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

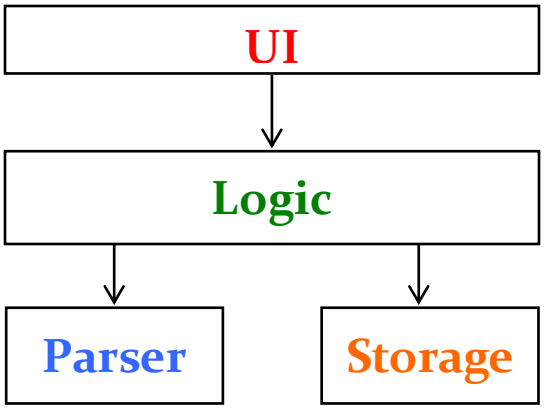
MapleSyrup is written in C++ programming language using Visual Studio 2012 (Professional). Hence, developers should be familiar with this programming language and developer software.

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.

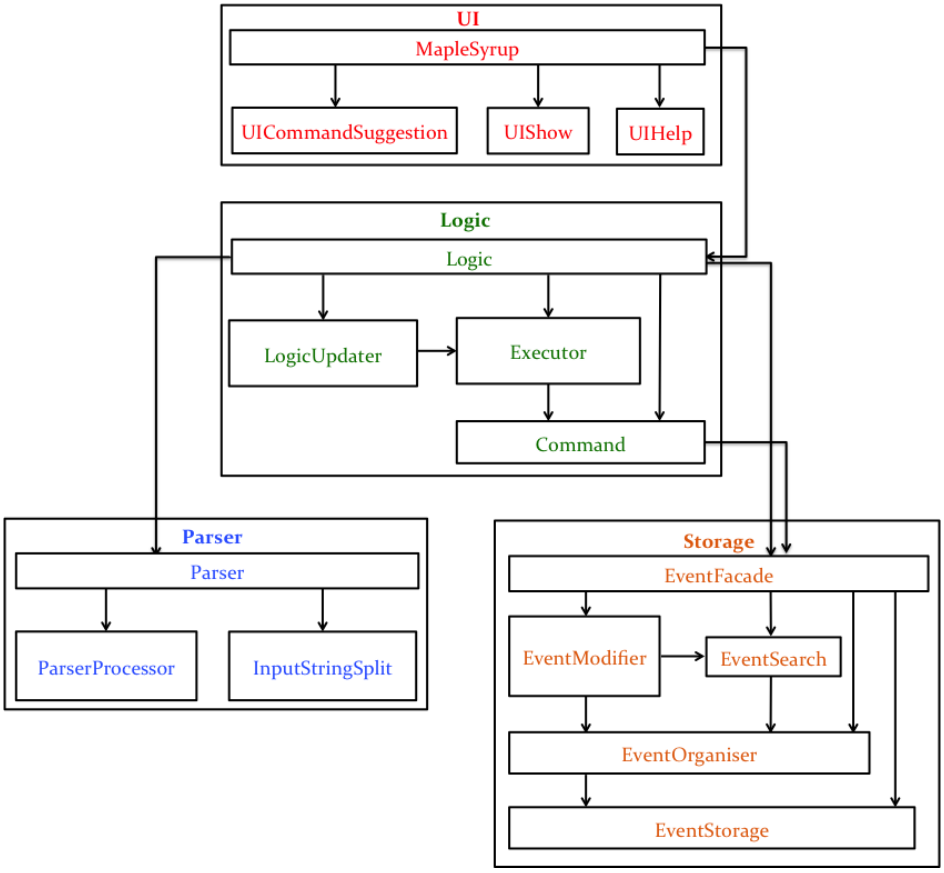
In implementing a command pattern, *MapleSyrup* also subscribes to the open-closed principle. The functionality of the program can be extended by implementing more commands to extend the virtual Command class, but without changing the source code.

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.



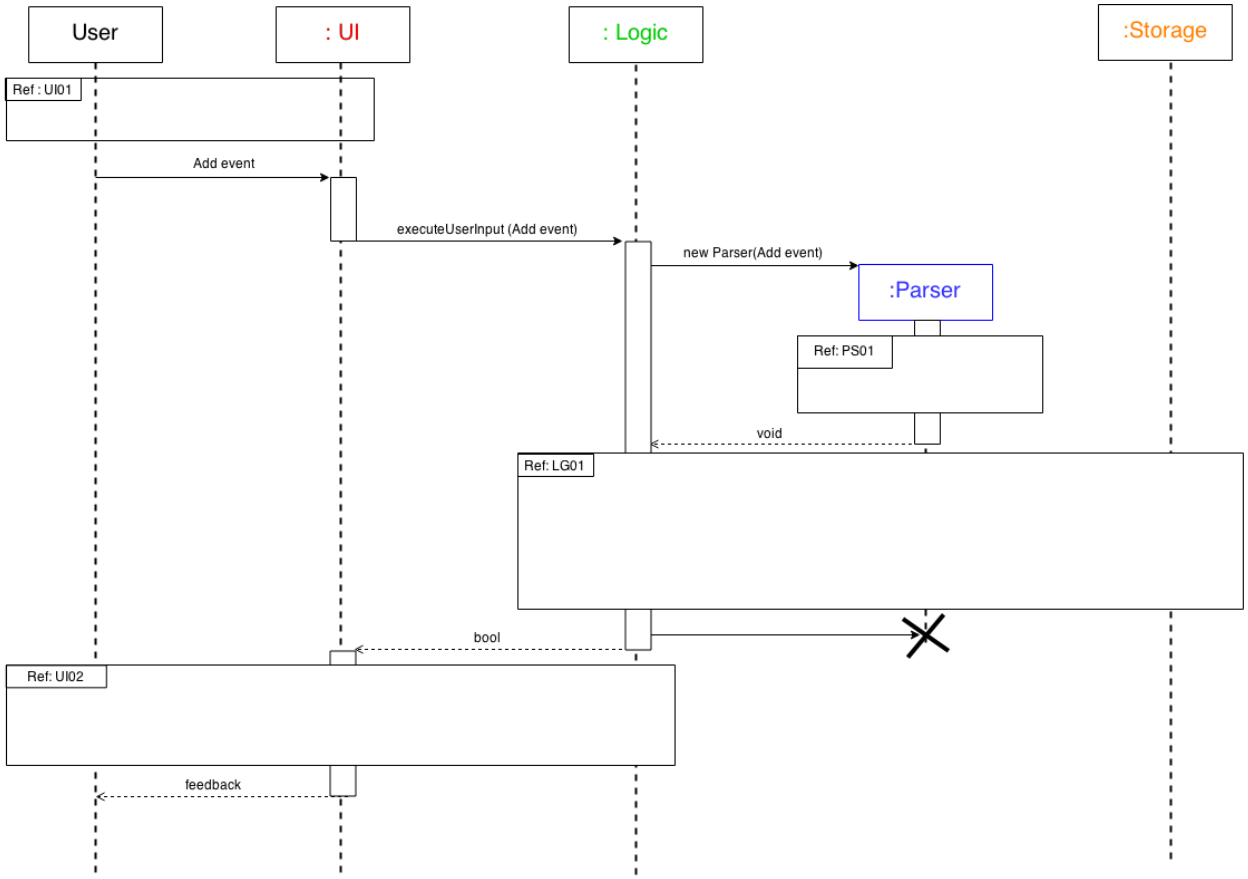
2.2 What Is An Event?

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

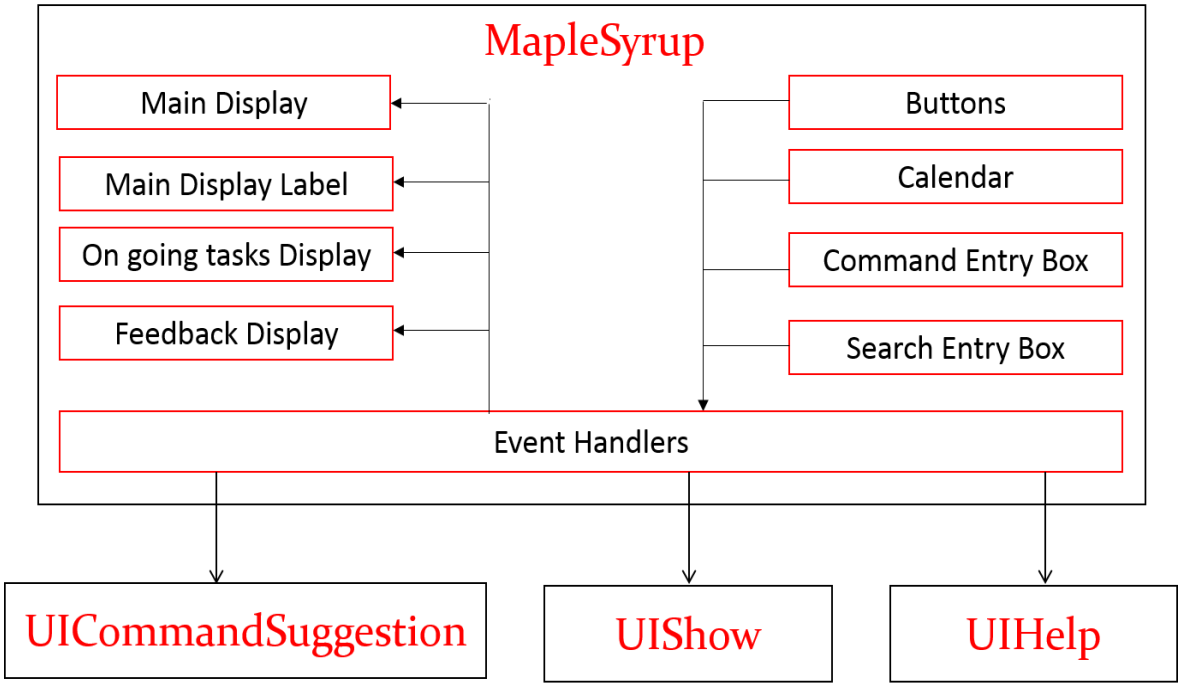
The sequence diagram for adding an Event is shown below to help facilitate the understanding of the basic interaction between and within each component. References can be found in the respective component section with more detailed descriptions showing how the internal components interact. The process 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)

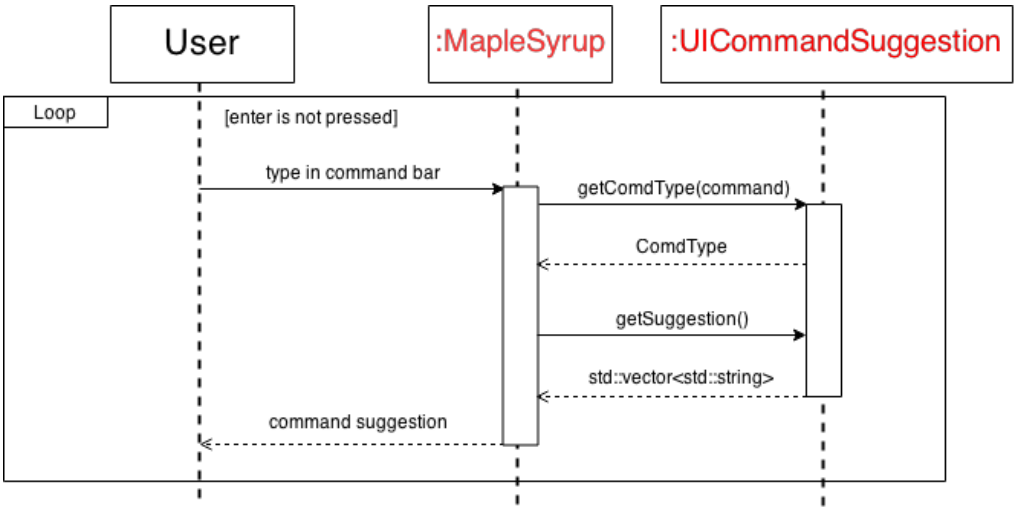
The UI is responsible for collecting all user input, converting internal data into a human readable format, and subsequently displaying it to the user. UI adopts a concept similar to that of facade when crafting its implementations. Inputs from users converge to only 1 point - function executeUserInput().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

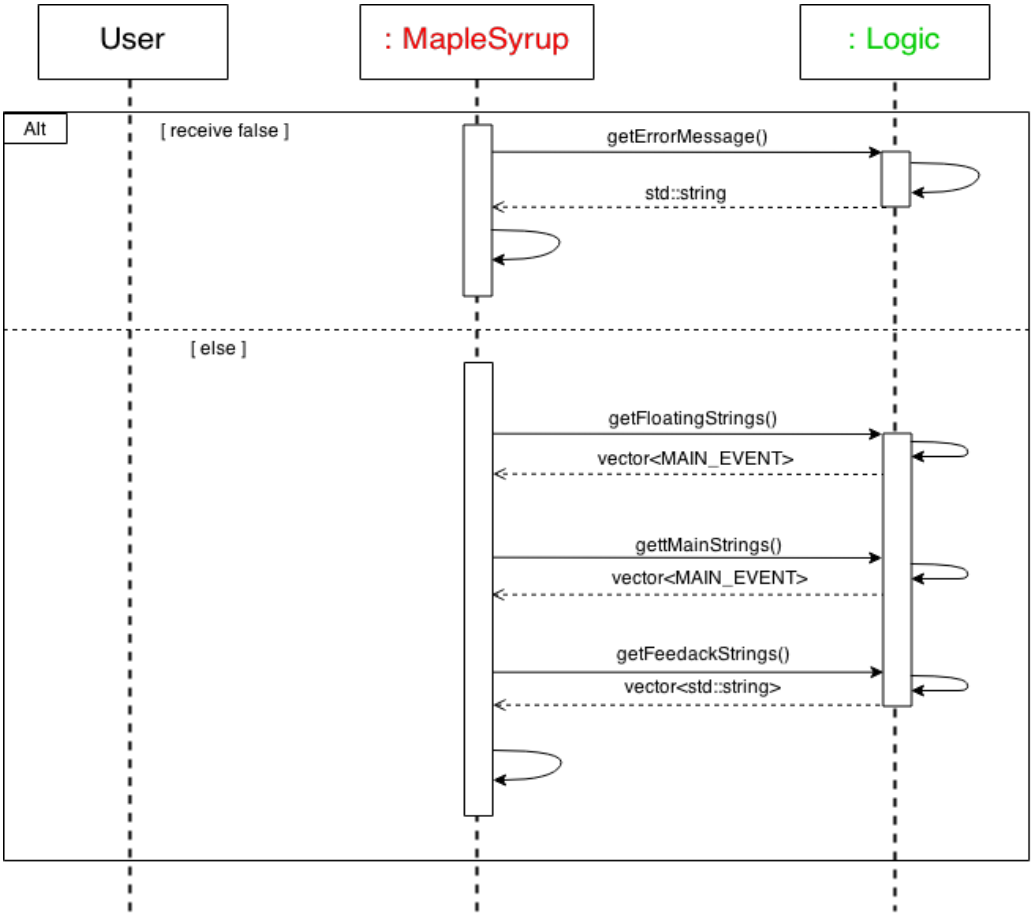


When a user action is detected, such as when a button is clicked, **Buttons** will be activated respectively to notify the respective **Event Handlers** of the user interaction. **Event Handlers** interact with UICommandSuggestion, UIShow and UIHelp within the UI component as well as with Logic. After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

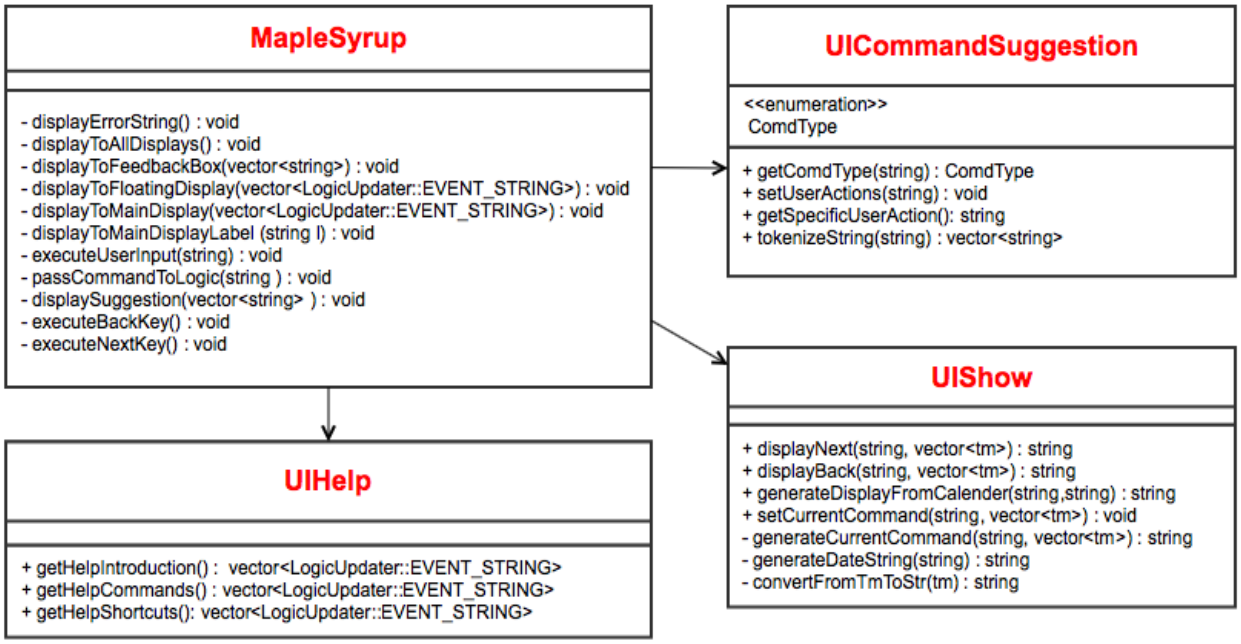
To illustrate interaction within the UI component, a sequence diagram of Add is included below.



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.

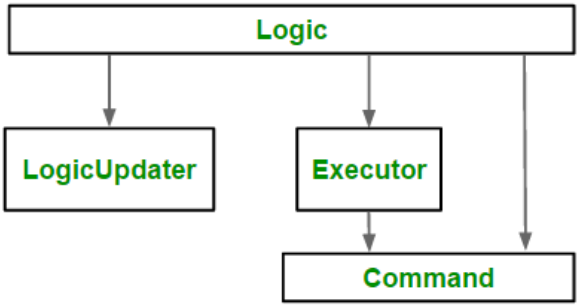


The main class of the UI component is MapleSyrup. It is responsible for the interaction between the back-end libraries and the user. It is being supported by another three classes within the UI component. These three classes act mainly as information storage and processor to Maplesyrup in order for it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



5. Logic

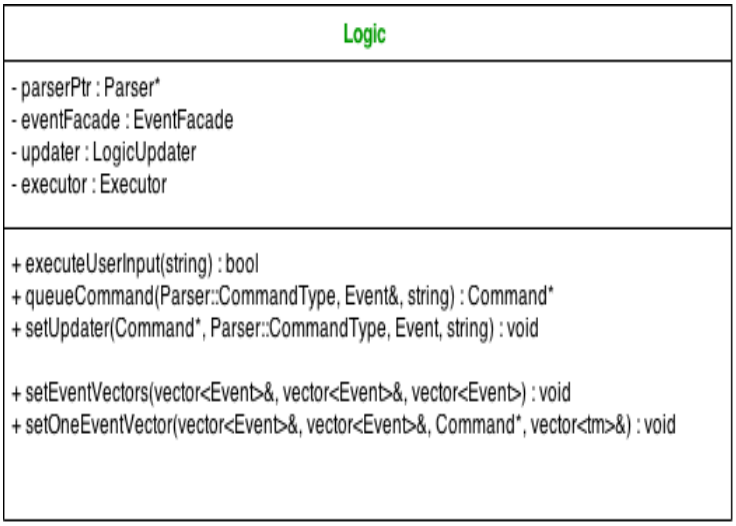
Overview

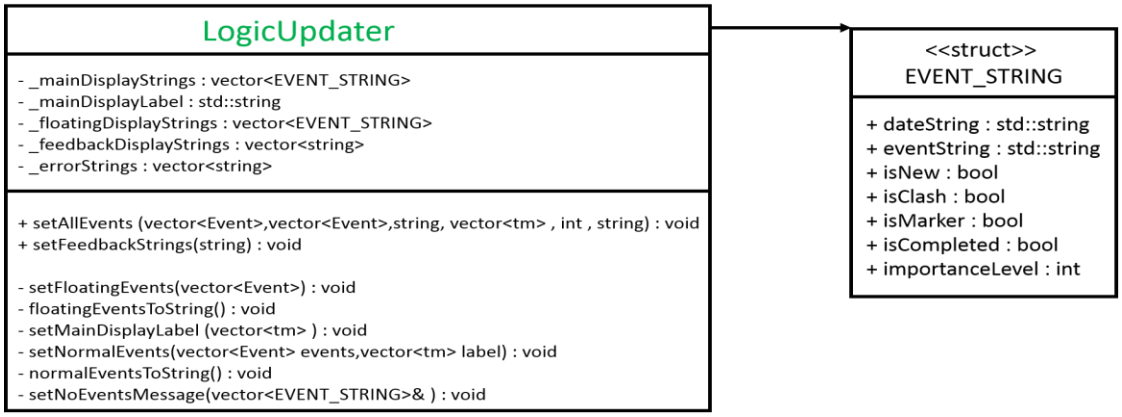


Logic has interactions with all other components, it is thus by nature more strongly coupled than these other components. Nevertheless, it applies several patterns and principles to limit coupling and increase cohesion as much as possible. Logic implements a command pattern as the Logic class is responsible 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

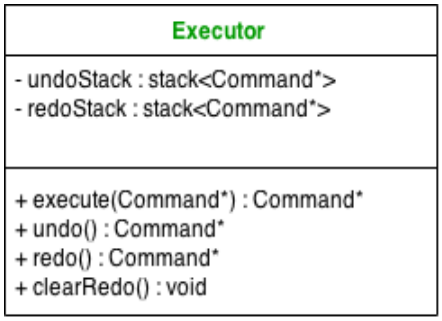
The Logic class can be considered the main class in the component. It is responsible for executing commands and updating the content that should be shown to the user via the LogicUpdater. LogicUpdater holds vectors of Event objects that should be updated for the user to see after every command is executed





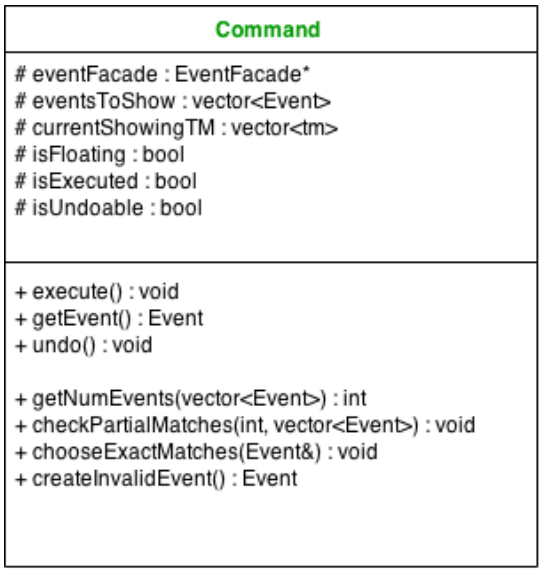
Executor

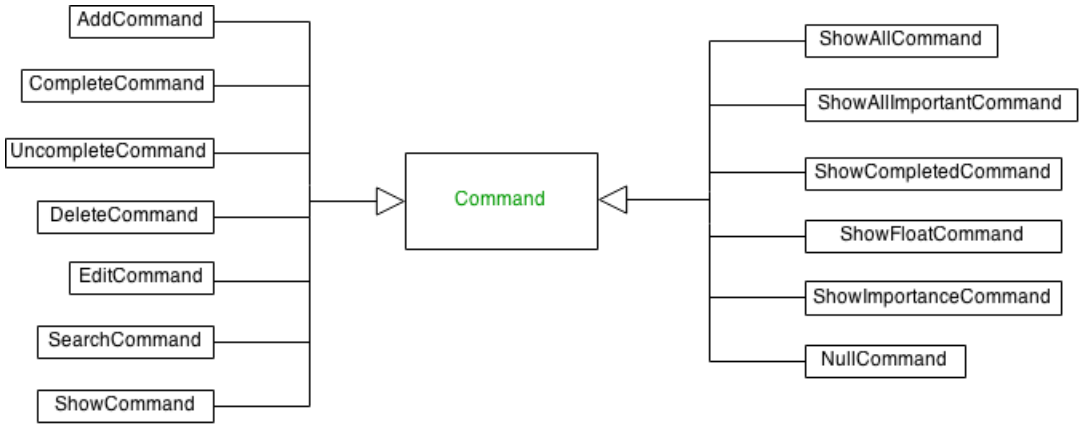
Executor is a small class which is only responsible for executing Command objects without being aware of what exactly is being executed. It has an undoStack and a redoStack that store undoable Commands. This supports undo and redo functionality for MapleSyrup.

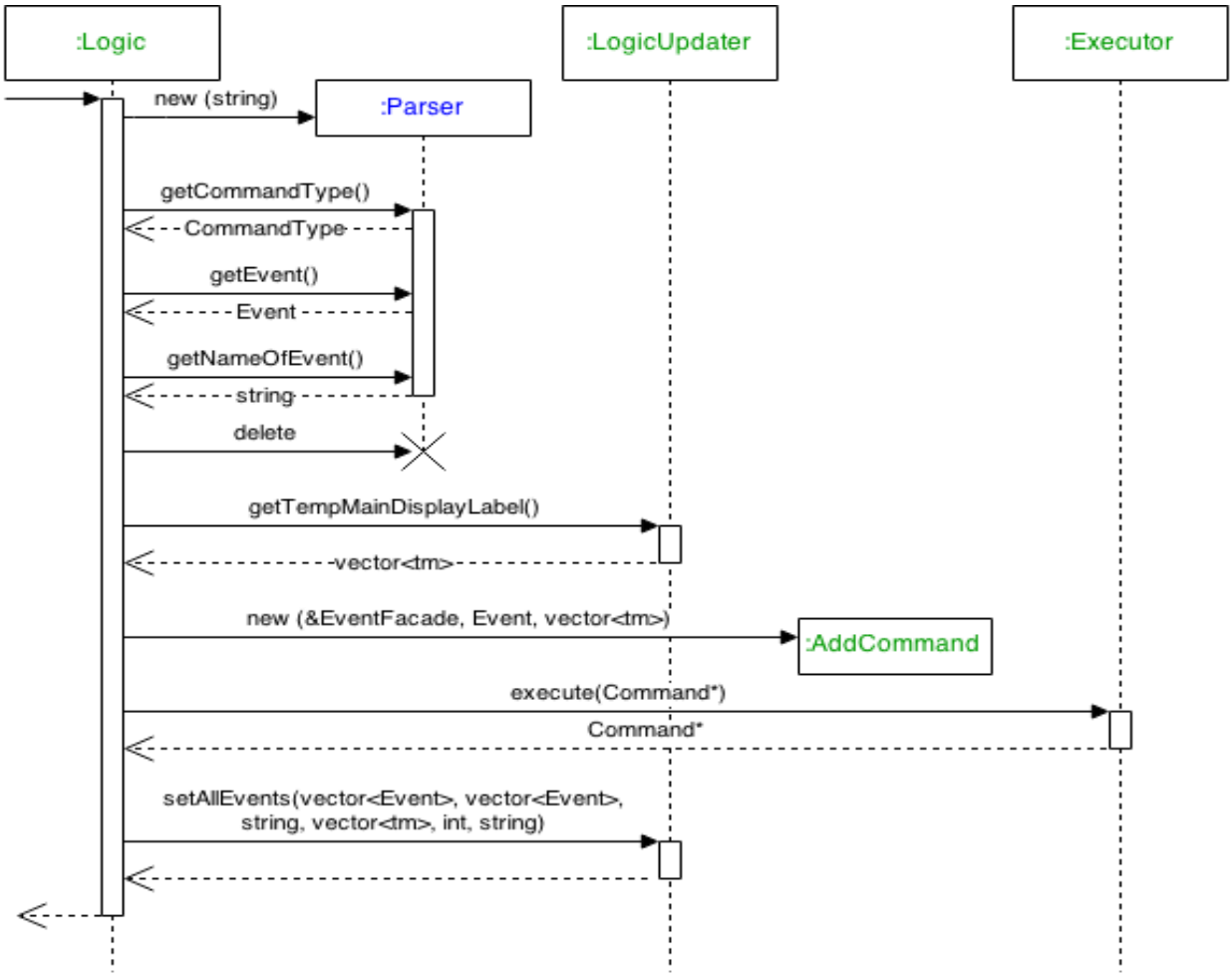


Command

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.







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

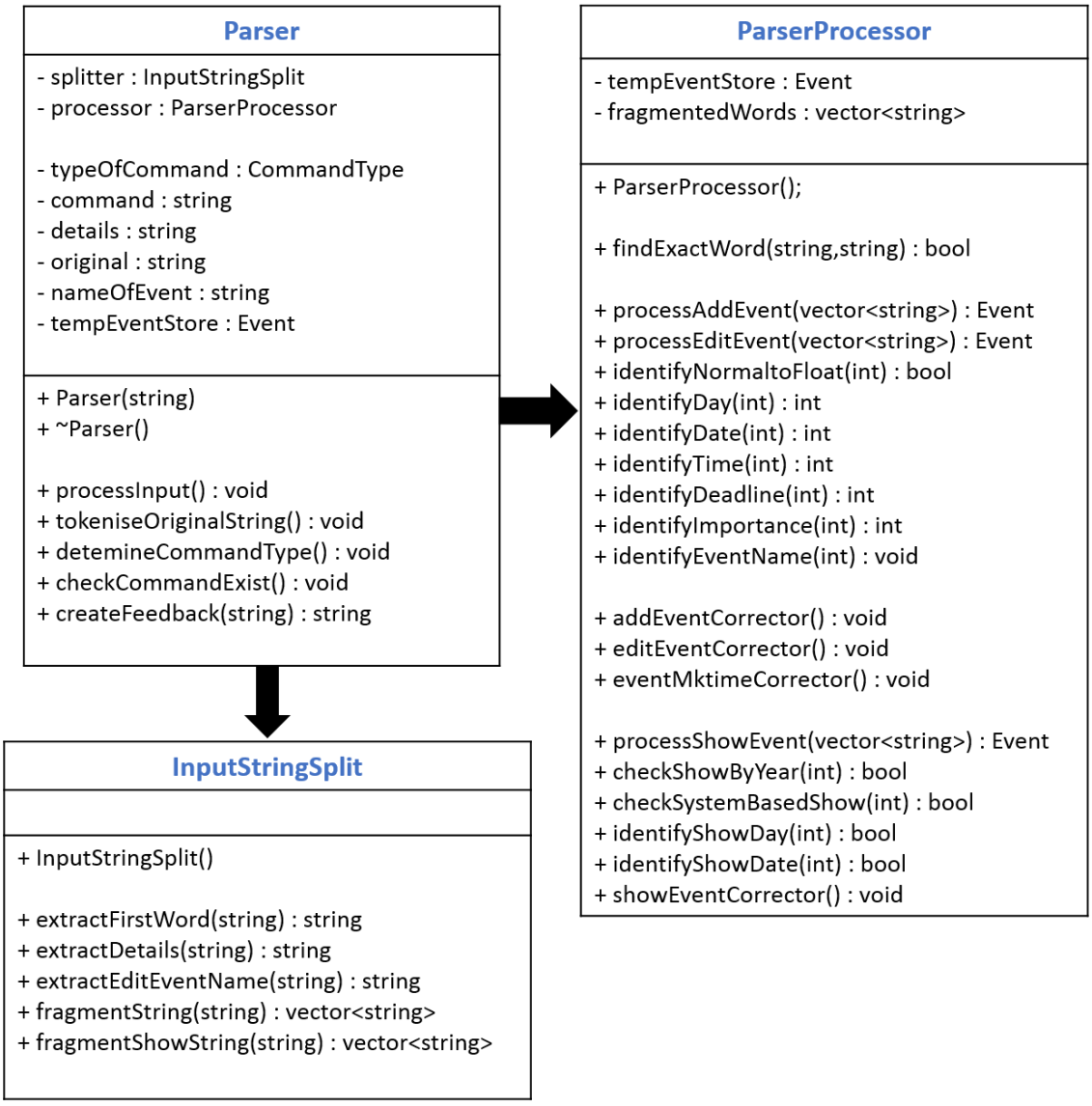
Parser is the main object within the parsing unit that Logic invokes. Therefore, it is required to hold all the information retrieved from the user input string, and organises them within its attributes to be read by Logic. In the parsing unit, it acts as the main control station.

InputStringSplit

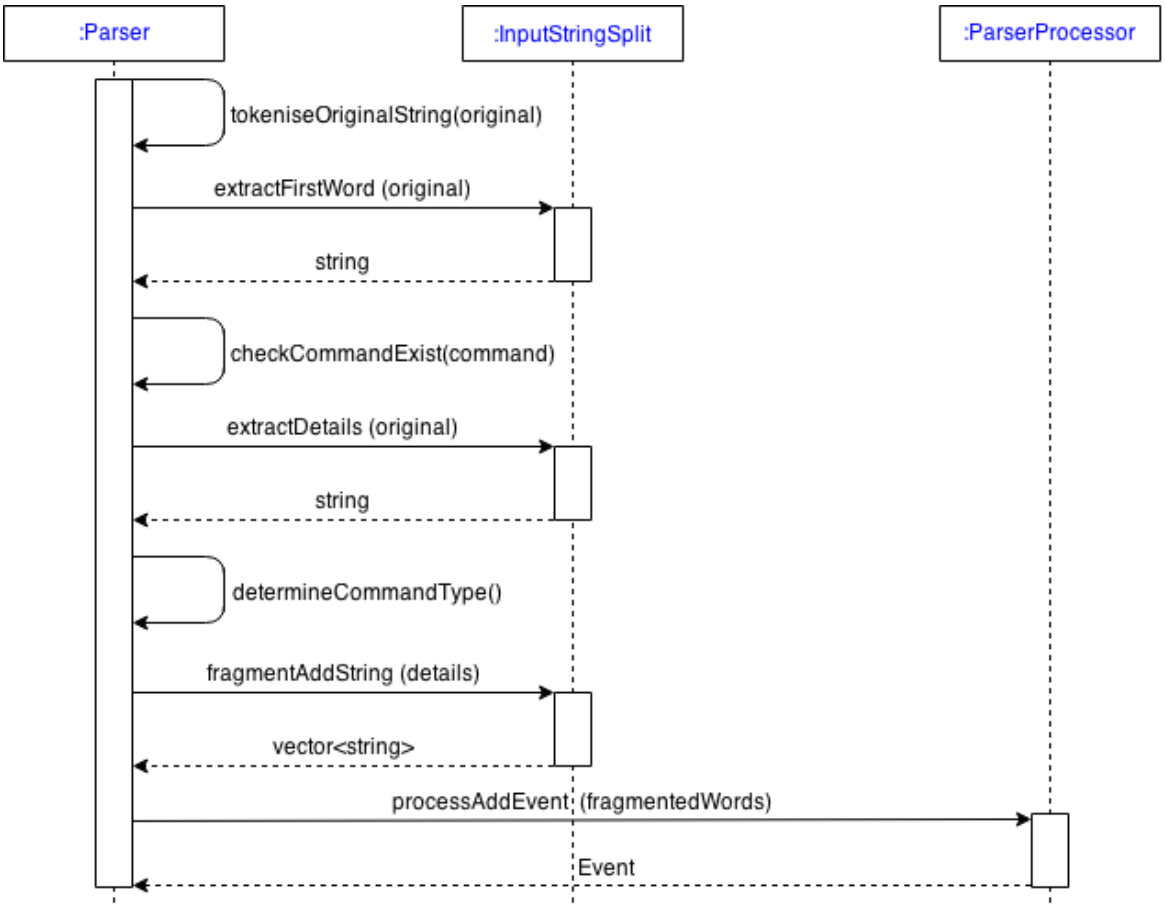
InputStringSplit is in charge of all the extracting, removing, and dissecting of the user input string to retrieve information like the intended command or the event name or index that the command is targeting. It then separates the additional details of the event from these information. It does not 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.



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.

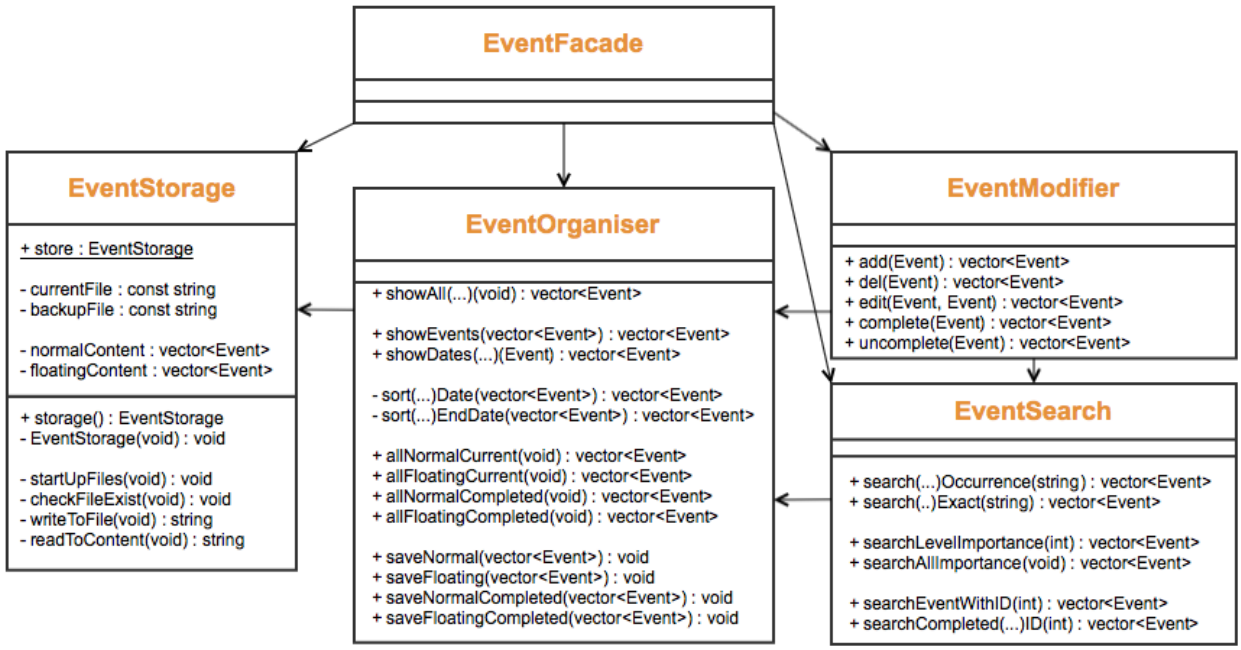


Given an input string, Parser will first call InputStringSplit to extract the first word which will be considered as the command, and then checks it if it is a valid command. Depending on the command, it will call InputStringSplit again to extract the details from the input string, excluding the command, 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

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.

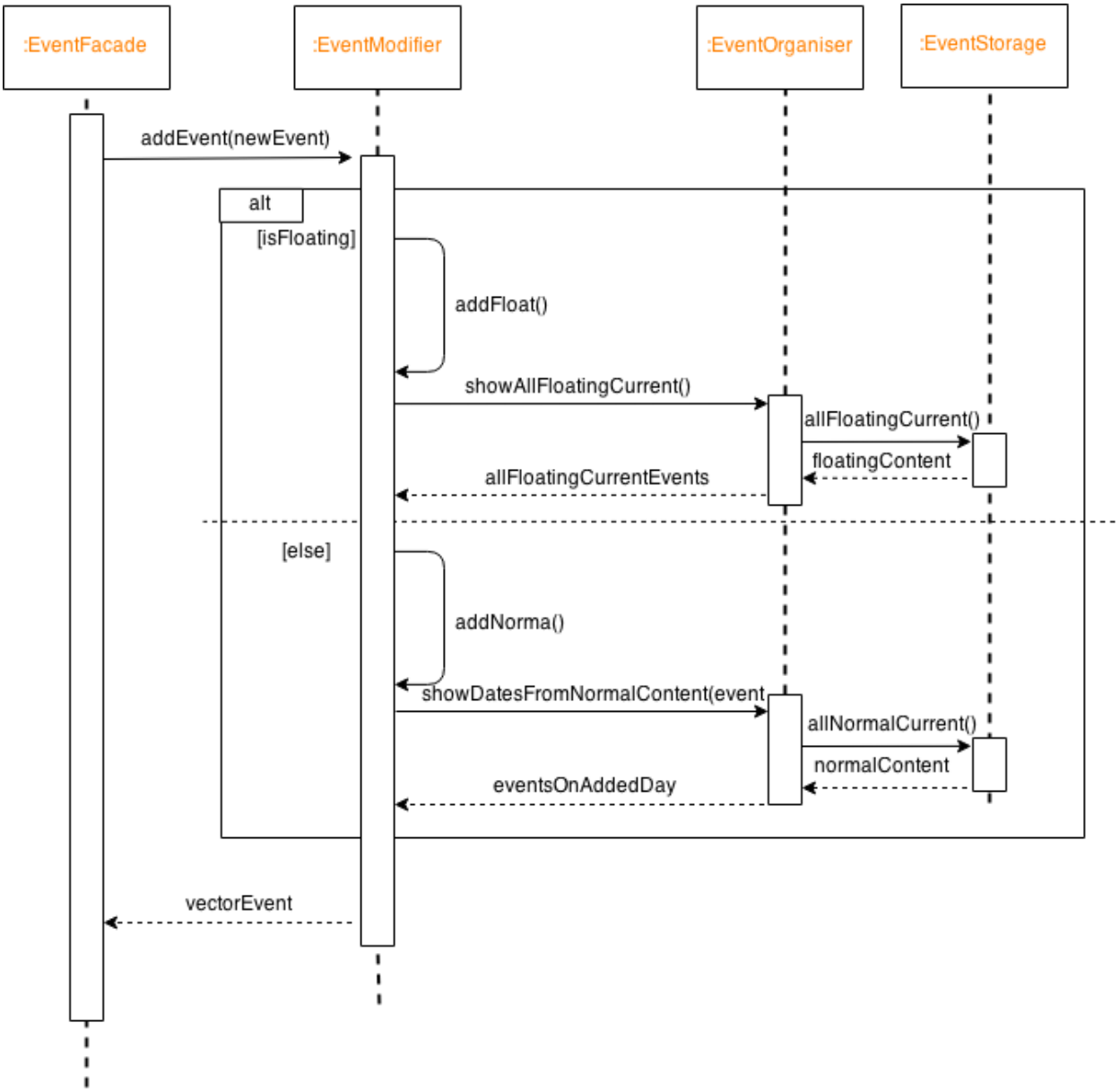
EventStorage

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.