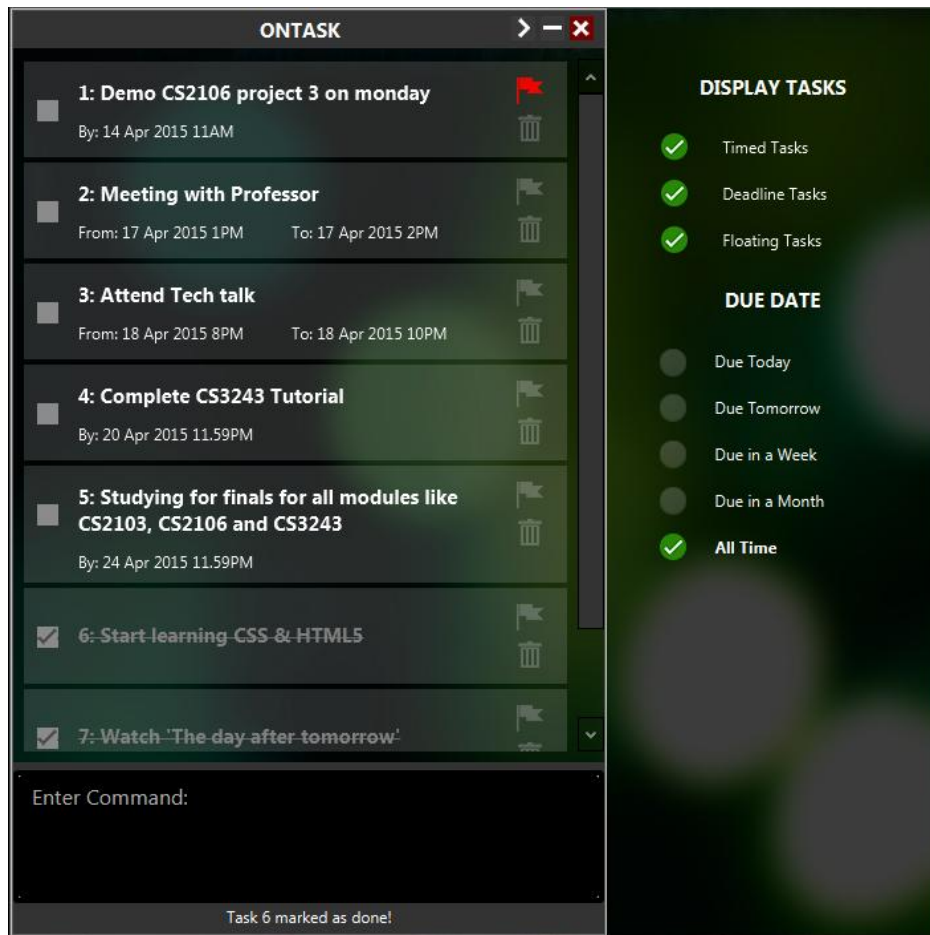





ONTASK v0.5



Supervisor: Ryan

Extra feature: GUI

 <p>Daren Tan</p> <p>Team lead, Documentation, Deliverables & Deadlines</p>	 <p>Parag Bhatnagar</p> <p>Software Expert, Integration, Code Quality</p>	 <p>Wang Minwei</p> <p>Testing, Documentation, Scheduling & Tracking,</p>
--	--	--

User Guide

Welcome to ONTASK!

1. Introducing our Software

1.1. Now that you've started using our software, here is some information on beginning to use our product!

Similar to all other GTD softwares, our product will provide the fundamental needs for tracking tasks and events. Users will be able to rely on this software to increase their productivity, manage emails, tasks, projects all in one place.

The program has a simple, minimalist GUI with task adding, editing, deleting, undo/ redo functions, search and sort capabilities.

2. Getting Started

2.1. No installation is required before using our product. Just run the program to get started!

2.2. The program uses an Ontask.xml file for persistent storage and requires user to specify its directory at startup. The file contains important attributes like task description, task type, start date-time, end date-time, prioritizing, completion and index. If there is no file present, the program creates a new empty file at the specified directory.

