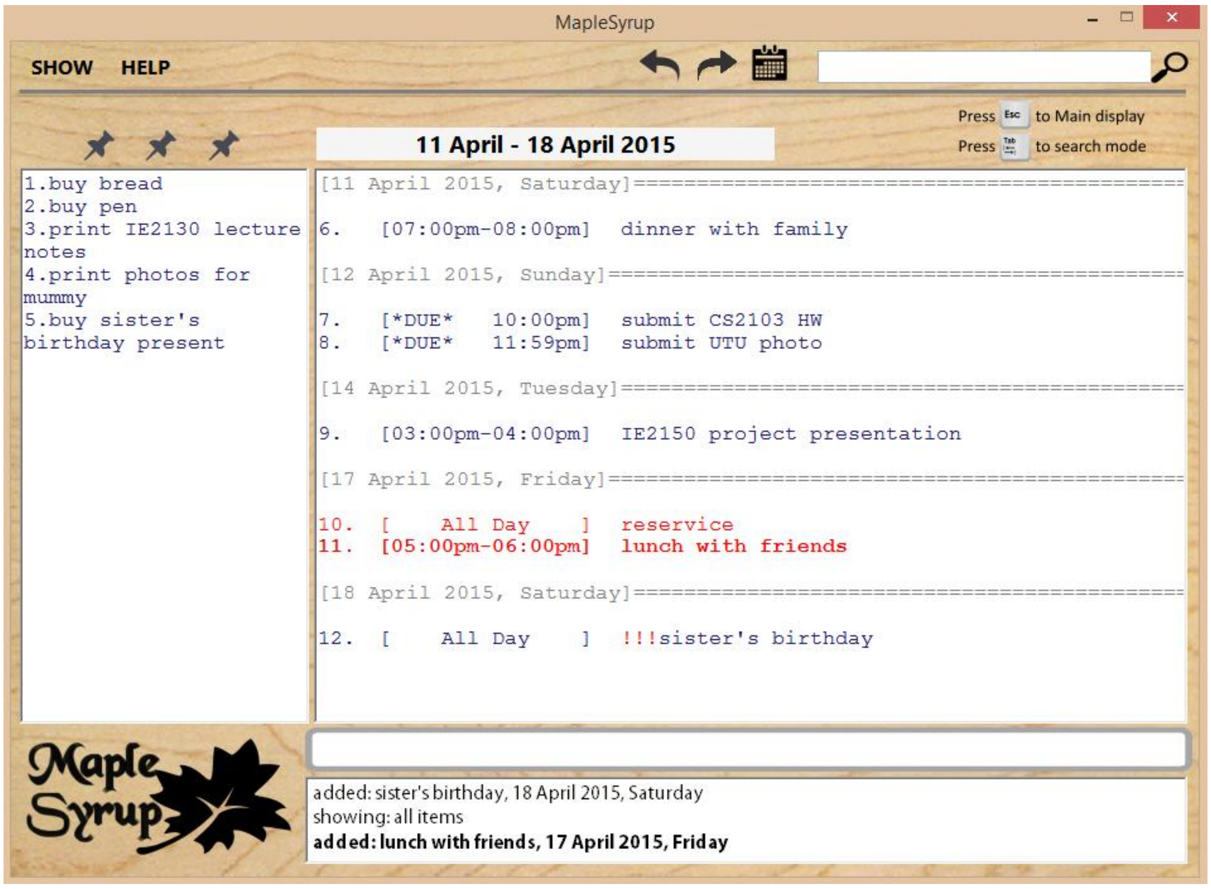


11 April - 18 April 2015

```
1.buy bread
2.buy pen
3.print IE2130 lecture notes
4.print photos for mummy
5.buy sister's birthday present
```

```
[11 April 2015, Saturday]==========================================

6.   [07:00pm-08:00pm]  dinner with family

[12 April 2015, Sunday]============================================

7.   [*DUE*    10:00pm]  submit CS2103 HW
8.   [*DUE*    11:59pm]  submit UTU photo

[14 April 2015, Tuesday]===========================================

9.   [03:00pm-04:00pm]  IE2150 project presentation

[17 April 2015, Friday]============================================

10.  [    All Day     ]  reservice
11.  [05:00pm-06:00pm]  lunch with friends

[18 April 2015, Saturday]==========================================

12.  [    All Day     ]  !!!sister's birthday
```

```
added: sister's birthday, 18 April 2015, Saturday
showing: all items
added: lunch with friends, 17 April 2015, Friday
```

Supervisor: Lim Yu De Extra feature: GoodGUI

Ong Wei Jee Team Lead Code Quality	George Lam Changwei Github Expert Documentation	Ter Yong Kang GUI Designer Deliverables and Deadlines	Che Jian Yong Joshua Integration Scheduling and tracking

Maple Syrup

User Manual

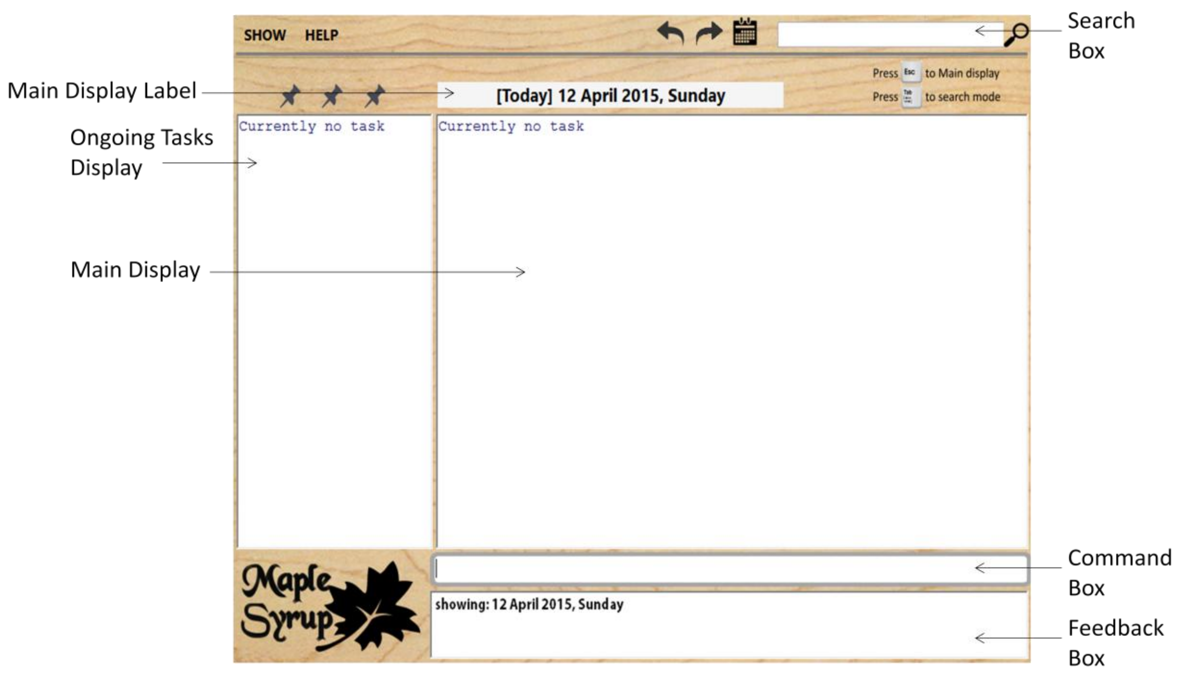
Introduction

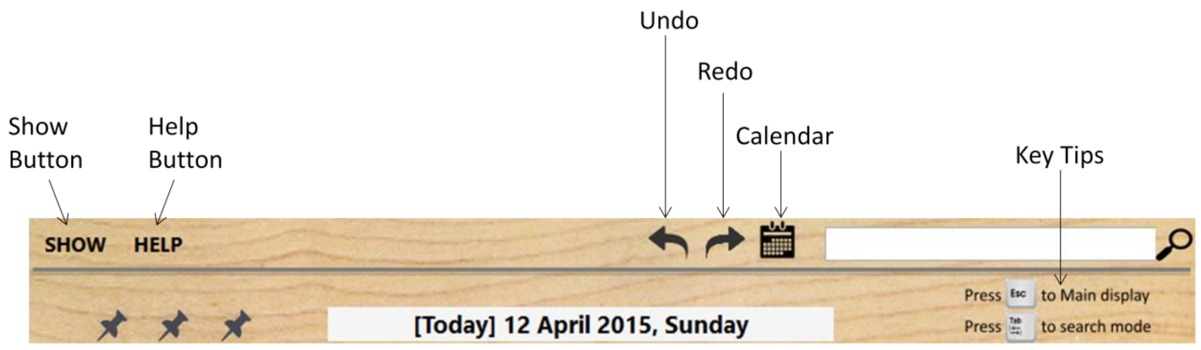
Welcome to MapleSyrup, your go to application to keep track of tasks and events! An event in MapleSyrup consists of several details: the name, date, time, and importance. This guide will walk you through the process of adding, deleting, editing, displaying events, and undoing and redoing your commands (because everybody makes mistakes).

Getting Started

Double click MapleSyrup on your desktop.

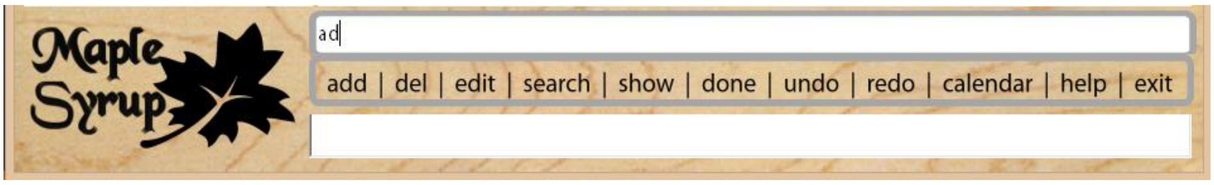
The main display will show all your tasks and events for today. On-going tasks will be displayed on the left panel.

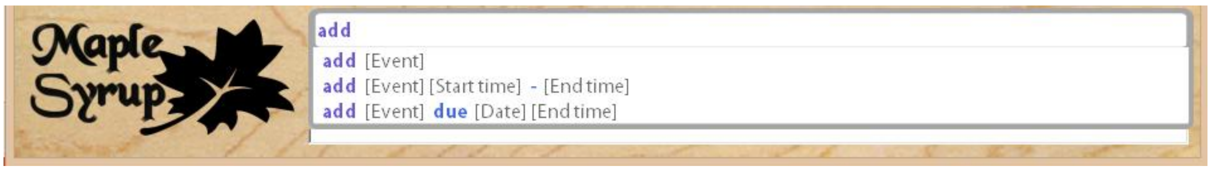




Command Mode

By default, Maplesyrup will be in the command mode. You will be able to see your cursor in the command box where you will be typing your commands. All the features in MapleSyrup are accessible from this mode. Here, you can add, delete, edit and show events. The tooltip guide below the command box will guide you in entering your commands. All keywords that you type will be highlighted in a different colour.


Maple Syrup
ad
add | del | edit | search | show | done | undo | redo | calendar | help | exit


Maple Syrup
add
add [Event]
add [Event] [Start time] - [End time]
add [Event] due [Date] [End time]

Navigate: To plan and direct the course of a ship, aircraft, or other form of transport, especially by using instruments... Wait, wrong one.
Navigate In Command Mode: Press TAB to enter the Search Mode. Press ESC to enter the Display Mode.

Search Mode
In search mode, your cursor will be in the search box at the top right of the window. You will be able to view your search results while typing in the search box.

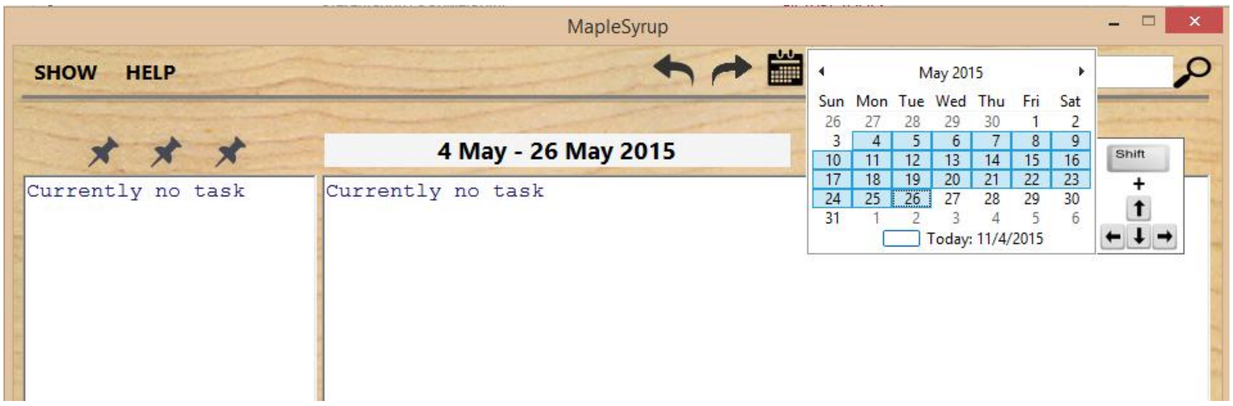
Navigate: Press TAB to enter the Command Mode. Press ESC to toggle between the ongoing-events display and the main display.

