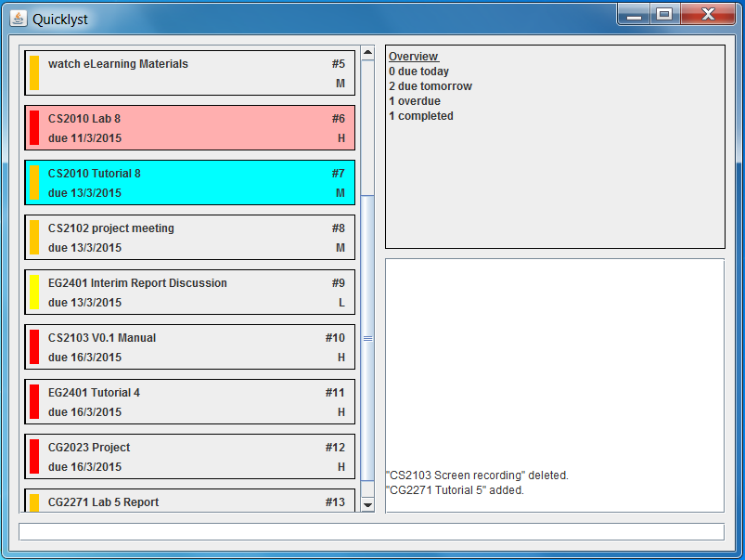
Quicklyst



Supervisor: *Michelle Tan*

Extra feature: *Google Integration*

|  |  |  |
| --- | --- | --- |
| photo  **Shao Fei** Team Leader Documentation Code quality | cheongkeyou  **Cheong Ke You** Team Member Testing Integration | luyanning  **Lu Yanning** Team Member Testing Scheduling |

Contents

[Getting started with Quicklyst 4](#_Toc414222425)

[1. Understanding the User Interface 4](#_Toc414222426)

[1.1. Primary Pane 4](#_Toc414222427)

[1.2. Secondary Pane 4](#_Toc414222428)

[1.3. Feedback Pane 4](#_Toc414222429)

[1.4. Command Line 4](#_Toc414222430)

[2. What is in a Task? 5](#_Toc414222431)

[2.1. Task Name 5](#_Toc414222432)

[2.2. Due Date 5](#_Toc414222433)

[2.3. Task Number 5](#_Toc414222434)

[2.4. Priority Level and Label 5](#_Toc414222435)

[3. How to use the Commands 6](#_Toc414222436)

[3.1. Task Commands 6](#_Toc414222437)

[3.2. Display Commands 7](#_Toc414222438)

[3.3. Other Commands 7](#_Toc414222439)

[Appendix A: Command Examples 9](#_Toc414222440)

[Add 9](#_Toc414222441)

[Edit 9](#_Toc414222442)

[Complete 9](#_Toc414222443)

[Delete 9](#_Toc414222444)

[Find 9](#_Toc414222445)

[Sort 10](#_Toc414222446)

[Quicklyst Developer’s Manual 11](#_Toc414222447)

[1. Architecture 11](#_Toc414222448)

[2. GUI Component 11](#_Toc414222449)

[2.1. User interaction sequence diagram 12](#_Toc414222451)

[3. Logic Component 13](#_Toc414222452)

[3.1. Task Class 13](#_Toc414222453)

[3.2. DateHandler Class 14](#_Toc414222454)

[3.3. CommandParser Class 14](#_Toc414222455)

[3.4. QLLogic Class 15](#_Toc414222456)

[3.5. Notable Algorithms 16](#_Toc414222457)

[4. Storage Component 18](#_Toc414222458)

[4.1. APIs 18](#_Toc414222459)

[5. Google Integration Component 19](#_Toc414222460)

[5.1. APIs 19](#_Toc414222461)

[6. Testing Methodology 20](#_Toc414222462)

[Appendix A- Sequence diagrams for QLLogic 21](#_Toc414222463)

