Contents

[1. Architecture 2](#_Toc414198821)

[2. GUI Component 2](#_Toc414198822)

[2.1. API 3](#_Toc414198823)

[2.2. Use case sequence diagram 3](#_Toc414198824)

[3. Logic Component 3](#_Toc414198825)

[3.1. Task Class 4](#_Toc414198826)

[3.2. DateHandler Class 4](#_Toc414198827)

[3.3. CommandParser Class 5](#_Toc414198828)

[3.4. QLLogic Class 5](#_Toc414198829)

[3.5. Notable Algorithms 6](#_Toc414198830)

[4. Storage Component 9](#_Toc414198831)

[4.1. Class diagram 9](#_Toc414198832)

[4.2. Notable APIs 9](#_Toc414198833)

[5. Google Integration Component 9](#_Toc414198834)

[5.1. API 10](#_Toc414198835)

[6. Testing Methodology 10](#_Toc414198836)

## Architecture

Quicklyst adopts an n-tier architectural style where higher level components make use of the services from lower level components. Hence higher level components are dependent on lower level components while lower level components are independent of higher level components.

The GUI component is at the highest level and is the only component that interacts with the user. It uses APIs provided Logic component to carry out the user’s commands. The Logic component implements the different functionalities of Quicklyst through the help of a few sub-components which will be further elaborated. Finally the Storage and Google Calendar component are at the lowest level and allows data to be loaded and stored. Figure 1 illustrates the architecture of Quicklyst.



*Figure 1. Quicklyst architecture*

## GUI Component

The Graphical User Interface (GUI) provides an interactive and visual indication for the user. By handling the command entered (e.g. add, delete, sort), GUI will update three main fields (task list, overview and feedback) accordingly. Figure 1 below shows the structure of the GUI component and its dependency.



*Figure 2. GUI component class diagram*

### API

Table \_\_ shows some of the notable API of the GUI component.

|  |  |  |
| --- | --- | --- |
| Method | Parameters | Description |
| updateUIWithTaskList (List<Task> taskList) : void | **taskList**: a List<Task> object containing the list of Task to be displayed | Obtain the list of tasks when the user launch the program at the beginning. For each command being processed, the taskList pane will be updated accordingly. |
| actionPerformed (ActionEvent keyEnter) : void | **keyEnter**: an event that detects the “Enter” key | Handles the event specified in the command field when the “Enter” key is pressed by user. |

*Table 1. GUI component API*

### Use case sequence diagram

A sequence diagram shown in *Figure \_\_* demonstrates some examples of the interaction between the user and the GUI.



*Figure 3. Use case sequence diagram*

## Logic Component

The Logic component processes and execute all user commands. It takes in commands from the GUI, executes them and pass a list of task that is required by the user to be displayed back to GUI. It consists of three sub-components- QLLogic, CommandParser and DateHandler, and handles Task objects. The class diagram in Figure \_\_ shows the relationship of the classes that are relevant to the Logic component.



*Figure 4. Logic component class diagram*

### Task Class

The Task Class instantiate a Task object which has attributes of a task in real life. Typical class methods and instance methods such as accessors and modifiers are omitted in the class diagram, and only the notable API is shown below in figure \_\_.

|  |  |  |
| --- | --- | --- |
| Method | Parameters | Description |
| clone(): Task |  | Returns a new instance of a Task with identical attributes as this Task. Used by QLLogic during for undo functionality. |

*Table 2. Task class API*

### DateHandler Class

The DateHandler Class handles anything that deal with dates. It is used by QLLogic when it needs to interpret a date, and also Task when it needs to set and modify its dates. Notable APIs are shown below in figure \_\_.

|  |  |  |
| --- | --- | --- |
| Method | Parameters | Description |
| convertToDateCalendar(  String: dateString): Calendar | **dateString:** a string in the form of ‘DDMM’, ‘DDMMYYYY’, ‘TDY’ or ‘TMR’. | Converts a String of date into a Calendar of date and returns it. |

*Table 3. DateHandler class API*

### CommandParser Class

The CommandParser Class handles commands that are keyed in by the user. It is used by QLLogic when it needs to interpret a command. Notable APIs are shown below in figure \_\_.

|  |  |  |
| --- | --- | --- |
| Method | Parameters | Description |
| splitActionAndFields(  String: command): String[] | **command**: the command string that is typed in by the user. | Split the command into ‘action’ and ‘fields’. ‘action’ is the type of operation such as add, delete, edit, etc. ‘fields’ are the fields of a Task that the action needs to apply on. Returns a String array of size = 2 where the first element is the ‘action’ and second element is the ‘fields’. |
| processFieldLine(String:fieldLine):  LinkedList<String> | **fieldLine**: a string that may contain some fields. | Extracts the individual fields and returns a LinkedList of fields. If there are no fields, returns an empty list. |
| getSortingCriteria(  LinkedList<String>: fields): LinkedList<char[]> | **fields:** a list of fields that may contain the sorting criteria in which higher level sorting criteria appears first in the list. | Interpret each field in the list and determine its soring criterion. Returns a list of char array of size 2. The first element in the char array is the field type and second element is the sorting order. |

