**BTO Management System**

SC2002 Object-Oriented Design & Programming Project

**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 honoured 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** |
| Chen XingWei | CSC | FCS3 |  |
| Afifuddin Yunior | MSE | FCS3 |  |
| Chloe Alexandria Lim Shi Ann | BCG | FCS3 |  |
| Allyson Leng Pei Wen | AISC | FCS3 |  |
| Balodi Shalok | CSC | FCS3 |  |

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. Design Considerations**

**1.1 Understanding the problem**

We first studied the requirements, identified all the entity classes, and noted down all the features that each of them has. From our understanding, the BTO Management System implements a BTO project application system for 3 users: Applicants, HDB Officer and HDB Manager. Applicants can view and apply for BTO projects, while HDB officers and HDB managers have an additional role of managing the projects. We employed the knowledge that we learnt about OODP and SOLID design principles to manage dependencies between classes and packages of classes to minimise the impact of change on other parts of the program.

**1.2 Explicit requirements and implicit expectations**

There were many requirements for this project, such as applying for a BTO and creating new BTO Projects. Thus, we decided to create a few packages to decompose the problem and segregate what each package should handle. The different packages are:

* **storage:** It stores all the files that handle the data that was provided. For example, the User Data, Project List, and the recorded BTO Applications.
* **users:** It stores all the files that are related to the different user types. For example, Applicant, HDB Officer, and HDB Manager.
* **ui:** It stores all the files that are related to the UX and UI. For example, handling of input and common messages that are used in the application.
* **system:** It stores the files that handle the flow of the application. For example, BTOSystem and Login.

**1.3 How we determined relationships**

**Users and Storage** **to** **BTOSystem**: we used a Has-a relationship and specifically Composition, as we think that the Users and Storage should not be able to exist without the BTOSystem, and by using composition, it also encourages encapsulation.

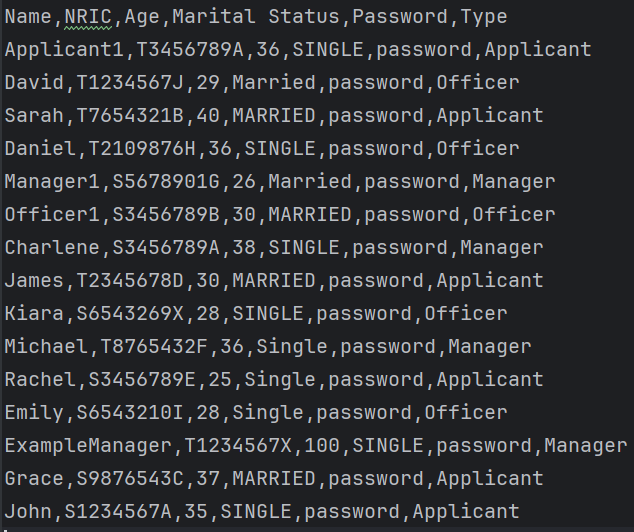
**Applicant, HDBOfficer, and HDBManager** **to User**: We used an is-a relationship, inheriting all the base functions that the abstract User has. So, this not only encourages code reusability, but it also allows for polymorphism. For example, the HDBOfficer inherits from Applicant. Both can view Enquiries, but HDBOfficer can view not just their Enquiries but also the Enquiries that belong to the Projects they are handling.

**All Users** **to Storage**:We used a use-a relationship for this, allowing them to rely on only 1 instance of the Storage that is created at the beginning of the program. This promotes loose coupling and the Dependency Inversion Principle (DIP), allowing for a more modular design.

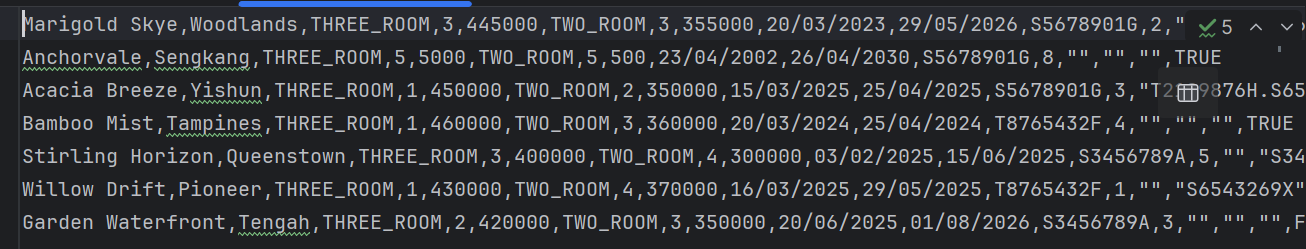
**1.3 Approach for storing data**

We decided to turn the Excel file that was provided by us into .csv files so that it is easier to read and write, and we also added a few more columns and created a few more .csv files to fulfil the requirements.

