Task Catalyst



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

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

# 

Contents

[1. User Guide 2](#_Toc403240693)

[2. Developer’s Guide Introduction 15](#_Toc403240694)

[3. Defining the Architecture 16](#_Toc403240695)

[4. Developing the Components 17](#_Toc403240696)

[4.1 Graphical User Interface 17](#_Toc403240697)

[4.2 Logic 19](#_Toc403240698)

[4.2.1 Action and Hint System 20](#_Toc403240699)

[4.2.2 Task Manager 28](#_Toc403240700)

[4.2.3 List Processor 29](#_Toc403240701)

[4.3 Storage 30](#_Toc403240702)

[5. Testing the System 32](#_Toc403240703)

# 1. User Guide

Main Interface

Task View

Command  
Bar

Default Hashtags

Hashtags

Custom Hashtags

Hashtags

Status and Help Bar

Hashtags

Exit to System Tray

Hashtags



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:

|  |  |
| --- | --- |
| Hashtag | Meaning |
| #all | All Tasks |
| #pri | Priority Tasks |
| #ovd | Overdue Tasks |
| #tdy | Today’s Tasks |
| #tmr | Tomorrow’s Tasks |
| #upc | Upcoming Tasks |
| #smd | Someday Tasks |
| #olp | Overlapping Tasks |
| #dne | Done Tasks |

The **Custom 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 for your actions. 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 also specify date ranges using the “**to**” keyword.



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.

The recent added task will be highlighted in cyan color.

The **Status and Help Bar** displays the successful message after you add a task.



Adding Deadline Tasks



If you need to add a task with deadline, simply use the keyword “**by**”.



The task will be tagged with “**Deadline**” icon.

Adding Someday Tasks



If you are not sure which day to enter for a task, you can simple type the descriptions without date and time.

Adding All Day Tasks



If you need to add an **All Day** event, simply specify a date without time.

Blocking / Reserving Timeslots



You can block out timeslots for a task using the “**or**” keyword.



The task will be tagged with “**Reserved**” icon.

Note: The time displayed on the task will change, due to today’s date and time. For example, if today is 10 Nov 8:00 AM, the time will change from 7:00 AM to 8:00 AM. In addition, the **Alternate timing** will only display 10 Nov 9:00 AM.

Tasks with Multiple Recurrences



You can also add tasks that occurs across multiple timeslots using the “**and**” keyword.



The task is automatically split into the various timeslots.

Viewing Custom Hashtags

You can quickly navigate through categories or custom hashtags simply by typing the hashtag in the command bar.

Furthermore, you can also use **ALT + UP/DOWN** hotkey to navigate through hashtag lists.



This will display the corresponding items with the hashtag.

******

Viewing Overdue Tasks

You can view overdue tasks by typing “**#ovd**”.

Overdue tasks is tagged with an “**Overdue**” icon



Viewing Upcoming Tasks

You can view upcoming tasks by typing “**#upc**”.

Note: Upcoming task does not display tasks that fall on today, tomorrow, overdue, someday and completed category.



Viewing Overlapping Tasks

You can view overlapped tasks by typing “**#olp**”.

Overlapping tasks is tagged with an “**Overlapping**” icon



Viewing Completed Tasks

You can view completed tasks by typing “**#dne**”.

Completed tasks is tagged with an “**Done**” icon



Searching for Tasks******

You can also search for tasks typing “**search**”, or alternative commands, followed by a search keyword.



Matching items are quickly displayed in the task view.

Editing a Task



You edit a task simply by typing “**edit**” followed by the task number.

The **Auto-Complete** feature will fill in your task details in the command bar.



Make your necessary changes and hit enter to save the changes.

Completing a Task

You can complete a task simply by typing “**done**” or “**complete**”, followed by the task number or keyword.

You can also complete all tasks by typing “**done all**”.





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

Deleting a Task

You can delete an unwanted task simply by typing “**delete**”, “**rm**” or “**del**”, followed by the specific task number or numbers.

You can also delete all tasks by typing “**delete all**”.





The item is deleted and removed from display.

Undoing an Action

You may undo an action by typing “**undo**” or using the **Ctrl+Z** hotkey.

The previous action is undone. You may also redo by typing “**redo**” or using the **Ctrl+Y** hotkey.



Setting Priority Task



You can set priority by **hashtagging** an entry with **#pri**.



Priority tasks are automatically labelled.

You can type “**#pri**” to view all priority tasks.

Hotkeys

There are many things you can do with hotkeys in Task Catalyst. You can use **Ctrl+H** to launch a comprehensive Quick Start menu.



Smart Paste

You can quickly paste highlighted text from other applications using a global hotkey **Ctrl+D**.

# 2. Developer’s Guide Introduction

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 uses familiar features like **hashtagging** for organization.

The design of Task Catalyst is built upon the Natural Bucket, which focuses to make Task Catalyst:

* User Friendly and Intuitive
* Simple yet Powerful
* Accessible

In this Developer’s Guide, you will first be introduced to the High-Level Architecture (Section 3) of the program. Next, we will talk about the components of the system from front-end to back-end (Section 4). Each component will be introduced using its class diagram and APIs, and then further elaborated with behavioral diagrams and code samples if available. Finally, we will guide you on the testing standards adopted in this project (Section 5).

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

# 3. Defining the Architecture



Figure 1 – Architecture Diagram

The overall architecture 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, status messages, and providing autocomplete functionality. 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, and provision of logical data structures. It depends on *Storage* for physical storage.

