A+ Student Planner

Adv. Software Architecture - SOEN 6471

Dr. Peter C Rigby

Winter 2013

Document: Milestone 3

Jamileh Mohammadkhani Ghiyasvand	**Student ID : 5990351**
Bahareh Mohammadpourbarghi	**Student ID : 5894336**
Shahla Noori	**Student ID : 5972671**
Robert Saliba	**Student ID : 6412033**
Harcharan Singh Pabla	**Student ID : 6547702**

Comparison: Our Ideal Architecture v/s Actual Architecture

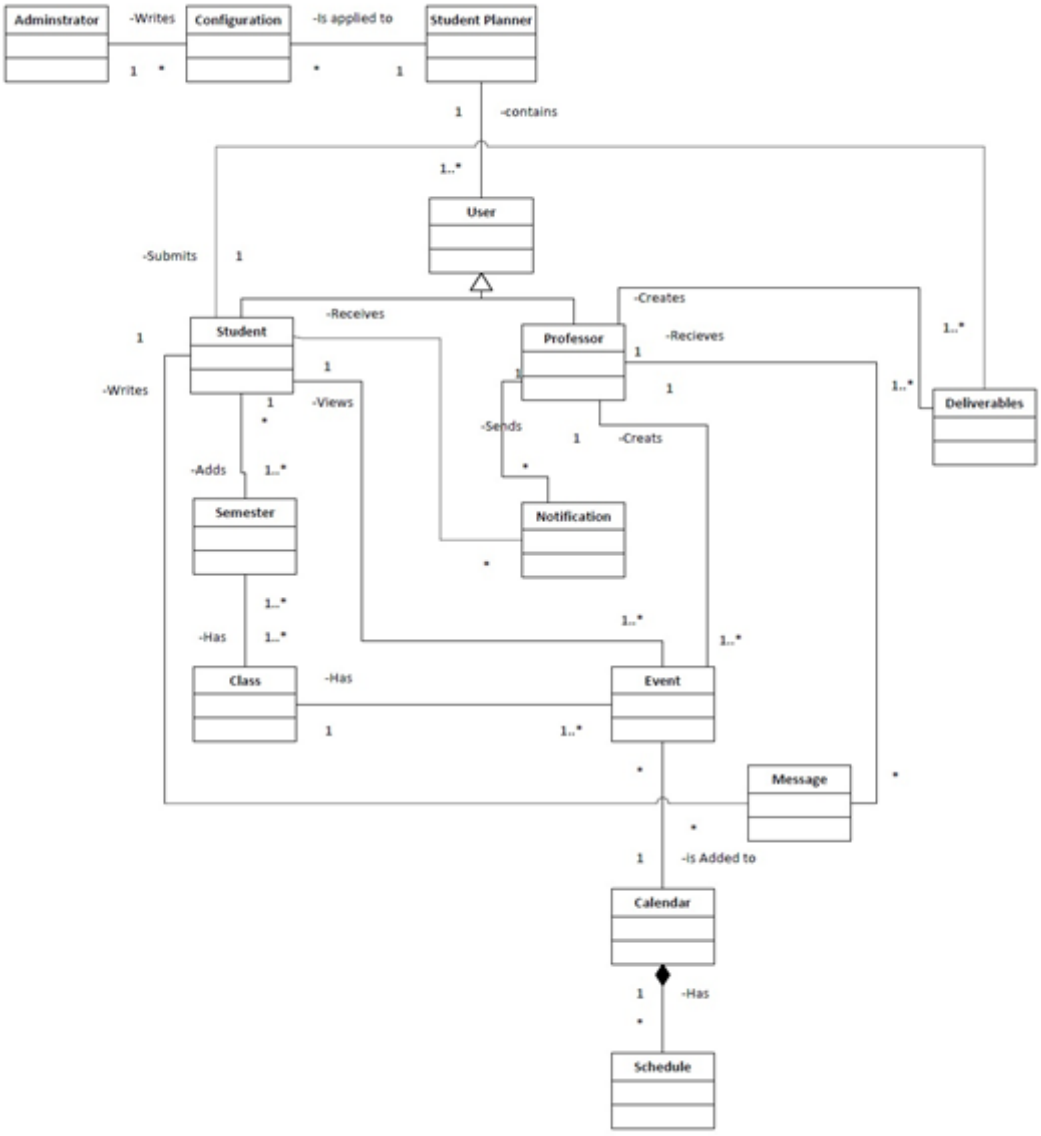

Figure 1. Class Diagram of Ideal System

In the domain model which we created for Student Planner, entities are somehow the same as the actual system. In this domain model we have 2 different users who interact differently with the system. Student and Professor are 2 users of our system. Our system has a calendar which contains all schedule information. Whenever professor creates an event it will be stored in calendar in order to be visible to student whenever he/she wants to view it. Professor and students can communicate with each other via messages through the system.

All information about the student semester and classes will be stored in the system. Student can check his/her class schedules in the calendar. Any kind of notifications regarding class cancellation or change of exam date can be created by professor and viewed by students. Any deliverables (projects and assignments) can be assigned for classes by professor and students can view and submit deliverables as well.

