main

Suru Task Manager

View on GitHub

Suru - Developer Guide

By : `Team W09-B3` Since: `Feb 2017` Licence: `?`

---

1. Introduction

Welcome to Suru, the innovative taskal assistant designed to help you manage your tasks like a boss. This developer guide aims to document every feature of Suru so you can get started contributing to this project. The guide teaches you all you need to know from setting up your development environment to deploying Suru for production.

2. Setting up

2.1. Prerequisites

1. Download and install **JDK version** `1.8.0_60` or later.

   > Having any Java 8 version is not enough.
   > This app will not work with earlier versions of Java 8.

2. Download and install **Eclipse** IDE.
3. Download and install **e(fx)clipse** plugin for Eclipse (Follow from step 2 onwards given in this page).
4. Download and install **Buildship Gradle Integration** plugin from the Eclipse Marketplace.
5. Download and install **Checkstyle Plug-in** plugin from the Eclipse Marketplace.

2.2. Importing the project into Eclipse

1. Fork this repo, 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`

   Note:

   > - If you are asked whether to 'keep' or 'overwrite' config files, choose to 'keep'.
   > - Depending on the speed of your connection and server load, it can take up to 30 minutes for the set up to complete (this is because Gradle downloads library files from servers during the project set-up process).
   > - If Eclipse automatically changed any settings during the import process, you can discard those changes.

2.3. Configuring Checkstyle

1. Click `Project` -> `Properties` -> `Checkstyle` -> `Local Check Configurations` -> `New...`
2. Choose `External Configuration File` under `Type`.
3. Enter an arbitrary configuration name e.g. taskmanager.
4. Import checkstyle configuration file found at `config/checkstyle/checkstyle.xml`.
5. Click `OK` once, go to the `Main` tab, use the newly imported checkstyle configuration.
6. Tick and select `files from packages`, click `Change...`, and select the `resources` package.
7. Click OK twice. Rebuild project if prompted.

Note:

> Click on the `files from packages` text after ticking in order to enable the `Change...` button.

2.4. Troubleshooting project setup

Problem: Eclipse reports compile errors after new commits are pulled from Git

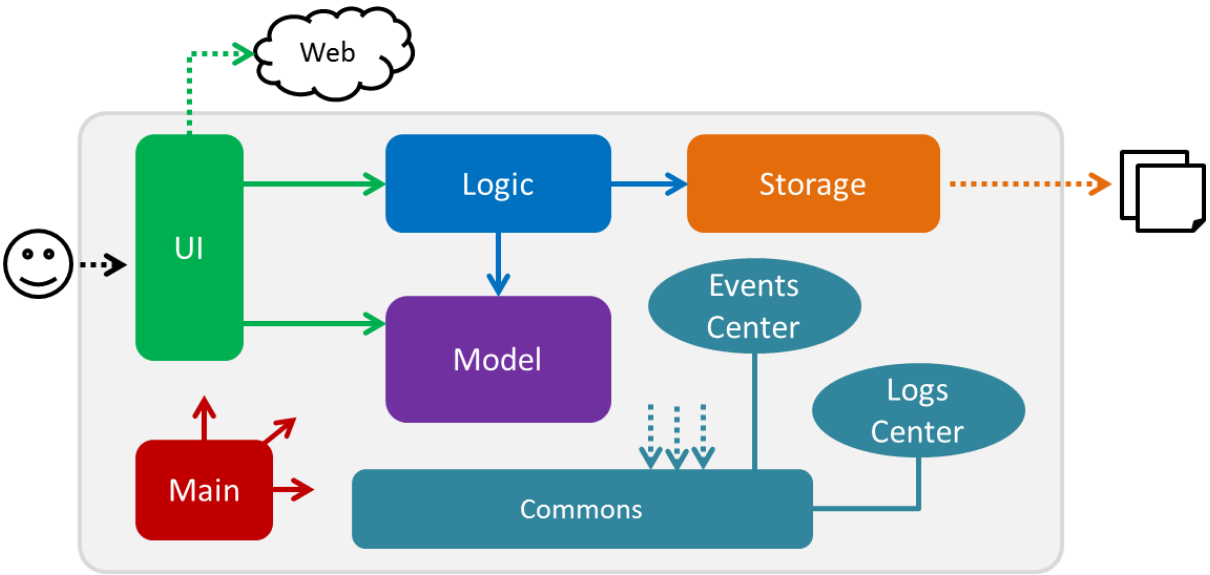
- Reason: Eclipse fails to recognize new files that appeared due to the Git pull.
- Solution: Refresh the project in Eclipse:
  Right-click on the project (in Eclipse package explorer), choose `Gradle` -> `Refresh Gradle Project`.

Problem: Eclipse reports some required libraries as missing

- Reason: Required libraries may not have been downloaded during the project import.
- Solution: Run tests using Gradle once (to refresh the libraries).

3. Design

3.1. Architecture



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

Note:

> - The `.pptx` files used to create diagrams in this document can be found in the diagrams folder.
> - To update a diagram, modify the diagram in the pptx file, select the objects of the diagram, and choose `Save as picture`.

`Main` has a single class called `MainApp`.

- At app launch it initializes the components in the correct sequence, and their constructors, passing necessary information to the relevant components.
- At shut down it shuts down the components and invokes cleanup methods where necessary.

