

# ENGENHARIA DE SOFTWARE

Padrões de projeto

# O QUE É UM PADRÃO DE PROJETO?

Um padrão de projeto descreve um problema e o cerne da sua solução (em OO), de forma que possa ser reusado, fazendo os ajustes necessários para cada caso.

# ELEMENTOS ESSENCIAIS DE UM PADRÃO DE PROJETO

**1 - Nome do padrão**

**2 - Problema**

**3 - Solução**

**4 - Consequências**

# DESCRIÇÃO DE UM PADRÃO DE PROJETO

1- Nome e classificação

2- Intenção e objetivo

3- Nomes similares

4- Motivação

5- Aplicabilidade

6- Estrutura

7- Participantes

8- Colaborações

9- Consequências

10- Implementação

11- Exemplo de código

12- Usos conhecidos

13- Padrões  
relacionados

# CATÁLOGO DE PADRÕES DE PROJETO

		Propósito		
		De criação	Estrutural	Comportamental
Escopo	Classe	Factory Method (112)	Adapter (class) (140)	Interpreter (231) Template Method (301)
	Objeto	Abstract factory (95) Builder (104) Prototype (121) Singleton (130)	Adapter Bridge (151) Composite (160) Decorator (170) Façade (179) Flyweight (187) Proxy (198)	Chain of Responsibility (212) Command (222) Iterator (244) Mediator (257) Memento (266) Observer (274) State (284) Strategy (292) Visitor (305)

# TIPOS/NÍVEIS DE PADRÃO DE PROJETO

## POR ESCOPO:

### Classe:

- Relacionamento entre classes e suas subclasses (herança).
- Estáticos, em tempo de compilação.

### Objeto:

- Relacionamentos entre objetos (potencialmente herança).
- Dinâmicos, em tempo de execução.

# TIPOS/NÍVEIS DE PADRÃO DE PROJETO

## POR PROPÓSITO (FINALIDADE):

- De criação
- Estruturais (composição de classes/objetos)
- Comportamentais (interação entre classes/objetos)

# EXEMPLO: PADRÃO ADAPTER

## Intenção

Converter a interface de uma classe em outra interface, esperada pelos clientes. O Adapter permite que classes com interfaces incompatíveis trabalhem em conjunto – o que, de outra forma, seria impossível.

Também conhecida como WRAPPER

# EXEMPLO: PADRÃO ADAPTER

Algumas vezes, uma classe de um *toolkit*, projetada para ser reutilizada não é reutilizável porque sua interface não corresponde à interface específica de um domínio requerida por uma aplicação.

Considere, por exemplo, um editor de desenhos que permite aos usuários desenhar e arranjar elementos gráficos (linhas, polígonos, texto, etc.) em figuras e diagramas. A abstração-chave do editor de desenhos é o objeto gráfico, o qual tem uma forma editável e pode desenhar a si próprio. A interface para objetos gráficos é definida por uma classe abstrata chamada *Shape*. O editor define uma subclasse de *Shape* para cada tipo de objeto gráfico: uma classe *LineShape* para linhas, uma classe *PolygonShape* para polígonos, e assim por diante.

Classes para formas geométricas elementares, como *LineShape* e *PolygonShape*, são bastante fáceis de ser implementadas porque as suas capacidades de desenho e edição são inherentemente limitadas. Mas uma subclasse *TextShape* que pode exibir e editar textos é mais difícil de ser implementada, uma vez que mesmo a edição básica de textos envolve atualizações complicadas de tela e gerência de *buffer*. Entretanto, pode já existir um *toolkit* para construção de interfaces de usuários, o qual já oferece uma sofisticada classe *TextView* para a exibição e edição de textos. Idealmente, gostaríamos de reutilizar *TextView* para implementar *TextShape*, porém, o *toolkit* não foi projetado levando classes *Shape* em consideração. Assim, não podemos usar de maneira intercambiável objetos *TextView* e *Shape*.

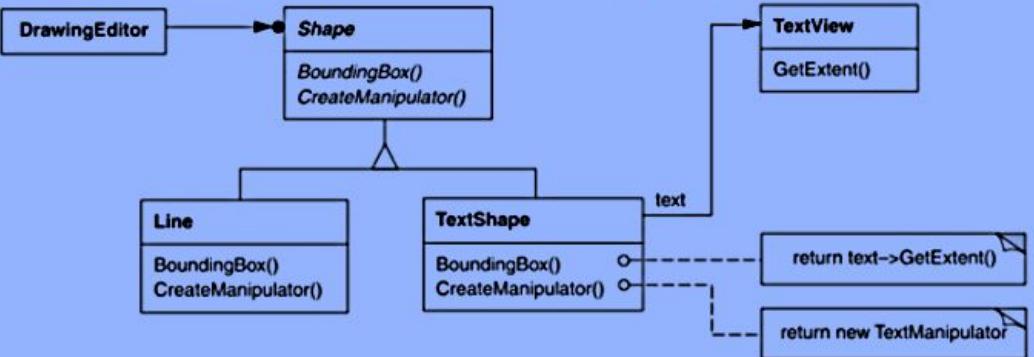
Como é possível que classes existentes e não-relacionadas, como *TextView*, funcionem em uma aplicação que espera classes com uma interface diferente e incompatível? Poderíamos mudar a classe *TextView* de maneira que ela fosse coerente com a interface de *Shape*, porém, a menos que tenhamos o código-fonte do *toolkit*, essa opção não é viável. Mesmo que tivéssemos o código-fonte, não teria sentido mudar *TextView*; o *toolkit* não deveria ter que adotar interfaces específicas de domínios somente para fazer com que uma aplicação funcione.

