|  |
| --- |
|  |
| **COMP 3111: Introduction to Software Engineering** |
| **Project: Architecture Document** |
|  |
|  |
|  |

**Group 9 Silly Pig Pig**

**Members:**

LEE, Wing Ki (20113413)

WU, Jimmy (20113786)

CHOY, Bing Son (20122218)

LAU, Tsz Hei (20113451)

|  |
| --- |
|  |

1. **Introduction**

This architecture document will describe the application of design patterns in the calendar project. We are going to identify the two existing design pattern, namely singleton and factory pattern, used in the calendar project. Apart from that, we will propose an additional design pattern, the strategy pattern for improvement.

1. **Design patterns used**
2. **First pattern**
3. Name:

Creational Pattern – **Singleton**

1. Motivation:

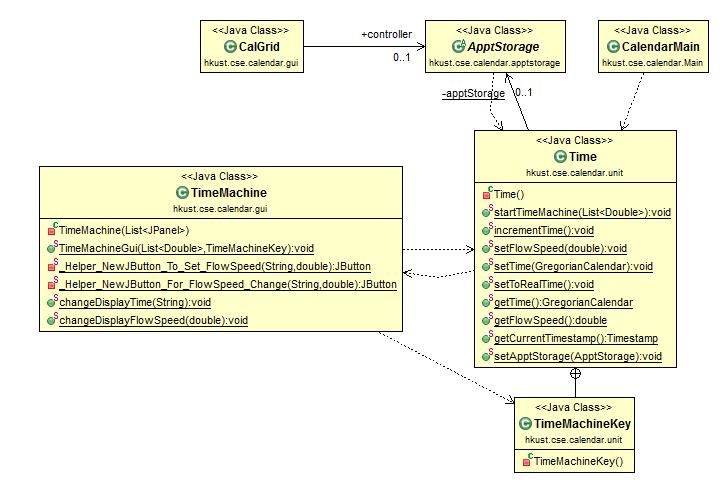
Time is a universal concept. It is absolute and should not be duplicated. Therefore, there should be only one unique instance of ‘Time’ being used. It would have been no problem if the code written for the calendar program is simple, or only a few classes and programmers were involved.

However, there might be problems when many classes and programmers are involved. Based on the beauty of encapsulation in computer software engineering, some programmers might not realize that the classes they are using have already created a ‘Time’ instance called ‘T1’ and thus creating an additional ‘Time’ instance called ‘MyTime’. If that is being the case, different `times’ are involved in the calendar, which might lead to the occurrence of time related conflicts. Thus, we made use of the Singleton pattern to solve this problem.

1. Solution:

Using Singleton, the constructor of the ‘Time’ is made private. Only one instance for `Time’ is allowed that no more additional instances can be created. Idea of singleton pattern can be achieved using a function ‘startTimeMachine’. ‘Time’ object will be constructed only when there is no any ‘Time’ object exists. Then, programmers use the same ‘Time’ instance to manipulate the other classes and functions. Time related conflicts could then be prevented.

1. Class diagram:



Explanation:

From the graph above, we can see that only one instance for `Time’ is allowed. Additionally, ‘TimeMachine’ is also a singleton class because its constructor is set to be private. In order to create a GUI to of the ‘TimeMachine’ object, ‘CalendarMain’ needs to call a function called ‘TimeMachineGui’ in ‘TimeMachine’ class. Although the function is set to be public, it cannot be called directly by any other class except for Time. It is because it requires a private parameter ’TimeMachineKey’, which is only owned by Time.

Therefore, all classes need to access this function through the class ‘Time’. This can prevent the time to be disrupted by other classes. Usage of singleton and private key in ‘TimeMachine’ can greatly keep the variables stored in ‘Time’consistent.

