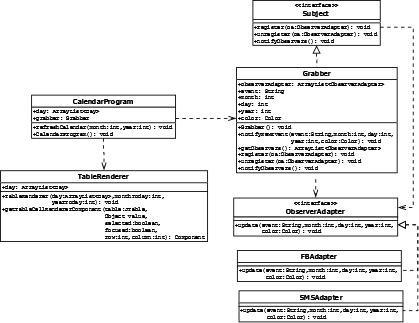
|  |  |
| --- | --- |
| Full Names: | Aquino, Kurt Neil / Caingles, John Israel / Cheng, Jan Kristoffer / Salceda, Francesco |
| Section: | S17 |

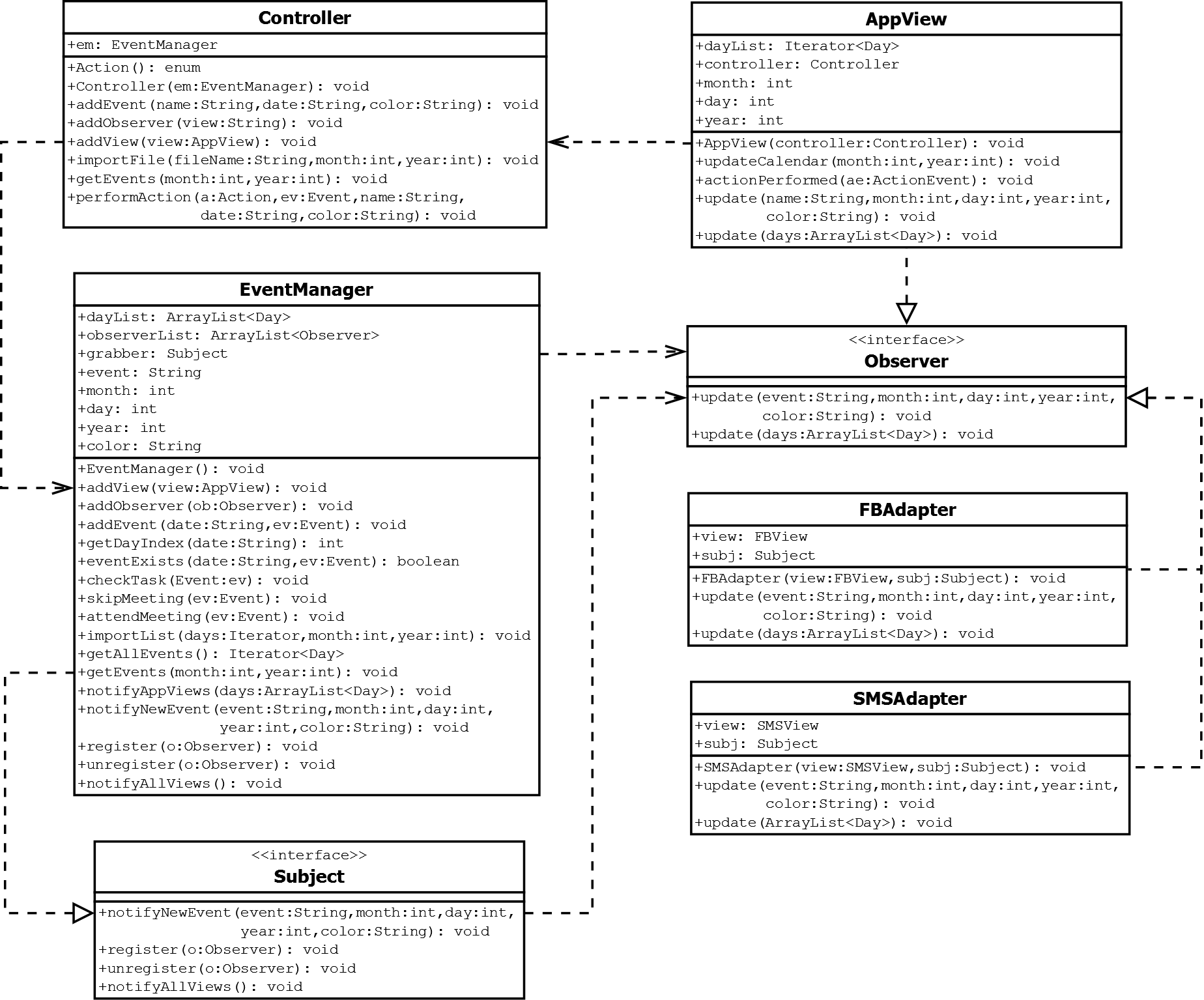
**Calendar App Part 2**

1. **Identify which aspects of your original design were affected. How was your design flexible to adapt to new requirements?**

