Table of Contents

QUICK GUIDE

Igor is a task manager that enables you to add tasks and manage them. You can perform operations such as add, delete, update, search and mark done by entering commands in the text box.

TO ADD AN EVENT

Command Format: add [event name] [time] [hash tag] [reminder]

Examples: add finish tutorial #high #ma3110

add project 5pm this Mon #high r- 5 mins

add career talk 5pm to 7pm tomorrow #enrichment #normal r- 15 mins

Quick Tips: Entering #high, #normal, or #low to classify an event as of high, normal or low priority.

TO DELETE AN EVENT

Command Format: delete [event index]

Examples: delete 2

TO UPDATE AN EVENT

- *Step 1: Choose event to update.*

Command Format: update [event index]

Example: update 1

- *Step 2: Event details will be displayed in the text box. Edit to your satisfaction.*

Command Format: update [event index] [new event content]

Example: update 1 finish tutorial #high #ma3110 12pm this Thursday r- 1 day

TO SEARCH FOR EVENTS

Command Format: search [search key words]

Examples: search project meeting #cs2103

Quick Tips: When you are in search view, press Ecs/Enter "back" into the text box to go back normal view

TO MARK DONE AN EVENTS

Command Format: done [event index]

Examples: done 2

TO MARK UNDONE AN EVENTS

Command Format: undone [event index]

Examples: undone 2

TO UNDO AN ACTION

Command Format: undo

USEFUL HOTKEYS

Command Format: ALT+M

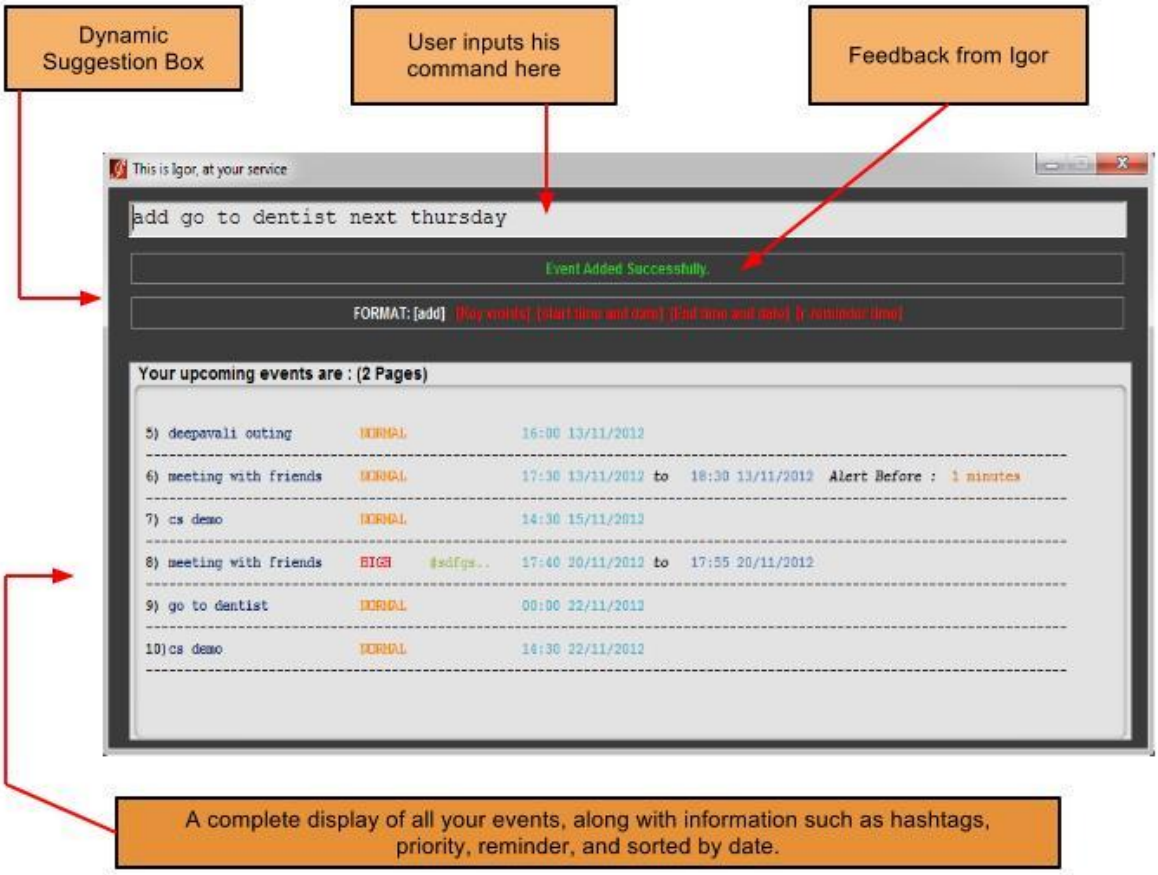
What It Does: - Minimizes the application to the system tray if application is running

- Maximizes the application if it is in the system tray

Command Format: ALT+C

What It Does: - Closes the application, irrespective of whether it is running in the system tray or in the front

2.1 Graphical User Interface



2.2 Event

There are 3 types of events supported by Igor:
- Floating event: events with no time. A Floating event can be considered as a to-do task.
- Deadline event: events with only one time.
- Timed event: events with start and end time.

Each event can have the following properties: Event Name Priority: HIGH/NORMAL/LOW Start time, End time, and Reminder.

