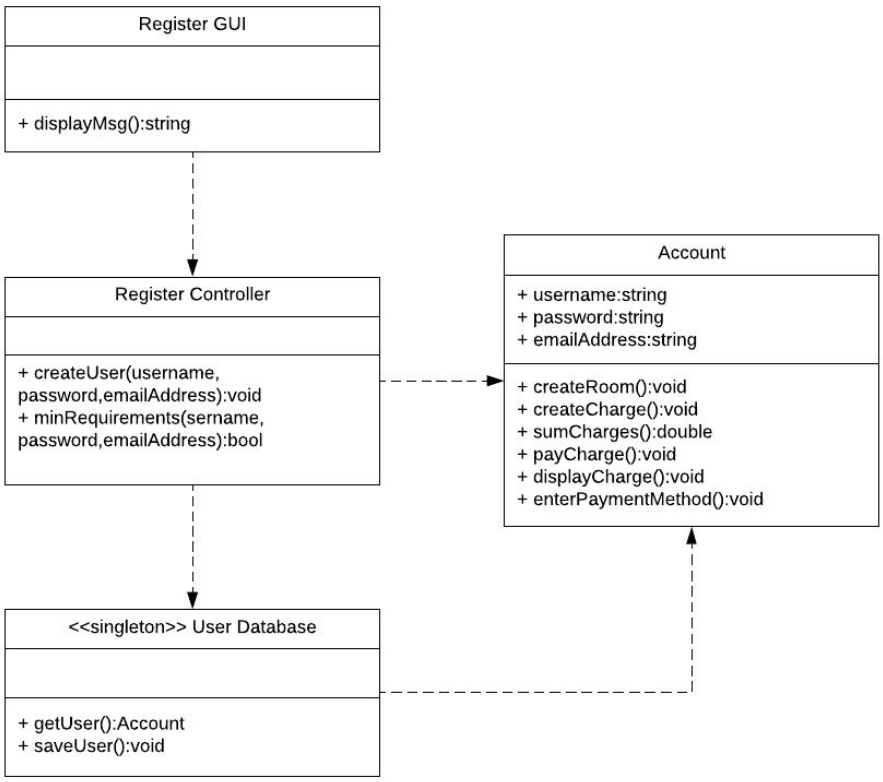
PALSS
Design Patterns and Models

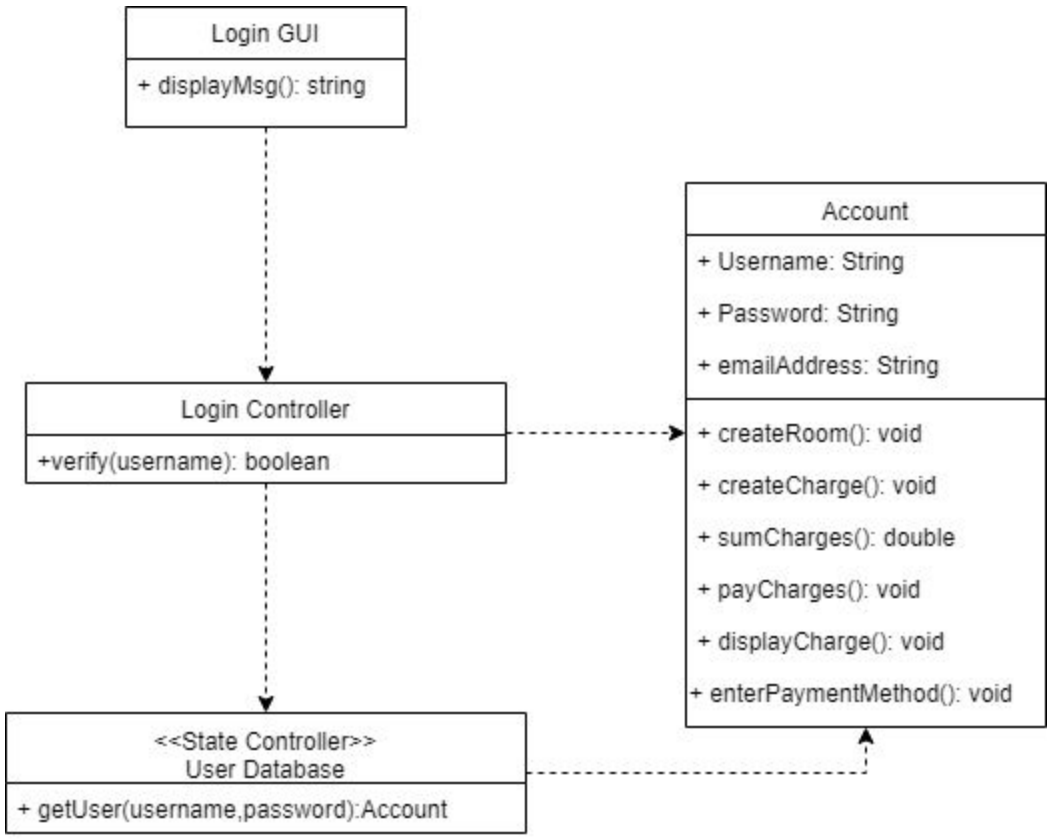
As a multifaceted financial app, we aim to build XPendit around a variety of design patterns. The use cases we will discuss in this document are as follows, and in this order:

- Create Account (User Story 012);
- Log In (US 012);
- Create Room (US 011);
- Add Members to Room (US 011);
- Create New Shared Expense (US 001, 003);
- Share Lists with Group (US 009, 010).

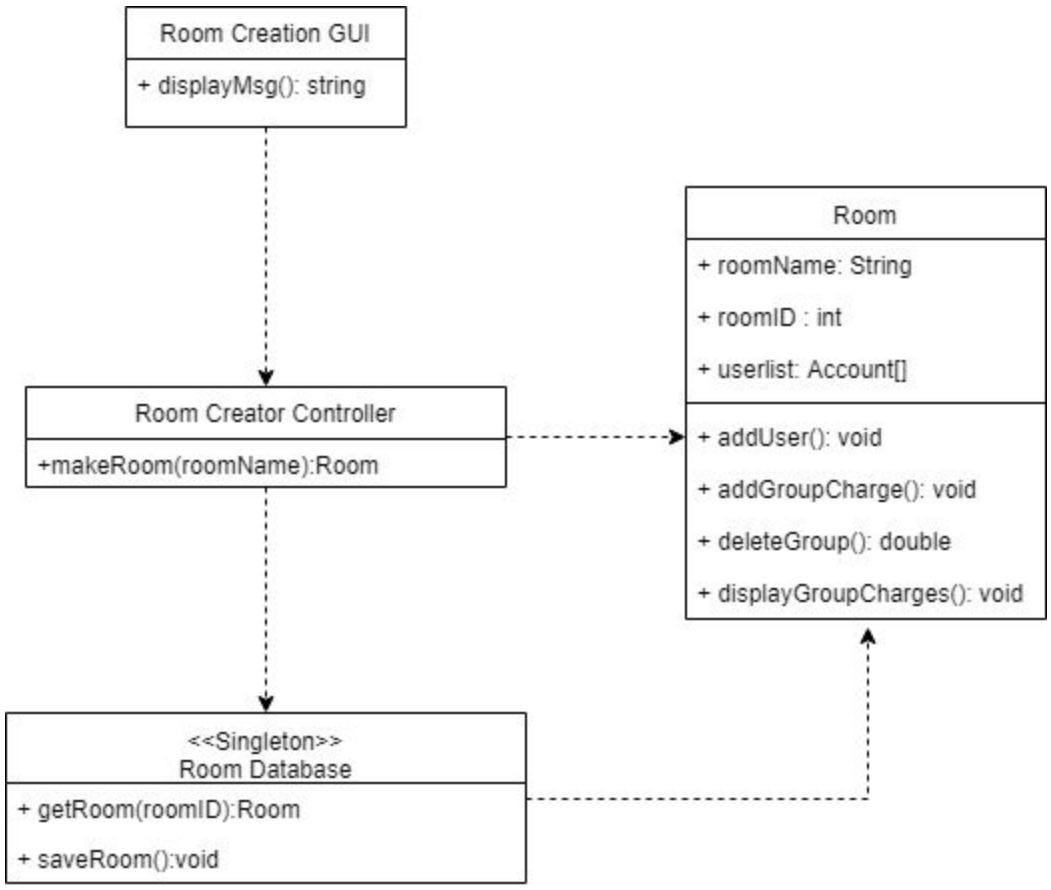
First, consider the use case “Create Account”. As the name suggests, this is a creational use case. However, there is only one type of account being created, as there is only one version of an account (i.e., there is not a “room owner” vs. a “room tenant”; all accounts are treated equally in the context of the room). Therefore, the simplest design pattern for the use case is a the Singleton approach.



