XIAMEN UNIVERSITY MALAYSIA

Course Code	:	CST209
Course Name	:	Object-Oriented Programming – C++
Lecturer	:	Geetha Kanaparan
Academic Session	:	2024/09
Assessment Title	:	Final Project
Submission Due Date	:	30 December 2024, before 11.59 p.m.

Prepared by :

Student ID	Student Name
CYS2309202	Sun Zhengwu

Date Received :

Feedback from Lecturer:

Mark:

Own Work Declaration

I/We acknowledge that my/our work will be examined for plagiarism or any other form of academic misconduct, and that the digital copy of the work may be retained for future comparisons.

I/We confirm that all sources cited in this work have been correctly acknowledged and that I/we fully understand the serious consequences of any intentional or unintentional academic misconduct.

In addition, I/we affirm that this submission does not contain any materials generated by AI tools, including direct copying and pasting of text or paraphrasing. This work is my/our original creation, and it has not been based on any work of other students (past or present), nor has it been previously submitted to any other course or institution.

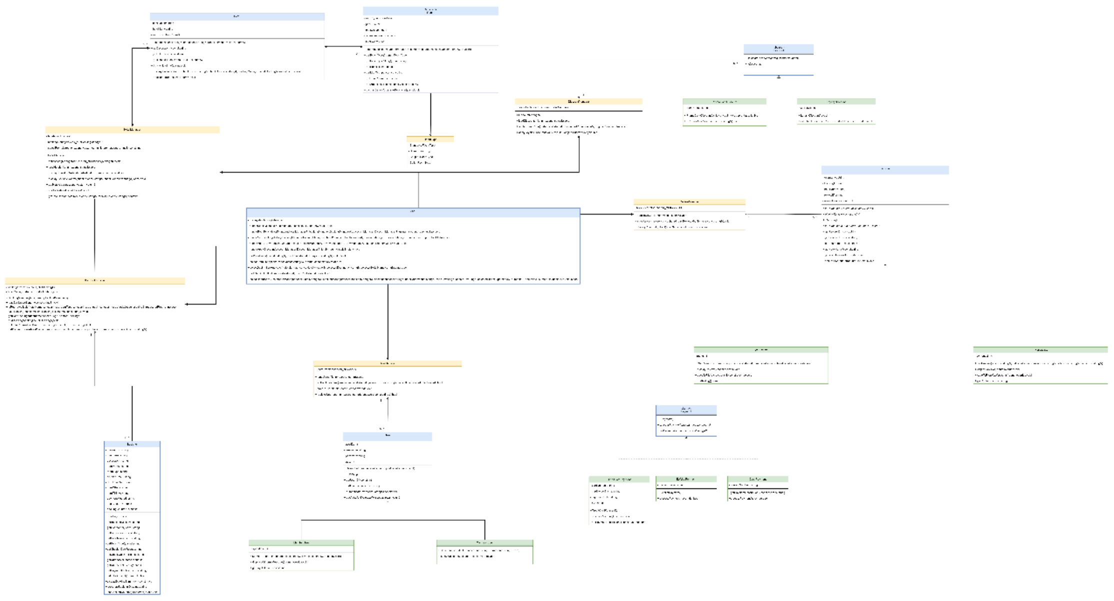
Signature: [illegible]

Date:2024-12-30

0. Content

1. UML diagram

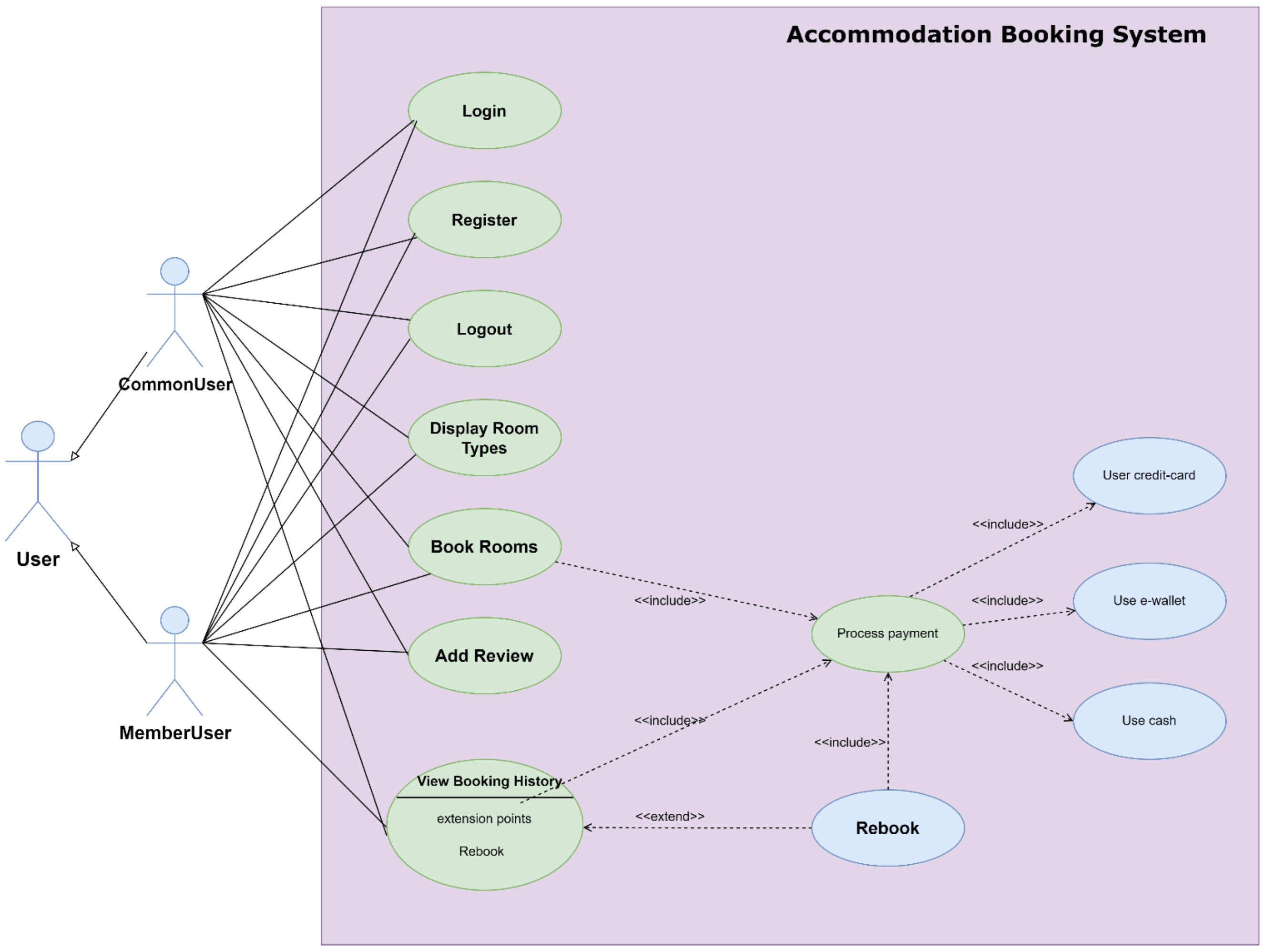
1.1 Class Diagram



To help you better check the class diagram, here is the link for your reference:

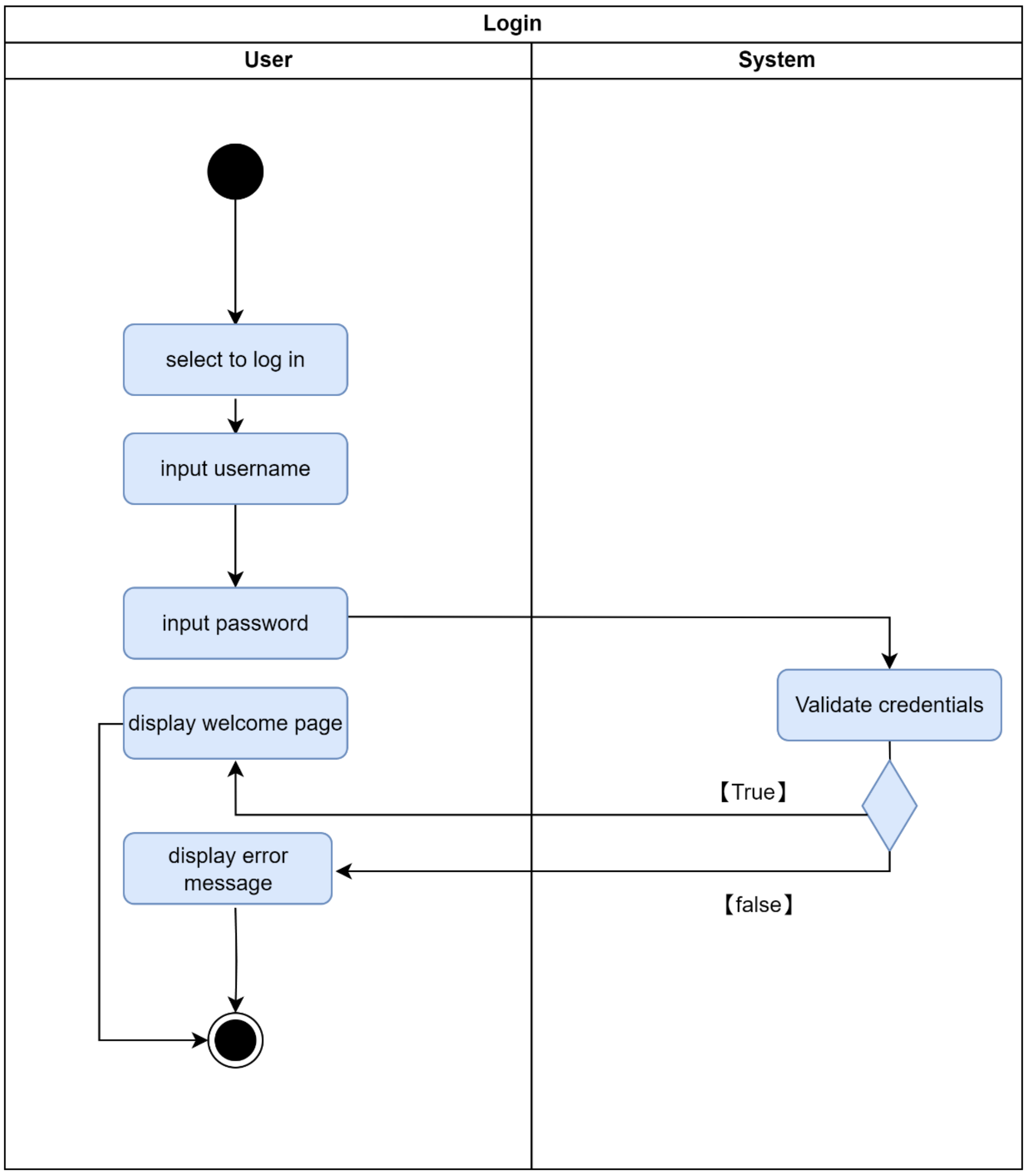
Class-diagram.pdf

1.2 Use-case Diagram

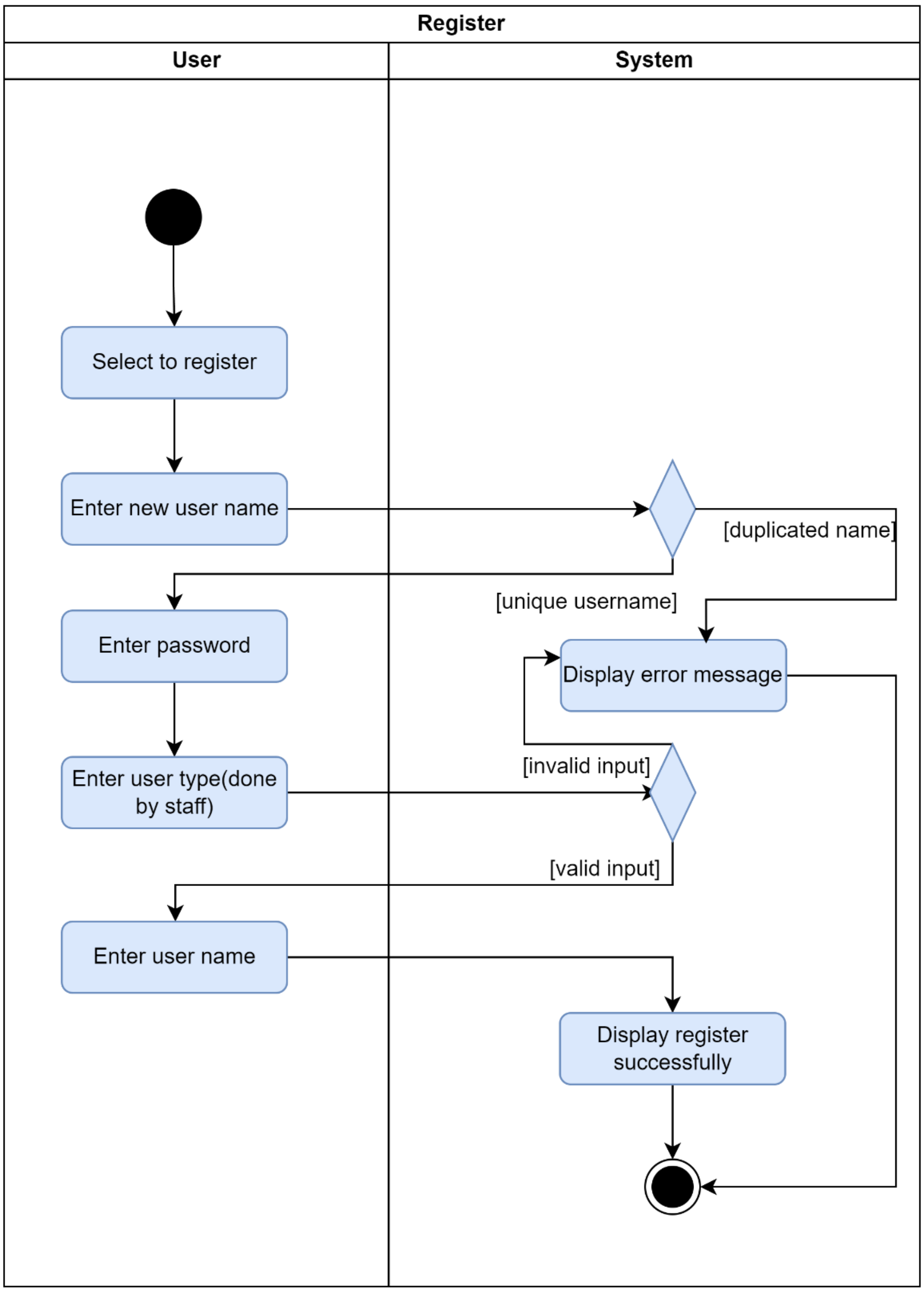


1.3 Activity Diagram

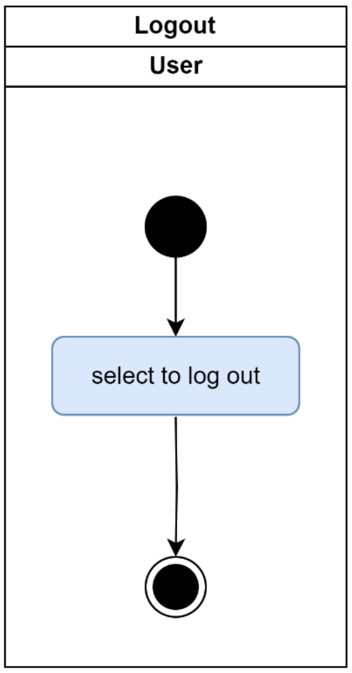
1.3.1 Login



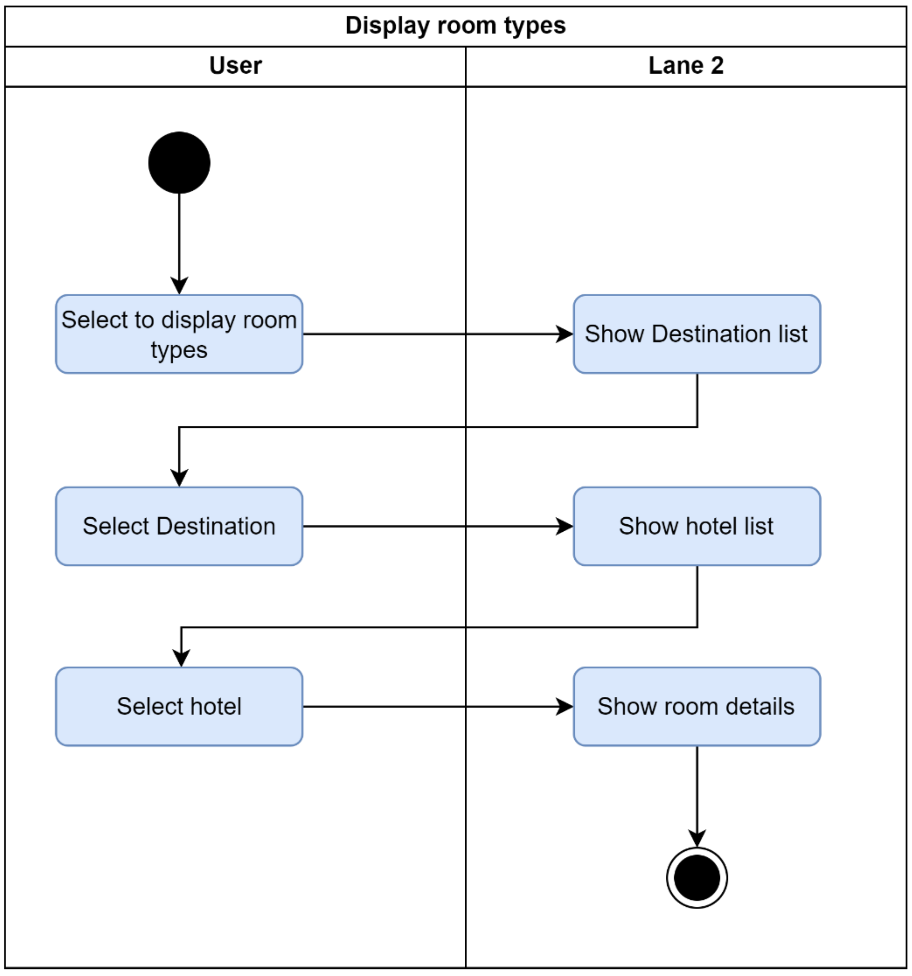
1.3.2 Register



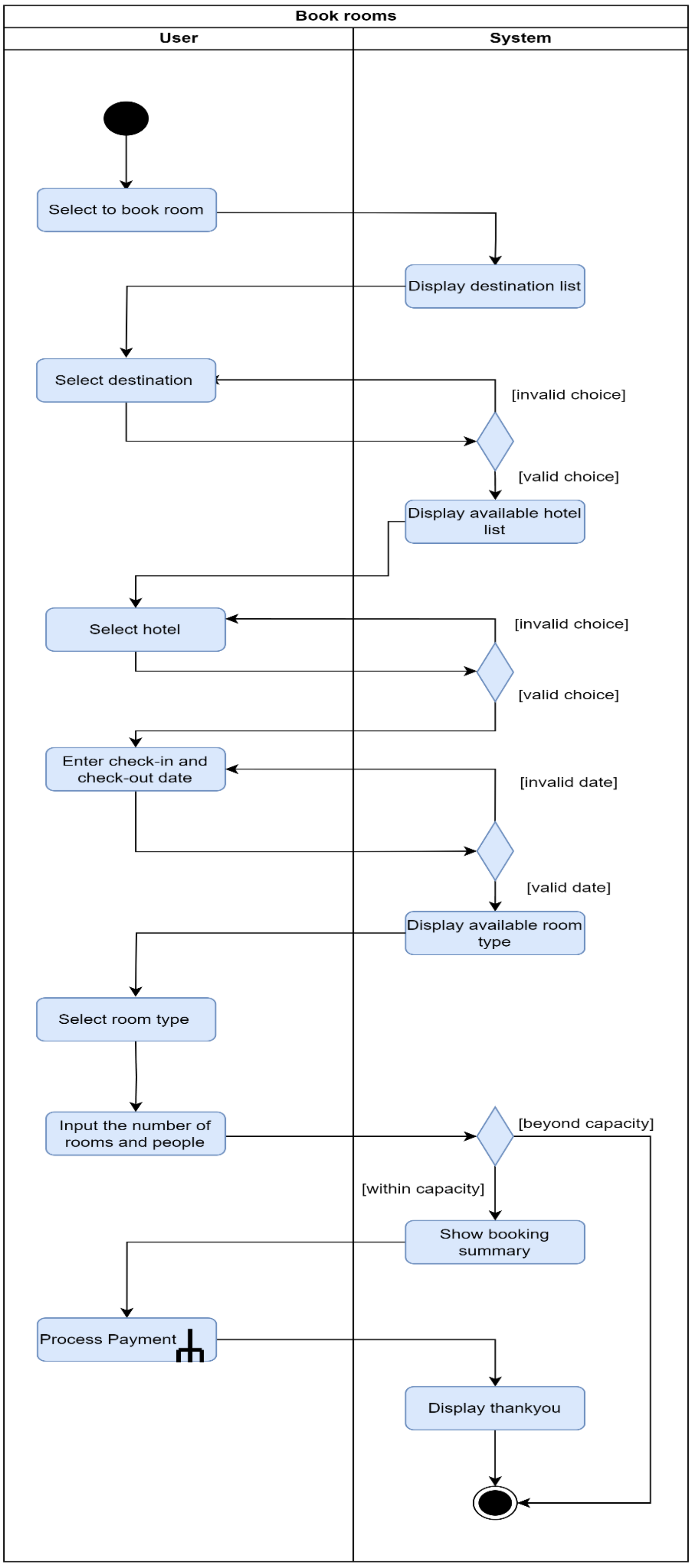
1.3.3 Logout



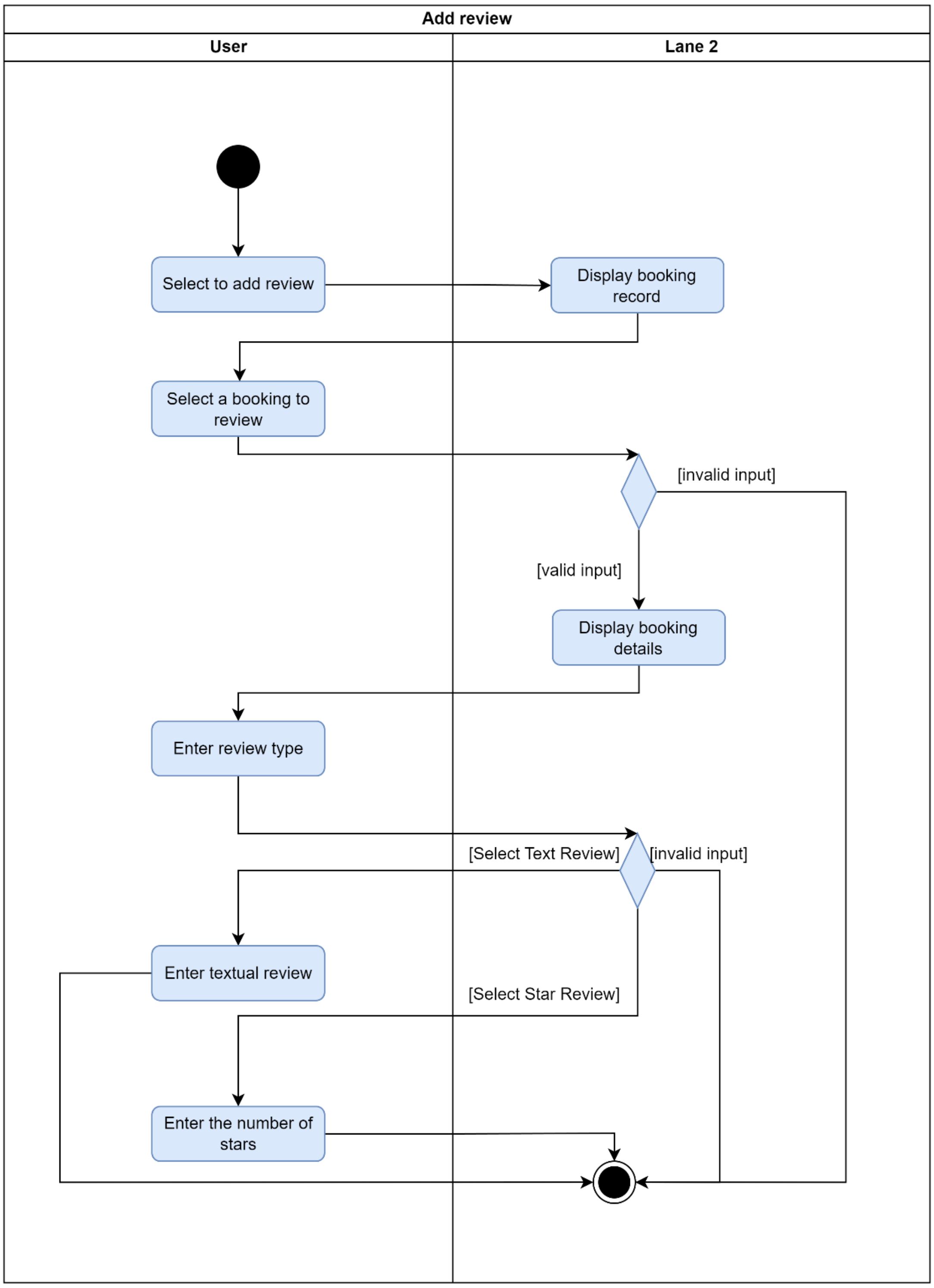
1.3.4 Display room types



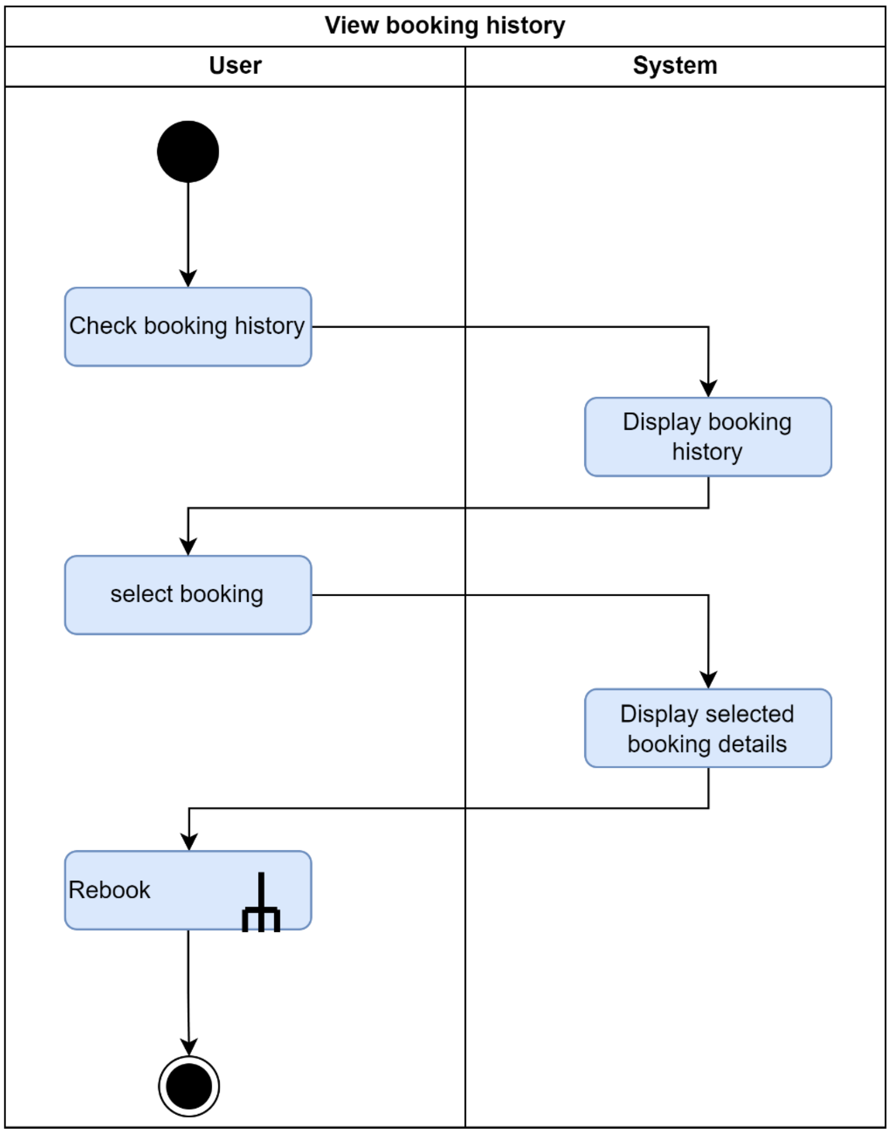
1.3.5 Book rooms



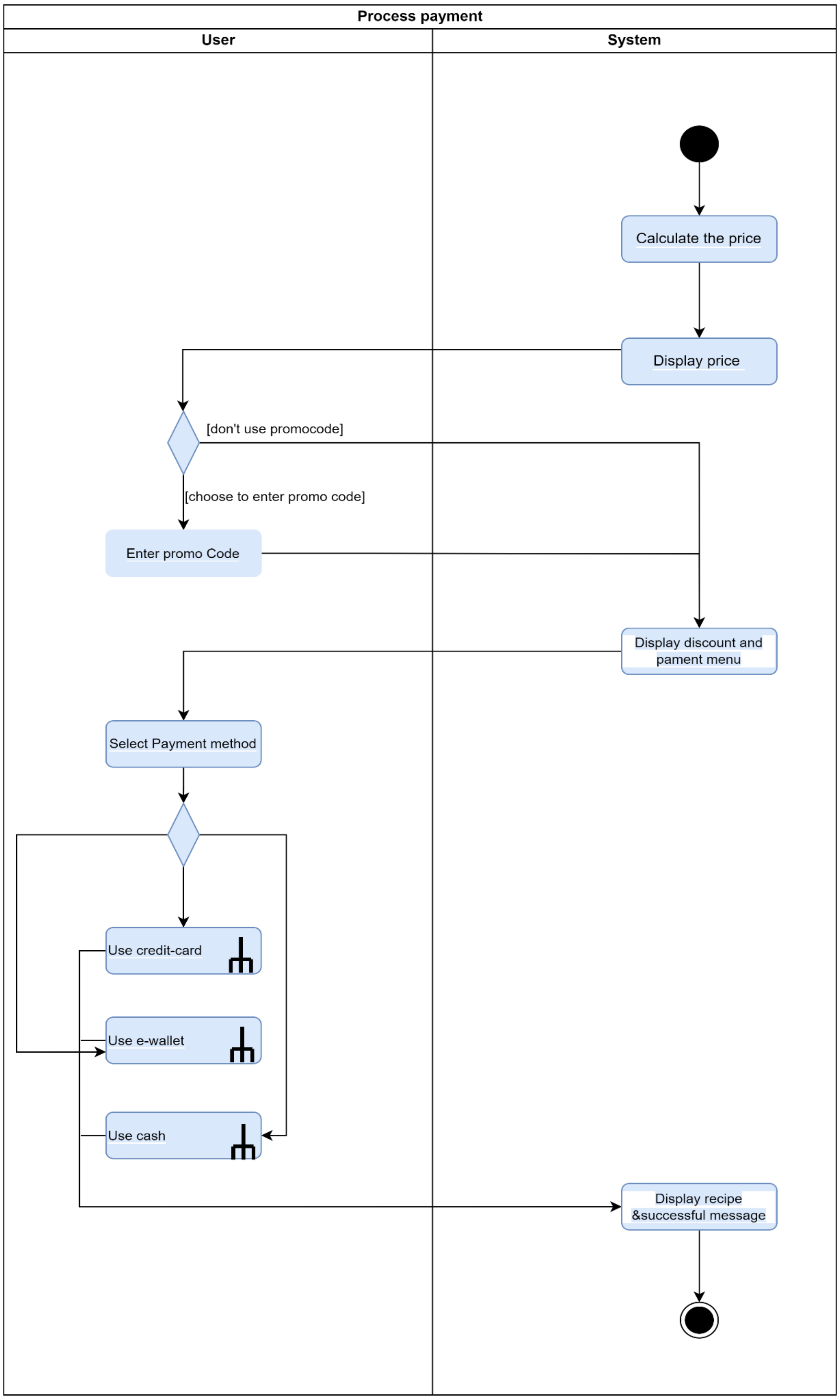
1.3.6 Add review



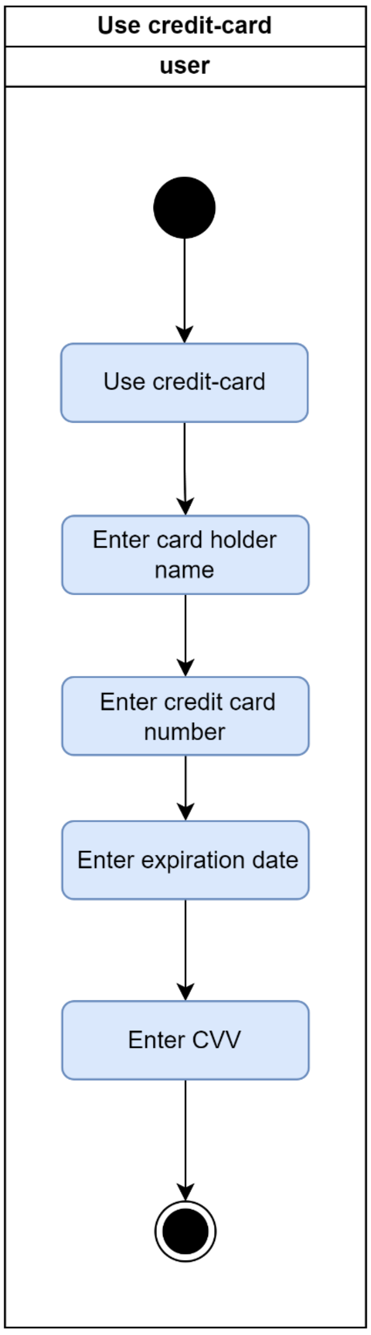