`Commons` represents a collection of classes shared by multiple other components. Two of those classes play important roles at the architecture level.

- `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). e.g. use events when communicating between `Model` and `UI`.
- `LogsCenter` : This class is used by many classes to write log messages to the App's log file.

The architecture consists of four other major components.

- `UI` : Initializes the UI for the app.
- `Logic` : Executes the commands.
- `Model` : Holds the data of the app in-memory.
- `Storage` : Reads data from, and writes data to, the hard disk.

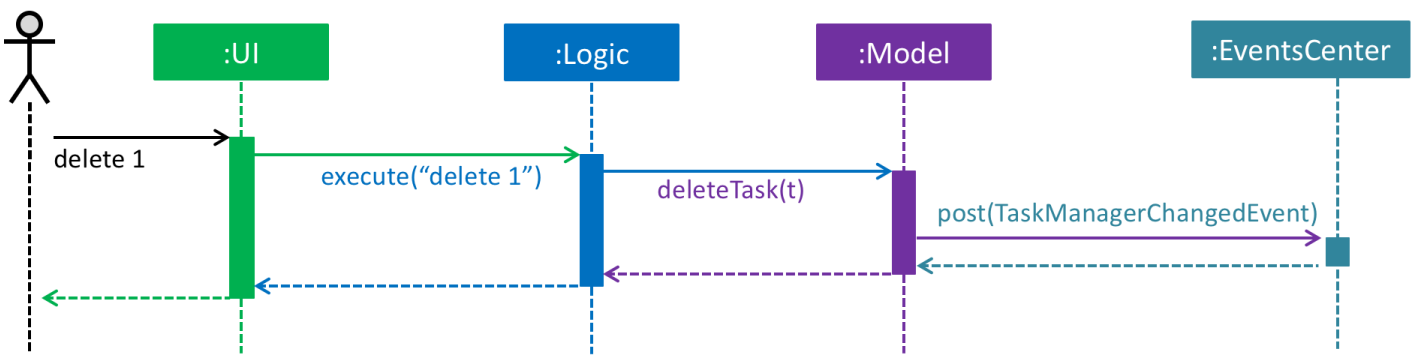
Each of the four components

- outlines all important methods in an `interface` with the same name as the Component.
- exposes its functionality using a `{Component Name}Manager` class.

For example, the `Logic` component defines its APIs in the `Logic.java` interface and exposes its functionality using the `LogicManager.java` class.

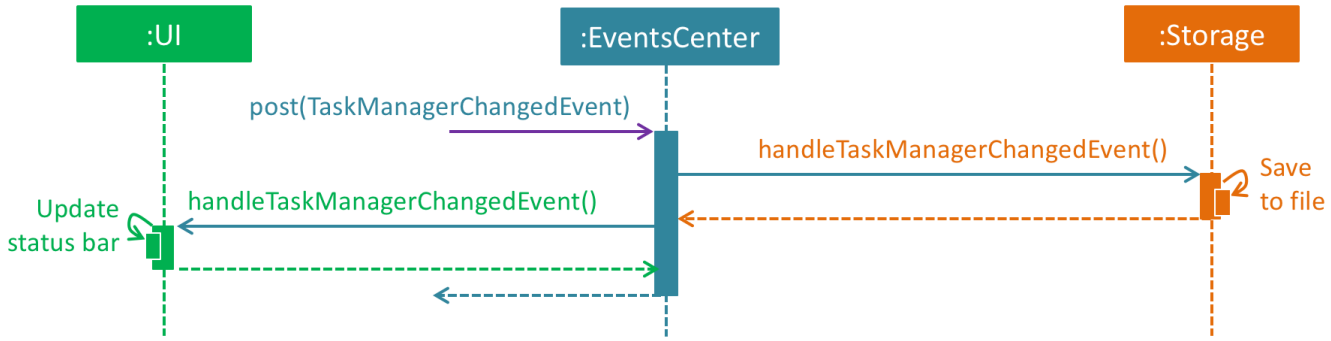
Events-Driven nature of the design

The *Sequence Diagram* below shows how the components interact for the scenario where the user issues the command `delete 1`.



> Note how the `Model` simply raises a `TaskManagerChangedEvent` when the data is changed, instead of asking the `Storage` to save the updates to the hard disk.

The diagram below 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.

