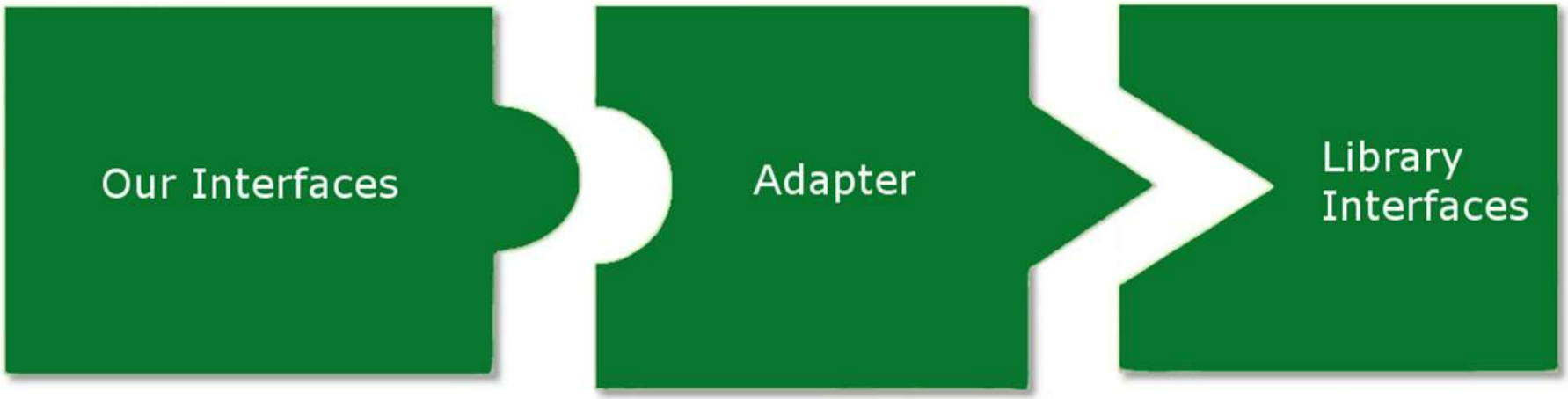
Adapter

Adapter – motivation and concept

- **Component designed to connect two "incompatible" devices**
- **In this scenario:**
  - A type of plug which makes it possible to connect two or more pieces of equipment (with different power plugs) to the same electrical supply

Adapter – design pattern

- **Also called “Wrapper”**
- **Structural Design Pattern**
  - Building simple hierarchies and relations between different classes
- **Links existing classes with an incompatible interface**
  - Without altering them – same underlying behaviour
  - Asymmetric - one class "connects to the other"



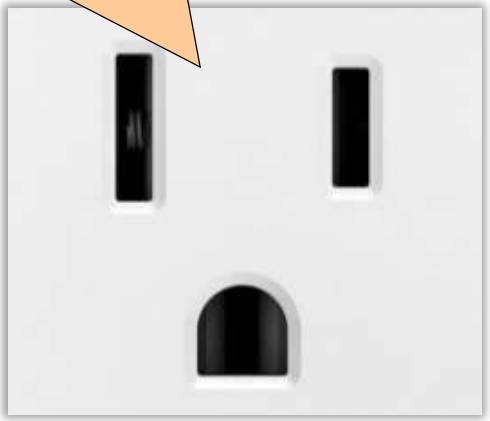
Adapter – participants

Client
(we adapt for)

Target
(defines the interface required by the Client)

