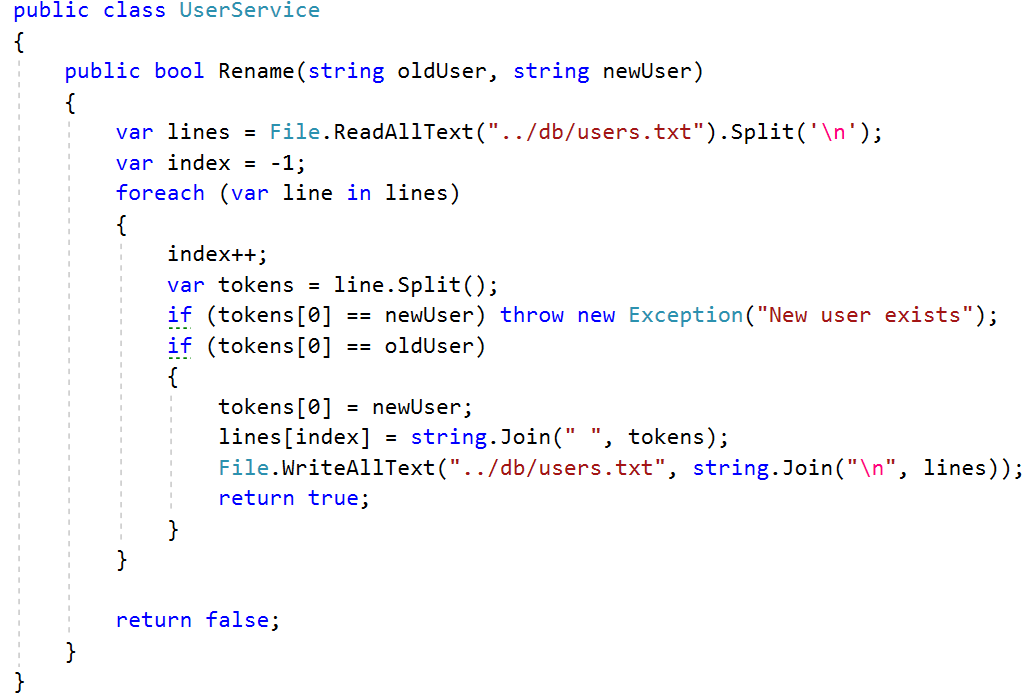
# Dependency Management Automation

## The Problem

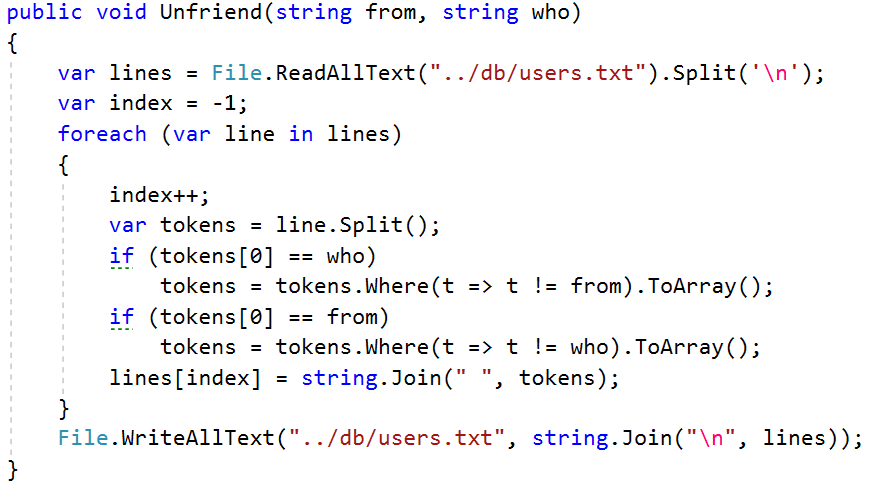
Once the application starts to grow, the operations and the data begins naturally to get organized, most commonly in classes and methods. But then, certain **operation starts to require another operation** to be done and data to be provided. This network of objects can be easily mistaken and can **lead to tight coupling, completely untestable code and hard extensibility**. All the things a developer **does not want** to happen.

The following scenario shows the problems from above and how it can affect the future of the project.



This sample code reads a text file with users’ data and changes the username of one of them. If users try to **rename their username to an already existing user, it raises an error**.

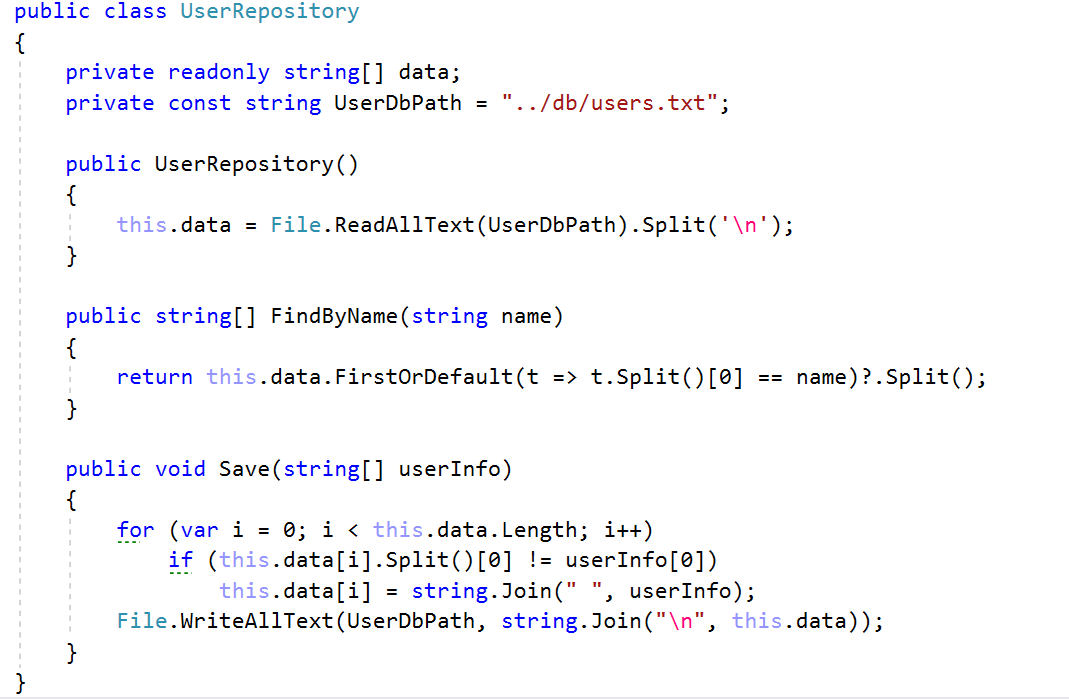
Despite the code **looks correct** at first sight, **it is not**. Not only that it has bugs, but there **are several other problems** here. To illustrate it better, let’s introduce a **second method that removes a friendship** between two users:

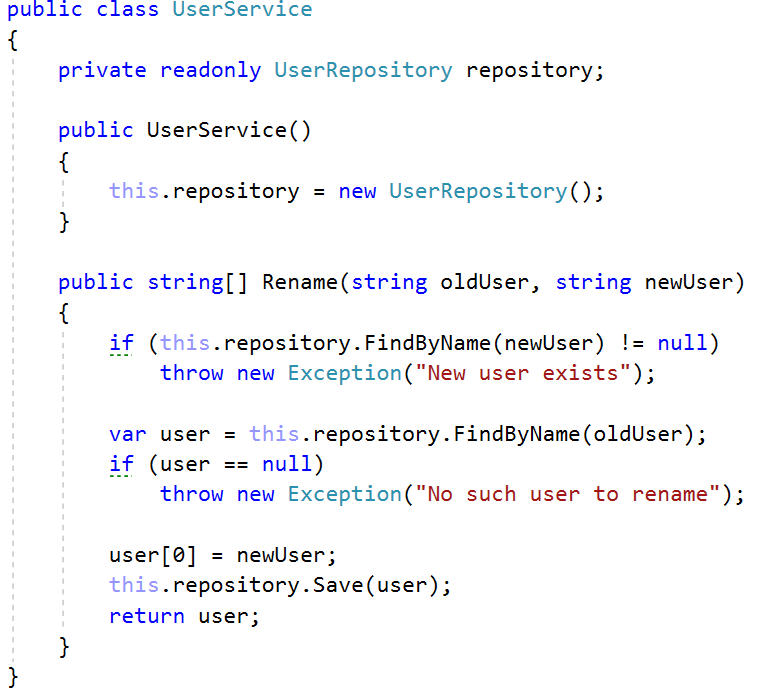


Let’s now examine the problems:

1. Once you need **to change where the “users” database is located**, you will need to find every single line of code it is used in, and **replace** it
2. These methods are **completely untestable**. We can never be sure what data exactly is in the database when the unit tests are running. **Faking data is impossible**, thus we can’t find the bugs in these methods easily
3. If we change the database logic from directly reading and writing files to SQL Database, we would need a **complete rewrite** of the UserService class (and any other using the data), because the **business logic is highly coupled to the data operation**

So, the most common step we need to take to resolve these problems, is to **decouple** the business logic and data retrieval:

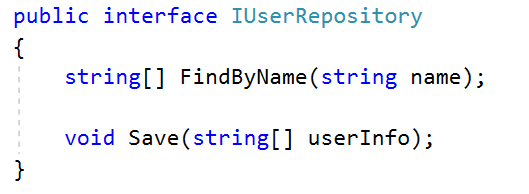


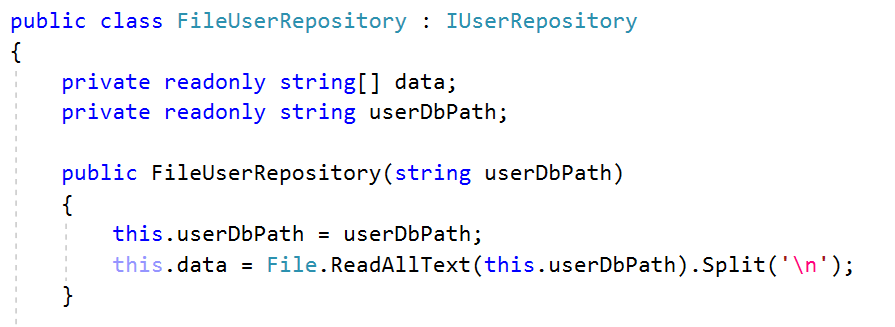


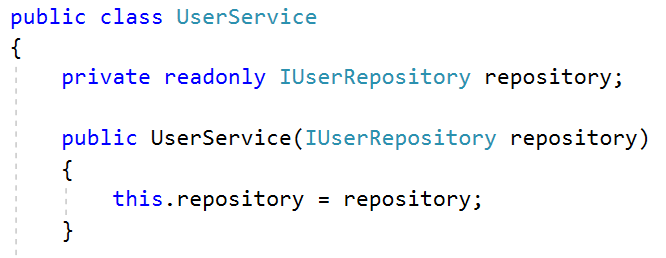
This is kind of better, yet with a lot of problems. If the UserRepository changes the logic of retrieving data, the UserService class will remain unchanged, which is good.

We still cannot test our UserService, because it **highly relies on this concrete Repository** (UserRepository). Also, the UserRepository operates only with **one database**. If we need to reuse the logic with another database of users, we cannot. Every **new instance of** UserRepository **will operate with one and the same file** (UserDbPath).

To fix these problems, we need to **stop creating concrete dependencies**. We need to **depend on abstractions** and receive them from outside. That’s what we call “**Dependency Injection**”:

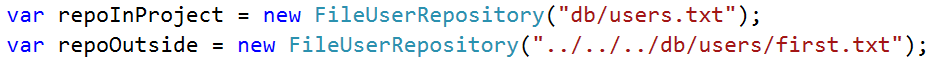




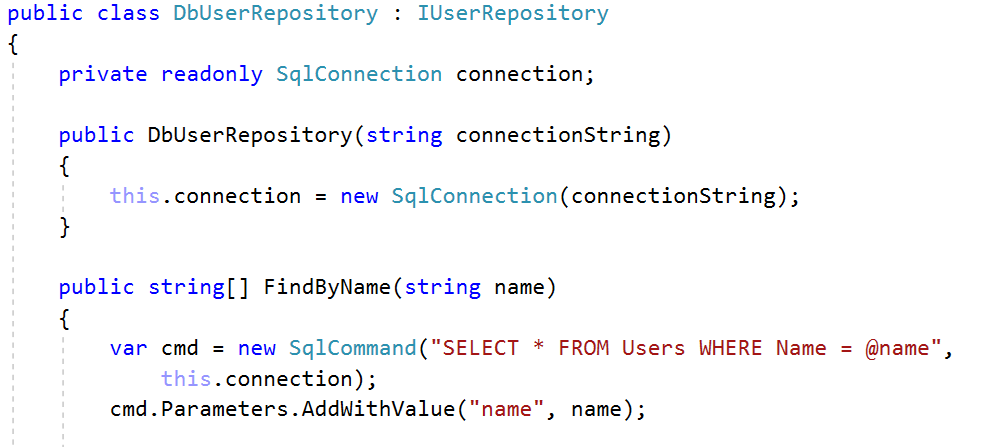


What we have achieved now:

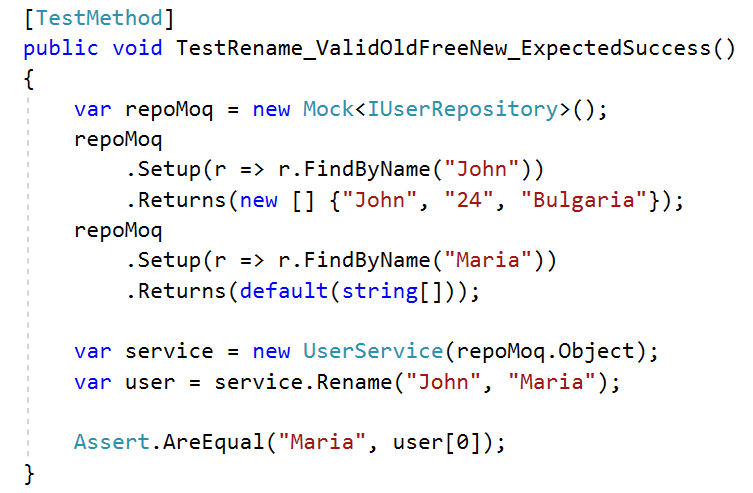
1. We can use the FileUserRepository with different database (file) paths depending on the situation



1. We can introduce a class that operates with **SQL Database, implementing the same interface** and using it in certain situations (giving it to the service as well)



1. We can **Mock** the IUserRepository to test our Service:





All cool we say, but as long as we have only these two objects in our applications. When **they grow to hundreds or thousands, their manual management will become hell**.

In the exercise below we will go through a **series of steps in order to setup an abstraction, which will manage the dependency graph** of an application, once it’s configured on startup. A technique called **Inversion of Control** often depends highly on **Dependency injection**, thus we will walk through **setting up an automation framework for resolving recursive dependencies**, as well as **inverting the control of the application**, so the framework starts the app and decides which object to call with which respective dependencies, **removing completely the need of manually creating service objects**.

## Final Result

A codebase which:

1. Allows the user to register **mappings between abstraction and implementation**
   1. If none is registered, to find if there is **only one** implementation and resolve it
2. Allows the user to register **mappings between abstraction and already created object**
3. Creates an object (or **takes it from cache** of already created objects) resolving all its dependencies **recursively**, if they are registered as mappings

## Overview

Once the application starts, it will **parse the container configuration**. In other words, a developer can tell **which implementation** to be used when an abstraction (interface) is required, by **configuring** the framework. This is done programmatically by filling a map (dictionary) **between Interface types and corresponding class type**; by filling a map (dictionary**) between Interface types and objects of classes** implementing this interface; by **annotating** a certain class with an attribute [Component] telling the framework to make the mapping or by not creating second implementation of an interface.

When the configuration is parsed and information is collected, the **application invokes some operations depending on the user events**. These operations might require some of the registered abstractions in order to work correctly. The **framework will provide these dependencies from the configuration**, which **magic** will make the application to work correctly.

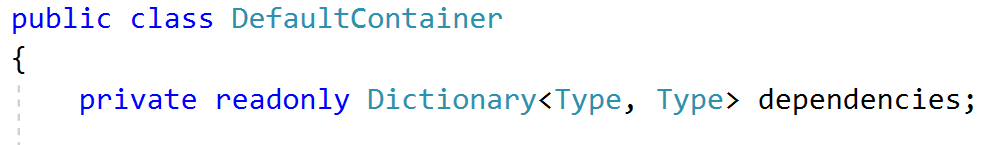
## Execution

It’s time to process to execution. Create a new project and dive deep into **reflection** and **recursion**.

### Storing Mappings

Basically, our framework will have **two main operations** – **storing** mappings and **getting** (or creating) objects from the stored mappings. .NET Library has a class named Type which stores **metainformation** about any **primitive or complex** type (interfaces, classes, enumerations, etc., …). We will use this metainformation to resolve correctly the objects: The Type class provides us an **information** about **what methods the class has, constructors, attributes** and so on and so forth.

