Software Design

Abstract

The Online Shopping Application is designed based on the commercial mode of business-to-customer (B2C). Its commercial target is the home-based customers group who are keen on a series of door-to-door shopping services experience, as well as customers who need to save time shopping in physical stores. In addition, the App can also help the enterprise save physical store operation costs, thereby reducing inventory pressure and increasing working capital. The application functions cover the basic daily online purchasing behaviors, needs, and habits of end users, such as a complete procedure: browsing category, product list → adding products in the shopping cart → selecting product items to the order → selecting payment method → generating order → products are released from the stock/product items are deducted from the stock. The application also considers the customer service model, the customer can return their products. In addition, a powerful database is configured to store the data information of category, product, stock, users, payment type, shopping cart, generated orders, and historical users' orders. Alongside an XUnit test function to monitor and assess the daily application performance.

Introduction

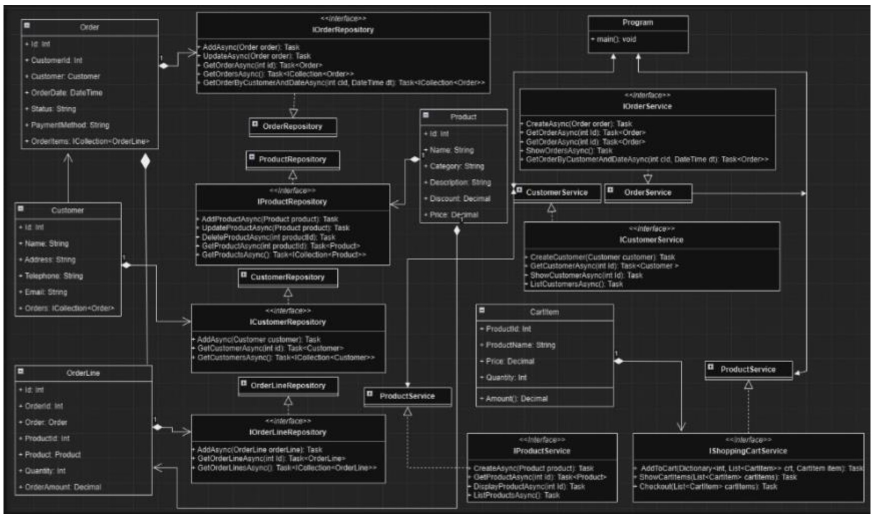
The Online Shopping Application is an object-oriented design and developed by using C# .Net programming language in Visual Studio. Through this extensive process of development, the developers have created a simple console application that simulates an online shopping application that prioritizes robust software design, clear architecture, and unit testing. Unit tests, interfaces, and classes are all part of the project, which aims to produce a workable and reusable application. The software developers utilized various modern technical skills on the online shopping App, which covered Unified Modeling Language (UML) design diagram, modern features in C# .Net, Solid Principles, three-layered architecture, MVC design patterns, XUnit tests, refactoring, and asynchronous programming with 'system.threading.task' support by using the .Net framework to handle this in the background. To improve the evaluation of the application performance, also to optimize the end user experience, the developers also utilized various tool packages in Visual Studio, which include 'coverlet.collect', 'Microsoft.EntityFrameworkCore/.IMemory/.Sqlte/.Tools', 'Microsoft.Extensions.Configuration/.FileExtensions/. Json', 'Microsoft.NET.TEST.Sdk', 'Moq', 'xunit'. 'Xunit.runner.visualstudio'. For data storage, the developers take advantage of DB Browser for SQLite, which can create, insert, modify the tables by SQL, and can export the data to different file types like SQL files, CSV files, and JSON files.

Techniques and Tools of Collaboration

The initial plan was to make use of the Live Share function in Visual Studio, which is an effective tool for group coding, pair programming, and more effectively requesting help. By allowing several developers to share code and project information in real-time, independent of their physical locations, this solution improves the development experience while the developers work remotely. In addition, developers can collaborate easily in a Live Share environment, which speeds up issue resolution, knowledge exchange, and cooperative problem-solving. However, due to the different computer operating systems used by developers, Macbook/IOS and Windows10, applications are developed in a traditional collaborative approach of zip file transformation online. This is also to avoid bugs that occur because of the version conflict so that the application can be completed in a limited development period.

UML Class Diagram

UML stands for Unified Modeling Language [6], it is a method to visually represent the architecture, design, and implementation of complex software systems. A UML class diagram [4] is UML structure diagram which shows structure of the designed system at the level of classes and interfaces, displays their features, constraints and relationships, associations, generalizations, dependencies and so on. The below diagram demonstrates the relationships among the classes in the App.


