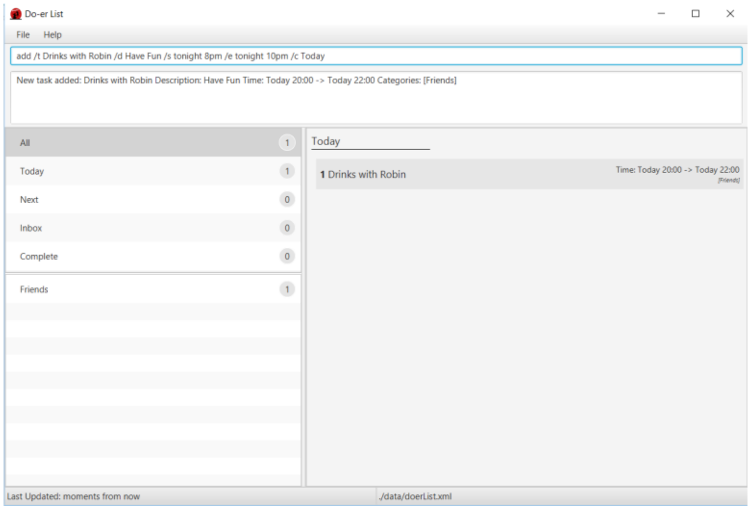

```
add /t Drinks with Robin /d Have Fun /s tonight 8pm /e tonight 10pm /c Today
```

```
New task added: Drinks with Robin Description: Have Fun Time: Today 20:00 -> Today 22:00 Categories: [Friends]
```

```
1 Drinks with Robin                Time: Today 20:00 -> Today 22:00
                                                          [Friends]
```

```
Last Updated: moments from now                ./data/doerList.xml
```

DEVELOPER GUIDE

TEAM PGP

ABSTRACT

This developer guide is a comprehensive document for all avid software engineers that are contributing to our to-do list application "Do-*er* List". We welcome all of you on board, to be a contributor of this awesome, life-changing application.

CS2101 / CS2103T

Effective Communication for Computing Professionals & Software Engineering

Contents

I. Purpose Statement and Overview

This developer guide is a comprehensive document for all avid software engineers that are contributing to our to-do list application "Do-*er* List". We welcome all of you on board, to be a contributor of this awesome, life-changing application.

This guide is maintained by the 4 original developers, Xiaopu, Jason, Benedict and Hai Long.

Do-er List is the next revolutionary to-do list application that is designed so extraordinarily where every command is so deceptively short yet so functionally powerful. So get ready to immerse yourself into our application!

II. Quick Start

Here are the basic steps needed to get started with 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
3. ***e(fx)clipse*** plugin for ***Eclipse***
4. 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 repo 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 `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*** changes any settings files during the import process, you can discard those changes.

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

III. Design

1. Overall Architecture

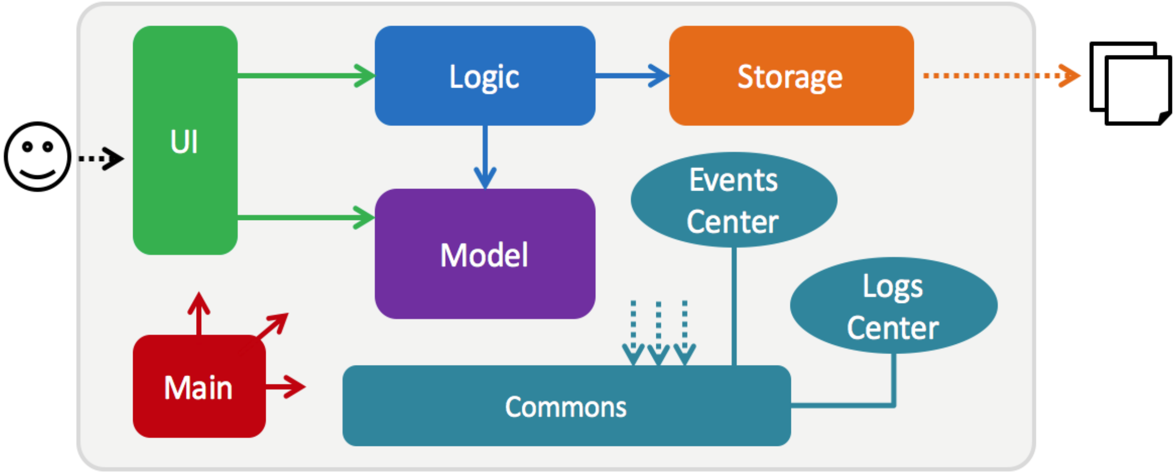

Figure 1: Architecture Diagram of high-level design

The *Architecture Diagram* 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 invoke 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 application's log file.

