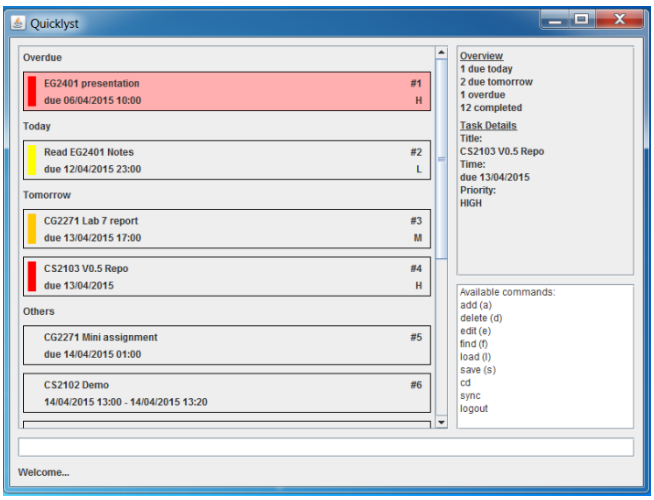
Quicklyst



Supervisor: *Michelle Tan*

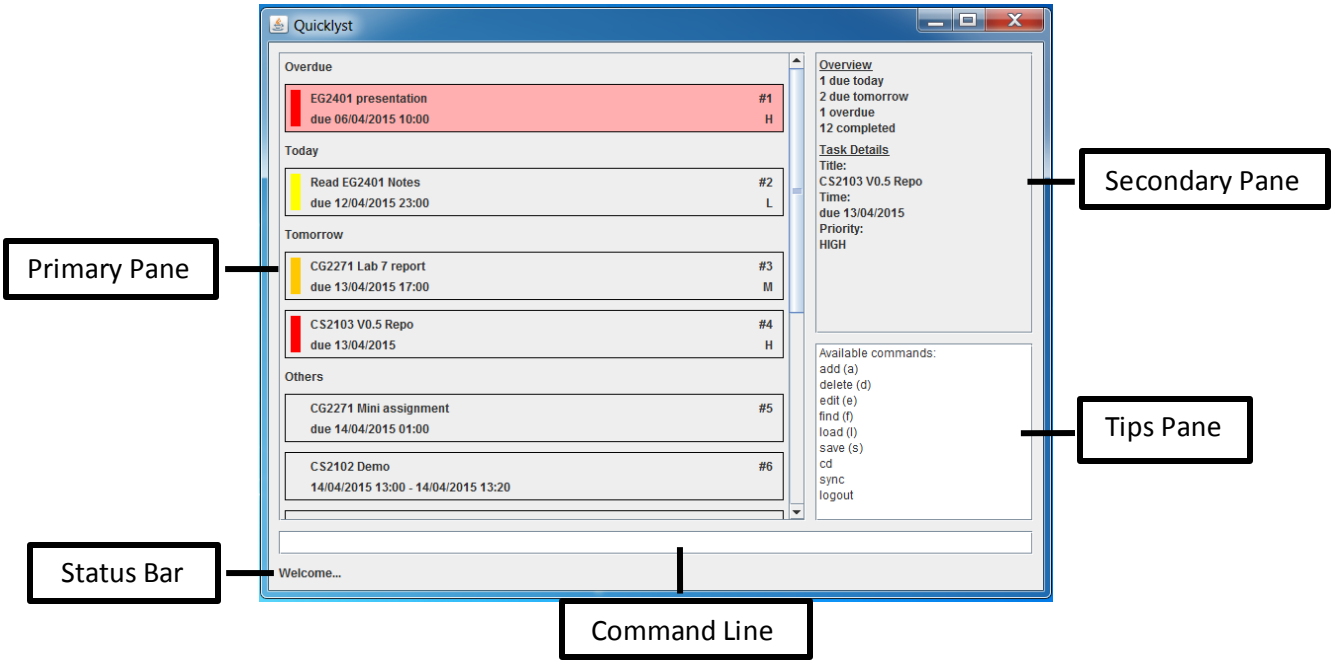
Extra feature: *Google Integration*

		
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

Understanding the User Interface



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 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. Details of the tasks will also be displayed as you hover your cursor over the tasks.

Tips Pane

If you ever need help using the commands in Quicklyst, the help page will be displayed in this pane as you enter the commands.

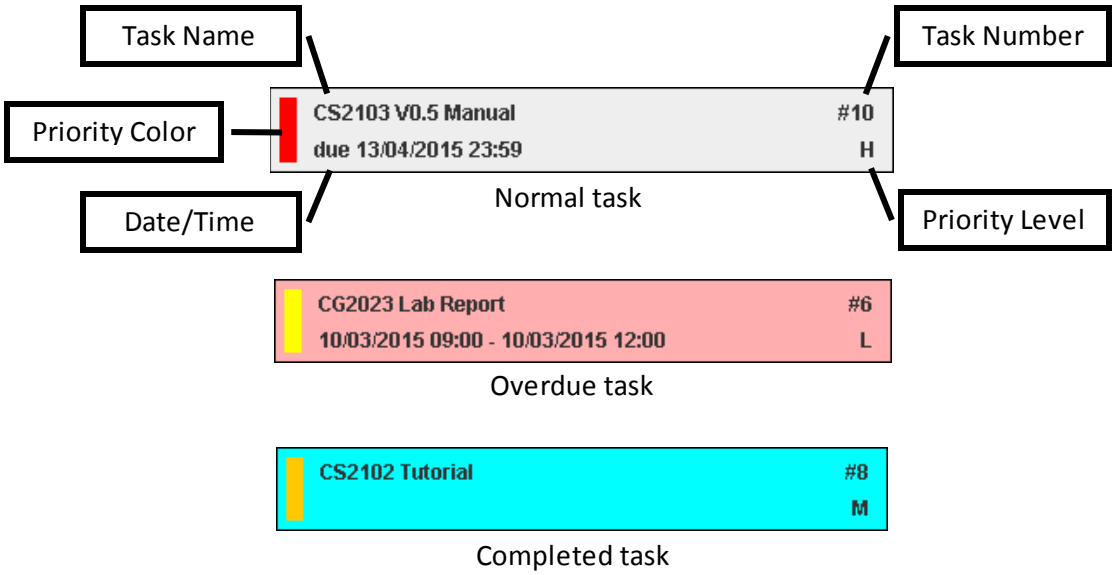
Status Bar

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

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.

Start and Due Date

This is where you can see when you should start and end your task. You have the option to include the start/due date when you add the task or go back to edit/add it any time you want. If you did not include a start/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.

Completed and Overdue tasks

In order to make it easier for you to look at your tasks, completed task will be highlighted in green while overdue task will be coloured red. If a task is both completed and overdue it will be hidden from the list and you need to use the Find command to display it. You can refer to Section 3.2.1 on how to use the Find command.

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 green 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 shorthand formats, while those enclosed in { } are in more natural formats. You are free to choose between the formats and can mix and match the formats while following certain rules. You can refer to **Appendix A** for examples of how to use the commands.

> Tip: As a beginner, you can start off with the natural formats if you find them more intuitive. However be sure to make use of the shorthand formats as you get a hang of Quicklyst to fully exploit the convenience it provides.

How to key in dates

As dates are the key to using Quicklyst, you need to understand the format of keying it in before you can properly use Quicklyst.

From now on, a "Date" is defined as an attribute that must have a date value and may have a time value. For example, 12/12/12, 7:30pm is a date, while 12/12/12 is also a date.

