refactoring.guru

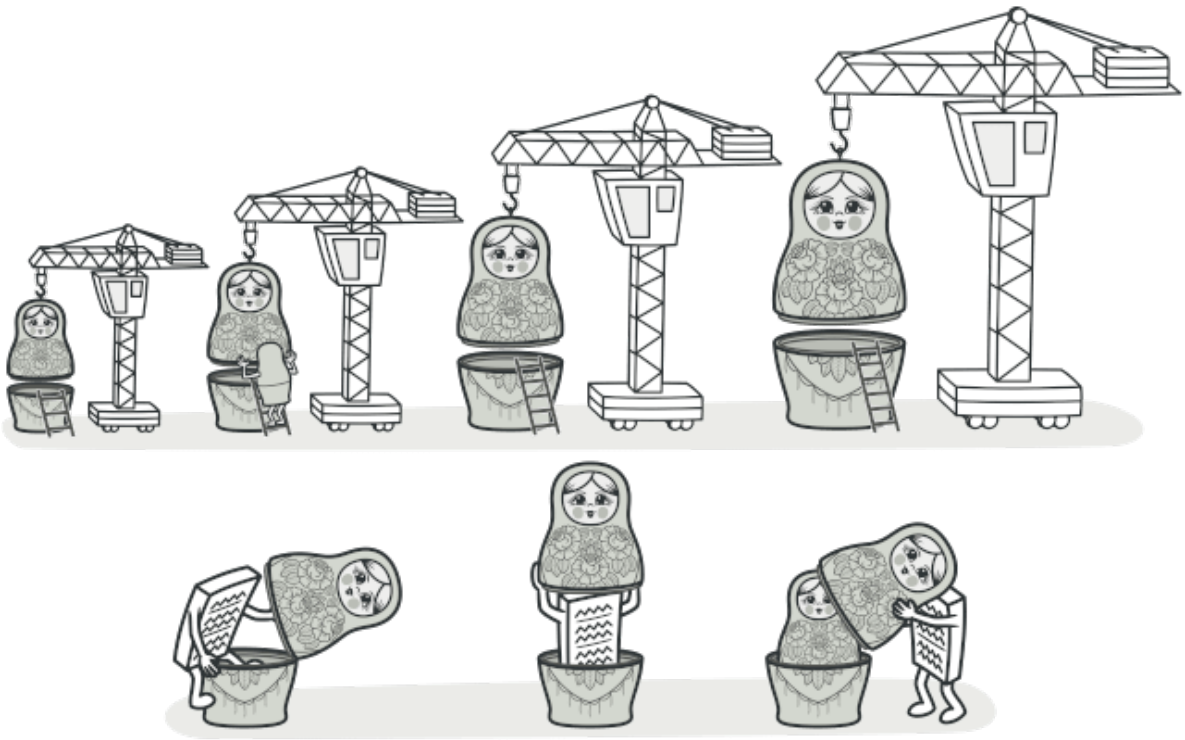
Decorator

17–23 minutes

Também conhecido como: Decorador, Envoltório, Wrapper

Propósito

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.

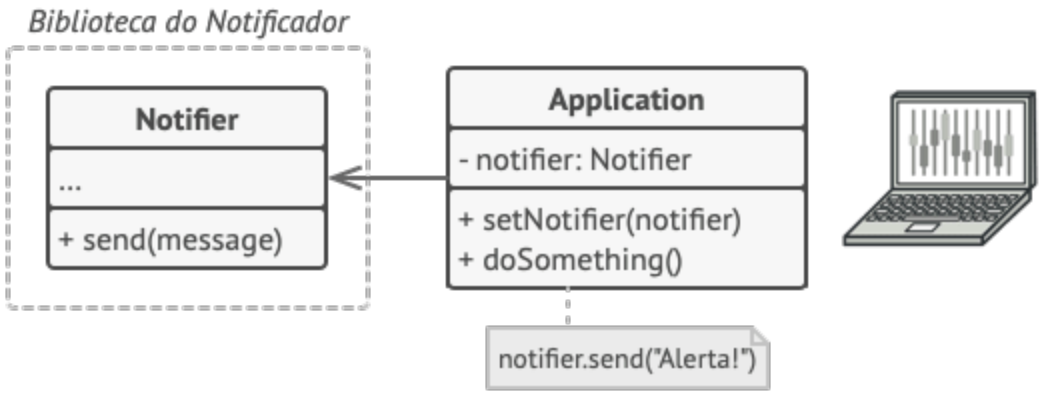


Problema

Imagine que você está trabalhando em um biblioteca de

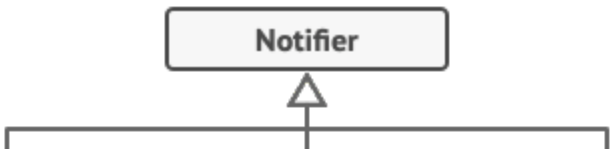
notificação que permite que outros programas notifiquem seus usuários sobre eventos importantes.

