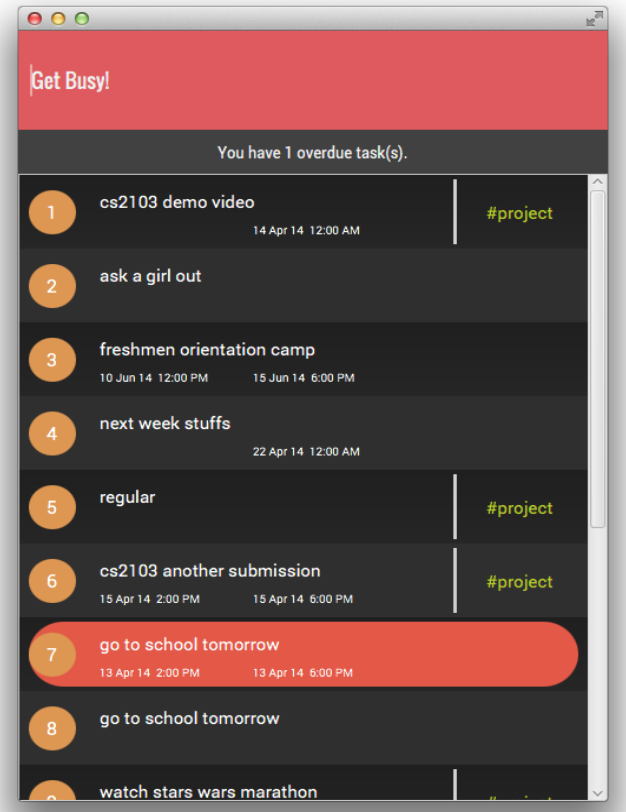
Task Panda Project Manual v0.5



Team members

		
Kaung Htet Team leader, Testing, Code quality	Clement Chong Integration, Documentation,Testing	Tan Zheng Jie (Matthew) Deliverables and deadlines, Scheduling and tracking

Task Panda User Manual

1. Introduction:

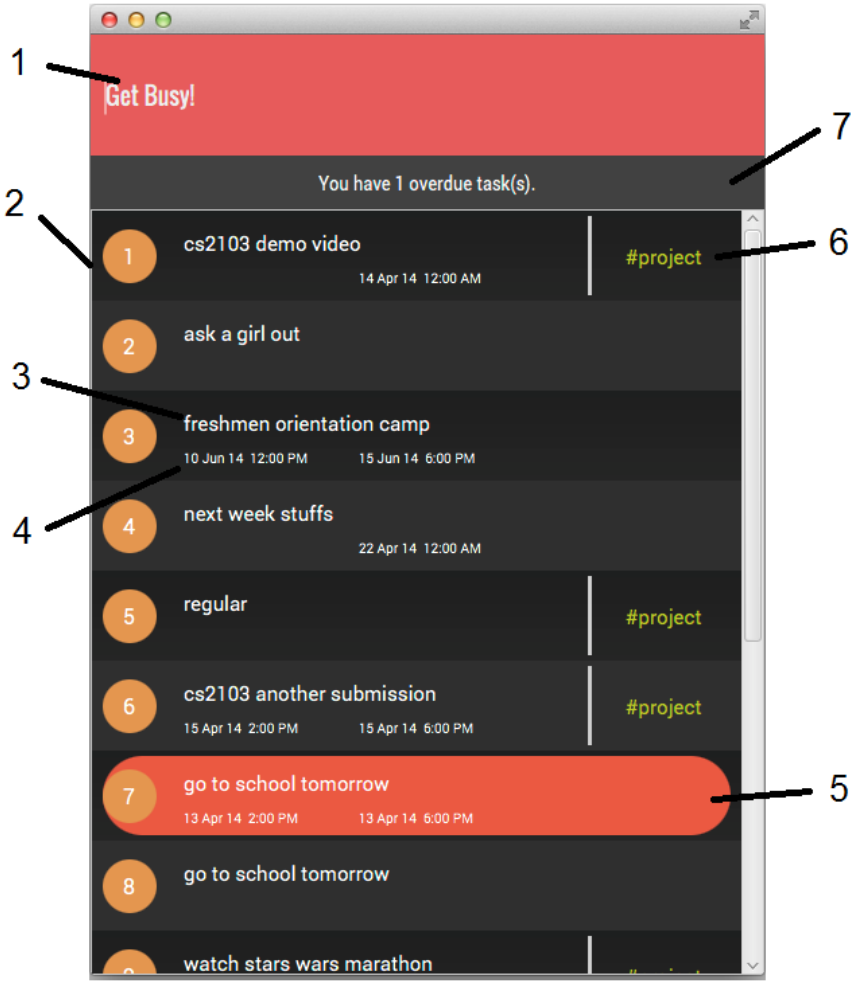
Task Panda is a lightweight to-do list manager that manages your tasks, deadlines and schedules. It focuses solely on keyboard inputs and thus, caters to users who find the need to switch between keyboard and mouse cumbersome.

This User Manual contains all essential information for the user to make full use of this system. This manual also includes a description of the system functions and capabilities, contingencies and alternate modes of operation, and step-by-step procedures for system usage.

To get started, simply double click `TaskPanda.jar`.

---

2. User Interface:



UI components:

1. Input text field
2. Task panel
3. Task descriptions
4. Task deadlines
5. Overdue Tasks
6. Task tags
7. Feedback panel

---

3. Specifying keywords

Task Panda relies on the use of keywords to correctly execute the user command. Therefore, a keyword must be specified into the input text field before any other input.

Supported keywords include:

- add
- delete
- edit
- list
- search
- undo
- redo
- done
- undone
- help

---

4. How to add a task:

To add an item, the first word that should be specified in the input text field is the keyword “add”. This will indicate that an item will be added upon entering the input. There are three ways that an item can be added, namely as a floating task, deadline task or timed task.

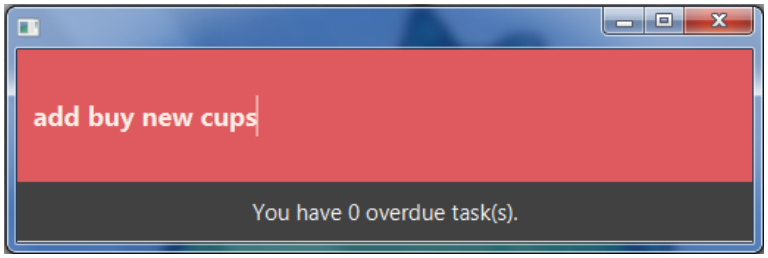
4.1 Adding a miscellaneous task

Miscellaneous tasks are events that do not have any specific completion date and time. You can simply enter your task description after the keyword “add”. All tasks fall under this category if no time or date is stated.

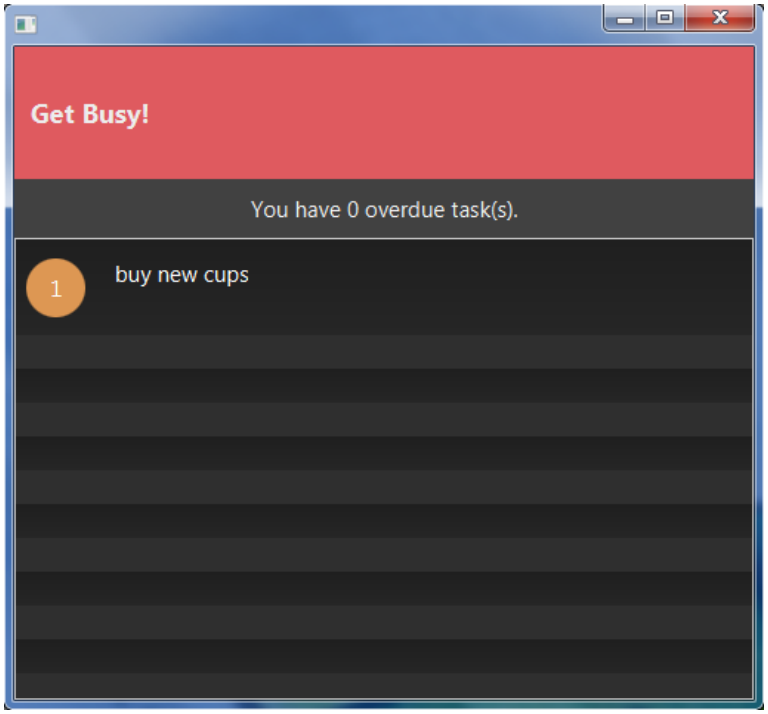
The format supported is:

- add *<task description>*

For example:



Result:



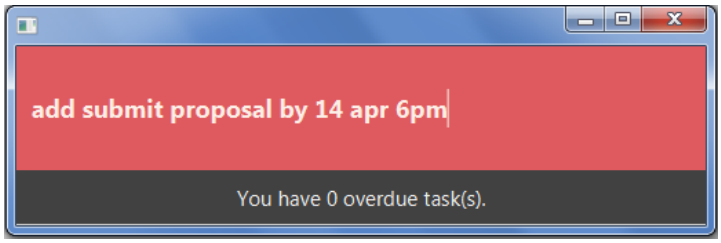
4.2 Adding a deadline task

Deadline tasks are tasks that are required to be completed by a certain time and date. For an input to be considered a deadline task, you must specify a deadline with a date or time in the input.

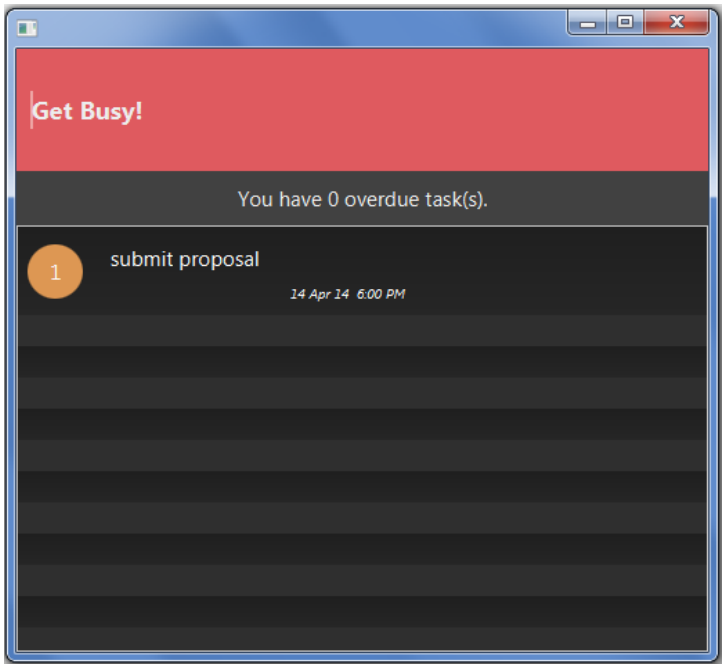
The formats supported are:

- add <*task description*> on <*date*> at/by <*time*>
- add <*task description*> at/by <*time*> on <*date*>
- add <*task description*> on/at/by <date> <*time*> (where time cannot be in the HHMM format)
- add <*task description*> on/at/by <time> <relative date>

For example:



Result:



Note:
The date of a deadline task will be set to the current day by default if it is not specified.
The time of a deadline task will be set to 12am by default if not specified.

For more information on supported time and date formats, refer to *Section 12 - Supported time and date format* of the User Manual.

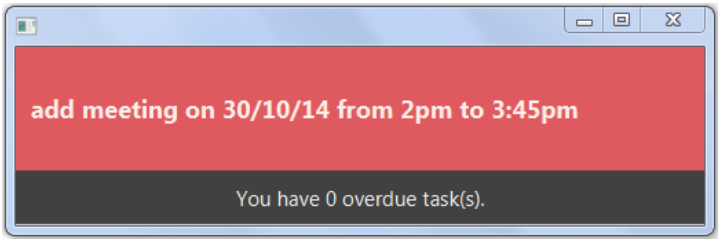
4.3 Adding a timed task

Timed tasks are tasks that falls on a certain date and contains specific start and end time. For an input to be considered a timed task, you must specify a starting and ending time period. The date of a timed task will be set to the current day if it is not specified.