Order
+ Id: Int
+ CustomerId: Int
+ Customer: Customer
+ OrderDate: DateTime
+ Status: String
+ PaymentMethod: String
+ OrderItems: ICollection<OrderLine>
<<interface>>
IOrderRepository
+ AddAsync(Order order): Task
+ UpdateAsync(Order order): Task
+ GetOrderAsync(int id): Task<Order>
+ GetOrdersAsync(): Task<ICollection<Order>>
+ GetOrderByCustomerAndDateAsync(int cId, DateTime dt): Task<ICollection<Order>>
Program
+ main(): void
OrderRepository
ProductRepository
Product
+ Id: Int
+ Name: String
+ Category: String
+ Description: String
+ Discount: Decimal
+ Price: Decimal
<<interface>>
IOrderService
+ CreateAsync(Order order): Task
+ GetOrderAsync(int Id): Task<Order>
+ GetOrderAsync(int Id): Task<Order>
+ ShowOrdersAsync(): Task
+ GetOrderByCustomerAndDateAsync(int cId, DateTime dt): Task<Order>>
<<interface>>
IProductRepository
+ AddProductAsync(Product product): Task
+ UpdateProductAsync(Product product): Task
+ DeleteProductAsync(int productId): Task
+ GetProductAsync(int productId): Task<Product>
+ GetProductsAsync(): Task<ICollection<Product>>
CustomerService
OrderService
Customer
+ Id: Int
+ Name: String
+ Address: String
+ Telephone: String
+ Email: String
+ Orders: ICollection<Order>
<<interface>>
ICustomerService
+ CreateCustomer(Customer customer): Task
+ GetCustomerAsync(int Id): Task<Customer >
+ ShowCustomerAsync(int Id): Task
+ ListCustomersAsync(): Task
CustomerRepository
<<interface>>
ICustomerRepository
+ AddAsync(Customer customer): Task
+ GetCustomerAsync(int id): Task<Customer>
+ GetCustomersAsync(): Task<ICollection<Customer>>
CartItem
+ ProductId: Int
+ ProductName: String
+ Price: Decimal
+ Quantity: Int
+ Amount(): Decimal
OrderLine
+ Id: Int
+ OrderId: Int
+ Order: Order
+ ProductId: Int
+ Product: Product
+ Quantity: Int
+ OrderAmount: Decimal
OrderLineRepository
ProductService
ProductService
<<interface>>
IOrderLineRepository
+ AddAsync(OrderLine orderLine): Task
+ GetOrderLineAsync(int Id): Task<OrderLine>
+ GetOrderLinesAsync(): Task<ICollection<OrderLine>>
<<interface>>
IProductService
+ CreateAsync(Product product): Task
+ GetProductAsync(int id): Task<Product>
+ DisplayProductAsync(int Id): Task
+ ListProductsAsync(): Task
<<interface>>
IShoppingCartService
+ AddToCart(Dictionary<int, List<CartItem>> crt, CartItem item): Task
+ ShowCartItems(List<CartItem> cartItems): Task
+ Checkout(List<CartItem> cartItems): Task
1
1
1
1
1
1

Classes and Interfaces

- 'Program.cs' is the entry point of the application with a 'main' method, which is typical for programming language in C#.
- 'Order.cs', 'Customer.cs', 'Product.cs', 'OrderLine.cs', and 'CartItem.cs' are classes representing entities within the App. Each has properties that define its attributes, such as 'Id', 'Name', 'Price' and so on.
- 'IOrderRepository.cs', 'IProductRepository.cs', 'ICustomerRepository.cs', and 'IOrderLineRepository.cs' are interfaces defining contracts for repository operations. These are likely used for data access operations related to their respective entities.
- 'IOrderService.cs', 'ICustomerService.cs', 'IProductService.cs', and 'IshoppingCartService.cs' are interfaces defining contracts for service operations. These services will contain business logic to process orders, manage customers, products, and shopping cart items.

Relationships and Dependencies

- Arrows with hollow diamonds indicate a composition relationship, meaning the class owns the object and it is created and destroyed with the class. For instance, an 'Order' has a collection of 'OrderLine.cs' objects which are part of the 'Order'.
- Arrows with simple lines indicate dependencies or associations. For instance, 'OrderService.cs' depends on 'IOrderRepository.cs', suggesting that 'OrderService.cs' uses the 'IOrderRepository.cs' to interact with 'Order' data.
- Dotted arrows with hollow triangles represent interface implementations. For instance, 'CustomerService.cs' implements 'ICustomerService.cs'.

Methods

- Methods in repositories ('AddAsync', 'UpdateAsync', 'DeleteAsync', etc.) suggest that this application is designed to work asynchronously. This is common in modern applications that deal with Input/Output-bound operations like database access.
- Methods in services ('CreateAsync', 'ShowCartItemsAsync', etc.) define the high-level operations that can be performed by the application.

Specific Class Details

- 'Order.cs' represents an order placed by a customer with properties like 'Id', 'CustomerId', 'OrderDate', 'Status', and a collection of 'OrderItems'.
- 'Customer.cs' displays a customer with personal details and a collection of 'Orders'.
- 'Product.cs' shows a product available for purchase.
- 'OrderLine.cs' illustrates a line item in an order, linking a product to an order and specifying the quantity and amount.
- 'CartItem.cs' demonstrates an real-time item in a shopping cart.

Services and Repositories

- Services encapsulate business logic and use repositories to interact with the database.
- Repositories handle the data persistence, allowing services to remain agnostic of the data access strategy.

This UML class diagram provides a high-level architecture overview of the application, emphasizing how different parts of the system interact with each other through interfaces and how data flows between objects. The use of interfaces suggests a design that favors dependency injection, which can be used to create flexible and testable code.

Solid Principles

The SOLID principles[2], a collection of five design principles that support maintainability, flexibility, and scalability in object-oriented software development, were followed in the creation of the shopping application.

Single Responsibility Principle (SRP): Each class in the project had a single responsibility, focused on a specific component of the app's functionality. For instance, the CustomerService class oversaw customer-related capabilities, whereas the ProductService class handled actions associated with products.

The Open/Closed Principle (OCP) states that the application classes are open to extension but closed for modification. For example, instead of changing the basic implementation of the classes, they were expanded to include additional features or functions. This was clear from the manner new features or methods were introduced without changing the existing code. By using interfaces like ‘ICustomerService.cs’, it enables the implementation to be extended without altering the interface itself. For instance, if new functionality needs to be added to ‘CustomerService.cs’, it can be done by adding new methods to the interface and implementing them in ‘CustomerService.cs’ without modifying existing code.