Firstly, we need to store an in**formation about the abstraction type mapped to the information about the implementation type**. The most natural way is to store the association (mapping) in a dictionary. Let’s create a class named DefaultContainer which has this dictionary as a field named dependencies:



### Resolving Mappings

Let’s imagine the following situation:

1. IUserService will be mapped to UserService:

public class UserService : IUserService

1. UserService requires in the constructor IUserRepository and IForumRepository

public UserService(IUserRepository userRepo, IForumRepository forumRepo)

1. IUserRepository will be mapped to DatabaseUserRepository

public class DatabaseUserRepository : IUserRepository

1. DatabaseUserRepository requires in the constructor IForumRepository

public DatabaseUserRepository(IForumRepository forumRepo)

1. IForumRepository will be mapped to FileForumRepository

public class FileForumRepository : IForumRepository

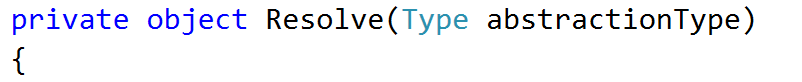
In this situation, the dependencies field will be filled as follows:

dependencies[typeof(IUserService)] = typeof(UserService);

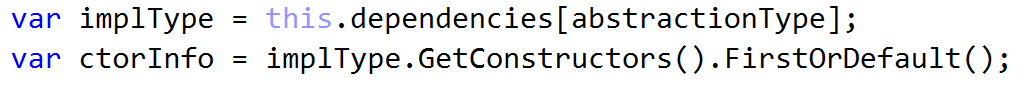
dependencies[typeof(IUserRepository)] = typeof(DatabaseUserRepository);

dependencies[typeof(IForumRepository)] = typeof(FileForumRepository);

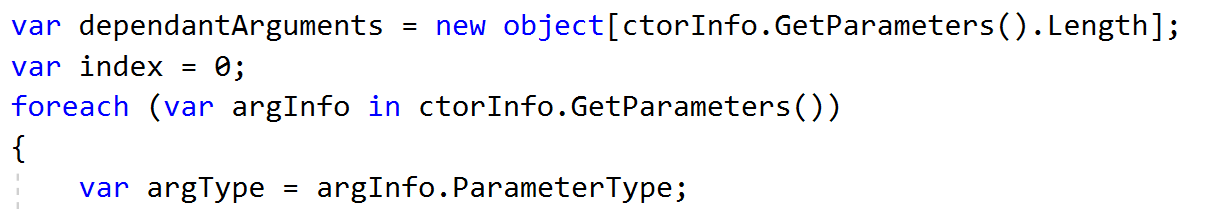
Let’s create in our DefaultContainer class a method called Resolve(Type) which will try to return an instantiated object by given abstraction type:



Once the resolving mechanism tries to get typeof(IUserService) it will find the associated UserService type and will start extracting **information about its constructor**.



ctorInfo variable now contains a special **reflection** type ConstructorInfo which, just like the Type class, gives **metainformation about constructors**. In our case we need to **extract the arguments** of the UserService constructor (IUserRepository and IForumRepository) in order to instantiate them (invoke their constructors). Once they are instantiated we need to instantiate our UserService (invoke the constructor through the ConstructorInfo again). The **invocation of constructor via reflection** needs an object[] with all of its arguments. That’s why before iterating through the arguments **we will create an object array – with size equal to the arguments count.**



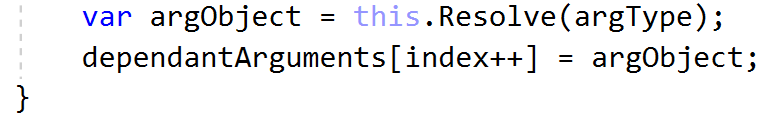
In the **first iteration** of the foreach loop the argType will be typeof(IUserRepository)and in the **second** it will be typeof(IForumRepository).

Once we have received typeof(IUserRepository) we need to **copy the logic from above**: extract its constructor, then its arguments.

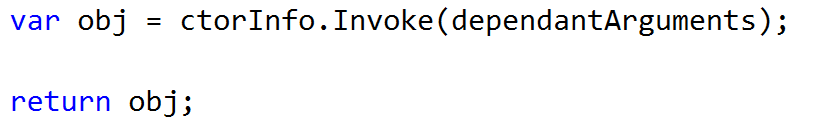
But all of a sudden, **its arguments start to require other arguments** in their constructors. And we **cannot simply nest** foreach **loops with constructors 10 levels down**. And what if there is an 11th level? We can never be sure when the chain breaks.

And **how to repeat all and the same logic over and over again**, until the last dependent object has no parameters in the constructor?

The answer is: **recursion**! Yes, we need to **recursively call the logic from above**, but this time – with the type of the parameters (typeof(IUserRepository) for example).

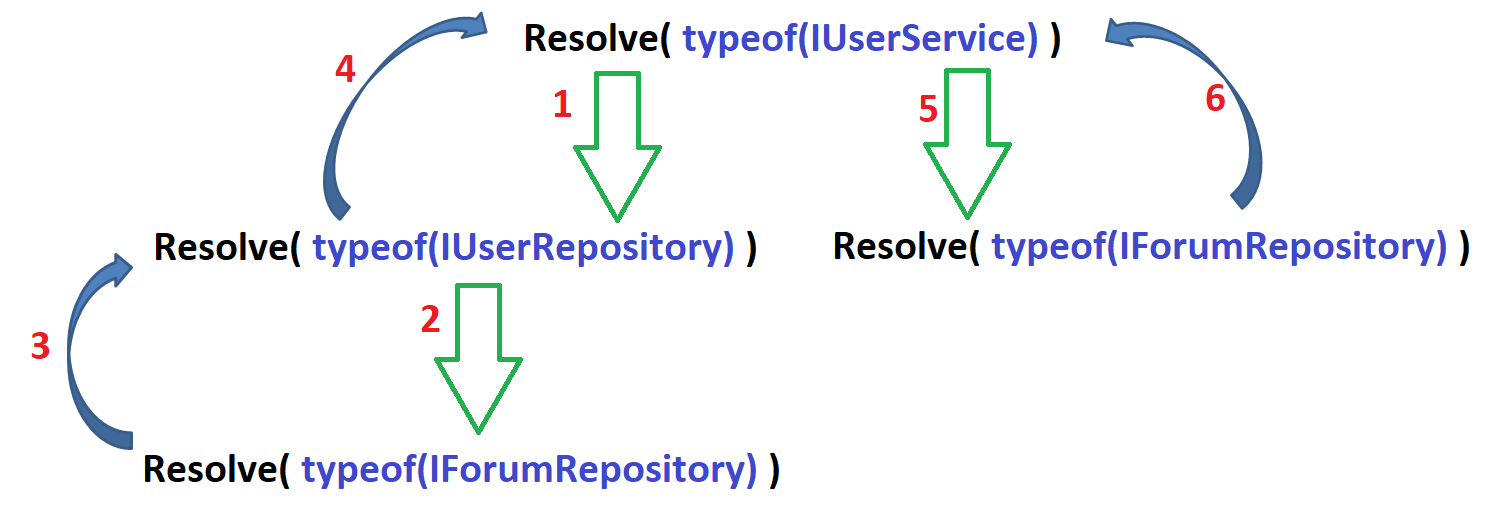


And then create the object (UserService) with its dependencies via Invoke() and then **return** it.



Let’s track the recursive calls.

1. Initially call Resolve(typeof(IUserService))
2. Because UserService has **2 parameters** in the constructor, it will call for first time with the first parameter: Resolve(typeof(IUserRepository))
3. Because DatabaseUserRepository has **1 parameter** in the constructor it will call Resolve(typeof(IForumRepositor))
4. Because FileForumRepository has **no parameters** in the constructor it will **not trigger the** foreachloop, thus **no recursive calls will be done here**. The object will be created and returned.
5. When (4.) returns, it will **go back to (3.)** and try to continue with the foreach loop, but **it has only one iteration**, so it will create DatabaseUserRepository and (3.) will return
6. When (3.) returns, it will **go back to (2.)** and try to continue with the foreach loop. It has one more iteration – IForumRepository – so it will call Resolve(typeof(IForumRepository))
7. Because FileForumRepository has **no parameters** in the constructor it will **not trigger the** foreach loop, thus **no recursive calls will be done here**. The object will be created and returned.
8. When (7.) returns it will **go back to (6.)** and try to continue with the foreach loop. It has only two iterations so **no more left**. It will create UserService and return it

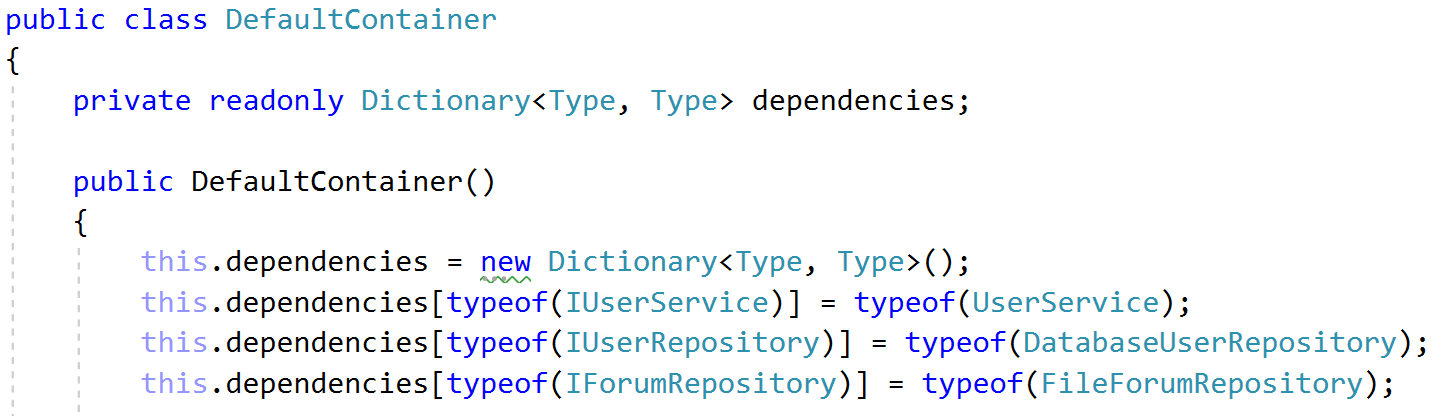


### First Try

Let’s have our objects from above, but they call each other to test that all of them are created.