To specify a Date in Quicklyst, you can either key in a *date*, *day*, *time* or a combination of two. These are the legal combinations accepted by Quicklyst:

1. *date* + *time*
2. *day* + *time*
3. *date*
4. *day*

The table below shows what you can key in for each of "date", "day" and "time".

Date	DD/MM/YYYY, DD/MM/YY, DD/MM (for current year only) Example: 01/01/2011, 1/1/11, 01/01, 1/1
Day	Tdy/Today, Tmr/Tomorrow, Mon/Monday, Tue/Tuesday, … Sun/Sunday
Time	HH:MM, 24 hour format Example: 7:30, 07:30, 13:30, 00:00

> Tip: You can type in "next" before a day in week to specify that day in the next week. E.g. "next mon", "next Saturday". In addition if the day in week is past, Quicklyst will automatically interpret it as that day next week.

Add

Description: Add a task into the list.

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

Fields:

1. Start date: [`-s Date`] {`from/start Date`}
2. Due date: [`-d Date`] {`to/due/by/end Date`}
3. Priority level: [`-p H/M/L`] {`priority high/medium/low`}

> Note: It is important that you enclose the task name with the identifier **backslash "\"**. If not Quicklyst will not be able to identify the name and your command will execute wrongly.

Edit

Description: Edit/add fields of an existing task.

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

Fields:

1. Name: [-n name \] {name name \}
2. Start date: [–s Date/CLR] {from/start Date/clear}
3. Due date: [-d Date/CLR] {to/due/by/end Date/clear}
4. Priority level: [-p H/M/L/CLR] {priority high/medium/low/clear}

> Note: Unlike in Add, you need to enter the **name/-n** identifier before the new name you entered. Don't forget the "/" too!

> Tip: In both Add and Edit, you can specify a timed task by only specifying the date once. E.g. You can type "from mon 7:30 to 9:30" or "from 12/3 23:00 to 1:00".

Complete

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

Command: `COMPLETE/C + Task Number + Y/N/yes/no`

> Tip: You can simply toggle the complete status of a task if you do not key in the last field yes or no

Delete

Description: Delete a task

Command: `DELETE/D + Task Number`

Find

Description: Find the tasks that fit certain criteria.

Command: `FIND/F + Fields`

Fields:

1. Task name: [-n name \] {name name \}
2. Start date: [-s + bf/af/on Date*** OR btw Date & Date]
   {due/end + before/after/on Date OR between Date and Date}
3. Due date: [-d + bf/af/on Date OR btw Date & Date]
   {start + before/after/on Date OR between Date and Date}
4. Priority level: [-p H/M/L] {priority high/medium/low}
5. Completed: [-c Y/N] {completed yes/no}
6. Overdue: [-o Y/N] {overdue yes/no}
7. Show all tasks: ALL

> Note: *** Find does not provide the feature of finding a time. Please only key in a Date with a date value

Undo

Description: Undo the previous command, except Save and Log out commands.

Command: `UNDO/U/Ctrl Z`

Redo

Description: Redo the previous command.

Command: `REDO/R/Ctrl Y`

> Tip: You can also access the previous commands entered by pressing the UP/DOWN arrow keys

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`

Sync with Google

Description: Synchronise all tasks with your Google Calendar. This command will redirect you to the Google Calendar log in page via your web browser. You need a valid Google account to enjoy this feature.

Command: `SYNC/SG`

Log out from Google

Description: Log out from the Google account you have previously logged in to during Sync. If you do not use this command, your account will continue to be logged in and you do not need to retype your password the next time you enter Sync.

Command: `LOGOUT/LG`

Appendix A: Command Examples

Add

1. Command: `A Task 1\ -d 16/08 -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 17:00 -d TMR 19:00`
   Result: Added *"Task 4", starts today 5pm, due tomorrow 7pm*
5. Command: `a Task 5\ from 13/03 to 13/02/2016 priority low`
   Result: Added *"Task 5", starts on 13 Mar, due on 13 Feb 2016, Low priority*
6. Command: `add Task 6\ from 13/3 5:00 to 7:00`
   Result: Added *"Task 6", starts from 13 Mar 5am to 13 Mar 7am*

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 17/08 -p L`
   Result: *#3, "Task 2", due on 17 Aug, Low priority*
2. Command: `EDIT 3 -d 17/08 6:30`
   Result: *#3, "Task 1", due on 17 Aug 6:30am, High Priority*
3. Command: `EDIT 3 -d CLR`
   Result: *#3, "Task 1"*
4. Command: `e 3 start 13/3 due 13/2/2016 priority CLR`
   Result: *#3, "Task 1", starts on 13 Mar, due on 13 Feb 2016*
5. Command: `Edit 3 name task one\ start today due next mon`
   Result: *#3, "Task one", starts tomorrow, due on next Monday, High priority*

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" or "one", with closer match listed at the top
2. Command: `FIND -o Y`
   Result: List all tasks that are overdue
3. Command: `FIND -c Y`
   Result: List all tasks that are completed
4. Command: `FIND -d on TDY`
   Result: List all tasks that are due today
5. Command: `F -p H -d btw TMR & 16/8`
   Result: List all tasks that are High priority, due between tomorrow and 16 Aug
6. Command: `FIND -s bf 1608 -d af 17/9`
   Result: List all tasks that start before 16 Aug and is due after 17 Sep
7. Command: `find start after 16/8 due before 15/9`
   Result: List all tasks that start after 16 Aug and due before 15 Sep
8. Command `find start from TDY to TMR due from 14/9 to 16/9 priority high`
   Result: List all tasks that start between today and tomorrow, and is due between 14 Sep and 16 Sep, that are High priority
9. Command: `find all`
   Result: List all tasks
10. Command: `find name task\ due after tomorrow`
    Result: List all tasks contains the word "task" and due tomorrow

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.

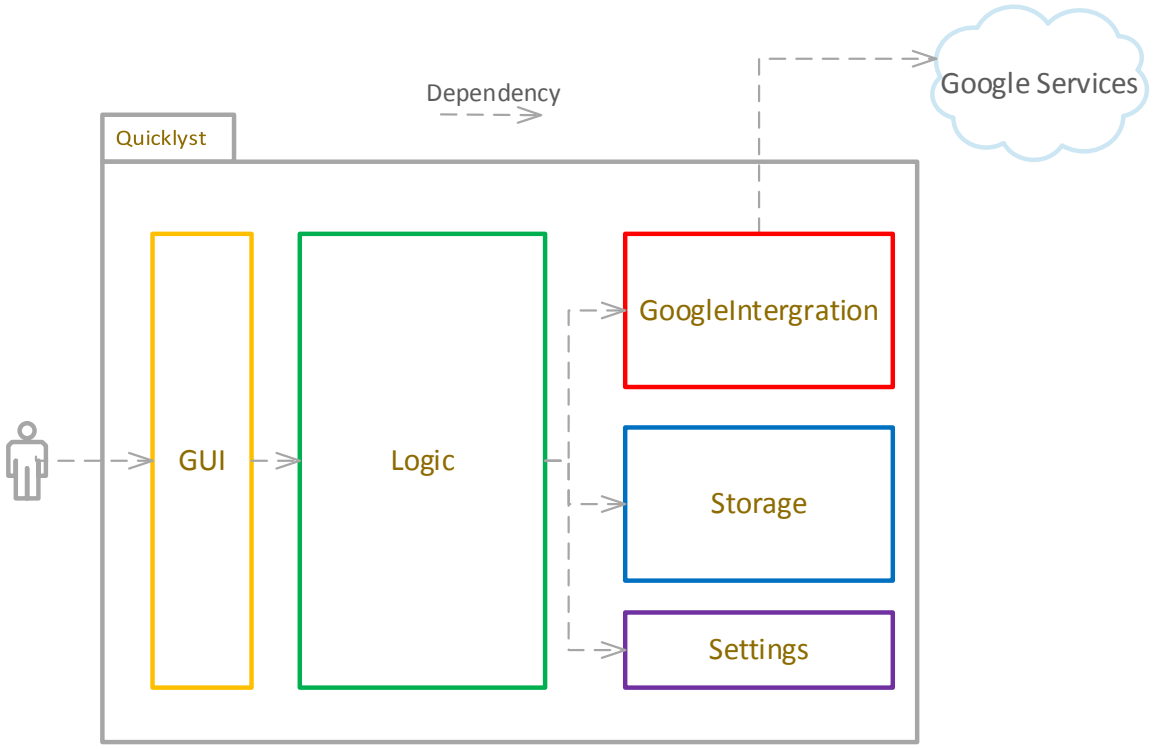
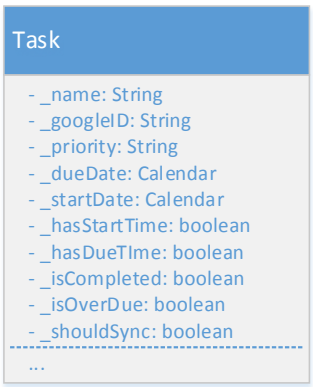


