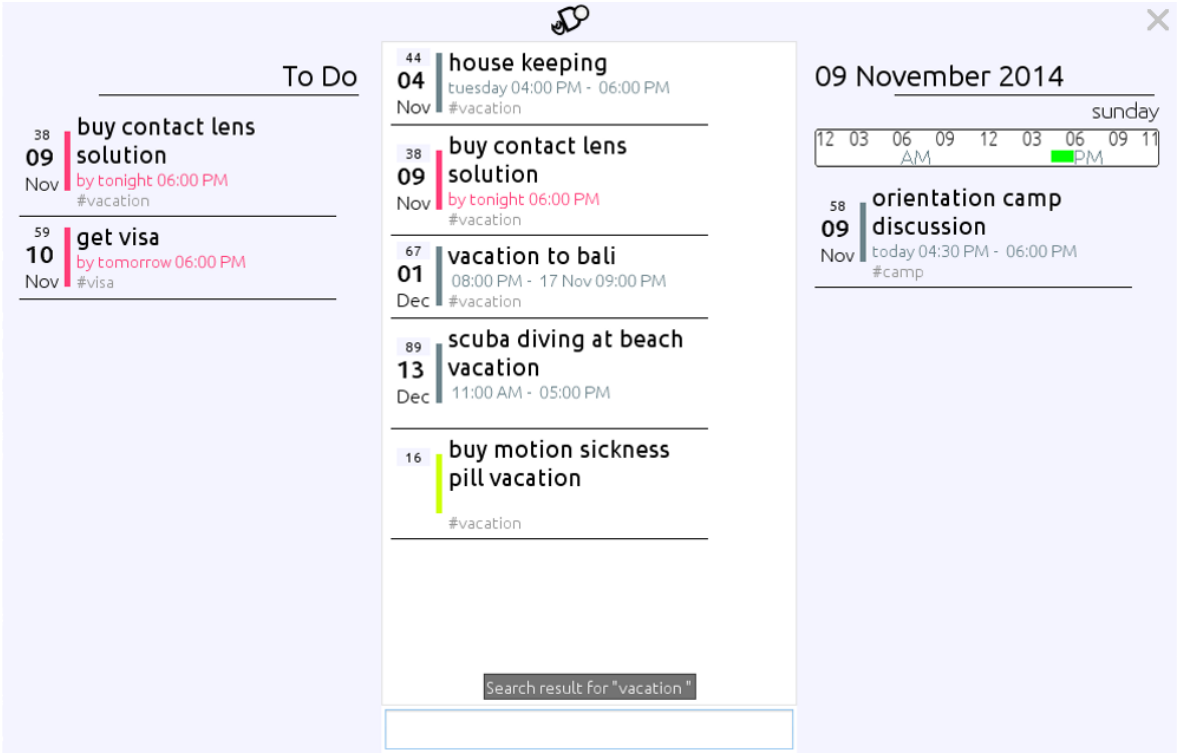
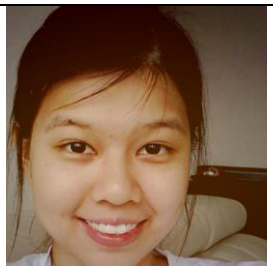
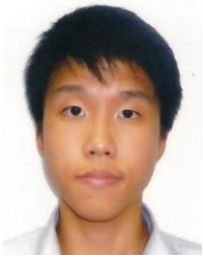
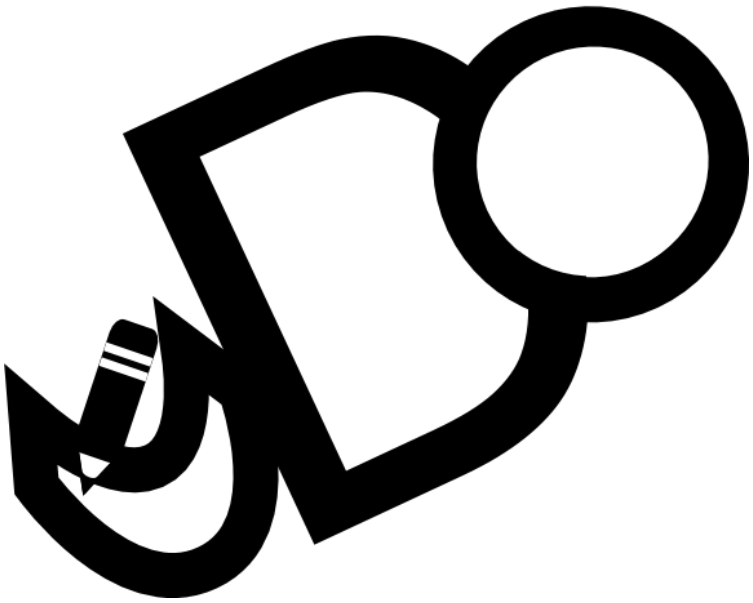
uDo



Supervisor: **CHANG YAN QIAN**

Special Feature: **Good GUI**

		
Nicholas Lum Team Lead Integration Tool Expert (Git)	Chong Jia Wei Scheduling and Tracking Deliverables and Deadlines Testing	Leonardo Sjahputra Code Quality Documentation Tool Expert (Swing)



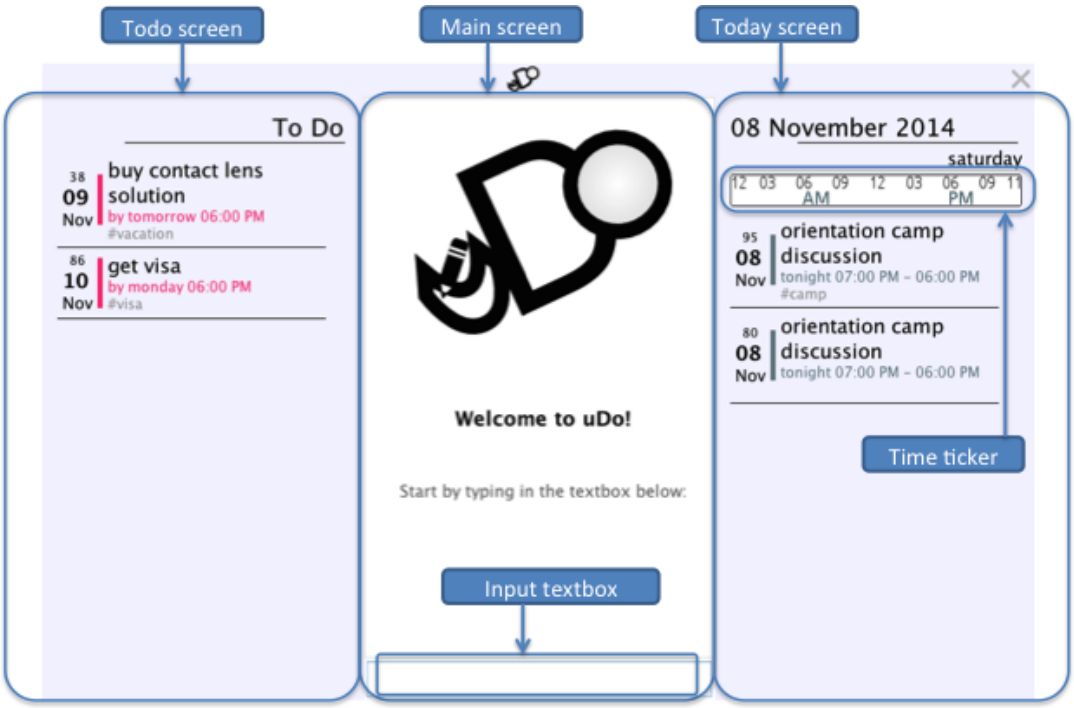
uDo User Guide

Table of Contents

1. Introduction

uDo is a schedule manager application that helps you keep track of the things you need to do. This guide will walk you through the various features of uDo and how to use them.

2. uDo at a glance



Main Screen
This is where the interaction with the program takes place. Every command you input will be reflected here.

To-Do Screen
To-Do screen shows all tasks not marked as done with deadlines up to 3 days away.

Today Screen
Today Screen shows all activities that are scheduled on that particular day.

Tip!
When the rows of entries shown grow longer than the screen, use Alt+Q and Alt+A, Alt+W and Alt+S, Alt+E and Alt+D to scroll up and down the To Do Screen, Main Screen and Today Screen respectively

Time Ticker
This represents the time of the day (12am to 11:59pm). Occupied time will be shown in green. If there are multiple activities with overlapping timing, it will be shown in red.

Input Textbox
This is where you type all commands into uDo.

