The Adapter Pattern

Graeme Geldenhuys

2009-01-20

The Adapter pattern is used to convert the programming interface of one class into that of another. We use adapters whenever we want unrelated classes to work together in a single program. We will implement a simple Adapter and will also show how the Adapter pattern can prevent you from being locked into a specific vendor's API, and why that is a good thing.

Adapters are everywhere

The adapter design pattern is very easy to understand because the real world is full of adapters. For example: if you wanted to use an American bought laptop in a European country, you would need an AC power adapter. You should also know what the adapter does. It changes the European wall outlet shape to the American shape that the laptop expects. See Figure 1 which summarises what the adapter does and how the real world adapter relates to what we want to do in software.

Not all adapters are simple. Some don't simply change the shape of the outlet. They may also change the wall outlet's power voltage to what the laptop requires. Some may have a fuse to protect the electronic device from a power overload.

The software based Adapter plays a similar role as the real world adapter. We could write a simple Adapter class that has the desired interface and then make it communicate with the class that has a different interface.

The Gang-of-Four book[1] has the following official explanation of what the intent is of the adaptor pattern: *"Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces."*

1 Eric Gamma, Richard Helm, Ralph Johnson & John Vlissides: Design Patterns: Elements of Reusable Object- Oriented Software. Addison-Wesley 1994. ISBN 0-201-63361-2, Page 139.

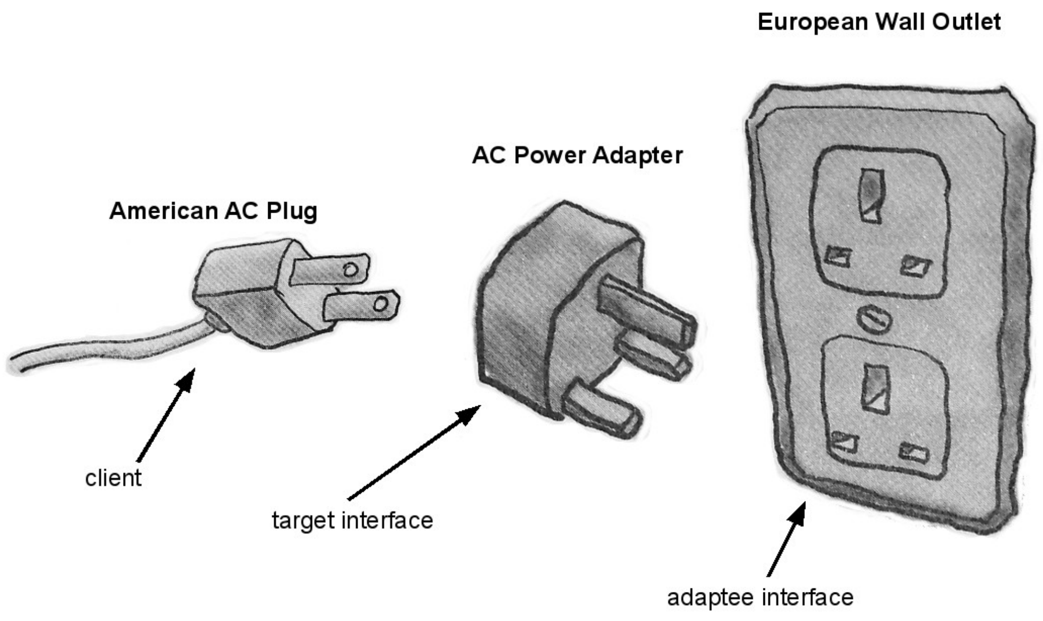


Figure 1: An example of a real world adapter.

There are two ways to accomplish this task: by inheritance and by object composition. In the Gang-of-Four book they call these *Class Adapters* and *Object Adapters*. Class Adapters use inheritance and works as follows. We derive a new class from the nonconforming one and add the methods we need to make the new class match the expected interface. Figure 2 shows a UML class diagram explaining the structure of a Class Adapter. In the Gang-of-Four book they use multiple inheritance which we can easily fake with Object Pascal's *Interfaces* language feature. The alternative adapter implementation known as Object Adapter, normally require more work to implement but is well worth the effort. We include the original class inside a new one and add the desired methods or properties to translate the calls within the new class. Figure 3 shows the UML class diagram of the Object Adapter.

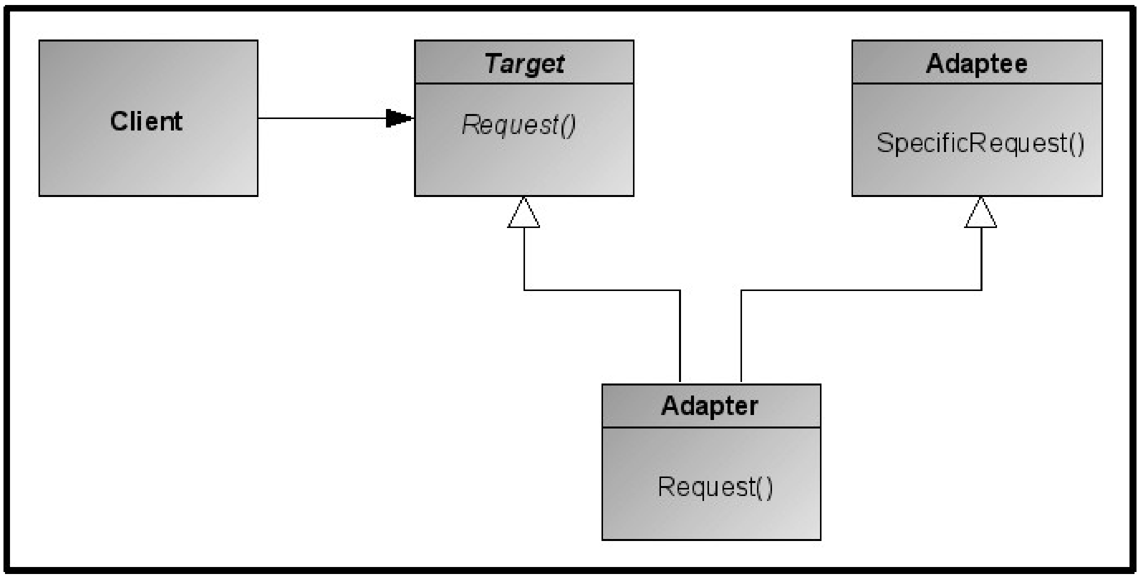


Figure 2: The Class Adapter uses multiple inheritance to adapt one interface to another.

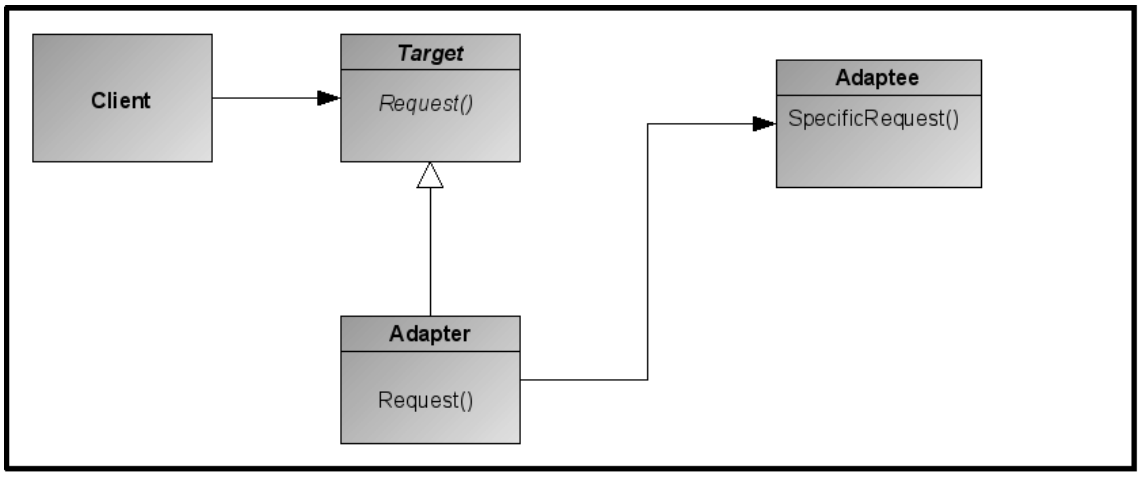


Figure 3: An Object Adapter relies on object composition.

Why do we need the Adapter?

With development tools like Delphi and Lazarus, we have more and more components to choose from when we write our software. As our software evolve over time, so do the components we use. There might be newer and better components available after a few years, or a vendor could have gone out of business and updates or support to those component have halted.

When we started our project, we would have had to make a choice of what components we wanted to use. For example: for SQL database access we could have chosen between BDE controls (TDatabase and TQuery) or something that talks directly to the database like IBObjects for Interbase & Firebird or SDAC for Microsoft SQL Server or DOA for Oracle access. The latter three giving us better performance and more specific features, or the BDE which is a more generic interface.

The same choices could have been made between encryption components, compression components, XML parsers etc. We have no guarantee that the choice we made at the time will be the correct or best choice in the future.

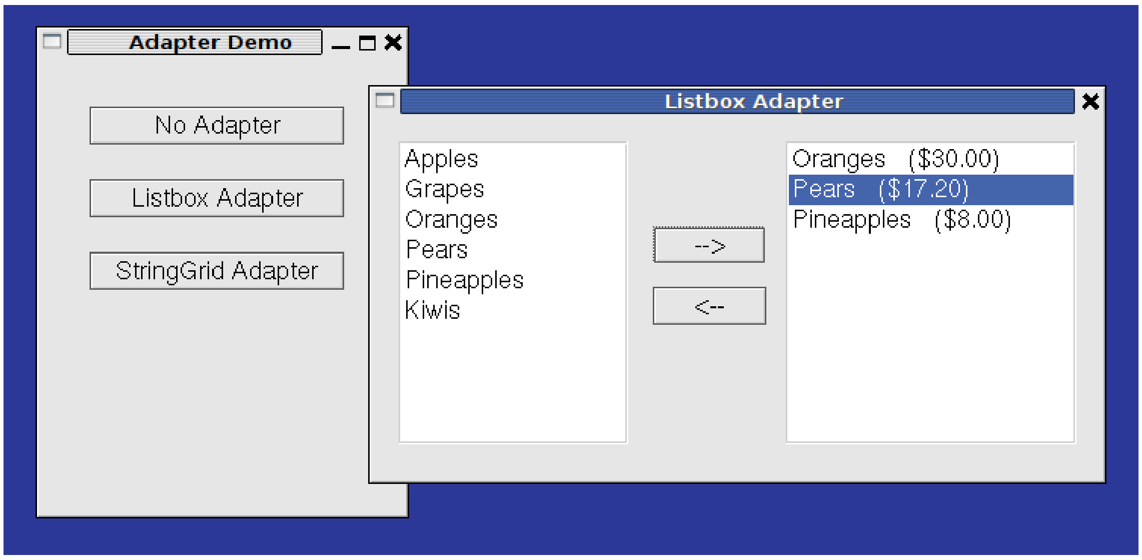
Dropping components on a form or data module will work fine for now, but what if circumstances change in the future and we need to change to a new vendor's components. Our existing vendor could have gone our of business or our company could have merged with another. There are various reasons for us needing to change our components. Now imagine the huge task of having to manually search and replace components in the data modules and various forms in our project. Then we would also have to change our code to work with the new interface of the new components. That would very likely be a huge undertaking and a costly one at that.