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

1. **UI**: The user interface of the application.
2. **Logic**: The command executor.
3. **Model**: Holds the data of the application 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.

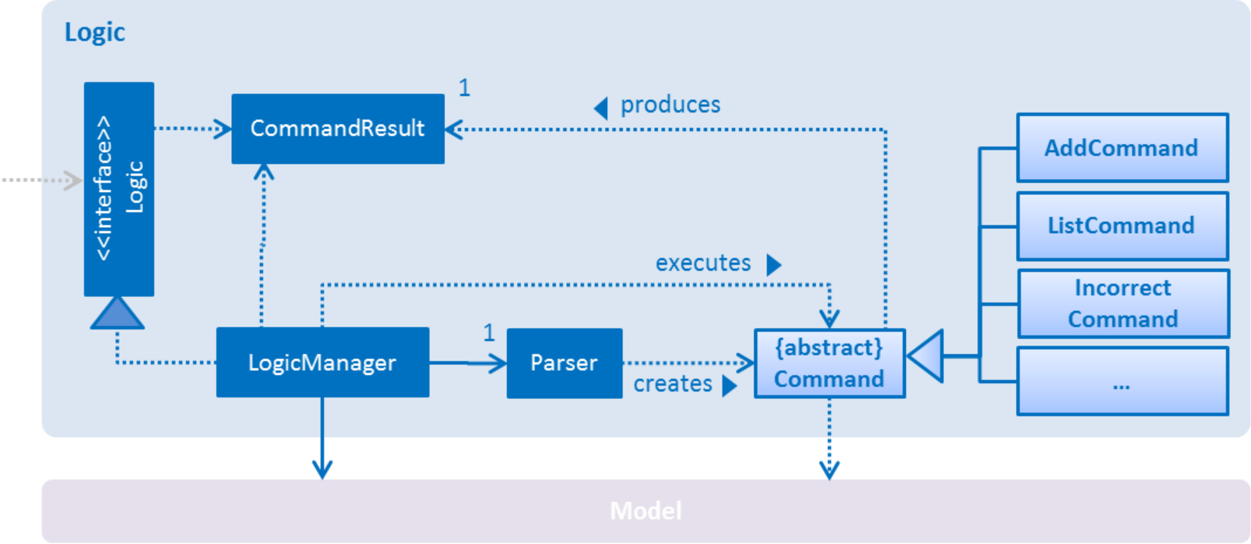

Figure 2: Architecture Diagram of the ***Logic*** *Component*

When a user issues a command, each component in the architecture diagram works together to deliver the result. *Figure 3* (*Sequence Diagram*) shows how the components interact with each other when the user issues the command `delete 1`.

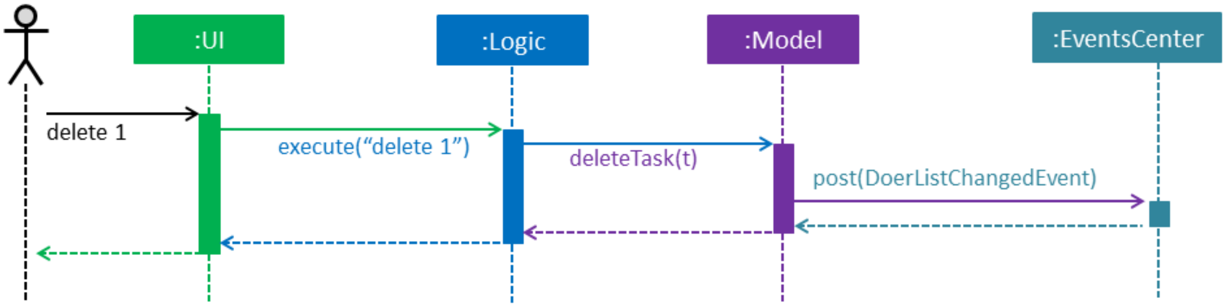

Figure 3: Sequence Diagram when user issues the command `delete 1`

Note how **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.

