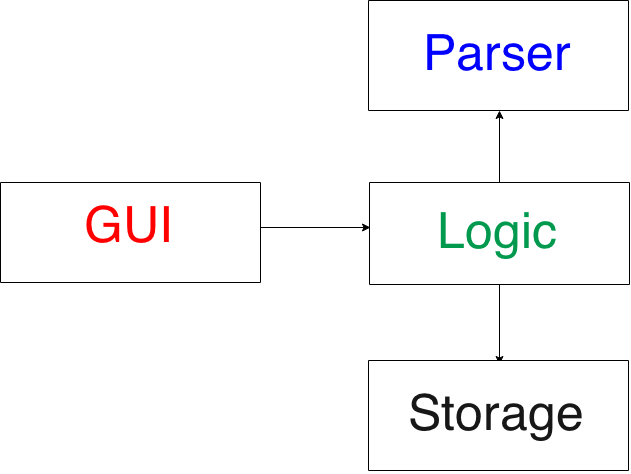
**MapleSyrup Developer Guide**

Table of Contents

1. Overview of Architecture .................................................................................................... 1
2. GUI  .................................................................................................................................... 3
   1. UI
   2. CommandSuggestion
3. Logic  .................................................................................................................................. 4
   1. Logic
   2. Display
4. Parser ................................................................................................................................. 5
   1. Parser
   2. InputStringSplit
   3. ParserProcessor
5. Storage ............................................................................................................................... 6
   1. EventStorage
   2. Search
   3. EventArchive
6. Global Classes  .................................................................................................................. 7
   1. Conversion
   2. Event
7. Sequence Diagrams .......................................................................................................... 7
   1. Add ........................................................................................................................ 7
   2. Delete  ................................................................................................................... 8
   3. Edit  ....................................................................................................................... 9
8. Important APIs .................................................................................................................. 10
   1. GUI
   2. Logic
   3. Parser
   4. InputStringSplit
   5. ParserProcessor
   6. EventStorage
9. Appendix  .......................................................................................................................... 11
   1. Sequence Diagrams
   2. APIs
10. **Overview of Architecture**

MapleSyrup utilizes a transaction processing architecture style with 4 components. They are the GUI, 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. All tasks/events entered by the user are saved as Event objects which may be passed between components (except GUI). This is illustrated in the overall architecture diagram on the right.

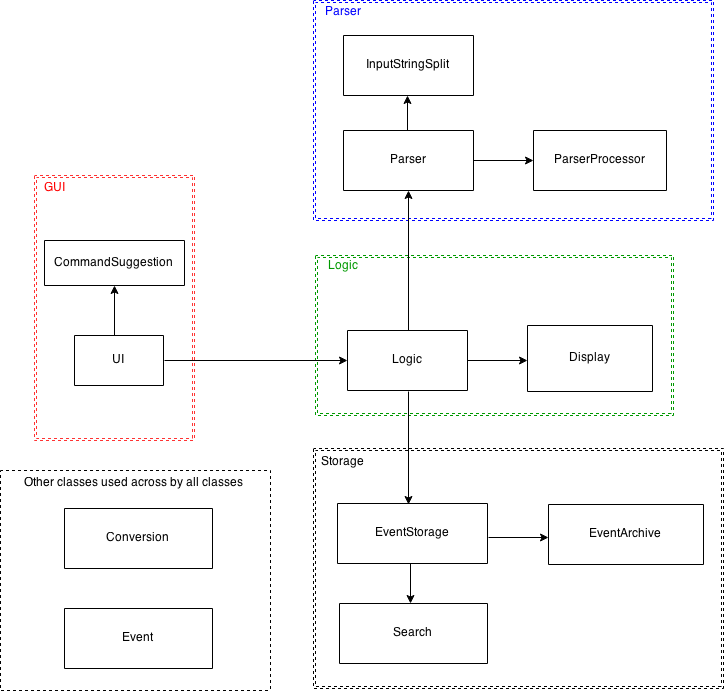
**GUI:** Users interact only with the GUI mainly in the form of text input, though there are certain buttons that provide alternatives to text input. Text typed in the command bar will be detected and the appropriate command suggestion will be received from CommandSuggestion class, and displayed to the user.  The GUI captures user input and conveys it to Logic through a one-point contact, and receives a cue from Logic through this same point. It moves on to obtain the specific data from Logic based on this cue and displays them to the user.

**Logic:** This acts as a facade shielding the more complex implementations from the GUI. It comprises the Logic class which interacts with Parser and Storage to implement the actual user command, and the Display class which stores all output required to be displayed by the GUI.

