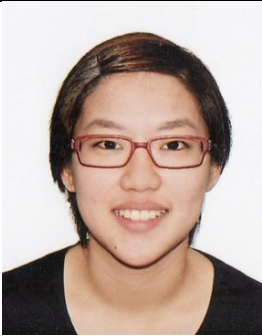
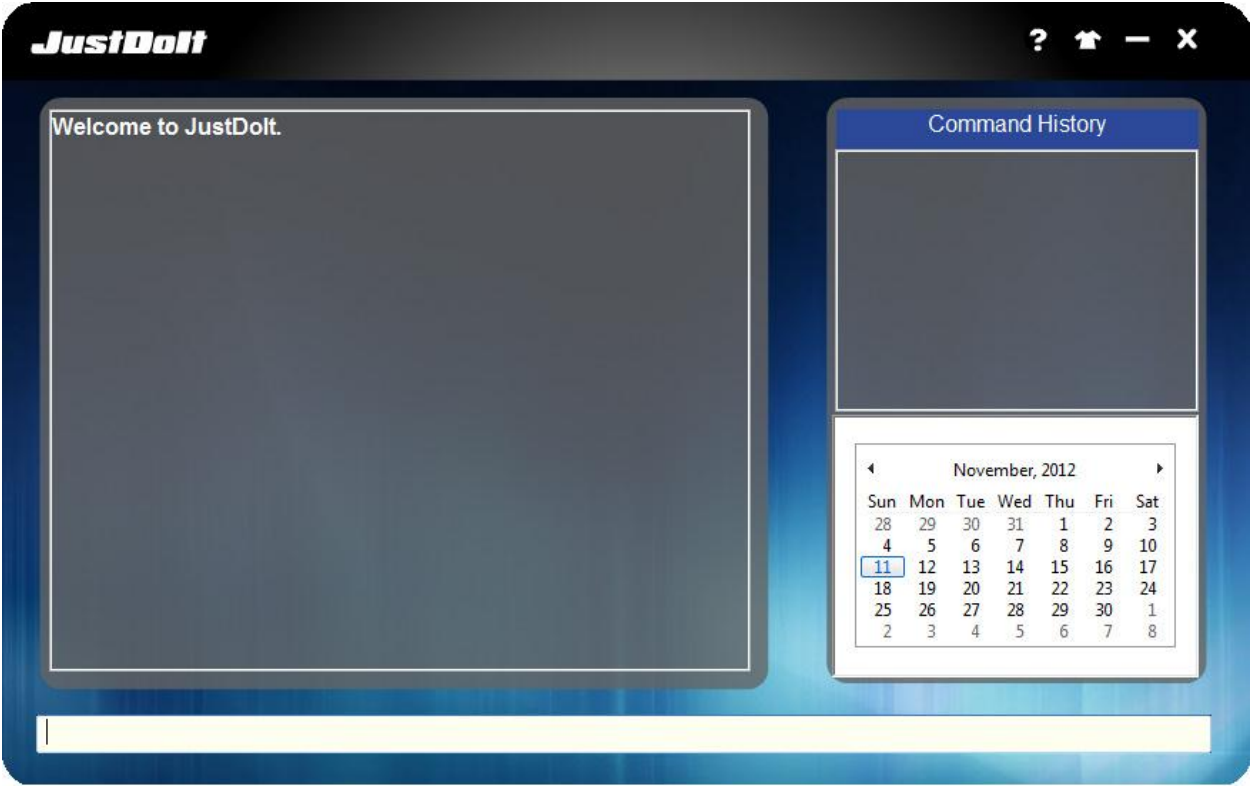


# JustDolt



Choo Cheng Mun  
Paulina  
Leader, Coder,  
Documentation



Liu Weiran  
Coder, UI Designer



Chen Zeyu  
Coder, Tester



Cui Wei  
Coder, Deadline  
Watcher

## Credits

YamlBeans (<http://code.google.com/p/yamlbeans/>) is used to store the data in a human readable format.

Joda-Time (<http://joda-time.sourceforge.net/>) is used to parse and process dates and times.

JUnit (<http://www.junit.org/>) is used to create test cases for JustDolt.



