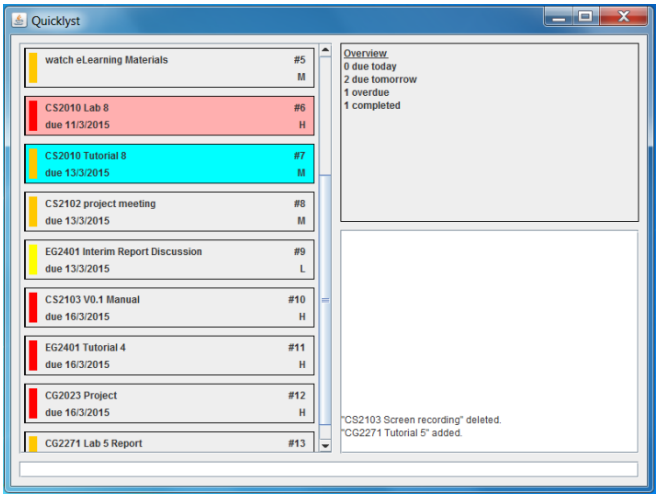
Quicklyst



Supervisor: *Michelle Tan*

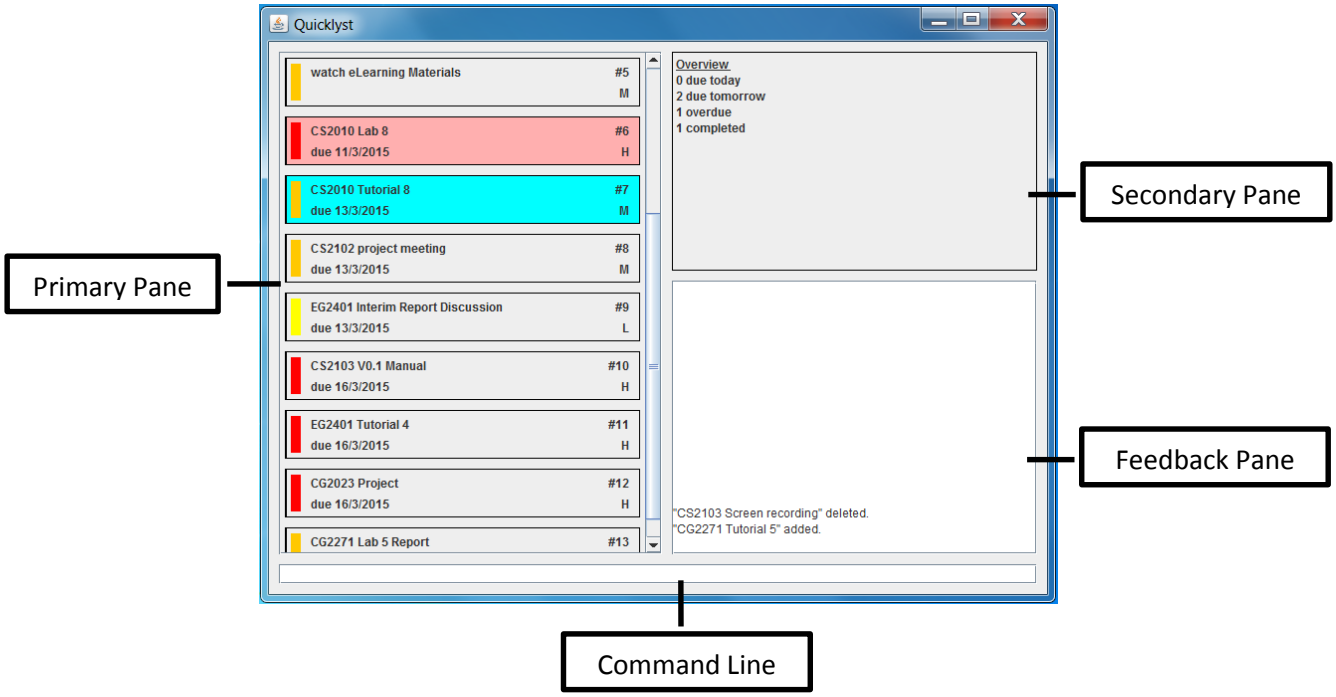
Extra feature: *Google Integration*

		
Shao Fei Team Leader Documentation Code quality	**Cheong Ke You** Team Member Testing Integration	**Lu Yanning** Team Member Testing Scheduling

Contents

Getting started with Quicklyst

1. Understanding the User Interface



1.1. 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.

1.2. 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.