2.3 Add an event

Command Format:	add [event name] [time] [hash tag] [reminder]

Example: *To add a Floating Event*
add floating #high #impt

Example: *To add a Deadline Event*
add deadline event #high #work 5pm today r- 30 mins

Example: *To add a Timed Event*
add timed event #normal #cs2013 #project 5pm to 7pm tmr r-15 min

Points to note:

- Acceptable Command words for Add: 'add', '+'
- The order of the event fields can be mixed up. However, the event name must be the first field.
- For example: add 5pm today meeting #high is unacceptable.
- To add a reminder for an event, start with 'r-', and specify the reminder period. For example: r- 15 minutes to add a reminder 15 minutes before the event time.
- The user can specify the priority of the event in the hash tag field. For example: "add project meeting #high #cs2103" will add an event named "project meeting" with a high priority.
- If priority is not specified in the hash tag field, the event will be defaulted to normal priority. Enter "#high"/"#h" to indicate HIGH priority, "#low"/"#l" to indicate LOW priority, "#normal"/"#n" to indicate NORMAL priority.
- Refer to section for acceptable time field.

Known Issue:

- If there are multiple priority entered, only the first priority will be taken as the event priority.
- If the entered start time is after the entered end time, Igor will automatically swap the two time fields.
- A floating event cannot have reminder. If the user adds a reminder time for floating event, the reminder will be ignored.

2.4 DELETE AN EVENT

Command Format: delete [event index]

Examples: *To delete the event with index 2 on the display list*

delete 2

2.5 Update An Event

- *Step 1: Choose event to update.*

Command Format: update [event index]

Example: update 1

- *Step 2: Event details will be displayed in the text box. Edit to your satisfaction, then press Enter.*

Command Format: update [event index] [new event content]

Example: update 1 finish tutorial #high #ma3110 12pm this Thursday r- 1 day

2.6 Search

Command Format: search [search key words]

Examples: search project meeting #cs2103

Quick Tips: When you are in search view, press Ecs/Enter "back" into the text box to go back normal view

2.7 Supported Time Format

Time separator: '.' or ':'

HH denotes hour, mm denotes minute, aa denote am or pm

12-hour time format:

HH + aa

HH + time separator + mm + aa.

24-hour time format: HH + time separator + mm

Example:

Acceptable: '7 pm', '7.00 am', '7.00 am', '07.00 pm', '7:00',

Unacceptable: '17:00pm', '700', '17 pm', '7:90am'.

2.8 Supported Date Format

dd denotes date, MM denotes month in integer, MMM denotes month in characters (full or short forms), yy denotes year in short/full

date separator: '/' or '-'

dd + date separator + MM

dd + date separator + MMM

dd + date separator + MM + date separator + yy

dd + date separator + MMM + date separator + yy

Note:

- The short form of month is the first 3 characters of the full form. For example, the short form of September is Sep.
- For date with month in MMM format, date separator can be ignored.
- For date with no year specified, the current year will be taken as year.

2.9 Supported Reminder Format

General format: r- + reminder period + time unit

Time unit can be minute, hour or day.

Forms of minute: minute(s)/min(s)

Forms of hour: hour(s)/h

Forms of day: day(s)/d

If time unit is minute, the reminder period can have value from 0 to 59. If time unit is hour, the reminder period can have value from 0 to 24.

Example:

Acceptable: r- 5 min, r- 5 mins, r-15minutes

Unacceptable: r-90min, r- 30 h, r-15 minutes

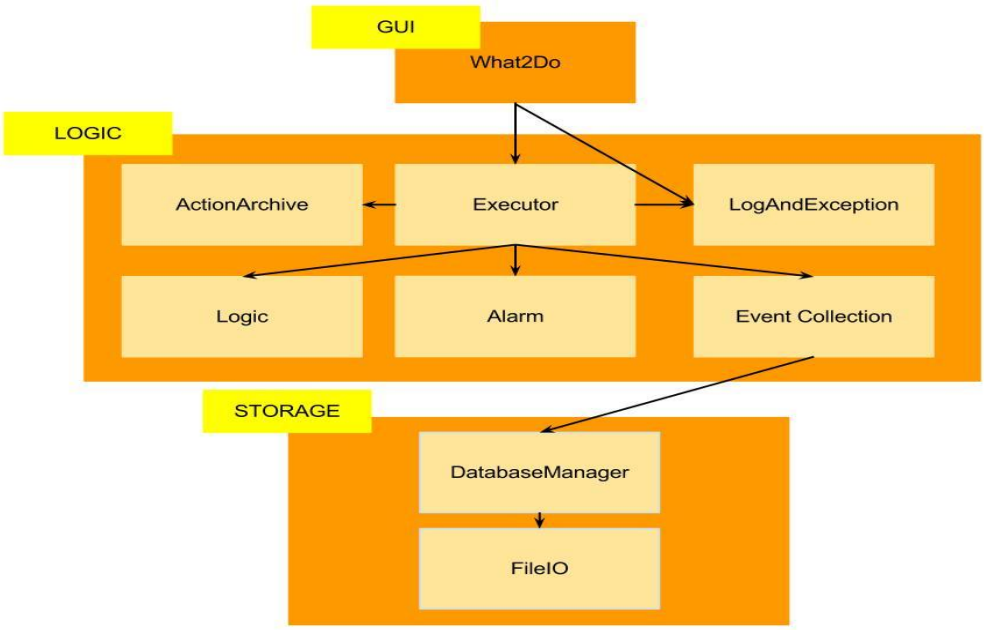
DEVELOPER'S GUIDE

Table of Content

1. INTRODUCTION

1.1. Component Overview

The internal design of What2Do utilizes the N-tier architecture. Specifically, the top tier is GUI, and below it is Logic, and Storage is the lowest layer. The figure below shows the main components of the internal architecture.



