Declaration of Original Work for SC2002 Assignment

We hereby declare that the attached group assignment has been researched, undertaken, completed, and submitted as a collective effort by the group members listed below.

We have honored the principles of academic integrity and have upheld Student Code of Academic Conduct in the completion of this work.

We understand that if plagiarism is found in the assignment, then lower marks or no marks will be awarded for the assessed work. In addition, disciplinary actions may be taken.

Name	Course	Lab Group	Signature / Date
Ewen Cheung Yi Wen	DSAI	FDAD	Ewen / 22 April 2025
Jerick Ho Cheng Hien	BCG	FDAD	Jerick / 22 April 2025
Chan Yun Han	DSAI	FDAD	YunHan / 22 April 2025
Makhija Eshaa Jiten	DSAI	FDAD	Eshaa / 22 April 2025
Jusvin Adrian Tan	DSAI	FDAD	Jusvin / 22 April 2025

Important notes:

1. Name must EXACTLY MATCH the one printed on your Matriculation Card.
2. Student Code of Academic Conduct includes the latest guidelines on usage of Generative AI and any other guidelines as released by NTU.

1.0 Design Considerations

1.1 Approach taken

Our team approached the BTO Management System project with a focus on creating a modular, maintainable, and extensible application that adheres to Object-Oriented Design principles. We employed a layered architecture with clear separation of concerns:

1. User Interface Layer: Implemented through menu classes that handle user interactions
2. Feature Access Layer: Interface-based design allowing role-specific access to features
3. Business Logic Layer: Handlers implementing the feature interfaces with business logic
4. Data Access Layer: File I/O operations for persistent storage

This architecture allows for components to be developed and tested independently, making the system more maintainable and enabling parallel development by team members.

1.2 Principles Used

1.2.1 Encapsulation

We have used encapsulation principles consistently across our entire BTO Management System project. This principle has been applied to all our classes to protect data integrity and control access to internal states:

```
public abstract class User {
    // Private attributes - hidden from outside access
    private String name;
    private String nric;
    private String password;
    // Additional attributes...

    // Controlled access through getters
    public String getNric() {
        return nric;
    }

    // Validation in setters
    public void setPassword(String password) {
        if (password != null && !password.trim().isEmpty()) {
            this.password = password;
        } else {
            throw new IllegalArgumentException("Password cannot be empty");
        }
    }
    // Other getters and setters...
}
```

We've implemented encapsulation throughout our system in several key ways:

1. Data Protection: All model classes (Project, Application, Enquiry) have private attributes with controlled access methods. This prevents invalid data from being stored and maintains object integrity.
2. Validation Logic: Setters include validation rules to ensure only valid data is stored. For example, password changes must meet security requirements, and age values must be within realistic ranges.
3. Information Hiding: Implementation details are hidden from client code. For instance, the internal representation of application status transitions is encapsulated within the ApplicationHandler, so only valid transitions are allowed.
4. Access Control: Different user types have different access levels to system data. An Officer cannot directly modify application outcomes, and an Applicant cannot see other users' application details - these constraints are enforced through encapsulation.

By making attributes private and providing controlled access through getters and setters across our project, we prevent direct manipulation of critical data and ensure all changes go through proper validation channels. This has improved the security and reliability of our application.

1.2.2 Inheritance

Inheritance has been leveraged throughout our project to create clear hierarchies and promote code reuse:

```java
public abstract class User {
    // Common attributes for all user types
    private String name;
    private String nric;
    // Other attributes...

    // Common method for all user types
    public boolean authenticate(String inputPassword) {
        return password.equals(inputPassword);
    }

    // Abstract method to be implemented by child classes
    public abstract boolean canApplyForProject(Project project);
}

public class Applicant extends User {
    // Constructor reuses parent class constructor
    public Applicant(String name, String nric, /* other params */) {
        super(name, nric, /* other params */);
    }

    // Implementation of abstract method specific to Applicants
    @Override
    public boolean canApplyForProject(Project project) {
        return project.isApplicationPeriodOpen() &&
               this.getAge() >= 21 &&
               !project.hasApplied(this.getNric());
    }
    // Applicant-specific methods...
}
```

```java
public class HDBOfficer extends User {
    private List<String> assignedProjects;

    // Officer-specific methods
    public void addAssignedProject(String projectName) {
        assignedProjects.add(projectName);
    }

    // Implementation of abstract method specific to Officers
    @Override
    public boolean canApplyForProject(Project project) {
        // Officer-specific implementation
        // ...
    }
    // More Officer-specific methods...
}
```

Our inheritance implementation provides several benefits throughout the system:

1. Code Reuse: Common user attributes and behaviors are defined once in the parent class and inherited by all user types, eliminating duplication.