A+ Student Planner Actual Architecture

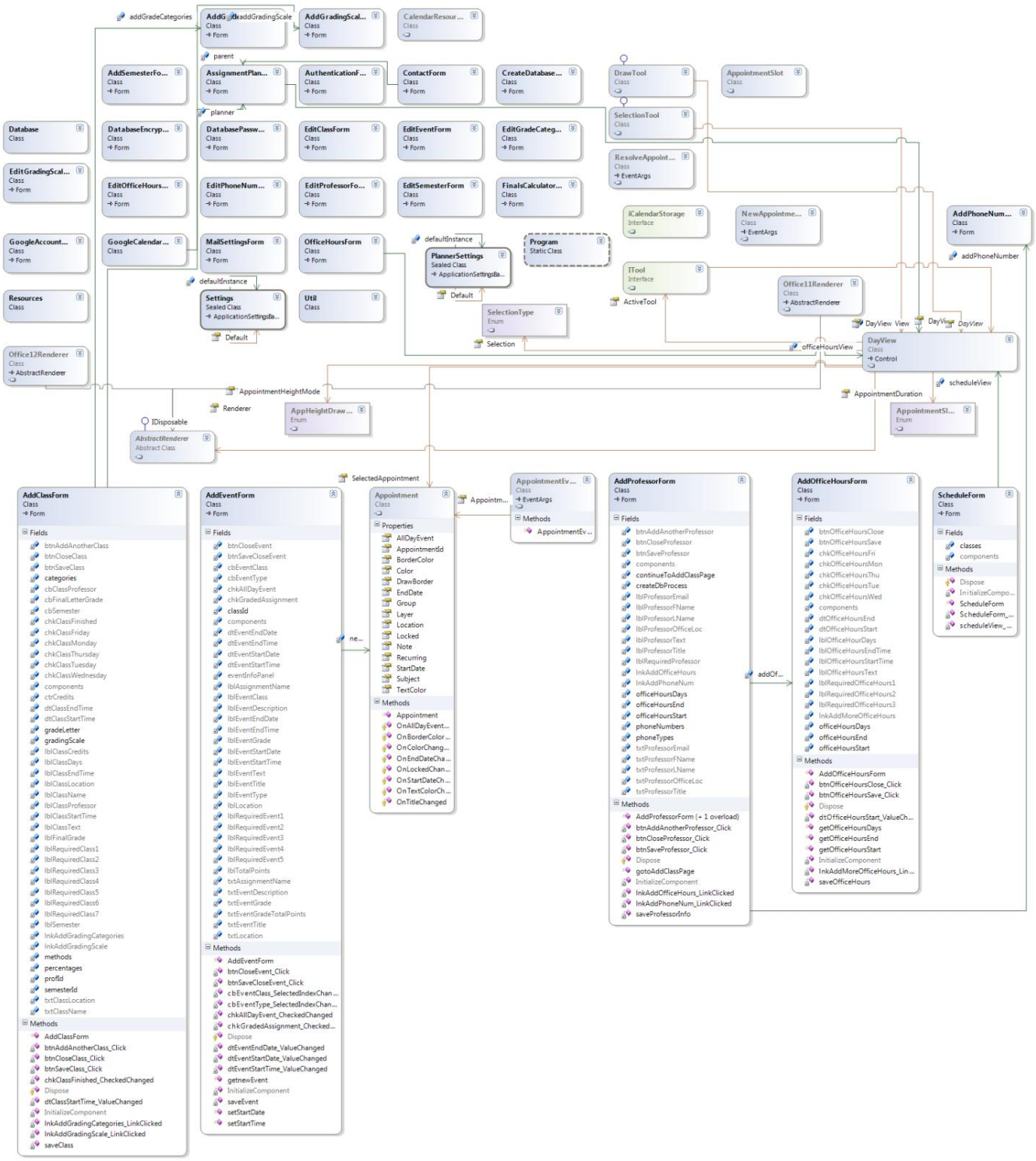

Figure 2. Extracted Class Diagram for the Actual Application

The focused part of application architecture is shown below. It has a lot of architectural flaws and code smells.