2. Graphical User Interface (GUI)

2.1 IgorUI

The user interacts with the program using the **IgorUI** class. The IgorUI class sets up the visual elements of the GUI, listens for user commands and displays the according reaction of the system.

The above diagram displays the important components of the JFrame, which display different kinds of information to the user.

The **IgorUI** is the controller class and contains the following important methods:

Methods	Description
initComponents()	Initializes the various JFrame components that form the Graphical User Interface, sets their dimensions and their layouts.
textField1KeyTyped(java.awt.event.KeyEventevt)	Listens for keys being typed by the user in the text box provided for user inputs and alerts the relevant functions in the GUI.
textField1KeyPressed(java.awt.event.KeyEventevt)	Listens for UP and DOWN arrow keys to access past inputs. Also listens for CTRL+U and CTRL+F, which shift the display to upcoming and floating events respectively.
actUponUserCommand(java.awt.event.KeyEventevt)	Checks the user input, and calls the relevant function in the Executor, which relays what has to be done in the display.
setSuggestionBox(java.awt.event.KeyEventevt)	Observes the input typed by the user, key-by key and dynamically changes the suggestions displayed.
displayDatabase(String message)	Obtains from the "DataToUser" class all the information that needs to be displayed to the user, in fully formatted style.

Points to note:

- The IgorUI class listens for user inputs, analyses it using the Executor class and then generates suitable response messages that are shown to the user. It then utilizes the DataToUser class to display on the screen all the database information with complete HTML formatting.

2.1 DataToUser

The DataToUser class obtains the database in the raw format from the Executor. It then formats the information according to the required final display format, converts to HTML code, and sends the information to the IgorGUI class, which simply displays the information to the user.

Methods	Description
getResults()	The controller method in the DataToUser class, it is called by the IgorUI class.
obtainDataFromDatabase()	Obtains stored data from the Executor class.
fillInDatabases()	Fills in the formatted databases with a combination of the raw information and the HTML formatting.

2.2SystemTrayIntegration

The SystemTrayIntegration class creates a tray icon for the program which runs in the system tray, so long as the program is not terminated. It also uses the JIntellitype library to set up global shortcut keys, to maximize and minimize the program to the taskbar and to exit the program.

Methods	Description
createSystemTray()	Creates tray icon and adds it to the system tray, sets up global shortcut keys and creates a pop-up menu for the tray icon.

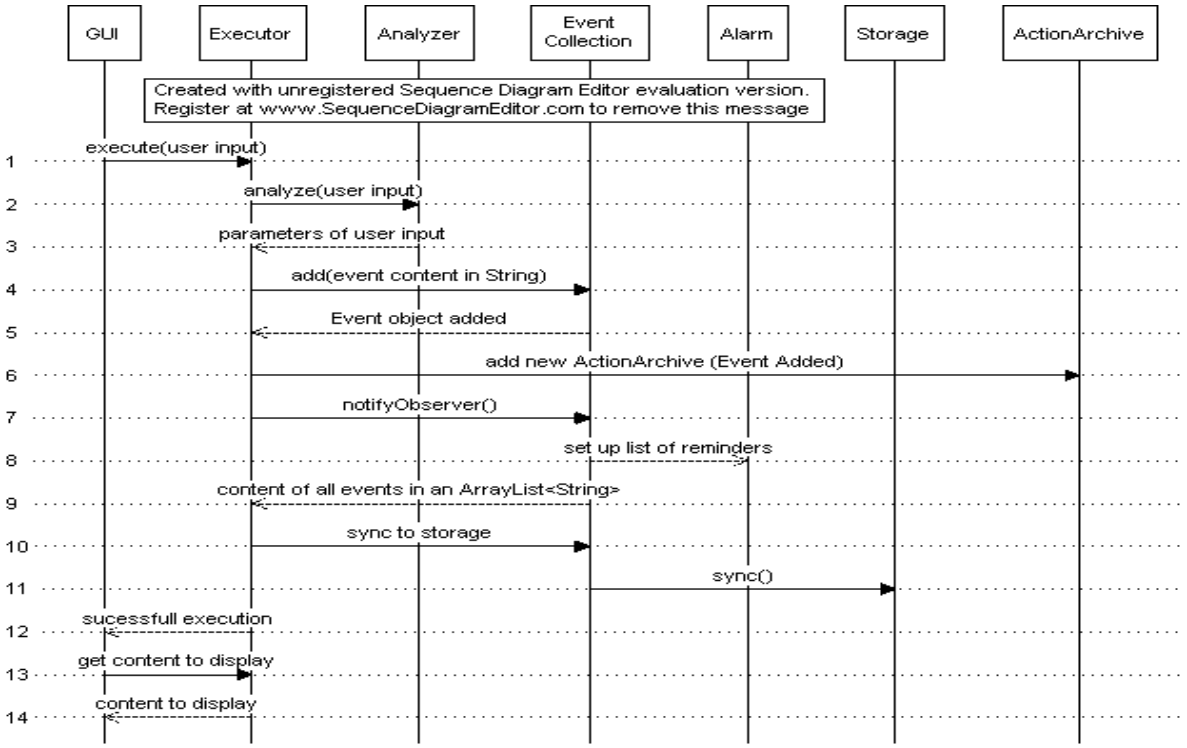
3. LOGIC

3.1. Logic Overview

Logic is the main part of the program. It receives user command from GUI, analyze the command and execute accordingly. From figure 1, LOGIC contains 6 main components, and the role of each component is as follow:

Component	Description
Executor	Executor is the façade class of LOGIC, GUI will pass an input to Executor to perform an execution and pull data from Executor to display to the user.
Analyzer	Split the input into parameters so that it can be process by the Executor.
Event Collection	Store all the events during run-time. Executor will call add, delete, update, search and mark done methods from this component to perform an execution.
Alarm	To store all reminders time. This component runs a thread that will trigger alarm.
Action Archive	To store a history of user command. Executor will call the undo function of this component when the user wants to undo his command.
LogAndException	This component contains a class for Logging, and a class called MessageHandler to store all the possible message that could be shown by GUI.