2. Specialized Behavior: Each user type extends the base functionality with specialized behaviors. For example, only HDBOfficers maintain a list of assigned projects.
3. Polymorphic Treatment: The system can treat different user types uniformly where appropriate, such as during authentication, while allowing specialized processing where needed.
4. Type Hierarchies: Beyond users, we've applied inheritance to create hierarchies for other system components, such as menu classes and feature implementations.
5. Method Overriding: Child classes override parent methods to provide specialized implementations while maintaining the same interface, ensuring consistency across the system.

This approach has allowed us to reuse code for common functionality while specializing behavior for various entity types throughout the entire application.

1.2.3 Polymorphism

We used polymorphism extensively through interfaces and method overriding to create a flexible and extensible system:

```java
// Interface defining project features for different user roles
public interface ApplicantProjectFeatures {
    List<Project> getVisibleProjects();
    // More applicant-specific methods...
}

public interface OfficerProjectFeatures {
    List<Project> getProjectsForOfficer(String officerNric);
    // More officer-specific methods...
}

// Handler implementing multiple interfaces
public class ProjectHandler implements ApplicantProjectFeatures, OfficerProjectFeatures {
    private List<Project> projects;

    @Override
    public List<Project> getVisibleProjects() {
        // Implementation for applicants
        return projects.stream()
                .filter(Project::isApplicationPeriodOpen)
                .collect(Collectors.toList());
    }

    @Override
    public List<Project> getProjectsForOfficer(String officerNric) {
        // Different implementation for officers
        return projects.stream()
                .filter(p -> p.isOfficerAssigned(officerNric))
                .collect(Collectors.toList());
    }

    // More implementations...
}
```

Polymorphism has been implemented throughout our system in several ways:

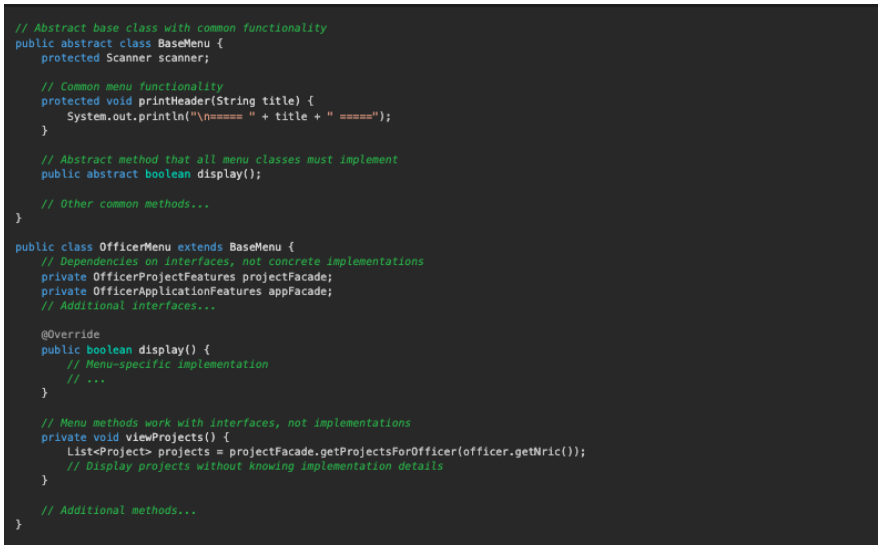
1. Interface-based Design: We defined specialized interfaces for each user role (Applicant, Officer, Manager), allowing different implementations while maintaining consistent method signatures.
2. Dynamic Method Dispatch: At runtime, the appropriate method implementation is called based on the actual object type, not the reference type.

3. Method Overloading: We provide multiple versions of methods with different parameters to handle various use cases.
4. Method Overriding: Child classes override parent methods to provide specialized behavior while maintaining the same interface.
5. Polymorphic Collections: Our system uses collections of interface types to store and process objects of different concrete types.