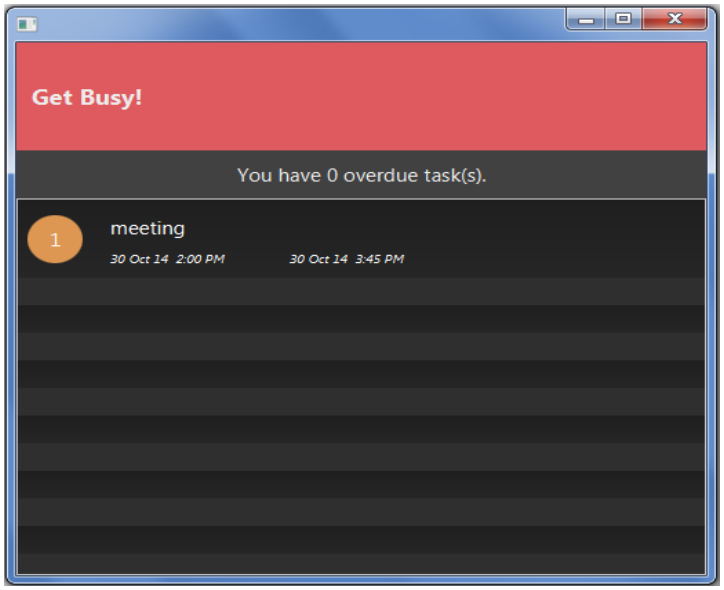
The formats supported are:

- add <*task description*> on <*date*> from <*time*> to <*time*>
- add <*task description*> from <*time*> to <*time*> on <*date*>
- add <*task description*> from <*time*> to <*time*>
- add <*task description*> from <*date*> <*time*> to <*date*> <*time*>

For example:



Result:



Note:
The date of a timed task will be set to the current day by default if it is not specified.
The time of a timed task will be set to 12am by default if not specified.

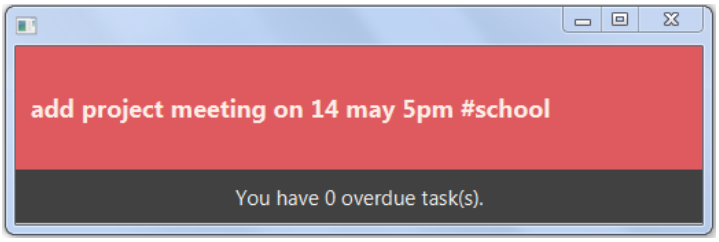
For more information on supported time and date formats, refer to *Section 12 - Supported time and date format* of the User Manual.

4.4 Hashtags

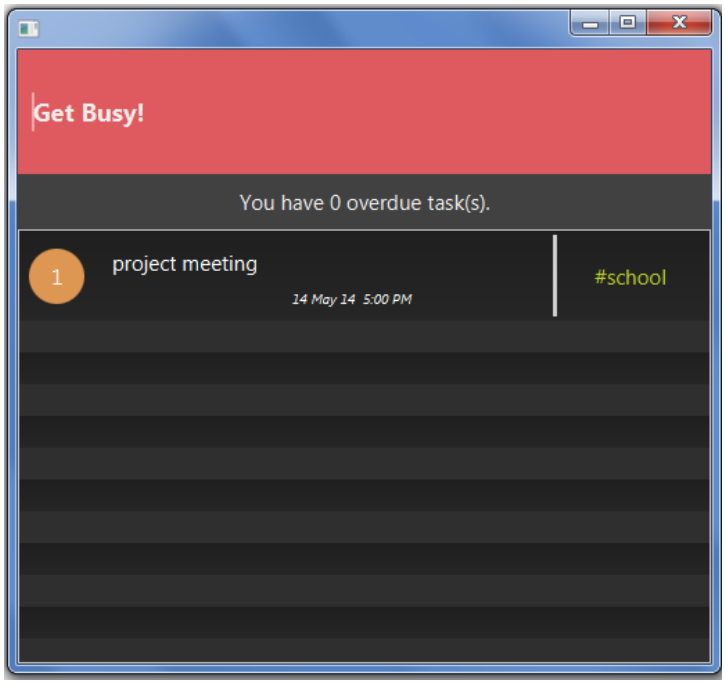
Finally, Task Panda also supports the use of multiple hashtags(#) so that users are able to specify certain keywords of the tasks that are being added. Hashtags categorizes tasks and by allowing you to search using hashtag keywords. More information can be found under *Section 11 - listing of tasks*.

To specify a hash tag, users can include a hashtag symbol followed by the hashtag keywords.

For example:



Result:



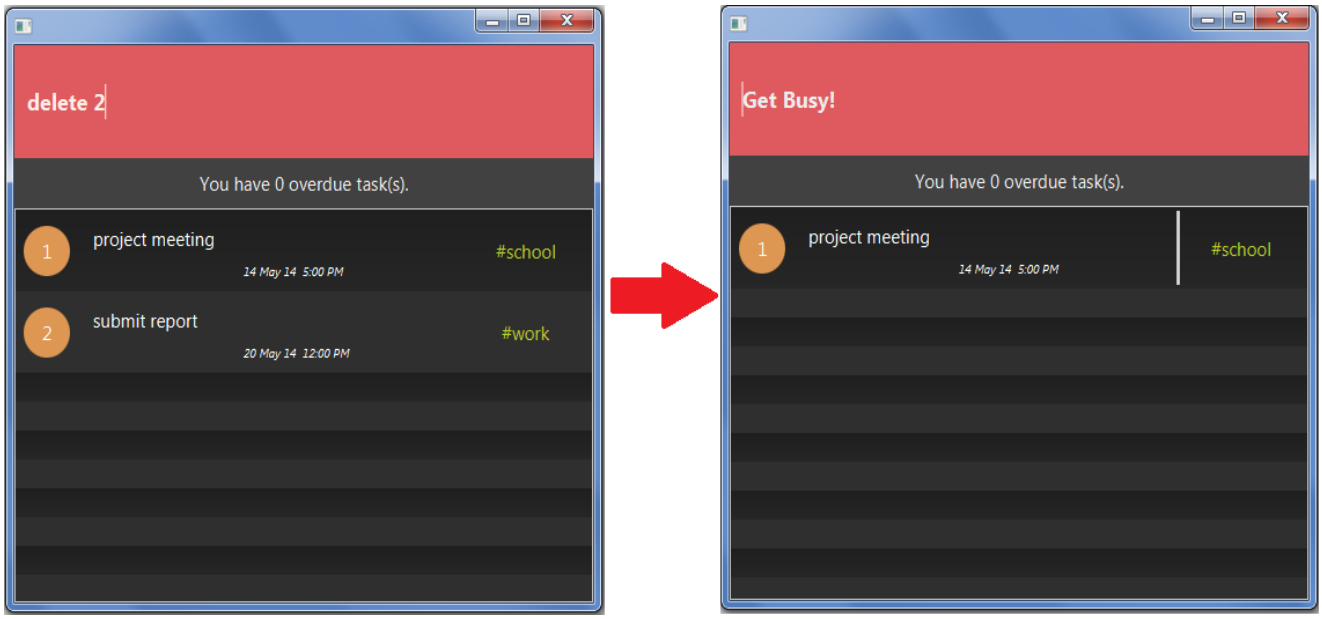
5. How to delete a task:

To delete a task, type the keyword “delete” into the input text field before specifying the index of deletion.

The format supported is:

- delete *<index>*

Example:



Task Panda also supports deletion after listing. For more information on listing formats, please refer to *Section 11 - listing of tasks.*

---

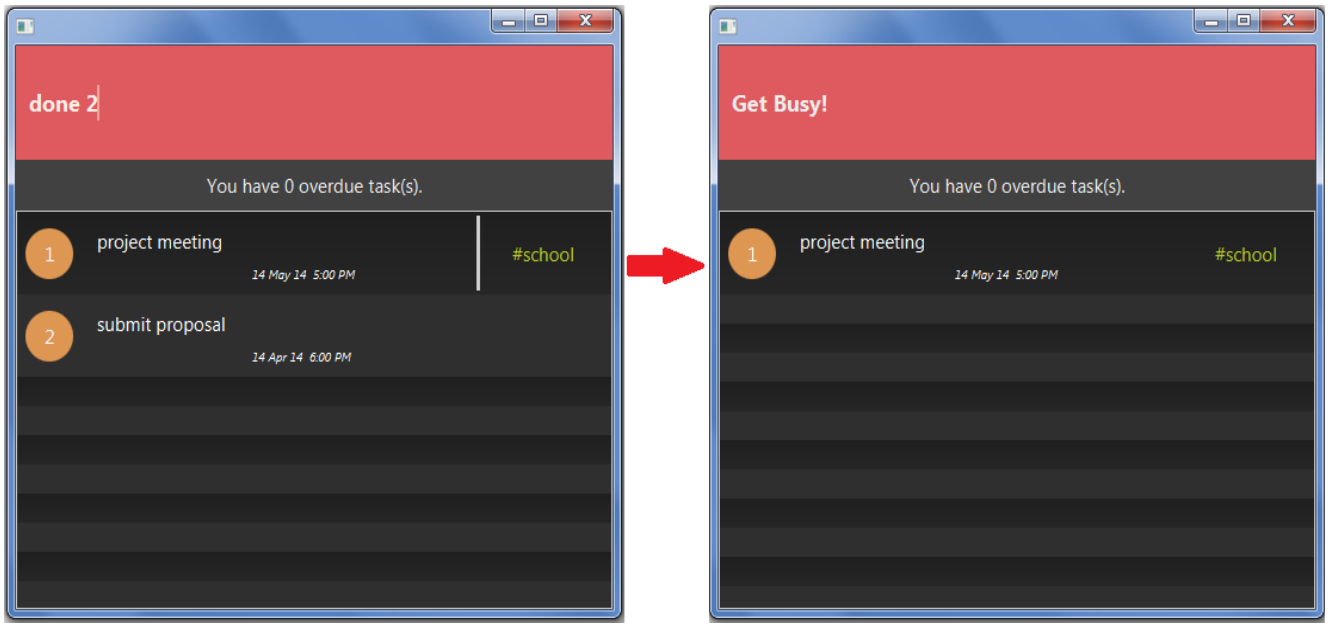
6. How to mark a task as done:

To mark a task as done or complete, users can simply type the keyword “done” into the input text field before specifying the index of task.

The format supported is:

- done *<index>*

Example:



Task Panda also supports the marking of tasks as done even after listing. For more information on listing formats, please refer to *Section 11 - listing of tasks*.

---

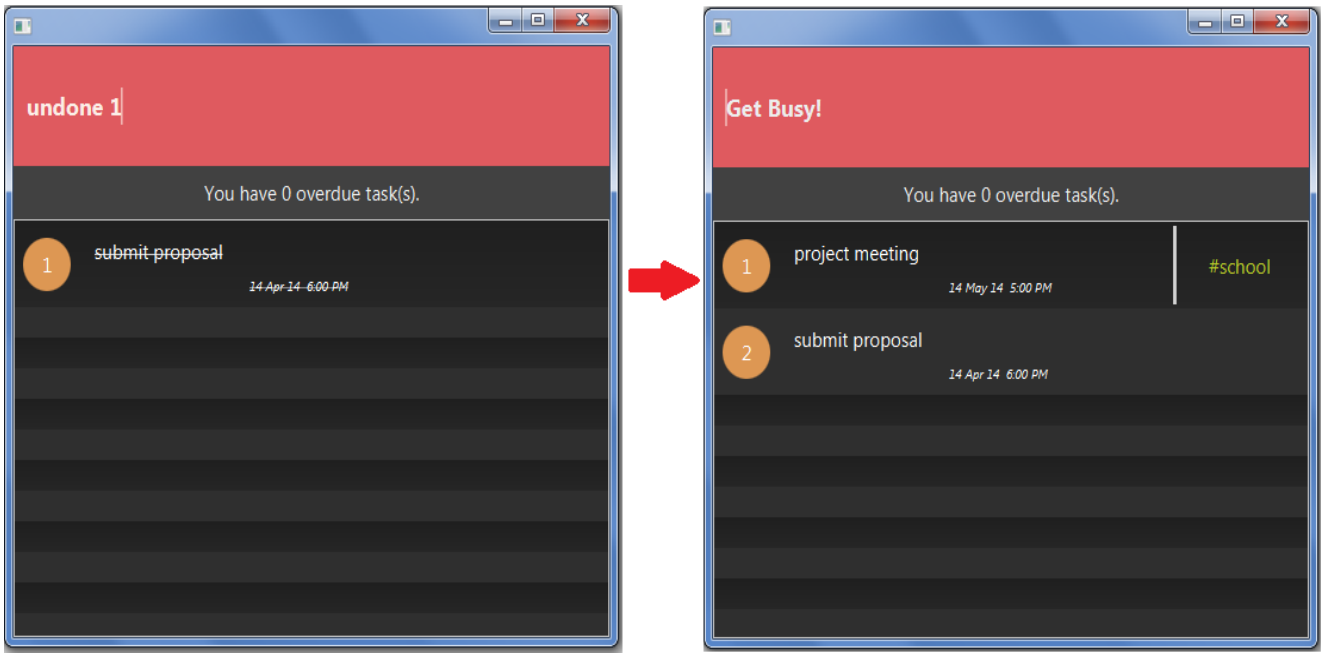
7. How to mark a task as undone

In order to unmark a task that has already been marked done or complete, users are required to type the keyword “undone” into the input text field before specifying the index of task.