Figure 4 shows how **EventsCenter** reacts to that event, which 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.

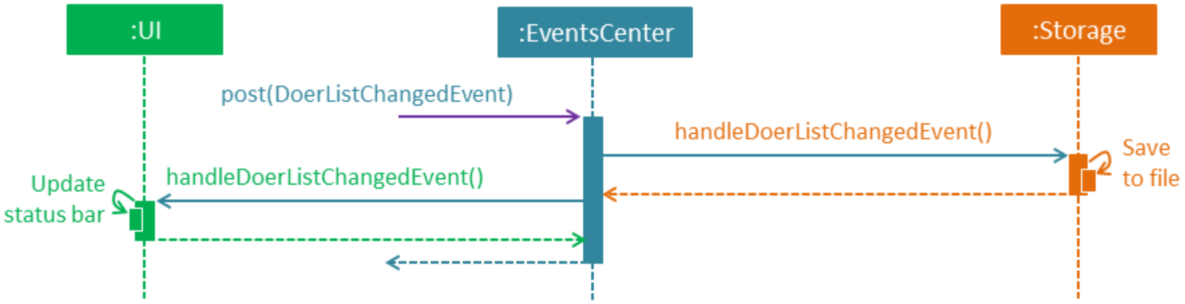
:UI
:EventsCenter
:Storage
post(DoerListChangedEvent)
handleDoerListChangedEvent()
Save
to file
Update
status bar
handleDoerListChangedEvent()

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

Notice how the event is propagated through the **EventsCenter** to the **Storage** and **UI** without having **Model** 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 the **UI**, **Logic**, **Model** and **Storage**. We will be exploring their class diagrams and their APIs to understand how they work and interact with each other to deliver a result. 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. *Figure 5* is its architecture diagram, displaying its smaller components and how they relate to each other.

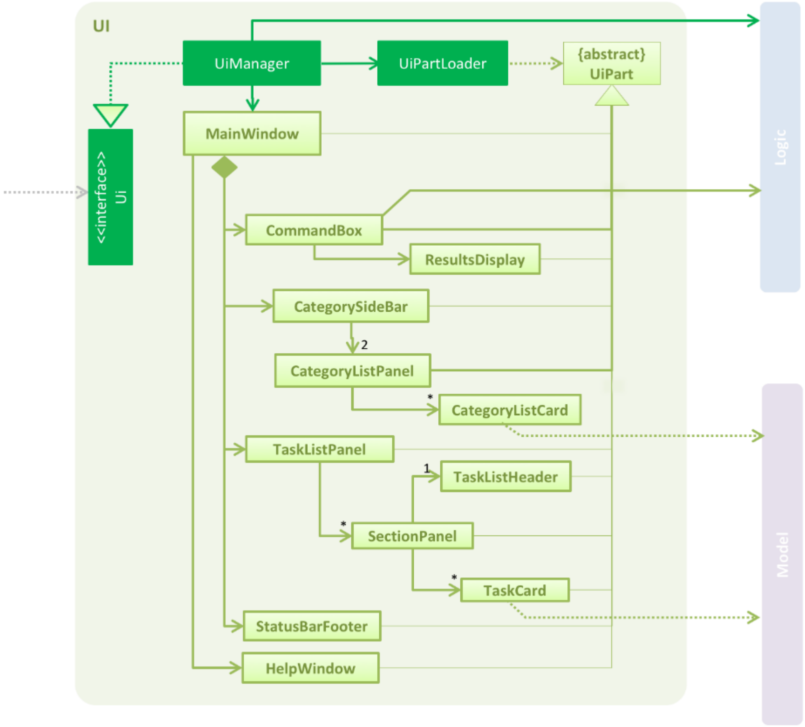

Figure 5: Architecture Diagram of the ***UI*** *Component*

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`, `TaskCard` and `HelpWindow`. 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`.

