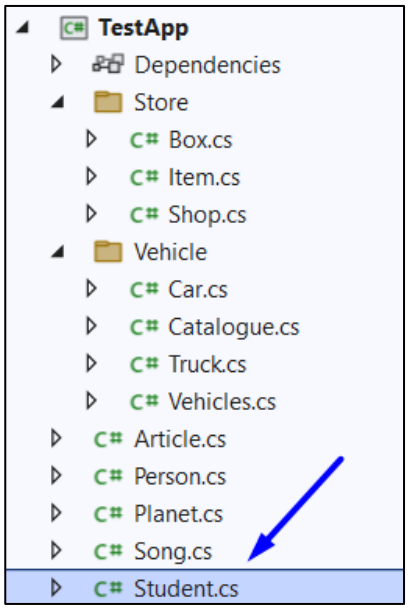
Exercises: Unit Testing Classes

Submit your solutions here: https://judge.softuni.org/Contests/4492/Objects-and-Classes-Unit-Testing-Exercise

1. Unit Test: Student

Look at the **provided skeleton** and examine the **`Student.cs`** class that you will test:



The class has **properties** for **first name**, **last name**, **age,** and **hometown**:

```
public class Student
{
    4 references
    public string FirstName { get; set; } = null!;

    4 references
    public string LastName { get; set; } = null!;

    3 references
    public int Age { get; set; }

    3 references
    public string Hometown { get; set; } = null!;
```

It also has a **method** that takes in a **string array** representing **students** in the form of:

"**`{first_name} {last_name} {age} {hometown}`**"

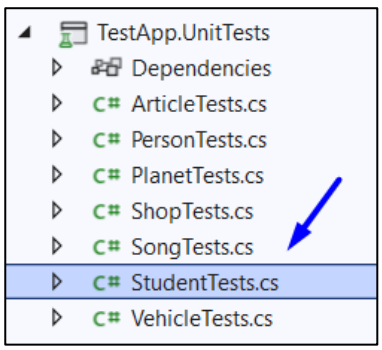
Also, a string representing which **town** the method should **filter** the students by and **return** them as a string:

```
    public string AddAndGetByCity(string[] students, string wantedTo
    {
        List<Student> studentList = new();

        foreach (string currentStudent in students)
        {
            string[] data = currentStudent.Split();
            string firstName = data[0];
            string lastName = data[1];
            int age = int.Parse(data[2]);
            string hometown = data[3];

            Student? student = studentList
                .FirstOrDefault(s:Student => s.FirstName == firstName
                                          && s.LastName == lastName);
```

```
            if (student is null)
            {
                studentList.Add(item: new Student()
                {
                    FirstName = firstName,
                    LastName = lastName,
                    Age = age,
                    Hometown = hometown
                });
            }
            else
            {
                student.FirstName = firstName;
                student.LastName = lastName;
                student.Age = age;
                student.Hometown = hometown;
            }
        }
```

```
        StringBuilder sb = new();
        foreach (Student student in studentList.Where(s:Student => s.Hometown == wantedTown))
        {
            sb.AppendLine($"{student.FirstName} {student.LastName} is {student.Age} years old.");
        }

        return sb.ToString().Trim();
    }
}
```

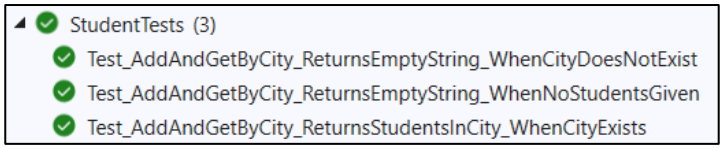
Then, look at the tests inside the **`StudentTests.cs`** class:



```
public class StudentTests
{
    private Student _student;

    [SetUp]
    0 references
    public void SetUp()
    {
        this._student = new();
    }

    [Test]
    0 references
    public void Test_AddAndGetByCity_ReturnsStudentsInCity_WhenCityExists()...

    [Test]
    0 references
    public void Test_AddAndGetByCity_ReturnsEmptyString_WhenCityDoesNotExist()...

    [Test]
    0 references
    public void Test_AddAndGetByCity_ReturnsEmptyString_WhenNoStudentsGiven()...
}
```

Notice the use of a **setup method**, so each test has a brand-new **student instance** to use.