The format supported is:

- undone <*index*>

Example:



> ***Note:***
> In order to view the list of tasks done. Simply key in “list done” into the input text field. For more information on supported listing formats, refer to *Section 11 - Listing of Tasks* of the User Manual.

---

8. How to edit a task:

There are three sections required in the input for you to edit a task:

1. Specify the keyword "*edit*" as the first word in the input text field.
2. Specify the index of that task to be edited.
3. New task input formats containing the description, date and time.

Note:
Task Panda requires you to specify all existing fields of the tasks again during the editing.
This is to ensure consistency in the program and the freedom to interchange between task types.

Alternatively, you can scroll up and down the task panel using the "up" and "down" key and select which task to edit with the "enter" key. The input text field will then be automatically filled with the three sections stated above.

8.1. Editing of task description

To edit task description, simply enter the task description after adding the "edit" keyword and the index.

The format supported is:

- edit <*index*> <*task description*>

8.2 Editing of date

To edit task date, simply enter the date after adding the "edit" keyword and the index.

The formats supported are:

- edit <*index*> <*task description*> on <*new date*> *by* <*time*>
- edit <*index*> <*task description*> from <*new date*> <*time*> to <new date> <time>

8.3 Editing of time

To edit task date, simply enter the date after adding the "edit" keyword and the index.

The formats supported are:

- edit <*index*> <*task description*> at/by <*new time*>
- edit <*index*> <*task description*> on <*date*> from <*new time*> to <*new time*>
- edit <*index*> <*task description*> from <*date*> <*new time*> to <*date*> <*new time*>

8.4 Editing tags

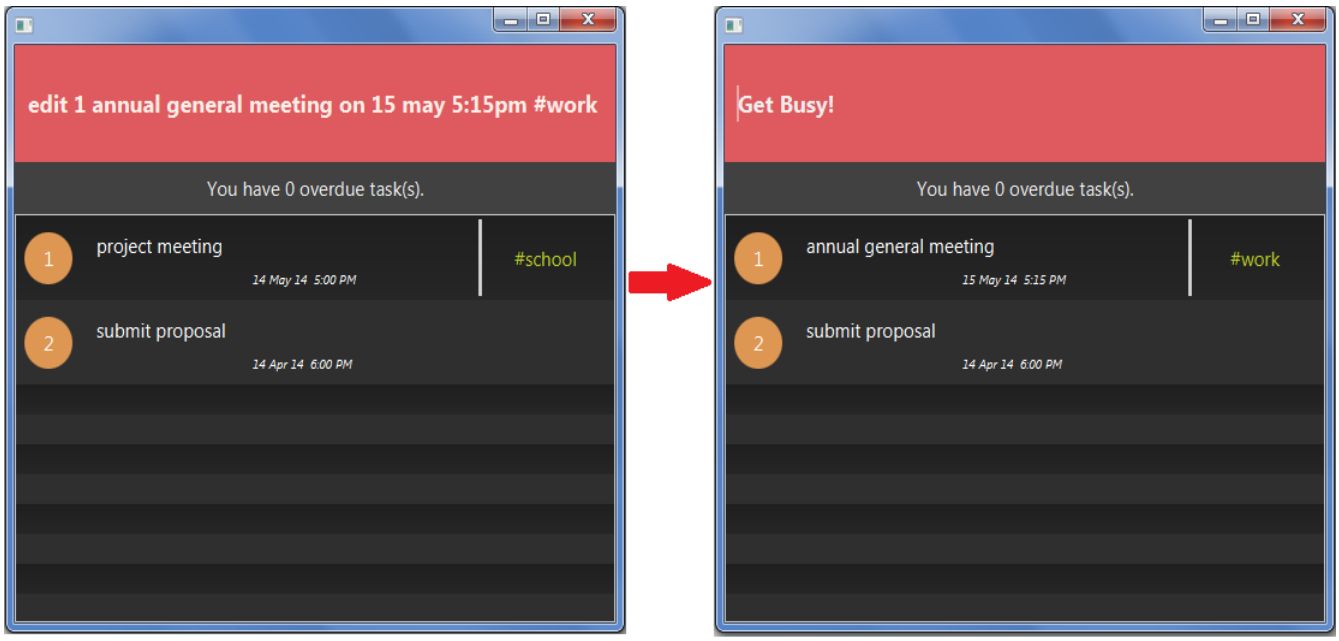
To edit the tag, simply enter # followed by the new hashtag description after adding the “edit” keyword and the index. You can also select the task to be edited by scrolling down with arrow key and press enter.

The format supported is:

- edit <*index*> <*task description*> #<new hashtag description>

8.5 Generic example

The example below shows a generic editing of a deadline task:



Note:
Entering the wrong format of date or time will result in the edition of task description. Should that happen, users are required to perform the “undo” operation if necessary *(refer to Section 9 - How to undo)*.

For more information on supported time and date formats, refer to *Section 12 - Supported time and date format* of the User Manual.

9. How to undo

The undo command is able to return the system the state prior to the execution of the last command. To perform an undo operation, the keyword “undo” must be specified.

Format supported is:

- undo

Task Panda undo feature achieves persistence by the implementation of an undo storage to store information of the previous state. Therefore the undo operation can still be called even when the session is new (i.e. running Task Panda again after closing).

10. How to redo

The redo command is able to return the system to the state prior to the last undo execution. To perform a redo operation, the keyword “redo” must be specified in the input text field.

Format supported is:

- redo

Similar to undo feature, Task Panda redo feature achieves persistence by the implementation of a redo storage to store information of the last undo state. Therefore the redo operation can still be called even when the session is new (i.e. running Task Panda again after closing).

> ***Note:***
> A user is unable to perform redo after performing any of the following commands excluding undo and redo.

---

11. Listing of tasks

All tasks are by default listed in the task panel. In listing, you have various criteria to search for various tasks. Following are the supported options for searching of tasks.

11.1 Listing of all tasks

To list all tasks that are yet to be done, simply specify the keyword “list” in the text input field.

Format supported is:

- list

11.2 Listing of miscellaneous tasks

To list all miscellaneous tasks that are yet to be done, simply specify the keyword “list” in the text input field followed by the “miscellaneous” task argument.

Format supported is:

- list misc

11.3 Listing of timed tasks

To list all timed tasks that are yet to be done, simply specify the keyword “list” in the text input field followed by the timed task argument.

Format supported is:

- list timed

11.4 Listing of deadline tasks

To list all deadline tasks that are yet to be done, simply specify the keyword “list” in the text input field followed by the deadline task argument.

Format supported is:

- list deadline

11.5 Listing of tasks with hashtag

To list tasks of specific alias, simply specify the keyword “list” in the text input field followed by the hashtag.
Formats supported are:

- list *#<word>*

11.6 Listing of tasks for weeks

Task Panda also supports the listing of tasks for the current week. Users are able to see what tasks are yet to be done for that week.

Format supported is:

- list this week

11.7 Listing of tasks today

Task Panda also supports the listing of tasks that are due or happening today.

Format supported is:

- list today

11.8 Listing of tasks for tomorrow

Task Panda also supports the listing of tasks that are due or happening tomorrow.

Format supported is:

- list tomorrow OR list tmw OR list tmr

11.8 Listing of tasks with a particular timestamp

Task Panda also supports searching of tasks with a particular timestamp. For deadline tasks, it will list out those whose deadlines fall on the same day as the timestamp and for time tasks, it will list out those in which the specified timestamp falls within the start and end interval.

Format supported is:

- list <date/time>

11.9 Listing of overdue tasks

To search for overdue tasks, simply type. Overdue tasks are highlighted in red color.

Format supported is:

- list overdue

> ***Note:***
> Refer to *Section 13* for combining different search criteria.

12. Supported time and date formats

12.1 Supported date formats

1. DD-MM-YYYY or DD-MM-YY (e.g. 12-12-2014, 12-12-14)
2. DD/MM/YYYY or DD/MM/YY (e.g. 12/12/2014, 12/12/14)
3. DD month YYYY or DD month YY (e.g. 12 nov 2014, 12 jan 14)
4. Relative dates (e.g. tomorrow, next monday, this monday)

12.2 Supported time formats

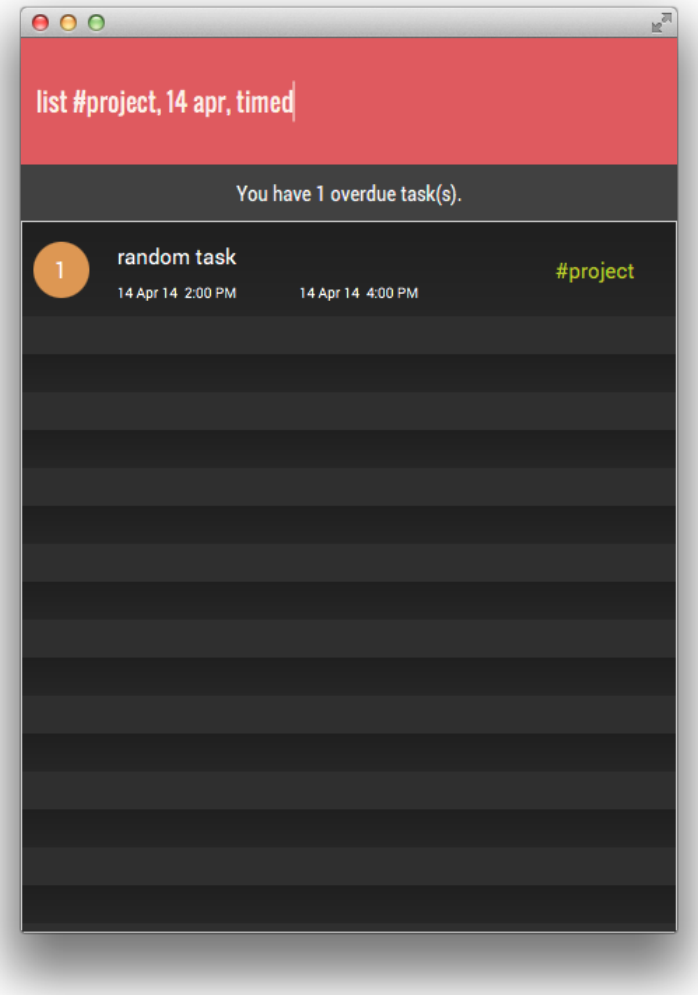
1. HH[am/pm] or HH:MM[am/pm] (e.g. 5pm, 6:15am)
2. HH:MM (e.g. 23:00, 00:00, 01:50)
3. HHMM (e.g. 2359, 1300)

---

13. Additional feature: Auto Complete, Power Searching

Task Panda also contains an auto complete feature. While the “search” keyword is active on the input text field, Task Panda will perform a concurrent search to provide users with real time information. This power search feature will allow you to search for their tasks easily by just specifying words off your mind.

Moreover, when listing, you can combine different criteria to locate your task more efficiently. Just use a comma as a delimiter. For instance, if you wish to search for timed tasks that have a “#project” hashtag, happening on 14th of April, you can type “list #project, 14 apr, timed”.



Task Panda Developer Guide

1. Introduction

Welcome to the developer guide of Task Panda. Task Panda is a powerful task management tool to help users manage their schedules and deadlines. It is catered specially to users who finds using the mouse cumbersome and prefer to work using the keyboard.

