MP 307 Part 1

2003115 Robbie Sangster

Contents

[Requirements Specification 2](#_Toc121760995)

[Functional Requirements 2](#_Toc121760996)

[Non-Functional Requirements 2](#_Toc121760997)

[UML Diagrams 3](#_Toc121760998)

[Class Diagram 3](#_Toc121760999)

[Sequence Diagram 4](#_Toc121761000)

[Design Principles and Patterns 5](#_Toc121761001)

[Reliability coding practices and testing 6](#_Toc121761002)

[Critical evaluation 8](#_Toc121761003)

[Git Repository Link 9](#_Toc121761004)

# Requirements Specification

## Functional Requirements

From the client brief the functional requirements.

1. Users shall be able to add, view, edit or delete asset data.
2. Users shall be able to add, view, edit or delete software data.
3. The system will store asset name, model, manufacturer, type, IP address, purchase date (if applicable) and an optional text note.
4. The system will store software name, version and last update.
5. A user must login before accessing the information.
6. The System shall be easy to learn with clear instructions.
7. The database shall be designed and run on windows 10.
8. The system shall have clear instructions and minimal learning curve.
9. The system shall get hardware date from the current machine.
10. The system will be search for vulnerabilities.

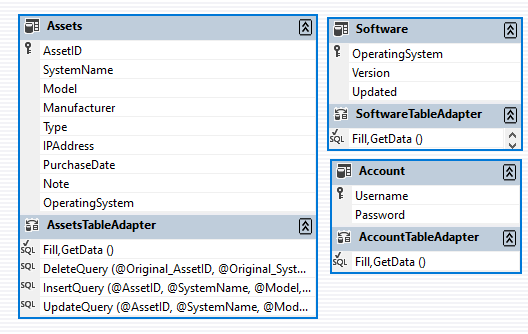
## Non-Functional Requirements

1. The system should have a minimal response time.
2. The system should take no more than 30 min of training to understand.
3. The system screen refresh time should not delay data entry.
4. The data should be displayed in a clear and concise manner that does not confuse the user.

# UML Diagrams

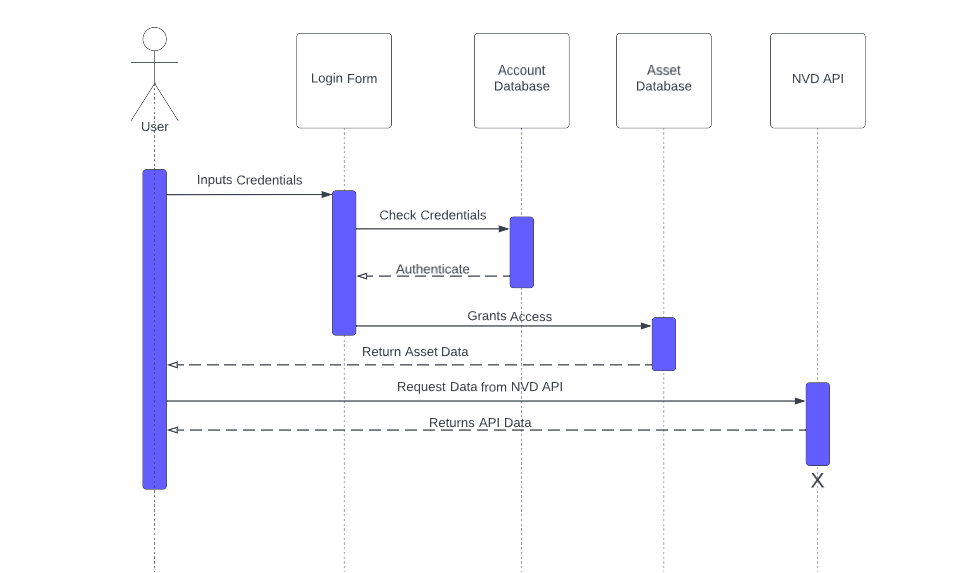
## Class Diagram

Class diagram taken from visual studio.



## Sequence Diagram

Sequence Diagram of user requesting data from the NVD API.



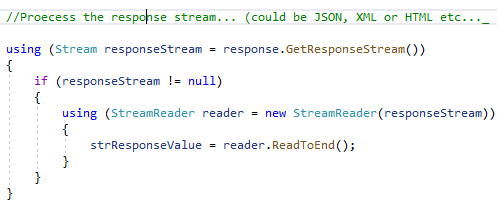
# Design Principles and Patterns

|  |  |
| --- | --- |
| Pattern Name | Singleton |
| Description | Singleton design pattern allows you to create a single instance of an object in which all classes can access. This provides a global access to the object and reduces the amount of repeat code. |
| Problem description | The code for this example is the connection to the database. Creating a new SQL connection and declaring the connection string consisting of a data source, initial catalog, username and password. This is called multiple times throughout the code and would be better suited to be available globally. |
| Solution Description | The solution for this is to make a global access point where all classes can call to connect to the database server. To do this we create the connection inside the Appconfig file. This way we never have to code out the entire connection string only the conn.open and declare the new query. |

|  |  |
| --- | --- |
| Pattern Name | Bridge |
| Description | Split large class into subclasses which can be developed and worked on independently from one another. |
| Problem description | The original class is very large as is for the windows form. This makes coding harder as you have so many moving parts to manage that by changing one part it could have consequences elsewhere. The aim of a bridge is to split up classes to make them independent and easier to manage. |
| Solution Description | The solution is to create multiple independent classes which do different things. We have 3 classes in the project, The main form, login and RESTClient each working on their own jobs. Login is used first to verify the user trying to login, the form is used to load assets into the data grid view and allow you to modify it, RESTClient is used to access the NVD API and return the value. Splitting it up makes coding easier to manage and reduces the negative consequences that can happen when edit other code. |