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.

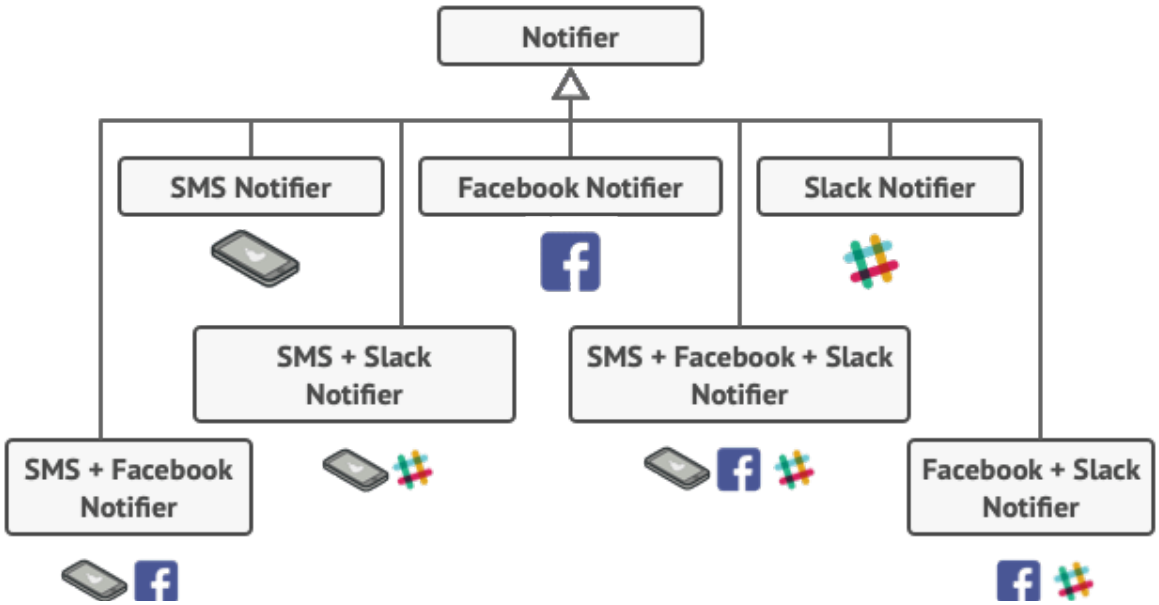


Cada tipo de notificação é implementada em uma subclasse do notificador.

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* 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

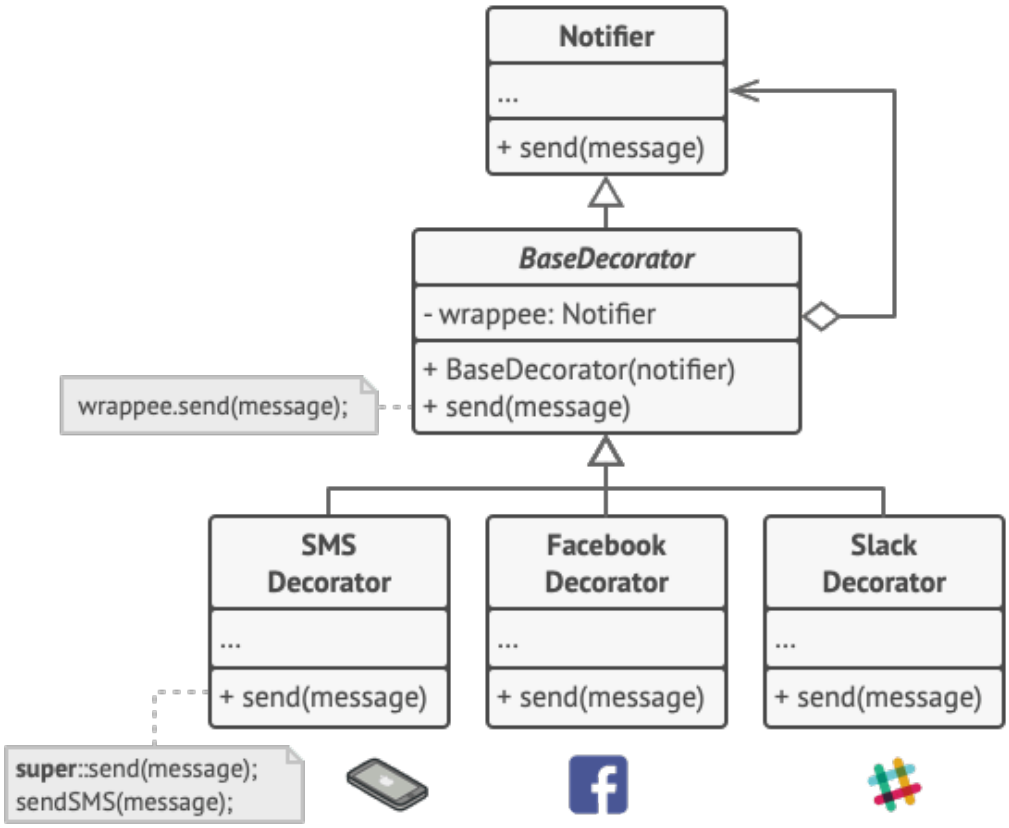
objetos, e delegar qualquer tipo de trabalho a eles. A agregação/composição é o princípio chave por trás de muitos padrões de projeto, incluindo o Decorator. Falando nisso, vamos voltar à discussão desse padrão.

