KWIC Implemented with Event Based Architectural Style (Assignment 3)

1. Introduction

1.1. Definition of the Problem

The Key Word In Context (KWIC) problem is defined as follows:

The KWIC index system accepts an ordered set of lines; each line is an ordered set of words, and each word is an ordered set of characters. Any line may be circularly shifted by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

1.2. Example

Consider the following set of lines:

- Star Wars
- The Empire Strikes Back
- The Return of the Jedi

The KWIC index system produces the following output (comparison is case-sensitive, that is capital letters are greater than small letters):

- Back The Empire Strikes
- Empire Strikes Back The
- Jedi The Return of the
- Return of the Jedi The
- Star Wars
- Strikes Back The Empire
- The Empire Strikes Back
- The Return of the Jedi
- Wars Star
- of the Jedi The Return
- the Jedi The Return of

1.3. Assignment 3

To accomplish the Assignment 3, you need first to get acquainted with the **existing KWIC system**. After that you need to extend the functionality of the existing system to meet the **following requirements**. Finally, you need to answer to these three **questions**.

2. Event Based Systems in General

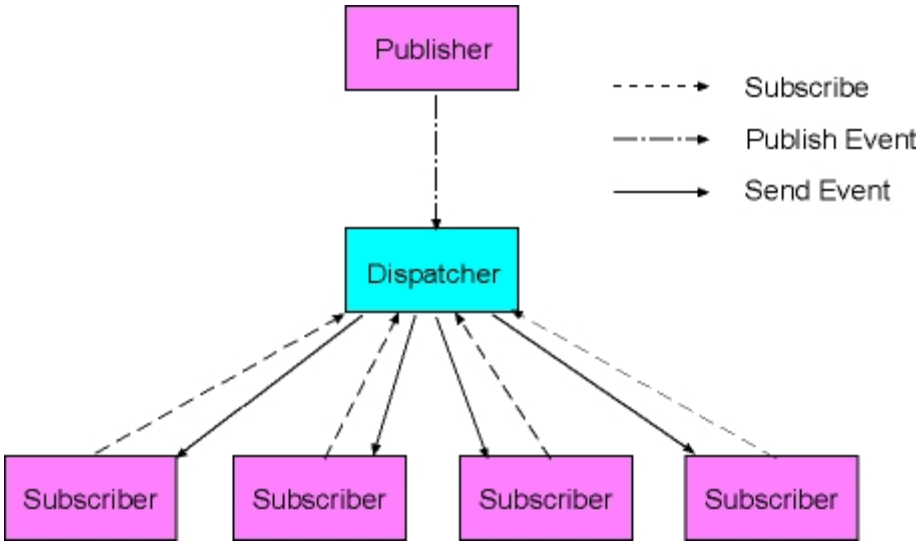
The idea behind event based systems is that instead of invoking a procedure directly, a component can announce or broadcast one or more events. Other components in the system can register an interest in an event by associating a procedure with it. When the event is announced, the system itself invokes all of the procedures that have been registered for the event. Thus, an event announcement "implicitly" causes

the invocation of procedures in other modules. For example, let us consider an Integrated Development Environment for Java. Such IDE consists of tools such as editors for source code, variable monitors, a debugger, etc. Usually, such systems utilize event based architecture. Thus, editors and variable monitors register for the debugger's breakpoint events. When a debugger stops at a breakpoint, it announces an event that allows the system to automatically invoke procedures of those registered tools. These procedures might scroll an editor to the appropriate source line or redisplay the value of monitored variables. In this scheme, the debugger simply announces an event, but does not know what other tools or actions (if any) are concerned with that event, or what they will do when the event is announced.

