A picture containing text

Description automatically generated

|  |  |
| --- | --- |
| Module code and name: | Object Oriented Programming |
| Assessment: | CW 1 |
| Student ID: | 00012938 |
| Submission deadline: | 07.12.2023 |
| Wordcount | 926 |
| Results date and type of feedback |  |
| Result (Pass or Fail) |  |
| **Feedback** | |
|  | |

**Оглавление**

[**Task1 2**](#_Toc152296495)

[**Task2 3**](#_Toc152296496)

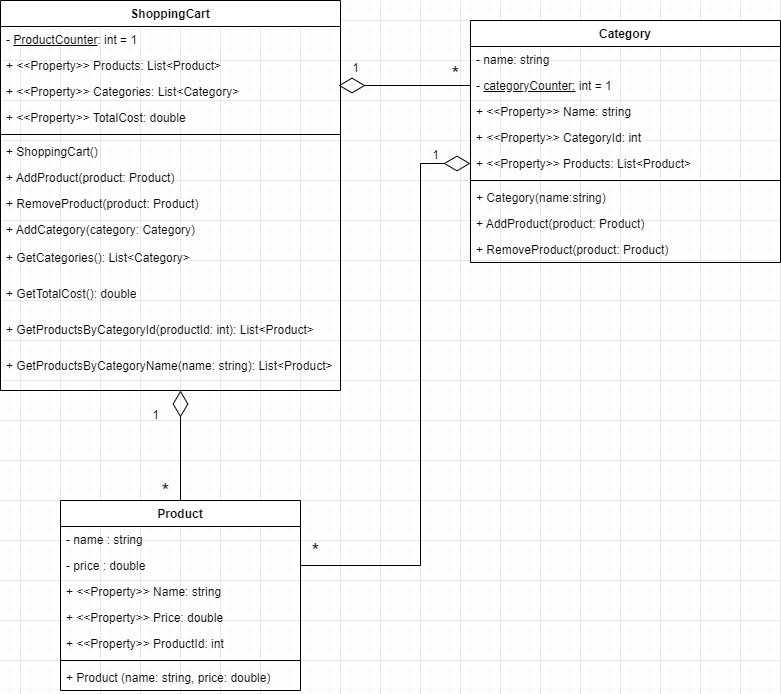
[**Task3 5**](#_Toc152296497)

[**Task4 6**](#_Toc152296498)

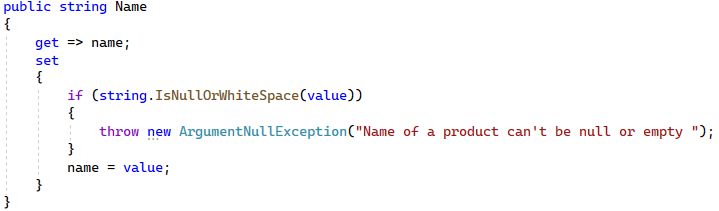
[**Main Program 8**](#_Toc152296499)

# Task1

The main requirement was creating ShoppingCart, Product and Category classes with the stated properties/methods. The following UML Diagram shows the details:

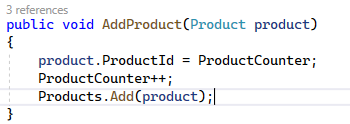


Let’s start with Product. Name and Price properties were required and because we wanted a validation, private fields were added accordingly.



However, from the given sample of the program, it was clear that adding another property that is “ProductId” is required in order to interact with a method that removes product by id. However, the sample tried to represent that the ProductId is generated only when it is added to the cart. Basically, it makes sense because, in terms of UX, it is better to have an ordered sequence to choose when a User sees his/her cart (1,2, 3.. instead of 345, 123, 12, 84 instead). ProductId will be required for that operation.

