Task Catalyst

Supervisor: Yeow Kai Yao **Extra feature**: Natural Bucket

Ang Kah Min, Kelvin	Toh Zhen Yu	Lin XiuQing, Thida	Lim Wei Jie
Project Team Leader **Code Quality** **Integration** Testing	**CS2101 Team leader** **Documentation** Code Quality Testing	**Scheduling and** **Tracking** **Resource Acquisition** Testing Integration	**Testing** Code Quality Resource Acquisition Integration

Contents

1. User Guide

1.1 Main Interface

The **Settings Button** will launch the settings window and the **Exit Button** will close the program.

The **Task View** is where tasks are displayed. It changes accordingly to category, hashtags and search term.

The **Command Bar** is where commands are entered. It is the main mode of operating the software.

The **Default Hashtags** contain categories that tasks are grouped into by default. Below is an explanation of what they mean:

#all	All Tasks
#pri	Priority Tasks
#tdy	Today's Tasks
#tmr	Tomorrow's Tasks
#upc	Upcoming Tasks
#smd	Tasks for Someday (no date information)
#dne	Done Tasks

The **User Hashtags** section displays hashtags used by the user when adding tasks. They can be quickly used to organize tasks.

The **Status and Help Bar** displays helpful context-sensitive hints and status messages.

1.2 Adding a Task
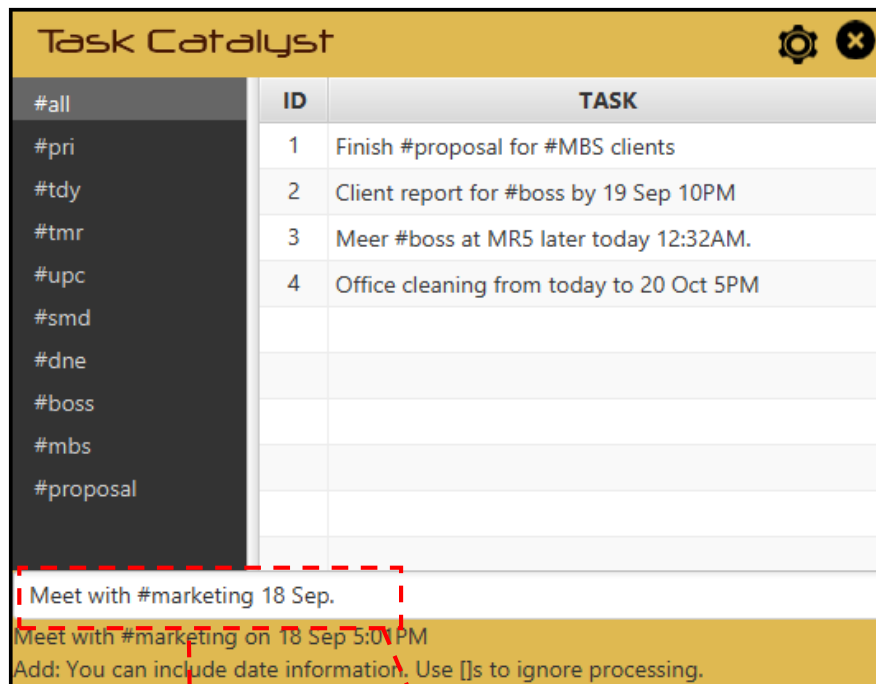
Adding a task is as simple as typing it into the command bar. **You do not have to follow strict formats** as the program naturally understands date and time.
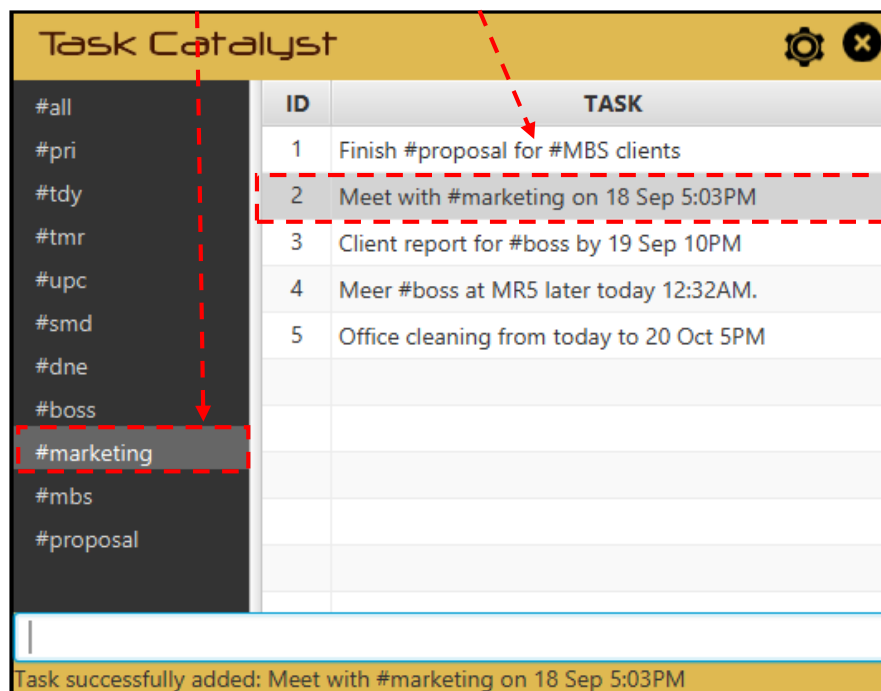
You can make use of the **hashtagging** feature to organize your tasks. When hashtags are specified, they appear in the hashtag list at the left.

1.3 Working with Hashtags
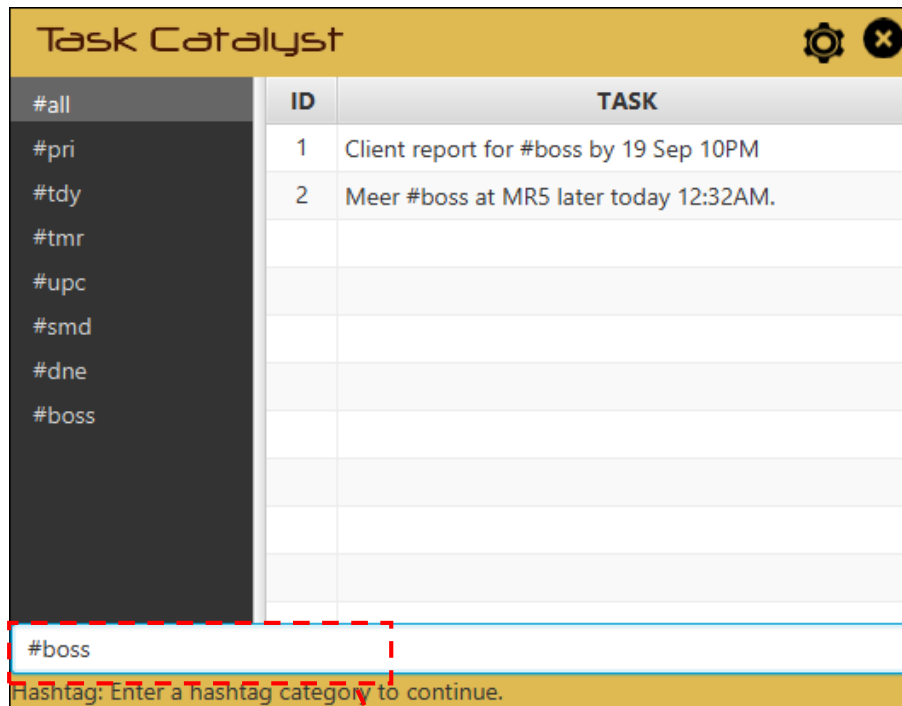
You can quickly navigate through categories or custom hashtags simply by typing the hashtag in the command bar.
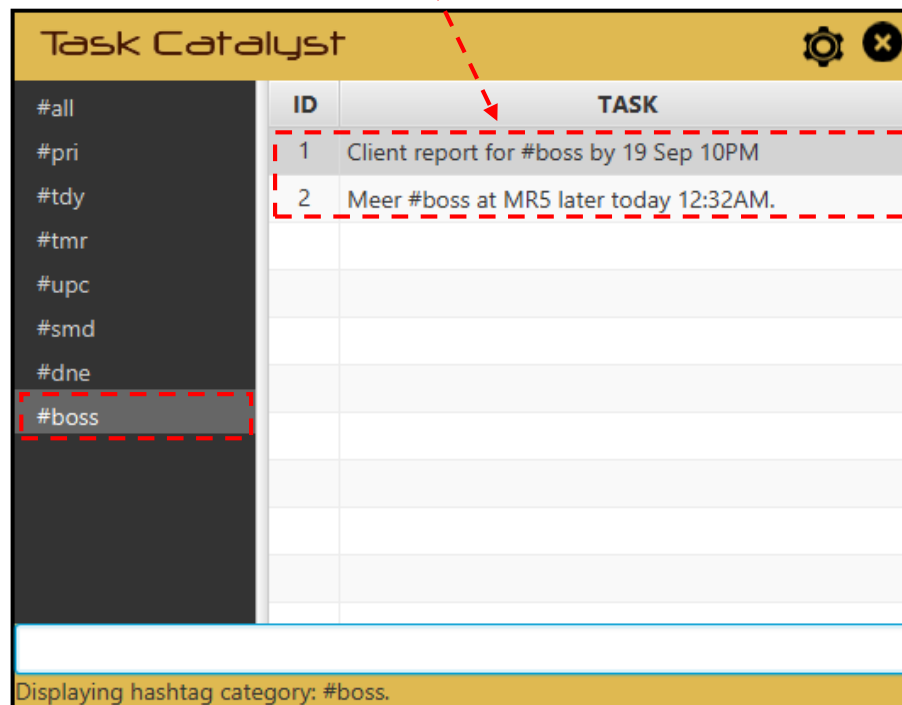
This will display the corresponding items with the hashtag. Notice that the selected hashtag is highlighted.