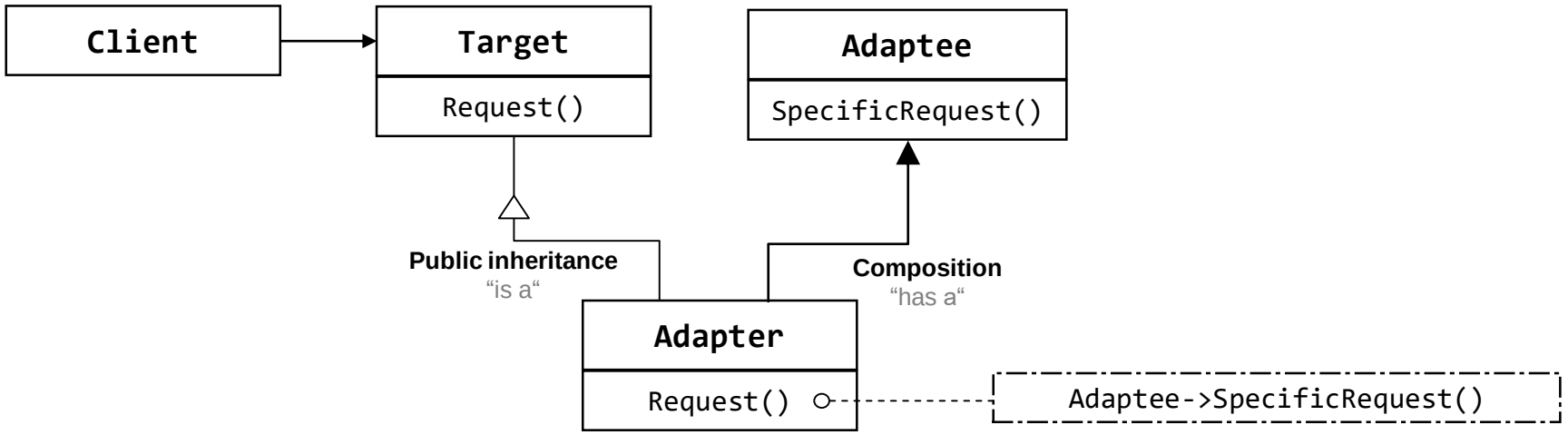
Adapter
(adapts Adaptee to Target Interface)

Adaptee
(defines the interface the Client wants to use)

