**Architecture Document**

**Pattern 1: Decorator Pattern**

**Motivation:**

Before the user logs onto the calendar client, the user is prompted to enter his username and password into the designated input boxes. Our goal was to implement input boxes that show grey background hint texts (i.e., “User Name” and “Password”) when there are no input in the text fields. When the user enters texts into either the User Name box or the Password box, the grey background hint texts should disappear and be replaced with the texts that the user had just entered, in black color.

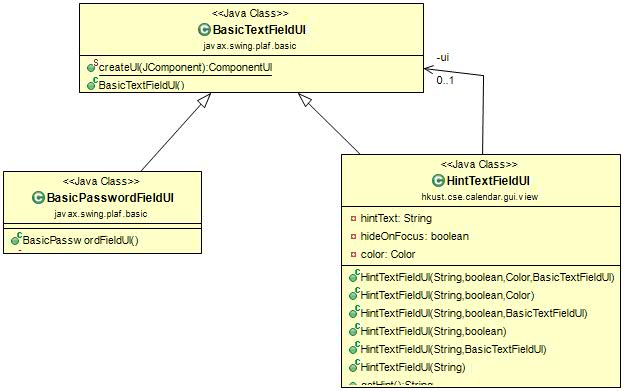
However, the User Name box was implemented using a JTextField and the Password box was implemented with a JPasswordField. Traditionally, we would have needed to create two new UI classes so that they can be used by either of the above mentioned TextField classes. This was the motivation to use a decorator pattern.

**Solution:**

Our abstract component class was the BasicTextFieldUI, and we created a decorator class HintTextFieldUI that is inherited from BasicTextFieldUI. The decorator was used to decorate the original BasicTextFieldUI as well as BasicPasswordFieldUI. Because in the decorator, we instantiated a BasicTextFieldUI object, the decorator can be used to decorator both a BasicTextFieldUI and a BasicPasswordFieldUI. With the decorator class implemented, we only needed to create one new class to show the background hint texts, instead of two separate new classes.

Note that we only needed one concrete decorator. Because of this we did not create a separate decorator class for concrete decorators to inherit from. Instead we only created the HintTextFieldUI concrete decorator, which is a generalization of the upper component class.

**Class** **Diagram:**



**Pattern 2: Observer Pattern**

**Motivation/Design Problem:**

Time machine can change the time of a user. The occurrence of many other changes is depend on the change of time. For example, calendar view is set according to the current time of the time machine. When the time changes to a different month, the calendar dashboard will switch to the new month and make other view changes. Not only does the calendar view, but also the notification center totally depends on time machine because when to send an event’s remainder is calculated by the time remaining.

Based on the above description, a communication way between the time machine and functions that depend on it needs to be established. One easy and traditional solution is to set up the communication mechanisms separately for each dependency, which means to build to communication mechanisms, one for the calendar view and the other for the notification. This approach, however, is awkward in the sense that the design of different communication mechanisms rather than only one significantly increase the programming workload. For example, the time machine and the notification center is tightly coupled so they have to know each other in details, leading to difficulties in both programming and further maintenance. And the same amount of workload is generated again when designing and maintaining the communication channel between the time machine and the calendar view.

**Solution:**

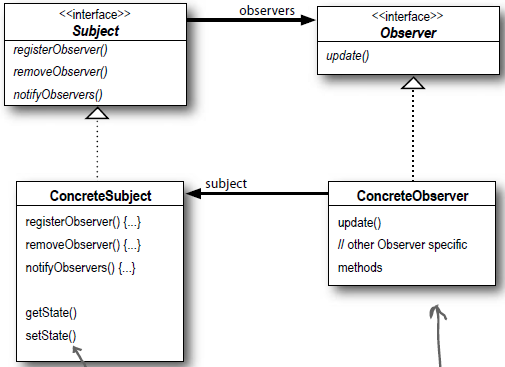
We use the observer pattern to solve the problem described. The subject is the time machine while the observers are the notification controller and the calendar view controller. In this way, the subject only needs to add the two observers as its listeners and they are notified automatically on data changes. It does not need to know how to update its observers. On the other hand, the two observers listen to the time machine and update themselves on data changes.