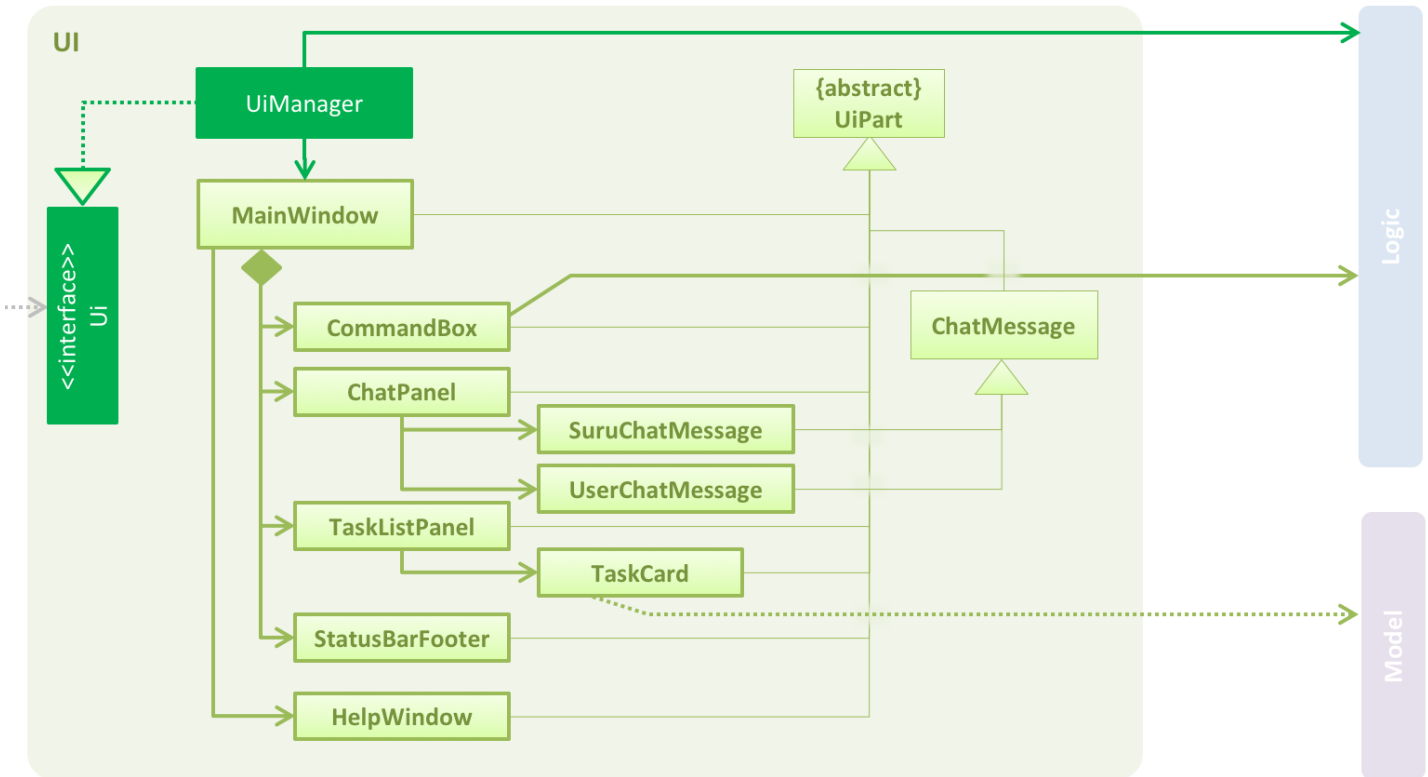


> Note how the event is propagated through the `EventsCenter` to the `Storage` and `UI` without `Model` having to be coupled to either of them. This is an example of how this Event Driven approach helps us reduce direct coupling between components.

The sections below give more details of each component.

3.2. UI component

Author: Shawn



API : `Ui.java`

The UI consists of a `MainWindow` that is made up of parts e.g. `CommandBox`, `ResultDisplay`, `taskListPanel`, `StatusBarFooter`, `BrowserPanel` etc. All these, including the `MainWindow`, inherit from the abstract `UiPart` class.

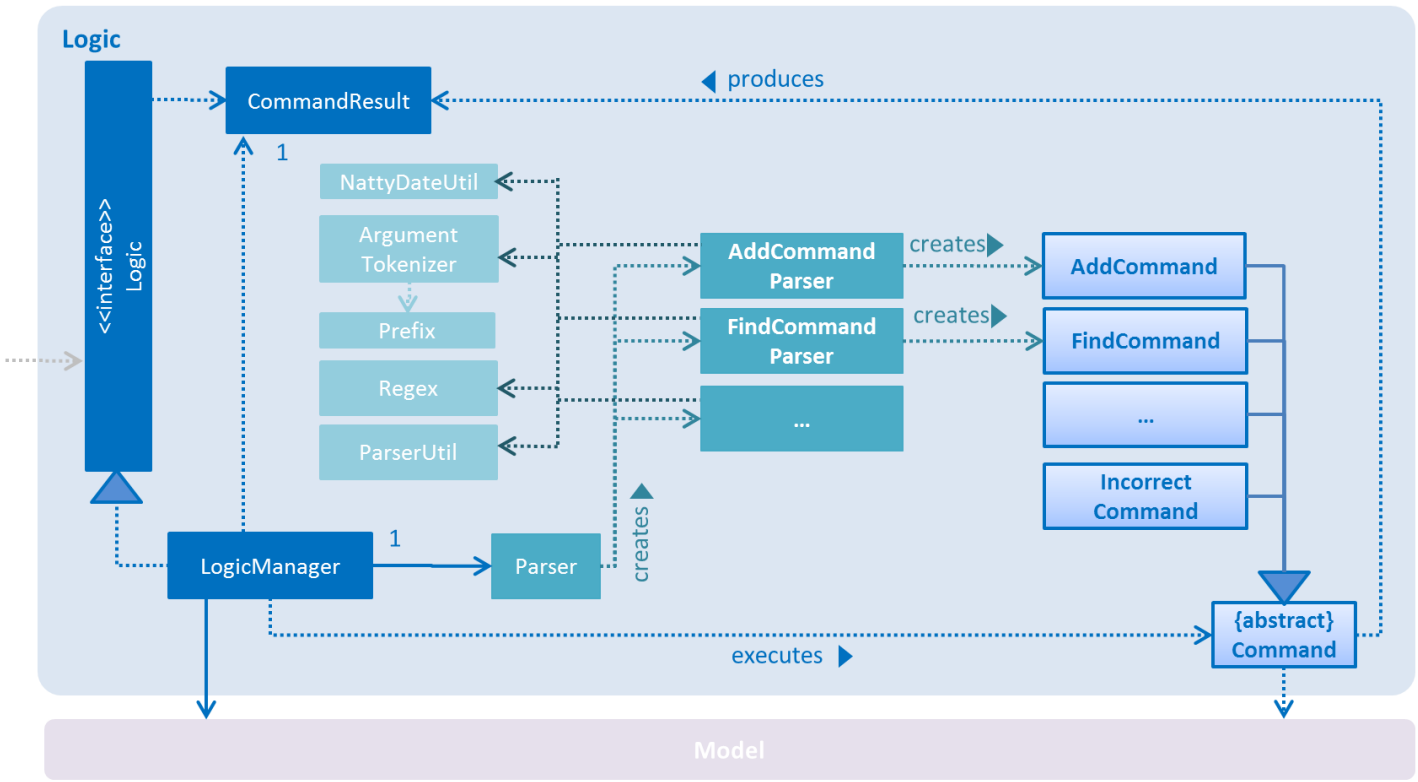
The `UI` component uses JavaFx UI framework. The layout of these UI parts are defined in matching `.fxml` files that are in the `src/main/resources/view` folder.
For example, the layout of the `MainWindow` is specified in `MainWindow.fxml`