This manual contains developer documentation for working with Task Panda and will equip you with the essential knowledge and tips on all the information you need as a developer of Task Panda.

Getting Started

You will be coding in Java Object-Oriented Programming (OOP) and is expected to have Java Development Kit 7 (JDK7) and Java Runtime Environment 7 (JRE7) installed. Furthermore, you are also required to have extensive knowledge of using Eclipse, an Integrated Development Environment (IDE). Task Panda is hosted on Google Code and uses Mercurial as its revision control system (RCS). Therefore, as a developer of Task Panda, you are required to have the aforementioned items installed and ready for use. You are also required to be familiar with the use of such technologies.

---

2. Infrastructure

2.1 Development environment

- Eclipse IDE for EE developers [version Kepler]
- TortoiseHg [version: latest stable]
- Google Code project hosting

2.2 Technologies and Third-party libraries used in implementation

- **JSON** (JavaScript Object Notation)
  JSON is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript.
- **Gson**
  Gson is a Java library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object.
- **Joda-time**
  Joda-Time provides a quality replacement for the Java date and time classes. The design allows for multiple calendar systems, while still providing a simple API. Supporting classes include time zone, duration, format and parsing.
- **JavaFX**
  For drawing GUI components. External styling is controlled via CSS. However, javafx has very limited support for CSS unlike W3C.
- **Natty**
  Natty is a third party library for date parser. This allows for more powerful parsing of date objects from the user input string. Natty recognizes dates described in a plethora of ways.
- **Guava**
  Contains very useful data structures and utility classes. TaskPanda makes use of String splitters and set utility methods mostly.

2.3 Tools used in testing

- **JUnit**
  JUnit is a unit test framework in Java.

---

3. Software Design

3.1 Architecture

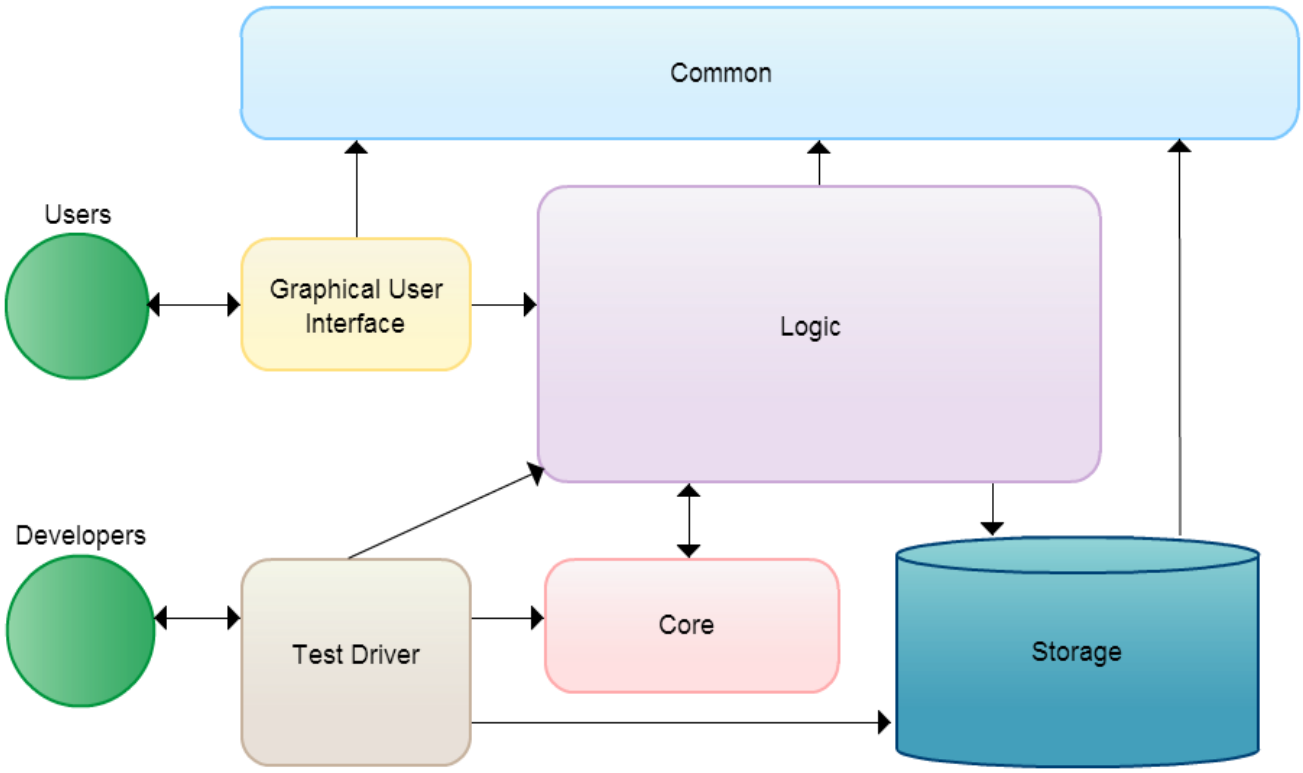


Fig 1 - Diagram of Components

Given above is the overview of the main components of Task Panda.

UI: The UI consists allows user to key in their input and a scroll pane which displays the list of tasks.
Logic: The main logic of the application is in the `CommandFactory` class which executes commands based on the user input and manipulates the task objects in a arraylist of task objects.
Storage: Tasks created by the users are stored in a JSON file. Also stores data needed to perform undo and redo operations.
Core: The core component facilitates the creation of `Task` objects based on user input while closely interacting with the logic components such as `TaskParser`.
Common: Consists of common components used across the application. Such components include our `PandaLogger` which is widely used across all other files.
Test Driver: Mainly consist of `JUnit` test cases for unit testing purposes.

The diagram below shows how the code is organized into packages inside each component and dependencies among them.

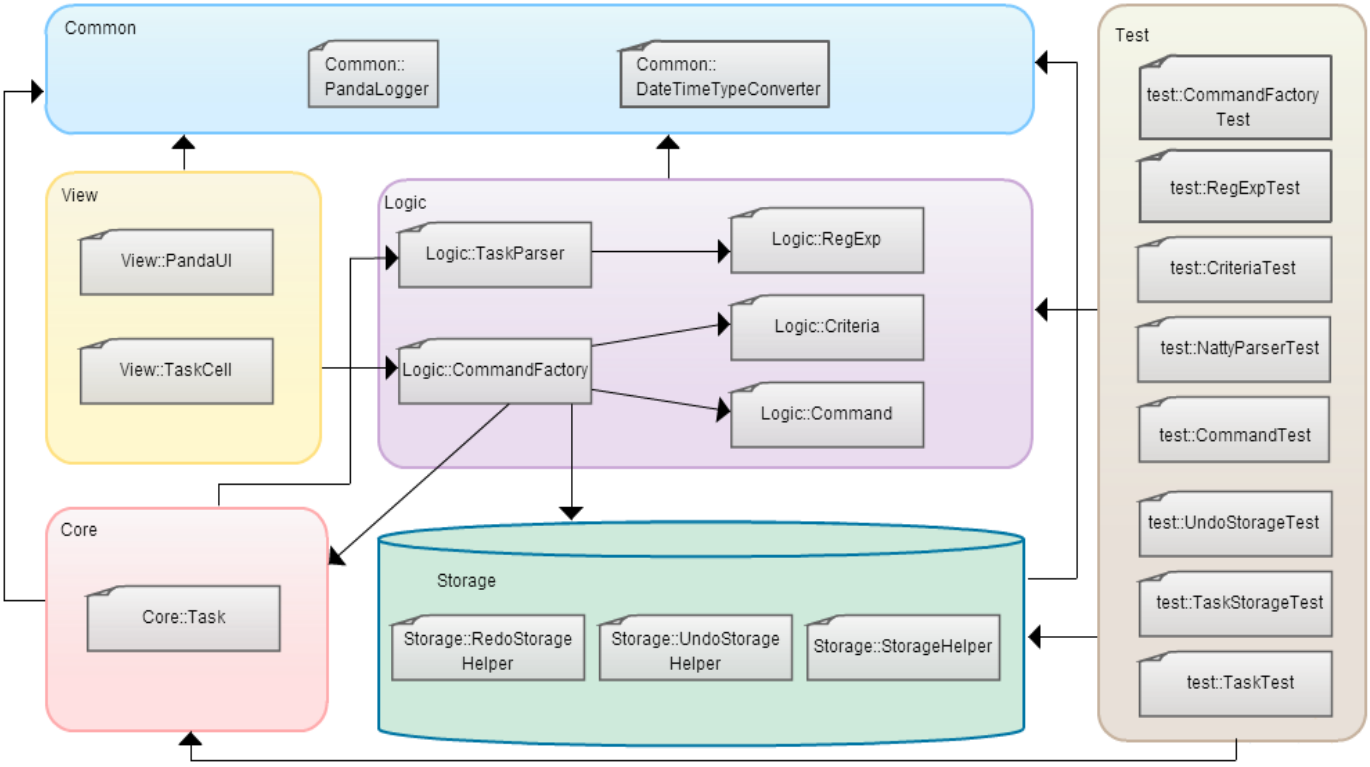


Fig 2 - Diagram of packages and dependencies

3.2 Request flow

1. User keys in "*add meeting on 24/4/14 from 2pm to 3pm*".
2. `TaskPandaUI` processes user input and passes input data to `CommandFactory` class.
3. `Command` object is created based on `COMMAND_TYPE` which is then passed back to `CommandFactory`.
4. `CommandFactory` validates commands and creates a `Task` object.
5. The task object calls on a `TaskParser` object to parse the input string of the user.
6. `TaskParser` then calls on `static` class, `RegExp` to obtain the date and time strings.
7. `TaskParser` then returns the initialized `DateTime` objects back to `Task`.
8. `CommandFactory` will then add the `Task` object into a `ArrayList<Task>` and calls on `StorageHelper` to store the list of tasks
9. `StorageHelper` will then store the list of tasks into the JSON file
10. The command along with the actual index of the task is pushed onto the undo stack which is to be stored in a JSON file using the `UndoStorage` class.
11. `CommandFactory` will then update `TaskPandaUI` and display the newly added `Task` object.

