

Adapter

Sayali Vidyadhar Naval
Biao A

Example

```
void chargeMyLaptop(USPlug plug){  
    plug.provideElectricity();  
}
```

```
interface USPlug{  
    void provideElectricity();  
}
```



Example

```
void chargeMyLaptop(USPlug plug){  
    plug.provideElectricity();  
}
```

```
interface USPlug{  
    void provideElectricity();  
}
```

```
class UKPlug{  
    public void giveElectricity();  
}
```



Intent

Convert the interface of a class into another interface clients expect.

Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



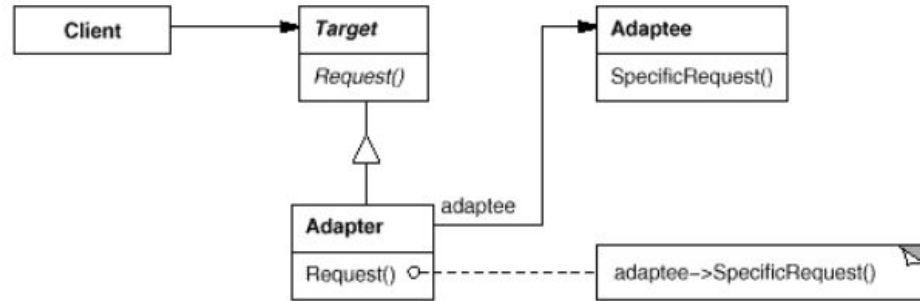
Structure

There are two types of adapter.

A class adapter uses multiple inheritance to adapt one interface to another.

An object adapter relies on object composition.

An object adapter relies on object composition:



Target: defines the domain-specific interface that Client uses.

Client: collaborates with objects conforming to the Target interface.

Adaptee: defines an existing interface that needs adapting.

Adapter: adapts the interface of Adaptee to the Target interface.

Example

```
void chargeMyLaptop(USPlug plug){  
    plug.provideElectricity();  
}
```

```
interface USPlug{  
    void provideElectricity();  
}
```

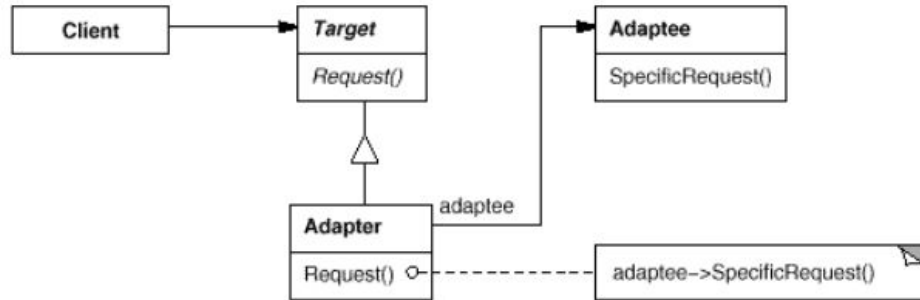
```
class UKPlug{  
    public void giveElectricity();  
}
```



```
void chargeMyLaptop(USPlug plug){  
    plug.provideElectricity();  
}
```

```
interface USPlug{  
    void provideElectricity();  
}
```

```
class UKPlug{  
    public void giveElectricity();  
}
```



Example

```
class Adapter implements USPlug{
    private UKPlug plug;

    public Adapter(UKPlug plug) {
        this.plug = plug;
    }

    @Override
    public void provideElectricity() {
        //detailed implementation...
        plug.giveElectricity();
    }
}
```

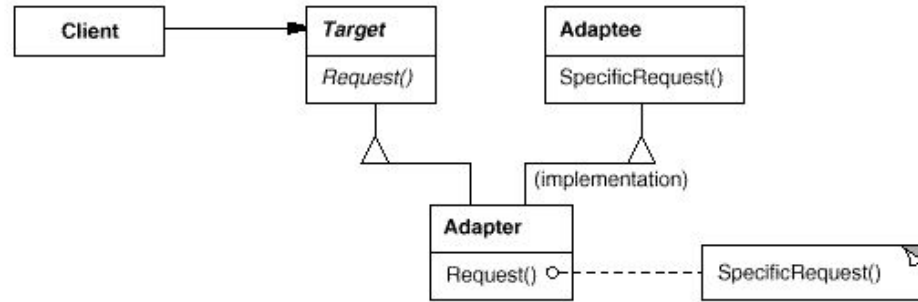
```
void chargeMyLaptop(USPlug plug){
    plug.provideElectricity();
}
```

UKPlug ukPlug = new UKPlug();

USPlug adapter = new Adapter(ukPlug);

chargeMyLaptop(adapter);

A class adapter uses multiple inheritance to adapt one interface to another.



Target: defines the domain-specific interface that Client uses.

Client: collaborates with objects conforming to the Target interface.

Adaptee: defines an existing interface that needs adapting.

Adapter: adapts the interface of Adaptee to the Target interface.

