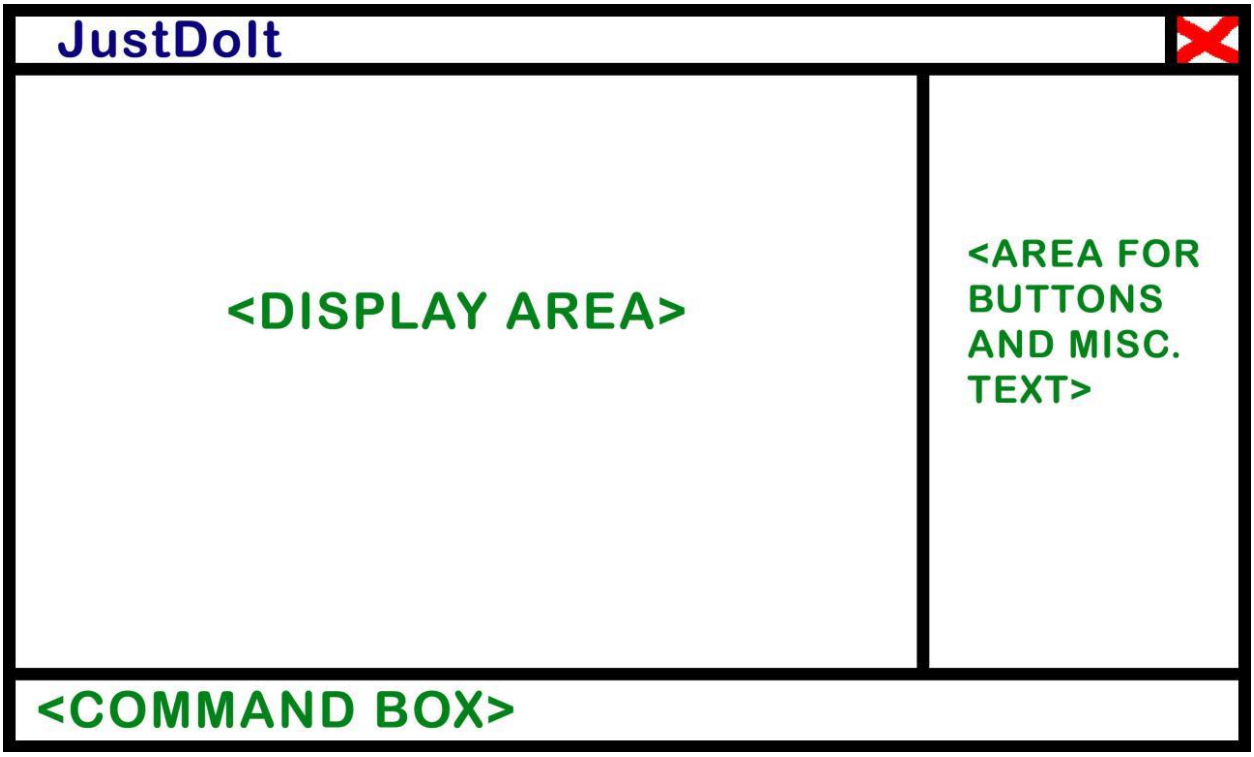
JustDoIt



			
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 JustDoIt.

User Guide for "JustDoIt"

Introduction

JustDoIt 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. JustDoIt 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 JustDoIt.

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.

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

Tagging Tasks

To TAG a task:

1. Type '**tag**', followed by the task name, **‘\#’** and the tag name.
2. Press ENTER.

Example: **tag** CS2103 homework **\#** homework

To remove a TAG:

1. Type **‘tag remove’**, followed by the task name.
2. Press ENTER key.

Example: **tag remove** CS2103 homework

Delete Tasks

To DELETE a task:

1. Type '**delete**', followed by keywords.
2. Press ENTER key.
3. If only one task matches the keyword, the task will be deleted.
4. If there is more than one matching task, a list of matching tasks will be displayed.
5. Type **‘delete’**, followed by the number of the task on the list.
6. Press ENTER key.

