



MTU

Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Refactoring Software Architecture to Adhere to Design Patterns

BSc. (Hons) In Computing With Software Development

TL_KOMP_BY4

Kyle Tenkrooden

12 December 2022

Table of Contents

1	Background Information	4
2	Existing structure	5
	Scalability	6
	Maintainability Costs	6
	Security Vulnerabilities	6
2.1	Functional Requirements	7
	Add Member	7
	Add Membership Type.....	7
	Update Member.....	7
	Issue reminder	7
2.2	Associations between requirements and use cases (including UML use case diagrams)	8
	Add Member	8
	Add membership type	8
	Update Membership Type	9
	Issue reminder	9
2.3	Non-Functional Requirements	9
	User Experience and Usability	9
	Security and Privacy	10
	Performance	10
	Availability -.....	10
2.4	Associations between Non-functional requirements and use cases	10
	Usability	10
	Security and Privacy	10
	Performance	11
	Availability.....	11
3	Design Rationale	11
3.1	Pros And Cons of Layered Architecture Pattern	11
	Pros	11
	Cons.....	12
3.2	Pros And Cons of Adapter Pattern	12
	Pros	12
	Cons.....	13

3.3	Pros And Cons of Command Pattern	13
	Pros	13
	Cons.....	14
3.4	Pros And Cons of Chain Responsibilities Design Pattern	14
	Pros	14
	Cons.....	14
4	Redesign and Refactor	15
4.1	Implementation of the Layered Architecture	15
	Description	15
	UML.....	16
	Code	16
4.2	Implementation of Adapter Pattern	22
	Description	22
	UML.....	23
	Code	23
4.3	Implementation of Command Design Pattern.....	27
	Description	27
	UML.....	28
	Code	28
4.4	Implementation of Chain Of Responsibilities Pattern	34
	Description	34
	UML.....	35
	Code	35
5	Conclusions and Reflections	39
5.1	Conclusions	39
5.2	Reflections.....	40
6	References	43
7	Appendices.....	44
7.1	Appendix A	44
7.2	Appendix B	45

1 Background Information

A fitness centre's daily operations involve several duties that employee staff must complete. These tasks take up a lot of time, which has a cumulative effect on the company's profitability. Gym Management Software takes the place of tedious administrative burdens which allows staff to focus on other tasks. The customer experience is also suggested to be greatly enhanced as memberships and payments are well managed which promotes the retention and attraction of customers which ultimately increases the organisation's profitability.

The elegance of such a system is revealed when conducting annual analysis or insights. For example, analysing membership types to produce a thorough chart on which membership types are producing the most revenue. Consequently, customer usage patterns and demographics can be considered to better understand the customer to make more informed business decisions. This leads to a significant reduction in the time spent on paperwork or manual calculations.

The gym membership system consists of four main functional components: Membership Types, Members, Renewals and Administration.

Membership types provides the necessary functionality to add or update a Membership Type. A typical example of this could be a 3-month Gym membership. The Members component will provide functionality for adding Members, editing a member's details, and finally showing the Members details recorded on the system. The Renewals component will issue reminders to all Members whose memberships are approaching their expiration date. These members will then have an option to renew their memberships if desired.

Figure 1.0 below illustrates the various functional components of the 'GymMembership_SYS' program.

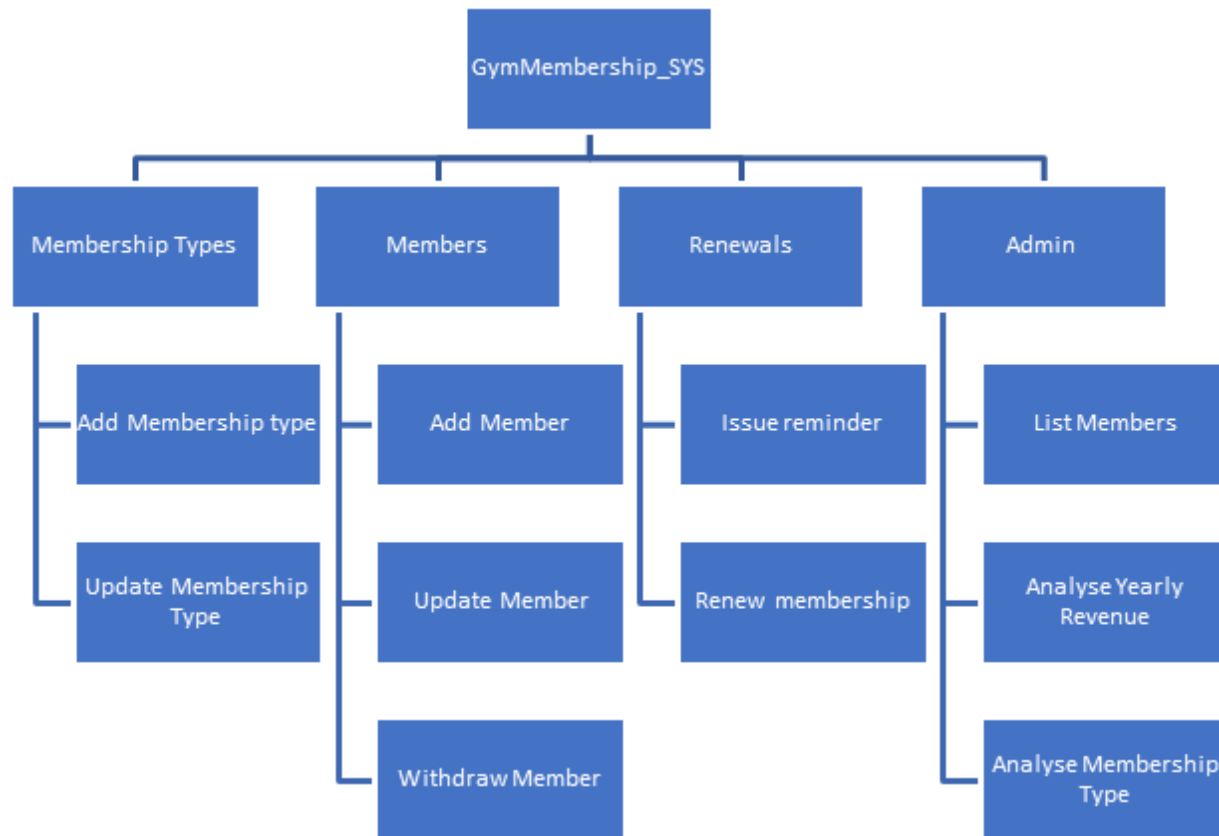


Figure 1.0 Gym Membership System Component Breakdown

2 Existing structure

The existing structure of the application originates from a past application of the 'Requirements Engineering' module. Although the application meets the functional requirements of a Gym Membership System, there are several issues with the application from a design point of view. The UML class diagram has been included in Appendix A and Appendix B. As shown, the existing structure has no relationships which creates a tightly coupled infrastructure. While this can occasionally be beneficial, it can lead to serious potential issues for the application. Firstly, as all the components are heavily dependent on one another, this creates a very difficult system to manage or extend (Gamma, et al., 1994). In the instance of extending the functionality of the software, this can result in duplication of code and a vulnerable system. Often, a tightly coupled program has a high level of complexity as all the components are closely integrated which often lead to the "God Class" code smell (McLaughlin, et al., 2006). Consequently, the testability is also

a major concern of the system as each class violates OOP principles, especially Single Responsibility Principles as the logic for the business, UI and data layers are tightly coupled. The Open closed Principle is also violated as the classes do not adhere to being “*Open for extension and closed for modification*” as any changes would entail modifying the class itself (McLaughlin, et al., 2006).

The main issues of the existing structure will be expanded on below:

Scalability: Programs that are tightly coupled are less scalable than the contrary loose coupled programs (Freeman, et al., 2004). This can be attributed to the close integration of components which can create bottlenecks and performance issues. Often this becomes a primary concern when the application is scaled to handle more complex tasks as modifying existing components could affect the rest of the system. As a result, refactoring the system is critical in adhering to OOP concepts to allow for the possibilities of scalability in the future.

Maintainability Costs: The maintainability of the application is greatly impacted when the application does not adhere to any OOP concepts (Freeman, et al., 2004). The tightly coupled relationship formed makes the application difficult to manage or debug as any changes has a cascading effect on the rest of the functional components within the application. This leads to additional costs and resources required in solving the issues which directly affect the costs of maintaining the system.

Security Vulnerabilities: Security is of paramount importance when handling User Data. Adhering to GDPR standards is a crucial step in securing customers legally. A system with tightly coupled components results in several system vulnerabilities and concerns (Gamma, et al., 1994). The vulnerabilities are produced as any flaws within a specific component can rapidly propagate throughout the system leading to increased risk of exposing sensitive user data.

2.1 Functional Requirements

Figure 2.0 below illustrates the functional components of the Gym Membership System.

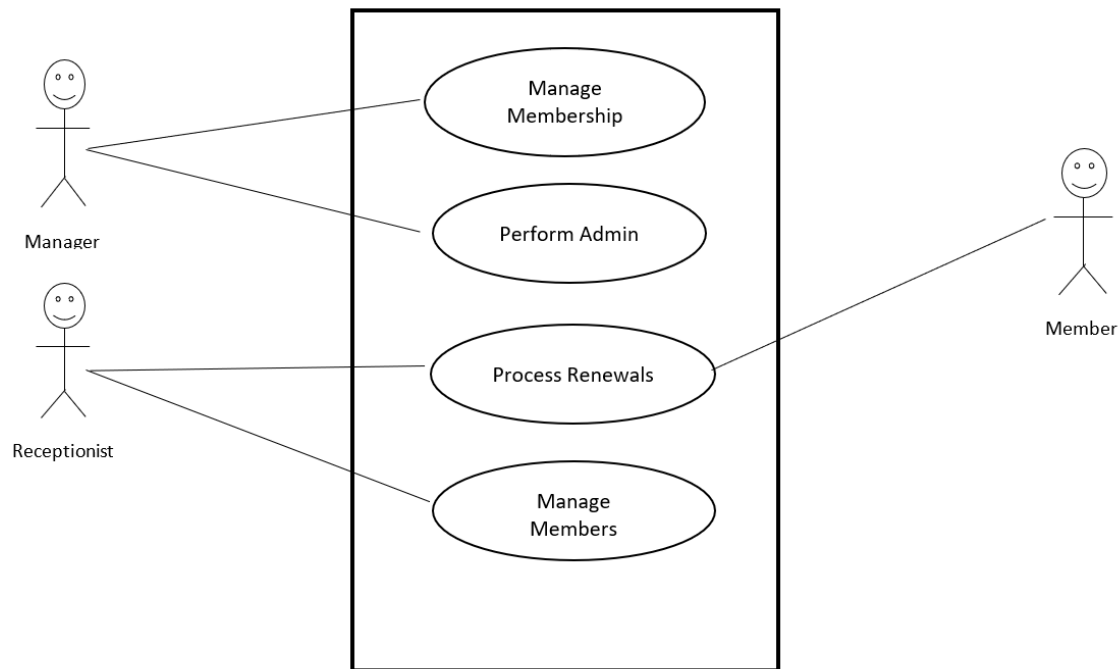


Figure 2.0 Existing Functional Requirements

Add Member - This function will add any individual who is interested in joining the gym on a monthly basis, all relevant details concerning the member will be added to the database.

Add Membership Type - All members are required to select their membership type of choice. These membership types will be allocated a price based on the respected duration of the membership.

Update Member - Membership types require regular updates to keep the Membership type table up to date. The update Membership type function will implement this!

Issue reminder - This function finds all members whose memberships are expiring in the next week and issues a reminder to these members via an email address provided.

2.2 Associations between requirements and use cases (including UML use case diagrams)

Add Member

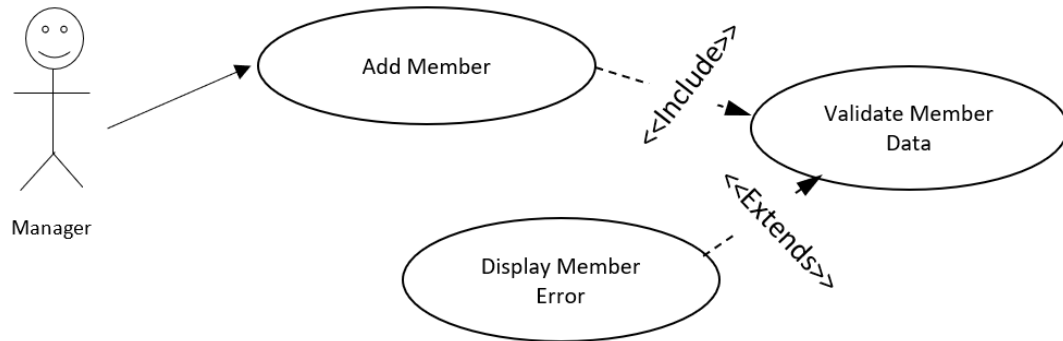


Figure 2.2.1 Add Member Functional Use Case Diagram

Add membership type

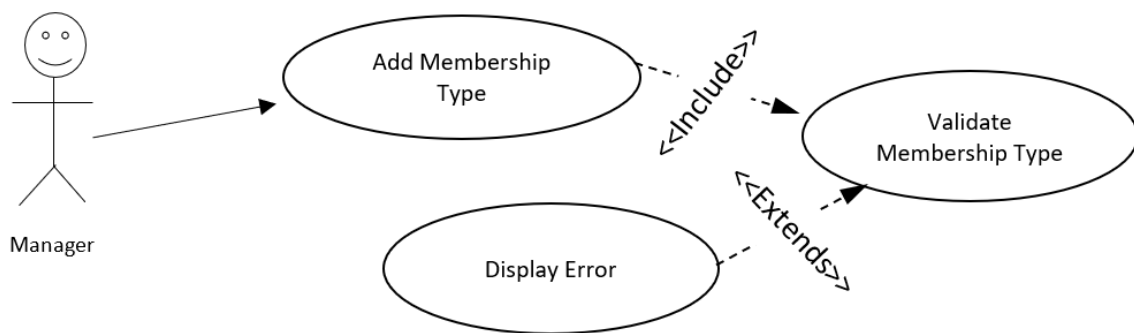


Figure 2.2.2 Add Membership Type Functional Use Case Diagram

Update Membership Type

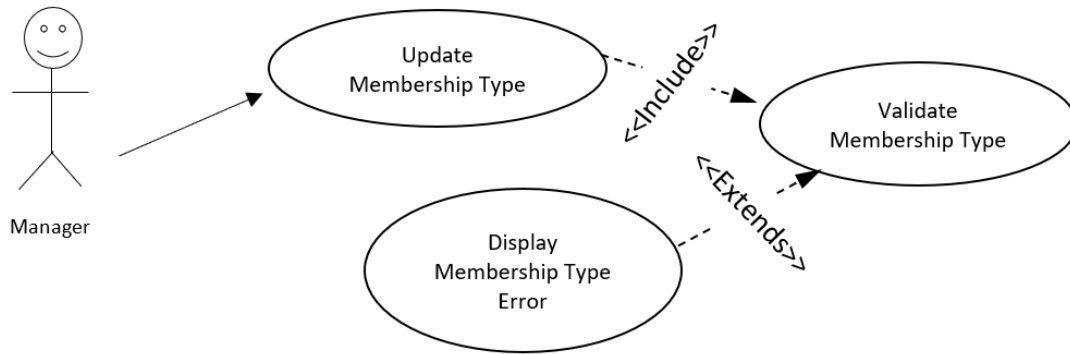


Figure 2.2.3 Update Membership Type Functional Use Case Diagram

Issue reminder

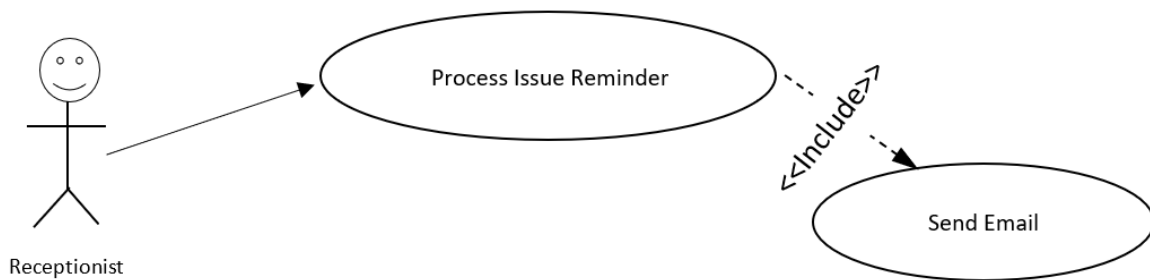


Figure 2.2.4 Issue Reminder Functional Use Case Diagram

2.3 Non-Functional Requirements

User Experience and Usability – The Gym Membership System must be easily understood from the client's point of view. This should be achieved by the integration of a user-friendly interface with clear navigation controls. Adding members for example, should be straight-forward and easily implemented. Consequently, the application should support individuals with disabilities by ensuring the relevant accessibility needed is adhered to.

Security and Privacy – Securing confidential user data is a paramount requirement in managing user data. All the customer information and payment details must conform to GDPR standards and the necessary payment security standards within the distributed region.

Performance - High numbers of concurrent users and transactions should be supported by the system without any performance issues presented.

Availability - The information provided by the application to management and clients should be always available with minimal downtime or disruption. This should be heavily considered when choosing a database to store the information. As an example, CouchDB is an excellent choice due to the consistency and constant availability of the data, even in the event of a server failure.

2.4 Associations between Non-functional requirements and use cases

Usability

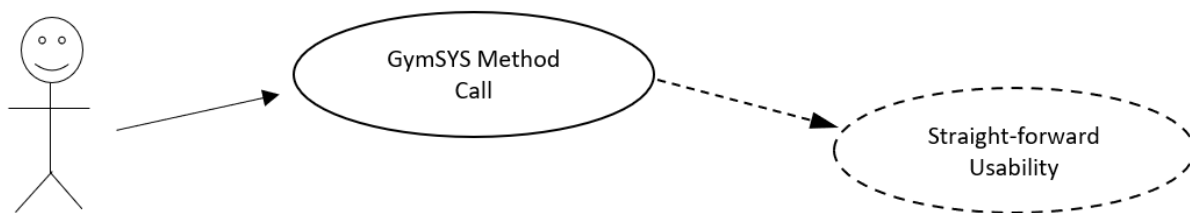


Figure 2.4.1 Usability Non-Functional Use Case Diagram

Security and Privacy

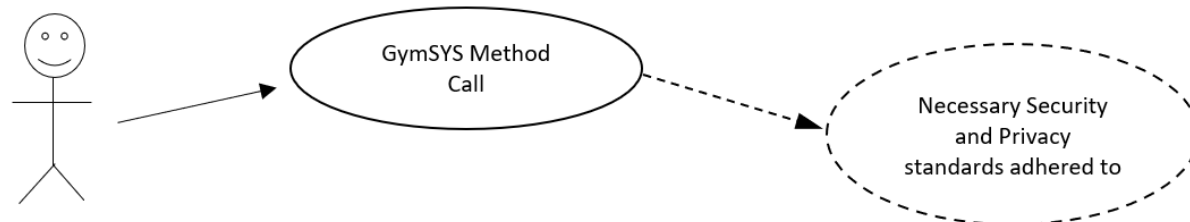


Figure 2.4.2 Security and Privacy Non-Functional Use Case Diagram

Performance

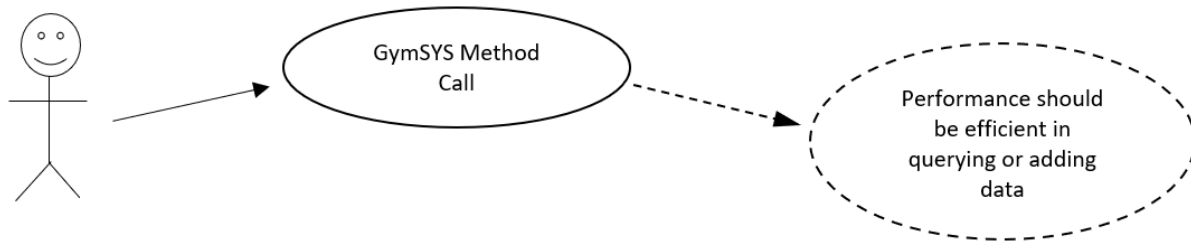


Figure 2.4.3 Performance Non-Functional Use Case Diagram

Availability

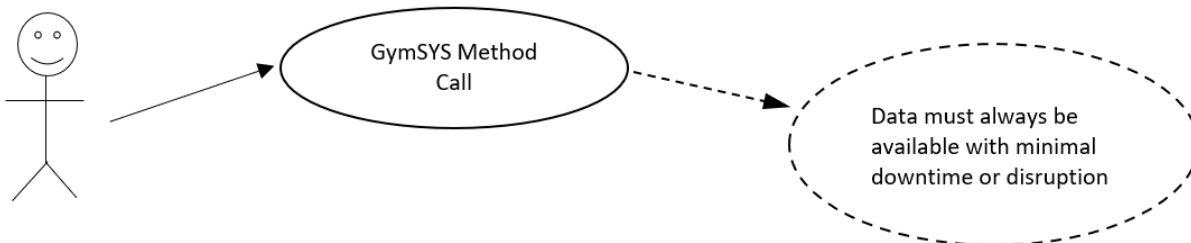


Figure 2.4.4 Availability Non-Functional Use Case Diagram

3 Design Rationale

3.1 Pros And Cons of Layered Architecture Pattern

Pros

- Allowing a separation of concerns approach promotes testability of each individual section (Freeman, et al., 2004)
- The software system as a whole becomes much more stable, which positively affects the maintainability of the application (Freeman, et al., 2004).
- Splitting the application into cohesive responsibilities allows for easy extensibility
- Troubleshooting is far more efficient as the root cause can be identified within each layer rather than from a 'God Class' with tightly coupled components (Gamma, et al., 1994)

- As the application is more isolated from other layers of the application, there is limited dependency as each layer functions separately from the other layers
- Implementing and learning this pattern is very easy and straightforward

Cons

- As an application scales, this can directly affect the performance of the application due to requests having to navigate through multiple layers before a response is processed and reverted to the client
- Using multiple processors (Parallel Processing) is not possible which affects the performance of the application. The effects will be more prominent the more the project scales (Freeman, et al., 2004)
- The maintainability can be impacted as when alterations are made, this can affect the other layers within the application

3.2 Pros And Cons of Adapter Pattern

Pros

- Single Responsibility Principle is adhered to as the data logic is separated from the business logic (Gamma, et al., 1994)
- The Open Closed Principle is adhered to as when a new adapter is implemented, there is no need to modify existing code, rather extending the Class is possible (McLaughlin, et al., 2006).
- Integrating incompatible interfaces within the software system is achieved which allows for an easy implementation of new functionalities or modification of existing components (Freeman, et al., 2004)
- Reduced duplication of code as the amount of dependencies and interconnections between different components of the system are reduced through abstraction of a class or object (Gamma, et al., 1994).
- Maintainability is enhanced as the implementation of various components is decoupled from the rest of the system. This also promotes the reusability of code and components.

Cons

- The complexity of the code is increased as new interfaces and classes need to be implemented to conform with the pattern.
- There is a limit on the flexibility and the adaptability of the system as it relies on a fixed interface with a set of conditions for the classes being adapted (Freeman, et al., 2004). If these classes change, the adapter may need to be modified which could result in a tedious refactoring process which is susceptible to human error.
- In certain situations, there could be a loss in functionality due to the adapted object not supporting all the features and operations as the original (Freeman, et al., 2004).

3.3 Pros And Cons of Command Pattern

Pros

- The open closed principle is adhered to as new functionality can be implemented without breaking existing client code (refactoring.guru, 2022). An example of this is if an engineer wants to add a welcoming email when users sign up to the gym. In the beginning, this would mean modifying a tightly coupled class to add functionality. Rather, with the command pattern the extensibility is achieved easily through extending a class.
- Reversible operations can be implemented as the command history keep a stack of all executed commands object with back-ups for the state of the application (Freeman, et al., 2004).
- You can easily queue operations and schedule execution which is very beneficial for scheduling reminders in the gym membership system
- The Single Responsibility Principle is adhered to as the classes become decoupled to classes which are solely responsible for the specific operation (refactoring.guru, 2022).
- A queue of commands can be executed chronologically which can be very useful for certain applications.

Cons

- The complexity of the program is greatly increased as several layers are added such as the invoker, receiver, and sender (refactoring.guru, 2022).
- As there are many classes working together, it is crucial to be meticulous in ensuring each class is implemented correctly.
- The maintenance can be increased as each individual command is concrete which leads to an increased volume of classes.

3.4 Pros And Cons of Chain Responsibilities Design Pattern

Pros

- Implementing the COR design pattern can greatly decouple your application which will enhance testability and maintainability (refactoring.guru, 2022)
- New handlers can be introduced without modifying existing code. Functionality can simply be added by extending a class which demonstrates the adherence to the Open/Closed Principle (refactoring.guru, 2022)
- The order of request handling can be defined to be executed in a defined order.
- The extensibility and scalability is greatly improved as new validation components can be added without affecting existing components (Freeman, et al., 2004). As a result, the monolithic validation process is avoided, and the scalability potential is enhanced.
- The validation process is broken down into smaller focused components rather than a monolithic approach which also enhances the maintainability of the system.

Cons

- As there are many interlinked components, the implementation and maintenance is easily susceptible to failure which leads to certain requests being unhandled (refactoring.guru, 2022).
- The can performance can be affected due to a long stack of traces (Freeman, et al., 2004)

4 Redesign and Refactor

4.1 Implementation of the Layered Architecture

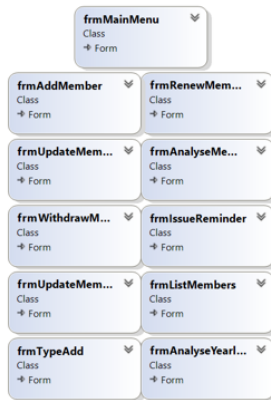
Description

The layered architecture pattern is utilised by dividing a software system into a set of layers, each serving a specific functionality for the system. Each layer communicates with the layer above it to exchange information or handle requests (Freeman, et al., 2004). As the initial structure of the Gym Membership System was tightly coupled, this pattern will isolate the functionality into the necessary classes to enhance the modularity, maintainability, and scalability of the system (Gamma, et al., 1994). Consequently, this leads to the system becoming decoupled and beginning to adhere to the Single Responsibility Principle.

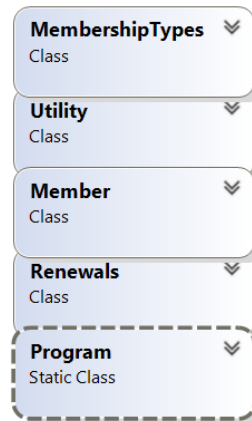
A common method of implementing the layered approach is the n-tier architecture (Microsoft, 2022). This method divides the system into three or more distinct layers including the presentation layer, the business layer, and the data access layer. The refactoring process of this has been illustrated in the next couple of sections. The presentation layer is responsible for presenting information to the client and handling any client prompts or interactions. In the context of the gym membership system, this is where all the forms to add a member, update a member etc will be refactored to. The business layer will be responsible for handling any logic or rules within the system. An example here would be handling a request to retrieve all members from the database, but no database query or data will be exposed. Finally, the data layer will manage access to the data and perform any necessary requests to retrieve data.

The layered architecture is well suited for applications with a high level of functionality and coupled components (Microsoft, 2022). For this reason, this pattern has been implemented to decouple the various components of the system to provide a more organised and manageable solution to promote the modularity, maintainability, and scalability of the Gym Membership System.

PRESENTATION LAYER



BUSINESS LAYER



DATA ACCESS LAYER

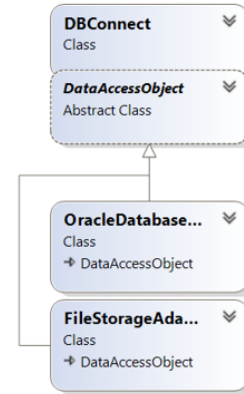


Figure 4.1.1 UML Diagram of the Layered Architecture

Code

The snippet shown below in Figure 4.1.2 is extracted directly from the existing structure before the n-tier architecture was applied. As shown by the red rectangle on the right-hand side of the screen, there was no existing structure in place. Additionally, all the logic was interconnected which impacted the maintainability of the code. The code is extracted directly from the members class whereby the blue rectangle shows the database logic hardcoded into the class. This violates OOP principles and results in an application that is very difficult to scale or extend (McLaughlin, et al., 2006).

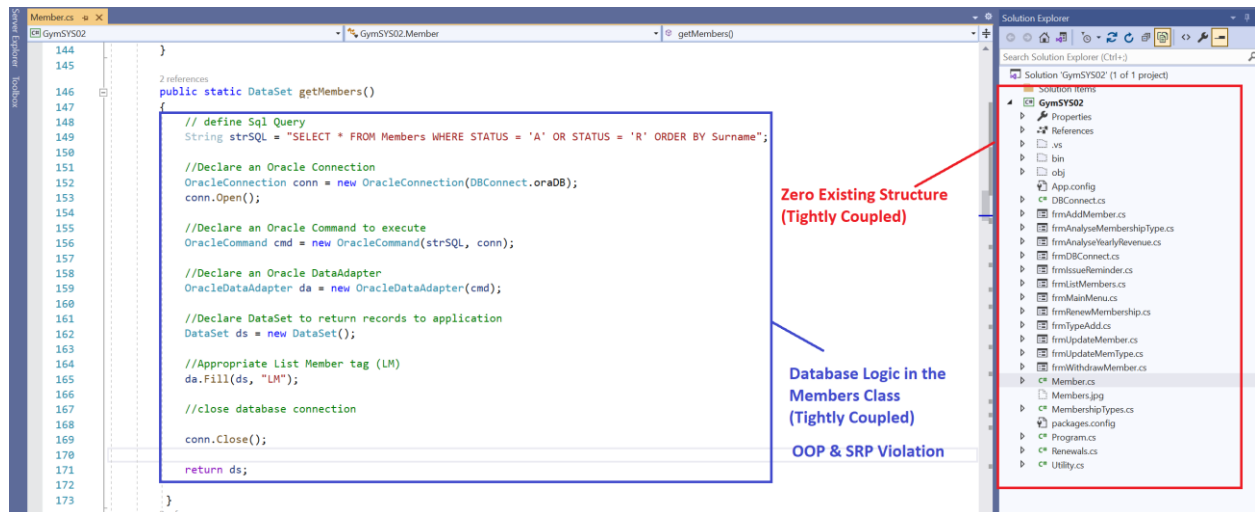


Figure 4.1.2 Existing Structure before Applying Layered Architecture

The solution to this issue begins with refactoring the existing code into separate classes each responsible for one purpose. In other words, each class must adhere to the Single Responsibility Principle. This principle states that “A class should have one and only one reason to change” (McLaughlin, et al., 2006).

Once the code has been refactored, an evaluation of the existing code must be performed to understand which components belong to which layers. All related functionalities should be added to the corresponding layer which contains a single, well-defined purpose such as the business layer, data layer or presentation layer.

The Microsoft “N-tier walkthrough” guide was closely followed in achieving the structure shown below (Microsoft, 2022). To summarise this procedure, two additional projects were created to adhere to the N-tier architecture. The BusinessLayer which is a “WCF Service” application and the DataLayer which is of type “Class Library”. The refactored layout to adhere to the layered architecture has been illustrated in Figure 4.1.3 below.

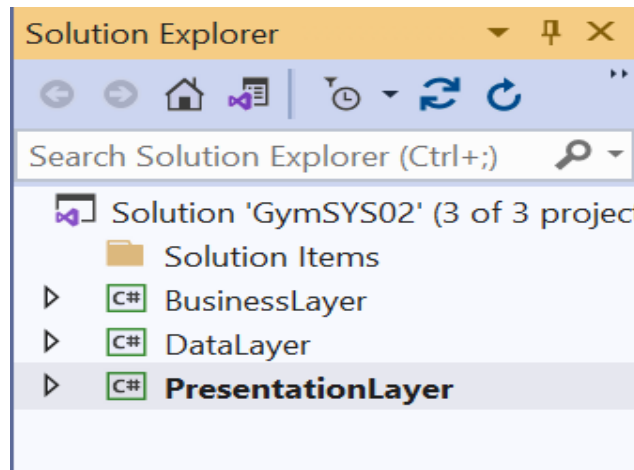


Figure 4.1.3 Refactored Layout to Adhere to Layered Architecture

All of the classes were then grouped into their relevant layer and refactored appropriately. The screen snippet in Figure 4.1.4 shown below illustrates the various classes that were refactored which can also be identified by the UML diagram in Figure 4.1.1 above.

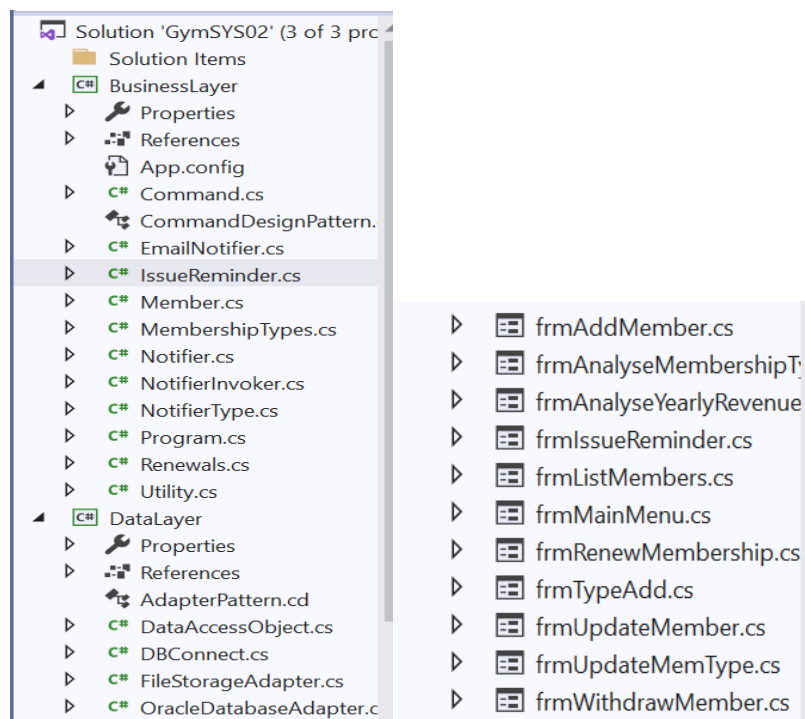


Figure 4.1.4 Various Classes Refactored to Appropriate Project

An integral part of linking the various layers involves the need to reference each layer which communicates together. Only the layers above one another communicate together. For example,

the Presentation Layer communicates directly to the Business Layer, but should know nothing about the Data Layer. Only the Business Layer knows about the Data Layer. The necessary references needed between the layers have been illustrated in Figure 4.1.5 and 4.1.6 below.

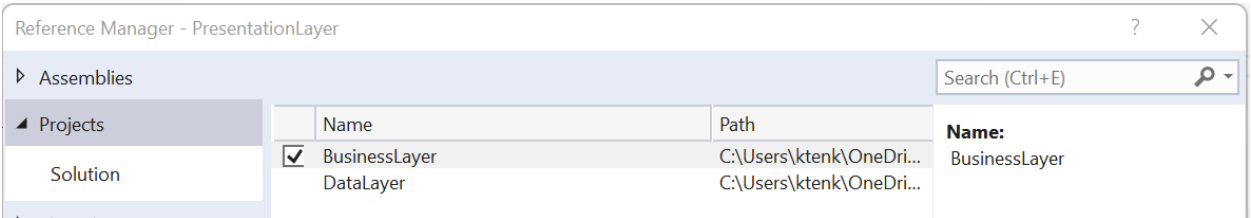


Figure 4.1.5 Reference between Presentation Layer and Business Layer

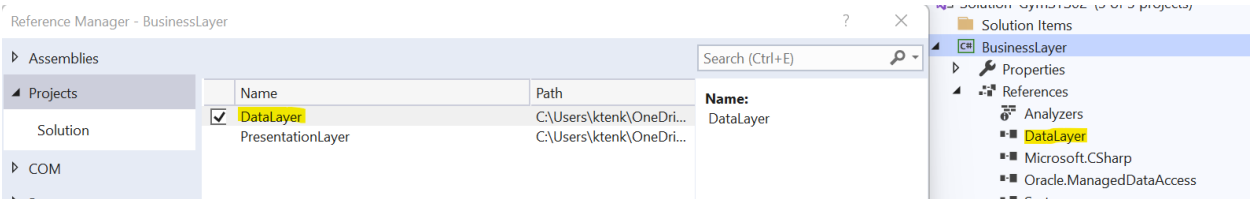


Figure 4.1.6 Reference between Business Layer and Data Layer

An example of a subsection of implementation process has been shown below. Line 10 is of critical importance as this code is paramount to enable the connection between the Presentation Layer and the Business Layer. As a prerequisite, it is essential to create a reference between the two layers before inserting this code, otherwise the “C# interpreter” will not be able to identify this layer.

The function of the code below in Figure 4.1.7 is to List all the Members in the Gym Membership System. The integration of the Layered Pattern occurs on lines 18 and 27. On line 18, there is a reference to the Member class within the Business Layer. Here, an instance of the class name ‘aMember’ is instantiated which allows access to all the methods within the class. Line 27 utilises one particular method of the class classed getMembers() which will retrieve a Data Set object once the data has completed the cycle through the layered architecture.

```

10  using BusinessLayer;
11
12  namespace PresentationLayer
13  {
14      public partial class frmListMembers : Form
15      {
16          //instance variable to remember where to go BACK to
17          frmMainMenu parent;
18          Member aMember = new Member();
19          public frmListMembers()
20          {
21              InitializeComponent();
22          }
23
24          private void ListMembers_Load(object sender, EventArgs e)
25          {
26              // retrieve all members
27              grdListMembers.DataSource = aMember.getMembers().Tables["LM"];
28              //Display the retrieved records
29              grdListMembers.Visible = true;
30          }
31      }

```

Figure 4.1.7 List all Members registered in the Gym Membership System

In continuation, the Members class in the Business Layer contains the 'getMembers()' function which was requested by the Presentation Layer. As the business layer is only in charge of logic, this request is then sent to the Data Layer to retrieve the necessary dataset. This is illustrated below in Figure 4.1.8.

```

2 references
public DataSet getMembers()
{
    DataSet ds = Program.DAO.getMembers();

    return ds;
}

```

Figure 4.1.8 Business Layer 'getMembers()' function

The code below in Figure 4.1.9 is a subsection of the methods contained in the 'DataAccessObject'. For illustration purposes, the 'getMembers()' method has only been shown. This method is of an abstract type which indicates that it must be implemented by the inherited class.

```

4 namespace DataLayer
5 {
6     3 references
7     public abstract class DataAccessObject
8     {
9         3 references
10        public abstract DataSet getMembers();

```

Figure 4.1.9 The 'getMembers()' function within the 'DataAccessObject'

Generally, the Data Access Object will contain all the method implementation code. This is often due to implementing a singleton design approach (Gamma, et al., 1994). In the instance of this implementation, an adapter pattern was used which will be touched on in the next section. In terms of the n-tier architecture, the code below demonstrates the final reference point to retrieve the necessary data set. The code in Figure 4.1.10 below retrieves all members from an oracle data source and returns the data of type "DataSet" on line 35 back to the DAO, Business Layer, and finally to the presentation layer where the client will be able to view a list of all members.

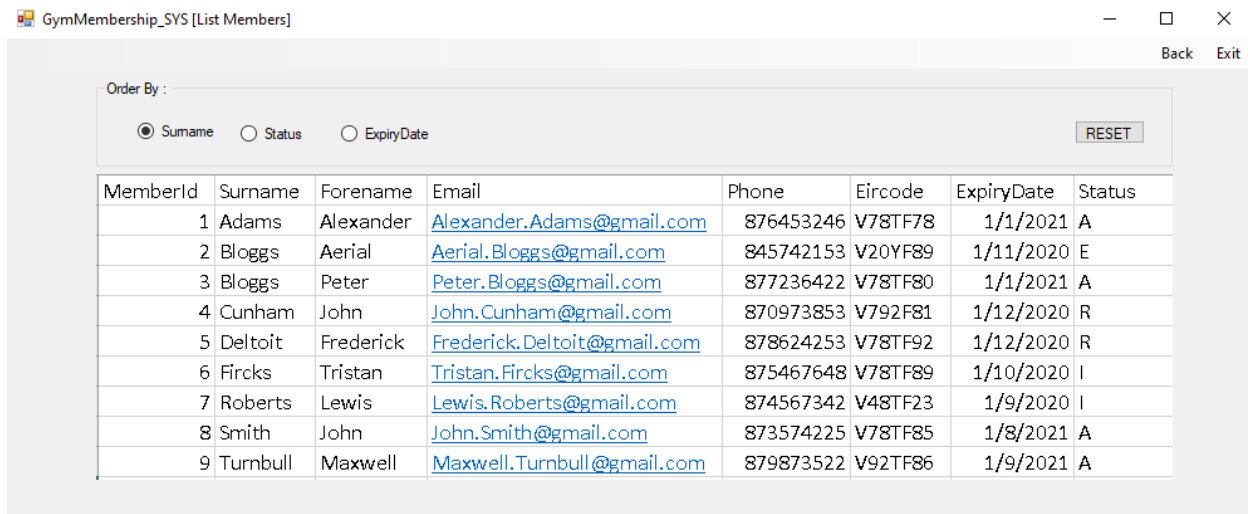
```

5 namespace DataLayer
6 {
7     1 reference
8     public class OracleDatabaseAdapter : DataAccessObject
9     {
10        2 references
11        public override DataSet getMembers()
12        {
13            // define Sql Query
14            String strSQL = "SELECT * FROM Members WHERE STATUS = 'A' OR STATUS = 'R' ORDER BY Surname";
15
16            //Declare an Oracle Connection
17            OracleConnection conn = new OracleConnection(DBConnect.oraDB);
18            conn.Open();
19
20            //Declare an Oracle Command to execute
21            OracleCommand cmd = new OracleCommand(strSQL, conn);
22
23            //Declare an Oracle DataAdapter
24            OracleDataAdapter da = new OracleDataAdapter(cmd);
25
26            //Declare DataSet to return records to application
27            DataSet ds = new DataSet();
28
29            //Appropriate List Member tag (LM)
30            da.Fill(ds, "LM");
31
32            //close database connection
33            conn.Close();
34
35            return ds;
36        }
37    }

```

Figure 4.1.10 Retrieval of all members using the 'OracleDatabaseAdapter'

Finally, once the request to List All Members has been processed by the necessary layers, the results will be presented to the client as shown in Figure 4.1.11 below.



Order By : ☒ Surname ☐ Status ☐ ExpiryDate RESET

MemberId	Surname	Forename	Email	Phone	Eircode	ExpiryDate	Status
1	Adams	Alexander	Alexander.Adams@gmail.com	876453246	V78TF78	1/1/2021	A
2	Bloggs	Aerial	Aerial.Bloggs@gmail.com	845742153	V20YF89	1/11/2020	E
3	Bloggs	Peter	Peter.Bloggs@gmail.com	877236422	V78TF80	1/1/2021	A
4	Cunham	John	John.Cunham@gmail.com	870973853	V792F81	1/12/2020	R
5	Deltoit	Frederick	Frederick.Deltoit@gmail.com	878624253	V78TF92	1/12/2020	R
6	Fircks	Tristan	Tristan.Fircks@gmail.com	875467648	V78TF89	1/10/2020	I
7	Roberts	Lewis	Lewis.Roberts@gmail.com	874567342	V48TF23	1/9/2020	I
8	Smith	John	John.Smith@gmail.com	873574225	V78TF85	1/8/2021	A
9	Turnbull	Maxwell	Maxwell.Turnbull@gmail.com	879873522	V92TF86	1/9/2021	A

Figure 4.1.11 A list of all members being returned to the client

4.2 Implementation of Adapter Pattern

Description

The adapter pattern is a structural design pattern which allows the functionality to combine incompatible interfaces. This pattern acts as middle layer class which interacts between two components (Freeman, et al., 2004). As a result, the data code is separated from the business logic of the program which adheres to the Single Responsibility Principle. In the context of the Gym Membership System, all of the classes are tightly coupled to the database code. In the instance of changing a database or extending the functionality to allow for file storage, this would mean visiting every single piece of code and adjusting it to allow for this change. This violates the Open-Closed principle which is very poor coding practice (McLaughlin, et al., 2006). Rather, the program should be open for extension without changing any code. By refactoring the program to the adapter pattern, this would mean that the existing code and the 'adaptee' code does not change, rather the 'Adapter' class contains the new code to allow the incompatible interfaces to connect. The adapter class in this example can be the fileStorageAdapter.

The UML diagram below shows the structure of this design pattern. As shown both the `OracleDatabaseAdapter` and the `FileStorageAdapter` implement the `DataAccessObject`'s methods. In the event of extending the functionality of the system to allow for a CouchDB database implementation, this can easily be implemented by extending the `DataAccessObject` class without having to rewrite the data access code. Consequently, in the data access object, dependency injection can be integrated to allow for different database systems or storage systems without having to modify existing code in the `DataAccessObject`.

UML

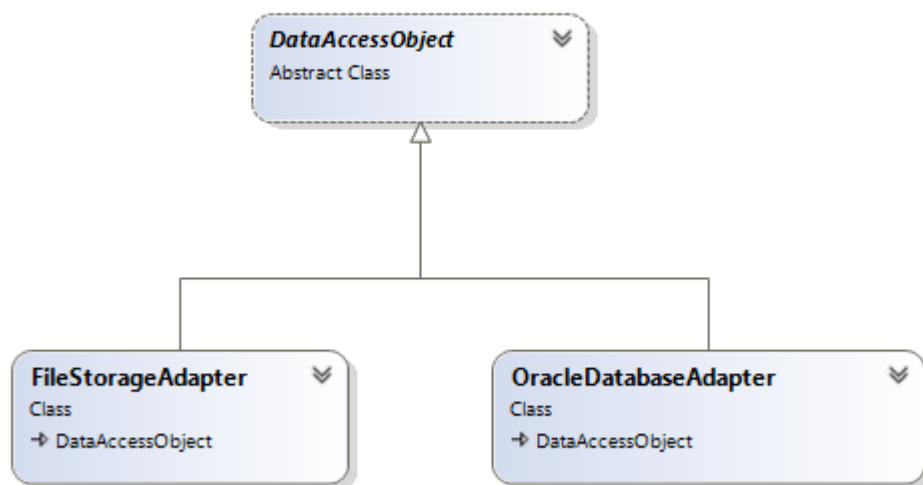


Figure 4.2.1 UML diagram of the implemented Adapter Pattern

Code

As illustrated in figure 4.2.2 below, the two methods shown below were hard coded into the "Members" class prior to implementing the adapter pattern. If management decided to switch databases, this would mean adjusting all of the code within the class to work with the new database. This violates the Open Closed Principle as in order to change or add functionality, the existing code will need to be modified. As every method call is hardcoded with Oracle syntax, the

refactoring process of changing everything to suit MySQL for example would have a huge impact on the organisation.

```
1 reference
public void addMember()
{
    //define sql query
    String strSQL = "INSERT INTO members VALUES (" + this.memberid + "," + this.typecode + "," + this.surname + "," + this.forename + "," + this.email + "," + this.phone + "," + this.eircode + "," + this.expDate + "," + this.status + ")";

    //Declare an Oracle Connection
    OracleConnection conn = new OracleConnection(DBConnect.oraDB);
    conn.Open();

    //declare an Oracle Command to execute
    OracleCommand cmd = new OracleCommand(strSQL, conn);

    cmd.ExecuteNonQuery();

    conn.Close();
}

3 references
public void getMember(int memberid)
{
    //define Sql Query
    String strSQL = "SELECT * FROM Members WHERE memberId = " + memberid;
    //Declare an Oracle Connection
    OracleConnection conn = new OracleConnection(DBConnect.oraDB);
    conn.Open();
    //declare an Oracle Command to execute
    OracleCommand cmd = new OracleCommand(strSQL, conn);
    OracleDataReader dr = cmd.ExecuteReader();
}
```

Figure 4.2.2 Hardcoded Database logic within Members class

Conversely, applying the adapter design pattern to solve this issue will allow for more modular, reusable, and maintainable code (Gamma, et al., 1994). In the event that management decides to switch databases, this process will be very easy without the need to rewrite the Data Access Object. Simply creating a new database implementation class will allow for this incompatible class to be easily integrated into the system.

Figure 4.2.3 below demonstrates the abstract methods which need to be implemented by the incompatible classes.


```

1  using System;
2  using System.Data;
3
4  namespace DataLayer
5  {
6      public abstract class DataAccessObject
7      {
8          public abstract DataSet getMembers();
9
10         public abstract int getNextId();
11
12         public abstract Boolean typeCodeExists(String typeCode);
13
14         public abstract DataTable fillChart(int year);
15
16         public abstract DataTable fillChartRenewals(int year);

```

Figure 4.2.3 Abstract Methods defined within DataAccessObject

Figures 4.2.4 and 4.2.5 below show a subsection of the Oracle Database Adapter code. This class 'implements' the Data Access Object along with all the methods through the 'override' keyword. Every single method listed in the Data Access Object will need to be implemented by this class. For this reason, this allows for consistency and integrity amongst the various adapters (Gamma, et al., 1994). Consequently, only the code relating to the Oracle Database will be included in the class. This is an example of the Single Responsibility Principle as the purpose is solely to retrieve data from one database provider.

```

1  using Oracle.ManagedDataAccess.Client;
2  using System;
3  using System.Data;
4
5  namespace DataLayer
6  {
7      1 reference
8      public class OracleDatabaseAdapter : DataAccessObject
9      {
10         2 references
11         public override DataSet getMembers()
12         {
13             // define Sql Query
14             String strSQL = "SELECT * FROM Members WHERE STATUS = 'A' OR STATUS = 'R' ORDER BY Surname";
15
16             //Declare an Oracle Connection
17             OracleConnection conn = new OracleConnection(DBConnect.oraDB);
18             conn.Open();
19
20             //Declare an Oracle Command to execute
21             OracleCommand cmd = new OracleCommand(strSQL, conn);
22
23             //Declare an Oracle DataAdapter
24             OracleDataAdapter da = new OracleDataAdapter(cmd);
25
26             //Declare DataSet to return records to application
27             DataSet ds = new DataSet();
28
29             //Appropriate List Member tag (LM)
30             da.Fill(ds, "LM");
31
32             //close database connection
33
34             conn.Close();
35
36             return ds;
37         }
38         1 reference
39         public override int getNextId()
40         {
41             int nextId = 0;
42
43             //Define SQL query to retrieve the last id assigned
44             String strSQL = "SELECT MAX(memberid) FROM members";
45
46             //Connect to the database
47             OracleConnection conn = new OracleConnection(DBConnect.oraDB);
48             conn.Open();
49
50             //define an oracle command
51             OracleCommand cmd = new OracleCommand(strSQL, conn);

```

Figure 4.2.4 OracleDatabaseAdapter Method implementation

```

1  using Oracle.ManagedDataAccess.Client;
2  using System;
3  using System.Data;
4
5  namespace DataLayer
6  {
7      1 reference
8      public class OracleDatabaseAdapter : DataAccessObject
9      {
10         2 references
11         public override DataSet getMembers()...
12         1 reference
13         public override int getNextId()...
14
15         2 references
16         public override bool typeCodeExists(string typeCode)...
17
18         2 references
19         public override DataTable fillChart(int year)...
20
21         2 references
22         public override DataTable fillChartRenewals(int year)...

```

Figure 4.2.5 4 OracleDatabaseAdapter Method implementation outline

4.3 Implementation of Command Design Pattern

Description

The command design pattern allows the necessary functionality to encapsulate an object which can be scheduled to be executed at a later time. The pattern stores a history of the executed commands which can be very useful for undo or redo functionality (refactoring.guru, 2022). In addition, parallel execution of operations is possible which greatly increases the performance of the system (refactoring.guru, 2022).

In the context of the Gym Membership System, this pattern can be integrated to process the reminders for expiring members. The system finds all members which have a membership within the week and issues one personalised reminder to each of these individuals, notifying them that their membership is expiring. The command pattern can be integrated in this process to schedule these reminders to be executed in the future. This will be achieved through a concrete implementation of an interface called 'IssueReminder' for example. Consequently, as this pattern adheres to SRP and OCP, the extension to allow for additional functionality is seamless. If management wishes to send a welcoming reminder to new members, this functionality can easily be integrated by creating a concrete implementation of the necessary interface. As this command will be performed in the future, the item will be encapsulated and scheduled to

execute when a new member signs up. As a result, this leads to a more flexible and maintainable code base allowing for seamless extensibility (Gamma, et al., 1994).

As seen in the UML diagram below the structure there is the main notifier interface which contains all the commands. The 'NotifierType' specifies which type of notifier to use i.e. 'EmailNotifier'. With this approach, it allows for a different form of notifying medium. The 'NotifierInvoker' which is responsible for instantiating the Command and calling the execute method of the instantiated object. The main receiver object which is called 'EmailNotifier' where the code for all the commands goes. The command interface which contains the execute and the undo method. The concrete implementation called IssueReminder with dependency injection is applied to promote flexibility.

UML

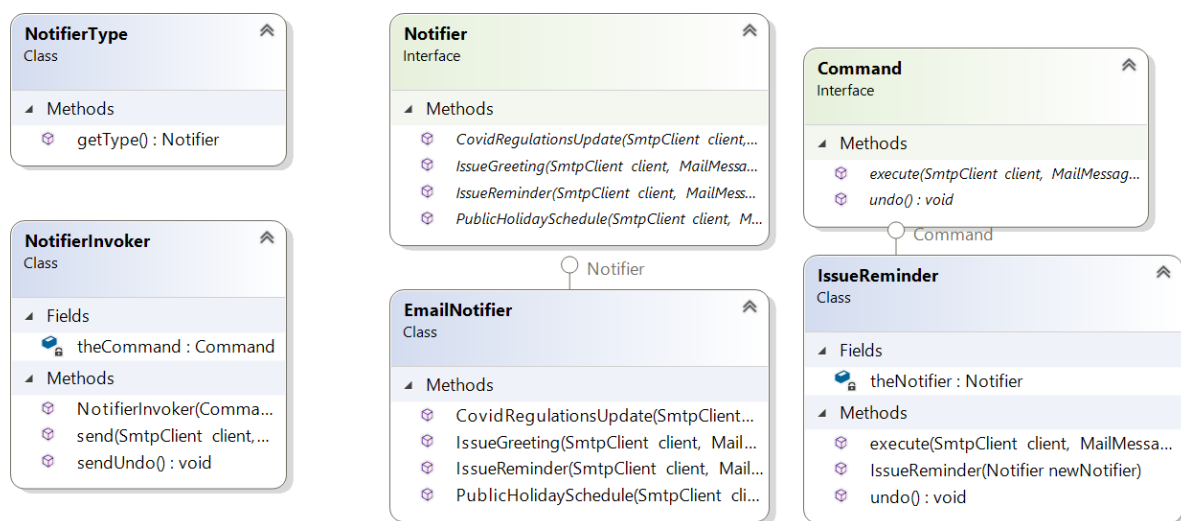


Figure 4.3.1 Command Design Pattern UML Diagram

Code

The first code snippet in Figure 4.3.2 contains the interface called `Notifier`. Here all the commands are specified which can be executed in the future. Each of these methods will have a concrete implementation command which defines the process of execution. In the event that additional

notifications need to be sent to clients, they can be inserted here which will cause all classes which inherit the interface to implement the commands which will be seen in Figure 4.3.3.

```
1 using System.Net.Mail;
2
3 namespace BusinessLayer
4 {
5     5 references
6     public interface Notifier
7     {
8         2 references
9         void IssueReminder(SmtpClient client, MailMessage message);
10
11         1 reference
12         void PublicHolidaySchedule(SmtpClient client, MailMessage message);
13
14         1 reference
15         void CovidRegulationsUpdate(SmtpClient client, MailMessage message);
16
17         1 reference
18         void IssueGreeting(SmtpClient client, MailMessage message);
19     }
20 }
```

Figure 4.3.2 The Notifier Interface with all predefined methods

The next snippet shows the receiver class called EmailNotifier. A huge benefit of this receiver class is the ability to decouple the client operation from the object performing the operation which is the receiver. This allows for the option to include different commands with the client code and have the ability to replace the receiver without affecting the client. For demonstration purposes, the only method implemented within the EmailNotifier class is the 'IssueReminder()' method where an email is sent to all expiring members. All other methods have been issued with "NotImplementedException()" as every method from the Notifier interface must be implemented.

```

1  //The Receiver of the Command Design Pattern
2  using System;
3  using System.Net.Mail;
4
5
6  namespace BusinessLayer
7  {
8      1 reference
8      public class EmailNotifier : Notifier
9      {
10         1 reference
10         public void CovidRegulationsUpdate(SmtpClient client, MailMessage message)
11         {
12             throw new NotImplementedException();
13         }
14
15         1 reference
15         public void IssueGreeting(SmtpClient client, MailMessage message)
16         {
17             throw new NotImplementedException();
18         }
19
20         2 references
20         public void IssueReminder(SmtpClient client, MailMessage message)
21         {
22             //Send the necessary email
23             client.Send(message);
24         }
25
26         1 reference
26         public void PublicHolidaySchedule(SmtpClient client, MailMessage message)
27         {
28             throw new NotImplementedException();
29         }
30     }
31 }
32

```

Figure 4.3.3 EmailNotifier class implementing the Notifier Interface

The NotifierInvoker shown in Figure 4.3.4 makes use of dependency injection which demonstrates the flexibility and extensibility of the pattern. Rather than instantiating the command within the 'send()' method, this is performed between lines 18-23. The code on line 27 will then result in the 'IssueReminder()' command of EmailNotifier() being called.

```

14 namespace BusinessLayer
15 {
16     3 references
17     public class NotifierInvoker
18     {
19         Command theCommand;
20
21         1 reference
22         public NotifierInvoker(Command newCommand)
23         {
24             theCommand = newCommand;
25         }
26
27         1 reference
28         public void send(SmtpClient client, MailMessage message)
29         {
30             theCommand.execute(client, message);
31         }
32
33         0 references
34         public void sendUndo()
35         {
36             theCommand.undo();
37         }
38     }
39 }

```

Figure 4.3.4 The NotifierInvoker Class

The notifier type shown in Figure 4.3.5 is the first command that is instantiated at run time which basically returns the type of notifier which will be used. As only one notifier has been created called 'EmailNotifier', this class will be instantiated at run time.

```

7 namespace BusinessLayer
8 {
9     1 reference
10    public class NotifierType
11    {
12        1 reference
13        public static Notifier getType()
14        {
15            return new EmailNotifier();
16        }
17    }
18 }

```

Figure 4.3.5 The NotifierType Class which returns the type of notifier

The command interface in Figure 4.3.6 contains two important methods which must be implemented by all classes which inherit from the interface. The execute command will be where all the emails are sent with a defined 'SmtpClient' which is the receiver of the email and the message of type 'MailMessage' which is the content of the email. The undo method is then also

included which is a key component of the command pattern which can closely reference a queue or stack and reverse certain operations performed.

```

8  namespace BusinessLayer
9  {
10     3 references
11     public interface Command
12     {
13         2 references
14         void execute(SmtpClient client, MailMessage message);
15
16         2 references
17         void undo();
18     }
19 }

```

Figure 4.3.6 The Command Interface

The IssueReminder class in Figure 4.3.7 shown below implements the Command pattern. This is a concrete implementation whereby dependency injection is applied to instantiate the Notifier and execute the IssueReminder() method of the instantiated object.

```

~
9  namespace BusinessLayer
10 {
11     3 references
12     public class IssueReminder : Command
13     {
14         Notifier theNotifier;
15
16         1 reference
17         public IssueReminder(Notifier newNotifier)
18         {
19             theNotifier = newNotifier;
20         }
21
22         2 references
23         public void execute(SmtpClient client, MailMessage message)
24         {
25             theNotifier.IssueReminder(client, message);
26         }
27
28         2 references
29         public void undo()
30         {
31             throw new NotImplementedException();
32         }
33     }
34 }

```

Figure 4.3.7 The IssueReminder class Implementing the Command Interface

Finally, Figure 4.3.8 and 4.3.9 illustrate the request coming from the presentation layer which the client interacts with directly. This is where the Notifier type is first instantiated whereby the type returned will be the EmailNotifier. Secondly the IssueReminder class is instantiated to include the

notifierType. Then NotifierInvoker is instantiated with the name onSend which has access to the send() command. This 'onSend' instance will then take two parameters, the Client and the Message to send to the client which will call a series of classes until it reaches the EmailNotifier, where the reminder will be sent to the individual.

The code provided to loop over all expiring members and tailor a message to the specified client address has also been included in the snippet below.

```

1 reference
private void btnSend_Click(object sender, EventArgs e)
{
    //get the notifierType to use
    Notifier newNotifier = NotifierType.getType();

    //IssueReminder contains the code to issue the reminder
    //When execute() is called, the method issueReminder()
    //in the EmailNotifier interface will be called

    IssueReminder reminderCommand = new IssueReminder(newNotifier);

    //Calling the invoker which will result in issueReminder() of EmailNotifier being called
    NotifierInvoker onSend = new NotifierInvoker(reminderCommand);

    Utility aUtility = new Utility();

    SmtpClient Client = aUtility.getSMTPClient();

```

Figure 4.3.8 The Client initiating the IssueReminder Command

```

//loop over every expiring member
for (int i = 0; i < grdListExpMems.Rows.Count - 1; i++)
{
    // if status is R, then don't send another reminder (member has already been issued a reminder)
    if (grdListExpMems.Rows[i].Cells[8].Value.ToString() != "R")
    {
        MailAddress ToEmail = new MailAddress(grdListExpMems.Rows[i].Cells[4].Value.ToString()); //retrieve the email address for expiring member
        MailMessage Message = new MailMessage() //create a custom message
        {
            From = FromEmail,
            Subject = "Expiring Gym Membership",
            //format body to display appropriate data for each member
            Body = "Dear " + grdListExpMems.Rows[i].Cells[2].Value.ToString() + " " + grdListExpMems.Rows[i].Cells[3].Value.ToString() +
                "\n\nA friendly reminder that your gym membership is about to expire on " + grdListExpMems.Rows[i].Cells[7].Value.ToString() +
                "\n\nTo renew your membership, please make your way to the front desk where your membership status for the year 2021" +
                " can be amended.\n\n" + "Current Member Details:  \n\nMember ID: " + grdListExpMems.Rows[i].Cells[0].Value.ToString() + "\nMembership Type: " +
                grdListExpMems.Rows[i].Cells[1].Value.ToString() + "\nStatus: " + grdListExpMems.Rows[i].Cells[8].Value.ToString() +
                "\n\nWe hope to welcome you back for the prosperous months that lie ahead!" + "\n\nThanks," + "\nGym Membership Team"
        };
        Message.To.Add(ToEmail);

        try
        {
            //send the formatted message above
            //Client.Send(Message);
            onSend.send(Client, Message);

            //Set status to R: Reminder Sent
            int Member = Convert.ToInt32(grdListExpMems.Rows[i].Cells[0].Value.ToString()); //retrieves memberid

            aMember.updateStatus(Member); //call updateStatus in member class

            //Show Confirmation Message
            MessageBox.Show("All email have successfully been sent", "Success!!");
        }
        catch { }
    }
}

```

Figure 4.3.9 The Code needed to Loop Over all Expiring Members and Issue a Custom Reminder Email

4.4 Implementation of Chain Of Responsibilities Pattern

Description

The Chain of Responsibilities design pattern is a behavioural pattern utilised through passing requests through a series of handlers (refactoring.guru, 2022). As the request progresses down the chain, each handler either processes the request or passes it to the next handler object in the chain. This approach leads to the decoupling of the request from the sender and receiver as each object in the chain can independently handle the request (Freeman, et al., 2004). The implementation of this pattern is achieved through the use of an interface which specifies the necessary methods that must be implemented by the handlers. This is particularly useful to integrate this pattern into a validation system as the request will be processed and validated by a chain of validators each handling a specific type of validation. Consequently, this will be in a defined order as the `'.setNext()'` method will be utilised to follow a chronological order of validation. Initially, the request will be passed to the first validator in the chain and continue to the end of the chain until the request has been handled. A huge advantage of implementing this pattern is that it allows new handlers to be introduced without modifying existing code (refactoring.guru, 2022). The necessary functionality can simply be added by extending a class. In addition, as each handler has one responsibility, the maintenance and testability of the system is positively impacted.

In the context of the Gym Membership System, this pattern can be used to validate the client input for each form. Initially all the validation steps would be added within each form even though the input fields were of the same type of validation. For example, when adding a member, you will need to validate the forename, email address, eircode etc. Moreover, this validation process is exactly the same when updating a member. As a result, when the validation is hardcoded within each form, this produces a large amount of duplicated code with limitations of extensibility as modifying the validation steps within the form could impact the other validation steps. By creating the various handlers illustrated in the UML diagram below (Figure 4.4.1), the code starts to adhere to OOP principles which results in a far more maintainable system.

UML

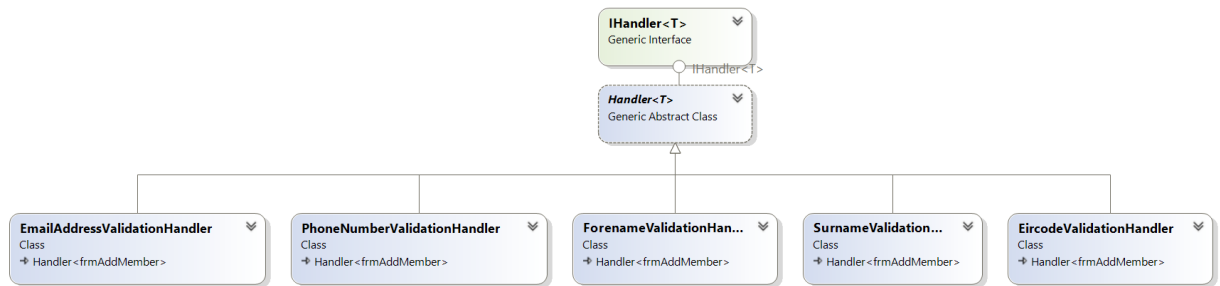


Figure 4.4.1 The UML Diagram for the Chain Of Responsibilities Design Pattern

Code

The initial structure of the validation before the design pattern was applied has been illustrated below in Figure 4.4.2. The code has been extracted from the “frmAddMember” on the left and the “frmUpdateMember” on the right. Notice several if statements are used to identify if the member’s details are valid. Additionally, the exact same validation steps are used in both forms, demonstrating the unnecessary duplication of code. Consequently, there is a large amount of logic being presented on the client side. This is poor coding practice as the maintainability of the code is negatively impacted, resulting in a difficult system to debug or test.

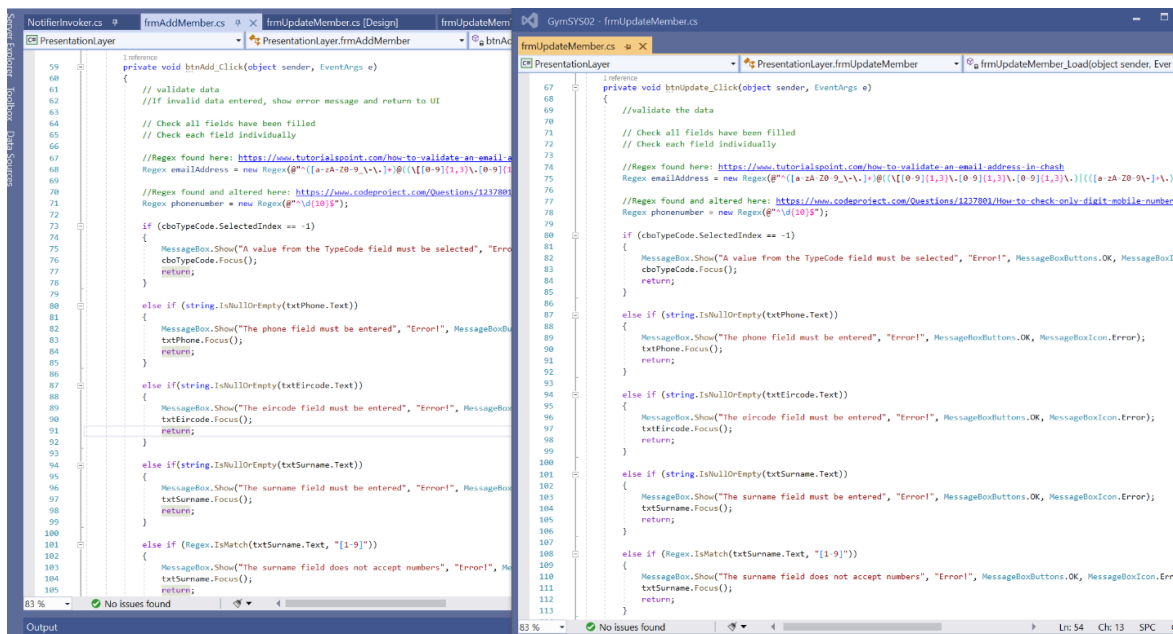


Figure 4.4.2 Initial Structure demonstrating the poor design with code duplication

In the event of trying to make a change to the logic in the code, this will need to be updated in multiple locations where the validation steps occur. As the validation was also spread across multiple classes, this leads to a very time-consuming and error-prone process.