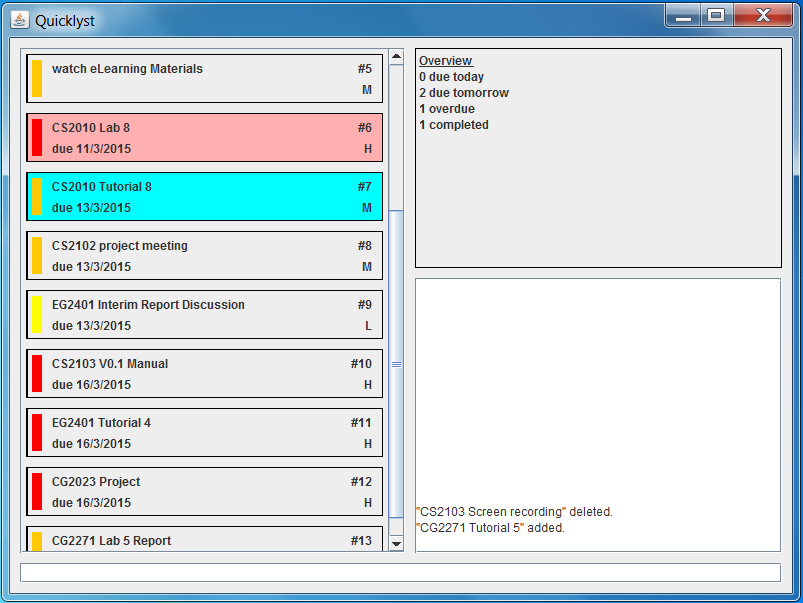
[Adding a task 21](#_Toc414222464)

[Editing a task 22](#_Toc414222465)

[Deleting a Task 23](#_Toc414222466)

# Getting started with Quicklyst

## Understanding the User Interface



Secondary Pane

Primary Pane

Feedback Pane

Command Line

### Primary Pane

In this pane, you will see all the tasks the way you want to see it. Whether is it viewing tasks for a certain day or period, streamlining them into categories or sorting them according to different criteria, Quicklyst offers simple commands to achieve any combinations of the above. You can refer to *Section 3. How to use the Commands* on how exactly to exploit the commands.

The default display when you open Quicklyst is all the uncompleted tasks sorted according to due date in ascending order. If you are using Quicklyst for the first time, this pane will be blank.

### Secondary Pane

In this pane, the default display is the Overview of your tasks shown as the number of tasks due today, due tomorrow, overdue and completed.

If you ever need help using the commands in Quicklyst, the help page will be displayed in this pane upon request. You can refer to *Section 3. How to use the Commands* on how to open the help page.

### Feedback Pane

This pane is where Quicklyst “talks” to you. Here, you can see how Quicklyst has responded to each command you entered.

### Command Line

This is where you type your commands. To execute a command, simply press ENTER after you have finished typing it.

## What is in a Task?



Task Name

Task Number

Due Date

Priority Level

Priority Color



Normal task

Overdue task



Completed task

### Task Name

This is the name of your task the way you entered it when you added the task. Every task must have a name. You can edit the task name any time you want.

### Due Date

This is the date when you need to do your task by/on. You have the option to include the due date when you add the task or go back to edit/add it any time you want. If you did not include a due date in a task, it will not be shown.

### Task Number

This is a numbering system that runs in ascending order down the list in the Primary Pane regardless of the way the tasks are displayed. This is to make things easy when you want to access (e.g. edit, delete, etc.) a task. It is decided by Quicklyst so you cannot change it.

### Priority Level and Label

This is the priority level you give to the task. There are 3 levels of priority- High, Medium and Low. You have to option to include the Priority Level when you add the task or go back to edit/add it any time you want. If you did not tag a Priority Level to a task, it will not be shown and there will be no Priority Label. The Priority Level is also reflected through the Priority Label which takes on Red, Orange, Yellow for High, Medium and Low priority tasks respectively.

## How to use the Commands

You have to type the commands in one line in the Command Line. When you finish typing a command, press ENTER to execute the command.

In this section, each command is explained by Description, Command and Fields. *Words in italics* are user defined and may follow certain formats. Key in only the Fields that you need in any order that you want. Commands are not case-sensitive. Commands enclosed in [ ] are in primitive formats, while those enclosed in { } are in more natural formats. The current version only supports primitive formats and features/ commands in grey will only be available in later versions. You can refer to **Appendix A** for examples of how to use the commands.

### Task Commands

#### Add

Description: Add a task into the list.