#### Hardcode Mappings

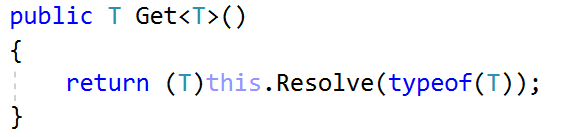
In our DefaultContainer constructor we will **hardcode** the three mappings from above.



#### Access to Resolving Mechanism

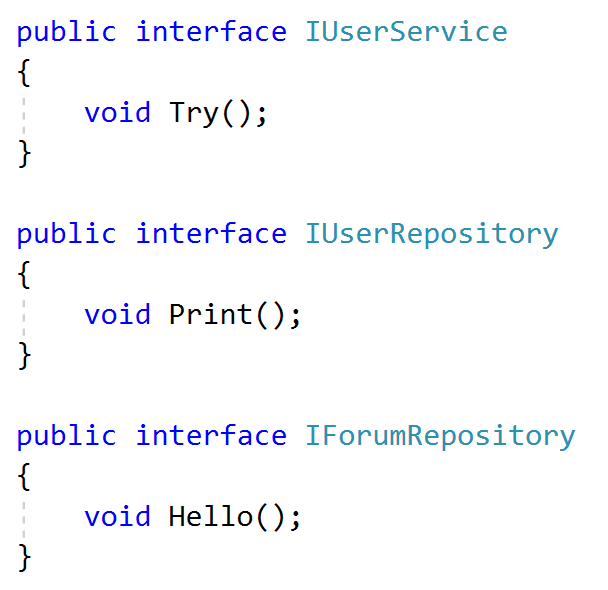
The Resolve method is private. And is not very handy for the outside world. It will be good for the users to just use the needed **type as generic**, and internally to be **converted** to typeof(Generic).

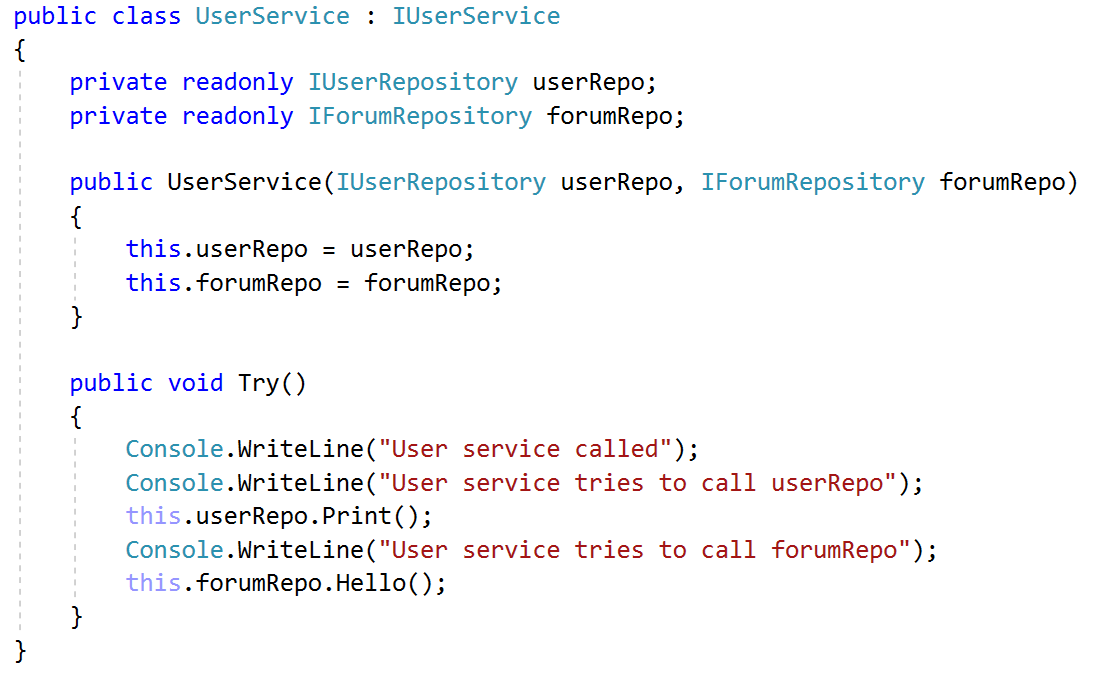
Create Get<T>() method which proxies the Resolve method

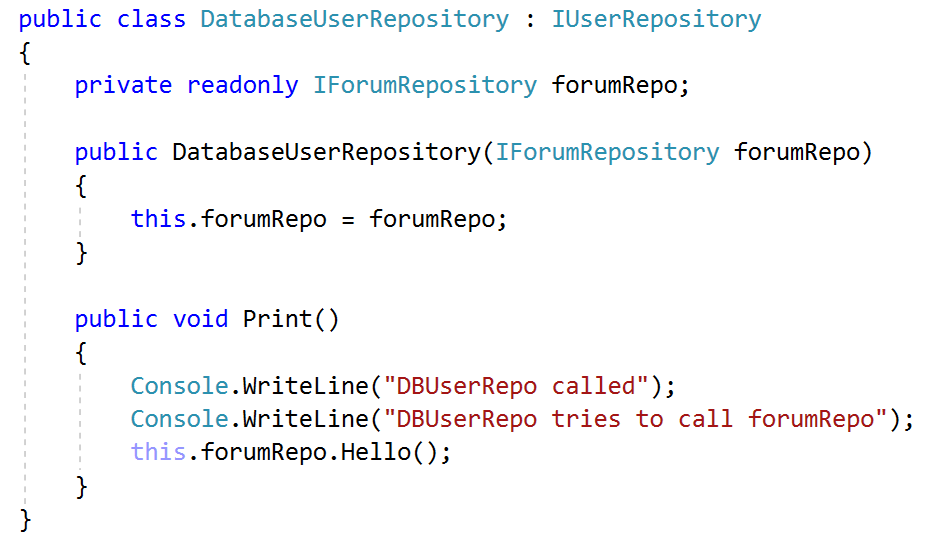


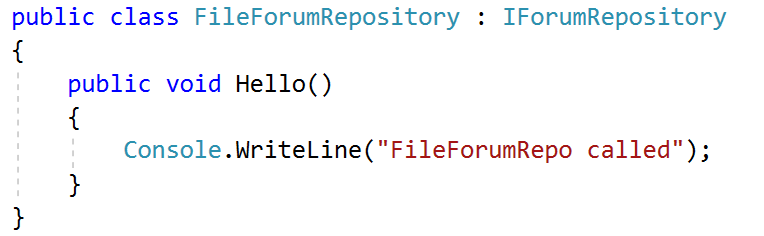
#### Create Classes

Let’s create somewhere in the application the relevant services and repositories



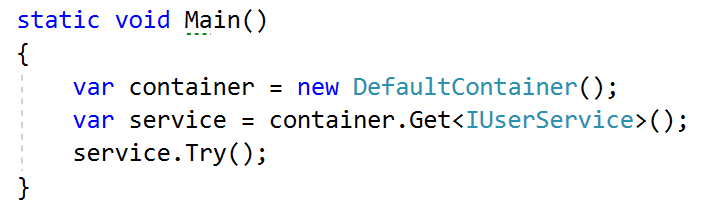




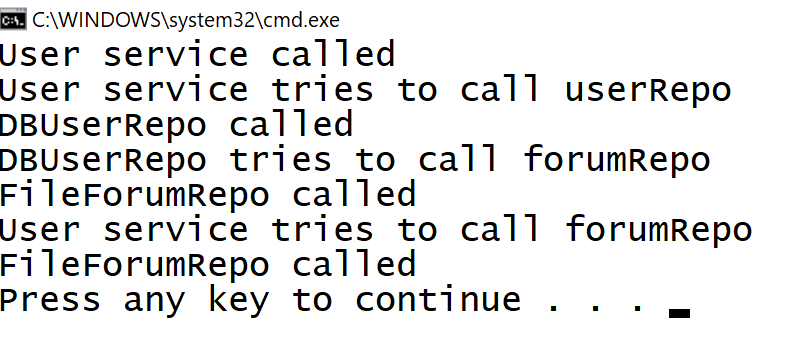


#### Use the Container

Let’s use the newly created Get<T> method to require IUserService. If everything is good we will receive UserService object with all its dependencies.