Display Mode
In display mode, no commands can be entered ☹. Pressing the left and right arrow keys will change the displayed range of dates. Pressing the up and down arrow keys will scroll through the displays ☺.

Navigate: Press TAB to enter the Command Mode. Press ESC to cycle between the ongoing events list to the normal events list.

Maple Tips

- NOT IMPORTANT: Maple syrup is usually made from the xylem sap of maple trees.
- IMPORTANT: Type "commands" or "shortcuts" to access... well what do YOU think they access?
- BREAKING NEWS: Press CTRL-D to call the calendar. Thereafter, use SHIFT + arrow keys to select the dates of your choice followed by ENTER to display events in your selected dates.



4 May - 26 May 2015

Currently no task

Currently no task

Sun	Mon	Tue	Wed	Thu	Fri	Sat
26	27	28	29	30	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	1	2	3	4	5	6

Functions

ADDING AN EVENT

Description:

Events (or tasks) can be added with the command **add**. An event must have a name, and can be assigned dates, from a starting date to an ending date; times, from a starting time to an ending time; and importance levels as indicated by ‘!’. Events can also be given a deadline.

General Format:

“add [event name] [importance] [date] [time]”

An event name is compulsory for all events and must be the FIRST word/words after the keyword add. Date, time, and importance are optional and can be omitted.

Types of Adding:

Ongoing Events

To add Ongoing Events, only an event name is needed.

“add drink maplesyrup”

Single Day Events

To add Single Day Event, add at least a date.

“add drink maplesyrup 1 jan”

“add drink maplesyrup 1 jan 3pm''

“add drink maplesyrup 1 jan 3pm - 4pm”

Multiple Day Events

To add Multiple Day Event, add a start date and end date.

“add drink maplesyrup 1 jan - 31 dec” (but don't forget your vegetables)

“add drink maplesyrup 1 jan 1am - 31 dec 11pm”

Deadline Events

Adding Deadline events requires a due or by keyword. Deadline events can only accept 1 date and 1 time input!

“add drink maplesyrup due 5pm”

“add drink maplesyrup by 31 dec 11pm”

Important Events

Events can also be assigned an importance level. There are 3 levels of importance (!, !!, !!!). Simply enter ! after your event name.

"add drink maplesyrup ! 1 jan"

"add drink maplesyrup by 31 jan 2pm !!!"

Recognisable forms of inputs:

Identifier:

- - / to

Date:

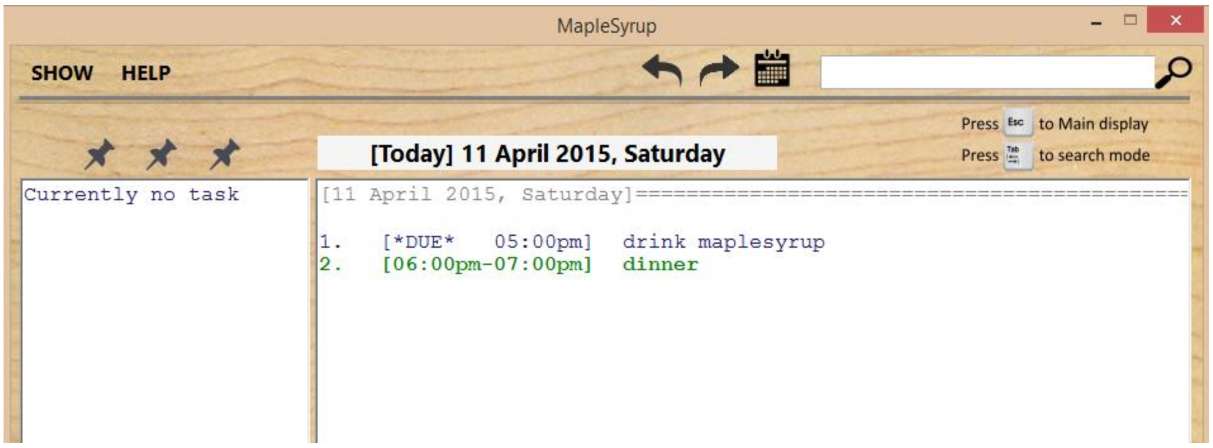
- 14apr / 14april / 14 apr
- today / tdy / tomorrow / tmr / mon / tues / wed / thurs / fri / sat / sun
- next mon / next tues /…
- jan / feb / ...

Time:

- 1am / 1 am / 1.30 am / 1.30am
- 1-3pm / 1pm-3pm / 1 pm to 3 pm

Display:

Newly added events are displayed in bold **green** and all other events displayed in blue.

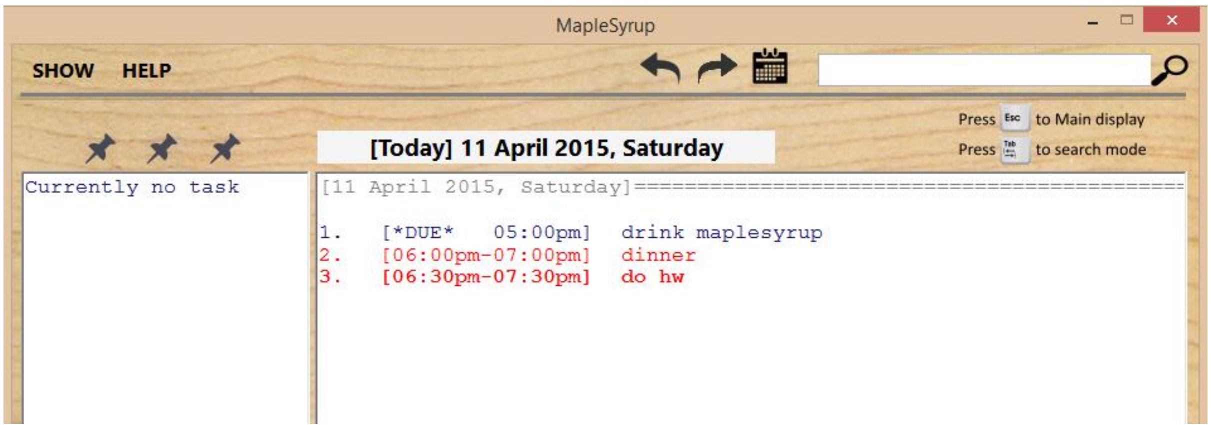


[Today] 11 April 2015, Saturday

```
Currently no task
```

```
[11 April 2015, Saturday]============================================

1.    [*DUE*    05:00pm]   drink maplesyrup
2.    [06:00pm-07:00pm]   dinner
```

In the case of a clash in timings between the new event and existing events, all clashing events will be highlighted in red to alert you, with the new event in **bold**.



[Today] 11 April 2015, Saturday

```
Currently no task
```

```
[11 April 2015, Saturday]============================================

1.    [*DUE*    05:00pm]   drink maplesyrup
2.    [06:00pm-07:00pm]   dinner
3.    [06:30pm-07:30pm]   do hw
```

Maple Tips:

- NOT IMPORTANT: Maple syrup is graded according to its density and translucency. Yummy!
- SOMEWHAT IMPORTANT: You can add events without even typing the keyword add.
- IMPORTANT: If you'd like a clearer way to differentiate the event name from the date, type ';' after the event name.
    - "drink maplesyrup**;** by tmr 2pm"

DELETING AN EVENT

Description:

Events can be deleted by typing the keyword **delete** or **del** followed by the event name or its index, which is displayed on the left of the event name.

General Format:

"del [event name/index]"

You can delete an event by name even if the event is not displayed. However, you can only delete an event by index if the index is displayed. Don't be difficult.

Types of Deleting:

By Name

If there is exactly one result found, that event will be deleted immediately. If there are partial matches or many results with the same name, they will all be displayed and you can take your pick. However, if the name you type is not found, nothing will happen. Comprehensive, right?

"del drink maplesyrup"

By Index

If the index is found in the display, that event will be deleted immediately. Multiple day events may be represented by more than one index even though they refer to the same event. In this case, deleting any of the corresponding indices will work.

"del 1"

Display:

Deleted events will be removed from the main display immediately and disappear from existence forever until the end of time. Unless you use undo. Then it will come back (amazing). The display will show all the dates where the deleted event used to be.

COMPLETING AN EVENT

Description:

Events can be completed by typing the keyword **complete** or **done** followed by the event name or its index. All other procedures are similar to delete (see above). After any subsequent commands are executed this completed event will be sent to the completed event storage, which can only be accessed by typing "show done" (more on this later).

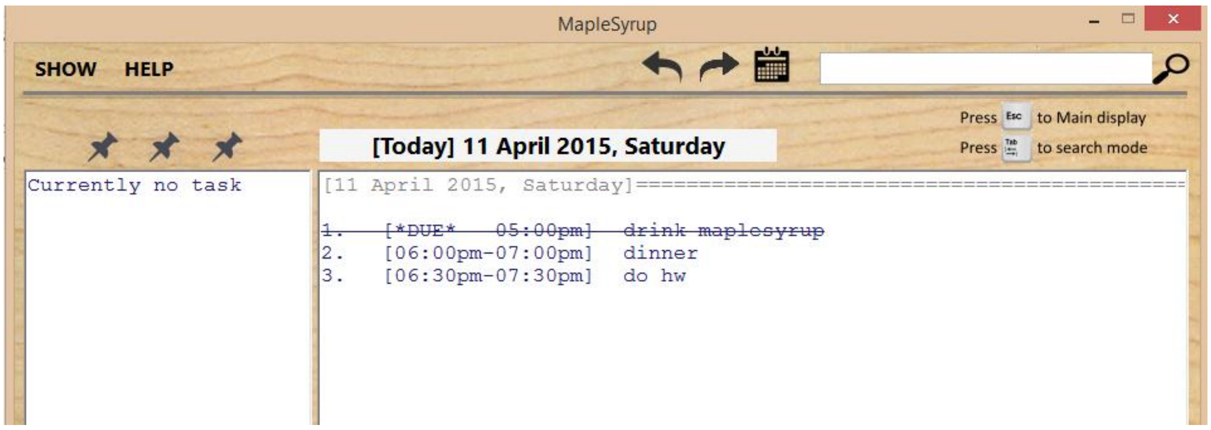
General Format:

"done [event name/index]"

"completed [event name/index]"

Display:

When you complete an event, it will be struck out in the display, just like how you would cross out your task on your notebook. Except that you don't have a notebook because it is the 21st century.



```
Currently no task
```

```
[11 April 2015, Saturday]============================================

~~1.    [*DUE*    05:00pm]   drink maplesyrup~~
2.    [06:00pm-07:00pm]   dinner
3.    [06:30pm-07:30pm]   do hw
```

Maple Tips:

- IMPORTANT IN CANADA: In Canada, maple syrup must have a density of 66° on the Brix scale. We do not know what this means.
- IMPORTANT: Once an event is completed, you can only view it using the show done command (think Recycle Bin).

- IMPORTANT: Completed events can ONLY be uncompleted (see next section). Deleting and editing them will not work.
- IMPORTANT: The object oriented way to get wealthy is inheritance.

UNCOMPLETING AN EVENT

Description:

Events can be uncompleted by typing the keyword **uncomplete** or **undone** followed by the event name or its index. All other procedures are similar to the procedures for delete and complete (see above). If successfully uncompleted, the event will be shown normally on the display.

General Format:

"undone [event name/index]"
"uncomplete [event name/index]"

Maple Tips:

- IMPORTANT: **Uncomplete** will only work when the completed event is currently shown on the display.
- IMPORTANT: Remember to check your feedback box after every command.
- VERY IMPORTANT: Maple syrup goes well with pancakes. But have you tried it with waffles?

EDITING AN EVENT

Description:

All details of an event: name, date, time, importance; can be edited by using the command **edit**.

General Format:

“edit [event name/index]; [new event name] [new importance] [new date] [new time]”

The event name or index of the event to be edited is compulsory. After the event name or index, you MUST type ‘**;**’ before adding the new details for the event .You must. These new details should only include those that you wish to be edited. All other details that are omitted will be left untouched.

Types of Editing:

<u>Normal Editing (both normal and ongoing events)</u>

Only type details that you wish to edit.

“edit drink maplesyrup; buy maplesyrup”

“edit drink maplesyrup; buy maplesyrup !!! 5may 7-8pm”

<u>Normal Event to Ongoing Event</u>

To remove the time and date of a normal event (i.e. convert to ongoing event), simply omit all new details.

“edit drink maplesyrup;”

<u>Ongoing Event to Normal Event</u>

Ongoing events can be edited to normal events if new dates or new timings are specified.

“edit drink maplesyrup; 5may 7-8pm

Display:

The edited event will be displayed in green, while all other existing events will be in blue. If this newly edited event clashes with other existing events, all clashing events will be highlighted in red. I mean red. We have mentioned this before. Pay attention.

SHOWING EVENTS

Description:

If you want to find specific types of events (e.g. within a time period, by importance level, completed events etc), you are going to NEED and LOVE the **show** command. When you use a **show** command, the display will be updated to what you specifically specify in the specification of your specified input.