The sequence diagram below shows a typical flow inside logic. In this case, the user is adding an event.



3.2. Executor

The two main classes of this package are Executor and Command. Command has child classes depending on what command the user has entered. Based on the user command, a Command object is created, which performs appropriate actions based on the user input, and state of Executor. Executor is not aware of the inner workings of the Command object. Command Pattern , thus has been implemented to an extent.

3.2.1. Executor

Executor is a singleton class. It is the class which actually decides on what action to perform based on the user's input. It has the following data members:

Data members	Description
`currentListOfUpcomingEventToDisplay`	A String ArrayList of the non–floating events to be displayed to the user via the GUI. The events are ordered according to their starting time. Events with the same time are ordered according to priority. It only contains those events whose end times are after the current system time.
`currentListOfFloatingEventToDisplay`	A String ArrayList of the floating events to be displayed to the user via the GUI. The events are ordered according to their priority as they have no starting or ending time. It only contains events that are not marked as "done".
`searchResults`	String ArrayList that contains search results. All search results (whether done or undone) are stored here. The events are sorted in order here as well.
`searchState`	A boolean variable used as a flag to check if the user is viewing events in Normal view (false) or as Search view (true). The action to be performed depends on this flag.
`feedback`	A String ArrayList that contains messages in case a user adds or updates an event with time before current time, or the time entered clashes with another event. The messages are displayed by the GUI. It is obtained from the Command object created.

The following are some of the important methods in Executor.

Return Type	Methods
`void`	`analyze(String[] content)` Performs the required action like Add, Delete, etc. based on user input. It checks for the command entered, creates a new Command type object based on that and executes the required action. It also stores the feedback, if any. It returns an integer value which denotes which message the GUI is to show to the user.
`void`	`searchToTrue()` Sets the searchState flag as true when user enters search mode.
`void`	`searchToFalse()` Sets the searchState flag as true when user exits search mode.
`void`	`updateListOfEvent()` Updates the data members storing events to be displayed via GUI. It is called after every operation.
`ArrayList<String>`	`printFloatingDataBase()` Returns *currentListOfFloatingEventToDisplay*. Called by the GUI.
`ArrayList<String>`	`printFloatingDataBase()` Returns *currentListOfUpcomingEventToDisplay*. Called by the GUI.
`ArrayList<String>`	`printSearchResults()` Returns *searchResults*. Called by the GUI.
`void`	`loadDatabase()` Loads from the Text file Database at the beginning of the program.

Example:*To add an Event, based on user command, the following is done in analyze*

```
Command action = new CommandAdd(userInput,searchState,size of search results,
                 size of displayed events);
action.execute();
feedback = action.getFeedback();
return action.getMessageValue();
```

Points to note:

- The integer values returned correspond to messages displayed by the GUI on successful or unsuccessful completion of an action.

3.2.2. Command

The Command class is the one which actually performs the actions the user requires. It has various child classes which have the same overridden method which is called by Executor to perform the appropriate action.

Data Fields	Description
`userInputString`	String entered by user in the GUI.
`userInputInteger`	Integer the user has entered as an index. 1 is subtracted from it to correspond to the actual index value. Stored as -2 if the user enters a non – integer.
`searchSize`	Size of searchResults. It is required for error handling if user enters an invalid index value when in Search mode.
`shownEventSize`	Number of Events (Floating and Non-floating) that are being displayed to the user . It is required for error handling if user enters an invalid index value when in Normal mode.
`searchState`	A Boolean variable used as a flag to check if the user is viewing events in Normal view (false) or as Search view (true). The action to be performed depends on this flag. It mirrors the flag in Executor.
`feedback`	A String ArrayList that contains messages in case a user adds or updates an event with time before current time , or the time entered clashes with another event. This is required by Executor'sfeedback .
`parameterList`	An array of String parameters created after splitting and parsing through the userInputString. Used for the execute() method .
`returnVal`	Integer value denoting what message to display to the user via GUI.Returned to Executor.

The following are some of the important methods in Executor.

Return Type	Methods
`Constructor`	`Command(String userInput, boolean searchState, int searchSize, int shownEventSize)` Constructor to initialize all the data members of the class. It also sets up the classes in the Logic package to parse through the user input .This constructor are also called by child classes.
`void`	`execute()` Method which actually performs the required actions as required by the user. It also writes messages to the Log. It decides the value of returnVal based on successful completion of action or encounter of an error. This method is **overridden** in the child classes (As each child class performs a different task)
`String`	`getFeedback()` Returns the Feedback obtained from ListOfEvent. Called by Executor
`int`	`getMessageValue()` Returns the value of returnVal. Called by Executor
`int`	`splitInput()` A **protected** method which splits userInputString using LogicSplitter class. Sets the values of parameterList elements. Returns -1 if no error encountered while

	splitting, otherwise returns an integer corresponding to the error.
void	`setLogAndMessage(int logType, int messageVal)` A **protected** method inherited by all child classes, it writes a message to the log and sets the value of returnVal.