The first test if **partially finished** so you have a **reference**, the rest are **empty,** and your task is to finish them. The tests should run when you're finished:



2. Unit Test: Song

The class **Song.cs** has **properties** for **list type**, **name**, and **time**:

```
public class Song
{
    2 references
    public string ListType { get; set; } = null!;

    2 references
    public string Name { get; set; } = null!;

    1 reference
    public string Time { get; set; } = null!;
```

It also has a **method** that takes in a **string array** representing **songs** in the form of:

"**{type} {name} {time}**"

Also, a string representing which **list** (**type**) the method should **retrieve** and **return** each song in it as a string:

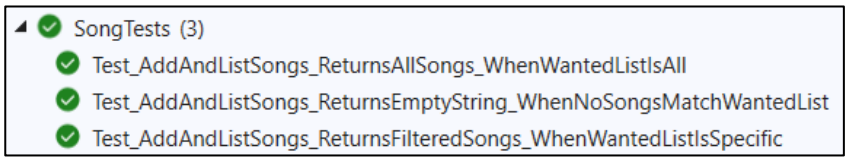
```
    public string AddAndListSongs(string[] songs, string wantedList)
    {
        List<Song> addedSongs = new();

        foreach (string currentSong in songs)
        {
            string[] data = currentSong.Split(separator: '_');

            string type = data[0];
            string name = data[1];
            string time = data[2];

            Song song = new()
            {
                ListType = type,
                Name = name,
                Time = time
            };

            addedSongs.Add(song);
        }
```

```
        List<Song> filtered = wantedList == "all"
            ? addedSongs
            : addedSongs.Where(s:Song => s.ListType == wantedList).ToList();

        StringBuilder sb = new();
        foreach (Song song in filtered)
        {
            sb.AppendLine(song.Name);
        }

        return sb.ToString().Trim();
    }
}
```

Now, look at the tests inside the **`SongTests.cs`** class:

```
public class SongTests
{
    private Song _song;

    [SetUp]
    0 references
    public void Setup()
    {
        this._song = new();
    }

    [Test]
    0 references
    public void Test_AddAndListSongs_ReturnsAllSongs_WhenWantedListIsAll()...

    [Test]
    0 references
    public void Test_AddAndListSongs_ReturnsFilteredSongs_WhenWantedListIsSpecific()...

    [Test]
    0 references
    public void Test_AddAndListSongs_ReturnsEmptyString_WhenNoSongsMatchWantedList()...
}
```

You are given a **setup method** again as well as one **partially finished** test, the rest are **empty,** and your task is to finish them. The tests should run when you're finished:



3. Unit Test: Store

The **folder Store** contains **3 classes**, in which the **main** one is **Shop.cs**.

The other 2 classes are smaller classes **representing real life objects** only holding **properties**.

Box.cs:

```
public class Box
{
    1 reference
    public Box()
    {
        this.Item = new();
    }

    2 references
    public long SerialNumber { get; set; }

    4 references
    public Item Item { get; set; }

    2 references
    public int ItemQuantity { get; set; }

    3 references
    public decimal BoxPrice { get; set; }
}
```

Item.cs:

```
public class Item
{
    2 references
    public string Name { get; set; } = null!;

    2 references
    public decimal Price { get; set; }
}
```

Shop.cs only has a **method** which takes in a **string array** representing **products** in the form of:

"**{serial_number} {name} {quantity} {price}**"

For each **product** a new **Item** is created and placed in a new **Box** then the box is **added** to a **list**. Finally, the list of boxes is returned as a **string of information**:

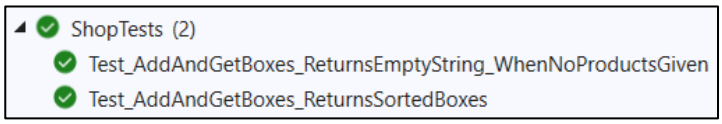
```
public class Shop
{
    2 references
    public string AddAndGetBoxes(string[] products)
    {
        List<Box> boxList = new();
        foreach (string product in products)
        {
            string[] data = product.Split();

            long serialNumber = long.Parse(data[0]);
            string name = data[1];
            int itemQty = int.Parse(data[2]);
            decimal price = decimal.Parse(data[3]);

            decimal boxPrice = price * itemQty;
            Item newItem = new()
            {
                Name = name,
                Price = price
            };
```

```
            Box newBox = new()
            {
                SerialNumber = serialNumber,
                Item = newItem,
                ItemQuantity = itemQty,
                BoxPrice = boxPrice
            };

            boxList.Add(newBox);
        }

        StringBuilder sb = new();
        foreach (Box box in boxList.OrderByDescending(box => box.BoxPrice))
        {
            sb.AppendLine(box.SerialNumber.ToString());
            sb.AppendLine($"-- {box.Item.Name} - ${box.Item.Price:f2}: {box.ItemQuantity}");
            sb.AppendLine($"-- ${box.BoxPrice:f2}");
        }

        return sb.ToString().Trim();
    }
}
```

Now, look at the tests inside the **ShopTests.cs** class:

```
public class ShopTests
{
    // TODO: write setup method

    [Test]
    0 references
    public void Test_AddAndGetBoxes_ReturnsSortedBoxes()...

    [Test]
    0 references
    public void Test_AddAndGetBoxes_ReturnsEmptyString_WhenNoProductsGiven()...
}
```

This time write the **setup method** on your own. You are given **one partial test**, the rest are **empty,** and your task is to finish them. The tests should run when you're finished:



4. Unit Test: Vehicle

The **folder Vehicle** contains **4 classes**, in which the **main** one is **`Vehicles.cs`**.

The other 3 classes **represent real life objects** only holding **properties**:

`Car.cs`:

```
public class Car
{
    3 references
    public string Brand { get; set; } = null!;

    2 references
    public string Model { get; set; } = null!;

    2 references
    public int HorsePower { get; set; }
}
```

`Catalogue.cs`:

```
public class Catalogue
{
    1 reference
    public Catalogue()
    {
        TruckList = new List<Truck>();
        CarList = new List<Car>();
    }

    3 references
    public List<Truck> TruckList { get; set; }

    3 references
    public List<Car> CarList { get; set; }
}
```

`Truck.cs`:

```
public class Truck
{
    3 references
    public string Brand { get; set; } = null!;

    2 references
    public string Model { get; set; } = null!;

    2 references
    public int Weight { get; set; }
}
```

`Catalogue.cs` only has a **method** which takes in a **string array** representing **vehicles** in the form of:

"**{type}/{brand}/{model}/{power}**"

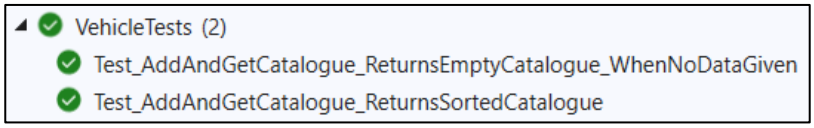
First, a new **Catalogue** is created. For each **vehicle** a new **Truck** or **Car** based on the **type** is created and added to the relevant list in the catalogue. Finally, a string is returned based on the catalogue:

```
public class Vehicles
{
    2 references | 0/2 passing
    public string AddAndGetCatalogue(string[] vehicles)
    {
        Catalogue catalogue = new();
        foreach (string vehicle in vehicles)
        {
            string[] data = vehicle.Split(separator: "/");

            string type = data[0];
            string brand = data[1];
            string model = data[2];
            int power = int.Parse(data[3]);

            if (type == "Truck")
            {
                catalogue.TruckList.Add(item: new Truck()
                {
                    Brand = brand,
                    Model = model,
                    Weight = power
                });
            }
            else
            {
                catalogue.CarList.Add(item: new Car()
                {
                    Brand = brand,
                    Model = model,
                    HorsePower = power
                });
            }
        }

        StringBuilder sb = new();
        sb.AppendLine("Cars:");
        foreach (Car car in catalogue.CarList.OrderBy(car => car.Brand))
        {
            sb.AppendLine($"{car.Brand}: {car.Model} - {car.HorsePower}hp");
        }

        sb.AppendLine("Trucks:");
        foreach (Truck truck in catalogue.TruckList.OrderBy(truck => truck.Brand))
        {
            sb.AppendLine($"{truck.Brand}: {truck.Model} - {truck.Weight}kg");
        }

        return sb.ToString().Trim();
    }
}
```

