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 from another interface.[1] It is often used to make existing classes work with others without modifying their source code.

Contents

Definition

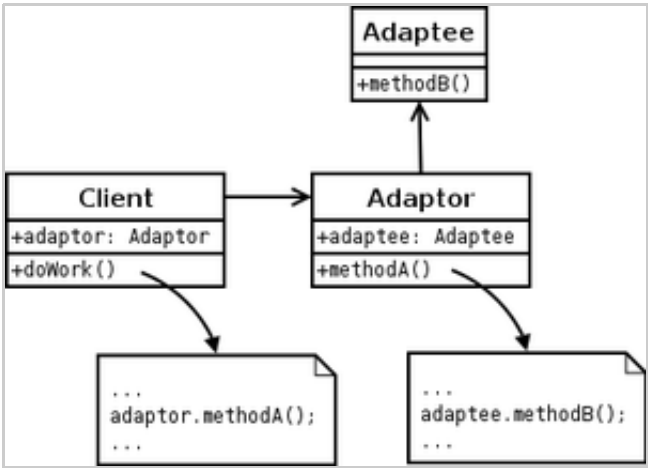
An adapter helps two incompatible interfaces to work together. This is the real world definition for an adapter. The adapter design pattern is used when you want two different classes with incompatible interfaces to work together. Interfaces may be incompatible but the inner functionality should suit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

Structure

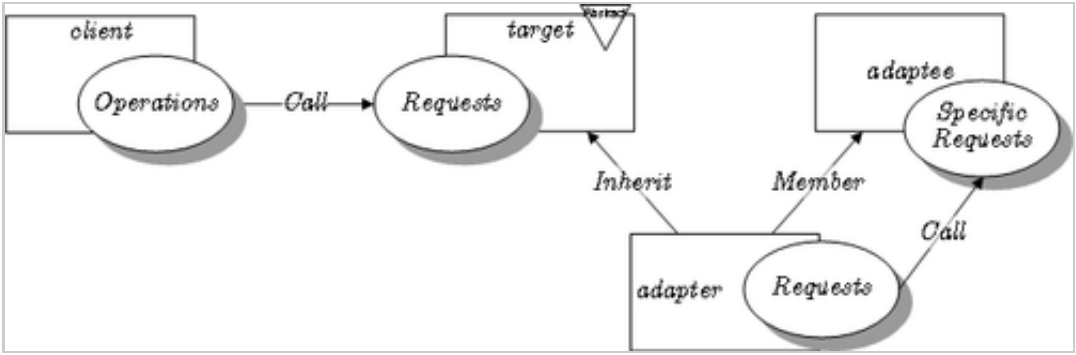
There are two types of adapter patterns:[1]

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.



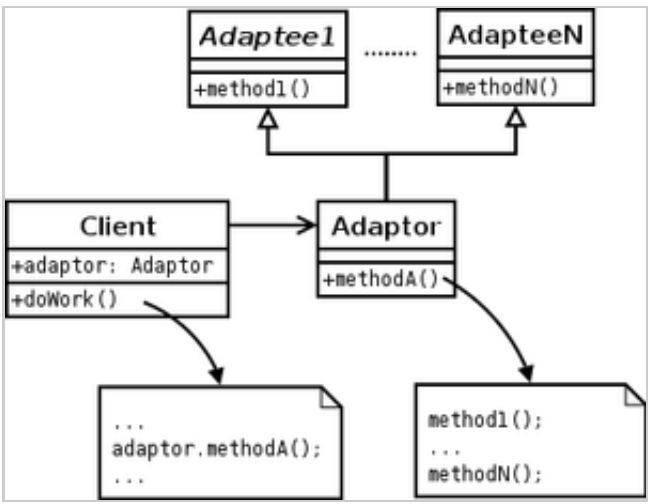
The object adapter pattern expressed in UML. The adaptor *hides* the adaptee's interface from the client.



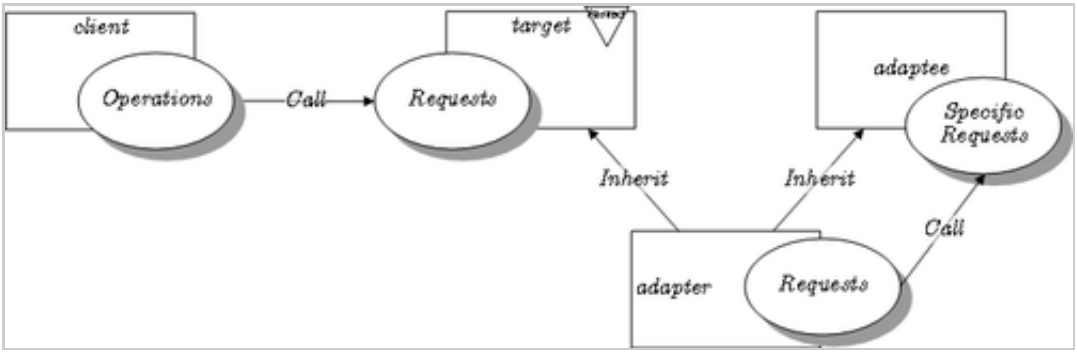
The object adapter pattern expressed in LePUS3.

Class Adapter pattern

This type of adapter uses multiple polymorphic interfaces to achieve its goal. The adapter is created by 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 that do not support multiple inheritance.[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 you need but does not use the interface you need. 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 link to a tutorial that uses the adapter design pattern is listed in the links below.

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:

```
Format1ClassA extends ClassA {
   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.toString());
    }
}
```

(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:

```
```

(iv) In your code, when you wish 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,
```

(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);
```

(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 use the class name `[AdapteeClassName]To[`**`Interface`**`]Adapter`, for example `DAOToProviderAdapter`. It should have a constructor method with adaptee class variable as parameter. This parameter will be passed to the instance member of `[AdapteeClassName]To[`**`Interface`**`]Adapter`.

```
public class AdapteeToClientAdapter implements Client {

    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
    }

}
```

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).
- Wrapper function
- Wrapper library

References

1. ^ ***a*** ***b*** ***c*** Freeman, Eric; Freeman, Elisabeth; Kathy, Sierra; Bates, Bert (2004). *Head First Design Patterns* (http://www.headfirstlabs.com/books/hfdp/) (paperback). O'Reilly Media. p. 244. ISBN 978-0-596-00712-6. OCLC 809772256 (https://www.worldcat.org/oclc/809772256). Retrieved April 30, 2013.