WESTMINSTER INTERNATIONAL UNIVERSITY IN TASHKENT

COURSEWORK SUBMISSION FORM

STUDENT USE		STAFF USE	
Module Name	Object Oriented Programing	First Marker's (acts as signature)	
Module Code	5COSC018C	Second Marker's (acts as signature)	
Lecturer Name	Avaz Khalikov	Agreed Mark	
UoW Student IDs		**For Registrar's office use only (hard copy submission)**	
WIUT Student IDs	00012860		
Deadline Date	15.12.2021		
Assignment Type	☐ Group [✓ Individual]		
Word Count			

SUBMISSION INSTRUCTIONS

COURSEWORKS *must* be submitted in *both* HARD COPY (to the Registrar's Office) *and* ELECTRONIC unless instructed otherwise.
For hardcopy submission instructions refer to:
http://intranet.wiut.uz/Shared%20Documents/Forms/AllItems.aspx - Coursework hard copy submission instructions.doc
For online submission instructions refer to:
http://intranet.wiut.uz/Shared%20Documents/Forms/AllItems.aspx - Coursework online submission instructions.doc

MARKERS FEEDBACK (Continued on the next page)

Contents

Introduction

As an architect, constructing core functional of any system is requires huge amount of time. Every business to develop, should have a maintainable, scalable, optimized solution. So, it is architects' responsibility to develop a software in a way that would not generate problems or system that is optimized. In position of software architect, I have three projects that I am assigned to build or struct part of its core functional. The list of scenarios is:

1. **Online Car Auction System.** This is an application in which Owners of some precious car, has ability to sell their property in auction. As name says, auction has different roles and processes. We can assume case like, every user can participate in online auction they want, they can bid on a car. The sellers can put their staff on auction when they are registered as a seller role. System has administrator that can control each auction process. Selling car must have starting price. Users can only offer amount that is higher than starting price or the last price that were recorded while bidding. In details, there should be 3 roles: Buyer, Seller, Administrator. Once time of auction ends, all users who participated should be notified results of auction.
2. **University Events Management System.** This is a system where all of university's events are shown. It has role-based access control of events. Staff can create events and every user in the system can view it, subscribe or unsubscribe. Events has starting and ending time, date, name, type and etc. Only staff can do CRUD operations over events. So involved users are students and staff who will have different calendar view according to their position. Usually, such kind of services are a subsystem or part of a huge learning platforms.
3. **Library Book Management System.**

   Library management systems are used by students, librarian and admin. Users can search a book they want, take it on loan or reserve it. Books are managed by librarian. Librarians can do create, delete, update books. Also, there are admins who can do another operation on library. Each book's information is stored in database and status of the book updated every time when it is available or on loan. There might be e-books, which are available every time and has no status. Book management application should have search system where everyone can find a book they need. Other than that, there should be services to check book return date, notify users about book when its status updated, check if loan time not ended. So, library management is a platform with tons of services.

Scenario 1 – Online Car Auction

One of auction requirements may be an advanced bid control. In case requirements, it is stated that there are three roles, and each role has its own control on auction management. Not every role has direct access to auction. For example, seller can create auction, change or describe property that is intended to sell, administrator can confirm auction creation and determine auction results. Buyers can only get information about auction and bid on it.

Naïve code

Auction.cs

```
public class AuctionRealObject
    {
        private Car _auction;

        public Car createAuction(string Name, string Brand,
                 int ProductionYear, int StartPrice)
        {
            _auction = new Car(Name, Brand, ProductionYear, StartPrice);
            return _auction;
        }

        public void deleteAuction()
        {
            _auction = null;
        }

        public void confirm()
        {
            _auction.IsConfirmed = true;
        }

        public void bid(int Price)
        {
            _auction.Bids.Add(new Bid(Price, _auction.Id));
        }
    }
```

The problem with this code is wherever this class is used it will cause some security issues. For example, deleting auction will be available to every user role. Or any other methods which are not right of some users to perform will create a risk of security break.

Solution

Looking for the solutions, we can come up with an idea that, we are able to create different proxy class objects dividing functionality. In details, we can have different access points with different operation allowed. So, Proxy Design pattern is the best solution this type of problem. What Proxy Design does is controls access to the real object, we can add extra functionality when accessing object. We define separate proxy object for different roles, to restrict direct entry to sensitive operations.

Proxy.cs

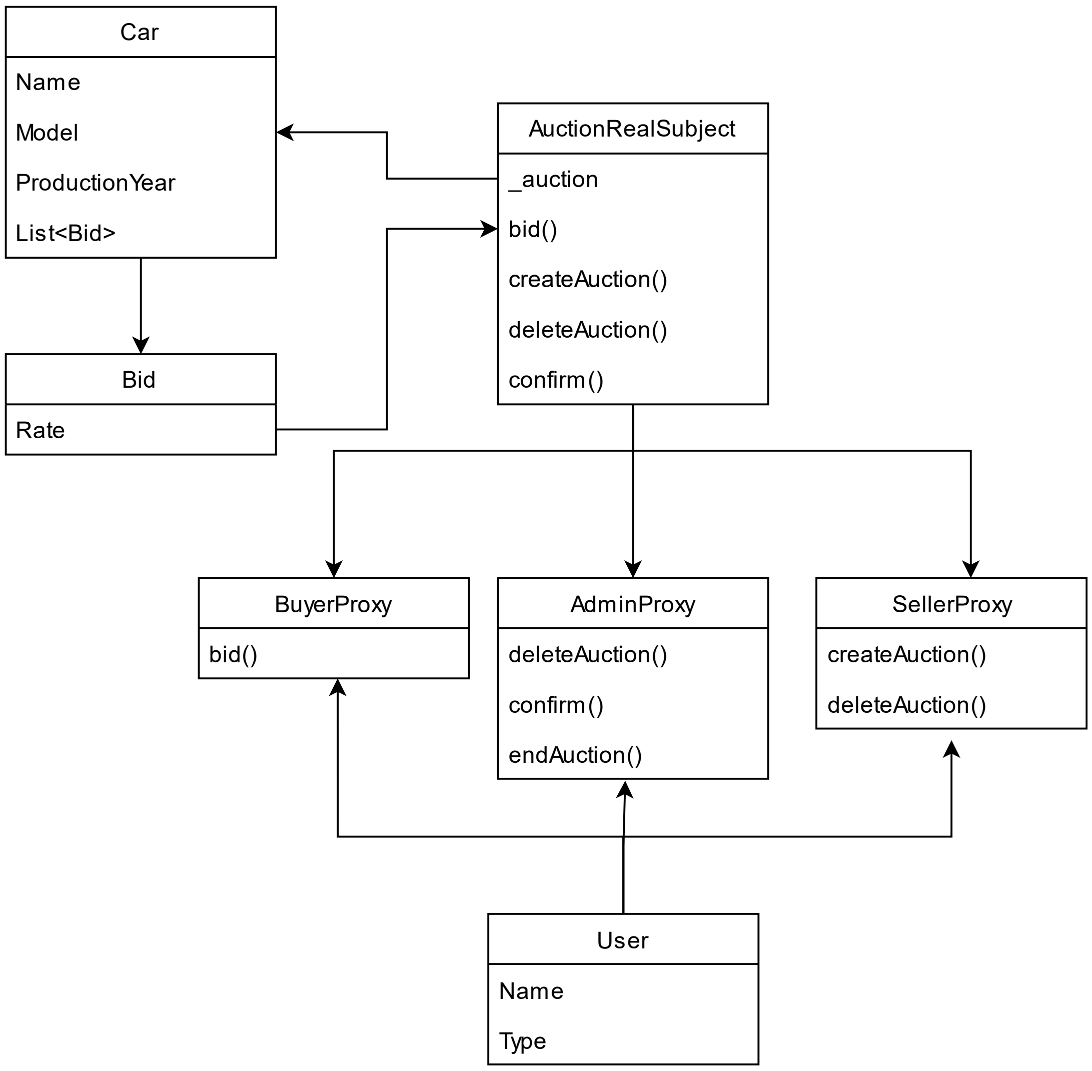
```
public class AdminProxy
    {
        private AuctionRealObject _subject;
        public AdminProxy(AuctionRealObject RealSubject)
        {
            _subject = RealSubject;
        }

        public void confirm()
        {
            _subject.confirm();
        }

        public void deleteAuction()
        {
            _subject.deleteAuction();
        }
    }

    public class BuyerProxy
    {
        private AuctionRealObject _subject;
        public BuyerProxy(AuctionRealObject RealSubject)
        {
            _subject = RealSubject;
        }

        public void bid(int Price)
        {
            _subject.bid(Price);
        }

    }
```

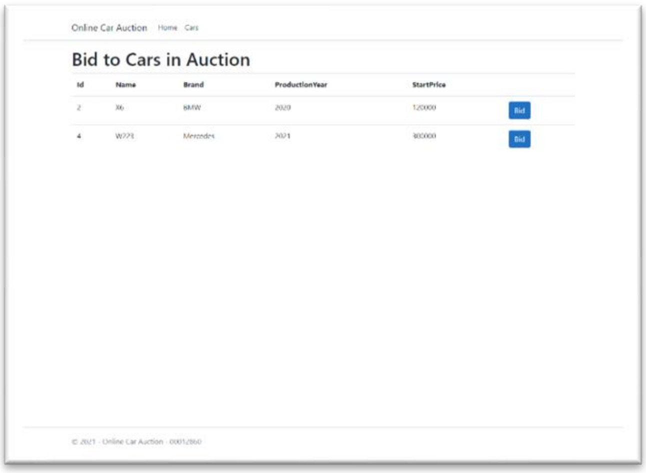
Program.cs

```
AuctionRealObject auctionObj = new AuctionRealObject();
auctionObj.createAuction("W223", "Mercedes", 2021, 200000);

AdminProxy adminProxy = new AdminProxy(auctionObj);
adminProxy.confirm();

BuyerProxy buyerProxy = new BuyerProxy(auctionObj);
buyerProxy.bid(210000);
```

Now we have different proxy classes for each role. AdminProxy cannot access bidding method, or BuyerProxy cannot delete Auction. In term of scalability and maintainability, if we have new another role, we can easily create another proxy specific for it. Or if we want to add more method, we can just add it to main subject class, and none of our proxies will be affected.

UML diagram

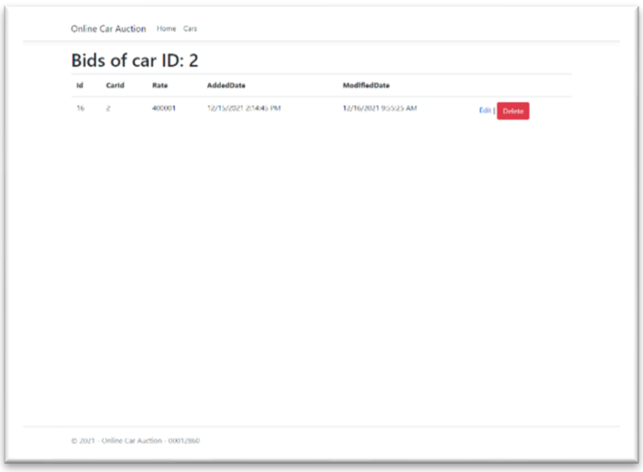



Online Car Auction Home Cars

Bid to Cars in Auction

Id	Name	Brand	ProductionYear	StartPrice	
2	X6	BMW	2020	120000	Bid
4	W223	Mercedes	2021	300000	Bid

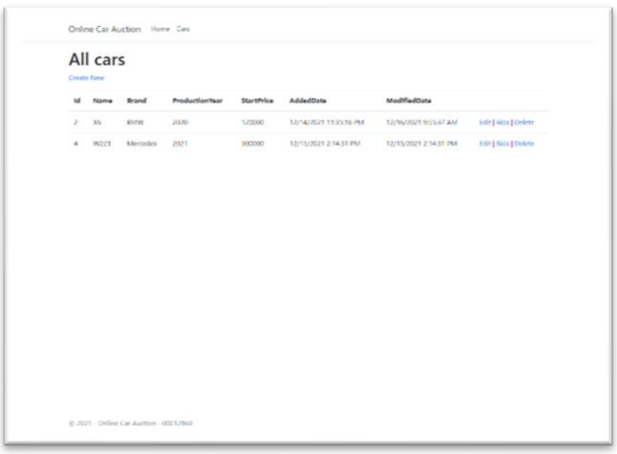
© 2021 - Online Car Auction - 00012860


Online Car Auction Home Cars

Bids of car ID: 2

Id	CarId	Rate	AddedDate	ModifiedDate	
16	2	400001	12/15/2021 2:14:45 PM	12/16/2021 9:55:25 AM	Edit | Delete

© 2021 - Online Car Auction - 00012860


Online Car Auction Home Cars

All cars

Create New

Id	Name	Brand	ProductionYear	StartPrice	AddedDate	ModifiedDate	
2	X6	BMW	2020	120000	12/14/2021 11:35:16 PM	12/16/2021 9:55:47 AM	Edit | Bids | Delete
4	W223	Mercedes	2021	300000	12/15/2021 2:14:31 PM	12/15/2021 2:14:31 PM	Edit | Bids | Delete

© 2021 - Online Car Auction - 00012860

Fully functioning CRUD, bidding

Scenario 2 – University events management system

Usually, Events Management System should have some type of notification service prior to event. So, events must have event subscription feature. This means, for each event there should be separate list of users more precisely event subscribers. Student should be able to subscribe one or more events. There should be one calendar object for everyone. Because all events must be only in one calendar.

Naïve code

There are many possible ways of implementing such event system. For example, we can create a student class, which will have list of events students has subscribed. However, this may create problems when implementing notification system. Because students have same event object in their event list property. Notifying will be mass strain, because our program should iterate each student and check if student has been subscribed to some particular event, then notify.

Solution

One of the best and only optimal solution would be using Observer Design Pattern. Students are observer who are notified of upcoming event. After implementation of observer design pattern, our program does much less computation. Main advantage of this design pattern is we will not have student class that has list of events, but event that has list of students who are subscribed to it. This solution is much better than standard/naïve approach, because we got a way to contact to events happening in other objects without coupling to their classes. Other than that, new solution has single calendar object every time. Usage of Singleton Design pattern ensure that we have only one instance of calendar.

Calendar.cs

```
public sealed class Calendar : ICalendar
    {
        private static Calendar instance = null;
        private static List<Event> Events;
        public static Calendar GetInstance
        {
            get
            {
                if (instance == null)
                {
                    instance = new Calendar();
                    Events = new List<Event>();
                }
                return instance;
            }
        }

        public void AddEvent(Event evt)
        {
            Events.Add(evt);
        }

        public void RemoveEvent(Event evt)
        {
            Events.Remove(evt);
        }

        // … … …
    }
```

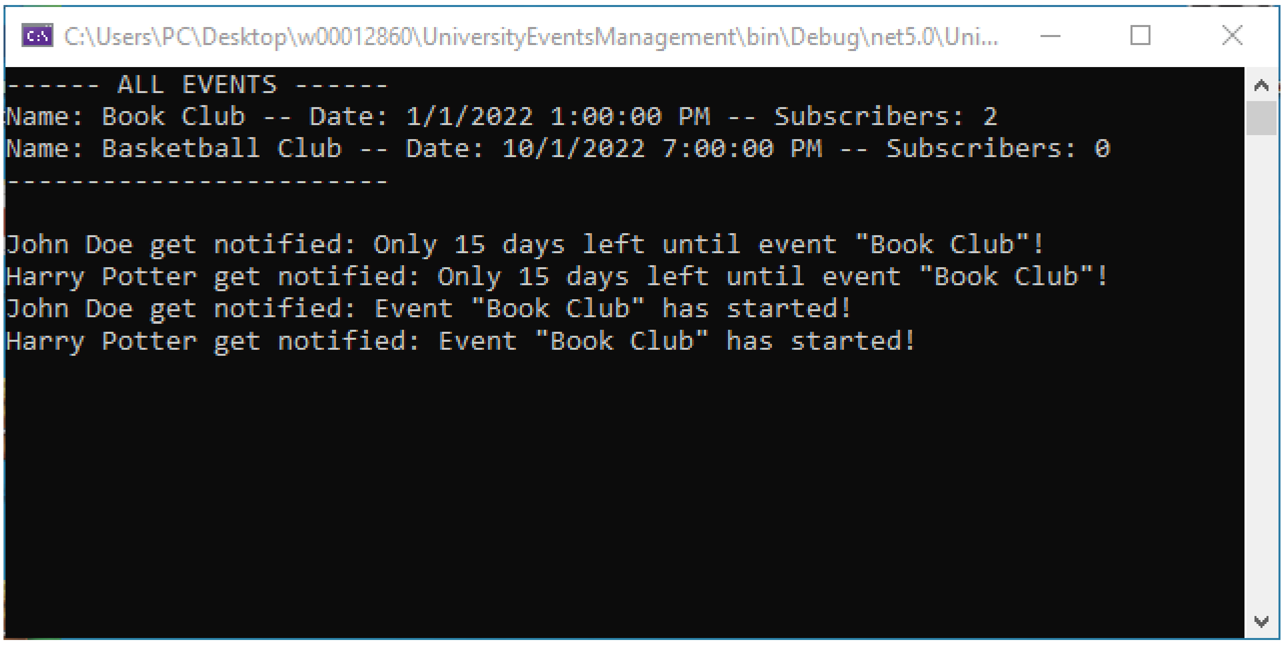
Event.cs

```
public class Event : IEvent
   {
       public string Name { get; set; }
       public DateTime Date { get; set; }

       List<Student> students;

       public Event(string name, DateTime date)
       {
           Name = name;
           Date = date;
           this.students = new List<Student>();
       }

       public void NotifyDaysLeft()
       {
           int daysLeft = (int)Math.Floor((Date - DateTime.Now).TotalDays);
           foreach (Student student in students)
           {
               student.GetNotified(this, daysLeft);
           }
       }

       public void Start()
       {
           Notify();
       }

       public void Notify()
       {
           foreach (Student student in students)
           {
               student.GetNotified(this);
           }
       }

       public void Subscribe(Student student)
       {
           students.Add(student);
       }

       public void Unsubsribe(Student student)
       {
           students.Remove(student);
       }

        // … … …
}
```

Program.cs

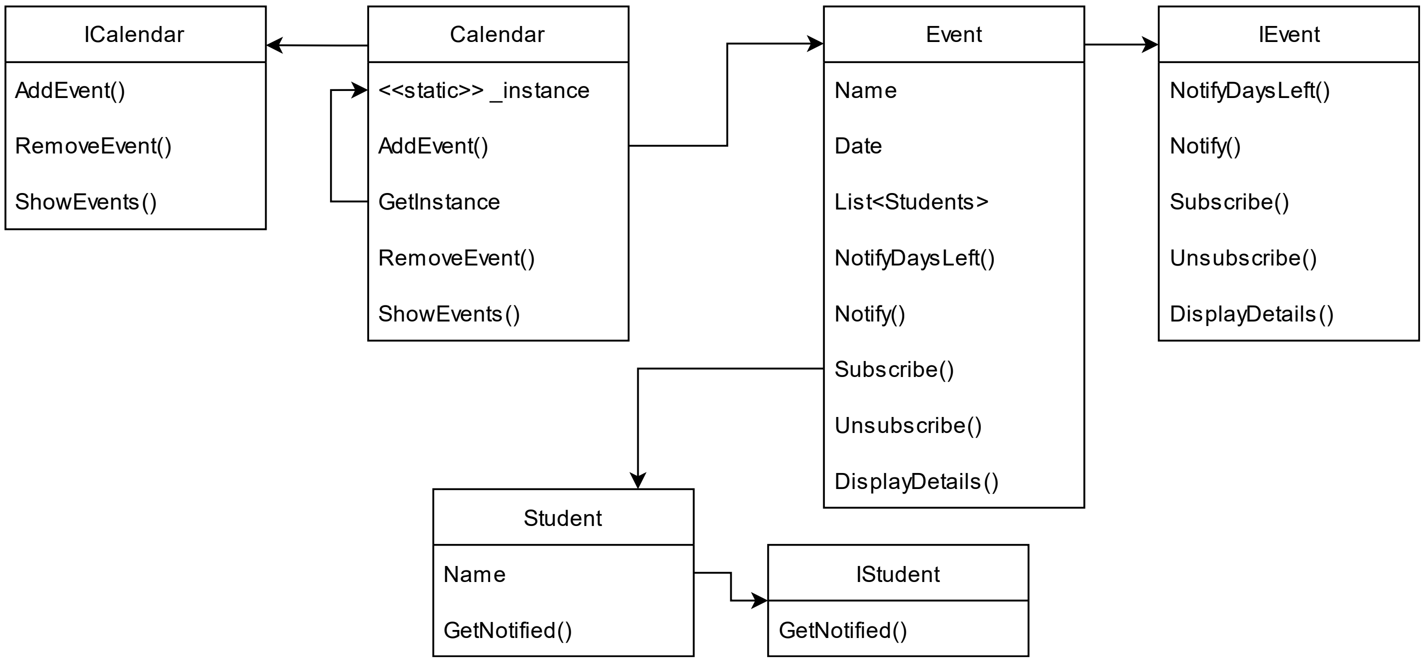
```
static void Main(string[] args)
        {

            ICalendar calendar = Calendar.GetInstance;

            Event evt1 = new Event("Book Club", DateTime.Parse("01-01-2022 13:00"));
            Event evt2 = new Event("Basketball Club", DateTime.Parse("10-01-2022 19:00"));

            Student st1 = new Student("John Doe");
            Student st2 = new Student("Harry Potter");

            evt1.Subscribe(st1);
            evt1.Subscribe(st2);
            calendar.AddEvent(evt1);
            calendar.AddEvent(evt2);

            calendar.ShowEvents();

            evt1.NotifyDaysLeft();
            evt1.Start();

        }
```

Above I used Observer and Singleton Design Patterns, because only one calendar is present. Final result is like this:


C:\Users\PC\Desktop\w00012860\UniversityEventsManagement\bin\Debug\net5.0\Uni...

```
------ ALL EVENTS ------
Name: Book Club -- Date: 1/1/2022 1:00:00 PM -- Subscribers: 2
Name: Basketball Club -- Date: 10/1/2022 7:00:00 PM -- Subscribers: 0
------------------------

John Doe get notified: Only 15 days left until event "Book Club"!
Harry Potter get notified: Only 15 days left until event "Book Club"!
John Doe get notified: Event "Book Club" has started!
Harry Potter get notified: Event "Book Club" has started!
```

UML diagram



Scenario 3 - Library Book Management System

As I said above, library book management system has lots of services to provide fully functioning interface. For example, our program should control users, books, events and offer other kinds of services to. Only Book object itself has many methods in different part of the application. In such cases, our system will be hard to analyze, maintain and refactor. Also using of different services from various classes confuse us and it will be hard to extend our application.

Naïve code

In standard, naïve implementation we can easily build our system step by step. Our program work as expected with full features. However, it has many drawbacks. For example, we have one service called search that helps to search some specific book in our database. And we have another class that does CRUD operations over Books. And there are many other functions in our project related to Book or some database entity. If we should have to use 2 or more services, we import them from different classes. This will create code duplication and generate hardness to find our services.

Solution

In such situations, we should have to implement something like wrapper class to our all-related sub classes. Façade design pattern is a good solution for this case. Façade design pattern is a structural design pattern that will simplify our interface. As our library system gets bigger, we will have complex set of classes. Façade design pattern creates us one wrapper class in which there are set of methods from different classes.

LibraryFacade.cs

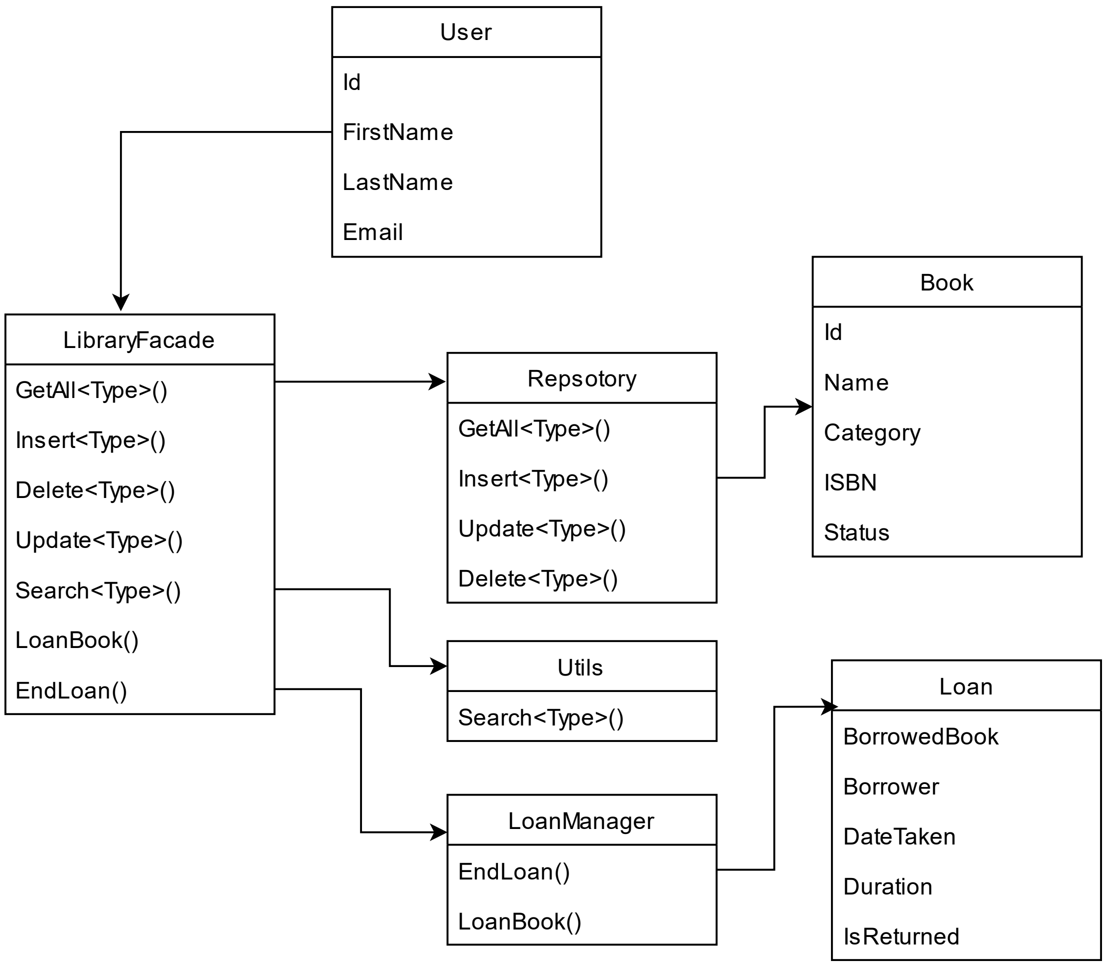
```
public class LibraryFacade
    {
        private readonly MyDbContext context;
        public LibraryFacade(MyDbContext context)
        {
            this.context = context;
        }

        public IEnumerable<T> GetAll<T>() where T : BaseEntity
        {
            return new Repository<T>(context).GetAll();
        }

        public  void  Insert<T>(T entity) where T : BaseEntity
        {
            new Repository<T>(context).Insert(entity);
        }

        public T Get<T>(int id) where T : BaseEntity
        {
            return new Repository<T>(context).Get(id);
        }

        public IEnumerable<T> Search<T>(string query) where T: BaseEntity
        {
            return new Utils<T>(context).Search(query);
        }

        public void Update<T>(T entity) where T : BaseEntity
        {
            new Repository<T>(context).Update(entity);
        }

        public void Delete<T>(T entity) where T : BaseEntity
        {
            new Repository<T>(context).Delete(entity);
        }

        public void EndLoan(long id)
        {
            new LoanManager(context).EndLoan(id);
        }

        public void LoanBook(Loan l)
        {
            new LoanManager(context).LoanBook(l);
        }
    }
```

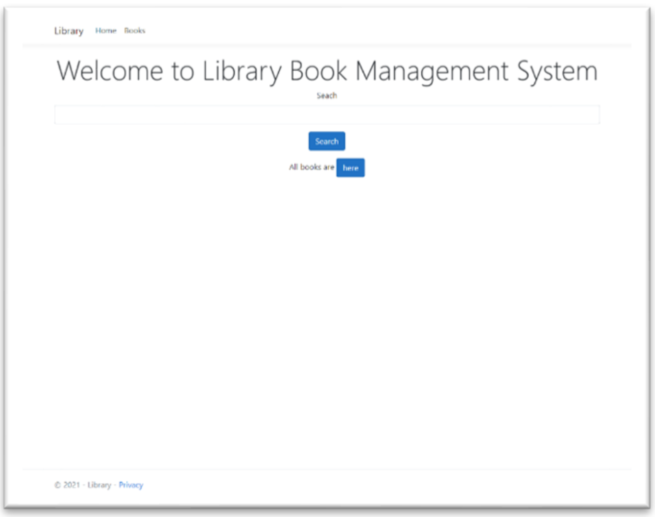
Somewhere else in our code

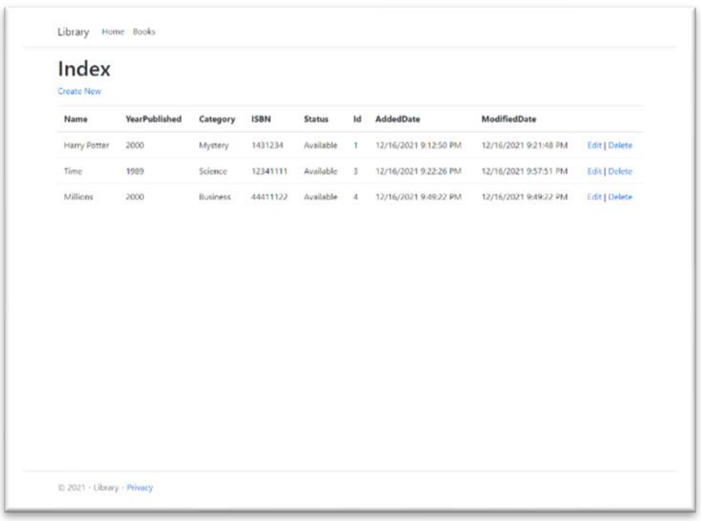
```
LibraryFacade _facade = new LibraryFacade(context);
_facade.LoanBook(Loan l);
_facade.EndLoan(long id);
_facade.Delete<Book>(Book b);
_facade.Update<Book>(Book b);
_facade.Insert<Book>(Book b);
_facade.Get<Book>(long id);
```

In the code above, we have combined our different services into one class. So, with this one Façade object we are able to do as many operations as we can. Thus, we will have created some easiness in our application. Without this pattern we would have many instances of same class in different part of our program. Other than that, in the code above, we have Repository Design pattern. In our database we have different table, and for each table we should have CRUD functionality. Instead of creating multiple CRUD services for each our models, we can create one service that fits all types of data. More precisely, repository design pattern mediates between the domain and the data mapping layers using a collection-like interface.

UML diagram




Library
Home
Books
Welcome to Library Book Management System
Seach
Search
All books are
here
© 2021 - Library - Privacy



Name	YearPublished	Category	ISBN	Status	Id	AddedDate	ModifiedDate	
Harry Potter	2000	Mystery	1431234	Available	1	12/16/2021 9:12:50 PM	12/16/2021 9:21:48 PM	Edit | Delete
Time	1989	Science	12341111	Available	3	12/16/2021 9:22:26 PM	12/16/2021 9:57:51 PM	Edit | Delete
Millions	2000	Business	44411122	Available	4	12/16/2021 9:49:22 PM	12/16/2021 9:49:22 PM	Edit | Delete

Search book, Book CRUD

Conclusion

In conclusion, I can say that underneath of any stable application there is a good use of different design patterns. Our software may work without design patterns. However, not all part of our solution is optimal in term of reusability, scalability and maintainability. With advantage of design patterns, we can build microservices or functions that work in different areas of our application without duplication. As an imaginary architect, finding right and compatible pattern was hard work and take vast amount of time. Learning of different kinds of design pattern was way challenging, because there were many uses of abstraction, rules, code separation. Applying design pattern was another tough task, as it changed project architecture a lot. By learning and using different types (structural, creational, behavioral) of design patterns, I gained much experience that enhanced my programming skills, because now I know that programming is not just coding, it is about design. From my personal experience I is clear that design pattern is one of the core principles of Object Oriented Programming.

References

1. ajcvickers (n.d.). *Querying and Finding Entities - EF6*. [online] docs.microsoft.com. Available at: https://docs.microsoft.com/en-us/ef/ef6/querying/ [Accessed 17 Dec. 2021].

2. GeeksforGeeks. (2020). *Class Diagram for Library Management System*. [online] Available at: https://www.geeksforgeeks.org/class-diagram-for-library-management-system/.

3. Wikipedia. (2020). *Facade pattern*. [online] Available at: https://en.wikipedia.org/wiki/Facade_pattern.

4. www.c-sharpcorner.com. (n.d.). *Repository Design Pattern In ASP.NET MVC*. [online] Available at: https://www.c-sharpcorner.com/article/repository-design-pattern-in-asp-net-mvc/#:~:text=By%20definition%2C%20the%20Repository%20Design [Accessed 17 Dec. 2021].

5. www.dotnettricks.com. (n.d.). *Observer Design Pattern - C#*. [online] Available at: https://www.dotnettricks.com/learn/designpatterns/observer-design-pattern-c-sharp [Accessed 17 Dec. 2021].

6. www.newthinktank.com. (n.d.). *Facade Design Pattern Tutorial*. [online] Available at: http://www.newthinktank.com/2012/09/facade-design-pattern-tutorial/ [Accessed 17 Dec. 2021].

7. www.youtube.com. (n.d.). *Code First Tutorial, Repository Pattern ASP.NET MVC C#, below github code*. [online] Available at: https://www.youtube.com/watch?v=pNfzjvk7ik0 [Accessed 17 Dec. 2021].