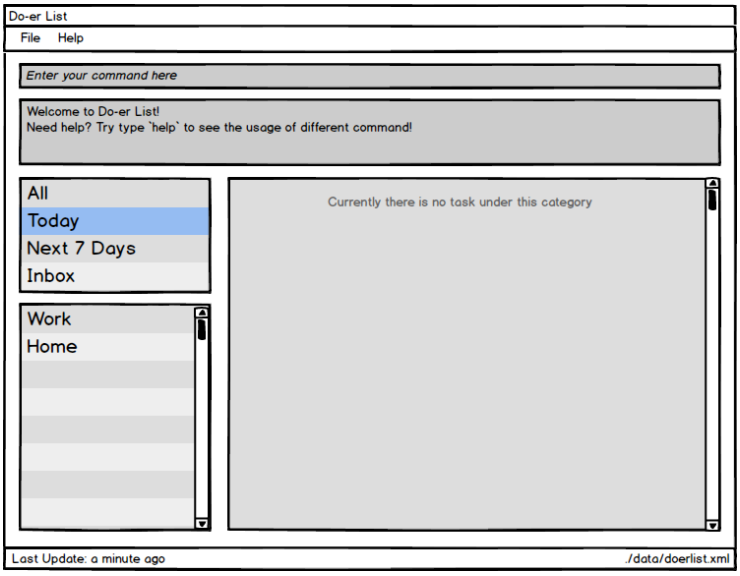

DEVELOPER GUIDE

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

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.

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

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 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
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' config 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 changes any settings files during the import process, you can discard those changes.

And that is it! You are now officially work on our program!

III. Design

1. Overall Architecture

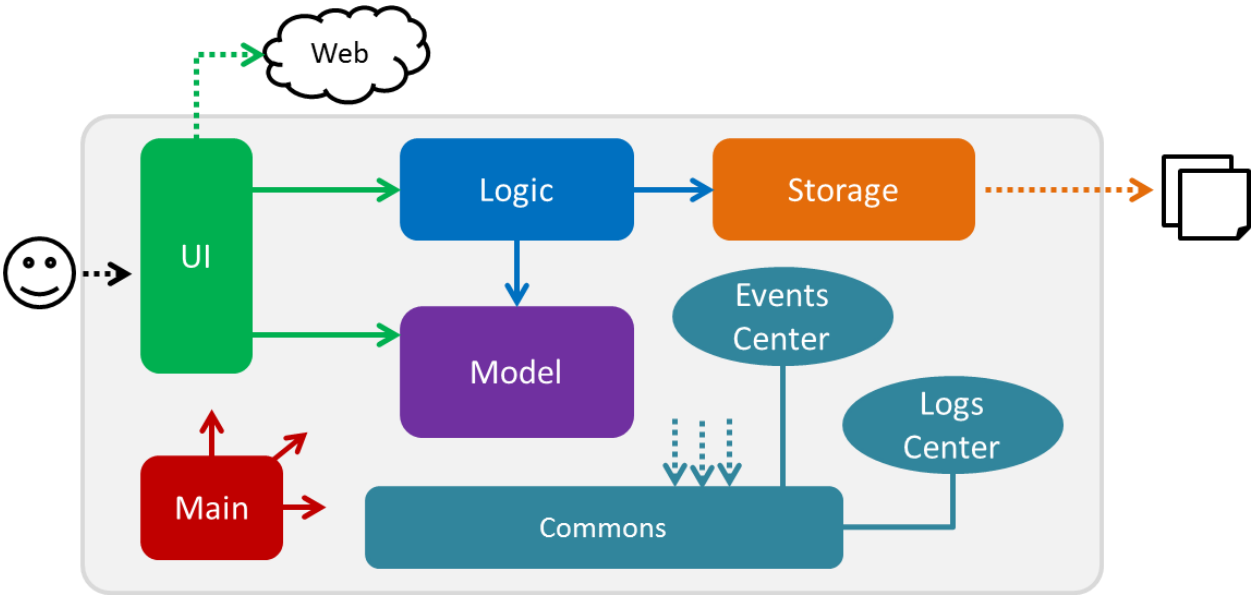

Figure 1: Architecture Diagram of high-level design

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

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

1. At app 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 cleanup 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 it's API in the **Logic.java** interface and exposes its functionality using the **LogicManager.java** class.