Now, look at the tests inside the **VehicleTests.cs** class:

```
public class VehicleTests
{
    // TODO: write the setup method

    [Test]
    0 references
    public void Test_AddAndGetCatalogue_ReturnsSortedCatalogue()...

    [Test]
    0 references
    public void Test_AddAndGetCatalogue_ReturnsEmptyCatalogue_WhenNoDataGiven()...
}
```

Write the **setup method** on your own. You are given **one partial test**, the rest are **empty,** and your task is to finish them. The tests should run when you're finished:



5. Unit Test: Person

The class **`Person.cs`** has **properties** for **name**, **id**, and **age**:

```
public class Person
{
    10 references | 0/2 passing
    public string Name { get; set; } = null!;

    10 references | 0/2 passing
    public string Id { get; set; } = null!;

    11 references | 0/2 passing
    public int Age { get; set; }
```

The first **method** it has, **`AddPeople()`**, takes in a **string array** representing **people** in the form of:

"**`{name} {id} {age}`**"

The method adds all people to a **list** and returns it:

```
    public List<Person> AddPeople(string[] people)
    {
        List<Person> peopleList = new();
        foreach (string data in people)
        {
            string[] split = data.Split();

            string name = split[0];
            string id = split[1];
            int age = int.Parse(split[2]);
```

```
            Person? searchPerson = peopleList.FirstOrDefault(person => person.Id == id);
            if (searchPerson is null)
            {
                peopleList.Add(item: new Person()
                {
                    Name = name,
                    Id = id,
                    Age = age
                });
            }
            else
            {
                searchPerson.Age = age;
                searchPerson.Name = name;
            }
        }

        return peopleList;
    }
```

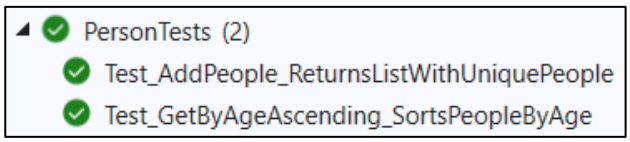
The next method, **`GetByAgeAscending()`**, takes in a **list of people**, **sorts the list by age**, and returns a **string** with **information**:

```
    public string GetByAgeAscending(List<Person> people)
    {
        StringBuilder sb = new();
        foreach (Person person in people.OrderBy(person => person.Age))
        {
            sb.AppendLine($"{person.Name} with ID: {person.Id} is {person.Age} years old.");
        }

        return sb.ToString().Trim();
    }
}
```

Now, look at the tests inside the **`PersonTests.cs`** class:

```
public class PersonTests
{
    // TODO: write the setup method

    [Test]
    0 references
    public void Test_AddPeople_ReturnsListWithUniquePeople()...

    [Test]
    0 references
    public void Test_GetByAgeAscending_SortsPeopleByAge()...
}
```

Write the **setup method** on your own. You are given **one partial test**, the rest are **empty,** and your task is to finish them. The tests should run when you're finished:



6. Unit Test: Article

The class **`Article.cs`** has **properties** for **title**, **content**, **author**, and **article list**:

```
public class Article
{
    7 references | 0/2 passing
    public string Title { get; set; } = null!;

    7 references | 0/2 passing
    public string Content { get; set; } = null!;

    7 references | 0/2 passing
    public string Author { get; set; } = null!;

    9 references | 0/2 passing
    public List<Article> ArticleList { get; set; } = new();
```

The first **method** it has, **`AddArticles()`**, takes in a **string array** representing **articles** in the form of:

"**`{title} {content} {author}`**"

The method adds all articles to a **list** and returns it:

```
    public Article AddArticles(string[] articles)
    {
        Article article = new();
        foreach (string data in articles)
        {
            string[] split = data.Split();

            string title = split[0];
            string content = split[1];
            string author = split[2];

            article.ArticleList.Add(item: new Article()
            {
                Title = title,
                Content = content,
                Author = author
            });
        }

        return article;
    }
```