*Table 4. CommandParser class API*

### QLLogic Class

The QLLogic class is where the user commands get executed. It executes the commands by accessing and operating on a “working list” of Task objects. The list of Task objects will represent the tasks that the users has specified. Notable APIs are shown below in figure \_\_.

|  |  |  |
| --- | --- | --- |
| Method | Parameters | Description |
| setup(String: filepath):  LinkedList<Task> | **filepath**: the path of the file to load a previously saved list of Tasks. | Sets up the working environment of QLLogic and loads saved Tasks into \_workingList and \_workingListMaster. Returns the loaded \_workingList. |
| executeCommand(  String: command, StringBuilder: feedback):  LinkedList<Task> | **command**: the command string that is typed in by the user.  **feedback**: the feedback to be displayed to the user after each operation. An empty StringBuilder should be passed in during each call of executeCommand | Executes the commands specified by the user. Returns the new \_workingList that satisfies this command. Edits feedback accordingly to correspond to the result of the execution. |

*Table 5. QLLogic class API*

#### \_workingList and \_workingListMaster

QLLogic holds Tasks objects LinkedLists referred to as “working lists”. \_workingList holds the Tasks that are passed to QLGUI to be displayed to the user while \_workingListMaster holds all the Tasks that has been added but not deleted by the user. The object diagram in figure \_\_ illustrates the relationship between the working lists.



*Figure 5. \_workingList & \_workingListMaster object diagram*

As seen from the object diagram, a \_workingList is a subset of \_workingListMaster, and they both hold the references to the same Task when it is present in both lists. Hence changing a Task in one list automatically changes the same Task in the other list. This allows Tasks to be edited and deleted in both lists at the same time.

#### executeCommand

The executeCommand method implements all functionalities of Quicklyst by calling on sub-methods in QLLogic. Current version of QLLogic supports the following sub-methods:

1. executeAdd()
2. executeEdit()
3. executeDelete()
4. executeFind()
5. executeSort()
6. undo()
7. redo()

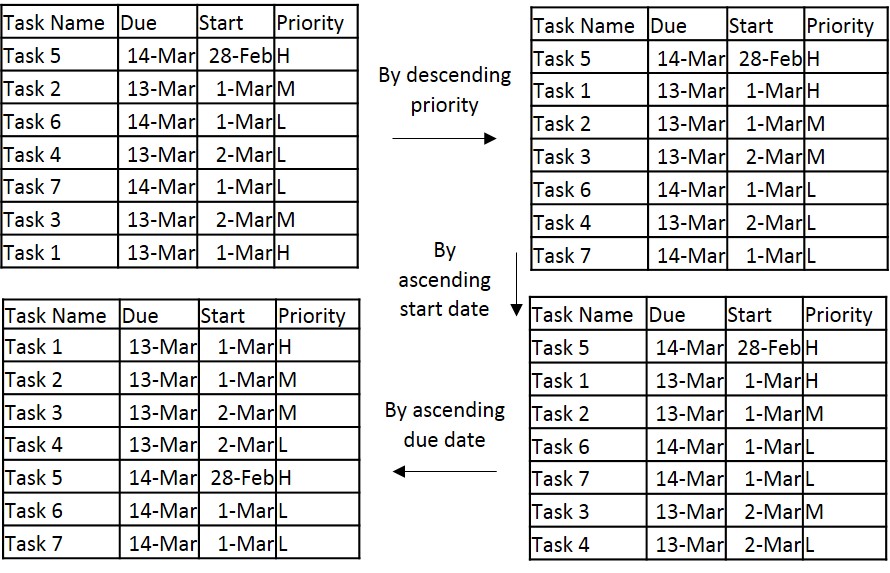
Sequence diagrams illustrating how some of these sub methods work can be found in Appendix \_\_.

### Notable Algorithms

#### Sorting tasks

Bubble sort is used to sort the tasks as it is simple and stable sorting algorithm. A stable algorithm is needed as the relative position of tasks from the previous sort must be preserved when the tasks are sorted again by the next criteria. This is to ensure that the result of a multiple criteria sort is correct sorted at all levels. To sort by multiple criteria, the tasks is sorted by the lowest level criteria first, followed by higher level criteria in the next iterations. The following example illustrates the idea.