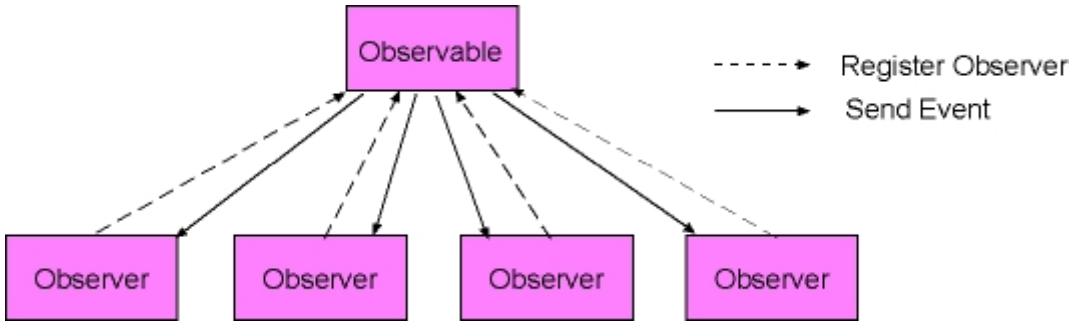
2.1. Strategy of Events Handling

When an event is announced in an event based system, the system itself invokes automatically all of the procedures that have been registered for the event. That means that the system must implement a certain strategy how events are handled in the system, i.e., how events are being dispatched to registered components in the system. There exist a number of different strategies of dispatching events. Generally, we may divide these strategies into two groups:

- Systems with a separate dispatcher module, which is responsible for receiving all incoming events and dispatching them to other modules in the system. The implementation of the dispatcher decides how events are sent to other modules. For example, the dispatcher may broadcast events to all modules in the system, and then modules themselves need to decide what they want to do with events. Another example is that the dispatcher manages lists of modules interested in particular events. In that case the dispatcher sends an event just to those modules that registered for that event. This strategy is usually called Publish/Subscribe strategy and may be seen on the following picture:



- Systems without a central dispatcher module, where each module allows other modules to declare interest in events that they are sending. Thus, whenever a module sends an event it sends the event itself to those modules that registered an interest in such event with that module. This strategy is usually called Observable/Observer and may be depicted as follows:

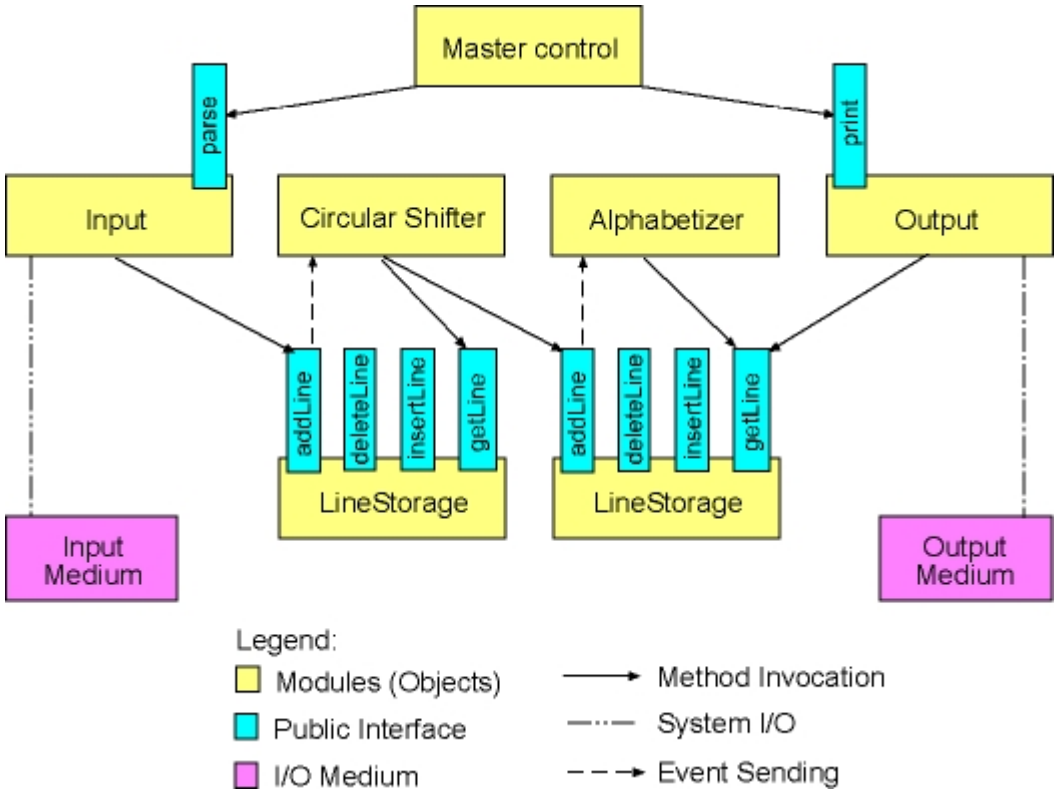


3. The Existing System

Event based KWIC system is composed of the following components:

- Two LineStorage modules. The first LineStorage module is responsible for holding all original lines, and the second LineStorage module is responsible for holding all circular shifts.