General Format:

"show [dates to show]"

"show [importance]"

"show done"

Types of Show:

Single Day

"show 24apr"

"show tmr"

Multiple Days

"show 22-23apr"

"show week"

Month

"show month"

"show april"

"show apr-may"

Year

"show year"

"show 2015-2016"

Importance

"show !!!"

"show important"

"show impt"

Completed

"show done"

"show complete"

"show completed"

<u>All</u>

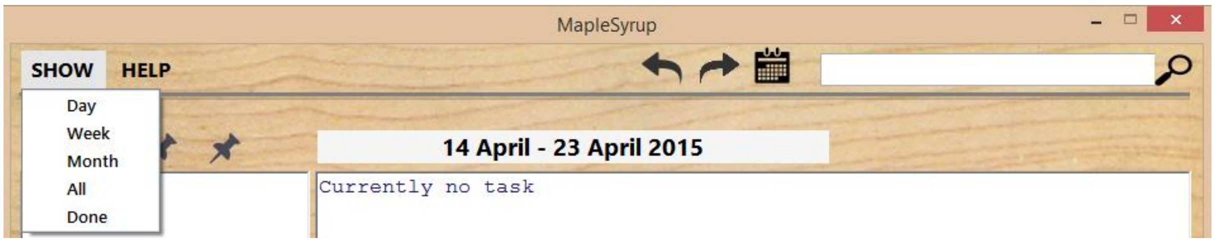
"show all"

<u>Tentative</u>

"show me the money"

Using the Mouse:

First get your hands on a mouse. No, not from the pet shop. Click the "show" button at the top left of the window (click the mouse, not the window). A drop down menu will appear with "day", "week", "month", "all" and "done". Click your desired choice. If you desire to stop using our program, click the cross at the top right hand corner and throw this user guide away. Also, you can click the calendar icon beside the search bar. A small calendar will appear. From there, click on any date or drag across dates to show on the display.

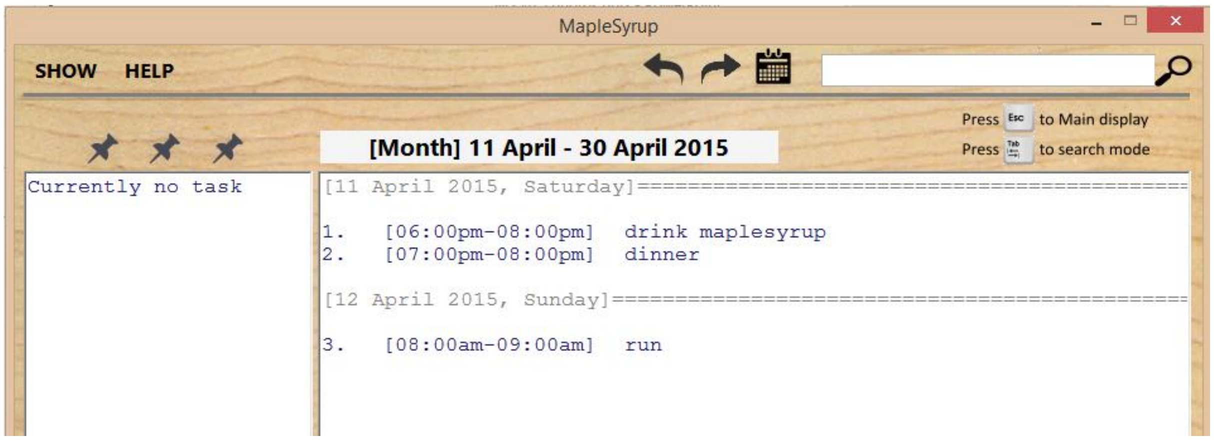


14 April - 23 April 2015

```
Currently no task
```

Display:

The main display label will be updated to reflect the dates that you wish to view. If you used a show week/month, the display will be tagged with [week] or [month] and it will stick (like maple syrup) to the week/month mode until you exit it.



[Month] 11 April - 30 April 2015

```
Currently no task
```

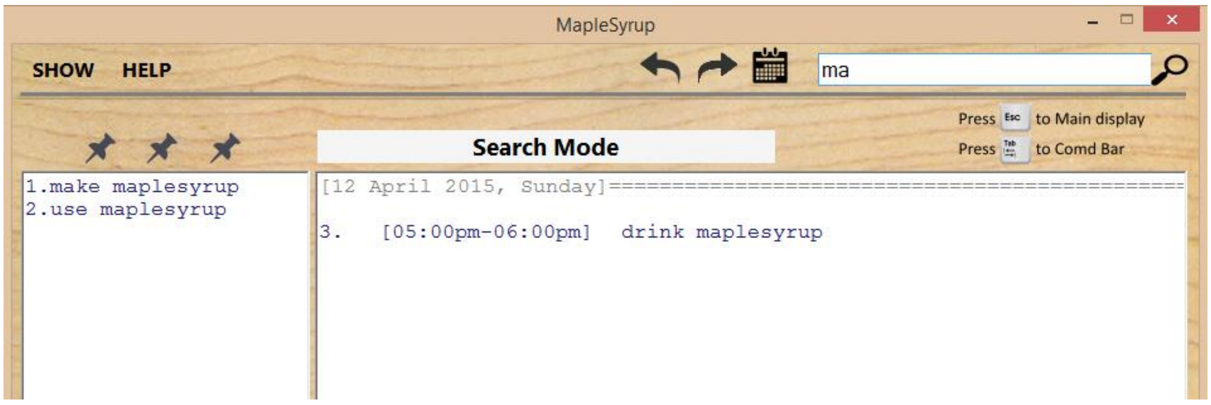
```
[11 April 2015, Saturday]==========================================

1.    [06:00pm-08:00pm]   drink maplesyrup
2.    [07:00pm-08:00pm]   dinner

[12 April 2015, Sunday]============================================

3.    [08:00am-09:00am]   run
```

SEARCHING FOR EVENTS

Description:

The **search** bar is located at the top right of the window and it can be accessed by pressing TAB when you are in the command box (or just click in the search bar, you rebel). The **search** results will update every time a new character is typed into the search bar.

Press TAB to go back to the command line after you are done searching. We hope you find what you are looking for.



```
1.make maplesyrup
2.use maplesyrup
```

```
[12 April 2015, Sunday]==============================================

3.   [05:00pm-06:00pm]   drink maplesyrup
```

UNDO & REDO

Description:

Made a mistake? Fear not. MapleSyrup is armed with an **undo** and **redo** command that allows you to **undo** all your previous commands until the point when you started up the program. If you happen to be too enthusiastic in using our incredible **undo** feature (you eager beaver), you can always **redo** all the commands until you execute a new command.

Here are the commands you can **undo** and **redo** (you mistake-maker): **add**, **delete**, **complete**, **uncomplete**, **edit**.

Types of Undo & Redo:

Just kidding! No more types! Just type "undo" or "redo"!

Feeling adventurous? Venture to the top of the window and click the undo or redo buttons (you explorer). They really work! Don't ask us how!

Maple Tips:

- IMPORTANT: Do you know the commonly used keyboard shortcuts for undo and redo? Good for you! Unfortunately only half of them work in MapleSyrup! No but seriously, CTRL-Z works, use CTRL-X to redo.
- IMPORTANT: MAPLESYRUPMAPLESYRUPMAPLESYRUPMAPLESYRUP

HELP

Description:

For lost souls.

General Format:

"help"

Maple Tips:

- Stop reading this already.

Appendix A: User stories. As a user, …

[Likely]

ID	I can … (i.e. Functionality)	so that I … (i.e. Value)
addFloating	add a task by specifying a task description only	can record tasks that I want to do some day
addLocation	Add location to an event	Will know where I need to be for that event
setRecurring	Set events to be recurring every day/week/month	Do not have to replicate similar events that occur every day/week/month
anyFormat	Enter new task in any format	Do not have to follow strictly to a certain format
setCategories	Organize my events/tasks by categories through tagging	Can search for any event/task easily
addAllDay	Add all day events	Do not have to specify time period spanning the whole day
setImportance	Rank my task by importance	Can sort tasks by importance and perform more important task first
viewAll	View all my task/events in an entire calendar	Can see how busy the week/month is going to be
setAbbreviations	Type in commands in the form of abbreviations	Save time by reducing amount of typing
setKeyboardCtrl	Open/close the software with keyboard commands	Have a quicker and simpler way of opening and closing the software
viewFreeSlots	View all free slots available	Better

setAutoSave	Save my inputs after every command	No loss of information if software accidentally closes
viewCompleted	View all completed tasks	Have a sense of accomplishment
reportClashing	Notified if there is any clashing events/tasks	Reschedule events/tasks to not have any clash
allowEdit	Edit my task information	Change any information at any time

[Unlikely]

ID	I can … (i.e. Functionality)	so that I … (i.e. Value)
addAlarm	Alarm/reminder when event is coming up	Will not miss important tasks
autoReschedule	Let the software auto reschedule my task if I fail to complete it during the set time	Do not need to set a new deadline again
setBufferTime	Set buffer time between events at different locations	Do not have a situation where I do not have time to travel from one event to the other
setAutoStart	Have the software auto start up when I turn on my computer	Can use the software immediately and be reminded of today’s event right away

Appendix B: Product survey

Product: Google Calendar Documented by: Ter Yong Kang

Strengths:
1. Event colour - allow user to classify events based on colour. This enhance the user experience.
2. Same event can be repeated more than 1 time.
3. Find time function - helps user to find a free time for the event to be added
4. Search function - allows user to search for events according to its details (name etc. etc.)
5. Display density allows user to choose the style of displaying calendar - enhances user experience
6. Displays only the important details of the event (Name & Time) on the summary page. Upon clicking the event, the full details of the event are being displayed. This provides a quick overview for the user without information overloading he/she.
7. A "help" section to provide a quick guide to new users on how to use the program

Weaknesses:
1. Unable to help user prioritise which event is most important given they overlaps
2. Unable to prioritise to user what event/activity to be complicated first
3. Does not allow user to add "floating tasks" without entering date and time
4. Unable to categorise events of the same type (e.g. work, family) together (except through colour)
5. Only able to display in Day Week Month 4Day. Should have more display options such as "display event according to priority" or "events of same type" etc.

Product: Macbook Calendar Documented by: George Lam Changwei

Strengths:
1. Able to add new events easily with detailed information such as location, day and time
2. Able to set alarms and reminder for to do events

3. Able to add invitees, notes, URL, attachments 4. Able to add recurring events and set the recurrence to weekly or monthly or yearly. 5. Able to tag an item by color tags to sort by different categories. 6. Able to sync to Iphone 7. Has a small calendar at the side which help to navigate through the different days and months easily. 8. Has a short cut input bar at the top where users can just describe their event in a sentence and it will be added to the day accordingly. So users do not need to find the add and add the event. Weaknesses: 1. Only on mac platform 2. Can only sync to apple products 3. Unable to recognise shortforms for users that prefers to type fast 4. Show a lot of information at once which might be overwhelming. Especially to a new user.
Product: Wunderlist Documented by: Che Jian Yong Joshua Strengths: 1. Able to set due date, and has options for recurring events daily, weekly, monthly, yearly, and even custom (every 2 days, or every 5 days) 2. Able to add additional details to the event 3. Able to star events to label as high priority. 4. Able to search for events based on event title as well as additional info typed on the event 5. Has sorting availability by due dates, and priority. 6. Has automatic grouping of events, by today and week. 7. Allows attaching of file to the event, such as emails or documents that contain additional information on the current event can be linked to 8. Has shortcuts available for quick addition, starring, completing, selecting 9. Keeps records of completed tasks for reference in the future if need be

Weaknesses:
1. Does not allow copying and pasting of events. Not able to set multiple due dates that does not follow a standard recurring structure.
2. Does not allow previewing of all the events in the form of a calendar for easier viewing. (Such as seeing how busy a certain week is)
3. Front page of website does not show all the events. Unable to set Today tab as the first tab viewed.
4. Events for the week cannot be sorted, fixed by due dates.

Product: Sony Calendar Documented by: Ong Wei Jee

Strengths:
1. Can view in day/week/month
2. Option to input location and detailed event description
3. Can specific exact time for events
4. Can set repetition for events

Weaknesses:
1. Setting time by scrolling through numbers is not efficient
2. Reminders only occur 10 minutes before the event
3. Must navigate to respective dates to view events, and it is not always obvious in the GUI

MapleSyrup Developer Guide

Table of Contents:

1. Introduction to *MapleSyrup* Developer Guide

Welcome to the *MapleSyrup* Developer Guide! This guide is intended to familiarise developers and maintainers with the design and implementation of *MapleSyrup,* and assist with program testing.

1.1 About *MapleSyrup*

MapleSyrup is a desktop task manager aimed at individuals comfortable with keyboard-based commands for rapid data entry and retrieval. It will appeal to users who are familiar with command-line style of calling and dismissing programs, with the addition of a simple but powerful GUI for clearer data organization.

The basic functionality of *MapleSyrup* is as follows:

- Adding, editing, completing and deleting of events.
- Setting deadlines and importance levels for events.
- Searching for events by date, completion and importance.