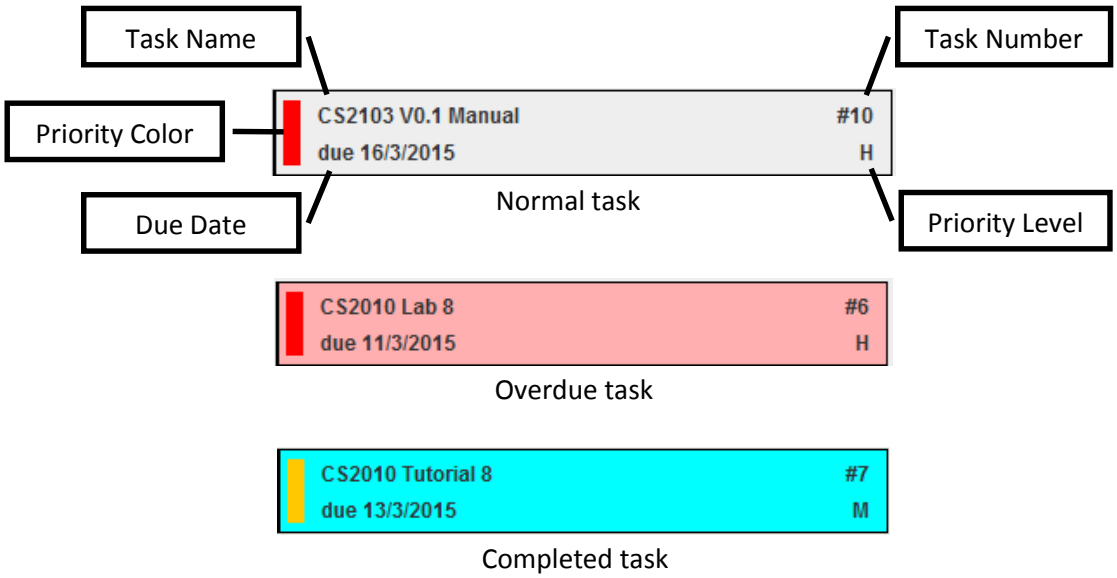
1.3. Feedback Pane

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

1.4. Command Line

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

2. What is in a Task?



2.1. 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.

2.2. 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.

2.3. 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.

2.4. 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.

3. 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.

3.1.Task Commands

3.1.1. Add

Description: Add a task into the list.

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

Fields:

1. Start date: [`-s date/day`[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`}

3.1.2. 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`}

3.1.3. Complete

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

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

3.1.4. Delete

Description: Delete a task

Command: `DELETE/D + Task Number`

[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)

3.2. Display Commands

3.2.1. 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` OR `btw range`[2]] {`due/end + before/after/on date/day` OR `between range`}
3. Due date: [`-d + bf/af/on date/day` OR `btw range`] {`start + before/after/on date/day` OR `between range`}
4. Priority level: [`-p H/M/L`] {`priority H/M/L`}
5. Completed: [`-c Y/N`] {`completed` OR `not completed`}
6. Overdue: [`-o Y/N`] {`overdue` OR `not overdue`}
7. Show all tasks: `ALL`

3.2.2. 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`}

3.3. Other Commands

3.3.1. Undo

Description: Undo the previous command.

Command: `UNDO/U`

3.3.2. Redo

Description: Redo the previous command.

Command: `REDO/R`

3.3.3. Sync with Google

Description: Synchronise all tasks to or from Google Calendar.

Command: `SYNC/SG + to/from`

3.3.4. Load file

Description: Load tasks from a specific file path.

[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`

Command: `LOAD/L + file path`

3.3.5. Save file

Description: Save tasks into a specific file path.

Command: `SAVE/S + file path`

3.3.6. 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*
3. Command: `ADD Task 3`
   Result: Added *"Task 3"*
4. Command: `a Task 4 -s TDY -d TMR`
   Result: Added *"Task 4", starts today, due tomorrow*
5. 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*
2. Command: `EDIT 3 -d 1708`
   Result: *#3, "Task 1", due on 17 Aug, High Priority*
3. Command: `EDIT 3 -d CLR`
   Result: *#3, "Task 1"*
4. 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 3`
   Result:
   a. Task #3 (uncompleted) that is currently displayed on the list in the Primary Pane is marked as completed
   b. 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
3. Command: `Complete 3 Y`
   Result: Task #3 is marked as complete regardless of its current status

Delete

Command: `D 3`
Result: Task #3 that 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
2. Command: `FIND -o Y`
   Result: List all tasks that are overdue

3. Command: `FIND -d on TDY -c Y`
   Result: List all tasks that are due today and completed
4. Command: `F -p H -d btw TMR:1608 -c N`
   Result: List all tasks that are High priority, due between tomorrow and 16 Aug and not completed
5. Command: `FIND -s bf 1608 -d af 1709`
   Result: List all tasks that start before 16 Aug and is due after 17 Sep
6. 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
7. 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
8. Command: `find all`
   Result: List all tasks

Sort

1. Command: `S -p A -d D -l A`
   Result: 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

1. 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 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.

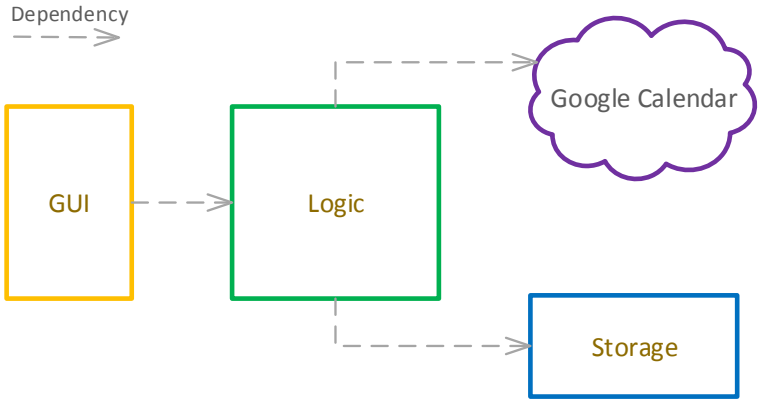


Figure 1. Quicklyst architecture

2. 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.

