Lab: Reflection & Attributes

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

In the current lab piece we are going to make the command pattern to be implemented using the reflection API from C#. That way we are going to create instances of the commands without the usage of a switch. This makes our application more flexible, because using it, will allow us to make a new child of the Command and implement it without changing any existing classes.

Problem 1. Making the Alias Attribute

Since we are not going to use the switch in our command interpreter, we will need to figure out another way to choose which command to create. This can be done by creating custom attributes and then scanning those attributes and depending on what value we've put in the constructor of the attribute, we will choose which our desired class to instantiate is.

We suggest you first create a new folder called Attributes.

The first attribute we will create is called Alias. It will have one property (name) which we will set by receiving the value for the string in the constructor.

```
public class AliasAttribute : Attribute
{
    private string name;

    public AliasAttribute(string aliasName)
    {
        this.name = aliasName;
    }
```

Now we will need a getter for the name:

```
public string Name
{
    get
    {
        return name;
    }
}
```

And finally for our ease we can override the Equals method, by comparing the name of the current instance with the given object.

```
public override bool Equals(object obj)
{
    return this.name.Equals(obj);
}
```

Finally, since we will be using this attribute for classes you might want to tell the compiler just that.

```
[AttributeUsage(AttributeTargets.Class)]
public class AliasAttribute : Attribute
```

Problem 2. Making the Inject Attribute

This attribute will point us which field we will set using the reflection. Now we are passing all the "managers/comparers/repos" in the constructor of the chosen command, but some of the users wishes have changed and since we are "Agile", we will follow the holy user's wishes and modify the desired functionality. Also note that the pass all utility classes to all commands works for now but if we add more utility classes like say a PeshoManager we will need to add it to all the constructors. Such code is not easy to maintain and extend so we need to fix it. It also causes some commands to know about stuff they don't need nor use.

But first let's create the Inject attribute class.

```
namespace Executor.Attributes
{
    [AttributeUsage(AttributeTargets.Field)]
    class InjectAttribute : Attribute
    {
        public InjectAttribute()
        {
        }
    }
}
```

As you can see, we do not have any fields to set in the current attribute. That's because we want to use it only as a notification that the field below it needs to be injected with a value from the command interpreter. Probably you notice that we've put a usage for the field only, and that is absolutely on purpose.

Problem 3. Changing the commands to fit the new approach for instantiation

Here are the three steps you need to do for all the commands in the application.

1. Delete everything from the constructor except input and data

2. Apply alias over the class by passing to the constructor of the attribute the string that is the actual command that you read from the console.

3. Leave only the fields that are used in the current scope of the class and put an inject attribute above them.

Here is how this should look:

1.

```
public ChangeRelativePathCommand(string input, string[] data)
    : base(input, data)
{
}
```

2.

```
[Alias("cdrel")]
class ChangeRelativePathCommand : Command
```

3.

```
[Inject]
private IDirectoryManager inputOutputManager;
```

And finally we are using this field in order to execute the current command:

```
if (this.Data.Length != 2)
{
    throw new InvalidCommandException(this.Input);
}

string relPath = this.Data[1];
this.inputOutputManager.ChangeCurrentDirectoryRelative(relPath);
```

Problem 4. Modify abstract command

Since you are not going to use the fields except the input and the data[], you can remove the fields, constructor parameters and properties for the once left.

Here is how you class should now look:

```
public abstract class Command : IExecutable
{
    private string input;
    private string[] data;

    protected Command(string input, string[] data)...

    public string Input...

    public string[] Data...

    public abstract void Execute();
}
```

Problem 5. Modifying the Command Interpreter:

Now it's time to make some changes in the command interpreter, to start creating the desired command using reflection and not the switch-case.

First thing you can do is delete the whole switch.

Now let's start preparing for the creation of the objects. First thing you need to make in the ParseCommand method is create an object array filling it with the input and the data:

```
private IExecutable ParseCommand(string input, string command, string[] data)
{
    object[] parametersForConstruction = new object[]
    {
        input, data
    };
```

Now we need to get the type of the command that we want to create. Here is the time to profit from the alias attribute. We want to take all types from the current assembly -> take the first one that has an Alias attribute with a name equal to the wanted command. Here is how this should look:

```
Type typeOfCommand =
    Assembly.GetExecutingAssembly()
        .GetTypes()
        .First(type => type.GetCustomAttributes(typeof(AliasAttribute))
                .Where(atr => atr.Equals(command))
                .ToArray().Length > 0);
```

Another thing we will need is the type of the command interpreter, but it could be easily taken.

```
Type typeofInterpreter = typeof(CommandInterpreter);
```

Now it's time to create the desired command and we are going to do this by using the activator, and passing it the type of command and the parameters for construction:

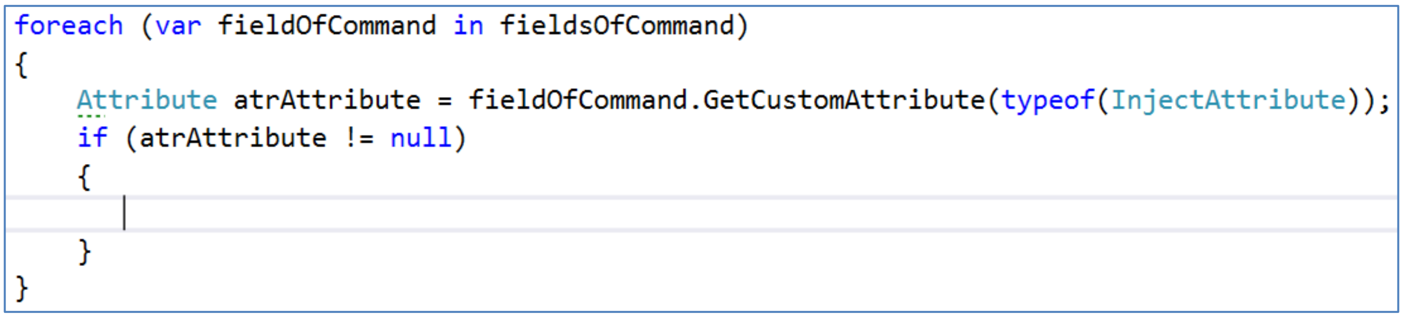
```
Command exe = (Command)Activator.CreateInstance(typeOfCommand, parametersForConstruction);
```

