INF011 – Padrões de Projeto

09 – *Adapter*

Sandro Santos Andrade
sandroandrade@ifba.edu.br

Instituto Federal de Educação, Ciência e Tecnologia da Bahia
Departamento de Tecnologia Eletro-Eletrônica
Graduação Tecnológica em Análise e Desenvolvimento de Sistemas

Adapter

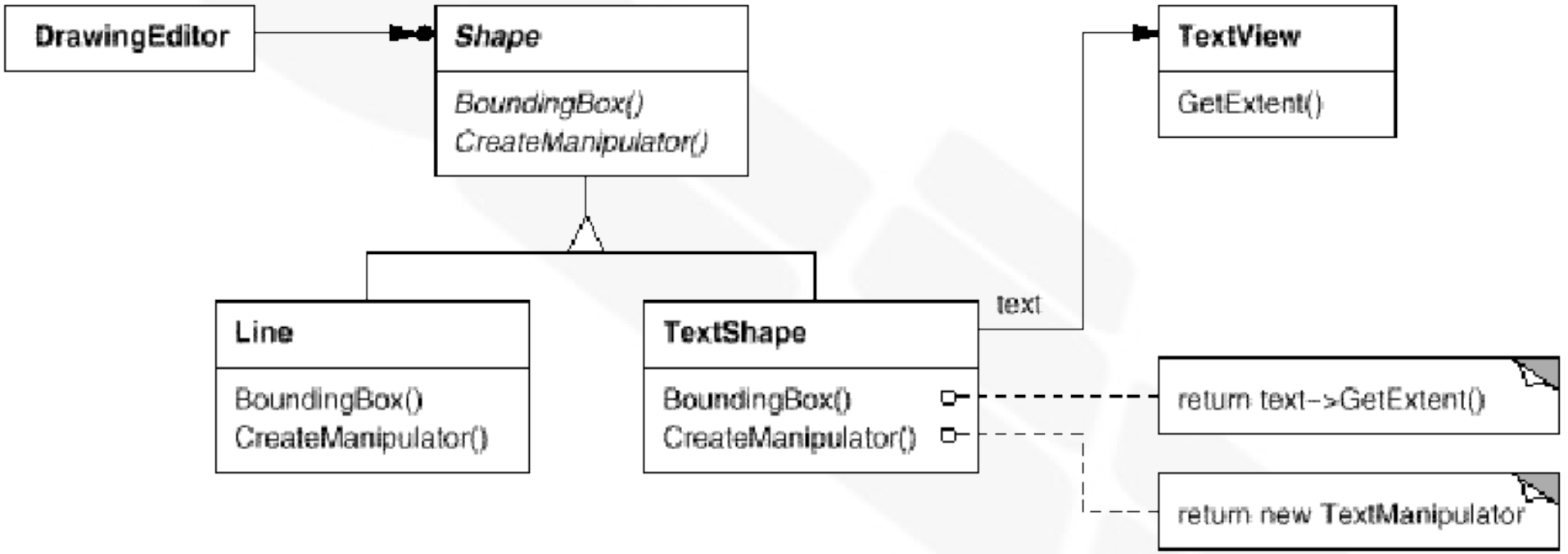
- Propósito:
  - Converter a interface de uma classe em uma outra interface, esperada pelo cliente. Permite que classes com interfaces incompatíveis trabalhem em conjunto
- Também conhecido como: *Wrapper*
- Motivação:
    - Considere um editor de diagramas que define a interface *Shape* e sub-classes para cada tipo de objeto gráfico: *LineShape*, *PolygonShape*, etc
    - Implementar *TextShape* é mais difícil e um *toolkit* já disponibiliza uma classe *TextView* para edição de textos
    - O *toolkit* porém não foi projetado para trabalhar com a interface *Shape*, que é específica da aplicação

Adapter

- Motivação:
  - Possíveis soluções:
    - Modificar o código de *TextView* se disponível
    - Mesmo se disponível não faria sentido requerer que o *toolkit* implemente interfaces específicas de uma aplicação
    - Melhor solução: definir *TextShape* de modo a **adaptar** a interface *TextView* à interface *Shape*
    - Pode-se fazer isso de duas formas:
      1) Herdando a interface de *Shape* e a implementação de *TextView* (*adapter* de classe)
      2) Agregando uma instância de *TextView* em *TextShape*, que implementa a interface *Shape* e a implementa com base nas funcionalidades disponibilizadas por *TextView* (*adapter* de objeto)

