Software Architecture & Design

**Helpdesk Mobile**

Bilal Uddin (33071276) - 646 Main Body Word Count

*30/11/2024*

Contents

Module 1 - Class Diagram & OOP Implementation

**1.1** Class Diagrams pg. 3

**1.2** OOP Implementation pg. 5

1.2.1 Benefits & Limitations

1.2.2 Workplace affects

1.2.3 Future Improvements

Module 2 - Design Principles

**2.1** Principles pg. 5

2.1.1 Single Responsibility Principle pg. 6

2.1.2 Open-closed Principle pg. 6

2.1.3 Liskov Substitution Principle pg. 7

Module 3 - Architecture Patterns

**3.1** Patterns pg. 7

**3.2** Reflection pg. 8

3.2.1 Workplace affects

3.2.2 Future Improvements

Module 4 - Design Patterns

**4.1** Creational pg. 8

4.1.1 Singleton

**4.2** Structural pg. 9

4.2.1 Facade & Repository Pattern

**4.3** Other Patterns pg. 9

1 - EXAMPLE MODULE TITLE NAME

1.1 - Class Diagrams

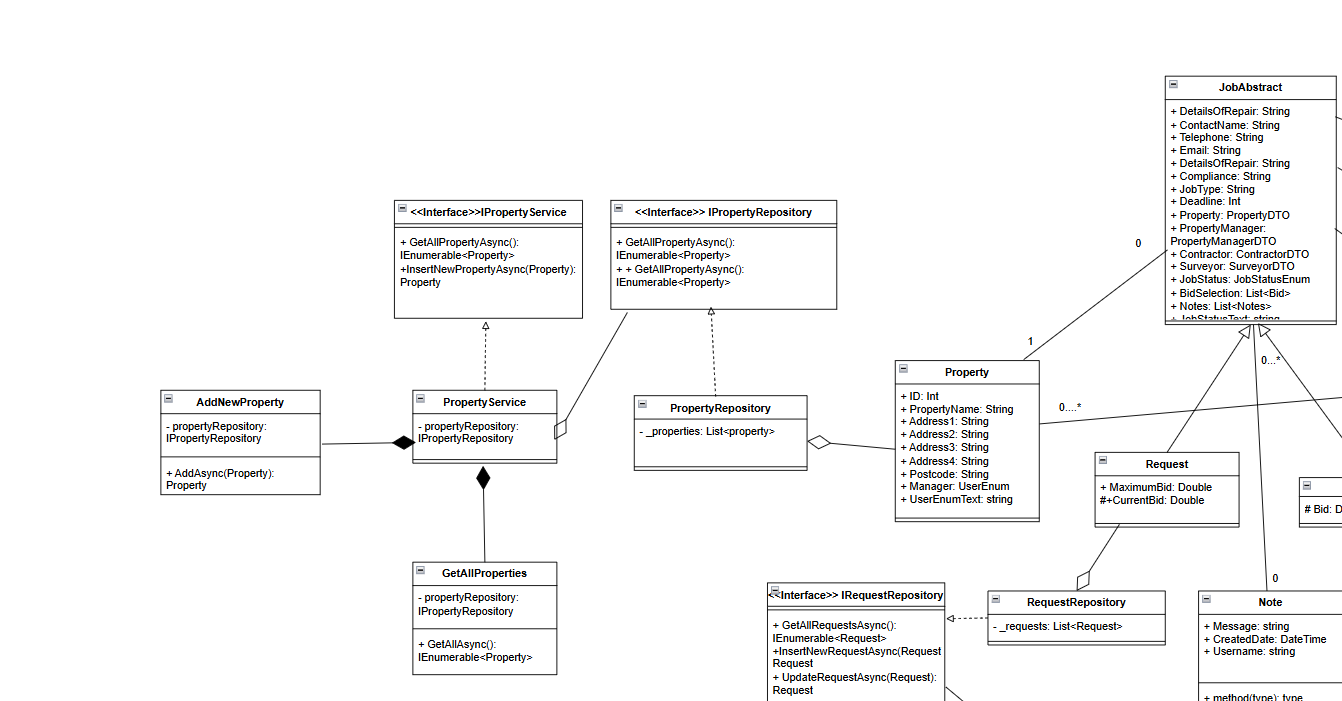


Figure 1.1.1 - Property Classes

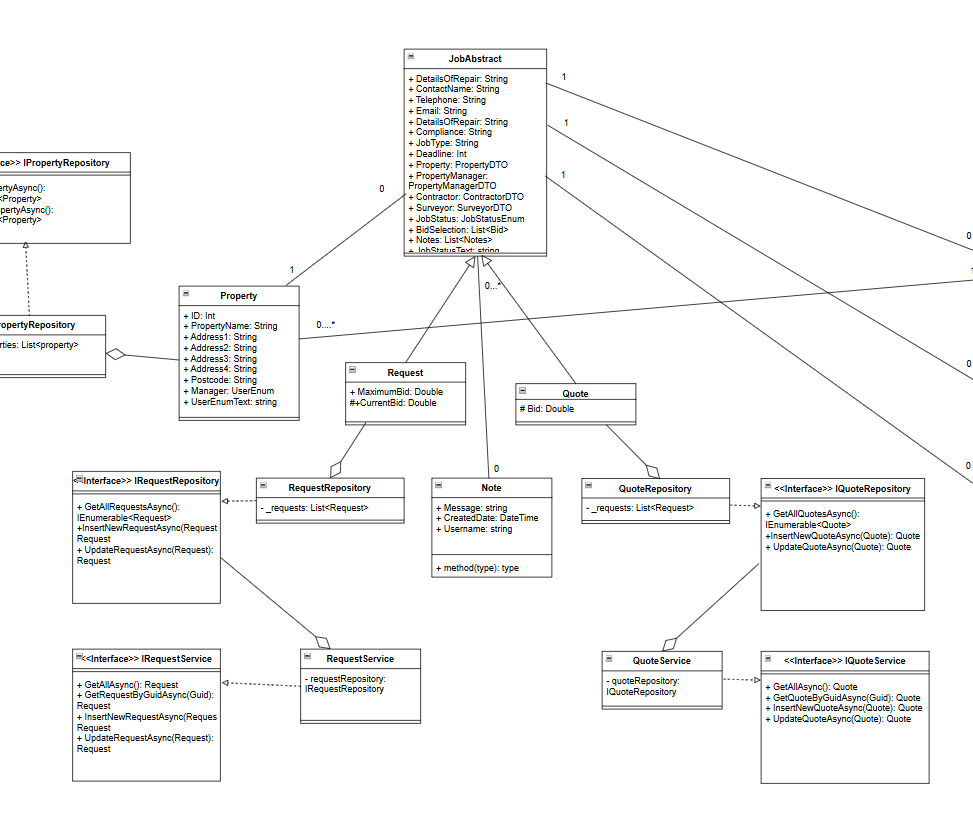


Figure 1.1.2 - Job Classes

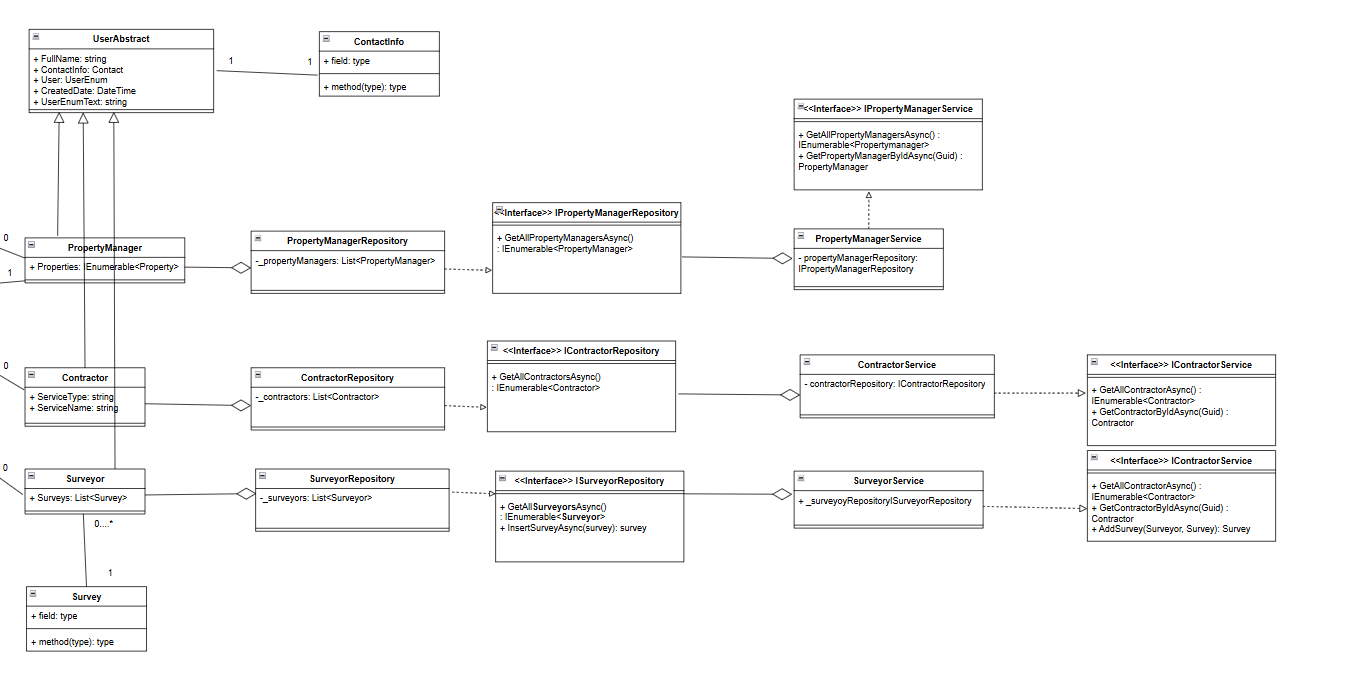


Figure 1.1.3 - User Classes