This polymorphic approach enables runtime binding of methods based on the specific interface being used, making our system more flexible and extensible.

1.2.4 Abstraction

We used abstraction to hide implementation details behind interfaces and abstract classes:



```
// Abstract base class with common functionality
public abstract class BaseMenu {
    protected Scanner scanner;

    // Common menu functionality
    protected void printHeader(String title) {
        System.out.println("\n===== " + title + " =====");
    }

    // Abstract method that all menu classes must implement
    public abstract boolean display();

    // Other common methods...
}

public class OfficerMenu extends BaseMenu {
    // Dependencies on interfaces, not concrete implementations
    private OfficerProjectFeatures projectFacade;
    private OfficerApplicationFeatures appFacade;
    // Additional interfaces...

    @Override
    public boolean display() {
        // Menu-specific implementation
        // ...
    }

    // Menu methods work with interfaces, not implementations
    private void viewProjects() {
        List<Project> projects = projectFacade.getProjectsForOfficer(officer.getNric());
        // Display projects without knowing implementation details
    }

    // Additional methods...
}
```

Abstraction has been implemented in our system through several mechanisms:

1. Abstract Classes: We used abstract classes like User and BaseMenu to define common attributes and behaviors while deferring specific implementations to child classes.
2. Interface-based Design: Our system defines functionality through interfaces, allowing multiple implementations and hiding implementation details from client code.
3. Feature Facades: Handler classes implement multiple interfaces to provide simplified access to complex system operations.
4. Information Hiding: Complex processing logic is hidden from UI components. For example, menus don't know how applications are stored or how status transitions are validated.

5. Data Abstractions: Complex data types are presented as high-level abstractions with meaningful operations rather than exposing their internal structure.

This approach allows the system to work with high-level concepts without needing to know implementation details.

1.3 SOLID implementation

1.3.1 Single Responsibility Principle (SRP)

Each class in our system has a single, well-defined responsibility.
For example from ApplicationHandler:

```
// ApplicationHandler focuses solely on application management
public class ApplicationHandler implements ManagerApplicationFeatures,
                                           OfficerApplicationFeatures,
                                           ApplicantApplicationFeatures {

    private List<Application> applications;
    private ApplicationSerializer serializer; // For persistence

    // One of many focused methods
    @Override
    public void processApplication(String applicationId) {
        Application app = findApplicationById(applicationId);
        // Validation and processing logic...
        app.setStatus(ApplicationStatus.BOOKED);
        saveChanges();
    }

    // File I/O responsibility is delegated to a separate class
    private void saveChanges() {
        serializer.saveToFile(applications);
    }

    // Other methods...
}
```

Our system implements SRP through clear separation of concerns:

1. Model Classes: Focus solely on representing domain entities (Project, Application, Enquiry).
2. Handler Classes: Each handler manages a specific business functionality.
3. Serializer Classes: Dedicated to data persistence concerns.
4. Factory Classes: Responsible exclusively for creating complex objects from raw data.
5. Menu Classes: Focus only on user interaction, delegating business logic to appropriate handlers.

1.3.2 Open/Closed Principle (OCP)

Our design is open for extension but closed for modification.

```
// Enum for application statuses - can be extended without changing application logic
public enum ApplicationStatus {
    PENDING("Pending"),
    UNDER_REVIEW("Under Review"),
    // Other statuses...
    BOOKED("Booked"),
    WITHDRAWN("Withdrawn");

    private final String displayName;
    // Constructor and methods...
}

// StatusTransitionHandler - encapsulates transition rules
public class StatusTransitionHandler {
    private static Map<UserType, Map<ApplicationStatus, Set<ApplicationStatus>>> allowedTransitions;

    // Validation method that doesn't change when new statuses are added
    public static boolean isTransitionAllowed(UserType userType,
                                              ApplicationStatus currentStatus,
                                              ApplicationStatus newStatus) {
        // Logic to check if transition is allowed
        // ...
    }
}
```

We've applied OCP in several ways:

1. Interface-based Design: New functionality can be added by creating new implementations without modifying client code.
2. Strategy Pattern: For features like application processing.
3. Enum-based Configurations: Status types and other enumerations can be extended without modifying the code that uses them.
4. Plugin Architecture: Handler classes can be extended through composition rather than modification.