* One of the main issues of our previous program is that the View is tightly coupled with the Model itself. Very much so that most of the logic regarding the updating of the calendar is in the GUI as well. This is because the pattern we implemented was the Observer Pattern. We delegated a Subject class to contain most of the logic for data manipulation and access, and assigned an Observer interface or the different AppViews; the problem was we did not consider the main calendar program as an Observer, but as a completely different type of View. Since we only considered the AppViews to be the only ones implementing the Observer interface, the main calendar did not. This resulted into the direct access of the View (CalendarProgram) to the Model (Grabber); this violates the Open-Closed Principle and if ever we decide to modify the GUI or the Model, we will have to modify most of the code regarding the updating of the Calendar Program.
* In order to fix this, we implemented the MVC Pattern, delegating the Grabber as the Model, the main Calendar Program as well as the different AppViews as the View, and implemented a Controller in which all operations regarding the updating of the View, to the accessing of the Model will be performed without having both of them to directly communicate with each other. This loosely couples the classes, allowing less code to be affected when modifications are to be added to the code or when changing or improving the GUI.

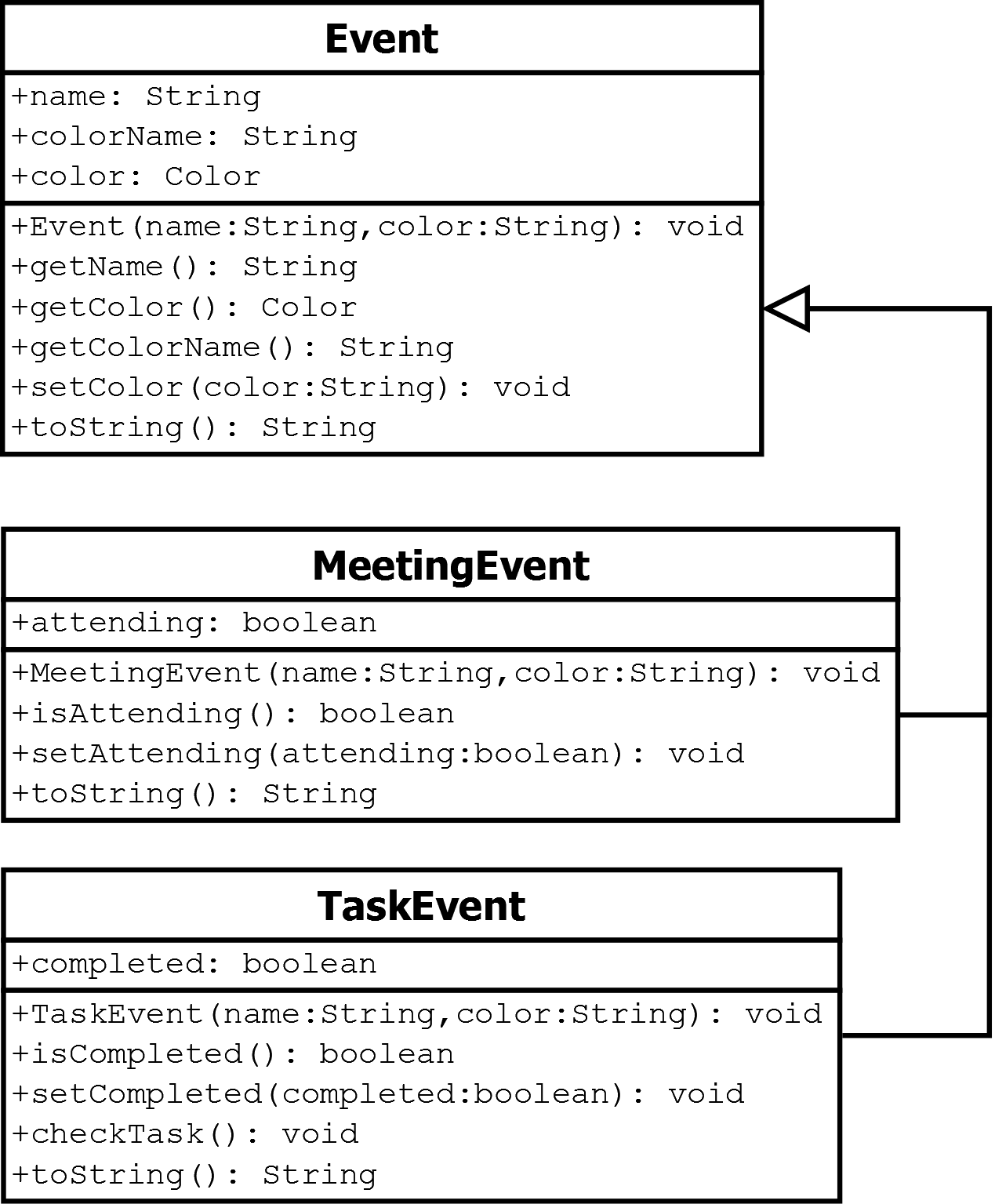


