Student Planner A+

Software Architecture / SOEN 6471

Dr. Peter C Rigby

Winter 2013

Document Name : Individual Part for M4

Shahla Noori 5972671

Strategy and Factory[1]:

As we have already mentioned in M3, the application we are working on doesn't have a smart design or architecture and there is no known pattern used in the code.

Therefore it was decided to see which part of the code is potential to be coded by a specific design pattern.

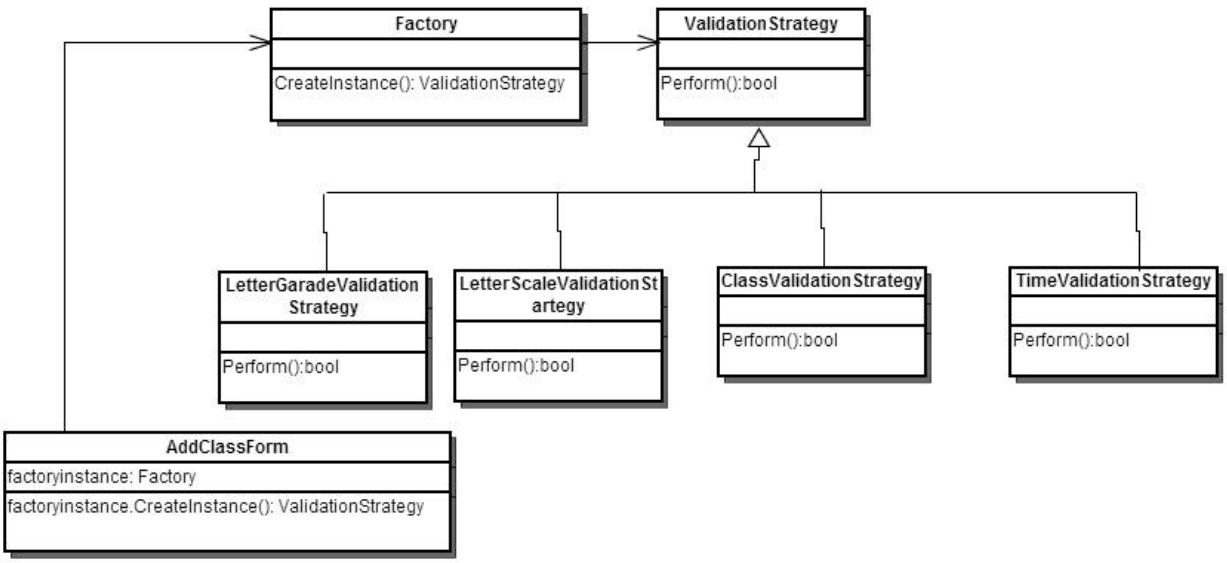
There is a method called “SaveClass” in the class AddClassForm which could have been handled by Strategy and Factory design patterns.

This method, by checking different conditions tries to do some specific operations. What we can do here is to create a separate class for the operations done in each condition block. So, we will technically have 4 different classes that each is responsible for its own operations.

We should create an abstract class Called ValidationStrategy which has a virtual method called Perform() that returns bool. All the other 4 classes that have been created in the previous step will inherit from this Strategy class and will override the method Perfom() in their classes following their own logic.

Next step is to create a factory class that has a method called CreateInstance that returns a ValidationStartegy object. Inside this method we should transfer those conditions in SaveClass and for each proper condition; its relevant class will be instantiated and will be returned.

The final modification is to get an instance of our factory class in AddClassForm and to call its perform() method. In this situation factory will decide which class to be instantiated.