1.3.3 Liskov Substitution Principle (LSP)

Child classes can be used wherever their parent classes are expected:

```
// MainMenu uses LSP to handle all user types interchangeably
private void handleUserSession(User user) {
    // User reference can be any User subtype
    if (user.getUserType() == UserType.APPLICANT) {
        new ApplicantMenu((Applicant) user, /* dependencies */).display();
    } else if (user.getUserType() == UserType.MANAGER) {
        new ManagerMenu((ProjectManager) user, /* dependencies */).display();
    } else if (user.getUserType() == UserType.OFFICER) {
        // Special handling for officers who can switch between roles
        // ...
    }
}
```

Our LSP implementation includes:

1. Behavioral Consistency: Child classes maintain the behavior guaranteed by their parent classes.
2. Precondition Preservation: Child classes don't strengthen preconditions.
3. Postcondition Preservation: Child classes maintain all guarantees of the parent class.

4. Exception Handling Consistency: Child classes don't throw exceptions not expected from the parent class.

1.3.4 Interface Segregation Principle (ISP)

We designed multiple focused interfaces tailored to specific client needs:

```
// Segregated interfaces for application features by user role
public interface ApplicantApplicationFeatures {
    String submitApplication(String applicantNric, String projectName, String unitType);
    List<Application> getApplicationsForApplicant(String applicantNric);
    // Only methods needed by applicants...
}

public interface OfficerApplicationFeatures {
    List<Application> getApplicationsForProject(String projectName);
    void processApplication(String applicationId);
    // Only methods needed by officers...
}

public interface ManagerApplicationFeatures {
    void reviewApplication(String applicationId, ApplicationStatus newStatus, String remarks);
    List<Application> getAllApplications();
    // Only methods needed by managers...
}

// Single implementation class
public class ApplicationHandler implements
    ApplicantApplicationFeatures,
    OfficerApplicationFeatures,
    ManagerApplicationFeatures {
    // Implementation of all interface methods
}
```

Our application of ISP includes:

1. Role-based Interfaces: Separate interfaces for each user role that include only relevant operations.
2. Feature-based Segregation: Each major feature area has its own set of interfaces.
3. Granular Interfaces: Interfaces are kept small and focused on specific capabilities.
4. Multiple Interface Implementation: Handler classes implement multiple interfaces for different clients.

1.3.5 Dependency Inversion Principle (DIP)

High-level modules depend on abstractions rather than concrete implementations:

```java
// OfficerMenu depends on abstractions, not concrete implementations
public class OfficerMenu {
    // Dependencies on interfaces instead of concrete classes
    private OfficerProjectFeatures projectFacade;
    private OfficerApplicationFeatures appFacade;
    // Other interfaces...

    // Constructor injection of dependencies
    public OfficerMenu(HDBOfficer officer,
                       OfficerProjectFeatures projectFacade,
                       OfficerApplicationFeatures appFacade,
                       /* other dependencies */) {
        // Initialize with injected dependencies
        this.projectFacade = projectFacade;
        this.appFacade = appFacade;
        // ...
    }

    // Methods use the interfaces, not concrete implementations
    private void viewProjects() {
        List<Project> projects = projectFacade.getProjectsForOfficer(officer.getNric());
        // Use projects without knowing implementation details
    }
}
```

Our implementation of DIP includes:

1. Interface-based Design: High-level modules like menus depend on interfaces rather than concrete classes.
2. Constructor Injection: Dependencies are injected through constructors rather than being created directly.
3. Abstraction Layers: Feature handlers implement multiple interfaces for different clients.
4. Composition Root: Dependencies are composed in the Main class, which serves as the composition root.

1.4 Assumptions Made

1. Data Persistence: We assumed text-based file storage would be sufficient for data persistence, with no need for database systems.
2. User Authentication: We assumed a simple username/password system would be adequate for demonstration purposes.
3. User Interface: We assumed a command-line interface would be sufficient as specified, without graphic user interface requirements.
4. Default Values: We assumed some default values for pricing and unit configurations to simplify implementation.