Command: ADD/A + *Task Name + Fields*

Fields:

1. Start date: [–s *date/day*[[1]](#footnote-1)] {from/start *date/day*}
2. Due date: [-d *date/day*] {to/due/by/end *date/day*}
3. Priority level: [-p *H/M/L*]{priority *H/M/L*}
4. Reminder: [-r *date/day*] {remind *date/day*}

#### Edit

Description: Edit/add fields of an existing task.

Command: EDIT/E + *Task Number* + *New/Updated* *Fields*

Fields:

1. Name: [-n *name*] {name}
2. Start date: [–s *date/day/CLR*] {from/start *date/day/CLR*}
3. Due date: [-d *date/day/CLR*] {to/due/by/end *date/day/CLR*}
4. Priority level: [-p *H/M/L/CLR*]{priority *H/M/L/CLR*}
5. Reminder: [-r *date/day/CLR*] {remind *date/day/CLR*}

#### Complete

Description: Complete/uncomplete a task. If Y/N is not defined, completed status is simply toggled.

Command: COMPLETE/C + *Task Number* + *Y/N*

#### Delete

Description: Delete a task

Command: DELETE/D + *Task Number*

### Display Commands

#### Find

Description: Find the tasks that fit certain criteria.

Command: FIND/F + *Fields*

Fields:

1. Task name: [-n *name*] {*name*}
2. Start date: [-s + bf/af/on *date/day* ORbtw *range*[[2]](#footnote-2)]

{due/end + before/after/on *date/day* OR between *range*}

1. Due date: [-d + bf/af/on *date/day* ORbtw *range*]

{start + before/after/on *date/day* OR between *range*}

1. Priority level: [-p *H/M/L*]{priority *H/M/L*}
2. Completed: [-c *Y/N*] {completed OR not completed}
3. Overdue: [-o Y/N] {overdue OR not overdue}
4. Show all tasks: ALL

#### Sort

Description: Sort the tasks currently displayed in the Primary Pane according to certain criteria. If more than one field is entered, the tasks will be sorted in the order that the fields are keyed in.

Command: SORT/S + *Fields*

Fields:

1. By due date: [-d *A/D*] {due/end *A/D*}
2. By task duration length: [-l *A/D*] {duration/length *A/D*}
3. By priority level: [-p *A/D*] {priority *A/D*}

### Other Commands

#### Undo

Description: Undo the previous command.

Command: UNDO/U

#### Redo

Description: Redo the previous command.

Command: REDO/R

#### Sync with Google

Description: Synchronise all tasks to or from Google Calendar.

Command: SYNC/SG + to/from

#### Load file

Description: Load tasks from a specific file path.

Command: LOAD/L + file path

#### Save file

Description: Save tasks into a specific file path.

Command: SAVE/S + file path

#### Help

Description: Open/close the Commands Directory in the Secondary Pane

Command: ? (enter ? again to close the Commands Directory)

## 

## Appendix A: Command Examples

### Add

1. Command: A Task 1 –d 1608 –p H  
   Result: Added *“Task 1”, due on 16 Aug, High priority*
2. Command: ADD Task 2 –p L

Result: Added *“Task 2”, Low priority*

1. Command: ADD Task 3

Result: Added *“Task 3”*

1. Command: a Task 4 –s TDY –d TMR

Result: Added *“Task 4”, starts today, due tomorrow*

1. Command: a Task 5 from 1303 to 13022016 priority L remind 3112

Result: Added *“Task 5”, starts on 13 Mar, due on 13 Feb 2016, Low priority, remind on 31 Dec*

### Edit

Examples are independent of each other.