Example: *The overridden execute() method in Command Undo*

```
intsplitError = this.splitInput();
if (splitError != -1) {
        setLogAndMessage(LOG_ERROR, splitError);
        return;
}
String undoMessage = ListOfActionArchive.undo();
if(undoMessage=="") {
        setLogAndMessage(LOG_ERROR, ERROR_UNDO);
        return;
}
ListOfEvent.notifyObservers();
try {
        // Save to file before each operation
        ListOfEvent.syncDataToDatabase();
} catch (Exception e) {
        ListOfEvent.formatListOfEvent();
        setLogAndMessage(LOG_ERROR, ERROR_DATABASE);
        return;
}
setLogAndMessage(LOG_MESSAGE, SUCCESSFUL_UNDO);
                return;
```

Points to note:

- ListOfEvent.notifyObservers() is called after every action at the end of execute() to get the updated list of events to display to the user.
- ListOfEvent.syncToDatabase() is also called at the end of execute() to save to the file after each user operation (except Search).
- Feedback is an empty ArrayList when there is no clash or “past event” issue.
- The message written to the log and displayed to user via the GUI is the same and depends on the value of returnVal.

3.3. Logic

Logic has two main parts – Logic Splitter and Logic Analyzer. Both are static in nature. These two classes together are responsible for analyzing and returning the user input in a particular format that can be stored in the database. It also has two additional supporting classes – PatternLib and PatternDateTime – for parsing various time formats.

3.3.1. Logic Splitter

Logic Splitter is a static class that is called by Executor, every time there is a user input that needs to be analyzed. It detects the type of command by taking in the first word and then splits the remainder of the string into various fields.

It is important to note that each type of command has its own splitter. For example – Add command will lead to the input being split into the following fields – COMMAND NAME, KEYWORDS, HASHTAGS, START-TIME, END-TIME and REMINDER TIME. On the other hand, a Delete Command is only split into 2 fields – COMMAND NAME and DELETE INDEX. Update Command also has an additional field called the Update Index - which is used for informing the executor for which event in the ListOfEvent is to be updated.

It contains protected methods which can be accessed by all the child classes.

Return Type	Methods
`void`	`setup()` To setup the various class parameters before splitting a user input.
`String`	`splitInput(String userInput)` Parses the type of command and accordingly calls a child class with the relevant splitter. For example – if the command was ADD, the function would call LogicSplitterAdd.

Example: *To Split a given input into an String Array -*

```
String userInput = "add project #high #cs2103 5pm to 6pm tmr r-1min";
Vector<String>parameterList = LogicSplitter.splitInput(userInput);
    // parameterList = {add, project, #high#cs2103, 2012-11-13T17:00+08:00,
                        2012-11-13T18:00+08:00, r-1min}
```

3.3.2. Logic Analyzer

LogicAnalyzer is also a static class. It is called by Executor when it needs to convert the splitted input into a system-processable format. The main function here is getAddUpdateEventString(String[]) which parses through the String array (which has been split by LogicSplitter into the relevant fields like – KEYWORDS, HASH-TAGS etc.) and returns a string with all event attributes separated by a ".." splitter.

Return Type	Methods
`void`	`setup()` To setup the various class parameters before analyzing a user input.
`String`	`getAddUpdateEventString(String[])` It analyzes all the elements of the string array, converts them into a specific format and then appends them in a particular order - (KEYWORDS..PRIORITY..HASHTAGS..”FALSE”..REMINDERTIME..STARTTIME ..ENDTIME..COMPLETEDTIME).
`int`	`getInteger(String)` It extracts the INTEGER from the user input and returns it. Returns -1 – if there was no integer.
`String`	`getHashTags(String[])` Searches throughout the String array and returns a list of hash tags.

Example: *To Analyze a add/update command String[] and get an event String -*

```
String[] parameterList = {“add”, “abcd #high”, “5pm tmr”, “r-1min”};
String event=LogicAnalyzer.getAddUpdateEventString(parameterList);
This will return a string – “abcd..HIGH….FALSE..16:59 13/11/2012..17:00
13/11//2012..INVALID..INVALID”.
```

Points to note:

- Keywords cannot be empty. RETURNS ERROR IF IT IS.
- Exchanges end time and start time if the end time specified is before start time.
- There can be no time fields in the keywords.
- Single Time field is taken as start time.

3.3.3. PatternDateTime

PatternDateTime is used to find/match a certain String to a date/time/reminder pattern. Each PatternDateTime object has two private fields – one is a Pattern object which compliled a regular expression of a date/time format, and the other is a String contains the time format that will be used to parse the input String to a DateTime object.

Example: To create a new PatternDateTime object that supports a certain date/time pattern

```
Pattern pat = Pattern.compile(“HH:mm dd/mm/yyyy”);
PatternDateTime patDateTime = new PatternDateTime(pat, “HH:mm dd/mm/yyyy);
```

Example: To check whether a String contains a date/time field.

```
String input = “project 10:30 20/10/2012”;
int[] subStringIndex = patDateTime.isFind(intput);
String detectedDateTime = input.subString(subStringIndex[0],
        subStringIndex[1]); // detectedDateTime = “10:30 20/10/2012”
```

Example (cont'd from previous): To parse a String to a DateTime object

```
DateTime detectedTime = patDateTime.getDateTime(detectedDateTime);
```

3.3.4. PatternLib

This is a singleton class which stores a list of PatternDateTime objects and is used by LogicSplitter to detect date/time subtring of a String, and by LogicAnalyzer to parse the substring to a DateTime object.