1.1 - OOP Implementation

**Benefits & Limitations**

Throughout the entire class diagram, it shows the pattern of the “entity” loosely associated with “repositories” and “repositories” loosely associated with “services”. This aggregated link is favoured due to maintaining important business data in the event a repository or service gets disposed of.

Figure 1.1.1 also demonstrates the composite link between the PropertyService and 2 other micro-services. In the event of the main service being disposed of, that should also be cascaded down to the micro-services due to the fact they’re not able to be utilised.

**Workplace Affects**

Scheming out the classes and their relationships before development has allowed greater clarity when working within the workplace, especially within a team.

**Future Improvements**

Even though I was able to effectively plan & design the diagrams before development, when it came time to develop, there were moments of difficulties and restrictions due to the rigidity of class diagrams. I believe moving forward it would be best to test functionality and ideas, then design the diagrams then develop further around those diagrams.

2 - Design Principles

2.1 - Principles

**Single Responsibility Principle**

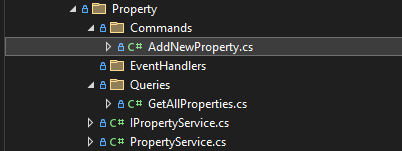
****

Figure 2.1.1 - Directory Of Property Services

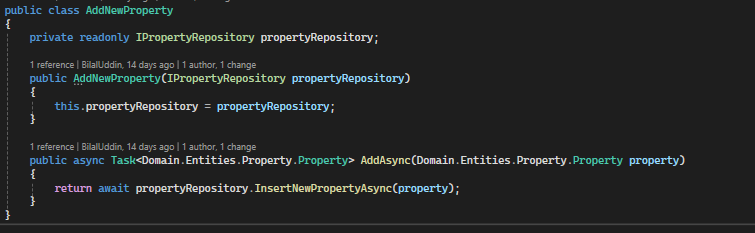


Figure 2.1.2 - Add New Property Micro-Services

Figure 2.1.1 shows an example of SRP in action. Each micro-service is designed to have one role and the consumed within the main PropertyService. This minimises the risk of merge conflicts when multiple developers work together and any unintended side effects.

**Open Closed Principle**

Unfortunately for this project, there are a lack of integration for this principle. In the future, the classes could be refactored to include a decorator pattern or use extension methods.

**Liskov Substitution Principle**

Referring back to figures 1.1.2 & 1.1.3 shows examples of inheritance and abstraction. Here the child classes act as an extension to the abstract class while maintaining all its base functionality.

