Archival View

The class diagram below depicts the important methods and attributes that provide us the ability to switch views between the main list and the archival list.

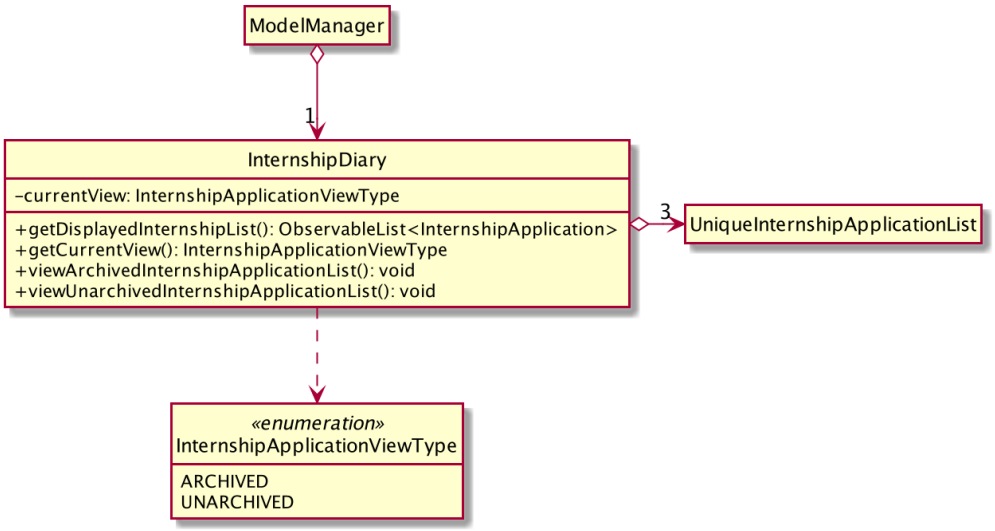

Figure 1. Structure of `InternshipDiary` that showcases the methods and attributes required for view-switching

The object diagram below illustrates the three `UniqueInternshipApplicationList` objects maintained by `InternshipDiary`.

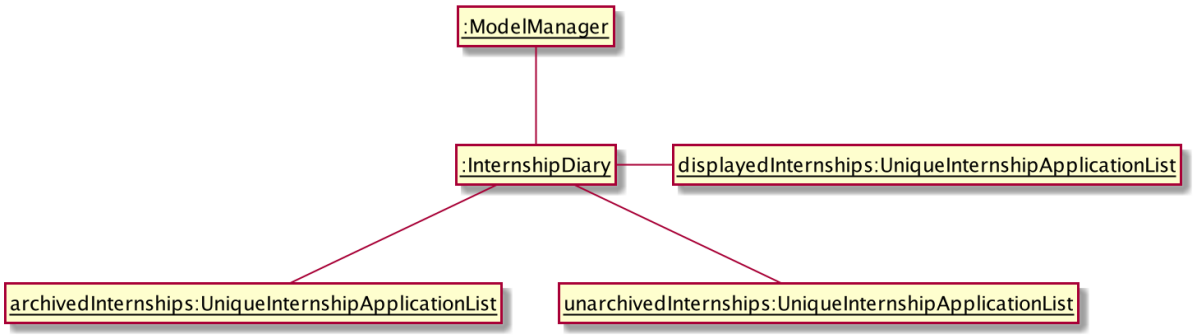

Figure 2. Object diagram to illustrate the three `UniqueInternshipApplicationList` maintained by `InternshipDiary`

As the name suggests, `displayedInternships` is the list that is shown to the user in the GUI. It references either `archivedInternships` or `unarchivedInternships` at any one time. When a user is viewing the main list, `displayedInternships` references `unarchivedInternships`. And when a user is viewing the archival list, `displayedInternships` references `archivedInternships`.

The following sequence diagram illustrates how an `archival` command is executed. The `list` command is similar to `archival`. You may use the same sequence diagram for the `list` command.