The next method, **`GetArticleList()`**, takes in an **instance of an article**, and a **print criteria**. Based on the criteria it **orders the list,** and returns a **string** with **information**:

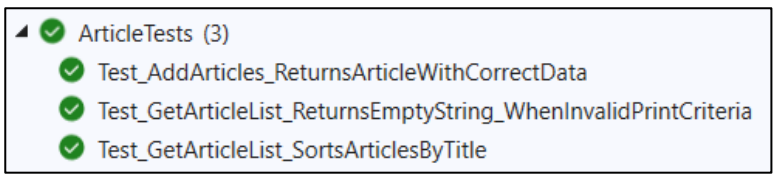
```
    public string GetArticleList(Article article, string printCriteria)
    {
        IOrderedEnumerable<Article>? list = null;
        switch (printCriteria)
        {
            case "title":
                list = article.ArticleList.OrderBy(a:Article => a.Title);
                break;
            case "content":
                list = article.ArticleList.OrderBy(a:Article => a.Content);
                break;
            case "author":
                list = article.ArticleList.OrderBy(a:Article => a.Author);
                break;
            default:
                return string.Empty;
        }
```

```
        StringBuilder sb = new();
        foreach (Article item in list)
        {
            sb.AppendLine($"{item.Title} - {item.Content}: {item.Author}");
        }

        return sb.ToString().Trim();
    }
}
```

Now, look at the tests inside the **`ArticleTests.cs`** class:

```
public class ArticleTests
{
    // TODO: write the setup method

    [Test]
    0 references
    public void Test_AddArticles_ReturnsArticleWithCorrectData()...

    [Test]
    0 references
    public void Test_GetArticleList_SortsArticlesByTitle()...

    [Test]
    0 references
    public void Test_GetArticleList_ReturnsEmptyString_WhenInvalidPrintCriteria()...
}
```

Write the **setup method** on your own. You are given **one partial test**, the rest are **empty,** and your task is to finish them. The tests should run when you're finished:



7. Unit Test: Planet

The class **`Planet.cs`** has **properties** for **name**, **diameter**, **sun distance**, and **moon number**. It also has a **private field** for **gravitation** and a **constructor**:

```
public class Planet
{
    private const double GravitationalConstant = 6.67430e-11;

    2 references | 0/2 passing
    public Planet(string name, double diameter, double distanceFromSun, int numberOfMoons)
    {
        this.Name = name;
        this.Diameter = diameter;
        this.DistanceFromSun = distanceFromSun;
        this.NumberOfMoons = numberOfMoons;
    }

    2 references
    public string Name { get; set; }

    4 references | 0/1 passing
    public double Diameter { get; set; }

    2 references
    public double DistanceFromSun { get; set; }

    2 references
    public int NumberOfMoons { get; set; }
```

The first **method** it has, **`CalculateGravity()`**, takes in a **number** representing **mass.** The method calculates the **planets gravity** with a calculation:

```
    public double CalculateGravity(double mass)
    {
        double radius = this.Diameter / 2.0;
        return mass * GravitationalConstant / (radius * radius);
    }
```

The next method, **`GetPlanetInfo()`**, returns a **string with information about the planet**:

```
    public string GetPlanetInfo()
    {
        StringBuilder sb = new();

        sb.AppendLine($"Planet: {Name}");
        sb.AppendLine($"Diameter: {Diameter} km");
        sb.AppendLine($"Distance from the Sun: {DistanceFromSun} km");
        sb.AppendLine($"Number of Moons: {NumberOfMoons}");

        return sb.ToString().Trim();
    }
}
```

Now, look at the tests inside the **`PlanetTests.cs`** class:

```
public class PlanetTests
{
    [Test]
    0 references
    public void Test_CalculateGravity_ReturnsCorrectCalculation()...

    [Test]
    0 references
    public void Test_GetPlanetInfo_ReturnsCorrectInfo()...
}
```

You are given **one partial test**, the rest are **empty,** and your task is to finish them. The tests should run when you're finished:

At the end make sure all tests pass: