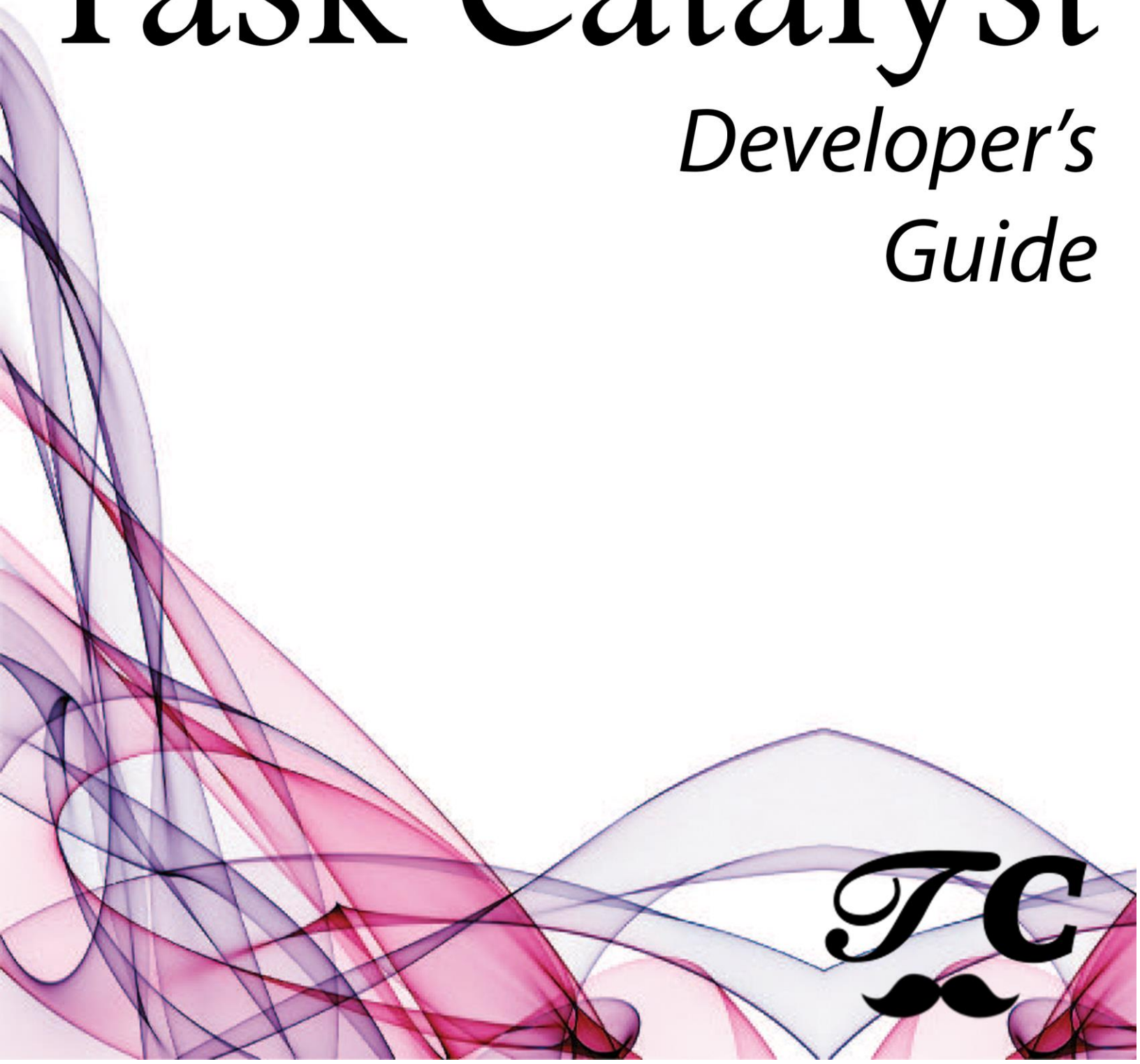
Task Catalyst

Developer's Guide



Contents

1. Introducing the Developer's Guide 2
2. Defining the Architecture 3
3. Developing the Components 4
3.1 Graphical User Interface 4
3.2 Logic 6
3.2.1 Action and Hint System 7
3.2.2 Task Manager 16
3.2.3 List Processor 17
3.3 Storage 19
4. Setting up the Environment 21
4.1 Using the Versioning Tool 21
4.2 Setting up the IDE 22
4.3 Testing the System 24
5. Upcoming Developments 27
6. Appendix 28
6.1 Glossary 28

1. Introducing the Developer's Guide

Our Audience

Task Catalyst is a lightweight, cross-platform application that caters to the modern urban crowd with a busy lifestyle. The program is optimized for keyboard-use, and hence provides for many command styles and familiar Social Media-inspired features like hashtagging for organization.

Our Vision

The design of Task Catalyst is based on the *Natural Bucket*, which focuses to make Task Catalyst:

- User Friendly and Intuitive
- Simple yet Powerful
- Accessible

Using this Guide

This guide will walk you through the basics required to maintain and develop Task Catalyst. First, you will be introduced to the **High-Level Architecture (Section 2)** of the program. Next, we will talk about the **System Components (Section 3)** from front-end to back-end. Each component will be introduced top-down using its class diagram and APIs, and then further elaborated with behavioral diagrams and code samples if necessary. Finally, we will orientate you to the **Development Infrastructure (Section 4)** of this project.

This guide assumes that you have some prior experience in Java and CSS.

Throughout the guide, we will be using the following markups to improve readability:

- *Class, Component, Library or Framework*
- Pattern or Principle
- `Commands, Code or Input/Output`

2. Defining the Architecture

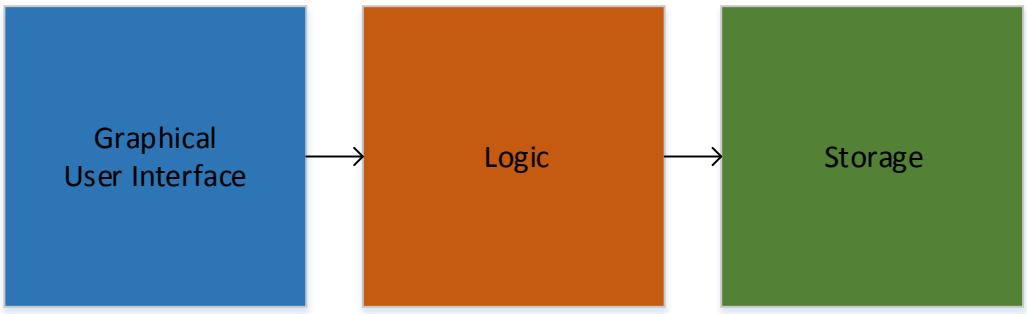


Figure 1 – Architecture Diagram

The overall architecture is illustrated in **Figure 1**. It is designed around the MVC (Model-View-Controller) 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.

GUI (Graphical User Interface) 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 and status messages. It is also responsible for many interactive features like hotkeys and autocomplete. It relies on the Logic component for command execution, low-level decision-making and data processing.

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

Storage is responsible for persistent physical storage. Its functionalities include *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

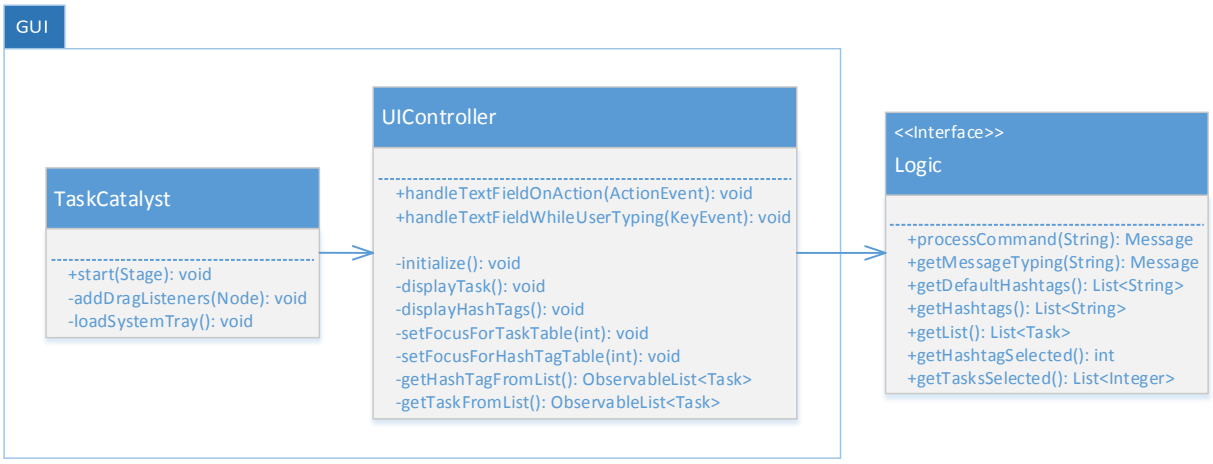


Figure 2 – Class Diagram of GUI Component

GUI was designed using *JavaFX Scene Builder*. The class diagram of the component is shown in **Figure 2**. *UIController* implements the Observer pattern internally to control the display elements and communicate with *Logic*.

The interactions between the *User*, *GUI* and *Logic* during initialization is depicted in **Figure 3**.