Now that we've created the wanted command, we might want to inject the needed field in the command, by passing it the instance from the command interpreter. This is why we will need to get the fields of both of the types of classes.

Here is how we do this:

```
FieldInfo[] fieldsOfCommand = typeOfCommand
                                .GetFields(BindingFlags.NonPublic | BindingFlags.Instance);
FieldInfo[] fieldsOfInterpreter = typeofInterpreter
                                .GetFields(BindingFlags.NonPublic | BindingFlags.Instance);
```

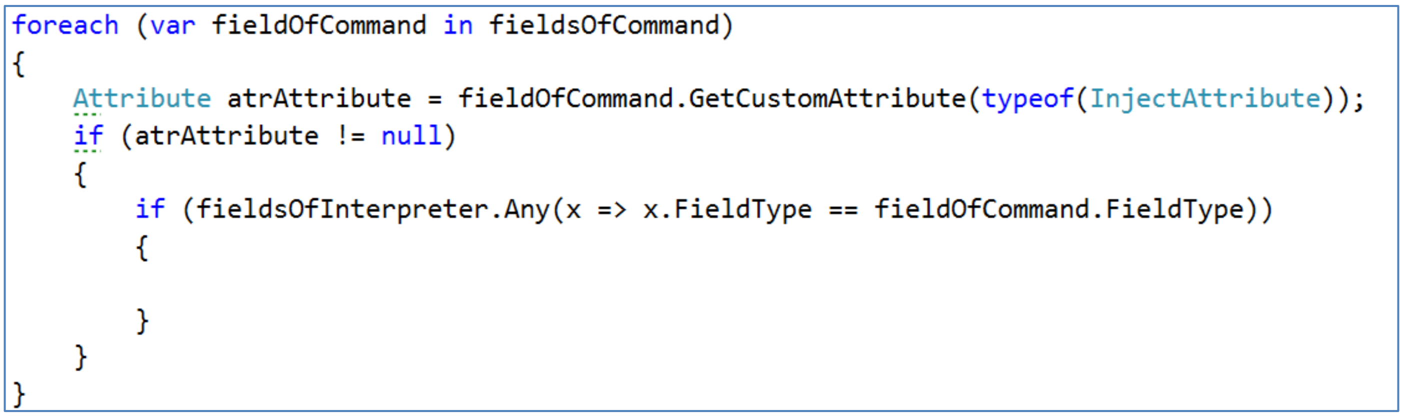
Now we would like to iterate through the fields of command. In it, take the custom attribute of type InjectAttribute and check whether it is not null:



```
foreach (var fieldOfCommand in fieldsOfCommand)
{
    Attribute atrAttribute = fieldOfCommand.GetCustomAttribute(typeof(InjectAttribute));
    if (atrAttribute != null)
    {

    }
}
```

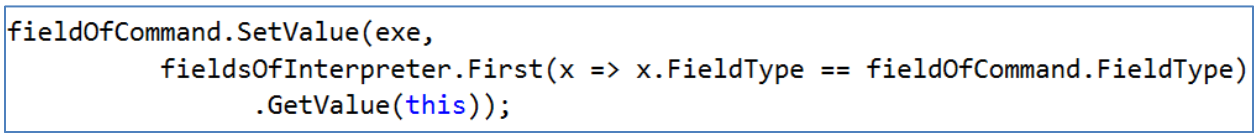
Next thing we need to check is whether there are any fields in the array of fields of the interpreter, that have the same type as the current one from the command :



```
foreach (var fieldOfCommand in fieldsOfCommand)
{
    Attribute atrAttribute = fieldOfCommand.GetCustomAttribute(typeof(InjectAttribute));
    if (atrAttribute != null)
    {
        if (fieldsOfInterpreter.Any(x => x.FieldType == fieldOfCommand.FieldType))
        {

        }
    }
}
```

Finally in the inner most if, we should set the value of the current field, by passing the exe(command) and for a value, you can get it from the fields of the interpreter, by taking the first that has the same type and taking its value from the current object (this)



```
fieldOfCommand.SetValue(exe,
          fieldsOfInterpreter.First(x => x.FieldType == fieldOfCommand.FieldType)
                .GetValue(this));
```

Finally after the for-each, we should return the current created command (exe)

Now we are done with all the refactoring and if you start the program, everything should be working as before. However, now if we need to implement a new command, all we have to do is inherit the abstract command class and set the corresponding attributes, but we do not need to change or add anything to the command interpreter. Not to mention this code is way cooler and shorter than the switch. However you should know that know we've reduced the performance, but this is a tradeoff we can afford.

Here is a cool article which you might want to check out:

http://www.codeproject.com/Articles/615139/An-Absolute-Beginners-Tutorial-on-Dependency-Inver