|  |  |
| --- | --- |
| Developer guide  TEAM PGP | Abstract  **This developer guide is a comprehensive document for contributing to our to-do list application "Do-er List" – for all avid software engineers, bet it a fresh and new to the programming scene or a seasoned and experienced coder. We welcome all of you on board, to be a contributor of this awesome, life-changing application.**  CS2101/CS2103T  Software Engineering & Effective Communication for Computing Professionals |

|  |  |
| --- | --- |
| Developer guide  TEAM PGP | Abstract  **This developer guide is a comprehensive document for contributing to our to-do list application "Do-er List" – for all avid software engineers, bet it a fresh and new to the programming scene or a seasoned and experienced coder. We welcome all of you on board, to be a contributor of this awesome, life-changing application.**  CS2101/CS2103T  Software Engineering & Effective Communication for Computing Professionals |

Contents

[I. Purpose Statement and Overview 3](#_Toc464079367)

[II. Quick Start 4](#_Toc464079368)

[1. Prerequisite Software 4](#_Toc464079369)

[2. Importing this project 4](#_Toc464079370)

[III. Design 5](#_Toc464079371)

[1. Overall Architecture 5](#_Toc464079372)

[2. UI component 8](#_Toc464079373)

[3. Logic component 9](#_Toc464079374)

[4. Model component 10](#_Toc464079375)

[5. Storage component 11](#_Toc464079376)

[6. Common classes 12](#_Toc464079377)

[IV. Implementation 13](#_Toc464079378)

[1. Logging 13](#_Toc464079379)

[2. Configuration 13](#_Toc464079380)

[V. Testing 14](#_Toc464079381)

[1. Running Test in Eclipse 14](#_Toc464079382)

[2. Using Gradle 14](#_Toc464079383)

[VI. Dev Ops 16](#_Toc464079384)

[1. Build Automation 16](#_Toc464079385)

[2. Continuous Integration 16](#_Toc464079386)

[3. Making a Release 16](#_Toc464079387)

[4. Managing Dependencies 16](#_Toc464079388)

[VII. Appendix A : User Stories 17](#_Toc464079389)

[VIII. Appendix B : Use Cases 20](#_Toc464079390)

[IX. Appendix C : Non Functional Requirements 27](#_Toc464079391)

[X. Appendix D : Glossary 28](#_Toc464079392)

[XI. Appendix E : Product Survey 29](#_Toc464079393)

[Review of TickTick: 29](#_Toc464079394)

[Review of WunderList: 29](#_Toc464079395)

[Review of Trello 30](#_Toc464079396)

[Review of Google Calendar 30](#_Toc464079397)

1. Purpose Statement and Overview

This developer guide is a comprehensive document for contributing to our to-do list application "Do-er List" - for both new and experienced software engineer, and even the young programmer who just started coding. We welcome all of you on board, to be a contributor of this awesome, life-changing application.

Do-er List is a task manager which is designed ever so simple that it only takes in one command line as its primary input, at the same time extremely powerful and feature-rich.

This guide is maintained by the 4 original developers, Xiaopu, Jason, Benedict, Hai Long, and we hope you will join us soon.

# Quick Start

Here are the basic steps needed to get started and begin your first contribution.

### 1. Prerequisite Software

Make sure you have the following software installed:

1. JDK 1.8.0\_60 or later
2. **Eclipse IDE** with **e(fx)clipse** plugin for Eclipse and **Buildship Gradle Integration** (from the Eclipse Marketplace)

