Adapter pattern

From Wikipedia, the free encyclopedia

In software engineering, the **adapter pattern** is a software design pattern that allows the interface of an existing class to be used as another interface.[1] It is often used to make existing classes work with others without modifying their source code.

Contents

Definition

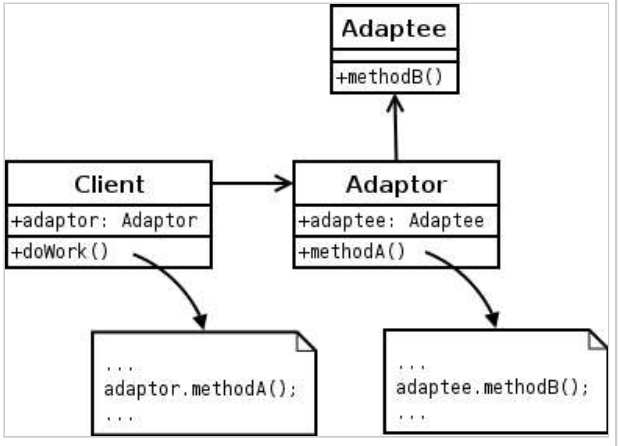
An adapter helps two incompatible interfaces to work together. This is the real world definition for an adapter. Interfaces may be incompatible but the inner functionality should suit the need. The Adapter design pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients.

Structure

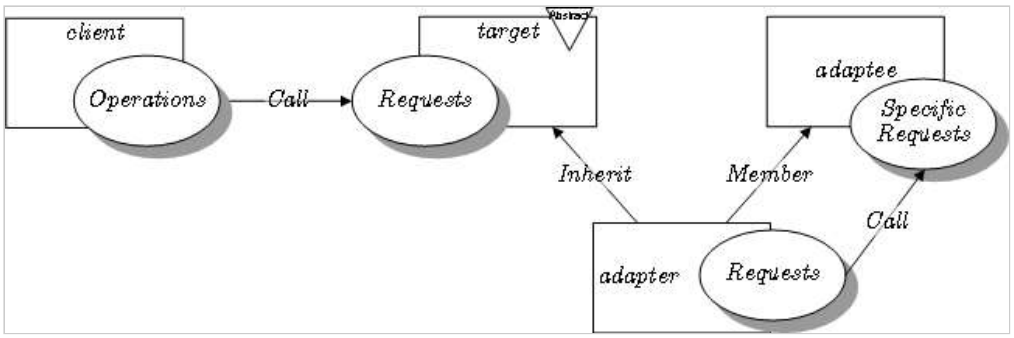
There are two types of adapter patterns:[1]

Object Adapter pattern

In this type of adapter pattern, the adapter contains an instance of the class it wraps. In this situation, the adapter makes calls to the instance of the wrapped object.



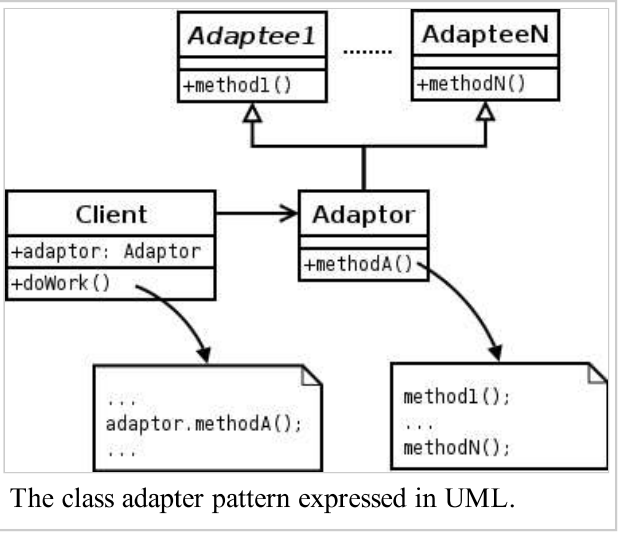
The object adapter pattern expressed in UML. The adapter *hides* the adaptee's interface from the client.



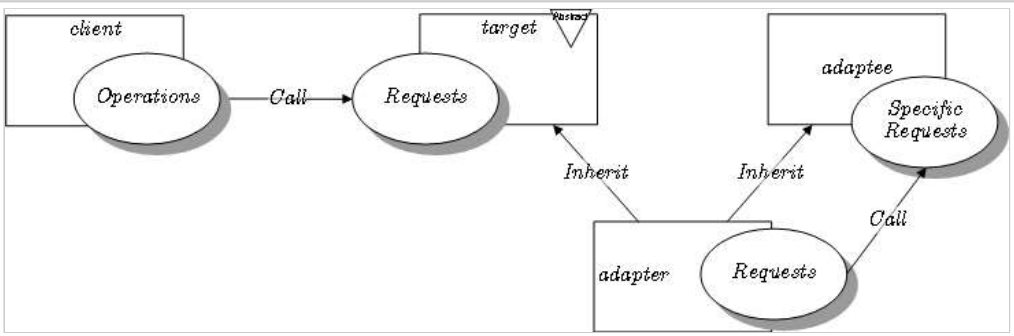
The object adapter pattern expressed in LePUS3.