Figure 1. Quicklyst architecture

Task Class

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

The Task object has attributes of a task in real life and are passed among the various components of the Quicklyst architecture to realise its different functionalities. The class diagram of Task and its notable API are shown in *Figure 2*. 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 2. Task Class Diagram (Left) and API (Right)

GUI Component

The Graphical User Interface (GUI) provides an interactive and visual feedback for the user. When the user is keying in commands, the tip text area shows the relative command syntax for user reference. Upon user's task creation, GUI will show the corresponding status and update two main panels – task list and overview accordingly. *Figure 3* shows the class diagram of the GUI component and its dependency.

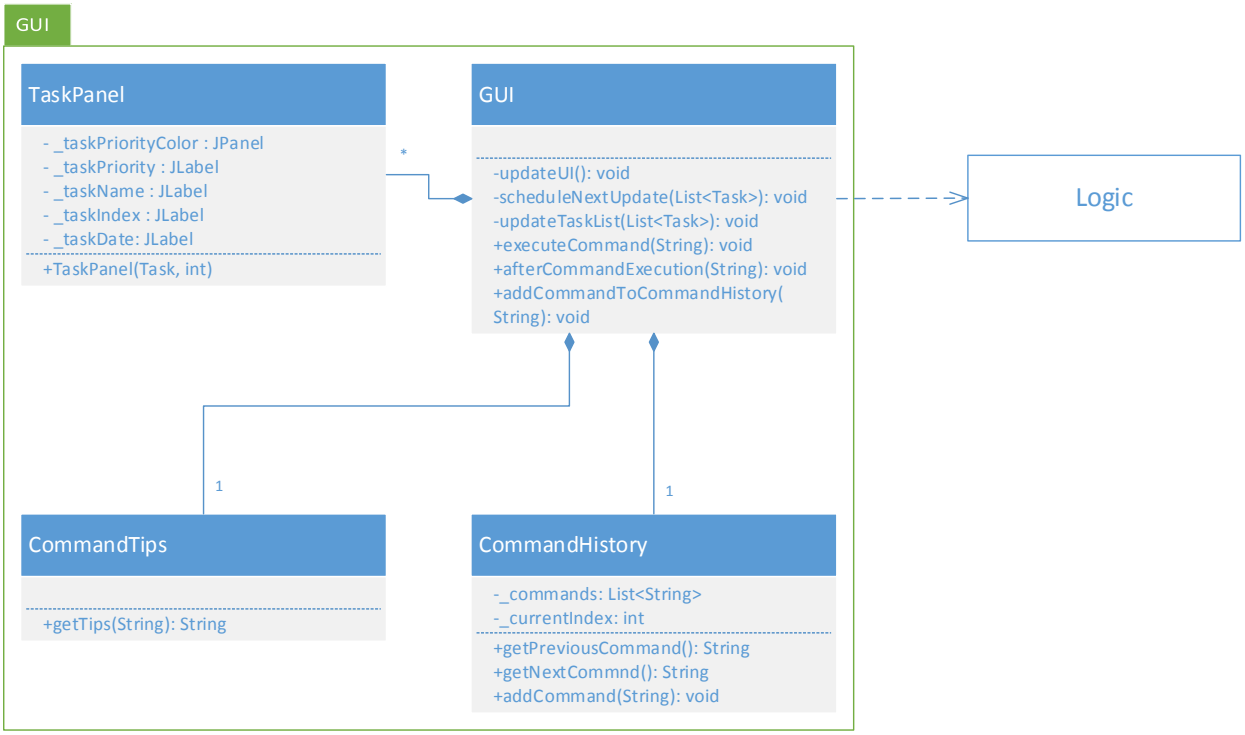


Figure 3. GUI component class diagram

GUI Class

The GUI class is implemented with the MVC (Model-View-Controller) pattern in mind. The component is solely responsible for the view portion of the pattern. Part of the controller (UI events) is also implemented by the GUI component. All other parts such as data processing and storage of data are handled by other backend components.

Task List Panel

The task list panel displays tasks according to their due dates. The tasks are shown up to maximum three categories (*Overdue*, *Two nearest due dates* and *Others*). Each task creates a TaskPanel object to be added to the task list. The TaskPanel class is responsible for the display of the details of each task.

Overview Panel

Overview panel consists of two parts: one part is the label that holds the overall tasks statistics and the other part is to serve as hover display. The computation of the overall task statistics is based on a master list from Logic that contains all the tasks relevant in the current session. This list may be different from the list of Tasks shown by the Task List Panel if the user has just executed a find command.

Command Tips Panel

The Command Tips Panel helps the user in the command syntax by utilising the *CommandTips* class. The process of generating command tips is illustrated in *Figure 4*.

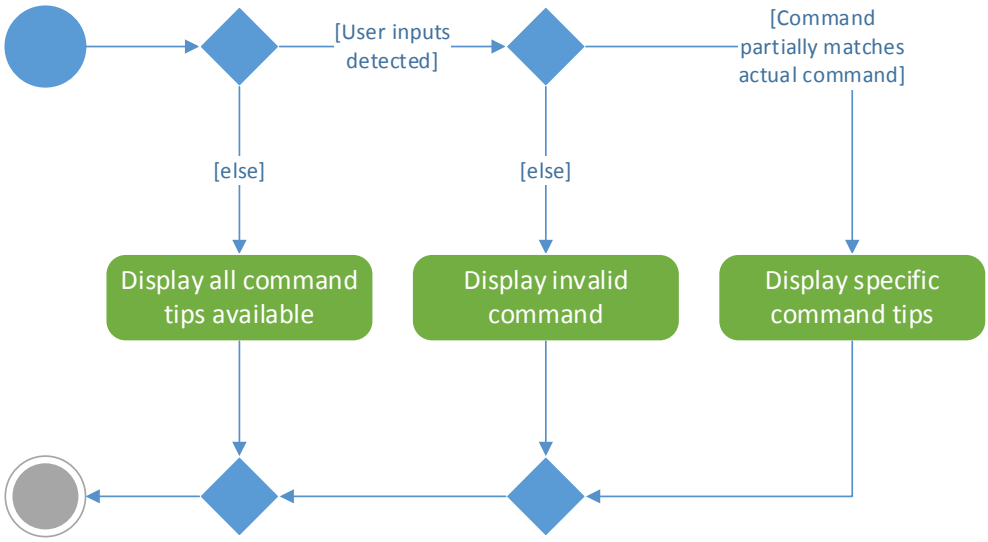


Figure 4. Command Tips generation Activity Diagram

User Interaction Sequence Diagram

The sequence diagram depicted in *Figure 5* demonstrates the interaction between the user and the GUI. In the below use case, before the user hits the Enter key, the command tips will be displayed accordingly. If the UP/DOWN key is pressed, the input will be processed by the CommandHistory class to generate previous commands. The user then enters the desired command.