Em vez disso, poderíamos definir *TextShape* de maneira que ele *adapte* a interface de *TextView* àquela de *Shape*. Podemos fazer isto de duas maneiras: (1) herdando a interface de *Shape* e a implementação de *TextView*, ou (2) compondo uma instância de *TextView* dentro de uma *TextShape* e implementando *TextShape* em termos da interface de *TextView*.

## Motivação

# EXEMPLO: PADRÃO ADAPTER

Essas duas abordagens correspondem às versões do *padrão Adapter* para classes e para objetos. Chamamos TextShape um adaptador.



Este diagrama ilustra o caso de um adaptador para objetos. Ele mostra como solicitações de BoundingBox, declarada na classe Shape, são convertidas em solicitações para GetExtent, definida em TextView. Uma vez que TextShape adapta TextView à interface de Shape, o editor de desenhos pode reutilizar a classe TextView que seria incompatível de outra forma.

Freqüentemente, um adaptador é responsável por funcionalidades não oferecidas pela classe adaptada. O diagrama mostra como um adaptador pode atender tais responsabilidades. O usuário deveria ser capaz de “arrastar” cada objeto Shape para uma nova posição de forma interativa, porém, TextView não está projetada para fazer isso. TextShape pode acrescentar essa função através da implementação da operação CreateManipulator, de Shape, a qual retorna uma instância da subclasse Manipulator apropriada.

Manipulator é uma classe abstrata para objetos que sabem como animar um Shape em resposta à entrada de usuário, tal como arrastar a forma geométrica para uma nova localização. Existem subclasses de Manipulator para diferentes formas; por exemplo, TextManipulator é a subclasse correspondente para TextShape. Pelo retorno de uma instância de TextManipulator, TextShape acrescenta a funcionalidade de que Shape necessita mas que TextView não tem.

## Motivação

# EXEMPLO: PADRÃO ADAPTER

## Motivação

Algumas vezes, uma classe de um *toolkit*, projetada para ser reutilizada não é reutilizável porque sua interface não corresponde à interface específica de um domínio requerida por uma aplicação.

# EXEMPLO: PADRÃO ADAPTER

## Aplicabilidade

□ Use o padrão Adapter quando:

- Você quiser usar uma classe existente, mas sua interface não corresponder à interface de que necessita;
- Você quiser criar uma classe reutilizável que coopere com classes não relacionadas ou não previstas, ou seja, classes que não necessariamente tenham interfaces compatíveis;

# EXEMPLO: PADRÃO ADAPTER

## Aplicabilidade

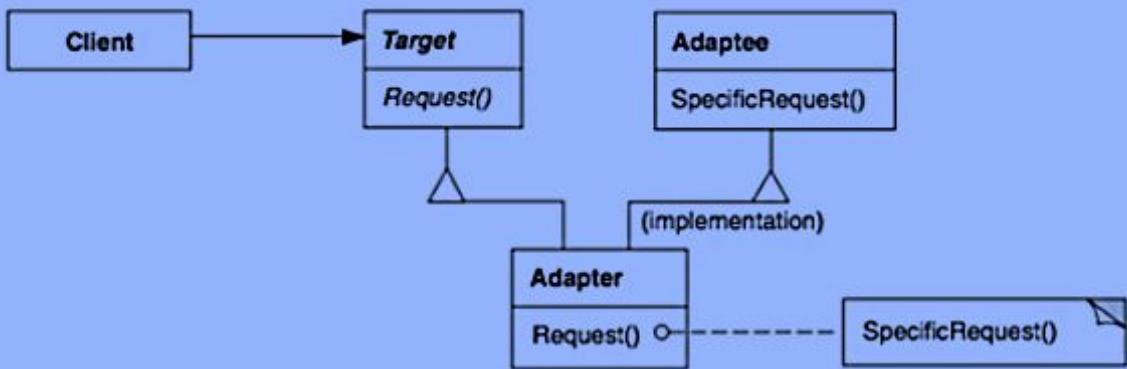
□ Use o padrão Adapter quando:

- (somente para adaptadores de objetos) Você precisar usar várias subclasses existentes, porém, for impraticável adaptar essas interfaces criando subclasses para cada uma. Um adaptador de objetos pode adaptar a interface da sua classe-mãe.