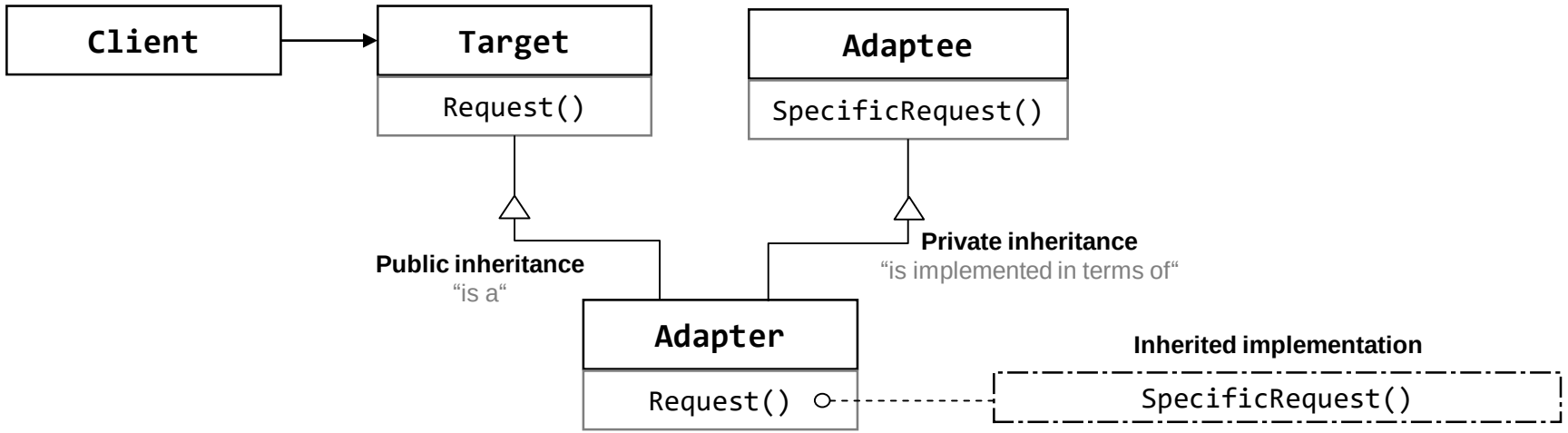


Adapter – two connection options

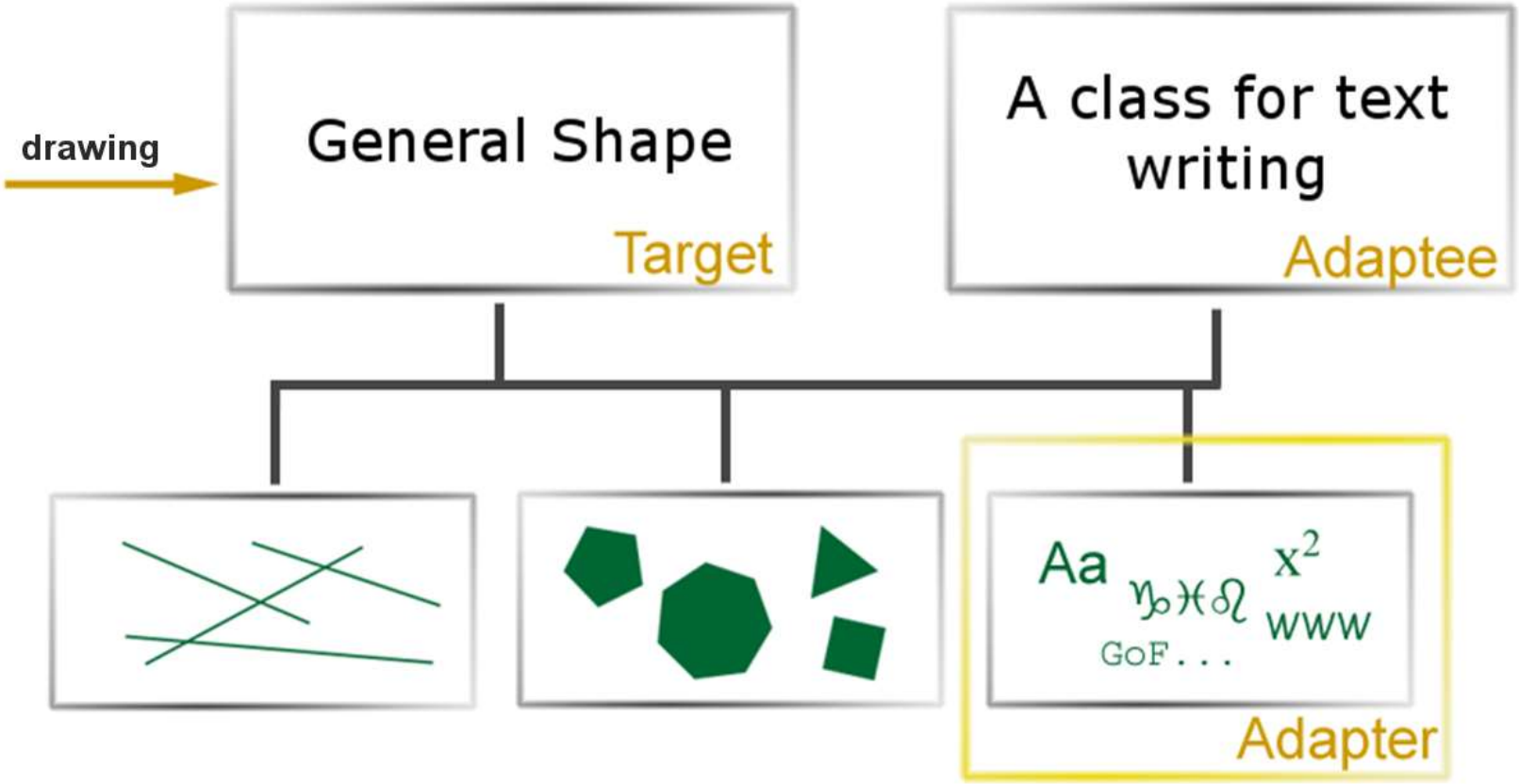
- **Object Adapter**



- **Class Adapter**



Adapter – Drawing App example



Adapter – example

Target Shape