The Liskov Substitution Principle (LSP) is upheld through interface implementation rather than classic inheritance. For instance, any class that implements the (ICustomerService) or (IProductService) interface can be substituted seamlessly in the application. This allows us to introduce new service implementations in the future that can interact with the existing system without requiring changes to the system's behavior or code that depends on these services.

The principle of Interface Segregation (ISP) states that the interfaces were created with the requirements of the classes using them in mind. For example, the ICustomerRepository interface concentrated on customer-related activities, whereas the IProductRepository interface had methods for accessing product data. Classes were unable to implement pointless methods as a result.

The project employed the Dependency Inversion Principle (DIP) by depending more on abstractions (interfaces) than on actual implementations. Dependency injection demonstrated this since high-level modules (like CustomerService) relied on abstractions (like ICustomerRepository) instead of specific implementations.

Three-Layered Architecture

The App is designed in a three-layered architecture user interface layer, business access layer, and data access layer.[5, 15] The three-layered procedure is demonstrated as below figure:

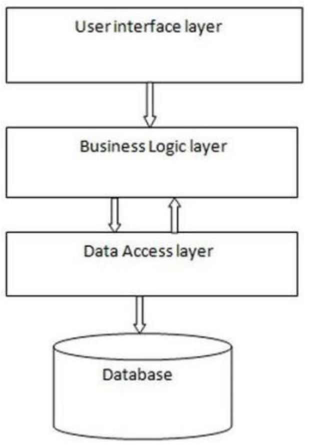
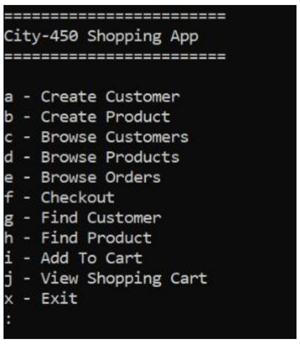


Figure source: [How to Implement 3 Layered Architecture Concepts in ASP.Net (c-sharpcorner.com)]

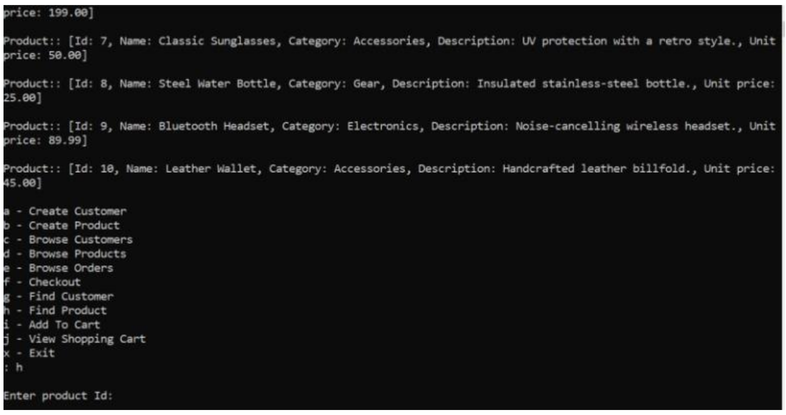
User Interface Layer is something that every user can view on the computer and mobile screen. When the App is executed, the end user will see the below menu and make their options. The menu enables the end user to input their options. In the App, the CustomerService.cs, ICustomerService.cs, IOrderLineService.cs, IOrderService.cs, IProductService.cs, IshoppingCartService.cs, OrderLineService.cs, OrderService.cs, ProductService.cs, and ShoppingCartService.cs files in the Service folder can achieve the goal by receiving the user input from the main Program.cs on this layer.

User menu



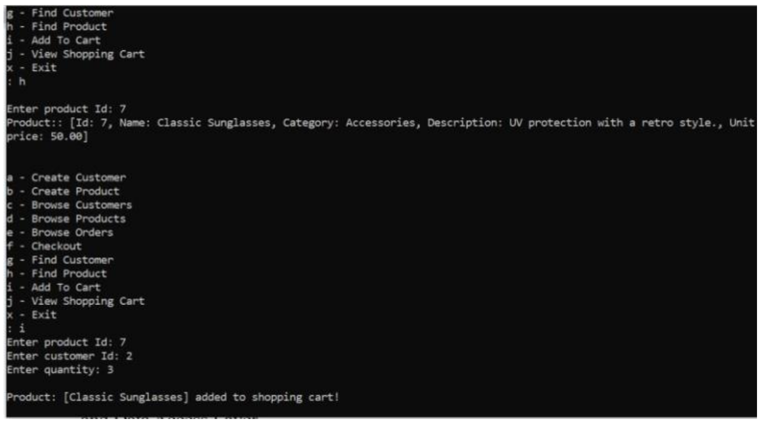
```
=======================
City-450 Shopping App
=======================

a - Create Customer
b - Create Product
c - Browse Customers
d - Browse Products
e - Browse Orders
f - Checkout
g - Find Customer
h - Find Product
i - Add To Cart
j - View Shopping Cart
x - Exit
:
```

The end user can select to browse the product, get the output of product ID, name, description, and price.