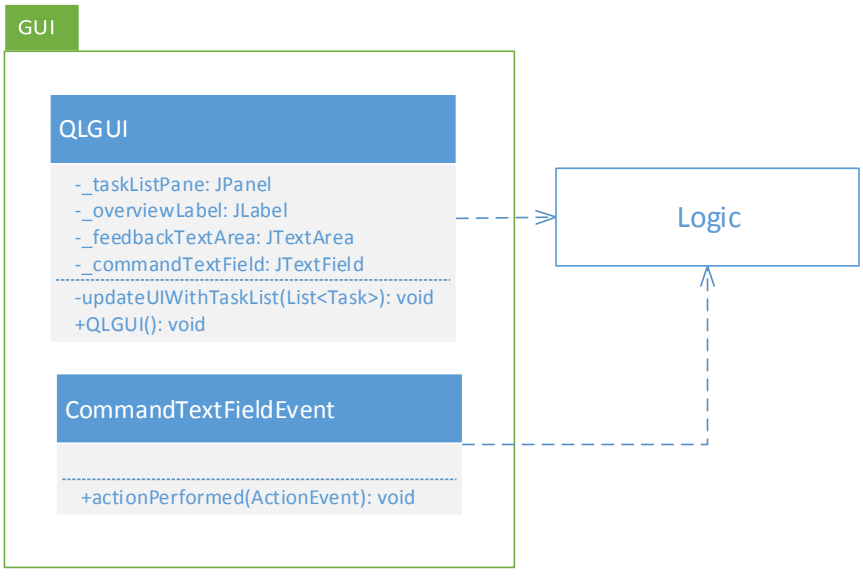


Figure 2. GUI component class diagram

2.1.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.

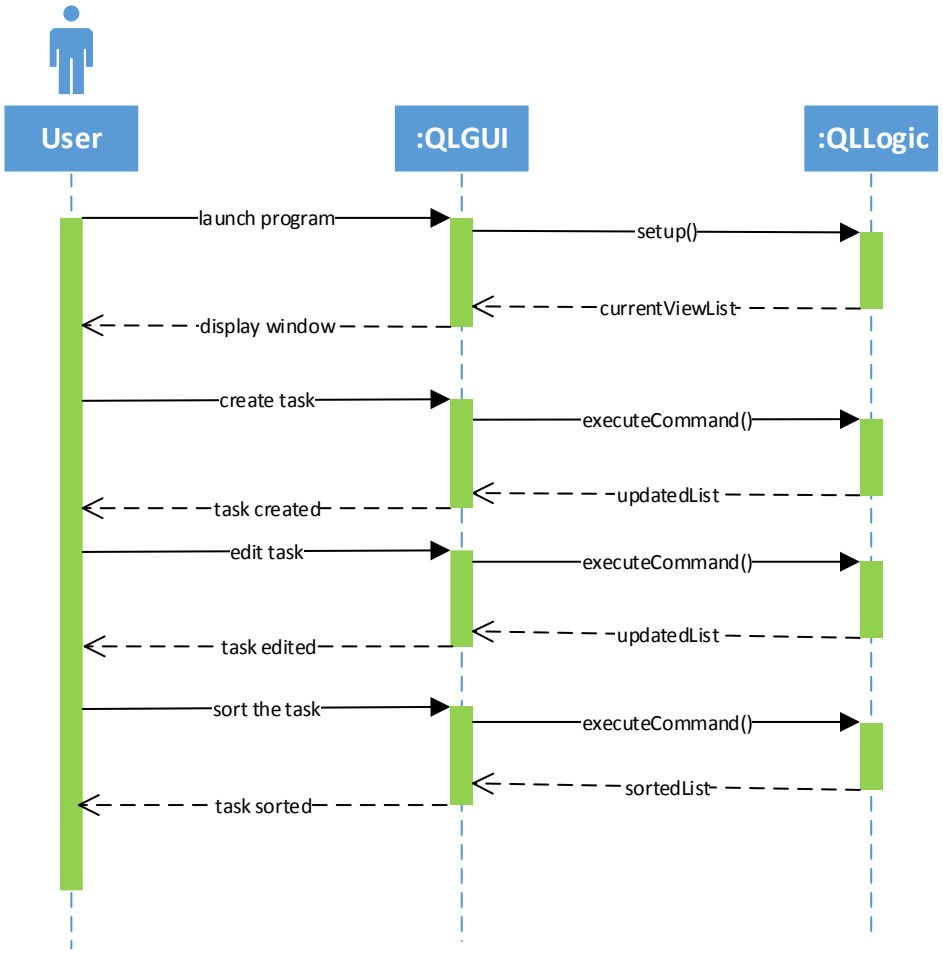


Figure 3. Use case sequence diagram

3. 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. It consists of three sub-components- QLLogic, CommandParser and DateHandler, and handles Task objects. The class diagram in *Figure 4* shows the relationship between the classes that are relevant to the Logic component.