Adapter

- Motivação:



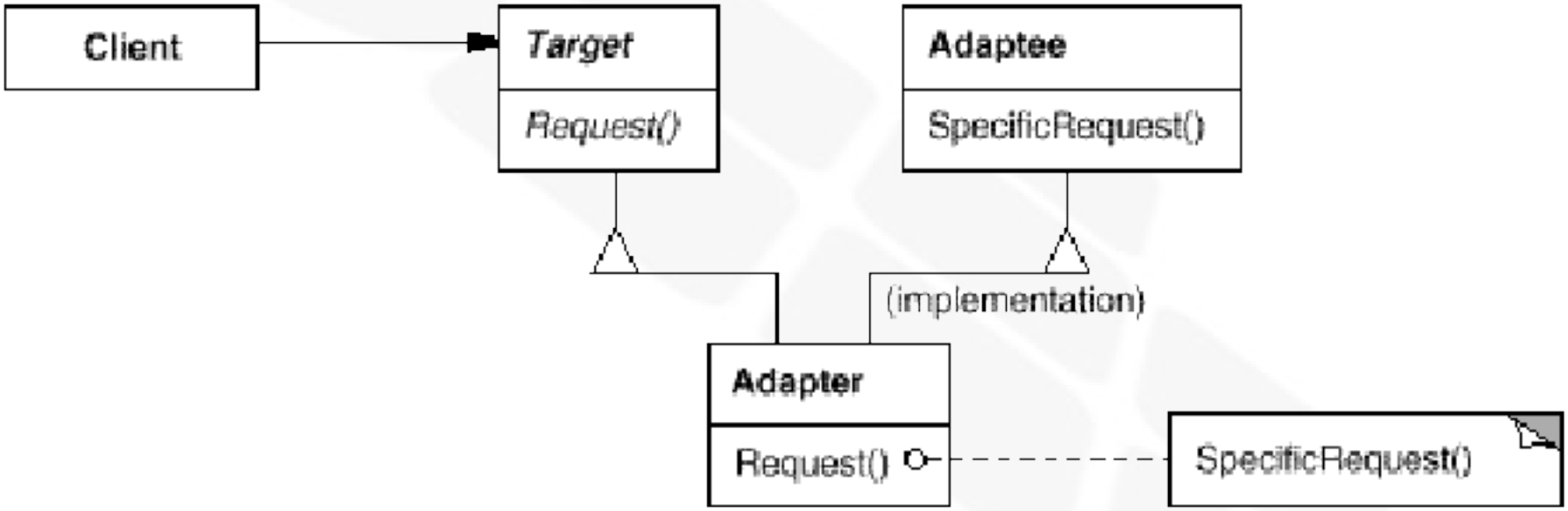
  - *Adapter* de objeto: o método *BoundingBox()* é implementado em termos da funcionalidade disponibilizada por *TextView - GetExtent()*

Adapter

- Aplicabilidade:
  - Deseja-se utilizar uma classe já existente porém sua interface não é compatível com a interface utilizada na aplicação
  - Deseja-se criar uma classe reutilizável que coopere com classes não relacionadas ou imprevistas, ou seja, classes com interfaces não necessariamente compatíveis
  - Precisa-se utilizar várias sub-classes já existentes mas é impraticável criar uma sub-classe para cada uma. Um *adapter* de objeto pode adaptar a interface da classe-pai das classes existentes