3. Features & Functions

Accepted Time Formats:

Format	Examples
24-hour clock	09:00, 13:00, 23:59, 01:25
12-hour clock	9am, 1pm, 11.59pm, 1.25am

Accepted Date Formats:

Format	Examples
Day Month Year	3 1 2015, 3-1-2015, 3/1/2015, 3 Jan 2015
Day Month	3 1, 3-1, 3/1, 3 jan
Month Day	Jan 3, january 3

Day of week	Monday, mon, thursday, thu
Special word	today, tdy, tomorrow, tmr

3.1. For new users who require **help** to display a list of commands when the program is initialized.

Command format:

help or <Press F1 key>

3.2. Add a **floating task** by specifying description only (no date and time required).

Please use quotation marks in order to input date or time formats into task description.

e.g. add *watch day after "tomorrow" with family*

Command format:

add *taskDescription*

create *taskDescription*

3.3. Add a **time-tasked event** that requires starting date, start-time, end date and end-time

Command format:

add *taskDescription <date,time>*

e.g. add Chalet with friends from 30 Apr 1pm to 1 May 8pm

e.g. add Chalet with friends 30/4 13:00 1/5 20:00

e.g. add Meet professor 30 Apr 2015 3pm 5pm

e.g. add Meet professor 30 Apr 2015 3pm - 5.25pm

3.4. Add a **due task** that has a due date and time.

Command format:

add *taskDescription* by *endDate endTime*

add *taskDescription endDate endTime*

3.5. To **delete** a completed task, refer to index number from Task list.

Command format:

delete *index*

3.6. To **undo** or **redo** recent actions. Number of undo operations is up to number of operations after starting the application. Undo for search, back is not supported. Number of redo operations is up to number of undo operations. User cannot redo after executing other commands than undo and redo.

Command format:

undo or <Press CTRL+Z key>

Redo or <Press CTRL+Y key>

3.7. To **edit** existing task, refer to index number from Task list. Edit is possible on any parameters of an existing task, provided that the parameter was passed to the task upon initialization (a due task will not have a starttime or startdate parameter, a floating task will only have the task description parameter).

User cannot change task type and is forbidden to change areas which are unspecified by its task type. Illegal operations such as making start time later than end time are disallowed.

The following parameters can be edited:

task (or) desc, startdate, starttime, enddate, and endtime

Command format:

edit *parameter index newValue*

e.g. edit endtime 2 12:00

edit task 3 this will be my new task description

3.8. To **search** for existing task, please specify a parameter to search. The search function is non case-sensitive. Once the search results are displayed, user can operate on sorted list. Press BACK_SPACE Key to go back to the main list.

Command format:

search desc *keyword*

search task *keyword*

search type floating / deadline / due / timedtask / event

search before *date*

search date *date*

3.9. To **return** to the task list (after search), just press enter.

Command format:

<Press Enter Key>

3.10. To **mark** a task as completed or **unmark** a task.

Command format:

mark *index*

done *index*

unmark *index*

undone *index*

togglemark *index*

3.11. To **flag** a task to take priority or **unflag** a task.

Command format:

flag *index*

prioritise *index*

unflag *index*

unprioritise *index*

toggleflat *index*

3.12. To **change the directory** where the Ontask.xml is stored.

Command format:

changedir or <Press F1 key>

3.13. To **clear** the task lists.

Command format:

clear

3.13. To **exit** the Ontask application.

Command format:

exit

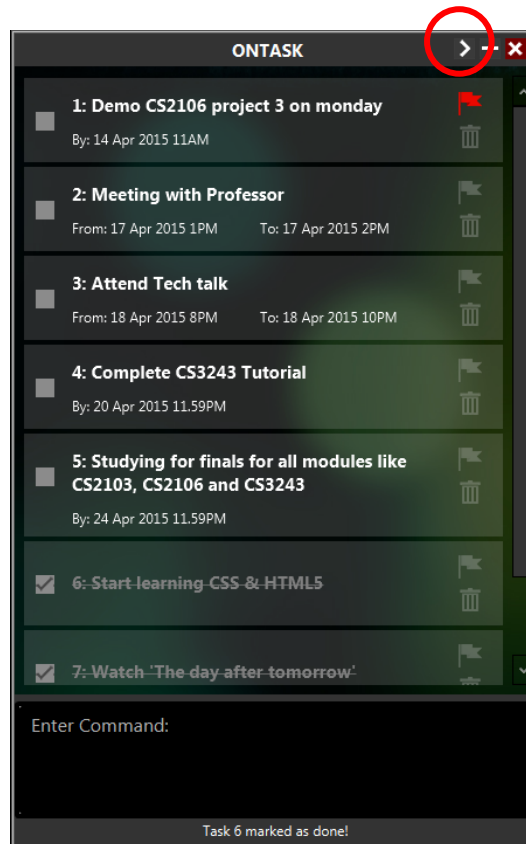
quit

3.14. To **autocomplete** on or off. The auto-complete function will be activated by the <Spacebar> key when a suitable command is recognized.