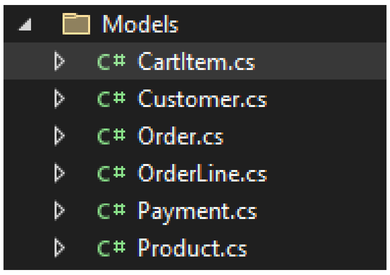
```
price: 199.00]

Product:: [Id: 7, Name: Classic Sunglasses, Category: Accessories, Description: UV protection with a retro style., Unit
price: 50.00]

Product:: [Id: 8, Name: Steel Water Bottle, Category: Gear, Description: Insulated stainless-steel bottle., Unit price:
25.00]

Product:: [Id: 9, Name: Bluetooth Headset, Category: Electronics, Description: Noise-cancelling wireless headset., Unit
price: 89.99]

Product:: [Id: 10, Name: Leather Wallet, Category: Accessories, Description: Handcrafted leather billfold., Unit price:
45.00]

a - Create Customer
b - Create Product
c - Browse Customers
d - Browse Products
e - Browse Orders
f - Checkout
g - Find Customer
h - Find Product
i - Add To Cart
j - View Shopping Cart
x - Exit
: h

Enter product Id:
```

The end user also can add the product in the shopping cart by entering the user ID, product ID and product quantity as below figure:

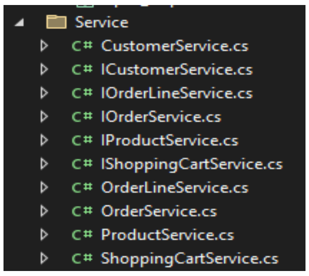


```
g - Find Customer
h - Find Product
i - Add To Cart
j - View Shopping Cart
x - Exit
: h

Enter product Id: 7
Product:: [Id: 7, Name: Classic Sunglasses, Category: Accessories, Description: UV protection with a retro style., Unit
price: 50.00]


a - Create Customer
b - Create Product
c - Browse Customers
d - Browse Products
e - Browse Orders
f - Checkout
g - Find Customer
h - Find Product
i - Add To Cart
j - View Shopping Cart
x - Exit
: i
Enter product Id: 7
Enter customer Id: 2
Enter quantity: 3

Product: [Classic Sunglasses] added to shopping cart!
```

Business Access Layer acts as a mediator layer that communicates between the User Interface Layer and the Data Access Layer. The layer is used to transfer the data between the User Interface Layer and Data Access Layer, it is mainly used for Validations and calculations purposes. The 'Model' folder contains the domain models which are used by the service layer to represent the business entities, such as 'Customer.cs', 'Order.cs', 'Product.cs' , and so on. The 'Service' folder contains service interfaces and their implementations. These services encapsulate the core business logic and rules of the application. The 'Model' folder is used by the 'Service' folder and the Data Access Layer to represent and manipulate the application's data structure throughout the business processes.



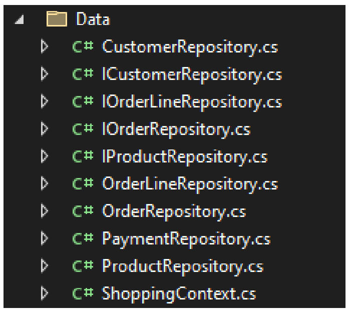
```
Models
  CartItem.cs
  Customer.cs
  Order.cs
  OrderLine.cs
  Payment.cs
  Product.cs
```



```
Service
  CustomerService.cs
  ICustomerService.cs
  IOrderLineService.cs
  IOrderService.cs
  IProductService.cs
  IShoppingCartService.cs
  OrderLineService.cs
  OrderService.cs
  ProductService.cs
  ShoppingCartService.cs
```

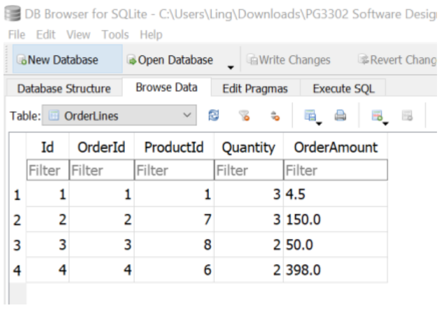
Data Access Layer communicates with the Business Access Layer, below data folder in the App contains the data access logic, including repositories and database context (‘ShoppingContext.cs’). This layer is responsible for communicating between the business logic layer and the database, performing CRUD operations, and persisting or retrieving data.

Data Access Layer



```
Data
    CustomerRepository.cs
    ICustomerRepository.cs
    IOrderLineRepository.cs
    IOrderRepository.cs
    IProductRepository.cs
    OrderLineRepository.cs
    OrderRepository.cs
    PaymentRepository.cs
    ProductRepository.cs
    ShoppingContext.cs
```

The orders are successfully saved in the database.



	Id	OrderId	ProductId	Quantity	OrderAmount
	Filter	Filter	Filter	Filter	Filter
1	1	1	1	3	4.5
2	2	2	7	3	150.0
3	3	3	8	2	50.0
4	4	4	6	2	398.0

The three-layered architecture ensures a separation of concerns, where each layer has a distinct responsibility and communicates with the adjacent layer through well-defined interfaces. The configuration facilitates maintenance, scalability, and testability of the application. It helps to easily maintain and understand large projects. It is more secure because the end users are not allowed to access the database directly. The developers can easily change any layer of code without affecting the other two layers. It makes the App more consistent.

Design Pattern

The Model-View-Controller (MVC) design pattern is used in the development of the shopping application to organize the programming in a way that improves maintainability, encourages modularity, and divides responsibilities.[11] The application's business logic and fundamental data structures are represented by the model. The models in our scenario are classes like Product, Customer, Order, and CartItem. The characteristics and actions related to goods, clients, orders, and things in the shopping cart are all encapsulated in these classes. The view in a conventional MVC application is in charge of displaying the user interface to the user. The several messages and prompts seen in the console in our console-based application indicate the perspective.