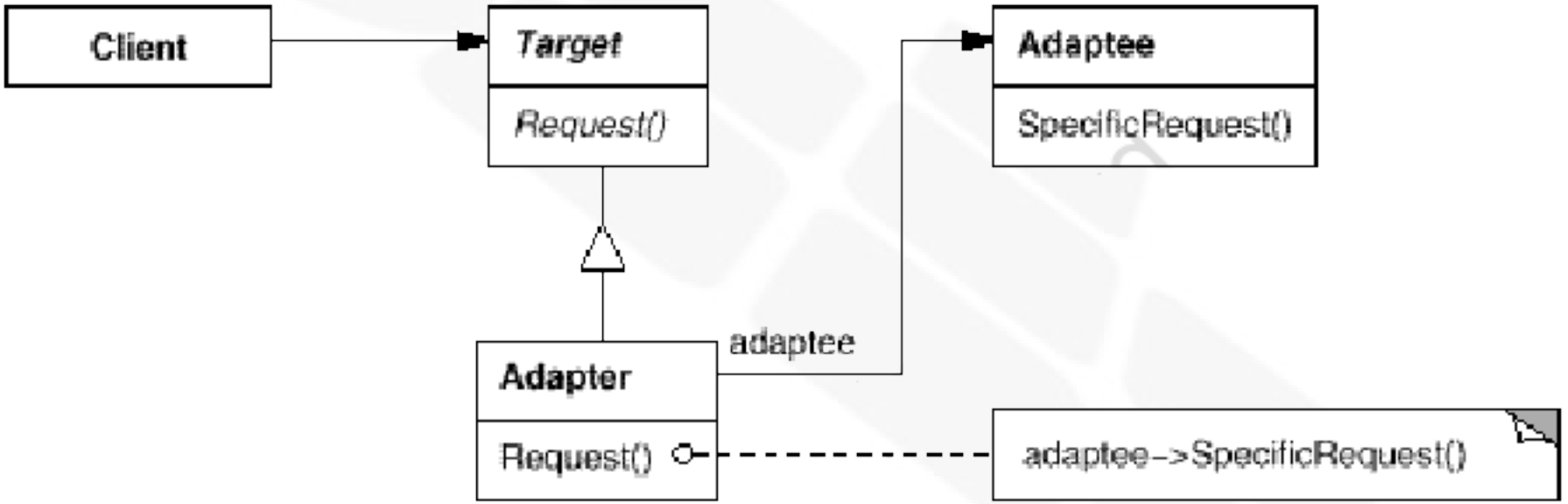
Adapter

- Estrutura *(Adapter* de classe*):*



Adapter

- Estrutura *(Adapter* de objeto*):*



Adapter

- Participantes:
  - *Target* (Shape): define a interface específica de domínio a ser utilizada pelos clientes
  - *Client* (Editor de diagramas): colabora com objetos em conformidade com a interface *Target*
  - *Adaptee* (TextView): define a interface pré-existente que necessita ser adaptada
  - *Adapter* (TextShape): adapta a interface de *Adaptee* para a interface *Target*

Adapter

- Colaborações:
  - O *Client* invoca operações na instância do *Adapter*, que por sua vez chama operações do *Adaptee* de modo a atender à requisição

Adapter

- Consequências:
  - Um *adapter* de classe:
    - Realiza a adaptação ao se comprometer com uma classe concreta do *Adaptee* e, portanto, não pode ser utilizado para adaptar uma classe **e** todas as suas sub-classes
    - Permite que o *Adapter* realize sobreposição de métodos do *Adaptee*, visto que o *Adapter* será uma sub-classe do *Adaptee*
    - Acrescenta somente um objeto e não inclui uma indireção a mais para acessar o *Adaptee*

Adapter

- Consequências:
  - Um *adapter* de objeto:
    - Permite que um único *Adapter* trabalhe com vários *Adaptees*, ou seja, o próprio *Adaptee* e todas as suas sub-classes
    - O *Adapter* pode adicionar funcionalidades a todos os *Adaptees* de uma única vez
    - Torna mais difícil realizar sobreposição dos métodos do *Adaptee*. Será necessário derivar o *Adaptee* e fazer com que o *Adapter* utilize a sub-classe ao invés do *Adaptee* original