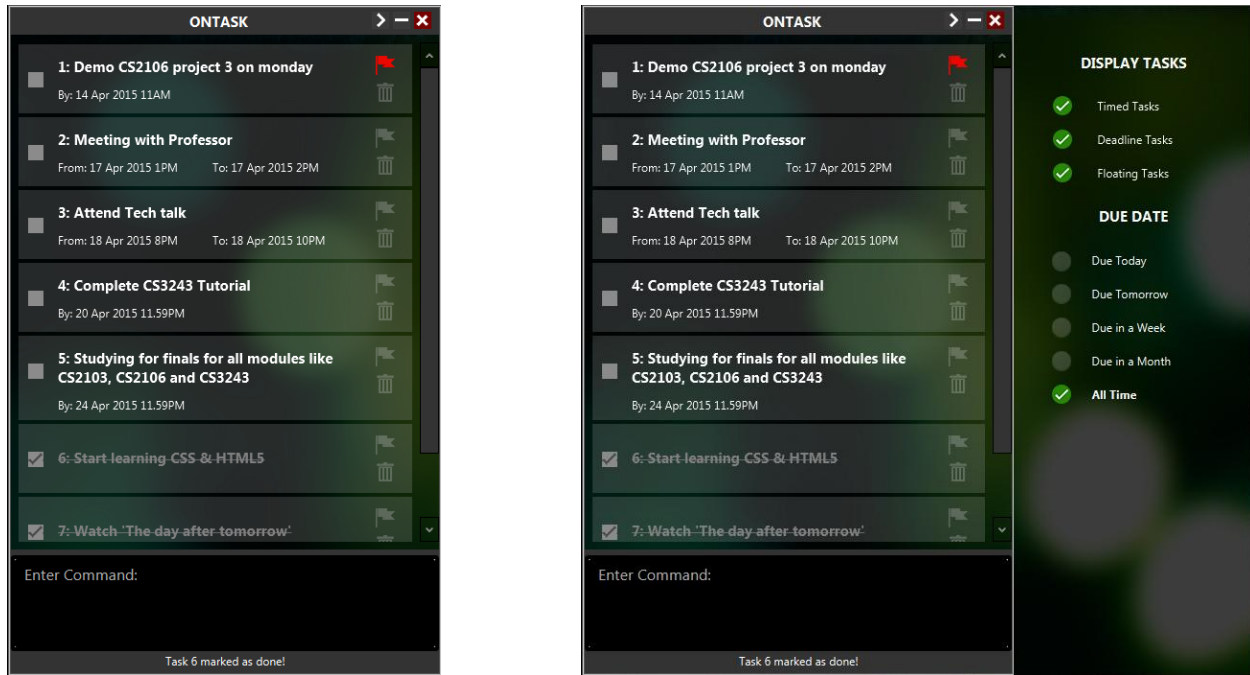
Command format:

<Press LEFT-ALT key>

3.15. To enable or disable the **Graphical User Interface Sidebar**. Click on the > Button.



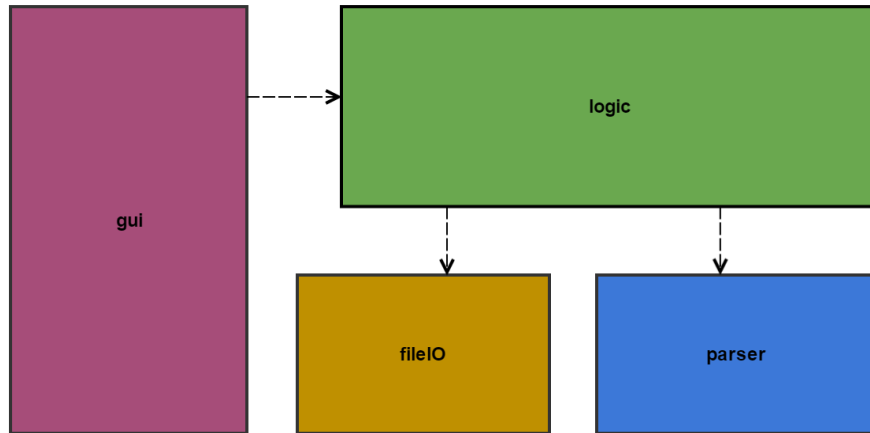
Developer Guide



4. Architecture & Components

Project Ontask relies on the top-down approach by first having an overview of the project requirements to formulate a system. It is then sub-divided into four main components: fileIO, logic, parser, gui and refined in detail base on each component. Figure below illustrates the basic architecture of how the program interacts on a component level.

