

Documentation

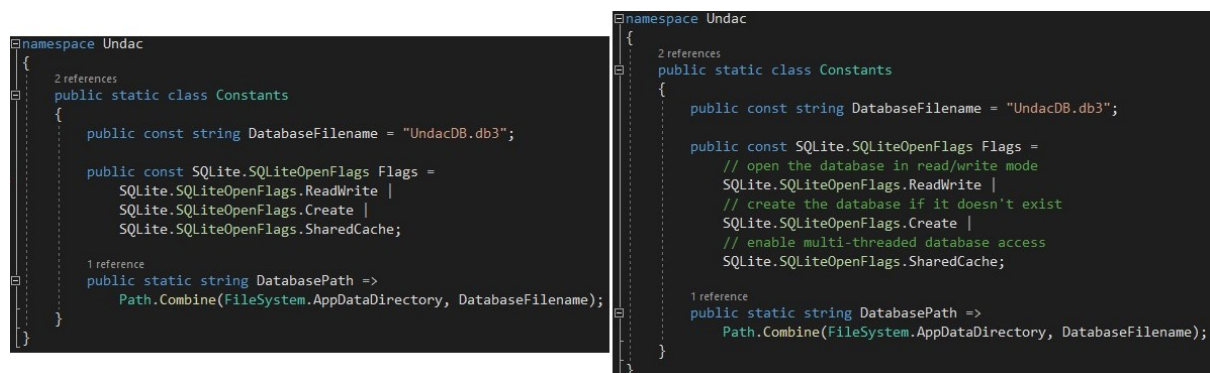
This document aims to describe the work done during week five, which consisted in getting a deeper understanding of the book "Clean Code" and of code documentation.

Applying Clean Code

We were asked to pick a few principles to which our codebase adheres. I decided to pick the following 6 as I think my code fully adopts them, so I can easily talk about how I decided to implement them:

1. **Good comments:** according to the book, the comments' function is to increase code clarity and reason, making it more understandable. However, most of the time, we should also be able to write code that is self-explanatory and does not need comments, which in fact can sometimes be seen as excessive and useless, even as a signal of potential issues. So comments should enhance the code's understanding, and I have the perfect example for that.

As you can see below, in my **Constants** class there is a property, called **Flags**, and a reader might not know what those "flags" are, despite the first two being quite self-explanatory. To solve this issue I added a comment, on each flag, explaining what it is used for, that way I enhanced my code:



```
namespace Undac
{
    2 references
    public static class Constants
    {
        public const string DatabaseFilename = "UndacDB.db3";

        public const SQLite.SQLiteOpenFlags Flags =
            SQLite.SQLiteOpenFlags.ReadWrite |
            SQLite.SQLiteOpenFlags.Create |
            SQLite.SQLiteOpenFlags.SharedCache;

        1 reference
        public static string DatabasePath =>
            Path.Combine(FileSystem.AppDataDirectory, DatabaseFilename);
    }
}

namespace Undac
{
    2 references
    public static class Constants
    {
        public const string DatabaseFilename = "UndacDB.db3";

        public const SQLite.SQLiteOpenFlags Flags =
            // open the database in read/write mode
            SQLite.SQLiteOpenFlags.ReadWrite |
            // create the database if it doesn't exist
            SQLite.SQLiteOpenFlags.Create |
            // enable multi-threaded database access
            SQLite.SQLiteOpenFlags.SharedCache;

        1 reference
        public static string DatabasePath =>
            Path.Combine(FileSystem.AppDataDirectory, DatabaseFilename);
    }
}
```