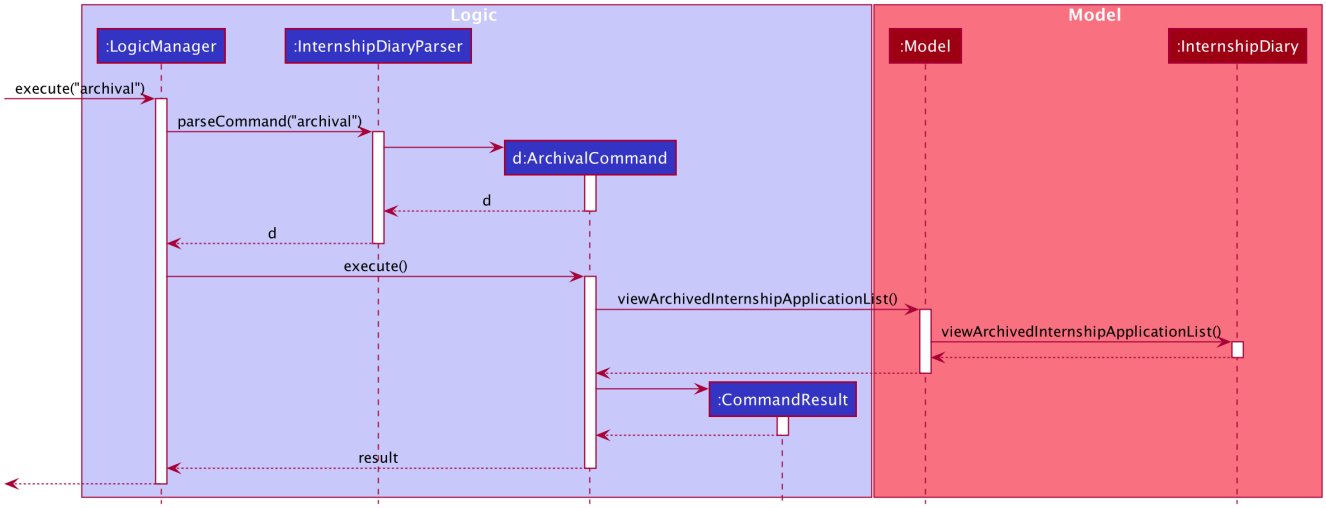

Figure 3. Sequence diagram for `archival` Command

The following code snippet is retrieved from the `InternshipDiary` class. It illustrates the internal workings of how we switch the view between the archived list and the main list.

```
public void viewArchivedInternshipApplicationList() {
    this.displayedInternships = archivedInternships;
    this.currentView = InternshipApplicationViewType.ARCHIVED;
    firePropertyChange(DISPLAYED_INTERNSHIPS, getDisplayedInternshipList());
}
```

It can be seen from the code snippet that we make use of referencing to switch between the views of archival and main list. However, such implementation brings about reactivity ssues — where elements that reference `displayedInternships` will not be aware of the reference change in `displayedInternships` whenever the user executes `archival` or `list`. Therefore, in the above scenario, users would still see the main list after executing the `archival` command.

To resolve this issue, we need to employ the **observer pattern design**. The broad idea is to assign each UI element to be an **observer** and `InternshipDiary` to be the **observable**. Consequently, whenever there is a state change to `InternshipDiary`, the list of observers will be notified and updated automatically.

To achieve this observer pattern, we made use of the `PropertyChangeSupport` class and the `PropertyChangeListener` interface. `PropertyChangeSupport` is a utility class to support the observer pattern by managing a list of listeners (observers) and firing `PropertyChangeEvent` to the listeners. A class that contains an instance of `PropertyChangeSupport` is an observable. On the other hand, a class that implements the `PropertyChangeListener` interface is an observer.