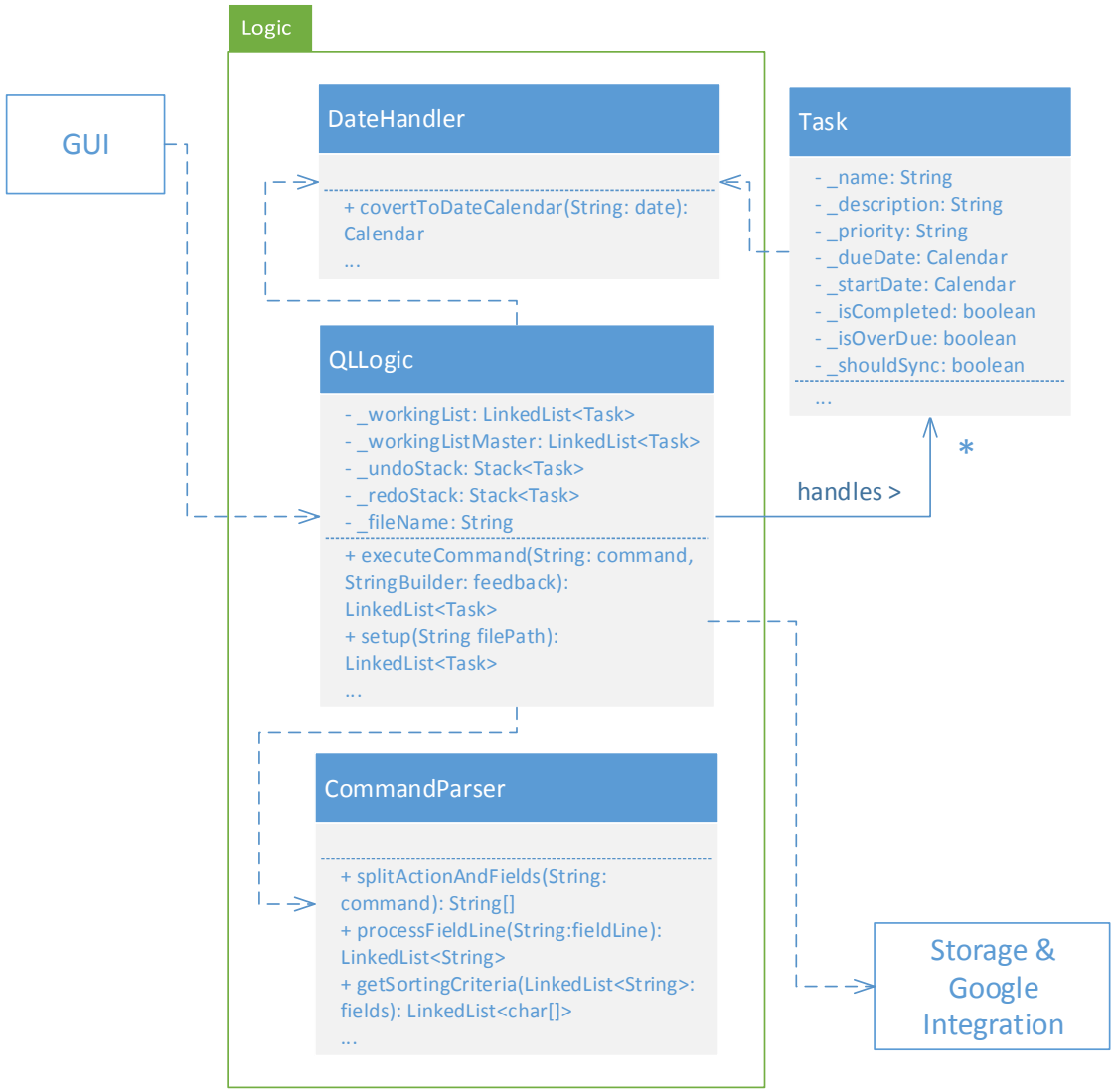


Figure 4. Logic component class diagram

3.1. 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 for conciseness, and only the notable API is shown in *Table 1*.

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

Table 1. Task class API

3.2. 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 in *Table 2*.

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

Table 2. DateHandler class API

3.3. 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 in *Table 3*.

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.
extractTaskName(String: fieldLine): String	**fieldLine**: a string that may contain a task name.	Extracts and returns a potential task name.
extractTaskNumber(String: fieldLine): String	**fieldLine**: a string that may contain a task number.	Extracts and returns a potential task number in String format.
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 3. CommandParser class API

3.4. 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 *Table 4*.

Method	Parameters	Description
setup(String: filepath): LinkedList<Task>	**filepath**: the path of the file that contains a previously saved list of Tasks.	Sets up the working environment of QLLogic and loads the list of 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 the method.	Executes the commands specified by the user. Returns new *_workingList* that satisfies the requirements of this command. Edits feedback accordingly to correspond to the result of the execution.