Fig.1 - Writing good comments

2. **Vertical and horizontal formatting:** once again another thing we should always look at when we are writing code, formatting. For our entire codebase to be clear and understandable, it should adhere to some standards, for example: follow the same styling format, use the same amount of space for indentation, not exceed the chosen line length limitation, group together functions that are related, etc. An example from my code would be the **GetItemsAsync_ReturnsAllItems()** method from the **UndacUnitTests** class: other than following the C# standard conventions like indentation and variable naming, it is also part of a class containing other functions with the aim of testing. After refinement it now also follows a specific spacing pattern between lines and abide by a line length rule as you can see below:

```
[Test]
0 references
public async Task GetItemsAsync_ReturnsAllItems()
{
    // Arrange

    if (mockDbService == null || undacDatabase == null)
    {
        Assert.Fail("Mock or UndacDatabase is null.");
        return;
    }

    var expectedItems = new List<RoomType> { new RoomType { ID = 1, Name = "Room 1" }, new RoomType { ID = 2, Name = "Room 2" }, new RoomType { ID = 3, Name = "Room 3" } };
    mockDbService.Setup(x => x.GetItemsAsync()).ReturnsAsync(expectedItems);

    // Act

    var result = await undacDatabase.GetItemsAsync();

    // Assert

    CollectionAssert.AreEqual(expectedItems, result);
}
```

```
[Test]
0 references
public async Task GetItemsAsync_ReturnsAllItems()
{
    // Arrange
    if (mockDbService == null || undacDatabase == null)
    {
        Assert.Fail("Mock or UndacDatabase is null.");
        return;
    }

    var expectedItems = new List<RoomType>
    {
        new RoomType { ID = 1, Name = "Room 1" },
        new RoomType { ID = 2, Name = "Room 2" },
        new RoomType { ID = 3, Name = "Room 3" }
    };
    mockDbService.Setup(x => x.GetItemsAsync()).ReturnsAsync(expectedItems);

    // Act
    var result = await undacDatabase.GetItemsAsync();

    // Assert
    CollectionAssert.AreEqual(expectedItems, result);
}
```

Fig.2 - Good formatting

3. **Do one thing:** well-organized functions are essential. They should have a clear objective and be concise. Their name should be appropriate and self-explanatory and they should have the least amount of parameters possible so they can be understood better. Last but not least they should only do one thing and have no side effects on other parts of the code.

As you can see below the `DeleteRoomTypeClicked()` method has a very descriptive name, it is concise, it only has two parameters and it is not doing anything else other than what is mentioned in its name:

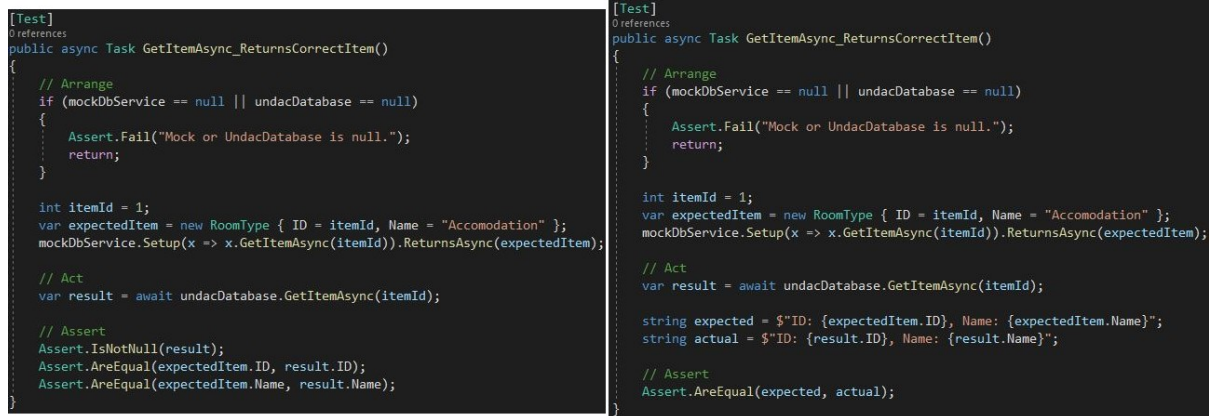
```
public async void DeleteRoomTypeClicked(object sender, EventArgs e)
{
    if (_currentRoomType is null) return;

    var confirmed = await DisplayAlert("Delete Room Type", "Are you sure you want to delete this room type?", "Yes", "No");

    if (confirmed)
    {
        await _database.DeleteItemAsync(_currentRoomType);
        _currentRoomType = null;
        LoadRoomTypes();
    }
}
```