Class Adapter pattern

This type of adapter uses multiple polymorphic interfaces implementing or inheriting both the interface that is expected and the interface that is pre-existing. It is typical for the expected interface to be created as a pure interface class, especially in languages such as Java (before jdk 1.8) that do not support multiple inheritance of classes.[1]



The class adapter pattern expressed in UML.



The class adapter pattern expressed in LePUS3

The adapter pattern is useful in situations where an already existing class provides some or all of the services needed but does not use the interface needed. A good real life example is an adapter that converts the interface of a Document Object Model of an XML document into a tree structure that can be displayed.

A further form of runtime Adapter pattern

There is a further form of runtime adapter pattern as follows:

It is desired for `classA` to supply `classB` with some data, let us suppose some `String` data. A compile time solution is:

```
classB.setStringData(classA.getStringData());
```

However, suppose that the format of the string data must be varied. A compile time solution is to use inheritance:

```
public class Format1ClassA extends ClassA {
   @Override
   public String getStringData() {
      return format(toString());
   }
}
```

and perhaps create the correctly "formatting" object at runtime by means of the Factory pattern.

A solution using "adapters" proceeds as follows:

(i) define an intermediary "Provider" interface, and write an implementation of that Provider interface that wraps the source of the data, `ClassA` in this example, and outputs the data formatted as appropriate:

```
public interface StringProvider {
    public String getStringData();
}

public class ClassAFormat1 implements StringProvider {
    private ClassA classA = null;

    public ClassAFormat1(final ClassA A) {
        classA = A;
    }

    public String getStringData() {
        return format(classA.getStringData());
    }

    private String format(String sourceValue) {
        //manipulate the source string into
        //a format required by the object needing the source object's
        //data
        return sourceValue.trim();
    }
}
```

(ii) Write an Adapter class that returns the specific implementation of the Provider:

```
public class ClassAFormat1Adapter extends Adapter {
   public Object adapt(final Object OBJECT) {
      return new ClassAFormat1((ClassA) OBJECT);
   }
}
```

(iii) Register the `Adapter` with a global registry, so that the `Adapter` can be looked up at runtime:

```
AdapterFactory.getInstance().registerAdapter(ClassA.class, ClassAFormat1Adapter.class, "format1");
```

(iv) In code, when wishing to transfer data from `ClassA` to `ClassB`, write:

```
Adapter adapter =
    AdapterFactory.getInstance()
        .getAdapterFromTo(ClassA.class, StringProvider.class, "format1");
```

```
StringProvider provider = (StringProvider) adapter.adapt(classA);
String string = provider.getStringData();
classB.setStringData(string);
```

or more concisely:

```
classB.setStringData(
    ((StringProvider)
            AdapterFactory.getInstance()
                .getAdapterFromTo(ClassA.class, StringProvider.class, "format1")
                .adapt(classA))
        .getStringData());
```

(v) The advantage can be seen in that, if it is desired to transfer the data in a second format, then look up the different adapter/provider:

```
Adapter adapter =
    AdapterFactory.getInstance()
        .getAdapterFromTo(ClassA.class, StringProvider.class, "format2");
```

(vi) And if it is desired to output the data from `ClassA` as, say, image data in `Class C`:

```
Adapter adapter =
    AdapterFactory.getInstance()
        .getAdapterFromTo(ClassA.class, ImageProvider.class, "format2");
ImageProvider provider = (ImageProvider) adapter.adapt(classA);
classC.setImage(provider.getImage());
```

(vii) In this way, the use of adapters and providers allows multiple "views" by `ClassB` and `ClassC` into `ClassA` without having to alter the class hierarchy. In general, it permits a mechanism for arbitrary data flows between objects that can be retrofitted to an existing object hierarchy.