Return Type	Methods
PatternLib	`getInstance()` Returns an instance of a PatternLib object.
int[]	`getDateTime(String input, int index)` Returns a DateTime Object by parsing the String input which had a pattern that matched the Time pattern at "index" in the list of Time patterns.
int	`isMatchDateTime(String)` Returns the index (in the List of Time Patterns) of the Time pattern which matches the given String. Returns -1 if not found.
int[]	`isFindDateTime(String)` Returns the Index of the Time pattern (from the List of Time Patterns) which matches the given string along with the start and end Index of the portion of the given String which matches the pattern. Returns an Integer Array with these three things. Returns {-1,-1,-1} if not found.
int[]	`isFindReminderTime(String)` Returns the Index of the Reminder pattern (from the List of reminder patterns) which matches the given string along with the start and end Index of the portion of the given String which matches the pattern. Returns an Integer Array with these three things. Returns {-1,-1,-1} if not found.

Example: *To detect a date/time field within a String*

```
String input = "project meeting 7pm tmr";
PatternLib patLib = PatternLib.getInstance();
int[] isMatch = patLib.isFindDateTime(input);
int patternIndex = isMatch[0];
int startStringIndex = isMatch[1];
int endStringIndex = isMatch[2];
String detectedTime = "";
if(patternIndex >= 0) {
        detectedTime = input.substring(startStringIndex, endStringIndex);
}
```

Example (cont'd from previous): *To parse a detected substring to a DateTime object*

```
DateTime date = patLib.getDateTime(patternIndex, detectedTime);
```

3.4. Event

Two main classes in this component are Event and ListOfEvent. ListOfEvent is a composition of many Event objects. ListOfEventObserver is an observer interface of ListOfEvent class.

3.4.1. Event

Each Event object will have 9 attributes – String ID, String name, PRIORITY_TYPE priority, String hashTag, DateTime start, DateTime end, DateTime reminder, DateTime timeCompleted. PRIORITY_TYPE is an enum that only allows HIGH, NORMAL, AND LOW priority values. Below is a description of important methods in Event class.

Return Type	Methods
`Void`	`parse(String[] content)` Change the value of the attributes according to the fields passed in as the parameter.
`String`	`toStringToDatabase()` Compose the content of the event into a single line of String to write to the database text file.
`String`	`toStringToDisplay()` Compose the content of the event into a single line of String to pass to the GUI. The GUI will format the content and display them to the user.
`Boolean`	`isBefore(Event anotherEvent)` Check if the Event is before another Event. This is called by the Comparator to sort the list of Event in ListOfEvent class.
`Int`	`getEventType()` Return 0 if the Event object is a floating event, 1 if it is a deadline event and 2 if it is a timed event.
`boolean`	`isClashedWith(Event anotherEvent)` To check whether the Event is clashed with another event.

Example: *To parse a String array to an Event*

```
String[] newEventInString = {"id", "name", "priority", "hashTag", "isDone",
                  "reminder", "start time", "end time", "completed time"};
Event newEvent = new Event();
```

```
newEvent.parse(newEventInString);
```

Points to note:

- The priority field in the String array above can only be either “high, “low”, or “normal”.
- The isDone field only takes “true” or “false”.
- Hash Tag field is a sequence of hash tag, each only has one word. For example: “#hash #tag” is valid, but “#hash tag #invalid” is invalid.
- Reminder, start time, end time and completed time field must be of the following time format: “HH:mm dd/MM/yyyy”.
- Invalid array of String passed in will not change the content of the event.

3.4.2. ListOfEvent

ListOfEvent is a static class. There are 3 main points to take note of this class:

1. ListOfEvent stores an ArrayList of Event objects. It contains methods to perform add, delete, update, search, and mark done on the list of Event objects. Note that there are different overloading methods with the same functionality. These are used in different scenarios by different classes. Refer to the examples below for two overloading methods for add.
2. ListOfEvent will call the load method in DatabaseManager at the start of the program to load database from the storage, and it will call sync method also from DatabaseManager to sync to the storage. Refer to section 4 on STORAGE for more information.
3. ListOfEvent is an Observable class of Observer classes that implemented ListOfEventObserver interface. Two observers include Executor and ListOfAlarm.

Example: *Executor add a new Event.*

```
String eventToAdd = “01..new event..high..#cs2103 #school..false..17:45” +
                    “07/07/2007..18:00 07/07/2007..invalid..invalid”;
ListOfEvent.add(eventToAdd);
```

Points to note:

- eventToAdd will be splitted by “..” and the resulting String array will be passed to parse(String[] args) of a new event object to change its fields.
- End time and completed time fields are “invalid” to denote no time.

Example: *ActionArchiveDelete wants to add back a deleted event*

```
ListOfEvent.add(_deletedEvent);
```

Points to note:

- _deletedEvent is an Event object. It is a field of an ActionArchiveDelete class. Refer to section 3.5. on ActionArchive for more information.