Fig.3 - Ideal function

4. **One assert per test:** tests are probably the most important thing to keep an eye on when programming. They help us to make sure that our code works, and if it doesn't, it helps us find where the issues are. Tests should be clear, independent and effective, they should obviously follow all Clean Code principles like avoiding duplication or being descriptive, and they should have one assertion. My code was initially doing more than one assertion but I found an easy way to make it a single one therefore adhere to the principle, as shown in figure 4:



```
[Test]
0 references
public async Task GetItemAsync_ReturnsCorrectItem()
{
    // Arrange
    if (mockDbService == null || undacDatabase == null)
    {
        Assert.Fail("Mock or UndacDatabase is null.");
        return;
    }

    int itemId = 1;
    var expectedItem = new RoomType { ID = itemId, Name = "Accommodation" };
    mockDbService.Setup(x => x.GetItemAsync(itemId)).ReturnsAsync(expectedItem);

    // Act
    var result = await undacDatabase.GetItemAsync(itemId);

    // Assert
    Assert.IsNotNull(result);
    Assert.AreEqual(expectedItem.ID, result.ID);
    Assert.AreEqual(expectedItem.Name, result.Name);
}

[Test]
0 references
public async Task GetItemAsync_ReturnsCorrectItem()
{
    // Arrange
    if (mockDbService == null || undacDatabase == null)
    {
        Assert.Fail("Mock or UndacDatabase is null.");
        return;
    }

    int itemId = 1;
    var expectedItem = new RoomType { ID = itemId, Name = "Accommodation" };
    mockDbService.Setup(x => x.GetItemAsync(itemId)).ReturnsAsync(expectedItem);

    // Act
    var result = await undacDatabase.GetItemAsync(itemId);

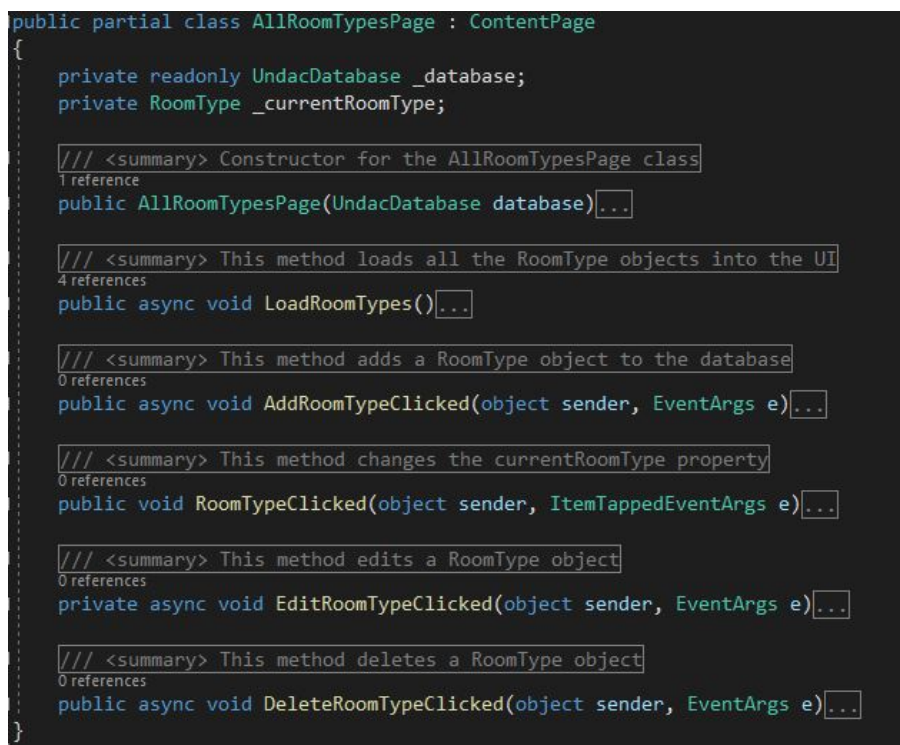
    string expected = $"ID: {expectedItem.ID}, Name: {expectedItem.Name}";
    string actual = $"ID: {result.ID}, Name: {result.Name}";

    // Assert
    Assert.AreEqual(expected, actual);
}
```

Fig.4 - Single assertion

5. **Classes should be small:** classes should be well structured and readable, encourage encapsulation and a right use of dependency injection; they should follow three of the five SOLID principles that mostly apply to classes which are SRP, OCP and LSP. Finally they should be kept small and well organised. As the book says, unlike functions, classes' size is not measured by lines count but by responsibilities count.