If the following result is shown

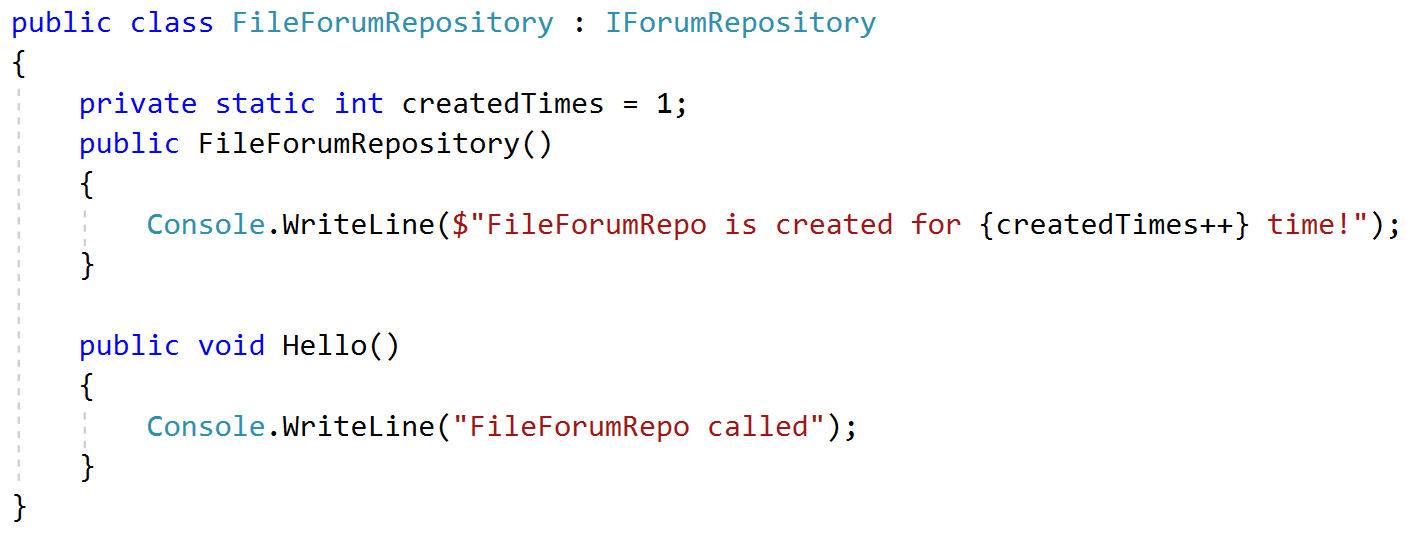


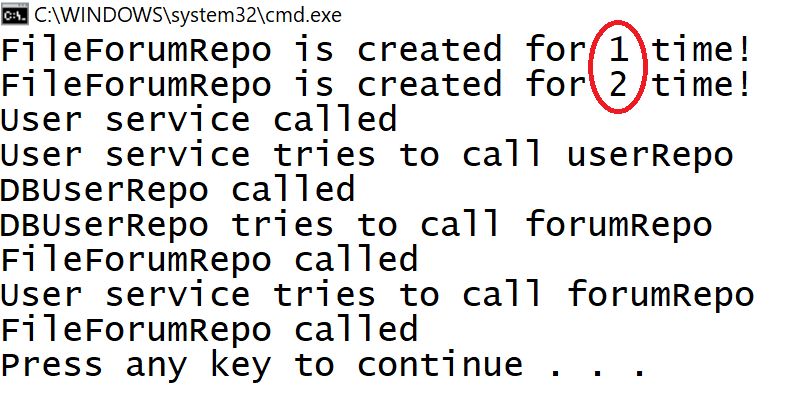
Then our framework fulfills **its basic requirements**. **Congratulations**! Now, **the hard work** is just about to start!

### Retrospection and Roadmap

So far, our framework **creates object by given abstraction** **and recursively resolves their dependencies**, which is a big leap. But there are several problems we cannot just omit:

1. Remember the recursive call graphic? IForumRepository is resolved **twice**. If it has its own dependencies they will be twice resolved too. You can track if it’s created twice by adding a WriteLine in its constructor. If it appears twice – we have a problem





1. We have promised ourselves that we will **configure the container not only manually**, but single implementations (as ours) will be **detected**, also we can **decorate** classes with an attribute. **We don’t have that functionality**; the configuration is hardcoded in the constructor.

Then our agenda is:

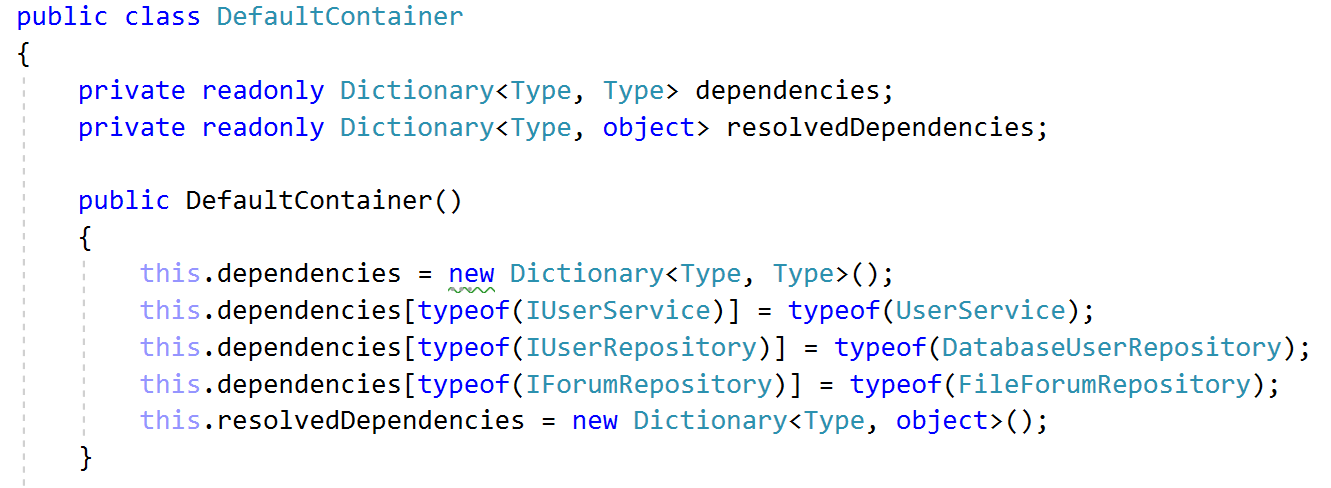
1. **Cache** already resolved objects
2. Implement [Strategy Pattern](https://en.wikipedia.org/wiki/Strategy_pattern) for different types of configuration
   1. Strategy for **adding manually** mappings
   2. Strategy for scanning **attributes**
   3. Strategy for scanning **single implementations**

### Object Cache

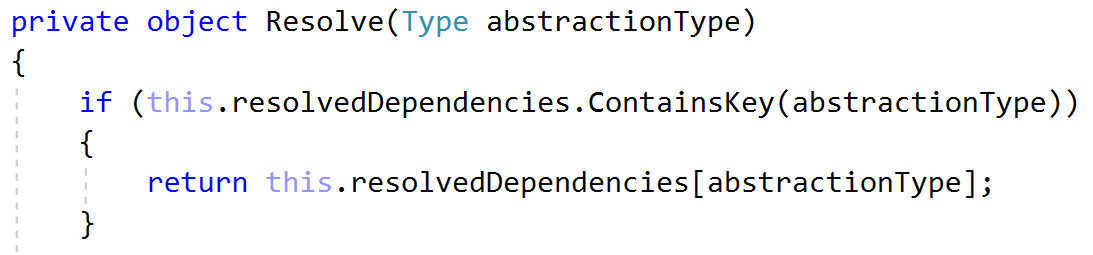
We need to **stop resolving the same object over and over again**. Once it is created, it should be **put in a cache** and next time it is required, to **lookup the cache** and take it from there, instead of using the recursive mechanism.

The cache will consist of **association between an abstraction** (typeof(Interface)) and the **created** object (**Type -> object**).

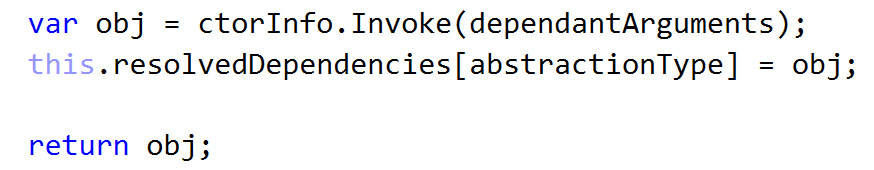
Create a field in the DefaultContainer class called resolvedDependencies and initialize it in the constructor.



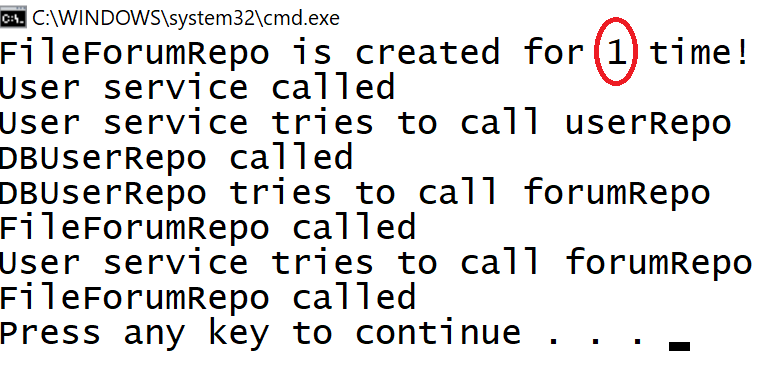
Every time an object is asked to be resolved, **we need to look up the cache**. If the object is **found** there, it can **simply be returned** and no recursive calls will be done:



But we will never enter in that block of code, because **we never add objects in this dictionary**. Every time we **create successfully an object it should be added in the cache** and then returned by the Resolve() method



Running the application again, should produce **only one instance** of the FileForumRepository class and we will see as output only one line printed in the constructor

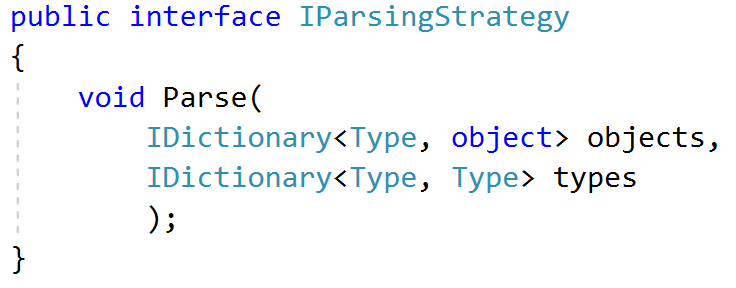


### Parsing Strategies

We need a **centralized way of parsing classes** in order to **configure** our container. Different implementation might behave differently (as we said before – one will parse **attribute**, another: interfaces with **only one implementation** and so forth). Every **Strategy** will have its own parsing logic, but will **receive the dictionaries with types association in order to fill them**, once it scans the relevant types.