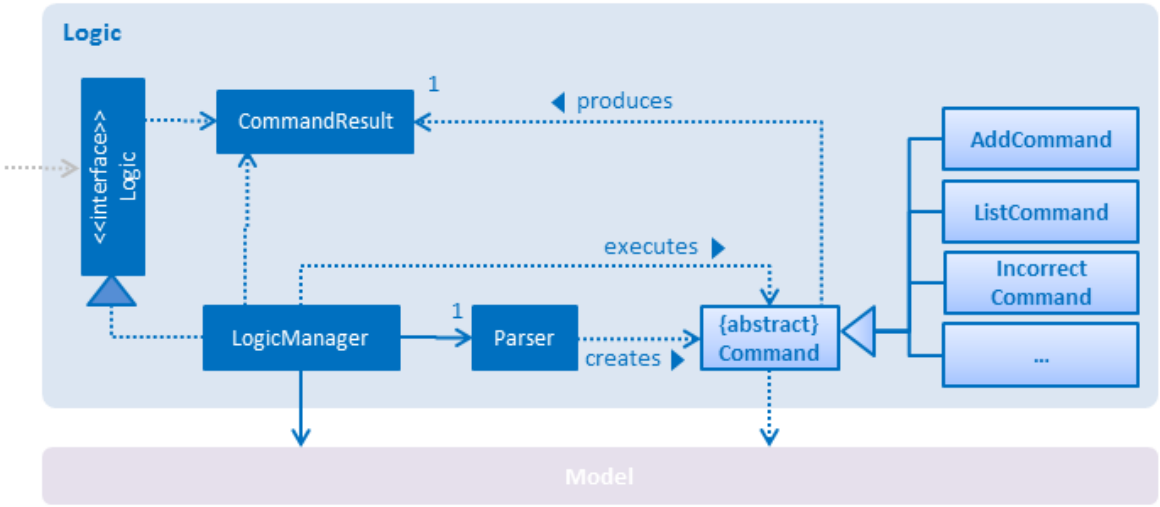

Figure 2: Architecture Diagram of the Logic Component

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**.

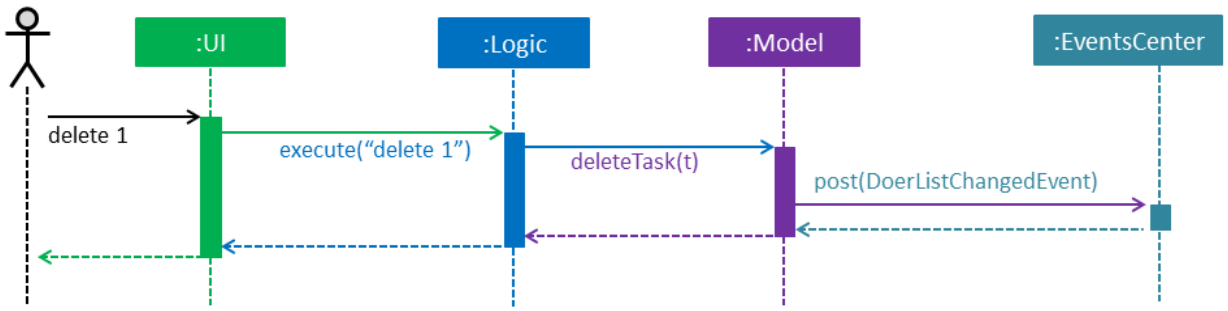

Figure 3: Figure Diagram when user issues the command delete 1

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.

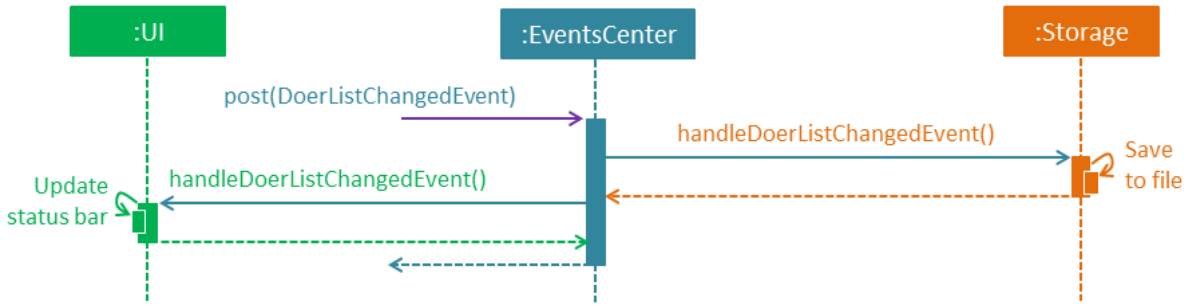

Figure 4: How the EventsCenter react to a change in event

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 diagram and their API 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 (figure 5) is its architecture diagram, displaying its smaller components and how they relate to each other.