- Input module, responsible for reading the data from a file and storing it in the first LineStorage module.
- CircularShifter module, responsible for producing circular shifts and storing them in the second LineStorage module.
- Alphabetizer module, responsible for sorting circular shifts alphabetically.
- Output module, responsible for printing the sorted shifts.
- Master control module, responsible for overall control of the system.



3.1. Properties of the Existing System

The existing KWIC system may be described by:

- **Run-Time Event Interactions**
- **Observable / Observers Dispatching Strategy**

3.2. Event Based KWIC System in Run-Time

The existing KWIC system works as follows. The act of adding a new line to the first LineStorage causes an event to be sent to the CircularShifter module. This allows the CircularShifter module to produce all circular shifts of the line and to store them into the second line storage. This in turn causes another event to be sent to the Alphabetizer module, so that this module may sort the newly added circular shifts.

Thus, we have the following event based interactions in the system:

- The CircularShifter registers its interest in the first LineStorage module. The first LineStorage module sends an event whenever a new line has been added to it. The CircularShifter module receives the event sent by the first LineStorage module. As a response to receiving the event the CircularShifter module produces all circular shifts of the newly added line and store those shifts in the second LineStorage module.
- The Alphabetizer module registers its interest in the second LineStorage module. The second LineStorage module sends an event whenever a new circular shifts has been added to it. The Alphabetizer module receives the event sent by the second LineStorage module. As a response to receiving the event the Alphabetizer module sorts circular shifts.

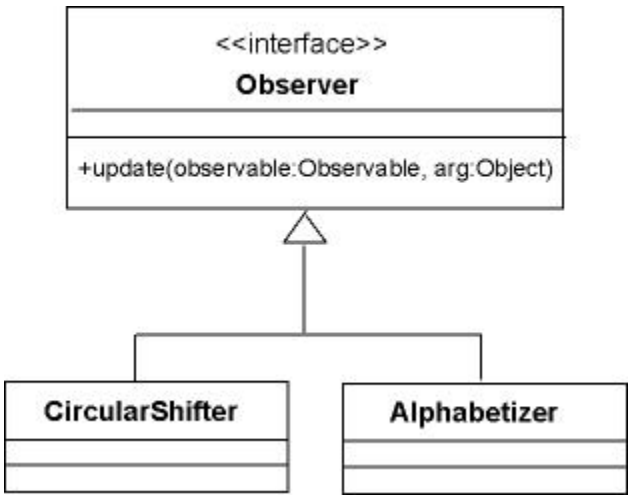
3.3. Observable & Observers

The current KWIC system implements Observable/Observers pattern for handling events in the system. Thus, the two LineStorage modules (the first one applied to store the original lines, and the second one applied to store the circular shifts) are implemented as Observable modules. On the other hand the CircularShifter and the Alphabetizer module are implemented as Observers. Thus, the CircularShifter is an Observer of the first LineStorage module, whereas the Alphabetizer is an Observer of the second LineStorage module.