Table 4. QLLogic class API

3.4.1. _workingList and _workingListMaster

QLLogic holds LinkedList<Tasks> objects 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 5* illustrates the relationship between the working lists.

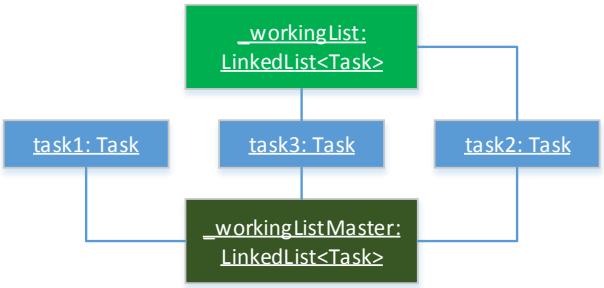


Figure 5. _workingList & _workingListMaster object diagram

As seen from the object diagram, *_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.

3.4.2. 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 A**.

3.5.Notable Algorithms

3.5.1. 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 -s 0103 -d 1403 -p H`

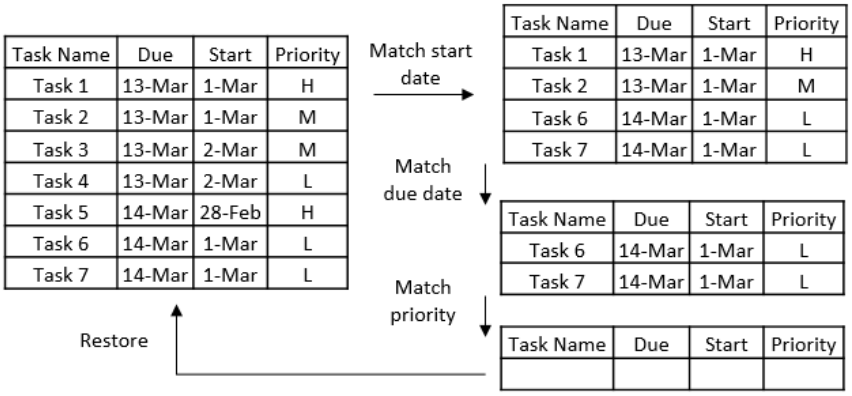


Figure 6. Finding tasks example

3.5.2. 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`

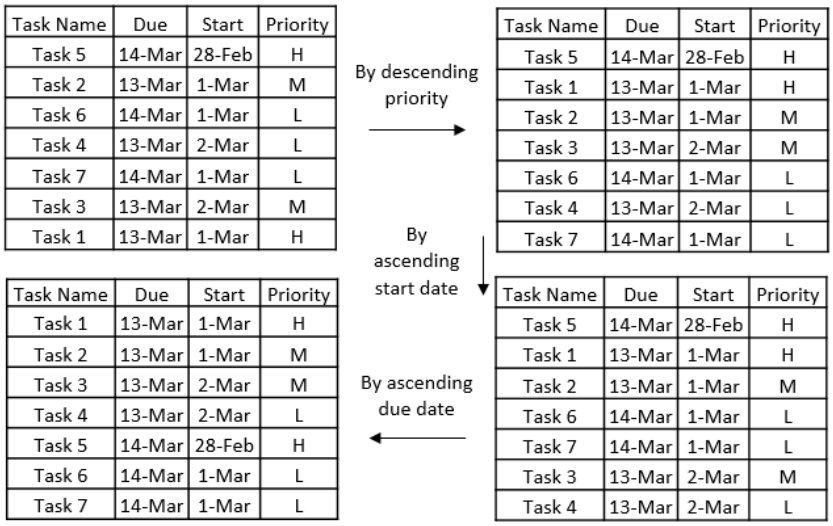


Figure 7. Sorting tasks example

3.5.3. 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 currently displayed working lists.

How it works

After each *add/ edit/ delete/ complete* operation, *_workingList* and *_workingListMaster* will be duplicated pushed onto *_undoStack* as a "snapshot" of the state of the lists. Since the working lists contain Tasks which are objects, new Tasks objects are created with identical attributes as those in the working lists when duplicating 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, which uses the clone() method in Task to create new identical Tasks. *Figure 8* illustrates this process.