defines the target interface

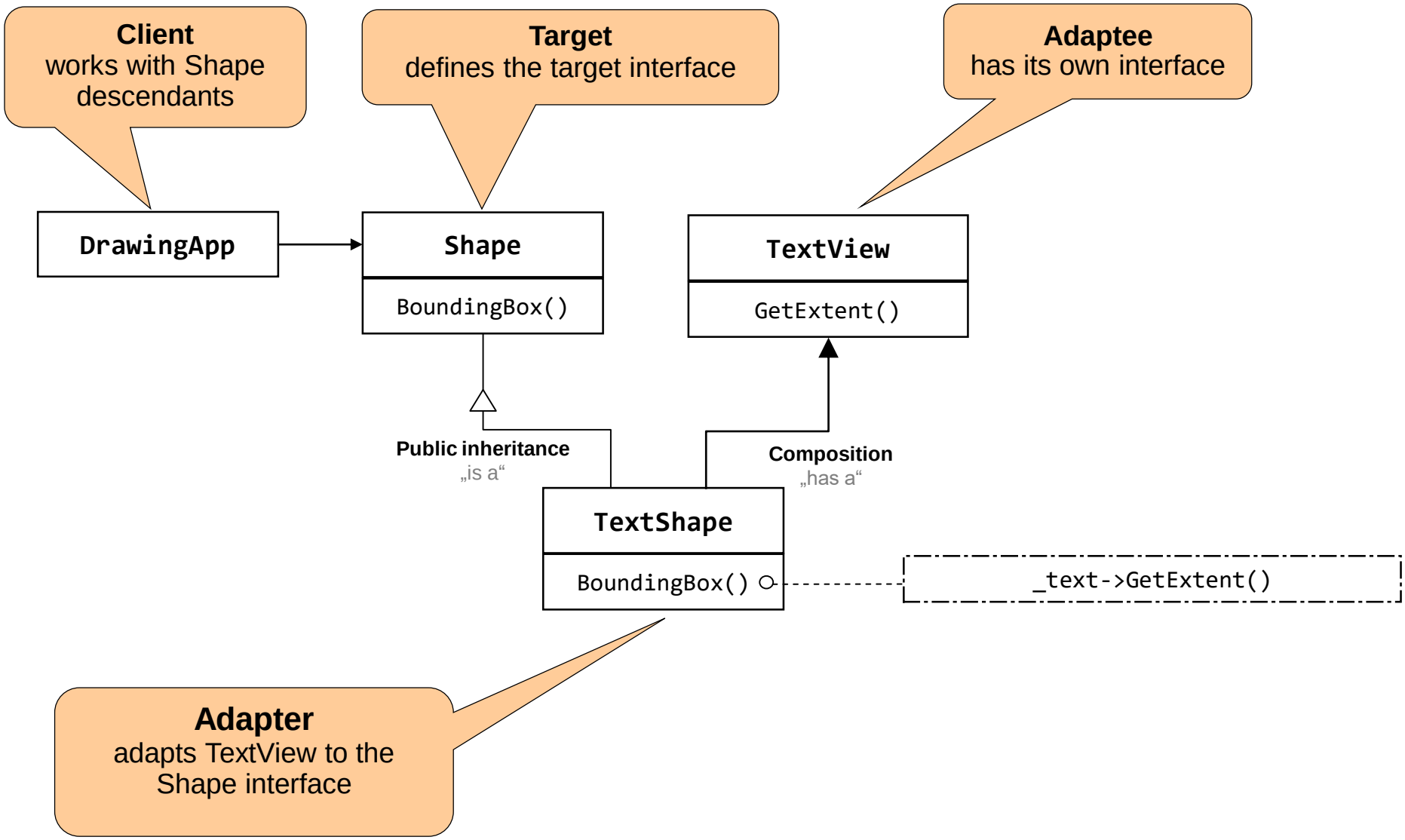
```
class Shape
{
   virtual void BoundingBox( Point& bottomLeft, Point& topRight ) const;
   virtual Manipulator* CreateManipulator() const;
};
```