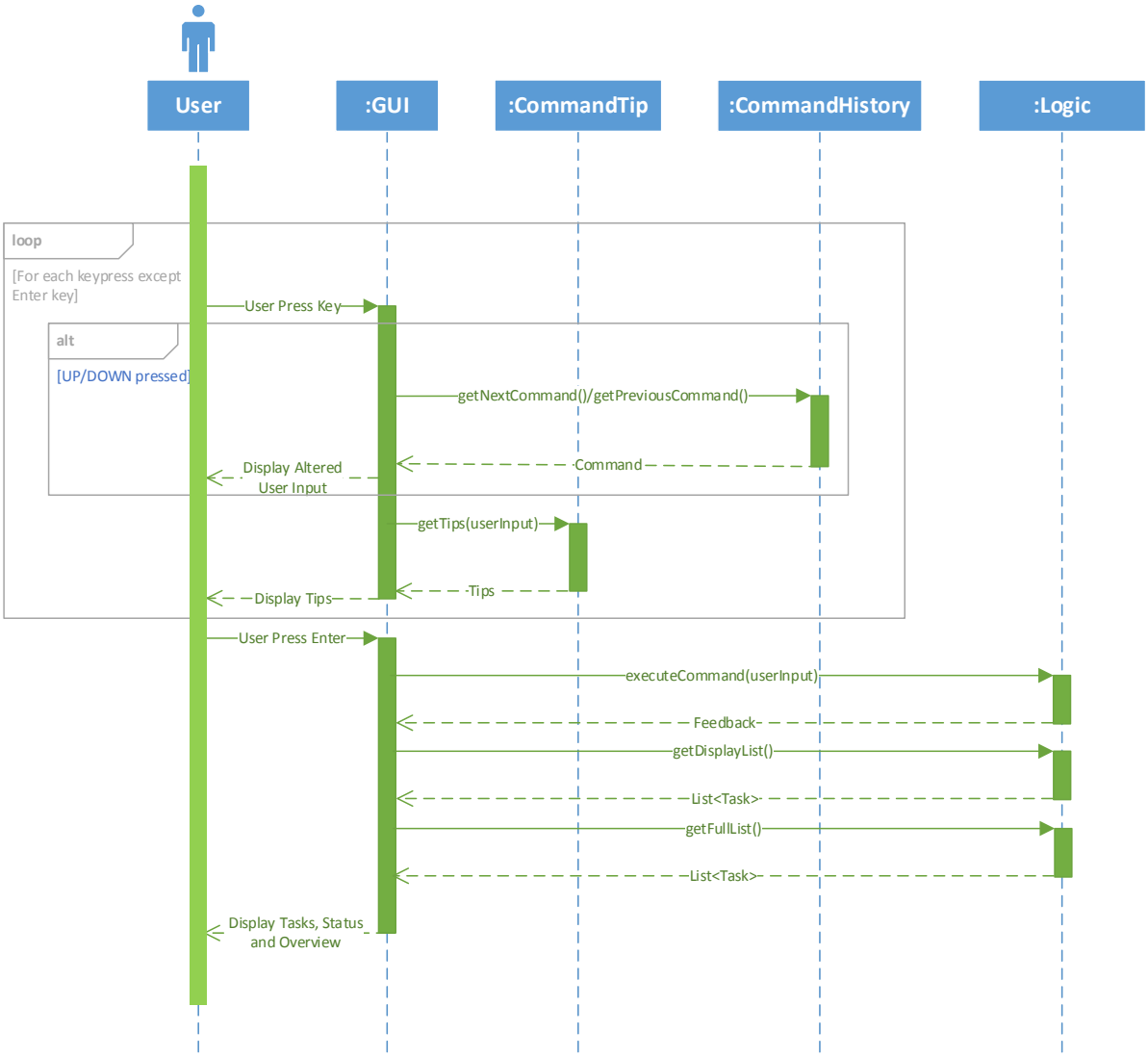


Figure 5. User interaction with GUI component Sequence Diagram

In addition to the above use case, hotkeys Ctrl + Z (Undo) and Ctrl + Y (Redo) also cause GUI to invoke the executeCommand API of Logic component.

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. *Figure 6* shows the relationship between the sub-components of the Logic Component under the Command Pattern.