1.3.7 View booking history



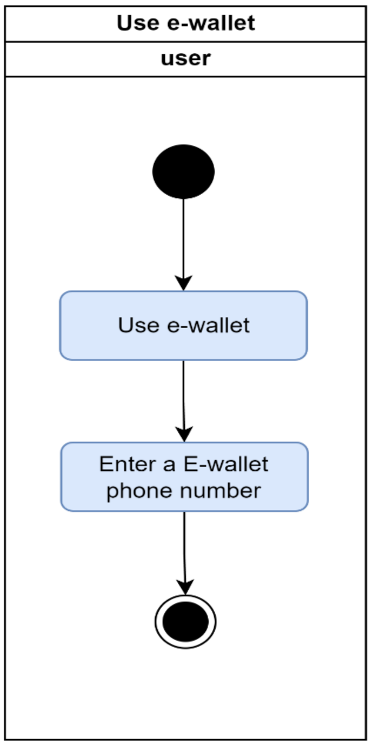
1.3.8 Process payment



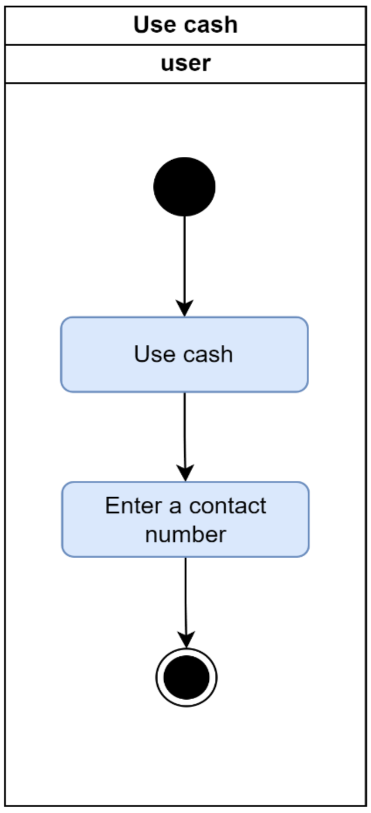
1.3.9 Use credit-card



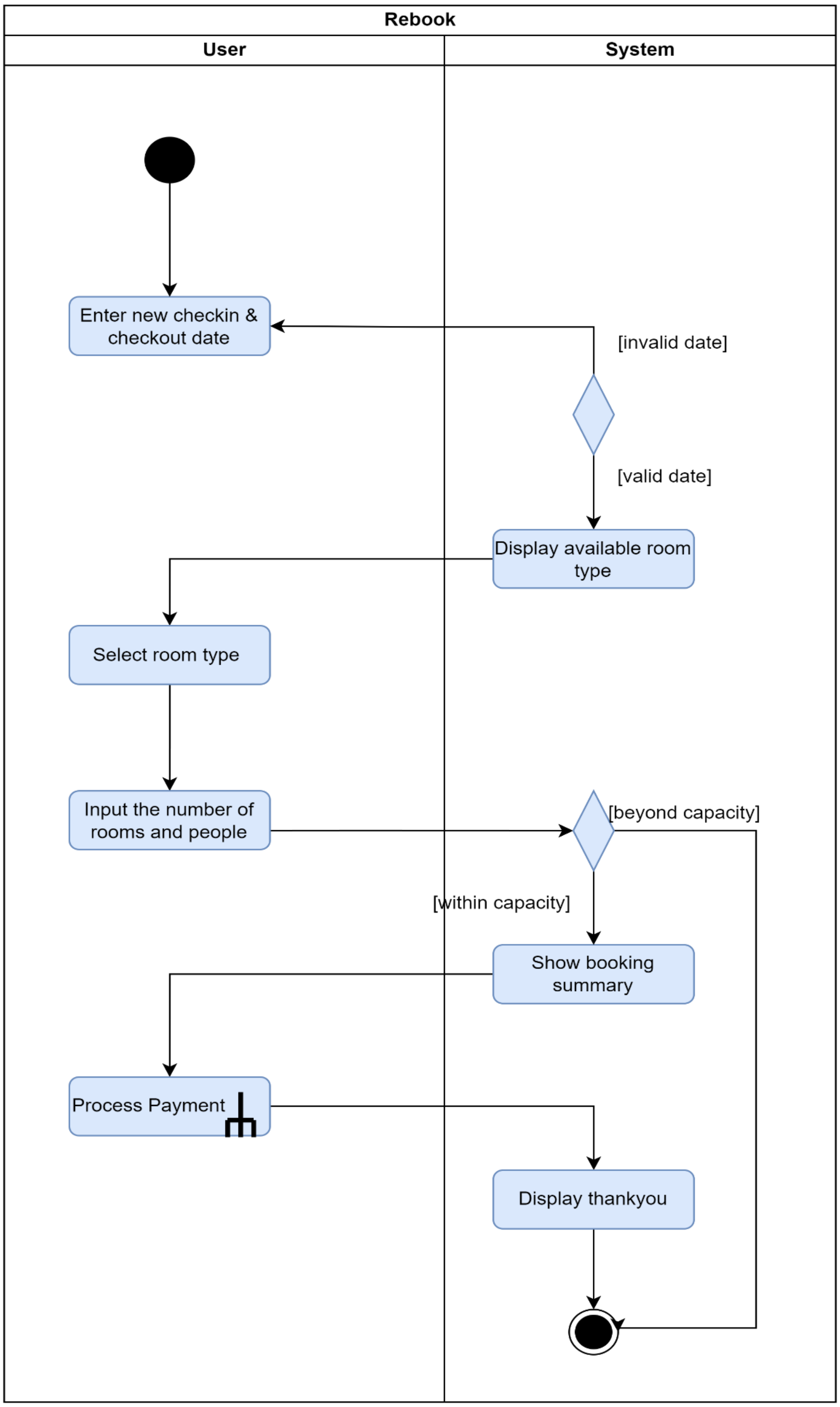
1.3.10 Use e-wallet



1.3.11 Use cash



1.3.12 Rebook


Rebook
User
System
Enter new checkin & checkout date
[invalid date]
[valid date]
Display available room type
Select room type
Input the number of rooms and people
[beyond capacity]
[within capacity]
Show booking summary
Process Payment
Display thankyou

2. STLs explanation