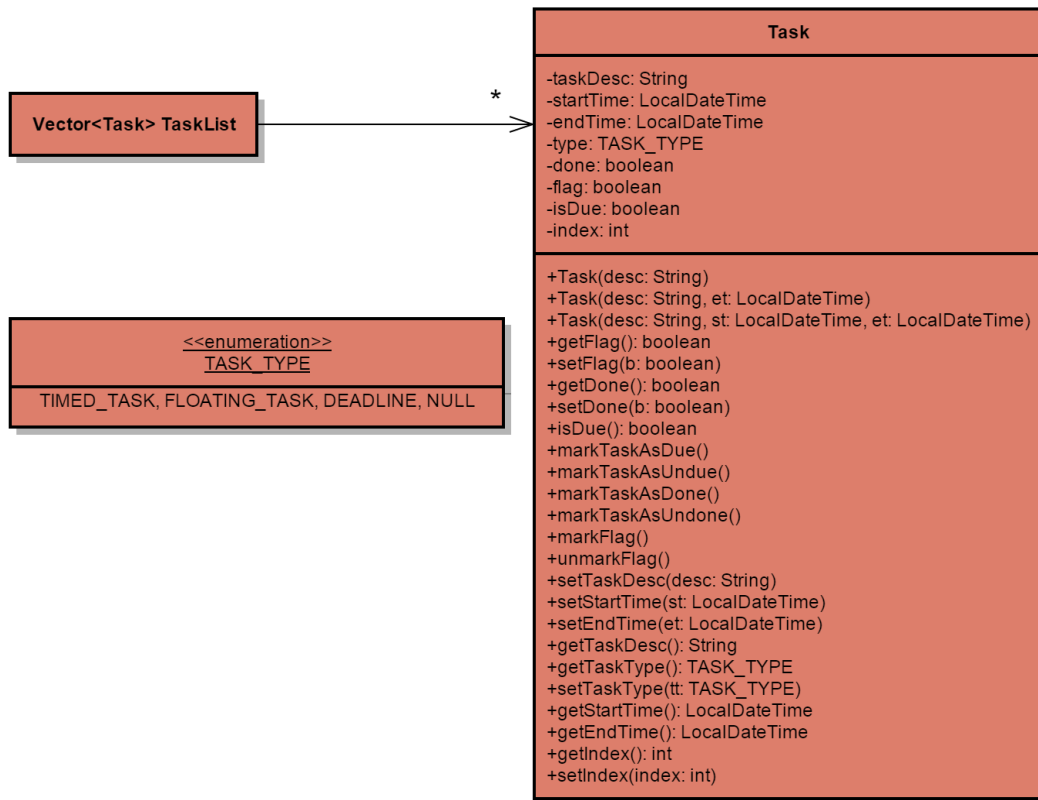
- The *gui* component is in control of the display and updating of Tasks to the graphical user interface, along with all the visual feedback and button controls.
- The *fileIO* component will be handling the file inputs and outputs to enable reading and writing into XML for persistent storage and keeping track of the directory.
- The *parser* component will parse the string into acceptable object-oriented formats for logic to process.
- The *logic* component will then execute the commands accordingly and return the Task List for the GUI to re-display.



Basic architecture diagram for Project Ontask

5. Introducing the Task object

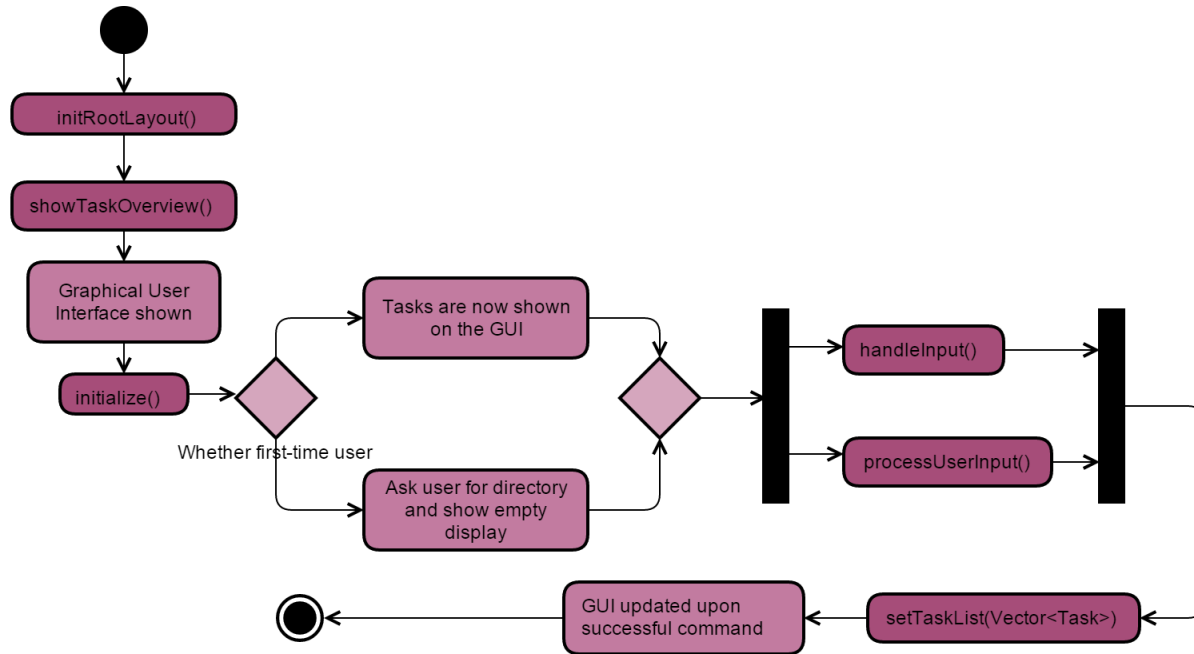
The program uses a vector of task objects of the following type with methods to get and set each attribute of the Task object.



