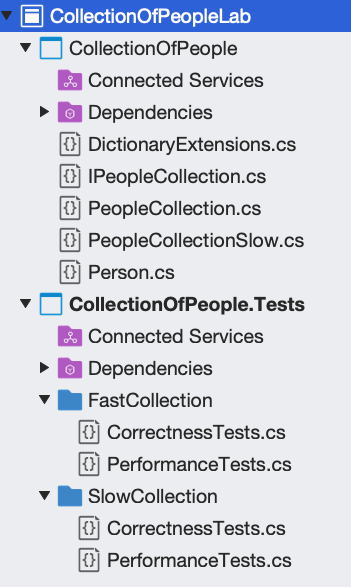
# Lab: Data Structures Augmentation

This document defines the lab for ["Data Structures – Advanced (C#)" course @ Software University](https://softuni.bg/trainings/4273/data-structures-advanced-with-csharp-november-2023).

## Project Skeleton

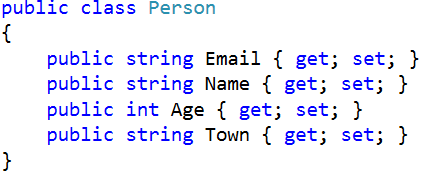
We are given a **Visual Studio project skeleton** (unfinished project) holding the unfinished classes **Person**, **PersonCollection** and **PersonCollectionSlow** and **unit tests** covering the functionality of the "person collection" data structure. The project holds the following assets:



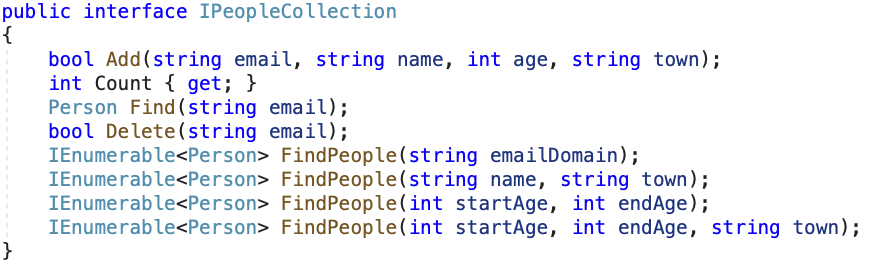
You need to support the following operations:

* **bool Add(email, name, age, town)**
  + **"**email**"** is unique for each person
  + Returns true if a new **Person** is created and false if the email already exists.
* **Person Find(email)** – returns the Person with that email or **null**.
* **bool Delete(email)** – returns true if the person is deleted, or false if it didn’t exist
* **IEnumerable<Person> FindPeople(emailDomain)** – returns a collection of matching people ordered by email ascending
* **IEnumerable<Person> FindPeople(name, town)** – returns a collection of matching people ordered by email ascending
* **IEnumerable<Person> FindPeople(startAge, endAge)** – returns a collection of matching people ordered by age ascending, then by email ascending. Borders are inclusive.
* **IEnumerable<Person> FindPeople(startAge, endAge, town)** – returns a collection of matching people ordered by age ascending, then by email ascending. Borders are inclusive.

First, let's look at the **Person** class. We may need to finish it later:



Next, let’s look at the interface **IPeopleCollection**. It defines the **operations** (methods and properties) to be implemented by the **"data structure"**:

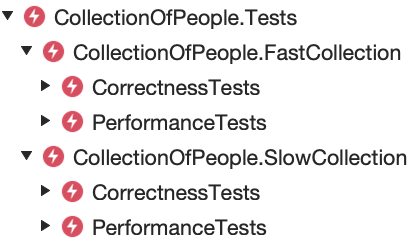


The next two classes **PeopleCollection** and **PeopleCollectionSlow** hold unfinished implementations of the **IPersonCollection** interface. These classes will be used to implement the data structure in two different ways – with list of people (simple but **slow implementation**) and by several dictionaries (**efficient implementation**):

The project comes also with **unit tests** and **performance tests** covering the functionality of the “**people collection**”.