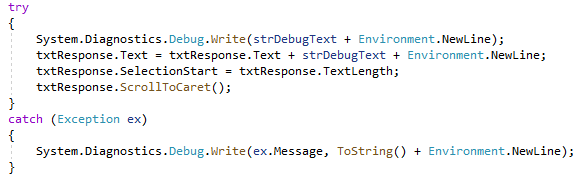
# Reliability coding practices and testing

When designing and creating this project reliable and secure code practices were used to make the code more readable and understandable to those who may be looking at it for the first time. Example of this can be seen in the code through comments. Comments can be seen through the code explaining what the sections do. Below is as example of code commenting which describes the API response handler.

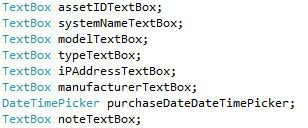


If someone new was to view my code it would be difficult to follow what this does without a description.

Try catch statements are used where code may throw exceptions and this will stop the application from crashing if there is an exception. This makes the program more reliable as it won’t stop running from an exception.



Naming conventions are another coding practice I used. All form items follow the convention of nameItem. Example assetIDTextBox, manufacturerTextBox. This is another way of making code more readable to first timer viewers as each item will be clearly identifiable.



Writing efficient code is also an important practice to make my code more efficient I declared the SQL connection string which contains the source address, username and password is written once which all classes can access.



When opening connections they are always closed when they have completed their task this way there are no stray connections left open.

To make the code itself secure passwords are stored in hash state so plain text is not being passed around. See below.



When entering the password the text is also hidden and the default is to require a login before any data information is loaded to the form.

When I came to testing my code I unit tested each individual section once implemented to ensure that I worked as it should before moving on to the next part. This way future additions will not be held up by previous work not functioning as intended. This is much more value than delaying testing till other sections are complete which will be harder to fix with more moving parts the later you wait. By having tested working code this can then be reused later. As an example I could reuse the data load code from the asset table in the software data.

Testing examples for adding an asset would be testing a duplicate primary key, using the wrong data types for data, leaving blank/null, doing everything possible to break the system and check it gives you the correct response. This was done for each part of the system. Assets retrieve, adding, deleting same for the software data, getting the system info, NVD API and the login system. Then once each system had been fully implemented and individually tested, I tested the system as a whole doing each system feature back to back in different orders multiple times until everything is done without failure. Rigorous testing is vital in any software system so it was important to test everything as much as possible.

# Critical evaluation

When creating this system my process started with the base prototype and planned out each change I needed to make from the functional and non-functional requirements. From there I worked on the new requirements in order of importance to the overall system. First was the getting the software data into a table on Microsoft SQL Management studio. The table doesn’t contain much data only the OS Name, version and the last updated date. If the client is interested in storing more data implementing more columns into the database and application won’t be a difficult task. Reading the system information was a part of the prototype requirements I never implement but is present in the final system. It contains lots of information from the ComputerSystem, Diskdrive, OperatingSystem, Processor, ProgramGroup, SystemDevices and StartupCommands. These are displayed in an array view and organised alphabetically. This system works by itself without any connections or databases. As there is a lot of data shown a future addition could be a search bar to make it easier find what you’re looking for.

One of the most important changes from the prototype is the login system. It adds a basic security to the database preventing unwanted users gain access to the data. The login system works by opening a login window before the main form which requires a username and password. The username and passwords are stored in an accounts table. The password is stored using bcrypt which is a hashing method to ensure plain text is not storing passwords. This is an important addition to any storage of valuable data. To further improve this system would be to add a create account option so more people can login. To further improve the security would be adding salt and pepper to the hashed password. The final change was the National Vulnerability Database (NVD) API to check for potential vulnerabilities in the software. Using the developer tools found on the NVD site gave a detailed guide on how to use the API and set parameters and what JSON it returns. As I had never used an API in c# before when it came to implementing this, I found a useful blog on how to use a rest client and get the data from the URL into the form.

Improvements that can still be made would be potential user interface changes. Right now the interface is very congested with all the different features. Having multiple tabs to manage the feature would make the UI more manageable. We could also look at acquiring more data from the NVD API, currently all we take is the vulnerability source and how exploitable it is in the form of a score. The response from the API contains lots more data such as severity, authentication, vectorString etc. I only took out the source and score as otherwise there would be too much information and most of the data returned won’t be very beneficial to the client.

# Git Repository Link

<https://github.com/RSangster1/307-Coursework.git>