Task Class Diagram

6. Graphical User Interface

The program initializes by first running the main function of the *ListViewGUI* class which launches the GUI, which is made using JavaFX (<http://www.edipse.org/efxdipse/index.html>). GUI relies on the *TaskDisplayPage.fxml* for the layout of the application and *Style.css* for the styling of the interface.



GUI Activity Diagram

The GUI is restricted to only 1 instance which conforms to the singleton pattern. The *ListViewGUI* calls the *FileStream* class which reads the *Ontask.xml* file and uses the *TaskListWrapper* class to create the vector of Task. The vector of Tasks is then wrapped into an observable list in *TaskDisplayController* which displays updates each Task in the display.

Each *TaskCell* object is a *hBox* (Horizontal Box) that consists of the following:

- Checkbox to mark/unmark a task as done
- *vBox* (Vertical Box) that contains 2 strings, task description and the task date inputs.
- *vBox* that contains a *flag* button to prioritize tasks and a *delete* button to delete a task.

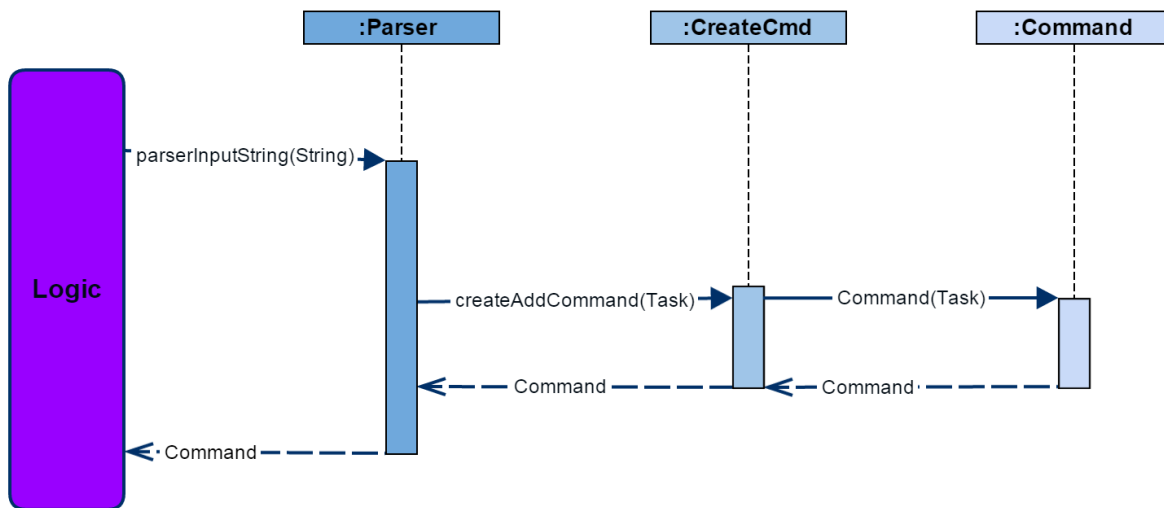
The GUI has a text area for the user to type in the string input, which is consumed upon pressing the Enter key. The string file is then sent to the *parser* class for parsing.

6.1 GUI SideBar

As an additional feature to our GUI, there is a sidebar that can be toggled to show or hide at the user's will. It consists of toggle and radio buttons that allows efficient sorting of the Tasks in the main display list.