## Run the Unit Tests to Ensure All of Them Initially Fail

**Run the unit tests**. All of them should fail:



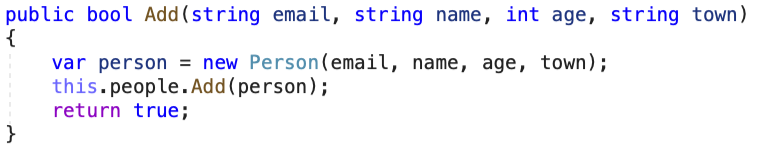
This is normal. We have unit tests, but the code covered by these tests is missing. Let's write it.

## Start with a Straightforward (and Slow) Solution

First, let’s start with a **simple, straightforward (and slow) solution** – implement the “people collection” data structure as **List<Person>**:

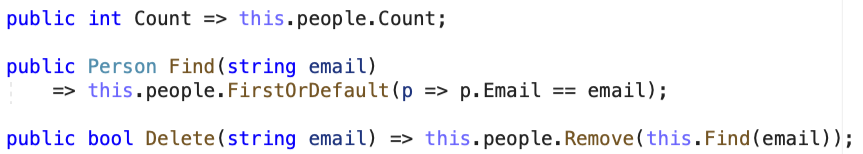


The **finder methods** could be implemented by **LINQ queries** with straightforward **filtering** and **sorting** using lambda expressions. First, let’s define the **Add(…)** method (You might need to create a constructor for the Person class):



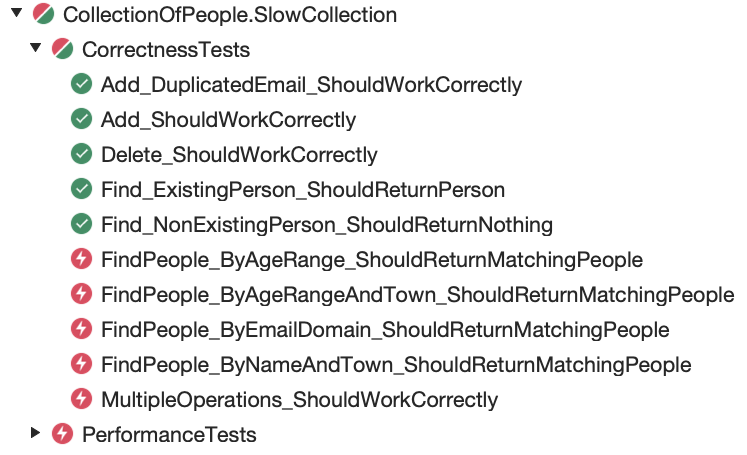
It just **creates a new person** and **appends it to the underlying list**. You have to check for existing person and **return false** if the person already exists.

Next, let’s add the **Count**, **Find(email)** and **Delete(email)** methods that work over the underlying list-based collection of people:



## Run the Unit Tests

Now **run the unit tests**. We should have five tests passed:

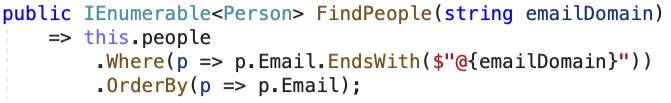


This is expected. Most functionality is still unimplemented. The **"**people collection**"** implementation works partially.

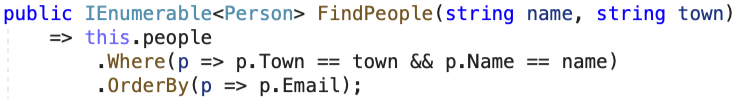
## Finder Methods in the Slow Solution

Now, let’s continue by implementing the **finder methods** in the slow list-based implementation.

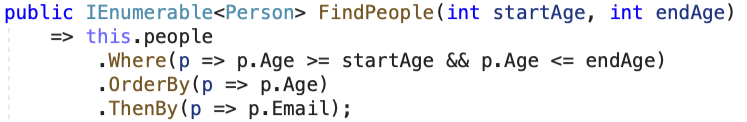
First, implement **finding people by email domain**. The results should be sorted by email:



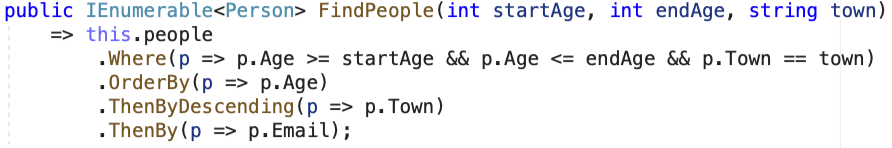
Similarly, implement **the other finder methods**. Finding people by name and town works the same way as finding by email domain:



Finding people **by age range** works by simple filtering the underlying list of people, then by applying the requested **sorting order** (sort first **by age**, then **by email** as second criteria):



Finally, we implement **finding by age range and town**, just like the previous finder methods:



All functionality seems to be implemented.

