Republic of Moldova

Technical University of Moldova

FCIM

**Report**

*Laboratory Work nr.3*

On **IPP**

**Topic:** *“The MVC pattern”*

Done by : Draghici V.

Verified by assis. lec. : Zarea I.

**Goals**

To familiarize with MVC pattern and to implement it on a specific language chosen by the student.

**Requirements**

Basic knowledge about MVC pattern, medium knowledge of a programming language to implement it.

**Objectives**

* Create 3 objects that have the responsabilites of the Model, View and Controller.
* Create a Model. Make sure to identify if it is a Domain Model or a Transaction Script
* Specify what the responsabilites of your Controller are and whether is a Page Controller or a Front Controller
* What are the responsabilities of the View ? Is it a Template View of a Transform View.
* Write a lot of explanations. Maintain the code as simple as possible and keep the explanations as clear as you can.

**Practical part**

The program that implements the requirements mentioned above is written in C# and contains 5 classes: *View, Controller, Model, TemperatureConverter, DataCheck.* As a general description the program converts temperature from Celsius to Fahrenheit and viceversa. A screenshot of the program can be found on Appendix A, figure 1. The program is a simple console application. The source code of the application can be found on: ***github repository*** Below there is a description of each class:

**View –** this class represents the entry point in the application, it holds the *main()* method. In general terms, this class represent the *View* from *MVC* pattern. It contains approximately no logic, it just interacts with the user. Analising the *Template View* and *Transform View* I think that in this program is implemented *Template View.* This two patterns mentioned are best applicable in web development, *Template View* is used to compose dynamic web pages using some static templates that are unchangeable and *TransformView* is used to process domain data element by element and transforms in into HTML. In my case the language is C#, but still there is some transformation that occur depending on the value of parameters transmitted to *Controller*. So, for menu items *Celsius to Fahrenheit* and *Fahrenheit to Celsius* a I have some special arguments *string temperature* and *temperature type* that are able to make my *View* to be dynamic and to change a little the interface of the view.

//converts Celsius to Fahrenheit

if (menu == 1)

{

string CelsiusTemperature = null;

System.Console.WriteLine("\t\tCelsius to Fahrenheit");

System.Console.WriteLine("- - - - - - - - - - - - - - - - - - - - - - - - - - -");

System.Console.Write("Celsius = ");

CelsiusTemperature = System.Console.ReadLine();

System.Console.WriteLine("Fahrenheit = {0}",controller.ConvertTemperature("Celsius",CelsiusTemperature));

System.Console.WriteLine("- - - - - - - - - - - - - - - - - - - - - - - - - - -");

}

So, here the arguments *“Celsius”* and *CelsiusTemperature* for the *controller.ConvertTemperature(“Celsius”, CelsiusTemperature)* method call change the info received from the *Controller.* In order to get the converted temperature value *View* is interacting with *Controller* by only one method call. I tryied all my best to make the coupling of the modules as less as possible. So, for both the *Celsius to Fahrenheit* and *Fahrenheit to Celsius* menu items is called the same function from the *Controller* modulenamed *ConvertTemperature()* that takes as arguments the type of input temperature as *Celsius* or *Fahrenheit*  and the actual temperature that is introduced by the user. This two arguments are of type string. For the first one is pretty clear why string, because its much easier to understand what is happening if you see a proper string argument rather then an int which you have no idea what that can mean. With the second argument is a little trick. It is easy to become confused and to ask why I pass a argument that is a number like a string variable. And the answer is a very simple one. The programmers should not trust the users. So, the user, with an intention or not can input invalid data as temperature value, like char or special characters. Passing string arguments, the data checking will be moved from *View* to *Controller*. The *View* module is very simple one, some outputs to the screen, some read lines and calling just one function from the *Controller* that is done as coupless as possible. When calling the *ConvertTemperature()* method from *Controller* was used the *Tell Don’t Ask* principle. So, the view don’t asks the *Controller* if you can please convert me this from this, it just says convert me this, and I don’t care if there is something wrong, I have no idea, I just want the result, I have no logic, I am smart enough to pass the user input to *Controller* and pass again the *Controller* result to user again, and thats all. Thanks to using the string arguments, the *View* has no idea that the user inputed a valid data or mumble jumble. The result obtained from *Controller* is a string too, and the *View* have no idea what that does mean, it just outputs to the screen, its the job of the *brain* to do all the job, the *View*  should be simple and beautiful. (O vorba des intilnita pe retelele de socializare: *Tata trebuie sa lucreze, dar mama trebuie sa fie frumoasa ☺*, ceia ce nu e aplicabil parintilor nostri). The only checking that the *View* is doing is making sure that the user chosed the correct menu item. In case the user chosed an indefinite menu item, *View* is posting an error message and asks the user to choose again. There was an idea to make *View* to be able to interact with *Model* directly. But very soon I realised that this was a bad idea. I don’t want that *View* will know something about the *Model*. This is not secure, and this implies that *View* should contain allready some logic in interacting with *Model,* but I don’t need this. I choose to pass some strings to the *Controller* instead of passing an object. I prefer the *Controller* to do all the job or better said to manipulate with data.

