**Software Architecture Overview**

**Name:** Gerard O’Sullivan

**T-Number:** T00196432

**Date:** 27/10/2022

Contents

[1 Background Information 2](#_Toc121769352)

[1.1 Overview 2](#_Toc121769353)

[2 Existing structure 3](#_Toc121769354)

[2.1 Functional and Non-functional requirements 3](#_Toc121769355)

[2.2 Associations between requirements and use cases 4](#_Toc121769356)

[2.3 Interaction between components 5](#_Toc121769357)

[2.4 Existing structure design patterns. 12](#_Toc121769358)

[2.5 UML class diagram 14](#_Toc121769359)

[3 Design rationale 14](#_Toc121769360)

[3.1 Factory Method pattern rationale 14](#_Toc121769361)

[3.2 Singleton pattern rationale 15](#_Toc121769362)

[3.3 Proxy pattern rationale 15](#_Toc121769363)

[3.4 State pattern rationale 16](#_Toc121769364)

[3.5 Decorator pattern rationale 16](#_Toc121769365)

[4 Redesign and Refactor 17](#_Toc121769366)

[4.1 Refactoring 17](#_Toc121769367)

[4.2 Factory method pattern implementation 17](#_Toc121769368)

[4.3 Singleton pattern implementation 20](#_Toc121769369)

[4.4 Proxy pattern implementation 22](#_Toc121769370)

[4.5 Decorator pattern implementation 24](#_Toc121769371)

[4.6 State pattern implementation 26](#_Toc121769372)

[5 Conclusion and Reflection 27](#_Toc121769373)

[6 Code links 29](#_Toc121769374)

[References 29](#_Toc121769375)

# 1 Background Information

## Overview

Design patterns in software engineering are a structural solution to reoccurring software design flaws (Washizaki, Uchida, Khomh, & Guéhéneuc, 2019). The assessment focuses on the use of design patterns to promote good software architecture on an established system.

The system chosen for the assessment is a hospital customer relationship management (CRM) application (Gjoni, 2022). A CRM system is designed to manage customer data for businesses (Pedron, Picoto, Colaco, & Arajo, 2018). Leart Gjoni is a software engineer who lives in Canada and is the creator of the C# CRM application being investigated (Gjoni, 2022). The primary language used in the user interface of the application is Albanian.

Examining the application will display design patterns implemented by the author. Language localization of the application will be implemented to further demonstrate the use of design patterns.

The overall objective of the assignment is to create a localization element for the application. The localization will allow users to select a language on the login page and have the language persist throughout the application.

# 2 Existing structure

## 2.1 Functional and Non-functional requirements

**Functional Requirements:**

The application being examined is a task management system. The primary purpose of the application is to allow hospital staff to manage patient interactions, display a staff timesheet, and view staff statistical information. The application connects to a Microsoft SQL database. When the application is launched a login screen is displayed. An example of the login screen can be seen in figure 2.1.1.

Graphical user interface, text, application

Description automatically generated

**Figure 2.1.1** Application login screen.

The application requests a username and password on the login screen. When a user is authorized, the application displays a new form panel based on the permissions of the logged-in user.

The administration panel is an example of a form that can be accessed post-login. The administration panel can only be accessed by users who have administrative privileges. An example of the administration panel can be seen in Figure 2.1.2.

Graphical user interface, table

Description automatically generated

**Figure 2.1.2** The administration panel in the application.

The administration panel is made up of multiple sub-panels including the access panel used to navigate the application and a user control panel used to handle system users. The administration panel uses CRUD features to create, read, update, and delete user records accessing the system. The centre list is retrieved from the database and displays information about the users in the system.

**Non-Functional Requirements:**

The ability to redirect users to different forms depending on the user's permissions is an element of security. The implementation of multiple forms post login is enabled to protect sensitive information from unauthorized users.

The application uses error messages if users enter invalid details when attempting to log in. The error messages provide a higher quality user experience when using the CRM application.

As the application contains data used for customer interaction, the speed of the system's performance was taken into consideration during the initial development of the application. The application uses essential classes and methods to reduce the response time required for the system to operate.

## 2.2 Associations between requirements and use cases

The primary interaction for users begins with the login screen. The user enters a username and password to be able to interact with the system.

Depending on the level of permissions that the user has, the user will be redirected to an appropriate form when completing successful login. An example would include a user logging into an administrator account who will be redirected to the administration form.

For brevity, examining the login form and the administration form will be the primary task.

The UML use case diagram for users interacting with the login page can be seen in figure 2.2.1.