## Run the Unit Tests Again

Now, let’s run again the **unit tests** to check whether the implementation works as expected. Some tests pass and some tests fail:



Analyzing the failed tests, we could make one **conclusion**:

* All **performance tests** fail, so the algorithm is most probably inefficient.

## More Efficient Underlying Data Structures

Now let’s implement an **improved solution**, which uses **more efficient underlying data structures**.

Let’s first **define the data structures** needed to perform efficiently the required operations:

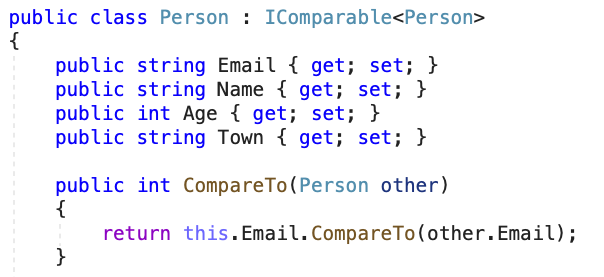
* To **find a person by email** we can use a **hash-table**. We expect zero or one person to match given email address (recall that the *email* is unique):



* To find all **people matching given email address domain**, we can use a **hash-table**. It will use the email domain as **key** and a sorted set of people as **value**:



Note that people for each email domain are **sorted internally by email**. In order this to work correctly, the **Person** class should implement **IComparable<Person>** and compare people by their email:



* To **find all** **people by name and town** we can use a hash-table. We can **combine the name + town** as a single string value and use it as **key** and use sorted set of people as value:



* **Finding all people by age range** needs a data structure that uses the **age as key** and keeps the **ages sorted**. The **values** could be a sorted set of people (sorted by email):



Use the **OrderedDictionary** from **Wintellect.PowerCollections** In the above structure, when we look for all people by range of ages, we will get all distinct ages in increasing order and for each age we will get a set of people sorted by email.

* Finding in a fast way all **people matching certain town and certain range of ages** is more complicated. We need **double mapping**: first **map towns** to some structure, which **maps ages** to **sets of people**. For example, we could have a dictionary that maps towns to sorted dictionary:



In the above structure, when we look for all people by town and a range of ages, we will first lookup to find the dictionary of ages for given town, then will get all distinct ages in the specified range in increasing order and finally for each age we will get a set of people sorted by email.

We have **five separate data structures** that work together to implement efficiently the operations from the “people collection” data structure. When we use a **combination of data structures,** we need to always keep all underlying data structures up to date:

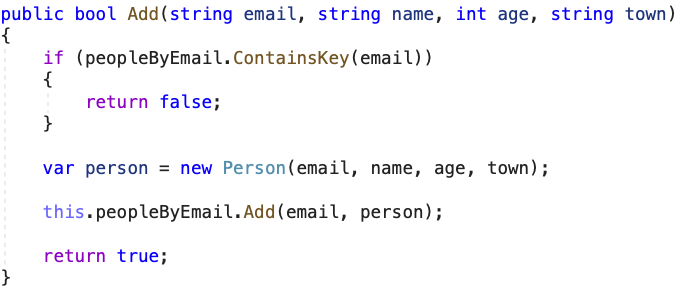
* **Add** needs to add the new data to all underlying data structures.
* **Modify** needs to update all the underlying data structures to hold correct data.
* **Delete** needs to delete the data from all underlying data structures.

We are ready to implement the operations from the **IPeopleCollection** interface based on the above underlying data structures. Let’s do it.

## Implement Add / Find / Delete

Let’s write the code **step by step**. We will implement the operations using only the hash-table that maps email to person. Later we will use the other underlying dictionaries for the other operations.

