Padrões relacionados

Freqüentemente, a ligação componente-pai é usada para o padrão Chain of Responsibility (212).

O padrão Decorator (170) é freqüentemente usado com o padrão Composite. Quando decoradores e composições são usados juntos, eles têm normalmente uma classe-mãe comum. Assim, decoradores terão que suportar a interface de Component com operações como Add, Remove e GetChild.

O Flyweight (187) permite compartilhar componentes, porém estes não mais podem referenciar seus pais.

O padrão Iterator (244) pode ser usado para percorrer os compostos.

O padrão Visitor (305) pode ser usado para localizar operações e comportamentos que seriam de outra forma distribuídos entre classes Composite e Leaf.

DECORATOR — estrutural de objetos

Intenção

Dinamicamente, agregar responsabilidades adicionais a um objeto. Os Decorators fornecem uma alternativa flexível ao uso de subclasses para extensão de funcionalidades.

Também Conhecido Como

Wrapper

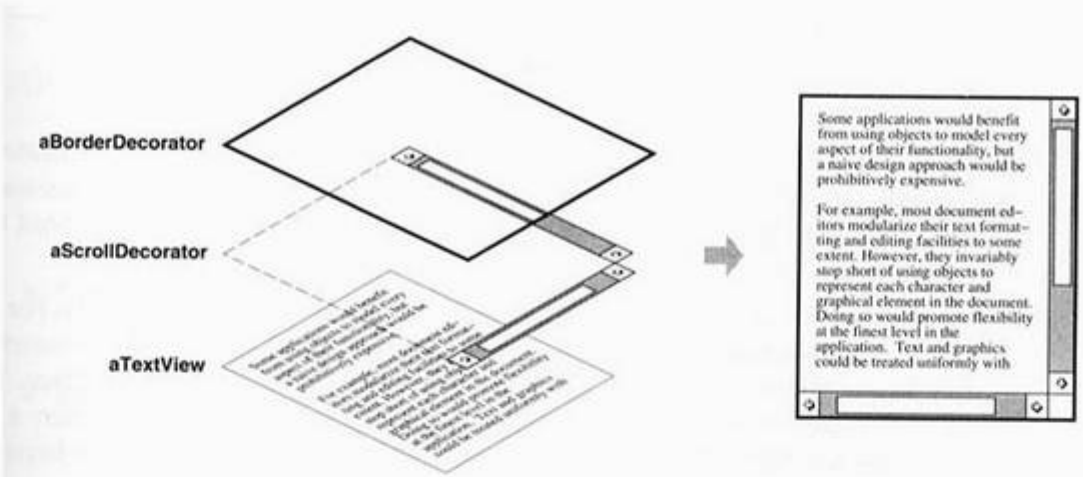
Motivação

Algumas vezes queremos acrescentar responsabidades a objetos individuais, e não a toda uma classe. Por exemplo, um *toolkit* para construção de interfaces gráficas de usuário deveria permitir a adição de propriedades, como bordas, ou comportamentos, como rolamento, para qualquer componente da interface do usuário.

Uma forma de adicionar responsabilidades é a herança. Herdar uma borda de uma outra classe coloca uma borda em volta de todas as instâncias de uma subclasse. Contudo, essa abordagem é inflexível, porque a escolha da borda é feita estaticamente. Um cliente não pode controlar como e quando decorar o componente com uma borda.

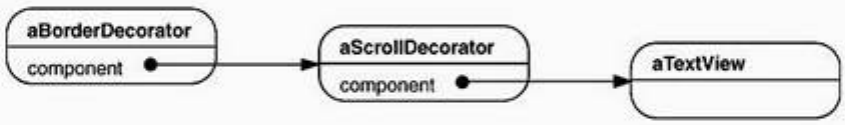
Uma abordagem mais flexível é embutir o componente em outro objeto que acrescente a borda. O objeto que envolve o primeiro é chamado de **decorator.** O decorator segue a interface do componente que decora, de modo que sua presença é transparente para os clientes do componente. O decorator repassa solicitações para o componente, podendo executar ações adicionais (tais como desenhar uma borda) antes ou depois do repasse. A transparência permite encaixar decoradores recursivamente, desta forma permitindo um número ilimitado de responsabilidades adicionais.