1.4 Searching for Tasks

You can also search for tasks typing "search", or alternative commands, followed by a search string.

Matching items are quickly displayed in the task view.

1.5 Editing a Task

You edit a task simply by typing "edit", or alternative commands, followed by the task number.

This will move your cursor focus to the task where you can make your changes. Simply hit ENTER to save the changes.

1.6 Completing a Task

You can complete a task simply by typing "done", or alternative commands, followed by the task number.

The specified task is moved into the #done list and will no longer show up in other categories.

1.7 Deleting a Task

You can delete an unwanted task simply by typing "delete", or alternative commands, followed by the task number.

The item is deleted and removed from view.

1.8 Undoing an Action
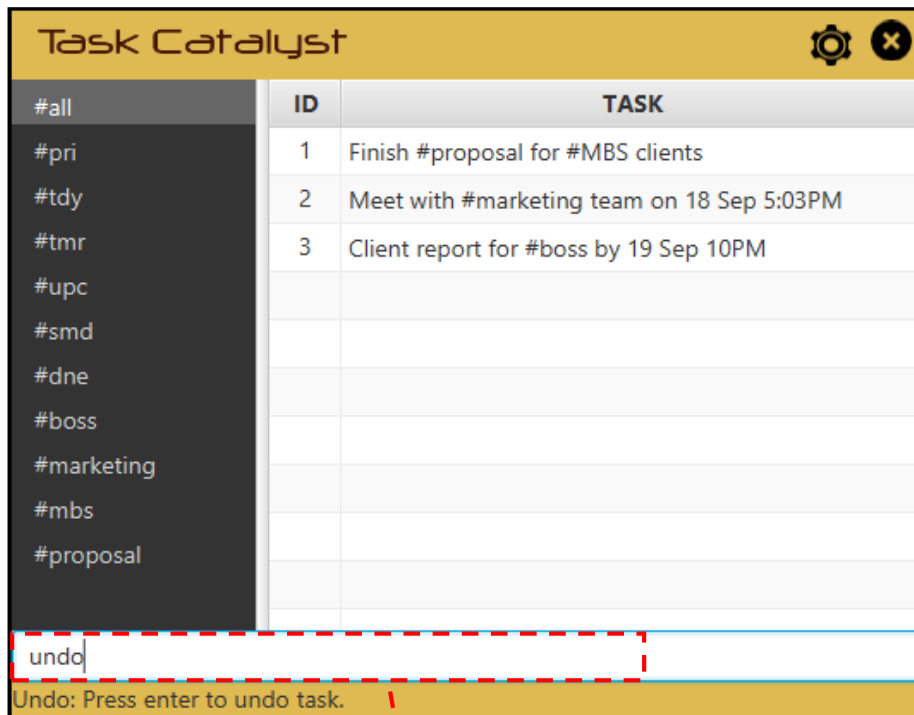
You may undo an action by typing "undo", or alternative commands.
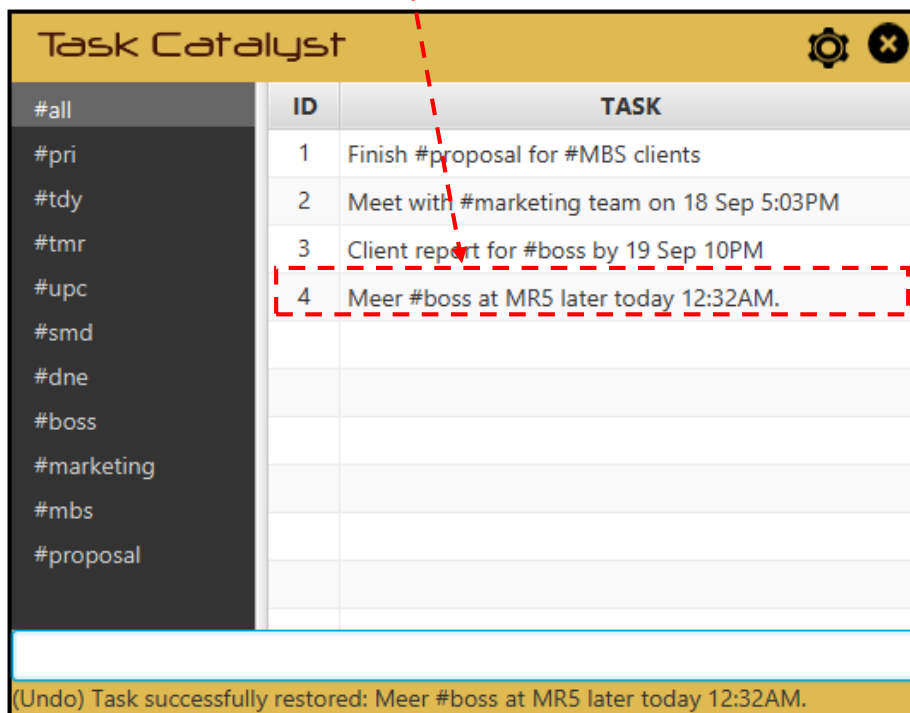
The previous action is undone. You may also redo by typing "redo".

2. Defining the Architecture

Graphical User Interface   Logic   Storage

| GUI | GUI Controller | | Logic Controller (Façade) | Logic Subsystem | | Storage Controller (Façade) | Storage Subsystem |

◀◀─View─▶▶ ◀◀────Controller────▶▶ ◀◀────────Model────────▶▶
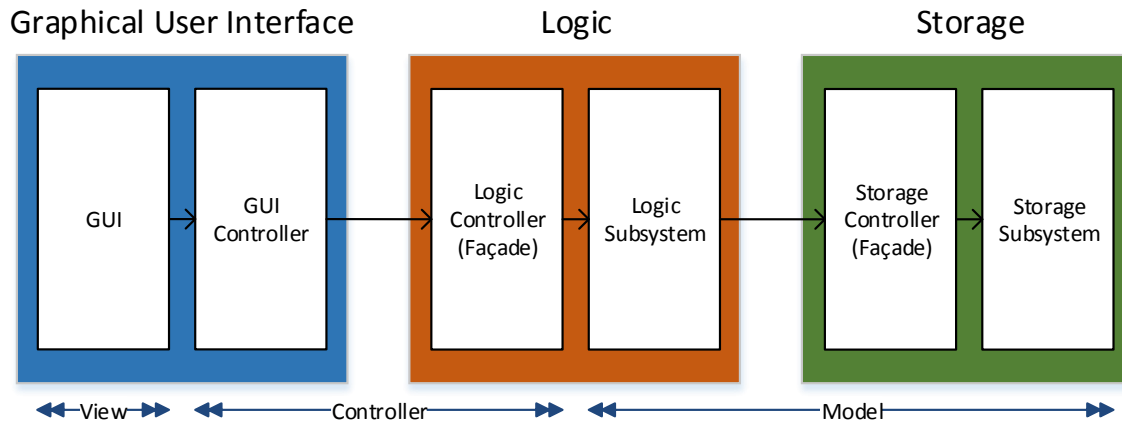
Figure 1 - Architecture

The overall architecture is designed around the <u>MVC (Model-View-Controller)</u> pattern in order to achieve the following objectives:

1) ***DUMB* View**   –   Minimal data processing in the View.
2) ***THIN* Controller**   –   Only data redirections in the Controller.
3) ***SMART* Model**   –   Full data processing in the Model.