### 2. Importing this project

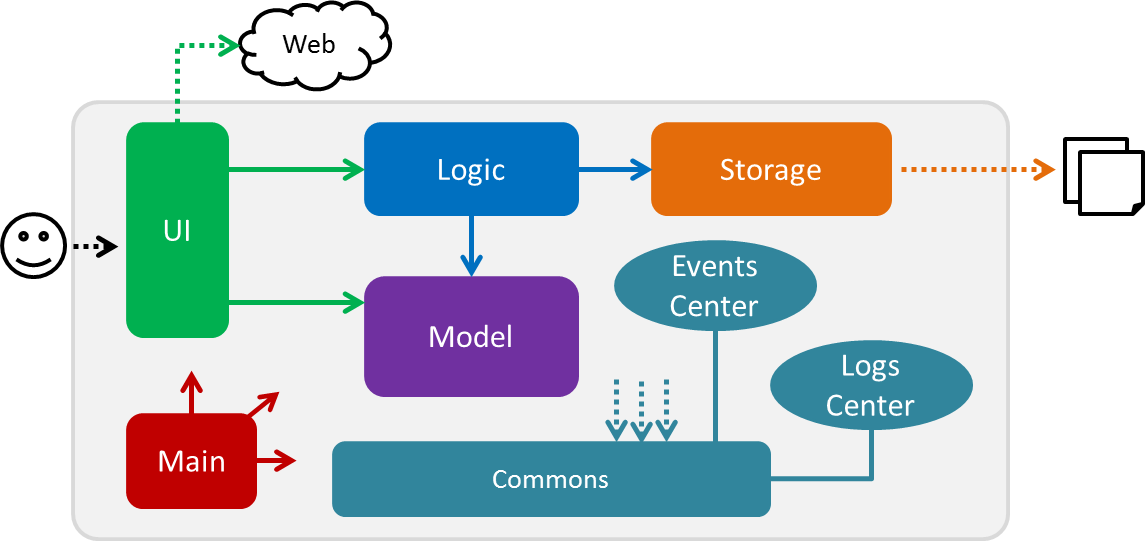
Do the following steps to successfully import this project into your workspace and work on our program:

1. Fork the repository at https://github.com/CS2103AUG2016-W09-C4/main, and clone the fork to your computer
2. Open Eclipse (**Note:** Ensure you have installed the e(fx)clipse and buildship plugins as given in the prerequisites above)
3. Click File > Import
4. Click Gradle > Gradle Project > Next > Next
5. Click Browse, then locate the project's directory
6. Click Finish
   * If you are asked whether to 'keep' or 'overwrite' configuration files, choose to **'keep'**.
   * Depending on your connection speed and server load, it can even take up to 30 minutes for the setup to finish. (This is because Gradle downloads library files from servers during the project set up process.)
   * If Eclipse automatically changes any settings files during the import process, you can discard those changes.
   * Remember to run all test cases and make sure they pass.

And that is it! You are now officially a part of our team!

# Design

### 1. Overall Architecture

****

*Figure 1: Architecture Diagram of high-level design*

The *Architecture Diagram* given above explains the high-level design of the application. Below is a quick overview of each component.

**Main** has only one class called **MainApp**. It is responsible for:

1. At application launch: Initializing the components in the correct sequence, and connect them up with each other.
2. At shut down: Shutting down the components and invoking clean-up method where necessary.

**Commons** represents a collection of classes used by multiple other components. Two of those classes play important roles at the architecture level. They are:

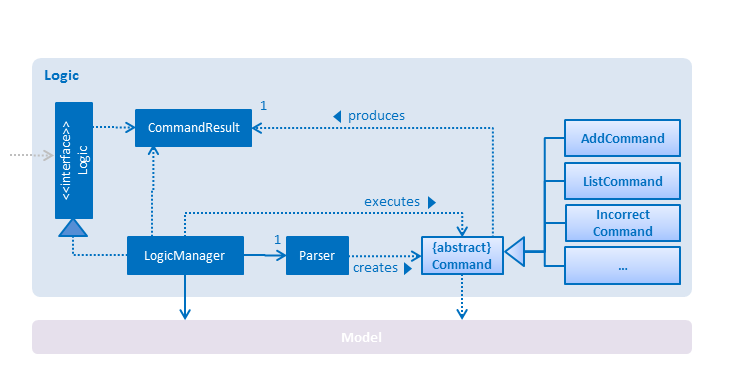
1. **EventsCenter**: This class (written using Google's Event Bus library) is used by components to communicate with other components using events (i.e. a form of *Event Driven* design)
2. **LogsCenter**: Used by many classes to write log messages to the App's log file.

The rest of the App consists of the following 4 elements:

1. **UI**: The UI of the App.
2. **Logic**: The command executor.
3. **Model**: Holds the data of the App in-memory.
4. **Storage**: Reads data from, and writes data to, the hard disk.

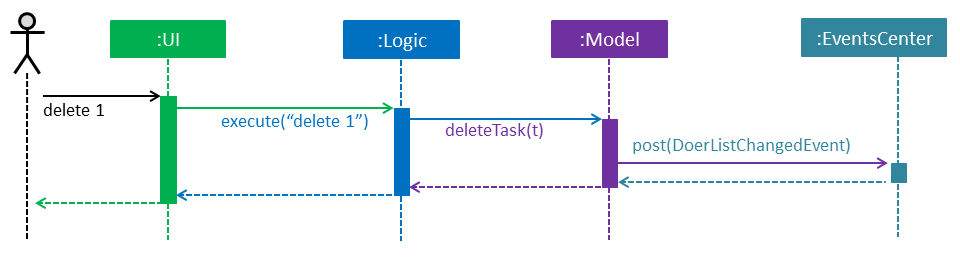
Then each of the above-mentioned four elements does the following:

1. Defining its *API* in an interface with the same name as the Component.
2. Exposing its functionality using a **{Component Name}Manager** class.