Diagram

Description automatically generated

**Figure 2.2.1** Use case diagram demonstrating how users interact with the login page.

When an administrator has accessed the system, the administrator will be redirected to the administration panel. The administration panel offers more features in comparison to the panel that is available for regular users. The extra features include the ability to create, read, update, and delete users who can access the system. The use case diagram for the administration panel can be seen in figure 2.2.2.

Diagram

Description automatically generated

**Figure 2.2.2** Administrative panel functionality.

## 2.3 Interaction between components

**Login form interaction:**

When a user clicks the login button on the main login screen, when attempting to login, a series of checks are performed to validate the user’s login attempt. The login button code can be seen in figure 2.3.1.

Text

Description automatically generated

**Figure 2.3.1** Application login button code.

The initial validation check seen in figure 2.3.1 verifies that the validation constraints are enabled for the form. When the initial validation check is completed, the code verifies the user’s login credentials from the text boxes on the login form page. An instance of the Login class called \_helper is used to call the LoggedIn function that verifies the users’ credentials. If the user’s credentials do not match a stored user within the database, then an error message will appear on the login form.

The code for the LoggedIn function in the Login class can be seen in figure 2.3.2.

Text

Description automatically generated

**Figure 2.3.2** LoggedIn code in the Login class.

The loggedIn function seen in figure 2.3.2 connects to the Microsoft SQL database and verifies that the user exists. If the user is determined to be a valid user, then the password entered in the password text box is compared with the password in the database.

The password stored in the database is stored using a password hashing algorithm. The HashingPassword class uses an HMACSHA1 type of hashing algorithm (Alattar, Farawn, & Ali, 2018).

The HMACSHA1 type of hashing algorithm uses a password masking technique called password salting. Password salting adds additional characters to the front or back of the plaintext to obstruct the plaintext password even further (Alattar, Farawn, & Ali, 2018).

The CheckPassword function converts the password that the user inputted in the text box from plaintext into an encrypted text called cypher text (Alattar, Farawn, & Ali, 2018).

The CheckPassword function also takes the hashed password from the database as an argument. If the user's password in the cypher text matches the hashed password from the database, then the user is authenticated.

When the password is verified the user's RememberMe status is checked using the RememberMe function. The user's details are also taken from the database and stored in the AuthUser class. The AuthUser class stores an instance of the User class called Model. The code for the AuthUser class called model can be seen in figure 2.3.3.

Text

Description automatically generated

**Figure 2.3.3**  Code for the AuthUser class.

The User class contains fields to store all the user details retrieved from the database. The code for the User class can be seen in figure 2.3.4.

A screenshot of a computer

Description automatically generated with medium confidence

**Figure 2.3.**4 Code for the User class.

When the user details are verified and stored in the AuthUser class, the LoggedIn function finishes execution returning true to the login button function in figure 2.3.1. Using the authenticated user’s details stored within the AuthUser class, the role of the user is retrieved from the user’s details.

Based on the authenticated user’s role id, the appropriate form object that the user will be redirected to will be created. The login form object is then hidden, and the new main form will be displayed.

**Admin form interaction:**

The administration panel uses CRUD features to create, read, update, and delete user records accessing the system. The administration panel is demonstrated in figure 2.1.2.

A list is generated in the administration form that can be used to read the users registered with the system.

Administrators can add new users to the system using the “Krijo” button which translates to add in English. The add button code can be seen in figure 2.3.5.

A screenshot of a computer

Description automatically generated with medium confidence

**Figure 2.3.5** Add users button code.

When the add button code is executed a new form is created that allows administrators to enter details for a new user. The form is used for both adding and updating user details. The form for adding details can be seen in figure 2.3.6.

Graphical user interface

Description automatically generated

**Figure 2.3.6** Add user details form.

The administrator can confirm the addition of the new user's details by clicking the "Ruaj" button which translates to save in English. If a user inputs invalid details, the form will present an appropriate error message for each field. An example of the error messages can be seen in figure 2.3.7.

Graphical user interface, application

Description automatically generated

**Figure 2.3.7** Errors in user details form.

If the user details are filled in and the "Ruaj" button is clicked, then the details from the text boxes are saved into the database. An example of the success screen when a new user is successfully added can be seen in figure 2.3.8.

Graphical user interface, application

Description automatically generated

**Figure 2.3.8** New user successfully added.

Editing a system user requires selecting a user from the registered user's list and clicking the "Modifiko" button which translates to edit in English. The code used for the edit button can be seen in figure 2.3.9.

Text

Description automatically generated

**Figure 2.3.9** Edit users button code.

The code for the edit button, seen in figure 2.3.9, attempts to get the user's details from the currently selected object. A check to verify that the user has selected a user from the list is also performed by checking that the user object is not null. Checking the status of the user object before creating the edit form avoids an error when attempting to load the edit form

The edit button form can be seen in figure 2.3.10.