Previous Implementation

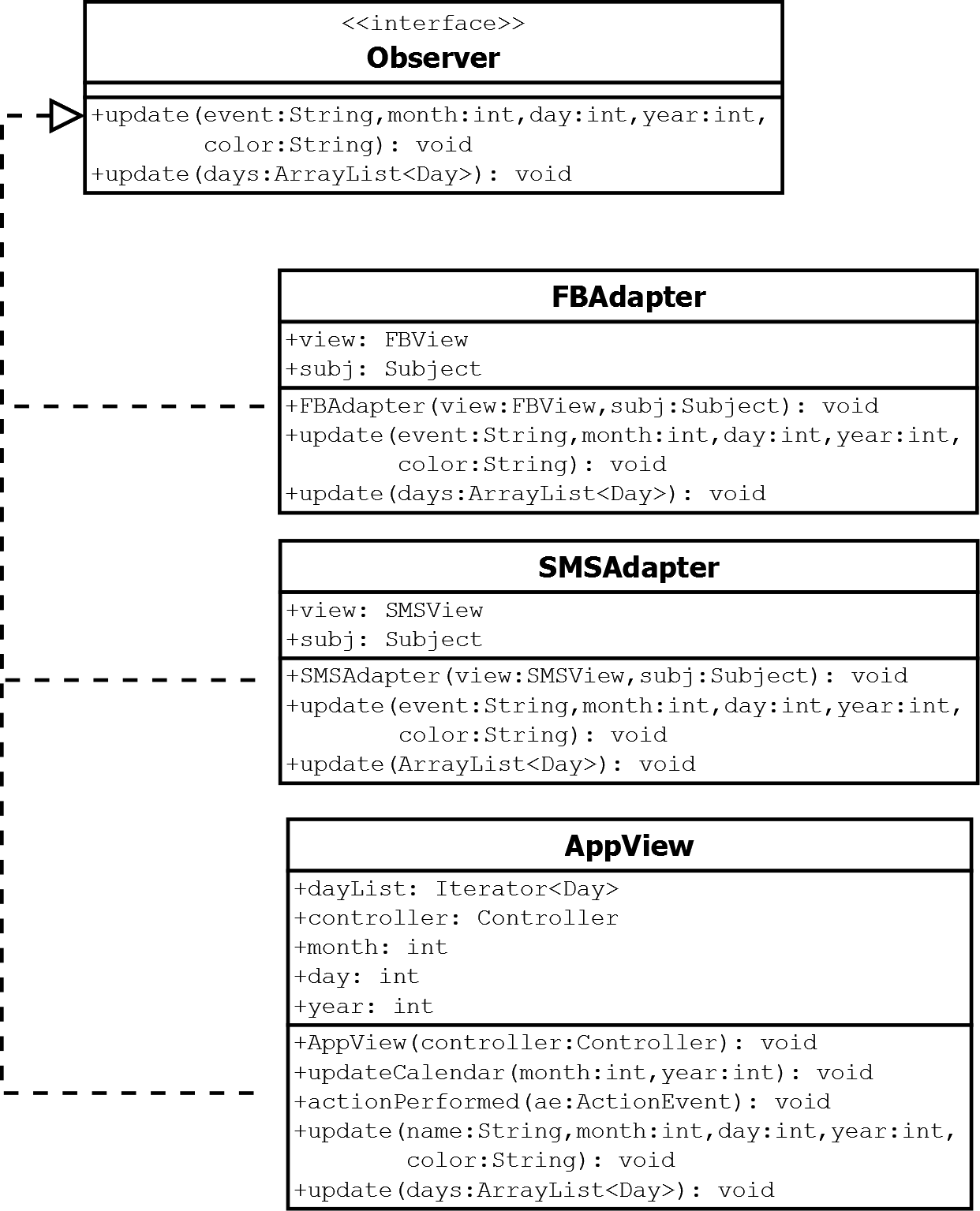


New Implementation

* Since we already made an Event class in our previous program, all we needed to do so that we could implement new types of Events was to extend the base class and create subclasses with their new corresponding attributes.

****

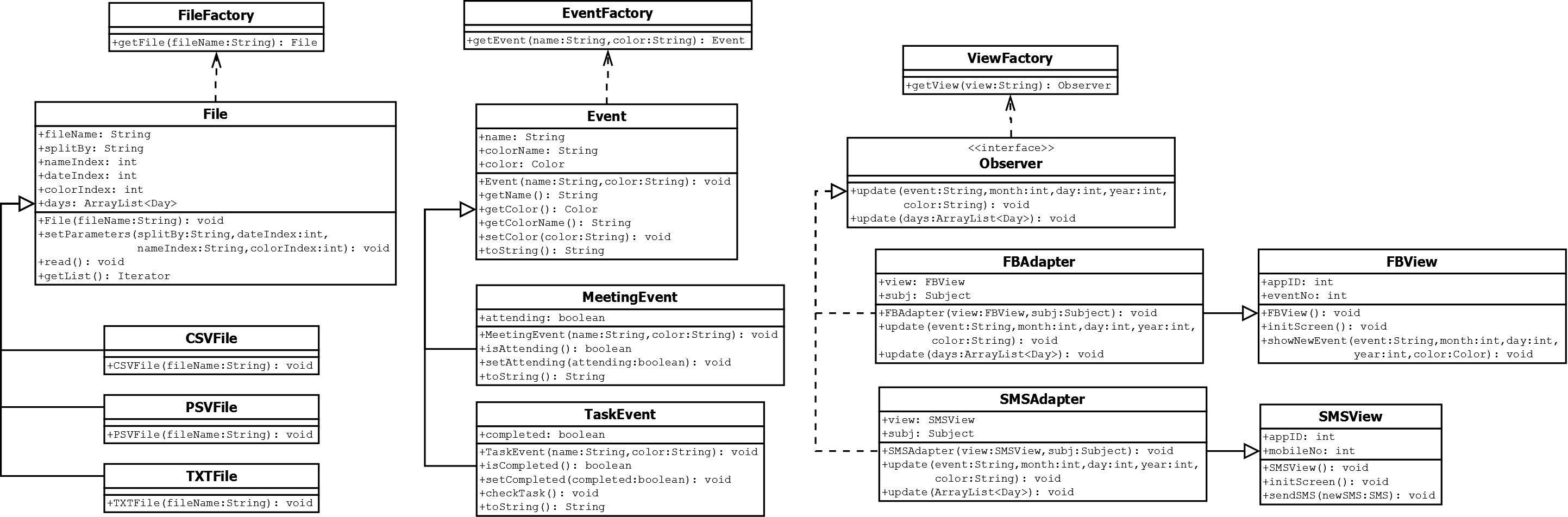
* Given that we have already implemented the Observer Pattern in our previous program, creating the new Calendar Program GUI was easy. Since it implements the previous Observer interface, updating the display, as well as the different views, only required to call the given method, thus removes the necessity of directly accessing the Model for getting information.