The crucial roles of the **UI** component are:

1. Executes user commands using **Logic** component.
2. Binds itself to some data in **Model** so that the **UI** can auto-update when data in **Model** change.

3. Responds to events raised from various parts of the application and updates the **UI** accordingly. *Figure 6* shows the interactions between the **UI** and `JumpToIndexedTaskRequestEvent` event.

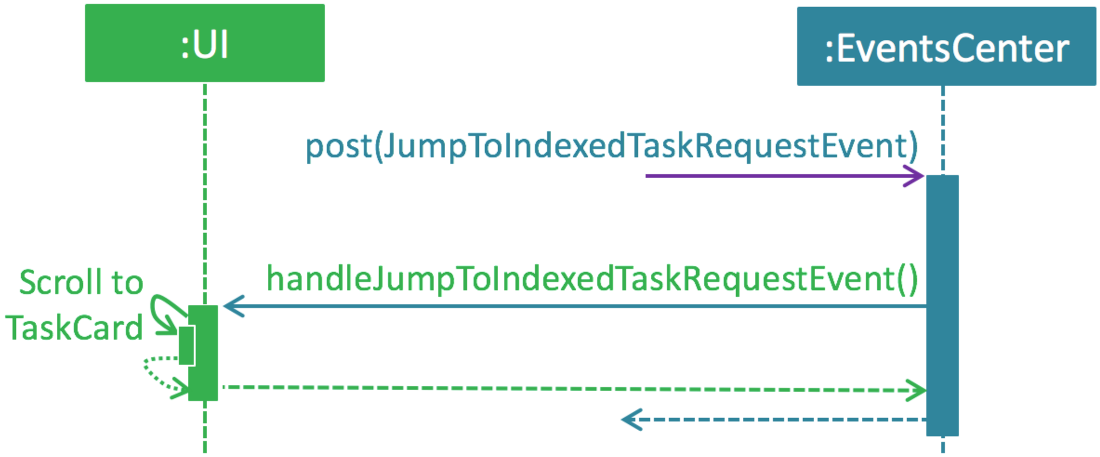
:UI
:EventsCenter
post(JumpToIndexedTaskRequestEvent)
handleJumpToIndexedTaskRequestEvent()
Scroll to
TaskCard

*Figure 6: Interactions between **UI** and **EventsCenter** components diagram*

3. Logic component

The **Logic** component is responsible for the logical functionalities in the application that includes all the possible commands and the parsing of users' inputs. *Figure 7* below shows the architecture diagram of the **Logic** component.