Example: *Add an Observer of ListOfEvent*

```
ListOfEvent.addObserver(ListOfEventObserver object);
```

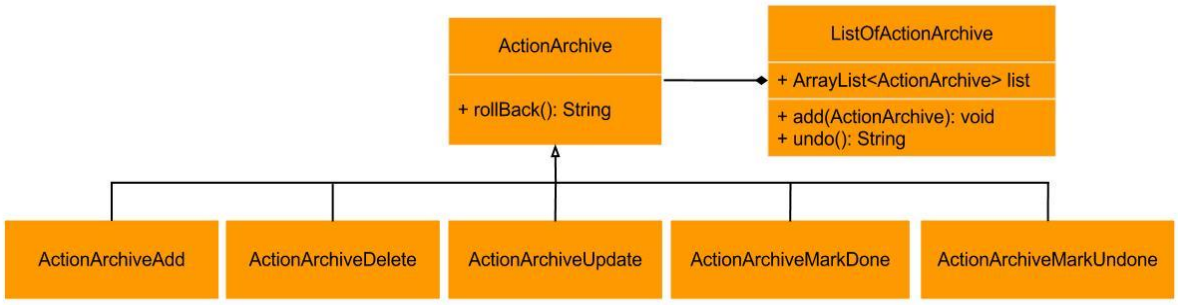
Example: *Prompt ListOfEvent to update its observers*

```
ListOfEvent.notifyObservers();
```

Points to note:

- Adding of Observer is done at the start of the program.
- Notifying Observers is done after the execution of every command.

3.5. ActionArchive



ActionArchive is a component to store the history of user command for undo purpose. Each ActionArchive object stores the Event object(s) on which changes was made in a single command. There are 5 children classes inherited from the ActionArchive class. These are to store changes made by 5 different types of undo-able user command.

If the user adds an event, the Event object that has just been added will be stored in a new ActionArchiveAdd object. The table below shows the field and methods in the ActionArchiveAdd class.

Class Field	Description
Event _addEvent	Each ActionArchiveAdd object will store the event that was added to ListOfEvent.
Return type	**Methods**
Constructor	ActionArchiveAdd(Event newAddedEvent)
String	rollBack() Roll back the previous action on ListOfEvent. This will delete the added Event from ListOfEvent. This method returns an undo message to Executor.

ListOfActionArchive class is a static class that stores a list of ActionArchive objects. After the execution of each undo-able command, Executor class will create and add a new ActionArchive object to ListOfActionArchive. When the user wants to undo his command, Executor will call the

undo() method in ListOfActionArchive, which in turn will call the *rollBack()* method in the last ActionArchive object in the list to undo the previous command.

Example: *Add a new ActionArchiveAdd*

```
Event addedEvent = ListOfEvent.add(newEvent);
ActionArchive newAddAction = new ActionArchiveAdd(addedEvent);
ListOfActionArchive.add(newAddAction);
```

Example: *Undo the previous command*

```
String undoMessage = ListOfActionArchive.undo();
```

Points to note:

- undoMessage is a formatted message to be sent back to GUI to display in the execution status box.

3.6. Alarm

The Alarm package has classes AlarmSound, AlarmThread, AlarmType, ListOfAlarm, AlarmRun, Dialogs and DialogRun.

3.6.1. AlarmSound

AlarmSound is a class which was created using an online software called SoundToClass. It basically converts a sound file to a class file (storing the sound as bytes) with additional methods. The following are some of the methods in AlarmSound.

Methods	Description
`void`	`play()` Plays the sound for its duration
`void`	`loop()` Loops the sound
`void`	`stop()` Stops the sound

Points to note:

- The class Dialogs has an object of this class

3.6.2. Dialogs

This class creates the settings for the Dialog box to display . It also has an AlarmSound object called sound.

Return Type	Methods
`Constructor`	`Dialogs(JFrameparent,String message)` Constructor to set up Dialog box to display the message.
`void`	`startAlarm()`

	Calls sound.play()

Points to note:

- The sound starts when DialogRun sets it to visible and stops either after 3 seconds or when the user clicks the “OK” button.

3.6.3. DialogRun

This class implements the **Runnable Interface** and thus can be run as a different thread. It has 2 data members

Data Fields	Description
`event`	String containing the name of the event to display
`reminder`	An object of the class Dialogs

Return Type	Methods
`Constructor`	`DialogRun(String ev)` Constructor to initialise the Dialogs reminder
`void`	`run(JDialog applet)` Setting the Dialogs reminder to true
`void`	`run()` Overridden public method which is called when the Object is run as a thread.

Points to note:

- The class Dialogs has an object of this class
- This class makes a Dialog Box run as a different thread which only closes when the user presses "OK"

3.6.4. AlarmType

This class implements has 2 data members,

Data Fields	Description
`eventName`	String containing the name of the event to display
`reminderTime`	A DateTime object storing the reminder time

Return Type	Methods
`Constructor`	`AlarmType(String Event, DateTime reminder)` Constructor to initialise the object.
`boolean`	`isAlarmTime()` Returns true if current System time matches reminderTime
`String`	`getEventName()` Returns eventName

Points to note:

- The class Dialogs has an object of this class
- This class makes a Dialog Box run as a different thread which only closes when the user presses "OK"

3.6.5. ListOfAlarm

This class is a Singleton class which implements ListOfEventObserver. Thus this class is updated whenever ListOfEvent.notifyObservers() is called (check Executor). It has the following data members:

Data Fields	Description
`alarmList`	A Collection of AlarmType objects, setup from ListOfEvent
`instance`	ListOfAlarm instance to implement singleton pattern

Return Type	Methods
`void`	`setListOfAlarm(ArrayList<AlarmType>newList)` Copies the elements of newList to alarmList
`void`	`updateListOfEvent()` Updates alarmList after ListOfEvent.notifyObservers is called
`void`	`runAlarm()` Continuously check if the elements in alarmList have a reminderTime that matches the current time to the nearest second. If it does, a DialogRun object is created which displays a box as another thread

3.6.6. AlarmThread

This class implements Runnable, and is run as a thread along with the main thread. This is a singleton class.

