Homework assignment 3.
Separation of responsibilities. Interfaces

1 Introduction

The primary goal of this assignment is to learn how to split responsibilities among classes in an object-oriented program and how to use interfaces to reduce coupling between program components.

However this assignment also covers a number of practical topics, e.g. repository pattern, WPF and class libraries.

2 Interfaces

The key principle of an interface in engineering is to establish rules of communication between system components. Take a well-known USB interface. Specification of the interface provides strict rules of how each device should behave during interaction with other devices. As a result, a PC does not need to know specific details of how a particular IO device is implemented as long as the device follows the specification. So any USB-compliant flash drive, mouse, keyboard can be successfully connected (both physically - a plug fits a socket - and logically - device is recognized by the operating system). [1].

The idea of an interface in programming is very similar. When two classes interact with each other, it is considered a good practice to establish rules of this interaction, especially when classes tend to change over time. When rules are standardized we can easily substitute components without breaking the whole system.

An interface provides a formal mechanism to describe capabilities that a class should support.

[1] There are however cases when proprietary functionality is needed - in this case a device driver is installed

2.1 Interfaces in C#

Below is an example of an interface declaration in C#:

```
public interface IPrinter
{
    bool IsOpen { get; }

    void Write(string message);
    void Flush();
}
```

An interface can also contain events and indexers.

Notice that interfaces do not contain any implementation. An interface is nothing more than a list of rules/capabilities. All members of the interface are public by convention and the explicit public keyword cannot be used.

Having an interface, classes that follow its specification can be created. Notice reference to the interface after class name:

```
public class FilePrinter : IPrinter
{
// All members of IPrinter must be implemented here
}

public class ConsolePrinter : IPrinter
{
// All members of IPrinter must be implemented here
}
```

Any of the two printers can now be used, however the crucial principle is: **Depend on abstraction, pass an implementation**.

This means that if a class needs to interact with a printer it should not know until runtime, which particular printer that would be. This in turn implies that no direct reference to the class name should be made. A reference to an interface should be used instead.

```
public class DataProcessor
{
// Some internal logic

void Process(/*other parameters*/, IPrinter printer)
{
// ...
printer.Write(...)
// ...
printer.Flush();
}
}
```

In the example above the Process method requires an external printer to output its results to. Notice that the passed parameter is of the interface type.

When calling Process from outside, a concrete implementation of the IPrinter interface should be provided:

```
DataProcessor dp = new DataProcessor();
// In this case we are using a file printer
dp.Process(..., new FilePrinter());

// In this call, a console printer
dp.Process(..., new ConsolePrinter());
```

3 Description of tasks

In this assignment you will work on a program that calculates rating of a student based on the course marks.

Browse the supplied template. The main logic of the application is concentrated in the MainWindow.xaml.cs file. This is an example of a poor object-oriented design as the window class has multiple responsibilities. Apart from the interface logic it stores application data (two lists), algorithms for generating data and calculating rating based on grades.

You will redesign the application so that the window class is only responsible for the interface and so that it doesn't know about concrete classes that implement other features.

- (0.5 point) The functionality of the program is split between two projects. One of them (StudentRating.Classes) is a class library, which contains definitions of domain classes and interfaces. The other (StudentRating) is a normal WPF application. First you need to fix library and namespace references, so that the solution compiles. You will need to add a reference from the main project to the class library and add the required "using" declarations.

- (0.5 point) Override the "Equals" method for "Grade" and "Course" classes. Two courses are considered equal when they have the same name. Two grades are equal when they reference the same course.

- (2 points) Look through the IRepository declaration. This interface will be implemented by two classes storing data.

  Inside the "StudentRating.Classes" project create a new "Repositories" folder and then a new class named "TestRepository" that implements the "IRepository" interface. Implement all interface members inside the newly created class. Notice that the "Remove" method accepts a predicate to filter items that should be removed. Move the logic that generates test items from MainWindow to TestRepository.

  Note that inside TestRepository items should be generated only once (not at every call to lists' "getters")

The "Save" method can be left blank as the test repository doesn't save items to external locations.

Modify code in the MainWindow class so that the project compiles and works as the original template. When declaring a variable of the repository class, make sure you specify its type as the interface, not the concrete class.

The following requirements apply to IRepository methods:

- "AddGrade" and "EditGrade" should verify the new/edited item. In case a null value is passed, an ArgumentNullException should be generated. If the item was previously added to the grades list, throw an ArgumentException. Catch the exception at the location of method call. For comparison checks use the "Equals" method that was added at the previous step.
- The Id property for a new item should be set in the "AddGrade" method in such a way that no two items in the list have the same Id.
- "RemoveCourse" should remove all courses that satisfy the predicate.
- Each of the three operations (Add, Edit, Remove) should call "Save" at the end and generate the "GradesChanged" event. The event will be handled in the main window.

- (2 points) Implement add, edit and remove logic for course grades inside the graphical interface. Use the supplied helper window (GradeWindow.xaml) for entering grade information. The same window should be used when adding and editing an item. The main window should subscribe to the GradesChanged event of the repository and update the interface.

  Note that you only need to implement adding, editing and removing for the grade list, the course list should remain unchanged

- (2 points) Add a second class named "FileRepository" to the "Repositories" folder that implements the same IRepository interface. This class should populate both lists from a file (any format is acceptable, serialization can be used as well). "Save" should also be fully implemented for this class.

  Change MainWindow code so that it now works with FileRepository instead of TestRepository.

- (1 point) Move the logic for calculating rating to a separate class (put it in a new "RatingCalculators" folder inside the class library) that implements the IRatingCalculator interface. Modify code in the window class so that the application works the same way as before.

  Write a second class implementing IRatingCalculator that uses the following formula for calculating rating (used at HSE):

$$Rating = \sum_{i=1}^{n} Mark_i \cdot Cred_i \quad (1)$$

where n is the number of courses,
$Mark_i$ - mark for the i-th course
$Cred_i$ - number of credits of the i-th course

Modify MainWindow so that it uses the new rating calculation class.

- (1 point) Apply interface segregation principle. Both IRepository and IRatingCalculator have references to a concrete collection class List<T> inside their methods'/propeties' declarations. Change List<T> to the lowest interface in the interface hierarchy implemented by the List class that still provides all the functionality required.
- (1 point) Implement the factory pattern for IRepository and IRatingCalculator interfaces.

Remark

After doing all these multiple refactorings you might come to a conclusion that the code only got more complicated. Although this is certainly true for simple programs like the one you have worked on, large software systems can't be properly maintained unless their code is carefully structured and responsibilities are split among classes.

4 Submission

Submit your solution following these steps:

1. Delete two temporary folders - "bin" and "obj" from both of the projects.
2. Add the whole solution to a ZIP **(not RAR or 7Z)** archive. **USE ONLY LATIN LETTERS AND THE UNDERSCORE SYMBOL (_) IN THE ARCHIVE NAME**
3. Upload the archive to WiseNetLabEDX in the corresponding section

5 Grading policy

The final grade for the assignment is determined by the number and quality of completed tasks. An instructor has the option to ask one or two additional questions in case of an unclear grade.