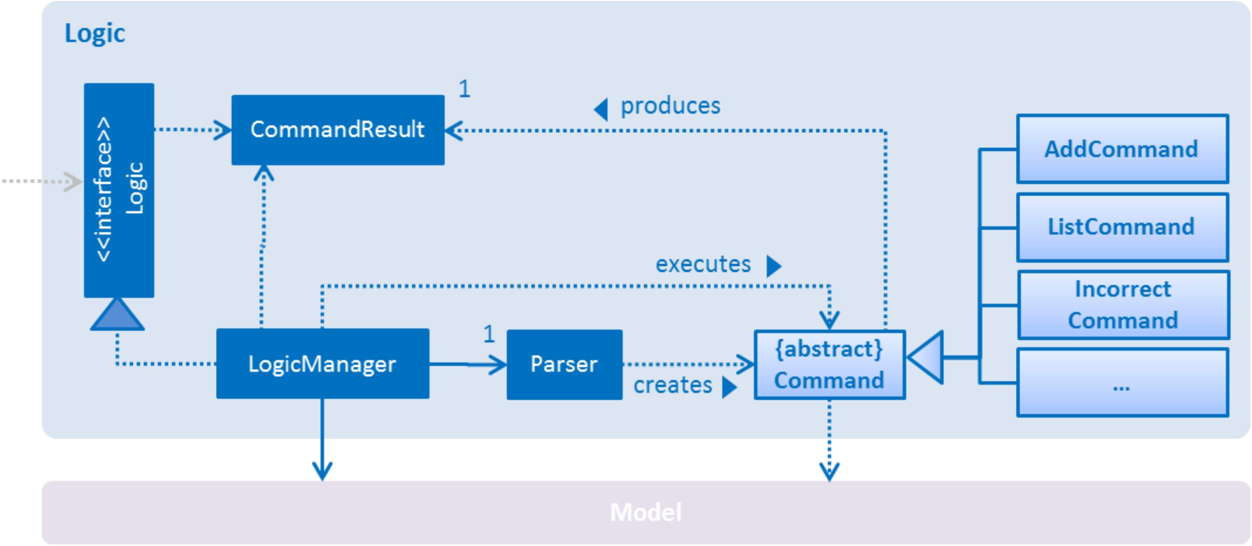

*Figure 7: Architecture Diagram of the **Logic** component*

This is what happens when a user types in a command:

Logic uses the `Parser` class to parse the user's command. This creates a `Command` object which will be executed by the `LogicManager`. The command execution can affect **Model** (i.e. 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 give you a concrete example, *Figure 8* (*Sequence Diagram*) shows the interactions within the **Logic** component when the API calls `execute("delete 1")`.

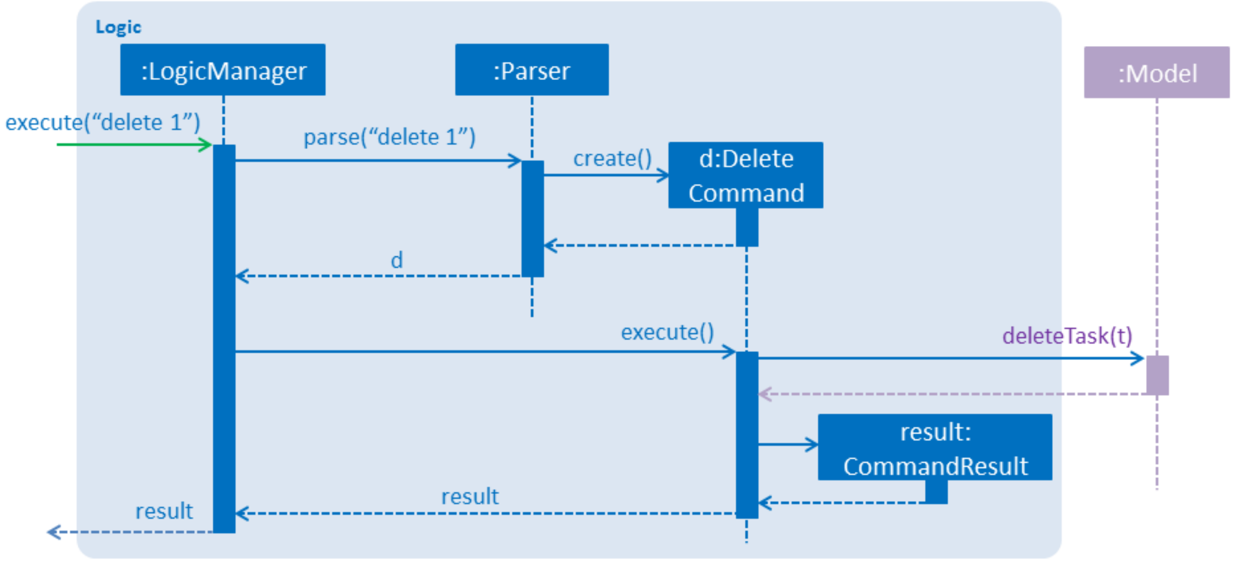

Figure 8: Sequence Diagram when `delete 1` is called.

4. Model component

The **Model** component stores the Do-er List's data into specific components. *Figure 9* gives an overall view of the architecture diagram of the **Model** component.