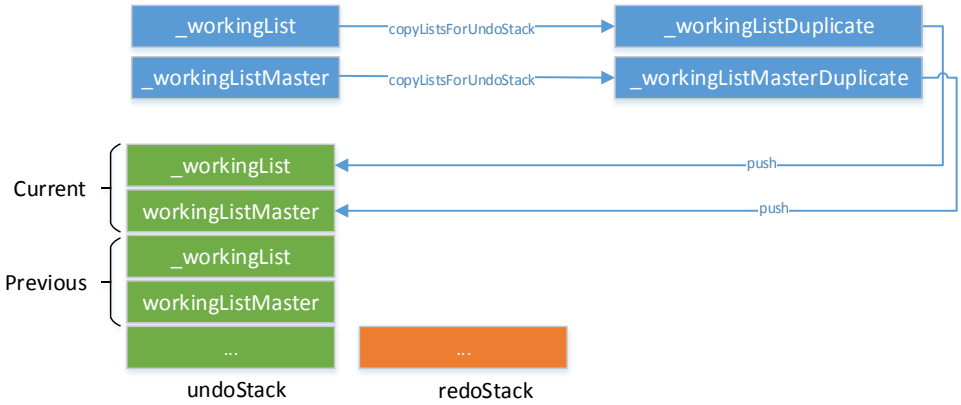


Figure 8. Updating of undoStack after executing a command

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 popped out of *_redoStack* and pushed into *_undoStack*, and the working lists are referenced back to them. *Figure 9* illustrates these processes.

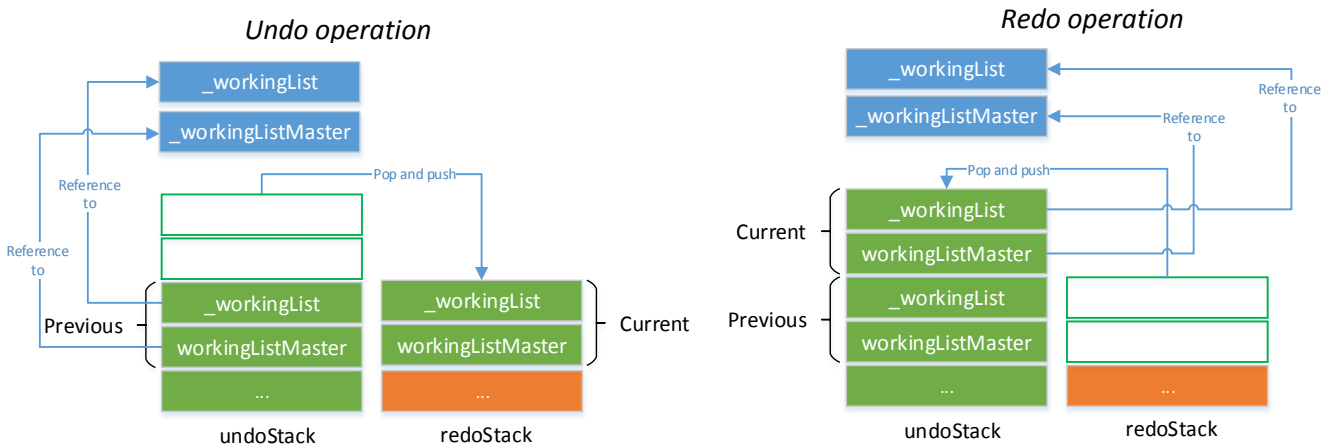


Figure 9. Behaviour of stacks during undo and redo

4. Storage Component

The Storage component manages 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. The class diagram in *Figure 10* shows the structure of the Storage component and its dependency.

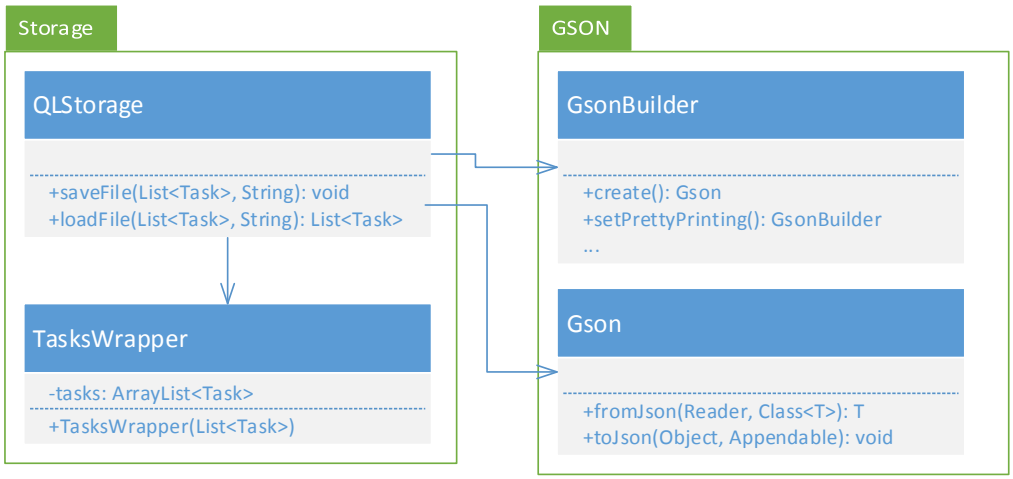


Figure 10. Storage component class diagram

4.1. APIs

Table 5 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. Stores the decoded Task objects into *taskList* and returns *taskList*. Returns an empty list if the file does not exist. Throws Error when fail.

Table 5. Storage component API

5. Google Integration Component

The Google Integration (GI) component handles the synchronisation of local data and data from the Google Calendar web service. The class diagram in *Figure 11* illustrates the structure of the GI component. 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.