Suponha que tenhamos um objeto TextView que exibe texto numa janela. Como padrão, TextView não tem barras de rolamento porque nem sempre a necessitamos. Quando as necessitarmos, poderemos usar um ScrollDecorator para acrescentá-las.

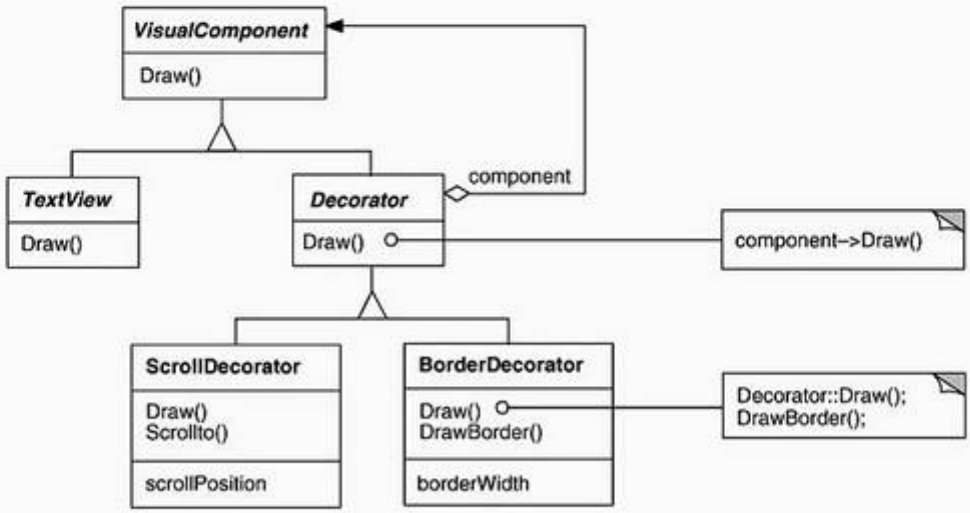


Suponha, também, que queiramos acrescentar uma borda preta espessa ao redor do objeto TextView. Também podemos usar um objeto BorderDecorator para esta finalidade. Simplesmente compomos os decoradores com TextView para produzir o resultado desejado.

O diagrama de objetos abaixo mostra como compor um objeto TextView com objetos BorderDecorator e ScrollDecorator para produzir uma visão do texto cercada por bordas e rolável:



As classes ScrollDecorator e BorderDecorator são subclasses de Decorator, uma classe abstrata destinada a componentes visuais que decoram outros componentes visuais.



VisualComponent é a classe abstrata para objetos visuais. Ela define suas interface de desenho e de tratamento de eventos. Observe como a classe Decorator simplesmente repassa as solicitações de desenho para o seu componente e como as subclasses de Decorator podem estender essa operação.

As subclasses de Decorator são livres para acrescentar operações para funcionalidades específicas. Por exemplo, a operação ScrollTo, de ScrollDecorator, permite a

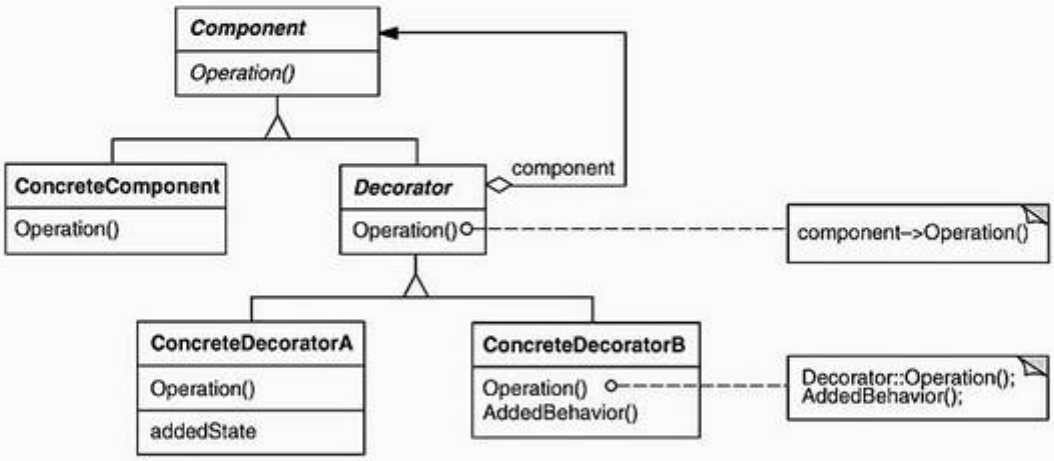
outros objetos fazerem rolamento na interface *se* eles souberem que existe um objeto ScrollDecorator na interface. O aspecto importante deste padrão é que ele permite que decoradores (*decorators*) apareçam em qualquer lugar no qual possa aparecer um VisualComponent. Desse modo, os clientes em geral não poderão distinguir entre um componente decorado e um não-decorado, e assim serão totalmente independentes das decorações.