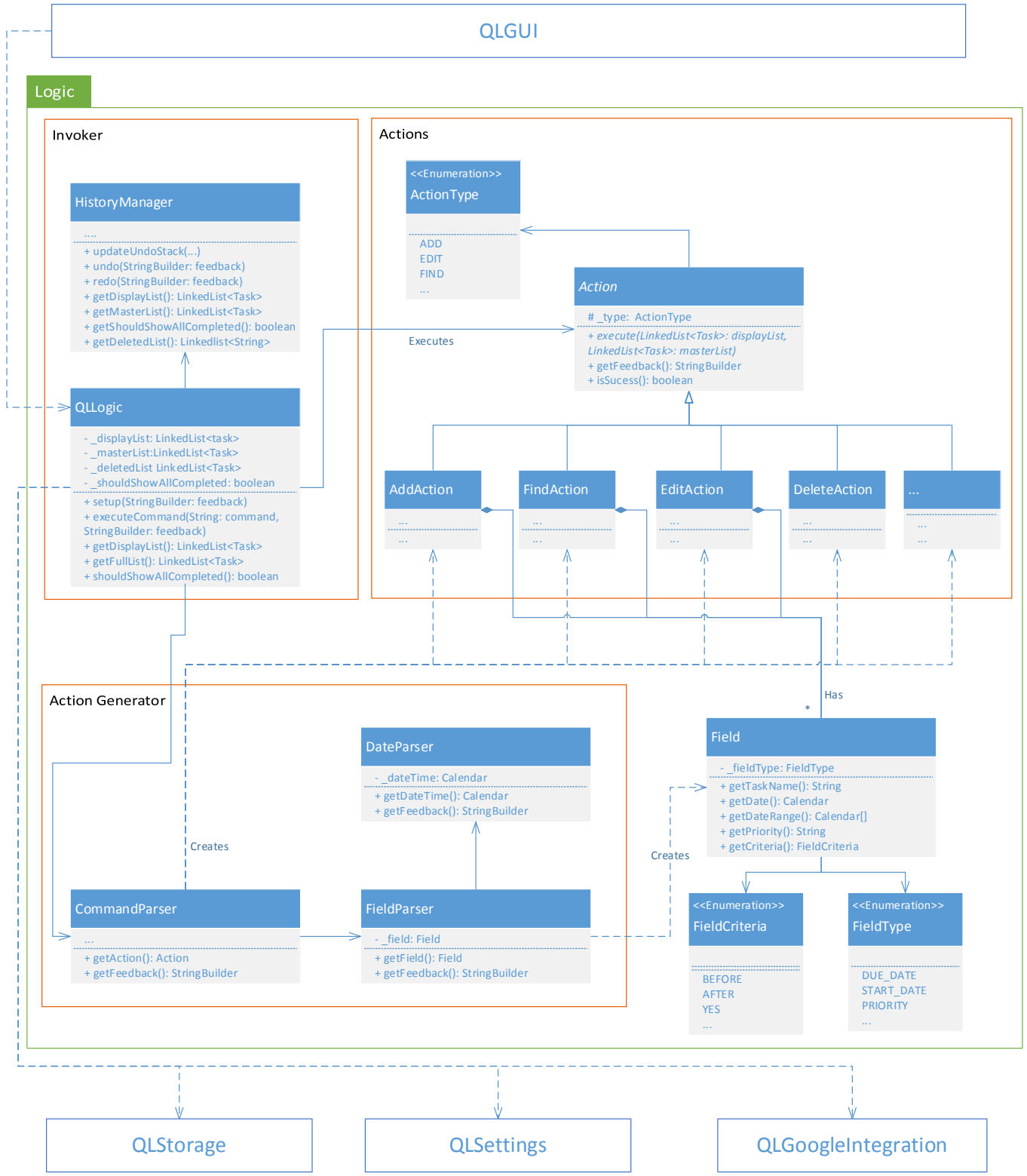


Figure 6. Logic component class diagram

As seen from the class diagram, the Logic Component implements the Command Pattern, where an "Action Generator" creates a concrete "Action" type and passes it to the "Invoker" 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 "Invoker" only need to be associated with one general "Action" object instead of many concrete "Action" objects.

The Invoker

Besides invoking the Actions generated by ActionGenerator, the Invoker sub-component holds and manages tasks that are relevant in the current session. It is made up of Logic Class and HistoryManager Class.

Logic Class

Logic Class is the single point of access between the sub-components of Logic and other components of Quicklyst. It main function is to receive the commands from GUI and decides what sub-components to call to perform a specific action, and returns the result of this action back to GUI. This applies the Façade Pattern where the Logic component can be accessed without exposing its details. *Table 1* 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 Logic 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 false if otherwise.

Table 1. 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 Logic and a *redoStack* to store the "states" that are ahead of the current state. The algorithm of the undo redo process can be found in Appendix A. *Table 2* shows some of the notable API of the HistoryManager class.

Method	Parameters	Description
updateUndoStack(LinkedList<Task>: displayList, LinkedList<Task> masterList, LinkedList<String> deletedList, boolean: shouldShowAllCompleted)	**displayList**: *_displayList* in Logic **masterList:** *_masterList* in Logic **deletedList:** *_deletedList* in Logic **shouldShowAllCompleted:** *_shouldShowAllCompleted* in Logic	Make current state of Logic available for undo in the future.
undo(StringBuilder: feedback)	**feedback**: the feedback to be displayed to the user after each operation. An empty StringBuilder	Set ups the previous state of Logic

	should be passed in during each call of the method.	
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 Logic 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.

Table 2. HistoryManager class API

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 Logic 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, FieldParser and Date Parser objects may be created to aid the parsing process. The Activity Diagram in *Figure 7* shows how the CommandParser class parses commands and generate Actions.

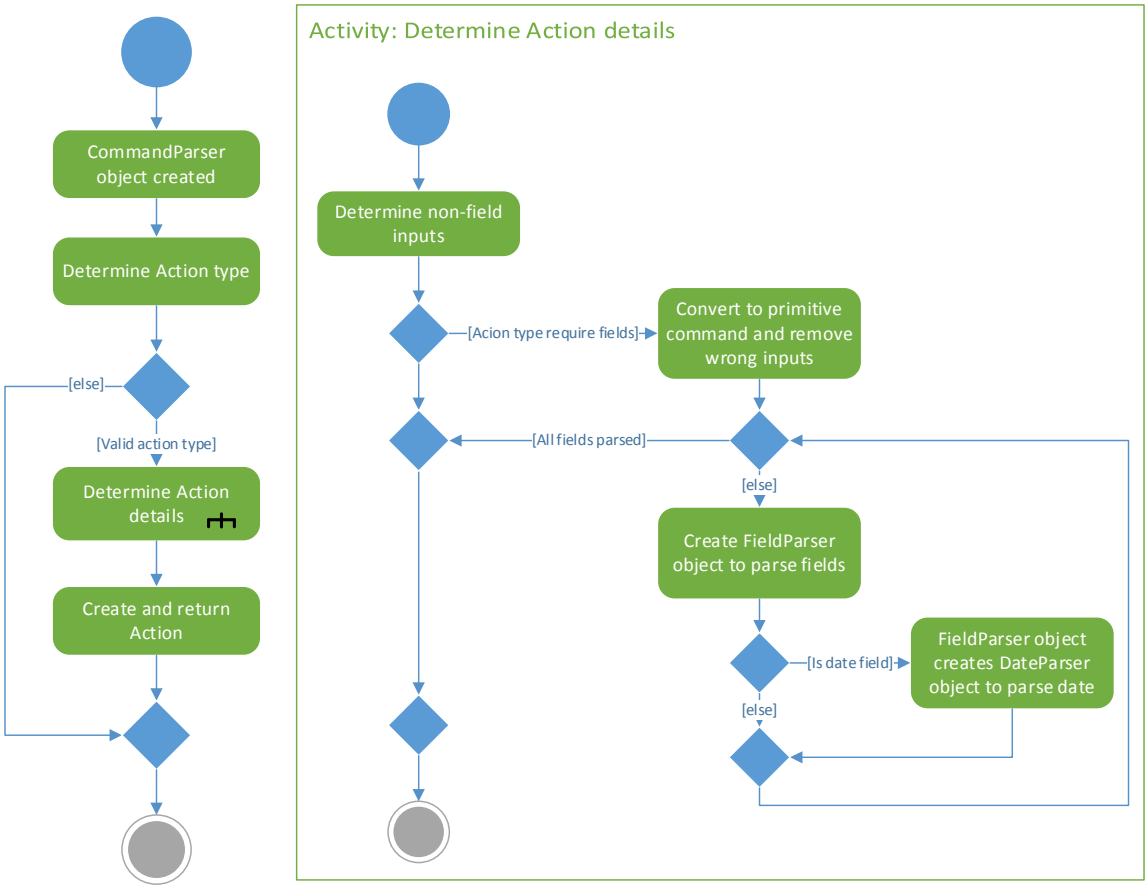


Figure 7. Activity Diagram for parsing commands and generating Actions

Notable APIs of the CommandParser, FieldParser and DateParser classes are shown in *Table 3*, *Table 4* and *Table 5* 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 4. 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 5. 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 Invoker 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 within one execute call. *Figure 7* shows the Class Diagram that depicts the relationships between all the Action subclasses. Notable algorithm of the FindAction can be found in Appendix A.