Adapter

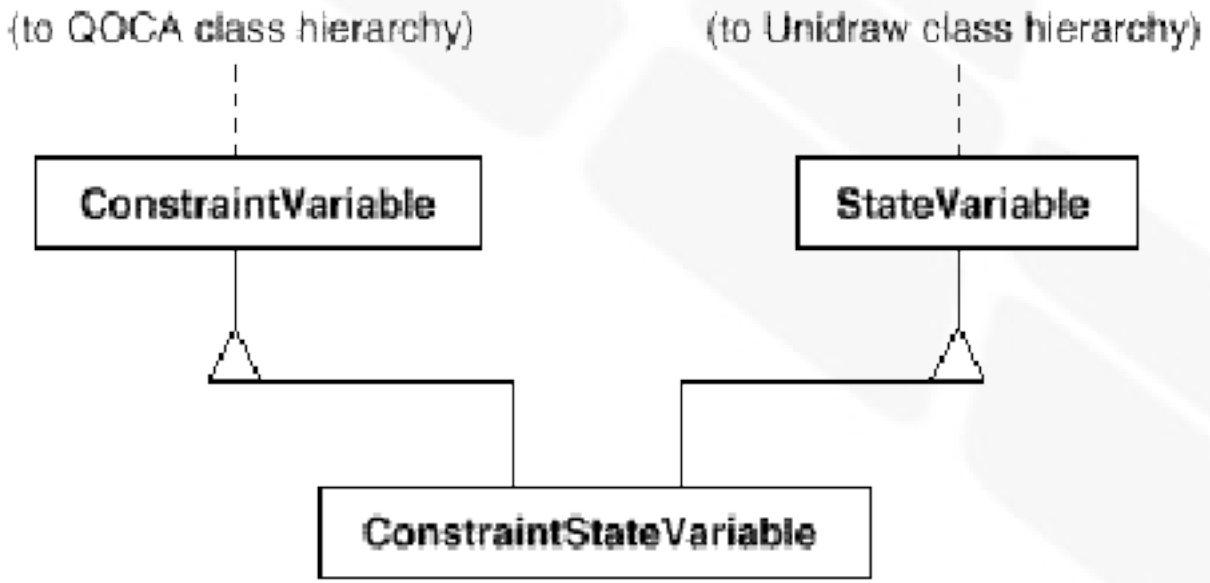
- Outras consequências:
  - Quanta adaptação o *Adapter* realiza ?
    - Realizam desde simples conversões de interfaces até a adaptação de um conjunto totalmente diferente de operações. Depende do grau de similaridade entre as interfaces do *Target* e *Adaptee*
  - *Pluggable Adapters*:
    - Suponha um *widget TreeDisplay* que apresenta graficamente estruturas hierárquicas
    - Se este *widget* for específico para uma aplicação pode-se exigir que os objetos apresentados tenham uma interface específica (ex: descendentes de *Tree*)
    - Se ele fizer parte de um *toolkit* reutilizável, entretanto, não pode depender de uma interface específica

Adapter

- Outras consequências:
  - *Two-way Adapters*:
    - Um objeto adaptado não mais disponibiliza a interface do *Adaptee* e, portanto, não pode ser utilizado por clientes desta interface
    - Quando dois clientes diferentes precisam ter visões diferentes de um mesmo objeto utiliza-se *Two-way Adapters*

Adapter

- Outras consequências:
  - *Two-way Adapters*:
    - Ex: Unidraw (*framework* para editores gráficos) e QOCA (*toolkit* para satisfação de restrições)