I primarily use $std::vector$ in this project because I need dynamic arrays to hold data such as the list of users, bookings, reviews, and hotel rooms. The flexibility of being able to add new items without manually managing memory or reallocation allows me to focus on the application logic rather than low-level resource handling. Using $std::vector$ also makes it straightforward to iterate, search, and sort elements, which is important when I'm performing operations like displaying all hotels in a particular destination or retrieving a user’s entire booking history.

I also incorporate $std::set$ for maintaining certain unique data, such as the distinct room types in each hotel or the distinct destinations for all hotels. The automatic avoidance of duplicates is crucial because it simplifies the process of building lists that must not contain repeated items. A set-like container enforces uniqueness, so I don’t need to manually check if the item already exists before inserting.

Beyond these two, I also rely on <string> and the stream library for parsing CSV lines, formatting booking IDs, and handling user input/output more efficiently. I use <algorithm> (for example, $std::find\_if$) to efficiently locate elements, while $< iomanip >$ (with features like $std::setw$ and $std::setfill$) ensures cleanly formatted textual output. Moreover, <regex>—enhanced in C++17—validates date strings, while $< ctime >$ combined with $< sstream >$ converts those strings into time values to check for overlaps between bookings. Additionally, <filesystem>, a C++17 feature, checks if the "data" folder exists and create it if necessary, thereby avoiding platform-specific file-handling code.