[1] http://sourcemaking.com/ , http://www.newthinktank.com/videos/design-patterns-tutorial/

```
public partial class AddClassForm : Form
    {
        #region Constructor
        public AddClassForm()
        {
         ...
        }
        #endregion

        AddGradingScaleForm addGradingScale = new AddGradingScaleForm();
        AddGradeCategoriesForm addGradeCategories = new AddGradeCategoriesForm();
        ...

        #region Methods
        private bool saveClass()
        {
            if (checkingClassNames())
            {
                return raiseInvalidClassNameError();
            }
            if (chkClassFinished.Checked == false && categories.Count == 0)
            {
                return raiseInvalidGradeCategory();
            }
            if (dtClassStartTime.Value.TimeOfDay > dtClassEndTime.Value.TimeOfDay)
            {
                return rasiInavlidTimeErorr();
            }

            ...
            return true;
        }

        private static bool raiseInvalidLetterGradeError()
        {
            Util.displayError("Please select a valid letter grade for the class", "Invalid Letter Grade");
            return false;
        }

        private static bool raiseInvalidGradeCategory()
        {
            Util.displayRequiredFieldsError("Grade Categories");
            return false;
        }
        private static bool raiseInvalidClassNameError()
        {
            Util.displayRequiredFieldsError(new string[] { "Class Name", "Days" });
            return false;
        }
        private static bool rasiInavlidTimeErorr()
        {
            Util.displayError("Invalid Start and End Times", "Error");
            return false;
        }
    }
```

Jamileh Mohammadkhani 5990351

Observer Pattern

Observer pattern is a proper way to notify many listeners in case of one Subject changes. One of the most common usages of observer is in **Model–view–controller (MVC**) software architecture. Although In our project they didn't use MVC architecture, Observer is a proper pattern to update view as in following I will explain.

Reference: http://en.wikipedia.org/wiki/Observer_pattern

Why observer?

In AssignmentPlanner class, both following methods are called time to time to update the calendar and LeftPanel view:

```
updateCalendar(eventsLoadedStartDate, eventsLoadedEndDate);
updateLeftPanel();
```

Snippet code

```
private void deleteFinishedEventsToolStripMenuItem_Click(object sender, EventArgs e) {
 //display confirmation message
 DialogResult reallyDelete = MessageBox.Show("Are you sure you really want to delete all
finished class events? This will remove all graded assignments and associated grading
information for the past events associated with finished classes.", "Really Delete
Finished   Class Events?", MessageBoxButtons.YesNo, MessageBoxIcon.Warning);
            if (reallyDelete == DialogResult.Yes) {
                Database.modifyDatabase("DELETE FROM Event WHERE EndDateTime <
DATETIME('now', 'localtime') AND EventID IN(SELECT EventID FROM GradedAssignment NATURAL
JOIN Class WHERE FinalLetterGrade IS NOT NULL);");
                calendarEvents.Clear();

                //reset calendar view to default settings
                …
                updateCalendar(eventsLoadedStartDate, eventsLoadedEndDate);
                updateLeftPanel();
            }
        }
  private void deleteFinishedGeneralEventsToolStripMenuItem_Click(object sender,
EventArgs e) {
     //display confirmation message
     DialogResult reallyDelete = MessageBox.Show("Are you sure you really want to delete
all  finished general events? This will not delete any current class assignments or
events.", "Really Delete Finished General Events?", MessageBoxButtons.YesNo,
MessageBoxIcon.Warning);
            if (reallyDelete == DialogResult.Yes) {
                Database.modifyDatabase("DELETE FROM Event WHERE EndDateTime <
DATETIME('now', 'localtime') AND EventID NOT IN(SELECT EventID FROM GradedAssignment);");
                calendarEvents.Clear();
                …
                updateCalendar(eventsLoadedStartDate, eventsLoadedEndDate);
                updateLeftPanel();
```

```
            }
        }
        private void updateLeftPanel() {
            updateEvents();
            updateGrades();
            updateGPA();
        }
//another Example of repetitive using these updates
enableDisableToolStripMenuItem.Checked = PlannerSettings.Default.SyncEvents;

            updateEvents();
            updateGrades();
            updateGPA();
            updateCalendar(eventsLoadedStartDate, eventsLoadedEndDate);
```

I think better way for implementing of these updates is using an observer pattern to update both these calendar and leftPanel. In this way by each change on form's data such as add, delete and modification; observer will be responsible for these updates.