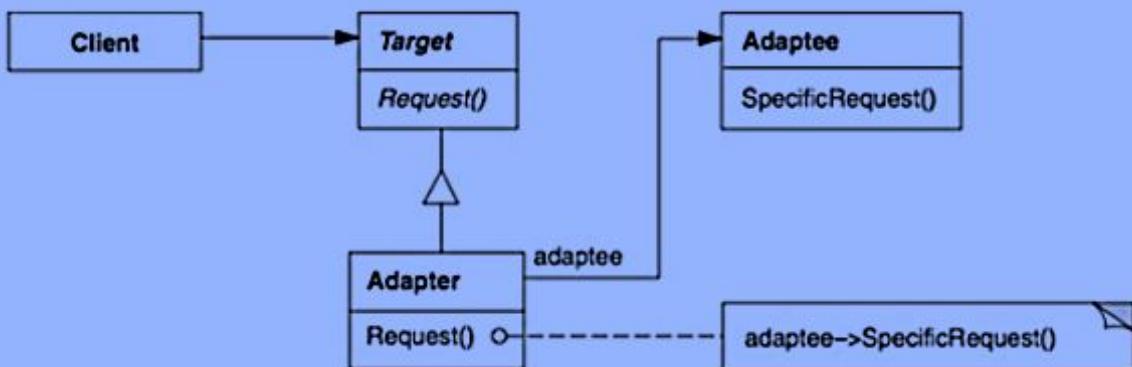
# EXEMPLO: PADRÃO ADAPTER

## Estrutura

Um adaptador de classe usa a herança múltipla para adaptar uma interface à outra:



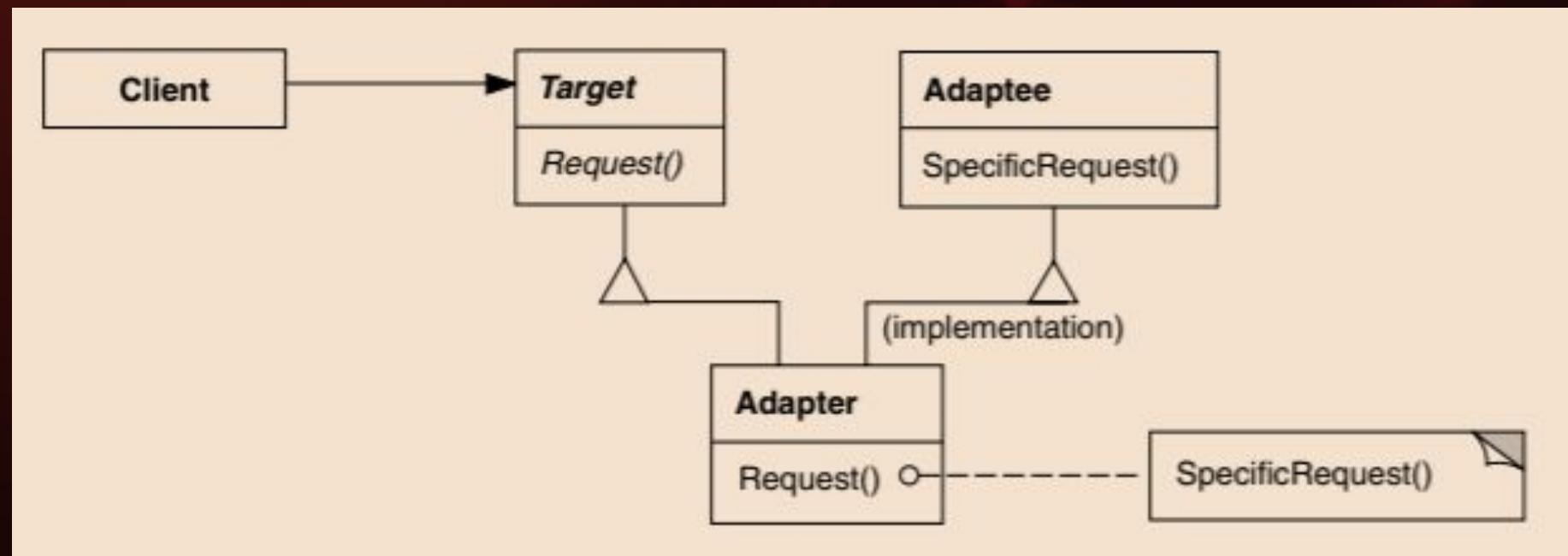
Um adaptador de objeto depende da composição de objetos:



# EXEMPLO: PADRÃO ADAPTER

## Estrutura

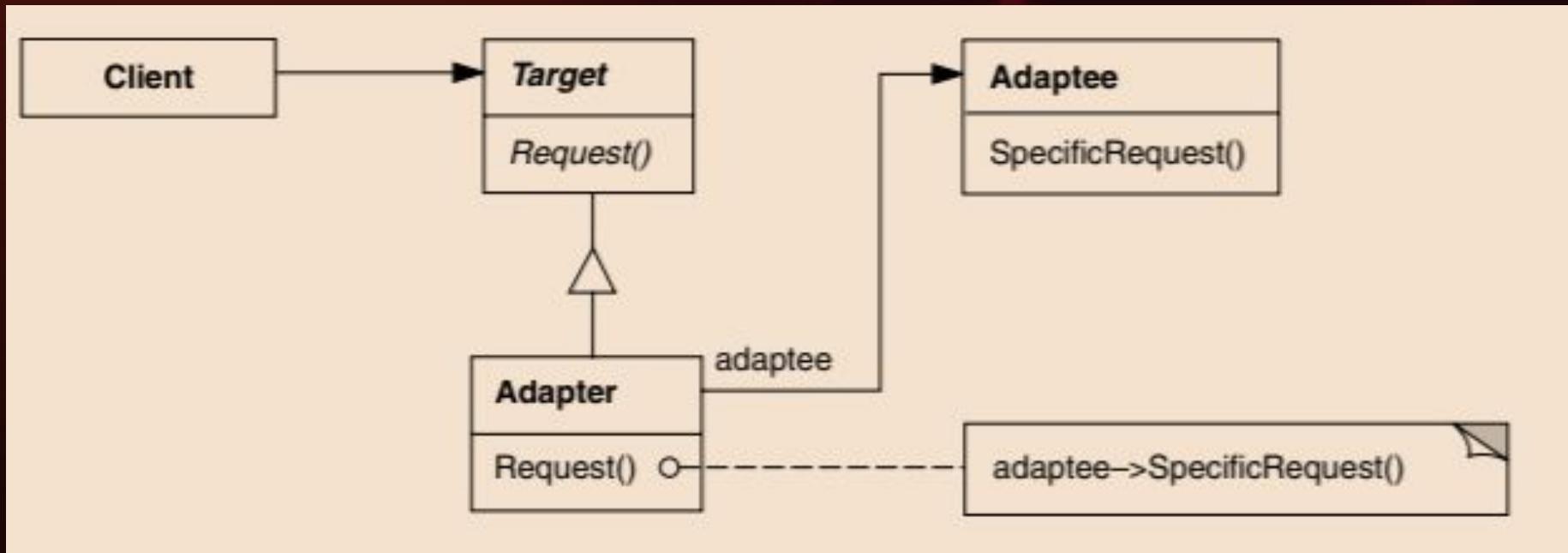
Um adaptador de classe usa a herança múltipla para adaptar uma interface à outra:



# EXEMPLO: PADRÃO ADAPTER

## Estrutura

Um adaptador de objeto depende da composição de objetos:



# EXEMPLO: PADRÃO ADAPTER

## Participantes

- **Target (Shape)**: Define a interface específica do domínio que Client usa.
- **Client (DrawingEditor)**: Colabora com objetos compatíveis com a interface de Target.
- **Adaptee (TextView)**: Define uma interface existente que necessita ser adaptada.
- **Adapter (TextShape)**: Adapta a interface do Adapter à interface de Target.

# EXEMPLO: PADRÃO ADAPTER

## Colaborações

- Os clientes chamam operações em uma instância de Adapter. Por sua vez, o adapter chama operações de Adaptee que executam a solicitação.

# EXEMPLO: PADRÃO ADAPTER

Os adaptadores de classes e de objetos têm diferentes soluções de compromisso. Um adaptador de classe:

- adapta Adaptee a Target através do uso efetivo de uma classe Adapter concreta. Em consequência, um adaptador de classe não funcionará quando quisermos adaptar uma classe e todas as suas subclasses;
- permite a Adapter substituir algum comportamento do Adaptee, uma vez que Adapter é uma subclasse de Adaptee;
- introduz somente um objeto, e não é necessário endereçamento indireto adicional por ponteiros para chegar até o Adaptee.

Um adaptador de objeto:

- permite a um único Adapter trabalhar com muitos Adaptees – isto é, o Adaptee em si e todas as suas subclasses (se existirem). O Adapter também pode acrescentar funcionalidade a todos os Adaptees de uma só vez;
- torna mais difícil redefinir um comportamento de Adaptee. Ele exigirá a criação de subclasses de Adaptee e fará com que Adapter refencie a subclasse ao invés do Adaptee em si.

## Consequências

# EXEMPLO: PADRÃO ADAPTER

## Consequências

Aqui apresentamos outros pontos a serem considerados quando usamos o padrão Adapter:

1. *Quanta adaptação Adapter faz?* Os Adapters variam no volume de trabalho que executam para adaptar o Adaptee à interface de Target. Existe uma variação do trabalho possível, desde a simples conversão de interface – por exemplo, mudar os nomes das operações – até suportar um conjunto de operações inteiramente diferente. O volume de trabalho que o Adapter executa depende de quanto similar é a interface de Target a dos seus Adaptees.
2. *Adaptadores conectáveis (pluggable)*. Uma classe é mais reutilizável quando você minimiza as suposições que outras classes devem fazer para utilizá-la. Através da construção da adaptação de interface em uma classe, você elimina a suposição de que outras classes vêm a mesma interface. Dito de outra maneira, a adaptação de interfaces permite incorporar a nossa classe a sistemas existentes que podem estar esperando interfaces diferentes para a classe. ObjectWorks/Smalltalk [Par90] usa o termo *pluggable adapter* para descrever classes com adaptação de interfaces incorporadas.