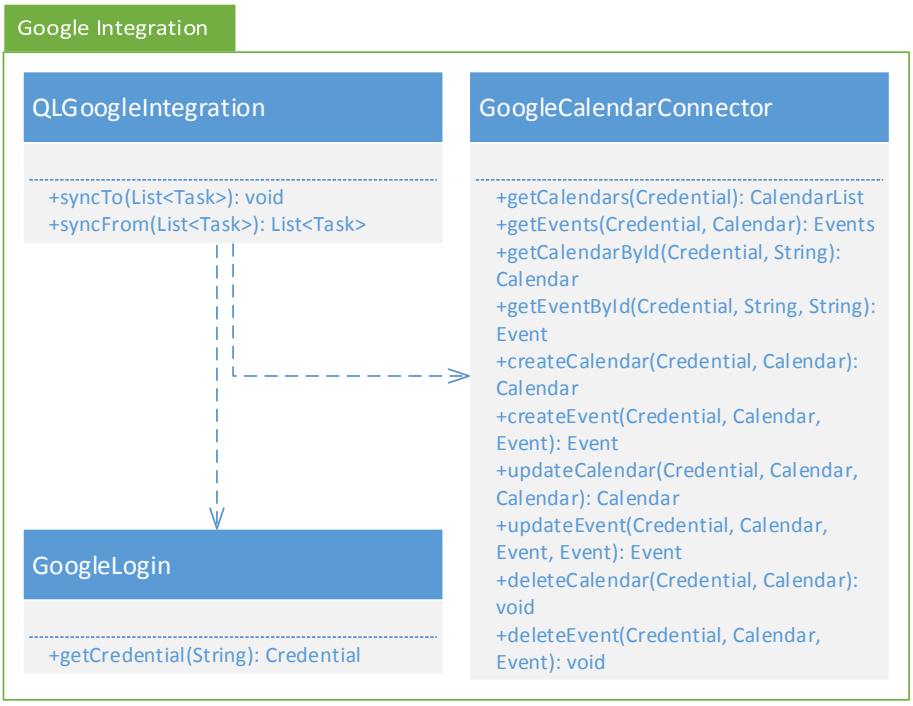


Figure 11. Google Integration component class diagram

5.1. APIs

Table 6 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

6. 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.