More broadly, C++17’s improvements, such as structured bindings and template argument deduction, help reduce boilerplate code, allowing me to focus on key booking logic instead of low-level details. Together, these standard library tools ensure safer memory usage—often in conjunction with $std::shared\_ptr$ —and consistent input/output routines, making them ideal for an application that prioritizes data integrity, file-based persistence, and user-friendly console interactions.

3. Rationale of using inheritance and polymorphism

I introduced inheritance in the user hierarchy primarily to differentiate general users from members in a natural way. The base User class provides essential attributes—like user ID, username, password, and age—and declares a pure virtual method, $displayWelcomeMessage()$, which forces derived classes ($CommonUser$ and $MemberUser$) to provide their own specialized welcome messages. By extending User into $MemberUser$, I could add loyalty-related data and behaviors (for example, storing the loyalty level and applying a membership discount). This structure allows shared code (login, ID management, etc.) to stay in one place, while enabling distinctive features (like greeting loyal users or awarding membership benefits) to be isolated in the $MemberUser$ subclass.

Another prominent example of inheritance and polymorphism occurs with the payment system, where an abstract base class Payment defines virtual methods $processPayment()$ and $getPaymentDescription()$. Specific payment types ($CreditCardPayment$, $EWalletPayment$, and $CashPayment$) override these methods to handle specialized user inputs and finalization steps (like card details, phone number, or simple cash confirmation). This design lets my booking process remain unaware of the exact payment mechanism; it only calls the base class interface. Each subclass presents a unique user dialogue and final descriptor (e.g., “Credit Card (****1234)”), demonstrating polymorphism at runtime.

I employed a similar approach for discounts and reviews. The Discount class is an abstract class that others, such as $PromoCodeDiscount$ and $LoyaltyDiscount$, can inherit and override with their respective discount calculation strategies. Meanwhile, Review is also an abstract class, and $StarReview$ plus $TextReview$ carry out specialized tasks (like storing numeric ratings versus textual comments). By introducing these polymorphic hierarchies, I gained the ability to extend or modify discount or review logic without forcing changes upon higher-level modules in the application. Ultimately, inheritance and polymorphism allowed me to segment code logically, reduce duplication, and dynamically select the proper functionality at runtime.

4. Rationale for using the static function

I designed the $generateBookingID()$ and $generateReviewID()$ functions as static methods so that each new booking or review can conveniently fetch a globally unique identifier. Using a static counter in these classes preserves a single, shared numeric state, eliminating the need for external or global code to keep track of how many bookings or reviews have been created. This approach keeps the logic centralized within the classes themselves, simplifying the process of ID generation and reducing the chance of accidental duplication or concurrency issues elsewhere in the code.

By implementing these generator methods as static functions, I avoid creating an unnecessary instance each time I need a new identifier. Rather than having to construct an object to produce IDs, a single static function call like $Booking::generateBookingID()$ or $Review::generateReviewID()$ does the job on demand, ensuring consistency across the application. It also clearly communicates that no individual instance is responsible for producing IDs; instead, the class as a whole governs the counting logic.

Beyond ID creation, the code also includes a static $setBookingCounter()$ (and a comparable one for reviews), which updates the static counter after loading existing data from CSV. This mechanism aligns the newly created items with the highest ID found in storage, guaranteeing that future IDs do not clash with already loaded records. In short, defining these as static functions makes sense for uniform identifier management, fosters straightforward data synchronization between file input and memory, and encapsulates ID logic tightly within each class without requiring global variables.

5. Additional functions explanation

I introduced three extra functionalities into my application—discounts, reviews, and registration for different types of users—that collectively enhance the system's flexibility and scope. The discount feature supports both a loyalty-based discount and a promo code discount, encapsulated in distinct subclasses ($LoyaltyDiscount$ and $PromoCodeDiscount$) inheriting from a common Discount interface. In the $PromoCodeDiscount$ class, a valid code reduces the total booking cost by a certain percentage, while the $LoyaltyDiscount$ rules subtract a fixed monetary amount for each loyalty level (for example, a Level 3 member automatically gets 30 RM off). This design allows me to add or modify any discount logic without changing the core booking process, because all discount classes share the same interface.

Reviews also exhibit polymorphism with $StarReview$ and $TextReview$, each inheriting from an abstract Review class. This ensures that whether a user rates their stay by typing out a paragraph or giving a star rating (from 1 to 5 stars), the system treats both reviews uniformly in terms of storing metadata such as booking ID and hotel name. At the same time, each subclass implements its own display and save format—$StarReview$ shows star symbols, while $TextReview$ holds descriptive text. Because all these details inherit from the shared parent, I can effortlessly add more review types in the future with minimal alterations to existing code.

The user registration feature allows individuals to join the system as either a $CommonUser$ or a $MemberUser$, both inheriting from the same User base class. The $MemberUser$ adds loyalty functionality that triggers the loyalty discount mentioned earlier, subtracting a fixed amount based on membership level. For instance, a user with loyalty level 2 may get 20 RM off each booking, while a level 3 user gets 30 RM off. By defining these specialized behaviors in $MemberUser$ and sharing common fields like username and password in the abstract User class, the code is cleaner and avoids repetition—enabling a clear path for future expansions, such as VIP tiers or corporate accounts.

6. Reference

[1] cppreference.com. (n.d.). *std::exception*. Retrieved December 29, 2024, from https://en.cppreference.com/w/cpp/error/exception

[2] cppreference.com. (n.d.). *Standard library header <sstream>*. Retrieved December 29, 2024, from https://en.cppreference.com/w/cpp/header/sstream

[3] cppreference.com. (n.d.). *Standard library header <iomanip>*. Retrieved December 29, 2024, from https://en.cppreference.com/w/cpp/header/iomanip

[4] cppreference.com. (n.d.). *Regular expressions library (since C++11)*. Retrieved December 29, 2024, from https://en.cppreference.com/w/cpp/regex