For example, the **Logic** component (see figure 2 given below) defines its API in the **Logic.java** interface and exposes its functionality using the **LogicManager.java** class. ****

*Figure 2: Architecture Diagram of the Logic Component*

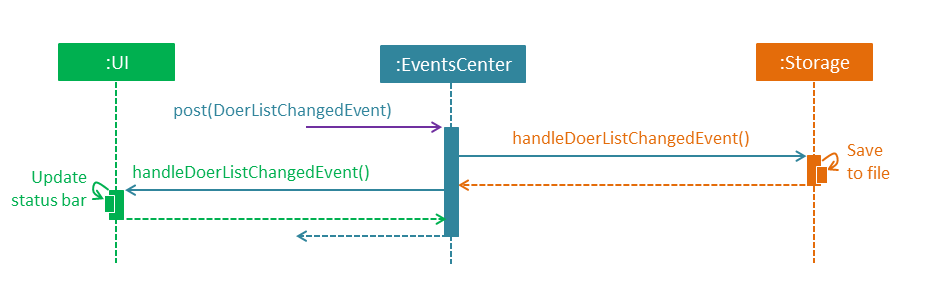
What happens when the user issues a command? How each component in the architecture diagram works together to deliver the result? The *Sequence Diagram* (figure 3) below shows how the components interact for the scenario where the user issues the command **delete 1**.

****

*Figure 3: Figure Diagram when user issues the command* ***delete 1***

Note how the **Model** simply raises a **DoerListChangedEvent** when the Do-*er* List data are changed, instead of asking the **Storage** to save the updates to the hard disk.

The diagram below (figure 4) shows how the **EventsCenter** reacts to that event, which eventually results in the updates being saved to the hard disk and the status bar of the **UI** being updated to reflect the 'Last Updated' time.

****

*Figure 4: How the* ***EventsCenter*** *react to a change in event*

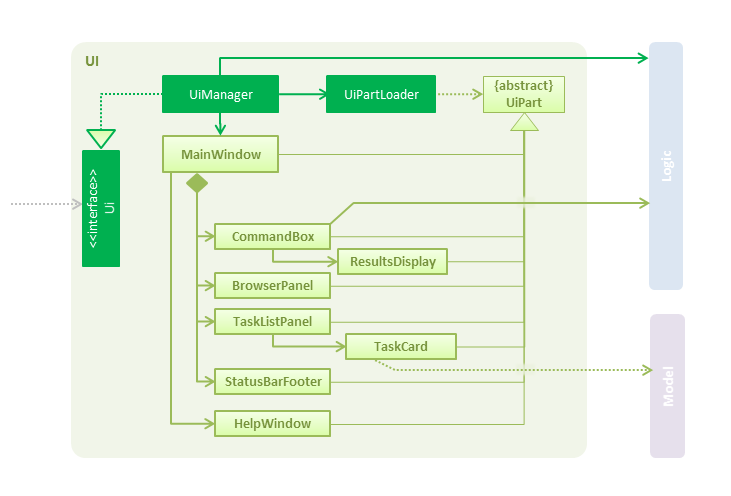
Note how the event is propagated through the **EventsCenter** to the **Storage** and **UI** without **Model** having to be coupled to either of them. This is an example of how this Event Driven approach helps us reduce direct coupling between components.

In the following sections, we will explore the four components, namely **UI**, **Logic**, **Model**, **Storage**. We will be exploring their class diagrams and their APIs to understand how they work and how they work together to deliver a results. It is **important** that you study the following section carefully as our software is written in the "Object-Oriented Paradigm", so it is crucial to know how various parts work together.

### 

### 2. UI Component

The **UI** component (Ui.java) is responsible for displaying user interface. The following figure 5 is its architecture diagram, displaying its smaller components and how they relate to each other.