3.3 Logic

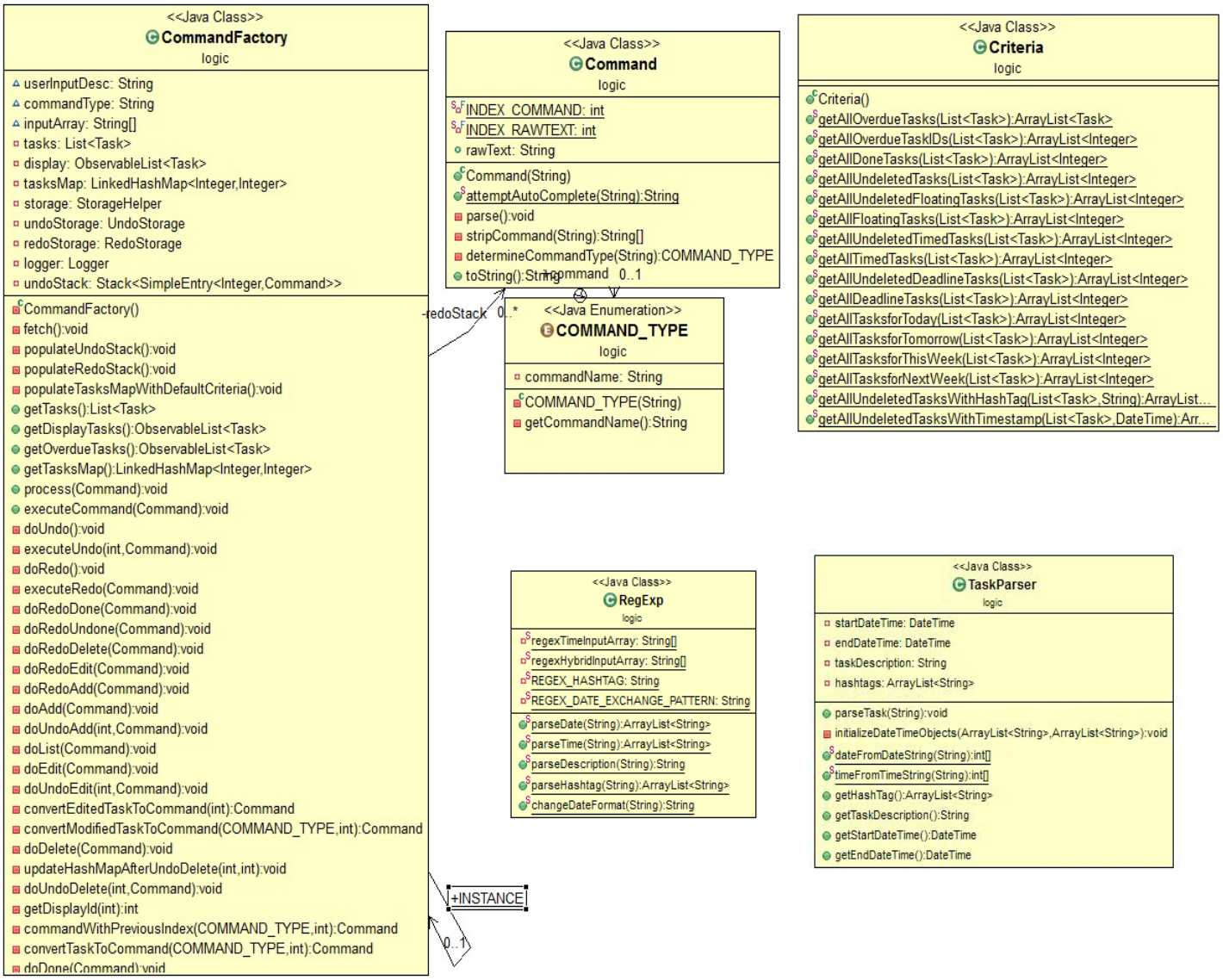


Fig 3 - Class Diagram of logic component

3.3.1 CommandFactory

The user interface creates a `CommandFactory` instance and passes the user input to it. The `CommandFactory` will then execute commands based on the command type determined by command class along with the extracted parameters to manipulate the task objects in the task arraylist. It follows the singleton design pattern. Therefore, only one instance of `CommandFactory` object will be created and used throughout the application.

As the tasks displayed to the users are different from the actual list of tasks stored in the JSON file, in order to ensure a consistent display of task index to the users, an internal `LinkedHashMap` is used.

The HashMap will store both the display ID and real ID of the task. At any point of time, the HashMap is correctly updated and will contain the list of mapping of display IDs to their real IDs. Whenever, a user performs a delete/done operation, the hashmap will be updated accordingly using the `updateHashMapAfterDelete()` method . For the undo operation of delete/done commands, the `updateHashMapAfterUndoDelete()` method will be called instead to update the hashmap and restore the task back to its original order. The sequence diagram below shows the execution of adding and deletion of task.

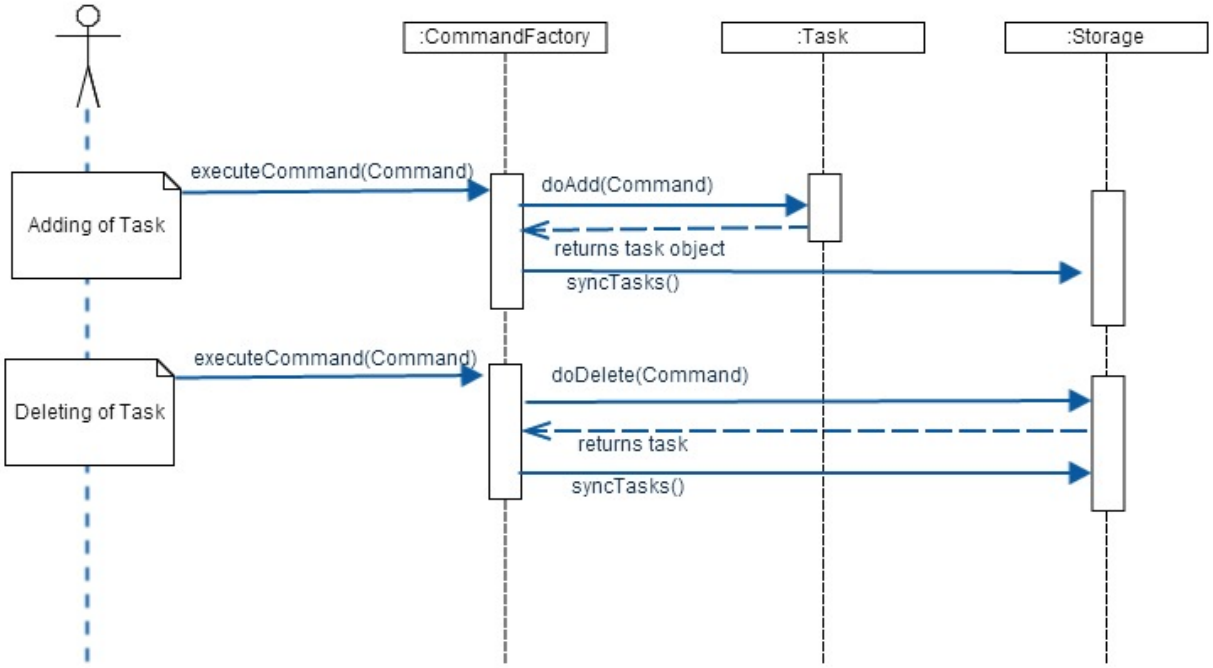


Figure 4 - Sequence diagram for CommandFactory

Addition of tasks

As seen in the diagram above, when the user keys in the add command followed by the respective task parameters, the command factory will parse the command to the task class which will create and return a task object. After which, the task object is added to the list of tasks before calling the `syncTasks()` method which will update the list of tasks to the data JSON file. Furthermore, the actual index along with the command executed is pushed to the UndoStorage JSON file for undo purposes.

Editing of tasks

Similar to the adding of task, the index of the task is first obtained from the command to locate the index of task to replace. Using the HashMap mentioned earlier, the actual index of the task is obtained using the index obtained from the command. A new task object is created based on the user input and is then used to replace the task object in the actual index. The `syncTasks()` method is

called to update the list of tasks to the data JSON file. Furthermore, the actual index along with the command executed is pushed to the UndoStorage JSON file for undo purposes.

Deletion / Marking a task as done

Deleting and marking a task as done is similar in a sense that both methods will require the locating of task and marking it as done or deleted. Using the index obtained from the command, the real index of the task is obtained using the HashMap. Depending on the operation, the task is mark as deleted or done using `setMarkAsDelete()` or `setTaskDone()`. In order to ensure the hashmap is updated correctly, the `updateHashMapAfterDelete()` method is called. In addition, to allow for undo operation and to restore the task back into its original order, the display ID as well as command is pushed onto the undo stack. Finally, the `syncTasks`() method is called to update the list of tasks to the data JSON file and the undo stack to the UndoStorage JSON file.

Undo / Redo Operation

Whenever the user launches the program, it will fetch the list of tasks along with the undo and redo stack from the JSON files. Both undo and redo operation is similar in that when the undo /redo operation is executed, the undo / redo stack is popped to obtain an entry. For undo operation an entry consisting of both the actual index as well as the command is obtained. The program will execute the respective undo operations based on the command along with the specified index to return to a state prior to the execution of the command. After which, the undo operation is converted to a command based on its undo operation before pushing onto the redo stack and calling the `syncTasks`() method. In the case for redo, the entry popped consist of only the command to be executed. For example in the case of the redo operation for the deletion of task, the method doDelete(Command) will be executed. Once again, after the redo operation, the `syncTasks`() method is called to update the JSON files.

Important APIs

Operation: `executeCommand(Command): void`
Description: Determine the action based on the Command object. After having decided, call respective method from the methods described below, passing the the user input.

Operation: `doAdd(Command):void`
Description: Creates a new Task object based on the command and write it to the tasks list. Command along with the real index of the task is pushed to the undo stack.

Operation: `doList(Command): void`

Description: Return the internal buffer of the list of all tasks. If there is a parameter to display differently, filter the buffer accordingly.

Operation: `doEdit(Command): void`
Description: The rawText must contain an ID to identify the task. Use the rest the of the information to create a new Task object. Afterwards, use the ID to locate the Task object from the buffer and replace with this newly created Task objects. In addition, the actual index of the task along with the command is pushed to the undo stack. Calls syncTasks() to update the database and undo storage JSON file.

Operation: `doDelete(Command): void`
Description: The rawText string must contain an ID to identify the task. Delete the task from the buffer and calls the writeTasks method of StorageHelper to update the database JSON file.

Operation: `doDone(Command): void`
Description: The rawText parameter must contain an ID to identify the task. Locate the task from the buffer and mark it is archive. Push the actual index of the done task along with the command onto the undo stack. Calls `syncTasks()` to update the database and undo storage JSON file.

Operation: `doUndone(Command): void`
Description: Unmark a done task and push the actual index of the task along with its command onto the undo stack. Calls `syncTasks()` to update the database and undo storage JSON file.

Operation: `doUndo(): void`
Description: Pops the entry from the undo stack and based on the command obtained, perform the respective undo operation to return to the state prior to the execution of the command.

Operation: `doRedo(): void`
Description: Pops the entry from the redo stack and based on the command obtained, perform the respective redo operation to return to the state prior to the execution of the last undo command.

3.3.2 Command

The Command class determines the command type from the user input and creates Command objects. This class uses an `Enum` of `COMMAND_TYPE` to define command types from a given command string.

3.3.3 Criteria

The criteria class provides a plethora of static methods to help filtering out the tasks in the buffer according to necessary criteria. Currently supported criteria are

- filtering floating (also known as misc) tasks
- filtering timed tasks
- filtering deadline tasks
- filtering tasks with a particular hashtag
- filtering tasks that falls on tomorrow, this week, next week, and finally
- filtering task with a particular timestamp.

This timestamp is also parsed through our parser. Therefore, it can be any format that we support.

These methods will return an integer list of the IDs of the tasks that match the criteria. Furthermore, the `doList()` function in `commandFactory` makes use of a Set data structure to provide searching of combining criteria in different ways. For instance, "list #project, timed, 14 apr 3pm" will search for timed tasks that have a hashtag of "#project" and "14 apr 3pm" timestamp falling in between the interval of task start and end time.

3.3.4 TaskParser

The `TaskParser` class is called by the `Task` class from the core component. It aids on the creation of `Task` objects by parsing user inputs given by the `Task` objects.

The sequence diagram below shows the workflow between `TaskParser` and `RegExp`.

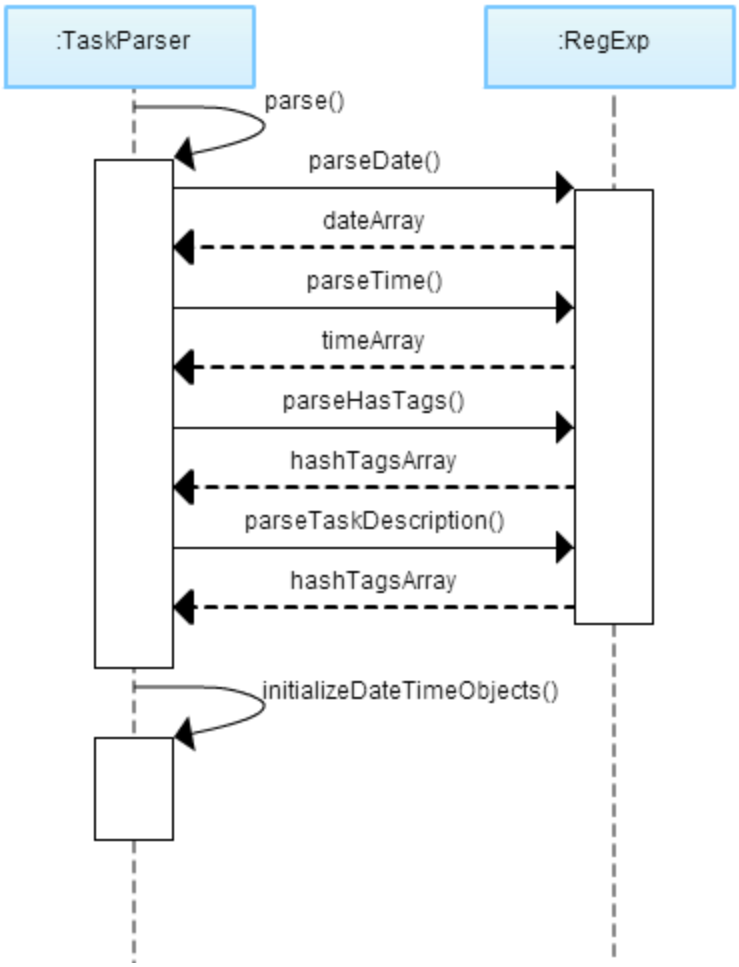


Figure 5 - Sequence diagram for TaskParser and RegExp

According to Figure 5, `TaskParser` first calls on the static class, `RegExp` to extract description, date, time and hashtag patterns in different arraylist of strings. More information of this extraction can be found in *Section 3.3.5 - RegExp*.