3.3.1. Java Support for Observer

Since the standard Java library, more specifically the standard java.util package provides the Observable class and the Observer interface, which implement the described behavior, these elements were reused to implement the

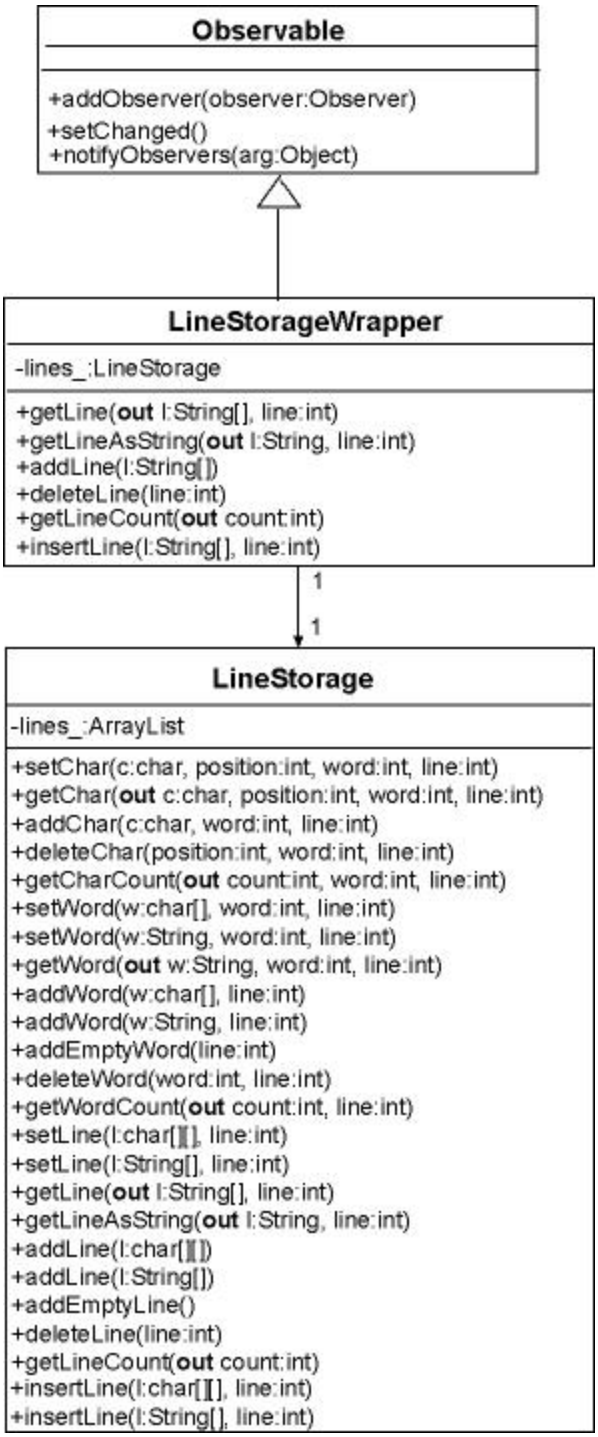
Observable/Observers mechanism in the KWIC system. The following diagram shows the CircularShifter and the Alphabetizer class.



Thus, the Observer interface from the standard java.util package has only one public method: update. This method is invoked each time when an Observable object (observed by that specific Observer object) sends an event in the system. Thus, the CircularShifter and the Alphabetizer class implement this method to take their action whenever the LineStorage module that they observe sends an event. The event itself is passed as the second argument in the update method.

3.3.2. Java Support for Observable

The second diagram shows the LineStorageWrapper class which reuses the LineStorage class from the second assignment, but extends its functionality by inheriting from the Observable class from the java.util package.



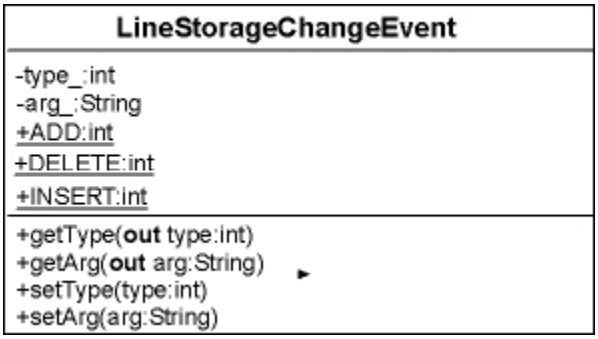
Thus, the Observable class from the standard java.util interface provides public methods that allow other objects to register themselves as Observers of objects of that class (addObserver() method). Further, the Observable class provides a method to notify Observers by sending an event in the system (notifyObservers() method). This method takes as an argument a single object of an Event class.

