CITS4401 Software Requirements and Design
Semester 1, 2020
Workshop week 9 – Project discussion and event-driven systems

Project questions

Discuss with your team:

- What additional information, beyond what is given in the project specification, would you need in order to do well in this project?

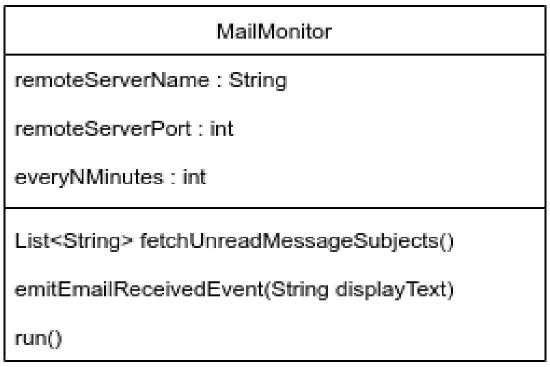
Event-driven programming

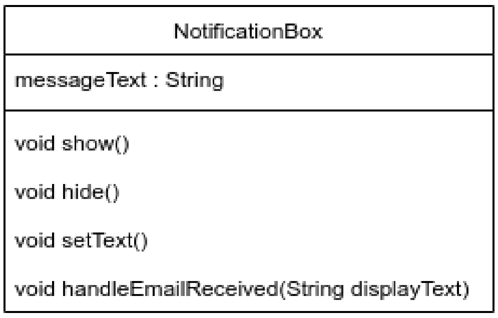
You may notice in the class diagram for the project that it is not at all clear what class methods are called by what. This is often the case with class diagrams – they don't show *behaviour*, a very important part of what are called *event-driven* systems.

Many of the programs you have written for university classes might be described as "procedural" (even if written in an OO language): the program starts, performs some set of tasks – in an order you specify – and then exits.

However, many real-world systems are instead event-driven: the program is "always running", and monitoring to see whether "events" of various sorts occur, and calling on particular objects to perform tasks in response to those events. What order methods are called in (or whether they are called at all) depends on what events occur, which isn't known in advance.

As a small example, an email-alert program might have the following classes:


MailMonitor
remoteServerName : String
remoteServerPort : int
everyNMinutes : int
List<String> fetchUnreadMessageSubjects()
emitEmailReceivedEvent(String displayText)
run()



These would typically make use of a GUI *framework* – a sort of library where the library has overall "control" of the program, and you supply individual classes that can be called.

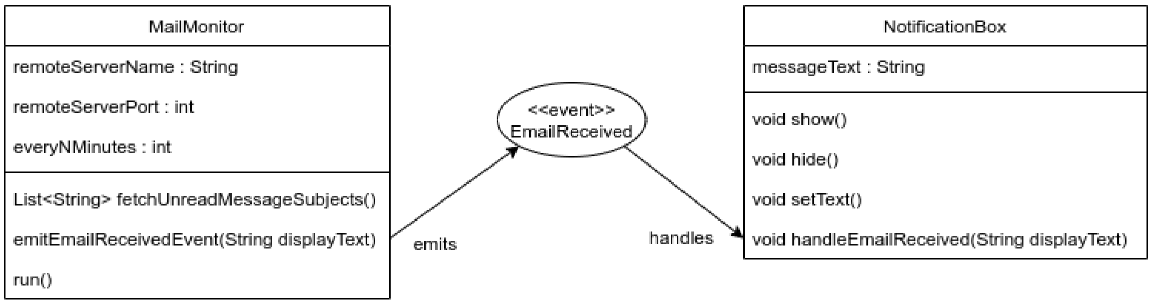
The classes are "wired up" so that:

- When the system starts, the framework calls the `MailMonitor`'s `run()` method. This periodically (`everyNMinutes`) calls `fetchUnreadMessageSubjects()` to see if new messages have arrived on the mail server.
- If messages have arrived, the `MailNotifier` emits an "event", by calling `emitEmailReceivedEvent()`.
- Whenever an event of type `EmailReceived` occurs: the framework calls the `NotificationBox`'s `handleEmailReceived()` method (which sets the text for the notification box, displays it for 3 seconds, then hides it).

Note that `MailMonitor` and `NotificationBox` never actually call or refer to each other – the GUI framework "decouples" classes that *generate* events, from those that *handle* them. (The setup could be done in code, or using a configuration file.)

Using UML, this system behaviour could be described using *state charts* and/or *sequence diagrams*, and various design patterns could be identified which the system uses. (Usually, the framework Observes classes that emit events, and classes that handle events in turn Observe the framework.)

But often, this is overkill, and the behaviour of the system can be described just by showing a sort of "wiring diagram": what event *emitters* are connected (indirectly, via the framework) to what *event handlers*.



Or, even more simply, using just a plain text table:

Event	**Can be generated by**	**Handled by**
EmailReceived	MailMonitor	NotificationBox.handleEmailReceived()

(And in fact, many GUI-designer programs will let you view and edit just such a table.)