With the arraylist of timestamps, `TaskParser` will logically create two `DateTime` objects through the `initializeDateTimeObjects()` method.

`TaskParser` will first initialize all primitive date and time variables through the use of two methods: `DateFromDateString()` and `TimeFromTimeString()`. These two methods will return an integer array of date or time for the instantiating of the primitive variables.

`TaskParser` will then logically deduce which task type was specified by the user by counting the number of specified timestamps and create the `DateTime` objects accordingly using the initialized primitive date and time variables.

The `DateTime` objects will then be returned to the `Task` object along with the list of hashtags and the parsed task description.

Note:
`DateFromDateString()` and `TimeFromTimeString()` methods uses Natty Date parser, a third party library. Therefore, it is important to update and run NattyParserTest to ensure that Natty Date parser supports any new date or time formats that will be included in the future.

Important APIs

Operation: `parseTask():void`
Description: Calls on `static` methods of the class, `RegExp.java` to obtain description, date and time variable. These methods include: 1. `parseDate(userInput)` 2. `parseTime(userInput)` 3. `parseDescription(userInput)` The method will then create `DateTime` objects using the `initializeDateTimeObject()` method and returned the `DateTime` objects to the `Task` object.

Operation: `initializeDateTimeObjects(ArrayList<String>, ArrayList<String>`
Description: Given two ArrayList of time and date as parameter, this method will call on `initializeDate(ArrayList<String>)` and `initializeTime(ArrayList<String>)` methods to initialize primitive date and time variables. It will then create two `DateTime` objects using the `finalizeDateTime()` method.

Operation: `dateFromDateString(String)`
Description: Given a date string (e.g. 16 march, tomorrow), this method will use Natty Date Parser to obtain an integer array of date.

Operation: `timeFromTimeString(String)`
Description: Given a time string (e.g. 5:15pm), this method will use Natty Date Parser to obtain an integer array of time

3.3.5 RegExp

`RegExp` is a class that consists of `static` methods to assist in parsing of date and time from the user inputs. This class is mainly utilised by the `TaskParser` class.

The main function of the class is to extract out substrings that resemble timestamps or hashtags and return it to `TaskParser`. It is done using the following methods: `parseDate(String)`, `parseTime(String)` and `parseHashTags(String)`. This can be illustrated in Figure 5.

The aforementioned methods compares user inputs with a set of predefined regex patterns. Once a timestamp or hashtag pattern is matched, it will extract the matched string into an `ArrayList<String>` and returned to `TaskParser`.

`RegExp` extracts task description by running through all the regex pattern arrays and replacing all matched date, time and hashtag regex patterns with an empty substring.

Therefore, to add additional support for new date and time formats, it must be done through the addition of regex patterns. New supported date and time formats must be added into `regexHybridArray` and `regexTimeArray` respectively.

Important APIs

Operation: `parseDate(String): ArrayList<String>`
Description: Matches user input to regex patterns and extract matched date patterns into an ArrayList <String> and returned to TaskParser.

Operation: `parseTime(String): ArrayList<String>`
Description: Matches user input to regex patterns and extract matched time patterns into an ArrayList <String> and returned to TaskParser.

Operation: `parseHashtags(String): ArrayList<String>`
Description: Matches user input to regex patterns and extract matched hashtagpatterns into an ArrayList <String> and returned to TaskParser.

Operation: `parseTaskDescription(String): ArrayList<String>`
Description: Matches all regex patterns and replace them with an empty substring.

3.4 Core

The class diagram below shows the dependencies of the Core component.

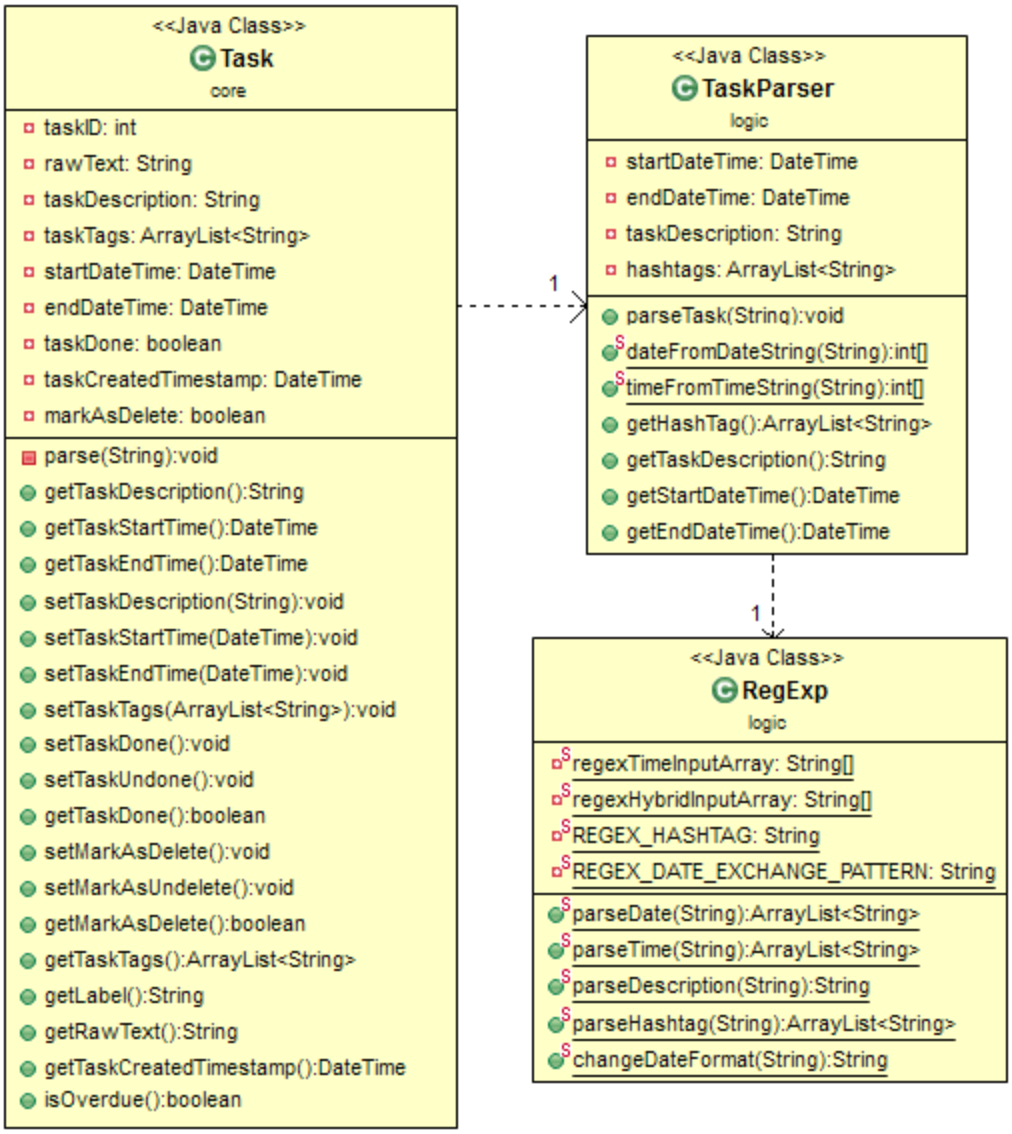
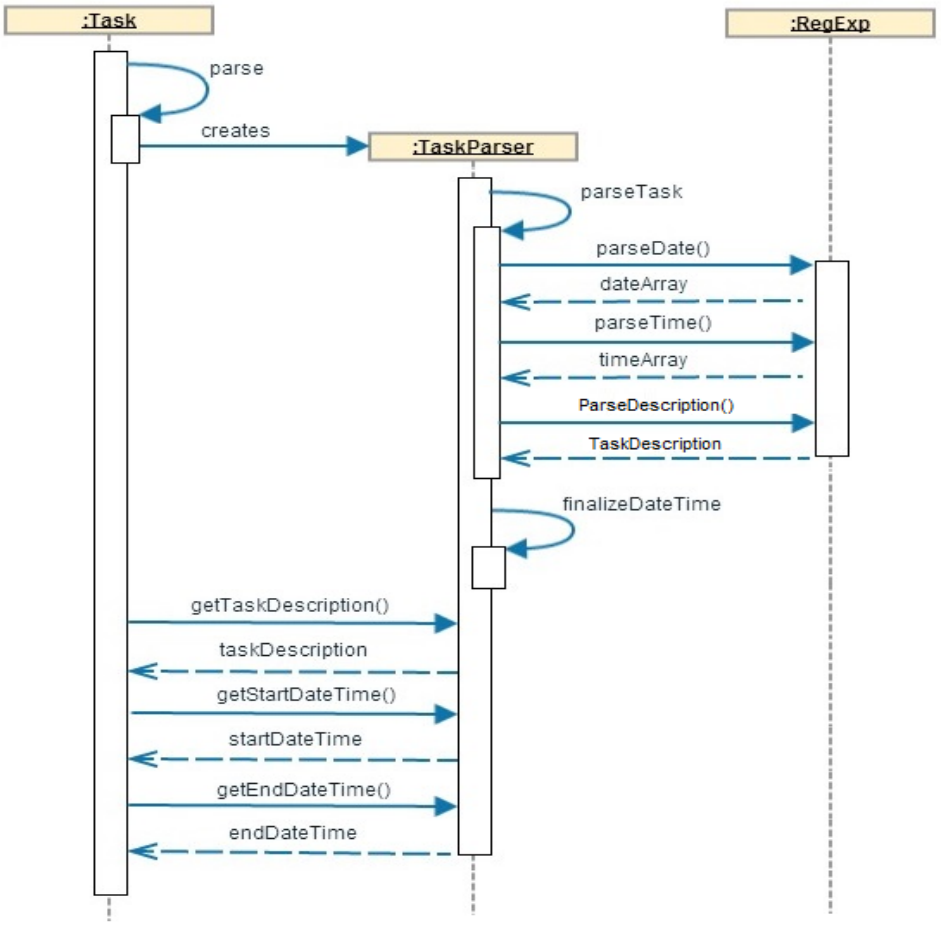


Fig X - Class Diagram of core component

Note:
Some internal methods which are not relevant were removed. For more information on those methods and their explanation, refer to *Appendix A - list of API references*.

The core component contains the `Task` class. It holds all important attributes of a single user task and it comprises mostly of getter and setter methods. These getter methods are mostly used to populate the user interface. On the other hand, the setter methods are mostly used for the editing, deleting, undo or redo of tasks.

The `Task` class is called by `CommandFactory` when user specify an **add** command. It facilitates the creation of task objects by calling `TaskParser` which will in turn, call the `RegExp` in the logic component. The sequence diagram below shows the process of the task creation when a `Task` object is being created.