Aplicabilidade

Use Decorator:

- para acrescentar responsabilidades a objetos individuais de forma dinâmica e transparente, ou seja, sem afetar outros objetos;
- para responsabilidades que podem ser removidas;
- quando a extensão através do uso de subclasses não é prática. Às vezes, um grande número de extensões independentes é possível e isso poderia produzir uma explosão de subclasses para suportar cada combinação. Ou a definição de uma classe pode estar oculta ou não estar disponível para a utilização de subclasses.

Estrutura



Participantes

- **Component** (VisualComponent)
  - define a interface para objetos que podem ter responsabilidades acrescentadas aos mesmos dinamicamente.
- **ConcreteComponent** (TextView)
  - define um objeto para o qual responsabilidades adicionais podem ser atribuídas.
- **Decorator**
  - mantém uma referência para um objeto Component e define uma interface que segue a interface de Component.
- **ConcreteDecorator** (BorderDecorator, ScrollDecorator)
  - acrescenta responsabilidades ao componente.

Colaborações

- Decorator repassa solicitações para o seu objeto Component. Opcionalmente, ele pode executar operações adicionais antes e depois de repassar a solicitação.

Conseqüências

O padrão Decorator tem pelo menos dois benefícios-chaves e duas deficiências:

1. *Maior flexibilidade do que a herança estática.* O padrão Decorator fornece uma maneira mais flexível de acrescentar responsabilidades a objetos do que pode ser feito com herança estática (múltipla). Com o uso de decoradores, responsabilidades podem ser acrescentadas e removidas em tempo de execução simplesmente associando-as e dissociando-as de um objeto. Em comparação, a herança requer a criação de uma nova classe para cada responsabilidade adicional (por exemplo, BorderScrollableTextView, BorderedTextView). Isso dá origem a muitas classes e aumenta a complexidade de um sistema. Além do mais, fornecer diferentes classes Decorator para uma específica classe Component permite combinar e associar (*match*) responsabilidades.
   Os Decorators também tornam fácil acrescentar uma propriedade duas vezes. Por exemplo, para dar a um TextView uma borda dupla, simplesmente associe dois BorderDecorators. Herdar de uma classe Border duas vezes é um procedimento sujeito a erros, na melhor das hipóteses.
2. *Evita classes sobrecarregadas de características na parte superior da hierarquia.* Um Decorator oferece uma abordagem do tipo "use quando for necessário" para adição de responsabilidades. Em vez de tentar suportar todas as características previsíveis em uma classe complexa e customizada, você pode definir uma classe simples e acrescentar funcionalidade de modo incremental com objetos Decorator. A funcionalidade necessária pode ser composta a partir de peças simples. Como resultado, uma aplicação não necessita incorrer no custo de características e recursos que não usa. Também é fácil definir novas espécies de Decorators independentemente das classes de objetos que eles estendem, mesmo no caso de extensões não-previstas. Estender uma classe complexa tende a expor detalhes não-relacionados com as responsabilidades que você está adicionando.
3. *Um decorador e o seu componente não são idênticos.* Um decorador funciona como um envoltório transparente. Porém, do ponto de vista da identidade de um objeto, um componente decorado não é idêntico ao próprio componente. Daí não poder depender da identidade de objetos quando você utiliza decoradores.
4. *Grande quantidade de pequenos objetos.* Um projeto que usa o Decorator freqüentemente resulta em sistemas compostos por uma grande quantidade de pequenos objetos parecidos. Os objetos diferem somente na maneira como são interconectados, e não nas suas classes ou no valor de suas variáveis. Embora esses sistemas sejam fáceis de customizar por quem os compreende, podem ser difíceis de aprender e depurar.