Tip!
When uDo is the active window, press enter return the focus to the textbox

Tip!
You can press the up or down button in the textbox to cycle through the past 5 commands

3. Quick start

Here is a quick list of things that uDo can do for you:

- Adding activities
- Listing
- Searching
- Undoing previous actions
- Editing
- Marking
- Deleting

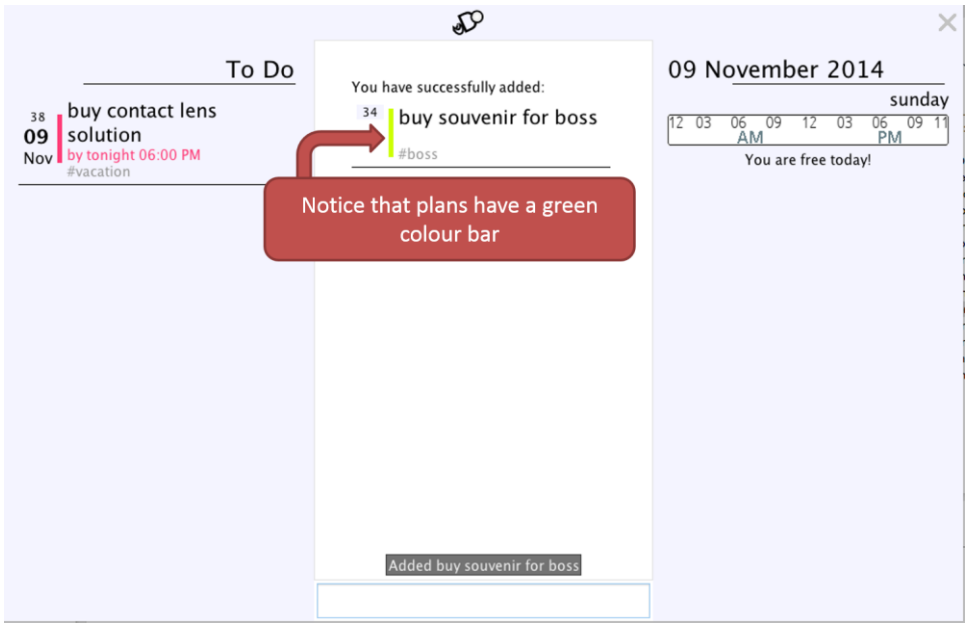
3.1. Adding activities

uDo allows hashtagging to be used within the activities you add. The hashtags help you group related items in uDo for easy retrieval. Hashtags are case-insensitive. In the following examples, hashtags will be weaved into commands to show you how uDo hashtags items.

To add a plan into uDo, enter add followed by the plan's description.

add buy souvenir for #boss

uDo will display the plan you have just added:



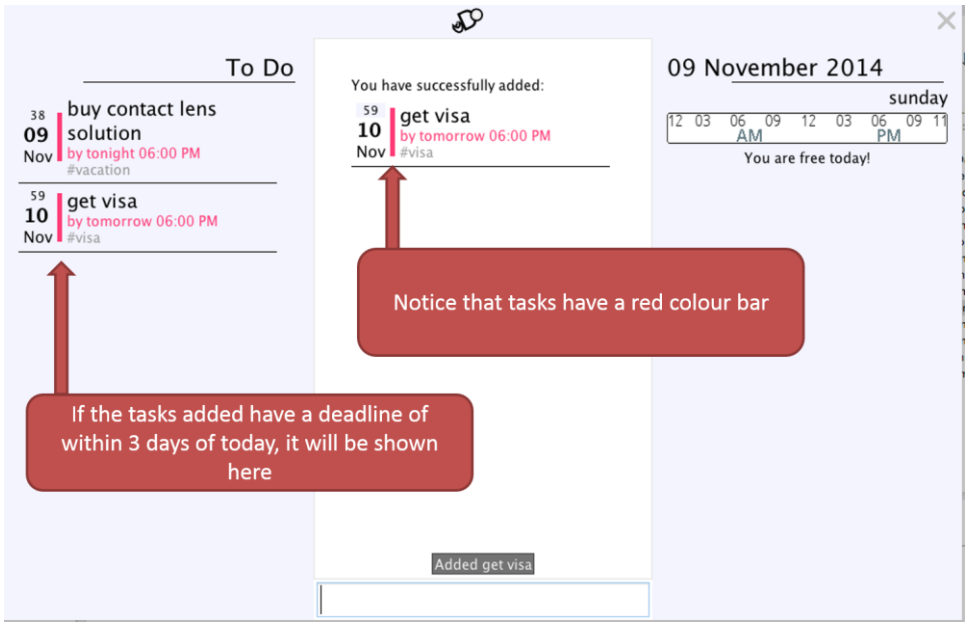
Note!

Plans are tasks with no deadlines, hence they will not have a deadline assigned to them

To add a task into uDo, enter add, followed by the task's description, the keyword "by", and the deadline. Your task deadline has to include both the date and time for it to be recognised by uDo. Refer to Section 4.2 for the date and time formats that uDo supports.

```
add get #visa by tomorrow 6pm
```

uDo will display the task you have just added:



Note!
Your tasks should only have one deadline assigned to them

To add an event into uDo, enter add followed by the event's description, the keyword "**from**" a starting date and time, "**to**" an ending date and time. Likewise, the date and time format is also flexible.

```
add orientation #camp discussion from today 4:30pm to 6:00pm
```

uDo will display the event you have just added:

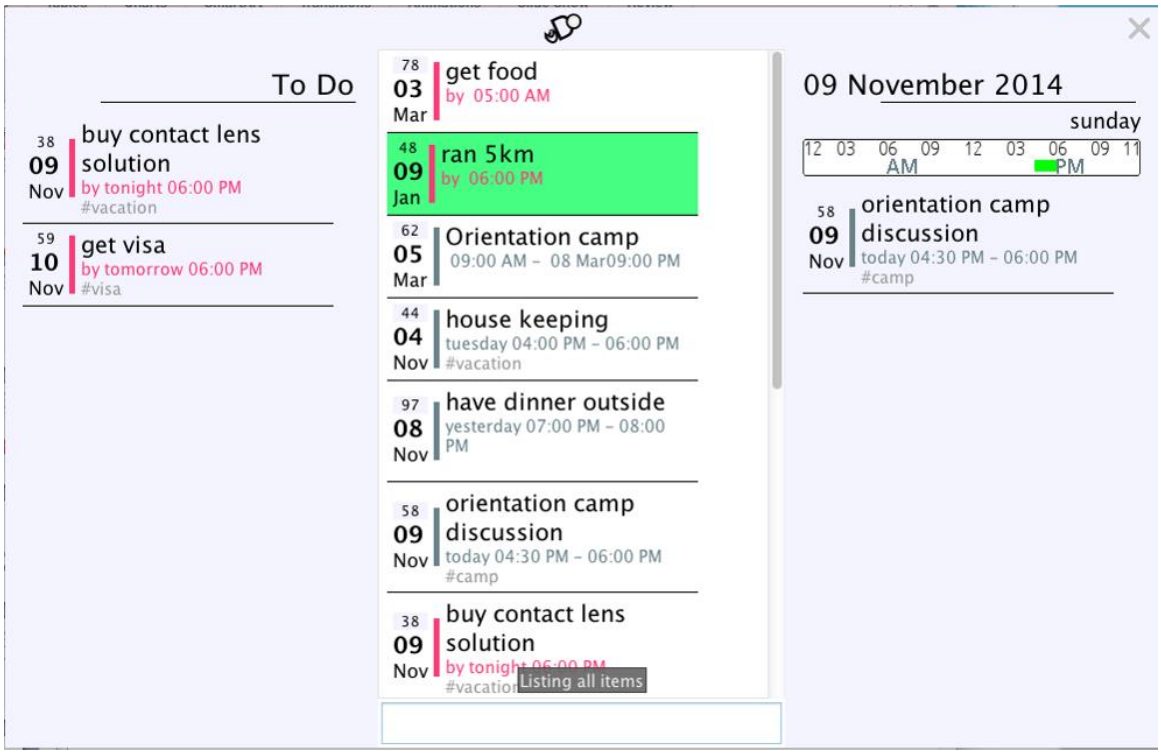
Note!
Your event has to have a starting date and time and an ending time, but the ending date may be omitted

3.2. Listing

To list all activities (sorted by date), enter:

```
list all
```

uDo will display the following:



To list all task and plans that have been marked as done, enter:

list done

To list all activities containing a certain hashtag (for example, vacation), enter:

list #vacation

To list all activities on a certain date, enter:

list 9/2

Again, refer to Section 4.2 for other date and time formats that uDo supports.

To list all plans, enter:

list plan

To list all tasks, enter:

list task

To list all events, enter:

list events

Note!

Listed activities will appear on the Main Screen

3.3. Searching

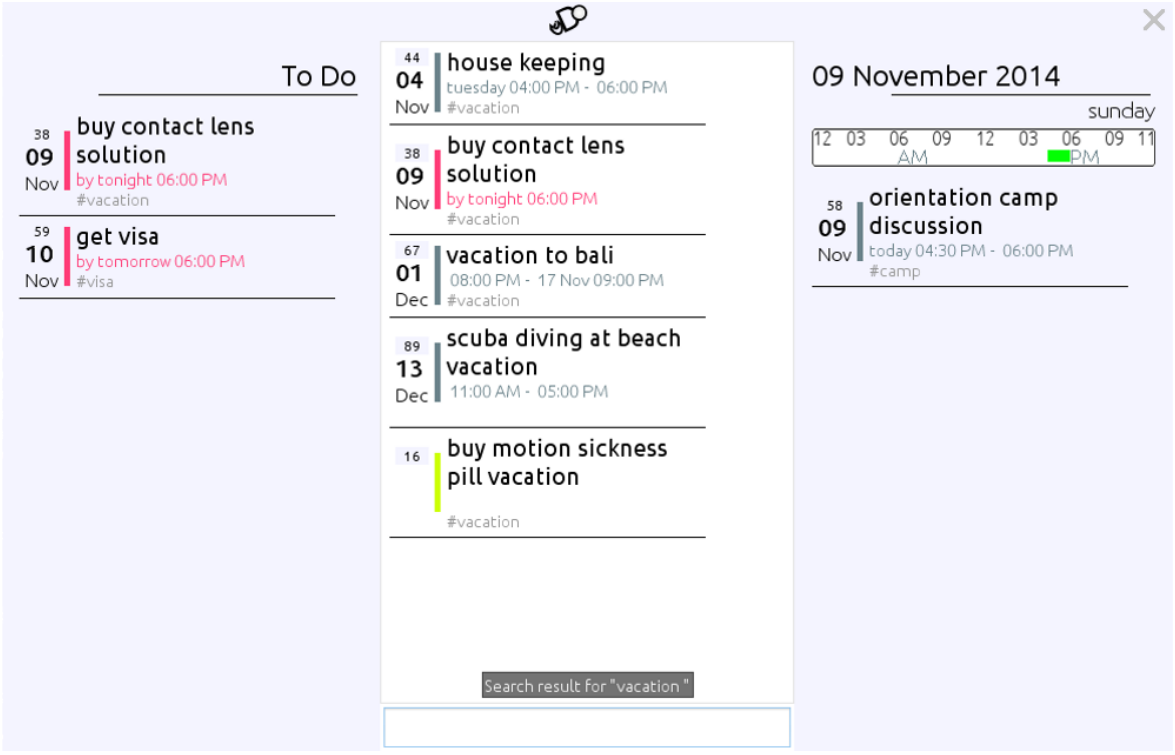
To search for activities, use the search command followed by the search keywords. You can enter multiple keywords after the search command. The search result will only return activities that contain all the entered keywords.

