**Adapter Pattern**

Continuing our series of articles, taking each design pattern one by one, we move onto the Adapter pattern. This pattern is used a lot in Eclipse, allowing plug-ins to be loosely coupled, yet still be integrated into the Eclipse runtime.

**Adapters in the Real World**

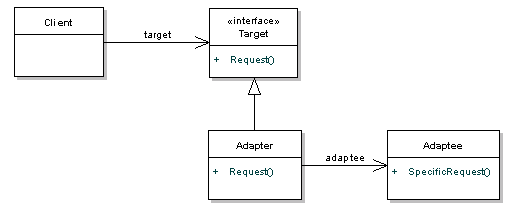
A real world analogy always helps with the understanding of a design pattern. The best example for the adapter pattern is based around AC power adapters. Say you're visiting Europe from the US, with your laptop, which expects a US power supply. To get your laptop plugged in, you're going to need to get a power adapter that accepts your US plug and allows it to plug in to the European power outlet. The AC adapter knows how to deal with both sides, acting as a middleman - this is the adapter pattern.

### The Adapter Pattern

The Adapter is known as a **structural** pattern,as it's used to identifying a simple way to realize relationships between entities. The definition of Adapter provided in the original Gang of Four book on DesignPatterns states:

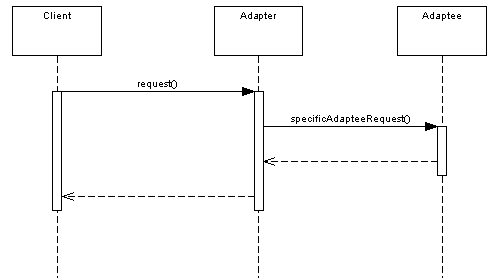
***Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.***

Let's take a look at the classic diagram definition of  the adapter pattern:



The **Target** interface defines the domain specific interface that the **Client** used, so the client collaborates with objects that implement the Target interface. On the other side of things, the**Adaptee** is the existing interface that needs adapting in order for our client to interact with it. The**Adapter** adapts the Adaptee to the Target interface - in other words, it translates the request from the client to the adaptee.

Let's take a look at the interactions in a sequence diagram:



In this example, as far as the Client is concerned it's just calling the **request** method of the Target interface, which the Adapter has implemented. In the background however, the Adapter knows that to return the right result, it needs to call a different method, **specificAdapteeRequest**, on the Adaptee.

**Note:***the pattern described here is the* *object* *adapter. There is a* *class* *adapter pattern, but you need multiple inheritance to use it. Seeing as Java doesn't support multiple inheritance, I'm going to leave this out.*

### Where Would I Use This Pattern?

The main use of this pattern is when a class that you need to use doesn't meet the requirements of an interface. As mentioned before, adapters are common across Eclipse plug-ins. For a particular object to contribute to the Properties view, adapters are used display the objects data. The view itself doesn't need to know anything about the object the it is displaying properties for.

**So How Does It Work In Java?**

The following example shows a simple implementation of the pattern. Consider that we have a third party library that provides sorting functionality through it's NumberSorter class. This is our **Adaptee**.

/\*\* This is our adaptee, a third party implementation of a

\* number sorter that deals with Lists, not arrays. \*/

public class NumberSorter{

public List<Integer> sort(List<Integer> numbers) {

//sort and return

return new ArrayList<Integer>();

}

}

Our **Client** deals with primitive arrays rather than Lists. *For the sake of this example, lets say we can't change the client to use Lists.*

int[] numbers = new int[]{34, 2, 4, 12, 1};

Sorter sorter = new SortListAdapter();

sorter.sort(numbers);

We've provided a Sorter interface that expects the client input. This is our **target**.

//this is our Target interface

public interface Sorter{

public int[] sort(int[] numbers);

}

Finally, the SortListAdapter implements our target interface and deals with our adaptee, NumberSorter

public class SortListAdapter implements Sorter{

@Override

public int[] sort(int[] numbers) {

//convert the array to a List

List<Integer> numberList = new ArrayList<Integer>();

//call the adapter

NumberSorter sorter = new NumberSorter();

numberList = sorter.sort(numberList);

//convert the list back to an array and return

return sortedNumbers; } }

While this example may be overkill, it illustrates how the adapter pattern can work.

### Watch Out for the Downsides

Some say that the Adapter pattern is just a fix for a badly designed system, which didn't consider all possibilties. While this is a fair point, it is an important part of a pluggable architecture.  It can also add a level of complexity to your code, making debugging more difficult.