

Tutorial – Refactoring

Spirit Du
October 13th, 2009 Ver. 1.4

Contents

About This Document	1
Tutorial – Refactoring	2

About This Document

Sometimes, a program does not have a good design. Or the implementation does not meet the original design. Or, a legacy system is to be maintained. These are the situations that a program needs to improve its design. In software engineering, a good mechanism, called refactoring, can be used to improve the design of a program. This tutorial introduces four refactoring methods offered in Visual Studio 2008: renaming of classes, namespaces, variables and functions, extracting methods, and formatting codes. Note that refactoring is quite different from the Find/Replace function, because Find/Replace doesn't know the type of the replaced object, and can not replace all instances of the same type in different files. But, refactoring is based on Abstract Syntax Tree obtained from compiler, and thus can do better than Find/Replace.

Tutorial – Refactoring

Step 1 Setup Environment

Create a new console project. Please use a default name, and then create a new class: MainEntry.

Copy the following code to overwrite all the contents of the new class. **Don't** key in the following code by yourself that is meaningless in this tutorial.

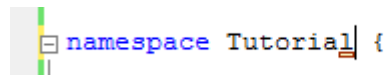
```
using System;
using System.Collections.Generic;
using System.Text;
namespace Project1 {
    public delegate void WeatherUpdated();
    class WeatherData {
        protected float _temp;
        protected float _humidity;
        protected string _location;
        protected event WeatherUpdated _stations;
        #region Properties
        public WeatherUpdated WearherStations {
            get { return _stations; }
            set { _stations = value; }
        }
        public float KelvinDegrees {
            get { return _temp; }
            set { _temp = value; }
        }
        foreach (WeatherUpdated s in _stations.GetInvocationList()) s();
        public float Humidity {
            get { return _humidity; }
            set { _humidity = value; }
        }
        foreach (WeatherUpdated s in _stations.GetInvocationList()) s();
        public string Location {
            get { return _location; }
        }
        #endregion
        public WeatherData(string location, float temp, float humility) {
            _location = location;
            _temp = temp;
            _humidity = humility;
        }
        abstract class WeatherStation {
            protected WeatherData _weather;
            protected abstract void showWeather();
            class TaiwanWeatherStation : WeatherStation {
                public TaiwanWeatherStation(WeatherData weather) {
                    _weather = weather;
                    _weather.WearherStations += new WeatherUpdated(showWeather);
                }
                protected override void showWeather() {
                    System.Console.WriteLine("Taiwan Weather Station:");
                    System.Console.WriteLine("The weather at " + _weather.Location);
                    System.Console.Write("Temperature: " + (_weather.KelvinDegrees - 273) + "C, ");
                    System.Console.WriteLine("Humidity: " + _weather.Humidity + "%\n");
                }
            }
            class NewYorkWeatherStation : WeatherStation {
                public NewYorkWeatherStation(WeatherData weather) {
                    _weather = weather;
                    _weather.WearherStations += new WeatherUpdated(showWeather);
                }
                protected override void showWeather() {
                    System.Console.WriteLine("New York Weather Station:");
                    System.Console.WriteLine("The weather at " + _weather.Location);
                    System.Console.Write("Temperature: " + (((_weather.KelvinDegrees - 273) * 9 / 5) + 32)
                        + "F, ");
                    System.Console.WriteLine("Humidity: " + _weather.Humidity + "%\n");
                    if ((_weather.KelvinDegrees - 273) <= 0 && _weather.Humidity > 50)
                        System.Console.WriteLine("It may start to snow!");
                }
            }
        }
        public class MainEntry {
            public static void Main() {
                WeatherData weather = new WeatherData("Los Angeles", 275.0f, 40.0f);
                TaiwanWeatherStation taiwan = new TaiwanWeatherStation(weather);
                NewYorkWeatherStation newYork = new NewYorkWeatherStation(weather);
                weather.KelvinDegrees = 274.0f;    weather.KelvinDegrees = 272.0f;
                weather.Humidity = 60.0f;
                System.Console.Read();    }
        }
    }
}
```

Step 2 Formatting Codes

Well, the copied code looks terrible, *if the option in “Options -> Text Editor -> Formatting -> Adjust indentation on paste” is not checked*. But it can be formatted by following the instructions: *Edit -> Advanced -> Format Document*, or press *Ctrl + E and then D* Key.