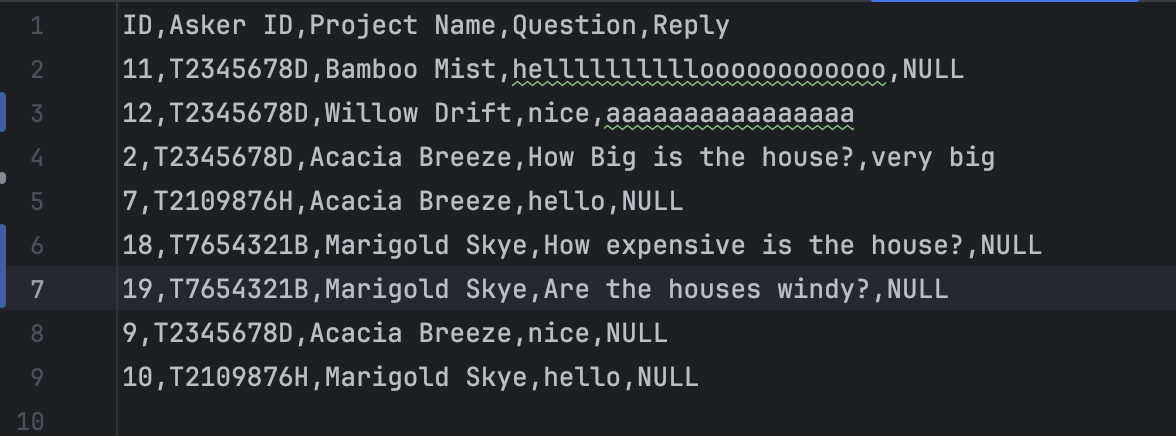
**UserList.csv:** Stores all info related to Users



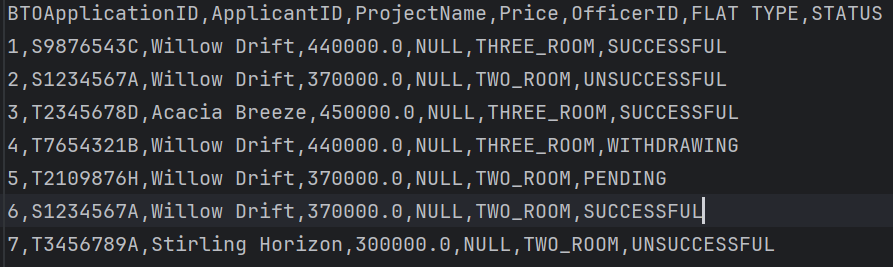
**ProjectList.csv:** Stores all info about different Projects



**Enquiry.csv:** Stores the different enquiries made by Users



**BTOApplication.csv:** Stores the info of BTO Applications made by Users



The main Data Structure that was used to store all the different data in the storage such as the UserList.csv was Map<String, Object>, as this allowed us to access the data we needed easily and it also makes reading and writing into the .csv easier as well without many trade-offs.

**1.4 Assumptions**

1. Users have unique and fixed User IDs.

2. The Project Name is always unique.

3. The Project Name and neighbourhood are in one set and cannot be changed.

4. The user’s Marital Status is either Single or Married.

5. The manager only manages BTO applications and withdrawals for their own projects.

6. Once the officer has been approved to manage a project, they can no longer withdraw.

**1.5 Design Trade-Offs**

We decided to only use and store the userID inside of objects like projectTeam instead of storing multiple user objects inside one projectTeam. This allowed for better security and extensibility when using the projectTeam object, while trading off with the simplicity that storing the user object can provide. One of the biggest challenges for this project was accessing and mutating the data that is inside the database. We heard that if we were to use Singleton, it would solve most of the issues regarding these problems. However, by using Singletons it will make the system less flexible, resulting in a potentially poor design. Hence, we chose to use composition and the Dependency Inversion Principle for the database so that if we were to add another kind of storage, the program will still work. Allowing it to be scalable but at the cost of increased difficulty to code.

**2. Object-Oriented Design**

**2.1 UML Class Diagram**

When reading the problem statement, we looked at nouns such as "User," "Project," "Application," "Enquiry," and "Registration". These entities form the backbone of the system. Specific system roles such as "HDB Officer," "Applicant," and "HDB Manager" were identified because they represent distinct users with different responsibilities. Enquiry represents the questions and responses between the applicant (or officer) and the system, while the projectdefines the BTO or housing project details.