1. **Second pattern**
2. Name:

Creational Pattern – **Factory**

1. Motivation:

In this calendar program, the functionality and layout of sign up dialog and edit user dialog are similar. In fact, these two dialogs can be implemented in an independent way, such as by constructing two classes for each dialog separately. However, this causes code duplication. The existence of duplicate code may increase maintenance costs and more time is needed for the programmers to have further modifications.

We found that these two dialogs can share common GUI and action listener codes. In order to prevent code duplication, we needed to find a way to reuse the common parts to construct sign up dialog and edit user dialog.

1. Solution:

The solution is to make use of factory pattern.

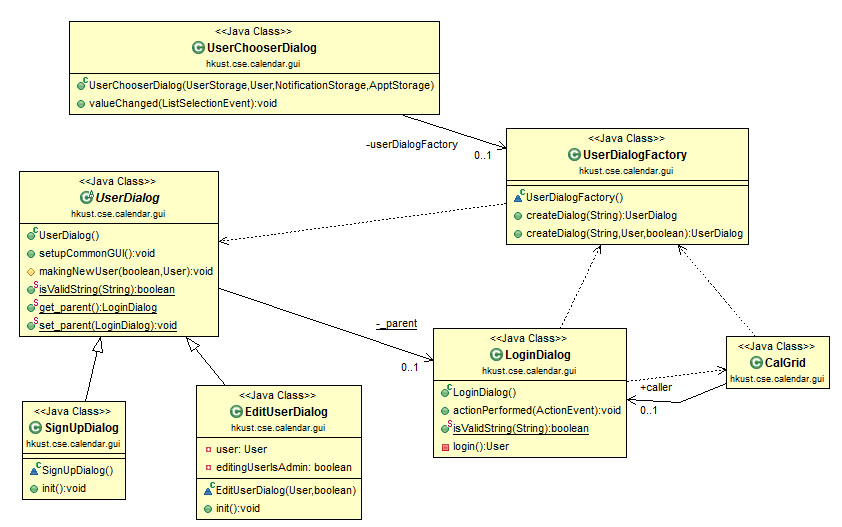
We created a class called ‘UserDialog’ (in UserDialog.java) to construct the fundamental GUI components and corresponding action listeners for dialogs first. As the layouts of two dialogs are not exactly the same, we have to construct two more classes (‘SignUpDialog’ and ‘EditUserDialog’), which inherit ‘UserDialog’, for further constructions of two dialogs. Such as instruction, title, etc.

The ‘UserDialogFactory’ class is created as a factory to create the corresponding dialog. There is a function called createDialog inside this class. Given parameters (string to determine which dialog to create, and other necessary information if needed), dialog construction will be done.

To construct a sign up or edit user dialog, what we need to do is to create a ‘UserDialogFactory’ object and call the function createDialog with suitable parameter. Then the ‘UserDialog’ object will be acted as a sign up/edit user dialog.

The advantage of using factory pattern is that its implementation is simple. This approach allows the reusability of existing functionality, and hence to prevent code duplication. Maintenance cost can also be reduced. When we need to construct a new dialog that has a similar layout and functionality with ‘UserDialog’ in the future, we can use the same approach to reuse the implemented codes.

1. Class diagram:



Explanation:

Three classes ‘UserChooserDialog’, ‘LoginDialog’ and ‘CalGrid’ are the clients that they want to call either sign up dialog or edit user dialog. As the class ‘SignUpDialog’ and ‘EditUserDialog’ inherit class ‘UserDialog’, they share common components such as GUI and action listener in ‘UserDialog’. After using the factory pattern, what clients need to do is to create a ‘UserDialogFactory’ object, and call ‘createDialog’ function (with certain parameters) for the dialog construction. Specific dialog (product) can be created from this factory.

1. **Improvement**
2. Name:

**Strategy pattern**

1. Motivation:

This calendar program needs to interact with different types of user that with different permission (i.e. administrative user and regular user). However, recently there is only one class ‘User’ to create users in this calendar program. Variable ‘mType’ is used in the determination. For each menu and dialog, a series of checking is needed to identify whether the current user type has permission to take certain actions or not.