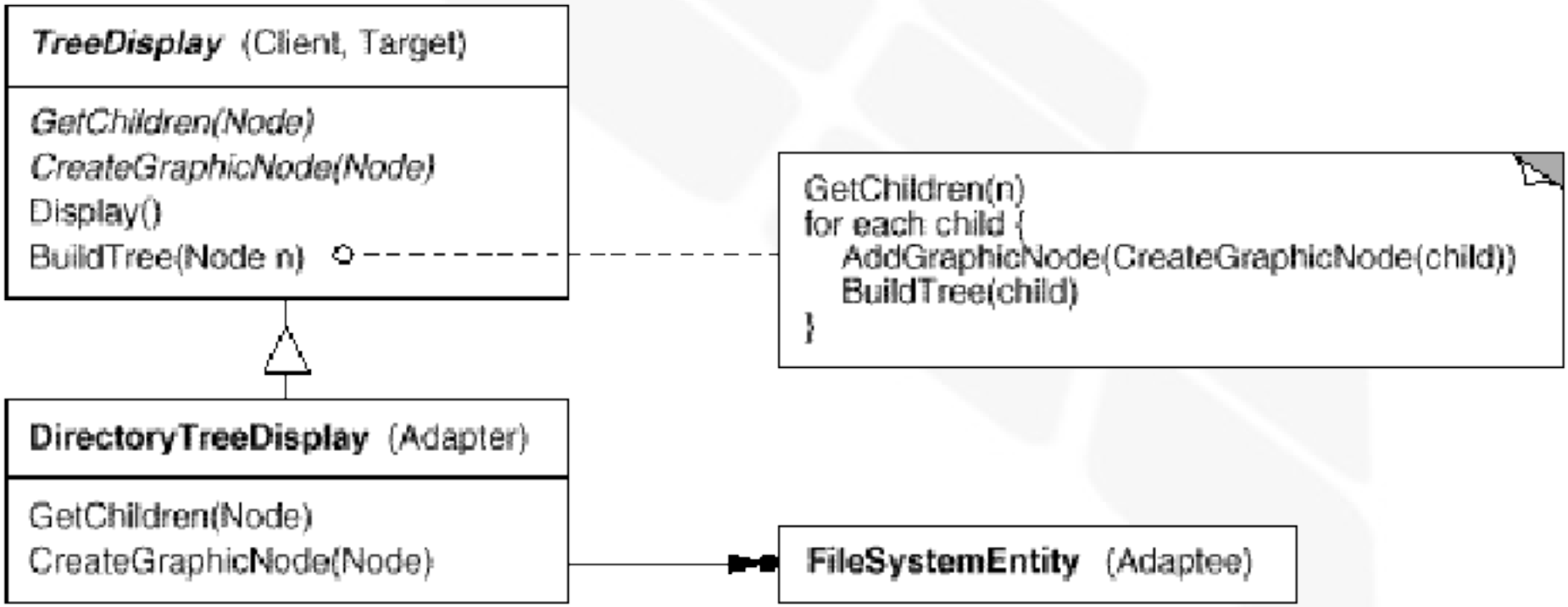


Adapter

- Implementação:
  - Implementando *Adapters* de classe em C++:
    - O *Adapter* deve fazer herança pública de *Target* e privada de *Adaptee*. Ele seria sub-tipo de *Target* mas não de *Adaptee*
  - *Pluggable Adapters*: três formas de implementação:
    - Passo comum a todas: definir o menor sub-conjunto possível de operações que possibilita a adaptação
    - Ex do *TreeView*: uma operação para exibição gráfica do nó e outra para recuperar os filhos de um determinado nó
    - A partir daí tem-se três abordagens possíveis ...