The `AllRoomTypesPage` class, which I created for taking care of the user interaction with the UI, can be easily considered a simple and small class with a perfect amount of responsibilities:



```
public partial class AllRoomTypesPage : ContentPage
{
    private readonly UndacDatabase _database;
    private RoomType _currentRoomType;

    /// <summary> Constructor for the AllRoomTypesPage class
    1 reference
    public AllRoomTypesPage(UndacDatabase database) {...}

    /// <summary> This method loads all the RoomType objects into the UI
    4 references
    public async void LoadRoomTypes() {...}

    /// <summary> This method adds a RoomType object to the database
    0 references
    public async void AddRoomTypeClicked(object sender, EventArgs e) {...}

    /// <summary> This method changes the currentRoomType property
    0 references
    public void RoomTypeClicked(object sender, ItemTappedEventArgs e) {...}

    /// <summary> This method edits a RoomType object
    0 references
    private async void EditRoomTypeClicked(object sender, EventArgs e) {...}

    /// <summary> This method deletes a RoomType object
    0 references
    public async void DeleteRoomTypeClicked(object sender, EventArgs e) {...}
}
```

Fig.5 - Ideal class

6. **Don't pass or return null:** in Chapter 7 the book emphasizes the importance of an appropriate error management using exceptions rather than returning error codes. But it also suggests using try-catch-finally blocks at the beginning of our functions and meaningful exception messages to debug more effectively. Finally it is against passing and using null as a parameter, suggesting to use any kind of alternative, which will be more reliable.

After noticing that Visual Studio was telling me about a potential issue with the fact that either `mockDbService` or `undacDatabase` could have been null, I decided to add a null check which solved the problem as shown in figure 6:

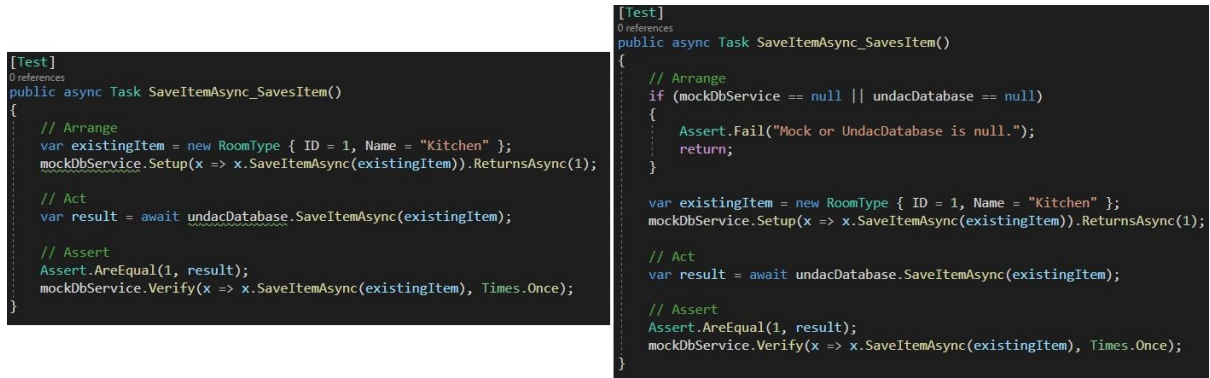


Fig.6 - Null avoiding

Doxygen for documenting

Doxygen is a program that scans thorough your project looking for XML comments, gathers them and finally puts them together on a web page. On this webpage it is possible to look at what each class looks like, all the different methods and inheritance relations. In the first example below you can see the project's structure, and next to each class there is their corresponding XML comment, **AllRoomTypesPage** is shown here:

```

/// <summary>
/// This class contains the logic for the AllRoomTypesPage page
/// </summary>

