Lab: Interfaces and Abstraction

This document defines the lab overview for the ["C# OOP Advanced" course @ Software University](). Please submit your solutions (source code) of all below described problems at the end of the course at [softuni.bg]().

Introduction

So far we've learned about **classes**, their **members** and the first three principles of OOP - **Encapsulation**, **Inheritance** and **Polymophism**. We've employed all of that and more in refactoring our **BashSoft**. Now it is time to go one step further and employ the fourth OOP principle in our code - namely **Abstraction**. We will do this by creating several **interfaces** that will correspond to the classes we have. We will also try to **segregate** them somewhat into more than one **interface** per class where we see fit.

Problem 1. Creating Interfaces for the IO package

As you've learned during the lecture the purpose of an **interface** is to define what methods a certain class needs to have in order to be sufficient for a certain task. Thus all methods in an interface are considered public and abstract.

Let's start with creating our first **interface** that will define what a command should have. Create a new package called "**contracts"** and in it an **interface** called "**IExecutable**". Then define the proper method signature for the execute method our commands have:

```
namespace Executor.Contracts
{
    public interface IExecutable
    {
        void Execute();
    }
}
```

Now implement the method in our abstract **Command** class:

```
public abstract class Command : IExecutable
```

If you defined the **interface** correctly there shouldn't be any issues. You could also implement the interface in the derived commands, although would be redundant, the code will be considered more readable. Finally replace the abstract **Command** in every **field**, **method** or **constructor return type / parameter**. This way we will raise the **abstraction** level of our application:

```
public void InterpredCommand(string input)
{
    string[] data = input.Split(' ');
    string commandName = data[0].ToLower();

    try
    {
        IExecutable command = this.ParseCommand(input, commandName, data);
        command.Execute();
    }
    catch (Exception ex)
    {
        OutputWriter.DisplayException(ex.Message);
    }
}

private IExecutable ParseCommand(string input, string command, string[] data)
```

Our next **Interface** will be for the **CommandInterpreter**. You can call it "**IInterpreter**" and define in it the public method of our class:

```
namespace Executor.Contracts
{
    public interface IInterpreter
    {
        void InterpretCommand(String command);
    }
}
```

Don't forget to **implement** the **interface** in the **CommandInterpreter** and raise the abstraction level in **fields**, **methods** or **constructor return types / parameters**.

In the **InputReader:**

```
private IInterpreter interpreter;

public InputReader(IInterpreter interpreter)
{
    this.interpreter = interpreter;
}
```

In the **main** method:

```
IInterpreter currentInterpreter
    = new CommandInterpreter(tester, repo, downloadManager, ioManager);
```

Next up is the **IOManager -** it is a good place to employ some **interface segregation.** First create an **interface** called **IDirectoryTraverser** - it will hold the **TraverseDirectory** method:

```
namespace Executor.Contracts
{
    public interface IDirectoryTraverser
    {
        void TraverseDirectory(int depth);
    }
}
```

Create an **interface** called **IDirectoryCreator** for the **CreateDirectoryInCurrentFolder** method:

```
public interface IDirectoryCreator
{
    void CreateDirectoryInCurrentFolder(string name);
}
```

Finally for this class create an **interface** called **IDirectoryChanger** for the remaining two methods:

```
public interface IDirectoryChanger
{
    void ChangeCurrentDirectoryRelative(string relativePath);
    void ChangeCurrentDirectoryAbsoulute(string absolutePath);
}
```

Creating these three **interfaces** would allow us to make classes that does **only one** of the things we described by each **interface**. However in the case of our **IOManager** we will need all three, so we will combine the three interfaces in a single one:

```
public interface IDirectoryManager : IDirectoryChanger, IDirectoryCreator, IDirectoryTraverser
{
}
```