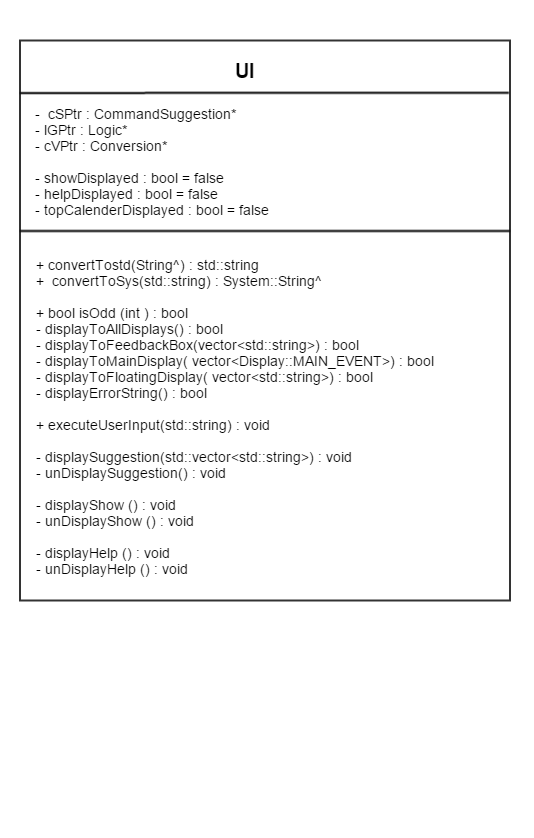
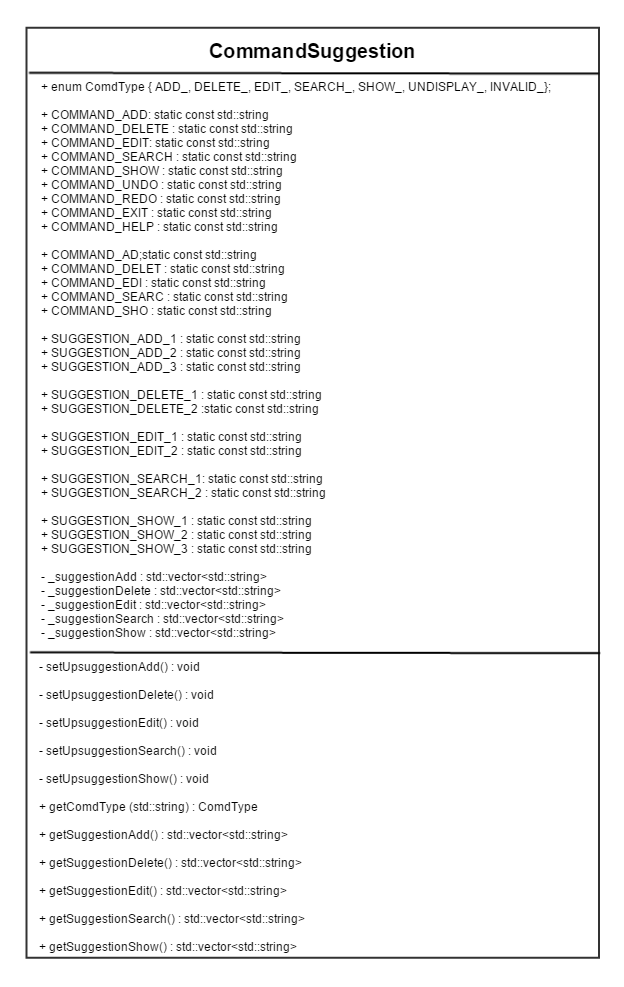
**Parser:** Parser translates raw user input into commands intelligible by the program by converting inputs into Event objects. This is done by passing inputs through classes ParserProcessor and InputStringSplit. Generally, Parser is able to track an Event and a command (which are part of its attributes). Logic gets this information through Parser's API, then uses it to call the APIs from Storage to execute the desired command.

**Storage:** In Storage, the class EventStorage keeps track of a vector of Events that have been entered by the user but not deleted. All actual adding, deleting and editing of Events is implemented by manipulating this vector. The information in the vector is written to a text file after every command is executed, and read from the text file whenever the program is started. The EventArchive class stores similar information but with additional information for command type. This is to facilitate undo operations. Finally, the Search class implements various search functions required by EventStorage to find certain Events.