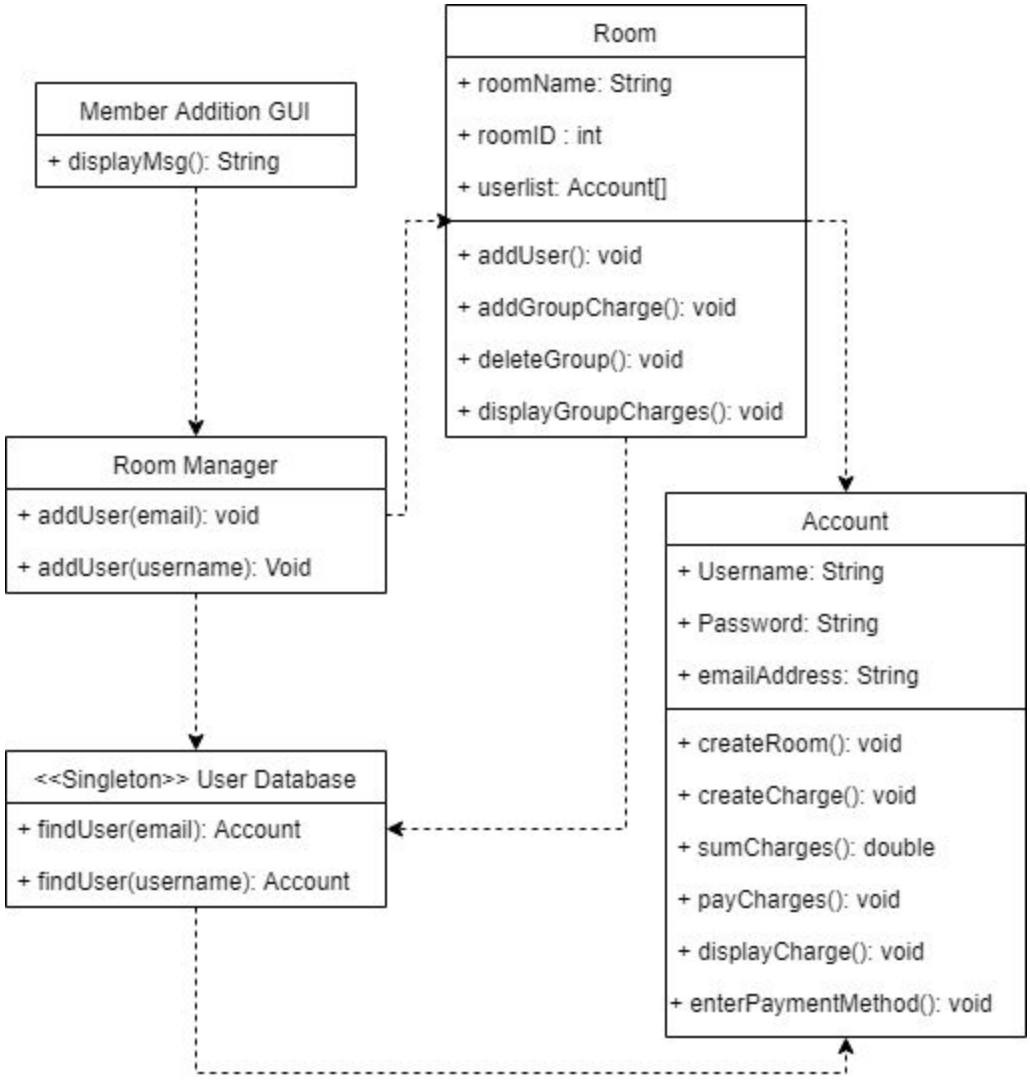
Next, consider the “Log In” use case. This is an extremely simple use case which chronologically comes directly after “create account” for new users, and is the first use case encountered by returning users. This use case is entirely behavioral and is ultimately based on a boolean-like communication between the login controller and user database. Therefore, we will approach this use case with a State design pattern.