1.3 *MapleSyrup* development

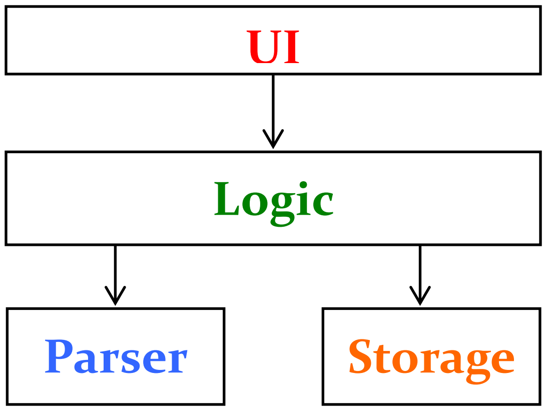
MapleSyrup is written in C++ programming language using Visual Studio 2012 (Professional). Hence, developers should be familiar with this programming language and developer software.

1.4 Development Environment

The recommended execution environment for the default build of *MapleSyrup* is Windows 7 or newer. *MapleSyrup* itself does not require installation and may be run straight from the executable binary.

2. Overview of Architecture

MapleSyrup utilizes a transaction processing architecture style with 4 components. They are the UI, which is in charge of all user displays and interactions; Logic, which is responsible for the internal processing of commands; Parser, which is responsible for translating user input into understandable program commands; and Storage, which handles the reading and writing of the internal data to savefiles. This is illustrated in the overall architecture diagram below.



2.1 Patterns and Principles

MapleSyrup is designed using the separation of concerns principle to achieve high cohesion. Lower levels have no knowledge of the higher level classes that depend upon their functionality, and each component has well-defined functionality with no functional overlap with each other.

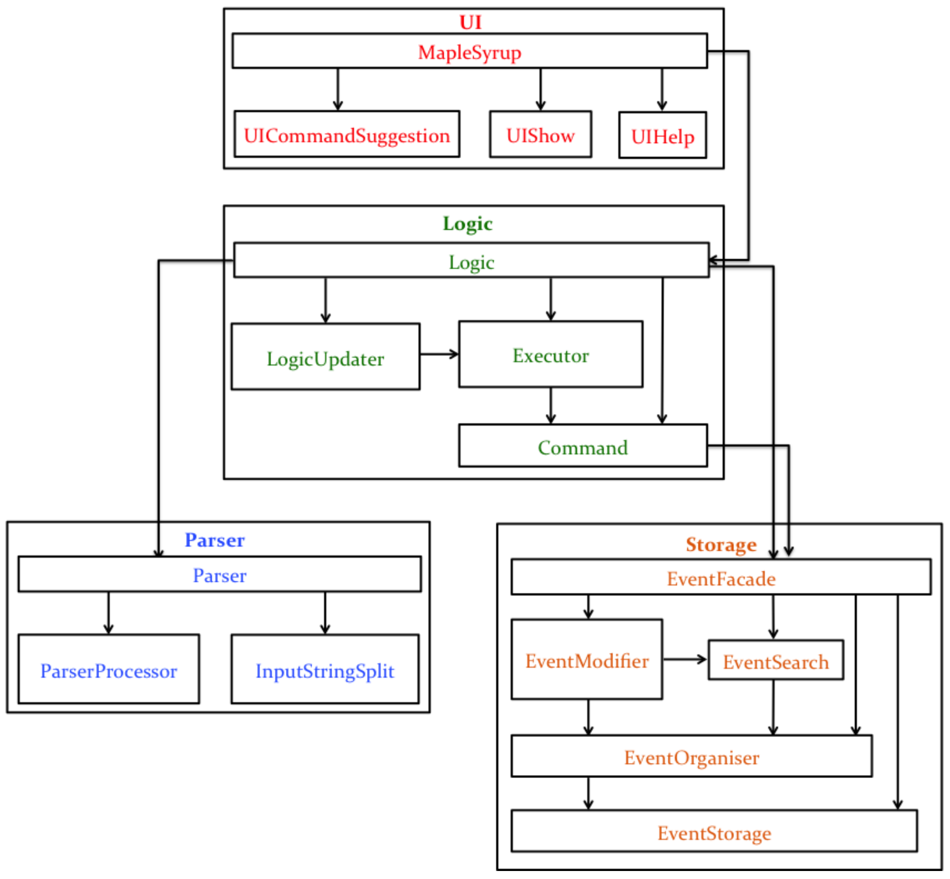
In implementing a command pattern, *MapleSyrup* also subscribes to the open-closed principle. The functionality of the program can be extended by implementing more commands to extend the virtual Command class, but without changing the source code.

The Model-View-Controller (MVC) pattern is also applied as follows:

1. View : Displays & feedbacks (UI)
2. Controller: Respective event handler (UI)
3. Model: back-end components (Logic, Parser, Storage)

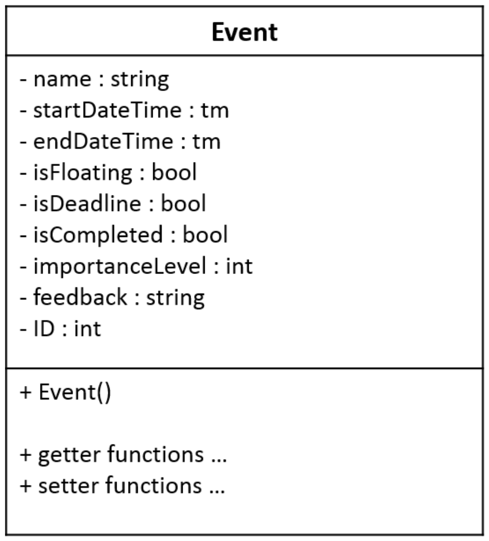
The use of this pattern reduces coupling resulting from the interlinked nature of the features described above by separating them into the 4 different components aforementioned.

The detailed architecture diagram is shown below.



2.2 What Is An Event?

The fundamental unit of information that is exchanged in our program is the Event object; all tasks/events entered by the user are saved as such. The Event object contains the event name, the start date and time, and end date and time, as well as its status (floating, deadline, completed, importance). Information about the user's events/tasks may be passed between components using this Event object.



Event

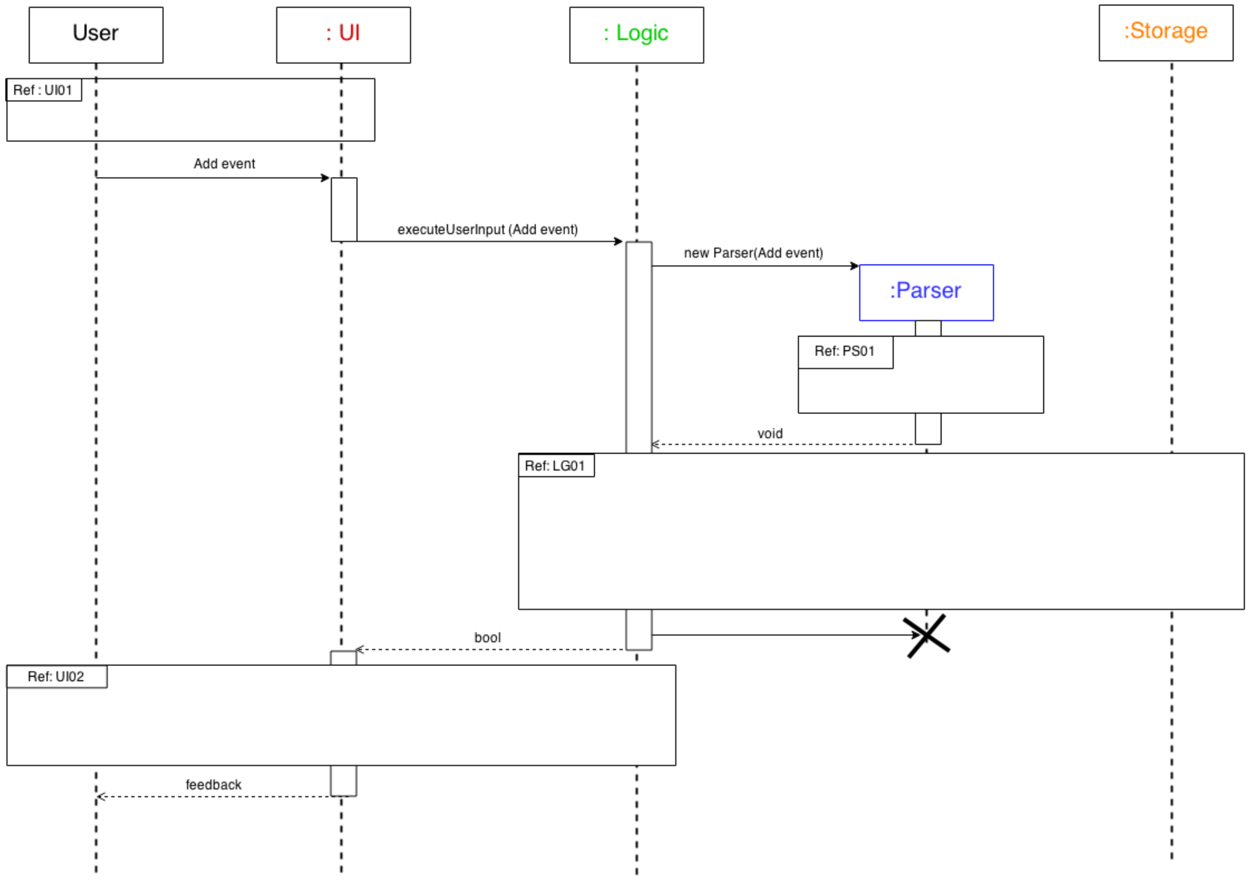
- name : string
- startDateTime : tm
- endDateTime : tm
- isFloating : bool
- isDeadline : bool
- isCompleted : bool
- importanceLevel : int
- feedback : string
- ID : int

+ Event()

+ getter functions …
+ setter functions …

3. Overall Sequence Diagram - Adding Events

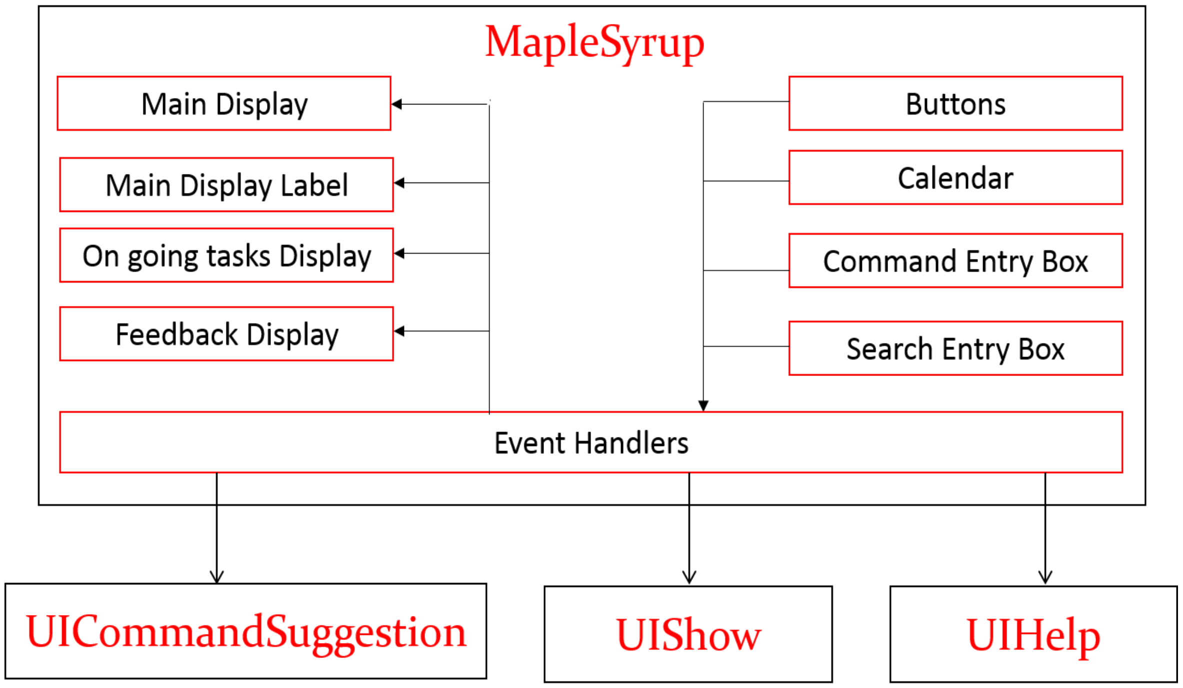
The sequence diagram for adding an Event is shown below to help facilitate the understanding of the basic interaction between and within each component. References can be found in the respective component section with more detailed descriptions showing how the internal components interact. The process begins when the user enters a valid add command. This command is accepted by the UI and immediately passed to Logic. Logic calls Parser to decipher the command and then creates the appropriate Command object. Logic interacts with Storage to save the Event and retrieve the necessary Events to display which are passed back to UI.