Herança vs. Agregação

"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.

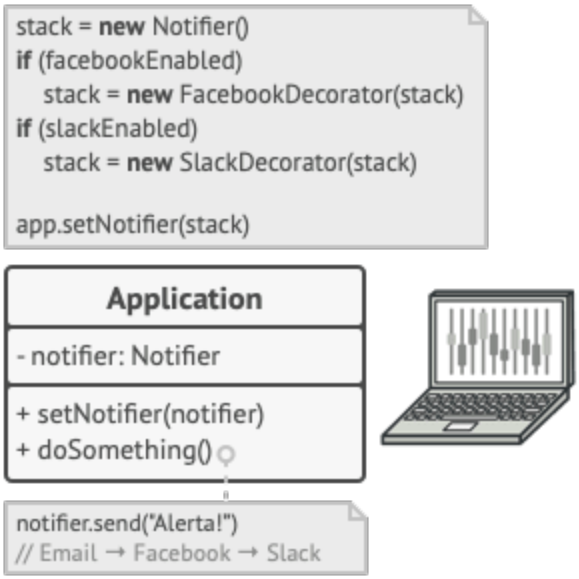
Quando um simples envoltório se torna um verdadeiro decorador? Como mencionei, o envoltório implementa a mesma interface que o objeto envolvido. É por isso que da perspectiva do cliente esses objetos são idênticos. Faça o campo de referência do envoltório aceitar qualquer objeto que segue aquela interface. 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.



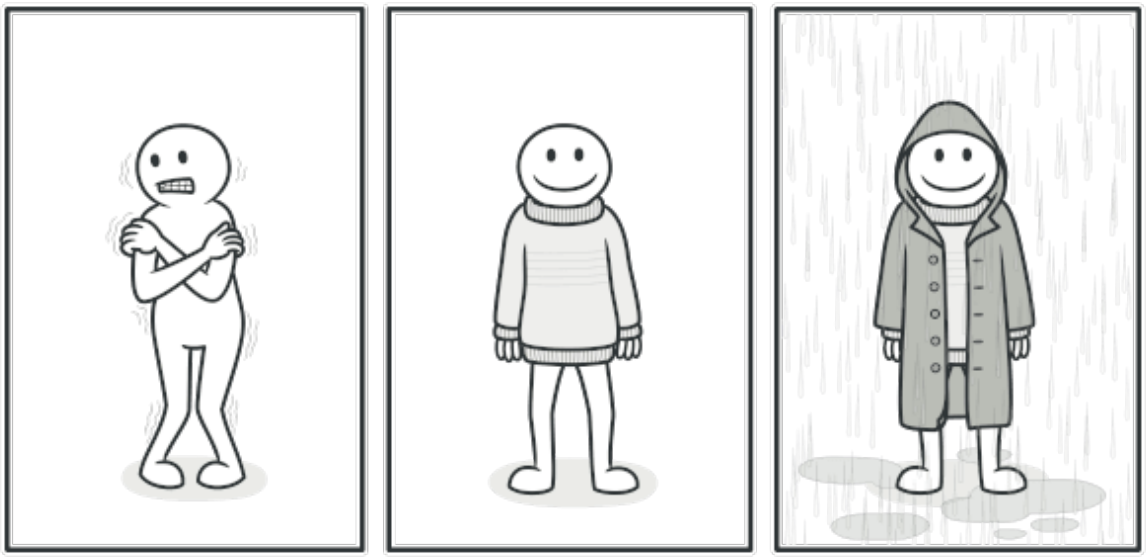
As aplicações pode configurar pilhas complexas de

notificações decoradores

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.