****

*Figure 5: Architecture Diagram of the UI Component*

As we can see from the diagram above, the **UI** consists of a **MainWindow** that is made up of various parts that are responsible for displaying the **UI**, for example, **CommandBox**, **ResultDisplay**, **TaskListPanel**, **StatusBarFooter**, **BrowserPanel**, etc. All these, including the **MainWindow**, inherit from the abstract **UiPart** class and they can be loaded using the **UiPartLoader**.

The UI component uses **JavaFx Ui** framework. The layout of these **UI** parts are defined in matching **.fxml** files that are in the **src/main/resources/view** folder.

For example, the layout of the **MainWindow** is specified in **MainWindow.fxml**

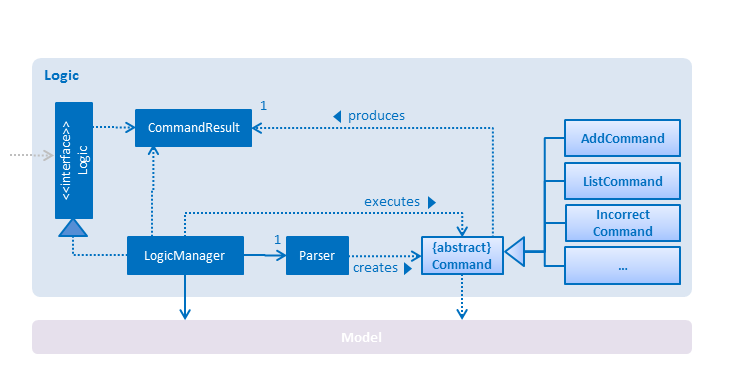
What are the crucial roles of the **UI** component? Its main roles are:

1. Executing user commands using the **Logic** component.
2. Binds itself to some data in the **Model** so that the **UI** can auto-update when data in the **Model** changes. E.g. **TaskCardListPanel** adds a listener to the **ObservableList** in **Model** so that it will be notified when the list is updated.
3. Responds to events raised from various parts of the application and updates the **UI** accordingly. The events will be handled by **UiManager** first and then it will notify other components to update.

### 3. Logic Component

The **Logic** component is responsible for logic in the application, such as all the possible commands

(E.g. add, delete, list, etc) and parsing user's inputs. Figure 6 below shows the architecture diagram of the **Logic** component.

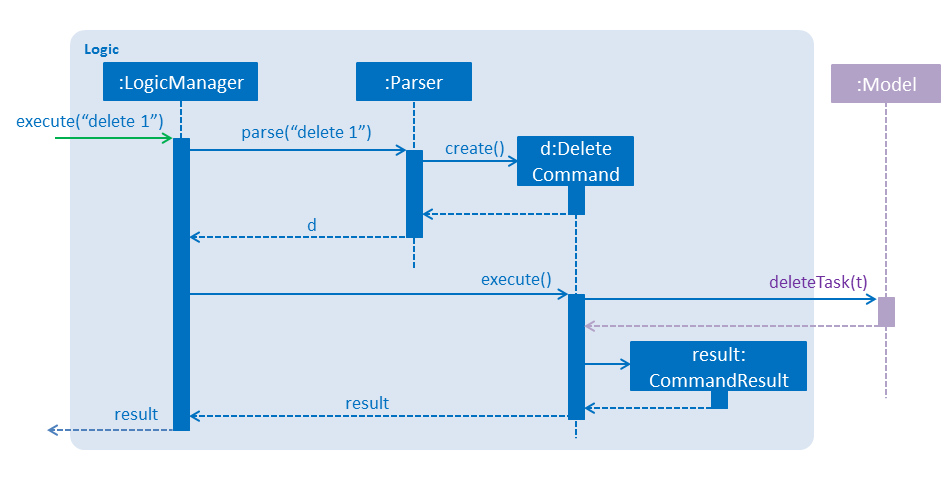
****

*Figure 6: Architecture Diagram of the Logic component*

What happens when the user type in a command?