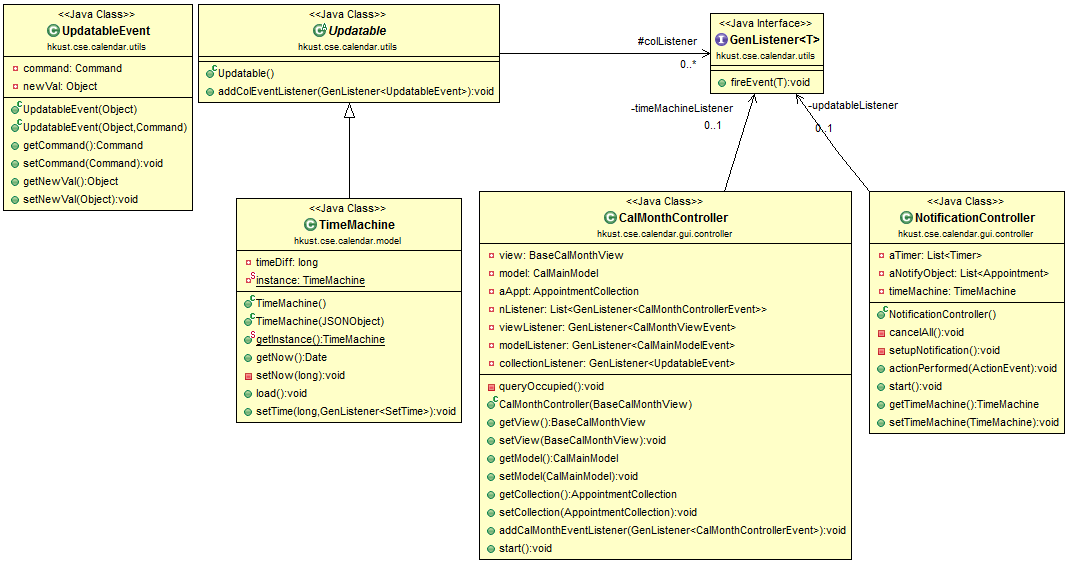
The observer pattern achieves loose coupling of the time machine and its observers by utilizing the programming to interface. This is also provides the further maintenance work with a greater convenience, for objects know each other via interfaces. Modification in any observer’s algorithm will not affect the time machine. When any observer wants to change the way it updates itself, it can merely modify the algorithm while remaining the same interface.

**Class Diagram:**

In our design, the Updatable class is an abstract class and is used as the subject interface. In the Updatable class, the addColEventListener function is equivalent to the registerObserver(). The UpdatableEvent class is an assistant class to the Updatable. The TimeMachine class is a concrete subject that implements time machine operations.

The GenListener class is the observer interface while the CalMonthController and the NotificationController are concrete observers that implement specific updating methods.





**Pattern 3: Factory Pattern**

**Motivation:**

In our project, the GUI is implemented with different types of views. The main application DCalendarApp creates an object viewManager from the class ViewManager. The viewManager is instantiated with attributes of different classes of views (for example PrimCalMainView). However, the creation and management is handled by individual getxxxView() functions in the viewManager. This creates unnecessary troubles in writing and maintaining the code. If a common part of the different view classes need to be changed, or if we need to add new view elements to improve the look and functionality of the calendar program, then the code inside every view class need to be modified. The viewManager needs to be modified as well. Thus it is reasonable that we implement a viewFactory class to incorporate the instantiation of all the view classes into one class in the future, instead of instantiating them one by one inside the viewManager class.

**Solution:**

We use the simple Factory Pattern to solve the described problem. The ViewManager class is the client of the factory. It will go through the ViewFactory to create the instances of views. The ViewFactory class now is the only part of codes that refers to concrete view classes. The View class is an abstract class, and is considered as the product of the ViewFactory. In it, common functions or actions of different xxxView classes may be declared. All the xxxView classes, such as the BaseCalMainView and the BaseLoginView, are concrete products. Each of them extends the abstract View class and becomes concrete. Finally, these products, created by the ViewFactory, will be handed back to the client, or the ViewManager.

**Class Diagram:**

Compare our design with the sample design in lecture:

The ViewManager is equivalent to the PizzaStore that orders products. The ViewFactory is equivalent to the SimplePizzaFactory that refers to concrete product classes. The View is equivalent to the Pizza that is an abstract class. And all xxxView are equivalent to xxxPizza that are concrete products.