For example, to search for vacation items, enter:

search vacation

Note!
Search function also searches for hashtags

uDo will display the following:



3.4. Undoing previous actions

To cancel your command, you can use the undo command. The undo command only applies to the last command you input.

undo

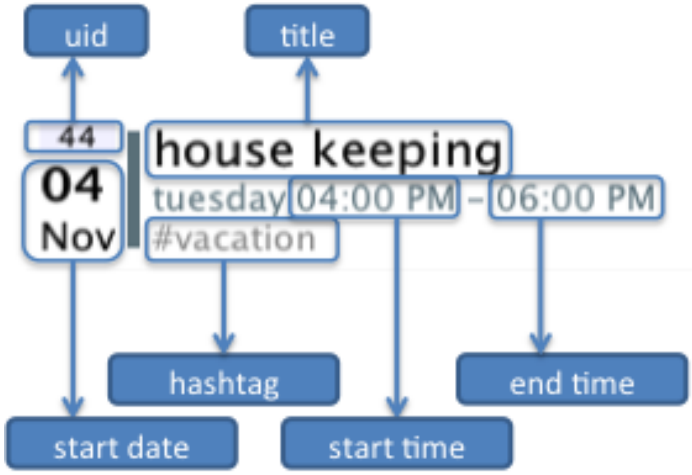
Note!
You can redo the last undo command by typing undo again

3.5. Editing

uDo allows you to edit your activities in these following fields:

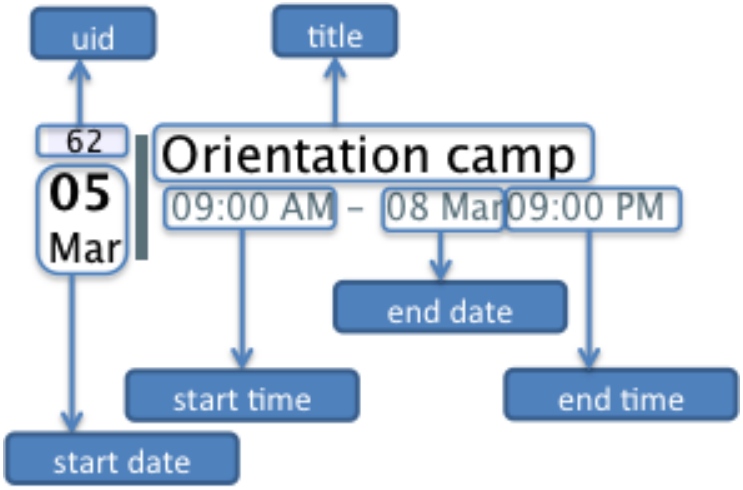
- title
- start time
- start date
- end time
- end date
- due time
- due date

The following is a one-day event activity. In a one-day event, the end date will not be shown in the item.



Note!
Though the end date is not shown in a one-day event, you may still edit the end date

The following is an event activity than spans more than one day.



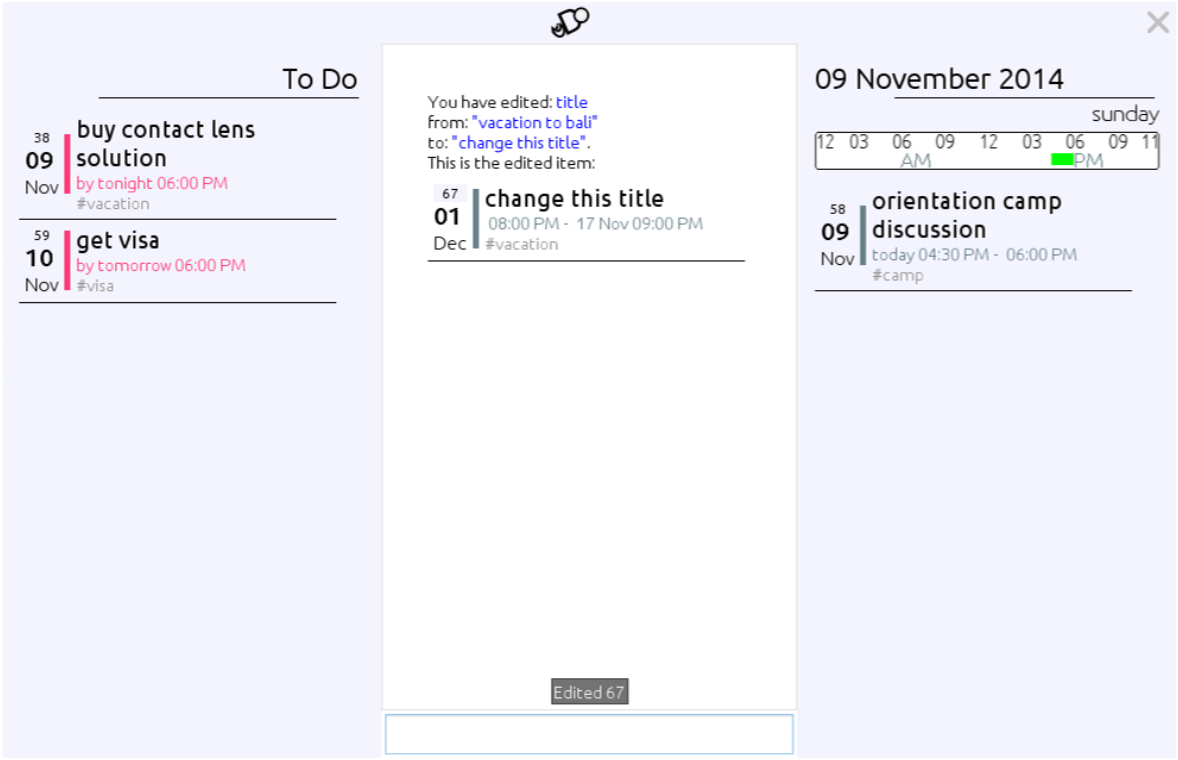
The following activity is a task.

You may refer to the summary for a better understanding on which field exists in which kind of activity.

To edit an activity's title, enter the edit command in the following sequence:

edit 67 title change this title

uDo will display the following:



To edit an event's starting time, enter the following sequence:

edit 29 start time 9:23am

To edit an event's starting date, enter the following sequence:

```
edit 28 start date tomorrow
```

To edit an event's ending time, enter the following sequence:

```
edit 80 end time 11:50pm
```

To edit an event's ending date, enter the following sequence:

```
edit 77 end date today
```

To edit a task's due time, enter the following sequence:

```
edit 23 due time 8am
```

To edit a task's due date, enter the following sequence:

```
edit 85 due date wednesday
```

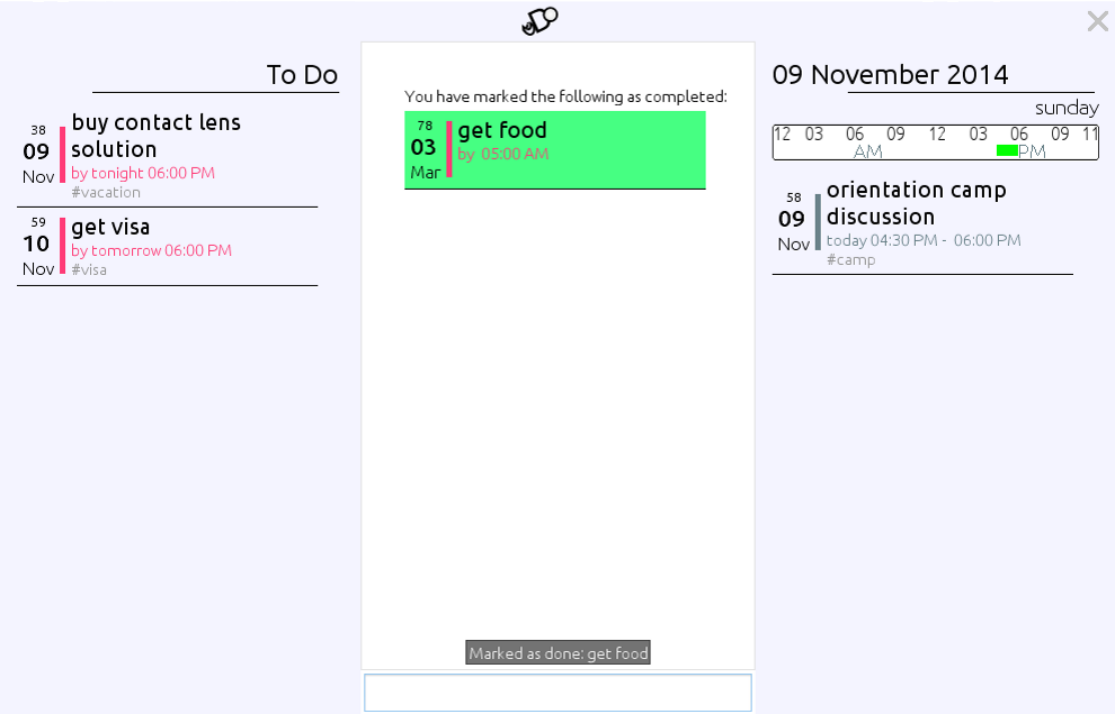
3.6. Marking as done

You may mark an activity as done by simply entering done, followed by the uid (refer to Field location summary). The activity marked done will be highlighted in green and will remain green until it is removed or marked "undone" (refer to toggle done command).