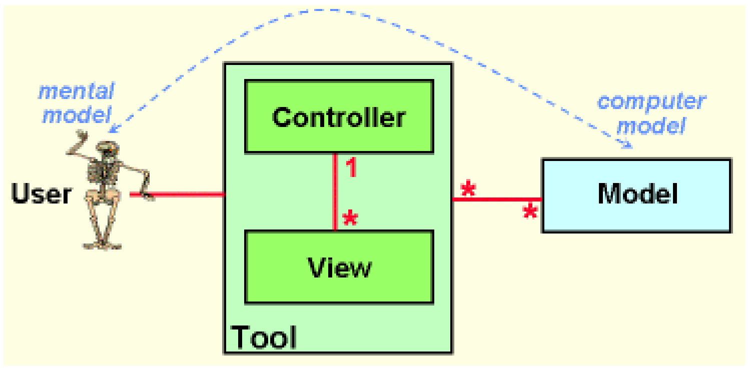


Figure source: Trygve/MVC (universitetetioslo.no)

In our application, we have the three parts of the MVC [7, 13, 8] software-design pattern that can be described as follows:

Models:

All the files within the Models folder are part of the Model component as these manage data and business logic[14]:

- CartItem.cs
- Customer.cs
- Order.cs
- OrderLine.cs
- Payment.cs

- Product.cs

Also, the ShoppingContext.cs is part of the Model in the MVC pattern because it describes how the data model is mapped to the database, it is also directly involved in setting up and configuring the database schema. This dual role is common in applications that use an ORM like Entity Framework.

View:

- In our console application, the Program.cs file is responsible for rendering the user interface to the console, thus serving as the View.

Controller:

- CustomerService.cs
- OrderService.cs
- ProductService.cs
- ShoppingCartService.cs

Furthermore, the corresponding interfaces (ICustomerService.cs, IOrderService.cs, IProductService.cs, IShoppingCartService.cs) define the contract for the services and are used by the Program.cs (acting as the controller in this context) to handle the operations.

The program class implements a console-based user interface and controls the flow of the application. The main function sets up the service scope for handling user input, migrates the database, and initializes the service provider. You may create orders, items, and customers from the application menu. You can also browse through the data that already exists and use the shopping cart.

The main classes that make up the application's structure each have a specific function. Functionalities of customer administration, product management, and shopping cart operations are encapsulated in the CustomerService, ProductService, and ShoppingCartService classes, respectively. These classes implement essential business logic and serve as the application's framework.

The introduction of interfaces like IShoppingCartService, IOrderService, IOrderLineService, ICustomerRepository, and IProductRepository facilitated dependency injection and encouraged modularity. This design decision improves the application's flexibility and testability by enabling component interchangeability.

Repository interfaces were injected into the service classes via dependency injection, which lessened tight coupling and encouraged modular architecture. The Microsoft ServiceCollection.Prolongations.The service provider was configured and constructed using DependencyInjection, which made it easier to inject dependencies into the application as a whole.

Asynchronous Programming and Threading

In our application, we've embraced asynchronous programming to enhance its performance, especially in tasks that involve database access or web requests. This is achieved through methods tagged with **async**, which allows these operations to be carried out without halting the execution of other code. This technique is particularly effective in ensuring that our application remains responsive and efficient. The use of **async** and **await** indicates that the operations are handled asynchronously, using threading in the background managed by the .NET framework.[3]

A key component of this approach is the use of the **await** keyword. By prefacing method calls with **await**, we instruct the program to pause at these points, waiting for the operation to finish. However, this doesn't render the program inactive. The executing thread remains active and can attend to other tasks, thereby avoiding any lag in application responsiveness.

This approach is exemplified in tasks like adding new customers. When the program initiates such an operation, it doesn't remain idle waiting for the task to complete. Instead, it temporarily hands over control and then resumes from where it stopped once the operation is completed. This responsive execution sequence is instrumental in keeping the user interface fluid and responsive, even during complex operations.

Our strategy of a non-blocking approach is essential for a smooth-running application. It allows the main thread to manage user interactions and other activities concurrently with ongoing operations. This means the application can continue to interact with the user or perform other tasks while waiting for an asynchronous operation to finish. It's also important to understand the distinction between asynchronous and parallel execution. In our use of **async** and **await**, we focus on handling tasks while waiting for other operations to complete, rather than executing multiple tasks at the same time. This approach is not about parallel processing but about efficient task management, ensuring our application remains highly responsive and effective.

Applying Threading for Responsiveness:

- **Threading in Action**: Our application employs threading to run asynchronous tasks, such as interacting with databases or APIs, without interrupting the main thread.
- **Real-World Example**: A practical use case is in our ShoppingCartService's Checkout function. It runs asynchronously, enabling users to interact with the console even during the checkout process, thereby enhancing responsiveness.

Specific Examples in our Application:

1. **Service Operations**: Functions like **await customerService.ListCustomersAsync();**, **await productService.ListProductsAsync();**, and **await orderService.ShowOrdersAsync();** utilize asynchronous programming to execute time-consuming tasks without blocking the main thread.
2. **Shopping Cart Checkout**: The checkout process, is triggered by **await shoppingCartService.Checkout(cartItems);**, is another example of asynchronous operation, ensuring the application remains responsive while processing user transactions.