Implementação

Vários tópicos deveriam ser levados em conta quando da aplicação do padrão Decorator:

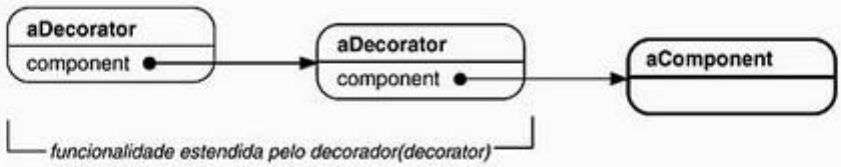
1. *Conformidade de interface.* A interface do objeto decorador deve estar em conformidade com a interface do componente que ele decora. Portanto, classes ConcreteDecorator devem herdar de uma classe comum (ao menos em C++).
2. *Omissão da classe abstrata Decorator.* Não há necessidade de definir uma classe abstrata Decorator quando você necessita acrescentar uma responsabilidade. Isso acontece freqüentemente quando você está lidando com uma hierarquia de classes existente em vez de estar projetando uma nova hierarquia. Nesse caso, pode fundir a responsabilidade de Decorator de repassar solicitações para o componente, com o ConcreteDecorator.
3. *Mantendo leves as classes Component.* Para garantir uma interface que apresente conformidade, componentes e decoradores devem descender de uma classe Component comum. É importante manter leve essa classe comum; ou seja, ela deve focalizar a definição de uma interface e não o armazenamento de dados. A definição da representação dos dados deve ser transferida para subclasses; caso contrário, a complexidade da classe Component pode tornar os decoradores muito pesados para serem usados em grande quantidade. A colocação de um volume grande de funcionalidades em Component também aumenta a probabilidade de que as subclasses concretas paguem por características e recursos de que não necessitam.
4. *Mudar o exterior de um objeto versus mudar o seu interior.* Podemos pensar em um decorador como sendo uma pele sobre um objeto que muda o seu comportamento. Uma alternativa é mudar o interior do objeto. O padrão Strategy (292) é um bom exemplo de um padrão para mudança do interior de um objeto.

   Strategies representam uma escolha melhor em situações onde a classe Component é intrinsicamente pesada, dessa forma tornando a aplicação do padrão Decorator muito onerosa. No padrão Strategy, o componente repassa parte do seu comportamento para um objeto strategy separado. O padrão Strategy permite alterar ou estender a funcionalidade do componente pela substituição do objeto strategy.

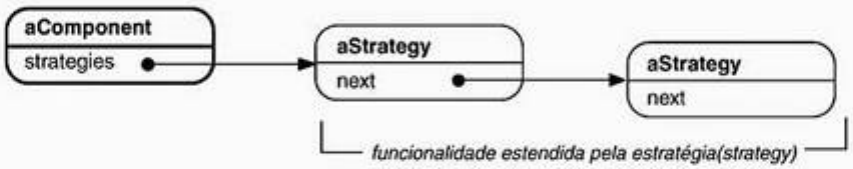
   Por exemplo, podemos suportar diferentes estilos de bordas fazendo com que o componente transfira o desenho de bordas para um objeto Border separado. O objeto Border é um objeto Strategy que encapsula uma estratégia para desenhar bordas. Através da extensão do número de estratégias de apenas uma para uma lista aberta (ilimitada) das mesmas, obtemos o mesmo efeito que quando encaixamos decoradores recursivamente.

   Por exemplo, no MacApp 3.0 [App89] e no Bedrock [Sym93a] componentes gráficos (chamados "views" (visões) mantêm uma lista de objetos adornadores (*adorners*) que podem ligar adornos adicionais, tais como bordas, a um componente visão. Se uma visão tem qualquer adorno ligado ou associado, então ela lhes dá a oportunidade de desenhar elementos adicionais. O MacApp e o Bedrock precisam usar essa abordagem porque a classe View é bastante pesada. Seria muito caro usar uma View totalmente completa somente para acrescentar uma borda.