4. User Interface (UI)

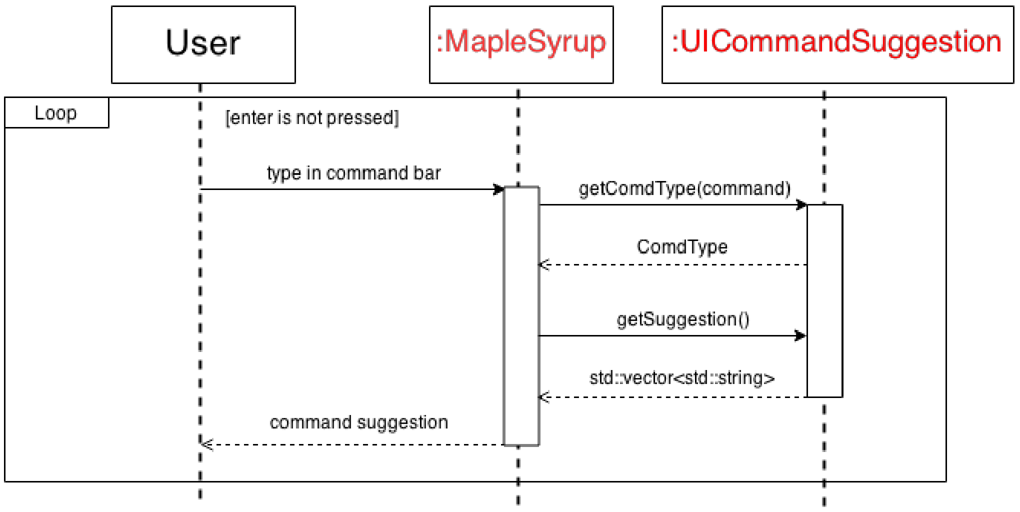
The UI is responsible for collecting all user input, converting internal data into a human readable format, and subsequently displaying it to the user. UI adopts a concept similar to that of facade when crafting its implementations. Inputs from users converge to only 1 point - function executeUserInput(). This function shields the back-end functions from the frontline components**.**

The UI architecture diagram with the intra and inter-classes dependencies is shown on the below. The important components of class MapleSyrup are also included.

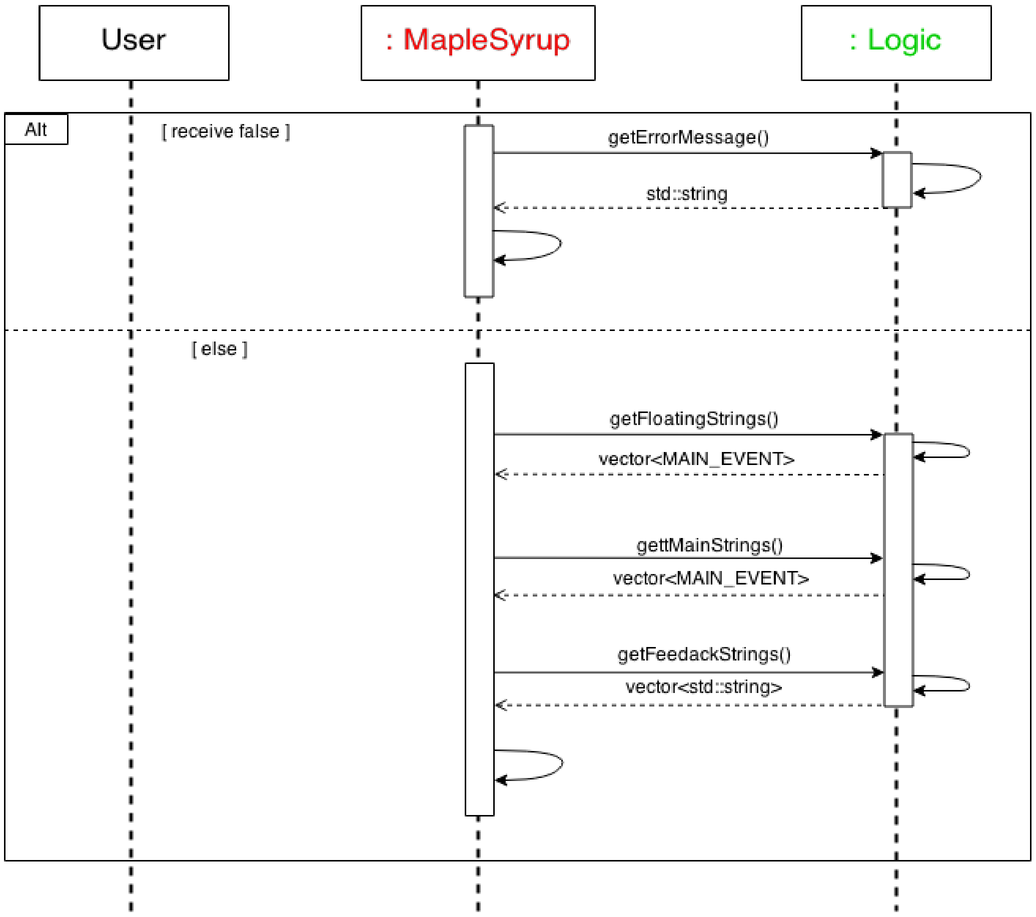


When a user action is detected, such as when a button is clicked, **Buttons** will be activated respectively to notify the respective **Event Handlers** of the user interaction. **Event Handlers** interact with UICommandSuggestion, UIShow and UIHelp within the UI component as well as with Logic. After which **Event Handlers** proceed to update the respective displays.

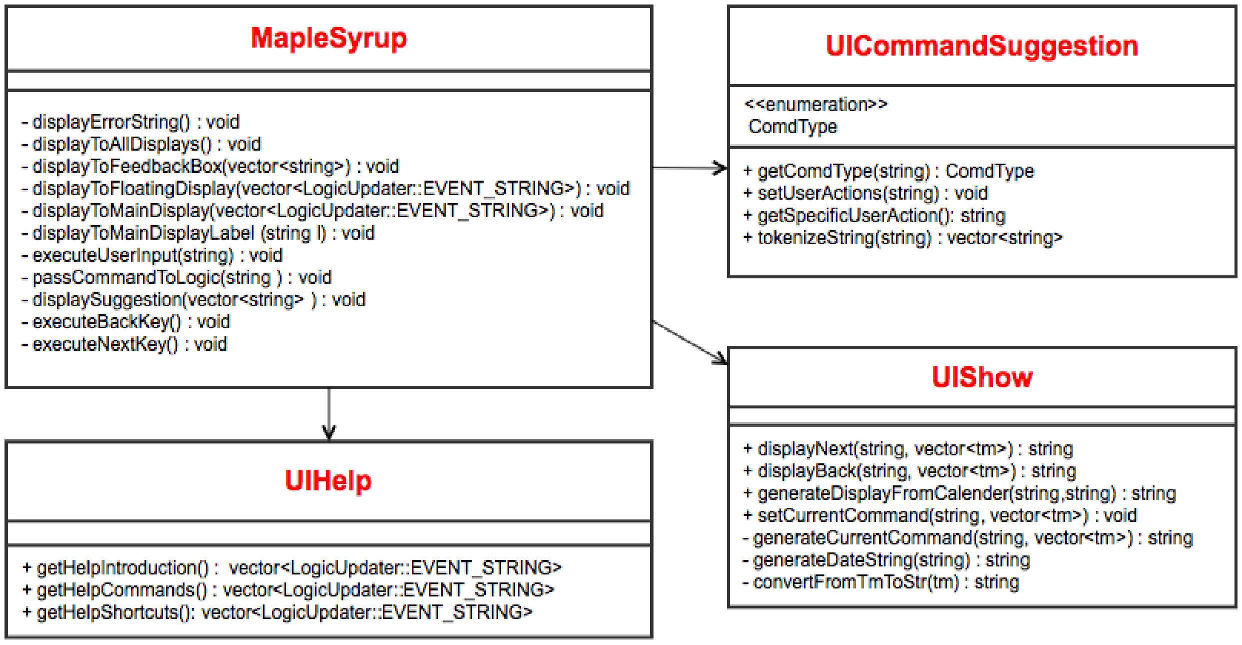
To illustrate interaction within the UI component, a sequence diagram of Add is included below.



For interaction with Logic, **Event Handlers** convert the input into a string to be passed to function executeUserInput() of class MapleSyrup, which is responsible for the interaction with Logic. This call returns an invocation to **Event Handlers**, which will proceed to retrieve information from Logic. A sequence diagram of Add is included to illustrate this interaction.



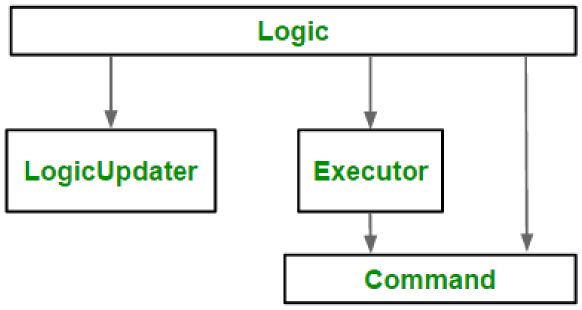
The main class of the UI component is MapleSyrup. It is responsible for the interaction between the back-end libraries and the user. It is being supported by another three classes within the UI component. These three classes act mainly as information storage and processor to Maplesyrup in order for it to have a seamless interaction with the user.


MapleSyrup
- displayErrorString() : void
- displayToAllDisplays() : void
- displayToFeedbackBox(vector<string>) : void
- displayToFloatingDisplay(vector<LogicUpdater::EVENT_STRING>) : void
- displayToMainDisplay(vector<LogicUpdater::EVENT_STRING>) : void
- displayToMainDisplayLabel (string l) : void
- executeUserInput(string) : void
- passCommandToLogic(string) : void
- displaySuggestion(vector<string>) : void
- executeBackKey() : void
- executeNextKey() : void
UICommandSuggestion
<<enumeration>>
ComdType
+ getComdType(string) : ComdType
+ setUserActions(string) : void
+ getSpecificUserAction(): string
+ tokenizeString(string) : vector<string>
UIShow
+ displayNext(string, vector<tm>) : string
+ displayBack(string, vector<tm>) : string
+ generateDisplayFromCalender(string,string) : string
+ setCurrentCommand(string, vector<tm>) : void
- generateCurrentCommand(string, vector<tm>) : string
- generateDateString(string) : string
- convertFromTmToStr(tm) : string
UIHelp
+ getHelpIntroduction() : vector<LogicUpdater::EVENT_STRING>
+ getHelpCommands() : vector<LogicUpdater::EVENT_STRING>
+ getHelpShortcuts(): vector<LogicUpdater::EVENT_STRING>

5. Logic

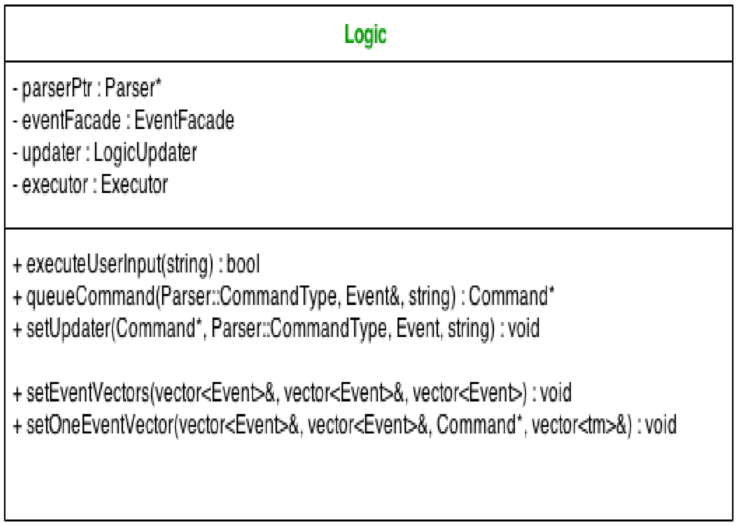
Overview

Logic has interactions with all other components, it is thus by nature more strongly coupled than these other components. Nevertheless, it applies several patterns and principles to limit coupling and increase cohesion as much as possible. Logic implements a command pattern as the Logic class is responsible for creating Command objects, which are passed to Executor to execute. This is also in line with the single responsibility principle as all classes in Logic are designed to encapsulate only one key objective entirely.