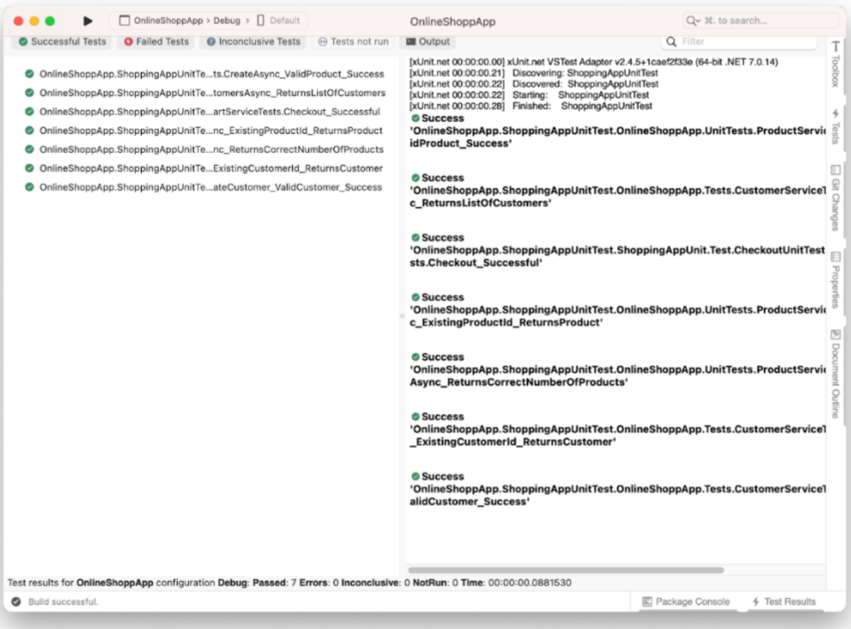
Through the strategic use of **async** and **await**, our application efficiently manages resource-intensive operations. This approach, a key part of our architectural design, significantly enhances the user experience by ensuring the application remains responsive and efficient, particularly during I/O-bound tasks such as database access or network communication.

XUnit Test

A strict unit testing approach was implemented during the development phase to guarantee the accuracy and dependability of the program. The Moq framework was essential in helping to create fake objects for repository interfaces so that service classes could be tested independently.

The testing approach adopted for this application is Xunit testing. While NUnit and Xunit share similarities, the decision to opt for Xunit was driven by its newer nature, excellent extensibility, support for parallel executions, and robust isolation between test methods. [12] Notably, Xunit's high-level isolation is particularly pivotal, as it generates a distinct instance of the test class for each test. This practice ensures a high degree of isolation, preventing interference between tests. [1]

For every service class, unit tests were developed that covered a range of scenarios, including adding customers, listing products, and checking out using the shopping cart. To provide a safety net for upcoming modifications and improvements, the tests concentrated on asserting the intended behavior of procedures under various scenarios. All tests were successfully run in debug mode, such as the image below depicts.



```
OnlineShoppApp.ShoppingAppUnitTe...ts.CreateAsync_ValidProduct_Success
OnlineShoppApp.ShoppingAppUnitTe...tomersAsync_ReturnsListOfCustomers
OnlineShoppApp.ShoppingAppUnitTe...artServiceTests.Checkout_Successful
OnlineShoppApp.ShoppingAppUnitTe...nc_ExistingProductId_ReturnsProduct
OnlineShoppApp.ShoppingAppUnitTe...nc_ReturnsCorrectNumberOfProducts
OnlineShoppApp.ShoppingAppUnitTe...ExistingCustomerId_ReturnsCustomer
OnlineShoppApp.ShoppingAppUnitTe...ateCustomer_ValidCustomer_Success
```

```
[xUnit.net 00:00:00.00] xUnit.net VSTest Adapter v2.4.5+1caef2f33e (64-bit .NET 7.0.14)
[xUnit.net 00:00:00.21]   Discovering: ShoppingAppUnitTest
[xUnit.net 00:00:00.22]   Discovered:  ShoppingAppUnitTest
[xUnit.net 00:00:00.22]   Starting:    ShoppingAppUnitTest
[xUnit.net 00:00:00.28]   Finished:    ShoppingAppUnitTest
Success
'OnlineShoppApp.ShoppingAppUnitTest.OnlineShoppApp.UnitTests.ProductServic
idProduct_Success'

Success
'OnlineShoppApp.ShoppingAppUnitTest.OnlineShoppApp.Tests.CustomerServiceT
c_ReturnsListOfCustomers'

Success
'OnlineShoppApp.ShoppingAppUnitTest.ShoppingAppUnit.Test.CheckoutUnitTest
sts.Checkout_Successful'

Success
'OnlineShoppApp.ShoppingAppUnitTest.OnlineShoppApp.UnitTests.ProductServic
c_ExistingProductId_ReturnsProduct'

Success
'OnlineShoppApp.ShoppingAppUnitTest.OnlineShoppApp.UnitTests.ProductServic
Async_ReturnsCorrectNumberOfProducts'

Success
'OnlineShoppApp.ShoppingAppUnitTest.OnlineShoppApp.Tests.CustomerServiceT
_ExistingCustomerId_ReturnsCustomer'

Success
'OnlineShoppApp.ShoppingAppUnitTest.OnlineShoppApp.Tests.CustomerServiceT
alidCustomer_Success'
```

```
Test results for OnlineShoppApp configuration Debug: Passed: 7 Errors: 0 Inconclusive: 0 NotRun: 0 Time: 00:00:00.0881530
Build successful.
```

Testing Overview

1. Checkout Unit Testing (CheckoutUnitTest.cs)

In the codebase, the testing framework is organized into three distinct parts, each serving a specific purpose. The first part, CheckoutUnitTest.cs, is focused on validating the success of the checkout process. This is achieved by constructing a list of cart items and establishing a mock behavior that simulates a customer placing an order and subsequently retrieving the order details.