*Original Task: #3, “Task 1”, due on 16 Aug 2015, High priority*

1. Command: E 3 –n Task 2 –d 1708 –p L

Result: *#3, “Task 2”, due on 17 Aug, Low priority*

1. Command: EDIT 3 –d 1708

Result: *#3, “Task 1”, due on 17 Aug, High Priority*

1. Command: EDIT 3 –d CLR

Result: *#3, “Task 1”*

1. Command: e 3 start 1303 due 13022016 priority CLR

Result: *#3, “Task 1”, starts on 13 Mar, due on 13 Feb 2016*

### Complete

1. Command: C 3Result:
   1. Task #3(uncompleted) that is currently displayed on the list in the Primary Pane is marked as completed
   2. Task #3 (completed) that is currently displayed on the list in the Primary Pane is marked as incomplete
2. Command: Complete 3 N

Result: Task #3 is marked as incomplete regardless of its current status

1. Command: Complete 3 Y

Result: Task #3 is marked as complete regardless of its current status

### Delete

Command: D 3Result: Task #3that is currently displayed on the list in the Primary Pane is deleted

### Find

1. Command: FIND –n task one

Result: Find all tasks that contain the word “task” and “one”, with closer match listed at the top

1. Command: FIND –o YResult: List all tasks that are overdue
2. Command: FIND –d on TDY –c YResult: List all tasks that are due today and completed
3. Command: F –p H –d btw TMR:1608 –c NResult: List all tasks that are High priority, due between tomorrow and 16 Aug and not completed
4. Command: FIND –s bf 1608 –d af 1709

Result: List all tasks that start before 16 Aug and is due after 17 Sep

1. Command: f start after 1608 due before 1509 not completed

Result: List all tasks that start after 16 Aug and due before 15 Sep that are not completed

1. Command find start from TDY to TMR due from 1409 to 1609 priority H

Result: List all tasks that start between today and tomorrow, and is due between 14 Sep and 16 Sep, that are High priority

1. Command: find all

Result: List all tasks

### Sort

1. Command: S –p A –d D –l AResult: Sort tasks in primary pane by Priority Level in ascending order, then by Due Date in descending order, then by task duration length in ascending order

# Quicklyst Developer’s Manual

## 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 by the Logic component to carry out the user’s commands. The Logic component implements the different functionalities of through the help of a few sub-components. Finally the Storage, Settings and Google Integration components are at the lowest level and allows data to be loaded and stored. *Figure 1* illustrates the architecture of Quicklyst.



*Figure 1. Quicklyst architecture*

### Task Class

Before going into the components of Quicklyst, the Task object shall be introduced as the central theme of Quicklyst is handling task.

The Task object has attributes of a task in real life and are passed are passed among the various components of the Quicklyst architecture to realise its different functionalities. The class diagram of Task and its notable API is shown in *Figure 1.* Typical instance methods such as accessors and modifiers are omitted for conciseness.

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

*Figure 1. Task Class Diagram (Left) and API (Right)*

## GUI Component

The Graphical User Interface (GUI) provides an interactive and visual indication for the user. By passing the command entered by the user to the Logic component, GUI will receive a list of tasks in return and update the three main field- task list, overview and feedback accordingly. *Figure 2* shows the class diagram of the GUI component.



*Figure 2. GUI component class diagram*

### User interaction sequence diagram

A sequence diagram shown in *Figure 3* demonstrates some examples of the interaction between the user and the GUI. Note that some methods are not shown to improve clarity of the sequence diagram.



*Figure 3.* *Use case sequence diagram*

## Logic Component

The Logic component processes and executes 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.

The Logic Component implements the Command Pattern, where an “Action Generator” creates a concrete “Action” type and passes it to the “Executer” for it to execute as a general “Action” type. This is advantages as Quicklyst provides many functionalities which need to be executed in different ways, and the Command Pattern enable Logic to execute them as a single general “Action”. This reduces coupling as the “Executor” only need to be associated with one general “Action” object instead of many concrete “Action” objects. *Figure 4* shows the relationship between the sub-components of the Logic Component under the Command Pattern.



*Figure 4. Logic component class diagram[[3]](#footnote-3)*

#### The Executor

The Manager subcomponent manages Task objects by passing the command to Action Generator and executing the Actions generated. It is made up of QLLogic Class and HistoryManager Class.