Adaptee TextView
implements some methods, but in another interface

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

Object Adapter

Object Adapter – example



Object Adapter – example

- **Object Adapter TextShape**

Object Adapter `TextShape`
publicly inherits from
Target Shape

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

    virtual void BoundingBox(Point& bottomLeft, Point& topRight) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator();

 private:
    TextView* _text;
};
```

Adaptee `TextView`
is a private member of
TextShape Adapter

Object Adapter – example

- **Object Adapter TextShape – implementation**

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

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

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

the constructor requires an existing Adaptee instance

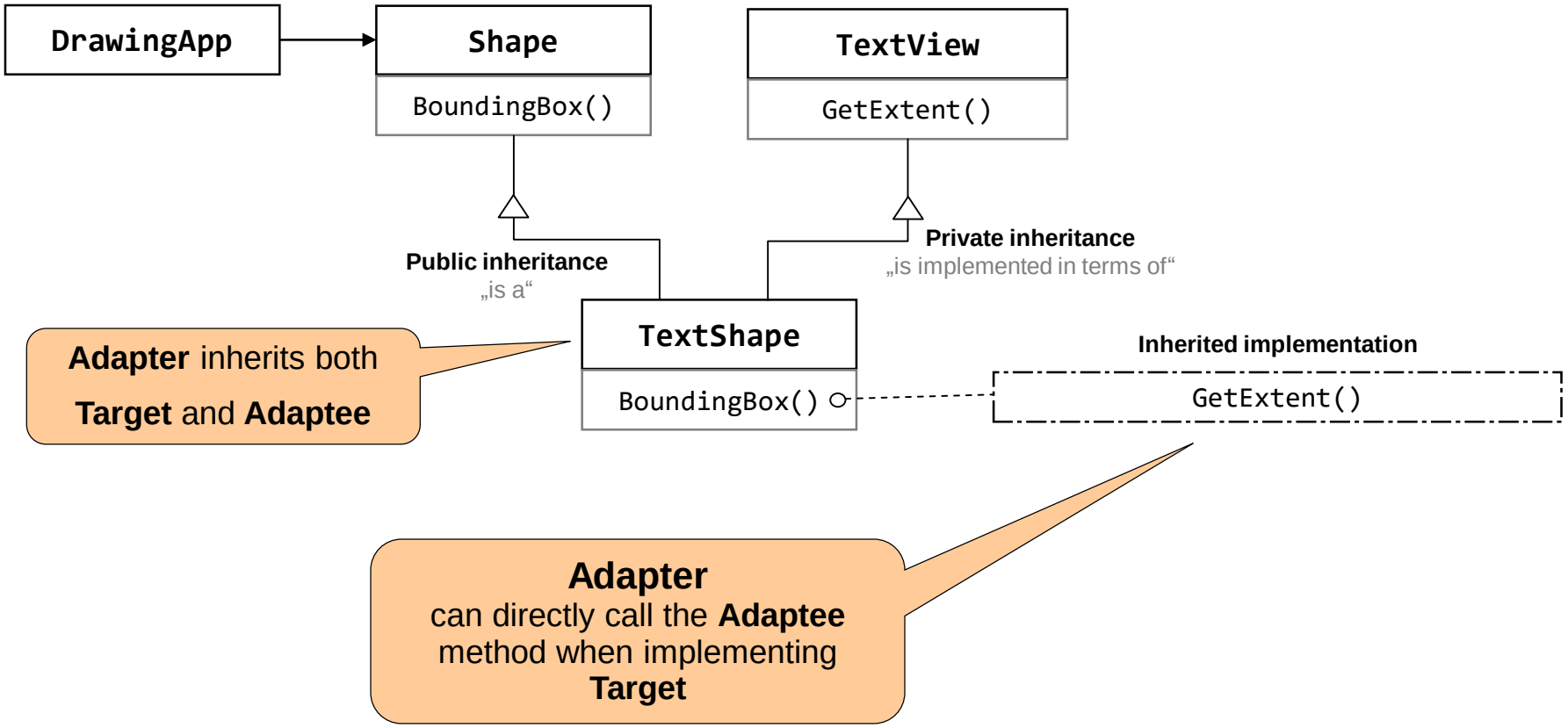
invoke Adaptee method and simply return its result

Object Adapter – properties

- **Uses composition**
  - **Adaptee** is a private data item of **Adapter**
- **May arise as a wrapper around an existing instance of Adaptee**
  - Example – `TextShape::TextShape(TextView* t)`
- **Can adapt all descendants of Adaptee**
  - Example – any descendant of `TextView`
- **Does not have access to the protected items of the Adaptee itself**
- **Cannot redefine Adaptee items**
  - Indirect - subclass **Adaptee,** redefined and used in **Adapter** instead of **Adaptee**

Class Adapter

Class Adapter – example



Class Adapter – example

- **Class Adapter TextShape**

Multiple Inheritance
public from **Target**
private from **Adaptee**