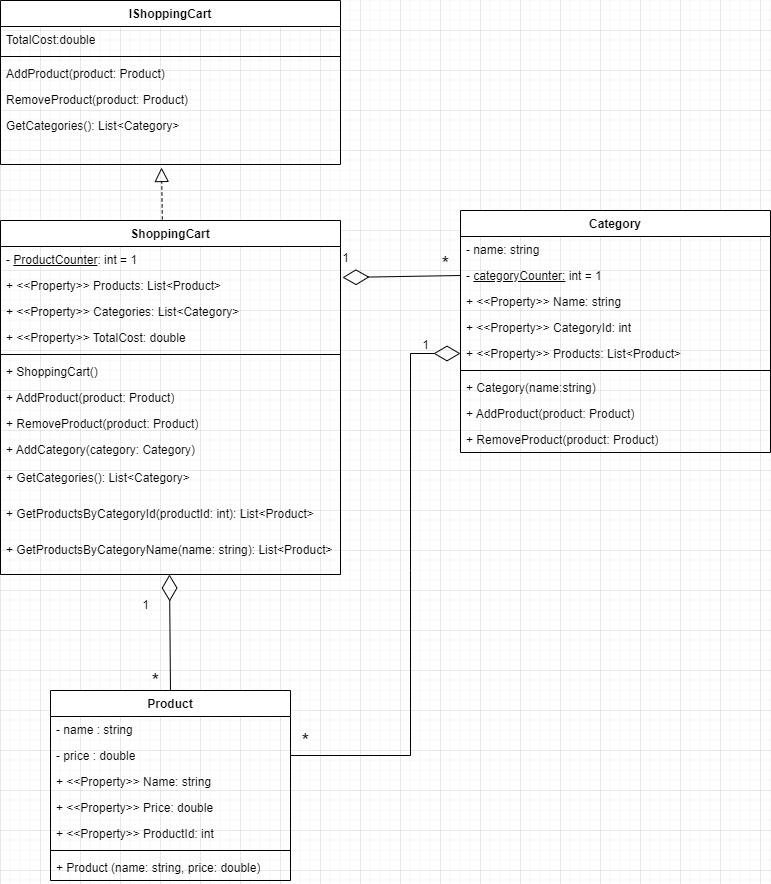
Turning to CategoryCart, we can observe a private field that is needed to generate the ProductId when a Product is added to the Cart:



Other methods/properties were added according to the requirements of Task 1. Finally, all properties/methods for Category Class were created according to the Task description, where CategoryId is automatically generated using CategoryCounter private field.

# Task2

Overall, the second task asks for existing class modifications for using the best practicing. Basically, few changes were applied because the created code has been already in the most optimal and well-structured way:



The first change was deleting the GetTotalCost method because we have a property TotalCost that is enough for getting the total cost. In task one TotalCost just used GetTotalCost() inside get{} statement. The second change was the addition of Interface IShoppingCart, where ShoppingCart implements it. The interface here defines the contract for a shopping cart, giving flexibility and function integrity. Also, it allows the creation of different shopping cart types and we will not be required to change the existing code (we are following the SOLID O principle)

Let’s go one by one through various aspects:

1. Well-defined Classes:

Each class has its unique responsibilities. Product class represents an individual product, while Category defines a group of products and the ShoppingCart class encapsulates the whole logic of product/Category handlings that includes adding, removing and calculating.

1. Well-formatted and easy-to-read code:

All OOP pillars are used for proper code formatting. The code is effectively indented, and meaningful variable/method names are used. Because of strict and classic formatting, the code is easy to read and understand for other developers.

1. Modular and easy-to-test code:

Because of separating different logic in various classes, we are achieving modularity that means testing a single model will not cause difficulty. Moreover, such kind of approach simplifies error detection.

1. Appropriate data structure and algorithms:

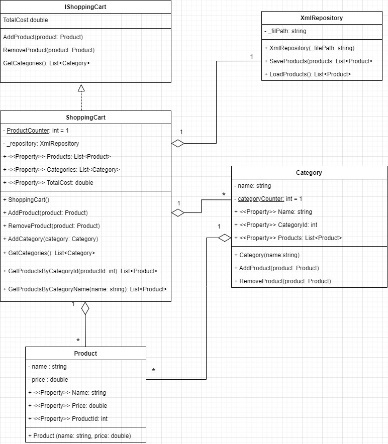
Well-stated data structure and algorithms are the keys to fast and effective application. Validation in the face of private fields is a part of it. The code uses List<Object> to store product/category objects, which is the most effective data structure for storing a collection of objects and having maximum productivity when working with filtering and calculation (which can be achieved by using LINQ).