3.3.3. Current Implementation

In the current KWIC system we apply two LineStorageWrapper objects to store the original lines and circular shifts respectively. The added functionality from the Observable class allows us to invoke notifyObservers() method whenever a new line is added, deleted or inserted in any of these two objects. This in turn causes update method in the CircularShifter or Alphabetizer to be implicitly invoked, thus allowing us to take desired actions, i.e., to produce and add circular shifts in the case of CircularShifter object or to sort circular shifts alphabetically in the case of the Alphabetizer object.

3.3.4. Event Class

Finally, here is the diagram of the Event class from the existing KWIC system.



Note that the Event class implements a simple protocol for describing the type of the event that occurred in a LineStorageWrapper object. Thus, by setting the type of the event we may tell the Observers what happened exactly. For example, if we set type of the event to ADD, we are telling Observers that a new line has been added to a LineStorageWrapper object. On the other hand, setting the type of the event to DELETE, we are informing Observers that a line was deleted.

4. Your Assignment

4.1. Programming Assignment

You need to modify the existing system in the following way:

- **Implement an interactive version of the system**
- **Extend the functionality of the system**

4.2. Understanding Check

Finally, you need to answer to these three **questions**.

5. Implementing Interactive KWIC System

You need to implement an interactive version of the KWIC index system. That means the system won't read lines from a file but lines may be manipulated interactively by means of a simple command line user interface. Here is a transcript of a sample session:

- Add, Delete, Print, Quit: a
- > Star Wars
- Add, Delete, Print, Quit: a
- > The Empire Strikes Back
- Add, Delete, Print, Quit: a
- > The Return of the Jedi
- Add, Delete, Print, Quit: p
- ----------
- Back The Empire Strikes
- Empire Strikes Back The
- Jedi The Return of the
- Return of the Jedi The
- Star Wars
- Strikes Back The Empire
- The Empire Strikes Back
- The Return of the Jedi

- Wars Star
- of the Jedi The Return
- the Jedi The Return of
- ----------
- Add, Delete, Print, Quit: d
- > Star Wars
- Add, Delete, Print, Quit: p
- ----------
- Back The Empire Strikes
- Empire Strikes Back The
- Jedi The Return of the
- Return of the Jedi The
- Strikes Back The Empire
- The Empire Strikes Back
- The Return of the Jedi
- of the Jedi The Return
- the Jedi The Return of
- ----------

Thus, the command that the interactive version of KWIC system should be able to execute are:

- Add command for adding a new line with key binding 'a'
- Delete command for deleting a line with key binding 'd'
- Print command for printing shifts sorted alphabetically with key binding 'p'
- Quit command for exiting the system with key binding 'q'

5.1. Implementation Hint

The current implementation of the system and the LineStorageChangeEvent already supports sending/dispatching a delete event. Thus, when a line is deleted from the first LineStorage a LineStorageChangeEvent with the DELETE type is sent to the CircularShifter object. Now, to implement the delete functionality you need to handle the DELETE event inside the update method in CircularShifter class and to delete all circular shifts of the deleted line as well.

6. Extending the Functionality of the System

To extend the functionality of the existing KWIC system you need to implement words index. This index should keep all words that appear in the original lines and the number of their occurrences in the original lines. For example, the words index for the following original lines looks as follows:

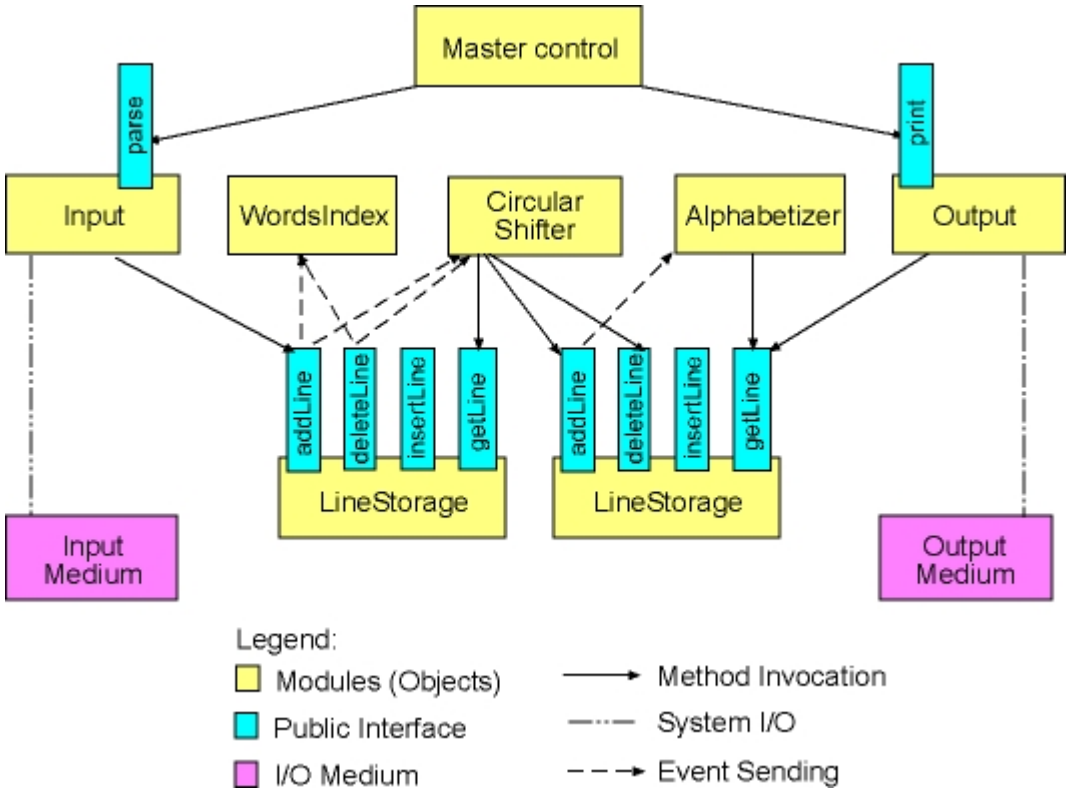
- Original Lines:
  - Star Wars
  - The Empire Strikes Back
  - The Return of the Jedi
  - The Phantom Menace
  - The Clone Wars
- Words Index:
  - Star: 1
  - Wars: 2
  - The: 4
  - Empire: 1
  - Strikes: 1
  - Back: 1
  - Return: 1
  - of: 1
  - the: 1
  - Jedi: 1
  - Phantom: 1
  - Menace: 1
  - Clone: 1

Further, you need to incorporate a new command in your command line user interface of KWIC system. This new command should allow users to print out the words index. Thus, your KWIC system need to support the following commands:

- Add command for adding a new line with key binding 'a'
- Delete command for deleting a line with key binding 'd'
- Print command for printing shifts sorted alphabetically with key binding 'p'
- Index command for printing words index with key binding 'i'
- Quit command for exiting the system with key binding 'q'

6.1. Implementation Hint

To implement the words index you may add a new module into the system, e.g. WordsIndex module. The new module should be declared as an additional Observer of the first LineStorage module. Thus, each event sent by the first LineStorage module will be sent to the WordsIndex module additionally to the CircularShifter module. In the update method of the WordsIndex you should then provide an implementation that will keep the WordsIndex up-to-date. Here is the architecture of the extended KWIC system:



7. Questions

Please, answer the following questions:

1. Which modules from the original KWIC system did you need to modify to implement the WordsIndex module? What can you conclude here? Do systems implemented with event based architecture allow for easy adding

of new modules, assuming that the new modules apply the already existing event protocol?

2. Please, describe the easiest way to implement a WordsIndex that should keep all words that appear in the original lines and the number of their occurrences in the circular shifts.
3. One of the main properties of event based systems is so-called "implicit" invocation of procedures. That means some of the procedures in the system are not invoked directly, but rather an event is sent into the system, and then the system itself after processing the events invokes these procedures. Now, taking into account that the processing of events and deciding which procedures to call might be quite complex in systems with a large number of modules and a complex event protocol what can you conclude about the performance of event based system? Do you think that these systems may achieve the same level of the performance as systems with direct procedure calls?