1. First Usage of Design Patterns: Singleton Pattern
2. Motivation

A location storage have to be passed in calendar grid as to access the locations’ details. So each time, after signing up and logged in as other user, a location storage did not be passed into the calendar grid. After logged out and go back to the login page, all locations’ information will be lost, as each time it is passed as a new constructor. So singleton pattern can help on maintaining a consistent locations storage.

1. Solution

CalGrid constructor need an appointment storage controller as parameter, an appointment storage controller constructor needs an appointment storage as parameter. So an instance controller have to be defined in the location storage and each time get the instance but not constructing a new one. Every time LocationStorageController getInstance() function will call getInstance() in LocationStorage to get the instance.

public static LocationStorage getInstance() {

if(LocationStorage.instance == null) {

LocationStorage.instance = new LocationStorageModel();

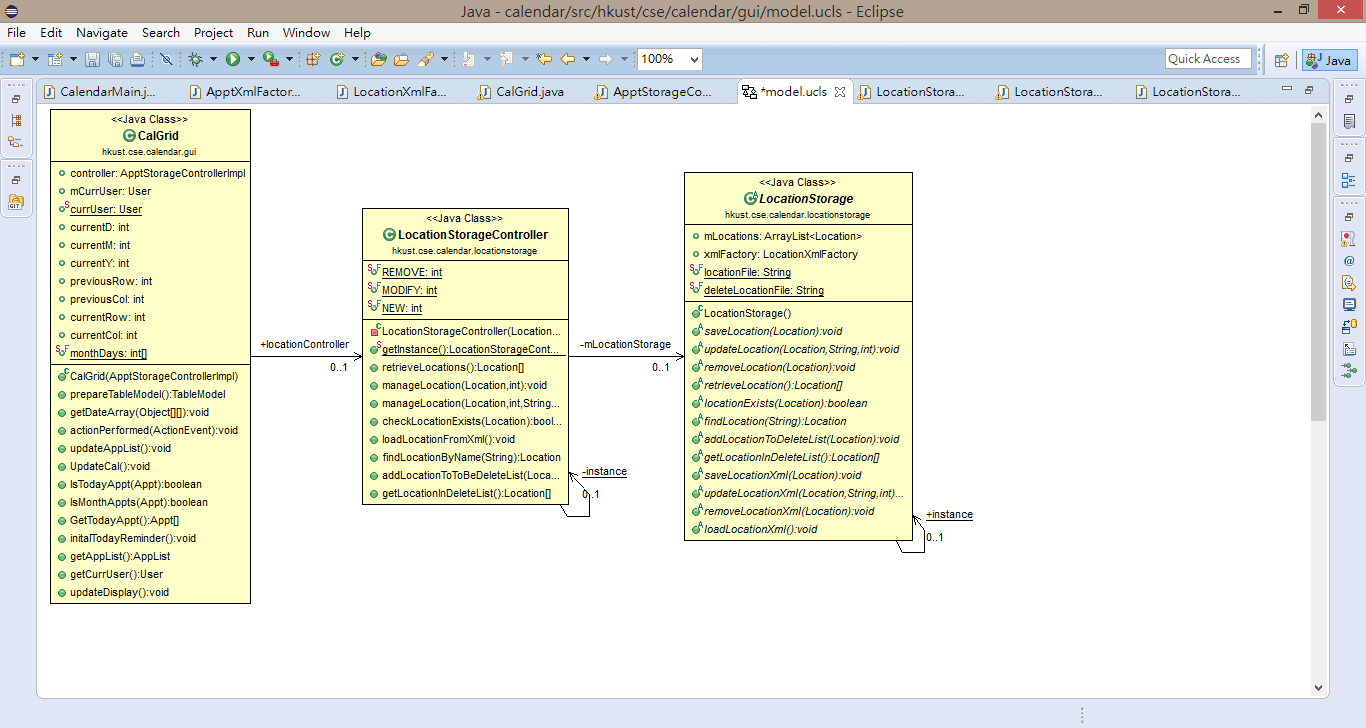
}

return LocationStorage.instance;

}

The getInstance() function, id the instance have been created, then return it, otherwise, create a new one and return it. So each time will access the same location storage, to avoid adding or modifying separately.

1. Class diagram



1. Second Usage of Design Patterns: Factory Pattern
2. Motivation

As there are two types of users - the regular users and the administrative users. They have different permission on accessing, modifying or deleting the appointments. In the base code provided, there is only a User class is given. So every time creating a user, is it the same type and have the same permission with other users. There have to be an identifier defined in the User class to show the difference in types of users.

1. Solution

We made use of the “pizza store” solution discussed in class. A factory is used to create the right type of inheritance class. UserFactory is created to prepare the construction of a user.

public User createAccount(…) {

if(type.equals("Admin")) {

user = new Admin(id, pw);

…

}

else if(type.equals("Regular")) {

user = new RegularUser(id, pw);

…

}

return user;

}

createAccount() function will automatically check the type of input, then constructor the corresponding type of user, where both Admin and RegularUser are extending the abstract User class . Then return it as a new user and store into UserStorage.

1. Class Diagram