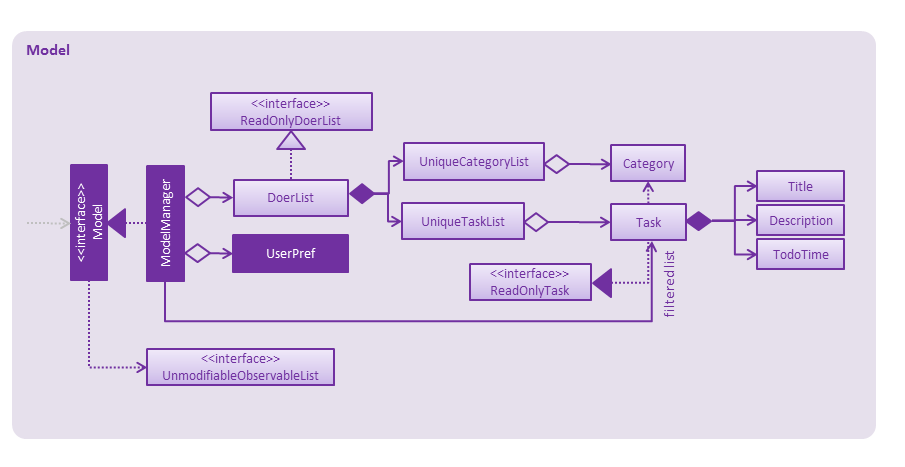
First, **Logic** uses the **Parser** class to parse the user command. This results in a **Command** object which is executed by the **LogicManager**. The command execution can affect the Model (e.g. adding a task) and/or raise events. The result of the command execution is encapsulated as a **CommandResult** object which is passed back to the **UI**.

To provide you with a concrete example, below is the Sequence Diagram for interactions within the Logic component for the **execute("delete 1")** API call. Note that the **UI** will also be updated as the delete operation will cause changes in the **ObservableList** in **Model**.

*****Figure 7: Sequence Diagram when "delete 1" is called.*

### 4. Model component

The **Model** component stores the Do-er List's data, such as how to represent a **Task** with its **Title** or **Description**. Figure 8 gives an overall architecture diagram of the **Model** component.

****

*Figure 8: Architecture Diagram of the Model Component*

The **Model** stores a **UserPref** object that represents the user's preferences besides storing the Do-er List data.

Importantly, it exposes a **UnmodifiableObservableList<ReadOnlyTask>** that can only be 'observed'. It means that the UI can be bounded to this list so that the UI automatically updates when the data in the list change, thus reflecting real-time changes.

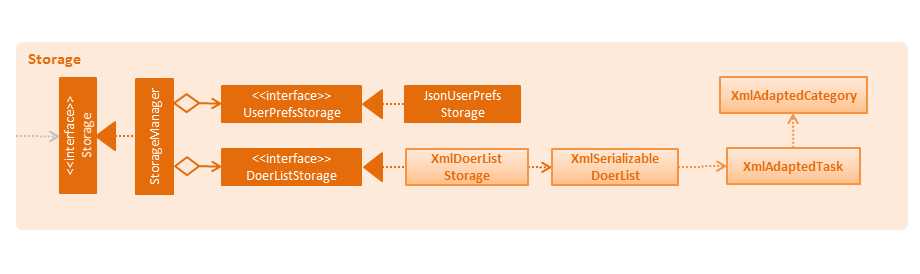
Note that besides category created by user, Do-er List also store a **BuildInCategoryList** in the model. They are **All**, **Today**, **Next**, **Inbox** and **Complete**. They are defined at the convenience of user to view different tasks under different days.

Model does not depend on any of the other three components.

### 

### 5. Storage component

The **Storage** component, like its name suggests, stores various data. It can save the Do-er List data in xml format and read it back. Additionally, it saves the **UserPref** objects in json format and reads it back.

Refers to figures 9 for the complete architecture diagram of this component. ****

*Figure 9: Architecture Diagram of Storage component*

### 

### 6. Common classes

Classes used by multiple components are in the **seedu.doerList.commons** package. Some of them are below:

* **EventsCenter**: The class is responsible for post events and handling events. When an event has been raised, the **EventsCenter** will check whether there is handler for the event and notify the correct handlers.
* **LogsCenter**: The class is responsible for recording the operation histories in execution. The records are useful for programmer in debugging.

# Implementation

### 1. Logging

We are using **java.util.logging** package for logging. The **LogsCenter** class is used to manage the logging levels and logging destinations.

1. The logging level can be controlled using the **logLevel** setting in the configuration file (See Configuration)
2. The **Logger** for a class can be obtained using **LogsCenter.getLogger(Class)** which will log messages according to the specified logging level
3. Currently log messages are output through: **Console** and to a **.log** file.

There are 4 logging level in total. They are:

1. SEVERE: Critical problem detected which may possibly cause the termination of the application.
2. WARNING: Can continue, but proceed with caution.
3. INFO: Information showing the noteworthy actions by the application.
4. FINE: Details that are not usually noteworthy but may be useful in debugging.

E.g. print the actual list instead of just its size.