```

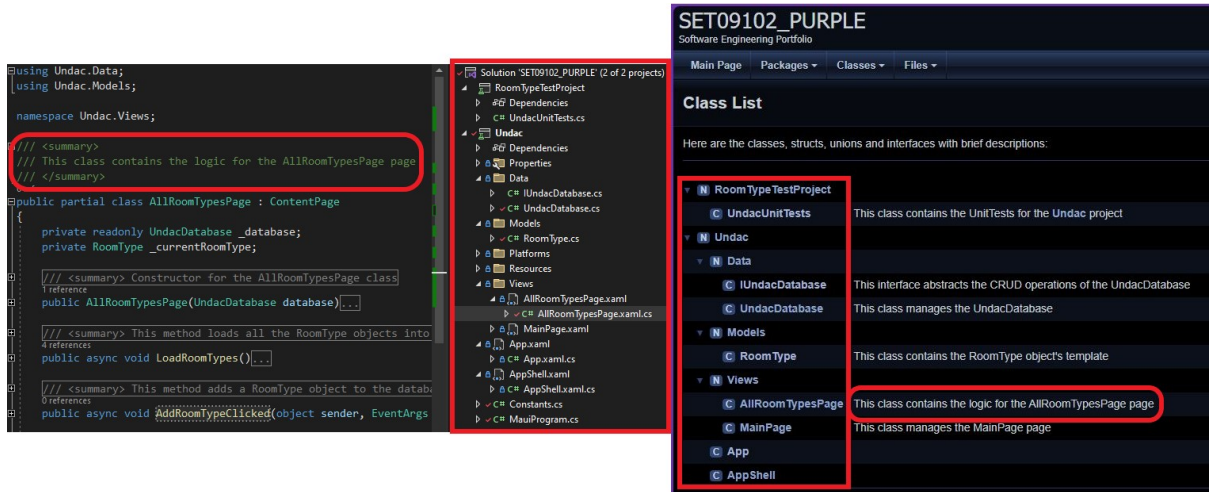


Fig.7 - Project overview

The second example shows what happens when you click on a class name. Doxygen gathers the different XML comments in the class to put together a very informative page showing the class' details like its name, its properties, any inheritance relation and the different methods it contains. Figure 8 shows the **UndacDatabase** class:

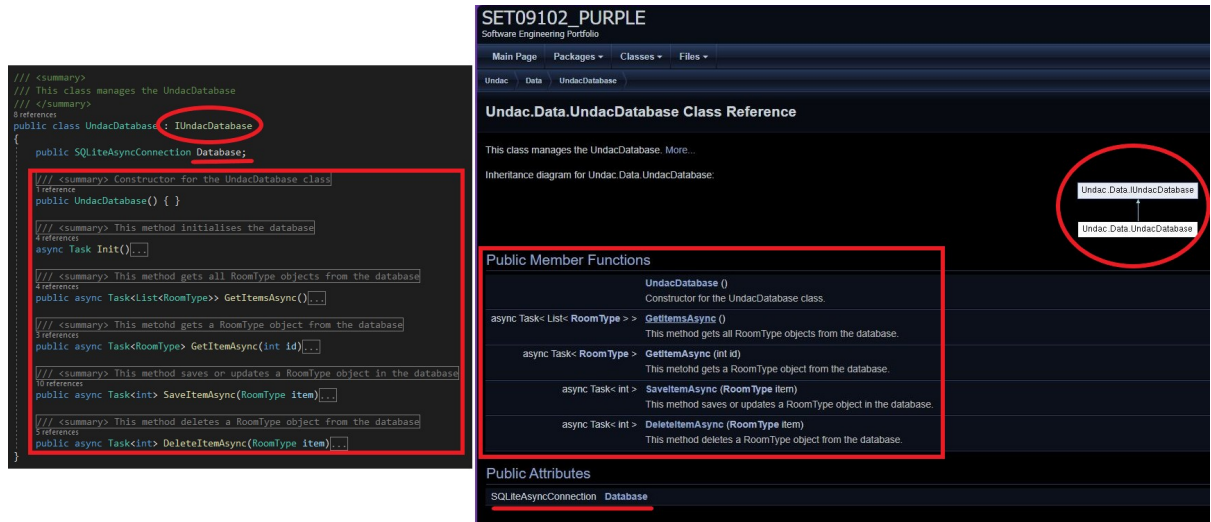


Fig.8 - Class overview

Finally each method is shown in even deeper details if you scroll down the page. You can see that the XML comment, containing a summary, a parameter and a return, is processed and an accurate description of the method, in this case `GetItemAsync()`, is displayed in the web page:

```
/// <summary>
/// This metohd gets a RoomType object from the database
/// </summary>
/// <param name="id">The id of the RoomType object</param>
/// <returns>A RoomType object</returns>
```