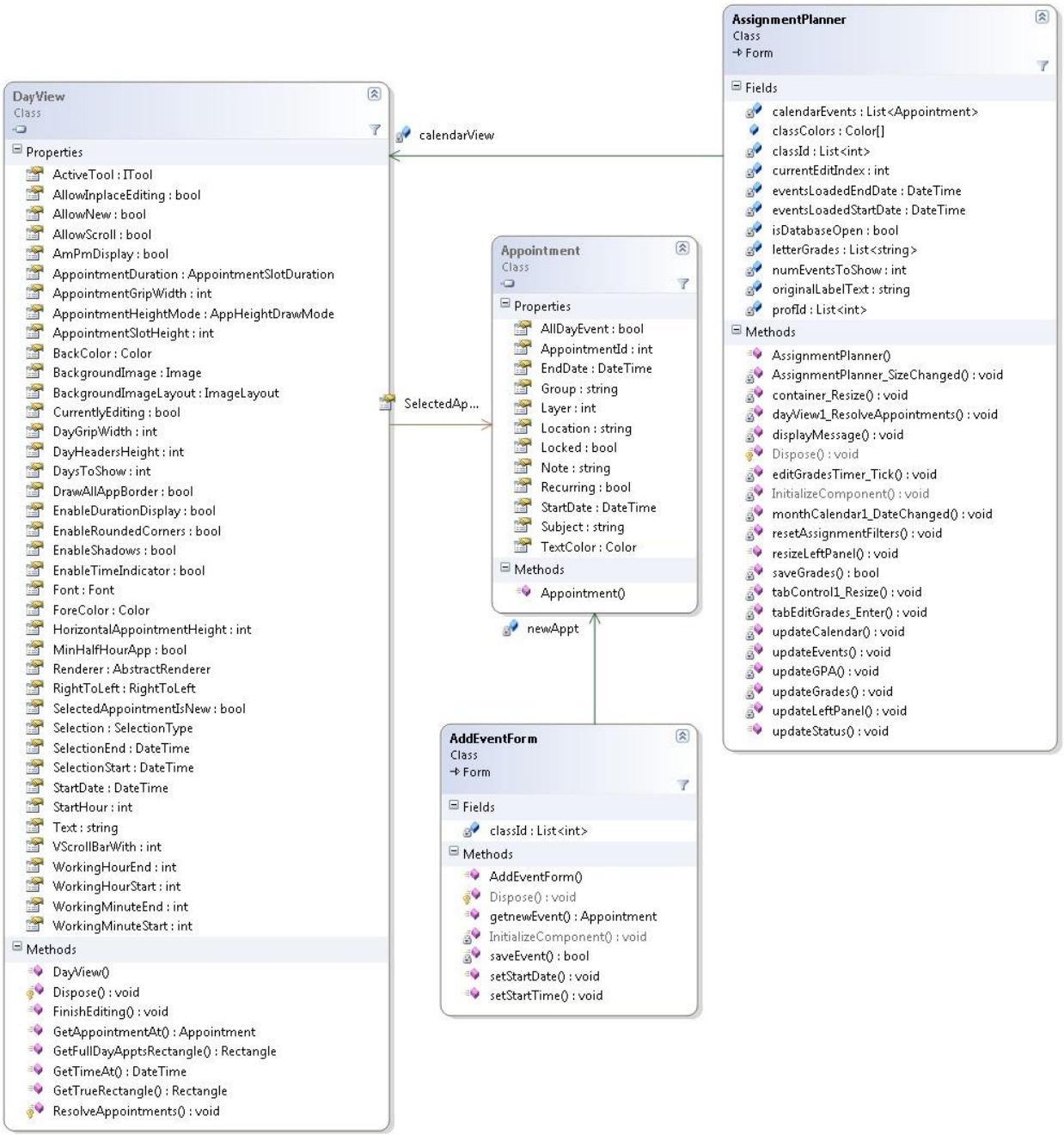

Figure 3. The most interesting part of the Application Architecture

Our system is a web based application which is totally different from the actual system which is a desktop application. Actual system has only one user (student) while our system can have multiple users at the same time since it is a web-base app. In our system students and professor can communicate via email. Professor can send notifications or messages to multiple students simultaneously. Data in our system is updated via databases automatically but actual system uses local databases and since most users do not want to have to

run a local server on their computer in order to run the application, A+ utilizes an embedded version of the database.

Unlike our imagination, the way that A+ Student Planner is designed is less clever than we thought. Since it is not using any specific pattern or even not utilizing the real power of object oriented concepts like polymorphism. Some significant aspects of a software design like low coupling, high cohesion and information expert is not well applied in the design and it makes it not enough maintainable, less flexible and hard and not clear to understand.

Roughly, all the operations are handled in the UI and it makes the system really low cohesive. For example all the events are handled in the code behind of the forms. While in our design we were going to create a new abstract class that could take care of the event handlers in a way that the UI would fire the events via the specific implementations of that class. In this condition we could keep the UI high cohesive since its responsibility is not controlling the events. And each Handler takes care of very specific actions.