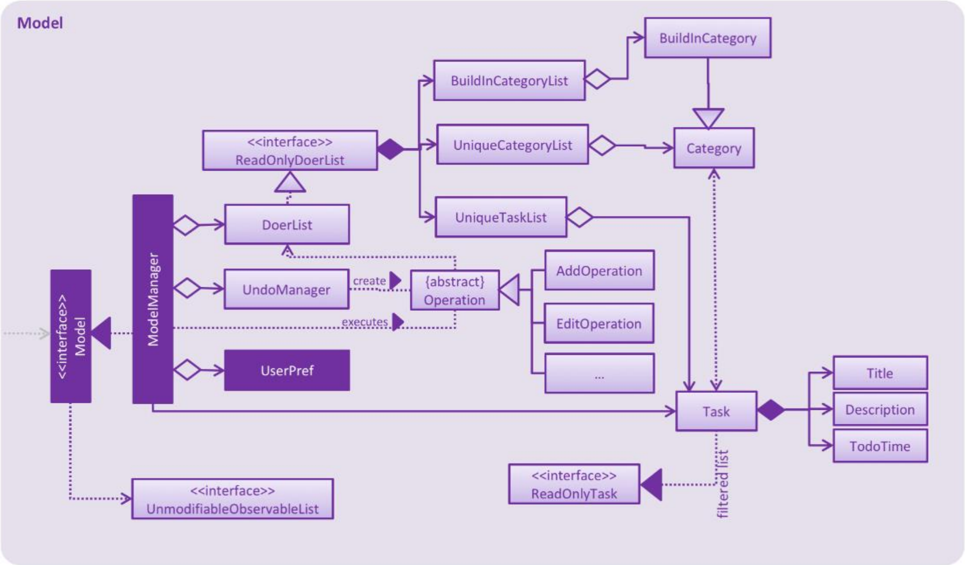

*Figure 9: Architecture Diagram of the **Model** Component*

Model stores a `UserPref` object that represents the user's preferences besides storing the Do-er List data. It also stores an `UndoManager` object that records the execution of all commands in the data.

More importantly, it exposes an `UnmodifiableObservableList<ReadOnlyTask>` and `UnmodifiableObservableList<Category>` that can be *observed*. It means that the **UI** can be bounded to this list so that it automatically updates when the data in the list changes, thus reflecting real-time changes.

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

5. Storage component

The **Storage** component, as 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.

Refer to *Figure 10* for the complete architecture diagram of this component.

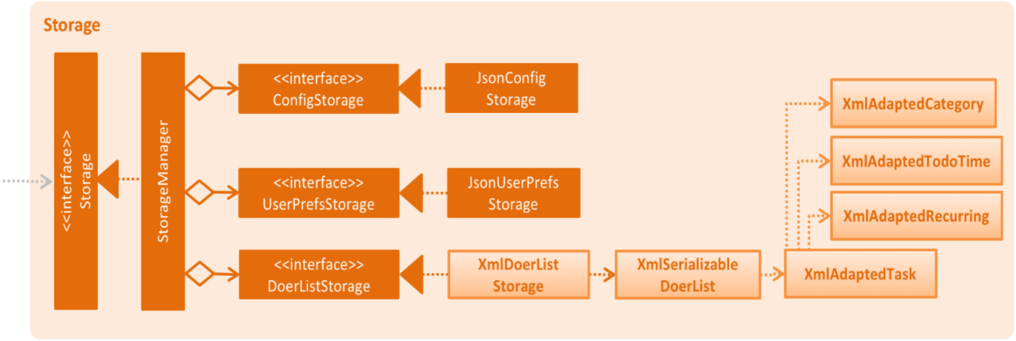

Figure 10: Architecture Diagram of ***Storage*** *component*

6. Common classes

Classes used by multiple components are in the `seedu.doerList.commons` package.

IV. 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**: Proceed with caution.
3. **INFO**: Information showing the noteworthy actions by the application.
4. **FINE**: Details that are usually not noteworthy but may be useful in debugging (print the actual list instead of just its size).

2. Configuration

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

V. Testing

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

1. Running Test in *Eclipse*

If you are not using a recent version of ***Eclipse*** (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

If you want to run all tests, right-click on the `src/test/java` folder and choose `Run as` > `JUnit Test`. If you 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 run 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,
   i. *Unit tests* targeting the lowest level methods or classes.
      - `seedu.doerList.commons.util.UrlUtilTest`
   ii. *Integration tests* that are checking the integration of multiple code units (those code units are assumed to be working).
      - `seedu.doerList.storage.StorageManagerTest`
   iii. Hybrids of unit and integration tests. These tests checks multiple code units as well as their connectedness.
      - `seedu.doerList.logic.LogicManagerTest`

Headless GUI Testing: Thanks to the ***TestFX*** library, our GUI tests can be run in *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.md` to learn how to run tests in headless mode.