To mark an activity as done, enter:

```
done 78
```

uDo will display the following:



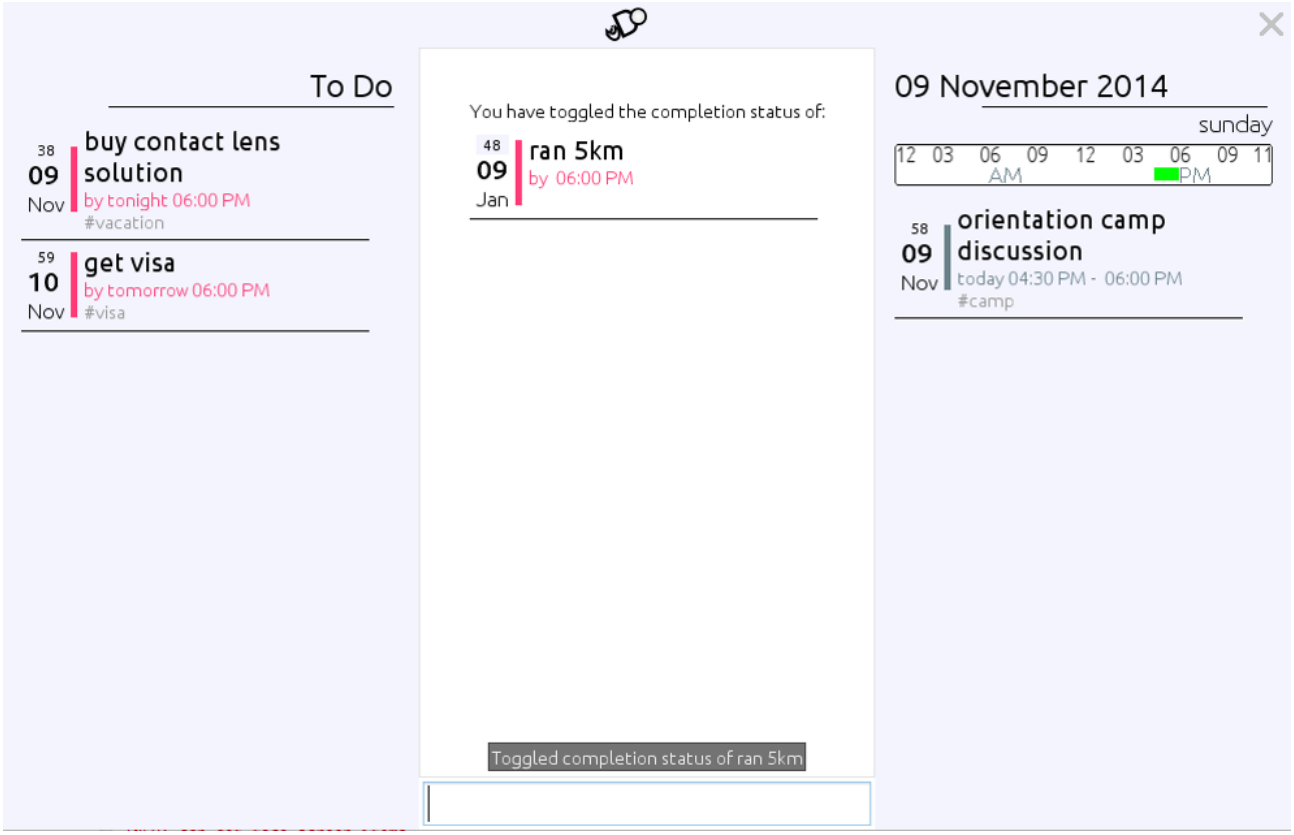
Note!
Only tasks and plans can be marked done. This command does not apply to events.

You may edit the “done” status by entering toggle done, followed by the uid (refer to Field location summary). A previously marked item will be unmarked after this command. After unmarking, the activities will return to its original grey color.

To do this, enter:

```
toggle done 48
```

uDo will display the following:



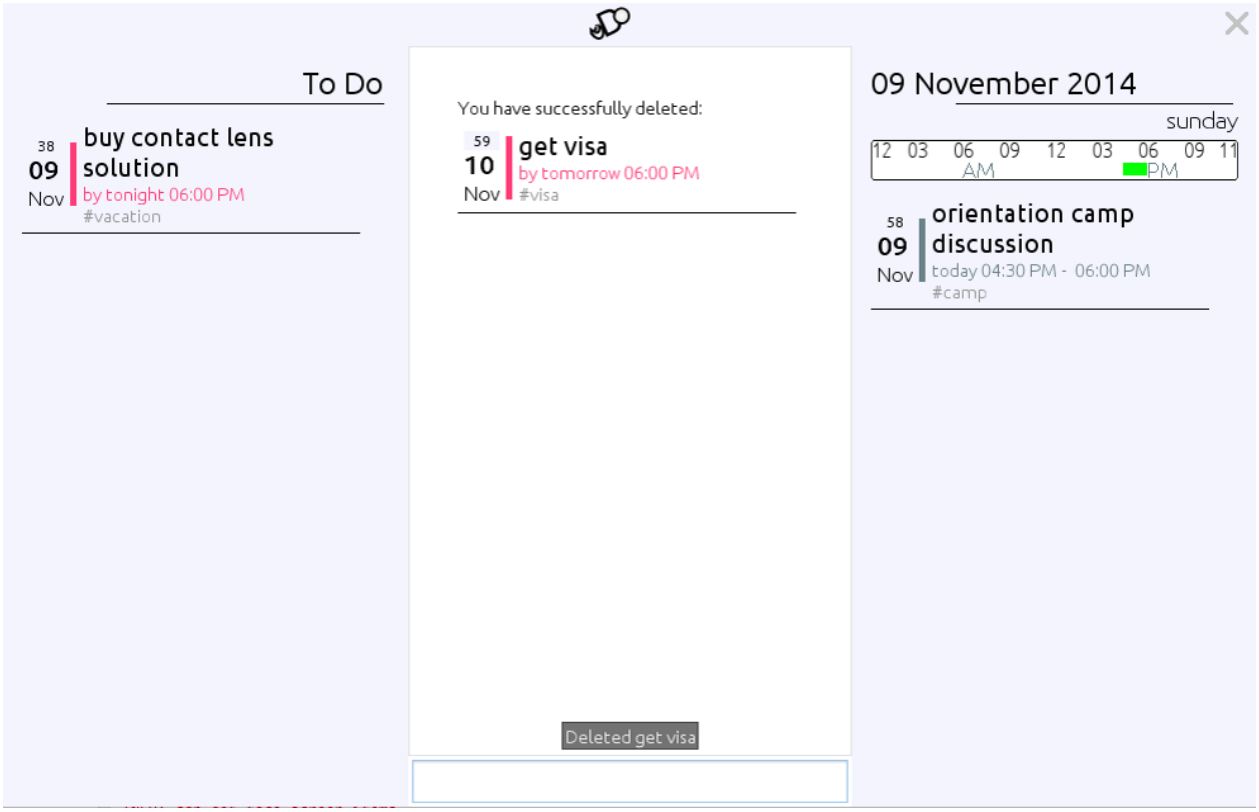
3.7. Deleting activities

You may delete an activity by using the command delete, followed by the uid (refer to Field location summary).

To delete an activity, enter:

```
delete 59
```

uDo will display the following:



3.8. More actions

uDo stores your activities in a backing data file. To save on power consumption, uDo only writes your activities to your hard disk when you tell it to.

To save your changes, type:

save

Additionally, uDo also allows you to exit the application without taking your hands off the keyboard.

To exit uDo, type:

exit

uDo will automatically save any changes you made before exiting.

4. Summary

4.1. Edit fields

The following table gives all the fields that can be edited for the different types of activities.