The *GUI (Graphical User Interface)* component is the main interface between the user and the system. Its main role is to handle high-level UI interactions, which include displaying tasks, hashtag categories, command hints, status messages, and providing autocomplete functionality. It relies on the Logic component for command execution, low-level decision-making and data processing.

The *Logic* component provides a variety of APIs (Application Programmable Interfaces) for the GUI. It handles parsing and execution of commands, generation of status, hint and autocomplete messages, filtration of task lists, and provision of logical data structures. It depends on the Storage component for physical storage.

The *Storage* component is responsible for persistent physical storage. Its functionality includes JSON (JavaScript Object Notation) encoding and decoding of task lists and settings, as well as read/write operations for physical storage.

3. Developing the Components

3.1 Graphical User Interface
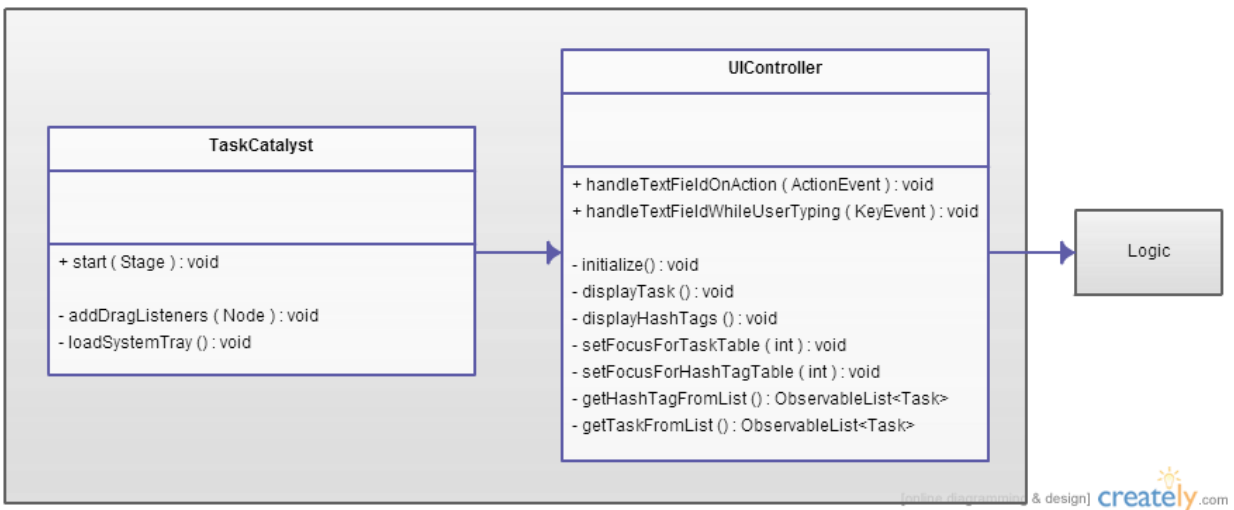
Figure 2: Class Diagram of GUI Component

The Graphical User Interface was designed using JavaFx Scene Builder. The Diagram above shows the class diagram of GUI. The UIController handles all the components displayed in the user interface and also communicating with Logic components.

An example of how the user interact with the GUI in the following sequence diagram:
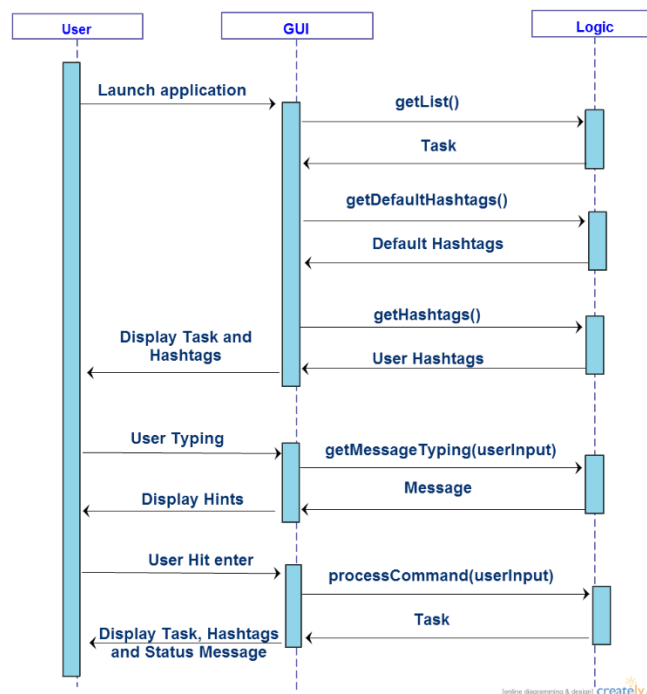
Figure 3: Sequence diagram for GUI

3.2 Logic
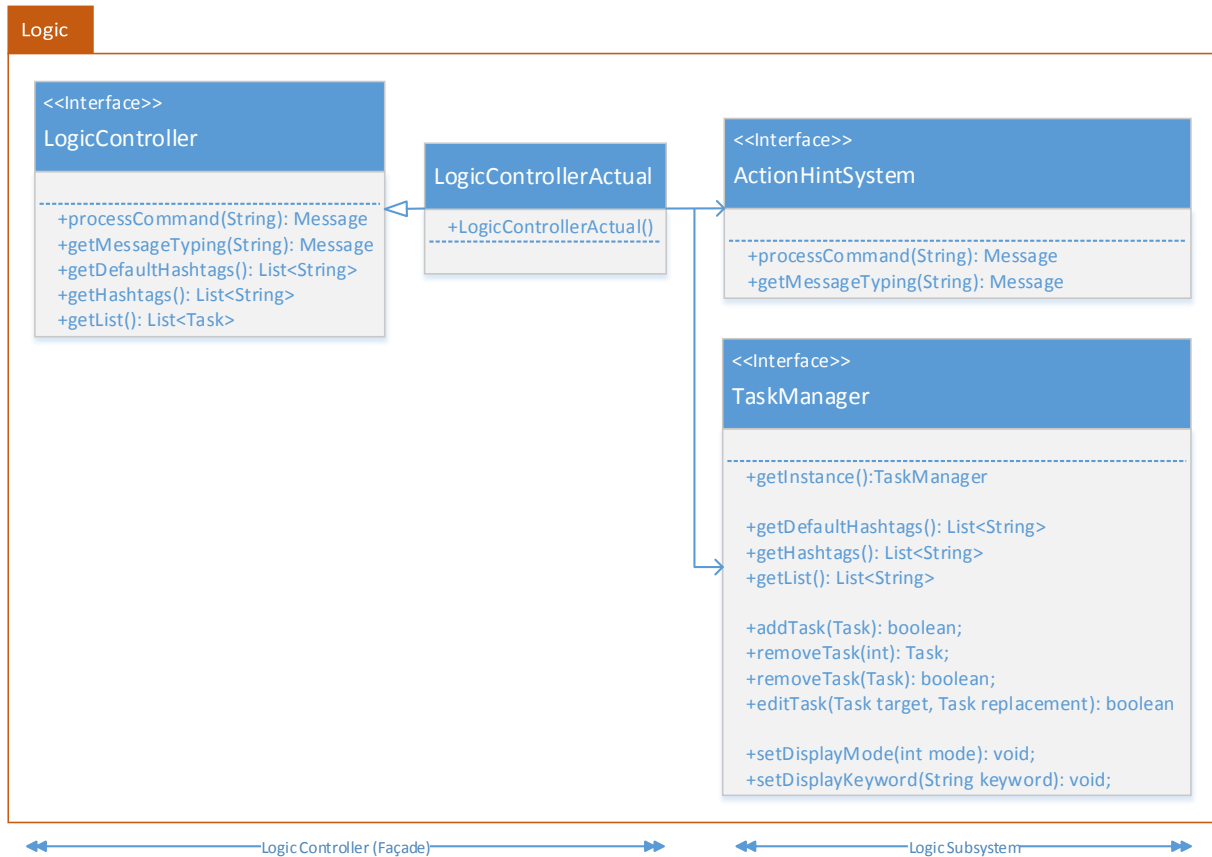
Figure 4 – Class Diagram of Logic Component

The *Logic* component is based on the Façade pattern. The *Logic Controller* abstracts the complexities of the *Logic Subsystem* from the GUI by acting as an intermediary.