****

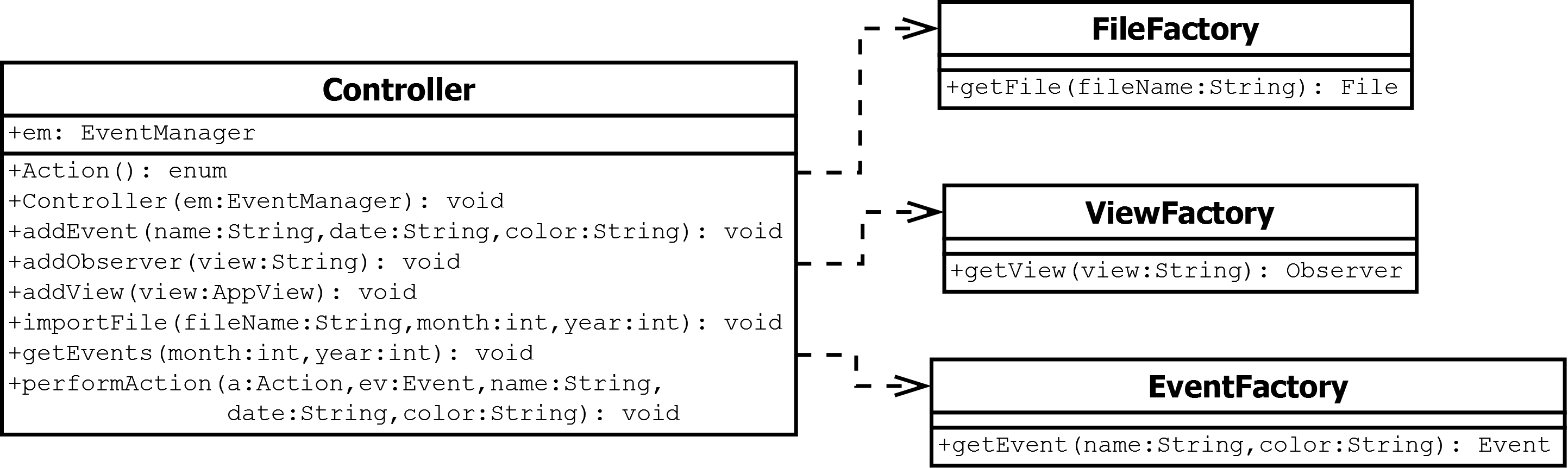
* With our previous program already implementing multiple Strategy and Adapter Patterns, implementing their respective Factory Methods was easy.

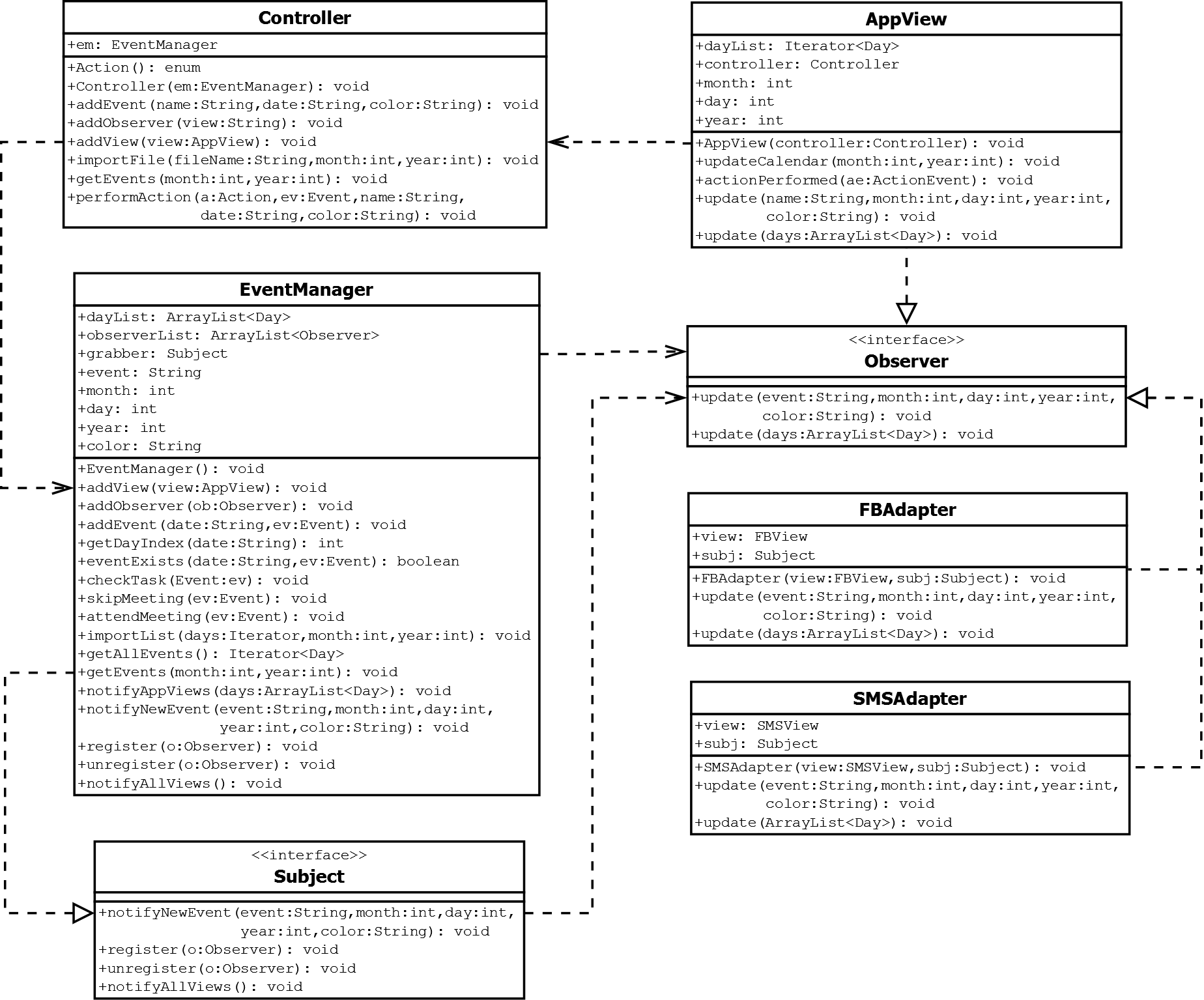
1. **How is your implemented design for this design challenge going to be flexible in anticipating similar changes to your calendar component in the future?**