**2.1 UML Sequence Diagram**

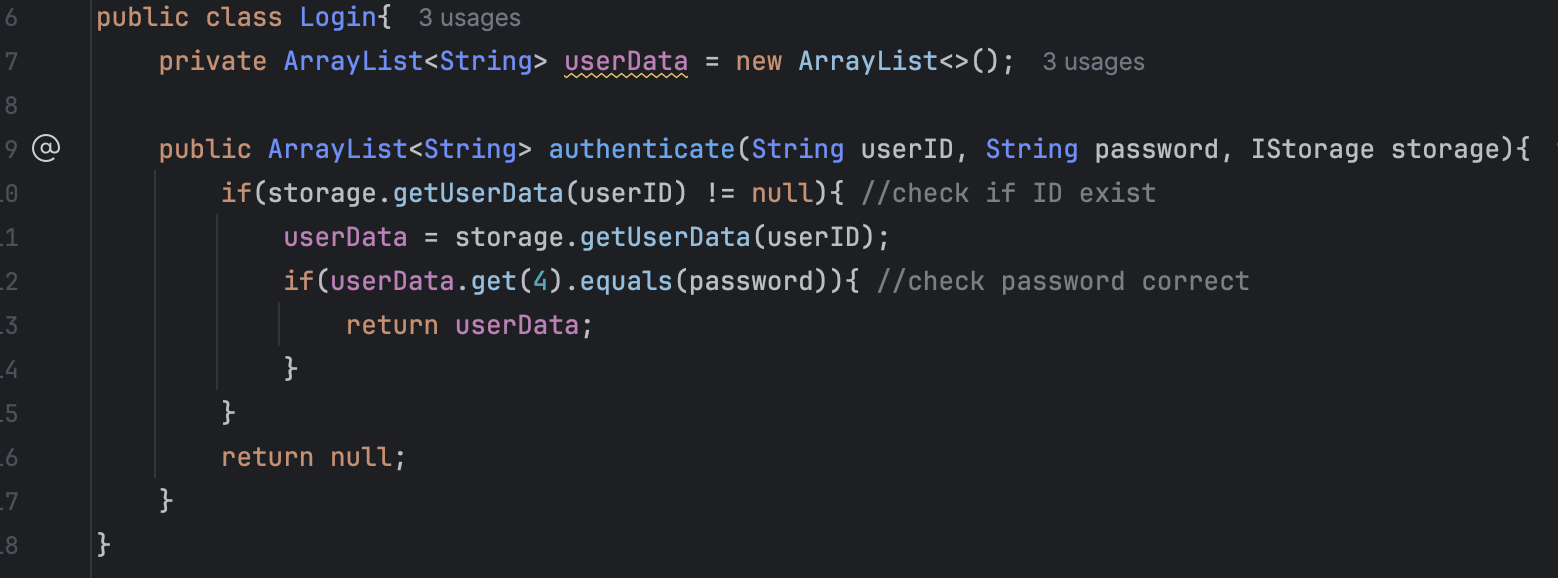
Link:<https://docs.google.com/spreadsheets/d/1rf49oRhrm2R8uH2RN43X4Of_rP-vz3hfxpq4VSKedtQ/edit?gid=304684806#gid=304684806>

The sequence diagram above is the action of the Officer to register for a Project. In this action, we are required to check the eligibility, such as not being an applicant, not already applying, and not being an officer in another project in the same time range.