The extracted actual entities are pretty well mapped with the ideal system's entities.

For instance:

Actual Classes	Conceptual Classes
AddEventForm, EditEventForm	Event
Appointment	Schedule
AddClassForm, EditClassForm	Class
AddProfessorForm,EditProfessorForm	Professor
AddSemesterForm, EditSemesterForm	Semester
Assignments	Deliverables

Table 1. Actual Entities v/s Ideal Entities

But there are still some limitations in the actual system; that are managed in the ideal design. As it's mentioned before the actual system intends to limit access to the database to a single user. This is because it intends to develop a typical desktop application that users will be able to run from their computers without being connected to the Internet. Because they wanted to eliminate the dependency on an Internet connection, this restricts the focus to local databases.

Microsoft Visual Studio gives us the ability to extract Class Diagram of the created project, we can add C# classes or namespaces from Architecture Explorer or dependency graphs to a UML class diagram. Also adding C# classes from Solution Explorer is possible. While doing right click on each project that is added to our solution we have an option called "View Class Diagram". Generating class diagram via Visual Studio doesn't support cross project relationship. Each class diagram can show only the classes within the same project. By doing a couple of research in the web we figured out that "ModelingPowerToys" which is a plug-in for visual studio can omit this restriction and lets us create a Class Diagram that shows cross project relationship.