Example: **delete** CS2103T homework

delete 2

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

Example: **update** Orientation Camp **\name** OC at SOC **\from** 13/11/2012 **\to** 15/11/2012
update Nap **\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

Example: **display all**

To DISPLAY pending tasks:

1. Type '**display**', followed by '**pending**'.
2. Press ENTER key.

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.

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.

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

To SEARCH for tasks by a tag:

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

Example: **search \#** homework

Read Help Document

To read the HELP document:

1. Type **'help'**.
2. Press ENTER key.

Undo Operations

To UNDO an operation:

1. Type '**undo**'.
2. Press ENTER key.

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.

National University of Singapore

Developer Guide for JustDoIt

Choo Cheng Mun Paulina, Chen Zeyu, Cui Wei, Liu Weiran

22 October 2012

Contents

1. Introduction to JustDoIt

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

JustDoIt is a text- based To-Do list creator and manager. By using JustDoIt, 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. JustDoIt also features a search function that is able to search using keywords or timings.

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

JustDoIt is targeted at heavy computer users who need to keep track of their tasks in a quick and efficient way. In particular, JustDoIt is made for users who prefer using the keyboard to input commands as opposed to using mouse or voice controls. As such, JustDoIt 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 JustDoIt, 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 JustDoIt.

2. Architecture

2.1. Overview

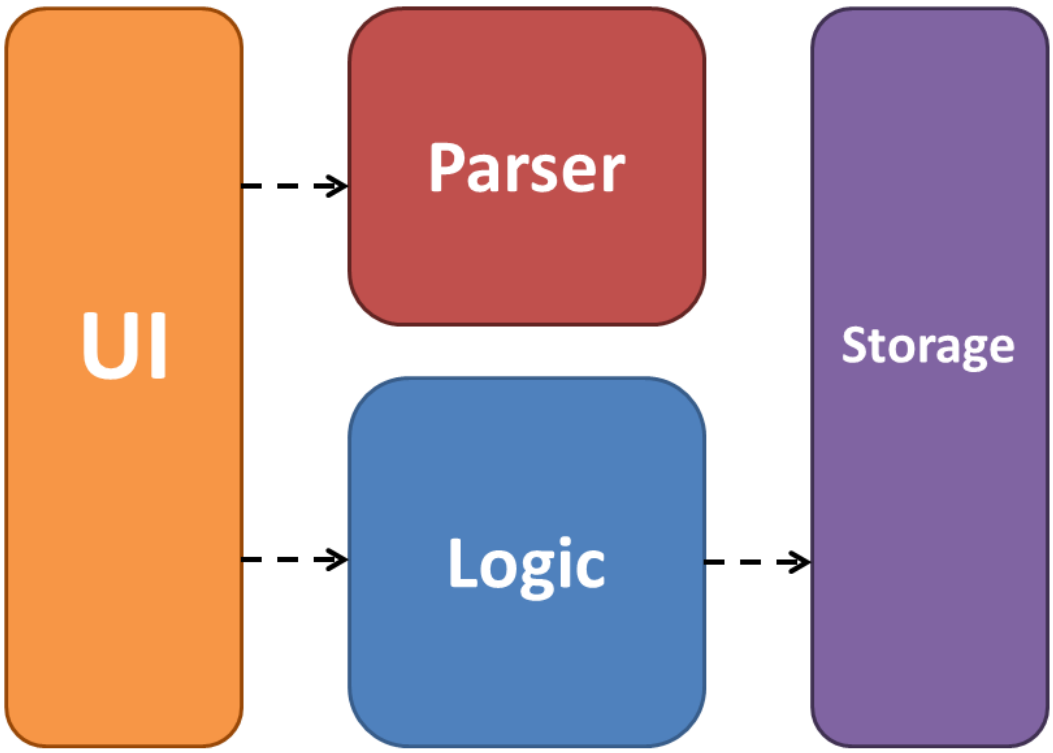


Figure 1: Software Architecture for JustDoIt

The software architecture of JustDoIt 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.