**3. SOLID Design Principles**

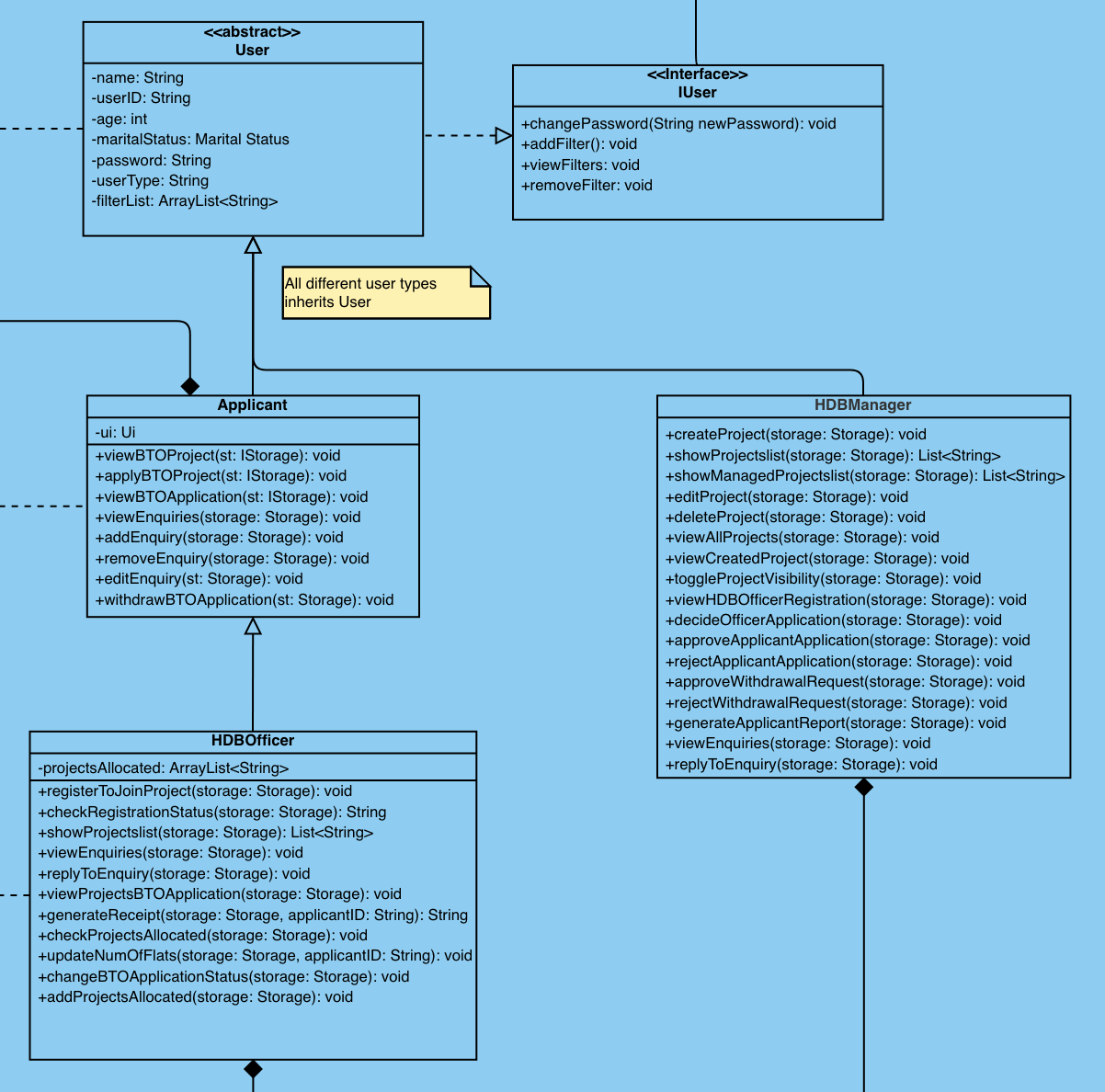
**3.1 Single Responsibility Principle (SRP)**

The Single Responsibility Principle states that every class should have a single responsibility or job or purpose. This ensures that the class is easier to read and understand, and more maintainable. An example in our code is the Login class, which only has 1 responsibility: to authenticate whether the login input is valid or not.



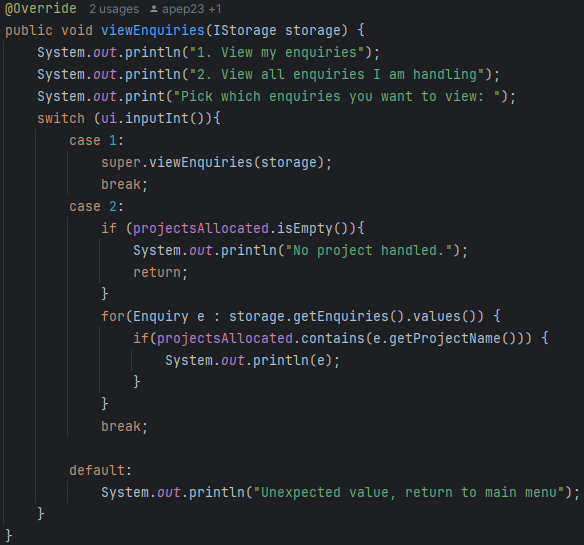
**3.2 Open/Closed Principle (OCP)**

The Open/Closed Principle states that classes should be open for extension but closed for modification, allowing for the addition of new functionality without changing the existing code. This can be done through inheritance and polymorphism. In our code, the Applicant class and the HDBManager class extend the abstract User class. The Applicant class extends the functionality of User by adding specific actions related to BTO projects, such as viewBTOProject() and applyBTOProject(), without modifying the original User class.