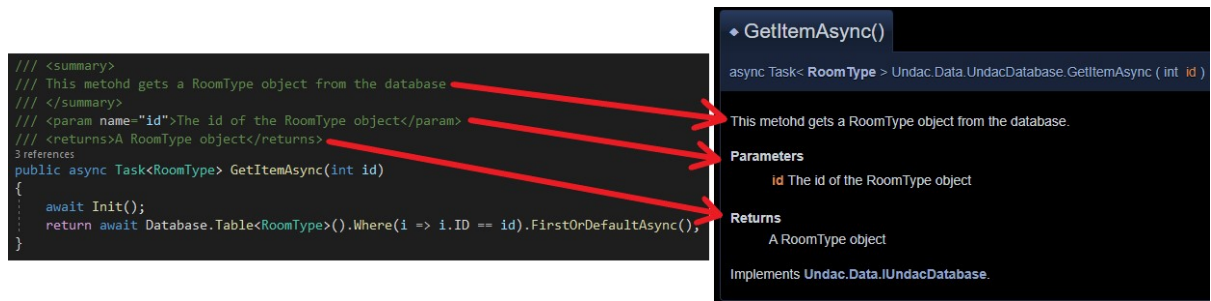


Fig.9 - Methods overview

Clean Code to eliminate comments

Adhering to the Clean Code principles also means that it will be very likely that you will have to remove some of the comments you have written since they will no longer be needed. In my case I can list three situations in which it happened:

1. In figure 10 it is possible to see and comprehend the importance of using **descriptive names**, in fact the `Add()` method needed some further explanation since it could have been doing anything. After renaming it to `AddRoomTypeClicked()` it was clear when the event was being fired and what it was doing, therefore it did not need a comment anymore:

```

/// <summary>
/// This method adds a RoomType object to the database
/// </summary>
/// <param name="sender">Sender</param>
/// <param name="e">Event data</param>
0 references
public async void Add(object sender, EventArgs e)
{
    var roomTypeName = roomTypePicker.SelectedItem?.ToString();

    if (!string.IsNullOrEmpty(roomTypeName))
    {
        await _database.SaveItemAsync(new RoomType { Name = roomTypeName });
        LoadRoomTypes();
    }
}

0 references
public async void AddRoomTypeClicked(object sender, EventArgs e)
{
    var roomTypeName = roomTypePicker.SelectedItem?.ToString();

    if (!string.IsNullOrEmpty(roomTypeName))
    {
        await _database.SaveItemAsync(new RoomType { Name = roomTypeName });
        LoadRoomTypes();
    }
}

```

Fig.10 - Descriptive names to avoid comments

2. From the example below it is possible to see that non-meaningful variables like `mock` or `db` will most likely need some further explanation, reason why I added the comment on top of it. This really superficial comment could be removed after applying **meaningful names** and renaming those two variables to something more meaningful that another developer who is reading can understand:

```

[Test]
0 references
public async Task DeleteItemAsync_DeletesItem()
{
    // Arrange
    // Ensure both mockDbConnection and undacDatabase are not null
    if (mock == null || db == null)
    {
        Assert.Fail("Mock or UndacDatabase is null.");
        return;
    }
}

[Test]
0 references
public async Task DeleteItemAsync_DeletesItem()
{
    // Arrange
    if (mockDbService == null || undacDatabase == null)
    {
        Assert.Fail("Mock or UndacDatabase is null.");
        return;
    }
}

```

Fig.11 - Meaningful names to avoid comments

3. The last one I will talk about is a very good example of applying **KISS**: as you can see I created an async function inside the method, which would ask the user for a new name, and afterwards I called it. This needed a comment as it might not seem clear to someone, but to solve this issue I simply tried to keep things simple and use the `DisplayPromptAsync()` method directly instead. That way no comment was needed anymore:

```
private async void EditRoomTypeClicked(object sender, EventArgs e)
{
    if (_currentRoomType is null) return;

    // Getting the new name to rename the room type
    var getNewName = async () => {
        var title = "Edit Room Type";
        var promptMessage = "Please enter the new name for the room type:";
        var confirmButtonLabel = "OK";
        var cancelButtonLabel = "Cancel";
        var defaultName = _currentRoomType.Name;

        var newName = await DisplayPromptAsync(title, promptMessage, confirmButtonLabel, cancelButtonLabel, defaultName);
        return newName;
    };
    var newName = await getNewName();

    if (!string.IsNullOrEmpty(newName))
    {
        _currentRoomType.Name = newName;
        await _database.SaveItemAsync(_currentRoomType);
        _currentRoomType = null;
        LoadRoomTypes();
    }
}

private async void EditRoomTypeClicked(object sender, EventArgs e)
{
    if (_currentRoomType is null) return;

    var newName = await DisplayPromptAsync("Edit Room Type", "Enter new name:", "OK", "Cancel", _currentRoomType.Name);

    if (!string.IsNullOrEmpty(newName))
    {
        _currentRoomType.Name = newName;
        await _database.SaveItemAsync(_currentRoomType);
        _currentRoomType = null;
        LoadRoomTypes();
    }
}
```

Fig.12 - KISS to avoid comments