Uma vez que o padrão Decorator somente muda o componente do ponto de vista exterior, o componente não precisa saber coisa alguma sobre seus decoradores; ou seja, os decoradores são transparentes para o componente.



Quando se usam estratégias, o próprio componente conhece as extensões possíveis. Assim, ele tem que referenciar e manter as estratégias correspondentes:



A abordagem baseada no Strategy pode exigir a modificação do componente para acomodar novas extensões. Por outro lado, uma estratégia pode ter uma interface especializada, enquanto que a interface de um decorador deve estar em conformidade com a do componente. Por exemplo, uma estratégia para implementar uma borda necessita somente definir a interface para definir uma borda (DrawBorder, GetWidth, etc.), o que significa que a estratégia pode ser leve, ainda que a classeComponent seja pesada.

O MacApp e o Bedrock utilizam essa abordagem para mais do que simplesmente adornar visões. Eles também a usam para estender o comportamento de tratamento de eventos dos objetos. Em ambos os sistemas, uma visão mantém uma lista de objetos "behavior" (comportamento), que pode modificar e interceptar eventos. A visão dá a cada um dos objetos comportamento registrados uma chance de manipular o evento antes de comportamentos não-registrados, na prática redefinindo-os. Você pode decorar uma visão com suporte especial para tratamento do teclado, por exemplo, registrando um objeto comportamento que intercepta e trata eventos de teclas.

Exemplo de código

O código a seguir mostra como implementar decoradores de interfaces de usuário em C++. Assumiremos que existe uma classe Component chamada `VisualComponent`.

```
class VisualComponent {
public:
    VisualComponent();

    virtual void Draw();
    virtual void Resize();
    // ...
};
```

Definiremos uma subclasse de `VisualComponent` chamada `Decorator`, para a qual introduziremos subclasses para obter diferentes decorações.

```
class Decorator : public VisualComponent {
public:
    Decorator(VisualComponent*);

    virtual void Draw();
    virtual void Resize();
    // ...
private:
    VisualComponent* _component;
};
```

`Decorator` decora o `VisualComponent` referenciado pela variável de instância `_component`, a qual é iniciada no constructor. Para cada operação na interface de `VisualComponent`, `Decorator` define uma implementação-padrão que passa a solicitação para `_component`:

```
void Decorator::Draw () {
    _component->Draw();
}

void Decorator::Resize () {
    _component->Resize();
}
```

As subclasses de `Decorator` definem decorações específicas. Por exemplo, a classe `BorderDecorator` acrescenta uma borda ao seu componente circunscrito. `BorderDecorator` é uma subclasse de `Decorator` que redefine a operação `Draw` para desenhar a borda. `BorderDecorator` também define uma operação de ajuda privada `DrawBorder`, que executa o desenho. A subclasse herda todas as outras implementações de operações de `Decorator`.

```
class BorderDecorator : public Decorator {
public:
    BorderDecorator(VisualComponent*, int borderWidth);

    virtual void Draw();
private:
    void DrawBorder(int);
private:
    int _width;
};

void BorderDecorator::Draw () {
    Decorator::Draw();
    DrawBorder(_width);
}
```

Uma implementação similar seria adotada para `ScrollDecorator` e `DropShadowDecorator`, a qual adicionaria recursos de rolamento e sombreamento a um componente visual. Agora podemos compor instâncias destas classes para oferecer

diferentes decorações. O código a seguir ilustra como podemos usar decoradores para criar um `TextView` rolável com bordas.

Em primeiro lugar, temos que colocar um componente visual num objeto janela. Assumiremos que a nossa classe `Window` oferece uma operação `SetContents` com esta finalidade:

```
void Window::SetContents (VisualComponent* contents) {
    // ...
}
```

Agora podemos criar a visão do texto e uma janela para colocá-la:

```
Window* window = new Window;
TextView* textView = new TextView;
```

`TextView` é um `VisualComponent` que nos permite colocá-lo na janela:

```
window->SetContents(textView);
```

Mas queremos uma `TextView` rolável com bordas. Assim, nós a decoramos de maneira apropriada antes de colocá-la na janela.

```
window->SetContents(
    new BorderDecorator(
        new ScrollDecorator(textView), 1
    )
);
```

Uma vez que `Window` acessa os seus conteúdos através da interface de `VisualComponent`, ele não é ciente da presença do decorador. Você, sendo cliente, pode ainda acompanhar a visão do texto se tiver que interagir diretamente com ela, por exemplo, quando precisar invocar operações que não são parte da interface de `VisualComponent`. Clientes que dependem da identidade do componente também poderiam referenciá-lo diretamente.