Edit fields	event	task	plan
title	✔	✔	✔
start time	✔		
start date	✔		
end time	✔		
end date	✔		
due time		✔	
due date		✔	

4.2. Date and time formats

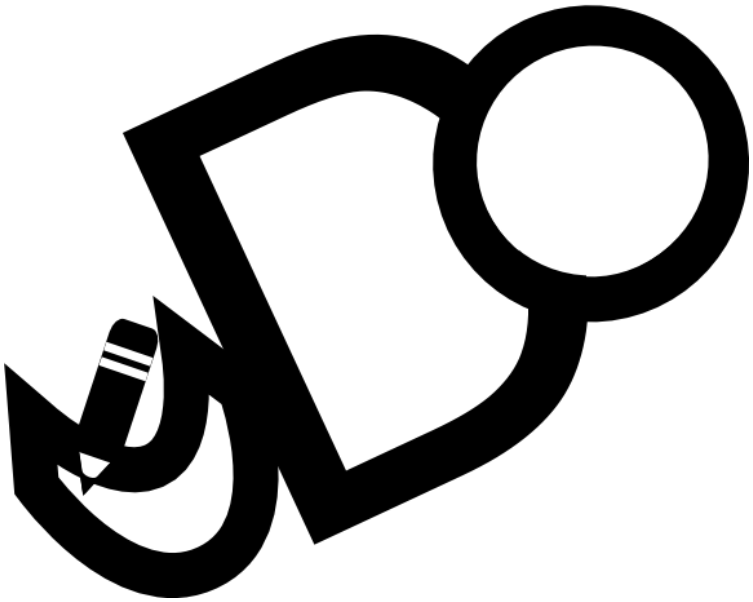
The following table lists the date and time formats that uDo accepts.

Day	Date	Time (12 hr clock)
today	dd/mm	hha
tomorrow	dd/mm/yy	hh:mma
sunday	dd/mm/yyyy	
monday		
tuesday		
wednesday		
thursday		
friday		
saturday		

4.3. Command Summary

The following table contains all possible commands with their command format for easy reference.

Command Type	Sentence Format	Example Input
Add Event (1 day event) (more than 1 day event)	Add <title-with-optional-hashtags> from <date> <start time> to <end time> Add <title-with-optional-hashtags> from <start date> <start time> to <end date> <end time>	Add sleep from Monday 1am to 11:11pm Add #orientation #camp from 9/10/14 7pm to 11/10 6pm
Add Task	Add <title-with-optional-hashtags> by <due date> <due time>	Add get #anniversary present for Jane by 16/10 5:15pm
Add Plan	Add <title-with-optional-hashtags>	Add make #friendship #bands
Mark Completed (for Task/ Plan)	Done <item-uid>	Done 12
Toggle Completed (for Task/ Plan)	Toggle done <item-uid>	Toggle done 73
Delete Item	Delete <item-uid>	Delete 45
Search Item	Search <content-to-search>	Search vacation
List All	List all	List all
List by hashtag	List <hashtag>	List #vacation
List by date or day	List <date OR day>	List 23/1 List friday
List Plans	List plan	List plan
List Tasks	List task	List task
List Events	List event	List event
Undo	undo	undo
Edit Fields	Edit <item-uid> <field> <new-info>	Edit 43 title meeting with boss Edit 89 start time 4:30am Edit 85 end date 12/2
Save	Save	save
Exit	Exit	exit



uDo Developer Guide

Table of Contents

1. Introduction

Welcome to uDo's development team!

As a new developer, you will find this guide useful in helping you get on board. Important background information will first be presented to you. This ensures that you are familiar with the development environment. The overall architecture will then be covered, followed by the internal structure of the main components. You will also be introduced to our testing methodologies. Finally, you will be given an example of extending uDo via language translations.

2. Background

uDo is a schedule manager application that helps to keep track of the activities that the user needs to do. All user interaction is done via the keyboard. This includes adding activities to uDo. The user's tasks and events are then shown on the Graphical User Interface (GUI).

It is important for you as you develop uDo to constantly keep the user's perspective in mind. Do also read uDo's User Guide to get more familiar with the desired user experience.

2.1. Design Principles

While you are developing for uDo, consider these three core design principles.

- **Convenience**: User interaction with uDo should be simple and fast. Ensure that users only need their keyboard to interact with the application.
- **Effectiveness**: Maintain a clean layout that is easy for the user to understand. Highlight important information to draw the user's attention.
- **Responsiveness**: Give dynamic and intuitive feedback to show that uDo has responded to the user's input and performed the action that the user intended.

2.2. Codebase and Standards

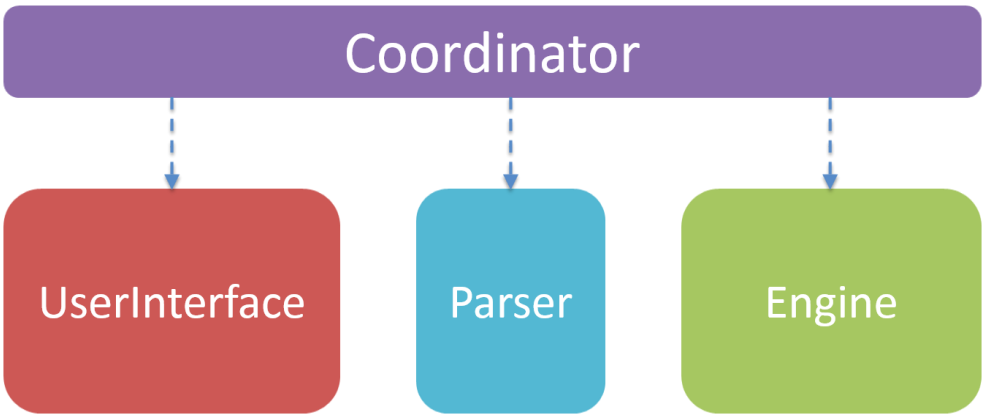
uDo is developed in the Java programming language. Testing is done with the JUnit framework. Our codebase is managed by the Git revision control software, and shared via GitHub.

To keep all the code in our codebase visually coherent, we adhere closely to an established coding standard. We also follow the *Collective Ownership* convention. All developers can improve and make changes to any part of the code. However to guard against regressions after a change, you are expected to write comprehensive test cases for your own code where applicable.

3. Architecture

3.1. Overall Architecture

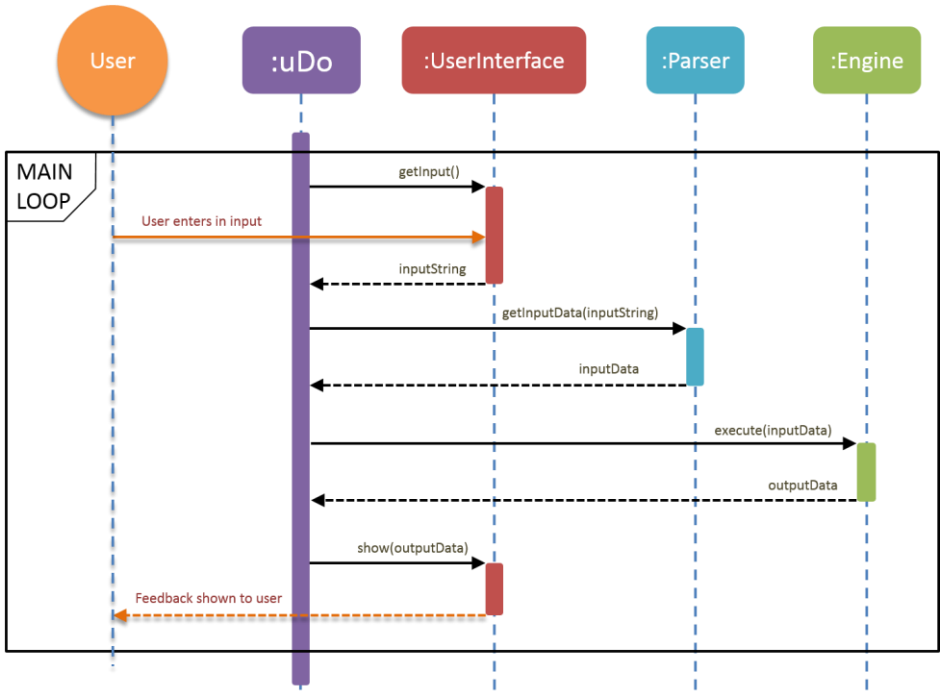
The architecture diagram below shows the relationship between the components.



The architecture design follows the *Law of Demeter*, where unrelated components neither communicate with nor have any knowledge of each other. You may refer to Section 4 for the detailed component descriptions.

3.2. Component Interaction

The sequence diagram below shows you how the components interact when responding to a generic input from the user.



Note that the sequence diagram shows the interactions between the representative classes for each component. As above, you may refer to Section 4 for the detailed component descriptions.