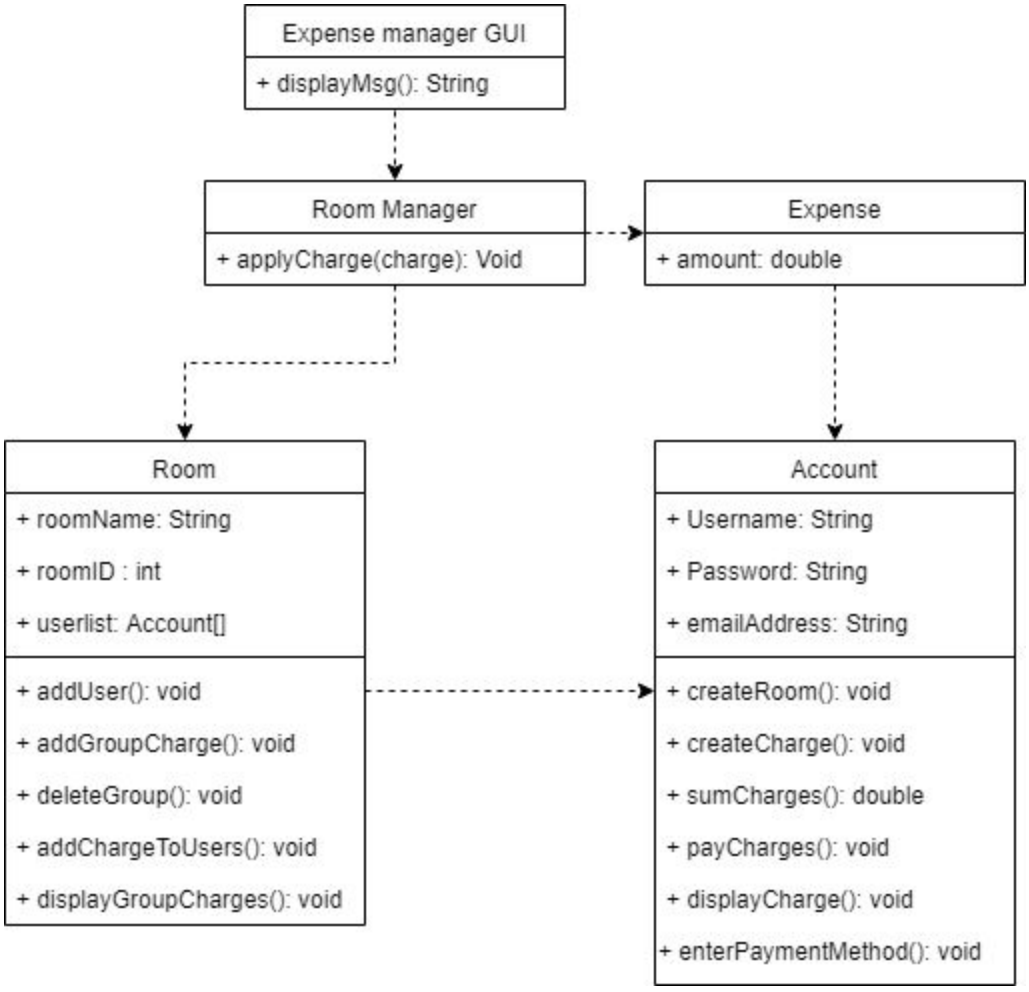
The “Create Room” use case shares many similarities with the “Create Account” use case - both require a creational approach, and only one type of object can be created. These similarities are fundamental enough that the “Create Room” use case also functions most efficiently with a Singleton approach.