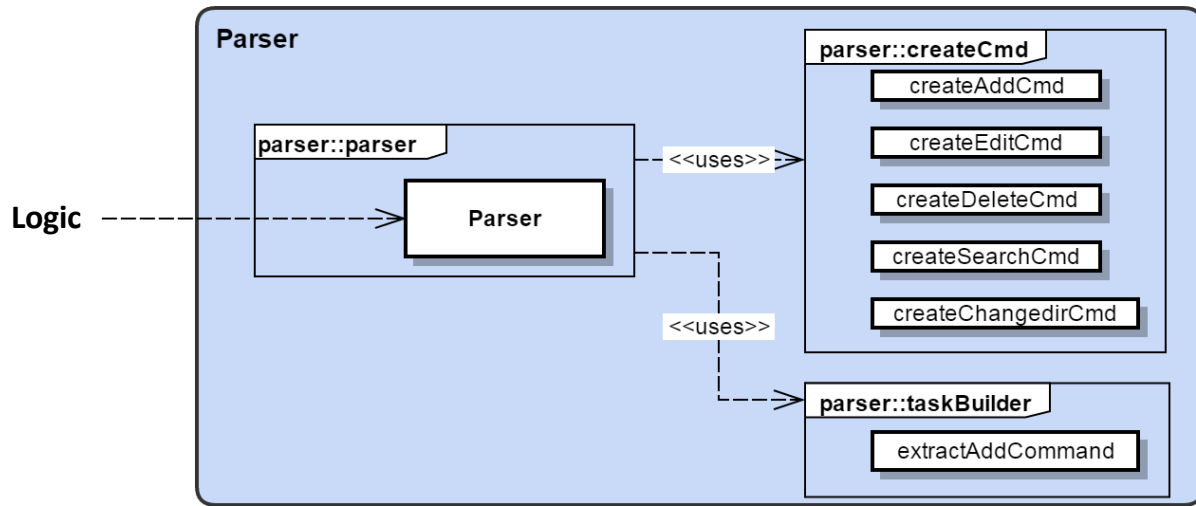
7. Parsing the string

Parser is an important component which takes in user input from logic and extracts meaningful information out of the string. It decides on the command type and call respective methods to perform parsing. After parsing, *Parser* stores extracted information in *Command* object and returns this to *Logic* class.



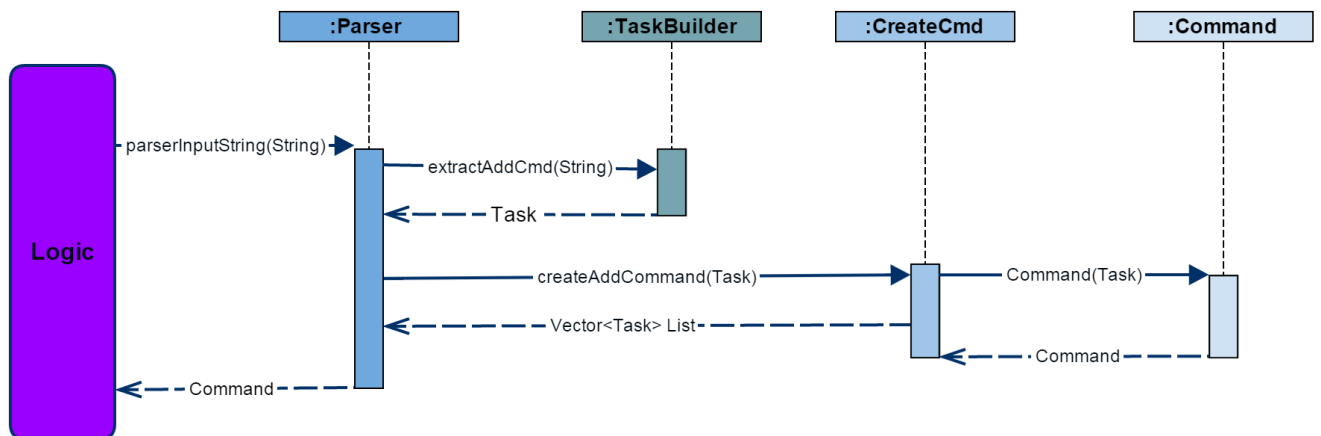
General workflow of parsing user input string

1. *Parser* takes in the string file and parses it.
2. *Parser* calls *CreateCmd* class with the information it extracts (Task, index, etc) to create *Command* object.
3. *Parser* return *Command* object to logic.



Parser package component

Parser class makes use of Command pattern, because it creates a *Command* Object for different types of operations. Thus *Logic* can treat them equally as *Command* Object and extract different information according to *Command* description. *Parser* class can be enriched with more time and date formats by defining new time/date String regex.