Testing Steps:

- **Cart Item List Creation:** A list of cart items is formulated to simulate a transaction.
- **Mocked Customer Order Behavior:** A mock behavior is set up to emulate a customer initiating an order and fetching the corresponding order information.

2. Customer Service Unit Testing (CustomerServiceUnitTest.cs)

The second testing module, CustomerServiceUnitTest.cs, is designed to assess the fundamental functionality of creating a customer within the customer list. The Moq framework is employed to create a mock ICustomerRepository, allowing for the addition of a customer to the repository. Subsequently, verification is performed to ensure the proper addition of the customer. The testing also includes retrieving the details of the added customer from the mock repository and obtaining a list of all customers present in it.

Testing Steps:

- **Mocked Customer Repository Setup:** Utilizing Moq, a mock ICustomerRepository is established to facilitate controlled testing.
- **Customer Addition Verification:** The created customer is added to the repository, and verification is conducted to confirm the successful addition.
- **Customer Information Retrieval:** The added customer's information is retrieved from the mock repository for validation.
- **All Customer Retrieval:** The ability to retrieve a list of all customers from the repository is tested.

3. Product Service Testing (ProductServiceTests.cs)

In the ProductServiceTests.cs module, our focus lies on the meticulous testing of adding a product deemed valid for inclusion in the system. This involves the creation of a mock IProductRepository, where the newly introduced product is meticulously integrated. The subsequent phase involves comprehensive verification to ascertain the accuracy of the product's successful addition.

Testing Steps:

- **Mock Product Repository Setup:** To facilitate controlled testing, a mock IProductRepository is established.
- **Product Addition Verification:** The process involves adding the newly created product to the mock repository, followed by a meticulous verification process. This ensures that the product is seamlessly integrated into the repository as expected.
- **Product Information Retrieval:** Validation is conducted to ensure the successful retrieval of detailed information about the added product from the mock repository.
- **All Products Retrieval:** The ability to retrieve a comprehensive list of all products present in the mock repository is tested, ensuring the system's capacity to manage and provide information about the entire product catalog.

This rigorous testing approach aims to confirm the reliability and accuracy of the product addition functionality within the ProductService, contributing to the overall robustness and quality of the application. To ensure brevity this class diagram only includes the classes that are worth mentioning or those classes that form the basic units of the application.

Refactoring

<u>Removal of Category Class:</u>

The project no longer contains the Category class. This modification is the result of a decision to simplify things, most likely as a result of reevaluating the requirement for a certain Category class.

<u>CartItem as a Simple Class:</u>

A more straightforward class has been created out of the CartItem class. The fact that it is no longer a part of the database context suggests that the cart items are not kept in the database permanently. This change is in line with the general e-commerce situation, where data related to shopping carts is frequently handled session-by-session rather than being saved indefinitely.

<u>CartItem class deleted from database context:</u>

The database context no longer contains the CartItem class. This suggests that cart contents are handled as temporary data, existing just while the user is using them and without being saved for later use.

<u>Removal of ShoppingCart from Database Context:</u>

The ShoppingCart class has been eliminated from the database context, just as CartItem. Data from shopping carts is usually only needed for a single session and is not kept in a database permanently.

<u>Order as a Basic Class:</u>

An easier class has been created out of the Order class. This modification raises the possibility that, depending on the needs of the application, order information may be handled and simplified differently, maybe as a transitory or non-relational structure.

<u>ShoppingCart Class Removal:</u>

The ShoppingCart class has been eliminated. This indicates a choice to manage shopping cart functionality without requiring a separate class, maybe leading to a lighter approach to maintaining cart-related data.

<u>SoldProduct Class Removal:</u>

There has been a removal of the SoldProduct class. Most likely, the products in this class were components of finished orders. One reason for the removal could be to make the product depiction in the context of orders simpler.

<u>Stock Class Removal:</u>

The project no longer includes the Stock class. This points to a potential change in the way product stock and inventory are handled, which could simplify the rationale behind inventory management as a whole.

<u>Removal of Stock Class from Database Context:</u>

The Stock class has been taken out of the database context, much like the CartItem and ShoppingCart classes. This indicates that the database is no longer permanently storing data on stocks.

<u>Refinement of Unit Testing:</u>

Instead of conducting individual unit tests for each class, a more effective approach is to perform unit testing for specific parts that require examination. This targeted testing strategy allows us to assess and ensure the proper implementation of the essential functionalities within the application's classes.

Discussion for Further Improvement

While the current version of the online shopping app successfully covers essential functionalities such as adding customers, processing orders, and managing products, there are several areas for further improvement to enhance the user experience and extend the application's capabilities. Consider implementing features such as:

1. User Reviews and Ratings: Foster user engagement by allowing customers to leave product reviews and ratings. This not only provides valuable feedback but also helps potential buyers make informed decisions.
2. Personalized Recommendations: Implement a recommendation system that analyzes user preferences and purchase history to suggest personalized product recommendations. This can significantly improve the user experience and drive sales.
3. Mobile Responsiveness: Optimize the application for mobile devices to reach a broader audience. A responsive design ensures a consistent and user-friendly experience across different screen sizes.

About Bugs

As of now, there are no notable bugs identified in the code. The seamless connectivity between all layers and the database ensures smooth and cohesive functioning of the entire system.