##### Logic Class

QLLogic Class is the single point of access between the subcomponents of Logic and other components of Quicklyst. This applies the Façade Pattern where the Logic component can be accessed without exposing its details. Table \_\_\_ shows some of the Notable API of the Logic class.

|  |  |  |
| --- | --- | --- |
| Method | Parameters | Description |
| setup(StringBuilder: feedback) | **feedback**: the feedback to be displayed to the user after each operation. An empty StringBuilder should be passed in during each call of the method. | Sets up the working environment of QLLogic and loads the list of saved Tasks. |
| executeCommand(  String: command, StringBuilder: feedback) | **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 the method. | Executes the commands specified by the user. |
| getDisplayList(): LinkedList<Task> |  | Returns the filtered list of Tasks that the users want to see, based on the result of the executeCommand(…) API. |
| getFullList(): LinkedList<Task> |  | Returns the unfiltered list of all Tasks in the current session. This includes all tasks that are last loaded, newly added and not deleted. |
| getDeletedList(): LinkedList<String> |  | Returns a list of Google IDs of the Tasks that are deleted in the current session. |
| shouldShowAllCompleted(): boolean |  | Returns true if users wants to see all completed Tasks, returns falls if otherwise. |

*Table 4. Logic class API*

##### HistoryManager Class

History Manager Class carries out the Undo and Redo functions. It uses an *undoStack* to store the previous “states” *(*attributes such as *\_fullList, \_displayList, \_deletedList*, etc) of QLLogic and *redoStack* to store the “states” that are ahead of the current state. The algorithm of the undo redo process can be found in Appendix\_\_\_\_

|  |  |  |
| --- | --- | --- |
| Method | Parameters | Description |
| updateUndoStack(LinkedList<Task>: displayList, LinkedList<Task> masterList, LinkedList<String> deletedList, boolean: shouldShowAllCompleted) | **displayList**: *\_displayList* in QLLogic  **masterList:** *\_masterList* in QLLogic  **deletedList:** *\_deletedList* in QLLogic  **shouldShowAllCompleted:** *\_shouldShowAllCompleted* in QLLogic | Make current state of QLLogic available for undo in the future. |
| undo(StringBuilder: feedback) | **feedback**: the feedback to be displayed to the user after each operation. An empty StringBuilder should be passed in during each call of the method. | Set ups the previous state of QLLogic |
| redo(StringBuilder: feedback) | **feedback**: the feedback to be displayed to the user after each operation. An empty StringBuilder should be passed in during each call of the method. | Sets ups the state of QLLogic ahead of the current state. |
| getDisplayList(): LinkedList<Task> |  | Returns the \_displayList in the previous state. |
| getMasterList(): LinkedList<Task> |  | Returns the \_masterList in the previous state. |
| getDeletedList(): LinkedList<String> |  | Returns the \_deletedList in the previous state. |
| getShouldShowAllCompleted(): boolean |  | Returns \_shouldShowAllCompleted status in the previous state. |

*Figure 4. Logic component class diagram*

#### The Action Generator

The Action Generator sub-component parses commands and generates the appropriate Action Class. It is made up of the CommandParser, FieldParser and DateParser class.

Every time QLLogic needs to execute a command, it creates a CommandParser object and the parsing of command is done in the constructor. Depending on the nature of the command, the FieldParser and Date Parser objects may be created to aid the parsing process. The Activity Diagram in Figure\_\_\_ shows how the CommandParser class parses commands and generate Actions.



Notable APIs of the CommandParser, FieldParser and DateParser classes are shown in *Table 3*, *Table 3* and *Table 3* respectively.

|  |  |  |
| --- | --- | --- |
| Method | Parameters | Description |
| getAction(): Action |  | Returns a concrete Action type that corresponds to the command |
| getFeedback(): StringBuilder |  | Returns the user feedback of the parsing process. |

*Table 3. CommandParser class API*

|  |  |  |
| --- | --- | --- |
| Method | Parameters | Description |
| getField(): Field |  | Returns a Field that corresponds to the fields in the command |
| getFeedback(): StringBuilder |  | Returns the user feedback of the parsing process. |

*Table 3. FieldParser class API*

|  |  |  |
| --- | --- | --- |
| Method | Parameters | Description |
| getDateTime(): Calendar |  | Returns a Calendar that corresponds to the date in the command |
| getFeedback(): StringBuilder |  | Returns the user feedback of the parsing process. |

*Table 3. DateParser class API*

#### The Actions

The Actions sub-component is where the actual execution of commands takes place. It consists the Action abstract class and the various subclasses that extends the Action class. The Action subclasses implement the different features of Quicklyst and new subclasses can be created to provide more features. Thus the Actions sub-component applies the Open-close principle, as it allows the Executor component to extent its functionalities without the need to modify itself, by creating new subclasses that extends Action.