### 

### 2. Configuration

Certain properties of the application can be controlled (E.g application name, logging level) through the configuration file (default:config.json).

# Testing

Tests can be found in the **./src/test/java** folder.

### 1. Running Test in Eclipse

If you are not using a recent Eclipse version (i.e. *Neon* or later), enable assertions in JUnit tests as described in this link:

[*http://stackoverflow.com/questions/2522897/eclipse-junit-ea-vm-option*](http://stackoverflow.com/questions/2522897/eclipse-junit-ea-vm-option)

If you want to run all tests, right-click on the **src/test/java** folder and choose **Run as > JUnit Test**. If you just want to run a subset of tests, you can right-click on a test package, test class, or a test and choose to run as a JUnit test.

### 2. Using Gradle

Please see **UsingGradle.md** that comes together when you download the project to properly run the tests using Gradle.

We have two types of tests:

1. GUI Tests - These are *System Tests* that test the entire application by simulating user actions on the GUI. These are in the **guitests** package.
2. Non-GUI Tests - These are tests not involving the GUI. They include,
   1. *Unit tests* target the lowest level methods/classes.

e.g. **seedu.doerList.commons.util.UrlUtilTest**

* 1. *Integration tests* check the integration of multiple code units (those code units are assumed to be working).

e.g. **seedu.doerList.storage.StorageManagerTest**

* 1. Hybrids of unit and integration tests. These tests are checking multiple code units as well as how the are connected together.

e.g. **seedu.doerList.logic.LogicManagerTest**

Headless GUI Testing : Thanks to the [TestFX](https://github.com/TestFX/TestFX) library that we are using, our GUI tests can be run in the *headless* mode. In the headless mode, GUI tests do not show up on the screen. That means the developer can do other things on his computer while the tests are running.

See UsingGradle.m[d](https://github.com/TeamPGP/main/blob/DeveloperGuide/docs/UsingGradle.md#running-tests) to learn how to run tests in headless mode.

# Dev Ops

### 1. Build Automation

See UsingGradle.md to learn how to use Gradle for build automation.

### 2. Continuous Integration

We use Travis CI to perform *Continuous Integration* on our projects. See UsingTravis.md that comes together when you download the project for more details.

### 3. Making a Release

Here are the steps to create a new release:

1. Generate a JAR file using Gradle.
2. Tag the repo with the version number. e.g. v0.1
3. Create a new release using GitHub and upload the JAR file you created.

### 4. Managing Dependencies

A project often depends on third-party libraries. For example, Do-er List depends on the Jackson library for XML parsing. Managing these *dependencies* can be automated using Gradle. For example, Gradle can download the dependencies automatically, which is better than manually downloading from each library.

# Appendix A : User Stories

Priorities: High (must have) - \* \* \*, Medium (nice to have) - \* \*, Low (unlikely to have) - \*

| **Priority** | **As a** | **I want to ...** | **So that I can...** |
| --- | --- | --- | --- |
| \* \* \* | new user | see usage instructions of all commands | how to use various commands in the application |
| \* \* \* | user | create task with title and description | put summary as title and more details in description |
| \* \* \* | user | create task without start time or end time | record tasks that need to be done but do not have deadlines |
| \* \* \* | user | create task with start time and end time or deadlines | know what task I should be doing or due on a certain timing |
| \* \* \* | user | edit task's title, description, start time, end time and categories | update it when I want to in the event I need to make some changes instead of creating a new task |
| \* \* \* | user | view all tasks | have an overview of all the tasks |
| \* \* \* | user | view a specific task | get more details of the specific task |
| \* \* \* | user | find a task by title and description | quickly locate the task if I can only remember few words in the title or description |
| \* \* \* | user | delete tasks | track only the required tasks |
| \* \* | user | add tasks to different categories | organise my tasks properly |
| \* \* | user | view the tasks under a certain category | examine different tasks under different categories |
| \* \* | user | view the tasks are going to happen or due today, tomorrow, next 7 days, etc. | know what is going to happen in the coming day(s) |
| \* \* | user | undo the most recent operations | revert back the changes when I make a wrong command by mistake |
| \* \* | user | redo the most recent operations | redo the operation after undoing it |
| \* \* | user | specify a storage location for data storage | specify my data location to the cloud syncing service provider to prevent data loss |
| \* \* | user | mark or unmark the task as [done](https://github.com/CS2103AUG2016-W09-C4/main/blob/master/docs/DeveloperGuide.md#done) | only keep track of the task(s) which are needed to be done and archive the finished task(s) |
| \* | user | type command parameters in arbitrary order | specify my parameter(s) without requiring me to remember the order |
| \* | user | add external ical file to the to do-lists | keep track of other events created by other applications |
| \* | user | create recurring tasks | be reminded to do the same task at the same time interval |
| \* | user | view events in Google Calendar | have a better pictorial view of my schedule. |