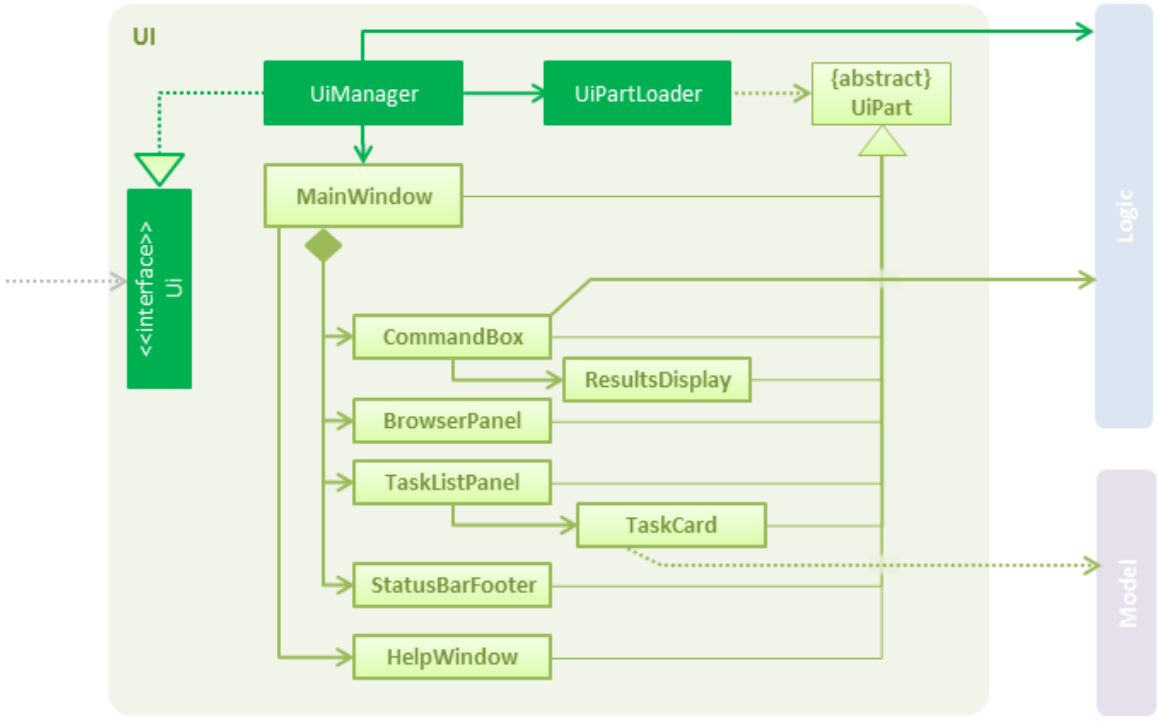

Figure 5: Architecture Diagram of the UI Component

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 change.
3. Responds to events raised from various parts of the App and updates the UI accordingly.

3. Logic component

The Logic component is responsible for logic in the app, such as all the possible commands (add, delete, list, etc) and parsing user's inputs. Figure 6 belows shows the architecture diagram of the Logic component.