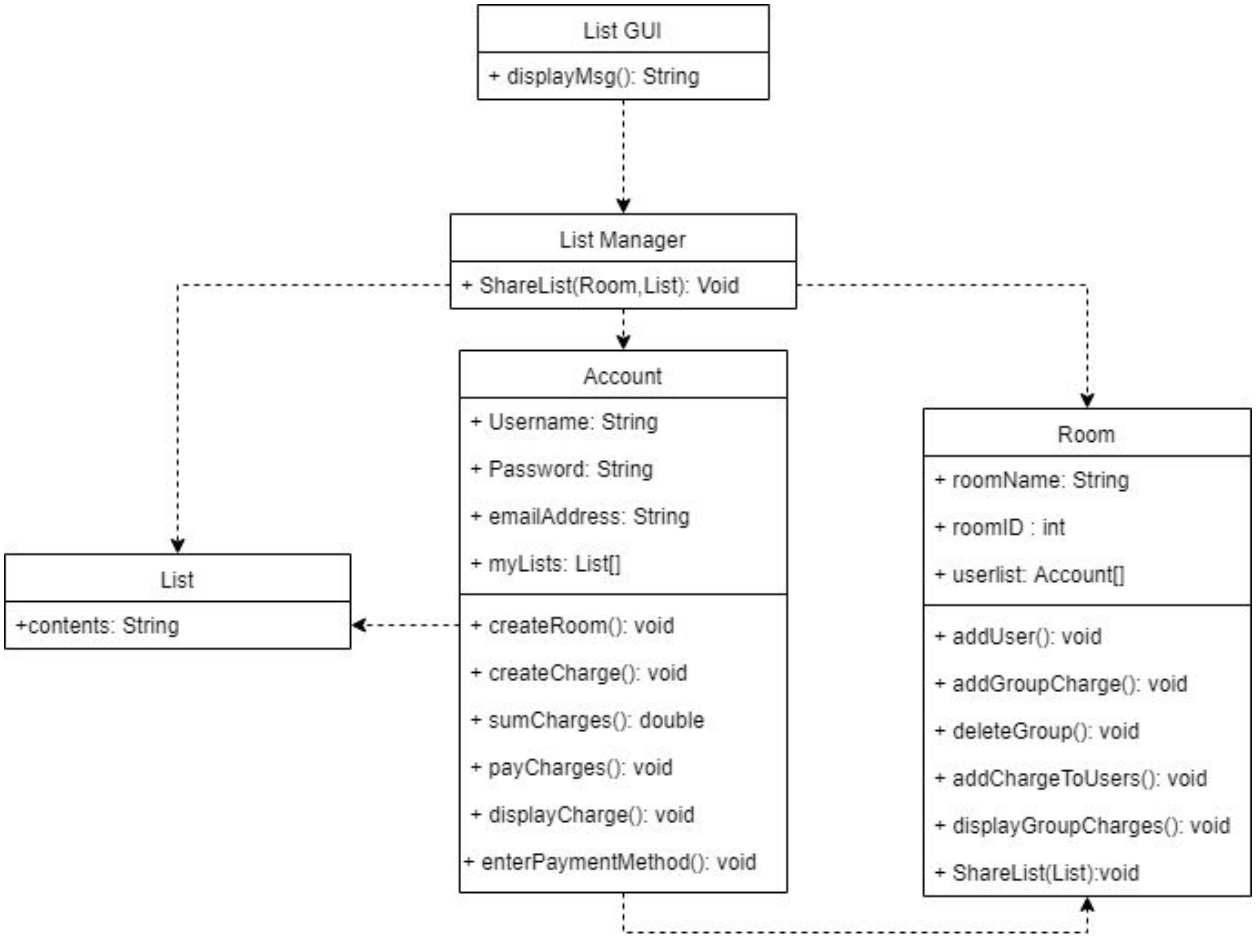
Whenever members need to “Add Members to a Group”, they may do so in a couple ways - for example, by looking up their in-app username or by their email address linked to the account. Therefore, in this use case, we use the behavioral Strategy approach, in which “all roads lead to Rome” - in this case, meaning there are multiple methods of creating the same successful outcome of adding an existing account to an existing room.



One of the cornerstones of the app comes in the user generation of shared expenses. When users employ the use case “Create New Shared Expense”, however, there are multiple generalizations to take into account (as noted in the use case document) - such as a recurring, evenly split charge or a variety of one-time, standalone options. The type of charge being created solely depends on the information the app is fed by the user, which prompts us to employ the creational Factory Method.



The final use case we aim to cover is “Share List with Group”. This requires a behavioral method, which does not have to be adaptable - it simply must take all of the list’s parameters and make them accessible by all group members. The Command design pattern is well-suited for the task, and is outlined below.



With this basic set of use cases outlined, XPendit can operate on a basic level, and with an adaptable, varied approach.