1. Efficient use of resources

In all cases, where built-in methods for item manipulation are available is used to reach max speed at less resource use.

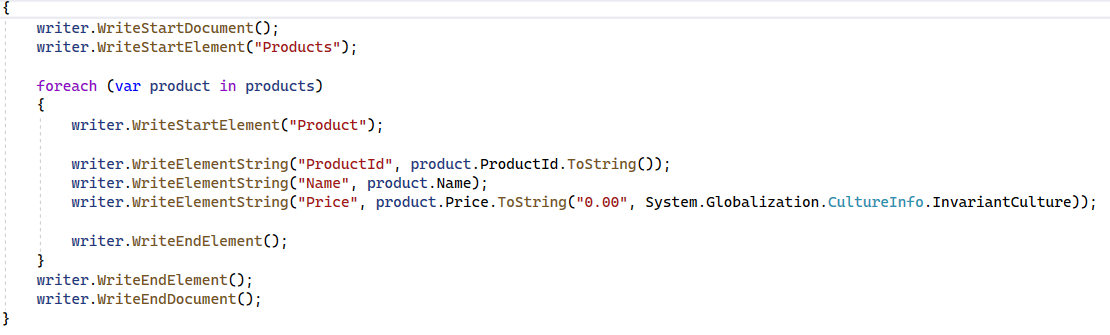
# Task3

The task was to integrate an XML Repository for our Shopping Cart. Purpose – saving product not temporarily, but permanently using .xml file:

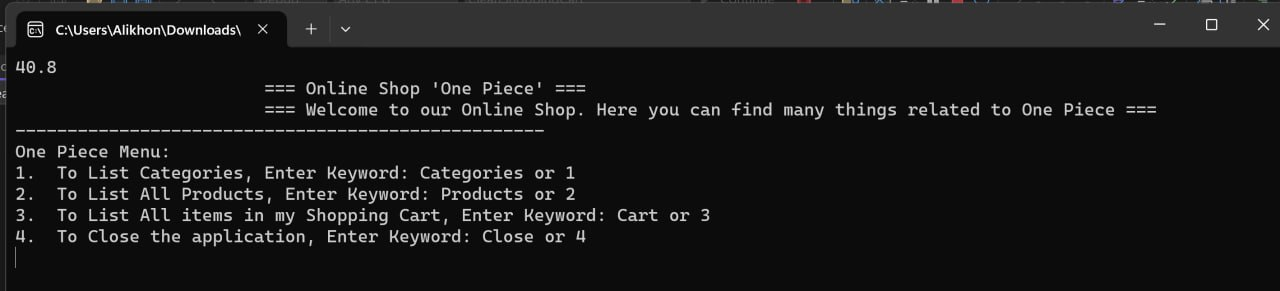


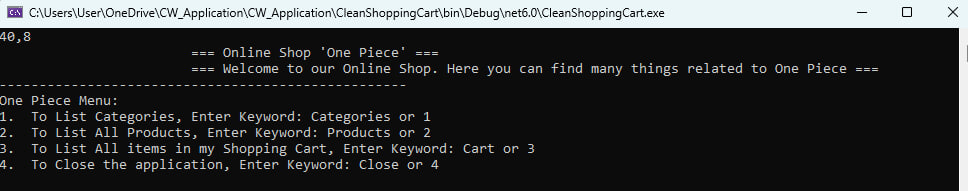
Regarding the Repository itself, XmlWriter and XmlReader are used as it was required. From personal perspective, using exactly this method (for instance instead of XmlSerializer and XmlDeserializer that would automatically identify content and we could write less code) is because of the efficiency and maintainability. Here we are responsible for specifying structure and such kind of approach gives the max possible speed, especially when we are talking about small files.