Step 3 Renaming of Namespace

The default namespace is “Project1” which will be changed to “Tutorial” in this step. There are several ways to rename and all of them can be used in renaming namespaces, classes, variables, and functions. Here, the first way is edit it directly and wait for prompt, a small red rectangle under the new name like Figure 1. Move your mouse over the prompt, a popup menu will show and choose the first suggestion Figure 2. Then the namespace is renamed no matter in which file; you can try it in your homework.



```
namespace Tutorial {
```

Figure 1 The refactor prompt

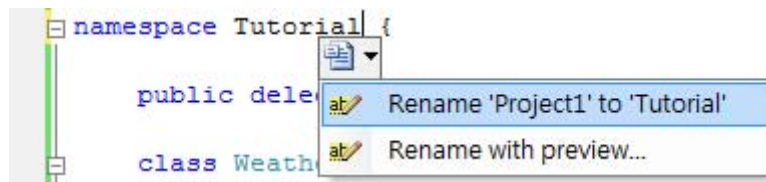


Figure 2 The refactor popup menu

Step 4 Renaming of Classes

In this step, the class *WeatherData* will be changed to *WeatherSensor*. Right click on the declaration, and a popup menu (Figure 3) is shown for you. Choose *Rename* option, and a rename dialog (Figure 4) will display. Key in the new name: *WeatherSensor*. Because the “Preview reference changes” option is checked, a preview dialog will show you what to change. Click *Apply* and all references will be changed to the new name.

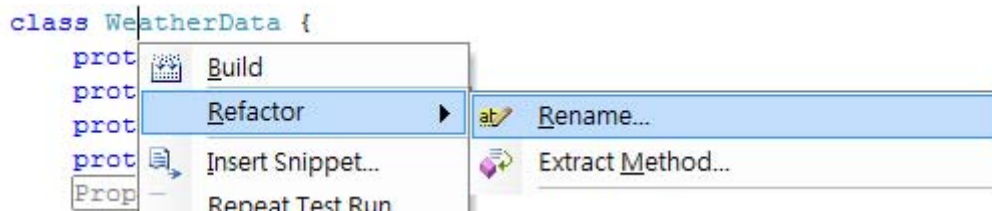


Figure 3 Right click to trigger the popup menu

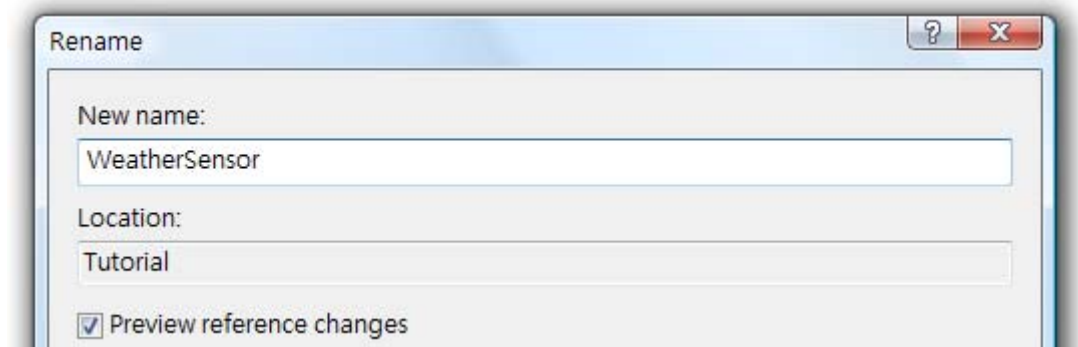


Figure 4 Rename Dialog

```
abstract class WeatherStation {
    protected WeatherSensor _weather;
    protected abstract void showWeather();
}

class TaiwanWeatherStation : WeatherStation {
    public TaiwanWeatherStation(WeatherSensor weather) {
        _weather = weather;
    }
}
```

Figure 5 Class renamed result: all references are changed, too.

Step 5 Renaming of Variables

The same as Step 4, variable names can be changed by refactor. The third way is the function key “F2”. Move the editing cursor (Figure 6), not pointer, on the member variable: `_temp`, and then press F2 key. The rename dialog appears again for you to rename. Rename it to `_temperature`. If you don’t want to see what to change, you can cancel the “Preview reference changes” in the rename dialog.

```
class WeatherSensor {
    protected float _temp;
    protected float _humidity;
    protected string _location;
}
```

Figure 6 Move the editing cursor on `_temp`

```
public float KelvinDegrass {
    get { return _temperature; }
    set {
        _temperature = value;
    }
}
```

Figure 7 Variable renamed result (Part)

Step 6 Renaming of Functions

The last way to rename is the Refactor menu in the menu bar. Select the `showWeather()` method, and then click the Refactor menu (Figure 8) to execute Rename function. In C#, the naming style for function name is usually starting with a capital letter in each word, so rename it to `ShowWeather()`.