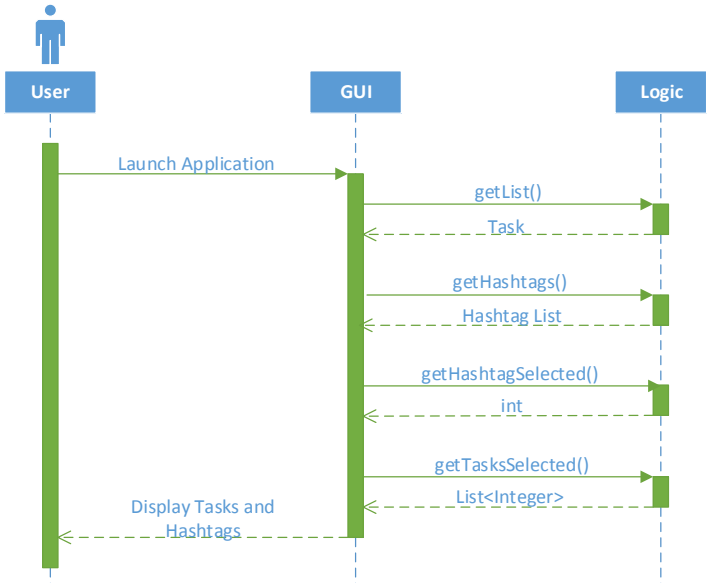


Figure 3 – Sequence Diagram for Initialization

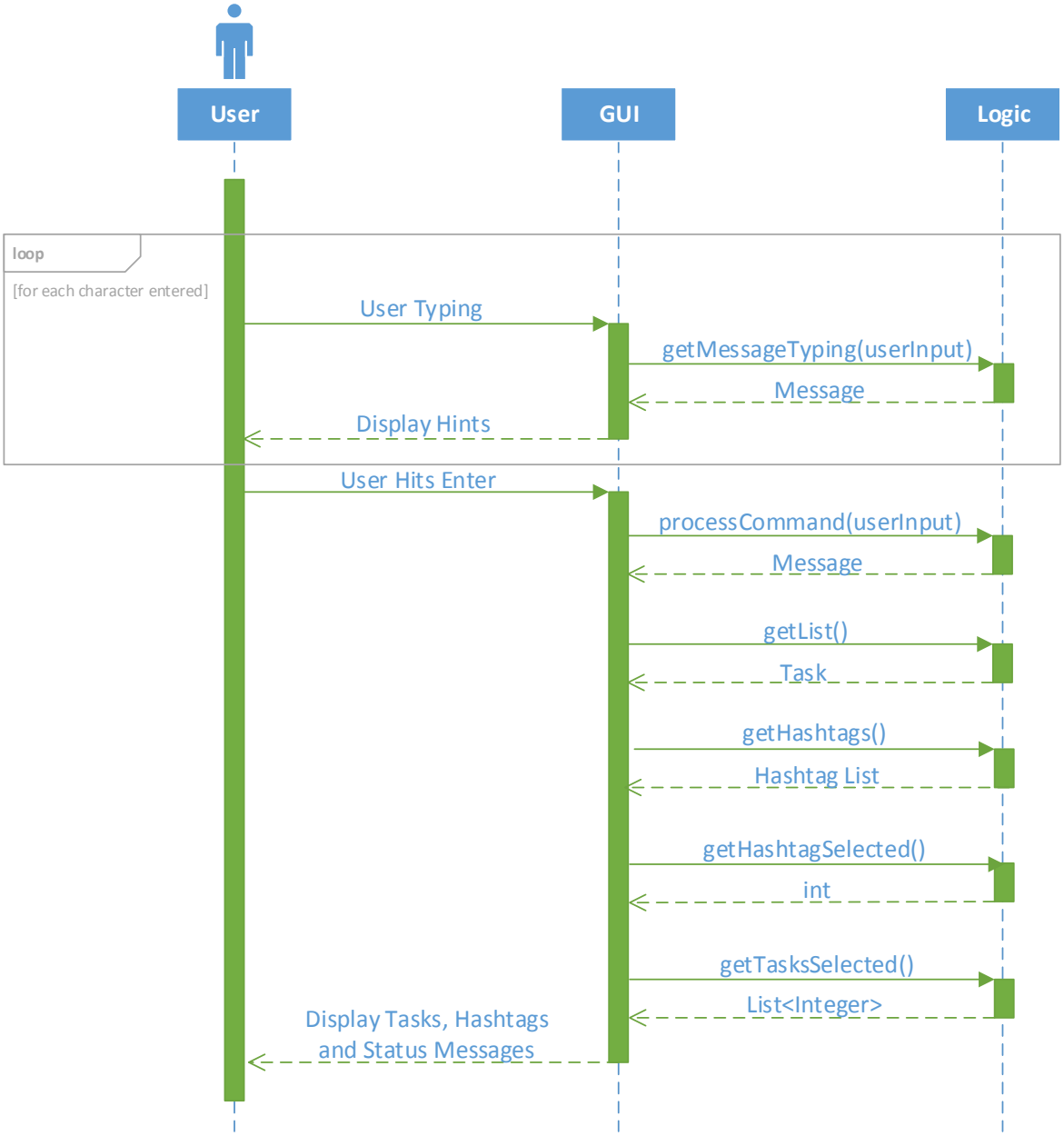


Figure 4 – Sequence Diagram for User Interactions

The standard sequence for generating hints and command execution is depicted in **Figure 4**. Each character entered will trigger the listener for the text field, which passes the user input to logic to generate a new hint. When the user confirms the command, the entire command string is sent to *Logic* without any preprocessing in the *GUI*.

Note:	The **Hashtag** and **Task** lists need to be refreshed with most successful commands, with the exception of repeated search or repeated category selection. Therefore, the Observer Pattern is not required between *Logic* and *GUI*.

3.2 Logic

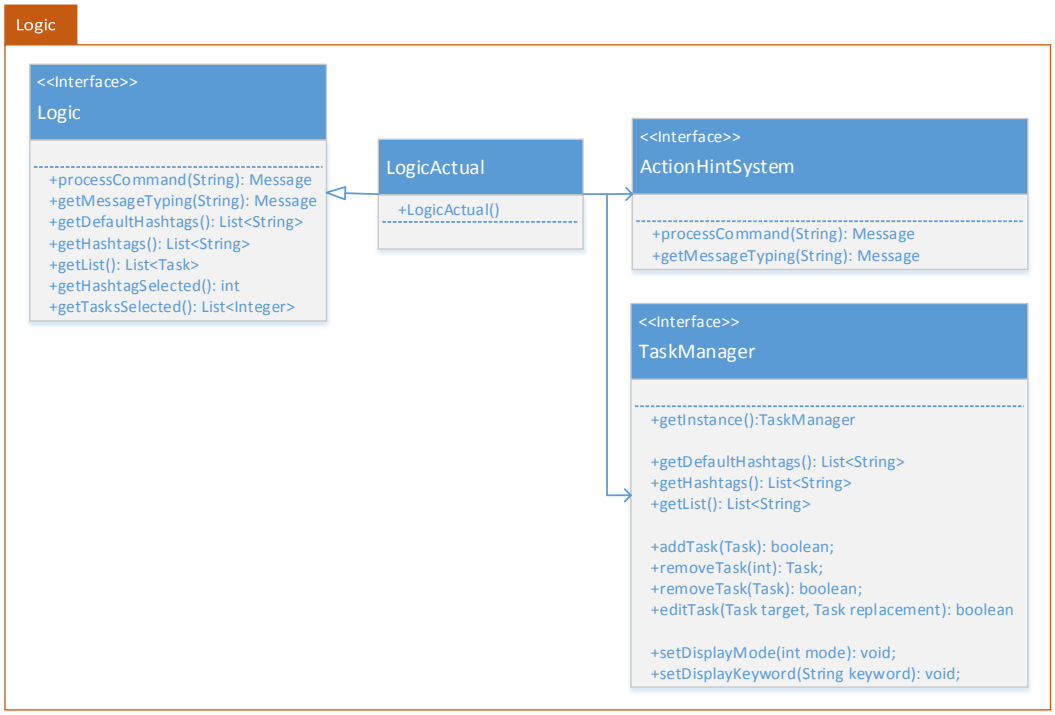


Figure 5 – Class Diagram of Logic Component

Logic is based on the Façade pattern. *Logic Controller* abstracts the complexities of the *Logic Subsystem* from the *GUI* by acting as an intermediary. The *Logic* class diagram is shown in **Figure 5**.

Logic's role is to provide all necessary backend functionality for the *GUI*, including command parsing, hints generation, list processing, and display state maintenance. These functionalities are achieved by relaying method calls to *ActionHintSystem* and *TaskManager*.

A quick overview of the methods specified by the *Logic* interface is shown below:

Field / Method	Description
`processCommand(String): Message`	Parses, interprets, and executes a user command.
`getMessageTyping(String): Message`	Generates a dynamic hint based on the current user command.
`getHashtags(): List<String>`	Returns the list of hashtags.
`getList(): List<Task>`	Returns the list of Task objects.
`getHashtagsSelected(): int`	Returns the hashtag index that should be selected (or highlighted).
`getTasksSelected(): List<Integer>`	Returns the indices of tasks that should be selected (or highlighted).

