**. Ao adicionar novos visitantes no futuro, podemos introduzir novas funcionalidades (como uma análise de saúde ou uma recomendação de dieta) sem alterar o código dos menus ou ingredientes.

Este padrão é útil sempre que queremos adicionar operações a uma estrutura composta de objetos, como menus de restaurantes ou hierarquias de produtos, sem alterar o código dos objetos base. Através do **Visitor**, podemos fazer análises, cálculos ou até gerar relatórios sem modificar a estrutura do código, tornando-o mais flexível e extensível.

**Vantagens Técnicas:**

* **Desacoplamento de operações e estrutura**: O padrão Visitor permite que as operações sejam desacopladas da estrutura dos objetos. Assim, a estrutura do menu não precisa saber nada sobre como calcular os valores nutricionais – a lógica de cálculo fica centralizada no NutritionVisitor.
* **Facilidade de adição de novas operações**: Quando há a necessidade de adicionar uma nova operação (como calcular a quantidade de gordura, ou gerar relatórios nutricionais em formato PDF), basta criar um novo visitante. Isso evita alterações no código das classes da estrutura composta.
* **Centralização de lógica**: O código que realiza a operação (neste caso, o cálculo das informações nutricionais) é centralizado no visitante, tornando a manutenção e a extensão do código mais fáceis. Para adicionar uma nova operação, não é necessário modificar a estrutura de elementos, basta criar um novo visitante.

**Desvantagens Técnicas:**

* **Quebra de encapsulamento**: Um dos problemas do padrão Visitor é que ele pode quebrar o encapsulamento das classes visitadas, uma vez que o visitante precisa acessar os dados internos dos objetos para realizar suas operações. Por exemplo, o visitante precisa de acesso direto aos detalhes de cada Ingredient (como calorias, proteínas, etc.) para realizar o cálculo.
* **Complexidade adicional**: Se a estrutura de objetos for alterada (por exemplo, se novos tipos de elementos forem adicionados), os visitantes terão de ser atualizados para lidar com esses novos tipos. Isso pode aumentar a complexidade do código à medida que a estrutura se torna mais extensa.

Desvantagens

 **Quebra de encapsulamento**: Um dos problemas do padrão Visitor é que ele pode quebrar o encapsulamento das classes visitadas, uma vez que o visitante precisa acessar os dados internos dos objetos para realizar suas operações. Por exemplo, o visitante precisa de acesso direto aos detalhes de cada Ingredient (como calorias, proteínas, etc.) para realizar o cálculo.

 **Complexidade adicional**: Se a estrutura de objetos for alterada (por exemplo, se novos tipos de elementos forem adicionados), os visitantes terão de ser atualizados para lidar com esses novos tipos. Isso pode aumentar a complexidade do código à medida que a estrutura se torna mais extensa.