The `UI` component

- executes user commands using the `Logic` component.
- binds itself to some data in the `Model` so that the UI can auto-update when data in the `Model` change.
- responds to events raised from various parts of the App and updates the UI accordingly.

3.3. Logic component

Author: Jeremy

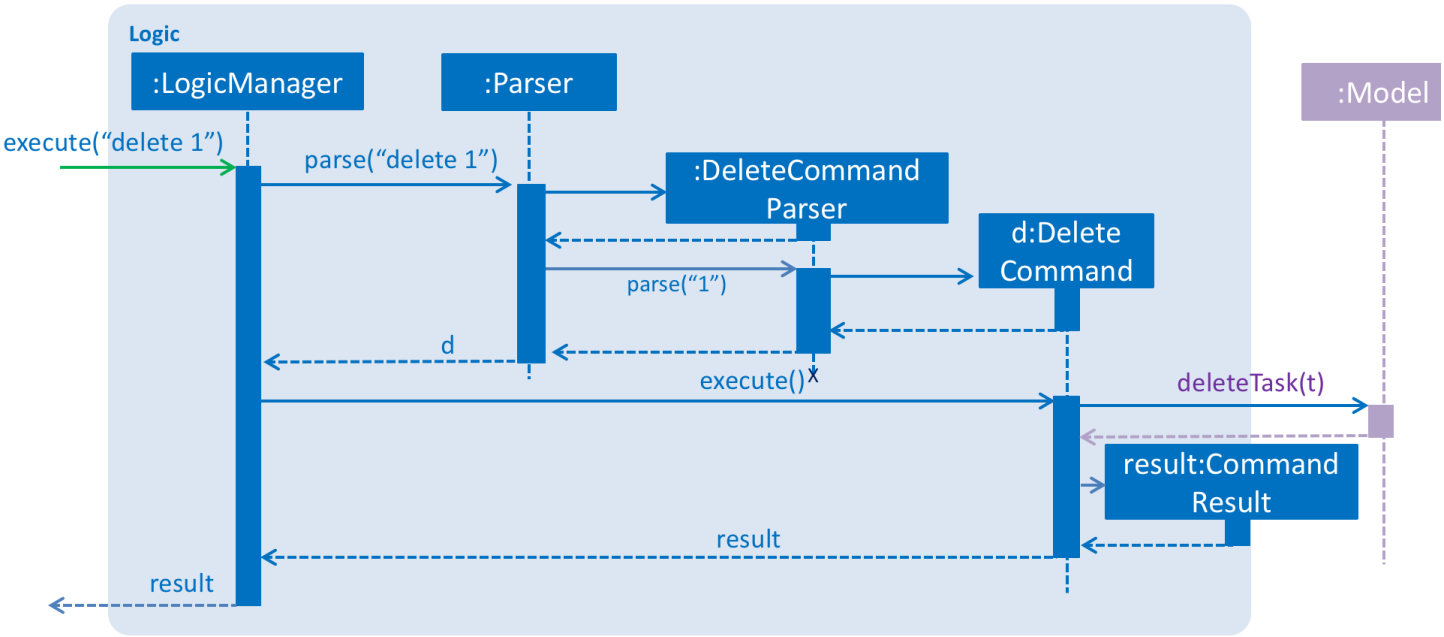


API : `Logic.java`

The `Logic` component consists of the `Logic Manager`, `Command Result`, `Parser`, `Command Parser` and `Command` classes. The logic component is responsible for parsing the input from the `UI` and affecting the corresponding `Model` objects.