One thing to discuss in xml:

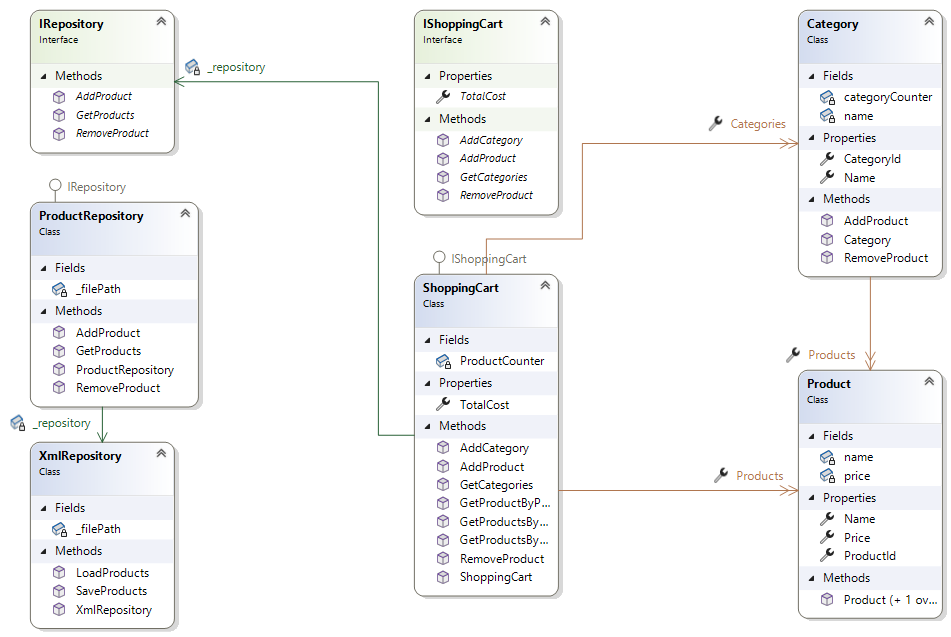


Here we are explicitly turning the price that double to specific format. The main reason – different cultures based on computer configuration:



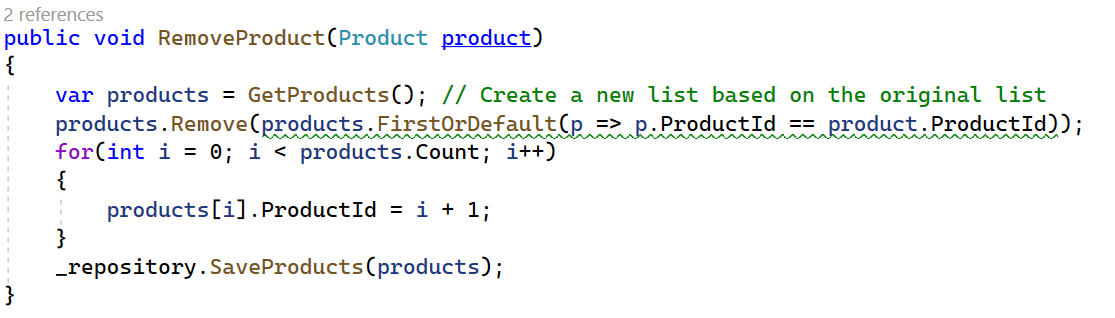


# Task4



Can be found inside the CleanShoppingCart project (ClassDiagram1.cd)

It was required to create another project that implements a Repository Pattern and is a proper extension of XmlRepository. This pattern is relatively a simple one, where we creating a contract in the face of the IRepository interface, which will have an aggregation connection with ShoppingCart. ProductRepository is a concrete implementation of IRepository that is assigned using Dynamic Polymorphism. All methods were created based on requirements. The only thing to mention: the connection with XmlRepository. GetProducts() is the same thing as LoadProducts(). Also, AddProduct() and RemoveProduct() can not work without getting and loading all products. This is a drawback of XML. We need to overwrite the products. That means we have two options: copy and paste the code from XmlRepository and in this case XmlRepository because legacy and will not be used anywhere or declare aggregation connection and use the methods from XmlRepository. Because we want to have well-defined and clean code, we are using the second option. The last thing to mention is RemoveProduct():



Why We are not using just products.Remove(product)? Again because of the validation of cultureinfo of different devices. For devices, where double is represented in “,” notation the product may be not removed because there will not be a product match and we will get the same product list (XML assumes double in “.” notation, e.g. “15.2”, but reading “15,2” will cause an error.

# Main Program



To follow the SOLID S (single responsibility) principle, methods that use the mentioned classes were created, where each class has a single responsibility. Main() wraps all the methods and we get a working application.