Considere um widget TreeDisplay que pode exibir graficamente estruturas de árvore. Se este fosse um widget com uma finalidade especial para uso em apenas uma aplicação, então, poderíamos requerer uma interface específica dos objetos que ele exibisse; ou seja, todos deveriam descender de uma classe abstrata Tree. Mas se quiséssemos tornar TreeDisplay mais reutilizável (digamos que quiséssemos torná-la parte de um toolkit de widgets úteis), então essa exigência não seria razoável. Aplicações definirão suas próprias classes para estruturas de árvore. Elas não deveriam ser forçadas a usar a nossa classe abstrata Tree. Diferentes estruturas de árvores terão diferentes interfaces.

Por exemplo, numa hierarquia de diretório, os descendentes podem ser acessados com uma operação GetSubdirectories, enquanto que numa hierarquia de herança, a operação correspondente poderia ser chamada GetSubclasses. Um widget TreeDisplay reutilizável deve ser capaz de exibir ambos os tipos de hierarquias ainda que usem interfaces diferentes. Em outras palavras, TreeDisplay deveria ter uma adaptação de interface incorporada a ele.

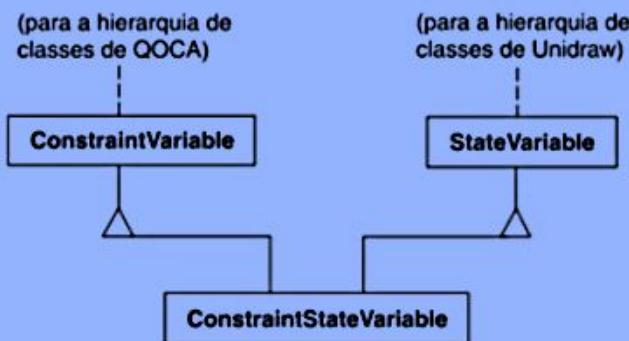
Examinaremos diferentes maneiras de construir adaptações de interfaces, dentro de classes, na seção Implementação.

# EXEMPLO: PADRÃO ADAPTER

## Consequências

3. Utilização de adaptadores de dois sentidos para fornecer transparência. Um problema potencial com adaptadores decorre do fato de que eles não são transparentes para todos os clientes. Um objeto adaptado não oferece a interface do objeto original, por isso ele não pode ser usado onde o original o for. Adaptadores de dois sentidos (*two-way adapters*) podem fornecer essa transparência. Eles são úteis quando dois clientes diferentes necessitam ver um objeto de forma diferente.

Considere o adaptador de dois sentidos que integra o Unidraw, um *framework* para editores gráficos [VL90], e QOCA, um toolkit para solução de restrições [HHMV92]. Ambos os sistemas possuem classes que representam variáveis explicitamente: Unidraw tem StateVariable e QOCA tem Constraint Variable. Para fazer com que Unidraw trabalhe com QOCA, Constraint Variable deve ser adaptada a StateVariable; para permitir que QOCA propague soluções para Unidraw, StateVariable deve ser adaptada a ConstraintVariable.



A solução envolve o uso de um adaptador de classe ConstraintStateVariable de dois sentidos, uma subclasse tanto de StateVariable como de ConstraintVariable que adapta as duas interfaces uma à outra. Neste caso, a herança múltipla é uma solução viável porque as interfaces das classes adaptadas são substancialmente diferentes. O adaptador de classe de dois sentidos é compatível com ambas as classes adaptadas, podendo funcionar em ambos os sistemas.

# EXEMPLO: PADRÃO ADAPTER

Embora a implementação do padrão Adapter seja normalmente simples e direta, apresentamos aqui alguns tópicos a serem sempre considerados:

1. *Implementando adaptadores de classe em C++.* Numa implementação em C++ uma classe Adapter deveria herdar publicamente de Target e privadamente de Adaptee. Assim, Adapter seria um subtipo de Target, mas não de Adaptee.
2. *Adaptadores conectáveis.* Vamos examinar três maneiras de implementar adaptadores "plugáveis" para o widget TreeDisplay descrito anteriormente, os quais podem formatar e exibir uma estrutura hierárquica automaticamente. O primeiro passo, comum a todas as três implementações discutidas aqui, é encontrar uma interface "mínima" para Adaptee, ou seja, o menor subconjunto de operações que permite fazer a adaptação. Uma interface mínima, consistindo em somente um par de operações, é mais fácil de adaptar que uma interface com dúzias de operações. Para o TreeDisplay, o adaptee é qualquer estrutura hierárquica. Uma interface minimalista pode incluir duas operações, uma que define como apresentar graficamente um nó na estrutura hierárquica e outra que recupera os filhos do nó.