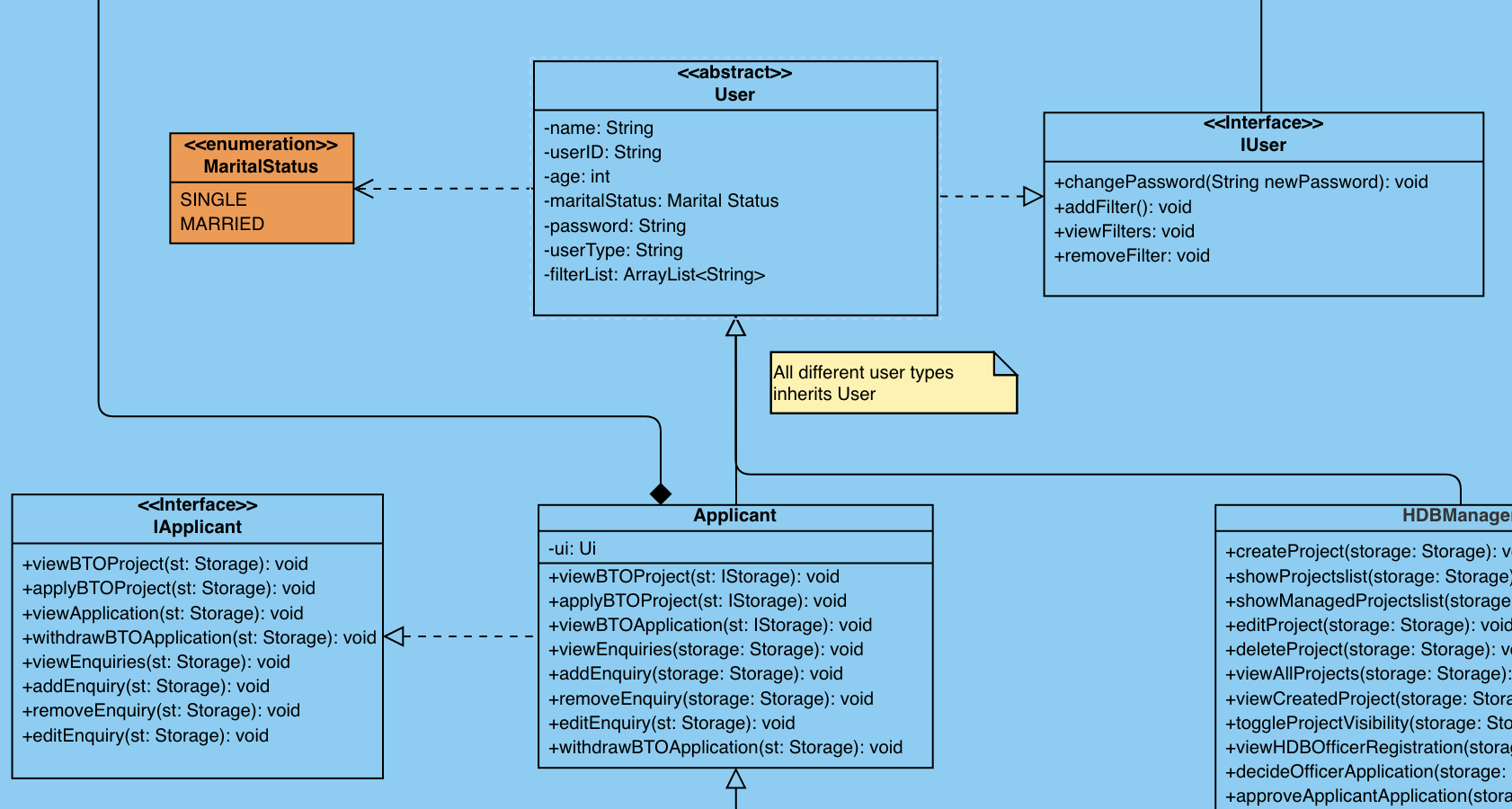
**3.3 Liskov Substitution Principle (LSP)**

The Liskov Substitution Principle states thatobjects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program. In our code, the class HDBOfficer extends the Applicant class, so HDBOfficer is a subclass of Applicant. Any instance of HDBOfficer can be used wherever an Applicant object is expected, without breaking the program. The HDBOfficer class inherits viewEnquiries() and overrides the functionality to suit the officer’s role. The Applicant can only view their enquiry, but the HDBOfficer can view and reply to all project-related enquiries and reply to the enquiry. Thus, an HDBOfficer object can substitute for an Applicant object without causing errors.