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

Given below is the Sequence Diagram for interactions within the `Logic` component for the `execute("delete 1")` API call.



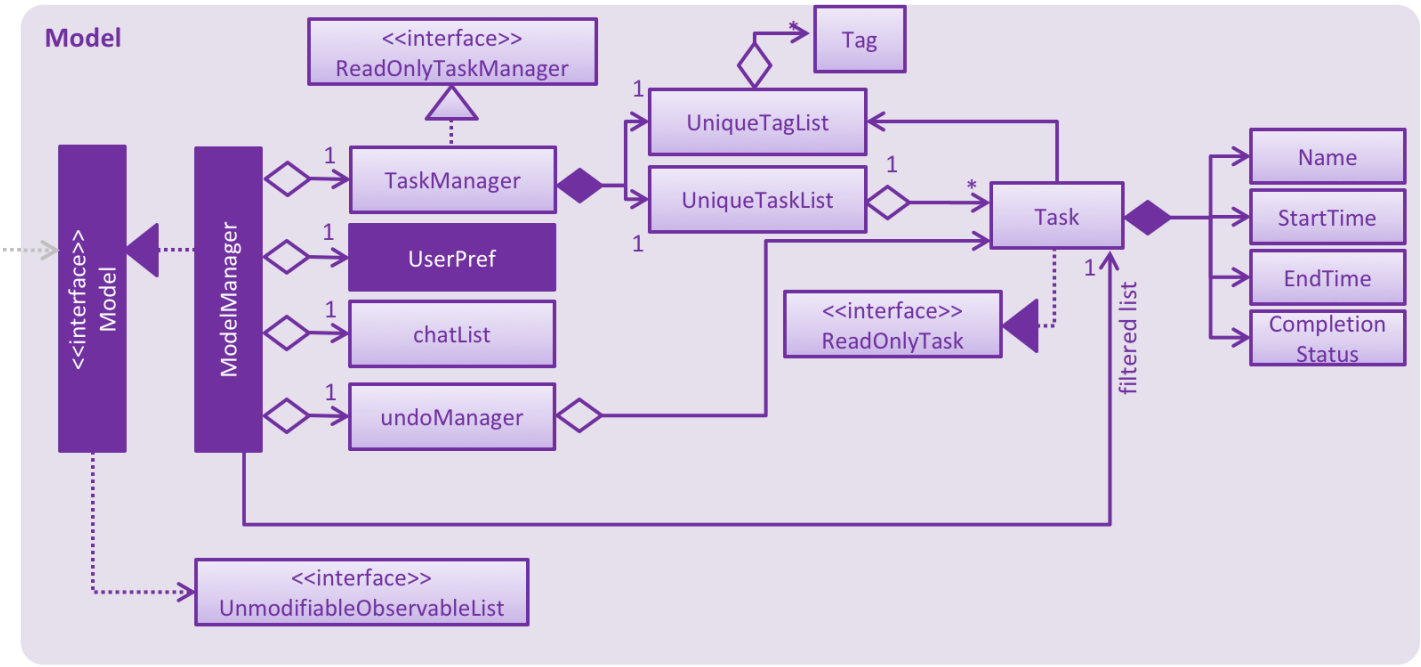
3.3.1 Natty Date Parser

Suru uses Natty for parsing natural language DateTime input. For instructions on how to set up Natty, follow the instructions here.

Suru has implemented customizations that wrap around Natty. These methods can be found in `NattyDateUtil`.

3.4. Model component

Author: Tian Song



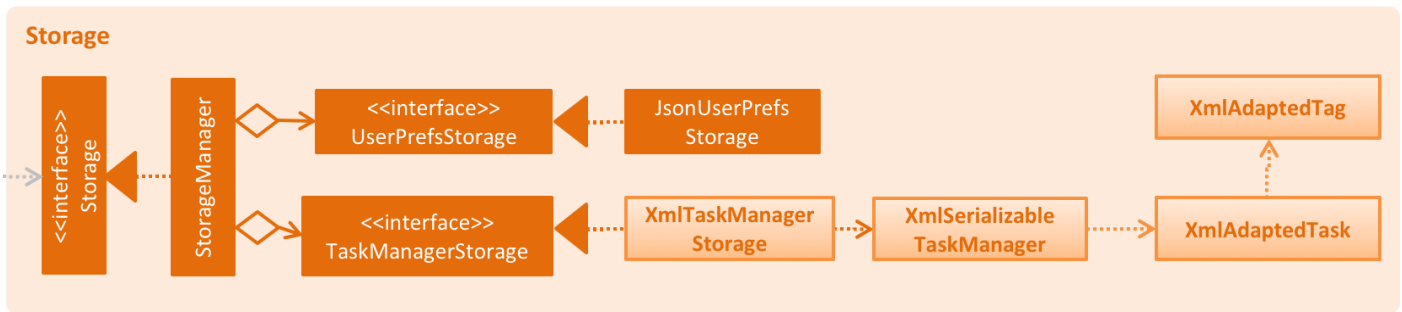
API : `Model.java`

The `Model` consists of several classes that contain information relevant to the data and data structures each object contains. The `Model`

- stores a `UserPref` object that represents the user's preferences.
- stores the Suru Task Manager data.
- exposes a `UnmodifiableObservableList<ReadOnlyTask>` that can be 'observed' e.g. the UI can be bound to this list so that the UI automatically updates when the data in the list change.
- does not depend on any of the other three components.