Example

```
class USPlug{  
public:  
    virtual void provideElectricity();  
}
```

```
class UKPlug{  
public:  
    virtual void giveElectricity();  
}
```

```
class Adapter :public USPlug, private UKplug{  
    public void provideElectricity() {  
        giveElectricity();  
    }  
}
```

```
USPlug *adapter = new Adapter();  
chargeMyLaptop(adapter);
```

Why do we need an adapter?

- Sometimes a toolkit class that's designed for reuse isn't reusable only because kit's interface doesn't match the domain-specific interface an application requires.
- The Open/Closed Principle

When to use adapter?

You want to use an existing class, and its interface does not match the one you need.

- Parse the input data
- Convert one type to another

Consequences

Class Adapter

Won't work when we want to adapt a class *and* all its subclasses.

Lets Adapter override some of Adaptee's behaviour.

Introduces only one object.

Cannot be used in Java and C# as they don't support multiple inheritance.

Consequences

Object Adapter

Lets a single adapter work with many Adaptees.

Makes it harder to override Adaptee behaviour.

Implementation

Implementing class adapters in C++

Adapter would inherit publicly from Target and privately from Adaptee. Thus Adapter would be a subtype of Target but not of Adaptee.

Implementation

Pluggable Adapters

Find a narrow interface for *Adaptee*, that is, the smallest subset of operations that lets us do the adaptation.

The narrow interface leads to three implementation approaches:

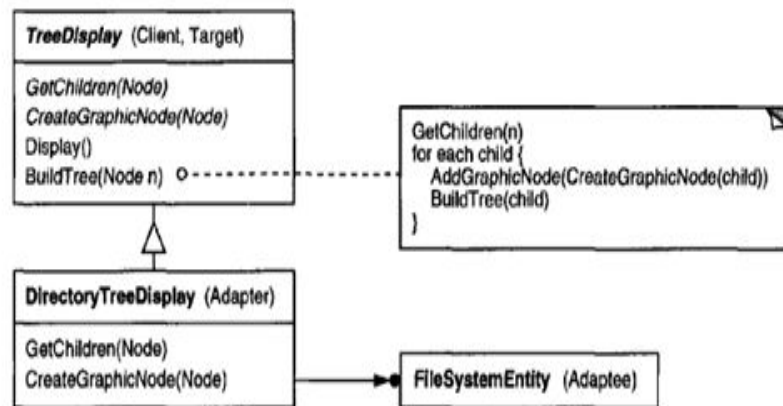
- a. Using abstract operation.
- b. Using delegate objects.
- c. Parameterized adapters.

Implementation

a. Using abstract operation

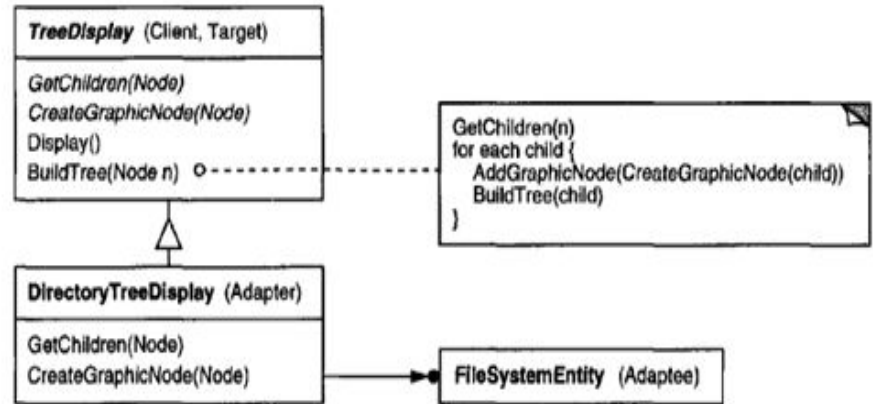
Define corresponding abstract operations for the narrow Adapter interface in the TreeDisplay class.

Subclasses must implement the abstract operations and adapt the hierarchically structured object.



A DirectoryTreeDisplay subclass will implement these operations by accessing the directory structure.

DirectoryTreeDisplay specializes the narrow interface so that it can display directory structures made up of FileSystemEntity objects.



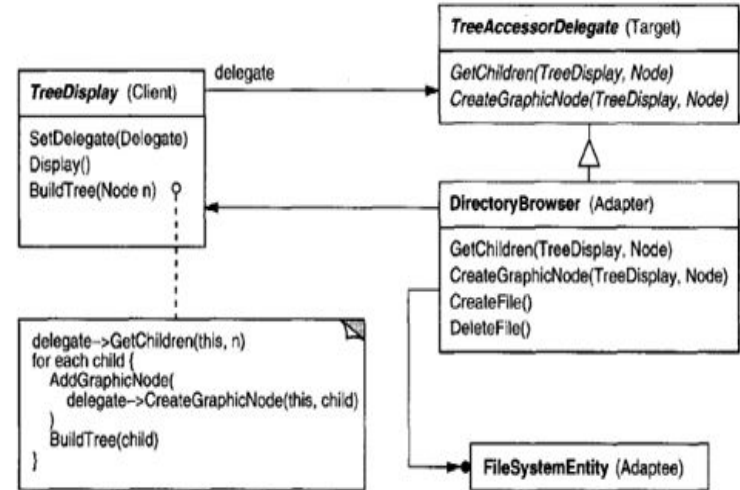
Implementation

b. Using delegate objects

In this approach, TreeDisplay forwards requests for accessing the hierarchical structure to a delegate object.

