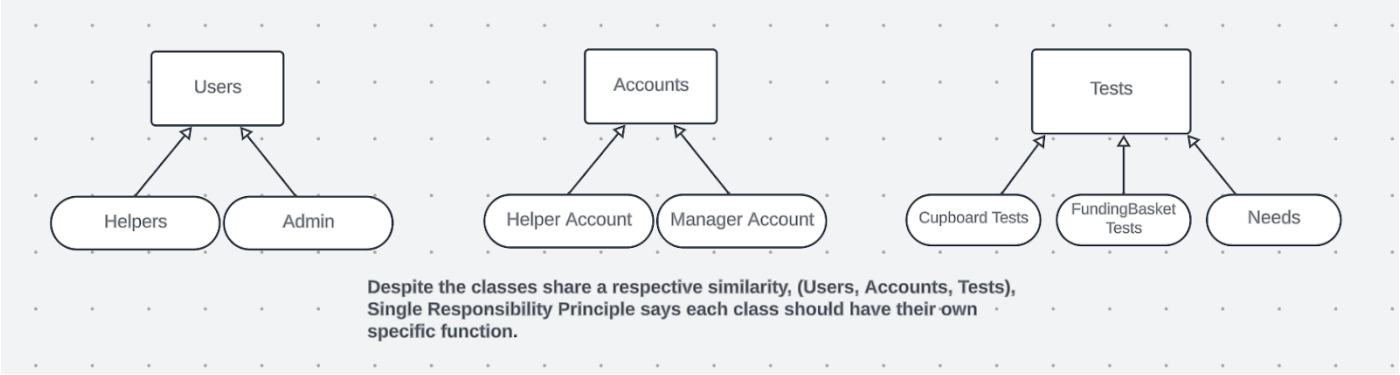
7gFundsInHighPlaces Adherence to Architecture and Design Principles

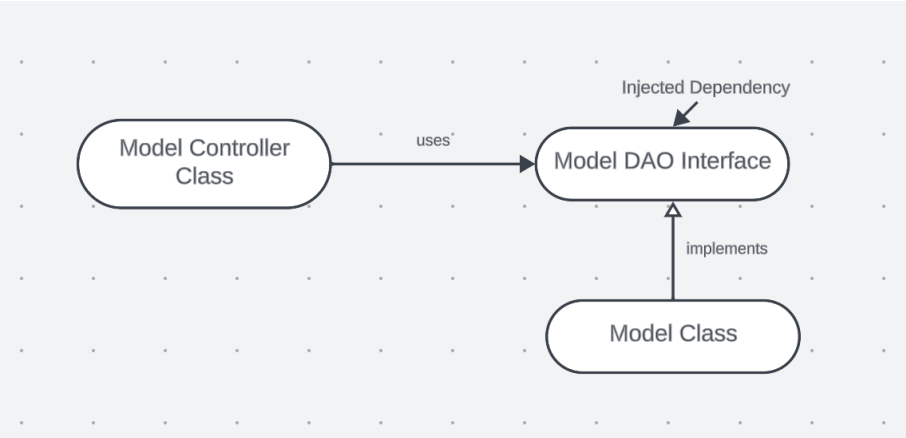
***Single Responsibility Principle*:**

One design principle that our code adheres to is the Single Responsibility principle. Essentially this principle says that each class should have one specific function. All the data required to do this function is encapsulated within this one class. We showcase this in our design as we separate modules into smaller components. In this sense, each component can have a specific functionality. For example, multiple users will use this application and we could put all user functionality in one class. However, putting all the user services in one class violates the single responsibility principle as each user type has a different functionality. Helpers are able to access the funding basket whereas admins are unable to. However, admins can update, add, and remove needs to the cupboard which is off-limits to helpers. Because the two components contain different functionalities, we must follow the single responsibility principle and separate them into different classes. We also utilize this principle in other areas such as the model controller classes which are separated by the type of API requests they handle. In unit testing, we separate them based on what class the code is testing instead of putting all test cases for the application in one file. By following the single responsibility principle, we are able to easily manage concurrent modifications and understand the scope of a change in a class.



***Dependency Injection*:**

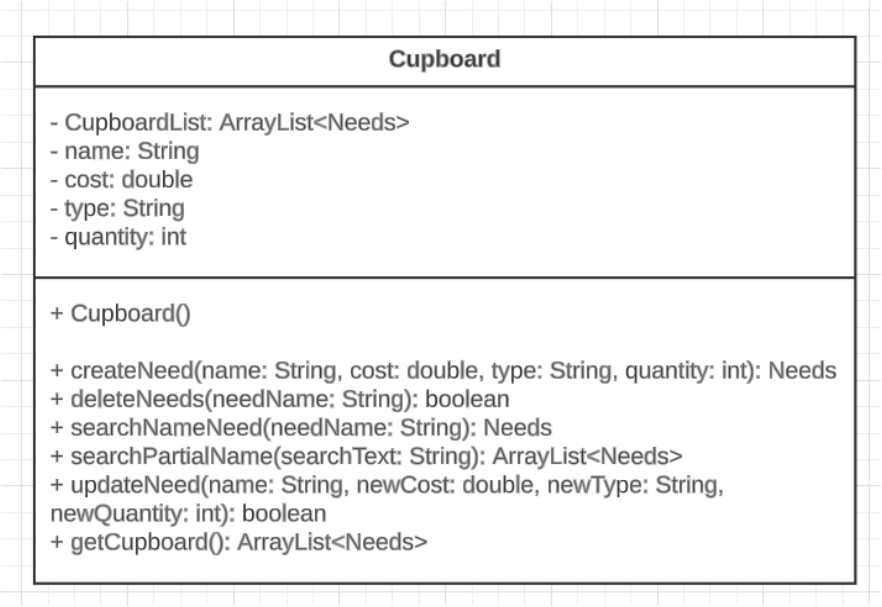
Another design principle that our code follows is the Dependency Injection Principle. It states that high-level modules should not depend on low-level modules but rather abstractions. The purpose of doing so is to loosen the coupling between modules. In our application, we used this principle many times by injecting a DAO interface between the module class and its respective controller. For example, Cupboard.java implements an interface named CupboardDAO. The Cupboard Controller does not directly use Cupboard.java but uses an instantiation of CupboardDAO. Practically, if the Cupboard were to break, we would not need to worry too much about the Cupboard Controller since they are not tightly coupled.



Information Expert:

Our model also adheres to the information expert principle. For example, in the Cupboard class, the cupboard is given the responsibility of maintaining the storage of Needs that an organization desires. Because the shopping cart object holds the needs ArrayList, we can assign it the

responsibility of adding removing, and editing needs while also being able to browse the needs within the cupboard. This keeps the class UML diagram simple and easy to understand without creating complications. For example, if an admin wants to delete a need from the cupboard, the Cupboard class can check whether the item exists in the cupboard by searching through its list of needs. If the need is found, the Cupboard class can remove it from the cupboard, as it holds the necessary information about the item.



All the data encapsulated within Cupboard.java are necessary for all functionality of the cupboard.