Lehrstuhl für Computergestützte Modellierung und Simulation
TUM School of Engineering and Design
Technical University of Munich

Professional Software Engineering

Lecture 4: Exercises

Revision – Interfaces

During last week's exercise session, you created an interface that was implemented by two vector classes. The interface contained signatures for calculating the L_2 norm of a vector, as well as printing the vector to the console.

Below is one of the ways of programming a method to calculate the dot product of two vectors to this exercise.

```
public interface IVector {
    //...last week's methods
    IVector CalculateDotProduct(IVector otherVector);
}

class Vector2D {
    //...constructors, properties and last week's methods...
    Vector2D CalculateDotProduct(Vector2D otherVector) {
        return this.X * otherVector.X + this.Y * otherVector.Y;
    }
}

class Vector3D {
    //...constructors, properties and last week's methods...
    Vector3D CalculateDotProduct(Vector3D otherVector) {
        return this.X * otherVector.X + this.Y * otherVector.Y + this.Z * otherVector.Z;
    }
}
```

Why would this implementation fail?

Hint: Look up upcasting and downcasting.

Exercise 1 Generic Vector Class

In this exercise, we will create a generic vector class that implements custom interfaces. Since this exercise will be quite lengthy, we have separated the tasks into the mini-exercises (a - i). You do not have to go through the mini-exercises in order, with the exception of exercises (a) and (b).

(a) Setup

Start by creating a new repository for this week and cloning it on your PC. In your repository's directory, create a main project and a testing (NUnit) project. Keep in mind that the former should contain a class with a main method and an empty vector class. Both classes should be included inside a namespace (e.g. *Lecture4*) and each should be included in their own, separate file. Starting from exercise (c), create a new branch for each mini-exercise.

Make sure the projects are set up correctly by outputting a string to the console from your main method (e.g. "test" or "Hello World").

Note: For this exercise, make sure you are using .Net 6.0 or higher, as some of the functions used in this exercise, are not available in the older .Net versions.

Note: One of the interfaces used in this exercise (*INumber*) is not released yet and only available in the experimental API. Therefore before moving onto the next exercise make sure to follow the steps below.

1. Include *System.Runtime.Experimental* as a dependency in Visual Studio
2. Add the following line to the project XML file.

<EnablePreviewFeatures>true</EnablePreviewFeatures>

(b) Optional Group Task – Test Driven Development

Set up groups of 2, where the first member's task is to create the Vector class described in Exercise 1, while the other member's task is to create a set of unit-tests to test the functionality of the Vector class. Keep on improving your Vector class until all the unit-tests pass.

Note: As mentioned last week, start small and work your way up. Focus on edge cases and try to write tests that can break your group member's Vector class.

Hint: If you set your repository to *private*, you can grant your group member access as follows:

In `gitlab.lrz.de`, go to your repository's page, then hover over project information (top left) and select members. In the members page, click on invite members and add your group member's email or username to invite them. Note: Make sure you change the role to *owner* from the default *guest*.

(c) Interfaces

While our Vector class is a generic class *i.e.* the user can initialize it with any type (int, double, complex etc.), we need to restrict the types to a set of types that would not break our class. Calculating the dot product of two vectors of type string (*Vector<string>*) for instance, will likely break our code and even if it doesn't, the output will be meaningless.

For this exercise, we will limit our class to the primitive, numeric types of C#, integers and floating-point numbers of different sizes (uint8, int32, float, double, decimal etc.). We will use the interface *INumber*[1] which is provided under the namespace *System.Runtime.Experimental*.

Furthermore, our Vector class itself should implement some interfaces to describe the operations that can be used on the class. For example, each vector should contain a method to add, or multiply it by another vector, therefore, they should implement *IAddable* and *IMultipliable*. Try to come up with additional interfaces that the Vector class should implement; increasing the rigidity of this implementation will decrease the likelihood of errors.

An example declaration of the vector class, along with some of the interfaces, is presented below. Keep in mind that the class and interfaces should be in separate files.

```
using System;
using System.Runtime.Experimental;

namespace Lecture4{
    public interface IAddable<T> where T : INumber<T> {
        Vector<T> Add (Vector<T> otherVector);
    }

    public interface IMultipliable<T> where T : INumber<T> {
        Vector<T> Multiply (Vector<T> otherVector);
    }
```

[1] Technically, INumber also includes characters (e.g. 'c', 'a' etc.), and hence the compiler will not complain if you initialize a character vector (*Vector< char >*), for example.

```
    //...other vector interfaces

    public interface IVector<T> : IAddable<T> , IMultipliable<T> , ...other vector
        interfaces {} // combining (technically inheriting) all interfaces into one
        interface -- makes no difference but helps keep the program more readable

    public class Vector<T> : IVector<T> where T : INumber<T> {
        //... class properties, constructors and methods
    }
}
```

Note: We will use the the signatures of both *IAddable* and *IMultipliable* in exercise (i) to program *operator overloads*.

(d) Properties

Now that we have a general setup for our Vector class, implement the following properties.

1. A container to store all the vector entries. While choosing the "right" container for your class, keep in mind that the size will constantly change; the size will not stay constant as is the case in an array for example.
2. An integer to store the current size of the vector.

Do not initialize the properties, as this will be done by the constructors.

Hint: You need a container that can dynamically change its size. Look through the lecture notes for some options. Choose the simplest one you can find (that fits your requirements), since the simpler the container, the less memory it demands.

Bonus: Use an array, with a fixed size, instead. How would you handle a change in the vector's size? Implement your solution as a private method.