***JustDolt***

Your ideal scheduler

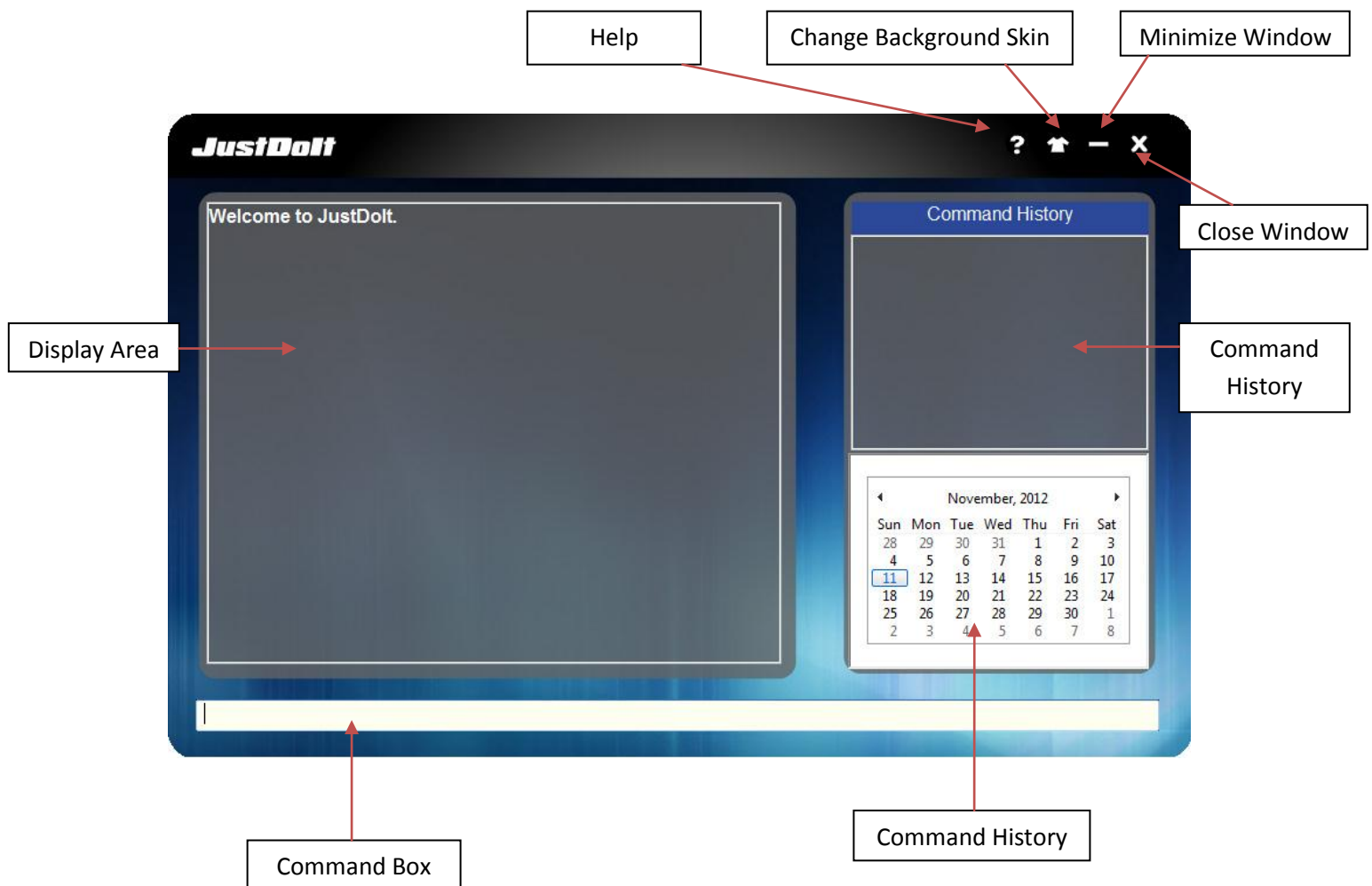
## **User Manual**

## Introduction

JustDolt is a text-based desktop task manager that users can use to create a To-Do list. Users can add tasks they need to do, including specifying deadlines, to a list. These tasks can later be read, modified, or deleted at later times. The tasks can also be searched for by using keywords and times, even if the keywords or times are not identical to the details in the task. JustDolt also supports tagging of tasks so that users can categorize their tasks.

In the following sections, this manual will guide through the various features and functions of JustDolt.

## User Interface



## How to Run

1. Download and install the latest version of Java  
(<http://www.oracle.com/technetwork/java/javase/downloads/index.html>).
2. Download the JustDolt.jar and save it in your desired location.
3. Double click on the jar file to begin running JustDolt.

## General Command Input Format

**<command>** <Task Name/ Additional Command> **\<command>** <Info> **\<command>** <Info>

**Important Note:** Commands indicating starting and ending times, as well as new Task names, are indicated by a “\” before the command.

## Shortcut Keys

Shortcut	Function
Ctrl – A	Display all
Ctrl – P	Display pending
Ctrl – U	Display Unfinished
Ctrl – Z	Undo
Up/Down	Previous/Next Command
ESC	Quit

## Features

### Add Tasks

To ADD an untimed task:

1. Type ‘**add**’, followed by the task name.
2. Optionally, tag the task by adding ‘\#’ followed by the tag.
3. Press ENTER key.

Example: **add** Clean my room \# chores

To ADD timed task:

1. Type '**add**', followed by the task name, as well as the start and/or end times.
2. Optionally, tag the task by adding '**#**' followed by the tag.
3. Press ENTER key.

Example: **add** Orientation Camp **from** 9/11/2012 **to** 12/11/2012

**add** CS2103T Homework **by** 9 September 23.59 **#** Work

## Delete Tasks

To DELETE a task:

1. Type '**delete**', followed by keywords.
2. Press ENTER key.

Example: **delete** CS2103T homework

To DELETE a task (Alternative method):

1. SEARCH for a task or DISPLAY multiple tasks.
2. Type '**delete**', followed by the number of the task on the displayed list.
3. Press ENTER key.

Example: **delete** 2

To DELETE all the tasks:

1. Type '**delete**', followed by '**all**'
2. Press ENTER

Example: **delete all**

To DELETE all the completed tasks:

1. Type '**delete**', followed by '**completed**'
2. Press ENTER

Example: **delete completed**

## Modify Tasks

To MODIFY the details of a task:

1. Type '**update**' followed by the task name.
2. Alternatively, enter the index of the task in the displayed list.

3. To change the task name, type '**\name**' followed by the new name.
4. To change the starting time, type '**\from**' followed by the new time.
5. To change the ending time, type '**\to**' followed by the new time.
6. To change the tag, type '**\#**' followed by the new tag.
7. Press ENTER key.

Example: **update** Orientation Camp **\name** OC at SOC **\from** 13/11/2012 **\to** 15/11/2012  
**update** 1 **\from** 15.30 **\#** needs

To mark a task as COMPLETED:

1. Type '**update**' followed by '**completed**' and the task name.
2. Press ENTER key.

Example: **update completed** CS2103T Homework

To mark a task as COMPLETED (Alternative method):

1. SEARCH for a task or DISPLAY multiple tasks.
2. Type '**done**', followed by the number of the task on the displayed list.
3. Press ENTER key.

Example: **done** 2

## Display Tasks

To DISPLAY all the tasks:

1. Type '**display**', followed by '**all**'
2. Press ENTER key.
3. Alternatively, press CTRL – A on the keyboard.

Example: **display all**

To DISPLAY pending tasks:

1. Type '**display**', followed by '**pending**'.
2. Press ENTER key.
3. Alternatively, press CTRL – P on the keyboard.

Example: **display pending**.

To DISPLAY a certain number of pending tasks:

1. Type '**display**', followed by '**pending**' and a number.

2. Press ENTER key.

Example: **display pending 5**

To DISPLAY unfinished tasks:

1. Type '**display**', followed by '**unfinished**'.
2. Press ENTER key.
3. Alternatively, press CTRL – U on the keyboard.

Example: **display unfinished**

## Search Tasks

To SEARCH for a task with keywords:

1. Type '**search**', followed by the keywords.
2. Press ENTER.

Example: **search CS2103 work**

To SEARCH for a task by time:

1. Type '**search**', followed by the time.
2. Press ENTER key.
3. Alternatively, click on a date on the calendar to search for that particular date.

Example: **search 11/11/2012 23.59**

To SEARCH for tasks by a time frame:

1. Type '**search**', followed by the time frame to search.
2. Enter a start time by typing '**\from**' and a start time.
3. Enter an ending time by typing '**\to**' and an end time.

Example: **search \from 10/10/2012 \to 17/10/2012**

To SEARCH for tasks within current week:

1. Type '**search**', followed by the '**this week**'
2. Press ENTER key.

Example: **search this week**



To SEARCH for a task by tags:

1. Type **'search'**, followed by the **'\#'** and the tag name.
2. Press ENTER key.

Example: **search \#** work

## Read Help Document

To read the HELP document:

1. Type **'help'**.
2. Press ENTER key.
3. Alternatively, click the help button at the top right of the window.

## Undo Operations

To UNDO the last add, modify or delete operation:

1. Type **'undo'**.
2. Press ENTER key.
3. Alternatively, press CTRL – Z on the keyboard.

## Exit the Program

To EXIT the program:

1. Type **'exit'**.
2. Press ENTER key.
3. Alternatively, click the close button at the top right of the window or press the ESC button on the keyboard.