A interface mínima conduz a três abordagens de implementação:

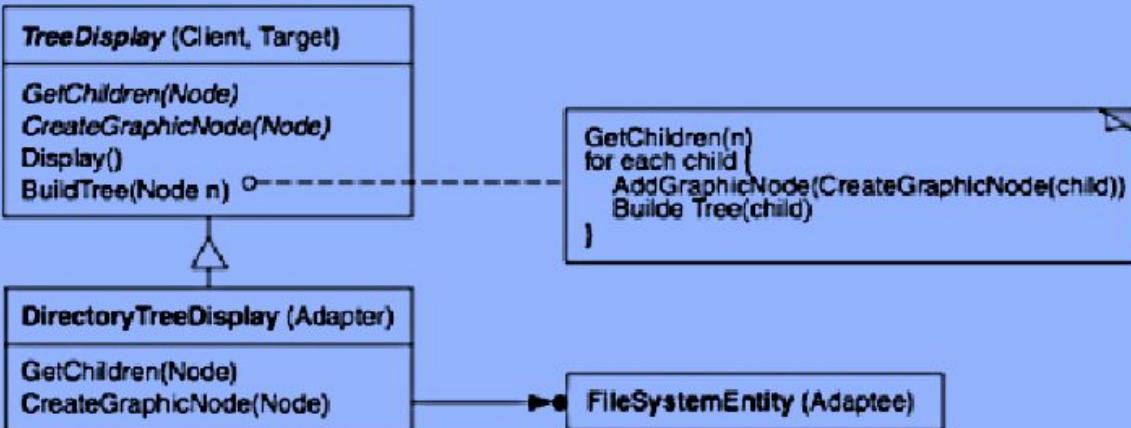
- (a) *Utilizando operações abstratas.*

Defina na classe TreeDisplay operações abstratas correspondentes à interface mínima de Adaptee. As operações abstratas devem ser implementadas por subclasses que também adaptam o objeto hierarquicamente estruturado. Por exemplo, uma subclass

## Implementação

# EXEMPLO: PADRÃO ADAPTER

DirectoryTreeDisplay implementará essas operações acessando a estrutura do diretório.



TreeDisplay armazena um bloco para converter um nó em um GraphicNode e outro bloco para acessar os filhos de um nó.

Por exemplo, para criar um TreeDisplay em uma hierarquia de diretório, escrevemos

```
directoryDisplay :=  
    (TreeDisplay on: treeRoot)  
    getChildrenBlock:  
        [:node | node getSubdirectories]  
    createGraphicNodeBlock:  
        [:node | node createGraphicNode].
```

Se você está construindo uma adaptação de interface em uma classe, essa abordagem oferece uma alternativa conveniente ao uso de subclasses.

## Implementação

# EXEMPLO: PADRÃO ADAPTER

Apresentaremos um breve esboço da implementação de adaptadores de classes e de objetos para o exemplo apresentado na seção Motivação, começando com as classes `Shape` e `TextView`.

```
class Shape {
public:
    Shape();
    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual Manipulator* CreateManipulator() const;
};

class TextView {
public:
    TextView();
    void GetOrigin(Coord& x, Coord& y) const;
    void GetExtent(Coord& width, Coord& height) const;
    virtual bool IsEmpty() const;
};
```

`Shape` supõe uma caixa delimitadora definida pelos seus cantos opostos. Em contraste, `TextView` é definido por origem, altura e largura. `Shape` também define uma operação `CreateManipulator` para criar um objeto `Manipulator`, o qual sabe como animar uma forma quando um usuário a manipula.<sup>1</sup> O `TextView` não tem uma operação equivalente. A classe `TextShape` é um adaptador entre essas diferentes interfaces.

Um adaptador de classe utiliza a herança múltipla para adaptar interfaces. O ponto-chave dos adaptadores de classe é a utilização de um ramo de herança para herdar a interface e de outro ramo para herdar a implementação. A maneira usual de fazer essa distinção em C++ é herdar a interface publicamente e herdar a implementação privadamente. Usaremos essa convenção para definir o adaptador `TextShape`.

A operação `BoundingBox` converte a interface de `TextView` para que esta fique de acordo com a interface de `Shape`.

```
class TextShape : public Shape, private TextView {
public:
    TextShape();
    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
};
```

```
void TextShape::BoundingBox (
    Point& bottomLeft, Point& topRight
) const {
    Coord bottom, left, width, height;

    GetOrigin(bottom, left);
    GetExtent(width, height);

    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}
```

A operação `IsEmpty` demonstra o repasse direto de solicitações, comum em implementações de adaptadores:

```
bool TextShape::IsEmpty () const {
    return TextView::IsEmpty();
}
```

Finalmente, definimos `CreateManipulator` (a qual não é suportada por `TextView`) a partir do zero. Assumimos que já implementamos uma classe `TextManipulator` que suporta manipulação de um `TextShape`.

```
Manipulator* TextShape::CreateManipulator () const {
    return new TextManipulator(this);
}
```

O adaptador de objeto utiliza composição de objetos para combinar classes que têm interfaces diferentes. Nesta abordagem, o adaptador `TextShape` mantém um apontador para `TextView`.

```
class TextShape : public Shape {
public:
    TextShape(TextView*);

    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
private:
    TextView* _text;
};
```

## Exemplo de código

# EXEMPLO: PADRÃO ADAPTER

TextShape deve iniciar o apontador para instância de TextView, e ele faz isso no constructor. Ele também deve invocar operações no seu objeto TextView sempre que suas próprias operações forem invocadas. Neste exemplo, presume que o cliente crie o objeto TextView e o passe para o constructor de TextShape:

```
TextShape::TextShape (TextView* t) {
    _text = t;
}

void TextShape::BoundingBox {
    Point& bottomLeft, Point& topRight
} const {
    Coord bottom, left, width, height;

    _text->GetOrigin(bottom, left);
    _text->GetExtent(width, height);

    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}

bool TextShape::IsEmpty () const {
    return _text->IsEmpty();
}
```