Example - Two classes and relationship between them

```
public class Appointment : iAppointment
  {
                        #region " Private Fields "
                        private int layer;
                        private string group;
                        private DateTime startDate;
                        private DateTime endDate;
                        ....
                        #endregion

     public Appointment()
     {
               color = Color.White;
               borderColor = Color.Blue;
               Subject = "New Appointment";
     }
     #region " Public Properties "
     public int Layer
     {
        get { return layer; }
        set { layer = value; }
     }
     .....

     public bool AllDayEvent
     {
        get
        {
           return allDayEvent;
        }
        set
        {
           allDayEvent = value;
           OnAllDayEventChanged();
        }
     }

#region iAppointment Members
     public DateTime StartDate
     {
        get
         {
               return startDate;
         }
       set
         {
               startDate = value;
               OnStartDateChanged();
          }
     }
    public DateTime EndDate
       {
          get
             {
               return endDate;
             }
           set
             {
               endDate = value;
               OnEndDateChanged();
             }
       }
               .....
    #endregion
}
```

```
public partial class AddEventForm : Form {
      //store the list of class ids
      private List<int> classId = new List<int>();
      //create a member variable for the appointment in the calendar view
      Appointment newAppt = null;
       public AddEventForm()
       {
         InitializeComponent();
         //dynamically add classes to combo box while storing class id's associated with class
name
         Util.addClasses(cbEventClass, classId, true, false, null, null);
         //set start and end date pickers to current date
         dtEventStartDate.Value = DateTime.Now;
         dtEventEndDate.Value = DateTime.Now;
      }
.....
      //get the information from the saved event once the form is closed
      public Appointment getnewEvent() {
         return newAppt;
      }
.....
      private bool saveEvent()
      {
         //ensure user entered a title since it is a required field
         if (txtEventTitle.Text.Equals("") == true) {
            Util.displayRequiredFieldsError("Event Title");
            return false;
         }
         //ensure start time is not later than end time (note this does not apply if an all day event)
         if (chkAllDayEvent.Checked == false && dtEventStartTime.Value.TimeOfDay >
dtEventEndTime.Value.TimeOfDay) {
            Util.displayError("Invalid Start and End Times", "Error");
            return false;
         }
....
         //begin transaction
         Database.beginTransaction();
.....
         //check if the event is a graded assignment
         if (chkGradedAssignment.Checked == true) {
            //get id of recently inserted event
            object eventID = Database.getInsertedID();
            double grade = Double.MaxValue;
            double gradeTotal = Double.MaxValue;
            //ensure an assignment name was given
            if (txtAssignmentName.Text.Equals("")) {
               Util.displayRequiredFieldsError("Assignment Name");
               return false;
            }
....
         //add event to calendar view
         newAppt = new Appointment();
         newAppt.StartDate = new DateTime(dtEventStartDate.Value.Year,
dtEventStartDate.Value.Month, dtEventStartDate.Value.Day, dtEventStartTime.Value.Hour,
dtEventStartTime.Value.Minute, 0);
         newAppt.EndDate = new DateTime(dtEventEndDate.Value.Year,
dtEventEndDate.Value.Month, dtEventEndDate.Value.Day, dtEventEndTime.Value.Hour,
dtEventEndTime.Value.Minute, 0);
         newAppt.Subject = txtEventTitle.Text;
         newAppt.Note = txtEventDescription.Text;
         newAppt.Location = txtLocation.Text;
         newAppt.AppointmentId = int.Parse(Database.getInsertedID().ToString()); //store unique
event id in calendar appointment note
         newAppt.Color = Color.Honeydew;
         newAppt.BorderColor = Color.DarkBlue;
         if (chkAllDayEvent.Checked == true) {
            newAppt.AllDayEvent = true;
            newAppt.EndDate = newAppt.EndDate.AddDays(1);
            newAppt.Color = Color.Coral;
         }
         else if (chkGradedAssignment.Checked == true) {
.....
         return true;
      }
....
}
```

Code Smells and Possible Refactoring

By looking at the weak design of the Student Planner application, we already knew that there would be a lot of code smells in the code by which applying a proper series of refactorings we could help in improvement of a better design.

Regardless of having Dead Code which is scattered in different parts of the code. Here is listed the obvious code smells we noticed:

1. Dispensable code: There is an interface named "iCalendarStorage" and a class named “AppointmentSlot” that are lazy classes since they are not used or implemented in any part of the code. These classes can be removed.

2. Extract Method: In AddEventForm, there is a method “SaveEvent”. It is a very large method, around 78 lines of code. We can use Extract Method and move method refactoring to make the method clear, so it’s easier to read and even modify. It contains nested switch statements and nested if statements inside the switch statements which make the method complex.

3. Low Cohesion: In AddEventForm, the class is performing database Read/Write operations that leads to LOW COHESION. We should have a separate class to tackle database related work.

4. Long Parameter List: In AddClassForm, AddEventForm, EditEventForm and EditClassForm there are couple of methods that have very complex if conditional statements which could be extracted in smaller methods to help reading the methods more easily. So we can use the “Consolidate Conditional Expression” refactoring technique to refactor those if statements.