* We applied multiple patterns into the code so as to ensure program flexibility as well as adaptability for possible future modifications.
* We have considered which patterns could be implemented to every aspect of the Design Challenge while keeping in mind Sir Briane’s advices and comments to the previous Design Challenge.
* New Patterns used:
  + Factory Pattern
    - One of the patterns mostly implemented into the program was Factory. Given that the code should be able to have a flexible design when it comes to adding similar elements or implementations, the main classes should not be heavily modified, in a sense that repetitive code will appear.
    - A Factory Pattern has been implemented to instantiate different types of Event classes. With the “Info Event” being the abstract or base class, both “Meeting” and “Task” Events will just be considered as a subclass with their additional corresponding attributes (e.g. attending for Meeting & complete for Task). The color of each respective event will be used as a determinant to identify what type of Event it is; Blue for Info Events, Yellow for a Meeting to be Attended, Orange for a Skipped Meeting, Green for a Completed Task, and Red for a Pending Task.
    - It also has been implemented for the instantiation of different File formats since our program is decide to import data and information from psv, csv, as well as txt files. This pattern will be used whenever we decide to import files to be loaded into our Calendar Program. The determinants for which type we want to open will be based on the filename (e.g. if it ends with “.psv”, “.csv”, or “.txt).
    - And finally, a Factory Pattern has also been implemented for the instantiation of different AppViews for the program. Given that we have already made Adapters for the different View APIs for DC1, all we needed to do was to integrate them with our ViewFactory. Since there might be a possibility of adding more APIs and AppViews in the future, it would be better to have a single Factory method in which it would determine what type of View the user wants to use, rather than adding a respective function call for every new type of view to be added.