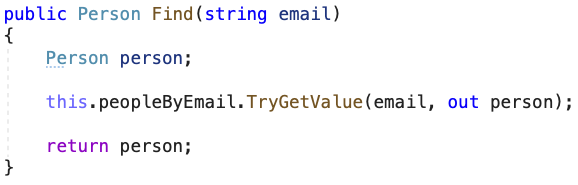
First, let’s write the **Add(…)** method:



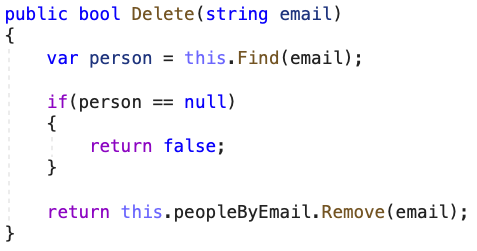
Next, let’s implement the **Count** property. It is trivial:



Next comes the **Find(email)** method:



Next, implement the **Delete(email)** method:



## Run the Unit Tests Again

Now **run the unit tests** to check what works and what does not work from the new **PeopleCollection** class:

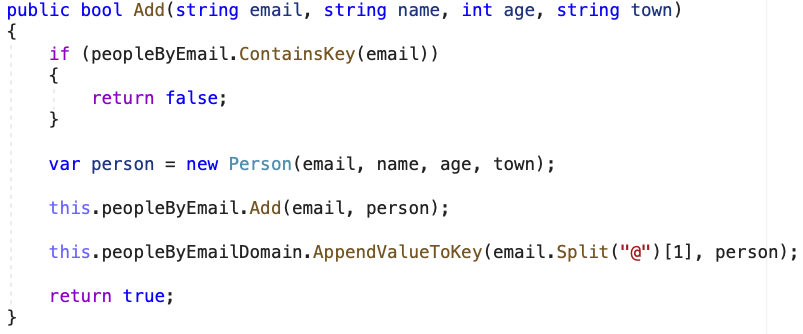


The results are hopeful: **add** / **find** / **delete** methods **work correctly** and at the same time their **performance is good**.

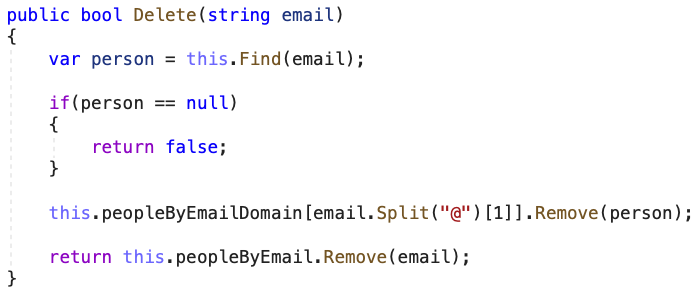
## Implement Find People by Email Domain

Now let’s implement the next operation: **find people by email domain**. We need 3 steps:

1. **Modify "add"** operation to append the new person to the dictionary **peopleByEmailDomain**:

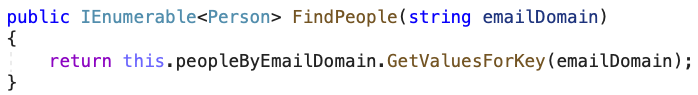


1. **Modify "delete"** operation to delete the person from the dictionary **peopleByEmailDomain**:



The **emailDomain** is guaranteed to exist in the **peopleByEmailDomain** dictionary, so we don’t need an additional check. The key is guaranteed to be created when the person is added by the **Add(…)** method.

1. Implement the method **Find(emailDomain)** to looking-up for people by email domain in the **peopleByEmailDomain** dictionary:



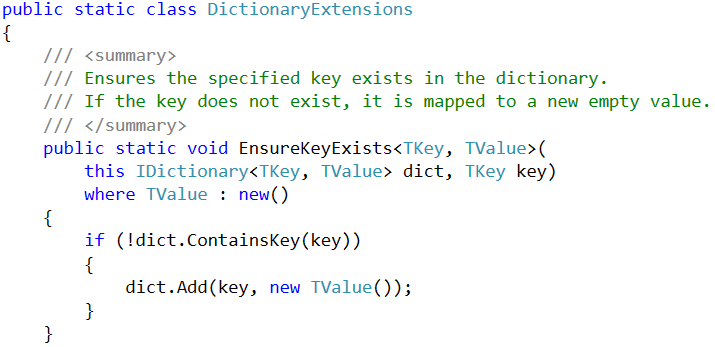
## Dictionary Extensions

Let’s now discuss the **extension methods** we use to **simplify working with dictionaries holding sets of values**. These methods are **generic** because we want to use them for any kind of dictionaries holding collections of values.