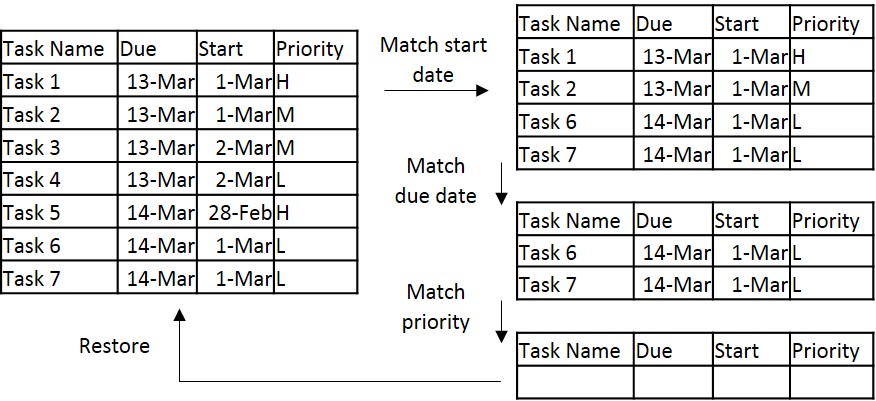
*Command: sort –d A –s A –p D*



*Figure 6. Sorting tasks example*

#### Finding tasks

To find tasks meeting a certain criteria, \_workingList is filtered by each criterion in the order they are keyed in. The result is a \_workingList that contains only the tasks that meet the criteria. If \_workingList is empty (i.e. no tasks found), \_workingList is restored to its previous state. Using the tasks in the previous section, the following example illustrates the idea.

**

*Figure 7. Finding tasks example*

#### Undo & redo

##### \_undoStack and \_redoStack

QLLogic uses stacks to perform undo and redo operations. \_undoStack is used to store previous versions of the working lists and \_redoStack is used to store versions of the working lists that are ahead of the current working lists.

##### How it works

After each add/ edit/ delete/ complete operation, a copy of \_workingList and \_workingListMaster will be pushed onto \_undoStack as a “snapshot” of the state of the lists. Since the working lists contain Tasks which are objects, new Tasks are created with identical attributes as those in the working lists when copying the working lists so that they do not get affected by edit functions when they are in the stack. This is achieved using the copyListsForUndoStack() method in QLLogic. The diagram below illustrates this process.



*Figure 8. State of stacks after executing a command*

When user calls undo, the “current” working lists are be popped out of \_undoStack and pushed into \_redoStack, and the working lists are referenced to the “previous” working lists on top of the \_undoStack. When the user calls redo, the “current” working lists are be popped out of \_redoStack and pushed into \_undoStack, and the working lists are referenced back to them. The diagrams below illustrate these processes.

*Undo operation*



*Redo operation*



*Figure 9. State of stacks after undo and redo command*

## Storage Component

### Class diagram

The Storage component manage the persistency of the list of Task between sessions by utilizing the physical storage. The data stored into the medium is encoded in JSON by utilizing the Gson library. **Figure X** shows the structure of the Storage component and its dependency.



*Figure 10. Storage component class diagram*

## Notable APIs

Figure \_\_ shows some of the notable API of the Storage component.

|  |  |  |
| --- | --- | --- |
| Method | Parameters | Description |
| saveFile(List<Task> taskList, String filepath): void | **taskList**: a List<Task> object containing the list of Task to be saved  **filepath**: the path to the file to store the list | Converts the List<Task> object into a JSON encoded string and write it into the specified file.  Throws Error when fail. |
| loadFile(List<Task> taskList, String filepath): List<Task> | **taskList**: an empty List<Task> object to contain the list of Task loaded  **filepath**: the path to the file to load the list from | Reads the JSON from the specified file and decode it back to Task objects and stores them into taskList which is returned from the method. Returns an empty list if the file does not exist. Throws Error when fail. |

*Table 6. Storage component API*

## Google Integration Component

The Google Integration (GI) component handles the synchronisation of local data and data from the Google Calendar web service. **Figure Y** illustrates the structure of the Google Integration. QLGoogleIntegration handles the logic of the synchronisation process. GoogleLogin does the authentication process and credential handling and GoogleCalendarConnector acts as a connector to the Google Calendar API.



*Figure 11. Google Integration component class diagram*

## API

Figure \_\_ shows some of the notable API of the GI component.

|  |  |  |
| --- | --- | --- |
| Method | Parameters | Description |
| syncTo(List<Task> taskList): void | **taskList**: a List<Task> object containing the list of Task to be synchronise to Google services | Synchronises to Google services from taskList by adding, updating and deleting events on the service.  Throws Error when fail. |
| syncFrom(List<Task> taskList): List<Task> | **taskList**: a List<Task> object to containing the current tasks | Retrieves events from Google services and updates current taskList to include the events.  Throws Error when fail. |

*Table 6. Google Integration component API*

## Testing Methodology

JUnit is the unit testing framework used in this project. When developing new functionalities, unit test can be used to test the outcome is within expectation. Although there is no strict policy on utilizing Test-Driven Development (TDD) approach, sufficient testing on the boundary cases is expected.

The following is a code snippet of a sample unit test.

Task task;

@Before

**public** **void** setUp(){

// initialization before tests

task = **new** Task("task 1");

}

@After

**public** **void** tearDown() {

// clean up after tests

}

@Test

**public** **void** testCase1() {

task.setDueDate("0305"); //DDMM

*assertEquals*(3,

task.getDueDate().get(Calendar.***DAY\_OF\_MONTH***));

*assertEquals*(Calendar.***MAY***,

task.getDueDate().get(Calendar.***MONTH***));

}

The setUp() method is used to initialize the environment for each test cases whereas the tearDown() method is used to clean up after each test cases. The functionality in question can be invoked within the test cases and expected outcome of the functionality should be asserted with the actual values.