Mechanics

1. Add Listener class to AssignmentPlanner class

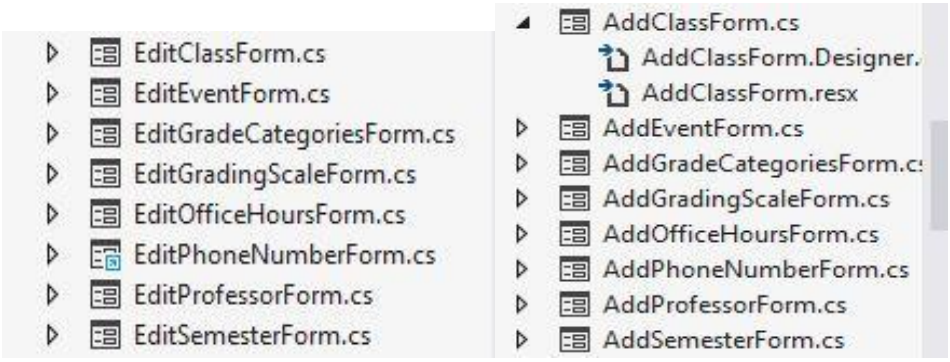
```
public interface Listener {

            public abstract void viewchangs();

      }

      static private ArrayList<Listener> listeners = new ArrayList<Listener>();

      static public void addListener(Listener listener) {

             listeners.add(listener);       }

      static public void notifyListeners() {

             for (int i = 0; i < listeners.size(); i++) {

                    Listener v = listeners.get(i);

                    v.viewchangs();           }    }
```

2. Make an update method :
   Public void Update (){
   ```
   updateCalendar(eventsLoadedStartDate, eventsLoadedEndDate);
   updateLeftPanel();
   ```
   }
3. Replace all
   ```
   updateCalendar(eventsLoadedStartDate, eventsLoadedEndDate);
   updateLeftPanel();
   ```
   with
   ```
   notifyListeners();
   ```

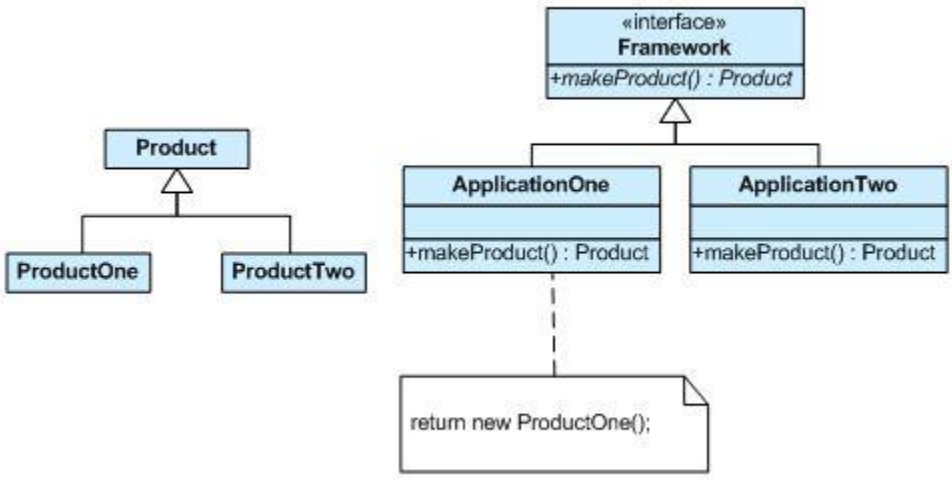
Bahareh Mohammadpourbarghi 5894336

Factory Design Pattern

The design pattern which I came up with for project A+ student planner is **Factory Method Design Pattern**. In this C# desktop application you can find lots of Forms for different views of GUI. As you open the code in Visual Studio you can easily notice so many classes like:

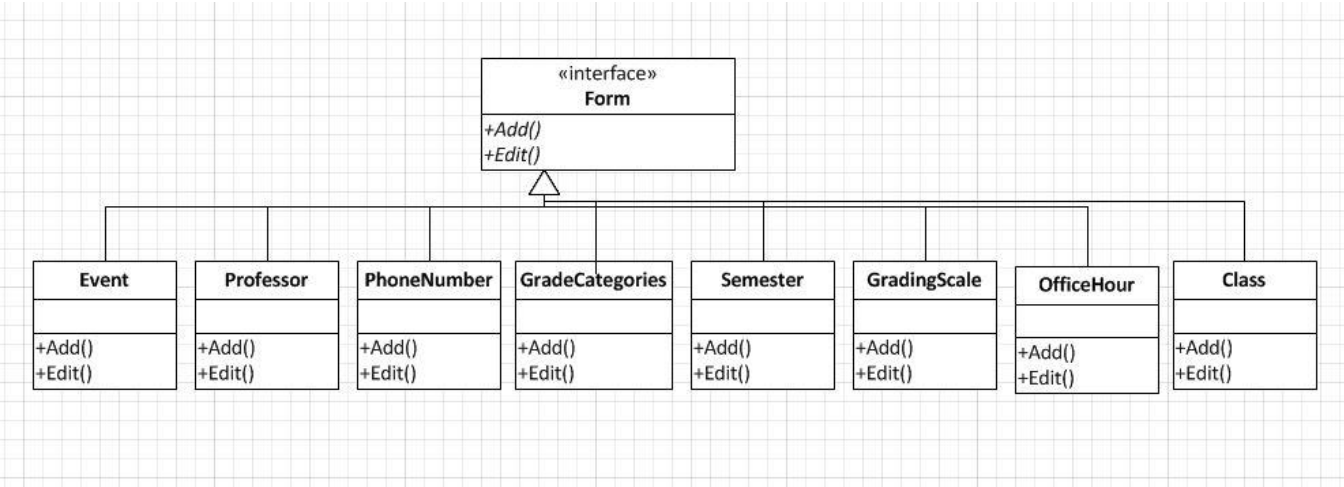


Factory Method makes a design more customizable and only a little more complicated



So my idea is that if we use factory Method Design pattern we can make code more modular. We can have one **<<interface>>** with name **Form** and sub classes would be Class, Event, GradeCategories, OfficeHour, PhoneNumber, Semester, GradingScale. The methods in Interface would be Add and Edit

with different functionalities and body for each subclass. The interface will decide which add or edit method for which Form would be called.



Harcharan Singh Pabla 6547702

Introduction

The A+ Student Planner is a combination of 2 systems – a Calendar and a Planner. The system, code wise is primarily consisting of 1500 lines of code for the Calendar and 9700 lines of code for the Planner. So when the projects were integrated into each other, no design patterns were used. The main focus was on integrating the functionality of both the solutions, so that the Planner could use the features of the Calendar. One design pattern that could be obviously used over here and was not, is Façade.

Façade

Definition: Façade provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. It is in-fact used to wrap a complicated system with a simpler interface, so that the other outside systems that is/are trying to access that system, can simply access the Façade.

An example to explain the use of Façade: There are 2 independent systems – System A and System B. And System A wants to access some methods or functionality of System B, so that the user has a new system which is a combination of the functionalities of both the systems.

Scenario: System A code is making a call to MethodOne(), MethodTwo(), MethodThree() and so on.., in System B project code.

Problem: High complexity between System A and B.

Why: System A makes direct calls **into** System B to access the classes and methods.

Co-relating the above example to our application – System A is the Calendar and System B is the Planner. The code in the Calendar is making calls to the Planner to perform specific functions like checking some condition before displaying the events on the Calendar. These calls are being made directly from one code base project to another code base project, which is detrimental to the code overall. The two projects are working by sharing namespaces and direct object calls.

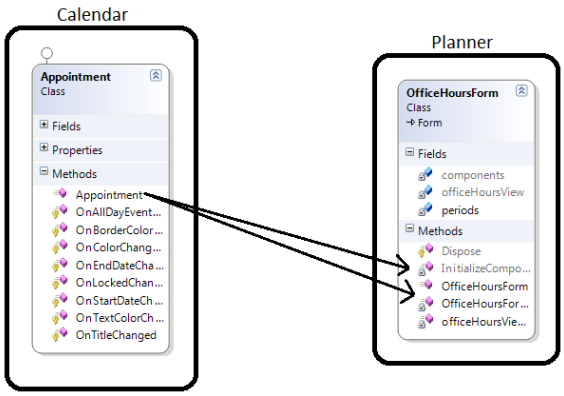


Figure 1. The current situation – Direct Method calling from Calendar to Planner code.

The above diagram shows the present working mechanism for making calls from Calendar to the Planner. The diagram shows that there is direct calling between the two code bases.