3.2.1 Action and Hint System

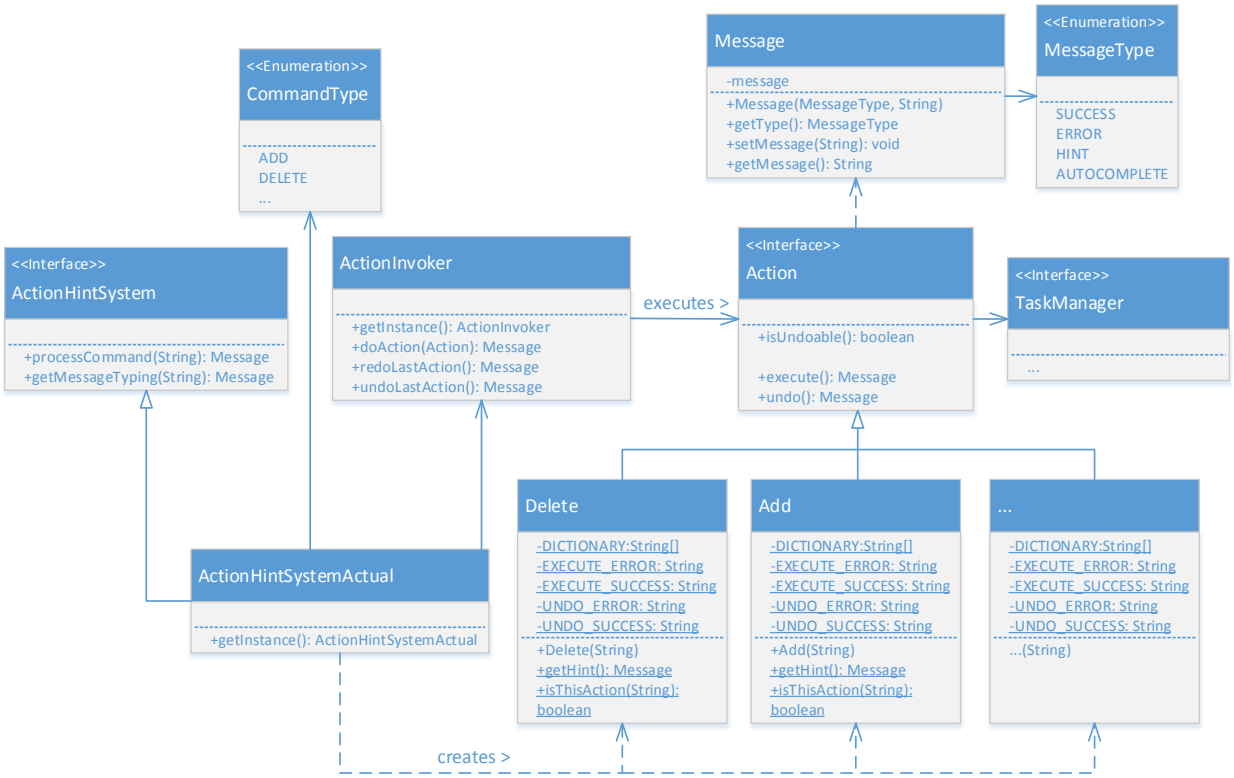


Figure 6 - Action and Hint System

ActionHintSystem applies the Command pattern. As shown in **Figure 6**, it provides two main API methods to handle execution of commands, and generation of hints and autocomplete messages.

Note:	Only critical APIs and relationships are shown in this class diagram. Dependencies on static libraries like the *TaskCatalystCommons* are not shown.

ActionHintSystemActual is responsible for interpreting and creating *Action*s from user commands. *Action*s encapsulates a complete specification of a command (to be elaborated in the next section). These *Action*s are passed to *ActionInvoker* for execution. The *ActionInvoker* is also responsible for maintaining command stacks for undo/redo functionality.

The *ActionInvoker* is a Singleton class as there should only be one command queue operating on the *Task* list at any instance of time.

A quick overview of the methods specified by the *ActionHintSystem* interface is shown below:

Field / Method	Description
`processCommand(String): Message`	Parses, interprets, and executes a user command.
`getMessageTyping(String): Message`	Generates a dynamic hint based on the current user command.

Action Class – Parsing and Executing Commands

ActionHintSystemActual parses user inputs and creates commands in the form of *Action* objects. These *Action* objects are sent to *ActionInvoker* for execution and, if undoable, are stored in a history stack. These *Action*s can then be undone or redone by calling the `undoFromStack()` and `redoFromStack()` methods of *ActionInvoker*.

Action implements the Template pattern. Each subclass of *Action* encapsulates a complete description of how an operation is performed and how hints are generated. Even though it is not specified in the *Action* interface, it is compulsory to implement various static methods for each *Action* subclass.

A summary of all mandatory methods and fields are shown below.

Field / Method	Description
`DICTIONARY: String[]`	All commands associated with this *Action*.
`EXECUTE_ERROR, EXECUTE_SUCCESS`	Status messages for execution.
`UNDO_ERROR, UNDO_SUCCESS`	Status messages for undo, if undoable.
`HINT_MESSAGE`	The hint message to return when `getHint()` is called.
`execute(): Message`	Code for executing the *Action*.
`undo(): Message`	Code for undoing the *Action*.
`isThisAction(String): boolean`	Static method for matching entries in the dictionary.
`getHint(String): Message`	Static method for generating a *Message* hint based on the input string.
`isUndoable(): boolean`	Static method for checking if the action is undoable.

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 following example, you can refer to Delete.java to supplement your understanding.

An abridged example of how the *Delete* operation is carried out is outlined in **Figure 7**.

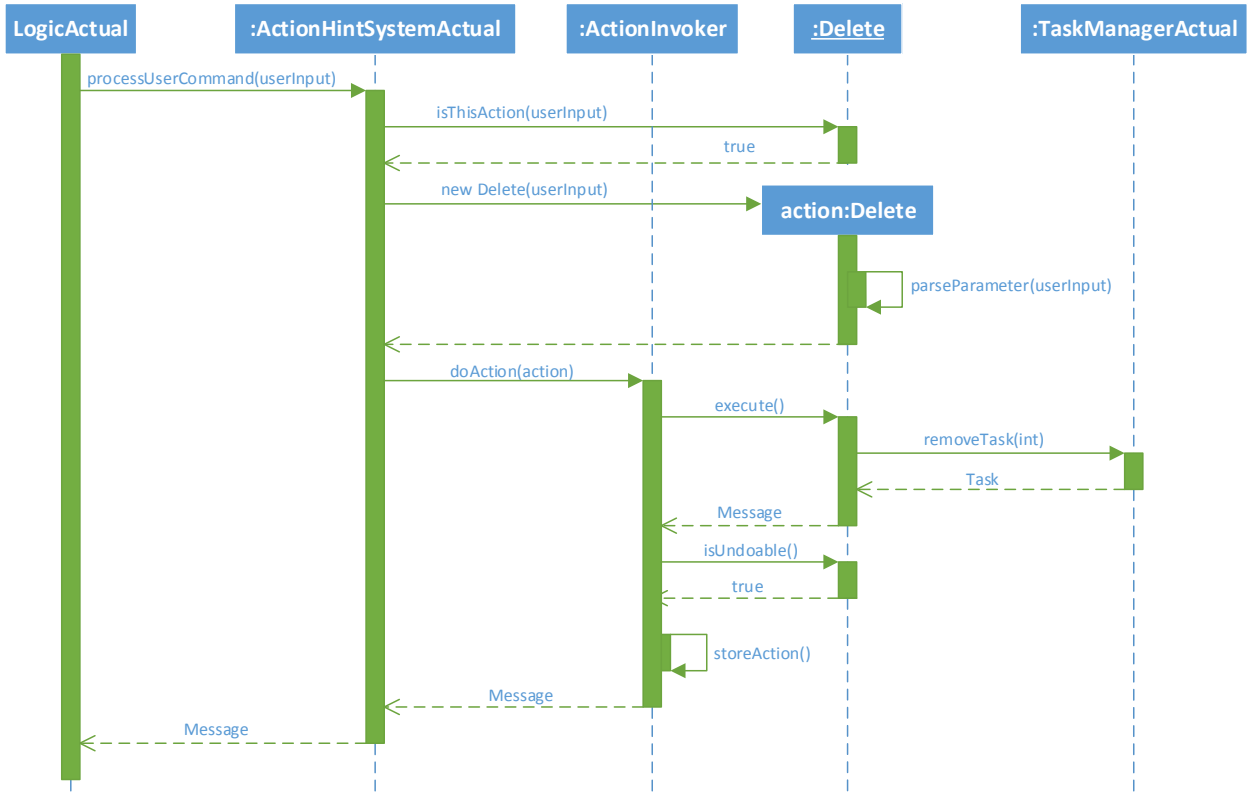


Figure 7 – Sequence Diagram for Delete Action

Whenever *LogicActual* requests for a command to be processed, *ActionHintSystemActual* first calls the `isThisAction()` methods of all *Action* subclasses until a match is found. An instance of the matching *Action* is created and the entire user input is passed to its constructor for further parsing.

All actions are executed by passing it to the *ActionInvoker*, which will also maintain the undo and redo stacks. *ActionInvoker* stores the actions based on whether it is undoable.

Upon completion of the *Action*, the *Message* is returned and forwarded back to the *GUI*.