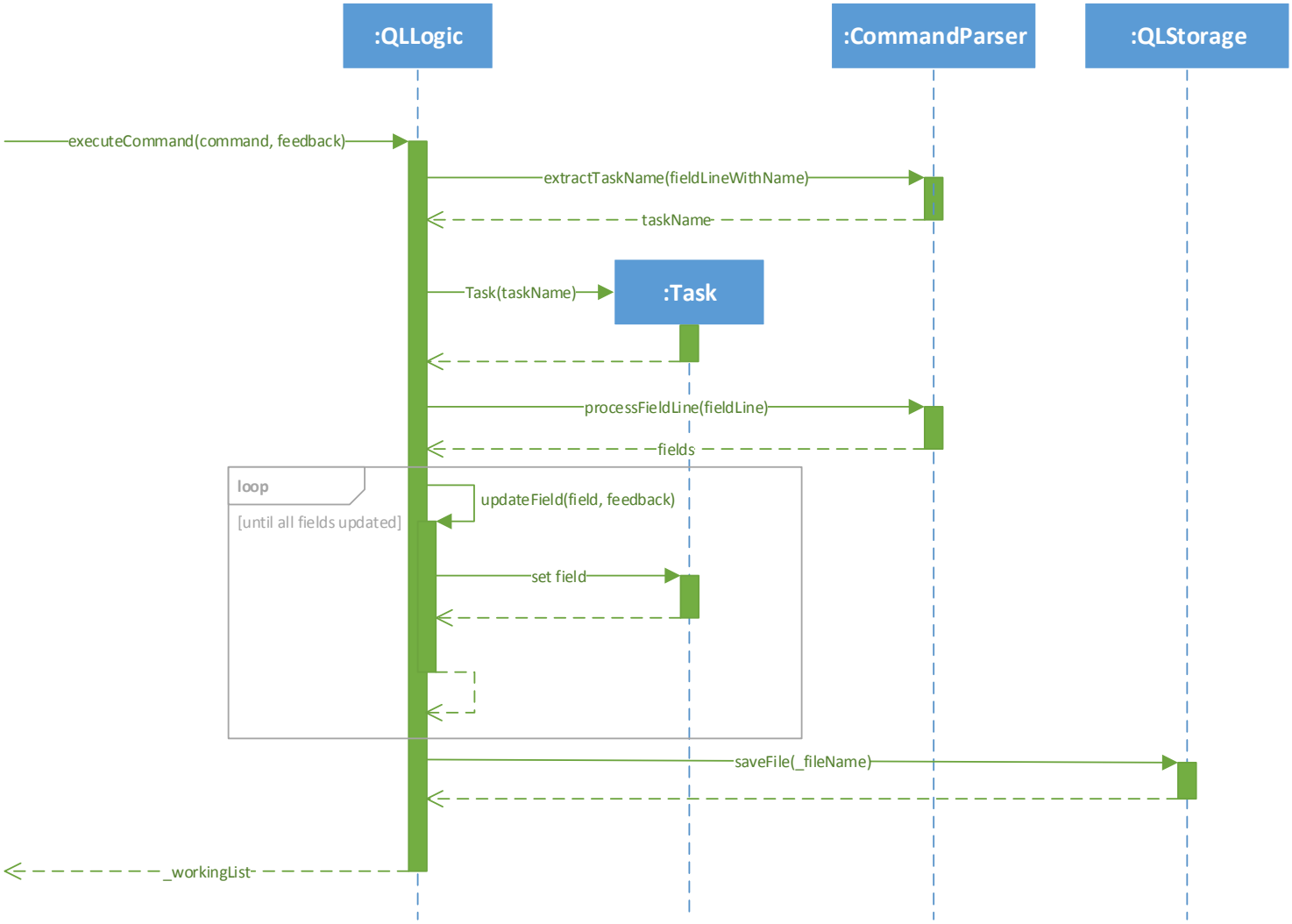
```
LinkedList<Task> testList;
StringBuilder feedback;
SimpleDateFormat sdf;

@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.

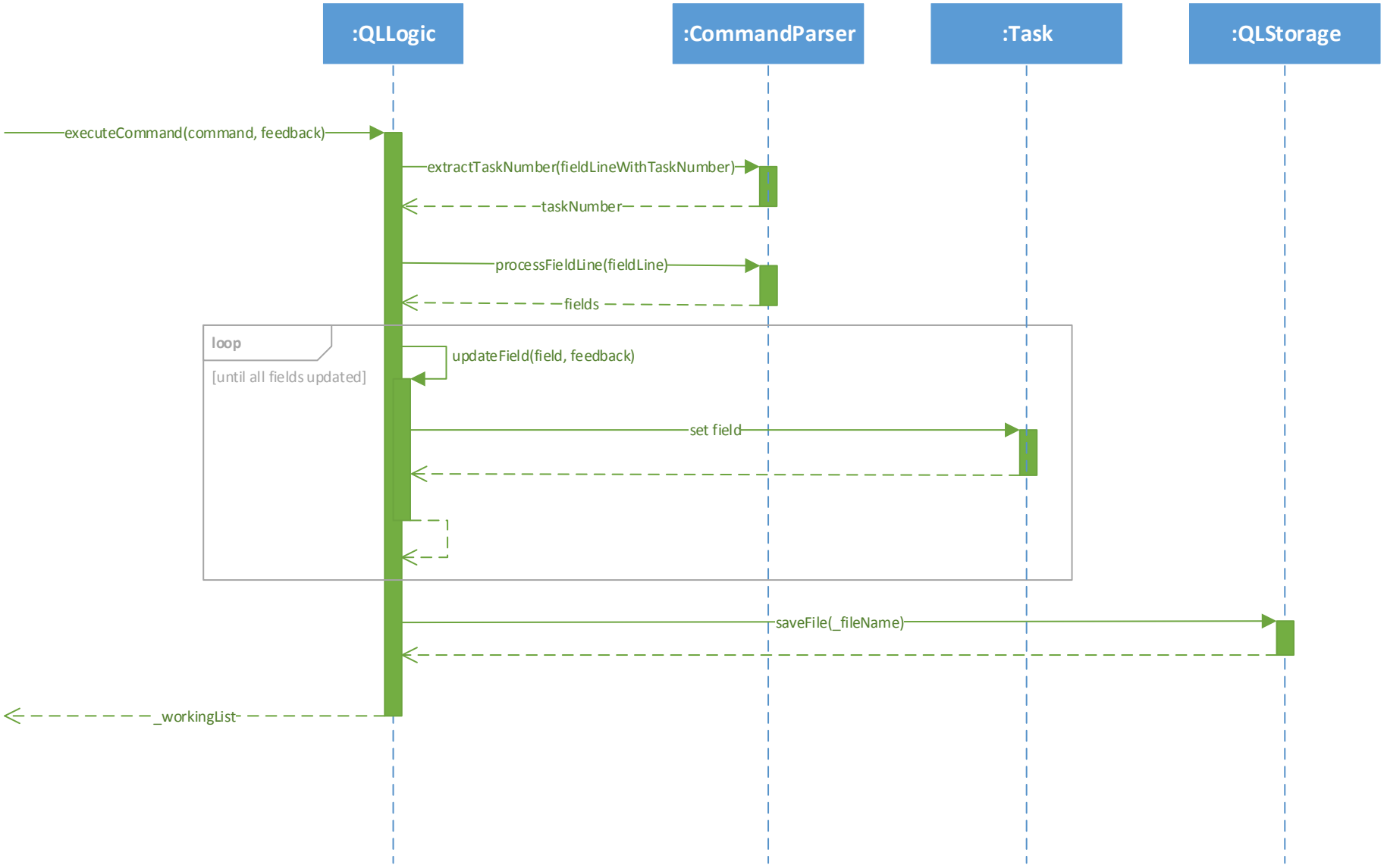
Appendix A- Sequence diagrams for QLLogic

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

Adding a task



Editing a task



Deleting a Task