**Controller –** it represents the brain of the application. So, *Controller* receives two arguments as string from *View*, and it must perform some data checking and temperature convertions. And plus to this the data is stored in a *Model.* But more about the *Model,* and stuff related to this will be represented bellow. The *Controller* interacts with *DataCheck* class which checks if the received string is ok and with *TemperatureConverter* class which in case the data is fine it converts the temperature. The result is stored in *Model*. As was mentioned before I tryied to maintain this module as simple as possible and coupless in relation with other classes. I think the coupling is thin because, in order to check data it just says to *DataCheck* object to replay a simple response, if the user data is fine or not. And in interacting with *TemperatureConverter* it somehow makes things easier by decomposing the problem. So, the *Controller* decides which method from *TemperatureConverter* it should run, depending on one argument. So, here it takes one responsability up to him to decide what to do, and this somehow can determine to be a little bit of coupling with *TemperatureConverter*, but I decided that this must do the *Controller.* Another plus of *Controller* to make decisions is that the *DataCheck* component is much easier to maintain, use and understand. Bellow is shown what the *Controller* does in order to convert the temperature from Celsius to Fahrenheit:

//if input temperature is in Celsius

if (type.Equals("Celsius"))

{

double convertedTemperature = 0;

temperature = Convert.ToDouble(input);

convertedTemperature = TempConv.ConvertCelsiusToFahrenheit(temperature);

model.setCelsius(temperature);

model.setFahrenheit(convertedTemperature);

retVal = Convert.ToString(convertedTemperature);

}

The *Controller* of this application is of type of *Front Controller* because a single component (*ConvertTemperature()* method) is responsible to process all the *View* requests.

**Model –** here the data is stored. According to my program, there is only a little need of this module. But, in order to implement *MVC* I must include a model too. This module can be helpful when adding new functionality to the application. At the moment it just stores the data (temperature in Celsius and Fahrenheit) and have the logic of simple manipulation with it like getting and setting *Model* attributes. And thats all. The *Model* is even more simple than *View* is. It has a very poor logic. The *Controller* uses it to store the Fahrenheit and Celsius temperature values, but after storing values it never uses them. Someone can ask why the program don’t use *Model* more active, and the answer is simple: because of the application specifics. The *Model*  will be very helpfull when I will add some new functionality to the application, but at the moment si good as it is. Because it has a very poor logic, I can say that it represents an *Transaction Script*, because it represents a pure representation of data. But, *Transaction Script*  is situated between *Presentation* and *DataSource*, but my *Model* is just an object that stores some info. On another hand I can think that is a *Domain Model*, becaus it incapsulates what we know about application. I am not sure, the *Model* in this application is vague and I choose that is a *Transaction Script,* but I am not 100% sure, not even 70%. Bellow you have a shortened representation of what *Model* looks like:

class Model

{

//Temperature in Celsius

private double Celsius;

//Temperature in Fahrenheit

private double Fahrenheit;

public Model()

{

Celsius = 0;

Fahrenheit = 0;

}

public void setFahrenheit(double F)

{

Fahrenheit = F;

}

public double getFahrenheit()

{

return Fahrenheit;

}

}

**DataCheck** – this object communicates with other objects using just one method call *CheckData()* which takes as input argument a string. Here is the place where is decided if the inputed user string is a valid double number representation or not. Also it is checked that the double number to be not too big (subjective opinion). It check that input string length should be in range of 10 chars, including the sign. The *CheckData()* method takes care to run the *CheckStringAsDouble()* method which ckecks if the string can be represented as a double and the *CheckInputRange()* method which checks the inputed range described above. The only response of this object is true or false, that meaning that the inputed string is ok or not for future converting. Bellow is the code for this object:

class DataCheck

{

//checks the input data to be a valid secured range double

public bool CheckData(string input)

{

if (CheckStringAsDouble(input))

if (CheckInputRange(input))

{

return true;

}

return false;

}

//checks if the string can be converted to double

private bool CheckStringAsDouble(string input)

{

try

{

Convert.ToDouble(input);

}

catch (Exception ex)

{

return false;

}

return true;

}

//check it the input is in secure ranges

private Boolean CheckInputRange(string input)

{

if (input.Length < 10)

return true;

return false;

}

}

**TemperatureConverter –** this object takes care to convert the temperature. It has two public methods which perform the job very well. Bellow is a short representation of this class:

public class TemperatureConverter

{

//converts Celsius to Fahrenheit

public double ConvertCelsiusToFahrenheit(double C)

{

return (C \* 1.8) + 32;

}

//converts Fahrenheit to Celsius

public double ConvertFahrenheitToCelsius(double F)

{

return (F - 32) \* 0.5;

}

}

As you can see is very simple one.

**Conclusion**

Performing this laboratory work I observed that there is much more explanations than the code itself. It can be constructed a theory about just one program, just commenting the choices made by the programmer. The conclussion of above mentioned thoughts is that the developer, and not only the developer has a great responsability when making a decision. Depending on decisions the future of the application can become brighter or not. Applying the MVC pattern I realised that knowing patterns is a very good idea. Also I remember one saying which stands that *Two weeks of hardwork can save one day of planning*. So, before beginning writing code, we should carefuly think about our choices and the consequences that will occure. Another aspect is that the Patterns are a little bit vague, some of them are more, and this makes things a little bit confusing. But, as our world is strongly subjective, we can admit ourselves to understand things as we want, not as others are trying as to constraint, and we can make our own decissions what will strongly punish us in case of failure.

**Appendix A**

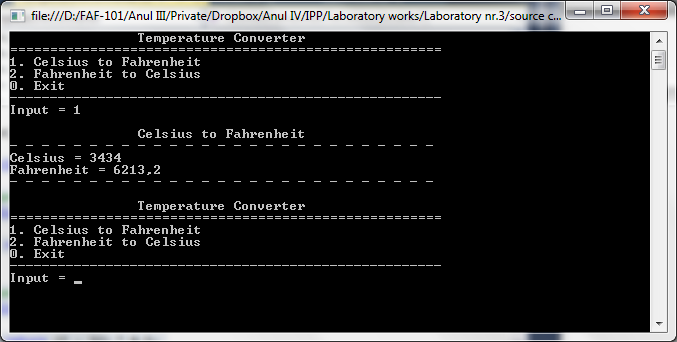


Fig. 1 *Main window of the application*