1.5 Additional Features Implemented

Beyond the core requirements specified in the assignment, we implemented several additional features to enhance the system's functionality:

1. Password Change Functionality: All user types can change their passwords through a secure interface
2. Advanced Filtering Options: Projects can be filtered by multiple criteria including neighborhood, flat type, and price range
3. Detailed Application History: System maintains a detailed history of application status changes
4. Receipt Generation and Management: Officers can generate and save booking receipts for successful applications
5. User-Friendly CLI Interface: Enhanced console display with formatted tables and visual separators
6. Input Validation: Comprehensive validation for all user inputs to prevent invalid data
7. Custom File Export Formats: Reports can be exported in various text formats for external use

2.0 Detailed UML Class Diagram

The UML Class Diagram represents the relationships between the major components of our system:

Class Relationships

- User Hierarchy: Abstract User class extended by Applicant, HDBOfficer, and ProjectManager
- Model Classes: Project, Application, Enquiry, OfficerRegistration, WithdrawalRequest
- Handler Classes: ProjectHandler, ApplicationHandler, EnquiryHandler, etc.
- Feature Interfaces: ApplicantProjectFeatures, OfficerProjectFeatures, ManagerProjectFeatures, etc.
- Menu Classes: MainMenu, ApplicantMenu, OfficerMenu, ManagerMenu

Key Design Patterns

- Facade Pattern: Handler classes provide simplified interfaces to complex subsystems
- Factory Pattern: Factory classes for creating complex objects
- Serializer Pattern: Serializer classes handle conversion between objects and storage format

To view the full class diagram, follow this link to access the class diagram folder.

3.0 Detailed UML Sequence Diagram

The UML Sequence Diagrams illustrates the flow of the HDB Officer's role in applying for a BTO and registering to handle a project:

Applying for a BTO

This illustrates the flow of the HDB Officer in applying for a BTO:

1. User Authentication: Officer logs in through the authentication system (in file "Login.pdf" attached)
2. Menu Navigation: Officer selects to submit application for BTO as applicant
3. Project Selection: System filters and displays available projects and officer selects one
4. Application Submission: Officer confirms application
5. Registration Processing: System creates and stores application record
6. Status Notification: System confirms application submission to officer

Register for Project

This illustrates the flow of the HDB Officer in registering to handle a project:

1. User Authentication: Officer logs in through the authentication system
2. Menu Navigation: Officer selects to register for a project
3. Project Selection: System displays available projects and officer selects one
4. Registration Submission: Officer confirms registration
5. Registration Processing: System creates and stores registration record
6. Status Notification: System confirms registration submission to officer

3.1 Further notes for sequence diagrams:

To view the full sequence diagram, follow this link to access the sequence diagram folder.

4.0 Testing

To see our full test cases table, follow this link to access the test cases.

5.0 Reflection

5.1 Difficulties encountered

1. Complex User Role Requirements: Managing the overlapping capabilities of different user types required careful design consideration. We overcame this by using interface segregation and proper inheritance hierarchies.
2. Date-Based Logic: Implementing date logic for project application periods and registration overlap detection was challenging. We created utility classes to centralize date manipulation and validation.

3. Data Persistence: Maintaining data integrity across multiple CSV files without a proper database was difficult. We developed a robust file I/O system with error handling and data validation.
4. Booking Logic: Implementing the complex booking process with multiple status changes and validations was challenging. We used the state pattern and careful workflow design to manage transitions.

5.2 Knowledge Learned

1. Practical OO Design: We gained hands-on experience applying OO principles and Solid Principles to a real-world problem, seeing how proper design leads to maintainable code.
2. Interface-Based Programming: We learned the value of programming to interfaces rather than implementations for creating flexible, extensible systems.
3. File I/O Management: We developed skills in efficient file operations for data persistence without relying on database systems.
4. Collaborative Development: We learned effective practices for parallel development through clear interfaces and responsibility separation.

5.3 Further improvement suggestions

1. Database Integration: The system could benefit from proper database implementation for improved data integrity and query capabilities.
2. GUI Implementation: A graphical user interface would enhance user experience and make the system more accessible.
3. Authentication Enhancement: Implementing proper hashing for passwords and more sophisticated authentication mechanisms would improve security.
4. Notification System: Adding email or SMS notifications for application status changes would enhance the user experience.
5. Reporting and Analytics: More sophisticated reporting capabilities with visual analytics would provide better insights into application patterns.

6.0 Appendix

Github Repository :https://github.com/EwenCheung/SC2002-BTO-Management-System
To view the report with larger referenced images, view REPORT.md in the github repository.
For setup, follow Installation & Setup in the README.md in the github repository.