Note: Only critical APIs are shown in this class diagram.

3.2.1 Action and Hint System
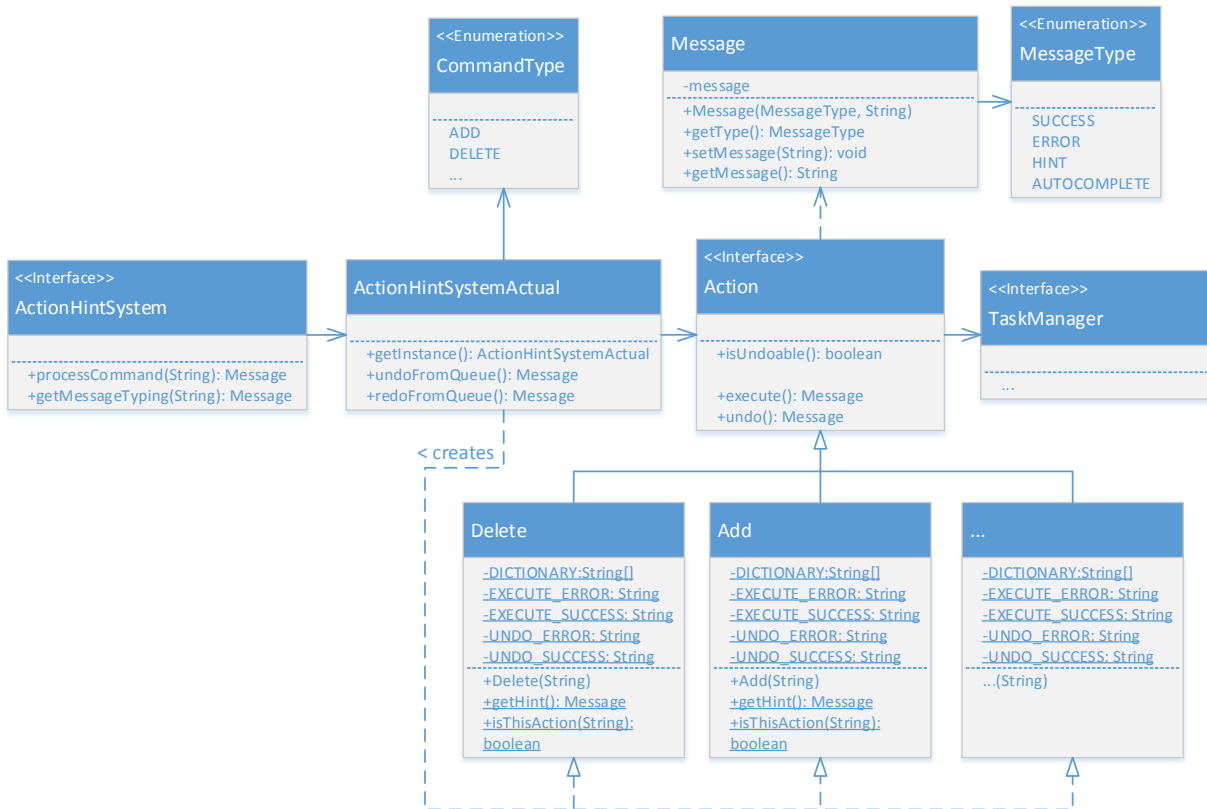
Figure 5 - Action and Hint System

The Action and Hint System applies the <u>Command pattern</u>. It provides two main API methods to handle execution of commands, and generation of hint and autocomplete messages.

> Note: Only critical APIs are shown in this Class Diagram. Dependencies on static libraries like the TaskCatalystCommons are not shown.

3.2.1.1 Executing Commands

The *ActionHintSystemActual* class parses and creates commands in the form of Action objects. These Action objects, if undoable, are stored in a history stack. These actions can then be undone or redone by calling the `undoFromStack()` and `redoFromStack()` methods.

Each subclass of *Action* encapsulates a complete description of how an operation is performed:

Field / Method	Description
DICTIONARY: String[]	All commands associated with this action.
isThisAction(String)	Static method for matching dictionary.
EXECUTE_ERROR, EXECUTE_SUCCESS	Status messages for execution.
UNDO_ERROR, UNDO_SUCCESS	Status messages for undo function, if undoable.
execute()	Code for executing the action.
undo()	Code for undoing the action.
isUndoable()	Instance method for checking if action is undoable.

Table 1 - Action Class Summary

Hint: To add functionality to the program, you simply have to create a new a new *Action* subclass, and add it to *ActionHintSystemActual*. For the example below, you can refer to Delete.java to supplement your understanding.

An abridged example of how the *Delete* operation is carried out is outlined in the following sequence diagram:
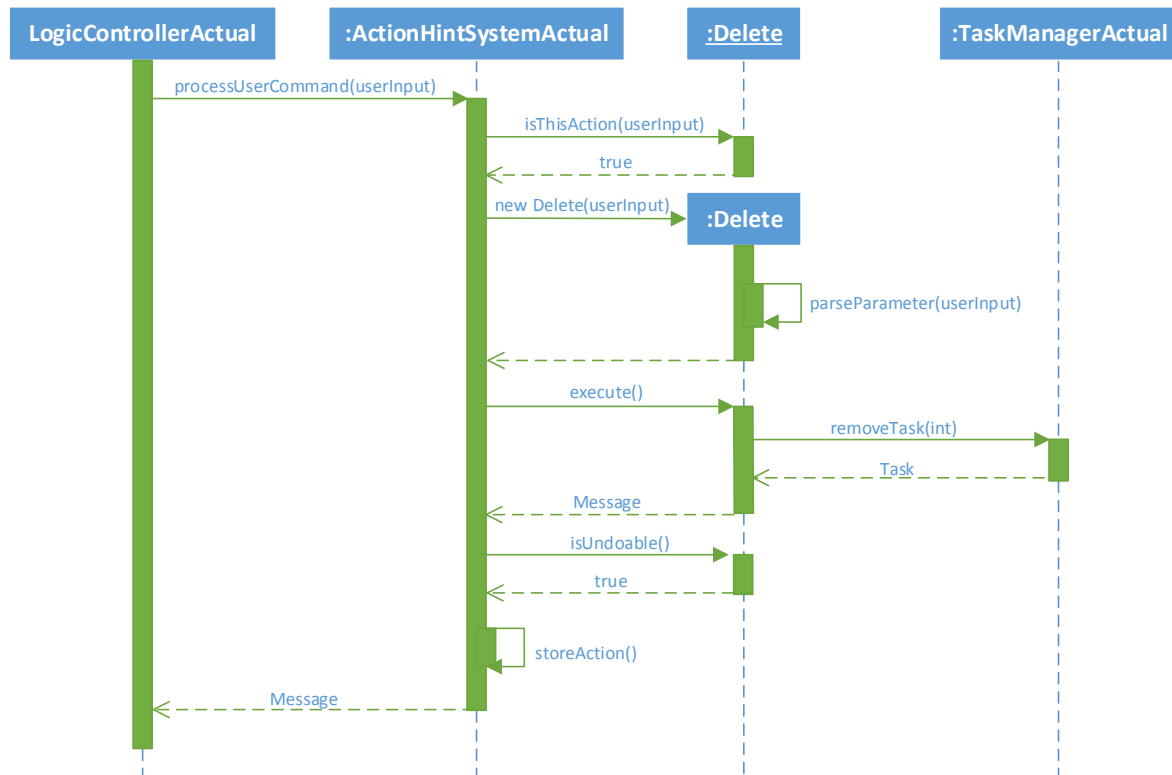
Figure 6 - Sequence Diagram for Delete Action

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

When *LogicControllerActual* requests for a command to be processed, *ActionHintSystemActual* first calls the `isThisAction()` methods of all *Action* subclasses until a match is found.