**3.4 Interface Segregation Principle (ISP)**

The Interface Segregation Principle states that many client-specific interfaces are better than one general-purpose interface. Larger interfaces should be split into smaller ones, so that implementing classes only need to be concerned about the methods that are of interest to them. We split the IUser interface into IUser and IApplicant interfaces. IApplicant is a specialised interface designed for the Applicant class that implements BTO application-related methods such as viewBTOProject() and applyBTOProject(). It doesn’t force HDBManager to implement methods they don’t need. Hence, no class should be implementing interfaces with abstract methods they do not need.



**3.5 Dependency Inversion Principle (DIP)**

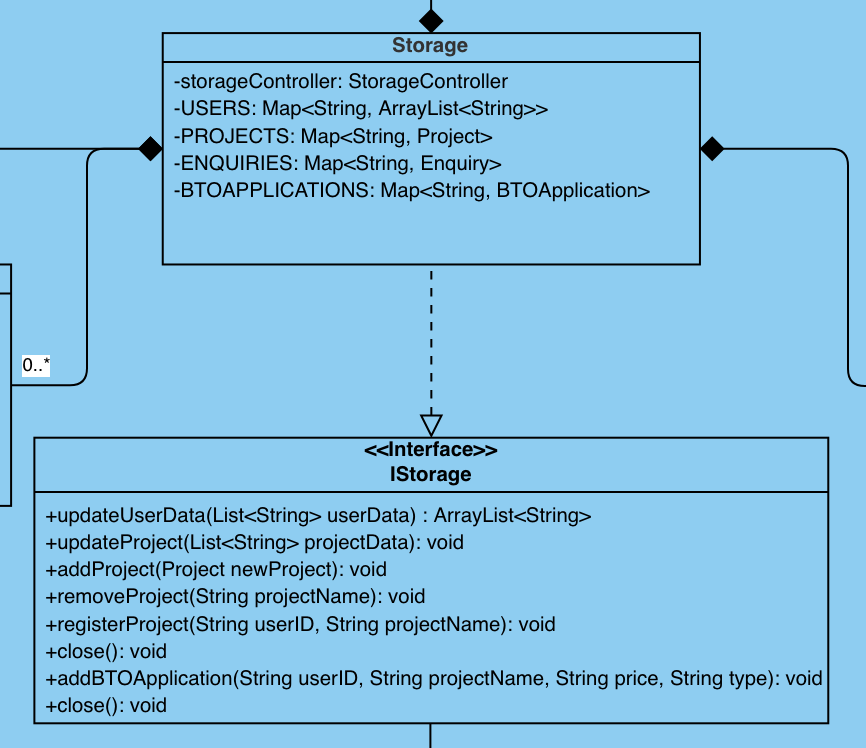
The Dependency Inversion Principle (DIP) states that both high-level and low-level modules should depend on abstractions. DIP in our BTO system ensures that high-level modules, such as business logic governing BTO applications and officer registrations, depend on abstractions rather than concrete implementations. For example, when an HDB Officer submits a BTO application, the application controller rather than calls methods on an abstract interface (e.g., IStorage or IDataRepository), directly invoking low-level database operations. This means that the business logic remains independent of the specific details of data storage.

The BTO System is the high-level module that defines all the actions applicants, officers, and managers can take. Within the BTO System, we create a new object of the Storage class, which will implement the methods in the IStorage interface. By using the IStorage interface, another class that implements IStorage can be used.

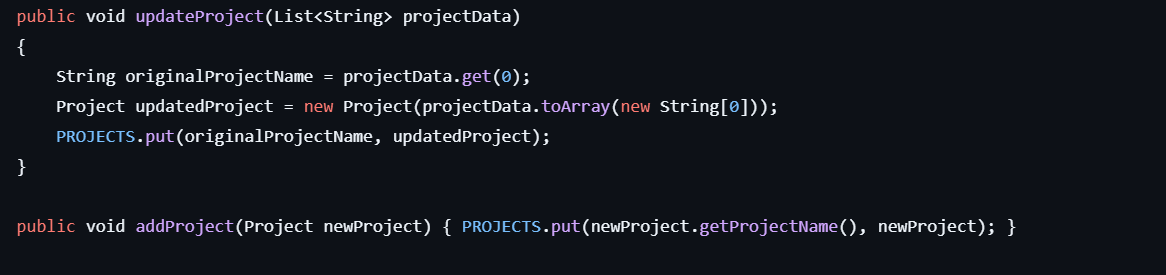
**4. Object-Oriented Design Principles**

**4.1 Abstraction**

The IStorage interface defines the essential methods updateProject() and addProject(), specifying what operations a storage system must support, but not how these operations are carried out. This promotes a clear separation between the definition and the implementation.



