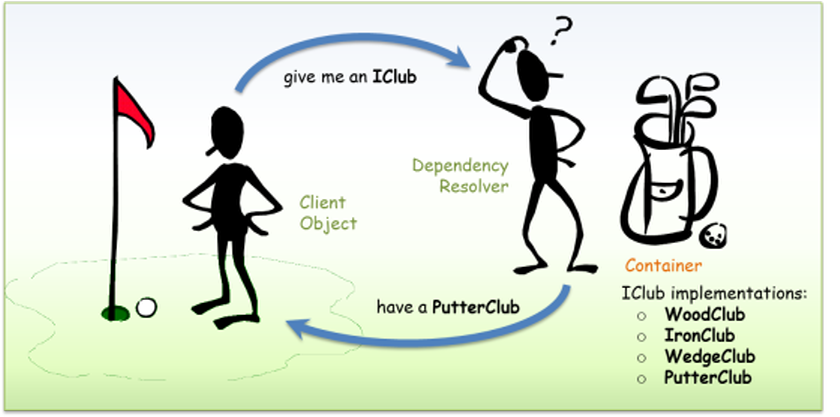
Exercises Dependency Injection



**Exercise 1:**In this exercise you refactor a small console application so that it works in accordance with the DI design pattern.  
  
Open the Console app ‘ExerciseDI\_FeedReader’. The project contains 3 entity classes (BlogFeedReader, PodCastFeedReader, YouTubeFeedReader). Objects of these classes are used in the FeedService class in order to return a specific feed to the Main().

Graphical user interface, text, application

Description automatically generated

When you run the project, the text “Audio:item 1” appears in the console.

Take a look at the FeedService class.

You’ll notice there is a tight coupling between the FeedService class and the PodCastFeedReader class.

Your task is to solve this dependency.

Make sure that the FeedService class can be used not only for returning podcasts but also for returning YouTube videos and blog posts.

Of course you must also test the GetFeed() method from the FeedService class.

**Exercise 2:**  
In this exercise you add extra functionality to an ASP.NET MVC application.

Open the ASP.NET MVC application ‘ExerciseDI\_MusicStore’.

Goal is to extend the Shopping Cart page by calculating the amount to be paid taking into account any possible discount.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

1. **Create folder ‘Services’**

This extra functionality or business logic is added to the application in a service layer. So first add a new folder ‘Services’. Inside this folder you code your business logic but taking into account the IOC principle, it’s important to abstract this service layer. The code below is just an example, the exact interface and classes you need to implement, will be explained in paragraph 2.

public interface IMyService

{

... some service methods

}

and then have your controllers work only with this abstraction through injection of the service class.

public class MyController: Controller

{

private readonly IMyService \_service;

public MyController(IMyService service)

{

\_service = service;

}

public ActionResult MyAction()

{

... call some methods on the service layer

}

}

1. **Add ‘IDiscountService’ and 2 implementations to the services layer**

Add the interface **IDiscountService** containing 1 method:  
  
int GetDiscount(List<CartItem> items);

Add the class **DiscountNumberOf** that implements the interface.  
GetDiscount() returns 0 when the shopping cart contains less than 5 items.  
When you buy more than 5 items but less than 10, you receive a discount from €5. Who buys 10 or more items gets a €10 discount.

Add the class **DiscountTotalPrice** that implements the interface.  
GetDiscount() returns 0 when the amount to be paid is less than €25. Between €25 and €50, the discount is set to €5. From €50 onwards, you receive a €10 discount.

1. **Add an MSTest Test project ‘MusicStore.Unittests’**

Rename the test class into *DiscountTotalPriceUnitTest*. We’ll use this class to test the methods of the *DiscountTotalPrice* class. Write the following test methods:

* GetDiscount\_ArticlesPrice10\_Returns0()
* GetDiscount\_ArticlesPrice30\_Returns5()
* GetDiscount\_ArticlesPrice60\_Returns10()

Add a *DiscountNumberOfUnitTest* and code some test methods.

When all the tests pass, you can use the DiscountService in the app.

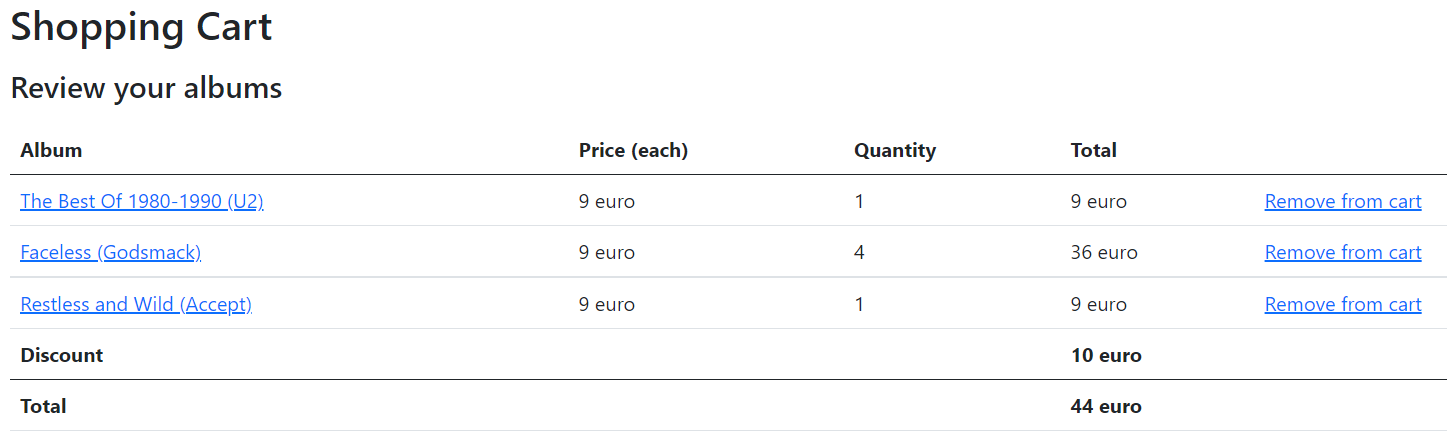
1. **Adjust the ShoppingCartController**

In the ShoppingCartController, inject the IDiscountService via the constructor.

Adjust the Index() action by calculating the discount. Pass the returned discount to the corresponding view by using a ViewData dictionary.

1. **Adjust the ShoppingCart Index view**

Calculate the amount to be paid taking into account the received discount. Also show the discount.



1. **Adjust Startup.cs**

Question is how the controller knows which implementation of the IDiscountService to use to calculate the discount. You can compare this with the question how the app knows which DBContext is used.

All the needed services are added to the *builder.Services* collection in *Program.cs*.  
  
Register the needed *IDiscountService* implementation:

builder.Services.AddScoped<IDiscountService, DiscountTotalPrice>();

// builder.Services.AddScoped<IDiscountService, DiscountNumberOf>();

Notice that this service is not registered as a singleton but scoped which means that each http request gets a new instance to calculate the discount.

Check if you get the discount according to the registered *IDiscountService*. Try both!

**Exercise 3:**

Understanding the life cycle of Dependency Injection (DI) is very important in ASP.Net Core applications. The DI Container has to decide whether to return a new instance of the service or provide an existing instance. In startup class when you configure the service, you must make a decision about the lifetime of each injected service.

There are 3 methods to define the lifetime of services:

* AddTransient
* AddScoped
* AddSingleton

To understand the difference, you can recreate the sample project explained on this link:

<https://www.c-sharpcorner.com/article/understanding-addtransient-vs-addscoped-vs-addsingleton-in-asp-net-core/>