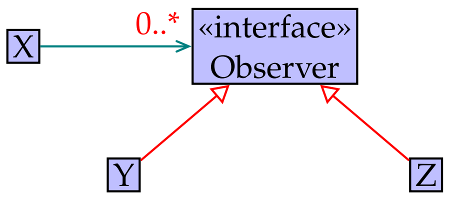
Worksheet 5: Event-Driven Programming

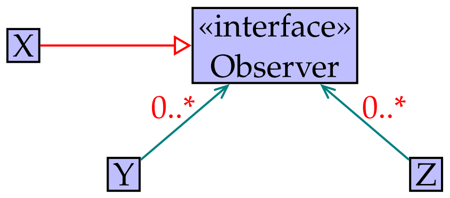
Updated: 24th September, 2014

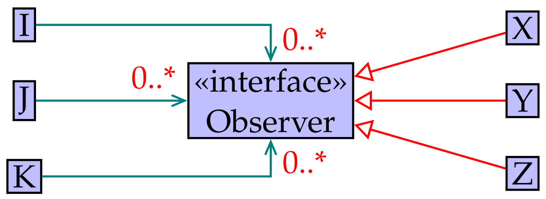
1. Discussion

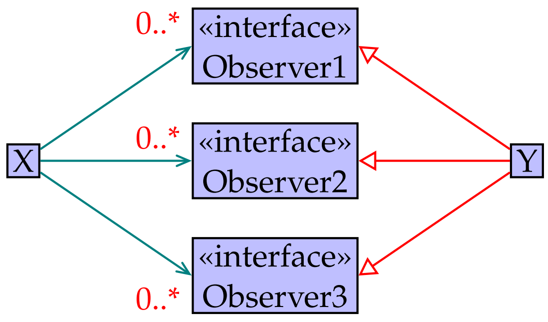
Each of the following diagrams represents a contrived but legitimate arrangement of the Observer Pattern. For each one:

(i) Explain what exactly is going on, and in particular how events are processed.
(ii) Write some brief example code that could be used to set up the object structure. (Assume that each event source class has appropriate mutators for managing its set of observers.)

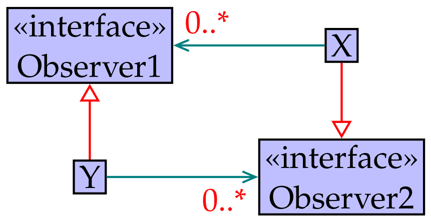
(a)


(b)


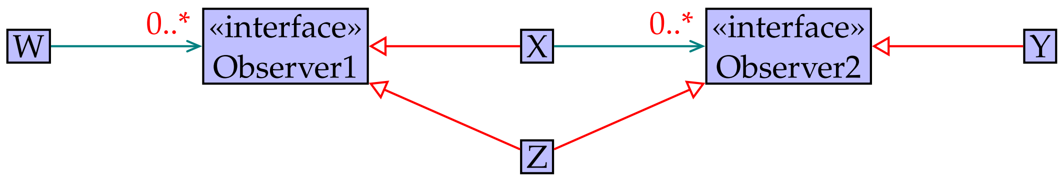
(c)


(d)


(e)



(f)



2. Theoretical Design

Consider the following problem:

A satellite-based weather monitoring system is being developed to detect (1) the formation and (2) the disappearance of cyclones (or hurricanes or typhoons) around the world.

To begin with, the system will have a `Satellite` class that communicates with weather satellites to perform the actual detection. Appearance and disappearance events both come with latitude and longitude (real numbers), to mark the cyclone's location.

When a cyclone is detected, several things must happen (all of which use the location):

- The system must record the incident in a log file.
- The system must email a notification to the local weather agency.
- The system must alert the user, showing the cyclone on a map.

A different part of the system handles each different action.

When a cyclone *dis*appears, the system must log the incident (including location), but no other actions are required. However, other kinds of actions *may eventually* be required in the future, for both formation and disappearance events.

(a) Apply the Observer Pattern to this problem, writing your design in UML.

(b) Explain *why* the Observer Pattern is useful here. What would the system look like *without* the Observer Pattern, and why would *that* design be a poor choice?

(c) Write the initialisation code necessary to create the objects and the relationships between them.

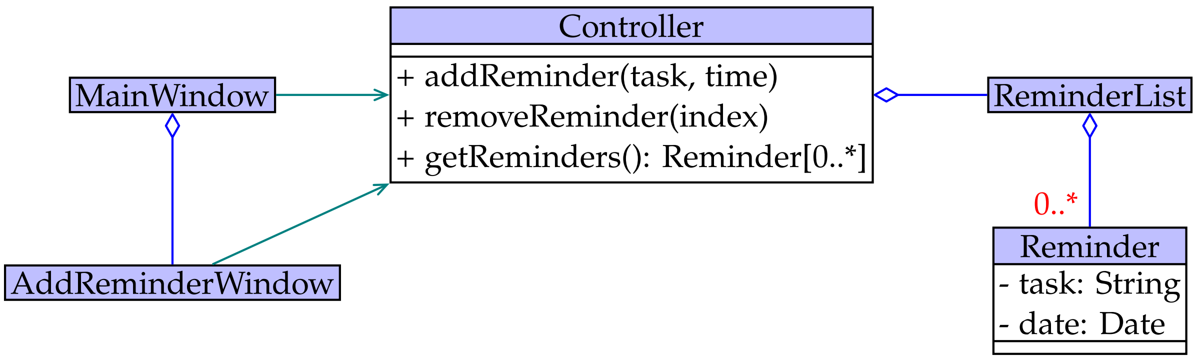
3. Implementing the Observer Pattern

Obtain a copy of Reminder.zip. This is a simple, but somewhat incomplete todo/reminder application.

Note: In fact, it's even less complete than Dave would like. It was intended to actually *remind* the user at specific times about each task. As it is, all it really does is keep track of a list of to-do notes.

You can try adding the actual reminder functionality if you like, but get the more basic stuff working first.

The application is designed as follows. (It uses the Java "Swing" GUI, and MainWindow and AddReminderWindow, both part of the UI, inherit from Swing's JFrame class.)



There are two more classes not shown above:

- ReminderApp, which contains the `main` method.
- FileManager, which contains static `read` and `write` methods for, well, reading and writing a reminder file.

In the design *so far*, the UI can tell the controller to add or remove reminders, and the controller then makes the changes to the model. However, the missing pieces of the puzzle are as follows:

- MainWindow contains a widget for displaying reminder text, but this is not updated when the model changes. The user can technically add a reminder, but nothing will actually show up.

- The file is not updated when the model changes. All changes will be lost when the program exits.

Your task is to use the Observer Pattern to fill in these gaps. For this purpose, you should treat the ReminderList class as the event source. You will then need to consider the following:

- What needs to be added to ReminderList?
- Which classes are going to function as observers, and how?
- How (and where) are you going to set up the connections between event source and observers?

Note: There's no one right answer, but there are many wrong ones!

Make sure you adhere to the principles of the Observer Pattern. In particular, *do not* couple the ReminderList class to any other class.

End of Worksheet