[Cover Page]

[Declaration]

**Table of Contents**

1. Introduction……………………………………………………………………………….
2. Design Principles…………………………………………………………………………
3. Single Responsibility Principle
4. Open-Closed Principle
5. Liskov’s Substitution Principle
6. Interface Segregation Principle
7. Dependency Injection Principle
8. UML Diagram……………………………………………………………………………..
9. UML Sequence Diagram…………………………………………………………………..
10. Test cases and Results………………………………………………………………………
11. **Introduction**

**Restaurant Reservation and Point of Sale System (RRPSS)** is a console-based application that facilitates restaurant tasks through a digitalized system, assisting with organising and managing processes like booking reservations, recording orders, printing receipts, generating sales reports and more. It is intended to be used by restaurant staff to allow an easier workflow of tracking various tasks, as much of the information is stored and managed by the computer as opposed to manual analogue bookkeeping. This report discusses the design principles and considerations of a particular implementation of the RRPSS, as well as demonstrating its effectiveness with testcases.

1. **Design Principles**

A well-designed application is one that shows modularity and strong ease of modification and expansion. In the RRPSS system, there are many different elements that are conceptually different entities, but there are many dependencies between them. The design principles implemented aim to manage these dependencies in a way to ensure that the core framework is future-proof, where changes or additions to certain subsets of the system will not result in any radical changes to the core system and its working principles.

1. *Single Responsibility Principle*

Responsibility under the Single Responsibility Principle (SRP) can be understood as “reason to change”; it states that a class or module should only have one responsibility. The reasoning behind this principle is that if a class has too many responsibilities, then naturally it will be undergoing many changes whenever modifications to one part of the class is required. This can put unrelated segments of the system at risk whenever another segment is being modified, as breaking the class due to one change will also affect other segments that the class aims to implement. To prevent such scenarios, identifying different responsibilities are important, and different, separate classes should be implemented to handle differing responsibilities.

1. *Open-Closed Principle*

The Open-Closed Principle ensures that modules are *open for extension but closed for modification.* Open for extension allows us to add new features and standalone modifications as our needs arise, while closed for modification dictates that the original framework or source code template remains unchanged. This provides a layer of protection against unnecessary reliance between different modules/features. If the source code and template were to be modified in the process of implementing a new feature, these changes may end up causing unintended repercussions in other modules which may require parts of the original source code that has been changed. The use of *interfaces* and *abstract classes* in the project facilitate the Open-Closed Principle implementation by ensuring different classes can implement different versions of abstract methods.