Actions object can also hold other Action objects in order to achieve multiple Actions execution with one execute call. The Class Diagram depicting the relationships between all the Action subclasses are shown in *Figure \_\_\_.* Notable algorithms of some of the subclasses of Action can be found in Appendi\_\_\_.



*Figure 4. Actions sub-component class diagram*

#### Interactions within the logic component

To illustrate how the different sub-components of Logic component working together to execute a typical command, figure \_\_\_ is a sequence diagram depicting a typical add command.

*Figure 4. Sequence diagram within Logic component for a typical add command*

## **Storage** Component

The Storage uses the Singleton pattern to ensure there is only one instance of the component. It maintain the persistency of the user data between sessions by utilizing the physical storage. The data stored into the medium is encoded in JSON by utilizing the Gson library. The class diagram in *Figure X* shows the structure of the Storage component and its dependency.



*Figure X. Storage component class diagram*

*Table X* shows some of the notable API of the Storage component.

|  |  |  |
| --- | --- | --- |
| Method | Parameters | Description |
| isValidFile(String filePath): boolean | **filePath**: the path to the file to be checked. | Returns if the filePath is valid to be used. Does not guarantee the success of saveFile and loadFile. |
| saveFile(List<Task> taskList, List<String> deletedIDs, String filePath): void | **taskList**: the list of Task to be saved  **deletedIDs**: the list of Task to be saved  **filePath**: the path to the file to store the list | Encodes taskList and deletedIDs into JSON and write it into the specified file. Throws Error when fail. |
| loadFile(List<Task> taskList, List<String> deletedIDs, String filePath): void | **taskList**: a list to contain the list of Task loaded  **deletedIDs**: a list to contain the list of Task loaded  **filePath**: the path to the file to load the list from | Decode JSON from the specified file. The decoded objects will be stored into taskList and deletedIDs respectively. Throws Error when fail. |

*Table X. Storage class API*

## Google Integration Component

The Google Integration (GI) component is based on Façade pattern. GoogleIntegration abstracts the internal details from other components to reduce coupling with the internal components. It handles the synchronisation of local data and data from the Google Calendar and Tasks web service. The class diagram in *Figure X* illustrates the structure of the GI component.



*Figure X. Google Integration component class diagram*

*Table X* shows some of the notable API of the GoogleIntegration component.

|  |  |  |
| --- | --- | --- |
| Method | Parameters | Description |
| sync(List<Task> taskList, List<String> deletedIDs): void | **taskList**: the list of Task to be synchronise to Google services  **deletedIDs**: the list of IDs deleted locally | Synchronises with Google services.  Throws Error when fail. |
| logout(): boolean |  | Delete Google credentials. Returns true if the credentials are deleted successfully. Returns false if credentials does not exist or the deletion fails. |

*Table X. GoogleIntegration class API*

The synchronisation process with Google Calendar is illustrated in *Figure X*. A similar process is also used for synchronisation with Google Tasks.



*Figure X. Google Calendar Synchronisation activity diagram*

## Settings Component

The Settings component is responsible for maintaining and storing of the application settings. Its design follows the law of Demeter to lower the dependency. *Figure X* illustrates the class diagram of the component.



*Figure X. Settings component class diagram*

## Testing Methodology

JUnit is the unit testing framework used in this project. When developing new functionalities, unit test can be used to test whether 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. Apart from testing for cases that users enter a correct command, the cases where the commands are invalid should also be tested to ensure invalid user commands do not result in failure of Quicklyst.