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

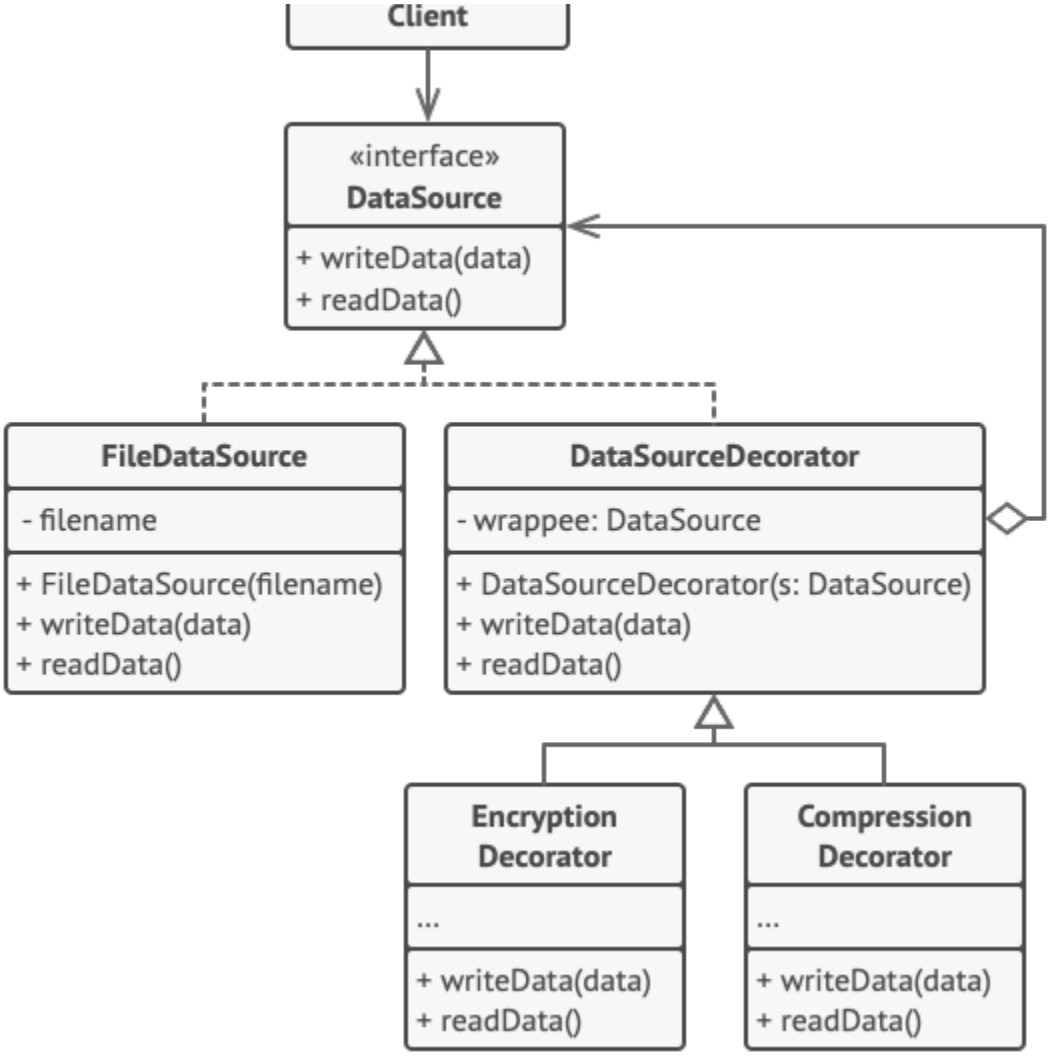
não são parte de você, e você pode facilmente remover uma peça de roupa sempre que não precisar mais dela.

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
    // 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 fornece uma interface completamente diferente para acessar um objeto existente. Por outro lado, com o padrão Decorator, a interface permanece a mesma ou é estendida. Além disso, o *Decorator* oferece suporte à composição recursiva, o que não é possível quando você usa o *Adapter*.
- Com Adapter, você acessa um objeto existente por meio de uma interface diferente. Com Proxy, a interface permanece a mesma. Com Decorator, você acessa o objeto por meio de uma interface aprimorada.
- 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.