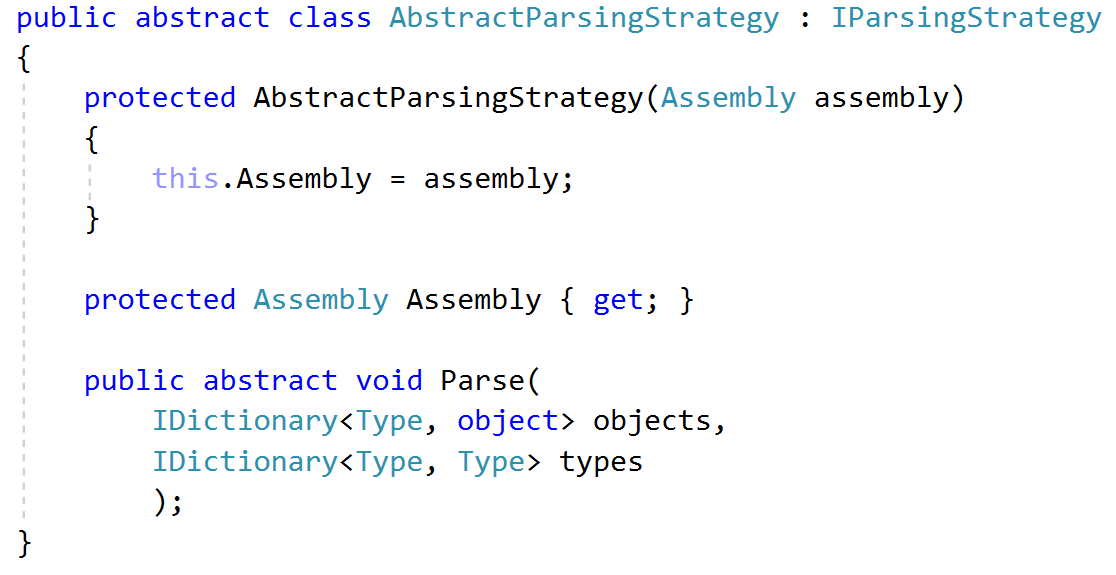
#### Strategy Interface

We will call our strategies – **Parsing Strategies**, so we need an interface IParsingStrategy with one method Parse() receiving dictionaries to fill



#### Abstract Strategy

All our strategies **will scan types in some area**. The area of types is often an **assembly**. It’s a special reflection type that provides **information to a whole project** (containing classes and other types, location, name, etc…). Let’s create an **abstract class** which will be parent for all our strategies, which needs an assembly for later usages



#### Attribute Parsing Strategy

This strategy will be responsible for **scanning types in an assembly which have some custom attribute**. All classes decorated with certain attribute should be added to the dictionary.

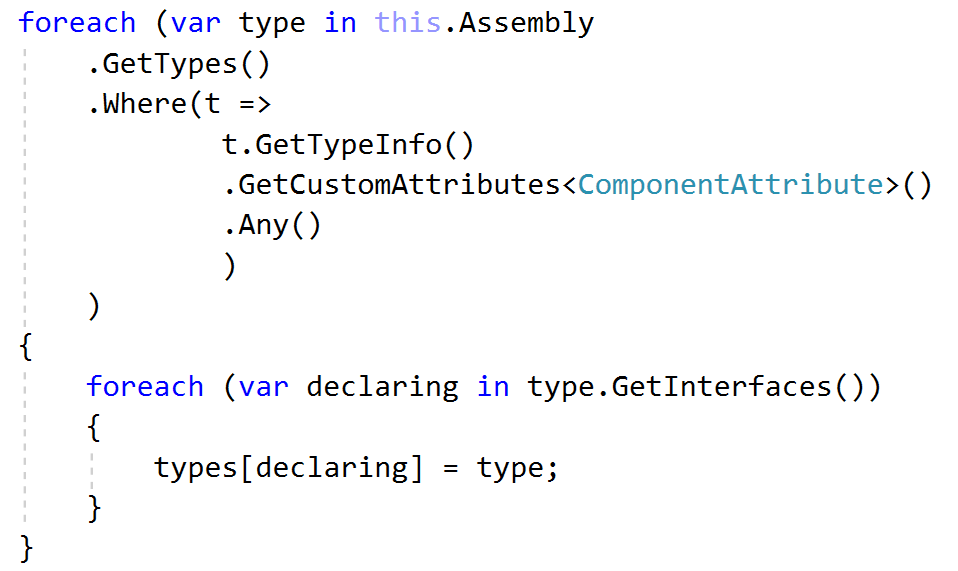
##### Create Attribute

To scan classes for a certain attribute, we need to **create it** first. Let’s call it Component



##### Create Strategy

Now we are ready **to implement our parsing logic**. It will be **encapsulated** in the Parse() method of a class extending the AbstractParsingStrategy. Let’s call it AttributeParsingStrategy. What we need there is logic that **goes through all types in the assembly which has custom attribute** [Component] and add them as values in the types dictionary. But **what about the keys**? The keys will be **all interfaces that this class implements**.

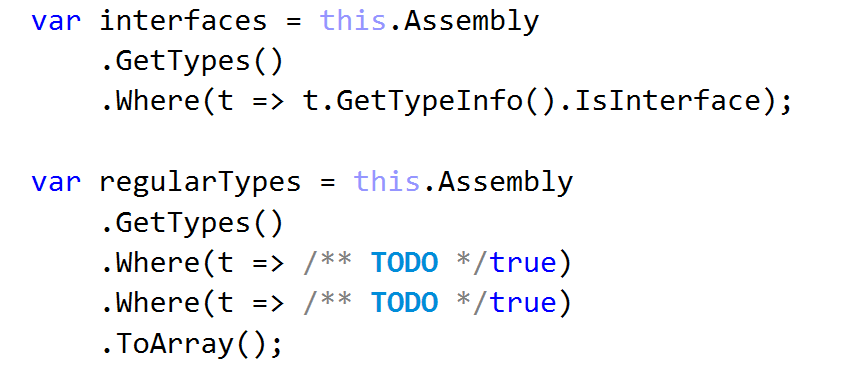


#### Single Implementation Strategy

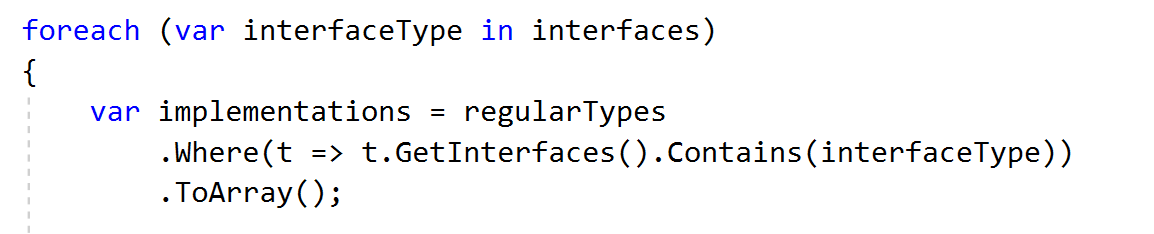
This strategy will scan the assembly for interfaces that has **only one implementation**. Let’s call it SingleImplementationParsingStrategy.

Unfortunately, in the Type API **there is no method for getting all implementations of a certain type**. Which is normal, because they can be in another assembly we don’t even know about.