*Storage* 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.

# 4. Developing the Components

## 4.1 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, controlling the display elements as well as communication with *Logic*.

**Figure 3** depicts the interactions between the *User*, *GUI* and *Logic* during initialization:



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 calls getMessageTyping(userInput) to generate a new hint. The entire command string is sent to *Logic* using the processCommand(userInput) method 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*. |

## 

## 4.2 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 component is depicted in **Figure 5**.

*Logic*’s role is to provide all necessary backend functionality for the *GUI*, including command parsing, hints generation, 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 in **Figure 6**:

|  |  |
| --- | --- |
| Field / Method | Description |
| processCommand(String): Message | Parses, interprets, and executes a user command. |
| getMessageTyping(String): Message | Generate a dynamic hint based on the current user command. |
| getDefaultHashtags(): List<String> | Returns the list of default hashtags. |
| getHashtags(): List<String> | Returns the list of user 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 tasks that should be selected (or highlighted). |

Figure 6 - API for Logic Interface

### 4.2.1 Action and Hint System

Figure 7 - Action and Hint System

*ActionHintSystem* applies the Command pattern. As shown in **Figure 7**, 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. |

#### Action Class – 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. Even though it is not specified in the *Action* interface, it is compulsory to implement various static methods for each *Action* subclass. These methods are shown in **Figure 8**.

|  |  |
| --- | --- |
| Field / Method | Description |
| DICTIONARY: String[] | All commands associated with this action. |
| isThisAction(String): boolean | 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(): Message | Code for executing the action. |
| undo(): Message | Code for undoing the action. |
| HINT\_MESSAGE and variants | The hint message to return when getHint() is called. |
| getHint(String): Message | Returns a *Message* hint based on the input string. |
| isUndoable(): boolean | Static method for checking if the action is undoable. |

Figure 8 - API for Action Subclasses

|  |
| --- |
| 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 9 – 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.

When an isThisAction(String) command evaluates to true, an object of that *Action* is created and the entire user input is passed to its constructor for further parsing within the *Action* object.

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 10 – Sequence Diagram for Undo Action

|  |
| --- |
| Note: How the delete function is undone is not shown, but the steps are similar to how it is executed. Please refer to the actual code for more information. |

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 *ActionInvoker* and calls the undoLastAction() method. This causes the undo() method of the *Action* object to be called, which generates a *Message* that is eventually returned to the *Logic*.

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 type set to MessageType.ERROR. |

#### Message Class – Generating Hint and Autocomplete

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

A *Message* object encapsulate the information shown in **Figure 11**.

|  |  |
| --- | --- |
| 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 11 – Message Class Summary

*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*.

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.

*ActionHintSystem* generates hints for partial command matches, as well as hints specific to a command if there is a match. **Figure 12** illustrates the hint generation process:



Figure 12 – Hint Generation Activity Diagram

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.



Figure 13 – 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.

#### Add Action – Building, Parsing and Adding Tasks



Figure 14 – Class Diagram for Add Action

The *Task Builder* 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*. 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 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 Input can be converted into a Relative String for further manipulation or a Display String for displaying.

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

|  |  |  |
| --- | --- | --- |
| 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. 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

After the conversion process, the *Interpreted Input* is returned as the *Interpreted String* 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 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 for displaying is shown below:

|  |  |
| --- | --- |
| 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

|  |
| --- |
| Note: There is a variant of Display String available that does not have date/time information embedded in the description. |

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”). 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 15 – Friendly Date Conversion Process

### 4.2.2 Task Manager



Figure 16 – Task Manager Class Diagram

The *Task Manager* Interface follows the Demeter’s Principle 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. In addition, it also keeps track of the list of tasks that was most recently modified for the GUI to highlight.

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

### 4.2.3 List Processor



Figure 17 – Class Diagram of List Processor

*ListProcessorActual* provides the API for processing the list of Tasks passed by *TaskManagerActual*.

When the user uses the search command, the searchByKeyword(List<Task> list, String keyword) method is called and *ListProcessorActual* will return a list of Tasks containing the specified keyword.

*TaskManagerActual* calls searchByHashtag(List<Task> list, String hashtag) method if the user keys in a hashtag category. *ListProcessorActual* will either return 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.

The table below 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

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

## 4.3 Storage

*Storage* handles the storage and retriving of task data in physical storage. *Storage* provides libraries to encode *Tasks* into JSON objects format, and decoding of JSON data back into *Tasks*. *Storage* also handles the automatic creation of storage files and folders if they do not exist.

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



Figure 18 – Class Diagram of Storage Component

**Figure 18** outlines the process of saving a list of Tasks passed by *Logic*, while **Figure 19** shows how tasks are read.



Figure 19 – Sequence Diagram for Saving Tasks



Figure 20 – Sequence Diagram for Reading Tasks

# 5. Testing the System

When developing new functionalities, 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 21 – /src/test/java Directory

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



Figure 22 – 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 23 – 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:

|  |
| --- |
| 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 24 – Running the JUnit Test

# 6. Appendix

## 6.1 Upcoming Developments

|  |  |
| --- | --- |
| Feature | Description |
| Custom Parser | Parse and recognize date and time formats without relying on external libraries. |
| Reminder System | Parse and recognize user requests for reminders in Logic.  Display notifications to the user in GUI. |

## 6.2 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. |
| 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. |