A implementação de CreateManipulator não muda em relação à versão para o adaptador de classe, uma vez que é implementada do zero e não reutiliza qualquer funcionalidade existente de TextView.

```
Manipulator* TextShape::CreateManipulator () const {
    return new TextManipulator(this);
}
```

Compare este código com o código do caso do adaptador de classe. O adaptador de objeto exige um pouco mais de esforço para escrever, porém, é mais flexível. Por exemplo, a versão do adaptador de objeto de TextShape funcionará igualmente bem com subclasses de TextView – o cliente simplesmente passa uma instância de uma subclasse de TextView para o constructor de TextShape.

## Exemplo de código

# EXEMPLO: PADRÃO ADAPTER

## Usos conhecidos

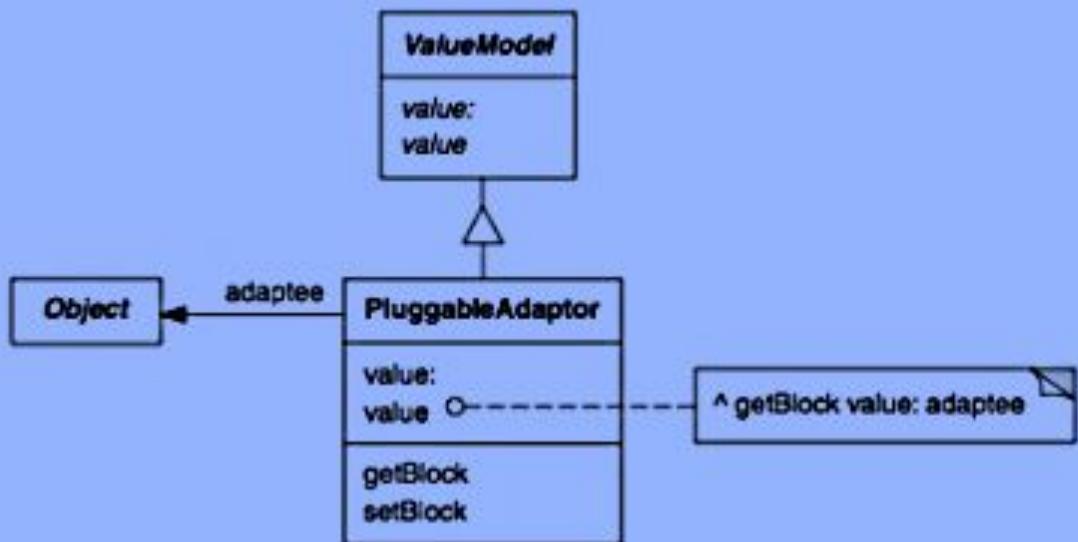
Outro exemplo proveniente de ObjectWorks\Smalltalk é a classe TableAdaptor. Uma TableAdaptor pode adaptar uma seqüência de objetos numa apresentação tabular. A tabela exibe um objeto por linha. O cliente parametriza TableAdaptor com um conjunto de mensagens que uma tabela pode usar para obter os valores de coluna de um objeto.

Algumas classes no AppKit do NeXT [Add94] usam objetos delegados para executar adaptação de interfaces. Um exemplo é a classe NXBrowser que pode exibir listas hierárquicas de dados. NXBrowser usa um objeto delegado para acessar e adaptar dados.

O “Casamento de Conveniência” (“*Marriage of Convenience*”), [Mey88] de Meyer é uma forma de um adaptador de classe. Meyer descreve como uma classe FixedStack adapta a implementação de uma classe Array à interface de uma classe stack. O resultado é uma pilha (Stack) que contém um número fixo de entradas.

# EXEMPLO: PADRÃO ADAPTER

Em vez disso, ObjectWorks\Smalltalk inclui uma subclasse de ValueModel chamada PluggableAdaptor. Um objeto PluggableAdaptor adapta outros objetos à interface de ValueModel (value, value:). Ela pode ser parametrizada com blocos para obter e atualizar (set) o valor desejado. PluggableAdaptor utiliza internamente estes blocos para implementar a interface value, value:. PluggableAdaptor também permite passar nomes no Selector (por exemplo, width, width:) diretamente por conveniência sintática. Ela converte estes seletores (selectors) nos blocos correspondentes, de forma automática.