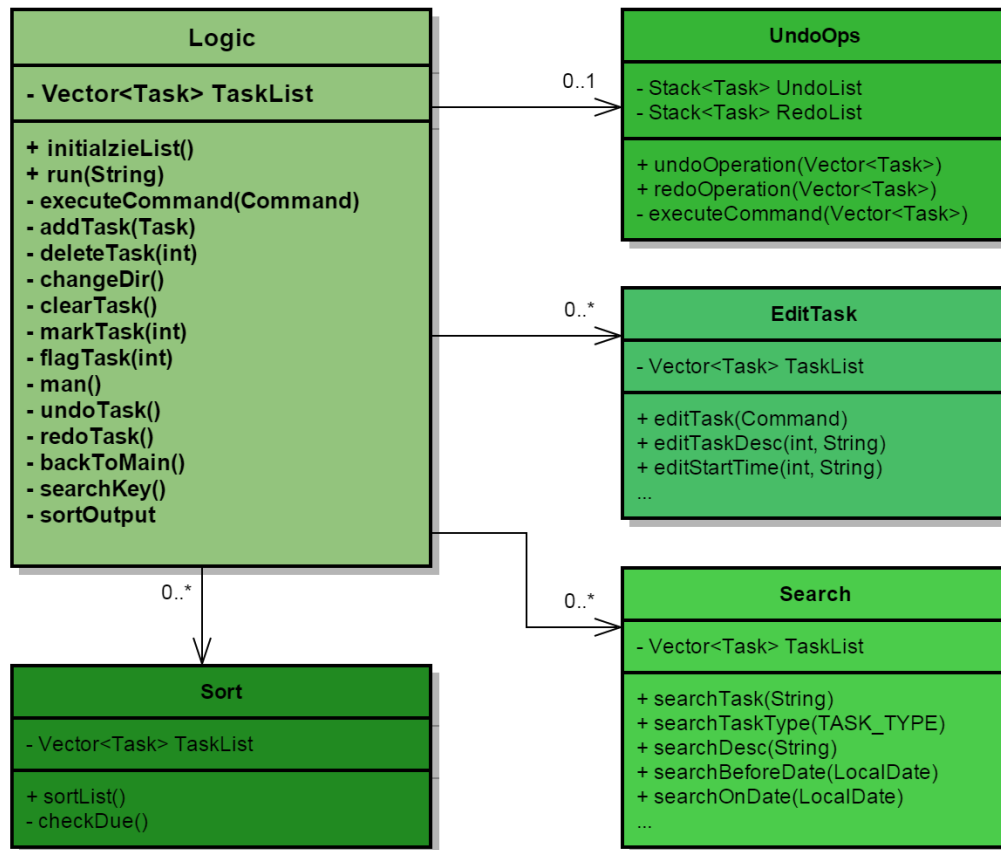


Parsing process for add command

Above is a graph describing the add command parsing process. It is more complex than normal parsing because our *parser* is able to take in more flexible formats. *TaskBuilder* class looks for date and time String regex in the String, and adjusts according to time indication words (by, at, from, to).

8. The Logic

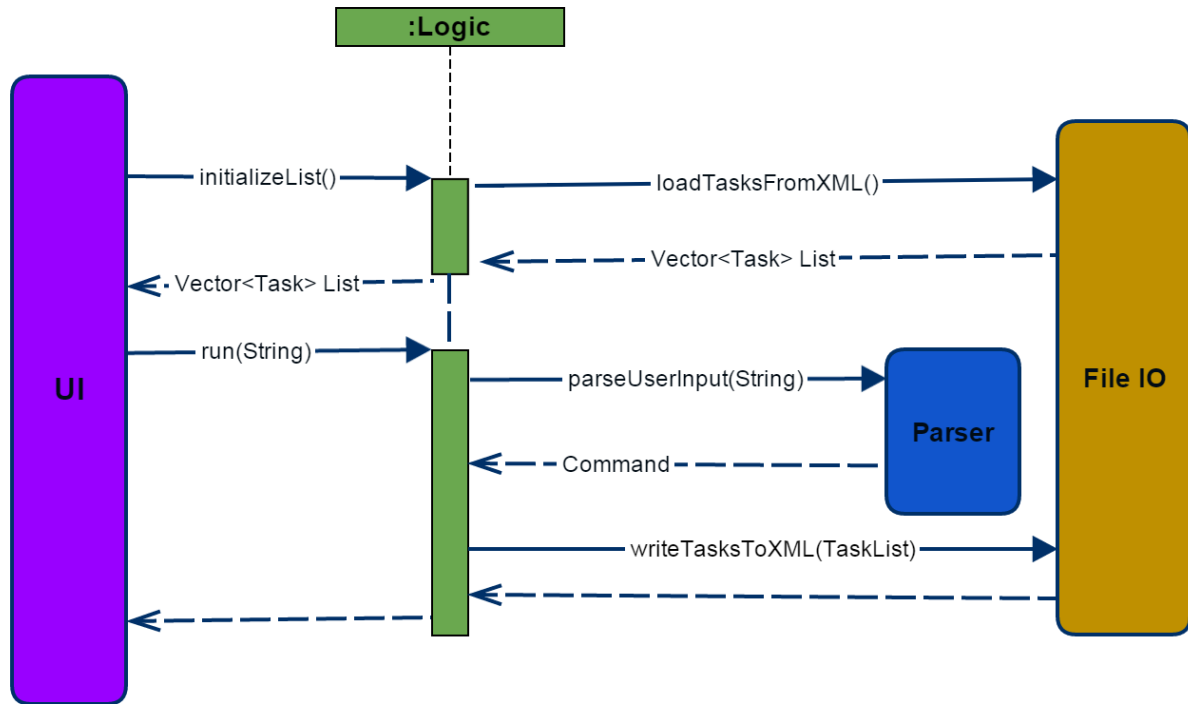
The *Logic* class serves as an intermediary that sits between GUI and other components of the program. *Logic* takes in user input as String, passes the String to *Parser*, updates TaskList accordingly, and, if the operation is successful, it calls upon *fileIO* to store the latest version of TaskList. *Logic* also serves the purpose of reading from *fileIO* and initializes list. *Logic* always returns a Vector of Tasks to the *gui* component for display purposes.