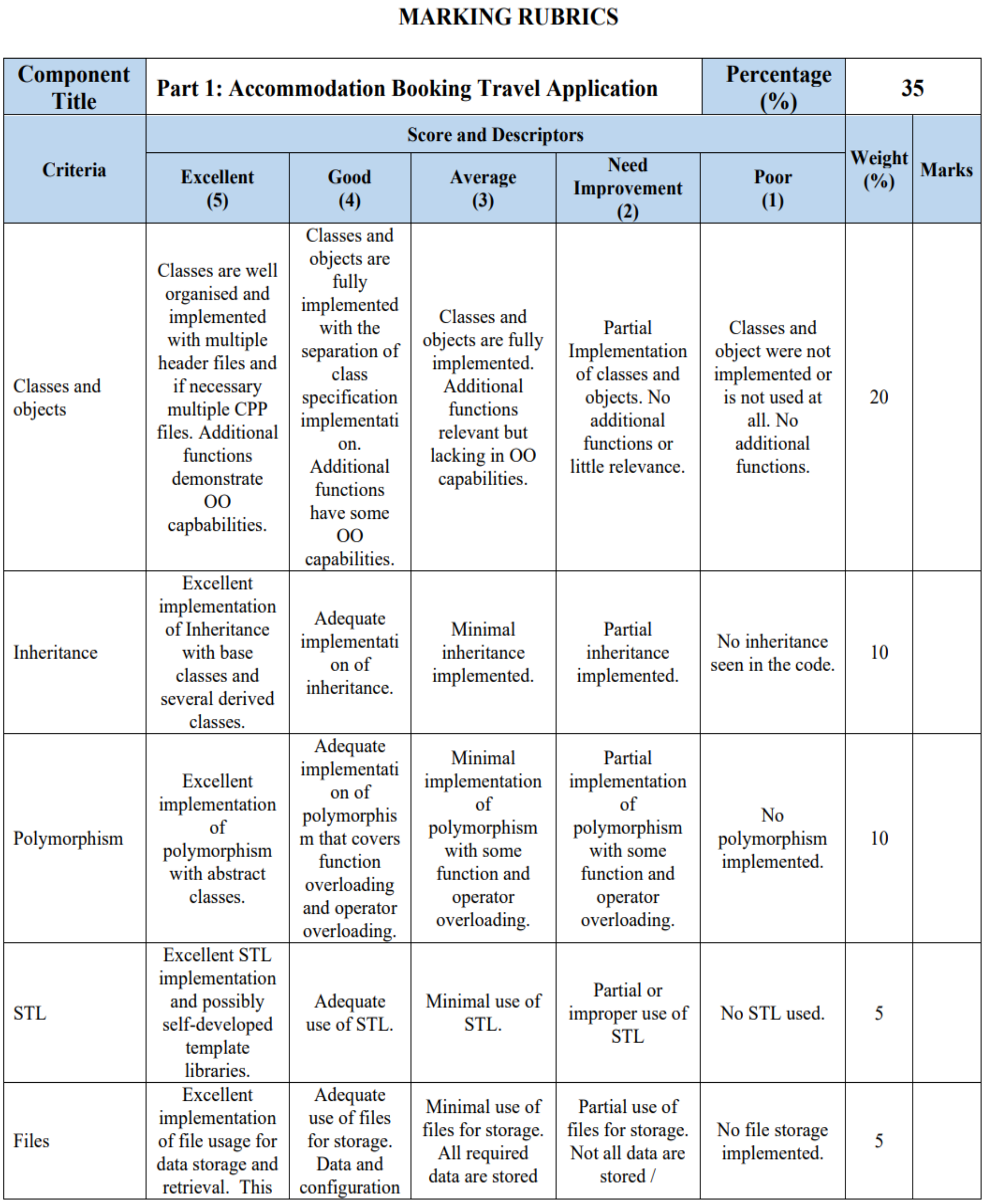
[5] cppreference.com. (n.d.). *std::ctime*. Retrieved December 29, 2024, from https://en.cppreference.com/w/cpp/header/ctime

[6] GitHub. (n.d.). *an-halim/Hotel-BookingAPP: Simple app to manage hotel room booking system Features*. Retrieved December 29, 2024, from https://github.com/an-halim/Hotel-BookingAPP

[7] Duke56. (2016, September 1). *Solution to the "_CRT_SECURE_NO_WARNINGS" error message*. CSDN. Retrieved December 29, 2024, from https://blog.csdn.net/duke56/article/details/52403458

[8] **utf8-bom-strip**. (n.d.). *GitHub Repository*. Retrieved December 29, 2024, from https://github.com/zer4tul/utf8-bom-strip

7. Marking Rubrics



MARKING RUBRICS

Component Title	**Part 1: Accommodation Booking Travel Application**				**Percentage (%)**	**35**	
Criteria	**Score and Descriptors**					**Weight (%)**	**Marks**
	Excellent (5)	**Good (4)**	**Average (3)**	**Need Improvement (2)**	**Poor (1)**		
Classes and objects	Classes are well organised and implemented with multiple header files and if necessary multiple CPP files. Additional functions demonstrate OO capbabilities.	Classes and objects are fully implemented with the separation of class specification implementati on. Additional functions have some OO capabilities.	Classes and objects are fully implemented. Additional functions relevant but lacking in OO capabilities.	Partial Implementation of classes and objects. No additional functions or little relevance.	Classes and object were not implemented or is not used at all. No additional functions.	20	
Inheritance	Excellent implementation of Inheritance with base classes and several derived classes.	Adequate implementati on of inheritance.	Minimal inheritance implemented.	Partial inheritance implemented.	No inheritance seen in the code.	10	
Polymorphism	Excellent implementation of polymorphism with abstract classes.	Adequate implementati on of polymorphis m that covers function overloading and operator overloading.	Minimal implementation of polymorphism with some function and operator overloading.	Partial implementation of polymorphism with some function and operator overloading.	No polymorphism implemented.	10	
STL	Excellent STL implementation and possibly self-developed template libraries.	Adequate use of STL.	Minimal use of STL.	Partial or improper use of STL	No STL used.	5	
Files	Excellent implementation of file usage for data storage and retrieval. This	Adequate use of files for storage. Data and configuration	Minimal use of files for storage. All required data are stored	Partial use of files for storage. Not all data are stored /	No file storage implemented.	5	

	includes searching and retrieving algorithm. Extensive data.	information is also stored and retrieved from files.	and retrieved from files.	retrieved from file			
Exception handling	Excellent exception handling with all errors caught and handled appropriately.	Adequate exception handling by handling all important errors as well as recovering from some of those errors.	Minimal exception handling where important errors are handled by printing out proper error statements.	Partial exception handling where error is acknowledged and program exits.	No exception handling implemented.	5	
Quality of implementation	Excellent coding format and standards with intuitive UI. Files are named appropriately. Development uses project files for multi-file source code.	Program runs very well with good UI. Code conforms to coding standards and format.	Program runs perfectly with simple UI.	Program can compile and run but crashes.	Program cannot compile and/or run.	15	
					TOTAL	**70**	

Component Title	**Part II: Project Documentation**				**Percentage (%)**	**15**	
Criteria	**Score and Descriptors**					**Weight (%)**	**Marks**
	Excellent (5)	**Good (4)**	**Average (3)**	**Need Improvement (2)**	**Poor (1)**		
UML Diagrams	Precise and accurate usage of diagrams.	A few minor mistakes were found in the diagram (e.g., wrong class name).	Some noticeable mistakes (e.g., inaccurate class relationships)	Not all the classes are included in the diagram.	The diagram is confusing, and the format is not standardised. or no attempt	10	
STL Explanation	The explanations are clear and concise to address the question accurately.	The explanations are good but missing a few essential points.	The explanations are acceptable but with a few inaccurate statements.	The explanations are vague and contain several inaccurate statements.	The explanations are almost irrelevant or wrong or no attempt	5	
Inheritance and polymorphism explanation	The explanations are clear and concise to address the question accurately	The explanations are good but missing a few essential points.	The explanations are acceptable but with a few inaccurate statements.	The explanations are vague and contain several inaccurate statements	The explanations are almost irrelevant or wrong or no attempt	10	
Static and additional functions	Statis usage is explained in detail. Additional functions thoroughly explained and highly relevant. Adds value to the solution.	Static usage is well-explained. Additional functions explained. Its relevance is clear.	Static explained with some gaps. Additional functions explained but needs more details.	Static usage not clear. Additional functions not clearly explained.	No static and/or additional functions.	5	
					TOTAL	**30**	