## Valid Input Formats

### Acceptable Commands

Command Type	Acceptable Commands
Add Task	"add", "create", "new"
Delete Task	"delete", "remove"

Modify Task	"modify", "change", "update"
Display Task	"display", "show"
Search Task	"search", "find"
Undo	"undo"
Exit	"exit", "quit", "close"
Start Time	"from", "start", "at"
End Time	"by", "end", "to"
New Task Name	"name", "description"
Tag	"#"
All	"all", "everything"
Complete	"complete", "completed", "finish", "finished", "done"
Pending	"pending"
Unfinished	"unfinished"

## Acceptable Date & Time Formats

**Legend:** Each symbol represents a character in the input

d – Date

m – Month

y – Year

H – Hour (24 hour)

m – Minute

h – Hour (12 hour)

a – AM/PM

### ***Date***

- dd/mm/yyyy
- dd-mm-yyyy
- dd mmm yyyy

### ***Time***

- HHmm
- HH.mm
- HH:mm
- hhmmaa
- hh.mmaa
- hh:mmaa

National University of Singapore

# Developer Guide for JustDoIt

Choo Cheng Mun Paulina, Chen Zeyu, Cui Wei, Liu Weiran  
12 November 2012

# Contents

1. Introduction to JustDolt.....	12
2. Architecture .....	13
2.1. Overview .....	13
2.2. User Interface .....	15
2.3. Parser .....	16
2.4. Logic .....	18
2.5. Storage .....	23
3. Task Model.....	24
4. Testing.....	26
5. Future Development .....	28
Appendix A – Change Log .....	29
Appendix B – List of Acceptable Commands.....	30
Appendix C – API .....	31
GUI class.....	31
UISwitch class.....	31
Helper class .....	31
commandFeedBack class .....	32
Parser class.....	32
TimeParser class.....	33
TypeParser class.....	33
Logic class.....	34
KeywordComparator class .....	35
TimeComparator class .....	35
Storage class .....	37
Task class.....	38

# 1. Introduction to JustDolt

Thank you for your interest in further developing our program, JustDolt. This guide will introduce to you, the new developer, the architecture and design of our program, as well as possible areas for improvement.

JustDolt is a text- based To-Do list creator and manager. By using JustDolt, users will be able to create tasks, with the ability to specify some details, such as deadlines. These tasks can later be read or modified as and when the user wishes to do so. JustDolt also features a search function that is able to search using keywords or timings.

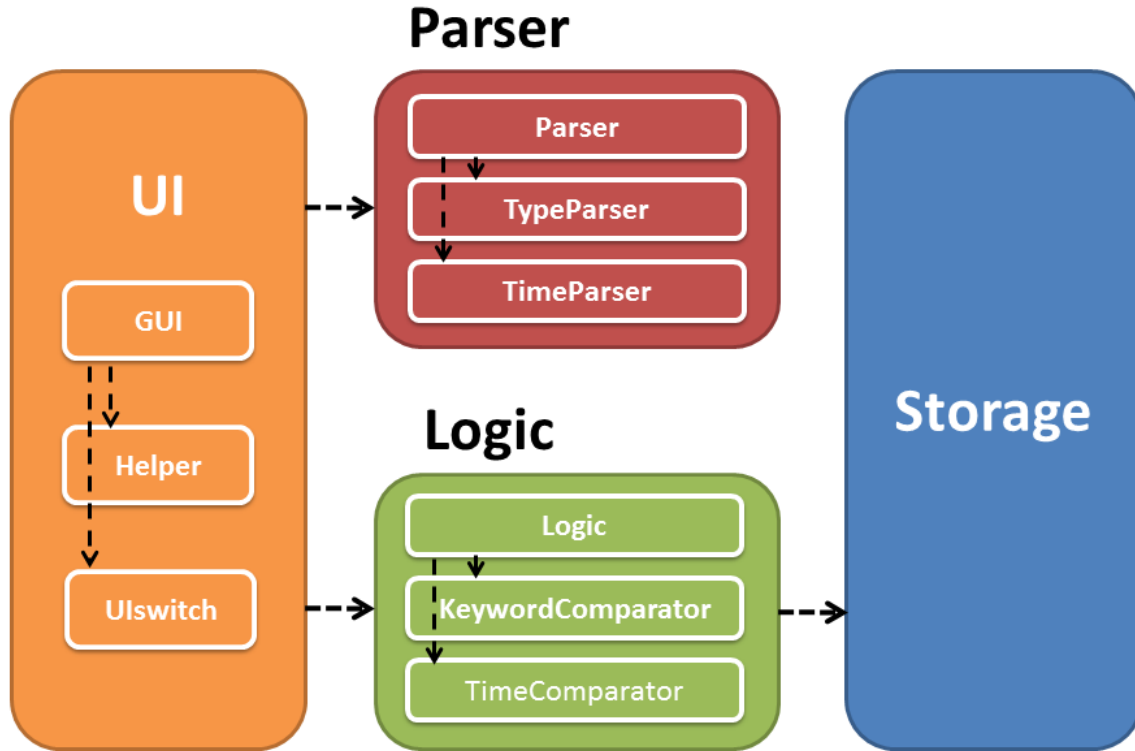
JustDolt is meant to be a lightweight program for Windows, hence the simple Graphic User Interface (GUI). The tasks are also stored offline in a text file which the user can use to refer to their tasks.

JustDolt is targeted at heavy computer users who need to keep track of their tasks in a quick and efficient way. In particular, JustDolt is made for users who prefer using the keyboard to input commands as opposed to using mouse or voice controls. As such, JustDolt utilizes a command-line style of inputting commands. Furthermore, the commands are designed to be as close to the natural English language as possible, making it easy to use even for non-Linux users.

In this guide, we will begin by introducing the architecture of JustDolt, followed by a detailed description of each component. Next, we will comment on the testing of the program and then report currently known issues with the program. Finally, we will provide suggestions for further development of JustDolt.

## 2. Architecture

### 2.1. Overview



**Figure 1: Software Architecture for JustDolt**

The software architecture of JustDolt is as shown in the figure above. It is composed of four major components, which are the User Interface (UI), the Parser, the Logic unit and Storage.

Basically, the UI is responsible for receiving the user's input and directing the information to the Parser and the Logic unit. It is also responsible for displaying the necessary information to the user. The responsibility of the Parser is to extract all information from the user's input. The Logic unit will then execute the command, and the Storage will save and retrieve the tasks as necessary.