```
class TextShape: public Shape, private TextView
{
 public:
     TextShape();

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

Class Adapter – example

- **Class Adapter TextShape – implementation metod**

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

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

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

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

Direct calling methods of the ancestor

Class Adapter – properties

- **There is no indirection when delegating methods**
  - Methods are called directly, not using pointers
- **Uses multiple inheritance**
  - Access to protected **Adaptee** items can be redefined
  - Can override some **Adaptee** behaviour
- **Multiple (class) inheritance is not supported by many languages**
  - But we can inherit from multiple interfaces
- **Cannot adapt subclasses of Adaptee**

Implementation notes

- **How much work needs to be done by the Adapter itself?**
  - Simple conversion between differently named methods

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

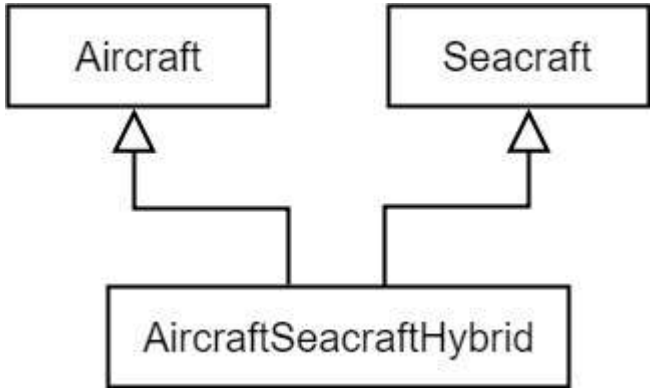
  - Operation not supported by **Adaptee** - Processing logic
  - Sometimes it implements the **Adapter** without the help of **Adaptee**

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

- **Adapter does not necessarily have to adapt only one class**

Some other variations of implementation

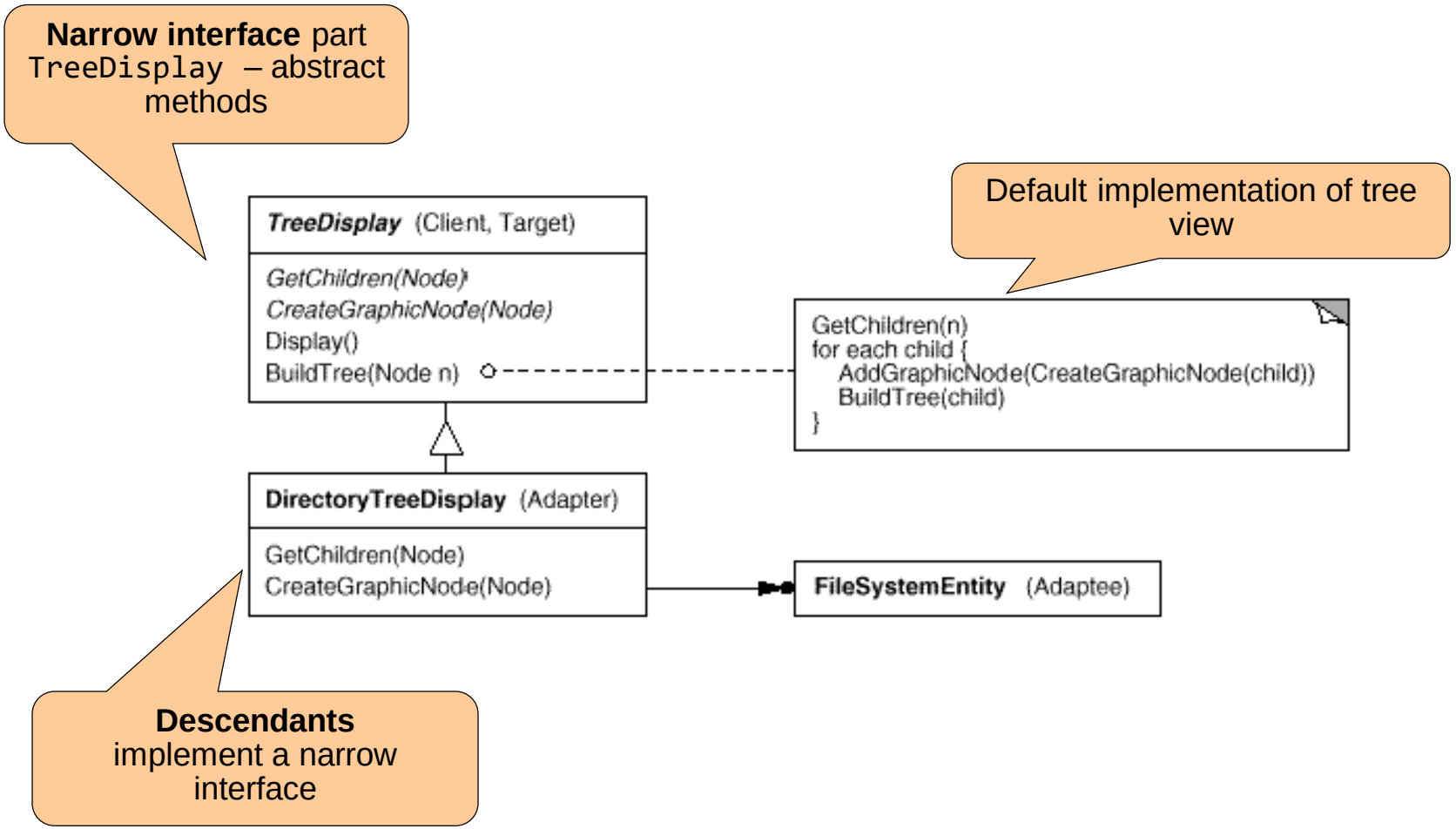
- **Two-way Adapter**
  - Interchangeability of the **Adapter** behind **Adaptee** objects
  - It behaves like **Target** and **Adaptee**
  - Typical implementation with **multiple public inheritance**