Since `Delete.isThisAction(userInput)` is true, a *Delete* object is created and the entire user input is passed to its constructor for further parsing. In this case, the task number is extracted from the user input.

Next, the `execute()` command is called. The *Delete* object gets the instance of the *TaskManager*, and calls the `removeTask(int)` method. The *Task* removed will be returned if it exists. By checking if the *Task* is null or not, the *Delete* object can decide whether it should return an error or success *Message*.

Assuming that *Task* is not null, its reference is stored and a success *Message* is returned to the caller. *ActionHintSystem* then checks if the task is undoable, which is true in this case. The *Delete* object is stored into the undo stack, and the *Message* is returned to *LogicController*.
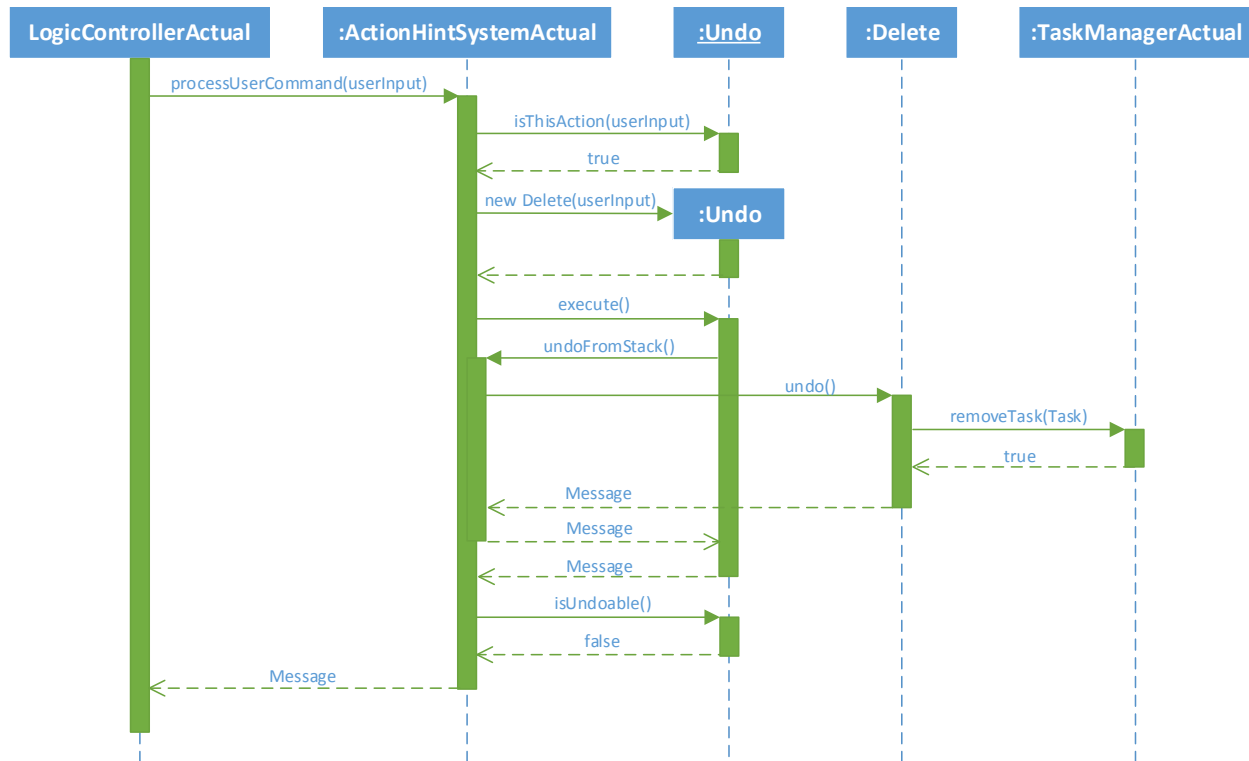
Figure 7 - Sequence Diagram for Undo Action

When undoing the previous command, an *Undo* object is created in the same fashion as the *Delete* object.

When the execute() method is called, the *Undo* object gets the instance of the *ActionHintSystem* and calls the undoFromStack() method. This causes the undo() method of the *Delete* object to be called, which generates a *Message* that is eventually returned to the *LogicController*.

Notice that since the *Undo* action is not undoable, it is not stored in the undo stack of *ActionHintSystem*.

Note: By convention, when implementing an action that is not undoable, the undo() method should return an error Message object.

3.2.1.2 Generating Hint and Autocomplete Messages
The *GUI* relies on the *Action and Hint System* to generate hint messages while the user is typing. This is done by passing the entire command to the getMessageUserTyping() method. The *Action and Hint System* would then generate the corresponding *Message* objects to either display a hint or perform an autocomplete operation.

A message object encapsulate the following information:

Field / Method	Description
message: String	All commands associated with this action.
type: MessageType	Static method for matching dictionary.
getType(): MessageType	Returns the message type.
getMessage(): String	Returns String stored in the message.

Figure 8 – Message Class Summary

The execute() and undo() methods of *Action* objects generate status *Message* objects with the SUCCESS and ERROR types, which are meant to be displayed in the *GUI's* status bar after commands.

On the other hand, the getHint() method of Action objects generate *Messages* of HINT and *AUTOCOMPLETE* types. Hints are displayed on the status bar like success and error messages, while autocomplete prompts the *GUI* to replace the user's input bar with the encapsulated message.

The *Action and Hints System* generates hints for partial command matches, as well as hints specific to a command if there is a match. The following flow chart illustrates the hint generation process:
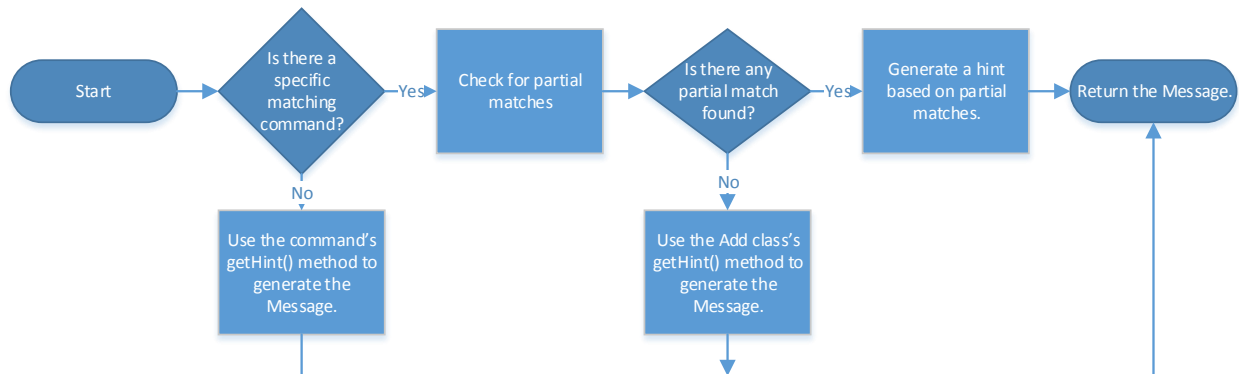
Figure 9 - Hint Generation Flow Chart

With the exception of *Edit* and *Add*, the getHint() methods of most commands generate static hints. *Edit* can return AUTOCOMPLETE *Messages*, while *Add* implements the *Live Task Preview* system.

Autocomplete and Live Task Preview

Figure 10 - Edit Autocomplete Flow Chart

Hint: Look in Edit.java to see the exact implementation of each conditional in the decision tree.

The above diagram shows the decision tree used by the `getHint()` method of the *Edit* Action. If the specified task exists, an AUTOCOMPLETE message is generated by pulling the *Task* from the *Task Manager* and appending its full description behind the command.