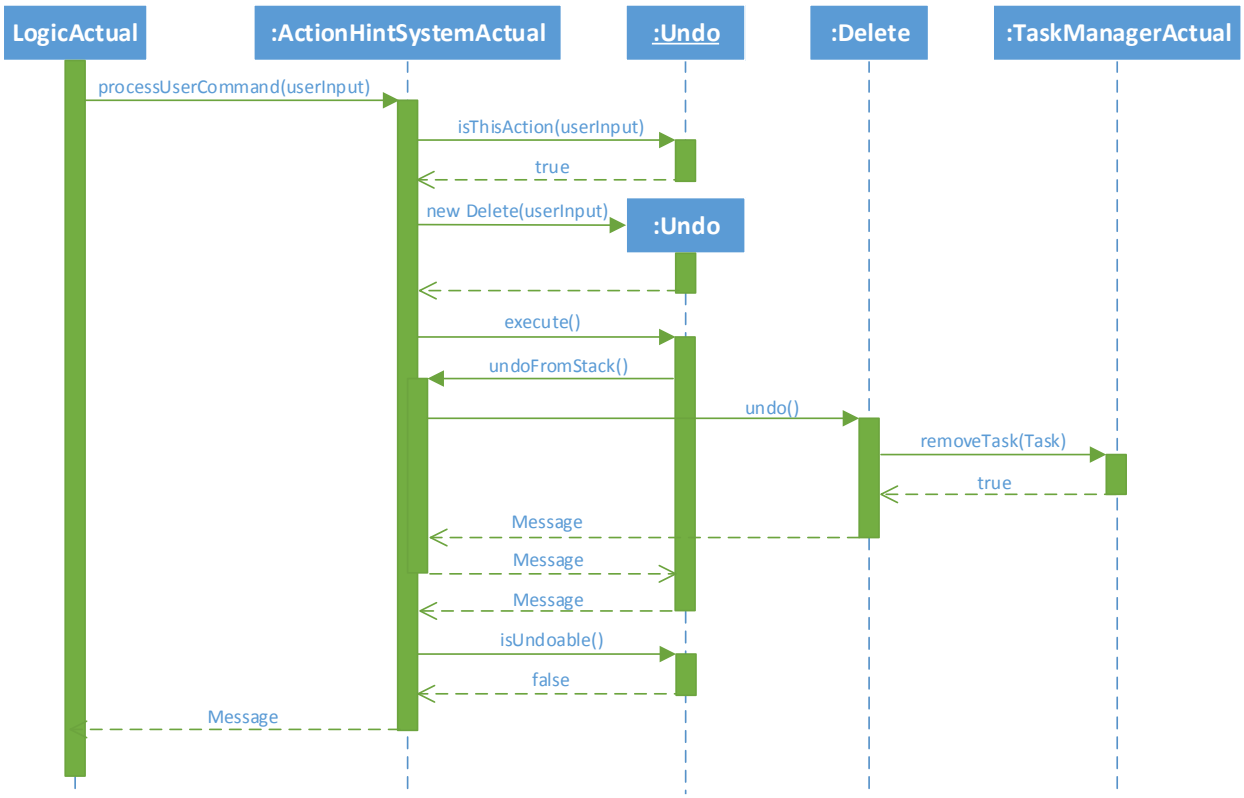


Figure 8 – Sequence Diagram for Undo Action

Note:	The `undo()` method of *Delete* is omitted, but the steps are similar to how it is executed. Please refer to the actual code for more information.

Figure 8 illustrates the process of undoing an *Action*. When undoing the previous command, an *Undo* object is created in the same fashion as the *Delete* object.

Upon execution, the *Undo* object gets the instance of the *ActionInvoker* and calls requests for the last action to be undone. *ActionInvoker* then requests for the previous command in the stack to undo itself.

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 a *Message* object with its `type` set to `MessageType.ERROR`.

Message Class – Status Messages, Hints and Autocomplete

GUI relies on *Logic* to generate hint messages while the user is typing. *Logic* relays these requests to *ActionHintSystem* to do the actual processing. By moving the user input through a decision tree, *ActionHintSystem* generates the corresponding *Message* objects to either display a hint or perform an autocomplete operation.

A *Message* object encapsulates the methods and fields shown below:

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.

Messages with their types set to `ERROR` or `SUCCESS` are generated by the `execute()` and `undo()` methods of *Action* objects. These *Messages* are typically displayed at the status bar of *GUI*.

The `getHint()` method of *Action* objects generates *Messages* of `HINT` and `AUTOCOMPLETE` types. `HINT` *Messages* are displayed on the status bar like `SUCCESS` and `ERROR` *Messages*, while `AUTOCOMPLETE` *Messages* prompt the `GUI` to replace the user's input bar with the encapsulated message.

ActionHintSystem generates hints for partial command matches, as well as hints specific to a command if there is a match. **Figure 9** illustrates the hint generation process.

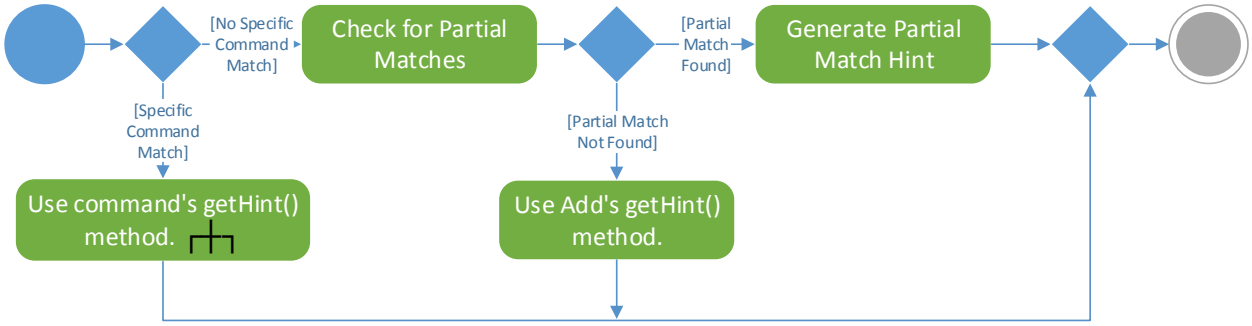


Figure 9 – Hint Generation Activity Diagram

With the exception of *Edit* and *Add*, the `getHint()` methods of most commands generate static hints. The *Edit* hint generation process is depicted in **Figure 10**. It can return `AUTOCOMPLETE` *Messages*, and provide *Live Task Preview* similar to *Add*.

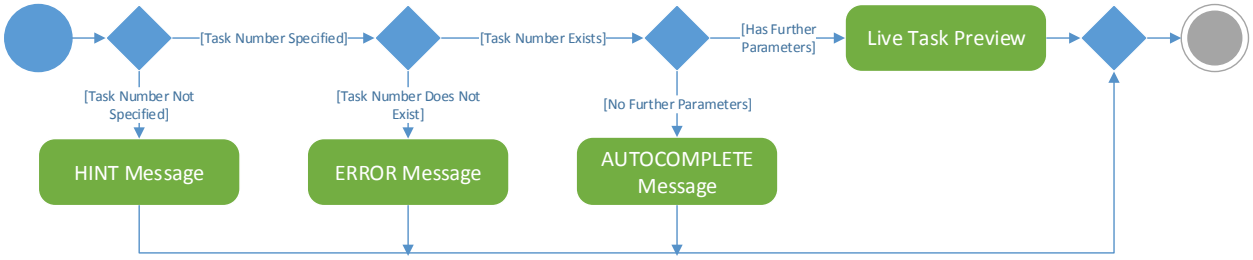


Figure 10 – Edit Hint Generation Process

Note:	When generating an `AUTOCOMPLETE` *Message*, 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).

Add Action – Parsing, Building and Adding Tasks

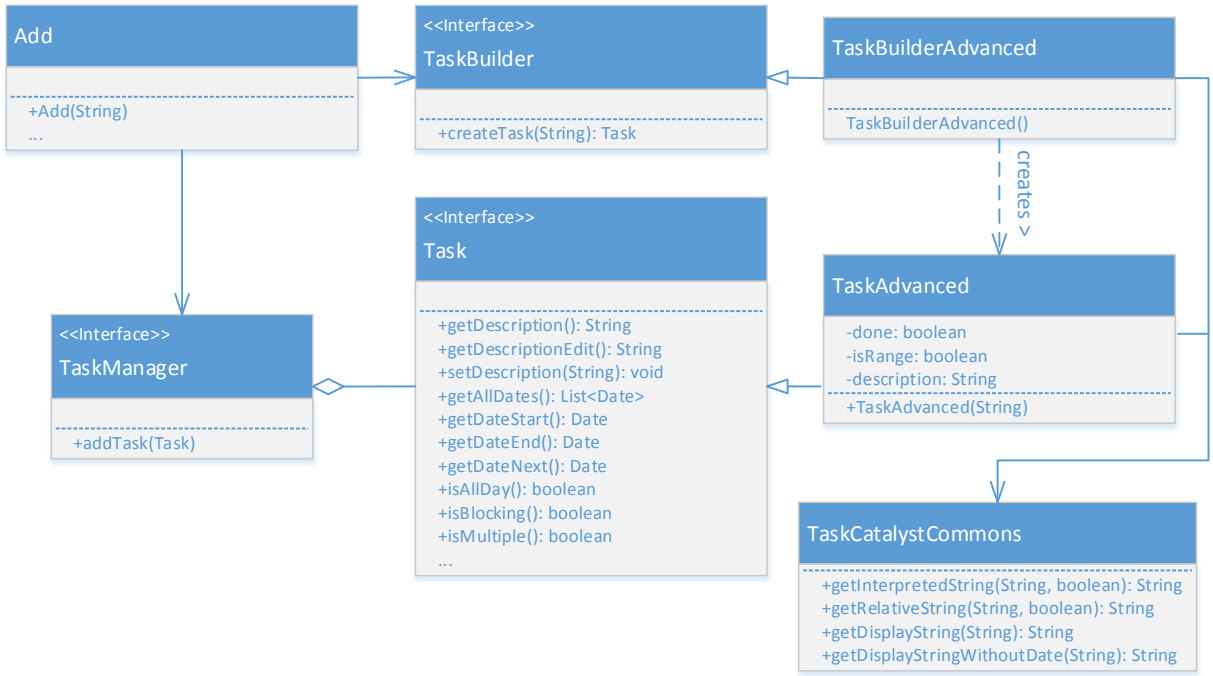


Figure 11 – Class Diagram for Add Action

The high-level structure of *TaskBuilder* is outlined in **Figure 11**. *TaskBuilder* is used by the *Add* action to parse and create *Task* objects. 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* to facilitate *Task Editing* and *Live Task Preview*. Therefore, the solution is to convert a *Task* description to something that is more easily understood, parsed and displayed later on.