The steps performed are outlined as follows:

1. `uDo` calls *`getInput()`* from `UserInterface`, where it waits for the user's input
2. The user keys in an input string to GUI layer
3. The *`getInput()`* method returns the input string when the user presses Enter
4. `uDo` passes the input string to `Parser` for parsing
5. `Parser` returns an InputData object to `uDo`
6. `uDo` passes this InputData to `Engine` to execute
7. `Engine` returns an OutputData object to `uDo`
8. `uDo` passes the OutputData to `UserInterface`
9. `UserInterface` shows the result of the input execution on screen to the user

4. Components

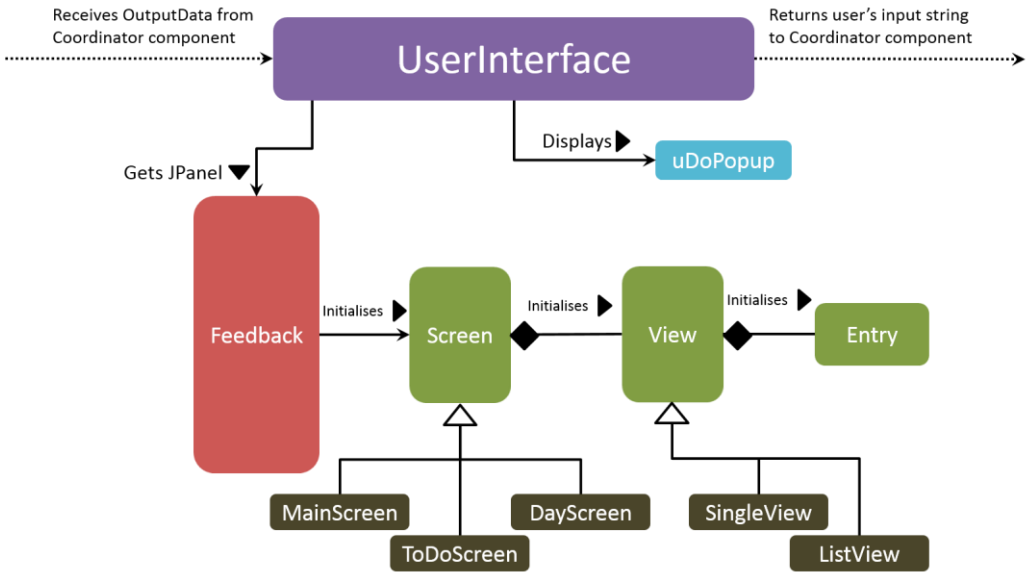
4.1. Coordinator

The Coordinator component comprises only one class, `uDo`. It manages the flow of data between the other components. `uDo` runs in an input-execute-display loop, as shown in Section 3.2.

Before starting a new loop iteration, `uDo` will check the OutputData returned from `Engine` for an **exit** command. `uDo` will stop the looping if it finds that `Engine` has executed the **exit** command successfully. The program will then close.

4.2. User Interface

The User Interface component is responsible for maintaining the GUI of uDo. The following diagram gives you an overview of its internal structure.



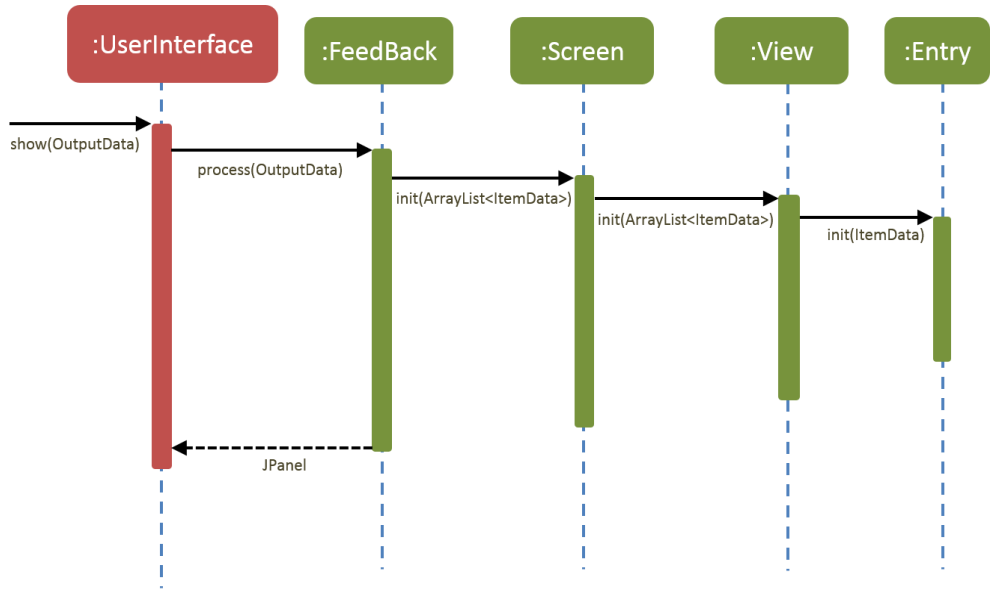
The development of the User Interface component requires at least a basic knowledge and familiarity with the Java Swing toolkit.

The User Interface component implements the *Singleton Pattern* for the class `UserInterface`. This ensures that there will be only one `UserInterface` object at all times.

`UserInterface` manages the returning of the user's input to the Coordinator and the displaying of any received OutputData.

To display the output, `Feedback` processes the given output and calls the necessary `Screen`. `Screen` then initialises as needed and calls the `View` to be displayed. `View` initialises, creates the `Entry` and displays it. Based on the result that `Feedback` processed, `UserInterface` will also call `uDoPopup` and display it accordingly.

The `Screen` (`DayScreen`, `ToDoScreen`, or `MainScreen`) and the `View` (`ListView` or `SingleView`) to be initialised depends on the attributes of the output sent to `UserInterface`. However, the overall sequence of operations are similar. The following sequence diagram shows the general logic sequence in UserInterface. The diagram below follows a **list** command which will show a list of entries in the main middle screen.

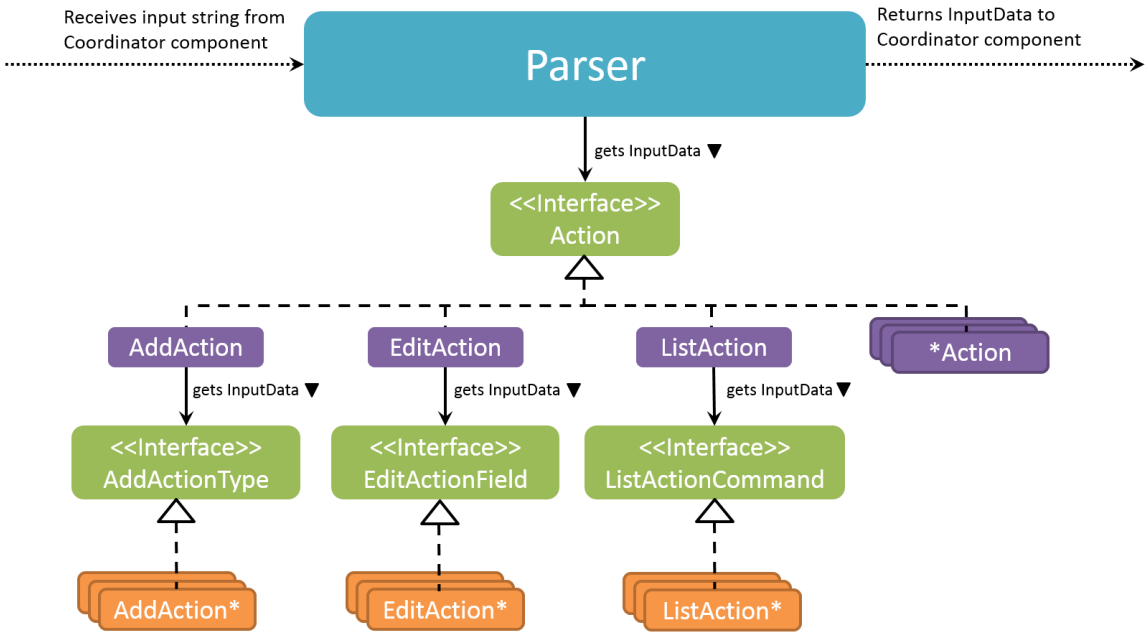


1. The *`show()`* method in `UserInterface` is called
2. `UserInterface` calls `Feedback` to process the OutputData
3. `Feedback` initialises a `MainScreen` with the list of ItemData that needs to be displayed
4. `MainScreen` initialises `ListView` to display the list of ItemData in
5. `ListView` creates an `Entry` for each item in the list
6. `Feedback` returns this `MainScreen` object, which contains the initialised `ListView` and `Entry` objects, as a JPanel to `UserInterface`
7. `UserInterface` renders the received JPanel on the GUI

4.3. Parser

Parser analyses a given user input and stores the information in an InputData object. This InputData is then returned to the Coordinator.

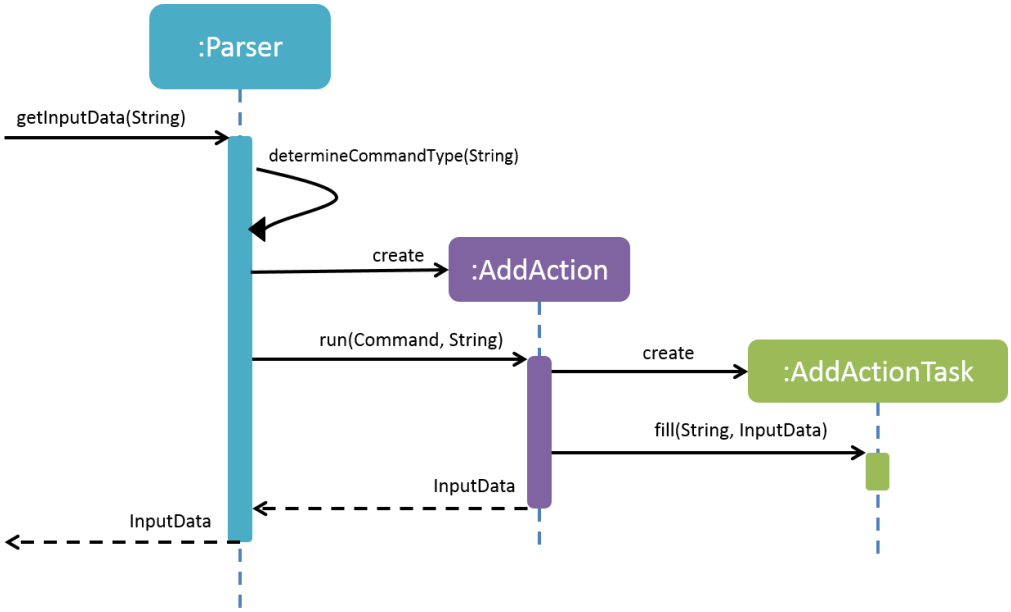
The class diagram for the Parser component is shown below.