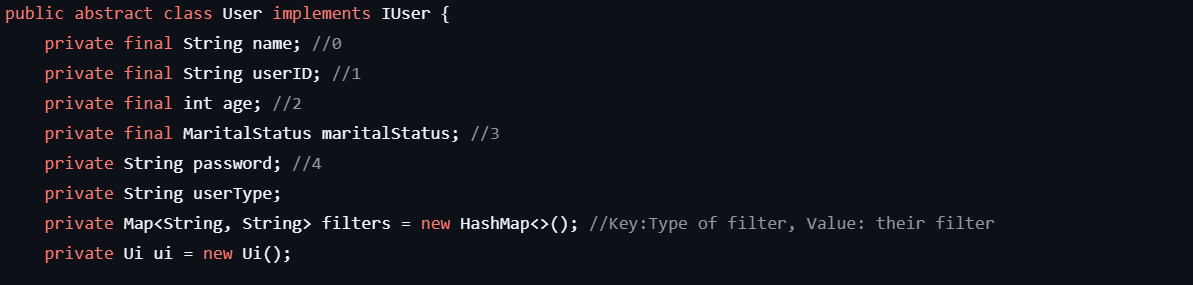
The actual logic, such as converting a list of project data into a Project object and storing it in a map, is provided in the Storage class, which implements the interface. By doing so, abstraction allows other parts of the program to interact with IStorage without needing to know the internal details of how the storage works.

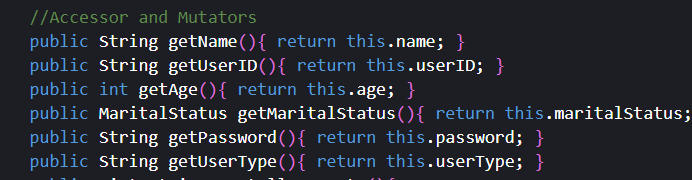


Implementation of abstract methods **updateProject** and **addProject** in the Storage class

**4.2 Encapsulation**

Encapsulation in the User class is achieved by restricting direct access to its internal data. For example, key attributes like name, userID, age, maritalStatus, and password are declared as private, meaning they cannot be accessed or changed directly by other classes. Similarly, the filters are maintained in a private Map and are manipulated through methods like addFilter(), removeFilter(), and viewFilters(), ensuring data integrity and fewer accidental modifications.



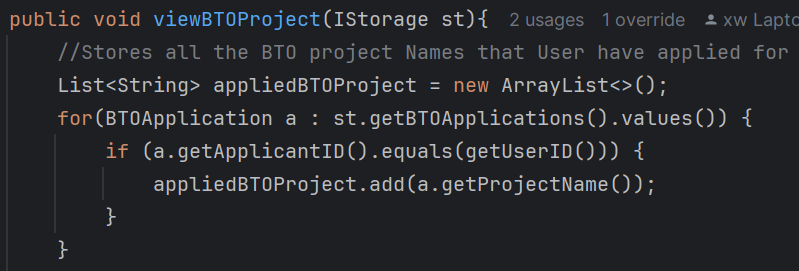


**4.3 Inheritance**

The Applicant class inherits all the fields and methods of the User class.