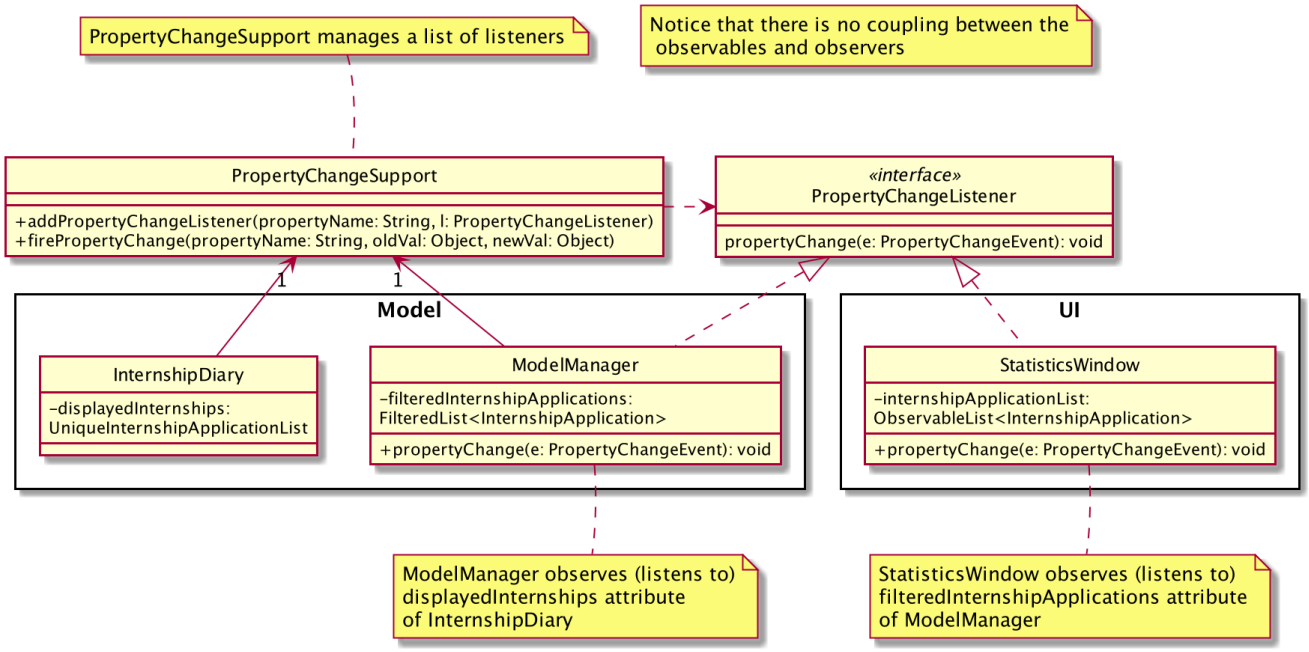

Figure 4. Implementation of a two-tier observer-observable structure

The class diagram above showcases our implementation of a two-tier observer-observable structure: 1) `InternshipDiary` is an observable, 2)`ModelManager` is both an observable and observer; it observes any changes to `displayedInternships` contained in `InternshipDiary`, 3) `StatisticsWindow` is an observer; it observes any changes to `filteredInternshipApplications` contained in `ModelManager`.

- `InternshipDiary` and `ModelManager` each contains an instance of `PropertyChangeSupport` to manage their listeners.
- `PropertyChangeSupport` serves as the intermediary and an abstraction between the **observables** and **observers**.
- Observers are generalized (polymorphism) as they implement the `PropertyChangeListener` interface; these observers are managed by `PropertyChangeSupport`.
- All the UI elements in our implementation follow the above class diagram; `StatisticsWindow` is an example.

We will briefly discuss how the observer pattern works in our implementation.

Whenever an object wants to observe changes in another object, it will call the `addPropertyChangeListener` function of the `PropertyChangeSupport` instance from the appropriate object that it wishes to observe. It will also have to specify which property of that object it wants to observe.

In our case, when `ModelManager` is created, it will call the `addPropertyChangeListener` function of the `PropertyChangeSupport` instance belonging to `InternshipDiary`. The function call will look like this: `addPropertyChangeListener("displayedInternships", this)` where `this` is a reference to `ModelManager` itself (so that it can be registered as a listener of the `displayedInternships` property of `InternshipDiary`).

The process is similar for any UI element that wants to observe the `filteredInternshipApplications` property of `ModelManager`.

As a result, whenever there is a change to the property `displayedInternships` in `InternshipDiary`, the `PropertyChangeSupport` instance of `InternshipDiary` will call `firePropertyChange` to emit a `PropertyChangeEvent` to `ModelManager`. The emitted event will trigger the `propertyChange` function of `ModelManager`. `ModelManager` can then retrieve the new reference from the event and update its `filteredInternshipApplications` accordingly. It will then repeat the event emission process to any UI element (e.g. `StatisticsWindow`) that is observing the `filteredInternshipApplications` property.

The following activity diagram gives a high-level overview of the above event-driven process.