The main design pattern implemented in the Parser component is the *Command Pattern*. The types `Action`, `AddActionType`, `EditActionField` and `ListActionCommand` are the *Commands* in this pattern.

The `Parser` class determines the command from the input string to create the appropriate `Action` class. `Action` then processes the user input. Additional command classes may be created for some `Action`s. For instance, an **add** command in the input string creates `AddAction`. `AddAction` then determines the item type to add. If the input string represents adding a task, `AddAction` creates `AddActionTask` to handle further processing of data from the input string. On the other hand, a **save** command in the input string results in only `SaveAction` being created.

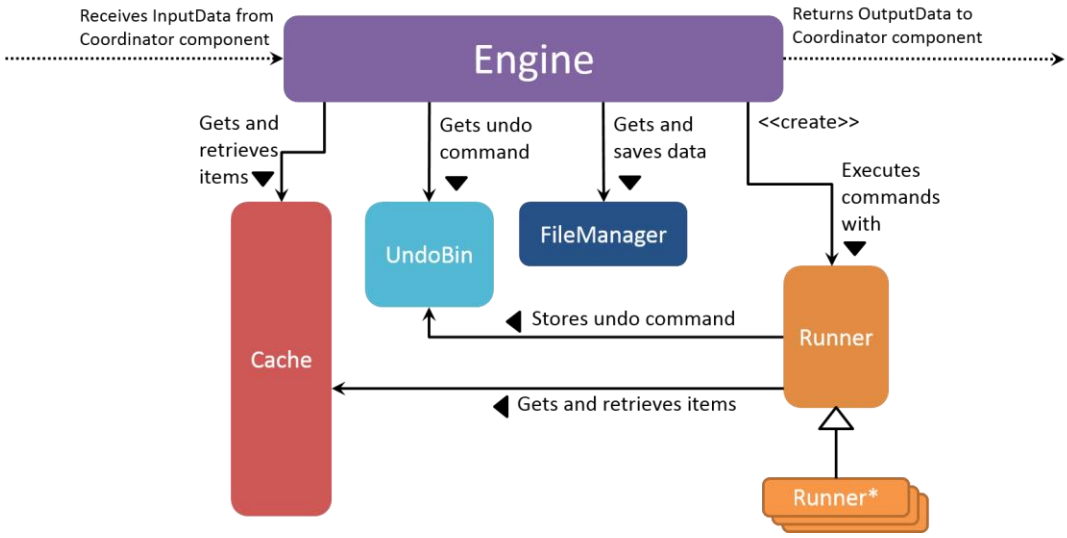
The interactions between components for **add**, **edit** and **list** commands are similar. The sequence diagram for adding a task is shown below.



1. `Parser` receives an input string from uDo
2. `Parser` determines the **add** command word and creates the `AddAction` object
3. `Parser` runs the command on `AddAction`
4. `AddAction` determines the item type added then creates an `AddActionTask` object
5. `AddAction` fills the InputData with `AddActionTask`
6. InputData from `AddAction` object is returned to `Parser`
7. `Parser` returns the InputData to uDo

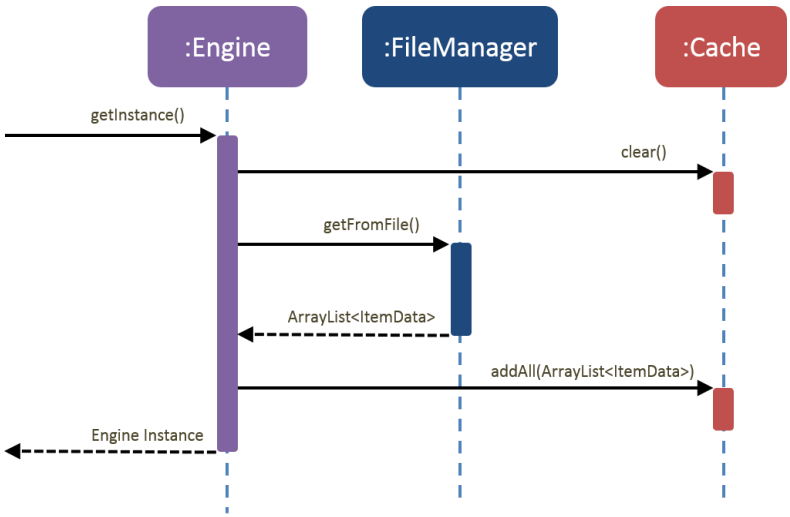
4.4. Engine

The Engine component executes commands. It also handles the storage and retrieval of application data. The diagram below shows how the classes in the Engine component are related to each other.



The *Façade Pattern* is implemented in this component. The `Engine` class is the *façade* class. `Engine` also implements the *Singleton Pattern*. This ensures that there is only one *façade* for the Engine component at all times.

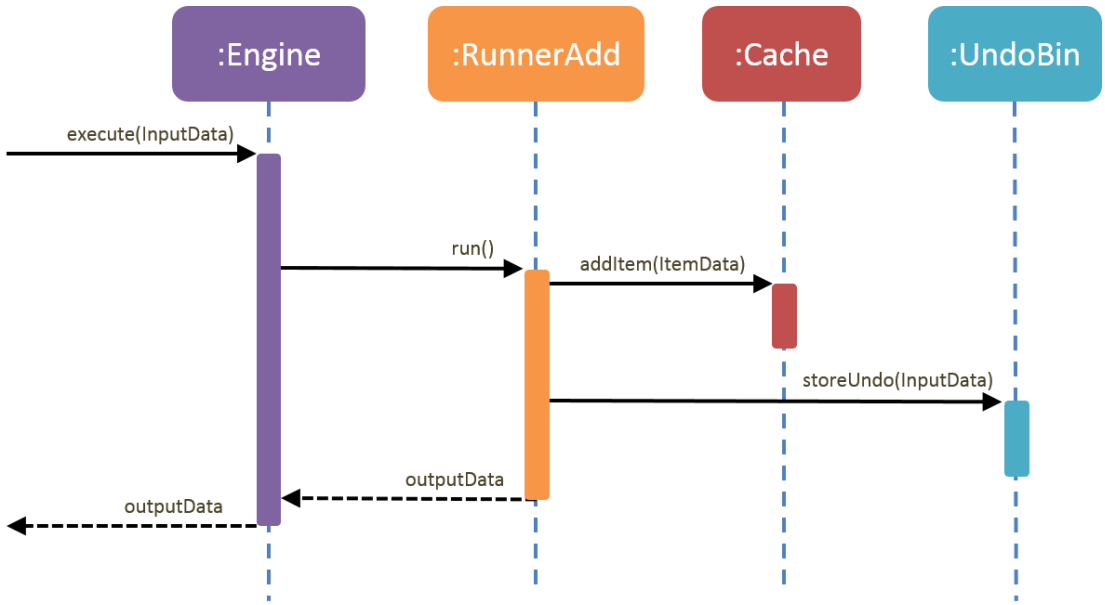
When uDo gets an instance of the `Engine` class, data is first loaded from the local file storage. The following sequence diagram shows the interaction between the internal classes during a *`getInstance()`* call to `Engine`.



1. `Engine` *`getInstance()`* is called
2. `Cache` is cleared before data is loaded
3. `FileManager` retrieves data from file and creates ItemData objects
4. `FileManager` returns the ItemData objects as an ArrayList
5. `Engine` adds the ItemData list to `Cache`
6. The `Engine` instance is then returned

User commands are executed by invoking a class of type `Runner`. Child classes of `Runner` are named after the command they are responsible for (e.g. `RunnerAdd`, `RunnerEdit`, etc.). `Engine` determines the command to execute, then instantiates the appropriate `Runner`.

The interactions between the `Runner` and other components are similar for most of the commands. The following sequence diagram describes a sample interaction between the classes for the **add** command, where the item added is an event.



1. `Engine` receives an InputData object containing an **add** command and item data
2. `Engine` creates the `RunnerAdd` object, which runs the **add** command
3. `RunnerAdd` creates the ItemData object then adds it to `Cache`
4. `RunnerAdd` constructs an InputData that represents the respective undo command, then stores it in `UndoBin`
5. `RunnerAdd` returns the OutputData to `Engine`
6. `Engine` returns the OutputData

The **undo** command does not have an associated `Runner`. `Engine` simply retrieves the InputData stored in the UndoBin and then calls *`execute()`* on it. The reverse command will hence be performed.

Similarly, the **exit** command also does not have an associated `Runner`. `Engine` creates `RunnerSave` and runs a **save** command with it before returning the OutputData. This OutputData still represents an **exit** command, but contains the execution status of the **save** command.

5. Instructions for Testing

As mentioned in Section 2.2, you are required to write test cases for your code where applicable.

We use the JUnit testing framework to conduct unit tests for our components. You may refer to **Appendix A: Using the JUnit Framework** for more information on how to set up and run JUnit test cases.

5.1 Testing the full application

The class `uDo` has a testing backdoor method: *`testParseAndExecute`*`(String)`. You should call this method on an input string that simulates an actual user's input. The method returns the OutputData after the input is parsed and executed. You should then compare the data in it with the expected values for the given input string.