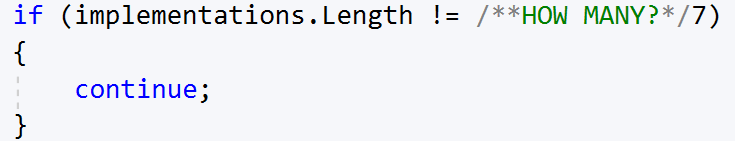
So, we need to **split** the types in two arrays – one for **interfaces** and one for **regular types**



Then **check** for each interface **how many** from the regular types implements it



If **none** or **more than one** class – it’s not our desired case. The size of this array **should be exactly 1**. Otherwise – skip this iteration

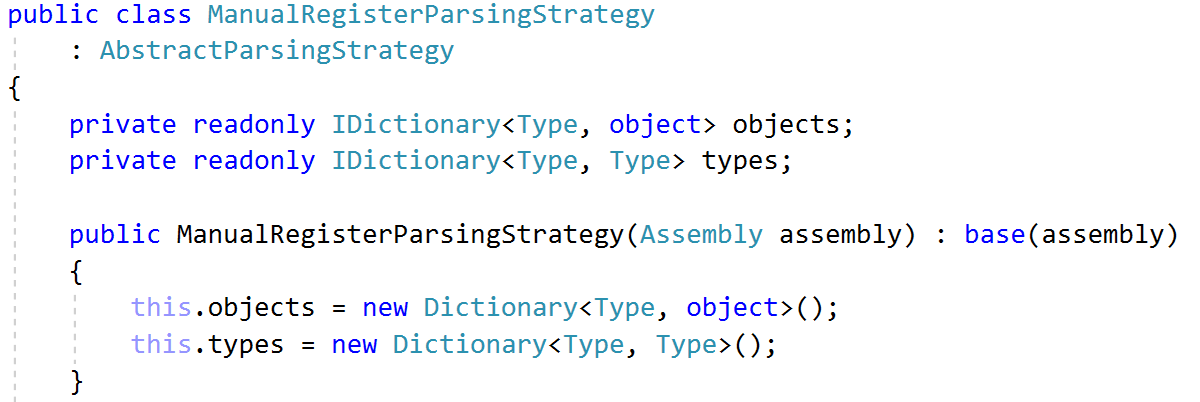


If it’s exactly one – take this **interface type for key** in our dictionary and **the only implementation for a value**



#### Manual Configuration Strategy

The last strategy here will be responsible for **manually** configuring the container. The user can register types which will go to internal dictionaries. Once the Parse() method is called, it will **move the registered types to the dictionaries** passed to the method.



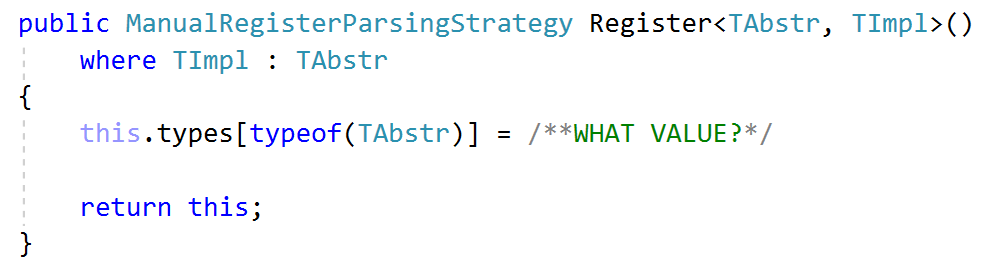
Now we need to provide the user two ways of configuring:

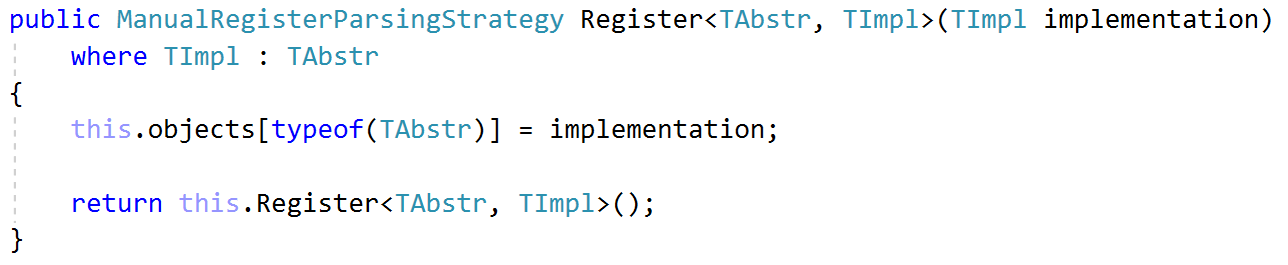
1. Abstract **Type** to Concrete **Type**
2. Abstract **Type** to Manually Create **Object** of Concrete Type

We can do it by providing **overloads** of a method giving access to the dictionaries. We may call it Register:

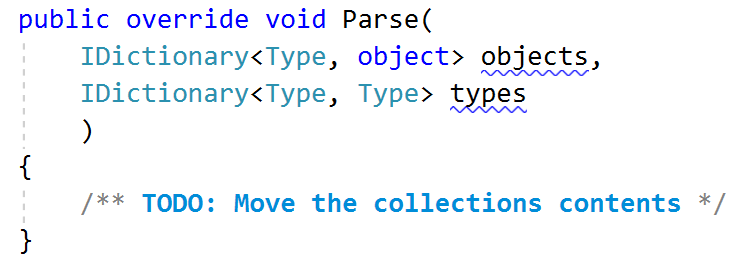
1. Register<T>(V impl) will register in dictionary – typeof(T) as key and impl as value
2. Register<T,V>() will register in dictionary typeof(T) as key and typeof(V) as value

There is a general rule here: **The <V> generic should always inherit from the <T> generic** (it will be real shame if we register an abstraction and implementation that does not match ☹)





Then the parsing logic will move **the contents from the internal collections to the passed** ones



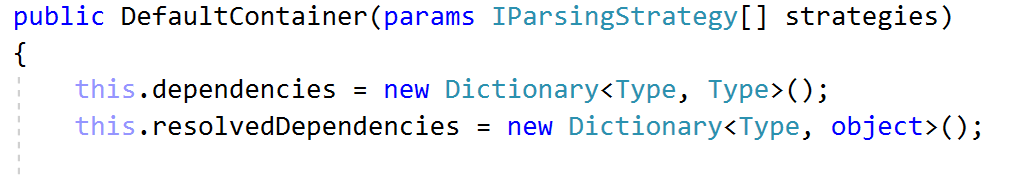
### Configuring the Container

We have our parsing strategies, but **how we can configure** the container?

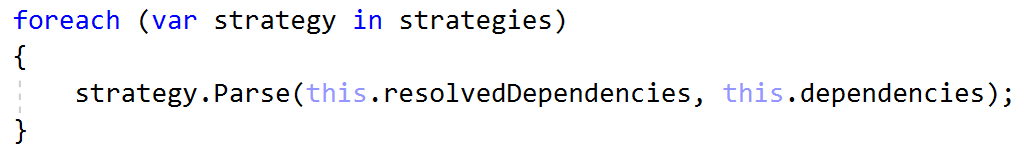
The answer is to **pass different amount and types of parsing strategies once the container is initialized**. It internally will invoke their Parse() method and pass its internal collections. Thus, the **container will receive all parsed classes** in its collections and will be ready to resolve objects

#### Pass Strategies to Container

Let’s make our container’s **constructor to receive a variable amount of strategies** upon initialization



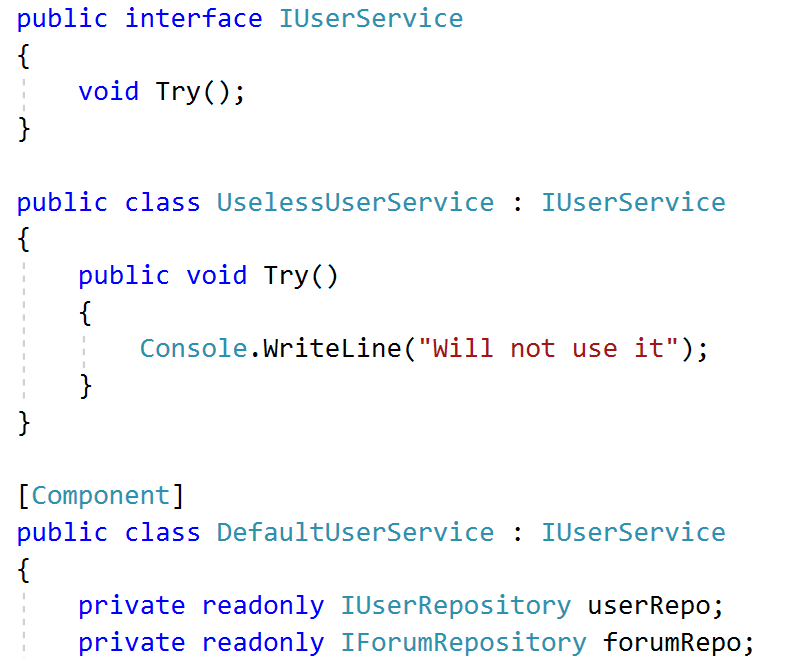
And invoke Parse() of each strategy giving it the internal collections, so they can fill them