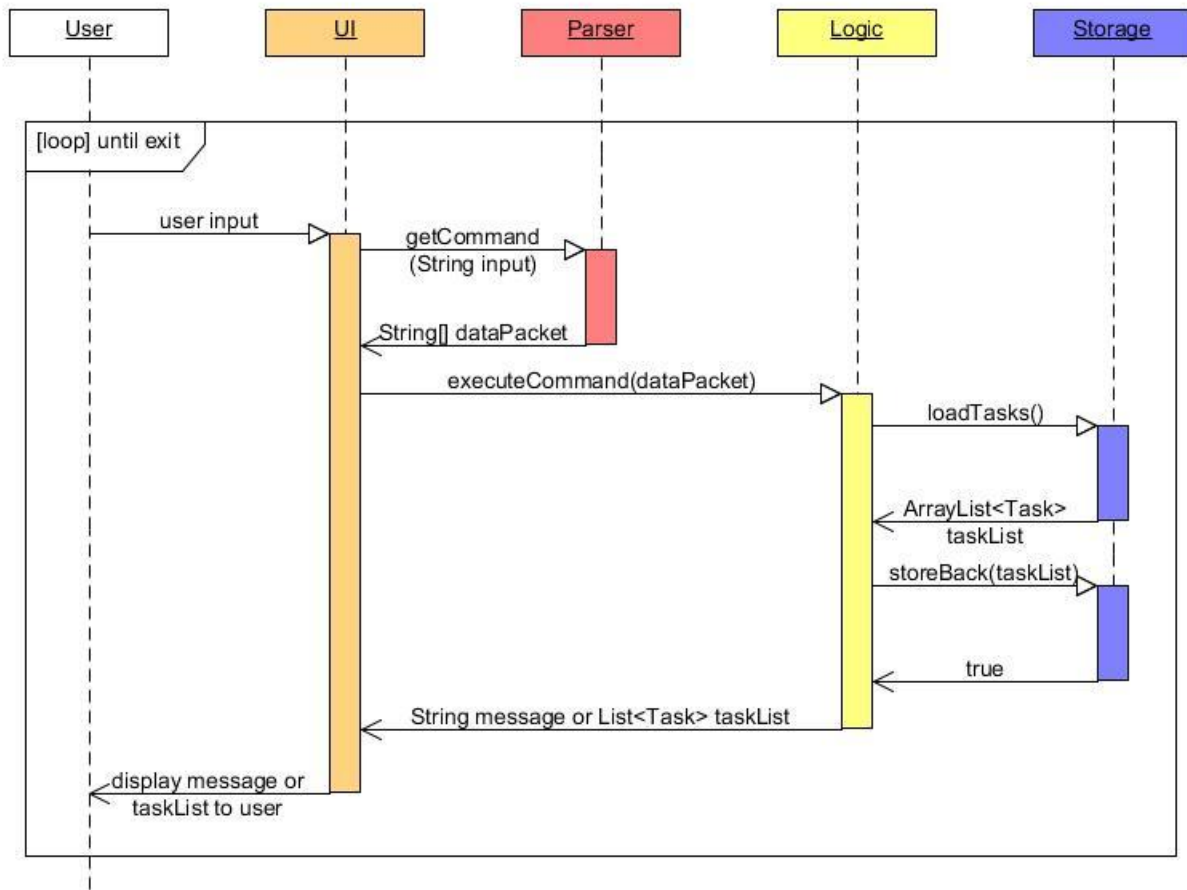


Figure 2: Sequence Diagram for JustDolt

The application begins running in the UI component, which will start off by instantiating a new Logic unit. As shown in Figure 2, upon receiving the user's command, the UI will tell the parser to `getCommand`. The parser will then return a `String[] dataPacket`. The UI then passes `dataPacket` to the Logic unit to execute. The Logic unit will subsequently communicate with the Storage component in order to retrieve the tasks to be processed and write back any new or edited tasks using `loadTasks` and `storeBack`. Lastly, the Logic unit will return any necessary details to be displayed back to the UI.

In the following sections, each component of the JustDolt architecture will be explained in greater detail.

## 2.2. User Interface

The UI component consists of 4 parts: the **GUI** class, the **Uswitch** class, the **Helper** class and the **commandFeedback** class. The former 2 classes are essential for the functionality of JustDolt, while the latter 2 classes are used to create a command input helper for the user.

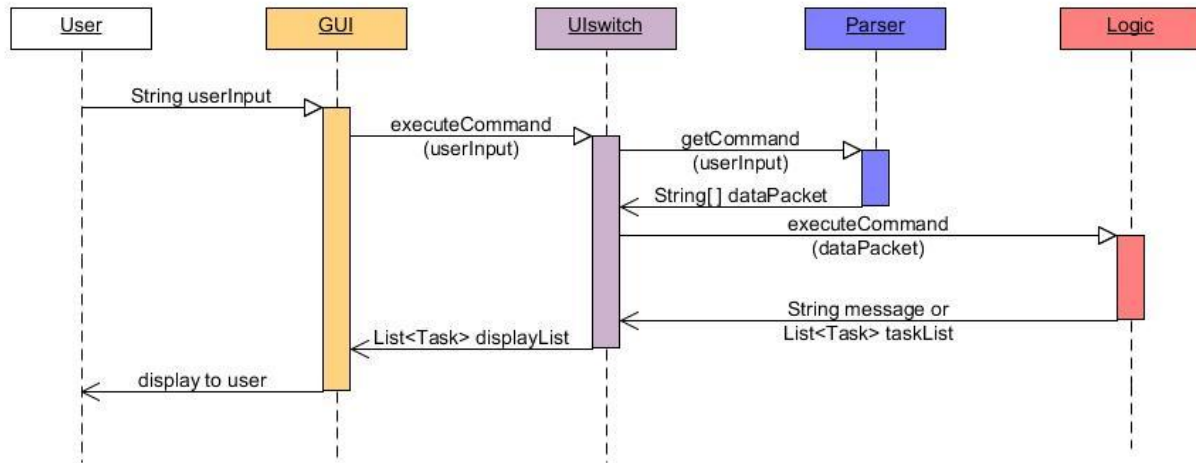


Figure 3: Sequence Diagram for UI component

As shown in figure 3, the **GUI** class is responsible for getting the command input from the user and displaying relevant information back to the user. The **GUI** begins by instantiating a **Uswitch** object. Then, until an exit command is given, the **GUI** listens continuously for a user input. When an input is received, it is passed on to the **Uswitch** using **executeCommand**. The **GUI** then displays the returned **displayList** accordingly.

The **Uswitch** class then acts as a façade between the **GUI** as well as the Parser and Logic components. When the instance of the **Uswitch** is first created, a Logic object is also created. When a call to **executeCommand** is made, the **Uswitch** will communicate with the Parser and Logic components in order to execute the command. The **Uswitch** class also handles any processing of the return message from the Logic component before returning the **displayList** to make it easier for the **GUI** class to handle.



**NOTE**

Since the Parser class does not return any message, the UI has to check the response from the Parser for an error ID and display the appropriate message.

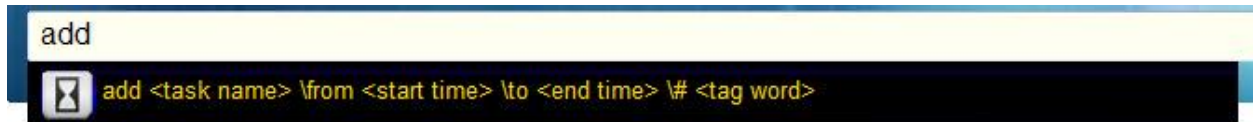


Figure 4: Example of Command Input Helper

JustDolt also implements a command input helper using the **Helper** and **commandFeedback** classes. Figure 4 is an example of how the helper looks like. The **Helper** class, which is instantiated in the **GUI** class, creates the dropdown window of the helper. With every keystroke from the user, the **onCommandChange** method in the **commandFeedback** class is then called to check the user input and display the appropriate help message in the window.