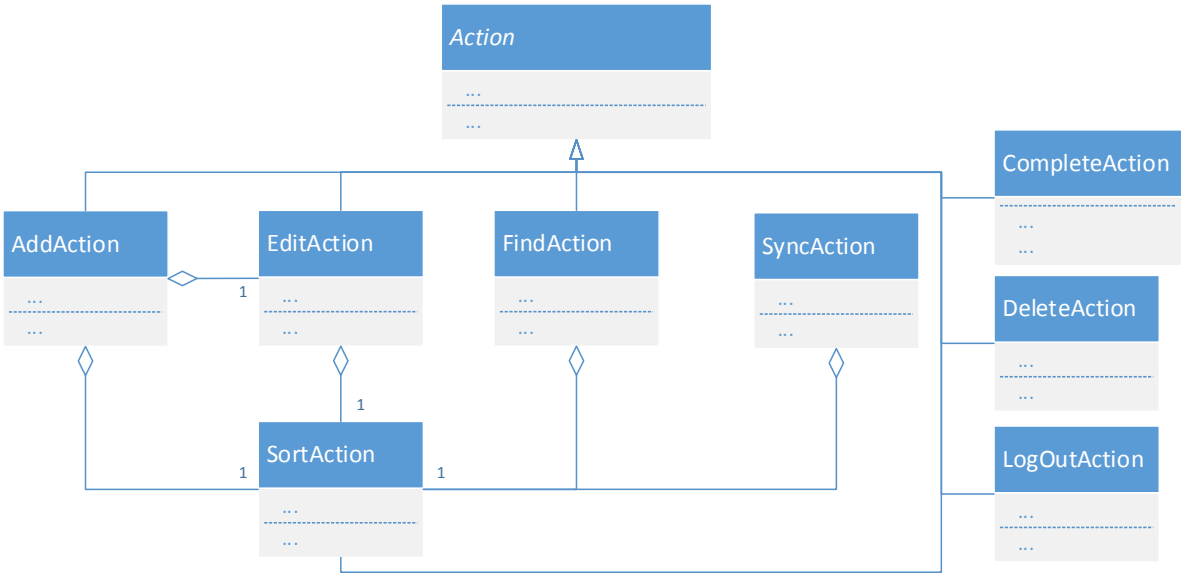


Figure 7. Actions sub-component class diagram

Interactions within the logic component

To illustrate how the different sub-components of Logic component working together to execute a typical command, *Figure 8* shows a sequence diagram depicting a typical add command.

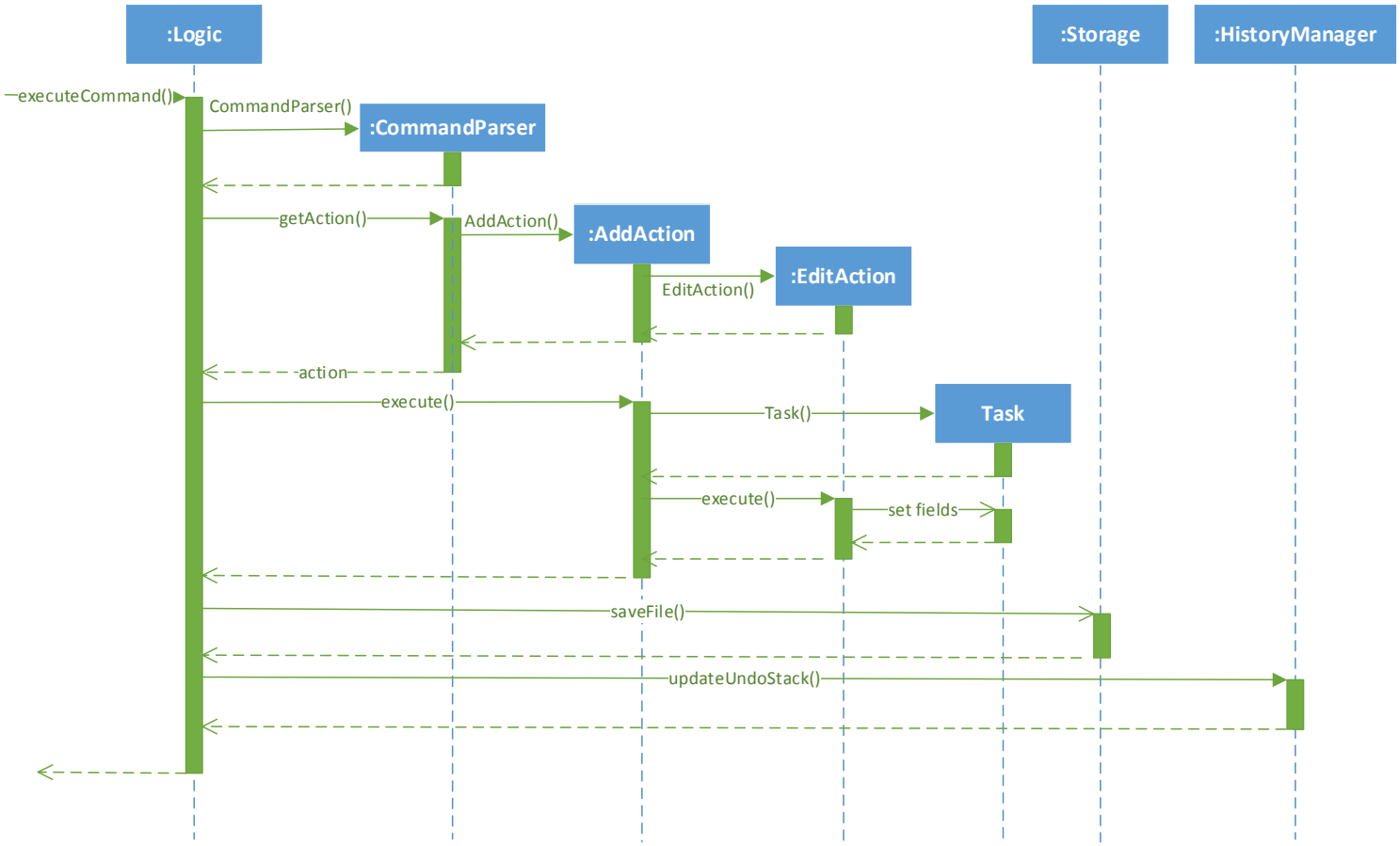


Figure 8. Sequence diagram within Logic component for a typical add command

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 9* shows the structure of the Storage component and its dependency.

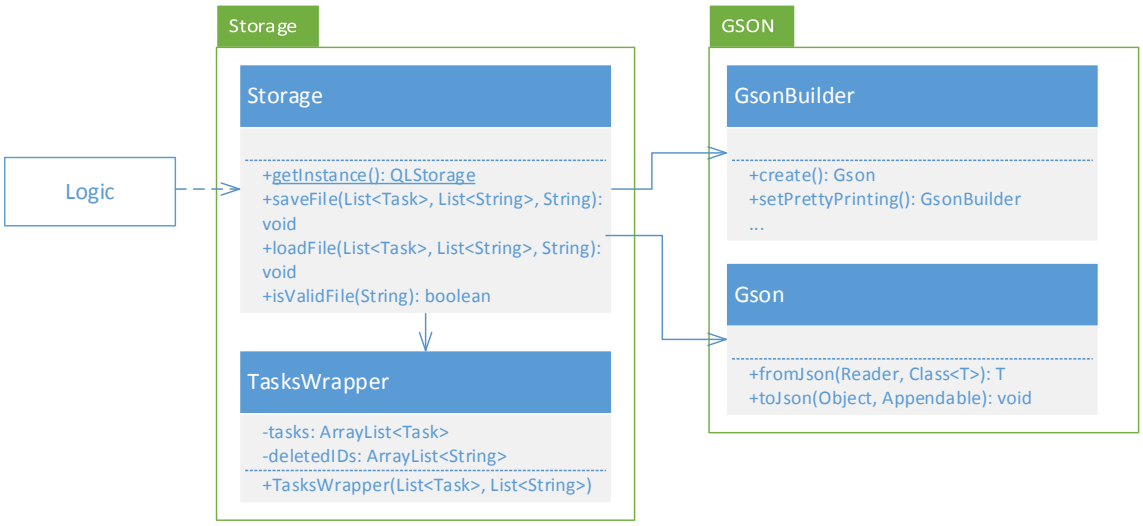


Figure 9. Storage component class diagram

Table 6 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 6. 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 10* illustrates the structure of the GI component.