3.5. Storage component

Author: Mustaqiim



API : `Storage.java`

The `Storage` component contain classes that save and load data in json and xml formats. The `Storage` should not directly communicate with the `Model` or `UI`. Interactions between these components should interact using `Events`. `Storage`

- can save `UserPref` objects in json format and read it back.
- can save the Suru Task Manager data in xml format and read it back.

3.6. Common classes

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

4. Configuration

4.1. Logging

The `java.util.logging` package is used for logging. The `LogsCenter` class is used to manage the logging levels and logging destinations.

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

Logging Levels

- `SEVERE` : Critical problem detected which may possibly cause the termination of the application.
- `WARNING` : Can continue, but with caution.
- `INFO` : Information showing the noteworthy actions by the App.
- `FINE` : Details that are not usually noteworthy but may be useful in debugging e.g. print the actual list instead of just its size.

4.2. Configuration file

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

5. Testing

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

In Eclipse:

- To run all tests, right-click on the `src/test/java` folder and choose `Run as` > `JUnit Test`.

- 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.

Using Gradle:

- See UsingGradle.md 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:
    1. *Unit tests* targeting the lowest level methods/classes. e.g. `seedu.task.commons.UrlUtilTest`
    2. *Integration tests* that are checking the integration of multiple code units (those code units are assumed to be working). e.g. `seedu.task.storage.StorageManagerTest`
    3. Hybrids of unit and integration tests. These test are checking multiple code units as well as how the are connected together. e.g. `seedu.task.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.

5.1. Troubleshooting tests

Problem: Tests fail because of NullPointerException when AssertionError is expected

- Reason: Assertions are not enabled for JUnit tests. This can happen if you are not using a recent Eclipse version (i.e. *Neon* or later).
- Solution: Enable assertions in JUnit tests as described here.
  Delete run configurations created from earlier tests.

6. Dev Ops

6.1. Build Automation

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

6.2. Continuous Integration

We use Travis CI and AppVeyor to perform *Continuous Integration* on our projects. See UsingTravis.md and UsingAppVeyor.md for more details.

6.3. Publishing Documentation

See UsingGithubPages.md to learn how to use GitHub Pages to publish documentation to the project site.

6.4. Making a Release

Here are the steps to create a new release.

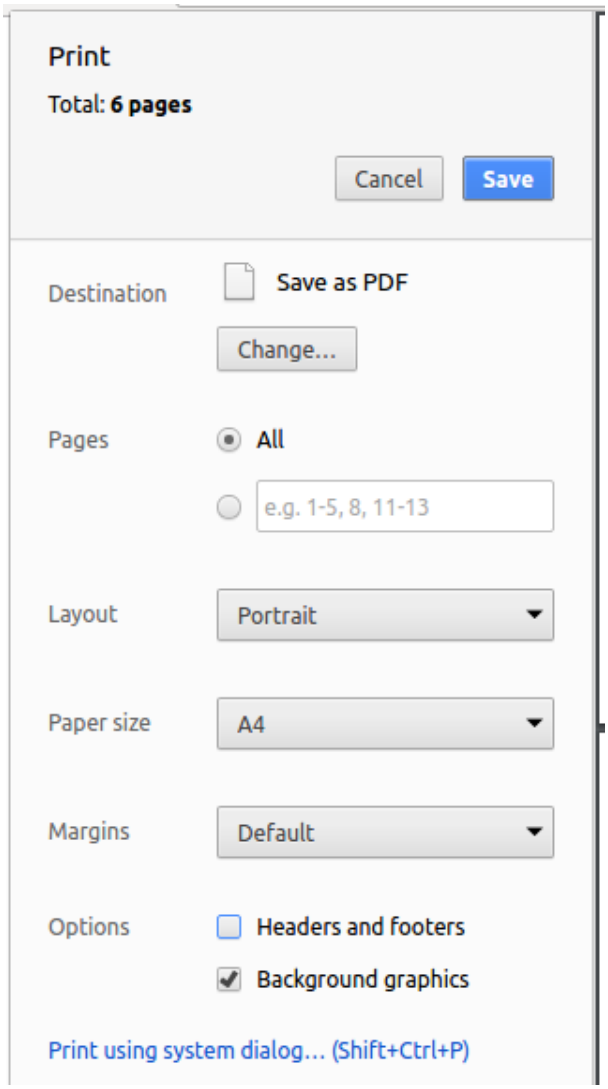
1. Generate a JAR file using Gradle.
2. Tag the repo with the version number. e.g. `v0.1`
3. Create a new release using GitHub and upload the JAR file you created.

6.5. Converting Documentation to PDF format

We use Google Chrome for converting documentation to PDF format, as Chrome's PDF engine preserves hyperlinks used in webpages.

Here are the steps to convert the project documentation files to PDF format.