### 2.3. Parser

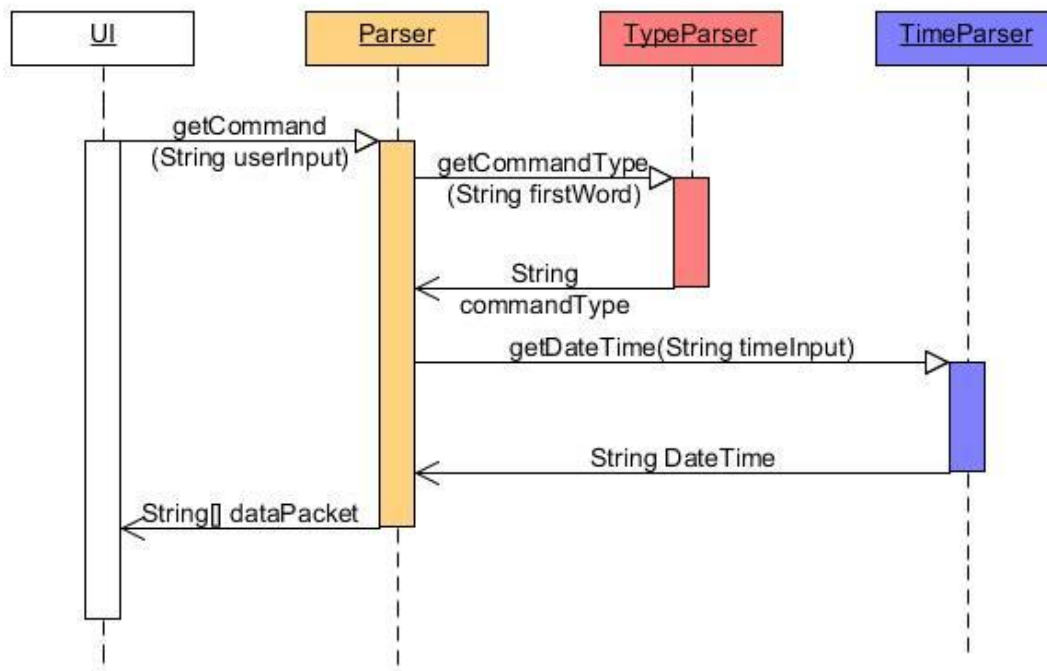


Figure 5: Sequence Diagram for Parser

The parser consists of 3 parts: the main **Parser** class, the **TypeParser** class and the **TimeParser** class.

As shown in Figure 3, the **Parser** receives the String for the user's input from the UI and breaks it up based on certain commands and demarcations using `getCommand` (Refer to Appendix B for the list of acceptable commands). In the process, it calls `getCommandType` in the **TypeParser** class to identify what type of function the user is calling. The information is then extracted by different methods based on the command type. If necessary, the information is further processed by the various methods.

In the event a time or date is input by the user, the methods for extracting information related to the Add, Modify and Search functions then call the **TimeParser** using `getDateTime`. The **TimeParser** receives the String for the time, parses it and returns the time as a String with the format yyyy-mm-dd hh:mm:00.00. This is required to standardize the date-time format processed by the Logic component since the user can input the date in various formats.

The information extracted from the user's input is stored in a **String array**. The information is stored in specific locations in the array that correspond to certain information needed for the various functions. The locations and corresponding information are as follows:

Array Index	Information
0	Command ID
1	Task name
2	Start time
3	End time
4	Tag
5	Any other information

This **String array** is then returned to the UI for processing by the Logic component (refer to section 2.4).

Any invalid input will return a command ID representing the error in the **String array**, which will then be displayed as an error message by the UI component.

## 2.4. Logic

The Logic component is responsible for executing the user's input, allowing the user to create, update, delete, search and display tasks. The Logic component consists of 3 parts: the **Logic** class, the **keywordComparator** class and the **timeComparator** class.

The most essential part of the Logic component is the **Logic** class. It is first instantiated and invoked by the UI. As shown in figure 2 (page 14), the UI uses the **executeCommand** method in the **Logic** class to start the execution of the user's command. The **Logic** class then operates based on the information in the **dataPacket** and returns the message or list of tasks corresponding to the command back to the UI. The **Logic** class also communicates with the Storage instance using the **loadTasks** method in order to access and retrieve tasks from the database located in the secondary storage (e.g. hard drive). It also tells the Storage instance when and what tasks to write back into the secondary storage using the **storeBack** method.

The following are some important methods in the **Logic** class:

Visibility	Method	Description
Public	String executeCommand(String[] info)	Call corresponding function to execute the user's command
Private	String add (String[] info)	Add task instance to the array list
Private	String deleteByIndex(int index)	Delete the task with the specified index.
Private	String deleteByKeyword(String keyword)	Delete all the tasks related to the specified keyword
Private	String updateTask(String taskName, String newName, String startTime, String endTime)	Update the name and time of the specified task
Private	ArrayList<task> searchKeyword(String keyword)	Return an array list of all the tasks related to the specified keyword

Private	ArrayList<task> displayPendingNum(int num)	Return an array list containing specified number of tasks with status "pending".
Private	ArrayList<task> sortTask(ArrayList<task>)	Sort a given array list of tasks in ascending order according to the end time of each task
Private	void undo()	Undo the previous operation
Private	void backUp()	Back up current tasks
Private	void writeBack()	Store all the tasks back to secondary storage

**NOTE**

1. Functions that take index as one parameters (namely, **deleteByIndex** function, **done** function should only be called after a search operation or display operation, because these functions will return an array list of tasks to be displayed. Each task will then be assigned an index, and users will use these indexes as parameters.
2. For each search or display function, the tasks in the returned array list are sorted in ascending order according to the end time.
3. For **displayPendingNum** function, if there are less pending tasks than the specified number, it will return all the pending tasks.
4. Repeated calls to the **undo** function results in multi-steps undo, until there are no more operations to cancel.

**Logic** class also has following important class variables:

Type	Name	Description
Logic	theOne	Default instance created in logic class
ArrayList<Task>	current	Stores updated tasks after the most recent operation
Stack<ArrayList<Task>>	previous	Each time <b>current</b> is modified the array list will be

		copied and stored in this stack for back up purposes.
ArrayList<Task>	toDisplay	Contains the tasks that need to be displayed to the user.
Storage	store	Storage object in charge of loading all tasks from the secondary drive when the software launches, and writing back the tasks in to array list <b>current</b> at the end of each operation

All the operations done by the **Logic** class are done completely in the memory. That is, every time the software is launched, the **Logic** class will call upon the Storage instance to load all the tasks stored in the secondary storage into memory in the form of an array list. Then, for each user request, this array list will be accessed by methods in the **Logic** class in order to execute the operation. At the end of each operation, the whole array list will be written back to secondary storage.