In the future, new types of user may be required in this program. If a new type of user added, it would be a mess for the programmer as they need to add such a new condition to each menu and dialog involved. Hence we have to find a new implementation method to prevent such a case.

1. Solution:

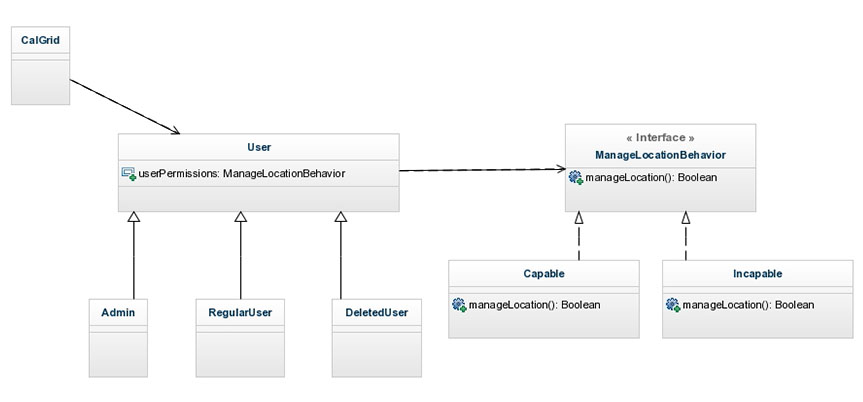
Strategy pattern is a solution to this problem. To apply this design pattern into the calendar program, we first improve ‘User’ class structure. Several classes that inherit ‘User’ will be created. They are ‘Admin’, ‘RegularUser’, ‘DeletedUser’, etc.

Then for each menu/dialog that we need to obtain the user’s permission data, we have to create an interface. Take location dialog as an example. If the current user is an administrative user, he or she has right to modify the location information. In this case, related actions of showing location dialog will be done only when the user is an administrator.

We make an interface called ‘ManageLocationBehavior’ that contains a function ‘manageLocation’. If the current user has the permission to manage location information, ‘manageLocation’ will take the corresponding actions to support such a feature. Otherwise, it would not take any action. Also, this function returns a boolean value to indicate whether the user has permission to manage location information or not. Two classes ‘Capable’ and ‘Incapable’ implement this interface. Using the new implementation, we can know which type of user can manage the information of location.

This design pattern takes what varies and “encapsulate” it. On the other hand, it means that the independent codes are separated. When changes are required, it would not affect the rest of the code. This enhances the code flexibility.

1. Class diagram:



Explanation:

In this class dialog, we used a feature ‘manage location information’ as an example. ‘CalGrid’ is a client to request the user permission. Using strategy pattern, client can obtain this information via user class (Admin, RegularUser and DeletedUser). It is because they inherit ‘User’ class, which has a variable called ‘userPermissions’. This variable contains the return value to identify user’s permission. If the return value is true, it means manage location feature is available to current user.

**IV. Conclusion**

All in all, this architecture document describes two design patterns used in the calendar program. The first one is singleton, while the second one is factory. Both of them belong to creational pattern.

We applied the pattern of singleton to construct ‘Time’ class. This implementation can assure only one and same instance of object every time.Therefore, the ‘time’ object will not be disrupted by other classes.

For factory pattern, we created ‘UserDialogFactory’ class to construct the corresponding products (sign up dialog and edit user dialog). Using this approach would enhance the reusability of code, and also prevent code duplication. Not only that, maintenance cost can also be reduced.

Moreover, we also proposed a new design pattern for improvement. It is strategy pattern that decided to improve the structure of ‘User’ object. This method separated independent codes, so that the rest of code will not be affected even there are changes.

With these improvements, we believed that the calendar program will become a more organized and understandable program, and also it is easy for users to make use of the calendar.