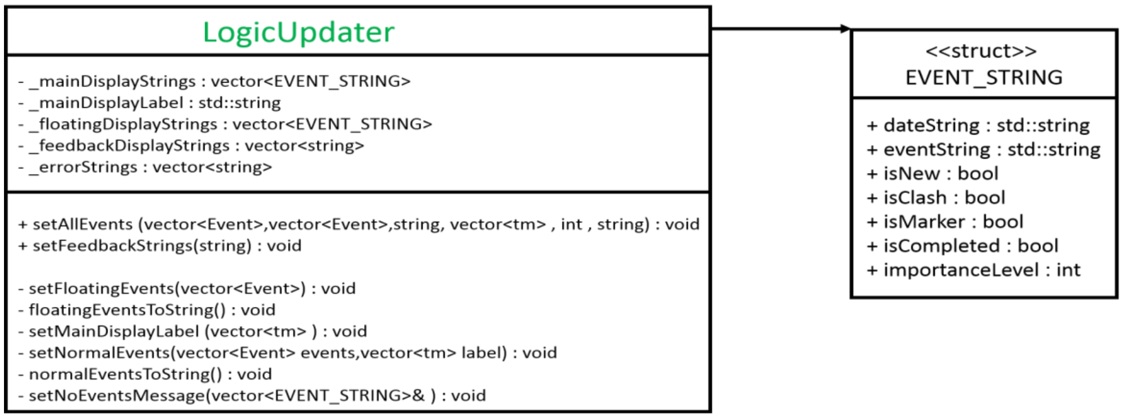


Logic

The Logic class can be considered the main class in the component. It is responsible for executing commands and updating the content that should be shown to the user via the LogicUpdater. LogicUpdater holds vectors of Event objects that should be updated for the user to see after every command is executed



```
Logic

- parserPtr : Parser*
- eventFacade : EventFacade
- updater : LogicUpdater
- executor : Executor

+ executeUserInput(string) : bool
+ queueCommand(Parser::CommandType, Event&, string) : Command*
+ setUpdater(Command*, Parser::CommandType, Event, string) : void

+ setEventVectors(vector<Event>&, vector<Event>&, vector<Event>) : void
+ setOneEventVector(vector<Event>&, vector<Event>&, Command*, vector<tm>&) : void
```

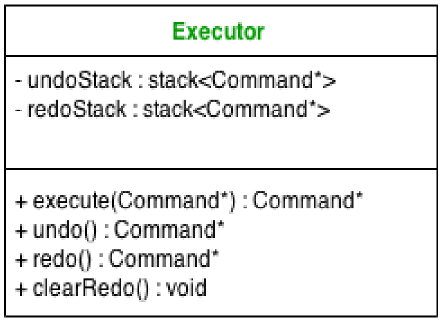


```
LogicUpdater

- _mainDisplayStrings : vector<EVENT_STRING>
- _mainDisplayLabel : std::string
- _floatingDisplayStrings : vector<EVENT_STRING>
- _feedbackDisplayStrings : vector<string>
- _errorStrings : vector<string>

+ setAllEvents (vector<Event>,vector<Event>,string, vector<tm> , int , string) : void
+ setFeedbackStrings(string) : void

- setFloatingEvents(vector<Event>) : void
- floatingEventsToString() : void
- setMainDisplayLabel (vector<tm> ) : void
- setNormalEvents(vector<Event> events,vector<tm> label) : void
- normalEventsToString() : void
- setNoEventsMessage(vector<EVENT_STRING>& ) : void
```

```
<<struct>>
EVENT_STRING

+ dateString : std::string
+ eventString : std::string
+ isNew : bool
+ isClash : bool
+ isMarker : bool
+ isCompleted : bool
+ importanceLevel : int
```

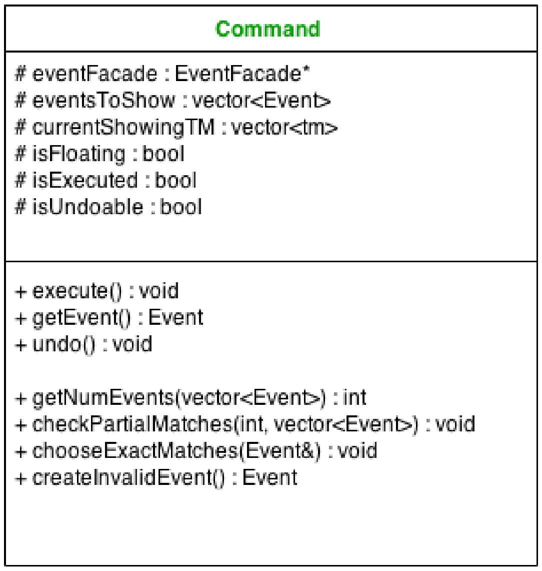
Executor

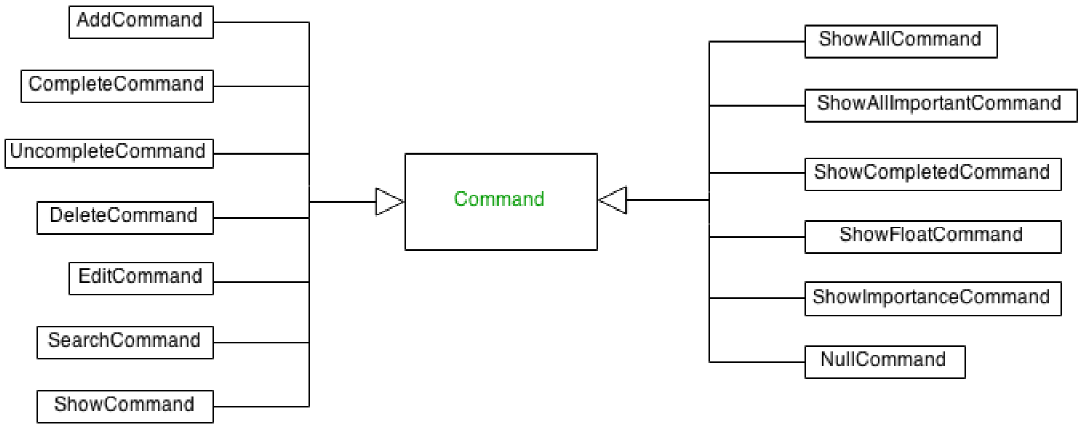
Executor is a small class which is only responsible for executing Command objects without being aware of what exactly is being executed. It has an undoStack and a redoStack that store undoable Commands. This supports undo and redo functionality for MapleSyrup.

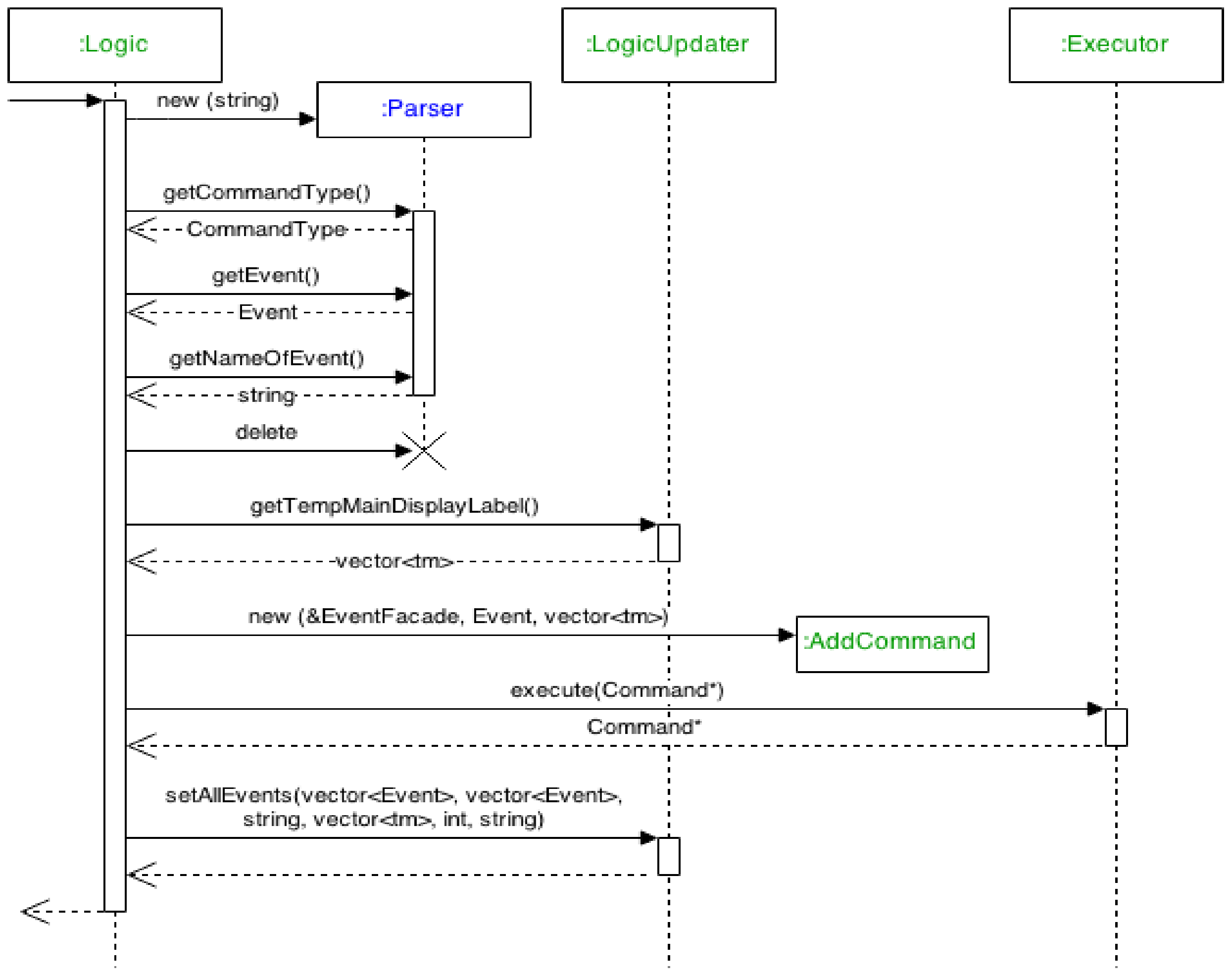

Executor
- undoStack : stack<Command*>
- redoStack : stack<Command*>
+ execute(Command*) : Command*
+ undo() : Command*
+ redo() : Command*
+ clearRedo() : void

Command

Command is an abstract superclass class from which subclasses of actual Command objects can be implemented. This is in accordance with the command pattern. The most important virtual methods are execute and undo; the latter must be implemented by all undoable commands to achieve proper undo and redo functionality.


Command
# eventFacade : EventFacade*
# eventsToShow : vector<Event>
# currentShowingTM : vector<tm>
# isFloating : bool
# isExecuted : bool
# isUndoable : bool
+ execute() : void
+ getEvent() : Event
+ undo() : void
+ getNumEvents(vector<Event>) : int
+ checkPartialMatches(int, vector<Event>) : void
+ chooseExactMatches(Event&) : void
+ createInvalidEvent() : Event





The sequence for the user adding an Event is shown here. UI will call executeUserInput(string) with the original string entered by the user. This string is used to create a Parser object, from which the command type, Event and relevant details are obtained through Parser's getters. After getting the range of tms currently showing from LogicUpdater, an AddCommand is created. This is passed to Executor to execute. The resulting AddCommand contains the required vector<Event> to update LogicUpdater. This is done by calling setAllEvents (vector<Event>, vector<Event>, string, vector<tm>, int, string) in LogicUpdater.

6. Parser

Overview

Parser handles the parsing of the user input to identify the command to be executed and to organise the additional information into a format readable by Logic. It also acts as the first line of defence against invalid commands that are either incomplete or unreadable and exceptions will be thrown, hence ensuring a valid format is passed to Logic.

Parser adopts a mediator pattern, where the Parser object acts as the main control station holding all the data and information, and coordinates all the actions of the InputStringSplit and ParserProcessor objects to process the user input string.

Parser

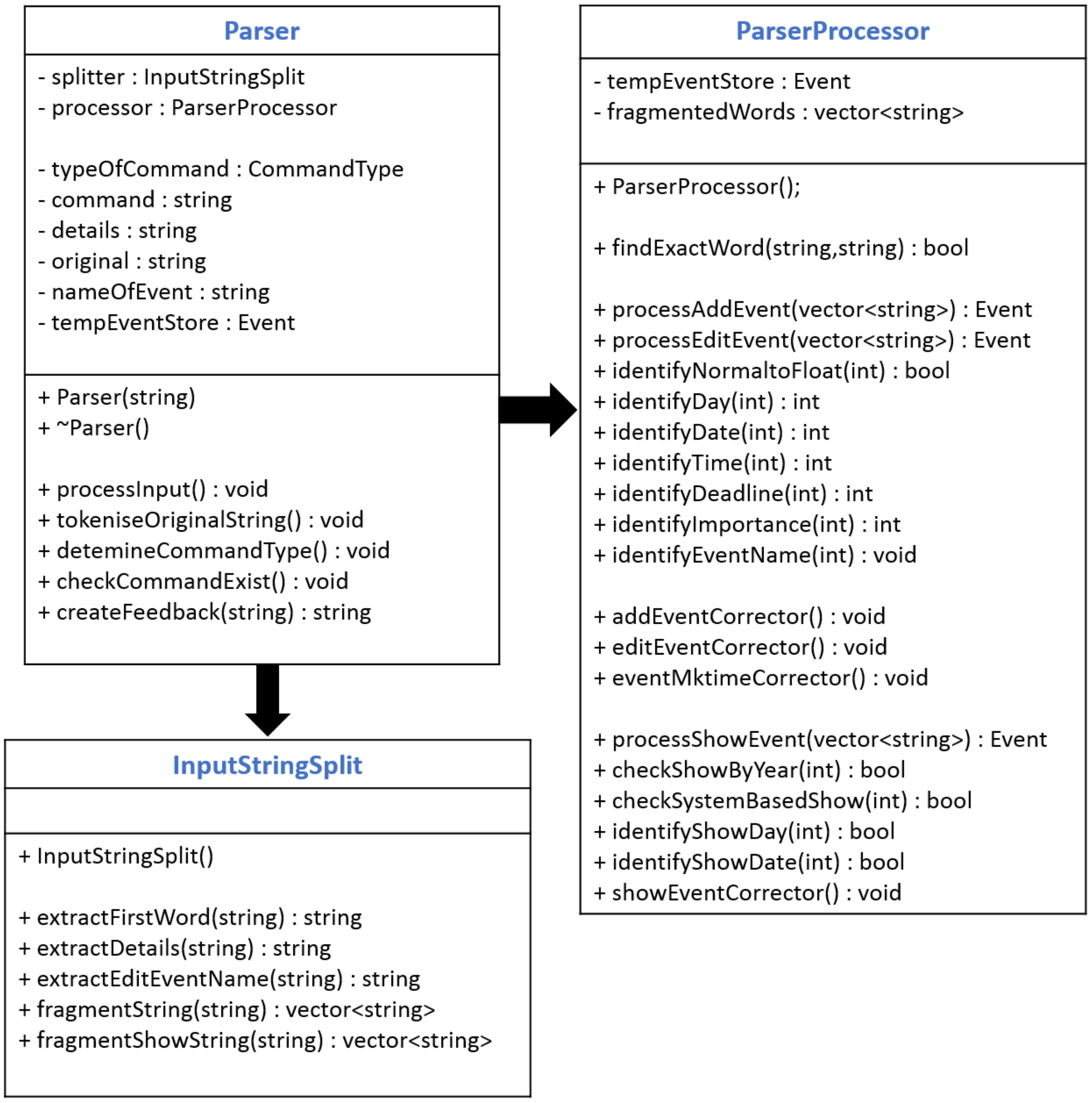
Parser is the main object within the parsing unit that Logic invokes. Therefore, it is required to hold all the information retrieved from the user input string, and organises them within its attributes to be read by Logic. In the parsing unit, it acts as the main control station.