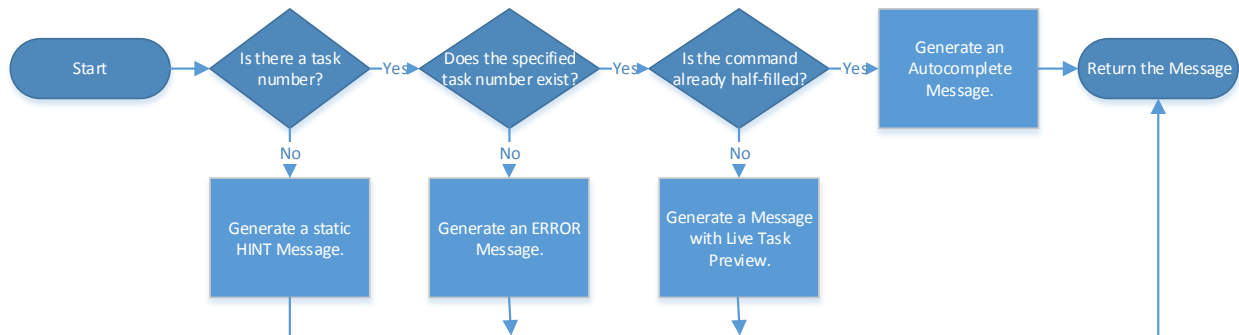Note: When generating AUTOCOMPLETE *Messages*, make sure it contains the exact command the user should type. For example, the parameter "edit 2 " should generate an AUTOCOMPLETE *Message* containing "edit 2 Meet boss at 5PM", and not simply "Meet boss at 5PM". Also, make sure to use `getTaskDescriptionEdit()` from the *Task* object to preserve ignore tags (explained in the parsing section below).

If the specified *Task* exists, and the command is already filled in, then Live Task Preview messages will be generated. These are messages of type HINT, which makes use of parsing libraries contained in *TaskCatalystCommons* to generate a preview of the system's NLP (Natural Language Processing) interpretation of the command.

Live Task Preview messages are also the main type of *Messages* generated by the *Add Action*. *Task* parsing and building will be discussed in the next section.

3.2.1.3 Adding Tasks
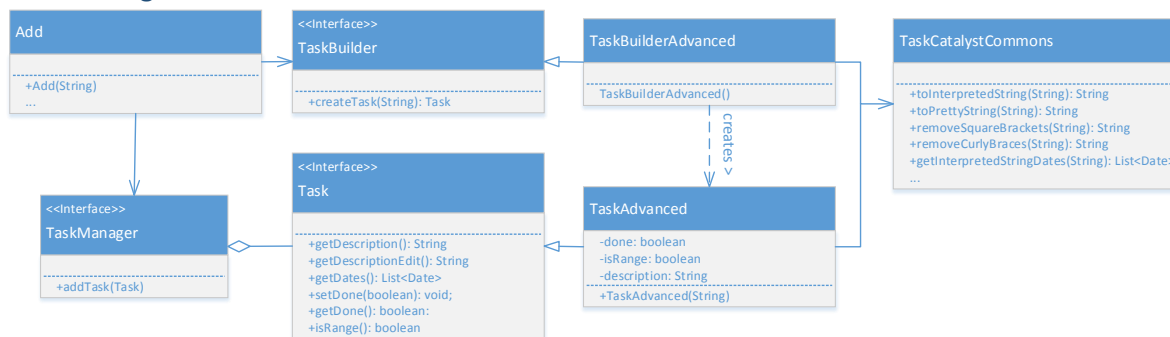
Figure 11 - Class Diagram for Add Action

The *Task Builder* is used by the *Add* action to parse and create *Task* objects. As the project implements the "Natural Bucket", there is a requirement for flexibility in command. The system makes use of the *PrettyTime* NLP library to recognize date and time formats. However, its behavior is inconsistent across various scenarios. There is also a need to have Relative Date Display. Therefore, the solution is to convert a *Task* description to something that is more easily understood, parsed and displayed later on.

An *Add* object passes the user input to *Task Builder*, which in turn sends it to *TaskCatalystCommons* for parsing. The following table shows an example of converting a *Task* description into a format that is more easily handled by the displaying function later on:

Process	Interpreted Input	Parsing Input
Original User Input	Meet client in MR5 at 5pm to 6pm. Phone number 91234567.	
Ignore all number strings longer than 4 digits.	Meet client in MR5 at 5pm to 6pm. Phone number [91234567].	

Ignore all words ending with a number.	Meet client in [MR5] at 5pm to 6pm. Phone number [91234567].	
Remove all ignored words for the Parsing Input.		Meet client in at 5pm to 6pm. Phone number.
Remove all PrettyTime buggy words for the Parsing Input.		Meet client 5pm to 6pm. Phone number.
Remove consecutive "and", "on" and whitespaces.		Meet client 5pm to 6pm. Phone number.
Send Parsing Input to PrettyTime, and replace each match that has absolute word boundaries and are outside of square brackets in Interpreted Input.	Meet client in [MR5] {12 Oct 2014 05:00 PM} to {12 Oct 2014 06:00 PM}. Phone number [91234567].	
Remove all prepositions before each date.	Meet client in [MR5] {12 Oct 2014 5PM} to {12 Oct 2014 6PM}. Phone number [91234567].	

The *Interpreted Input* is returned to *TaskBuilder* and stored as the *Task's* Description. Whenever the `getDescription()` method of the *Task* is called, it uses the *TaskCatalystCommons* library to convert it into a friendlier format for displaying.

Note: Square brackets are used to ignore parts, while curly braces are used to denote date and time information.

The process of converting from an Interpreted Input to a Friendly String for displaying is shown below:

Process	Friendly String
Original Interpreted Input	Meet client in [MR5] {12 Oct 2014 05:00 PM} to {12 Oct 2014 06:00 PM}. Phone number [91234567].
Parse items in brackets and replace them with relative dates.	Meet client in [MR5] {today 5PM} to {6PM}. Phone number [91234567].
Remove all square brackets and curly braces.	Meet client in MR5 today 5PM to 6PM. Phone number 91234567.

When there are more than one date in a sentence, the following code snippet is used by the conversion process to determine relative dates and ensure that there is no repeated information (i.e. "Saturday 5PM to Saturday 6PM" instead of "Saturday 5PM to 6PM").

```java
if (!TaskCatalystCommons.isSameDate(previousDate, currentDate)) {
    // Can add some more, like yesterday, last Tuesday, etc.
    if (TaskCatalystCommons.isToday(currentDate)) {
        formatString = "'today'";
    } else if (TaskCatalystCommons.isTomorrow(currentDate)) {
        formatString = "'tomorrow'";
    } else if (TaskCatalystCommons.isThisWeek(currentDate)) {
        formatString = "'on' E";
    } else {
        formatString = "'on' d MMM";
    }
    if (!TaskCatalystCommons.isThisYear(currentDate)) {
        formatString = formatString + " yyyy";
    }
}
if (!TaskCatalystCommons.isSameTime(currentDate, nextDate)) {
    if (!formatString.isEmpty()) {
        formatString = formatString + " ";
    }
    formatString = formatString + "h";
    if (TaskCatalystCommons.hasMinutes(currentDate)) {
        formatString = formatString + ":mm";
    }
    formatString = formatString + "a";
}
SimpleDateFormat formatter = new SimpleDateFormat(formatString);
friendlyUserInput = friendlyUserInput.replace(dateGroups.get(i).getText(),
formatter.format(currentDate));
```