# Appendix B : Use Cases

(For all use cases below, the **System** is the Do-er List and the **Actor** is the user, unless specified otherwise)

#### Use case: Add task

**MSS**

1. User requests to add in a task.
2. To-Do List creates task with title, description, start date and end date.
3. The task is moved into the categories according to the parameters.
4. System displays the details of the created task.  
   Use case ends.

**Extensions**

1a. add command followed by the wrong parameters

1a1. System indicates the error and display the correct format for user.

Use case ends.

1b. TITLE is empty string

1b1. System indicates the error that task\_name is empty.

Use case ends.

1c. User does not supply START or END parameters

1c1. Event is created and categorized to INBOX.

1c2. System displays the created task.

Use case resumes from steps 2.

1d. User does not supply START parameter

1d1. Event is created with START as today.

Use case resumes from steps 2.

1e. System is able to parse START or END which is not in standard format.

Use case resumes from steps 2.

1f. System is not able to parse START or END which is not in standard format.

1f1. System will create the task without START and END date.

1f2. System indicates the error to user.

Use case resumes from steps 2.

#### Use case: Edit task

**MSS**

1. User types in the command.
2. To-Do List finds the task at that index.
3. The task details are changed accordingly (E.g. title, description, start time, end time, category).
4. System displays the details of the newly edited task.   
   Use case ends.

**Extensions**

1a. edit command followed by the wrong parameters.

1a1. System indicates the error and display the correct format for user.

Use case ends.

1b. edit command is followed by the non-existent INDEX.

1.b.1 System indicates the error that the INDEX is non-existent.

Use case ends

1c. TITLE is empty string.

1c1. System indicates the error that TASK\_NAME is empty.

Use case ends.

1d. System is not able to parse START or END which is not in standard format.

1d1. System will create the task without START and END date.

1d2. System indicates the error to user.

Use case resumes from steps 2.

#### Use case: Delete task

**MSS**

1. User types in the command.
2. System finds the task at that index.
3. System confirms with the user if he wants to delete the task.
4. User confirms.
5. System deletes the task.   
   Use case ends.

**Extensions**

1a. delete command is followed by the wrong parameters.

1a1. System indicates error and display the correct format to user.

Use case ends.

1b. delete command is followed by a non-existent INDEX.

1b1. System indicates the error in the INDEX is non-existent.

Use case ends.

4a. User rejects the confirmation.

4a1. System indicates that the delete order was not carried out.

Use case resumes from step 1.

#### Use case: List task by category

**MSS**

1. User types the list command with specific category name as parameter.
2. System displays all the task under CATEGORY.   
   Use case ends.

**Extensions**

1a. User does not supply CATEGORY.

1a1. System displays all the tasks.

Use case ends.

2a. The category does not exist in the system.

2a1. System indicates the error.

Use case ends.

#### Use case: Undo Command

**MSS**

1. User types in the undo command.
2. System tries to find the last operation that caused a change of data.
3. System undoes the operation.
4. System indicates the change to user.   
   Use case ends.

**Extensions**

2a. The last operation which involve the change of the data does not exist.

2a1. System indicates the error.

Use case ends.

#### Use case: Clear Command

**MSS**

1. User types in the command.
2. System confirms if user wants to clear the entire all of the tasks.
3. User confirms.
4. System deletes all the tasks.   
   Use case ends.

**Extensions**

3a. User rejects the confirmation.

3a1. System indicates that the clear order was not carried out.

Use case resumes at step 1.

#### Use case: Help Command

**MSS**

1. User types in the command.
2. System finds with the details of a command in its parameters.
3. System displays the details.   
   Use case ends.

**Extensions**

1a. help command is followed by the wrong parameters.

1a1. System indicates the error and displays the correct format for user.

Use case ends.

1b. help command is followed by no parameters.

1b1. System displays all the commands available with all of the details.

Use case ends.

#### Use case: View a task

**MSS**

1. User types in the view command.
2. System retrieves the task list based on the index parameter in the recently displayed list.
3. System displays the details of the task.   
   Use case ends.

**Extensions**

2a. There is no recently displayed list.