When the user wants to add a task, the *Add* object passes the user input to *TaskBuilder*, which in turn uses the parsing libraries in *TaskCatalystCommons*. The parsing process produces an *Interpreted String* which is of the following format:

```
This is a sample task. Some sample dates are {08 Nov 2014 02:00:00 PM}, {08
Nov 2014 03:00:00 PM} and {08 Nov 2014 05:00:00 PM}. [This text is ignored].
```

Notice that each date is stored in absolute format and enclosed in curly braces. The *Interpreted String* can be converted into a *Relative String* for further manipulation or a *Display String* for displaying.

The following table outlines the conversion methods in *TaskCatalystCommons*.

Field / Method	Description
getInterpretedString(String, boolean): String	Converts a *User Input String* into *Interpreted String*.
getRelativeString(String, boolean): String	Converts an *Interpreted String* into a *Relative String*.
getDisplayString(String): String	Converts a *User Input String* into a *Display String* (used for *Live Task Preview*).
getDisplayStringWithoutDate(String): String	Converts a *Relative String* into a *Display String*.

Table 1 shows an abridged example of how user input is converted into an *Interpreted String*. The full process can be found in TaskCatalystCommons.java.

Process	Interpreted Input	Parsing Input
Original ***User Input String***	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 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. The correct prepositions will be generated later.	Meet client in [MR5] {12 Oct 2014 5PM} to {12 Oct 2014 6PM}. Phone number [91234567].	

Table 1 – Interpreted String Conversion Process (Abridged)

The *Interpreted String* is generated by converting the *User Input* into an *Interpreted Input* and *Parsing Input*, and then combining them afterwards. The *Interpreted String* is passed to *TaskBuilder* and used to instantiate a *Task*. Whenever the `getDescription()` method of the *Task* is called, it is converted into a *Display String*.

Note:	Square brackets are used to exclude text from processing, while curly braces are used to denote date and time information.

The process of converting an *Interpreted String* to a *Display String* is shown in **Table 2**.

Process	Display String
Original ***Interpreted String***	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.

Table 2 – Display String Conversion Process (Abridged)

When there are more than one date in a sentence, the following code snippet in **Figure 12** is used by the conversion process to display relative dates and ensure that there is no repeated information (i.e. "`Saturday 5PM to Saturday 6PM`" instead of "`Saturday 5PM to 6PM`"). Whether the date or time should be shown is determined by looking at the previous and next date in the sentence.

```
if (isShowDate) {
	if (isYesterday(currentDate)) {
		formatString = "'yesterday'";
	} else if (isToday(currentDate)) {
		formatString = "'today'";
	} else if (isTomorrow(currentDate)) {
		formatString = "'tomorrow'";
	} else if (isThisWeek(currentDate) && isFirstDate) {
		formatString = "'on' E";
	} else if (isThisWeek(currentDate)) {
		formatString = "E";
	} else if (isFirstDate) {
		formatString = "'on' d MMM";
	} else {
		formatString = "d MMM";
	}
	if (!isThisYear(currentDate)) {
		formatString = formatString + " yyyy";
	}
}
if (isShowTime) {
	if (!isDateEmpty) {
		formatString = formatString + " ";
	}
	formatString = formatString + "h";
	if (hasMinutes(currentDate)) {
		formatString = formatString + ":mm";
	}
	formatString = formatString + "a";
}
```

Figure 12 – Display Date Conversion Process

3.2.2 Task Manager

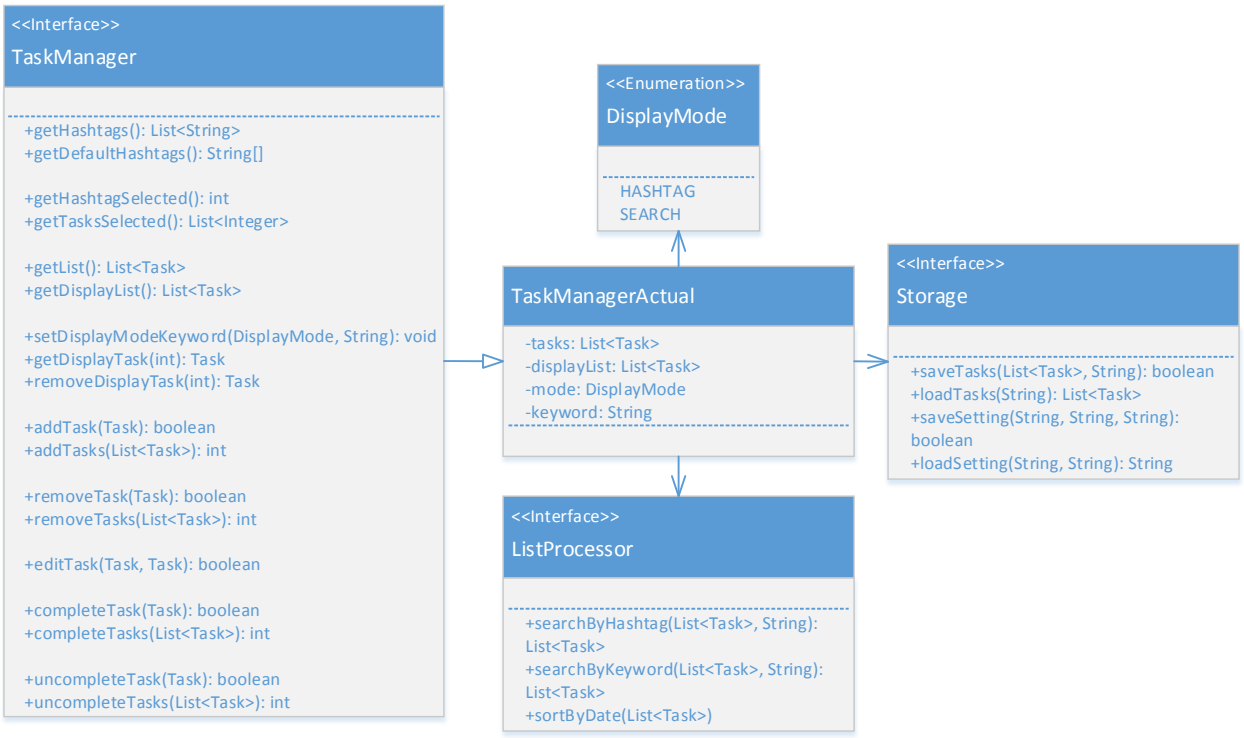


Figure 13 – Task Manager Class Diagram

TaskManager follows Demeter's Principle closely by ensuring that most common operations can be done using APIs without low-level manipulation of *Tasks*. An overview of *TaskManager* is illustrated in **Figure 13**. *TaskManager* 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.

TaskManager is responsible for maintaining the full list of tasks, and depends on *ListProcessor* to generate the display list and identify overlaps whenever the list is refreshed. In addition, it also keeps track of the list of tasks that was most recently modified for the *GUI* to highlight.

Whenever the list of tasks is modified, *TaskManager* automatically sends it to *Storage* for saving.

3.2.3 List Processor

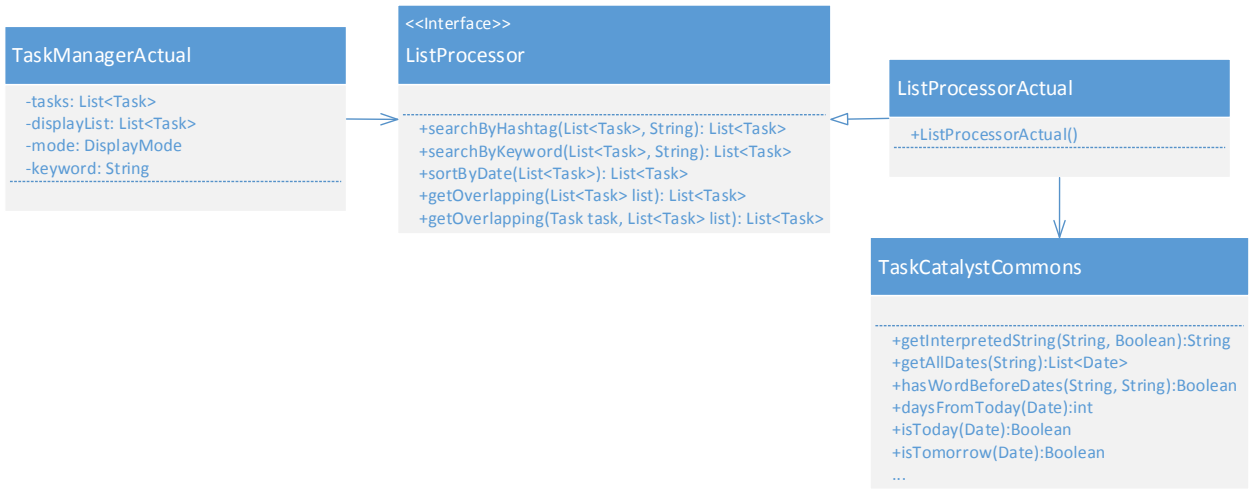


Figure 14 – Class Diagram of List Processor

ListProcessor provides the API for processing the list of *Tasks* passed by *TaskManager*. The structure of ListProcessor is shown in **Figure 14**. *ListProcessor* is responsible for processing the list and provides functionalities like searching by hashtags, searching by keywords, sorting by date, and identifying overlapping tasks.