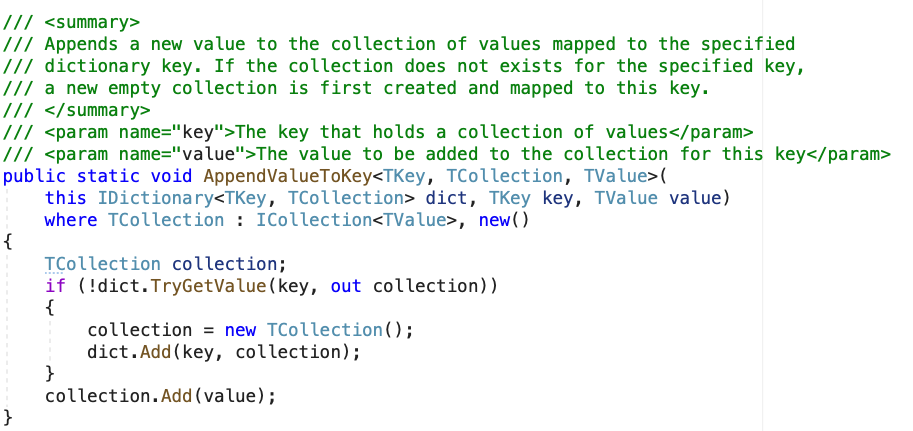
**Extension methods in C#** provide a special syntax for attaching methods to a class without changing or inheriting it. **Generic methods in C#** allow data types of the input and output method parameters to be generic – data types become parameters. Combining extension methods with generics can extend the C# dictionaries (**IDictionary** interface) and add very helpful methods to them that **save a lot of code**.

Look at the **DictionaryExtensions** class in your main project. It provides several extension methods for dictionaries.

The method **dictionary.EnsureKeyExists(key)** ensures a **key always exists in a dictionary** and **always** **holds a collection of elements**. It creates a new value (e.g. and empty set of people **new** **SortedSet<Person>()**) for the specified key when it does not exist:



The next helpful method is **dictionary.AppendValueToKey(key,** **value)**. It **adds a value to the collection of keys mapped to certain key**. If the key does not exist, a collection is first created:



Another very helpful extension method for dictionaries holding collection of values is **IEnumerable<values>** **dictionary.GetValuesForKey(key)**. It returns **all values for certain dictionary key**. It assumes the key holds a collection of values or does not exist. When the key does not exist, am empty sequence of values is retuned.

Graphical user interface, text, application

Description automatically generated

**Note**: if you are not expert in C# programming, generics, interfaces and extension methods, the above code might be complex for understanding, but this is not a problem. Just assume “**the code does its magic**” and you can use it like any other external library or API.

## Run the Unit Tests

We have implemented the **"find people by email domain"** functionality. Let’s **test it**. All tests covering the add / find / delete / find-by-email-domain functionality works correctly and fast:

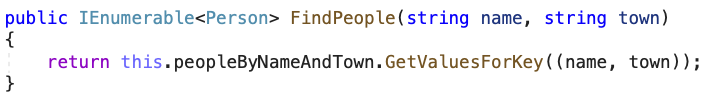


## Implement Find by Name and Town

Now let’s implement the **"find by name and town" functionality**. It is very similar to “find by email domain” First, it **combines the name and town** to get a **combined key**. This key is used to lookup in the underlying dictionary **peopleByNameAndTown** that maps {name + town} to set of people.

**Combining several keys** to obtain a single string value to be used as **combined key** can be done two ways:

* Concatenate several values into a single string.
  + For example, we can use the **formula:** **name + "|!|" + town**.
* Use the .NET class **Tuple<T1,** **T2,** **…>**. In our case we will use **Tuple<name,** **town>** as combined key.