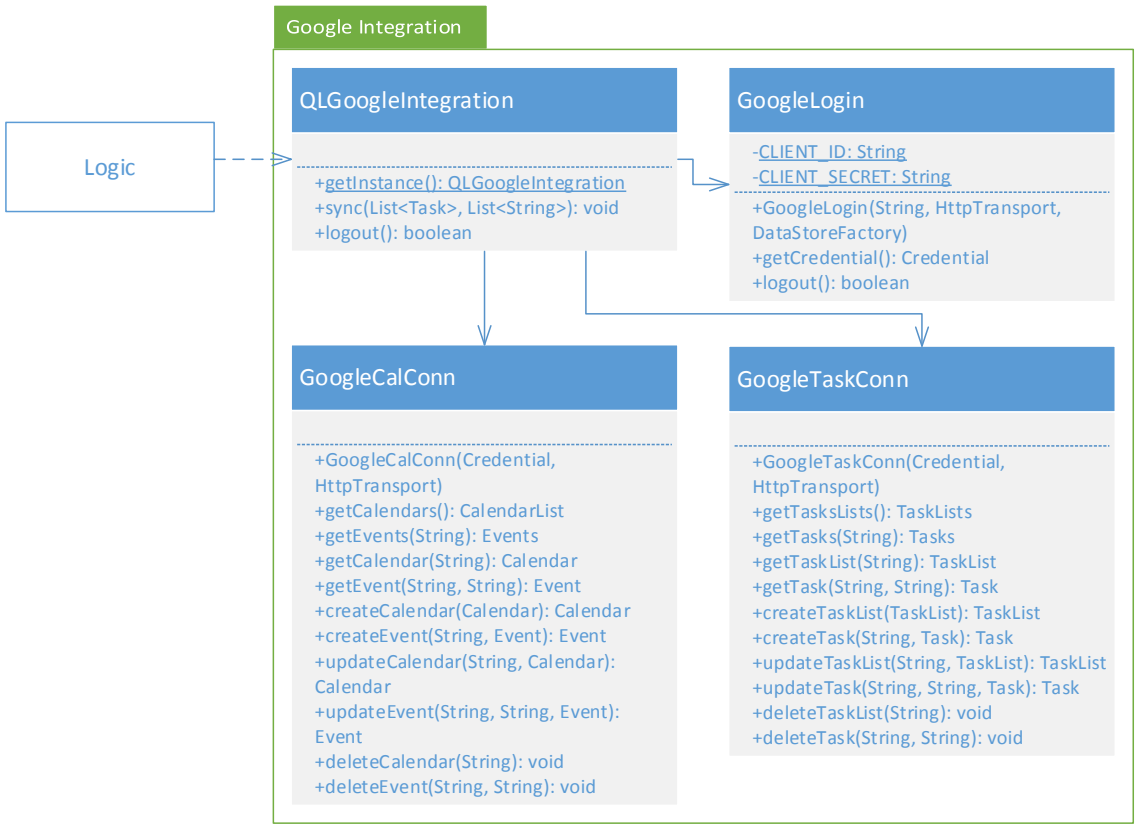


Figure 10. Google Integration component class diagram

Table 7 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 7. GoogleIntegration class API

The synchronisation process with Google Calendar is illustrated in *Figure 11*. A similar process is also used for synchronisation with Google Tasks.

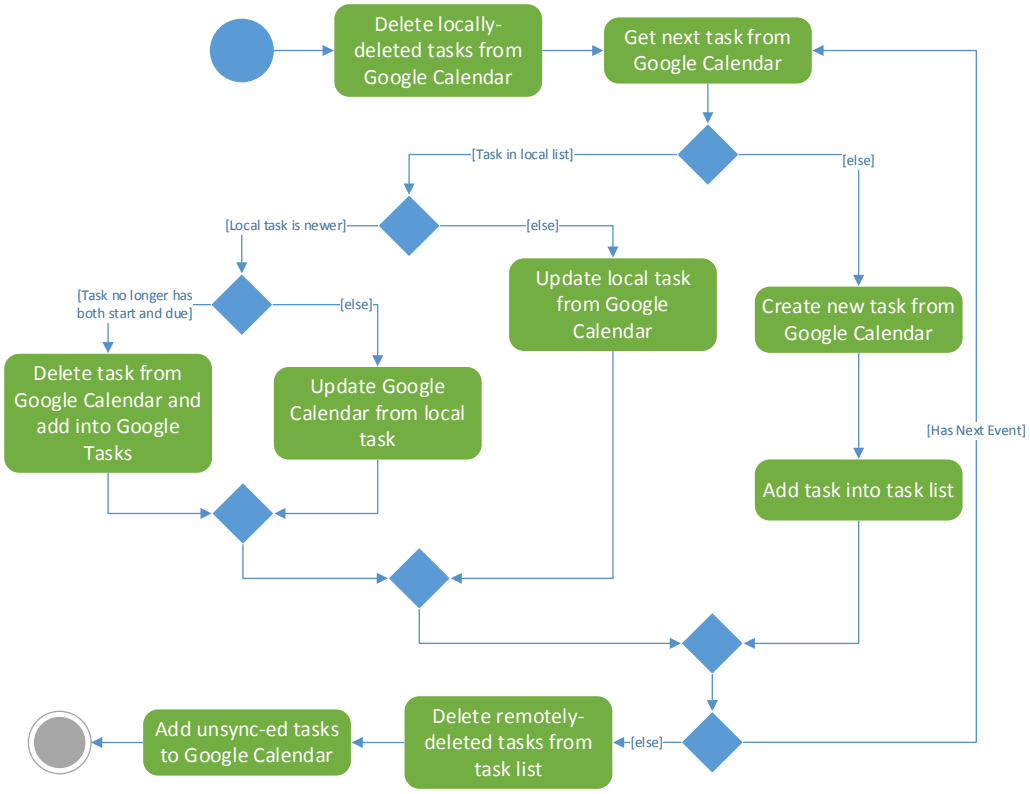


Figure 11. Google Calendar Synchronisation activity diagram

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 12* illustrates the class diagram of the component.

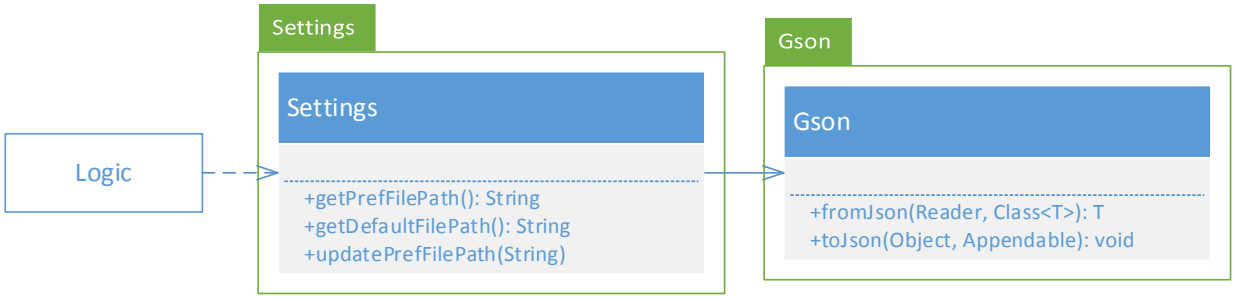


Figure 12. Settings component class diagram

Testing Framework

Quicklyst uses jUnit framework to develop unit test for all its components. To run the unit tests of Quicklyst in the Eclipse IDE, select the test to be run under Package Explorer, right click to bring up the context menu, go to Run As, and select JUnit Test. *Figure 13* depicts the context menu.