5. Dead Code/Commented Code - As mentioned above, there is Dead Code at a lot of places in the application. To mention, in file Office12Renderer.cs, there is a block of commented code starting at Line Number 308. If a code block is commented, that means it is no longer used in the execution. So we can simple remove them.

6. AddClassForm.cs has a numeric type code that does not affect its behavior. Here we can replace it with a new class. First we create a new class for the LetterGrade. Then we change the implementation of the source class to use our new class. For all the methods on the source class that uses the code, a new method that uses the new class should be created. In every parts of the code where the old code was used we should modify it in order to use the new class. When making sure that it compiles successfully we should remove the old part that uses the codes. Compile and test is highly recommended to make sure all the changes are done successfully. The figure below is the UML diagram of the expected refactoring.

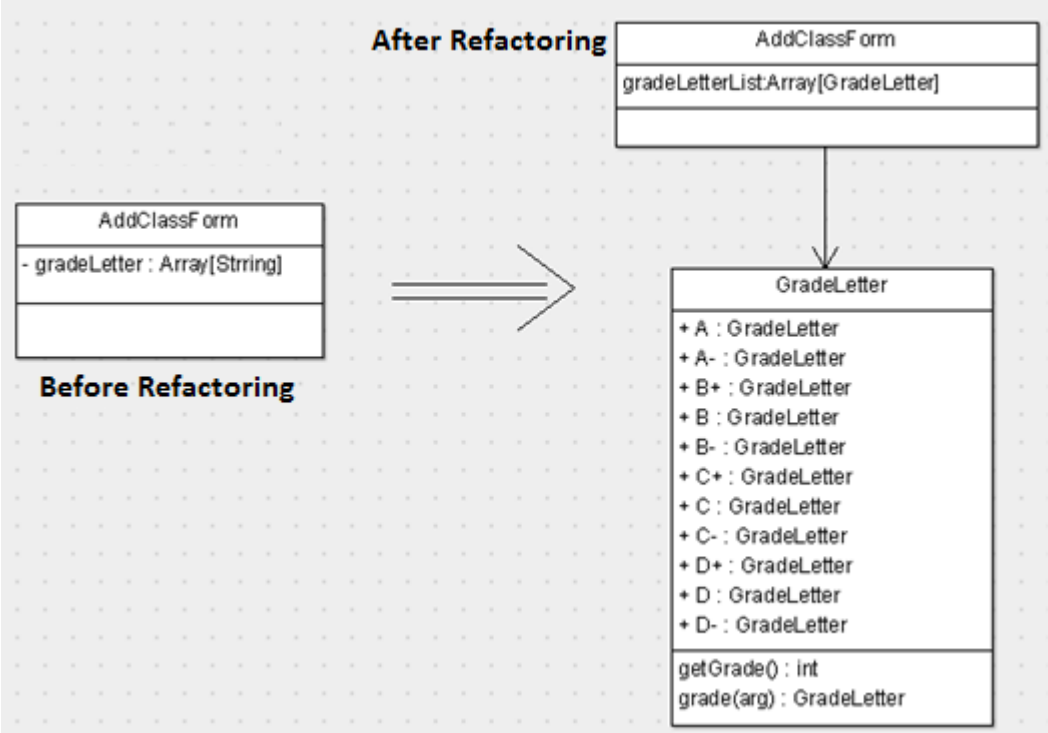

Figure 4. UML Diagram for Code Smell No. 6

Inter-relation between the Code Smells

The AddEventForm has a lot of code smells, which if all rectified one-by-one would lead to a much better code architecture. There is a very long method called SaveEvent. First of all we can extract that method to a new class to make it clearer and increase the modularity. The Save Event is performing some unrelated functions which decreases the cohesion. It is performing database read/write operations which should actually be performed by another entity which takes care of such operations exclusively. So we can extract this functionality from the SaveEvent and create a new class for it. Apart from this, the AddEventForm has a bad smell of long parameter lists, as mentioned above. This can be refactored using the Consolidate Conditional Expression technique and thus make the "if" statements look simpler and more clear. Similarly for EditEventForm, the class is performing unrelated tasks like displaying Error Messages, modifying the Database, etc. Here is shown the EditEventForm class, as an example of the inter-related code smell refactoring's to make the code better.