Graphical user interface

Description automatically generated

**Figure 2.3.10** Edit user details form.

The selected user details populate the text box fields. When the administrator edits the user details and clicks the "Ruaj" button the new user information is updated in the database and the success dialog box in figure 2.3.8 is displayed.

Administrators also can remove users from the system. When a user is selected from the registered system users list, an administrator may choose the "Elimino" button, which translates to delete in English, to remove the selected user. The code used to delete a user can be seen in figure 2.3.11.

Text

Description automatically generated

**Figure 2.3.11** Delete user button code.

When the “Elimino” button is selected, a dialog box appears on the administration panel. The delete button dialog box can be seen in figure 2.3.12.

Graphical user interface, table

Description automatically generated

**Figure 2.3.12** Delete user dialog box.

If a user chooses the yes option of the dialog box, then the record is removed from the database and the record on the registered system users list is also updated.

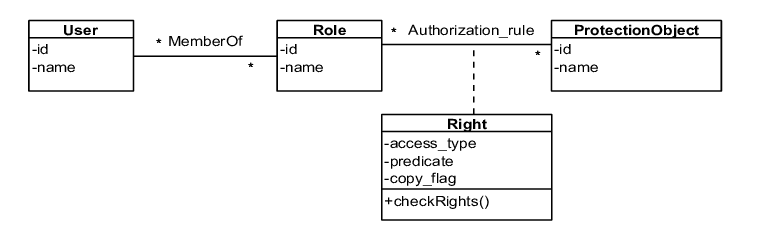
## 2.4 Existing structure design patterns.

**Role-Based Access Control pattern:**

A critical element of the CRM application is to delegate roles and responsibilities to system users to restrict access and permissions. Controlling user access using roles is an example of the Role-Based Access Control pattern (RBAC).

The RBAC is an access control pattern used to enforce the access rights of users based on the user’s roles and permissions (Lee, Woo, Lee, & Joo, 2019). System users are assigned roles and the roles contain permissions needed to perform specific system functions. Users are not assigned the permissions directly but instead acquire the permissions through the user's role (Lee, Woo, Lee, & Joo, 2019).

A class diagram demonstrating the structure of the RBAC can be seen in figure 2.4.1.



**Figure 2.4.1** RBAC structure (Kobashi, et al., 2015).

The structure demonstrates how a user contains a particular role. The user's role determines if the access rights of the user meet the acceptance criteria for interacting with protected objects.

An example of a role being used within the CRM application would include when users attempt to log in. The user's role determines the type of form that should be displayed to the user. The code for the RBAC implementation can be seen in figure 2.3.1.

The ability to assign roles and responsibilities to system users has many advantages. The primary advantage is the ability to manage user access rights. Limiting the access rights of users in confidential system applications is a fundamental security benefit of RBAC.

RBAC also allows businesses to manage the permissions of all employees throughout the company with minimal alterations. The ability of RBAC to manage all employees’ permissions allows RBAC to act as an abstraction layer.

RBAC is more effective in larger companies where the application requires a large number of users. Using RBAC on smaller system applications can become more labour-intensive and more difficult to maintain.

**Model-View- ViewModel pattern and Mediator pattern:**

Another pattern observed when examining the application is the Model-View-ViewModel (MVVM) pattern. The MVVM pattern is an architectural pattern that separates the program logic into three interconnected elements (Hoefling, 2022).

The three elements are the Model, the View, and the ViewModel. The Model contains all the backend data logic. The View contains all the frontend graphical user interface logic. The ViewModel acts as a mediator between the Model and the View where all the ViewModel logic is passed directly to the view (Hoefling, 2022).

The ViewModel element of the MVVM architecture implements the mediator design pattern. The mediator design pattern is a behavioural pattern that directs communications between objects (Mediator, 2022).

An example of the MVVM architecture used in the CRM application file structure to separate the programming logic can be seen in figure 2.4.2.

Text

Description automatically generated

**Figure 2.4.2** Application file structure.

The MVVM architecture decouples the code. Decoupling the code allows the code to be easily maintained and managed which is the primary benefit of using the MVVM architecture. The package structure is easier to navigate, and new features can be added more easily (Hoefling, 2022).

Adding extra classes can cause the code to become more complex and as a result, become more difficult to maintain.

## 2.5 UML class diagram

UML contains various graphical notations that can be used to capture the structure of an application (Planas & Cabot, 2020). UML class diagrams are one form of graphical notation and can demonstrate relationships between classes within the application. An example of a class diagram can be seen in figure 2.5.1.