Design Pattern

I plan to introduce a Façade on the Planner side, which will act as an intermediate interface with whom the Calendar will interact to get its function performed. In our case, the Calendar is calling to the Planner to check for the timings of availability of the Professor, before displaying it finally on the Calendar. The modified architecture after introducing a Façade would look like this:

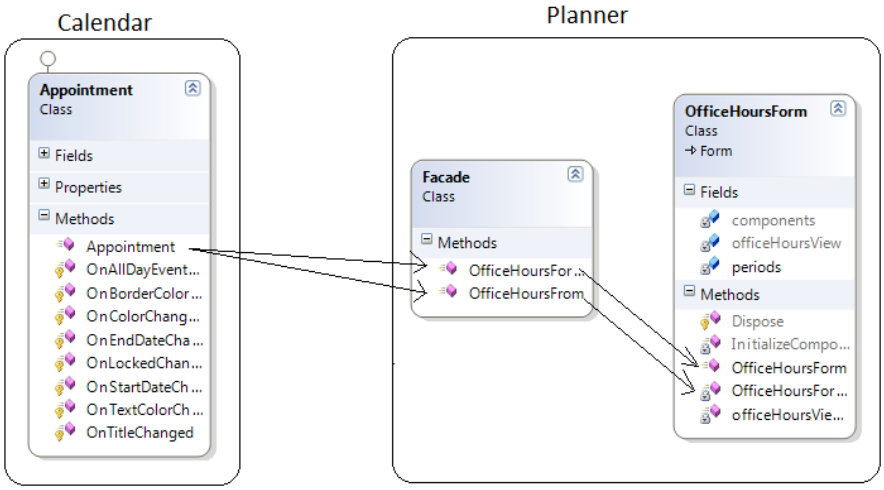


Figure 2. The Class Diagram after adding the Façade Design Pattern.

The Façade would not only help in this case, when we are dealing with two projects being used together. In future, when the project will be expanded to be used with some other third party project, the Façade will be really very useful and the architecture will be really steady.

Conclusion: So introducing a Façade will be beneficial architecturally – not just for the current scenario where we are using two projects but also for future project expansions.

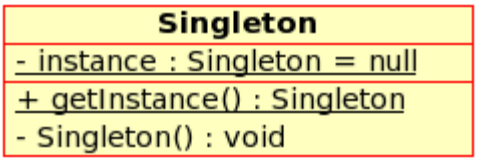
Robert Saliba 6412033

Singleton

Singleton is one of the first design patterns described in *"Design Patterns: Elements of Reusable Object-Oriented Software by E. Gamma, R. Helm, R. Johnson, J. Vlissides".* It restricts object creation for a class to only one instance.

We could use Singleton in the class Office12Renderer. This class is the renderer class for the control DayView which is the main control in the application.

public class Office12Renderer : AbstractRenderer



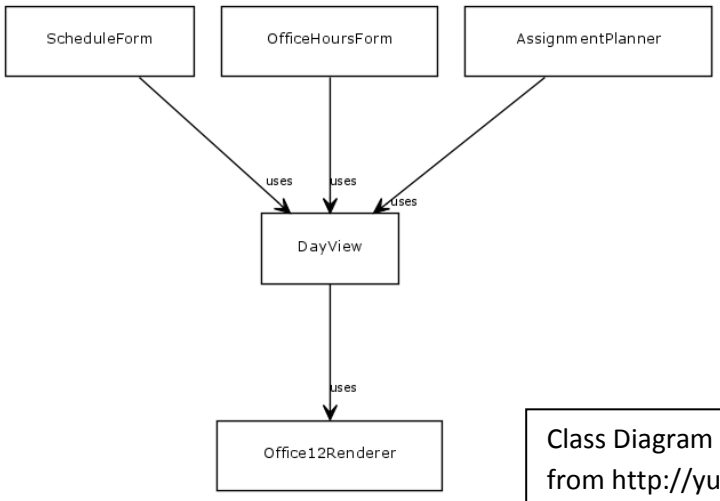
This doesn't have any coded constructor. The following piece of code should be added to change the class to a singleton

```
private Office12Renderer _instance;
private Office12Renderer()
{
}
public Office12Renderer GetOffice12Renderer()
{
if (_instance == null)
{
_instance = new Office12Renderer();
}
return _instance;
}
```

Then all the code that creates new objects from Office12Renderer should be changed to call the public method GetOffice12Renderer instead of calling the constructor.

Singleton pattern in this case is very useful in this application since Office12Renderer is used only by the control DayView

```
public class DayView : Control
{
public
AbstractRenderer
}
```



Renderer //Field

By this way we could guarantee that the view of the control is unified though all the application.

Class Diagram using the online tool from http://yuml.me