For each search and display function (i.e. **searchKeyword**, **searchTime**, **searchTag**, **displayPending**, **displayAll**, **displayPendingNum**, **displayUnfinished**), the returned array list will be passed to the UI to display to the user. When UI displays the tasks in the array list, it will give each task an index starting from 1. These indexes can be used as parameters for certain delete or update functions.

For search functions, we also distinguish between "exact matches" and "similar matches". By "exact matches", we mean the task name and the user input are exactly the same, or the task name contains the user input. For example, if a task description is "MA3110 Midterm Test", it will be considered "exact match" if the user input is "MA3110 Midterm Test" or keywords such as "MA3110" or "Midterm Test". However, if the user input is something like "MA3115 Midterm Test", the above task will be categorized as "similar match". The search functions will search for "exact matches" first, and if no exact matches are found, it will then try to find "similar matches". Therefore all the returned task objects must fall under only 1 category, either "exact matches" or "similar matches". In order to indicate which category the returned tasks fall under, at the head of the array list, there is a dummy task object indicating whether it contains "exact matches" or "similar matches" in the task name. In the event that no matches are found, the array list will contain only one dummy task object indicating that there are no matches found.

The **Logic** class will also communicate with two other classes: **keywordComparator** and **timeComparator**. Their purposes are as follows:

Class Name	Functionality
<b>keywordComparator</b>	<ol style="list-style-type: none"><li>1. Determine whether two strings are similar</li><li>2. Determine whether a given string is similar to another array of strings</li></ol>
<b>timeComparator</b>	<ol style="list-style-type: none"><li>1. Determine whether two dates are the same</li><li>2. Determine whether two dates are similar</li><li>3. Determine whether a given date is in current week</li><li>4. Determine whether a given date is in the specified time range</li></ol>

The **Logic** class will use the instances of two these classes when a search operation is performed. The following is a sequence diagram of a search by date operation.

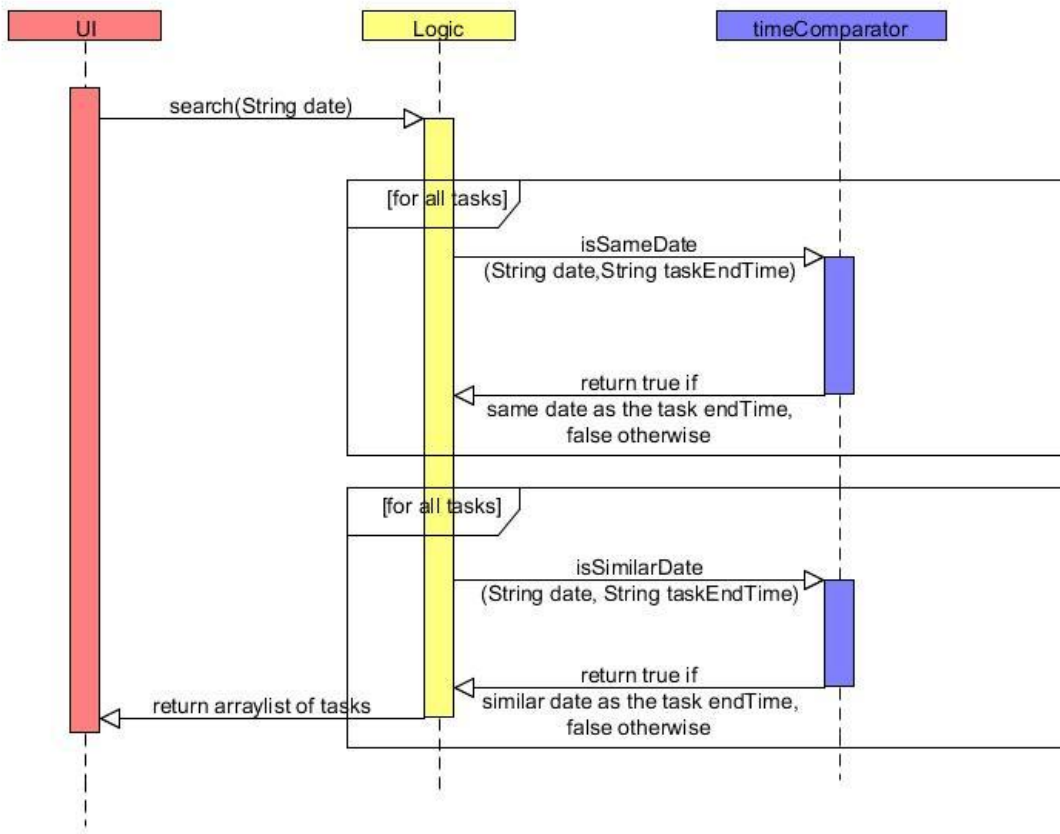


Figure 6: Sequence Diagram for Search by Time Operation

When the search by date function is called, for each task in memory, the **Logic** instance will call the method `isSameDate` in the **timeComparator** class to determine if date of the task matches the input date. If it matches, the task will be added to the `toDisplay` array list. After the first for-loop, if tasks have been added to `toDisplay`, it means that exact matches have been found. Therefore, a dummy task indicating an exact match will be added to the head of `toDisplay` and `toDisplay` will then be returned to the UI.

However, if `toDisplay` is still empty, this means that no exact matches have been found. Therefore, for each tasks stored, the function will then take the user input string representing a date and the task's end time as two parameters, and call the function `isSimilarDate` in the **timeComparator** class to determine if these dates are the similar. If it is, the task will be added to `toDisplay`. By the end of the second loop, if the `toDisplay` array list is not empty, it means that similar matches have been found. Therefore, a dummy task indicating a similar match will

be added to the head of `toDisplay` and `toDisplay` will then be returned to the UI. If the `toDisplay` array list is still empty after the second loop, it means that no matches were found. Therefore, a dummy task indicating that no matches were found will be added to the head of `toDisplay` and `toDisplay` will then be returned to the UI.

Similarly, when the search by keyword function is called, the **Logic** class will repeat the above operations using the **keywordComparator** class.

## 2.5. Storage

The Storage component loads and stores tasks as directed by the Logic component. It is able to load tasks from the local database file into the memory and write back to the file if necessary. The storage class contains the following methods:

Visibility	Method	Description
Public	<code>loadTasks()</code>	Return an <code>ArrayList&lt;Task&gt;</code> which contains all the task instances in local yaml file.
Public	<code>storeBack(ArrayList&lt;Task&gt; allTasks)</code>	Write all task instances in the input <code>ArrayList&lt;Task&gt;</code> back to local yaml file, return true if write back is successful.

The Logic component will invoke `loadTask` when it is initialized, loading all tasks from the local yaml file into the primary memory so that the tasks can be manipulated and modified. The `loadTask` method also checks if the loaded tasks have passed their deadline since the last load. If so, the task status would be changed to unfinished. Once the modifications are done, `storeBack` is called in order to write the updated tasks back in to the local yaml file. `storeBack` should be called after every operation in the **Logic** component to reduce loss of data in the event of an error.