Return Type	Methods
`void`	run() Overridden public method which starts when the class is run as a thread. It continuously calls ListOfAlarm.runAlarm()

Points to note:

- The class is created in the main class when the software first starts

3.7. Log and Exception

This component comprises of two separate classes – Log and Message Handler.

3.7.1. MessageHandler

Message Handler is to store an ArrayList of all possible messages that will be display by the GUI. This class is used by 2 components - Executor and GUI, in two scenarios:

- After each execution, the Executor will return an integer value to the GUI, indicating the index of the message in the MessageHandler class. GUI will pull the message accordingly from MessageHandler, and display the message to the user.
- The Executor pulls the message from MessageHandler as a Log message to write to the Log file.

Return Type	Methods
void	setUp() Set up the list of message at the start of the program.
String	getMessage(int index) Return the message at the index in the list of message.
void	setMessage(int index, String formattedMessage) Set the message at index as the formattedMessage. This is used only in one scenario when Executor set the undo message to return to the GUI.

3.7.2. Log

The purpose of Log class is for Logging. After the execution of each command, the Command object will compose a Log and write to the Log file. The written Log comprise of two pieces of information: Execution status, and the user command that has just been executed. For each time the program is run, there are two separate Log files, one is ErrorLog (when error occurs or exception was caught), and the other is MessageLog (when things go as planned).

Return Type	Methods
void	setUp() Set up at the start of the program.
String	toLog(int level, String message) Write the message together with the level of severity to the Log file. level = 1 indicates LEVEL_WARNING. level = 0 indicates LEVEL_INFO. Other values indicate LEVEL_SEVERE.

3.8. Global

This component includes a static class – Clock, which is used by other components of the program. The rationale behind this is to contain methods that are used frequently during the program to perform operations on DateTime. This class is used by LogicSplitter, Logic Analyzer and Event to convert between String and DateTime, as String operations on DateTime objects is required to write data to database, and to display date and time to the user in GUI.

Below are the methods in Clock class and some examples of their uses.

Return Type	Methods
DateTime	getBigBangTime() Return an instance of DateTime object, which is the time at 00:00 01/01/1970. The program uses this time as an indication of null DateTime fields in Event class.
boolean	isBigBangTime(DateTime time)

	To check whether time is a Big Bang time.
String	toString(DateTime time) Return a DateTime object in a String with format HH:mm dd/MM/yyyy.
DateTime	parseTimeFromString(String timeInString) Return a DateTime object parsed from the input String of format HH:mm dd/MM/yyyy.
DateTime	changeToDate(DateTime from, DateTime to) To change the date of a DateTime object to the same date of another DateTime object. The resulting DateTime object remains the same hour and minutes as the initial object.

Example: *Parse from a String to a DateTime objects*

```
String dateInString = "10:30 10/10/2012";
DateTime result1 = Clock.parseTimeFromString(dateInString);
dateInString = "invalid";
DateTime result2 = Clock.parseTimeFromString(dateInString);
dateInString = "10.30 10-10-2000";
DateTime result3 = Clock.parseTimeFromString(dateInString);
```

Points to note:

- Case 1: result1 = 10:30 10/10/2012.
- Case 2: "invalid" is the String representation of Big Bang time. result2 = 00:00 01/01/1970.
- Case 3: Wrong DateTime format. result3 = 01:00 01/01/1970 (Note: Different from Big Bang time!!)

Example: *Get a String representation of a DateTime object*

```
DateTime date = new DateTime(2012, 10, 10, 10, 30);
String dateString = Clock.toString(date); // dateString = "10:30 10/10/2012"
date = new DateTime(1970, 1, 1, 0, 0);
dateString = Clock.toString(date); // dateString = "invalid"
```

4. STORAGE

4.1. FileIO

FileIO is a class to read and write data to the database (i.e. a text file named "IgorDatabase.txt". ListOfEvent is in charge of creating a FileIO object and write its own content to the database/read data from the database.

Another role of FileIO is to read from test files, and write to output files during system testing. Refer to section 5 on TESTING for more information.

Below are examples of how FileIO is used by ListOfEvent.

Example: *ListOfEvent set up data from database*

```
String databaseFileName = "IgorDatabase.txt";
FileIO database = new FileIO(databaseFileName);
database.setUpDatabase();
```

Example: *ListOfEvent sync data to database*

```
String databaseFileName = "IgorDatabase.txt";
FileIO database = new FileIO(databaseFileName);
ArrayList<String> currentListOfEventInString =
                        ListOfEvent.getContentToSyncToDatabase();
database.syncToDatabase(currentListOfEventInString);
```

5. TESTING

Testing of Igor is done in two fashions: Testing of individual components and integration testing of the whole system. All test files are contained in the Test package.

Junit test is already done on the following classes: Event, ListOfEvent, PatternDateTime, LogicSplitter, LogicAnalyzer, and Clock.

SystemTest is a class under Test package, it will read test commands from a text file name "SystemTest.txt" and call relevant methods in Executor to process the commands. At the end of each execution, it pull the list of floating, upcoming and search results from Executor and respectively write to 3 different text files – "ExpectedFloating.txt", "ExpectedUpcoming.txt", "ExpectedSearch.txt". A file comparision between these actual output and the expected output from the files "ExpectedFloating.txt" and "ExpectedUpcoming.txt" and"ExpectedSearch.txt" will determine whether the system works as expected.