Usos conhecidos

Muitos *toolkits* orientados a objetos para construção de interfaces de usuário utilizam decoradores para acrescentar adornos gráficos a widgets. Os exemplos incluem o InterViews [LVC89, LCI+92], a ET++ [WGM88] e a biblioteca de classes ObjectWorks/Smalltalk [Par90]. Aplicações mais exóticas do Decorator são o DebuggingGlyph, do InterViews, e o PassivityWrapper, do ParcPlace Smalltalk. Um DebbuggingGlyph imprime informação para depuração (*debbugging*) antes e depois que ele repassa uma solicitação de *layout* para o seu componente. Esta informação para rastreamento pode ser usada para analisar e depurar o comportamento de *layout* de objetos participantes numa composição complexa.

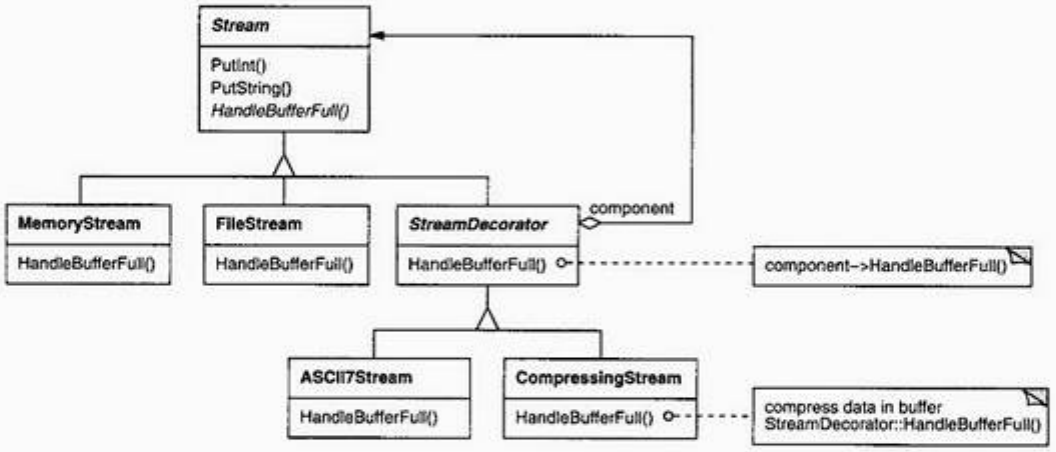
O PassivityWrapper pode habilitar ou desabilitar interações do usuário com o componente.

Porém, o padrão Decorator não tem seu uso limitado a interfaces gráficas do usuário, como ilustra o exemplo a seguir, baseado nas classes *streaming* da ET++[WGM88].

Streams são uma abstração fundamental em muitos recursos de programas para entrada e saída. Um *stream* pode fornecer uma interface para conversão de objetos em uma seqüência de bytes ou caracteres. Isso nos permite transcrever um objeto para um arquivo, ou para um *string* na memória, para posterior recuperação. Uma maneira simples e direta de fazer isso é definir uma classe abstrata Stream com as subclasses MemoryStream e FileStream. Mas suponha que também queiramos ser capazes de fazer o seguinte:

- Comprimir os lados do *stream* usando diferentes algoritmos de compressão (runlength encoding, Lempel-Ziv, etc.).
- Reduzir os dados do *stream* a caracteres ASCII de sete bits, de maneira que possa ser transmitido por um canal de comunicação ASCII.