1. Make sure you have set up GitHub Pages as described in UsingGithubPages.md.
2. Using Chrome, go to the GitHub Pages version of the documentation file. e.g. For UserGuide.md, the URL will be `https://github.com/CS2103JAN2017-W09-B3/main/docs/UserGuide.html`.
3. Click on the `Print` option in Chrome's menu.
4. Set the destination to `Save as PDF`, then click `Save` to save a copy of the file in PDF format. For best results, use the settings indicated in the screenshot below.



6.6. Managing Dependencies

A project often depends on third-party libraries. For example, Suru 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 these alternatives.
a. Include those libraries in the repo (this bloats the repo size).
b. Require developers to download those libraries manually (this creates extra work for developers).

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	refer to instructions when I forget how to use the application

Priority	As a …	I want to …	So that I can…
`* * *`	new user	can view more information about a particular command	learn how to use various commands
`* * *`	user	add task by specifying task description only	record tasks that need to be done ‘someday’
`* * *`	user	add task with due date	record task that has to be done by given date
`* * *`	user	add event with specific duration	record event that takes place within a given timeframe
`* * *`	user	add follow-up tasks using start time	record tasks that has to be done after a certain date
`* * *`	user	delete task	get rid of task that I no longer care to track
`* * *`	user	edit task	edit task name, description
`* * *`	user	view all tasks	
`* * *`	user	view incomplete tasks	
`* * *`	user	view completed tasks	
`* * *`	user	undo previous action	revert previous action in case of mistakes
`* * *`	user	redo	reverse the effects of the last undo
`* * *`	user	search incomplete tasks by date or date range	check tasks due on a certain day or within a time frame
`* * *`	user	search for tasks	find a specific task
`* * *`	user	check off task	track what tasks are done
`* * *`	user	uncheck a task	correct accidental check off of a task
`* * *`	advanced user	Define save and load directory	specify which directory the program read and write from

Priority	As a …	I want to …	So that I can…
`* *`	user	add tags to task	categorize tasks by tags
`* *`	user	remove tags from task	
`* *`	user	receive email reminders	
`*`	advanced user	use shorter versions of a command	type command faster

Appendix B : Use Cases (UC)

(For all use cases below, the **System** is `Suru Task Manager` and the **Actor** is the `user`, unless specified otherwise)

`UC01 - Add task`

MSS

1. User requests to add new task specifying task description and no date.
2. Suru adds task to list of tasks.
3. Suru GUI is refreshed to show the updated list of tasks. Use case ends.

Extensions

1a. User requests to add new task specifying tasks and due date/time.

> 1a1. Suru adds task to list of task with due date/time. Use case resumes at step 3.

1b. User requests to add new task specifying tasks and time frame for task.

> 1b1. Suru adds task to list of task with time frame. Use case resumes at step 3.

1c. User requests to add new task specifying tasks and start date/time.

> 1c1. Suru adds task to list of task with start date/time. Use case resumes at step 3.

1d. User requests to add task without task description or using invalid syntax.

> 1d1. Suru displays error message. Use case ends.

`UC02 - View tasks`

MSS

1. User requests to shows all tasks.
2. Suru displays all tasks. Use case ends.

Extensions

1a. User requests to show only incomplete tasks.

> Use case resumes at step 2.

1b. User requests to show only complete tasks.

> Use case resumes at step 2.

2a. The requested list is empty.

> 2a1. Suru displays error message. Use case ends.

`UC03 - Search tasks`

MSS

1. User requests to find tasks with specific keyword(s).
2. Suru displays all tasks containing specified keyword(s). Use case ends.

Extensions

2a. The requested list is empty.

> 2a1. Suru displays error message. Use case ends.

`UC04 - Edit tasks`

MSS

1. User **view all tasks (*UC02*).**
2. User requests to update a task by index with changed details.
3. Suru requests user for confirmation.
4. User confirms changes.
5. Suru GUI is refreshed to show updated list of tasks. Use case ends.

Extensions

2a. The given index is invalid.

> 2a1. Suru displays error message. Use case ends.

2b. The new details are given using invalid syntax.

> 2b1. Suru displays error message. Use case ends.

4a. User rejects changes.

> Use case ends.

`UC05 - Delete task`

MSS

1. User **view all tasks (*UC02*).**
2. User requests to delete a task by index.
3. Suru requests user for confirmation.
4. User confirms changes.
5. Suru GUI is refreshed to show updated list of tasks. Use case ends.

Extensions

2a. The given index is invalid.

> 2a1. Suru displays error message. Use case ends.

4a. User rejects changes.

> Use case ends.

`UC06 - Undo previous command`

MSS

1. User requests to undo command.
2. Suru displays list of tasks according to before previous action. Use case ends.

Extensions

2a. Undo is the first command entered by User during current start up.

> 2a1. Suru displays error message. Use case ends.

`UC07 - Redo previous 'undo'`

MSS

1. User requests to redo previous ‘undo’.
2. Suru displays list of tasks as according to before previous undo. Use case ends.

Extensions

2a. Undo is the first command entered by User during current start up.

> 2a1. Suru displays error message. Use case ends.

`UC08 - Search for incomplete task by date or date range`

MSS

1. User requests to search for task with either date or date range.
2. Suru displays list of tasks that falls on specified date or within date range. Use case ends.

Extensions

2a. The requested list is empty.

> 2a1. Suru displays error message. Use case ends.