Field / Method	Description
`searchByHashtag(List<Task>, String): List<Task>`	Returns a list of *Task*s containing the specified hashtag.
`searchByKeyword(List<Task>, String): List<Task>`	Returns a list of *Tasks* matching the specified keyword or dates.
`sortByDate(List<Task>): List<Task>`	Sorts the *Task* list by date.
`getOverlapping(List<Task>): List<Task>`	Returns a list of *Tasks* overlapping with at least one other *Task* in the list.
`getOverlapping(Task,List<Task>): List<Task>`	Returns a list of *Tasks* overlapping with the *Task* specified in the parameter.

When searching for keywords, *ListProcessor* is able to filter the list of *Tasks* based on the specified keyword, dates or date ranges. Example searching formats are summarized in **Table 3**.

Criteria	Example
Keyword	• boss • marketing team
Single Date	• 4 Nov
Multiple Dates	• 30 Oct, 1 Nov, 3 Nov
A Range Of Dates	• 2 Feb to 4 Feb • between 3 Mar and 6 Mar

Table 3 – Example Search Formats

When the user requests to display a hashtag category, the `searchByHashtag(List<Task> list, String hashtag)` method is used. *ListProcessor* returns a list of *Tasks* with the specified hashtag if it is a custom hashtag, or a list of *Tasks* within the specified category if it is a default hashtag.

Table 4 lists the default hashtags used in Task Catalyst.

Default Hashtag	Description of the list returned
#all (All)	Returns a list of *Tasks* which are not completed.
#pri (Priority)	Returns a list of *Tasks* which are marked as priority.
#ovd (Overdue)	Returns a list of *Tasks* which are overdue.
#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 dates.
#olp (Overlapping)	Returns a list of *Tasks* which are overlapping.
#dne (Done)	Returns a list of *Tasks* which are completed.

Table 4 – Default Hashtags

The sorting function uses the built-in comparator of the *Task* class to perform sorting. The comparator in *Task* is summarized in **Figure 15**.

```
…
if (isThisDateNull && !isOtherDateNull) {
	return 1;
}
if (isOtherDateNull && !isThisDateNull) {
	return -1;
}
if (isThisDateNull && isOtherDateNull) {
	return 0;
}

if (isSameDate && isThisAllDay && !isOtherAllDay) {
	return -1;
}

return thisDateTime.compareTo(otherDateTime);
…
```

Figure 15 - Task Comparator Snippet

3.3 Storage

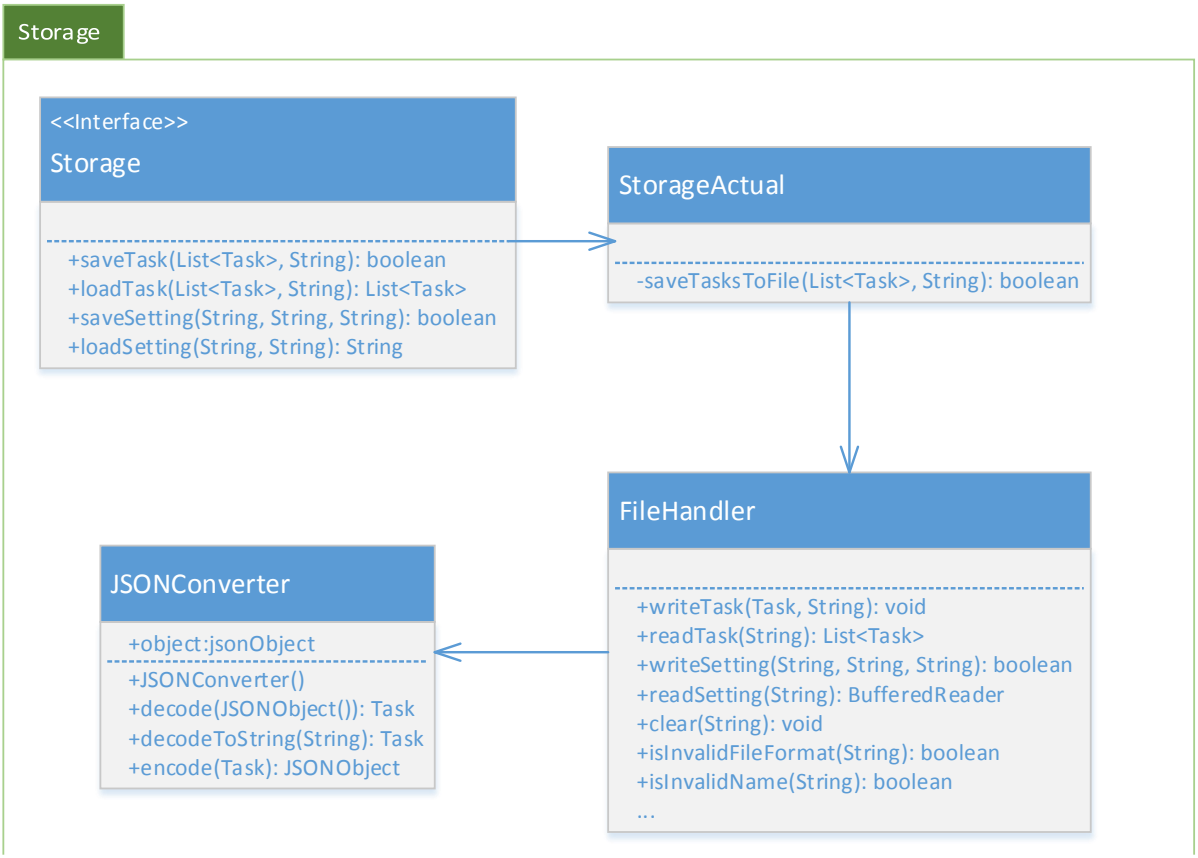


Figure 16 – Class Diagram of Storage

Storage handles the storing and retriving of *Task* data in physical storage. **Figure 16** illustrates the structure of *Storage*. *Storage* provides libraries to encode *Tasks* into *JSON* objects format, and decode *JSON* data back into *Tasks*. *Storage* also handles the automatic creation of storage files and folders if they do not exist.

The API of *Storage* is summarized in the following table.

Field / Method	Description
`saveTask(List<Task>, String): boolean`	Saves the list of tasks from the specified file path and returns true if operation is successful.
`loadTask(List<Task>, String): List<Task>`	Loads the list of tasks from the specified file path and returns a list of tasks. If the save file is not found, an empty list will be returned.
`saveSettings(String, String, String): boolean`	Saves a setting to the specified file and returns true if operation is successful.
`loadSettings(String, String): String`	Loads a setting from the specified file and returns it.

Figure 17 outlines the process of saving a list of *Tasks* passed by *Logic*, while **Figure 18** shows how *Tasks* are read.