(e) Constructors

Implement the following constructors in your class:

1. A default constructor (no arguments), where the container should be initialized with 1 entry with a value of 0.
2. A constructor that takes one argument *initValue*, where the container should be initialized with 1 entry with a value equal to *initValue*.
3. A constructor that takes two arguments: *initValue* and *vectorSize*. Initialize the container with *vectorSize* entries, each with a value equal to *initValue*
4. A constructor that takes an array *initArray* as an argument. Initialize the container with the values of *initArray*

In all constructors, the size property should be set to the size of the container.

(f) Data Access Methods

Implement the following methods in your Vector class.

1. GetValue: Returns the value stored at an index i. Make sure to check if the index is out-of-bounds.

   public T GetValue(int i);

2. Exists: Returns a boolean indicating whether a certain value is stored in the container.

   public bool Exists(T v);

(g) Container Modifying Methods

Now, we will add useful methods that can be used to modify the content of the container property. Implement the following methods in your Vector class.

1. PushBack: This method should take a value as an argument and place it as the last item in the container.

   public void PushBack(T value);

2. Delete: This method should take an index i as an argument and delete the value stored at i.

   public void Delete(int i);

3. SetValue: This method should take two arguments: an index i and a value v. Update the value stored at i with v.

   public void SetValue(int i , T v);

4. Four overloads of the method Reset: Resets all properties to the default state. The logic should be similar to the logic of the constructors.

Hint: You can also move the logic of the constructors to private methods, and then call the private methods in the constructors and the overloads of the method Reset.

Note: Make sure you update your size property once the size of the container changes. You can also add a private method that checks the size of the container, and call the method inside your size property's getter, as shown below. This way, you do not need to keep updating your size property every time the size changes.

```
//...class declarations and other properties
private int size; //field-where the data is stored
public int Size { //modified property
    get {
        return GetSize(); // whenever the user asks for the size the method is called --
            will always be up-to-date
    }
    set {
        this.size = value;
    }
}
```

(h) Interface Methods

Implement the following methods in your Vector class:

1. Add: Add the entries placed at the same index together and return a third vector.

   public Vector< T > Add(Vector< T > otherVector);

2. Multiply: Call the method CalculateDotProduct (eq. (1)) and return the result

   public T Multiply(Vector< T > otherVector);

3. IsEqual: Check if for every index i, the i^{th} entry of the first and second vector are equal. If they all are, return *true*.

   public bool IsEqual(Vector< T > otherVector);

4. Assign: Update your class' properties with the properties of *otherVector* [2]

   public void Assign(Vector< T > otherVector);

Hint: The logic of the method assign is very similar to the reset methods.

[2] This method is a custom assignment method (similar to the assignment operator '=').

(i) Overloaded Operators

Operator overloading is the process of modifying the outcome an operator, such as "+", "−" or "∗".

For example, suppose we created a custom integer class *CustomInt* and overloaded the addition operator (+) to return 0 instead of adding both integers as follows.

```
class CustomInt {
    public int Number {get; set;}

    CustomInt(int number) {
        this.Number = number;
    }

    public static int operator + (CustomInt customInt , CustomInt otherCustomInt) {
        return 0;
    }
}
```

Adding two integers and two CustomInts would result in the following output.

```
class MainClass {
    public static void Main(string[] argArray) {
        CustomInt customInt1 = CustomInt(4);
        CustomInt customInt2 = CustomInt(5);

        int int1 = 4;
        int int2 = 5;

        var customAddition = customInt1 + customInt2;
        var addition = int1 + int2;

        Console.WriteLine(addition); //outputs 5;
        Console.WriteLine(customAddition); //outputs 0;
    }
}
```

We can use this concept to modify the operators in our Vector class. For instance, an addition of two vectors should return a third vector, where the items at the same index i should be added and placed in index i of the third vector. Since we added similar methods in exercise (h), we just have to call them in our operator overloads.

Implement the following operator overloads in your Vector class:

1. The addition operator "+": Call the method *Add* and return the result

   public static Vector< T > operator + (Vector< T > vector1 , Vector< T > vector2);

2. The equality operator "==": Call the method *IsEqual* and return the result

   public static bool operator == (Vector< T > vector1 , Vector< T > vector2);

3. The inequality operator "! =": Call the equality operator and return the opposite of the result.

   public static bool operator != (Vector< T > vector1 , Vector< T > vector2)

4. The multiplication operator "∗": Call the method *Multiply* and return the result.

   *public static T operator * (Vector< T > vector1 , Vector< T > vector2)*

Once you are done, overload the square bracket operator [], where you should call the method GetValue and return the result. Keep in mind that this operator is overloaded in a different way in C#, since it is treated as a property and not a method, as shown in the code-snippet below.

```
 public T this[int i]
{
   get { return GetValue(i); }
   set { SetValue(i , value); }
}
```

(j) Misc. Methods

Finally, implement the following methods in your class.

1. CalculateDotProduct: Returns the dot product of two vectors (u and v in eq. (1)).

$$\textit{public T CalculateDotProduct (Vector otherVector);}$$

$$v.u = v_i u_i = v_1 u_1 + v_2 u_2 + v_3 u_3 \ldots \tag{1}$$

2. GetNorm2: Returns the square of the L_2 norm of a vector (v in eq. (2)).

$$\textit{public T GetNorm2 ();}$$

$$||v||_2^2 = v_i v_i = v_1 v_1 + v_2 v_2 + v_3 v_3 \ldots \tag{2}$$

3. PrintItem: Prints the item stored at an index i to the console.

$$\textit{public void PrintItem (int i);}$$

4. Print: Prints the vector to the console.

$$\textit{public void Print ();}$$

Hint: A Euclidean norm can be thought of as the square root of the dot product of a vector with itself. Use this to minimize duplicate code in your class.