The concept of the Adapter pattern is quite simple. We could write a class that has the interface we would like and in return the adapter will communicate with the class that has a different interface. So if we now want to change from one vendor's components to another, all that has to change is for the adapter to talk to a different vendor's components. Our application still only communicates with the adapter, and it's interface has stayed the same.

Initially this would be a lot of work to code up, but as with most design patterns, there are huge rewards. I will start off by showing you a simple example so you can get an understanding of how the Adapter works, then later I will talk about more advanced examples.

Moving Data Between Lists

Lets consider a simple program that allows us to move data between lists. The one list will contain product names only. As we move the data item to the second list, we will get a more detailed view of the associated item.



Apples
Grapes
Oranges
Pears
Pineapples
Kiwis

Oranges ($30.00)
Pears ($17.20)
Pineapples ($8.00)

Figure 4: A simple demo to display item details.

To add items to the listbox on the right, we click the button with the right arrow. The button's event handler looks as follows.

```
procedure TNoAdapterForm.btnAddClick(Sender: TObject);
var
  i: integer;
  obj: TProduct;
begin
  i := ListBox1.ItemIndex;
  if i < 0 then
    Exit; //==>
  obj := TProduct(ListBox1.Items.Objects[i]);
  ListBox2.Items.Add(
      Format(cLBDisplay, [obj.Name, obj.Price]));
  ListBox1.ItemIndex := -1;
end;
```

We first make sure that an item was highlighted in the left Listbox. We then extract a object reference from the `Items` property of the Listbox. We then use the `cLBDisplay` constant and format a string which we then add to the Listbox on the right. Afterwards we deselect the previously highlighted item in the left Listbox.

Nothing very complicated, but it is a bit awkward having to use very `TListBox` specific properties and methods. In this small amount of code we are very much tied into the `TListBox` design by referring to properties and methods like: `ItemIndex`, `Items.Objects[]` array and `Items.Add()` method. This means if we ever wanted to change our detailed display component to something other than a `TListBox`, we have a lot of code that needs to change.

What we would prefer is a class that hides these complexities and `TListBox` dependencies and adapts the interface to something we would like. We are looking for a simpler interface that doesn't surface the internal properties and methods of the display components we use.

To solve this problem, we create a `TListAdapter` class which gives us a simpler interface and internally operates on a Listbox instance. The class declaration of the TListAdapter is shown below.

```
TListAdapter = class(TObject)
private
  FListBox: TListBox;
public
  constructor Create(lb: TListBox);
  procedure Add(s: string);
  function SelectedIndex: integer;
  procedure Clear;
  procedure ClearSelection;
end;
```

We simply pass in the target ListBox instance as a parameter to the constructor. The rest of the methods operate on the ListBox instance, but simplifies the interface. The following code shows how we can slightly improve our code. The `TListAdapter` instance is stored in the `lstNew` variable.

```
procedure TListAdapterForm.btnAddClick(Sender: TObject);
var
  i: integer;
  obj: TProduct;
begin
  i := ListBox1.ItemIndex;
  if i < 0 then
    Exit; //==>
  obj := TProduct(ListBox1.Items.Objects[i]);
  lstNew.Add(Format(cLBDisplay, [obj.Name, obj.Price]));
  ListBox1.ItemIndex := -1;
end;
```

If we are always going to do the same string formatting in the new Listbox, we can simplify the code even further. Instead of calling `Format(...)` and then calling `lstNew.Add(...)` with the resulting string as a parameter, we can actually pass in the `TProduct` instance directly. The adapter class can then do the string formatting for us. Here is the improved `TListAdapter.Add()` method...

```
procedure TListAdapter.Add(AProduct: TProduct);
begin
  FlistBox.Items.Add(
    Format(cLBDisplay, [AProduct.Name, AProduct.Price]));
end;
```

...and the new `btnAddClick` event handler.

```
procedure TListAdapterForm.btnAddClick(Sender: TObject);
var
  i: integer;
  obj: TProduct;
begin
  i := ListBox1.ItemIndex;
  if i < 0 then
    Exit; //==>
  obj := TProduct(ListBox1.Items.Objects[i]);
  lstNew.Add(obj);
  ListBox1.ItemIndex := -1;
end;
```

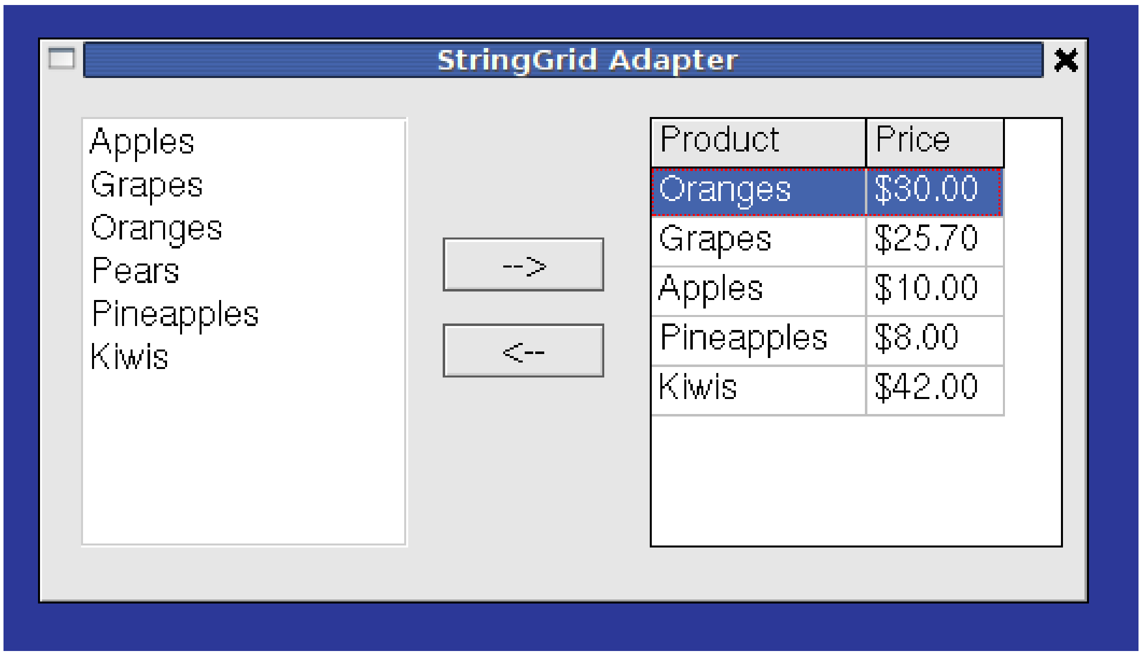
So lets summarise what we have done so far. We have created an Adapter class that contains a ListBox reference. The Adapter class has also simplified how we use the ListBox, without revealing any ListBox specific properties or methods. Next I will show how we can apply a similar adapter to a StringGrid component. We will then use the StringGrid to display the more detailed view of our products. This will also visually improve how our detailed product data is displayed, using grid columns to align values. The important thing is also that we want to keep the same interface we created for the ListBox, but use the StringGrid component instead.

Here follows the implementation for the `TGridAdapter` class.

```
constructor TGridAdapter.Create(AGrid: TStringGrid);
begin
  inherited Create;
  FGrid := AGrid;
  FGrid.Options := FGrid.Options + [goRowSelect];
  FGrid.FixedCols := 0;
  FGrid.RowCount := 1;
end;

procedure TGridAdapter.Add(AProduct: TProduct);
var
  i: integer;
begin
  i := FGrid.RowCount;
  FGrid.RowCount := FGrid.RowCount+1;
  FGrid.Cells[0, i] := AProduct.Name;
  FGrid.Cells[1, i] := Format('%m', [AProduct.Price]);
end;

function TGridAdapter.SelectedIndex: integer;
begin
  Result := FGrid.Row;
end;

procedure TGridAdapter.Clear;
begin
  FGrid.Clear;
end;

procedure TGridAdapter.ClearSelection;
begin
  if SelectedIndex < 1 then
    Exit; //==>
  FGrid.DeleteColRow(False, SelectedIndex);
end;
```

As you can see, we have the exact same interface as we did for the Listbox Adapter. The only interface change that we made here is that we have to pass in a `TStringGrid` instance into the constructor. We did achieve our goal though - not wanting to change the `btnAddClick` and `btnRemoveClick` event handler code. We can swap out the adapter classes, switching between a ListBox or StringGrid without changing the Add or Remove button event handlers.

The adapter is really an easy pattern to implement. We could continue and adapt a TTreeview or any other component, but I think you get the idea. Please note that accompanying this article, on the Toolbox DVD, is the complete source code for the TListAdapter and TGridAdapter including the Lazarus demo project.



Apples
Grapes
Oranges
Pears
Pineapples
Kiwis

Product	Price
Oranges	$30.00
Grapes	$25.70
Apples	$10.00
Pineapples	$8.00
Kiwis	$42.00

Figure 5: The same item details but with a StringGrid Adapter.

Other uses and improvements

Now that we have a working example, the next question people normally ask me is: "Where or how do we create the Adapter instance?" There are various ways this can be done.

The simplest method is to specify the appropriate concrete class directly in the code. This is what we have done in the demo project. The down side of using this method is that it locks us into using a specific class from design-time. Plus we will not have the ability to swap adapters at runtime.

The second method is a slightly more flexible method, by using *class references*. This will make it easier to vary our programs behaviour. Again, we will not have the flexibility of runtime changing, but it will make it a lot easier for compile-time changing. At least this method will allow you to switch behaviour with a single line code change.

The third method and by for the most flexible of the lot is to use the Factory Pattern[2]. I have already covered this pattern in a previous Toolbox issue, so will not go into more details. To recap, the Factory can be implemented in Object Pascal as a `TList` of objects that map a string that identifies a class to a class reference that can be used to create an instance of the class. Using this method we can vary the applications behaviour at design-time and runtime.

For more advanced examples of the Adapter pattern, I would recommend you look at the tiOPF (TechInsite Object Persistence Framework) project[3]. The tiOPF uses OOP extensively and implements a lot of different design patterns. In the tiOPF the Adapter pattern is used for the following tasks - to name but only a few. The first example is where it wraps the compression library ZLib, which is included with Delphi and Free Pascal, resulting in a much easier interface for us to use. It also uses the Factory pattern

2 Factory Pattern. Toolbox issue 5/2008
3 tiOPF project: http://tiopf.sourceforge.net/

so that the compression algorithm can be changed at runtime. Another example is where the tiOPF wraps various encryption algorithms and gives them a uniform and easy to use interface. Again the Factory pattern is used so that the encryption algorithms can be changed at runtime.

Probably the most important usage of the Adapter in tiOPF is where the Adapter wraps data access components. tiOPF supports over ten different persistence layers all using different data access components from various vendors. The framework is so flexible that you can even switch the persistence layers at runtime - a near impossible task with component-on-form style of development. To find out how the tiOPF has accomplished this, you can study the starting unit for database access called `tiQuery.pas` where it defines various virtual abstract classes to handle navigation and field access - similar to what `TDataset` does.

Summary

In summary, we have studied how the Adapter pattern works and how it can be used in a simple project. We have also explored the more advanced usage options like wrapping data access components and preventing vendor lock-in. We have also mentioned how the Adapter pattern can be used with other design patterns like the Factory pattern. I hope you found this information informative and that you now have the knowledge to use yet another design pattern in your projects. I hope you find many uses for the Adapter Pattern.