3 - Architecture Patterns

3.1 - Patterns

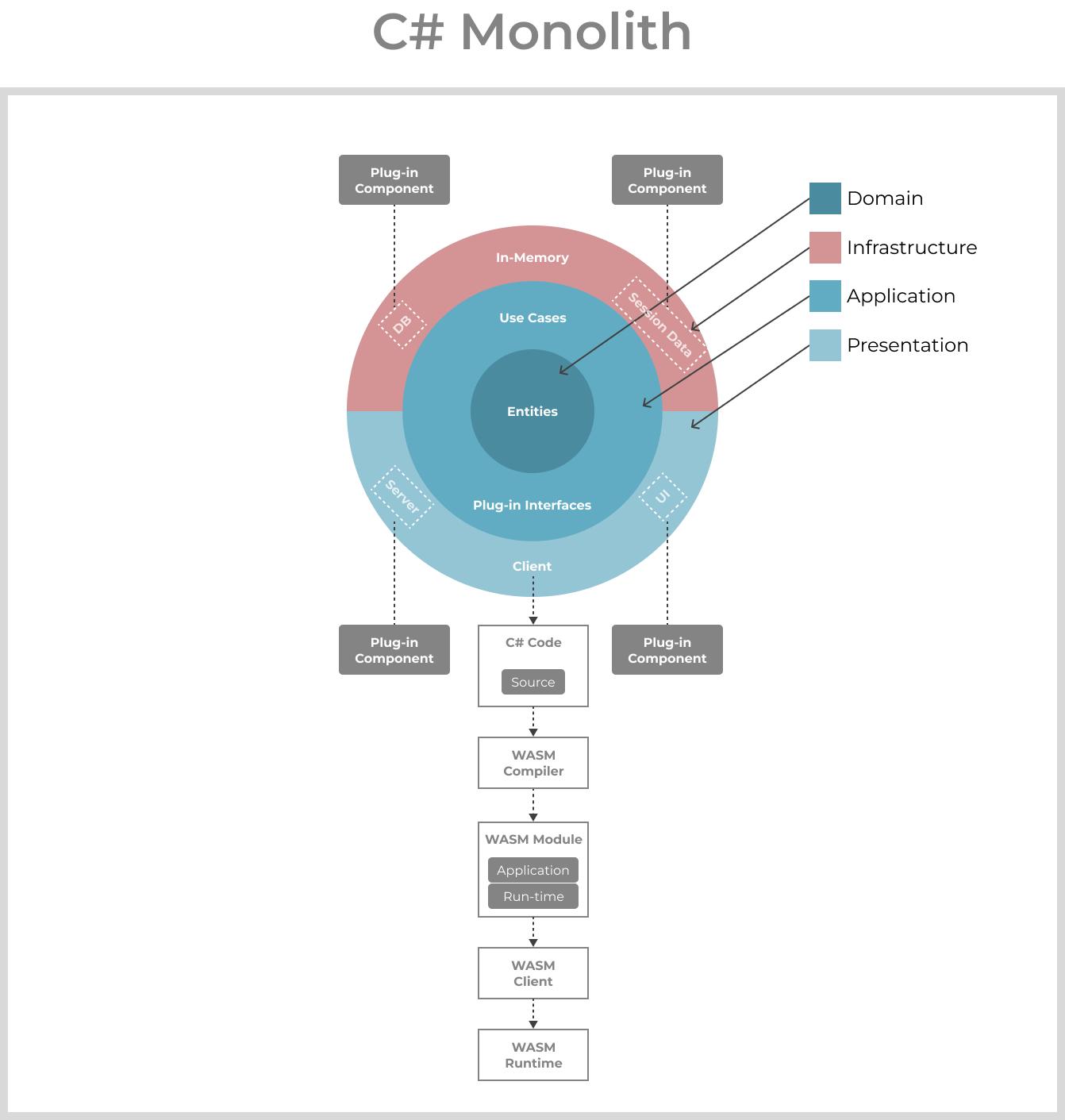


Figure 3.1.1

Benefits - business data & use cases are the core of the architecture and therefore can work alongside multiple different infrastructures (web, mobile, API etc). Due to the focus on the domain, the infrastructure & presentation layer allows for easy plugin implementation between various storage mediums/project types.

3.2 Reflection

**Workplace Affects**

The C# code is compiled into web assembly code which is executable on the browser. This allows the entire project to be done via C# which is very friendly for developers within the team who know little front-end frameworks/languages.

**Potential Improvements**

This form of architecture has strict dependencies and project references, where the core domain cannot have any. This makes it difficult to apply and run tests.

4 - Design Patterns

4.1 - Creational

**Singleton**

The “CurrentUserSingleton” and “UndoStack” stores hold the current in-memory data of the user which will be used within every razor file within the program. Therefore, it made most sense to set them as a singleton which is accessible within the client project.

A singleton benefitted here due to the stores also acting as state management which allowed the singletons to act as plugins within each component.

4.2 - Structural

**Facade & Repository Pattern**

This pattern is a specialisation of the facade pattern. It hides all the complexities of data access through a simple service interface.

This benefits the service user by simplifying and removing the need for worrying about how the data will be handled.

One issue with the repository pattern is that there are many parts with repeated code which violates the DRY principle. In the future, it may be beneficial to refactor the repositories with one generic base repository and use type generics to remove the repetition.

4.3 - Other Patterns

Composite Pattern (Structural)

Command (Behaviour)

Template Method (Behaviour)