2a1. System indicates the errors to user.

Use cases ends.

2b. The index is not valid.

2b1. System indicates the errors to user.

Use cases ends.

#### Use case: Find keywords

**MSS**

1. User requests to find keyword.
2. To-Do List shows the requested keywords in all categories.   
   Use case ends.

**Extensions**

2a. Keyword does not exist in the list.

Use case ends.

#### Use case: Task Due Command

**MSS**

1. User requests to find all tasks due by end date.
2. To-Do List shows all of the tasks due by end date.   
   Use case ends.

**Extensions**

2a. No tasks are due by end date.

Use case ends.

#### Use case: Redo Command

**MSS**

1. User types the command.
2. To-do List reverses the changes caused by the most recent undo.   
   Use case ends.

**Extensions**

1a. No recent undo is called.

1a1. System indicates the error and shows the error message.

Use case ends.

#### Use case: Mark Command

**MSS**

1. User marks task of TASK\_NUMBER done.
2. To-Do List shows if task could be marked as done.   
   Use case ends.

**Extensions**

2a. No such task of TASK\_NUMBER.

2a1. To-Do List shows an error message.

Use case ends.

2b. Task of TASK\_NUMBER is already marked done.

Use case ends.

#### Use case: Unmark Command

**MSS**

1. User marks task of TASK\_NUMBER undone.
2. To-Do List shows if task could be marked as undone.   
   Use case ends.

**Extensions**

2a. No such task of TASK\_NUMBER.

2a1. To-Do List shows an error message.

Use case ends.

2b. Task of TASK\_NUMBER is already marked undone.

Use case ends.

# Appendix C : Non Functional Requirements

1. The program should work on any mainstream OS as long as it has Java 1.8.0\_60 or higher installed.
2. It should be able to hold up to 1000 tasks.
3. Automated unit tests and open source code for this program should be readily available.
4. Every operation executed should be logged to the log file.
5. The program should favour DOS style commands over Unix-style commands
6. The product should not have dependencies on other packages.
7. The software can be launched, without installing, by clicking on the executable file.

# Appendix D : Glossary

##### Mainstream OS

##### Windows, Linux, Unix and OS-X

##### Deadline

##### A time interval with the start day as the day the task was created and the end day represents the date of the deadline.

##### Done

The built-in category in the To-Do list which stores all the tasks that are marked as “done”.

# Appendix E : Product Survey

### Review of [TickTick](https://ticktick.com/):

#### Strengths:

* Desktop software is provided, so we can launch it quickly without using a browser.
* Shortcuts for opening the software is provided, so that the todo lists can be opened quickly for those who prefer using the keyboard.
* User can create their own categories for tasks and allocate tasks to different categories.
* Elegant GUI is provided; the UI is not wordy and icons are quite intuitively.

#### Weaknesses:

* A constant network connection is required. If there is no network connection, the software cannot be opened.
* The parser for input text can only deal with simple command.
  + E.g. Adding the start time of event. If the command cannot be recognized, it will be automatically added as task title.

### Review of [WunderList](https://www.wunderlist.com/zh/):

#### Strengths:

* Ease of usage is its biggest strength. A user can easily add multiple items just by entering his desired items.
* Apple Watch integration is a nice bonus for Apple Watch owners.
* Slick user interface that allows background customizations.

#### Weaknesses:

* The free version has very limited features. Users only get 25 assigns per shared to-do list and 10 background choices.
* A constant network connection is required. If there is no network connection, the software cannot be opened.
* Wunderlist lacks IFTTT integration compared to other to-do list applications.

### Review of [Trello](https://trello.com/)

#### Strengths:

* Online/Cloud based program that allows it to be transferrable to other computers.
* Ease to add notes and description into the Trello cards.
* Customizable looks.

#### Weaknesses:

* It cannot link up with other calendar software like Google calendar, which makes it hard to keep track of tasks done.
* The free version is much more limited than the paid version, hence making certain customisation features difficult to accomplish.

### Review of [Google Calendar](https://calendar.google.com/)

#### Strengths:

* Adds different kind of colouring to the schedule.
* Undo addition or deletion of events
* Creates multiple calendar for different purposes
* GUI is quite intuitive. The formatting is clear and it does not require guidelines
* Able to use calendar in offline mode.

#### Weaknesses:

* Unable to view all deleted events or reminders.
* It does not have command-line inputs to modify the calendar; most operations require a user to click, which can be time-consuming.
* Only accessible via browsers; no desktop application available.