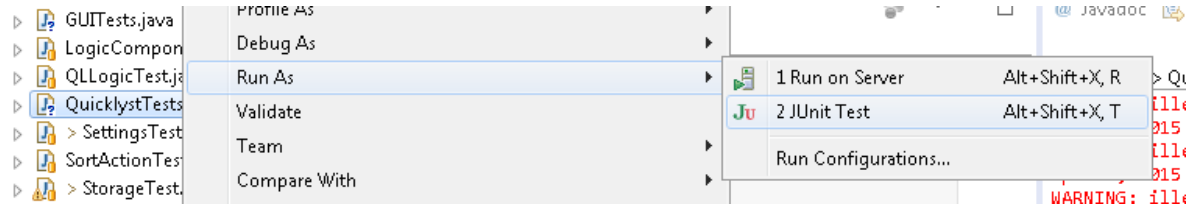


Figure 13. Eclipse screenshot of context menu

To run unit test on a particular class, choose <Classname>Test.java files. To test a particular component, choose <Component name>Test(s).java files. To run all available unit test, choose QuicklystTests.java. *Table 8* lists the unit test available in alphabetical order.

Class unit tests	Component unit tests
AddActionTest.java	GoogleIntegrationTest.java
CommandHistoryTest.java	GUITests.java
CommandParserTest.java	LogicComponentTests.java
CommandTipsTest.java	SettingsTest.java
DateParserTest.java	StorageTest.java
EditActionTest.java	
FIeldParserTest.java	
FindActionTest.java	
LogicTest.java	
SortActionTest.java	

Table 8. Quicklyst unit test list

Appendix A - Notable Algorithms

FindAction

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 A* illustrates the idea.

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

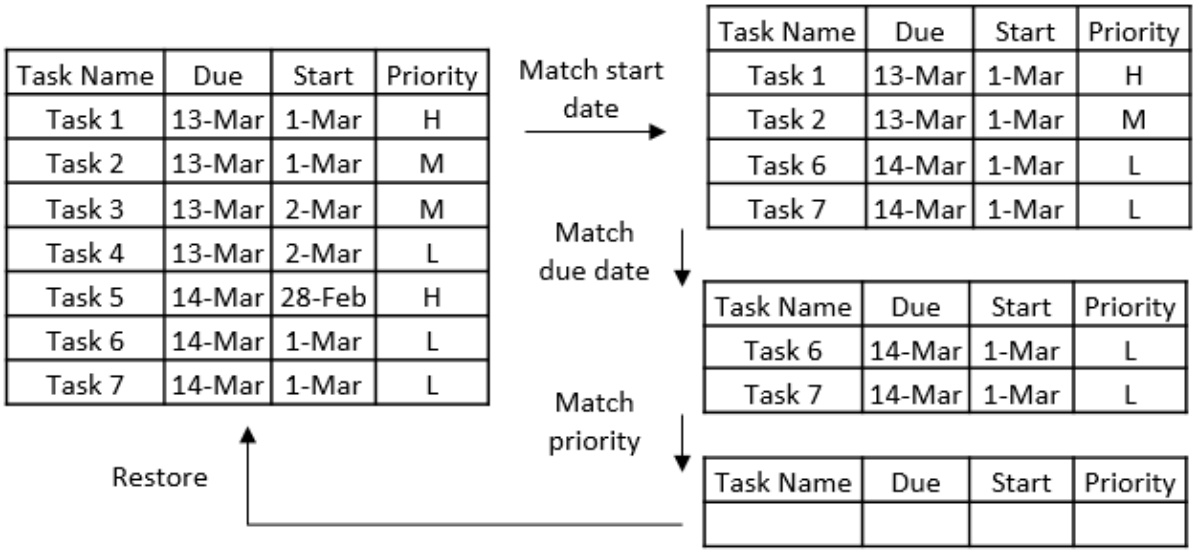


Figure A. Finding tasks example

Undo/Redo

After each change in "state", Logic 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 B* illustrates this process.

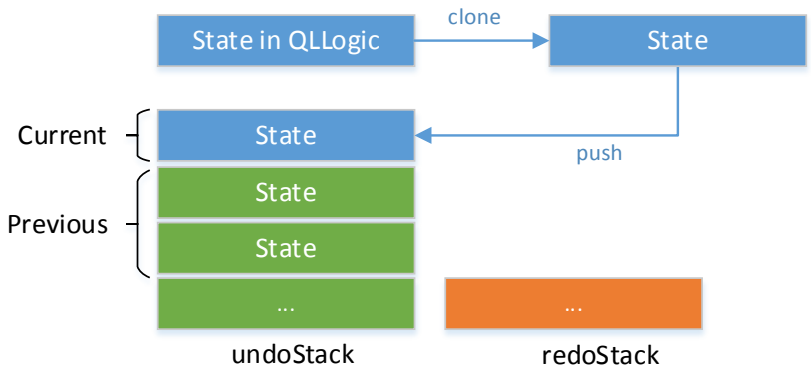


Figure B. Updating of undoStack after executing a command

When Logic 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 Logic. When the Logic calls redo(), the "state" that is ahead of the current "state" is popped out of *redoStack*, pushed into *undoStack*, and retuned to Logic. *Figure C* illustrates these processes.

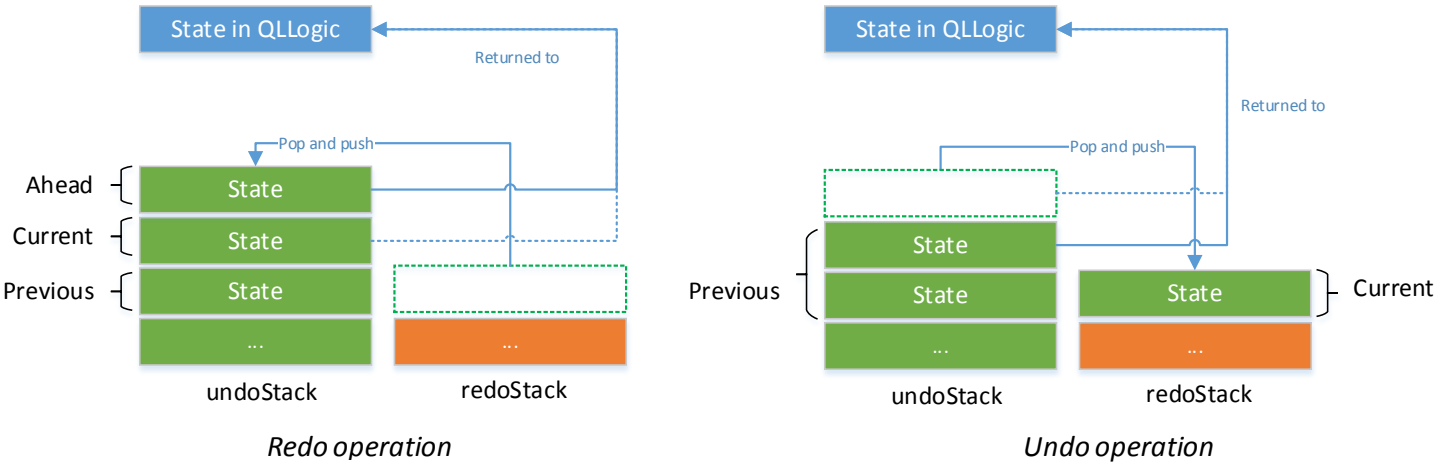


Figure C. Behaviour of stacks during undo and redo