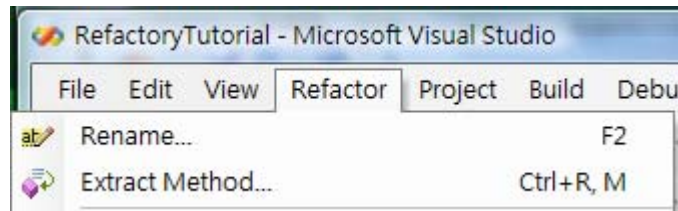


Figure 8 Refactor menu

```
abstract class WeatherStation {
    protected WeatherSensor _weather;
    protected abstract void ShowWeather();
}

class TaiwanWeatherStation : WeatherStation {
    public TaiwanWeatherStation(WeatherSensor weather) ...
    protected override void ShowWeather() ...
}

class NewYorkWeatherStation : WeatherStation {
    public NewYorkWeatherStation(WeatherSensor weather) ...
    protected override void ShowWeather() ...
}
```

Figure 9 The function renamed result (Part).

Step 7 Extracting Methods

At last, there is one more thing convenient: extracting methods. If you start with a not well design, you may find that there exist some redundant codes to do the same job. In this case, you can extract the codes into a function call, and then remove the redundant by calling the new method. In the tutorial, the notify method is removed purposely and the codes are redundant in the setter of KelvinDegrees and Humidity property. Select the redundant codes in the setter of Humidity, like Figure 10. Then, you click right mouse button to trigger the popup menu (Figure 11). In the popup menu, choose "Extract Method" and a dialog (Figure 12) will appear to ask you give the new method a new name. Key in the new name: NotifyWeatherStations, and then click OK. **However, only the selected codes will be replaced by the new method call; you need to replace the redundant codes in the KelvinDegrees property by yourself. And the result should like Figure 13. Finally, you can execute the program to see what it does!**

```
set {
    temperature = value;
    foreach (WeatherUpdated s in _stations.GetInvocationList())
        s();
}
```

Figure 10 Select the codes to extract

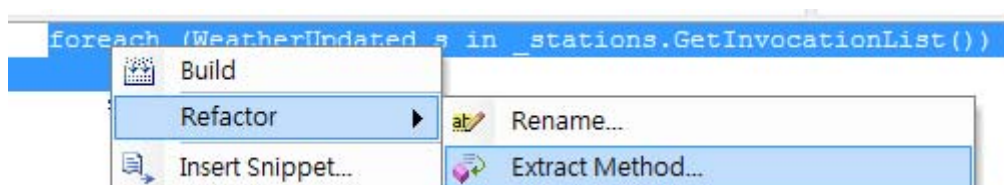


Figure 11 Popup menu: choose Extract Method

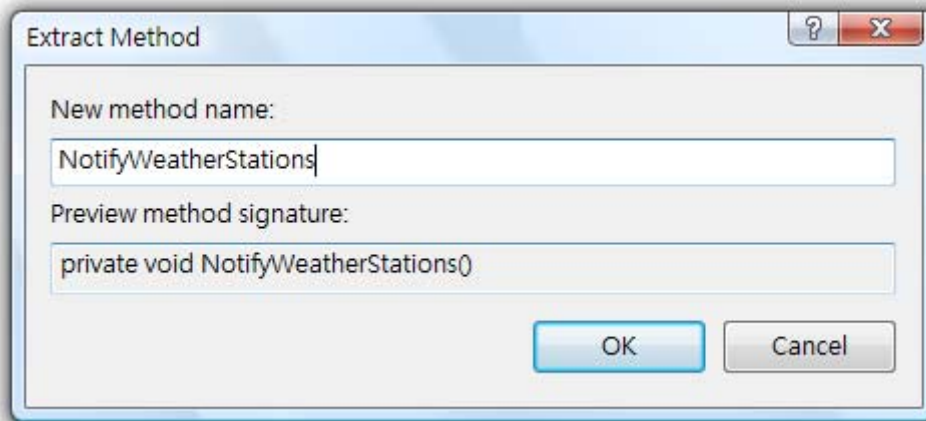


Figure 12 Extract method dialog

```
public float Humidity {  
    get { return _humidity; }  
    set {  
        _humidity = value;  
        NotifyWeatherStations();  
    }  
}  
  
private void NotifyWeatherStations() {  
    foreach (WeatherUpdated s in _stations.GetInvocationList())  
        s();  
}
```

Figure 13 Extract method result

- The End -