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

LinkedList<Task> testList;

StringBuilder feedback;

SimpleDateFormat sdf;

@Before

**public** **void** setup() {

QLLogic.*setupStub*();

feedback = **new** StringBuilder();

sdf = **new** SimpleDateFormat("dd.MM.yyyy");

}

@After

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

}

@Test

**public** **void** testExecuteAdd() {

// success case

testList = QLLogic.*executeCommand*(feedback,

"add task one -p L -d 2202 ");

*assertEquals*(feedback.toString(),

"task one added. Priority level updated. Due date updated");

*assertEquals*(testList.peekLast().getName(),

"task one");

*assertEquals*(testList.peekLast().getPriority(),

"L");

*assertEquals*(sdf.format(testList.peekLast().getDueDate().getTime()),

"22.02.2015");

// failure case

testList = QLLogic.*executeCommand*("add", feedback);

*assertEquals*(feedback.toString(),

"Invalid task name entered. Nothing is executed.");

}

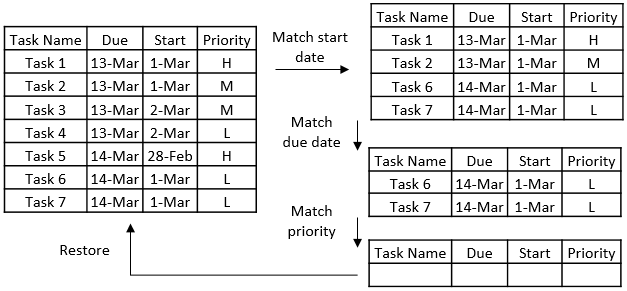
The *setUp()* method is used to initialize the environment for each test cases whereas the *tearDown()* method is used to clean up after each test cases. The functionality in question can be invoked within the test cases and expected outcome of the functionality should be asserted with the actual values.

### Notable Algorithms

#### 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) at the end, *\_workingList* is restored to its unfiltered state. The example in *Figure 6* illustrates the idea.

*Command: find start on 1/3 due on 14/3 priority H*

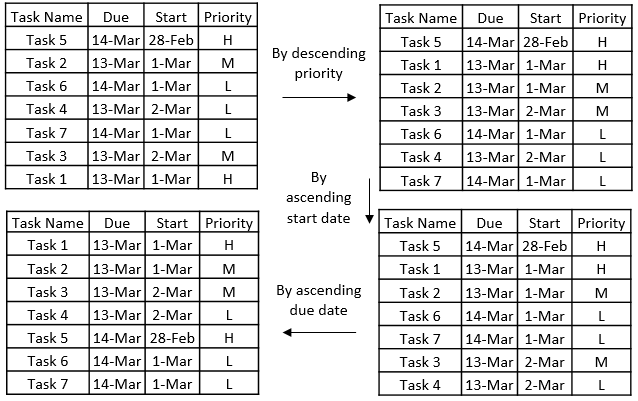
**

*Figure 6. Finding tasks example*

#### Sorting tasks

Bubble sort is used to sort the tasks as it is a 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 correctly 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 example in *Figure 7* illustrates the idea.

*Command: sort –d A –s A –p D*



*Figure 7. Sorting tasks example*

##### How it works

After each change in “state”, QLLogic will call updateUndoStack() and the new “state” will be cloned and pushed onto *\_undoStack* as a “snapshot” of the state of the lists. When cloning the Task lists, new Tasks objects are created with identical attributes so that they do not get affected by edit functions when they are in the stack. This is achieved using the clone() method in Task Class. *Figure 8* illustrates this process.



*Figure 8. Updating of undoStack after executing a command*

When QLLogic calls undo(), the current “state” is popped out of *undoStack* and pushed into *redoStack*, and the previous state now at the top of the *undoStack* will be return to QLLogic. When the QLLogic calls redo(), the “state” that is ahead of the current “state” is popped out of *redoStack*, pushed into *undoStack*, and retuned to QLLogic. *Figure 9* illustrates these processes.

*Redo operation*

*Undo operation*



*Figure 9. Behaviour of stacks during undo and redo*

## Appendix A- Sequence diagrams for QLLogic

Note: Some methods are not shown to improve clarity of the sequence diagram.

### Adding a task



### Editing a task

### Deleting a Task

1. Date can be represented in the format DDMM (for current year) and DDMMYYYY.

   Day can be represented by TDY (today), TMR (tomorrow) and MON – FRI (for current week, if passed then next for next week) [↑](#footnote-ref-1)
2. Range refers to a range of date, it can be represented in the primitive format *date/day*:*date/day* and the natural format from *date/day* to *date/day* [↑](#footnote-ref-2)
3. Some of the relationships in Actions sub-component are omitted for clarity. They will be elaborated in later sections [↑](#footnote-ref-3)