Conclusion and Prospective Considerations

In conclusion, the successful completion of this project marks a significant achievement in the realm of C# development, showcasing a comprehensive and well-architected online shopping application. Employing a three-layered architecture and embracing the Model-View-Controller (MVC) design pattern, this project stands as a testament to effective software engineering principles and practices.

The project's core functionality revolves around the creation of a robust online shopping application. Leveraging various classes, the application seamlessly manages and orchestrates interactions between different components. The three-layered architecture provides a high-level structure and modular approach, enhancing maintainability and scalability.

The application's adherence to the MVC design pattern fosters a clean separation of concerns, promoting code organization and maintainability. The Model encapsulates the data and business logic, the View manages the user interface, and the Controller orchestrates interactions between the Model and View. This architectural paradigm not only enhances code readability but also simplifies future enhancements and modifications.

XUnit testing plays a pivotal role in ensuring the reliability and correctness of the implemented features. The incorporation of extensive unit tests, such as those in Xunit, contributes to the overall quality assurance of the application. Each unit test, whether for the checkout process, customer service, or product handling, has been meticulously designed to validate the intended functionality and prevent regressions.

Furthermore, the integration of a database to store customer and product information adds a layer of persistence and data management to the application. This database-centric approach ensures data integrity and facilitates seamless retrieval and storage of essential information.

In summary, this C# project exemplifies a holistic and well-engineered solution for an online shopping application. The three-layered architecture, XUnit testing practices, and adherence to the MVC design pattern collectively contribute to the project's success. The careful consideration of each component and their seamless integration culminate in a reliable and scalable application, poised to meet the demands of an ever-evolving digital marketplace. In the future, the program might benefit from more features like increased product management capabilities, better error handling, and user authentication. As the application develops, performance testing and scalability issues could also become critical.

Individual Contribution

Candidate 2041 contributed on:

- Project Structure Update: Enhanced organization and readability of the codebase.
- Data Model Simplification: Removed Category, ShoppingCart, SoldProduct, and Stock classes; converted CartItem and Order to simple classes.
- Database Context Refinement: Eliminated CartItem, ShoppingCart, and Stock from the database context for a streamlined schema.
- UML design
- XUnit-testing
- Asynchronous Programming and Threading
- MVC

Candidate 2075 contributed on:

- UML Class Diagram Design
- Initial Generation of the App coding framework
- Layering design and coding
- Establish of Database
- User interface and menu design and coding

Candidate 2007 contributed on:

- Xunit testing
- Provided ideas on further improvement
- Refactoring of unit testing
- Storage of data in data access layer

Reference

[1] Erin. 2022. "XUnit vs. NUnit Demystified: A Comprehensive Explanation." AI-Driven E2E Automation with Code-like Flexibility for Your Most Resilient Tests. April 1, 2022.

https://www.testim.io/blog/xunit-vs-nunit/

[2] Geeksforgeeks 2022. "SOLID Principle in Programming: Understand with Real Life Examples." GeeksforGeeks. 05 Dec, 2022. https://www.geeksforgeeks.org/solid-principle-in-programming-understand-with-real-life-examples/

[3] Kathleen Dollard. 2022. "Asynchronous Programming with Async and Await - Visual Basic." Learn.microsoft.com. September 21, 2022. https://learn.microsoft.com/en-us/dotnet/visual-basic/programming-guide/concepts/async/

[4] Kirill Fakhroutdinov. 2018. "UML Class and Object Diagrams Overview - Common Types of UML Structure Diagrams." Uml-Diagrams.org. April 21, 2018. https://www.uml-diagrams.org/class-diagrams-overview.html

[5] Kumar, Sharath. 2023. "How to Implement 3 Layered Architecture Concepts in ASP.net." Www.c-Sharpcorner.com. August 23, 2023. https://www.c-sharpcorner.com/UploadFile/7be47b/how-to-implement-3-tier-architecture-concepts-in-Asp-Net/

[6] Lucidchart. 2019. "Introducing Types of UML Diagrams | Lucidchart Blog." Lucidchart.com. April 5, 2019. https://www.lucidchart.com/blog/types-of-UML-diagrams

[7] Microsoft 2022. "Overview of ASP.NET Core MVC." Learn.microsoft.com. June 27, 2022. https://learn.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-8.0&WT.mc_id=dotnet-35129-website

[8] Mozilla. 2023. "MVC." MDN Web Docs. 2023. https://developer.mozilla.org/en-US/docs/Glossary/MVC

[9] Negi, M. (2019). Fundamental of database management system: Learn essential concepts of database systems. BPB Publications.

[10] Price, M. J. (2021). C# 10 and .NET 6 – Modern cross-platform development: Build apps, websites, and services with ASP.NET core 6, Blazor, and EF core 6 using visual studio 2022 and visual studio code. Packt Publishing.

[11] Saltzer, J. H., & Kaashoek, M. F. (2019). Principles of computer system design: An Introduction. Morgan Kaufmann.

[12] Sheth, Himanshu. 2021. "NUnit vs. XUnit vs. MSTest: Comparing Unit Testing Frameworks in C#." LambdaTest. March 22, 2021. https://www.lambdatest.com/blog/nunit-vs-xunit-vs-mstest/

[13] Trygve. n.d. "Trygve/MVC." Folk.universitetetioslo.no. https://folk.universitetetioslo.no/trygver/themes/mvc/mvc-index.html

[14] Voorhees, D. P. (2020). Guide to efficient software design: An MVC approach to concepts, structures, and models. Springer Nature.

[15] Xu, D. (2021). Modern software engineering: Principles & practices : Writing clean dependable code.