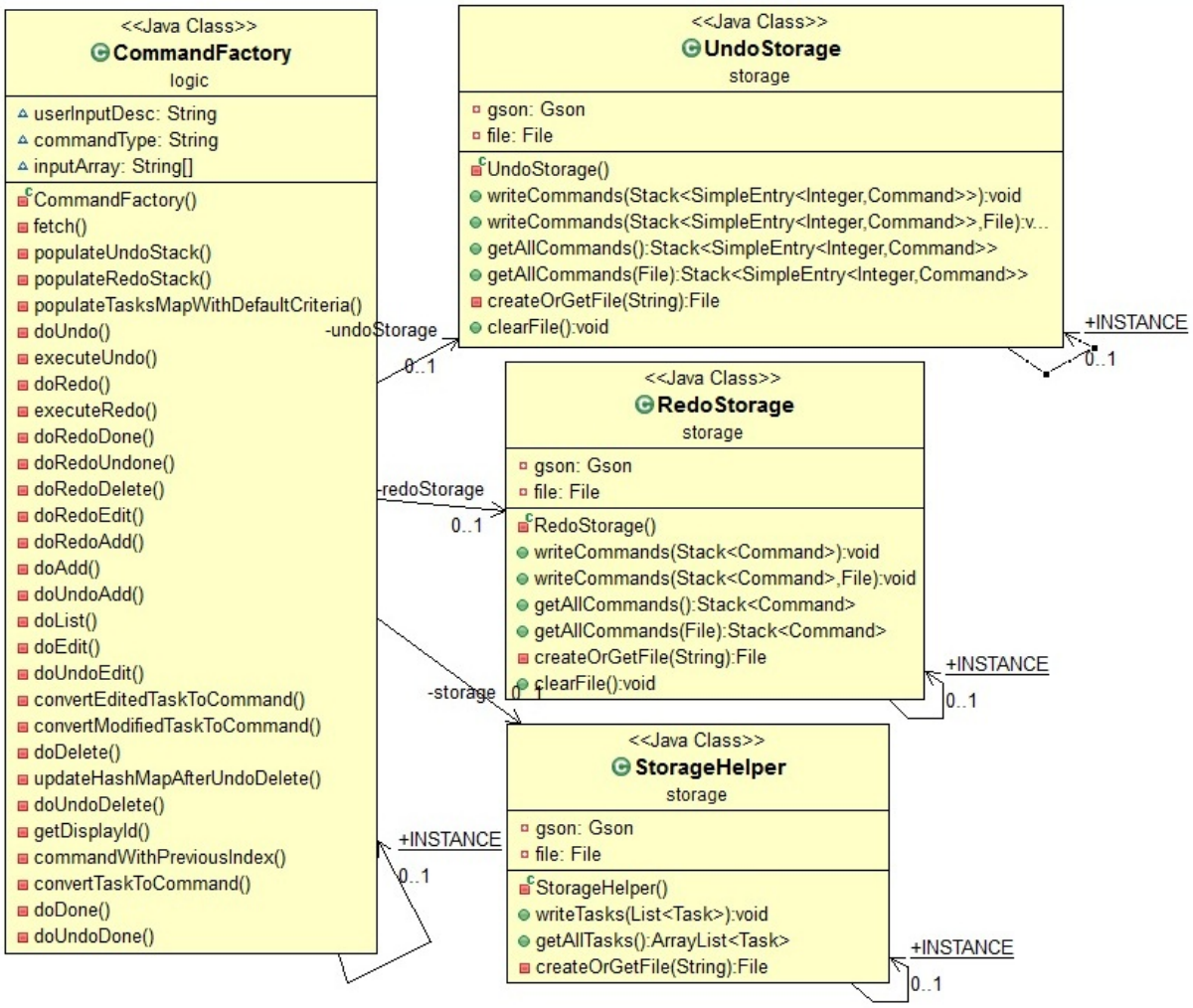
The `Task` object created by `CommandFactory` invokes its internal `parse()` method to create a `TaskParser` object. The TaskParser object which will then call on `static` class `RegExp` to parse the time and date and store the data temporarily. The `Task` object will then get the necessary information from the getter methods in `TaskParser`.

There are several functions of `Task` objects. As each `Task` object contains information of a single task, the `CommandFactory` class stores a list of `Task` objects and uses it for manipulation according to inputs specified by the user. Furthermore, `Task` objects are also used for conversion into JSON objects for file storing.

Important APIs

Operation: `parse():void`
Description: Constructs a `TaskParser` object followed by calling three methods to initialize its description and `DateTime` variables: 1. `getTaskDescription()` 2. `getStartDateTime()` 3. `getEndDateTime()`

2.5 Storage



2.5.1 StorageHelper

The `Task` objects in the task arraylist are first converted to JSON objects before storing in a JSON file using the `StorageHelper` class. After execution of respective commands, `CommandFactory` will call the `syncTasks()` method to write the modified tasks to the data JSON file as well as to write the undo and redo stacks to the UndoStorage and RedoStorage JSON file respectively.

`getAllTasks()` method retrieves all the tasks from the database JSON file and populates the internal buffer of task list in CommandFactory.

Important APIs

Operation: `StorageHelper()`
Description: The constructor of `StorageHelper` class. Initializes the database JSON file as well as creates a `Gson` object with required properties such as - custom DateTime serializer - enabling complex may key serialization

Operation: `createOrGetFile(String filename): File`
Description: If there is already a database JSON file, locate it. Otherwise, create a new File instance and return it.

Operation: `writeTasks(List<Task> t):void`
Description: Converts all the tasks in the list to its string representation forms and write it into the database JSON file. Throws an exception should there be any error regarding file I/O.

Operation: `getAllTasks(): ArrayList<Task>`
Description: Locate the database JSON file, read its content and create an `ArrayList` of Task objects and return it.

Operation: `clearFile(): void`
Description: Under the hood, it deletes the database JSON file and creates a new one. Internally calls `getOrCreateFile()` method.

The sequence diagram below shows the common flow between the Logic component and Storage component.

2.5.2 UndoStorage

The undo feature of Task Panda achieves persistence by the implementation of an undo storage. Whenever a command is executed, the command along with the actual task ID is pushed onto an undo stack. The stack is then updated onto a JSON file using the UndoStorage class. When the user exits the program, the undo data is pushed to the undo

stack. Upon the execution of an undo command, the top entry will be popped from the undo stack and return the program to the previous state of operation.

Important APIs

Operation: `writeCommands(Stack<SimpleEntry<Integer, Command>>):void`
Description: Converts all the entries in the stack to its string representation forms and write it into the database JSON file. Throws an exception should there be any error regarding file I/O.

Operation: `createOrGetFile(String filename): File`
Description: If there is already a database JSON file, locate it. Otherwise, create a new File instance and return it.

Operation: `getAllCommands(File file): Stack<SimpleEntry<Integer, Command>>`
Description: Locate the database JSON file, read its content and create a stack of both integer and command objects and return it.

2.5.3 RedoStorage

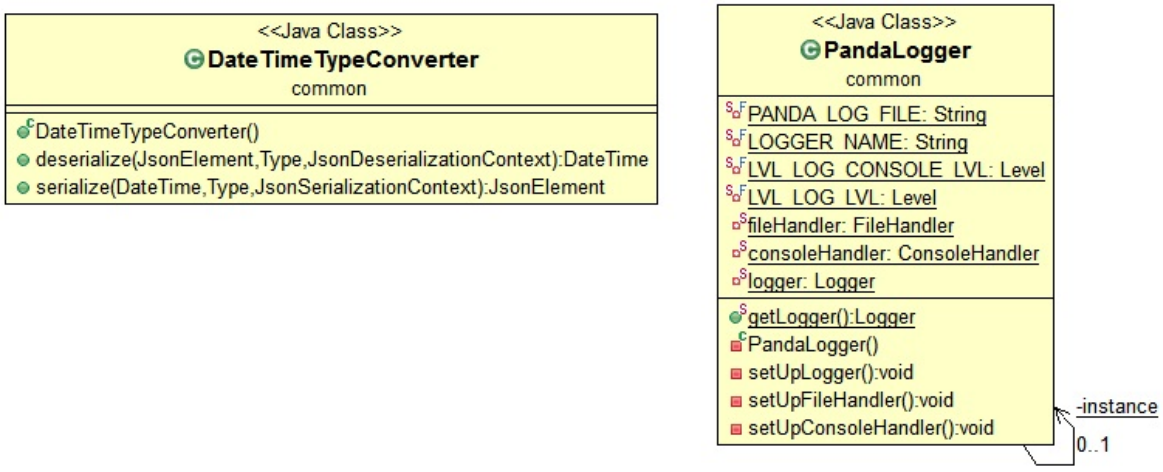
Similar to undo storage, the redo storage is stored into a stack and stored in a JSON file. Whenever an undo command is executed, the command is pushed onto a redo stack. The stack is then updated onto a JSON file using the RedoStorage class which will call `writeCommands()` to update and write the stack to the JSON file. Upon the execution of a redo command, the top entry will be popped from the redo stack and manipulate the command within the entry to return the state of the program prior to the undo operation.

Operation: `writeCommands(Stack<Command> c) : void`
Description: Converts all the entries in the stack to its string representation forms and write it into the database JSON file. Throws an exception should there be any error regarding file I/O.

Operation:	`createOrGetFile(String filename): File`
Description: If there is already a database JSON file, locate it. Otherwise, create a new File instance and return it.	

Operation:	`getAllCommands(File file): Stack<Command>`
Description:Locate the database JSON file, read its content and create a stack of command objects and return it.	

3.6 Common



The common component contains common utilities used across the application.

The `DateTimeTypeConverter` is a custom serializer class to aid in the conversion of Joda Datetime into a readable string to save in the JSON file.

`PandaLogger` is used mainly for logging purposes and consists of a few wrapper methods. This is used during development for console logging so that problematic areas and errors can be identified easily. A singleton object is instantiated from this class once and used throughout the application.

Important APIs

DateTimeConverter

Operation: `deserialize(JsonElement JSON, Type type,JsonDeserializationContext context)`
Description: Retrieve the JSON string from the JsonElement and pass it to the constructor of Joda `DateTime` class. Joda DateTime constructor is flexible and intelligent enough to correctly convert it into a `DateTime` object.

Operation: `serialize(DateTime datetime, Type type, JsonSerializationContext context)`
Description: Get the string representation of DateTime object and return it.

3.7 View

The view component consists of classes required for creating a graphical user interface mainly responsible for user interaction.

Up till V0.2, TaskPanda makes uses of Java Swing library for drawing UI, but from V0.3 onwards, we have switched to JavaFX library since it provides more control and flexibility to design a better user interface. The design decision remains the same, a text field for command input and a dynamic display of task listview based on the specific command.

Input Field

This is glorified text field with various key listeners and a text property listener for concurrent keyword searches. Enter key will execute the specified command. However, for strings starting with "search" keyword, as soon as the user finishes typing a space afterwards, it will start perform a concurrent keyword search and update the listview. This is done in `handleSearch()` method.

Overdue Label

This is the text label to display the number of overdue tasks. To achieve this, its text property is bound to the size of overdue tasks. However, if there is an error in user input, it temporarily switches its status to show the error message. This is done by temporarily unbinding with overdue tasks and setting the text to error message.

Task ListView

Consists of a `observableList` which is an interface comes along with JavaFX. On startup, this `observableList` is populated with all the tasks ordered by the created timestamp. Overdue tasks are highlighted red. This listview also have key listeners for “Up” and “Down” arrows to enable scrolling one by one with keyboard as well as “PageUp”/”PageDown” for faster scrolling up and down. Moreover, the “Esc” key will shift the focus back to the inputField. If the user presses “Enter” on a selected task, it will assume that the user wants to edit the task and populates the input field with the selected task details. Likewise, if the user presses “Delete”, it will assume that the user wants to perform a delete operation and populates the input field with delete command and the index of the selected task.

Below are the classes being used in view component.

PandaUI.java - A class inherited from JavaFX Application. Consists of main method as well as initialization of text field for input and list view for displaying task list. Furthermore, it also initializes necessary key listeners. The input field also have a listener providing concurrent keyword searches.

TaskCell.java - A class for customizing default list cell. It sets the layout as a grid, add necessary UI components such as the task description, timestamps, hashtags. In addition, each component has its own CSS classes and ids.

resources/css/style.css - Controls mostly styling and paddings of UI components.

API

The method interacting with other components is

Operation: `processCommand(String inputText): void`
Description: This API is called to called to create a Common object from the user input text. Afterwards, it passes the newly created Command object to `commadFactory.process(command)` to execute the command accordingly. Upon processing, the bottom feedback panel is redrawn to update the display.

---

4. Future Milestones

Currently, in addition to the basic functionalities, TaskPanda supports a simple power search feature as well as a flexible command format that allows natural language. However, there are definitely areas for improvement and we have identified areas for further work as listed below.

Search for empty slots
An improved searching system should be implemented to allow users to search for empty slots available so that the user can easily plan his schedule.