A screenshot of a computer

Description automatically generated with medium confidence

**Figure 2.5.1** Class diagram from hospital CRM application.

The class diagram seen in figure 2.5.1 demonstrates the interaction between the login class and the EPatient Context class. The EPatient Context class is used to generate the administration panel. The AdminForm class and the LoginForm class are child elements of the MetroForm class. The button class handles various menu buttons within the AdminForm class.

The UML class diagrams are an essential visualisation tool used to understand the scope of the project being worked on. The class methods and relationships help establish how the application functions.

# 3 Design rationale

## 3.1 Factory Method pattern rationale

The primary objective of redesigning the application is to localise the project. To obtain localisation within the project, language classes need to be created based on the user's choice of language. Using the factory method will allow the selection of classes based on the user’s chosen language.

An interface class called words will be created to store the generic word types. The implementation of 3 buttons to allow the user to choose between the different classes will be visible on the Login form.

Using the factory method allows the application to have a generic words class object to be created that can instantiate new language classes based on the user’s chosen language. Using the factory method allows the words class to handle the text on the login form while the language classes will manipulate the text into the selected language.

Having the words class handle only the text on the login form enforces the Single Responsibility Principle (Oktafiani & Hendradjaya, 2018). The Single Responsibility Principle is part of the SOLID software design principles and ensures that a class should only have one responsibility (Oktafiani & Hendradjaya, 2018).

Using the factory pattern also reinforces the Open/Closed Principle in the SOLID software design principles. The Open/Closed principle ensures that classes should be “open for extension but closed for modification” (Oktafiani & Hendradjaya, 2018). The words class will not be modified to add additional languages as the Words class and the language classes are separated.

Separating the responsibility of the Words class and the language classes also helps to avoid tight coupling in the application. Tight coupling occurs when software classes are heavily dependent on other classes (Fritzmann, Sigl, & Seplveda, 2020). Reducing tight coupling in an application allows the application to become more maintainable (Fritzmann, Sigl, & Seplveda, 2020).

The primary disadvantage of using the factory method is the complexity of the code implementation. The introduction of new subclasses when implementing the pattern has the potential to increase the complexity of the application.

## 3.2 Singleton pattern rationale

When a user initially starts the CRM application, a new login form object is created. When a user proceeds to enter the username and password, the user will be redirected to the appropriate form. If the user attempts to log out of the redirected form a new login form object will be created.

The creation of the new login form object becomes an issue when localisation is in place. For example, if a user logs in with the chosen language of English and then attempts to log out from the form the new Login form object will default the language to Albanian.

The singleton pattern can be used to allow continuity between the different forms by instantiating a single login form and calling the same login form when required. The singleton pattern will allow the user's login form object to remain the same as the language on the login form.

The singleton pattern's primary purpose is the pattern's greatest strength and weakness. By using the singleton pattern the class will instantiate a single object that can be accessed throughout the entire application. Having a single instance of an object is beneficial for objects where the values of the class are required to remain the same.

The purpose of using design patterns is to promote quality code. The singleton pattern can be considered an antipattern. An antipattern goes against the principle of using the patterns design by encouraging poor-quality code (Bogner, 2019).

Using the singleton pattern breaks the Single Responsibility Principle. The class manages the creation of a single instance of the class while also attempting to manage the life cycle of the object. Creating a class with the singleton pattern creates a god class. A god class is a code smell within a software application that does too many things within the code (Bogner, 2019).

## 3.3 Proxy pattern rationale

When attempting to connect to the database a user must open a connection to retrieve the username and password. The proxy pattern can be used on the login function to reduce the number of calls to open the connection to the database. If a chosen username is used to attempt to connect to the database and the username is not in the database, then the username is added to a list of invalid usernames. If the user attempts to log into the CRM application using the same username, then the username will be compared with the list of invalid usernames and return false before opening the database connection. Using the proxy method can help reduce the number of unnecessary open connections to the database and improve performance.

An alternative recommended use for the proxy pattern would be to request passwords for users. If a user's password was null or an empty string, then the proxy login could request that the user enters a new password and then proceed with the login request. Having the proxy be used to generate new passwords would allow database administrators to remove the passwords from the database and request users to generate new passwords.

The biggest advantage of using the proxy pattern is that it allows the additional implementation of service objects without altering the original service object. Adding additional proxies to service objects upholds the Open/Closed Principle of the software design principles (Proxy, 2022).

The biggest disadvantage of using the proxy pattern is the introduction of new classes can cause the code to become unnecessarily complex (Proxy, 2022).

## 3.4 State pattern rationale

When a user has selected a language from the login menu, the system must then store the user's selection so that the selected language can be used throughout the rest of the application. The state behavioural pattern can be implemented to alter the application based on the user's chosen language.