The difficulties of validating these forms are resolved in a fraction of the code on the client side. As illustrated in the snippet below, the complex if statements have been eradicated and replaced with concise validation handlers for each field. This approach allows for a more flexible and extensible system by splitting the responsibility amongst several decoupled handlers which adhere to the Single Responsibility Principle.

```
try
{
    frmAddMember userInput = new frmAddMember();
    var handler = new EmailAddressValidationHandler();

    handler.setNext(new ForenameValidationHandler())
        .setNext(new SurnameValidationHandler())
        .setNext(new EircodeValidationHandler())
        .setNext(new PhoneNumberValidationHandler());

    handler.Handle(userInput);
}
catch
{
    throw new Exception("The input validation failed!");
}
```

Figure 4.4.3 The Client Initiating the Chain of Validation Handlers

The implementation of this pattern is straight-forward as it does not require any complex algorithms or data structures. Instead, the pattern utilises inheritance and polymorphism which makes it uncomplicated to integrate with existing code without the need for major changes of the code. As shown in the code snippet in Figure 4.4.4 below, an interface called 'IHandler' is added which takes a generic object T of type class which allows for the flexibility of validation in the context of the Gym Membership System. The benefits of having this handler only implement a class is that its open to extension. In the event of this only allowing a string, you will need to pass the specific input of each form item i.e "txtEircode" to the handler which will soon become difficult to maintain. The IHandler interface has two methods, setNext and Handle. The setNext method is simply in place to coordinate which handler processes the request next. On the other hand, the "Handle" method processes the request as a whole as seen in the code "handler.Handle(userInput)".

```
6 references
public interface IHandler<T> where T : class
{
    9 references
    IHandler<T> setNext(IHandler<T> next);
    9 references
    void Handle(T request);
}
```

Figure 4.4.4 The interface 'IHandler' of generic object type <T>

The Handler abstract class shown in Figure 4.4.5 then implements this method to coordinate the chain of responsibilities. As seen in the methods defined, each method utilises the 'Next' generic object which will be either handled or passed on to the next handler through the "setNext" method.

```

5 references
public abstract class Handler<T> : IHandler<T> where T : class
{
    3 references
    private IHandler<T> Next { get; set; }
    9 references
    public virtual void Handle(T request)
    {
        Next?.Handle(request);
    }

    9 references
    public IHandler<T> setNext(IHandler<T> next)
    {
        Next = next;

        return Next;
    }
}

```

Figure 4.4.5 The Handler abstract class implementing the IHandler interface

Finally, an example of one of the handlers has been shown below in Figure 4.4.6. This handler represents the “EmailAddressValidationHandler” which implements the Handler abstract class, while specifying the class to be validated. The validation steps utilise a regular expression and “.isEmpty” predefined method to validate whether the email contains valid characters and if any value has been included to start with. In the event that an exception is thrown, the system will issue the necessary error message to the client. If no exception occurs, the request will be passed to the next handler in the chain until it reaches the end of the chain whereby the request will be deemed valid.

```

2 references
public class EmailAddressValidationHandler : Handler<frmAddMember>
{
    5 references
    public override void Handle(frmAddMember request)
    {
        //Regex found here: https://www.tutorialspoint.com/how-to-validate-an-email-address-in-csharp
        Regex emailAddress = new Regex(@"^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.]|([a-zA-Z0-9\-\.]|([a-zA-Z]{2,4}|[0-9]{1,3}))(\.|\[?))$");

        if (string.IsNullOrEmpty(request.getEmail()))
        {
            throw new Exception("The email field must be entered");
        }

        else if (!emailAddress.IsMatch(request.getEmail()))
        {
            throw new Exception("Not a valid Email Address!");
            //MessageBox.Show("Not a valid Email Address!", "Error!", MessageBoxButtons.OK, MessageBoxIcon.Error);
        }
    }
}

```

Figure 4.4.6 The 'EmailAddressValidationHandler' implementing the Handler abstract class with the "frmAddMember" class type

5 Conclusions and Reflections

5.1 Conclusions

The existing structure of the Gym Management System was poorly designed and engineered. Although the system functionally performed all necessary components of the requirements specification. There was very little consideration that addressed the non-functional aspects of the system such as the security and privacy of sensitive information, or the consistent availability of information. This can highly be attributed to the initial structure not adhering to any Object-Oriented Programming principles. As a result, this created a very difficult system to manage or scale which heavily impacts the technical debt of the organisation (J. Holvitie, 2014).

Four software design patterns were integrated into this system to resolve the poor design of the system. These patterns included the Layered Architecture (N-tier) pattern, Adapter Pattern, Command Pattern and the Chain of Responsibilities pattern. This refactored approach of utilising these patterns proved to improve the quality by transitioning into a more modular, manageable, and scalable system (Crosby, 1979) (Deming, et al., 2012). Consequently, the integration of these patterns will improve the efficiency and productivity of the development team as there will be a logical structure in place which is open for extension and closed for modification.

Therefore, it is suggested that software design patterns can be integrated into existing systems to provide a more manageable and scalable system which lead to a higher quality product or service (Crosby, 1965). It is also suggested to integrate design patterns when necessary, as using too many patterns can increase the complexity of the system and result in performance bottlenecks. The common phrase to keep it simple (KISS) in software is often disregarded which can lead to an over-engineered system which is very difficult to maintain (McLaughlin, et al., 2006).

5.2 Reflections

Four software design patterns were integrated into the Gym Management System to address the poorly designed implementation.

The first pattern used was the Layered Architecture which decoupled the functionality of the system into 3 logical components called the Presentation Layer, Business Layer and Data Access Layer. These components mutually work together to create a more modular, manageable, and scalable system (Microsoft, 2022). This pattern was the most difficult to implement out of the three as there was an extensive refactoring process which was not supported in 'Visual Studio 2019'. As a result, the process was very manual and time consuming. However, once the refactoring process was complete, the system was more manageable and followed a logical structure which enhanced the non-functional components of the system including the security and availability.

The second pattern used was the adapter pattern which allowed for an incompatible class to integrate seamlessly into the system by extending the functionality of the adapter interface (Freeman, et al., 2004). The biggest advantage of implementing this pattern was to allow for seamless extensibility in either switching database providers or utilising a different storage medium (Gamma, et al., 1994). In the past the code needed to be refactored by revisiting every occurrence where the database syntax needed to be changed. However, this pattern adds an extra layer of abstraction which increases the complexity of the code. In the context of the Gym Membership System, the benefits of implementing this pattern greatly exceed the shortfalls.

The third pattern used was the command design pattern which encapsulated an object to be scheduled for execution at a later time (refactoring.guru, 2022). The main advantage which was seen after implementing this pattern was the decoupling of the operations. By decoupling these operations, the OCP and SRP were adhered to which promote scalability and testability (refactoring.guru, 2022). However, this pattern increases the underlying complexity in the code as there could be a large influx of small classes which clutters the codebase. Moreover, if there will only be a couple concrete implementations, the complexity required to refactor the existing solution can be disadvantageous. Therefore, it is recommended to integrate this pattern where there is a clear scope for the project, and when the sender request needs to be decoupled from the receiver. This results in a request being processed without details of how it was handled (Freeman, et al., 2004).

The fourth pattern used was the chain of responsibilities design pattern which is utilised by passing requests through a series of handlers (refactoring.guru, 2022). This pattern was integrated to solve the underlying issues of code duplication and maintenance issues. As the validation of user input was hardcoded and unified into each form through the use of nested if statements, this created an ideal use case for the chain of responsibility pattern (refactoring.guru, 2022). The process of refactoring the application to adhere to this pattern involved; designing an interface to handle the requests, creating a base class which provides a default implementation of the interface, and creating a concrete class for each validation handler. On the client side, the chain of responsibility will be established by creating references between the handlers using the "setNext()" method. The refactoring process performed proved to convert the initial design into a more maintainable and testable solution. However, as the stack will exponentially grow when new handlers are added, this can directly affect the performance. In the context of the Gym Membership System, it is suggested that the handlers will not exceed a threshold capable of putting the system under strain as the input validation for the forms will not change often. Consequently, this leads to a very effective pattern to implement for simple user validation.

To conclude the reflections of design patterns, it was observed that the integration of software patterns could greatly improve the quality of a software system if used where necessary. It is essential to carefully evaluate a piece of software to understand whether a pattern is needed. In

the exercises performed above, the existing structure was evaluated to understand its shortfalls and areas for possible improvement. All the design patterns implemented were meticulously selected based on drawbacks of the system, which demonstrated the effectiveness of these patterns.

6 References

Crosby, P., 1965. *Cutting the Cost of Quality*. Boston, MA, Industrial Education Institute.

Crosby, P., 1979. *Quality is Free*. New York, NY: McGraw-Hill.

Deming, W. E., Orsini, J. & Cahill, D. D., 2012. *The Essential Deming: Leadership Principles from the Father Of Quality*. New York: McGraw-Hill.

Freeman, E., Robson, E., Bates, B. & Sierra, K., 2004. *Head First Design Patterns*. 1st ed. California: O'Reilly Media, Inc..

Gamma, E., Vlissides, J., Johnson, R. & Helm, R., 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. MA, United States: Addison-Wesley.

HeydarNoori, A., 2003. *Software Quality*. [Online]
Available at: <https://cs.uwaterloo.ca/~apidduck/CS846/Seminars/abbas.pdf>
[Accessed 15 10 2022].

J. Holvitie, V. L. a. S. H., 2014. *Technical Debt and the Effect of Agile Software Development Practices on It - An Industry Practitioner Survey*, pp. 35-42.

McLaughlin, B., Pollice, G. & West, D., 2006. *Head First Object-Oriented Analysis & Design*. 1st ed. California: O'Reilly Media, Inc..

Microsoft, 2022. *Walkthrough: Create an n-tier data application*. [Online]
Available at: <https://learn.microsoft.com/en-us/visualstudio/data-tools/walkthrough-creating-an-n-tier-data-application?view=vs-2022&tabs=csharp>
[Accessed 15 11 2022].

refactoring.guru, 2022. *Chain of Responsibility*. [Online]
Available at: <https://refactoring.guru/design-patterns/chain-of-responsibility>
[Accessed 01 12 2022].

refactoring.guru, 2022. *Command Design Pattern*. [Online]
Available at: <https://refactoring.guru/design-patterns/command>
[Accessed 28 11 2022].

7 Appendices

7.1 Appendix A



7.2 Appendix B