## Usos conhecidos

# EXEMPLO: PADRÃO ADAPTER

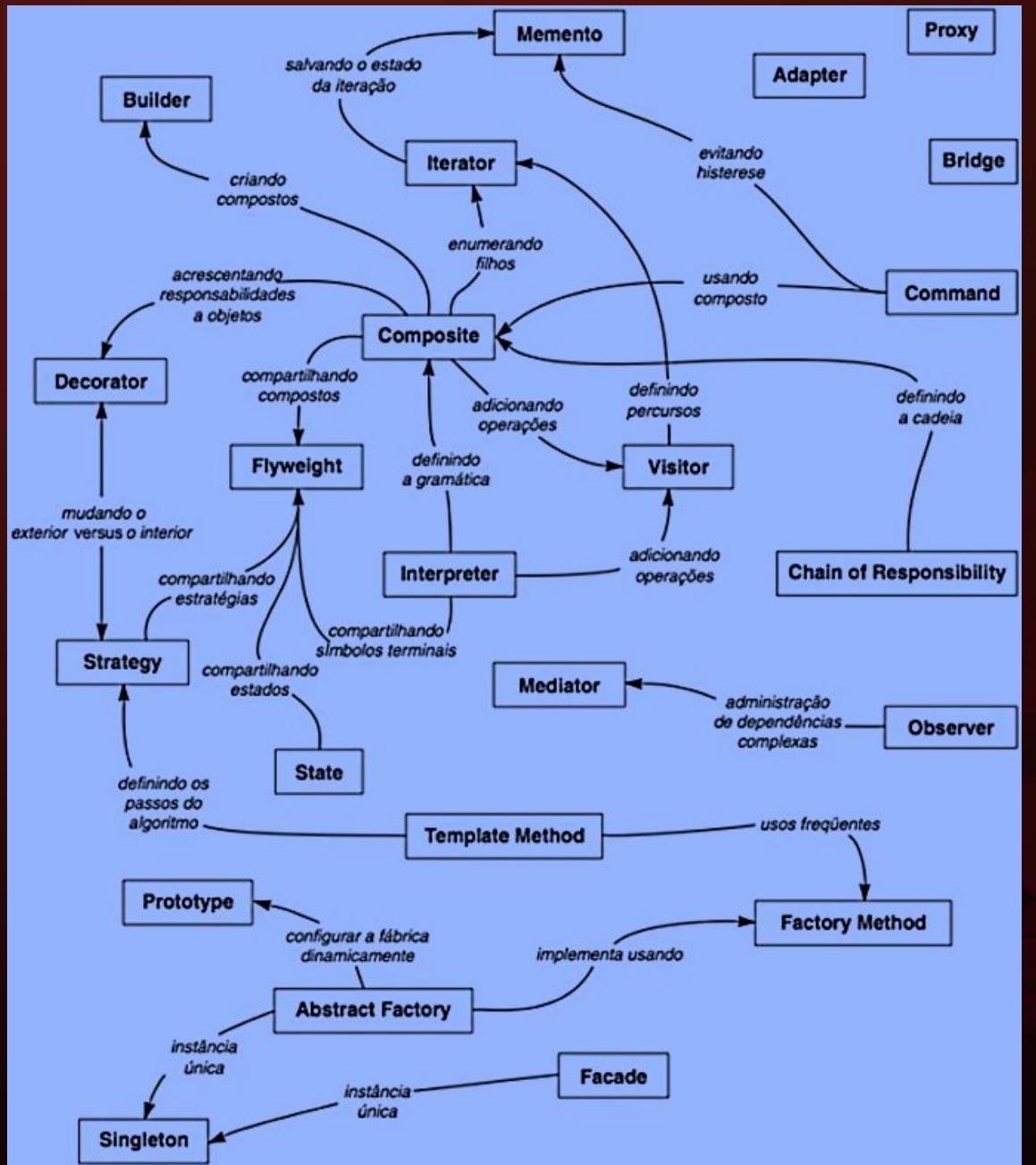
## Padrões relacionados

O padrão Bridge(151) tem uma estrutura similar a um adaptador de objeto, porém, Bridge tem uma intenção diferente: tem por objetivo separar uma interface da sua implementação, de modo que elas possam variar fácil e independentemente. Um adaptador se destina a mudar a interface de um objeto existente.

O padrão Decorator (170) aumenta outro objeto sem mudar sua interface. Desta forma, um Decorator é mais transparente para a aplicação do que um adaptador.

Como consequência, Decorator suporta a composição recursiva, a qual não é possível com adaptadores puros.

O Proxy (198) define um representante ou “procurador” para outro objeto e não muda a sua interface.



# Relacionamentos entre padrões

# SELECIONANDO UM PADRÃO

1. Entenda como padrões de projeto solucionam problemas de projeto, de forma geral.
2. Examine as seções “Intenção”.
3. Estude como os padrões se inter-relacionam.
4. Estude padrões de finalidades semelhantes.
5. Examine uma causa de reformulação de projeto.
6. Considere o que deveria ser variável no seu projeto.



# SELECIONANDO UM PADRÃO

1. Leia o padrão por inteiro uma vez, para obter a sua visão geral.
2. Volte e estude as seções “Estrutura”, “Participantes” e “Colaborações”.
3. Olhe a seção “Exemplo de código”.
4. Escolha nomes para os participantes do padrão que tenham sentido no contexto da aplicação.
5. Defina as classes.
6. Defina nomes específicos da aplicação para as operações no padrão.
7. Defina nomes específicos da aplicação para as operações no padrão.

# REFERÊNCIAS

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. Padrões de projeto: Soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2000.