Description

"Convert the interface of a class into another interface clients expect." [GoF] In other words, you define an API to use locally and then create implementations of that API that will convert the calls to another class's API. The main benefit of the adapter pattern is that it allows you to use classes that were not originally designed for your purposes. Please note that I use the wording "API" rather than "interface" (as GoF does) since the adapter pattern does not depend on an actual interface implementation (as used in Java/C#). What GoF meant by 'interface' is the method signatures that make up the interactions with some class (ie it's programming interface).   
  
When and when not to use

The adapter pattern is commonly used in applications where some package or third party application is being replaced with some other package. Rather than rewrite the application to use the new package's API, it's commonly less expensive to write an adapter to simply adapt the old API to the newer API. The application will simply call the old API via the adapter and the adapter will then manage the calls to the new package.   
  
The adapter pattern is common in component design/development also. However, the focus in component development is in integrating other components into the overall design, usually the adapter is simply another implementation to some strategy pattern that offers the adapting class's capabilities to the pattern. The designer will spot functionality in a different component that they view as useful for this component. The designer can then create an adapter to adapt that functionality to what the component expects. This allows the designer to quickly reuse other components in a design without having to ask for modifications to the other component or having to explicitly support the other component's API.   
  
An adapter pattern should not be used when the component categorically depends on the functionality of the other class. In other words, if the other component provides functionality that is critical to the component and cannot be replaced, then the component should call the API directly and not bother with an adapter class between them.   
  
Considerations

The biggest consideration when using the adapter pattern is in identifying the pattern appropriately. The adapter pattern is very similar to the bridge, decorator and proxy patterns. Each of these patterns has similar features to the adapter pattern – the difference is in the intent of the pattern. The intent of the adapter pattern is to simply convert a local interface to another class's interface. The other patterns intent is much different. The bridging pattern intent is to allow two implementations to vary independently. The decorator pattern's intent is to enhance another class and the proxy pattern intent is to be a placeholder for the other class. The intent of the class is what determines if the class is an adapter or not.   
  
Another consideration is when the adapting class is no longer an adapting class but a complex or composite class. Adapter classes can range from simple, one to one translation, to complex translation functionality. The complexity of the adapter class largely depends on how similar the two APIs are to each other. However, as more functionality and state information is added to an adapter class, the less like an adapter it is and the more like a complex class (that happens to use the other API as part of it's operations) it becomes. Like the first consideration, it all comes back to intent of the class. If the complexity is due solely to enable API-to-API translation, then the class can still be counted as an adapter. However, if the complexity starts to add functionality beyond the other's API, the more like a complex class the adapter becomes.   
  
Lastly, an adapter class should only adapt a single API. From the wording in GoF – "Convert the interface of a class…." If the adapter translates some of the API to one class's API and the rest of the API to a different class's API, the class is more like a composite class than an adapter class. This doesn't mean that you can't use more than one class in the translation process. What it means is that the source of the API should come for a specific class API.   
  
Implementation details  
The adapter pattern is very easy to implement. Simply identify the interface that your component will deal with and then identify the component's API that you want to use. Create a simple class (generally called "xxxAdapter") that implements your interface and executes the other API. Please note that it is reversed if you are providing an adapter to another component.   
  
Adapter as a Subclass

An adapter pattern can be implemented as a subclass to the class you wish to adapt. The subclass would then implement your own API and translate to the API of the super class. The approach presents several issues:

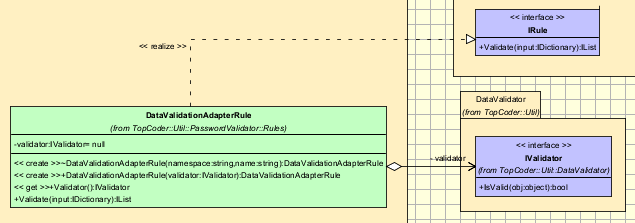
1. The adapter class is tied directly to the underlying class via inheritance. This creates a hard association between the components that must be considered when either component changes and is simply not possible if the adapter class needs to subclass a different tree.
2. The adapter class only works with the underlying class and can't use any subclasses or alternative implementations of the underlying class. In other words, the approach limits you to specifically what the underlying class has to offer. If the underlying class is later decorated or subclassed with additional functionality, the adapter class will not be able to use that functionality since it's tied to a specific class structure.

There are no examples in the TopCoder Software catalog of this type of implementation (at least that I could find). However, I believe the implementation description above is simple enough to understand without an example. Please feel free to post a message in the forum related to this article if you'd like to discuss implementation details.

Adapter as an Interface  
The adapter pattern is generally implemented as part of a strategy pattern. In fact, the best benefit comes when the adapting class is utilizing the strategy pattern on both sides. Combining the strategy pattern with the adapter pattern results in a powerful combination that:

* Allows the application to decide if it will use the Adapter and which Adapter implementation to use.
* Allows the application to decide the actual implementation the Adapter is converting to.

To implement an adapter as an interface, create a class that implements some local interface with a variable whose type is the other interface we are adapting to. The constructor of the adapter class will then take an instance of the other interface and store it into that instance field. Calls to the local interface can then call the other interfaces methods without regard to the actual implementation of the other interface.   
  
Let's take a look at an example – the .NET Password Validator component. The Password Validator component provides a centralized location to store enterprise password rules sets. A rule set provides a number of validation rules that implement the IRule interface (see Figure 1 below). This component provides a number of prebuilt rules for some of the most common password validations (such as minimum length, expiration time, common words, etc). Customers can provide their own implementations to the IRule interface to customize the component to their own local password validations. The designer realized that the customer may already have a number of pre-existing validations built with the .NET Data Validation component and may want to utilize those validations within the password validation component. An adapter class was created to adapt the IRule interface of the .NET Password Validator component to the IValidator interface of the .NET Data Validation component.   
  
As you see in Figure 1 below, the designer created an adapter class called DataValidationAdapterRule that implements the IRule interface. The adapter class's constructor takes an instance of the IValidator interface (either directly or through a configuration option) and stores it to an instance variable. When calls are made to the Validate method of IRule interface, the class 'converts' those calls to the IsValid method of the IValidator interface. This allows the component to utilize the functionality in another component that wasn't specifically designed for this component.

**Figure 1**  
  
[](http://www.topcoder.com/i/education/adapter1_lg.gif)