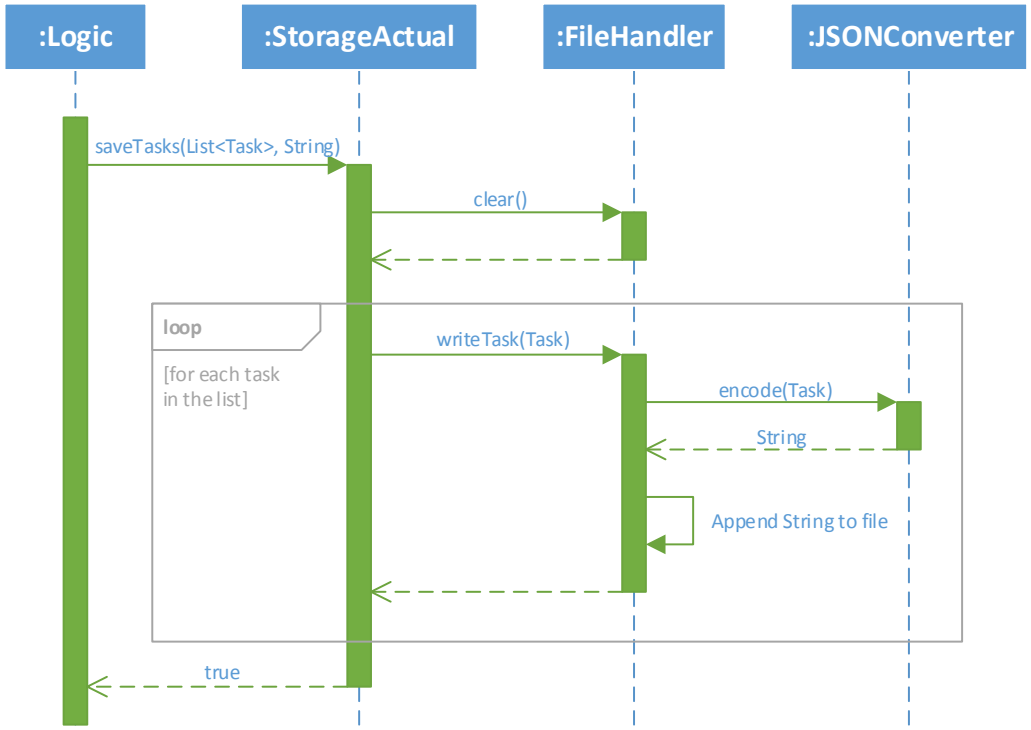


Figure 17 – Sequence Diagram for Saving Tasks

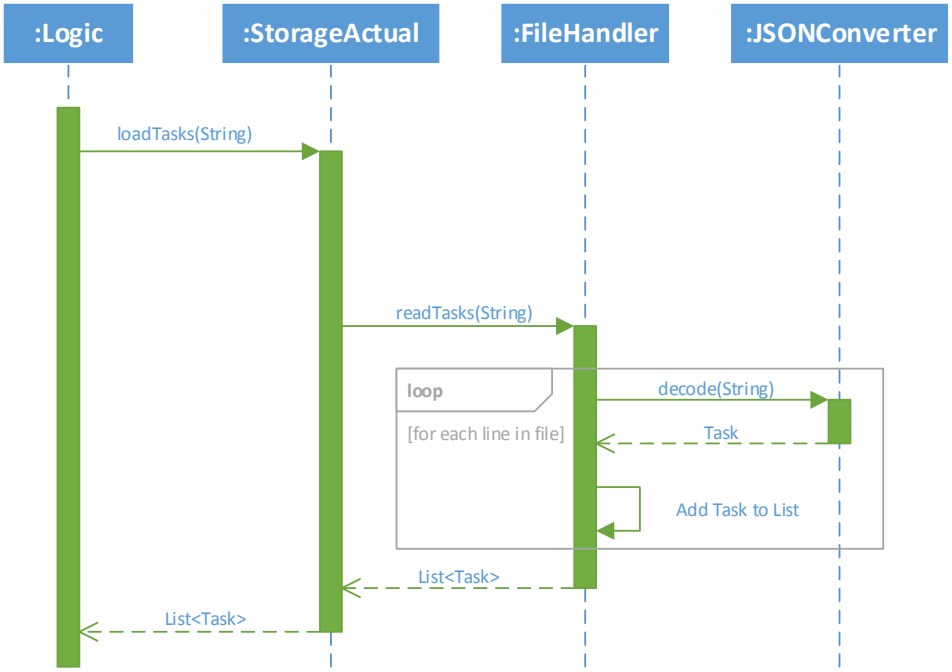


Figure 18 – Sequence Diagram for Reading Tasks

4. Setting up the Environment

4.1 Using the Versioning Tool

Task Catalyst is hosted on *GitHub*. To begin working on Task Catalyst, you will need to synchronize a copy of the source code using the *GitHub* client from the following page:

https://windows.github.com/

After downloading and installing a suitable version, you will have to log into your account and clone the remote repository to the local file system. The steps to clone a repository are outlined in **Figure 19**.

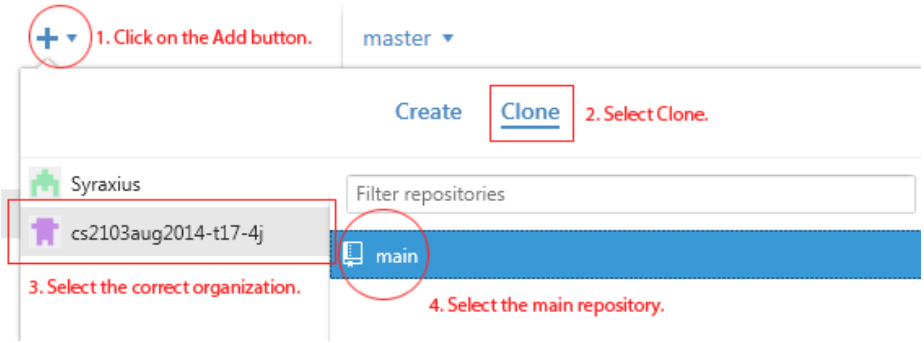


Figure 19 – Cloning the Task Catalyst Repository

You will be prompted to select a folder as the local repository. For the rest of this guide, we will assume that the default folder is selected as shown in **Figure 20**. The files will be synchronized to your local system upon confirmation.

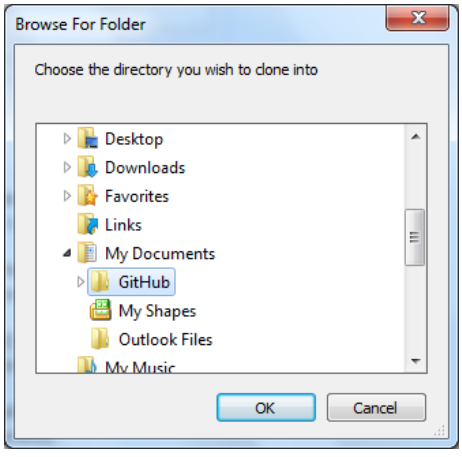


Figure 20 – Selecting Clone Folder

4.2 Setting up the IDE

The project relies on *Maven* framework for dependency management. In order to import *Maven* projects, you will need to download the *Java Developers* version of *Eclipse* from the download page:

https://www.eclipse.org/downloads/

After you have downloaded *Eclipse*, you will need to import the project as a *Maven Project*. Select **File > Import** and browse to "Existing Maven Projects" as shown in **Figure 21**.

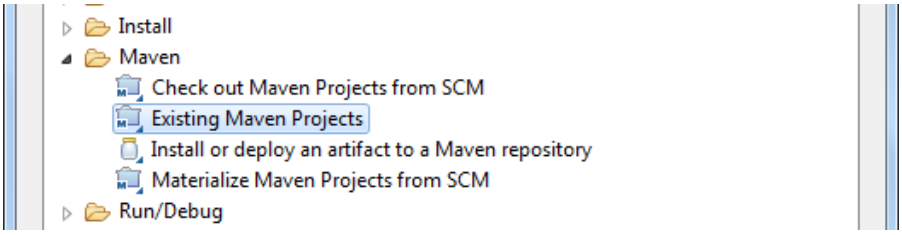


Figure 21 – Import Existing *Maven* Projects

Ensure that you select only the main pom.xml file as shown in **Figure 22**.

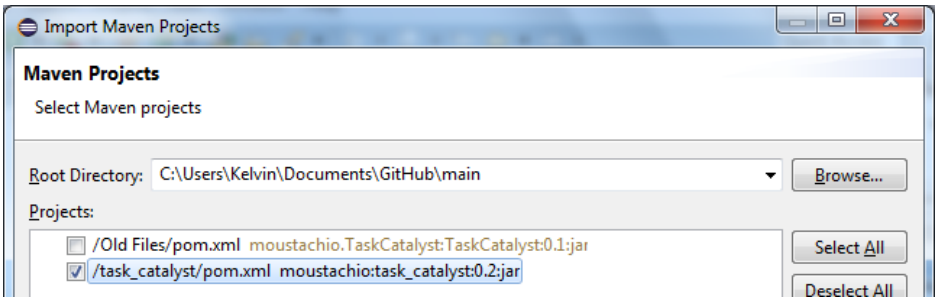


Figure 22 – Selecting the Correct POM File

You will have to use the `Alt + F5` key combination to bring up the *Update Maven Project* dialogue as shown in **Figure 23**. This will automatically download and refresh all dependencies into your system.

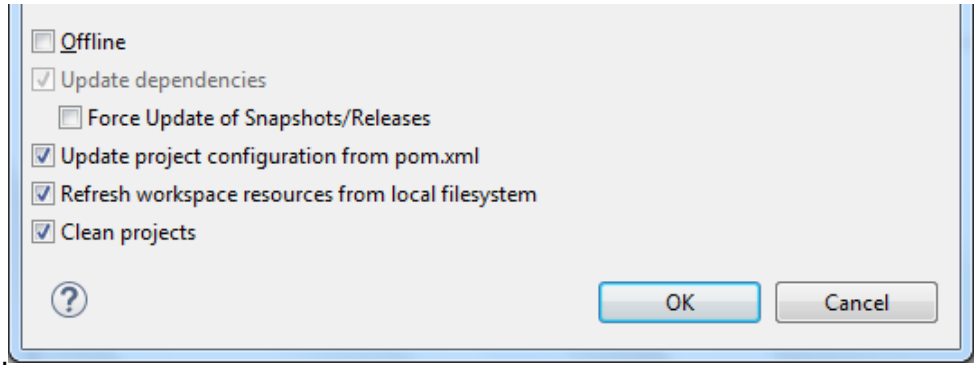


Figure 23 – Update *Maven* Project

Whenever you wish to add a new *Maven* dependency into the system, you will need to insert a new entry into pom.xml as shown in **Figure 24**.

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.ocpsoft.prettytime</groupId>
    <artifactId>prettytime-nlp</artifactId>
    <version>3.2.5.Final</version>
</dependency>
<dependency>
    <groupId>com.googlecode.json-simple</groupId>
    <artifactId>json-simple</artifactId>
    <version>1.1</version>
</dependency>
```

Figure 24 – Adding a Dependency

Whenever you add a dependency, remember to use `Alt + F5` to update the project.

To build the project into a JAR (Java Archive), ensure that you have the latest *JDK* selected in the build paths, then create a new *Maven* Build configuration as shown in **Figure 25**. The program will be built when you select **Run**.

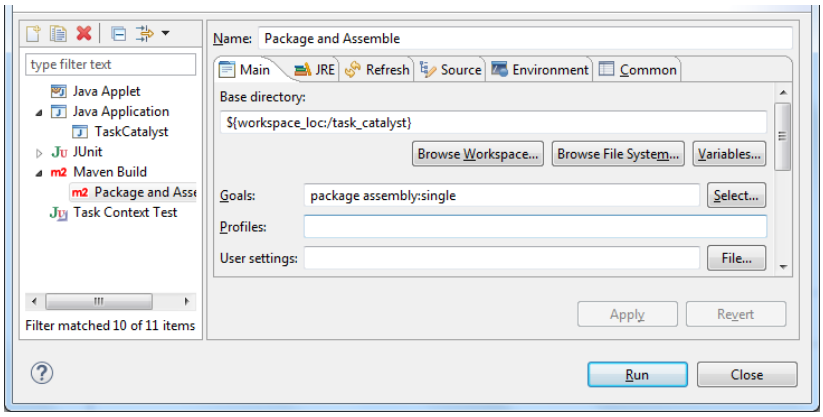


Figure 25 – Build Configuration

4.3 Testing the System

When developing new functionalities for Task Catalyst, 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 as shown in **Figure 19**.

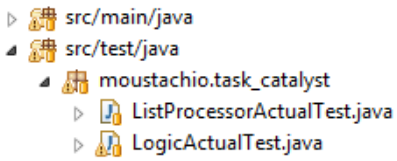


Figure 19 – `/src/test/java` Directory

To create a new *JUnit* test case, right-click on the project package, and select **New > JUnit Test Case** as shown in **Figure 20**.

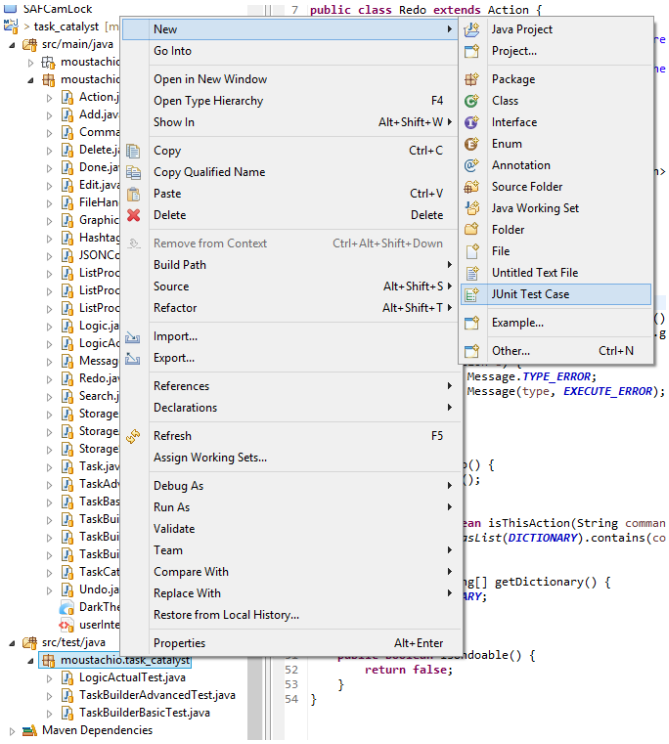


Figure 20 – Creating a new *JUnit* Test Case

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. An example is shown in **Figure 21**.

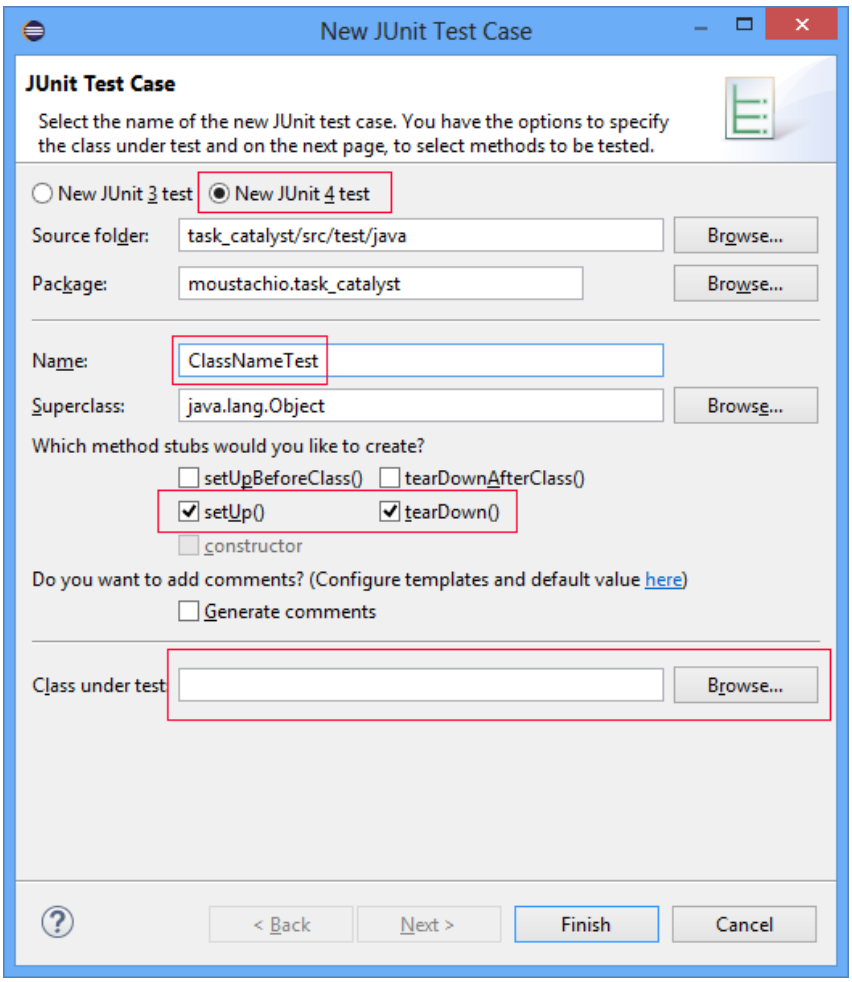


Figure 21 – Creating a new *JUnit* Test Case

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. A code snippet is shown in **Figure 22**.

```
List<Task> tasks;
TaskBuilder taskBuilder;
ListProcessorActual listProcessor;

@Before
public void setUp() throws Exception {
	tasks = new ArrayList<Task>();
	taskBuilder = new TaskBuilderAdvanced();
	listProcessor = new ListProcessorActual();
}

@After
public void tearDown() throws Exception {
	BlackBox.getInstance().close();
}

/* This is a boundary case for the case 'empty list' partition */
@Test
public void testSearchByHashtag() {
	assertEquals(listProcessor.searchByHashtag(tasks, "boss").size(), 0);
}
```

Figure 22 - Test Case Code Snippet

You can write test cases as shown in the above code. Remember to write comments for your test cases to specify its purpose and partitions as shown whenever possible.

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.

When you are done, simply right-click the test case and select **Run as > JUnit Test** to run the test as shown in **Figure 23**.

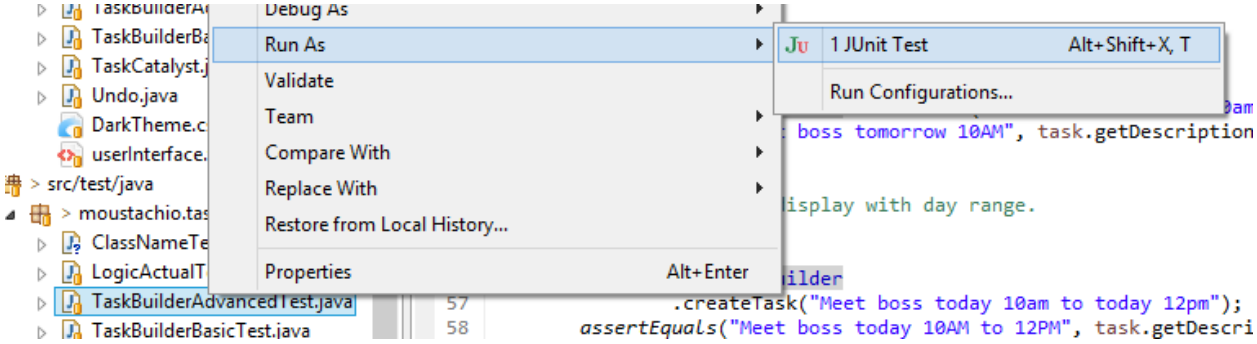


Figure 23 – Running the JUnit Test

5. Upcoming Developments

Feature	Description
Abstraction-Occurrence Pattern in *Task*	Able to handle multiple dates better by abstracting *Tasks* and its associated dates using the Abstraction-Occurrence Pattern.
Color Themes	Able to switch to various color themes in *GUI*.
Custom Parser	Able to parse and recognize date and time formats without relying on third-party libraries.
Google Calendar Integration	Able to login and synchronize *Tasks* with the user's Google account.
Instant Overlap Notification	Able to display overlaps immediately during Live Task Preview.
Reminder System	Able to parse and recognize user requests for reminders in *Logic* and display notifications to the user in *GUI*.
Settings Page	Able to modify and save various settings (Color Theme, Preferred Date Formats, etc.) using a settings page.

6. Appendix

6.1 Glossary

Term	Description
Action	A complete specification of a command, including its command dictionary, actual implementation and related hints.
Blocking Task	A *Task* with multiple dates defined separated by the “or” connector.
Display String	A String to be displayed to the user.
Floating Task	A *Task* without any date specified.
Hashtag	A # symbol appended to the start of a word. Can be used as a verb to describe the act of appending a hashtag to the start of a word.
Task	A collection of description, date and time information used to describe an entry in the task manager.
Interpreted String	A String directly formatted from the user's input, with date/time information converted into absolute values (i.e. `08 Nov 2014 08:30:00 PM`).
Message	A message paired with a type.
Multiple Task	A *Task* with multiple dates defined separated by the “and” connector.
Relative String	An Interpreted String with date/time information converted into relative terms (i.e. today, tomorrow).
Ranged Task	A *Task* with a start and end time.
Undoable	An Action that can be undone.