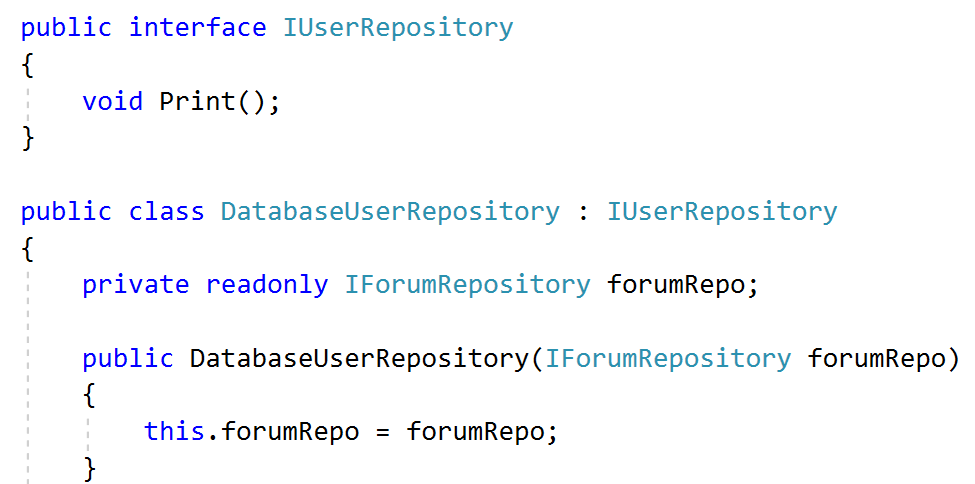


### Last Try

**Congratulations**, we are near the **end of our journey**. The only thing we need is to **test** the framework.

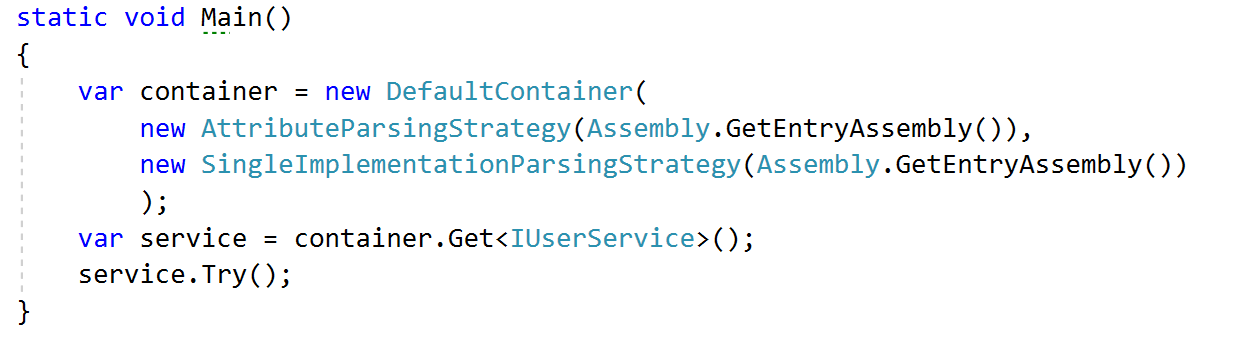
Let’s test it by **having an interface with two implementations**, one of which is **decorated** with [Component] and by **another interface that has only single implementation**



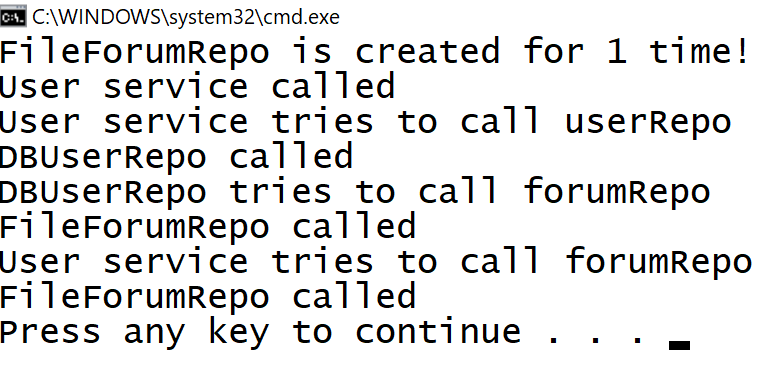


The only difference from the last time is we have created a **new class implementing** IUserService named UselessUserService and **rename** UserService to DefaultUserService

Since DefaultUserService is decorated with [Component] and the DatabaseUserRepository and FileForumRepository are **single implementations of their interfaces**, this means that the AttributeParsingStrategy and the SingleImplementationParsingStrategy will be enough



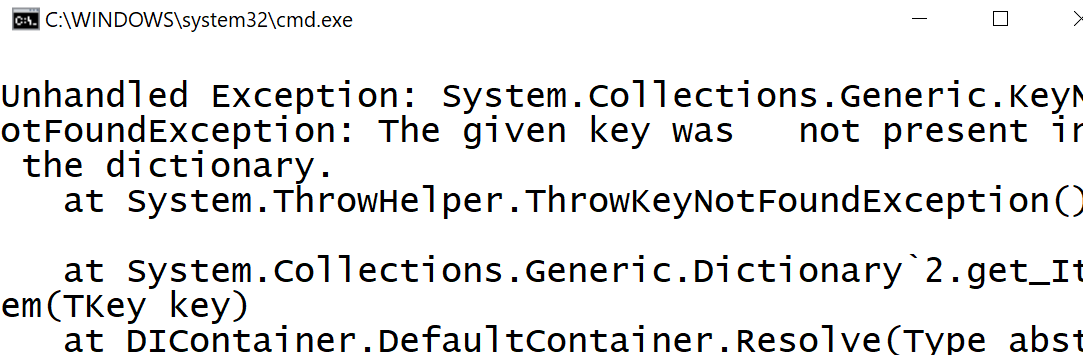
The result should be

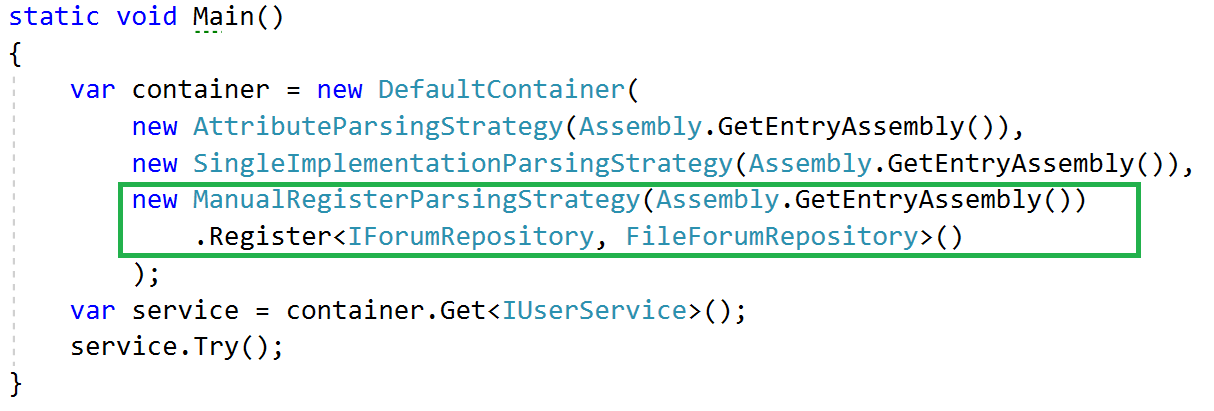


But what if we introduce **another implementation** of IForumRepository and **none of both** is decorated with [Component]?



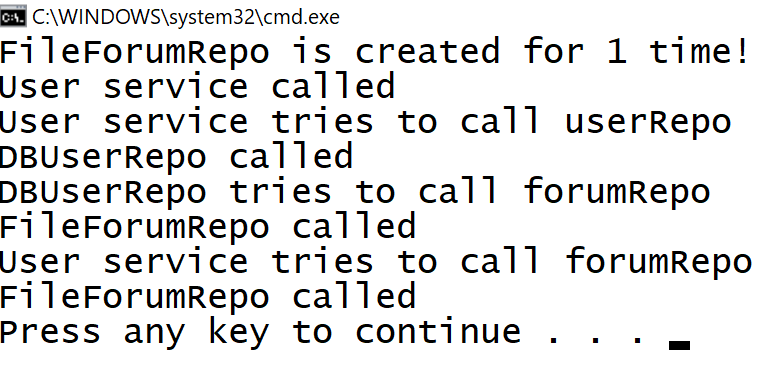
The result is



That’s because **none of the strategies has parsed an implementation of** IForumRepository and typeof(IForumRepository) is not found in dependencies dictionary. We need either to remove one of the implementations, decorate one of them or **use the Manual Configuration strategy** and register manually which of them to be used 

Because the Register<>() method in ManualRegisterParsingStrategy returns itself (return this) we still have a strategy returned object and can be passed to the constructor. This **would not be possible**, if the methods return type was **void**.

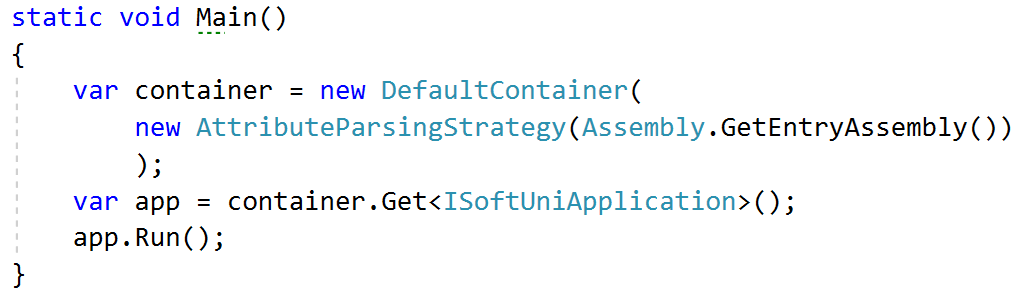
Upon run, we should receive



If so – **congratulations**, we have just created a **decent dependency injection container for educational purposes**. If not, go back and find what is mistaken.

## Inversion of Control

For the sake of **good practices**, the container should **not be used more than once** – for creating the application and running it. All the other dependencies should be created accordingly and relevant objects to be passed.



An example of such application can be found in the [Marketplace](https://judge.softuni.bg/Contests/267/CSharp-OOP-Advanced-Exam-Sample-2-August-2016) exam preparation. But you will have to refactor it, first. Go ahead!