The State behavioural pattern helps maintain the Single Responsibility Principle by designating each state with the individual state’s responsibilities. The Open/Closed Principle can also be upheld as additional states can be added without modifying the existing state classes (State, 2022).

The State pattern can be an unnecessary feature in some applications if the application only requires a few states or rarely changes state (State, 2022).

## 3.5 Decorator pattern rationale

The initial Words class contains all the words that need to be changed for the login page. To reduce the opportunity for tight coupling within the Words class, additional Words classes can be created for additional forms. Adding additional functionality to the Words class is an example of the Decorator pattern.

As part of the implementation, the AdditionalWords class will be created for the implementation of the administration panel. Additional decorator classes may be implemented in future implementation to add additional localisation for other forms such as the operator, doctor, or nurse forms.

Using the Decorator pattern allows additional object behaviour. The Decorator pattern maintains the Single Responsibility Principle by separating the functionality of objects into multiple decorators. Additional decorators can be added to the decorator stack to increase the functionality of the base object (Decorator, 2022).

The Decorator pattern's biggest disadvantage is that the pattern can become difficult to maintain. Applying multiple decorators can become a problem if decorators are required to be removed. The decorator’s implementation may become reliant on the current decorator's order within the decorator stack and as a result can become difficult to maintain (Decorator, 2022).

# 4 Redesign and Refactor

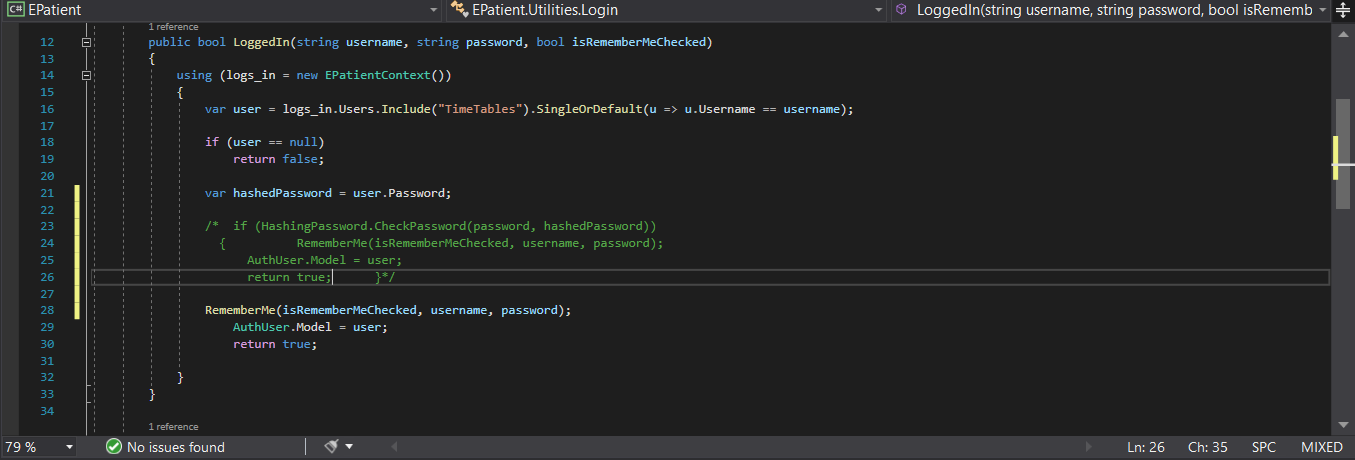
## 4.1 Refactoring

The complex hashing algorithm used on passwords within the application proved to be an obstacle when attempting to access the application.

The Login class contains a LoggedIn function which returns a bool determining if the user has successfully submitted login details. When the function checks the password within the LoggedIn function, the function calls another class called “HashingPassword”.

The HashingPassword class uses an HMACSHA1 type of hashing algorithm. The hashing algorithm converts a password in plaintext into ciphertext. The password also uses password salting, which adds additional characters to the front or back of the plaintext to obstruct the actual password even further. The password is decrypted within the HashingPassword class using the CheckPassword function (Alattar, Farawn, & Ali, 2018).

To allow for the development of the application, the CheckPassword function is commented out. The commenting out of the CheckPassword function can be seen in figure 4.1.1. A recommendation for future implementation would include the clearing of passwords within the database. A proxy class could be used to ask users to enter new passwords if the password is null. Alternatively, if the passwords are known then the user could simply uncomment the CheckPassword function.



**Figure 4.1.1** CheckPassword function commented out in Login class.

## 4.2 Factory method pattern implementation

Implementing a language factory in the CRM application will enable the class to return a language object. Additional language buttons are added to the login form to allow users to select the language. The language buttons can be seen in figure 4.2.1.