InputStringSplit

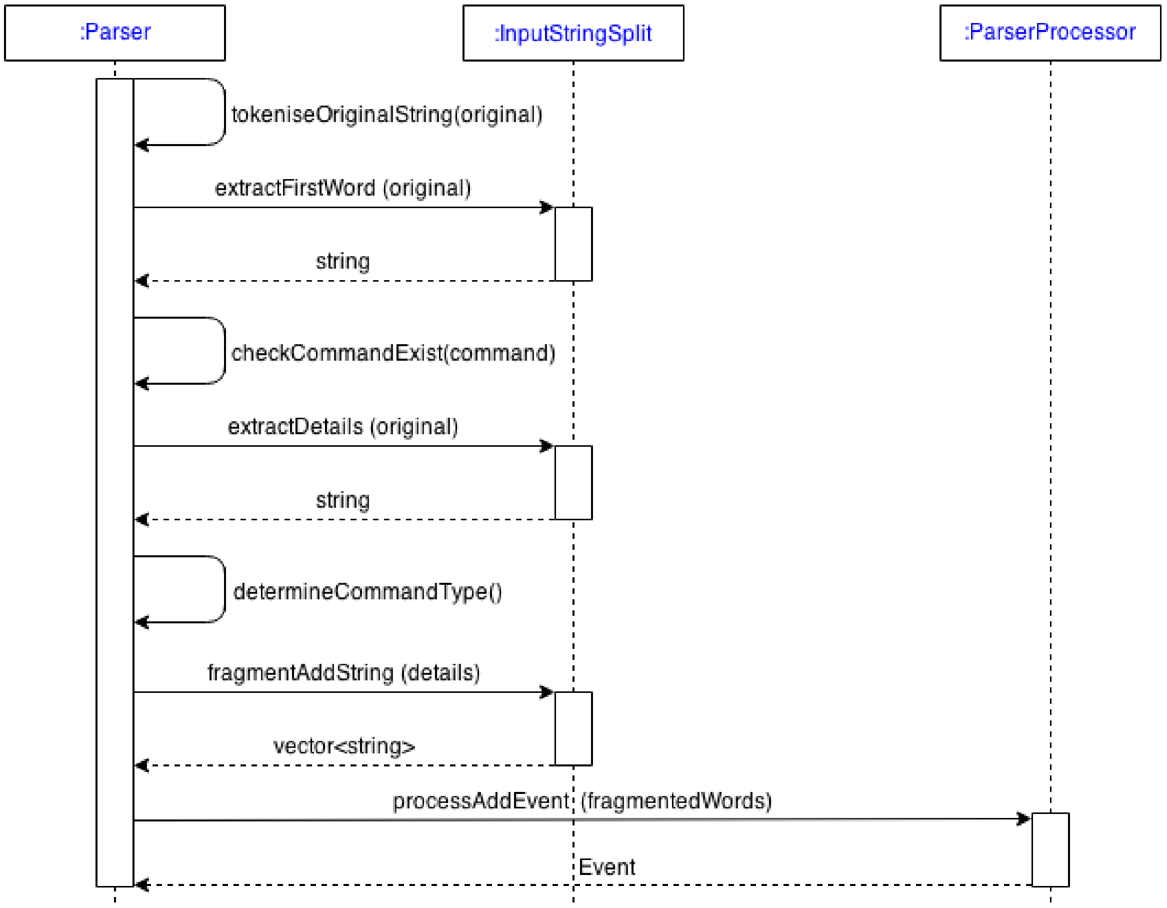
InputStringSplit is in charge of all the extracting, removing, and dissecting of the user input string to retrieve information like the intended command or the event name or index that the command is targeting. It then separates the additional details of the event from these information. It does not interpret or process the retrieved information any further and only returns it to Parser, who will decide the next course of action.

ParserProcessor

ParserProcessor is the processing unit that converts the string of details that is sent to it into an Event format that can be read by Logic. It has to identify all keywords within the string of details that indicate the dates, timings, importance and the event name and eventually to set up a fully completed Event with all its attributes determined. With the need to recognise different types of user input format of date and time and converting them to a single format understandable by the program, ParserProcessor has to contain many methods to identify the variations.


Parser
- splitter : InputStringSplit
- processor : ParserProcessor
- typeOfCommand : CommandType
- command : string
- details : string
- original : string
- nameOfEvent : string
- tempEventStore : Event
+ Parser(string)
+ ~Parser()
+ processInput() : void
+ tokeniseOriginalString() : void
+ detemineCommandType() : void
+ checkCommandExist() : void
+ createFeedback(string) : string
ParserProcessor
- tempEventStore : Event
- fragmentedWords : vector<string>
+ ParserProcessor();
+ findExactWord(string,string) : bool
+ processAddEvent(vector<string>) : Event
+ processEditEvent(vector<string>) : Event
+ identifyNormaltoFloat(int) : bool
+ identifyDay(int) : int
+ identifyDate(int) : int
+ identifyTime(int) : int
+ identifyDeadline(int) : int
+ identifyImportance(int) : int
+ identifyEventName(int) : void
+ addEventCorrector() : void
+ editEventCorrector() : void
+ eventMktimeCorrector() : void
+ processShowEvent(vector<string>) : Event
+ checkShowByYear(int) : bool
+ checkSystemBasedShow(int) : bool
+ identifyShowDay(int) : bool
+ identifyShowDate(int) : bool
+ showEventCorrector() : void
InputStringSplit
+ InputStringSplit()
+ extractFirstWord(string) : string
+ extractDetails(string) : string
+ extractEditEventName(string) : string
+ fragmentString(string) : vector<string>
+ fragmentShowString(string) : vector<string>

As the Parser object acts as a mediator, it dictates when the InputStringSplit and ParserProcessor objects will step in to process the user input, without the need for these two objects to have any knowledge of what the other object is doing or has done. This ensures low coupling between these two objects. An example of this is illustrated below in the form of a sequence diagram for the parsing of an add command.

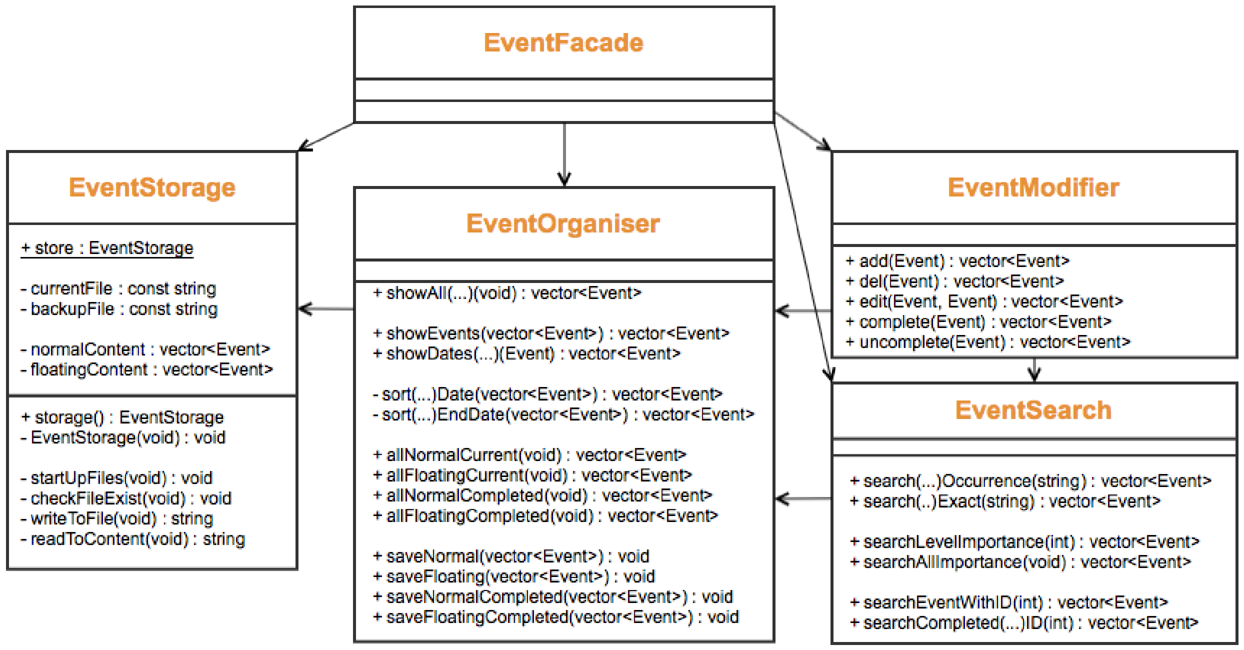

:Parser
:InputStringSplit
:ParserProcessor
tokeniseOriginalString(original)
extractFirstWord (original)
string
checkCommandExist(command)
extractDetails (original)
string
determineCommandType()
fragmentAddString (details)
vector<string>
processAddEvent (fragmentedWords)
Event

Given an input string, Parser will first call InputStringSplit to extract the first word which will be considered as the command, and then checks it if it is a valid command. Depending on the command, it will call InputStringSplit again to extract the details from the input string, excluding the command, and then fragments the string into a vector of strings. Finally, based on the command, it will call ParserProcessor to process, identify and organise the details into an Event format to be stored within Parser, and eventually be used in Logic.

7. Storage

Overview

The Storage component is responsible for all internal processing of Events while maintaining both internal and external storages by reading and writing from the text file. Storage makes use of the facade pattern. All input received from Logic should pass through EventFacade for redirection. This conforms to the Law of Demeter as it hides several classes behind an API. Clients calling Storage will only need to invoke EventFacade. In addition, uses a singleton pattern as there can be only one instance at any time. Storage is composed of five classes, namely EventFacade, EventModifier, EventSearch, EventOrganiser and EventStorage. These classes are developed with the principle of the single responsibility in mind with their responsibilities entirely encapsulated within the respective classes. Their dependencies are shown in the Storage class diagram below.


EventFacade
EventStorage
+ store : EventStorage
- currentFile : const string
- backupFile : const string
- normalContent : vector<Event>
- floatingContent : vector<Event>
+ storage() : EventStorage
- EventStorage(void) : void
- startUpFiles(void) : void
- checkFileExist(void) : void
- writeToFile(void) : string
- readToContent(void) : string
EventOrganiser
+ showAll(...)(void) : vector<Event>
+ showEvents(vector<Event>) : vector<Event>
+ showDates(...)(Event) : vector<Event>
- sort(...)Date(vector<Event>) : vector<Event>
- sort(...)EndDate(vector<Event>) : vector<Event>
+ allNormalCurrent(void) : vector<Event>
+ allFloatingCurrent(void) : vector<Event>
+ allNormalCompleted(void) : vector<Event>
+ allFloatingCompleted(void) : vector<Event>
+ saveNormal(vector<Event>) : void
+ saveFloating(vector<Event>) : void
+ saveNormalCompleted(vector<Event>) : void
+ saveFloatingCompleted(vector<Event>) : void
EventModifier
+ add(Event) : vector<Event>
+ del(Event) : vector<Event>
+ edit(Event, Event) : vector<Event>
+ complete(Event) : vector<Event>
+ uncomplete(Event) : vector<Event>
EventSearch
+ search(...)Occurrence(string) : vector<Event>
+ search(..)Exact(string) : vector<Event>
+ searchLevelImportance(int) : vector<Event>
+ searchAllImportance(void) : vector<Event>
+ searchEventWithID(int) : vector<Event>
+ searchCompleted(...)ID(int) : vector<Event>

EventFacade

EventFacade is a thin class that does not process any data. Instead, it purely redirects all commands received from Logic to the appropriate Storage components. Hence, providing a simple interface for interaction with external clients while hiding all the internal complexity of the storage.