### NOTE

Storage does not check if the input to `storeBack` is null or not. This should be done in the Logic component. Storage will not handle such exceptions and will assume that every input is legal.



### 3. Task Model

The Task class is used to create a Task object for every task that the user creates. Each Task instance has 5 attributes:

Type	Name
LocalDate	startTime
LocalDate	endTime
String	taskName
String	taskStatus
String	taggedWord

There are some tasks that do not have specific values for `startTime`, `endTime` and `taggedword`. Therefore we have some default values for those attributes to avoid exceptions. `startTime`'s default value is "1991-11-11 00:00:00.000", `endTime`'s default value is "2091-11-11 00:00:00.000" and `taggedword`'s default value is "k".

Each task object models a task within memory. To retrieve a variable, the getter method for the variable is used. Modifications can then be done by calling the setter methods of each attribute. The following are examples of getter and setter methods:

Visibility	Method	Description
Public	<code>getEndTime()</code>	Return the end time as a String.
Public	<code>setEndTime(String endTime)</code>	Replace the current end time with the given new endTime.

Developers should note that we have 3 constructors in Task class. One receives a `String[] args` and each element of `args` contains specific information. `args[0]` is the `taskName`, `args[1]` is the `startTime`, `args[2]` is the `endTime` and `args[3]` is `taggedWord`. `taskStatus`

is set to be “pending” by default. Another constructor receives another Task object and clones a new one. The third constructor is a do-nothing constructor, reserved for the yaml format.

**NOTE**

All class variables' visibilities are set to be public even though they are usually private. This is done in order to create the yaml file for local storage.

## 4. Testing

Testing is very crucial to the development process as it helps to spot bugs and prevent regressions as early as possible. Our code makes use of the JUnit library included with Eclipse to create unit tests. Even though currently not all methods have unit tests for them, the most important methods are covered in the tests to ensure that there are no problems with the core functionality.

Here are some tips that developers can consider when creating their test cases.

- Test cases should be created to simulate the event when unexpected inputs, such as null inputs, are received. Likewise, functions should be designed with error handling in mind. Below is an example of such a test case.

```
@Test
public void addTest() {
    test = Parser.getCommand("add eat lunch \\by 8/12/2012 1200pm");
    assertTrue(test[0].equals("add"));
    assertTrue(test[1].equals("eat lunch"));
    assertTrue(test[2] == null);
    assertTrue(test[3].equals("2012-12-08 12:00:00.000"));
    assertTrue(test[4] == null);
    assertTrue(test[5] == null);

    test = Parser.getCommand("add");
    assertTrue(test[0].equals("errorParam"));

    test = Parser.getCommand("add eat lunch \\by bahbahblacksheep");
    assertTrue(test[0].equals("errorParam"));
}
```

**Code Sample 1: (Partial) Unit test for Parser class, method extractAdd**

In Code Sample 1, the 1<sup>st</sup> test case checks if the correct information is extracted when the user input is valid. The 2<sup>nd</sup> and 3<sup>rd</sup> test cases then test that the correct error message is given when the user inputs an invalid command.

- Functions should also be designed such that the return values can be compared with an expected output. This is because unit tests usually compare actual output of functions with expected output to tell whether these functions are working or not. Below is an example of such a function.

```
public boolean storeBack(ArrayList<Task> allTasks) {  
    if (clearStorage()) {  
        if(allTasks.size() == 0) return true;  
        try {  
            YamlWriter writer = new YamlWriter(new  
                FileWriter(storagePath));  
            for (int i = 0; i < allTasks.size(); i++) {  
                writer.write(allTasks.get(i));  
            }  
            writer.close();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
    return true;  
}
```

**Code Sample 2: storeBack Method from Storage class**

In Code Sample 2, the method returns a Boolean variable to indicate whether the operation is successful or not. This allows the method to be easily tested by checking if the return value is true or not.

## 5. Future Development

The following are some suggestions for improvements that future developers can work on.

- Improvements to the GUI (e.g. use of table to display tasks)
- Integration with Google Calendar
- Support for tagging of tasks with more than one tag
- Greater flexibility with command format and date-time inputs
- Alerts or notifications when a task is reaching its deadline
- Storage pattern optimization to avoid loading whole storage files when launching the software

## Appendix A – Change Log

### Version 0.1

- First working version
- Able to create, read, delete and modify tasks
- Simple search function implemented
- Text UI

### Version 0.2

- Added tagging function
- Added power search function
- Added search by time
- First version of GUI

### Version 0.3

- Added search by time period
- Added search for current week
- Improvements to GUI

### Version 0.4

- Improvements to GUI
- Implement multi-step undo
- Bug fixes

### Version 0.5

- Final release

## Appendix B – List of Acceptable Commands

### General Input Format

<command> <Task Name/ Additional Command> \<command> <Info> \<command> <Info>

**Important Note:** Commands indicating starting and ending times, as well as new Task names, are indicated by a “\” before the command.

Command Type	Acceptable Commands
Add Task	"add", "create", "new"
Delete Task	"delete", "remove"
Modify Task	"modify", "change", "update"
Display Task	"display", "show"
Search Task	"search", "find"
Undo	"undo"
Exit	"exit", "quit", "close"
Start Time	"from", "start", "at"
End Time	"by", "end", "to"
New Task Name	"name", "description"
Tag	"#"
All	"all", "everything"
Complete	"complete", "completed", "finish", "finished", "done"
Pending	"pending"
Unfinished	"unfinished"

## Appendix C – API

### *GUI class*

#### **GUI**

```
public GUI() throws IOException
```

Creates an instance of GUI.

#### **Throws:**

IOException – unable to create the interface

---

### *Ulswitch class*

#### **Ulswitch**

```
public Ulswitch()
```

Constructs an instance of Ulswitch

#### **executeCommand**

```
public List<Task> executeCommand(String userInput)
```

This operation executes the user input.

#### **Parameters:**

userInput – The users' input received from GUI class

#### **Returns:**

Returns a list of tasks to be displayed, the first task in the list is always a message.

---

### *Helper class*

#### **executeCommand**

```
public List<Task> executeCommand(String userInput)
```

This operation executes the user input.

#### **Parameters:**



userInput – The users' input received from GUI class

**Returns:**

Returns a list of tasks to be displayed, the first task in the list is always a message.

---

***commandFeedBack class***

**getInstance**

```
public static commandFeedback getInstance()
```

Constructs an instance of commandFeedback. Only 1 instance of commandFeedback can exist at any time.

**Returns:**

Returns an instance of commandFeedback.

**onCommandChange**

```
public String onCommandChange(String text)
```

This operation checks the input from the user and returns the appropriate help message.

**Parameters:**

text – The user input

**Returns:**

Returns the message to be displayed to the user.

---

***Parser class***

**getCommand**

```
public static String[] getCommand(String userCommand)
```

This operation extracts all necessary information from the user's input.