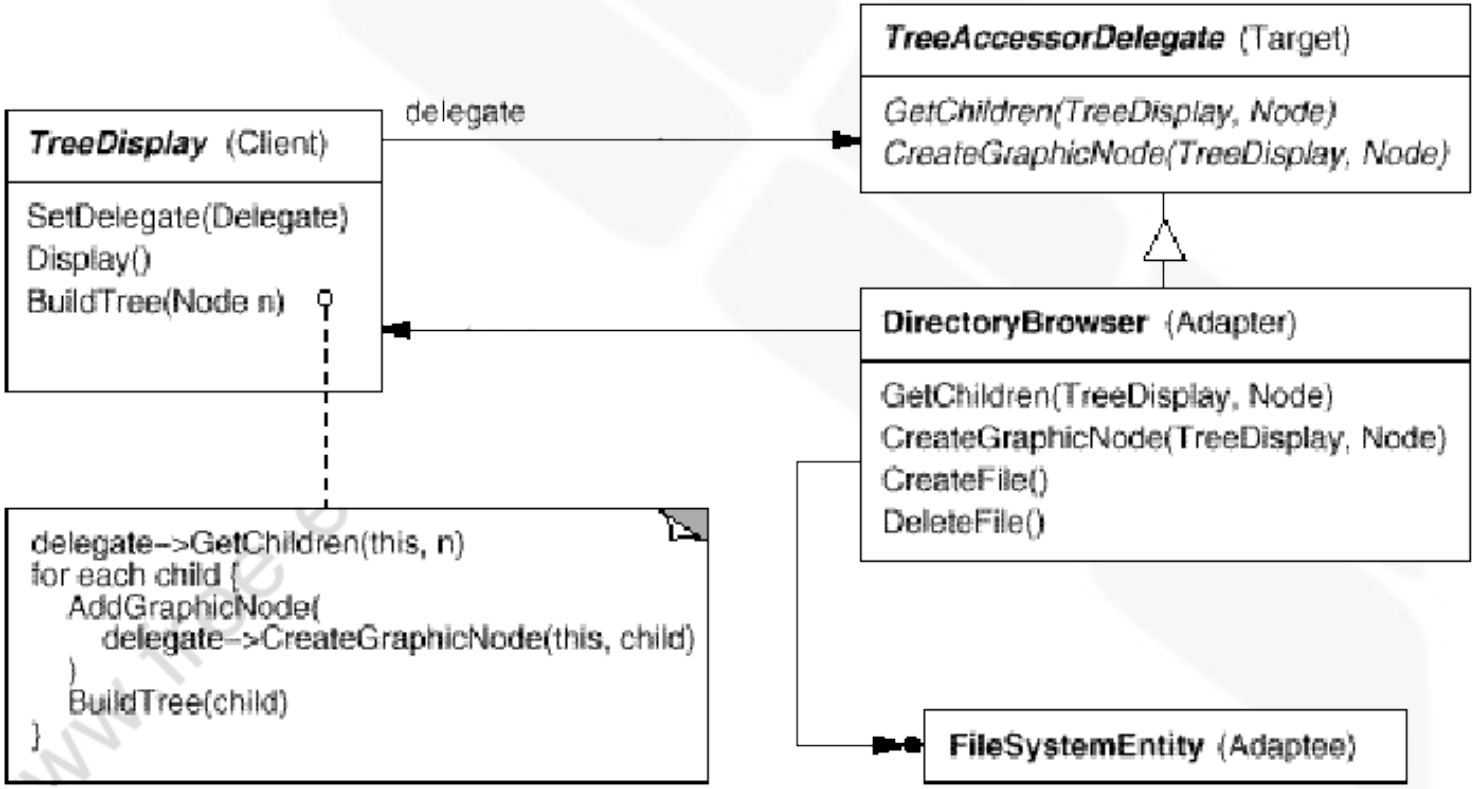
Adapter

- Implementação:
  - *Pluggable Adapters*: três formas de implementação:
    1) Usando métodos abstratos:



Adapter

- Implementação:
  - *Pluggable Adapters*: três formas de implementação:
    2) Usando *delegates*:



Adapter

- Implementação:
  - *Pluggable Adapters*: três formas de implementação:

    3) Adapters parametrizados:

    - O *Adapter* é parametrizado com um ou mais *blocks*
    - Existe um *block* para cada método do *Adaptee* a ser adaptado

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

Adapter

- Código exemplo (*Adapter* de classe):

```
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;
};
```

Adapter

- Código exemplo (*Adapter* de classe):

```
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);
}
```

Adapter

- Código exemplo (*Adapter* de classe):

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

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

Adapter

- Código exemplo (*Adapter* de objeto):

```
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;
};
```

Adapter

- Código exemplo (*Adapter* de objeto):

```
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();
}
```

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

Adapter

- Usos conhecidos:
  - ET++Draw: exemplo *TextShape*
  - InterViews 2.6
  - ObjectWorks / Smalltalk: *pluggable adapters*
  - NeXT AppKit

Adapter

- Padrões relacionados:
  - O *Bridge* possui uma estrutura semelhante ao *Adapter* de objeto porém tem uma intenção diferente: seu objetivo é separar uma interface de sua implementação, de modo que eles possam variar de forma fácil e independente. O *Adapter* modifica a interface de um objeto que já existe
  - O *Decorator* melhora um objeto sem modificar a sua interface e é, portanto, mais transparente para a aplicação do que o *Adapter*. O *Decorator* suporta composição recursiva enquanto o *Adapter* não
  - O *Proxy* define um representante ou substituto para um objeto e não modifica a sua interface

INF011 – Padrões de Projeto

09 – *Adapter*

Sandro Santos Andrade
sandroandrade@ifba.edu.br

Instituto Federal de Educação, Ciência e Tecnologia da Bahia
Departamento de Tecnologia Eletro-Eletrônica
Graduação Tecnológica em Análise e Desenvolvimento de Sistemas