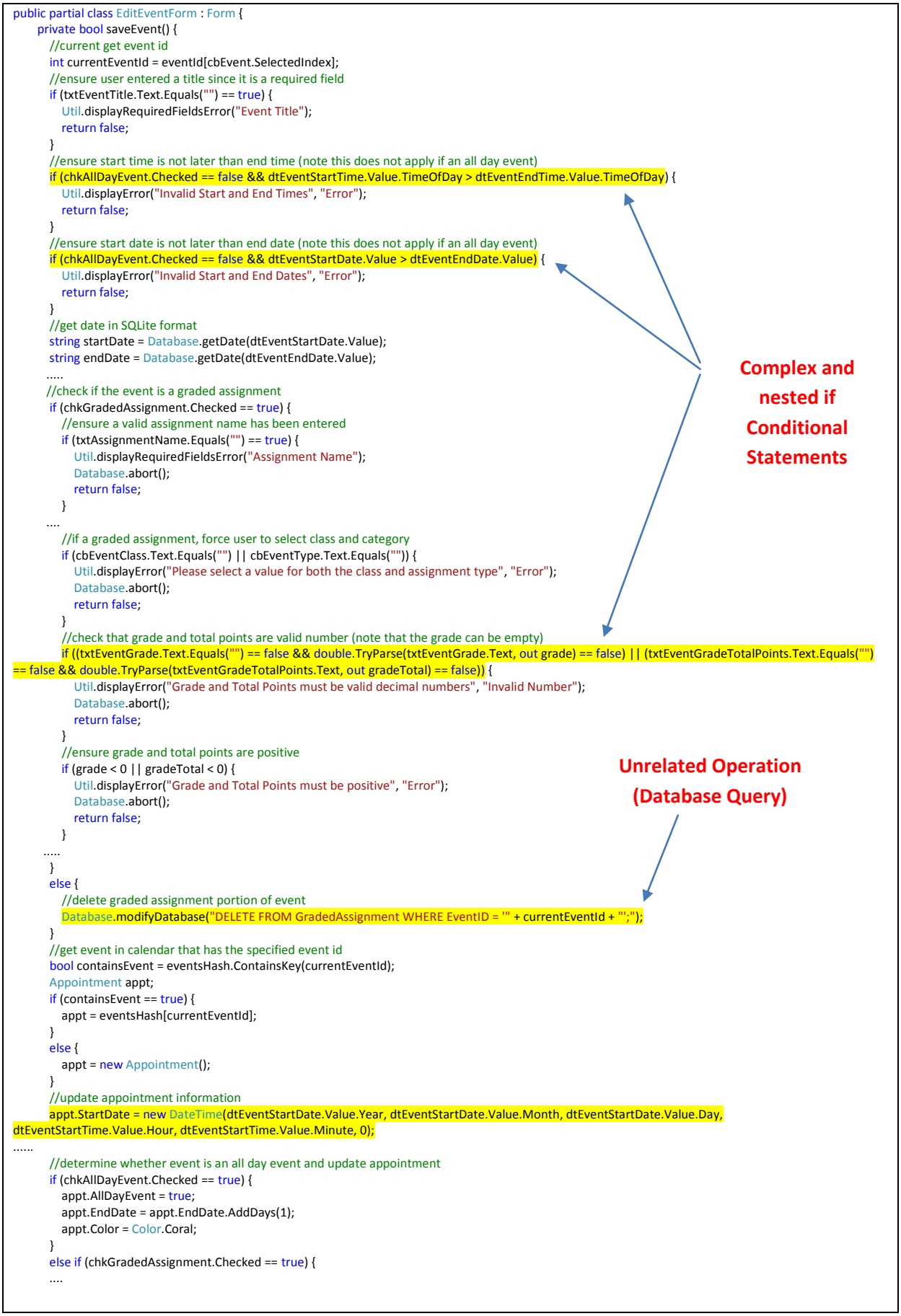

Figure 5. Multiple Refactoring's can be performed to address the Multiple Code Smells