# Refactoring
# CKD Calculator
# (Team 33)
# NephroTrack

This section documents the key refactoring activities carried out during the development of NephroTrack. The goal of refactoring was to improve the code quality, maintainability, and security without altering the external behaviour of the application. Techniques such as encapsulation, access control adjustments, and modularisation were applied to enhance the structure and readability of the codebase. These improvements were informed by early testing, security audits, and feedback from team review sessions.

Refactoring was approached iteratively, with each cycle informed by internal testing, code reviews, and security audits. We focused on applying widely recognised best practices, such as encapsulation, modularisation, and the DRY (Don't Repeat Yourself) principle, to reduce technical debt and enhance future extensibility.

**Refactoring and Code Improvements**

Throughout both development iterations, several refactoring techniques were implemented to improve code structure and enforce separation of concerns. A primary area of focus was encapsulation. Initially, sensitive database elements – such as user IDs and password fields – were exposed through public variables in Unity scripts. These were refactored using private access modifiers alongside secure getter/setter methods, ensuring that sensitive logic was only accessible within explicitly defined scopes.

Other improvements included:

-Method Extraction: Repeated logic for tasks such as input validation, login attempts, and CSV parsing was modularised into reusable methods. This reduced redundancy and improved readability.

-Lockout Logic Isolation: The brute force protection mechanism was separated into a discreet logic handler to better control flow and error handling.

-Password Hashing: The introduction of SHA-256 hashing was encapsulated into a secure method call, aligning password storage practices with NHS and GDPR expectations.

-Code Commenting and Naming Conventions: Variables and functions were renamed for clarity, and comments were added to improve collaboration and long-term maintainability.

Each refactoring decision was guided by feedback from our testers and informed by real-world security concerns. These improvements contribute to the stability, safety, and clarity of NephroTrack as it continues into future development cycles.

```
public void SelectFile()
{
    NativeFilePicker.Permission persimssion = NativeFilePicker.PickFile(async (path) =>
    {
        if (!string.IsNullOrEmpty(path) && Path.GetExtension(path).ToLower() == ".csv")
        {
            resultText.text = string.Empty;
            try
            {
                Debug.Log("CSV file selected: " + path);
                filePath = path;
                List<eGFRData> fileContent = await ParseCSV();
                if (!resultCanvas.isActiveAndEnabled) resultCanvas.gameObject.SetActive(true);
                int elementCount = 1;

                foreach (eGFRData data in fileContent)
                {
                    if (elementCount == 11 || fileContent[fileContent.Count - 1] == data)
                    {
                        pages.Add(resultText.text);
                        resultText.text = string.Empty;
                        elementCount = 1;
                    }
                    FormatText(data);
                    elementCount++;
                }
                OnFormatComplete?.Invoke(pages);
                Debug.Log(transform.root.gameObject.name);
            }

            catch (Exception ex)
            {

                Debug.LogError("Error: fileContent is empty. CSV might not be parsed correctly.");
                resultText.text = "Please select a valid csv file with the correct formatting";
                if (!resultCanvas.isActiveAndEnabled) resultCanvas.gameObject.SetActive(true);
                if (resultCanvas.gameObject != transform.root.gameObject) transform.root.gameObject.SetActive(false);
            }
```

*Figure 1: encapsulated CSV Upload Handler - demonstrates structured, modular parsing logic with condition checks and method invocation.*

```
if (!lockoutStarted && incorrectCount >= maxIncorrect)
{
    lockoutStarted = true;
    userIdField.gameObject.SetActive(false);
    userIdField.text = string.Empty;
    passwordField.gameObject.SetActive(false);
    passwordField.text = string.Empty;
    await Task.Delay(lockoutSeconds * 1000);
    incorrectCount = 0;
    lockoutStarted = false;
    userIdField.gameObject.SetActive(true);
    passwordField.gameObject.SetActive(true);
}
```

*Figure 2: Logic Lockout Functionality - refactored to prevent brute force attacks using conditional logic and field reset controls.*

```
private string HashData(string data)
{
    using (SHA256 sha256 = SHA256.Create())
    {
        byte[] hashedBytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(data));
        return BitConverter.ToString(hashedBytes).Replace("-", "").ToLower(); // Convert to HEX
    }
}
```

*Figure 3: SHA-256 Password Hashing Implementation - shows secure password storage using cryptographic hashing with method encapsulation.*

Collectively, these refactoring practices improved code robustness, reduced technical debt, and laid the foundation for secure future enhancements.