VI. Dev Ops

1. Build Automation

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

2. Continuous Integration

We used ***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 repository with the version number. (i.e. v0.1)
3. Create a new release using ***GitHub*** and upload the `JAR` file your 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 them from their respective libraries.

VII. 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 the usage format of all commands	use various commands in the application
* * *	user	create a task with title and description	summarise my task into a title and provide details in its description
* * *	user	create a task without start or end time	record tasks that need to be done without a deadline
* * *	user	create a task with start and end time or deadlines	prepare for the task that is happening or due at certain time
* * *	user	edit the task's title, description, start time, end time and categories	update my task in the event a mistake is made
* * *	user	view all tasks	have an overview of all 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 with certain keywords in the event I forget
* * *	user	delete tasks	remove unwanted tasks
* *	user	add tasks to different categories	organise my tasks more effectively
* *	user	view the tasks under a certain category	examine different tasks under different categories
* *	user	view the tasks that are going to happen or due today, tomorrow or in the next 7 days.	remind myself about the upcoming tasks
* *	user	undo the most recent operations	revert back to my unmodified data
* *	user	redo the most recent operations	revert to my modified data

Priority	As a	I want to ...	So that I can...
* *	user	specify a storage location for data storage	synchronise with online data servers in the event I lose my data locally
* *	user	mark or unmark the task as done or undone	choose to only keep track of the tasks which are needed to be done and archive them when completed.
*	user	type command parameters in arbitrary order	choose not to remember the order of parameters but still able to use the command properly
*	user	add external `ical` file to the to do-lists	keep track of other events created by others
*	user	create recurring tasks	be reminded to do the same task at every fixed-time-interval
*	user	view events in Google Calendar	have a pictorial view of my schedule.

VIII. Appendix B: Use Cases

(For all of the 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 a 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 is followed by the wrong parameters.
  1a1. System indicates the error and display the correct format for user.
  Use case ends.
1b. `TITLE` is an 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's details are changed accordingly.
    - title, description, start time, end time, category
4. System displays the details of the newly edited task.
   Use case ends.

Extensions

1a. `edit` command is 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`
  1b1. System indicates the error that the `INDEX` is non-existent.
  Use case ends.
1c. `TITLE` is an 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 tasks 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 which involves change of data.
3. System undoes the operation.
4. System indicates the change to user.
   Use case ends.

Extensions

2a. The last operation which involves 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 display 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 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 detail 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. `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 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 the task of `TASK_NUMBER` done.
2. To-Do List shows if the specific 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 as 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.

Use case: Save Command

MSS

1. User attempts to save his data in a specified location.
2. To-Do List shows if the data could be save in that location.
   Use case ends.

Extensions

2a. Invalid `SAVE_LOCATION` specified.
    2a1. To-Do List shows an error message.
    Use case ends.

IX. 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 have no dependency on other packages.
7. The software can be launched, without installing, by clicking on the executable file.

X. Appendix D : Glossary

Mainstream OS

Windows, Linux, Unix and OS-X

Deadline

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

Done or Undone

The build-in category in the To-Do list which store all the tasks that are marked as `done` or `undone` respectively.

XI. Appendix E : Product Survey

Review of TickTick:

Strengths:

- Desktop software is provided, so we can launch it quickly without using browser.
- Shortcuts for opening the software is provided, so the To-Do lists can be opened quickly to those who prefer using the keyboard.
- Users can create their own category 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.
    - Addition of the start time of event. If the command cannot be recognized, it will be automatically added as task title.

Review of WunderList:

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 customization.

Weaknesses:

- The free version is seriously limited. 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:

Strengths:

- Cloud-based (Online) program that allows it to be transferrable to other computers.
- Ease to add notes and descriptions 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, making certain customisation features is hard

Review of Google Calendar:

Strengths:

- Add different kinds of colouring to the schedule.
- Undo addition or deletion of events.
- Create 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 and troublesome.
- Only accessible via browsers; no desktop application available