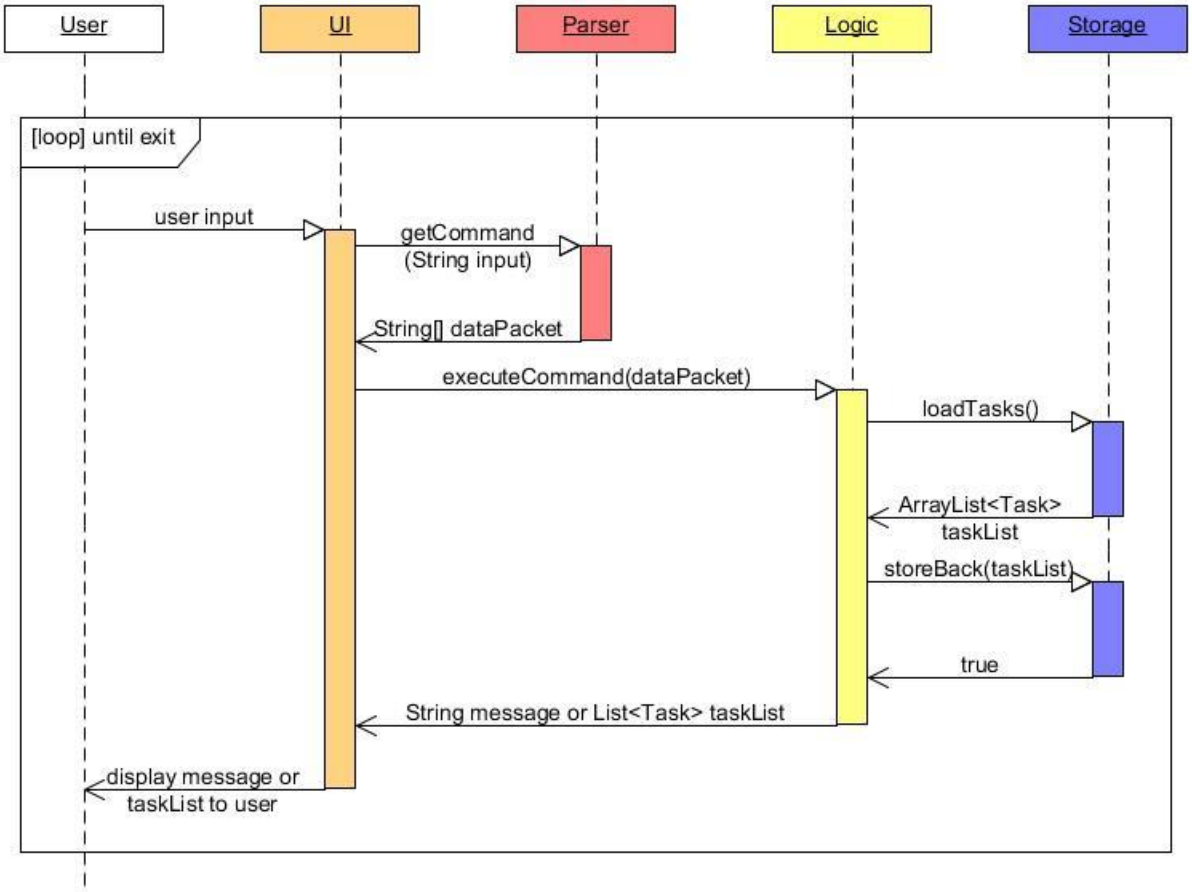


Figure 2: Sequence Diagram for JustDoIt

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 JustDoIt architecture will be explained in greater detail.

2.2. User Interface

The responsibility of the UI is to display relevant information to the user, as well as retrieve any input from the user. The UI is also the class that starts the execution of the program. It instantiates the Logic object, and communicates with the Parser and Logic classes to execute the user command. (Refer to Figure 2) The most important methods for the UI are as follows:

Visibility	Method	Description
private	beginExecution()	Retrieves user input, tells parser to parse it, then tells logic to start executing the command
public	display(Object message)	Displays the message or list of tasks to the user

Until the user inputs an exit command, the UI will continue to watch for input from the user and execute them accordingly with `beginExecution`. Upon receiving a response from the Logic class, the UI will display the response using the `display` method.