2b. The date or date range is in invalid format. (e.g end date earlier than start date)

> 2b1. Suru displays error message. Use case ends.

`UC09 - Check off task`

MSS

1. User **view all tasks (*UC02*).**
2. User requests to check off a task by index.
3. Suru requests user for confirmation.
4. User confirms changes.
5. Suru GUI is refreshed to show updated list of tasks. Use case ends.

Extensions

2a. The given index is invalid.

> 2a1. Suru displays error message. Use case ends.

2b. The requested task to check off is already checked.

> 2b1. Suru displays error message. Use case ends.

4a. User rejects changes.

> Use case ends.

`UC10 - Uncheck a task`

MSS

1. User **view all tasks (*UC02*).**
2. User requests to uncheck a task by index.
3. Suru requests user for confirmation.
4. User confirms changes.
5. Suru GUI is refreshed to show updated list of tasks. Use case ends.

Extensions

2a. The given index is invalid.

> 2a1. Suru displays error message. Use case ends.

2b. The requested task to uncheck is not checked.

> 2b1. Suru displays error message. Use case ends.

4a. User rejects changes.

> Use case ends.

`UC11 - Add tags to a task`

MSS

1. User **view all tasks (*UC02*).**
2. User requests to add a tag or a few tags to a task by index.
3. Suru requests user for confirmation.
4. User confirms changes.
5. Suru GUI is refreshed to show updated list of tasks. Use case ends.

Extensions

2a. The given index is invalid.

> 2a1. Suru displays error message. Use case ends.

2b. The requested task to add tags already exists.

> 2b1. Suru displays error message. Use case ends.

4a. User rejects changes.

> Use case ends.

`UC12 - Delete tags in a task`

MSS

1. User **view all tasks (*UC02*).**
2. User requests to delete tags for a task by index.
3. Suru requests user for confirmation.
4. User confirms changes.
5. Suru GUI is refreshed to show updated tasks witout tags. Use case ends.

Extensions

2a. The given index is invalid.

> 2a1. Suru displays error message. Use case ends.

2b. The requested tags to be deleted from the task does not exists.

> 2b1. Suru displays error message. Use case ends.

4a. User rejects changes.

> Use case ends.

`UC13 - Filter tasks by tags`

MSS

1. User **view all tasks (*UC02*).**
2. User requests to filter tasks by tags.
3. Suru requests user for name of tags to filter the task by.
4. Suru GUI is refreshed to show updated list of tasks by tags.

Use case ends.

Extensions

2a. The given tags does not exists.

> 2a1. Suru displays error message. Use case ends.

`UC14 - Define save and load database`

MSS

1. Suru scans default directory for previous save database during start-up.
2. Save database is loaded into memory from default file location.
3. Suru runs as per normal.

Use case ends.

Extensions

2a. File not found in default directory.

> 2a1. Suru request user to locate database file or create new database file.
> Use case resumes at step 2.

`UC15 - Email Reminders`

MSS

1. Suru sends list of tasks to server
2. Server sends email to user 1 hour before task is due.

Use case ends.

`UC16 - Command shortcut`

MSS

1. Suru detects specific command shortcut during runtime.
2. Suru maps to main command.

Use case ends.

Extensions

1a. Command entered not found or invalid.

> 1a1. Suru displays error message. Use case ends.

Appendix C : Non Functional Requirements

1. Should work on any mainstream OS as long as it has Java `1.8.0_60` or higher installed.
2. Should be able to hold up to 1000 tasks without a noticeable sluggishness in performance for typical usage.
3. A user with above average typing speed for regular English text (i.e. not code, not system admin commands) should be able to accomplish most of the tasks faster using commands than using the mouse.
4. Should be able to have data file synced using cloud syncing services (e.g dropbox) if save/load directory is in the appropriate location.

Appendix D : Glossary

Mainstream OS

> - Windows 7 and above
> - Mac OSX
> - Ubuntu

Appendix E : Product Survey

IKE Author: Shawn Lin Jingjue

Pros:

- Prioritizes task by urgency and importance
- Due dates
- Checklists

Cons:

- No syncing across devices
- Only Android platform

Wunderlist

Author: Muhammad Mustaqiim Bin Muhar

Pros:

- Able to share group tasks easily with teammates.
- Cross platform support on most devices and Operating Systems.
- Able to auto add reminders and due dates using rss link to a calendar of your choice.

Cons:

- No import function of existing task from other apps to Wunderlist.
- Assigning of to-dos to your team members is a paid feature (Organisation Features).

Habitica

Author: Jeremy Heng Wen Ming

Pros:

- Gamification paradigm extends to the social features of the application in the form of team quests. This is essentially a mechanism to share group tasks and to mantain accountability.
- A loot and shop system allows users to obtain and purchase cosmetic items to signify character progress.
- Splits tasks into dailies, habits and to-dos.
- Tasks can be split into subtasks which can be assigned individual reward values.

Cons:

- The interface is not efficient to use when adding many tasks.
- There is no integration with any calendar applications.

Trello

Author: Teo Tian Song

Pros:

- Has due date capability.
- Keeps track of completed task.
- Has checklist to break down tasks into components.

Cons:

- Requires internet connection to utilize.
- Requires consistent usage of mouse to perform commands.