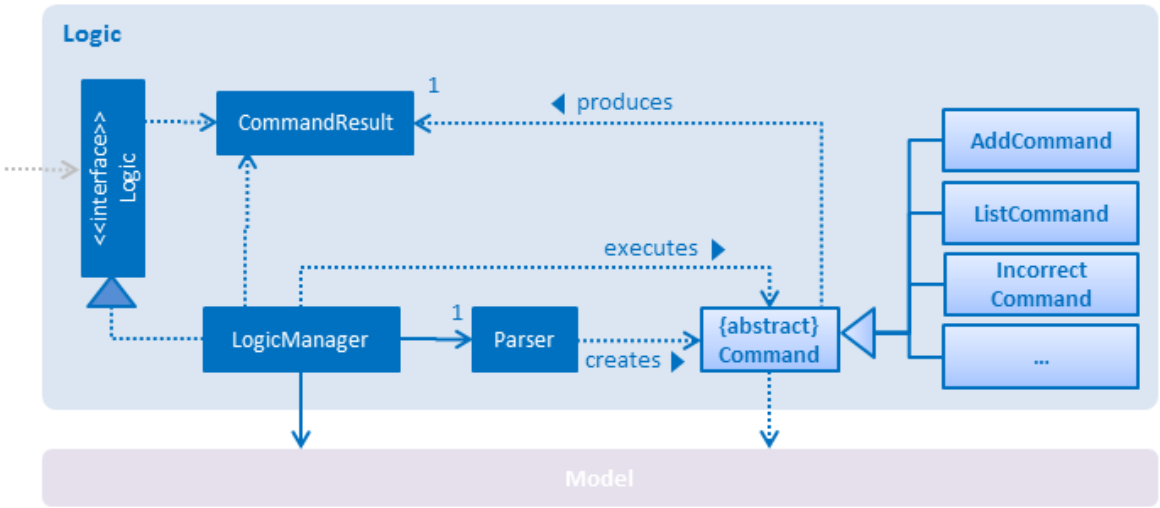

Figure 6: Architecture Diagram of the Logic component

What happens when the user type in a command?

First of all, 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 give you a concrete example, below is the Sequence Diagram for interactions within the Logic component for the execute("delete 1") API call.

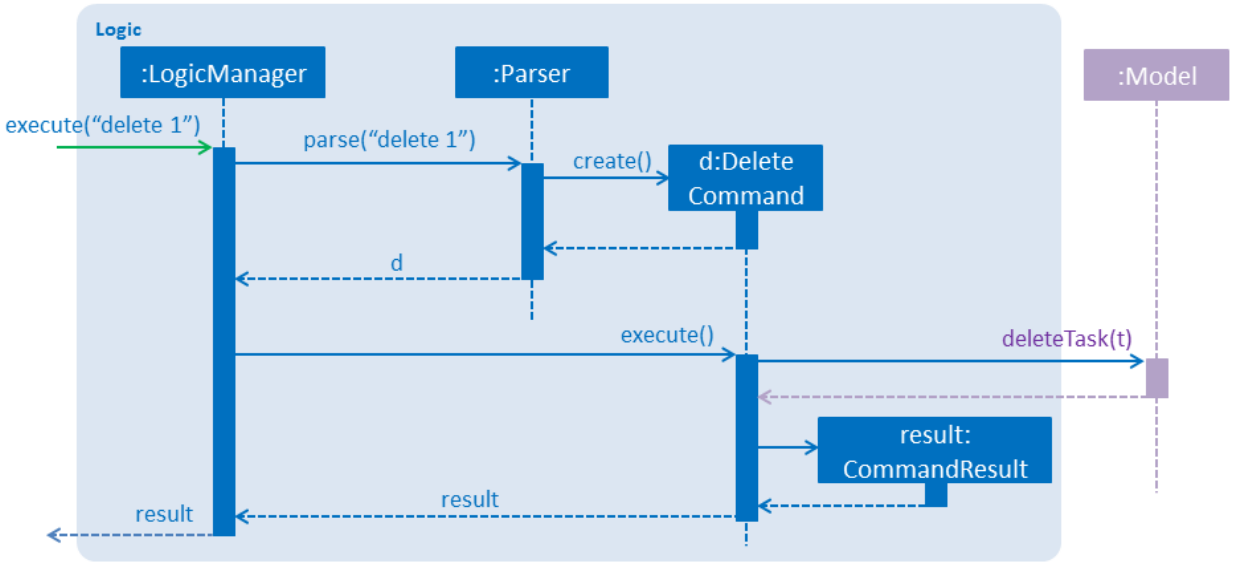

Figure 7: Sequence Diagram when "delete 1" is called.

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.