Google Calendar Integration
Allow integration with google calendar to allow users to identify scheduled tasks quickly just be taking a look at the calendar.

Improve the feedback system
The feedback system needs to be improved to allow the display of error messages to the user so that the user is aware of the mistakes made.

Custom settings
Users should have the choice of modifying the colour scheme as well as font of the program depending on their preference.

Provide more search power by combining AND, OR, NOT searches
In this scenario, users can make use of AND, OR and NOT keywords to have more power in searching for their desired tasks.

Appendix A - List of API references

PandaUI class

Method	Description
start(Stage)	Initializes the stage. Set the main UI layout as borderPane layout and fill the top part with input field and center part with the listview. Set necessary attributes search as the title and icon of the application
createTrayIcon(final Stage)	If the OS platform supports system tray, create a system tray icon and a menu.
setupGlobalHotkey	When the application is in background, attach a global hotkey listener to listen for “Ctrl+Space” key combination to bring the application to front.
show(final Stage)	Helper function to bring the application to front
setUpScene(BorderPane)	Setup necessary attributes such as the width and height of the application. Attach the CSS file for design and add listeners for width and height changes to account for dynamic resizing of the application
addInputField()	Initializes and create a custom textfield which will become our input field.
handleInputKeyListener()	Attach a key listener to the input field. Does various actions based on the key provided by the user. “Enter” key will execute the command and “Down” key will allow the user to shift selection focus to listview.
addBottomComponents()	Initializes and create a listview for displaying task lists
addHelpText()	Initializes the label for displaying help text manual
addOverStatus()	Initializes the label for displaying number of overdue tasks
addTaskList()	Add the listview into the stage
handleListKeyListener()	Add necessary key listeners for the listview.
handleEnterKey()	Typing “enter” key will go into editing mode of the task
handleEscKey()	Typing “esc” key will switch the focus back to the input field
handleDeleteKey()	Typing “delete” key will go into deleting mode of the task

handleSearch()	Handle concurrent searches using partial keywords. Provides a similar autocomplete experience in UI
updateTaskList()	A helper method to make sure the listview is updated and showing correct data at all times

Taskcell class

Method	**Description**
TaskCell()	Initializes the components inside and set their respective attributes such as CSS classes, ids, width and height values. CSS classes and ids are added so that each component can then be styled by external stylesheet.
addControls()	Add UI components to the grid layout
addContent()	Initializes mentioned UI components with necessary details to be displayed

CommandFactory class

Method	**Description**
populateUndoStack()	Fetches the undo stack from the UndoStorage JSON file
populateRedoStack()	Fetches the redo stack from the UndoStorage JSON file
populateTaskMapWithDefaultCriteria()	Fill the entries of TaskMap with all the tasks that are not deleted and done
getTasks()	Returns the list of task
getDisplayTasks()	Collect all the ids of the tasks to be displayed
getOverdueTasks()	Search for all overdue tasks.
getTasksMap()	Returns the TasksMap list used for mapping of indexes
process(Command)	Calls executeCommand(Command) method
executeCommand(Command)	Executes command based on the Command given
executeUndo(int, Command)	Performs an undo operation to return to the previous state based on the command and task index
executeRedo(Command)	Performs a redo operation to return to the previous state prior to the execution of the last undo command

doRedoDone(Command)	Calls doDone(Command) method
doRedoUnDone(Command)	Calls doUndone(Command) method
doRedoDelete(Command)	Calls doDelete(Command) method
doRedoEdit(Command)	Calls doEdit(Command) method
doRedoAdd(Command)	Calls doAdd(Command) method
doUndoAdd(int, Command)	Deletes the previously added task
doList(Command)	Display the list of tasks based on the criteria specified
doUndoEdit(int, Command)	Edits the edited task back to its original state
convertEditedTaskToCommand(int)	Method to convert tasks that are edited into command for redo edit
convertModifiedTaskTo Command(Command.COMMAND_TYPE, int)	Method to convert tasks that are added or edited previously into command for undo operation
updateHashMapAfterUndoDelete (int, int)	Updates hashmap back to the original order prior to deletion
doUndoDelete(int, Command)	Unmark the task marked as deleted
getDisplayId(int)	Obtains the actual display ID of task
commandWithPreviousIndex(Command.COMMAND_TYPE, int)	Method to get command for Undo Done/Delete operations
convertTaskToCommand(Command.COMMAND_TYPE, int)	Method to get command for Redo Delete/Done/Undone operations
doUndoDone(int, Command)	Unmark the task that is marked done
doUndoUndone(int, Command)	Mark the task that has just been undone as done
checkDeleteInput(String)	Method to check delete parameter if it is > 0 and not exceeding the size of task list
checkEditIndexInput(String)	Method to check edit parameter if it is > 0 and not exceeding the size of task list
checkIfNumberBelowArraySize(String)	Checks if the number specified is within the size of task list
isPositiveNonZeroInt(String)	Checks if a number is > 0

obtainUserEditInput(String)	Removes the index from the raw text input and return the remaining text
getFirstWord(String)	Obtain the first word of the string
isValidString(String)	Checks if a string is valid and not null
checkIfFileEmpty()	Checks if task list is empty
showToUser()	Display feedback to the user
getFakeIDbyRealID(int)	Return the fake index of task based on its actual index
syncTasks()	Writes and updates the undo stack, redo stack and tasks list to the respective JSON file
updateHashMapAfterDelete(int)	Updates hash map after deletion to ensure order is correct
testGetDisplayId(int)	Calls the getDisplayId(int) method to return the display id for testing purposes
clearUndoRedoAfterTesting()	Removes the undo/redo stack entries after testing
getLastIndex()	Returns the last index of the tasks list
testAdd(Command)	Calls doAdd(Command) method for testing purposes
testEdit(Command)	Calls doEdit(Command) method for testing purposes
testUndo()	Calls doUndo(Command) method for testing purposes
testRedo()	Calls doRedo(Command) method for testing purposes
testDelete(Command)	Calls doDelete(Command) method for testing purposes
testDone(Command)	Calls doDone(Command) method for testing purposes
testUndone(Command)	Calls doUndone(Command) method for testing purposes

Command class

Method	**Description**
parse()	Method will determine what command type is being specified by user
stripCommand()	Method obtains the command from user input by stripping the first word
determineCommandType(String)	Method determines command type based on user command

toString()	Method used for debugging purposes

Criteria class

Method	Description
getAllOverdueTasks(List<Task>)	Returns a list of overdue tasks
getAllOverdueTaskIDs(List<Task>)	Returns the ID of all overdue tasks
getAllDoneTasks(List<Task>)	Returns a list of all done tasks
getAllUndeletedTasks(List<Task>)	Returns a list of all undeleted tasks
getAllUndeletedFloatingTasks(List<Task>)	Returns a list of undeleted floating tasks
getAllFloatingTasks(List<Task>)	Returns a list of all floating tasks
getAllUndeletedTimedTasks(List<Task>)	Returns a list of undeleted timed tasks
getAllTimedTasks(List<Task>)	Returns a list of all timed tasks
getAllUndeletedDeadlineTasks(List<Task>)	Returns a list of undeleted deadline tasks
getAllDeadlineTasks(List<Task>)	Returns a list of all deadline tasks
getAllTasksForToday(List<Task>)	Returns a list of all tasks that fall on today
getAllTasksForTomorrow(List<Task>)	Returns a list of all tasks that fall on tomorrow
getAllTasksForThisWeek(List<Task>)	Returns a list of all tasks that falls within this week
getAllTasksForNextWeek	Returns a list of all tasks that falls on next week
getAllUndeletedTasksWithHashTag(List<Task>, String)	Returns a list of tasks that has matched hashtag and are undeleted
getAllUndeletedTasksWithTimestamp(List<Task>, DateTime)	Returns a list of undeleted tasks with timestamp

Task class

Method	Description
parse(String)	Method parses user input via TaskParser.java and obtain necessary information from it
setID(int)	Setter method to set TaskID
getID(int)	Getter method to get TaskID
setTaskDescription(String)	Setter method to set TaskDescription
getTaskDescription()	Getter method to get TaskDescription
getTaskStartTime()	Getter method to get start DateTime
setTaskStartTime(DateTime)	Setter method to set start DateTime
setTaskEndTime(DateTime)	Setter method to set end DateTime
getTaskEndTime()	Getter method to get end DateTime
addNewTag(String)	Method adds additional tags to task
setTaskTags()	Setter method to replace with new list of task tags
setTaskDone()	Setter method to mark task as done
getTaskDone()	Getter method to check if task is done
setTaskUndone()	Setter method to set task as undone
setMarkAsDelete()	Setter method to delete task
setMarkAsUndelete()	Setter method to undelete task
getMarkAsDelete()	Getter method to check if task was deleted
getTaskTags()	Getter method to get array list of hash tags
getTags()	Getter method to get a string of hash tags
getLabel()	Getter method to get task type label
getRawText()	Getter method to return raw text of task
getTaskCreatedTimeStamp()	Getter method to return CreatedTimeStamp
setTaskCreatedTimeStamp(DateTime)	Setter method to set CreatedTimeStamp
isOverdue()	Method checks if task is overdue

TaskParser class

Method	Description
parseTask(String)	Method parses user input by feeding it into reg exp and obtaining necessary information from it
initializeDateTimeObjects(Array List<String>, Arraylist<String>)	Method initializes start and end DateTime objects by calling initializeTime, initializeDate, intializeTaskType and finializeDateTime
initializeTime(ArrayList<String>)	Method initializes start and end time accordingly given a list of time strings
initializeStartTime(int[])	Method initializes primitive start time variables
intializeEndTime(int[])	Method initializes primitive end time variables
initializeDate(ArrayList<String>)	Method initializes start and end date accordingly given a list of date strings
initializeStartDate(int[])	Method initializes primitive start date variables
initializeEndDate(int[])	Method initializes primitive start date variables
initializeTaskType(int, int)	Given number of time and date specified, method sets
finializeDateTime()	Given determined task type, method finalizes the DateTime objects
finalizeFloatingTask()	Creates null start and end DateTime objects
finalizeTimedTask()	Logically creates start and end DateTime objects
finalizeDeadlineTask()	Logically creates an end DateTime objects
intializeDateToToday()	Initializes date to current date
initializeTimeToMidnight()	Calls intializeStartTimeToMidnight() and initializeEndTimeToMidnight()
intializeStartTimeToMidnight()	Initializes start time to 12am
initializeEndTimeToMidnight()	Initializes end time to 12am
dateFromDateString(String)	Method uses NattyTime parser to parse a given string into an integer array of date

timeFromTimeString(String)	Method uses NattyTime parser to parse a given string into an integer array of time
getHashTag()	Getter method to return ArrayList of hashtags
getTaskDescription()	Getter method to return parsed task description
getStartDateTime()	Getter method to return start DateTime object
getEndDateTime()	Getter method to return end DateTime object

RegExp class

Method	**Description**
parseDate(String)	Method extracts matched date string patterns and store it in an ArrayList
parseTime(String)	Method extracts matched time string patterns and store it in an ArrayList
parseDescription(String)	Method removes all matched date, time and hashtags patterns by calling the 3 methods: removeHybridPatterns, removeTimePatterns and removeHashtagPatterns
removeHybridPatterns(String)	Method removes all matched date patterns
removeTimePatterns (String)	Method removes all matched time patterns
removeHashtagPatterns(String)	Method removes all matched hashtag patterns
parseHashtag(String)	Method extracts matched hashtag patterns and store it in an ArrayList
changeDateFormat(String)	Method changes date format from DD/MM/YYYY to MM/DD/YYYY to cater to NattyTime's parsing of US date format

DateTimeTypeConverter class

Method	**Description**
serialize(DateTime, Type, JsonDeserializationContext)	Method gets the string representation of DateTime object and return it.
deserialize(DateTime, Type, JsonSerializationContext)	Method retrieves the JSON string from the JsonElement and pass it to the constructor of Joda DateTime class. Joda DateTime constructor is flexible and intelligent enough to correctly convert it into a DateTime object.

PandaLogger class

Method	Description
setUpLogger()	initializes a logger instance as well as calls the below two methods to setup file and console loggers
setUpFileHandler()	initializes external log file
setUpConsoleHandler()	initializes logging in console as well