Figure 12 - Friendly Date Conversion Process
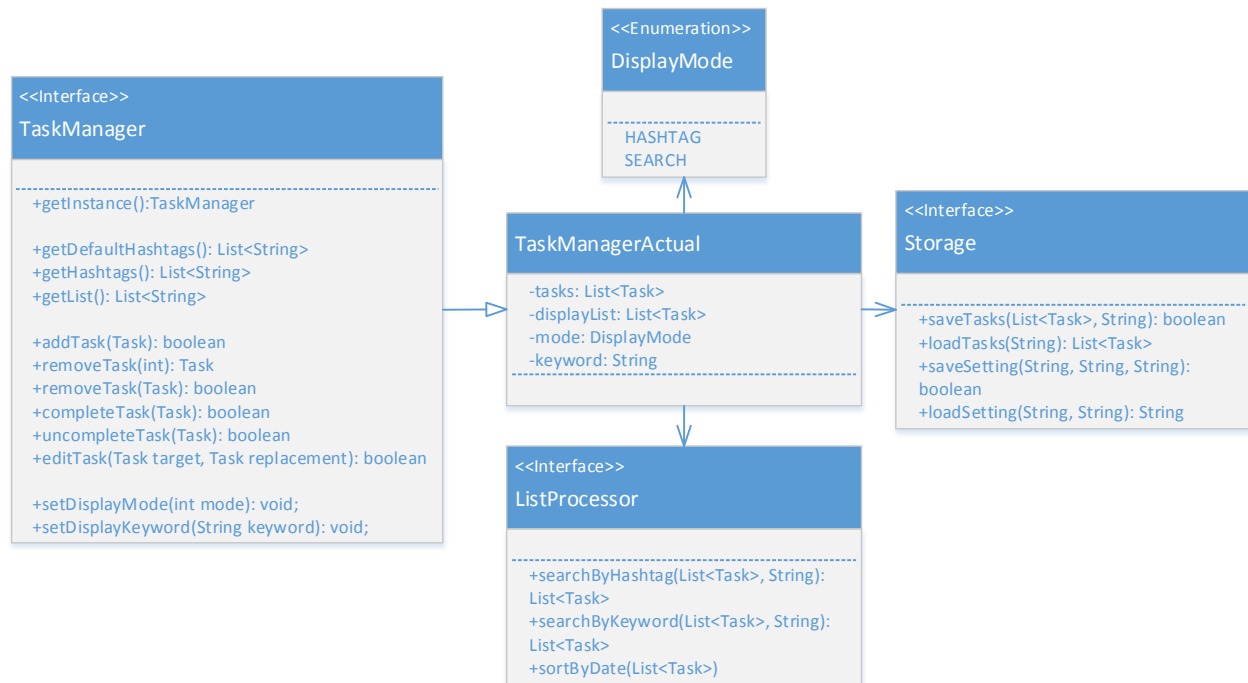
3.2.2 Task Manager

Figure 13 - Task Manager Class Diagram

The *Task Manager* Interface follows the <u>Demeter's Principle</u> closely by ensuring that most common operations can be done using APIs without low-level manipulation of *Tasks*. The *Task Manager* generates the actual *Task* list displayed to the user by keeping track of the last display mode and keyword used by the user. The keyword can be a hashtag or search key depending on the display mode.

TaskManagerActual is responsible for maintaining the full list of tasks, and depends on a *ListProcessor* to generate the display list whenever the *getList()* method is called.

Whenever tasks are added or removed, *TaskManagerActual* automatically sends the whole list of tasks using the *Storage* interface of the *Storage* component.
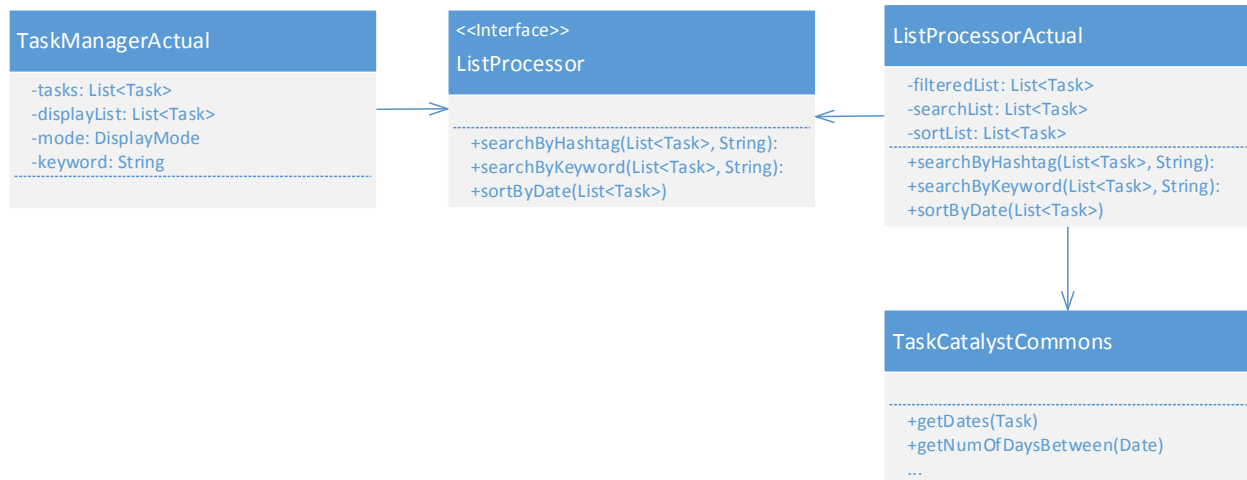
3.2.3 List Processor

Figure 14: Class Diagram of List Processor

ListProcessorActual is responsible for processing the list of tasks which is passed by *TaskManagerActual* according to whether the user wants to search by hashtag or search by keyword.

If the user keys in a keyword he wants to search, `searchByKeyword(List<Task> list, String keyword)` is called by *TaskManagerActual* and *ListProcessorActual* will return a list of tasks that contains the keyword user keyed in.

TaskManagerActual calls `searchByHashtag(List<Task> list, String hashtag)` if the user keys in a hashtag he wants to search and *ListProcessorActual* will return a list of tasks that contains the hashtag user keyed in.

If the user wants to display a default hashtag which is not manually created by the user, the same method is called for *ListProcessorActual* to process the list. The table below is the description of lists which the method will return when a default hashtag is entered.

Default Hashtag	Description of the list returned
#all	Returns a list of tasks which are not completed.
#pri (priority)	Returns a list of tasks which are marked as priority.
#tdy (today)	Returns a list of tasks which are due today.
#tmr (tomorrow)	Returns a list of tasks which are due tomorrow.
#upc (upcoming)	Returns a list of tasks which are due at least two days later.
#smd (someday)	Returns a list of tasks which do not have due date.
#dne (done)	Returns a list of tasks which are completed.

Table 2: Default Hashtags

For the `sortByDate(List<Task>)` method, ListProcessorActual will return a list of tasks which are sorted chronologically to TaskManagerActual when it is called.

3.3 Storage

The Storage Component does the functions of storing task data in the file and loading the data to perform displaying tasks or editing the contents of the tasks. When the data is stored, you need to convert the list of tasks into JSON objects to save in the file. Similarly, you have to converts JSON data of the file to tasks while loading the list of tasks.

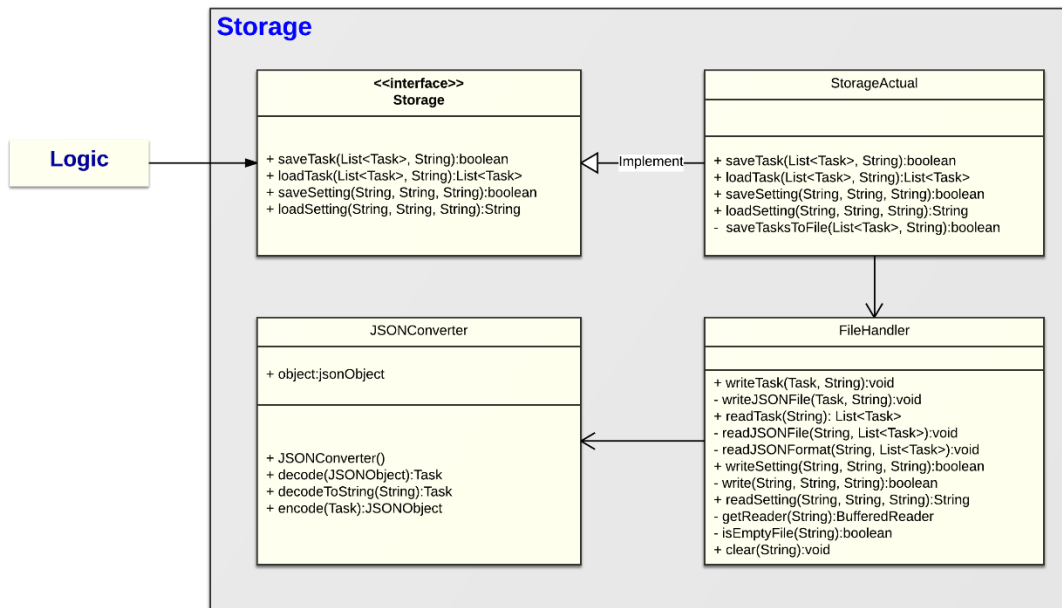The below class diagram demonstrates the structure of the Storage component.