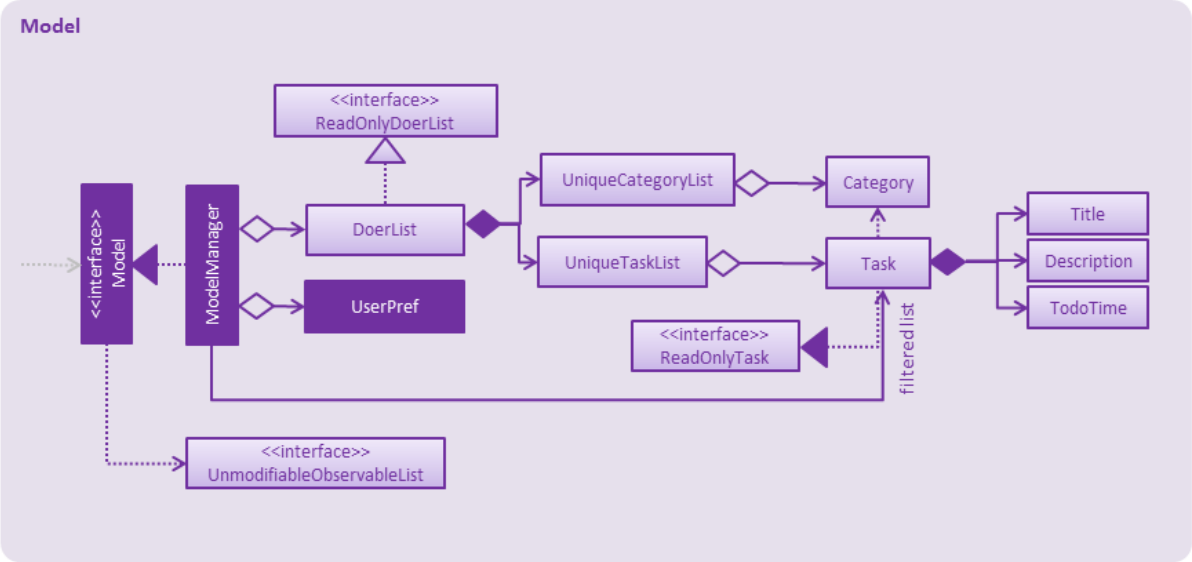

Figure 8: Architecture Diagram of the Model Component

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 be 'observed'. It means that the UI can be bound to this list so that the UI automatically updates when the data in the list change, thus reflecting real-time changes.

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 read it back.

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

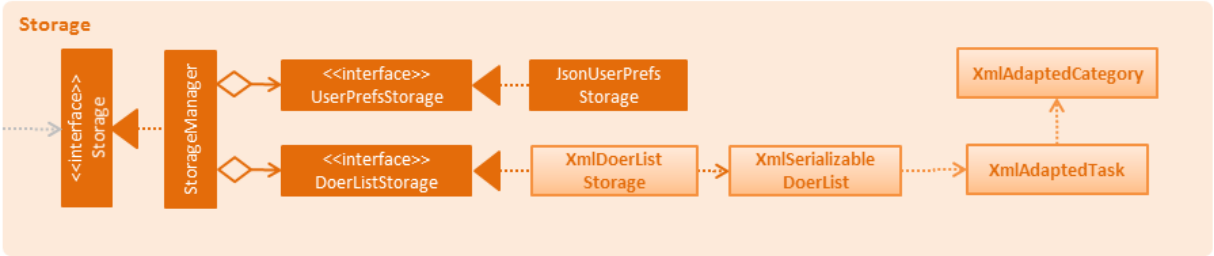

Figure 9: Architecture Diagram of Storage component

6. Common classes

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

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: Can continue, but with caution
3. INFO: Information showing the noteworthy actions by the App
4. FINE: Details that is 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 App 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 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

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. for how to run tests using Gradle.

We have two types of tests:

1. GUI Tests - These are *System Tests* that test the entire App 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/classes.
    ii. e.g. seedu.doerList.commons.util.UrlUtilTest
    iii. *Integration tests* that are checking the integration of multiple code units (those code units are assumed to be working).
    iv. e.g. seedu.doerList.storage.StorageManagerTest
    v. Hybrids of unit and integration tests. These test are checking multiple code units as well as how the are connected together.
    vi. e.g. seedu.doerList.logic.LogicManagerTest

Headless GUI Testing : Thanks to the TestFX library we use, 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 the 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 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. Crete 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 from each library.