Implementation of Adapter pattern

When implementing the adapter pattern, for clarity one can apply the class name `[ClassName]To[Interface]Adapter` to the provider implementation, for example `DAOToProviderAdapter`. It should have a constructor method with an adaptee class variable as a parameter. This parameter will be passed to an instance member of `[ClassName]To[Interface]Adapter`. When the clientMethod is called it will have access to the adaptee instance which allows for accessing the required data of the adaptee and performing operations on that data that generates the desired output.

```
public class AdapteeToClientAdapter implements Adapter {

    private final Adaptee instance;

    public AdapteeToClientAdapter(final Adaptee instance) {
         this.instance = instance;
    }

    @Override
    public void clientMethod() {
       // call Adaptee's method(s) to implement Client's clientMethod
    }

}
```

And a Scala implementation

```
implicit def adaptee2Adapter(adaptee: Adaptee): Adapter = {
  new Adapter {
    override def clientMethod: Unit = {
    // call Adaptee's method(s) to implement Client's clientMethod */
    }
  }
}
```

PHP

```
// Adapter Pattern example

interface IFormatIPhone
{
    public function recharge();
    public function useLightning();
}

interface IFormatAndroid
{
    public function recharge();
    public function useMicroUsb();
}

// Adaptee
class IPhone implements IFormatIPhone
{
    private $connectorOk = FALSE;

    public function useLightning()
    {
        $this->connectorOk = TRUE;
        echo "Lightning connected -$\n";
    }

    public function recharge()
    {
        if($this->connectorOk)
        {
            echo "Recharge Started\n";
            echo "Recharge 20%\n";
            echo "Recharge 50%\n";
            echo "Recharge 70%\n";
            echo "Recharge Finished\n";
        }
        else
        {
            echo "Connect Lightning first\n";
        }
    }
}

// Adapter
class IPhoneAdapter implements IFormatAndroid
{
    private $mobile;

    public function __construct(IFormatIPhone $mobile)
    {
        $this->mobile = $mobile;
    }

    public function recharge()
    {
        $this->mobile->recharge();
    }

    public function useMicroUsb()
```

```
    {
        echo "MicroUsb connected -> ";
        $this->mobile->useLightning();
    }
}

class Android implements IFormatAndroid
{
    private $connectorOk = FALSE;

    public function useMicroUsb()
    {
        $this->connectorOk = TRUE;
        echo "MicroUsb connected ->\n";
    }

    public function recharge()
    {
        if($this->connectorOk)
        {
            echo "Recharge Started\n";
            echo "Recharge 20%\n";
            echo "Recharge 50%\n";
            echo "Recharge 70%\n";
            echo "Recharge Finished\n";
        }
        else
        {
            echo "Connect MicroUsb first\n";
        }
    }
}

// client
class MicroUsbRecharger
{
    private $phone;
    private $phoneAdapter;

    public function __construct()
    {
        echo "---Recharging iPhone with Generic Recharger---\n";
        $this->phone = new IPhone();
        $this->phoneAdapter = new IPhoneAdapter($this->phone);
        $this->phoneAdapter->useMicroUsb();
        $this->phoneAdapter->recharge();
        echo "---iPhone Ready for use---\n\n";
    }
}

$microUsbRecharger = new MicroUsbRecharger();

class IPhoneRecharger
{
    private $phone;

    public function __construct()
    {
        echo "---Recharging iPhone with iPhone Recharger---\n";
        $this->phone = new IPhone();
        $this->phone->useLightning();
        $this->phone->recharge();
        echo "---iPhone Ready for use---\n\n";
    }
}

$iPhoneRecharger = new IPhoneRecharger();

class AndroidRecharger
{
    public function __construct()
    {
        echo "---Recharging Android Phone with Generic Recharger---\n";
        $this->phone = new Android();
```

```
        $this->phone->useMicroUsb();
        $this->phone->recharge();
        echo "---Phone Ready for use---\n\n";
    }
}

$androidRecharger = new AndroidRecharger();

// Result: #quanton81

//---Recharging iPhone with Generic Recharger---
//MicroUsb connected -> Lightning connected -$
//Recharge Started
//Recharge 20%
//Recharge 50%
//Recharge 70%
//Recharge Finished
//---iPhone Ready for use---
//
//---Recharging iPhone with iPhone Recharger---
//Lightning connected -$
//Recharge Started
//Recharge 20%
//Recharge 50%
//Recharge 70%
//Recharge Finished
//---iPhone Ready for use---
//
//---Recharging Android Phone with Generic Recharger---
//MicroUsb connected ->
//Recharge Started
//Recharge 20%
//Recharge 50%
//Recharge 70%
//Recharge Finished
//---Phone Ready for use---
```

See also

- Delegation, strongly relevant to the object adapter pattern.
- Dependency inversion principle, which can be thought of as applying the Adapter pattern, when the high-level class defines their own (adapter) interface to the low-level module (implemented by an Adaptee class).
- Shim
- Wrapper function
- Wrapper library

References

1. Freeman, Eric; Freeman, Elisabeth; Kathy, Sierra; Bates, Bert (2004). "Head First Design Patterns" (paperback). O'Reilly Media: 244. ISBN 978-0-596-00712-6. OCLC 809772256. Retrieved April 30, 2013.