Figure 15 - Class Diagram of Storage Component

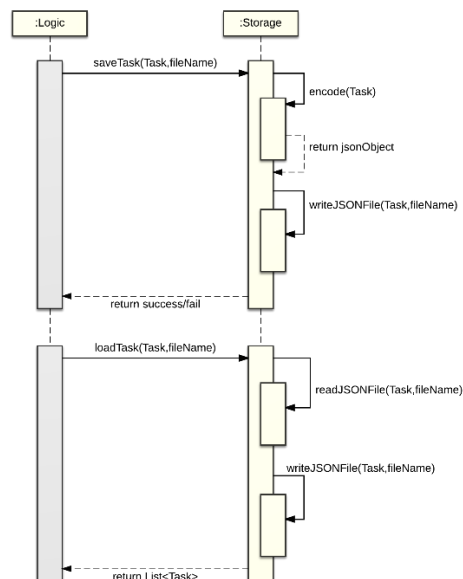The following sequence diagram shows the main functions of saving and loading tasks.

Figure 16 - Sequence Diagram of Saving and Loading Task

4. Testing the System

When developing new functionality, the TDD (Test-Driven Development) approach should be applied.
More information on how to use the TDD approach can be found in the following URL:

http://agiledata.org/essays/tdd.html

JUnit is the main unit testing system used in the project. As the project structure follows the
specifications of the Maven dependency management system, JUnit test cases are stored under the
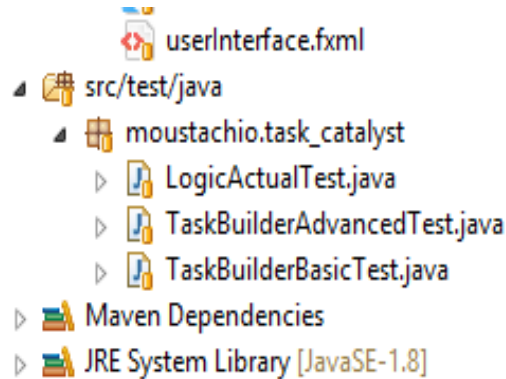/src/test/java directory.

Figure 17 - /src/test/java Directory

To create a new JUnit test case, right click on the project package, and select **New > JUnit Test Case**.
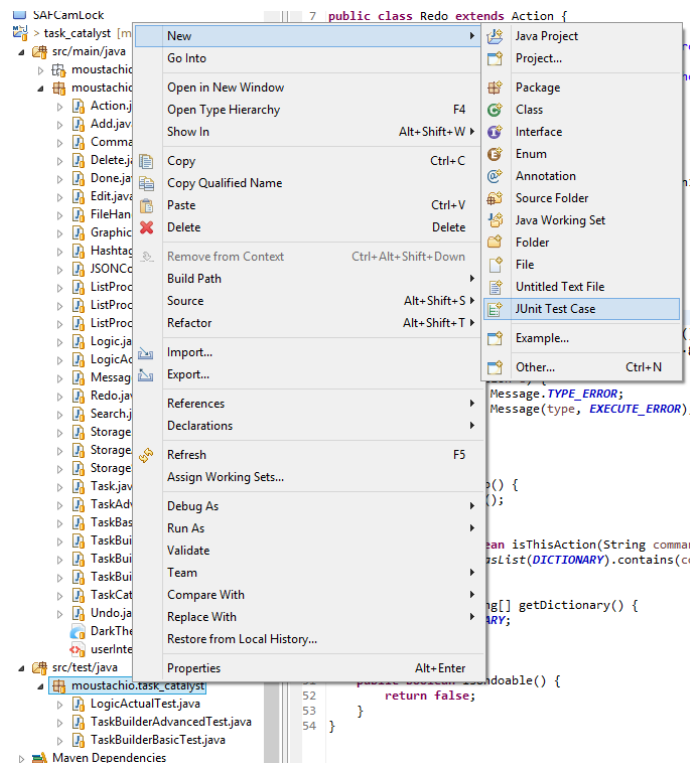

Figure 18 - Creating a new JUnit Test Case

Ensure that your test case follows the naming convention of *ClassName*Test where *ClassName* is the name of the Class Under Test. Also, ensure that JUnit 4 is in use, and the correct class is selected for the "Class under test" field.
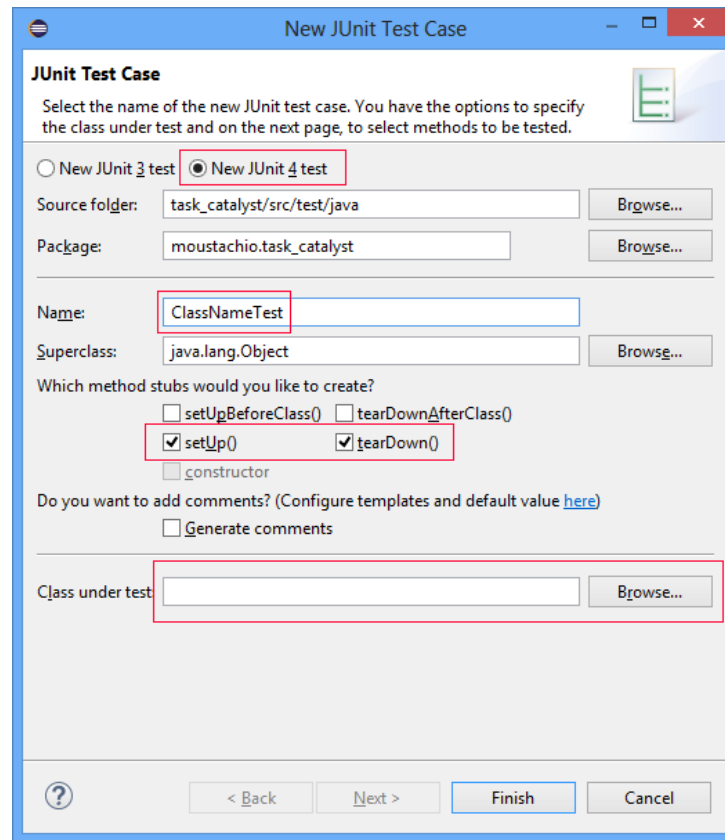

Figure 19 - Creating a new JUnit Test Case

The setUp() and tearDown() methods are called before and after respectively after each test case. Use setUp() to instantiate an instance of the Class Under Test, and tearDown() to perform any cleaning up operations. An example is shown below:

```java
TaskBuilder taskBuilder;
@Before
public void setUp() throws Exception {
	taskBuilder = new TaskBuilderAdvanced();
}
@After
public void tearDown() throws Exception {
}
// Test for basic date recognition.
@Test
public void tc1() {
	Task task = taskBuilder.createTask("Meet boss 21 Jun 10:05am");
	assertEquals("Meet boss on 21 Jun 10:05AM",
task.getDescriptionEdit());
}
…
```

You can write test cases as shown in the above code. When using TDD, remember to create the smallest test case possible, and pass each test case using the simplest code. You can create additional test cases simply by prefixing them with the @Test directive.

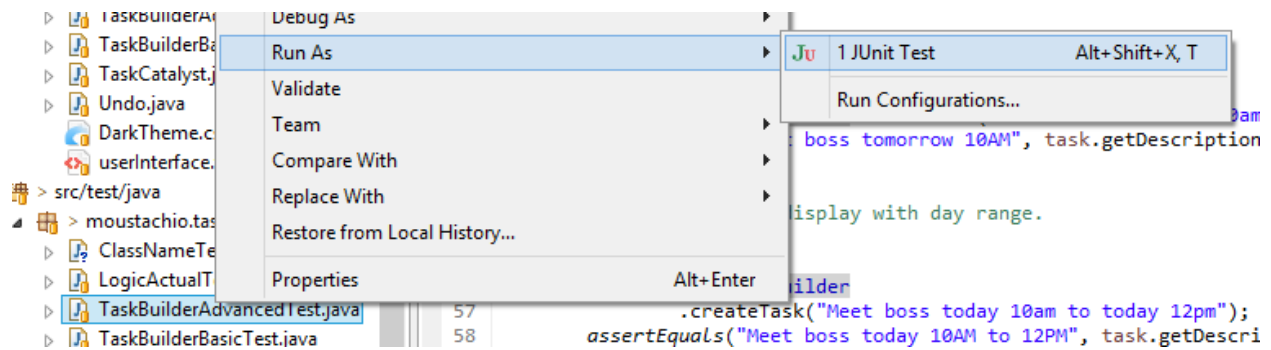Simply right click the test case and select **Run as > JUnit Test** to run the test.

Figure 20 – Running the JUnit Test