5.2 Testing the UserInterface

The UserInterface component is tested via exploratory testing instead of scripted JUnit tests. Once a new feature is up or a bug is fixed, test it by launching the program and issuing input that will trigger the feature under test. You then compare the output on the GUI with the expected output.

5.3 Testing the Parser

Scripted testing is used for the Parser component. You should first specify an input string then pass it into the method *`getInputData`*`(String)` of `Parser`. An InputData object will be returned. You should then compare the data in this object with your expected values.

5.4 Testing the Engine

Scripted testing is also used for the Engine component. The method *`execute`*`(InputData)` of `Engine` should be called on an InputData object constructed by you. The data in the returned OutputData object should then be compared with the expected values.

6. Extending uDo: Language

uDo has the capability to have a language extension with a little code addition. If you are keen on translating uDo, follow the steps below to make uDo run in your desired language. This guide assumes that you already have uDo's code, which you can get from GitHub.

1. Under the *udo.language* package, create a new class and name it anything you like (for example, `MyLanguagePack`). Make sure it starts with capital letter.
2. Copy the contents of *EnglishLanguagePack.java* to your new language class. Don't forget to rename the public class `EnglishLanguagePack` to your class name.
3. Change the value to all variables to your desired language. Take care not to add on to or remove any existing variables.
4. Go to *LanguagePack.java* and locate the line:

   ```
   public static final LanguagePack USER_LANGUAGE_PACK = new EnglishLanguagePack();
   ```

   Replace `EnglishLanguagePack()` with your new class.

Appendix A: Using the JUnit Framework

This guide assumes you are using the Eclipse IDE (Luna Release) with the built-in JUnit4 Testing Framework.

Writing a new JUnit Test Case

1. Decide on the class and the functionality that will be tested.
2. On the toolbar, select File, then New. Select "JUnit Test Case" from the dropdown menu.
3. A pop-up window titled "New JUnit Test Case" should appear, with various fields to fill up.
4. Give your new test case a descriptive name.
5. Ensure that the "Class under test" field is pointing to the class you chose earlier.
6. Leave the other fields at their default values. Click Finish.
7. A pop-up asking you to add the JUnit Library to the project's build path may appear. Click OK.
8. Observe the default test method and note how it is written.
    a. Test methods have the annotation `@Test` above the method body.
    b. Test methods are always `public void` and without arguments. This is by convention.
    c. Inside the method body, there should be at least one JUnit *`assert`* function. The default code uses the *`fail()`* function, which will always return a test failure.
9. Follow the guidelines to write your own test methods for the functionality you have previously selected.

As a reference, here is a code sample from an actual JUnit Test Case for uDo.

```
@Test
public void testEngineDelete() {
     // Setting up the objects under test
     Engine e = new Engine();
     InputData input = new InputData(Command.DELETE);
     input.setParsingStatus(ParsingStatus.SUCCESS);
     input.put(Keys.UID, TEST_UID);
     OutputData output = e.execute(input);

     assertFalse("Output must not be null",
               null == output);

     assertEquals("Execution must be successful",
               ExecutionStatus.SUCCESS,
               output.getExecutionStatus());
}
```

Running an existing JUnit Test Case

1. Check the side views for a JUnit tab. If it exists, bring it to focus and skip the next step.
    a. You have to open the JUnit tab if you do not have it on the side views.
    b. On the toolbar, navigate to Window -> Show View -> Other. A pop-up dialog should appear.
    c. Look for the Java folder and expand it. Select JUnit, then click OK.
2. Check your side views again for the JUnit tab.
3. Ensure that you have the source code for the test case on the main view.
4. On the toolbar, select Run -> Run. You may also use the shortcut Ctrl-F11.
5. Observe the changes on the JUnit tab.
    a. A green bar means all the test methods have passed.
    b. A red bar means that one or more test methods have failed.
    c. If the bar is red, a list of all the test methods will appear under it. Methods that threw test failures or errors will be highlighted.
6. You may return to the class under test to rectify any test failures.
7. You may also double-check the test case itself to ensure that the test results are valid.
8. Repeat from Step 3 to re-run the test once changes are made.

Appendix B: Useful APIs for Data Structures

InputData

METHOD	RETURN TYPE
INPUTDATA(COMMAND)	[constructor]
INPUTDATA(COMMAND, PARSINGSTATUS)	[constructor]
GETCOMMAND()	Command
SETPARSINGSTATUS(PARSINGSTATUS)	void
GETSTATUS()	ParsingStatus
PUT(STRING, OBJECT)	void
GET(STRING)	Object
CONTAINS(STRING)	boolean

OutputData

METHOD	RETURN TYPE
OUTPUTDATA(COMMAND)	[constructor]
OUTPUTDATA(COMMAND, PARSINGSTATUS)	[constructor]
OUTPUTDATA(COMMAND, PARSINGSTATUS, EXECUTIONSTATUS)	[constructor]
SETPARSINGSTATUS(PARSINGSTATUS)	void
SETEXECUTIONSTATUS(EXECUTIONSTATUS)	void
GETCOMMAND()	Command
GETPARSINGSTATUS()	ParsingStatus
GETEXECUTIONSTATUS()	ExecutionStatus
PUT(STRING, OBJECT)	void
GET(STRING)	Object
CONTAINS(STRING)	boolean

ItemData

METHOD	RETURN TYPE
ITEMDATA(ITEMTYPE)	[constructor]
PUT(STRING, OBJECT)	void
GET(STRING)	Object
CONTAINS(STRING)	boolean

Appendix C: Useful APIs for Component Classes

uDo

METHOD	RETURN TYPE
UDO()	[constructor]
STATIC MAIN(STRING[])	void
TESTPARSEANDEXECUTE(STRING)	OutputData

UserInterface

METHOD	RETURN TYPE
STATIC GETINSTANCE()	UserInterface
GETINPUT()	String
SHOW(OUTPUTDATA)	Void
UPDATETODAYSCREEN(ARRAYLIST<ITEMDATA>)	Void
UPDATETODOSCREEN(ARRAYLIST<ITEMDATA>)	Void

uDoPopup

METHOD	RETURN TYPE
UDOPOPUP()	[constructor]
GETALPHA()	float
SETALPHA()	void

Feedback

METHOD	RETURN TYPE
FEEDBACK()	[constructor]
INITTODAYVIEW(ARRAYLIST<ITEMDATA>)	JPanel
INITTODOVIEW(ARRAYLIST<ITEMDATA>)	JPanel
PROCESS(OUTPUTDATA)	void
GETCOMMAND()	String
GETFINALVIEW()	JPanel

SingleView

METHOD	RETURN TYPE
SINGLEVIEW()	[constructor]

INIT(ITEMDATA, STRING)	void

ListView

METHOD	RETURN TYPE
LISTVIEW()	[constructor]
INIT(ARRAYLIST<ITEMDATA>)	void

Screen

METHOD	RETURN TYPE
SCREEN(INT, INT)	[constructor]
INIT(DATE, ARRAYLIST<ITEMDATA>)	void
GETSCROLLPANE()	JScrollPane
REMOVEALL()	void

Entry

METHOD	RETURN TYPE
ENTRY(ITEMDATA, STRING)	[constructor]

Parser

METHOD	RETURN TYPE
PARSER()	[constructor]
GETINPUTDATA(STRING)	InputData

Action

METHOD	RETURN TYPE
RUN(COMMAND)	InputData
RUN(COMMAND, STRING)	InputData

AddActionType

METHOD	RETURN TYPE
FILL(STRING, INPUTDATA)	void
GETTAGS(STRING)	ArrayList<String>
GETTITLE(STRING)	String

EditActionField

METHOD	RETURN TYPE
FILL(STRING, INPUTDATA)	void

ListActionCommand

METHOD	RETURN TYPE
FILL(STRING, INPUTDATA)	void

DateGetter

METHOD	RETURN TYPE
DATEGETTER()	[constructor]
GETDATE(STRING)	Calendar

TimeGetter

METHOD	RETURN TYPE
TIMEGETTER()	[constructor]
GETTIME(STRING)	Calendar

Engine

METHOD	RETURN TYPE
GETINSTANCE()	Engine
EXECUTE(INPUTDATA)	OutputData
GETTODAYSCREENITEMS(CALENDAR)	ArrayList<ItemData>
GETTODOSCREENITEMS(CALENDAR, CALENDAR)	ArrayList<ItemData>

Runner

METHOD	RETURN TYPE
RUN()	OutputData

Cache

METHOD	RETURN TYPE
CACHE()	[constructor]
ADDITEM(ITEMDATA)	void

ADDALL(ARRAYLIST<ITEMDATA>)	ArrayList<ItemData>
GETITEM(INT)	ItemData
GETALLITEMS()	ArrayList<ItemData>
SEARCHALLITEMS(STRING)	ArrayList<ItemData>
DELETEITEM(INT)	ItemData
SIZE()	int
CLEAR()	void
GENERATEUID()	int

FileManager

METHOD	**RETURN TYPE**
FILEMANAGER()	[constructor]
FILEMANAGER(STRING)	[constructor]
GETFROMFILE()	ArrayList<ItemData>
WRITETOFILE(ARRAYLIST<ITEMDATA>)	void

UndoBin

METHOD	**RETURN TYPE**
UNDOBIN()	[constructor]
STOREUNDO(INPUTDATA)	void
GETUNDO()	InputData
CLEAR()	void

LanguagePack

METHOD	**RETURN TYPE**
GETINSTANCE()	LanguagePack
GETUSERLANGUAGEPACK()	LanguagePack
SETLANGUAGE()	void