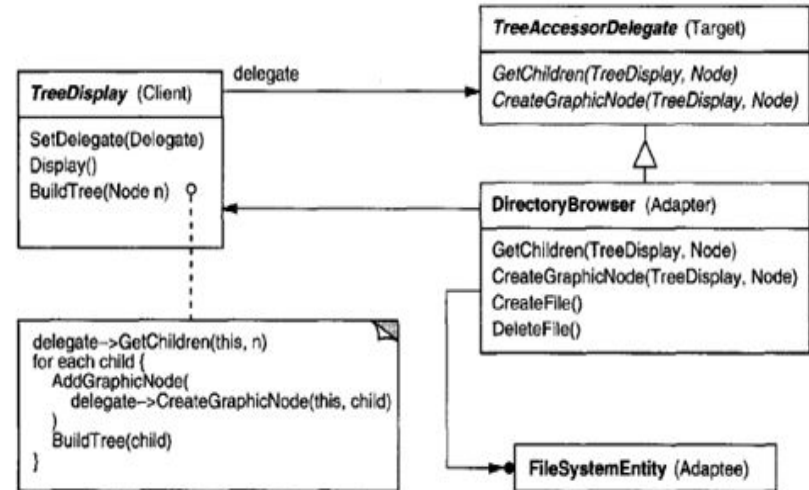
Suppose there exists a DirectoryBrowser that uses a Tree-Display.

DirectoryBrowser might make a good delegate for adapting TreeDisplay to the hierarchical directory structure.



Putting the narrow interface that TreeDisplay requires into an abstractTreeAccessorDelegate class.

Then we can mix this interface into the DirectoryBrowser delegate using inheritance.



Implementation

c. **Parameterized adapters**

To parameterize an adapter with one or more blocks.

Supports adaptation without subclassing.

A block can adapt a request, and the adapter can store a block for each individual Request.

Example:

TreeDisplay stores one block for converting a node into a GraphicNode and another block for accessing a node's children.

To create TreeDisplay on a directory hierarchy, we write

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

Known Uses

ET++Draw

ET++Draw reuses the ET++ classes for text editing by using a TextShape adapter class.

InterViews 2.6

Defines an object adapter called GraphicBlock, a subclass of Interactor that contains a Graphic instance and adapts the interface of the Graphic class to that of Interactor.

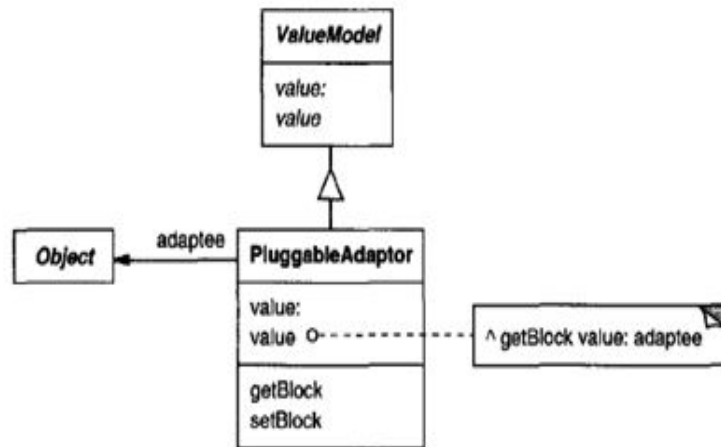
Known Uses

ObjectWorks\Smalltalk

Includes a subclass of ValueModel called PluggableAdaptor.

A PluggableAdaptor object adapts other objects to the ValueModel interface.

It can be parameterized with blocks for getting and setting the desired value.



Known Uses

Meyer's "Marriage of Convenience"

A form of class adapter.

Adapts implementation of Array class to the interface of a Stack class.

The result is a stack containing a fixed number of entries.

Advantages & Disadvantages

Advantages

Helps achieve reusability and flexibility.

Client class is not complicated by having to use a different interface and can use polymorphism to swap between different implementations of adapters.

Disadvantages

All requests are forwarded, so there is a slight increase in the overhead.

Sometimes many adaptations are required along an adapter chain to reach the type which is required.

Related patterns

Bridge

Decorator

Proxy

References

https://www.tutorialspoint.com/design_pattern/adapter_pattern.htm

<https://www.geeksforgeeks.org/adapter-pattern/>

Design Patterns: Elements of Reusable Object-Oriented Software

Thank You!

Questions?