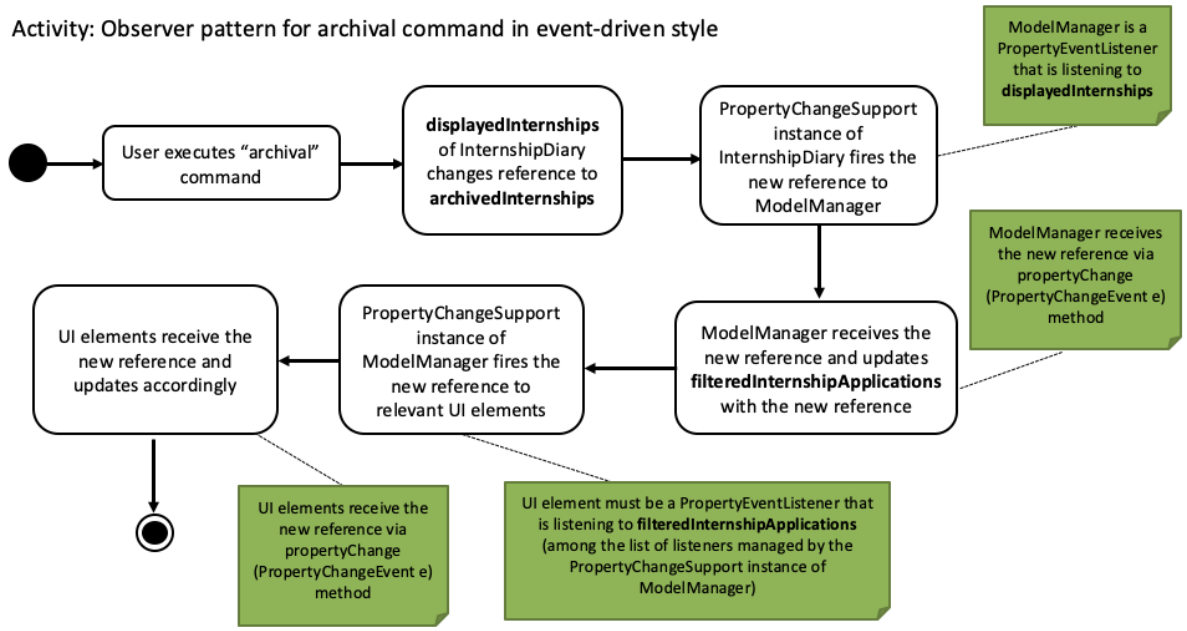

Figure 5. Activity diagram to illustrate the Observer Pattern using `archival` command

Note that the two-tier observer-observable structure is **necessary**.

This is because `list` and `archival` only changes the reference of `displayedInternships`. When 'ModelManager' updates its property `filteredInternshipApplications` with the new reference, UI elements that reference `filteredInternshipApplications` will not be aware of the reference update to `filteredInternshipApplications`. Thus, `ModelManager` has to notify and update the UI elements as well.

As an extension, our team also implemented enumeration for each property that is being observed. This modification ensures type safety and a way for us to track what properties are observed. This is especially important when many properties are being observed.

Below is the updated class diagram with the implementation of `ListenerPropertyType` enumeration.

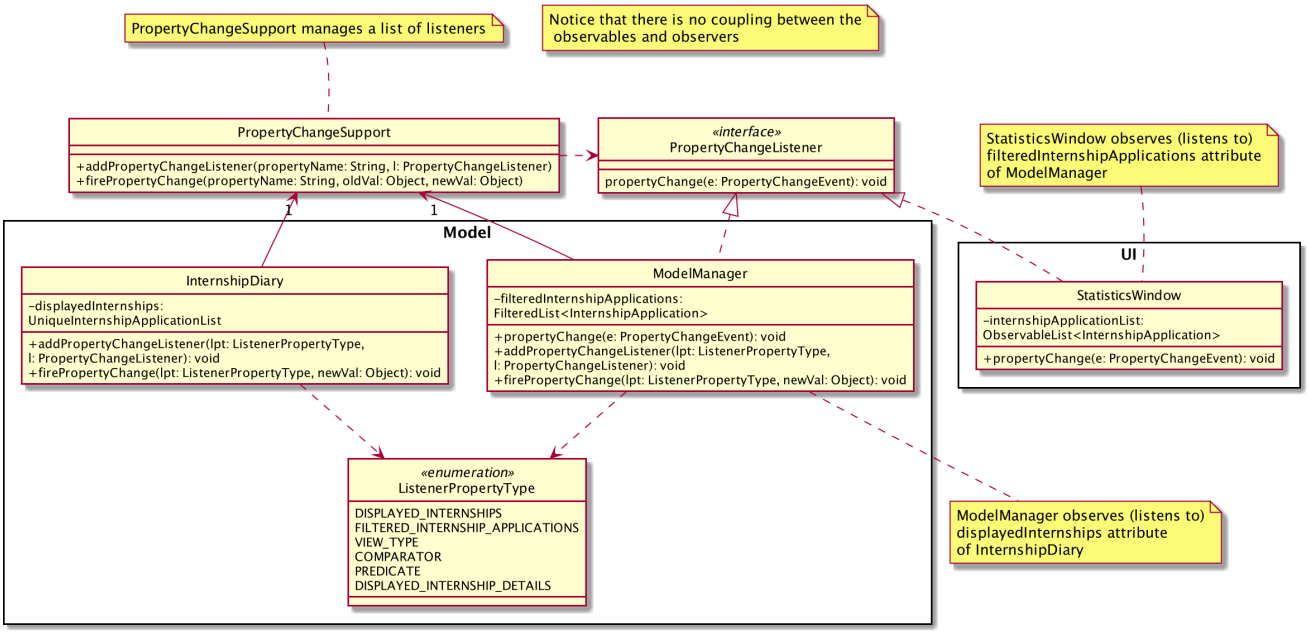

Figure 6. Updated class diagram of the two-tier observer-observable structure with `ListenerPropertyType`

As seen from the diagram above, each observable will implement two additional methods to use `ListenerPropertyType` enumeration as parameters:

1. `addPropertyChangeListener(ListenerPropertyType propertyType, PropertyChangeListener l)`
2. `firePropertyChange(ListenerPropertyType propertyType, Object newValue)`

This forms a layer of abstraction as we would not be allowed to call the `addPropertyChangeListener` and `firePropertyChange` methods of `PropertyChangeSupport` directly.