O padrão Decorator fornece uma maneira elegante de acrescentar estas responsabilidades a *streams*. O diagrama abaixo mostra uma solução para o problema:



A classe abstrata Stream mantém um *buffer* interno e fornece operações para armazenar dados no *stream* (PutInt, PutString). Sempre que o *buffer* fica cheio, Stream chama a operação abstrata HandleBufferFull, que faz a transferência efetiva dos dados. A versão FileStream desta operação redefine a mesma para transferir o buffer para um arquivo.

Aqui a classe chave é StreamDecorator, a qual mantém uma referência para um componente *stream* e repassa solicitações para o mesmo. As subclasses de StreamDecorator substituem HandleBufferFull e executam ações adicionais antes de chamar a operação HandleBufferFull de StreamDecorator.

Por exemplo, a subclasse CompressingStream comprime os dados, e ASCII7Stream converte os dados para ASCII de sete bits. Agora, para criar uma FileStream que comprime os seus dados *e* converte os dados binários comprimidos para ASCII de sete bits, nós decoramos um FileStream com um CompressingStream e um ASCII7Stream:

```
Stream* aStream = new CompressingStream(
    new ASCII7Stream(
        new FileStream("aFileName")
    )
);
aStream->PutInt(12);
aStream->PutString("aString");
```

Padrões relacionados

Adapter (140): Um padrão Decorator é diferente de um padrão Adapter no sentido de que um Decorator somente muda as responsabilidades de um objeto, não a sua interface; já um Adapter dará a um objeto uma interface completamente nova.

Composite (160): Um padrão Decorator pode ser visto como um padrão Composite degenerado com somente um componente. Contudo, um Decorator acrescenta responsabilidades adicionais – ele não se destina a agregação de objetos.

Strategy (292): Um padrão Decorator permite mudar a superfície de um objeto, um padrão Strategy permite mudar o seu interior. Portanto, essas são duas maneiras alternativas de mudar um objeto.

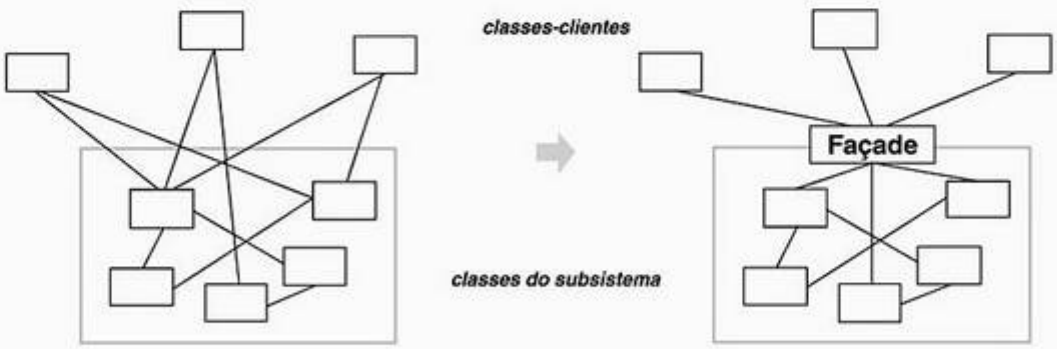
FAÇADE — estrutural de objetos

Intenção

Fornecer uma interface unificada para um conjunto de interfaces em um subsistema. Façade define uma interface de nível mais alto que torna o subsistema mais fácil de ser usado.

Motivação

Estruturar um sistema em subsistemas ajuda a reduzir a complexidade. Um objetivo comum de todos os projetos é minimizar a comunicação e as dependências entre subsistemas. Uma maneira de atingir esse objetivo é introduzir um objeto **façade** (fachada), o qual fornece uma interface única e simplificada para os recursos e facilidades mais gerais de um subsistema.



Considere, por exemplo, um ambiente de programação que fornece acesso às aplicações para o seu subsistema compilador. Esse subsistema contém classes, tais como Scanner, Parser, ProgramNode, BytecodeStream e ProgramNodeBuilder, que implementam o compilador. Algumas aplicações especializadas podem precisar acessar essas classes diretamente. Mas a maioria dos clientes de um compilador geralmente não se preocupa com detalhes tais como análise e geração de código; eles apenas querem compilar seu código. Para eles, as interfaces poderosas, porém de baixo nível, do subsistema compilador somente complicam sua tarefa.