Logic Class Diagram

The workflow of *Logic* is shown in the sequence diagram and does the following procedure:

1. When the program first starts, *UI* component calls *Logic* to initialize list, *Logic* calls upon *fileIO* and returns a Vector of Tasks.
2. In the following commands, *UI* calls *run* method in *Logic* to process user input.
3. *Logic* passes the String to *Parser* and operates on the *Command* Object returned by *Parser*.
4. After performing the operations, *Logic* sorts the TaskList and updates backup list by calling *fileIO*.

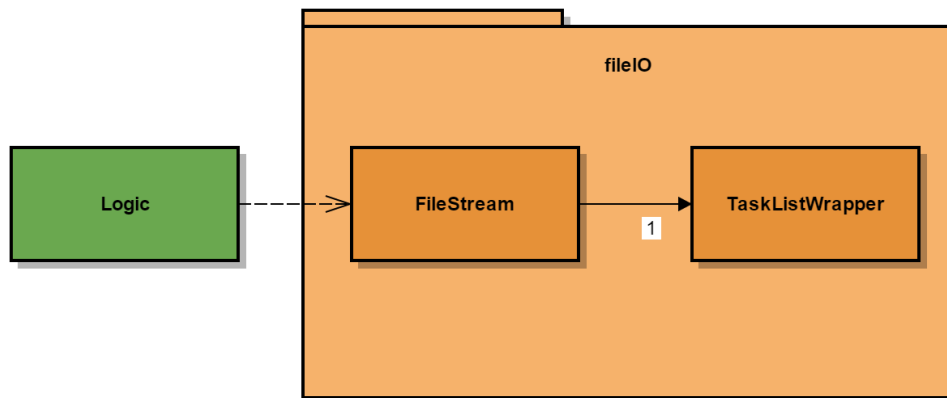


Logic Sequence Diagram

9. Persistent Storage: fileIO

The *fileIO* component is responsible for converting the Task List to be stored in the XML and vice versa for persistent storage. It is also responsible for monitoring the directory location which includes initializing and changing of directory path.

There are two classes in *fileIO*, the *Filestream* and *TaskListWrapper*. This component adheres to the single responsibility principle where each of the classes have their single responsibility entirely encapsulated within the class.



The `FileStream` class is in charge of all the methods that is required by logic to keep track of the `Vector<Task> TaskList` and the directory where it is stored which relies on the Preference API. Preference API provides a way for the application to store and retrieve the user preference using the window's registry.

The `TaskListWrapper` class helps to wrap and unwrap the vector of Task objects into XML readable formats which dedares the elements when written into `Ontask.xml`.

10. Quality assurance through testing

The `SystemTest` class in the `util` component uses the JUnit framework and it's addon (JUnit-addons 1.4) to simulate a system test for specified commands defined in the specifications to ensure correctness. It is used to test for corner cases as some commands are more prone to errors. This is conveniently used as a form of regression testing during implementation phase to prevent any accidental changes.

`SystemTest` launches the application and makes use of the `TaskDisplayController` class to process the user input indirectly as a string instead of through the GUI. The drawback of this method is that the GUI buttons are not in the test. Upon completion of all command inputs, we then assert the task list with an expected list to make sure they are conforms with the correct output.

```
public void test() throws Exception {
    TaskDisplayController tdc = new TaskDisplayController();

    ...
    //Testing format for due task without keyword 'by'
    tdc.processUserInput("add due task Fri 5pm");

    //Testing special words: today, tomorrow
    tdc.processUserInput("add timed task today 3.25pm tomorrow 5.00pm");

    //Test final command exit
    tdc.processUserInput("exit");
    ...

    checkCurrentAgainstExpectedXML("Ontask.xml", "Expected.xml");
}

public void checkCurrentAgainstExpectedXML(String currentS, String expectedS) {
    File current = new File(currentS);
    File expected = new File(expectedS);

    FileAssert.assertEquals(expected, current);
}
```

Credits

JUnit & JUnit addon v1.4 (JUnit.org)

Stack Overflow (<http://stackoverflow.com/>)

Scene Builder 2.0 (<http://fxexperience.com/2014/05/announcing-scenebuilder-2-0/>)

JavaFX (<http://www.eclipse.org/efxclipse/index.html>)

Gliffy (<https://www.gliffy.com/>)