getUserID() method in the User class

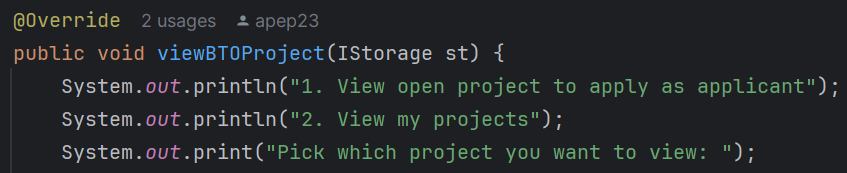


viewBTOProject() in Applicant class

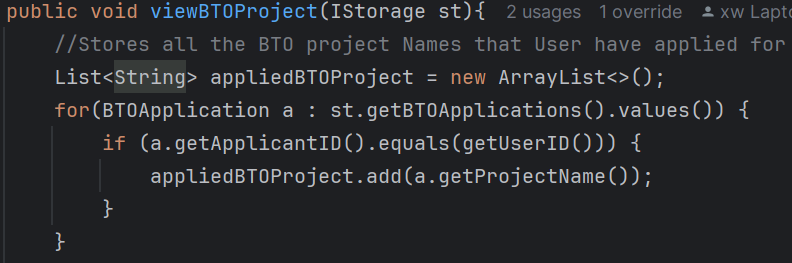
**4.4 Polymorphism**

Polymorphism means that objects of different classes can be used interchangeably because they share a common interface or superclass. In our BTO System, the high-level module defines the actions all users can take. For example, the Applicant class defines a method called viewBTOProject(), which is part of the user interface. The HDBOfficer class, which extends Applicant, overrides viewBTOProject() to incorporate additional options specific to officers. This design enables the BTO System to reference these objects using a common type such as Applicant and when it calls viewBTOProject(), the system dynamically determines and executes the appropriate version based on the actual object at runtime. As a result, the system remains flexible and extensible: if a new user role is introduced, its custom behavior can be integrated seamlessly by overriding the common method, thereby adhering to principles like low coupling and high cohesion.





viewBTOProject method in HDB Officer class



viewBTOProject() method in the Applicant class

**5 Test Cases**

Link: <https://docs.google.com/spreadsheets/d/1rf49oRhrm2R8uH2RN43X4Of_rP-vz3hfxpq4VSKedtQ/edit?usp=sharing>

**6 Reflection**

**6.1 Difficulties Encountered and the way to conquer**

We didn’t know what kind of data needed to be added to the base data that was provided, so that it would allow us to complete the specified requirement. We just started and worked with whatever we had, slowly discussing and figuring out what needed to be included. By doing this, it allowed us to only include things that are needed.

Another difficulty that we encountered was deciding whether the User class should use a has-a or a use-a relationship when using information from the storage. We first started off using a has-a relationship because it allowed ease of access. However, after working on it for a while we felt that by allowing the user to own the storage was not the best design, judging from a security standpoint. This is why we changed to giving it a use-a relationship, at the cost of making the BTOSystem class harder to maintain.

After running the first use case test, we found that our code hadn’t included all the requirements and the test cases. We need to repetitively test and edit our code to be able to cover all the test cases. For example, in the registerToJoinProject() method, at first, we thought that once the officer is registered in a project, they cannot apply as an applicant, so we added a new column in the project list and checked for each project whether it is overlapping with the current project.

While testing our initial system implemented in accordance with the assignment brief, we found that although the system was functional, it was inconvenient for the user.

Therefore, we added an additional feature for number selection for the projects and applications, printing the list of possible choices, and getting an integer input to choose, rather than requiring ID or names to be typed manually. Additionally, we added a feature of entering 0 to exit the function, in case the user misselected the action they wished to take.

Other features included allowing selection of visibility, rather than toggling, to avoid toggling to the incorrect visibility.

There were parsing issues with the dates for creating and editing projects, as the date formatting would accept any 4-digit combination. To combat this, we ensured that the years were limited, such that it was only recent years (30-year span from the current year), as well as checking that the closing date was only after the opening date.

We also added a “Withdrawal Rejected” status to the application status, so that applicants would be able to confirm the outcome of their withdrawal application.

**6.2 Further Improvements**

Some further improvements that we could consider is making the system more robust, so that it allows for projects with different numbers of Flat Types. For instance, projects with more types of Flat types, instead of just 2 rooms and 3 rooms. We could also add more features of the Projects, such as distance from the nearest MRT and cost per square meter/foot. Additionally, to allow projects to gain traction with users quickly, a feature could be added to showcase new projects at the top or recommend new projects that suit the applicants' or officers’ preferences.

**Individual Contributions**

|  |  |
| --- | --- |
| Name | Contributions |
| Chen XingWei | Project Management and Base Classes |
| Afifuddin Yunior | Sequence Diagram and Officer Class |
| Allyson Leng | Class Diagram and Report |
| Chloe Lim | HDB Manager Class and Reflection |
| Balodi Shalok | OO Design Principles and Colours Class |

**Appendix**

GitHub Link: <https://github.com/dozetype/BTO_Project>

Test Cases and Sequence Diagram: [SC2002 BTO Project](https://docs.google.com/spreadsheets/d/1rf49oRhrm2R8uH2RN43X4Of_rP-vz3hfxpq4VSKedtQ/edit?gid=304684806#gid=304684806)