**Parameters:**

userCommand - The command input by the user

**Returns:**

Returns dataPacket, which is a String[] with all information

---

## ***TimeParser class***

### **getDate**

**public static** String getDate(String inputDate) **throws** IllegalArgumentException

This operation converts the input date into a standardized format.

#### **Parameters:**

inputDate - The date input by the user

#### **Returns:**

Returns dateTime, which is the date as a String in a standard format.

#### **Throws:**

IllegalArgumentException - If date entered is in an invalid format

### **checkDate**

**public static boolean** checkDate(String date)

This operation checks if the input is a date.

#### **Parameters:**

date - The date input by the user

#### **Returns:**

Returns true if the input is a date, false otherwise.

---

## ***TypeParser class***

### **getCommandType**

**public static** String[] getCommandType(String command)

This operation gets the type of command.

#### **Parameters:**

command - The command input by the user

**Returns:**

Returns a String that indicates the type of command

---

***Logic class***

**getInstance**

```
public static Logic getInstance()
```

This function gets an instance of Logic. Only 1 Logic instance can exist at any time.

**Returns:**

Returns an instance of the Logic class.

**executeCommand**

```
public Object executeCommand(String[] info)
```

This function executes commands of the user.

**Parameters:**

info - The String[] containing all information needed for commands

**Returns:**

Returns a String if only one line of message needs to be displayed.

Returns an ArrayList of Tasks if a list of tasks needs to be displayed.

**resetTaskKeyword**

```
public void resetTaskKeyword(Task Temp)
```

This function sets the Keyword attribute of a task to null.

**Parameters:**

temp - The Task whose keyword needs to be reset

**resetShouldRefresh**

```
public void resetShouldRefresh()
```

This function sets the boolean variable shouldRefresh to false.

## ***KeywordComparator class***

### **similar**

```
public boolean similar(String first, String second)
```

This function determines if two strings are similar

#### **Parameters:**

first – One string to be checked

second - The other string

#### **Returns:**

Returns true if two strings are similar, and false otherwise

### **similar**

```
public boolean similar(String first, String[] info)
```

This function determines if a string is similar to an array of strings

#### **Parameters:**

first - The string to be checked

info - The array of strings

#### **Returns:**

Returns true if the string is similar to the array of strings, and false otherwise

---

## ***TimeComparator class***

### **isSameDate**

```
public boolean isSameDate(String one, String other)
```

This function determines if the dates represented by two strings are the same date

#### **Parameters:**

one - The string represents the first date

other - The string represents the second date

#### **Returns:**

Returns true if the dates represented by two strings are the same date, and false others

### **isSimilarDate**

```
public boolean isSimilarDate(String one, String other)
```

This function determines if the dates represented by two strings are the similar date

#### **Parameters:**

one - The string represents the first date

other - The string represents the second date

#### **Returns:**

Returns true if the dates represented by two strings are the similar date, and false others

### **inCurrentWeek**

```
public boolean inCurrentWeek(String time)
```

This function determines if a given date is within current week

#### **Parameters:**

time - The string represents the date to be checked

#### **Returns:**

Returns true if the date is within current week and false otherwise.

### **withinPeriod**

```
public boolean withinPeriod(String time,String start, String end)
```

This function determines if a given date is within the specified period.

#### **Parameters:**

time - The string represents the date to be checked

start - The string represents the starting date of the period

end - The string represents the ending date of the period

#### **Returns:**

Returns true if the date is within the given period and false otherwise.

## ***Storage class***

### **getInstance**

```
public static storage getInstance()
```

This function returns an instance of Storage. Only 1 instance of Storage can exist at the same time.

#### **Returns:**

Returns an instance of Storage.

### **loadTasks**

```
public ArrayList<Task> loadTasks()
```

This method load all tasks in local yaml file to an ArrayList.

#### **Returns:**

Returns the ArrayList contains all tasks in local yaml file.

### **clearStorage**

```
public boolean clearStorage()
```

This method clears local yaml file.

#### **Returns:**

Returns true if successful, false otherwise.

### **storeBack**

```
public boolean storeBack(ArrayList<Task> allTasks)
```

This method stores the given ArrayList back to the local yaml file.

#### **Parameters:**

allTasks – The ArrayList of tasks to be written back to the yaml file

#### **Returns:**

Returns true if successful, false otherwise.

## ***Task class***

### **getTaskName**

```
public String getTaskName()
```

Getter method for taskName

#### **Returns:**

Returns the name of task

### **setTaskName**

```
public void setTaskName(String taskName)
```

Setter method for taskName

#### **Parameters:**

taskName - The String of the taskName

### **getStartTime**

```
public String getStartTime()
```

Getter method for startTime

#### **Returns:**

Returns the startTime of task

### **setStartTime**

```
public void setStartTime(String startTime)
```

Setter method for startTime

#### **Parameters:**

startTime - The String of startTime

### **getEndTime**

```
public String getEndTime()
```

Getter method for endTime

#### **Returns:**

Returns the endTime of task

### **setEndTime**

```
public void setEndTime(String endTime)
```

Setter method for taskName

#### **Parameters:**

endTime - The String of endTime

### **getTaskStatus**

```
public String getTaskStatus()
```

Getter method for taskStatus

#### **Returns:**

Returns the status of task

### **setTaskStatus**

```
public void setTaskStatus(String taskStatus)
```

Setter method for taskName

#### **Parameters:**

taskStatus - The String of taskStatus

### **getTaggedWord**

```
public String getTaggedWord()
```

Getter method for taggedWord

#### **Returns:**

Returns the taggedWord of task

### **getKeyWord**

```
public String[] getKeyWord()
```

Getter method for KeyWord

#### **Returns:**

Returns the KeyWord of task

### **setKeyWord**



```
public void setKeyWord(String keyWord)
```

Setter method for KeyWord

**Parameters:**

keyWord - The String of keyWord

**setTaggedWord**

```
public void setTaggedWord(String taggedWord)
```

Setter method for taggedWord

**Parameters:**

taggedWord - The String of taggedWord

**isDefaultStart**

```
public boolean isDefaultStart()
```

To determine if a task's startTime is equal to default value

**Returns:**

Returns true if startTime is equal to defaultStartTime, false otherwise.

**isDefaultEnd**

```
public boolean isDefaultEnd()
```

To determine if a task's endTime is equal to default value

**Returns:**

Returns true if endTime is equal to defaultEndTime, false otherwise.

**isDefaultTag**

```
public boolean isDefaultTag()
```

To determine if a task's taggedWord is equal to default value

**Returns:**

Returns true if taggedWord is equal to defaultTaggedWord, false otherwise.

**isUrgent**

```
public String isUrgent()
```

To determine if a task is urgent(i.e. due within 24 hours)

**Returns:**

Returns amount of time before deadline in a format as [in %1\$s h %2\$s m!], otherwise return null if not urgent.

**isValidTime**

```
public boolean isValidTime()
```

To determine if the start time of a task is not after its end time.

**Returns:**

Returns true if the start and end times are correct, false otherwise.