Graphical user interface, text, application

Description automatically generated

**Figure 4.2.1** Login form with language buttons.

A LanguageFactory class is used to create the chosen language. A function called getLanguage, which is within the LanguageFactory class returns a Words class object. The function takes a string argument which will be used to determine which Words class object to create.

The code for the LanguageFactory class can be seen in figure 4.2.2

Text

Description automatically generated

**Figure 4.2.2** Language factory class code.

Each language class contains variables of all the words required for the application. Each of the language classes implements the Words class interface. An example of the EnglishWords class code can be seen in figure 4.2.3.

Text

Description automatically generated

**Figure 4.2.3** EnglishWords class code.

The words class contains the functionality for the language class objects. An example of the words class code can be seen in figure 4.2.3

Text

Description automatically generated

**Figure 4.2.3** Words class code.

The Login form class instantiates a LanguageFactory class object. The LanguageFactory object uses the function called getLanguage to instantiate a Words object called language. The default language applied to the getLanguage function is Albanian. The code for instantiating the LanguageFactory as well as the Words class objects can be seen in figure 4.2.4.

A screenshot of a computer

Description automatically generated with medium confidence

**Figure 4.2.4** LanguageFactory and Words class instantiation.

When one of the language buttons is selected from figure 4.2.1 the state of the chosen language is stored. The state will be explained in more detail in figure 4.6. A function called Update\_Values was created to update the language based on the chosen language state which is stored as a string. The text in the UI is then updated based on the chosen language. The code for the language buttons and the Update\_Values button can be seen in figure 4.2.5.

Text

Description automatically generated

**Figure 4.2.5** Language button code and the Update\_Values function.

**Factory Pattern UML diagram:**

The UML diagram for the factory pattern implemented into the CRM application can be seen in figure 4.2.6.

Graphical user interface

Description automatically generated

**Figure 4.2.6** Factory pattern UML diagram.

## 4.3 Singleton pattern implementation

The singleton factory pattern was implemented to demonstrate the potential of the pattern and remove an error that occurred when logging out of the administration panel. The administration panel would create a new login form object when logging out which would reset the language chosen in the original login form.

The LoginFormSingleton class creates a single instance of the class only if the single instance object is null. The LoginFormSingleton class contains a method called getInstance that returns the single instance object. As a result, the object can be accessed globally allowing for the same object to be used throughout the application. The code for the LoginFormSingleton can be seen in figure 4.3.1.

Text

Description automatically generated

**Figure 4.3.1** LoginFormSingleton class code.

The getLoginForm method seen in figure 4.3.1 can be used to get the instance of the login form instantiated in the LoginFormSingleton class.

The LoginFormSingleton class is initially instantiated in the Main class and the getLoginForm function is used to get the Login form of the object. The instantiation of the LoginFormSingleton class object can be seen in figure 4.3.2.

Text

Description automatically generated

**Figure 4.3.2** LoginFormSingleton instantiation in Main class.

The instantiation of the LoginFormSingleton class object in the AdminForm can be seen in figure 4.3.3. The getInstance method returns the same instance of the LoginFormSingleton object that is used in the Main class in figure 4.3.2.

Text

Description automatically generated

**Figure 4.3.3** LoginFormSingleton instantiation in AdminForm class.

Using the getLoginForm method allows the same instance of the Login form to be returned to the user to allow the language of the application to remain consistent. The implementation of the getLoginForm method being used on the Logout button in the Admin form can be seen in figure 4.3.4.

Text

Description automatically generated

**Figure 4.3.4** AdminForm logout button code.

**Singleton Pattern UML diagram:**

The UML diagram for the Singleton pattern implemented into the CRM application can be seen in figure 4.3.5.

Text

Description automatically generated

**Figure 4.3.5** Singleton pattern UML diagram.

## 4.4 Proxy pattern implementation

The proxy pattern was used to remove the unnecessary opening of the database connection. The ProxyLogin class checks for invalid users.

The code opens the connection to check if the user is valid only once. If the username is null, then the username does not exist in the database. The username is then added to a list of invalid users. If the user attempts to log in again then the invalid user list will be checked. If the username appears on the list of invalid usernames, then the LoggedIn function will return false to the login page without opening the database connection.

If the username is determined to be acceptable, then the original Login classes LoggedIn function is called.

The code for the ProxyLogin class can be seen in figure 4.4.1.

Text

Description automatically generated

**Figure 4.4.1** ProxyLogin class code.

The Proxy login class is instantiated in the LoginForm class instead of the original Login class. The code for the ProxyLogin instantiation can be seen in figure 4.2.4 and the \_helper object is implemented in the login button which can be seen in figure 2.3.1.

**Proxy Pattern UML diagram:**

The UML diagram for the Proxy pattern implemented into the CRM application can be seen in figure 4.4.2.