Take note that 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.

2.3. Parser

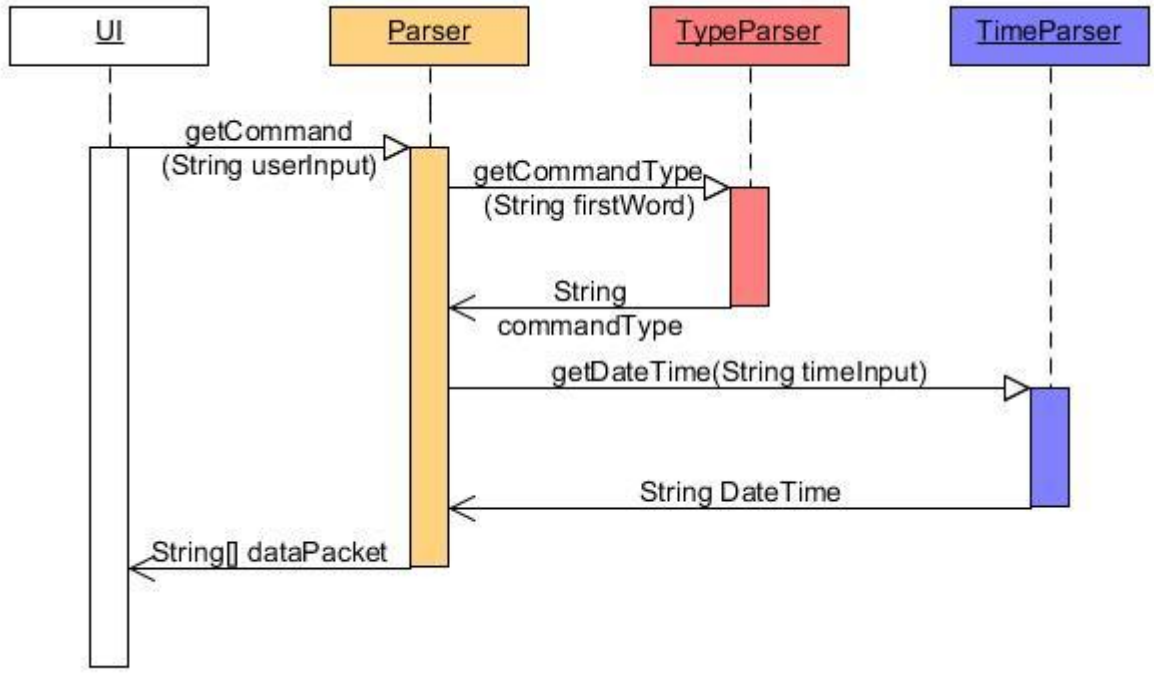


Figure 3: Sequence Diagram for Parser

The parser consists of 3 parts: the main **Parser** class, **TypeParser** class and **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 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 **TimeParser**. **TimeParser** receives the String for the time, parses it and returns the time as a String with the format yyyy-MM-dd. This is required to standardize the date-time format processed by the Logic component.

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.

2.4. Logic

Logic instance executes the user's input, allowing the user to create, update, search, display and delete tasks. It is instantiated and invoked by the UI**.** Logic instance acquires commands from the UI and returns the message or list of tasks corresponding to the command back to the UI. Logic instance also communicates with the Storage in order to access and retrieve tasks from the database file in the secondary storage (e.g. hard drive). It also tells Storage when and what tasks to write back into the memory.

Logic class implements the ILogic interface with the following functions:

Visibility	Method	Description
Public	String executeCommand(String[] info)	Call corresponding functions execute the user's command
Public	String add (String[] info)	Add task instance to the array list
Public	String deleteByIndex(int index)	Delete the task with the specified index. Developer should notice that **the user should only use this function after a search operation or display operation**
Public	String deleteByKeyword(String keyword)	Delete all the tasks related to the specified keyword
Public	String updateTask(String taskName, String newName, String startTime, String endTime)	Update the name and time of the specified task
Public	ArrayList<task> searchKeyword(String keyword)	Return an array list of all the tasks related to the specified keyword
Public	ArrayList<task> displayPendingNum(int num)	Return an array list containing specified number of tasks with status "pending". Tasks are sorted in **ascending** order according to the **end time**. If there are less pending tasks than the specified number, it will all the pending tasks

Public	ArrayList<task> sortTask(ArrayList<task>)	Sort a given array list of tasks in ascending order according to the end time of each task
Public	void undo()	Undo the previous operation. This function only supports **one-step-back** undo operation
Public	void backUp()	Back up current tasks
Public	void writeBack()	Store all the tasks back to secondary storage

Developers should notice that all the operations are done completely in the memory. That is, every time the software is launched, all the tasks stored in the secondary storage will be loaded into memory in the form of an array list. Then for each user request, functions of the logic instance will access this array list to execute the operation. At the end of each operation, the whole array list will be sent back to secondary storage.

Logic class has four class variables:

```
private ArrayList<Task> current;

private ArrayList<Task> previous;

private ArrayList<Task> toDisplay;

private storage store;
```

The array list `current` stores updated tasks after the most recent operation, while the array list `previous` stores the tasks right before the most recent operation so we can execute the undo operation. `current` and `previous` store the same content if the most recent operation is an undo operation. The array list `toDisplay` contains the tasks that need to be displayed to the user. Note that this variable is useful only when the user's command is "search" or "display" as it helps in identifying the correct task to operate on by keeping track of the current index assigned to the tasks. The `store` variable is the Storage object in charge of loading all tasks from the secondary drive when the software launches, and storing back the tasks in to array list `current` at the end of each operation.

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. Please note that when UI display tasks

in the array list, it will give each task an index starting from 1. For example, suppose if the user executes the `displayAll` function, the output will be as follows:

```
---Task 1---
Task Name:CS2101 Developer Guide
Start Time:2012-10-09
End Time:2012-10-15

---Task 2---
Task Name:MA3110 Midterm Test
Start Time:
End Time:2012-10-15


---Task 3---
Task Name:MA3238 Homework 2
Start Time:
End Time:2012-10-12
```

The function `deleteByIndex` should only be executed after one of the above search or display function, because users should use the index from the screen output as its parameter.

Internally, the logic class can be optimized to be able to handle the operations better and more efficiently. For example, the present implementation the software only supports one-step-back undo operation. Developers can use a HashMap to store the `current` array list after each operation, so that the user can go back as many steps as possible.

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	loadTasks()	Return an ArrayList<Task> which contains all the task instances in local yaml file. When loading it will check if a task has passed its deadline. If it has, change the status of the task to be incomplete.
Public	storeBack(ArrayList<Task> allTasks)	Write all task instances in the input ArrayList<Task> back to local yaml file, return true if write back is successful.

The Logic component will invoke `loadTask` when Logic is initialized, loading all tasks in the local yaml file into memory so that the tasks can be manipulated and modified.

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

2.6. 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 `startTime`, `endTime` and `taggedword`. Therefore we have some default values for those attributes to avoid exceptions. `startTime's` default value is "1991-11-11", `endTime's` default value is "2091-11-11" and `taggedword's` default value is "k".

Each task object models a task within memory. Modifications can be done by calling the setter methods of each attribute (e.g. `setEndTime`).

Developers should notice 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.

One more thing to take note of is that all class variables' visibilities are set to be public even though they are usually private. This is also designed purposely for yaml format.

3. Testing

Testing is very crucial in the development process as it helps spot bugs and prevents possible regression problems in the early stage. Our code makes use of the JUnit library included in Eclipse for unit testing. Note that it is not compulsory to do a unit test for every method in each class, but at least test significant methods in the code for bugs.

Currently, the Parser, Logic, and Storage components have implemented classes for testing. Some unit testing examples are shown below.

- Unit test for Parser component, method **`extractModify`**

```
@Test
   public void modifyTest() {
          test = Parser.getCommand("modify eat lunch \\by 123 \\ from 456");
          assertTrue(test[0].equals("updateTime"));
          assertTrue(test[1].equals("eat lunch"));
          assertTrue(test[2].equals("456"));
          assertTrue(test[3].equals("123"));
          assertTrue(test[4] == null);
          assertTrue(test[5] == null);

          test = Parser.getCommand("update complete eat lunch");
          assertTrue(test[0].equals("updateStatus"));
          assertTrue(test[1].equals("eat lunch"));
          assertTrue(test[2] == null);
          assertTrue(test[3] == null);
          assertTrue(test[4] == null);
          assertTrue(test[5].equals("complete"));

          test = Parser.getCommand("change");
          assertTrue(test[0].equals("errorParam"));
   }
```

- Unit test for Storage component

```
@Test
public void test() {
       storage s = new storage();
       String[] args1 = {"tell a true story","1991-12-10","1991-12-
       11","#cs2103"};
       String[] args2 = {"tell a true story","1992-12-12","2013-11-
       01","#cs2103"};
       String[] args3 = {"tell a true story",null,null,"#cs2103"};
       Task t1 = new Task(args1);
       Task t2 = new Task(args2);
```

```
        Task t3 = new Task(args3);
        //both tasks' status are pending
        System.out.println(t1);
        System.out.println(t2);
        ArrayList<Task> ta = new ArrayList<Task>();
        ta.add(t1);
        ta.add(t2);
        ta.add(t3);
        //test storeback
        assertTrue(s.storeBack(ta));
        //test load,
        assertNotNull(s.loadTasks());
        assertTrue(s.storeBack(s.storage));
}
```

Developers should also create test cases for possible unexpected inputs, such as null inputs. Likewise, functions should be designed with error handling in mind.

- Function designing in consideration of unit testing.

```
@Override
public Object executeCommand(String[] info) {

        String commandType = info[0];
        switch (commandType) {
        case "add":
              return add(info);
        case "deleteName":
              return deleteByKeyword(info[5]);
        case "deleteNum":
              return deleteByIndex(Integer.parseInt(info[5]));
        case "deleteAll":
              return deleteAll();
        case "updateTask":
              return updateTask(info[1],info[5],info[2],info[3]);
        case "updateStatus":
              return updateStatus(info[1],info[5]);
        case "searchKey":
              return searchKeyword(info[5]);
        case "searchTime":
              return searchTime(info[5]);
        case "displayAll":
              return displayAll();
        case "displayPending":
              return displayPending();
        case "displayPendingNum":
              return displayPendingNum(Integer.parseInt(info[5]));
        case "undo":
              return undo();
        }

        return GENERAL_ERROR_MESSAGE
```

```
}
```

Unit tests usually compare actual output of functions with expected output to tell whether these functions are working or not. Hence, functions should also be designed such that the return values can be compared with an expected output.

4. Known Issues

At current stage we have implemented the CRUD (create, read, update, delete) functions, as well as a simple search function. However, issues about the flexibility of command inputs still exist. For now, the full name of tasks must be included in the users' command if they want to indicate a certain task to be operated on. For instance, in functions like modifying and deleting, users must type commands like this:

update CS2103 Midterm Presentation in forms of groups \by 2012-10-15

delete Picnic at East Coast Park with Mr.John and His Family

Such inputs can be tedious and inconvenient for users. Therefore, JustDoIt needs an update that supports partial task name input or an auto-complete function to resolve this issue.

5. Future Development

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

- Creation of a Graphic User Interface (GUI) to replace the current TUI
- 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

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 partial search
- Added search by time

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"`
All	`"all", "everything"`
Complete	`"complete", "completed", "finish", "finished", "done"`
Pending	`"pending"`
Unfinished	`"unfinished"`