- **Pluggable Adapter**
  - Up to now - one **Client,** one **Adaptee**
  - Many **Adapters** for Many Adjusted Classes
  - An adapter that isn't hard-coded against a specific **Adaptee**
  - Can adapt to different Adaptees with different interfaces
  - **Unified interface** for adapting all representations
  - Minimal – **“narrow interface“**
  - Implementation types:
    - Abstract methods
    - Delegate

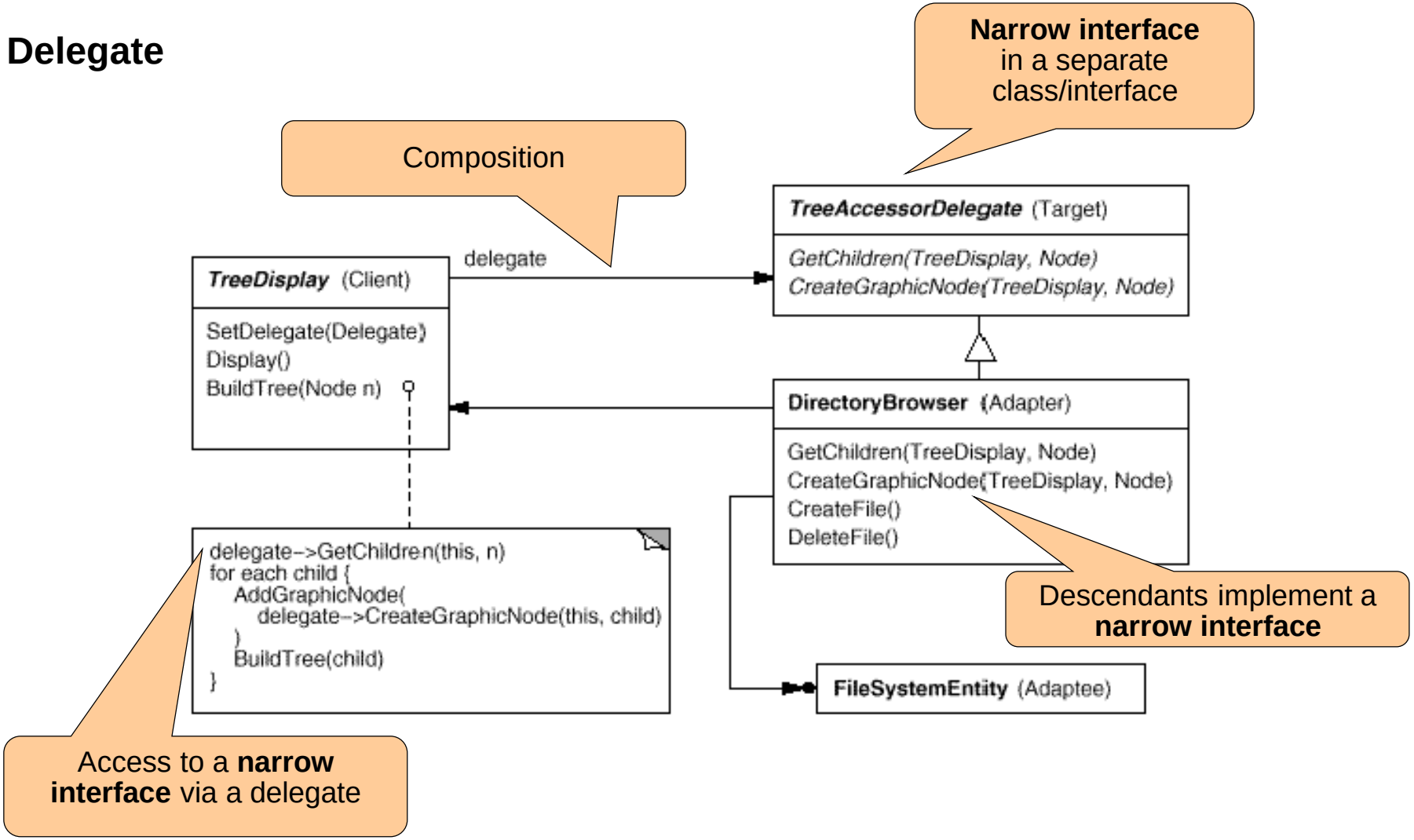
Pluggable Adapter – Abstract methods

Abstract methods



Pluggable Adapter – Delegate

Delegate



Pluggable Adapter – Delegate (C# example)

```
class Cat //Adaptee 1
{public static void Meow() { Console.WriteLine("Meow!"); }}

class Dog //Adaptee 2
{public static void Bark() { Console.WriteLine("Woof!"); }}

class Adapter
{
    public Action MakeSound { get; private set;} // "pluggable" adapter

    public Adapter(Action makeSoundAction)
    {
        MakeSound = makeSoundAction;
    }
}

class SoundMaker //Client
{
    static void Main(string[] args)
    {
        (new Adapter(Cat.Meow)).MakeSound();
        (new Adapter(Dog.Bark)).MakeSound();
    }
}
```

Action Delegate encapsulate some method

Pluggable Adapter – properties

- **Abstract methods**
  - **Client** is also **Target**, inheriting directly from it
  - The **Adaptee** interface is only suitable for the object-adapter
    - **Target** is not just an interface - multiple inheritance required
- **Delegate**
  - Delegating a narrow interface to a delegated class
    - Access methods through a pointer
    - A pointer back to the **Client**
  - Possible object and class adapter
  - Two objects – **Client** and **Adapter**

Adapter – usage

- **When to use**
  - Use an existing class with an incompatible interface
  - Creating a reusable class that uses classes with a potentially incompatible interface

- **Known uses**
  - C# – ADO.NET
    - **`DataAdapter`** – data queries independent of the data source
    - Usage in **`DataSet`**
  - Java I/O
    - **`StringBufferInputStream`** – adapts **`StringBuffer`** as **`InputStream`**
  - Pluggable Adapter in Swing
    - **`javax.swing.tree.TreeModel`** – **`JTree`** component viewing interface

Adapter – related design patterns

- **Related DPs**
  - Bridge
    - Decouple interface from implementation
    - Does not change the interface
    - Used at design time
  - Decorator
    - Does not change the interface of the object
    - Adds functionality
    - Support recursive composition
  - Proxy
    - Does not change the interface
    - Hides the actual location of the object (e.g. object on disk, on another computer)
  - Facade
    - Provides a simplified interface
    - Hides the complexities of a larger system