Implement it:

```
public class IOManager : IDirectoryManager
```

Finally replace **IOManager** with **DirectoryManager** in all **fields**, **methods** or **constructor return types / parameters** in the **CommandInterpreter** and all the **Command** classes:

```
private IDirectoryManager inputOutputManager;

public CommandInterpreter(Tester judge, StudentsRepository repository,
    DownloadManager downloadManager, IDirectoryManager inputOutputManager)
{
    this.judge = judge;
    this.repository = repository;
    this.downloadManager = downloadManager;
    this.inputOutputManager = inputOutputManager;
}
```

```
private IDirectoryManager inputOutputManager;

public Command(string input, string[] data, Tester tester, StudentsRepository repository,
    DownloadManager downloadManager, IDirectoryManager ioManager)
{
```

```
protected IDirectoryManager InputOutputManager
{
    get { return this.inputOutputManager; }
}
```

```
public OpenFileCommand(string input, string[] data,
                       Tester tester, StudentsRepository repository,
                       DownloadManager downloadManager, IDirectoryManager ioManager)
    : base(input, data, tester, repository, downloadManager, ioManager)
```

Finish the rest of the command classes yourself.

Make an interface for the **InputReader** yourself. You can call it **IReader** with method **`StartReadingCommands`.** Don't forget to implement it and replace **InputReader** with its interface where possible (psst - the main method).

We will not make an interface for our **OutputWriter** because it is not meant to be an instance class - it only has static methods and those don't belong in interfaces.

Problem 2. Creating Interfaces for the Models package

Let's begin by changing the name of both the classes to **SoftUniStudent** and **SoftUniCourse**:

We are doing this because we need the old more **abstract** names for the **underlying interfaces** we are about to create. Now let's do just that - create interfaces called **Course** and **Student.** They will have all the public methods of our **SoftUni** classes. This is how the **Course** interface should look (notice how we use the **Student** interface instead of the **SoftUniStudent** class in the method signatures):

```
public interface Course
{
    String getName();

    Dictionary<String, Student> GetStudentByName();

    void EnrollStudent(Student student);
}
```

And the **SoftUniCourse** class should now look like this:

```
public class SoftUniCourse : Course
{
    public const int NumberOfTasksOnExam = 5;
    public const int MaxScoreOnExamTask = 100;

    private string name;

    private Dictionary<string, Student> studentsByName;

    public SoftUniCourse(string name)...

    public string Name...

    public IReadOnlyDictionary<string, Student> StudentsByName...

    public void EnrollStudent(Student softUniStudent)...
}
```

Do the **Student** interface and **SoftUniStudent** classes yourself, following the pattern:

1) Put all **public** method signatures of the **class** in the **interface**.
2) Implement the interface in the class
3) Change every **SoftUniStudent** with **Student** in both of them.
4) Change every **SoftUniCourse** with **Course** in both of them.
5) Fix the **StudentRepository** to work through both the **Student** and **Course interfaces -** you can start off with the fields:

```
private Dictionary<string, Course> courses;
private Dictionary<string, Student> students;
```

Problem 3. Creating the remaining Interfaces

Create the following interfaces yourself:

- For the **Tester** class

    - **IContentComparer** with method **compareContent**
- For the **DownloadManager**
    - **IDownloader** with method **Download**
    - **IAsynchDownloader** with method **DownloadAsync**
    - **IDownloadManager implementing the upper 2 interfaces**
- For the **StudentsRepository**
    - **IDatabase** with methods: **LoadData, UnloadData** and extending sub-interfaces:
        - **IRequester** with methods: **GetStudentMarkInCourse, GetStudentsByCourse**
        - **IFilteredTaker** with methods: **FilterAndTake**
        - **IOrderedTaker** with methods: **OrderAndTake**
- For the **RepositorySorter**
    - **IDataSorter** with methods: **PrintSortedStudents(rename the method in the repository sorter)**
- For the **RepositoryFilter**
    - **IDataFilter** with methods: **PrintFilteredStudents(rename ….)**

Once you are done your main method should look something like this:

```
static void Main()
{
    IContentComparer tester = new Tester();
    IDownloadManager downloadManager = new DownloadManager();
    IDirectoryManager ioManager = new IOManager();
    IDatabase repo = new StudentsRepository(
                        new RepositorySorter(), new RepositioryFilter());

    IInterpreter currentInterpreter
        = new CommandInterpreter(tester, repo, downloadManager, ioManager);

    IReader reader = new InputReader(currentInterpreter);

    reader.StartReadingCommands();
}
```

If the application still compiles at the end - congratulations you have completed the lab for **Interfaces and Abstraction**!