EventModifier

EventModifier class deals with the modification of an Event or its attributes. Modifications that can be administered to an Event includes add, delete, edit, complete and uncomplete. EventModifier will simultaneously update EventStorage according. EventModifier makes use of EventSearch to locate the modified Event from the internal storage and also EventOrganiser to format the modified Event.

EventSearch

EventSearch is responsible for locating an Event from EventStorage. This class is used mainly by EventModifier to located the modified Event. However, it can also be called by EventFacade if the external client requires a specific Event. For instance, when a search is called. EventSearch obtains the raw data from EventOrganiser after they are filtered. It also uses EventOrganiser to sort and format the Event.

EventOrganiser

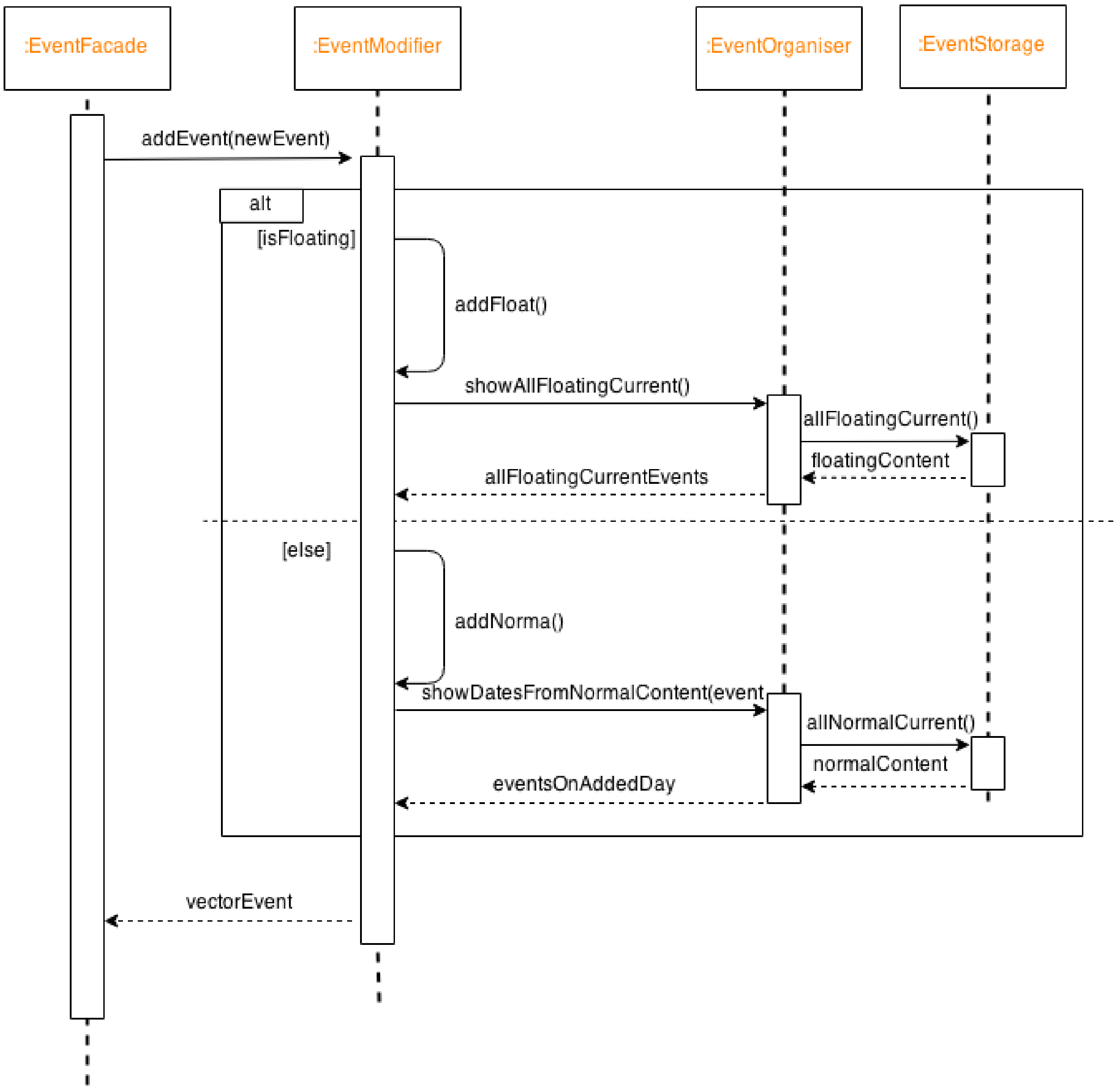
EventOrganiser is the only class that gets and sets Events with EventStorage and is responsible for the organisation and formatting of the Event. There are 4 main types of organisation methods. Firstly, it gets Event from EventStorage and filters them into 4 categories. Namely, current normal, current floating, completed normal and completed floating Events. These filtered results are used by the other Storage components. Secondly, it sorts the Events according to date and time and separates them with a marker. Thirdly, show functions which retrieves and formats Event can be called by show(...)(). Lastly, it merges completed and uncompleted events and sets them in EventStorage using the save(...)() method.

EventStorage

EventStorage is implemented with a singleton pattern as there can be only one instance of EventStorage at any time. This is to prevent read/write errors and data corruption. EventStorage stores the internal data in 2 vectors of Event. A read operation is performed during program start-up in order to read the saved information from an external text file. This function is provided by the startUpFiles() method and called by EventFacade. On the other hand, EventStorage will write to the text file each time an Event is modified. The method writeToFile() is called to save the content externally.

MapleSyrup stores the user's Event in a human readable and modifiable plaintext file. This is for the benefit of advanced users, who may choose to directly modify the text file or transfer it to another computer. This process, however, runs the risk of the user modifying the text file to the extent that it is unable to be read by *MapleSyrup*. To address this issue, a backup file is written every time *MapleSyrup* successfully reads the current file upon start-up. In the case of any errors in reading the original save file, *MapleSyrup* will automatically switch to the backup and subsequently overwrite the original file.

A sample execution of adding an event in Storage is shown here in the sequence diagram. When EventFacade receives a commands from Logic, it redirects the command addEvent() to EventModifier which executes add(newEvent). The method will check if the Event is a floating event or normal Event and proceed to retrieve the existing vector of Events, push the new Event in and save it into EventStorage. It will then retrieve the events that will be return to Logic. If a floating Event is added, all floating Events will be returned. If a normal Event is added, all existing events within the new Event date range will be return.



8. Appendix

8.1 Important APIs

8.1.1 UI

MapleSyrup

void executeUserInput(string) : This function centralises all the calls from the various parts/event handlers from the UI. It first checks and matches commands that are related to developer or UI-handled. If yes, it proceeds with executing these commands. If no, it proceed to pass command in string form to function to passCommandToLogic()

void passCommandToLogic(string): This function links UI to Logic by passing a command in string to Logic for execution. Thereafter, based on the Boolean variable it received from Logic's executeUserInput() function, it proceeds to call functions to display the relevant information to the various displays on the UI.

void displayToAllDisplays() : Get the display vectors from Logic when invoked by function executeUserInput(). It proceed on to display these vectors to the respective displays, namely main display, floating tasks display and feedback box.

void displayErrorString(): Get the error string from Logic when invoked by function executeUserInput(). It proceed on to display this error string onto the feedbackBox of the UI.

UIShow

string displayNext(string, vector<tm>) : This function takes in a string that contains that date(s)/labels that is being displayed in the main display currently. It returns the string which contain the command to display the next date(s)/labels based on what it has received.

string displayBack(string, vector<tm>) : This function takes in a string that contains that date(s)/labels that is being displayed in the main display currently. It returns the string which contain the command to display the previous date(s)/labels based on what it has received.

string generateDisplayFromCalender(string, string) : This function should combine with COMMAND_SHOW (at the front) to generate a proper command. This function takes in a string that contains that date(s) from calendar in its specific format of dd/mm/yyyy in string form and return its equivalent show command.

8.1.2 Logic

Logic

bool executeUserInput(string): Called with the exact user input string, then creates Parser object to determine the correct action to take. After creating the appropriate Command object and calling Executor to execute it, returns bool. Return value is true by default, false if command not fully executed.
void queueCommand(Parser::CommandType, Event, string): Dynamically creates Command object, calls Executor to execute it.

void setUpdater(Command*, Parser::CommandType, Event, string): Updates new information for UI to display. This information is obtained from the executed Command.

Command

virtual void execute(): Must be implemented by all Command objects that implement the Command abstract class. Contains code that executes actual command by invoking EventFacade.

virtual void undo(): Must be implemented by all Command objects that are undoable. Contains code that will undo the executed command.

Executor

void execute(Command*): Calls execute() method of Command object. Pushes this Command into the undoStack if the command is undoable and was executed correctly.

void undo(): Calls undo() method of the Command object that is at the top of the undoStack. Pushes this Command into the redoStack.

void redo(): Calls execute() method of the Command object that is at the top of the redoStack. Pushes this Command back into the undoStack.

LogicUpdater

void setAllEvents (vector<Event>,vector<Event>,string, vector<tm>, int, string) : This function is the single point assessed by the setter to updated the information to be displayed to user. Information passed into this function will be processed and stored in 2 forms in the respective private attribute of this Class

void setFeedbackStrings(string) : This function is the single point assessed by the setter to updated the feedback to be displayed. This function is purely to update the feedback only. It is being used in the case where only feedback required update which the other information remain the same.

8.1.3 Parser

Parser

void Parser::tokenizeOriginalString(): Separates input string into a command and additional details. Based on the command, calls InputStringSplit object to further split the remaining string, then calls ParserProcessor object to process the split string. Command type will be determined and additional information will be stored in Event object within the Parser object.

InputStringSplit

vector<string> fragmentString(string): Splits input string into components by removing spaces and ".-" symbols, then stores them in a vector<string>. Returns this vector.

ParserProcessor

Event processAddEvent(vector<string>): Identifies names, dates and time in their respective formats from argument vector<string>, stores them in an Event object. Dates and time will be converted from string to integer and missing details filled in. Returns completed Event.

8.1.4 Storage

EventFacade

vector<Event> addEvent(Event): adds an event to Storage

vector<Event> deleteEvent(Event): deletes an event from Storage vector<Event> editEvent(Event, Event): edits an event from Storage

vector<Event> completeEvent(Event): completes an event in Storage

vector<Event> uncompleteEvent(Event): uncompletes an event in Storage

vector<Event> findNameOccurrence(string): find occurrence of event name from existing events in storage

vector<Event> findNameExact(string): find exact matches of event name from existing events in storage

vector<Event> findLevelImportance(int): finds events with specified level of importance

vector<Event> showDates(Event): show Event dates sorted by days and separated by a marker.

EventOrganiser

vector<Event> showEvents(vector<Event>): showEvent takes the start and end date of the vector of Event given and sort them according to date. Markers are then added to separate the Events on different days.

vector<Event> showDatesFromNormalContent(Event): takes the start and end date of the Event given and sort them according to date. Markers are then added to separate the Events on different days.

EventStorage

void startUpFiles(void): Used when program is started. It is called by eventFacade, then it checks if the text files myCurrent and myBackup exist. If they do not exist, new text files will be created. Next, readToContent(currentFile) will read myCurrent into 2 vectors. If reading fails due to data corruption, myBackup will be read instead. myCurrent will also be updated with myBackup.

8.2. Testing

MapleSyrup was built with constant regression testing. Each component was tested on its own with relevant unit tests before being tested with the rest of the system. System testing was performed mainly on the Logic component as a whole to ensure that *MapleSyrup* remained robust with every iteration. Unit tests for each component are written and built with the NUnit testing framework for Microsoft.NET 4.5, bundled with Visual Studios 2012 Professional Edition where *MapleSyrup* was developed. The unit test files for individual components are available together with the source code of *MapleSyrup* and can be run using the following steps:

1. Open MapleSyrup with Visual Studios 2012, Professional Edition

2. Right click on the UI project in the Solution Explorer and select Properties

a. Ensure that the Configuration option at the top left is set to Debug

b. Ensure that the library is being built as a Static Library under Project Defaults -> Configuration Type

3. Navigate to Test -> Windows -> Test Explorer

4. Click on Run All in the Test Explorer window to run the tests

Should you desire to add tests of your own, simply navigate to UnitTest -> Source Files in the Solution Explorer. Choose the source file with the name of the class you intend to test (e.g. if you want to test Event methods, choose EventTest.cpp). Finally, add a public TEST_METHOD to the TEST_CLASS; this is where you can write your test. All tests should be written with Asserts provided for by the namespace, and all test names should be prefixed with the name of the class and method being tested.

It is highly recommended that you test your new or changed components in isolation first. Once the code is stable and basic bugs have been removed, you should then integrate his changes with the most relevant components, and run both your own tests as well as those currently implemented. Only after this stage has proved successful should you move on to integration with the entire system and subsequently, system testing. Should you desire to access or add tests of your own, simply navigate to SystemTest -> Source Files in the Solution Explorer.