Modify the **Add(…)** method to ensure the new person is added to the **peopleByNameAndTown** dictionary:



Modify **DeletePerson(…)** method to ensure the person is deleted from the **peopleByNameAndTown** dictionary:



## Implement Find by Age Range

Now let’s implement the **"find by age range" functionality**. It will use the ordered dictionary **peopleByAge** that maps ages to sets of people. Ages are sorted in increased order. Mapped people are sorted by email.

First, modify **Add(…)** method to ensure the new person is added to the **peopleByAge** dictionary:

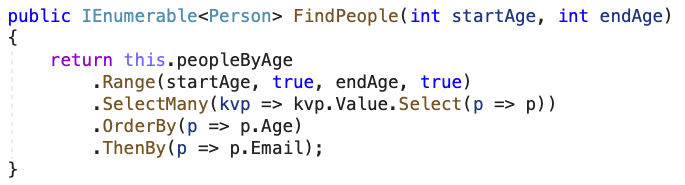


Next, modify **Delete(…)** method to ensure the person is deleted from the **peopleByAge** dictionary:



We don’t need to check whether the key **"person.Age"** exists in the **peopleByAge** dictionary, because it is guaranteed to exist. If the people exist by given email, the **Add(…)** method should have added it to **peopleByAge** dictionary and its age should be a valid existing key in this dictionary.

Finally, write the code that finds the people in given age range, sorted by age (as first criteria) and then by email (as second criteria):



The above method is more complex. It first performs a **lookup** in the balanced tree to **extract all people for given age range**. The result holds a dictionary:

* Its **keys** hold the **distinct ages** in the specified age range.
* Its **values** hold for each key a **sorted set of people** of this age (sorted by email).

The above method iterates over the returned range by using the **"SelectMany"** method from LINQ. This flattens the nested sets into a single enumerable. Finally, the matched people are returned as sequence of people **IEnumerable<Person>**, sorted by age and email (as second criteria).

## Implement Find by Town and Age Range

All failing tests should be related to the unimplemented **"find by town and age range" operation**. Let’s implement it. It is like the **"find by age range"** operation but is a bit more complicated. It first maps towns to **"**age-dictionary**"**, then lookups the age range and finally returns people in the specified age range.

First, let’s write the additional code in the **Add(…)** method:



Due to the extension methods, we have in the **DictionaryExtensions**, the above code is very short and simple:

* First it ensures the **town** exists as key in the dictionary **peopleByTownAndAge**. If it does not exist, it is created to hold an empty dictionary mapping ages to sets of people.
* Next it appends in the dictionary for the specified **town** a key **age** holding as a value the new **person**. For each town we have **"**age-dictionary**"**. For each age we have sorted set of people in the **"**age-dictionar**"**. Each person is added to the set of people for its town and age.

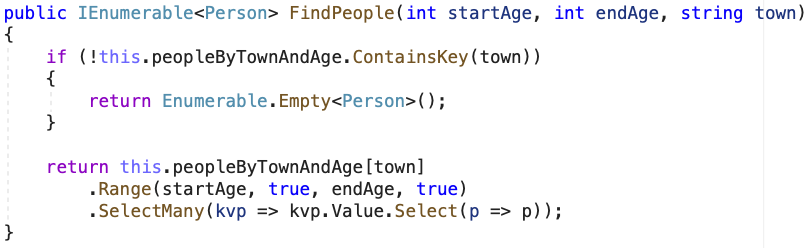
Recall that the dictionary **peopleByTownAndAge** maps towns to sorted dictionary that maps ages to sorted sets of people:



Second, let’s write the additional code for the **DeletePerson(…)** method. It is quite simple:



Finally, implement the **"find by town and age range"** functionality. It should return the matched people ordered by age (as first criteria) and the by email (as second criteria):



It first lookups the **town** in the **peopleByTownAndAge** dictionary. If it does not exists, returns an empty sequence of people. Next, if the town exists, we can look up the age range and return the expected sequence of people like in the **FindPeople(startAge,** **endAge)** method.

## Run the Unit Tests (Hopefully for the Last Time)

Run the **unit tests** again, hopefully for the last time. Now we have **passed all the tests** (functional tests and performance tests):



**Congratulations!** We have implemented correctly and efficiently the "people collection" data structure.