[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 (OCP) 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* like the **Printer** and **UserInterfacePrinter** respectively facilitate the OCP, by allowing different classes to implement different submenus for their respective functions (OrderSubMenu prints the sub-menu for order functions in the console, ReservationMenu prints the sub-menu for reservation functions, etc.) without affecting other subclasses.

1. *Liskov’s Substitution Principle*

Liskov’s Substitution Principle (LSP) states that any property present in a superclass must also be true for its subclasses, thus implying that the superclass must be completely replaceable by its subclasses without causing any breakage. Thus the subclass also cannot introduce additional restrictions or parameters that cannot be handled by the superclass.

This can be seen in the implementation of **AMenuItem** and its subclasses **MainDish, Dessert** and **Beverage**, where the subclasses have different enumeration variables already defined in the superclass to act as identifiers but are functionally the same as their superclass and can be stored in data structures that are meant for **AMenuItem** like **ArrayLists** of type **AMenuItem**.

1. *Interface Segregation Principle*

The Interface Segregation Principle (ISP) states that classes should not depend on methods they do not use. This applies to interfaces, where they should not be implemented with methods such that there exist classes that may use some but not all methods in the interface. This is to reduce the dependency of the class on methods, as a change in an unused method in the interface breaking the interface will affect the unrelated class.  
  
The implementation of the **Printer** interface only uses one method, *print()* for various SubMenu classes to implement. All classes that use the Printer interface implement this method and there are no other unrelated methods that do not associate with the concept of printing inside the interface, thus strictly adhering to the ISP.

1. *Don’t Repeat Yourself*

The Don’t Repeat Yourself (DRY) Principle ensures that we do not add or implement what already exists; thus a functionality or value that can be reused in different places should not be reimplemented, only referenced.

The subclasses **MenuSubMenu**, **OrderSubMenu** and **ReservationSubMenu** all require the **Scanner** object to receive user input to perform their various functions. Instead of creating a new Scanner object for each and every one of them, the Scanner is initialized in their common superclass, UserInterfacePrinter and referenced by the three subclasses, thereby eliminating repetition and adhering to the DRY principle.