A picture containing graphical user interface

Description automatically generated

**Figure 4.4.2** Proxy pattern UML diagram.

## 4.5 Decorator pattern implementation

To keep the different words of the panels separate, the AdditionalWords class was created. The Words class implements the AdditionalWords class allowing the additional words methods to be used for the additional panels. The Words class implementation of the AdditionalWords class can be seen in figure 4.2.3. The AdditionalWords class contains the words used for the AdminForm.

A sample of the code of the AdditionalWords class can be seen in figure 4.5.1.

Text

Description automatically generated

**Figure 4.5.1** AdditionalWords class code.

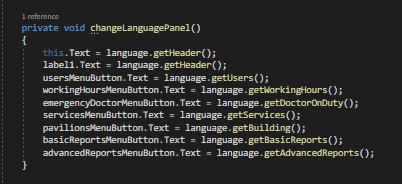
The additional words required for the AdminForm are added to the language classes the SpanishWords class can be seen in figure 4.5.2

Text

Description automatically generated

**Figure 4.5.2** SpanishWords class containing the Admin panel words required for the additional words class.

Using the methods in the AdditionalWords class, the Words object uses the AdditionalWords class methods to obtain the new words for the AdminForm. When the AdminForm loads, a function called changeLanguagePanel is called. The function updates all the text fields on the AdminForm determined by the language state chosen in the login form. The code for the changeLanguagePanel can be seen in figure 4.5.3.



**Figure 4.5.3** The changeLanguagePanel function that uses the AdditionalWords class methods to update the AdminForm class on load.

**Decorator Pattern UML diagram:**

The UML diagram for the Decorator pattern implemented into the CRM application can be seen in figure 4.5.4.

Graphical user interface

Description automatically generated

**Figure 4.5.4** Decorator pattern UML diagram.

## 4.6 State pattern implementation

Implementing a state allows the application to maintain continuity throughout the application. The LanguageState class contains a string to contain the selected language state. The LanguageState class contains a getter and a setter to interact with the language state variable.

The LanguageState class also contains a method called selectedLanguage that is used to return a Words class object based on the current value of the languageState variable.

An example of the code of the LanguageState class can be seen in figure 4.6.1.

Text

Description automatically generated

**Figure 4.6.1** LanguageState class code.

When a language button is selected in the LoginForm the LanguageState is set to the chosen language. The code for the language buttons, where the state is set, can be seen in figure 4.2.5.

The language state is used with the factory method to alter the current language of the objects on the login form.

When users successfully log in to the administration panel, the administration form page is altered based on the current language state. In the AdminForm class, an instance of the LanguageState is instantiated to create a languageState object. The languageState object calls the selectedLanguage function to instantiate the Words class object called language. The Words class object will contain the selected language from the LoginForm because of the LanguageState class.

The code instantiating the LanguageState class and the Words class in the AdminForm can be seen in figure 4.6.2.

A screenshot of a computer

Description automatically generated with medium confidence

**Figure 4.6.2** LanguageState class object instantiation and Word class object instantiated with the language state.

**State Pattern UML diagram:**

The UML diagram for the State pattern implemented into the CRM application can be seen in figure 4.6.3.

Text

Description automatically generated with medium confidence

**Figure 4.6.3** State pattern UML diagram.

# 5 Conclusion and Reflection

The objective of the assessment was to evaluate the software architecture of an established system and apply design patterns to the system. The work conducted focused on five software design patterns implemented in a CRM application. The five software design patterns included the Factory Method pattern, the Singleton pattern, the Proxy pattern, the State pattern, and the Decorator pattern.

The implementation of the design patterns required research into each pattern to understand the functionality of the given pattern and how the application of the pattern could improve the current software architecture structure.

Exploring the original application demonstrated design patterns used by the author of the application. The Role-Based Access Control pattern and the Model-View-ViewModel pattern were used in the creation of the application. Using the two patterns allowed the application to be easily accessed, managed, and maintained.

The application required refactoring to allow the development of the application. The refactoring included removing the password authentication element. An alternative solution to removing the password would have been to reset the administration password or create an Administrator account with a known password.

The work conducted focused on language localisation of the UI elements in the application. The three languages chosen for the study include Albanian, English, and Spanish languages.

An alternative solution to using design patterns to implement language localisation is to use a localisation plugin for the IDE. An example of a localisation plugin includes the ResXManager plugin for visual studio (Englert, 2022).

Implementing the different design patterns demonstrated the effectiveness of each design pattern and how different design patterns can be used in cohesion to produce a better-quality application.

Using the Factory pattern allows for future language elements to be added to the application with minimal refactoring.