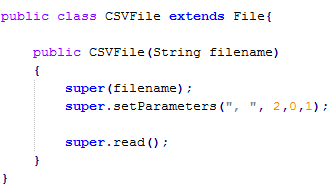


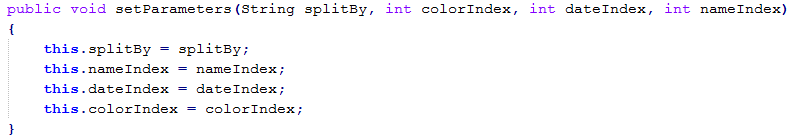
* + Model View Controller Pattern
    - Given that our previous implementation in DC1 already applied the Observer Pattern in order to delegate a Subject class containing the different Observers and Observables as well as updating them, all we needed to do was to integrate this pattern into our new code in a way that the Observers, or the Views, are not as tightly coupled as with our previous code.
    - By assigning the Subject class as the model, and the different Observers as the View, all that is left is to create a Controller in which most of the communication between the View and the Model will occur.
    - Our implementation of the MVC Pattern is designed in a way that before the Views are updated, given a command from the Controller, the Model must first be updated as according to the received instruction. If the Model has been updated successfully, it then notifies the Views to be updated.
    - The Views are not directly connected to the Model as to ensure the Single Responsibility Principle. Since the Views are just there to display the information and accept user input, all it is required to do is fetch and send information to the Controller, and the Controller itself will perform the operation requested by the view.
    - And since the Controller is the one who fetches and performs the operations to update, access, or modify the data in the model, it is the class that has the access to most of the Factory methods so that it could instantiate the different types of classes required for updating the Model, based on the requested instruction from the user in the View.





* Reused / Modified Patterns
  + Strategy Pattern
    - Since the previous implementation of our File parsers had repetitive code, for example, the reading from both csv and psv files used the same syntax for parsing through the file, but the only differences between then is the order of the data to be fetched as well as the split character, the implementation itself is inefficient.
    - In order to modify this part of the program, we used a base class named File, which has all of the methods required for parsing through a file and saving the data. It now only requires a split character, and the index of the required information based on the format of the file, as parameters to differentiate between the multiple types of files to be read. For example, all csv files may require a comma (,) as a split character, and the indexes of the data required, in our program’s case, the Event Name, Date, and Color are 2, 0, and 1 respectively, all we need to do is to create a subclass of File for the CSV reading, setting these information as parameters for the file parsing in the superclass. The same goes as well for both psv and txt files.

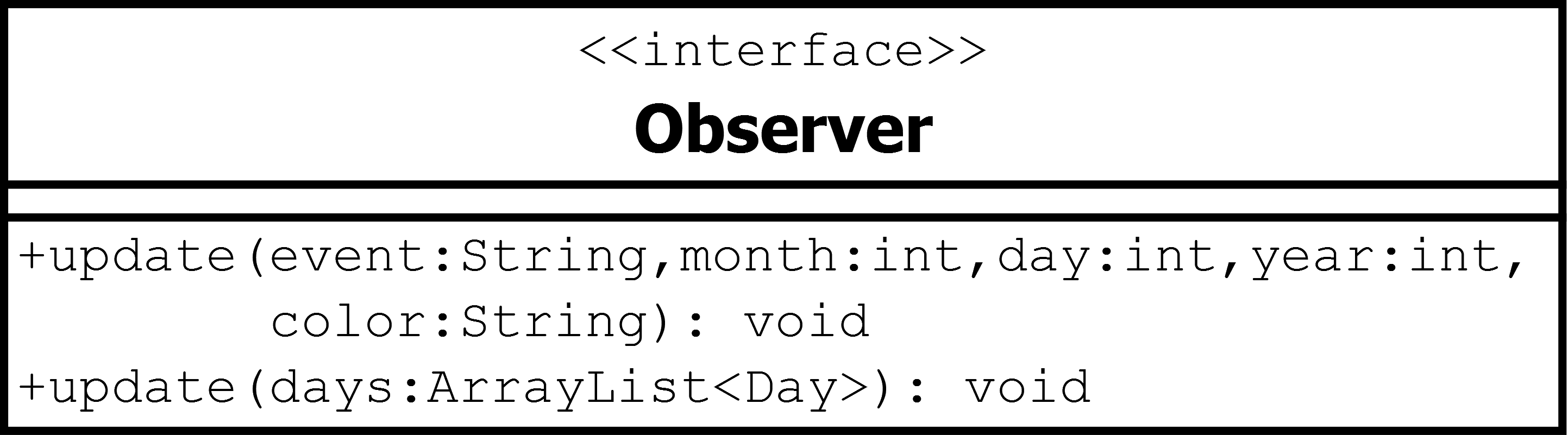




* + Observer Pattern
    - As mentioned earlier, we integrated the Observer pattern we have implemented in our previous code, and reused it to comply with the MVC pattern. The only difference is that there is another update method, different from the implantation of the previous one. The first update method is just used when updating the different AppViews from our previous code. With the new implementation of the code to apply the MVC Pattern, the second update method is used to update the data in the main Calendar Program itself.



Previous Observer Interface



New Observer Interface