The singleton pattern used for the Login form was an overengineered element of the application. Using the singleton pattern produced the results required to enable language continuity throughout the application. An alternative solution to using the singleton pattern would have been to hide the original login form object and show the login form object again when the user would log out of the administration form.

The Proxy patterns implementation removes unnecessary calls to the database. An alternative use of the Proxy pattern would have been to allow the users to change their passwords by prompting the users to change their password with a dialog box. When the user’s passwords would have been reset the Login classes LoggedIn function could have been called to complete the login process.

The Decorator pattern has the potential to add additional functionality to the CRM application. Upon reflection on the code for the decorator pattern, a more applicable naming convention would be more suitable for future implementation. The Words class should be renamed to the LoginFormWords class and the AdditionalWords class should be renamed to the AdminFormWords class.

The State pattern stored the language state used after login to update the administration form page. The State pattern is a powerful design pattern that was well suited for the localisation element of the application. Future language implementation within the CRM application will become more manageable because the language state class handles the language implementation.

The design patterns implemented in the assessment allow for a clean code base and a more maintainable application for future development. Initial exploration of all the elements of the patterns used within the assessment required an investment of time. The effectiveness of the patterns produced a better quality application with additional features.

# 6 Code links

The code for the original application before modification can be found at the following URL:

<https://github.com/leartgjoni/c-sharp-hospital-crm>

The code was forked from the original repository and the code with the design pattern implementation can be found at the following URL:

<https://github.com/GerardOSullivan/c-sharp-hospital-crm>

As the application focused on an administration account the username “Admin” will display the administration panel. The password system was removed for demonstration purposes, so no password is required to access the application.

# References

Alattar, M. H., Farawn, A. A., & Ali, N. S. (2018). Anti-continuous collisions user-based unpredictable iterative password salted hash encryption. *Int. J. Internet Technology and Secured Transactions*, 619-634.

Bogner, J. a. (2019). Towards a Collaborative Repository for the Documentation of Service-Based Antipatterns and Bad Smells. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)* (pp. 95-101).

*Decorator*. (2022, 12 01). Retrieved from refactoring.guru: https://refactoring.guru/design-patterns/decorator

Englert, T. (2022, 12 03). *ResXManager*. Retrieved from visualstudio.com: https://marketplace.visualstudio.com/items?itemName=TomEnglert.ResXManager

Fritzmann, T., Sigl, G., & Seplveda, J. (2020). RISQ-V: Tightly coupled RISC-V accelerators for post-quantum cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 239-280.

Gjoni, L. (2022, 10 27). *Desktop Based Hospital CRM in C#*. Retrieved from github.com: https://github.com/leartgjoni/c-sharp-hospital-crm

Hoefling, S. (2022). Model-View-ViewModel (MVVM). In *Getting Started with the Uno Platform and WinUI 3* (pp. 191-206). Springer.

Kobashi, T., Yoshizawa, M., Washizaki, H., Fukazawa, Y., Yoshioka, N., Kaiya, H., & Okubo, T. (2015). TESEM: A Tool for Verifying Security Design Pattern Applications by Model Testing. *2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015 - Proceedings*.

Lee, J., Woo, J., Lee, C., & Joo, K. (2019). A software development methodology for secure web application. *International Journal on Advanced Science, Engineering and Information Technology*, 336-341.

*Mediator*. (2022, 12 01). Retrieved from refactoring.guru: https://refactoring.guru/design-patterns/mediator

Oktafiani, I., & Hendradjaya, B. (2018). Software metrics proposal for conformity checking of class diagram to SOLID design principles. In *2018 5th International Conference on Data and Software Engineering (ICoDSE)* (pp. 1-6).

Pedron, C. D., Picoto, W. N., Colaco, M., & Arajo, C. C. (2018). CRM System: The role of dynamic capabilities in creating innovation capability. *BBR. Brazilian Business Review*, 494-511.

Planas, E., & Cabot, J. (2020). How are UML class diagrams built in practice? A usability study of two UML tools: Magicdraw and Papyrus. *Computer Standards & Interfaces*, 103363.

*Proxy*. (2022, 12 01). Retrieved from refactoring.guru: https://refactoring.guru/design-patterns/proxy

*State*. (2022, 12 01). Retrieved from refactoring.guru: https://refactoring.guru/design-patterns/state

Thakur, R. N., & Pandey, U. (2019). The role of model-view controller in object oriented software development. *Nepal Journal of Multidisciplinary Research*, 1-6.

Washizaki, H., Uchida, H., Khomh, F., & Guéhéneuc, Y.-G. (2019). What Do We Know about the Effectiveness of Software Design Patterns? In S. S. Systems, *2019 10th International Workshop on Empirical Software Engineering in Practice (IWESEP)* (pp. 49-495).