**Detailed Architecture Diagram**

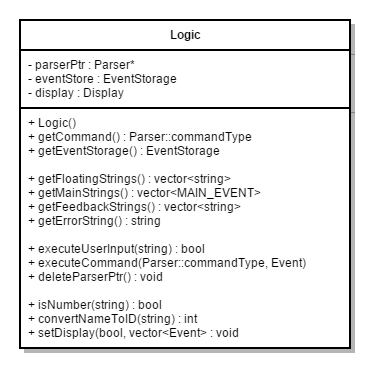


**2. GUI**

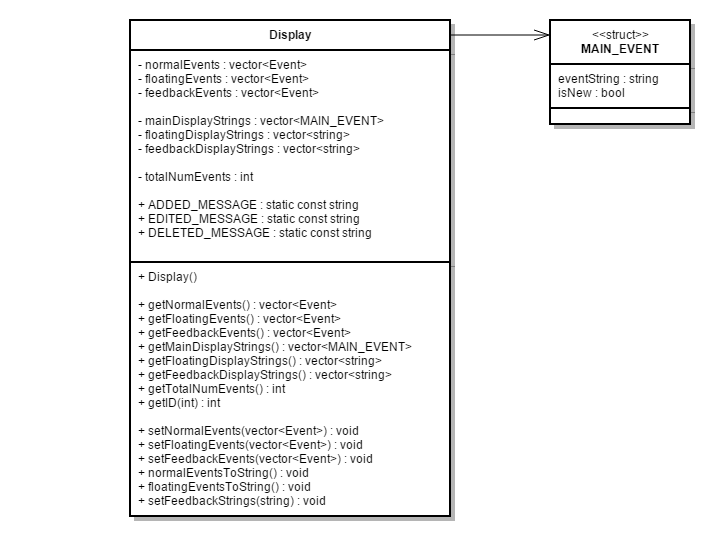
**2.1 UI 2.2 CommandSuggestion**

**3. Logic**

**3.1 Logic**

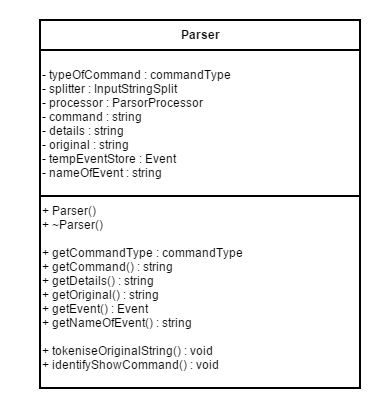


**3.2 Display**

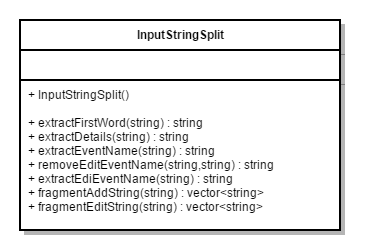


**4. Parser**

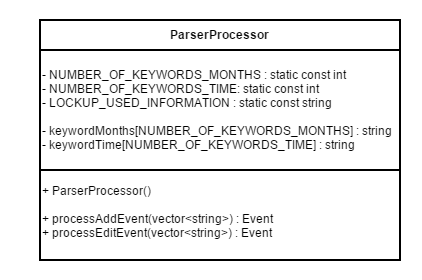
**4.1 Parser**

****

**4.2 InputStringSplit**

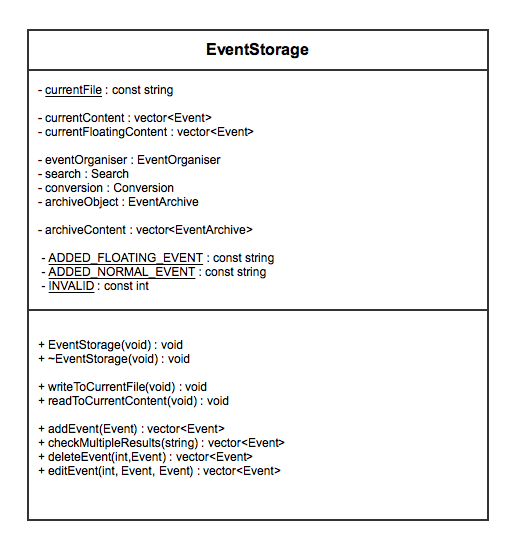
****

**4.3 ParserProcessor**

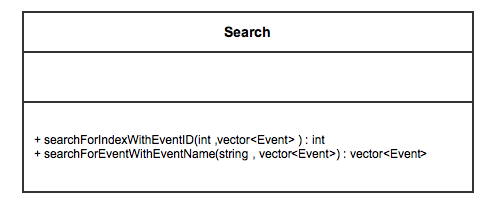
****

**5. Storage**

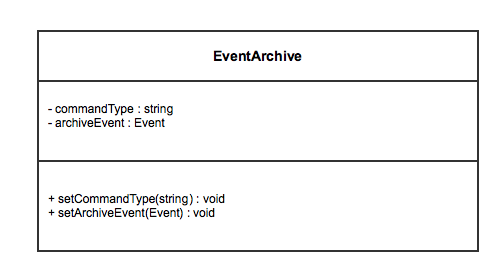
**5.1 EventStorage**

****

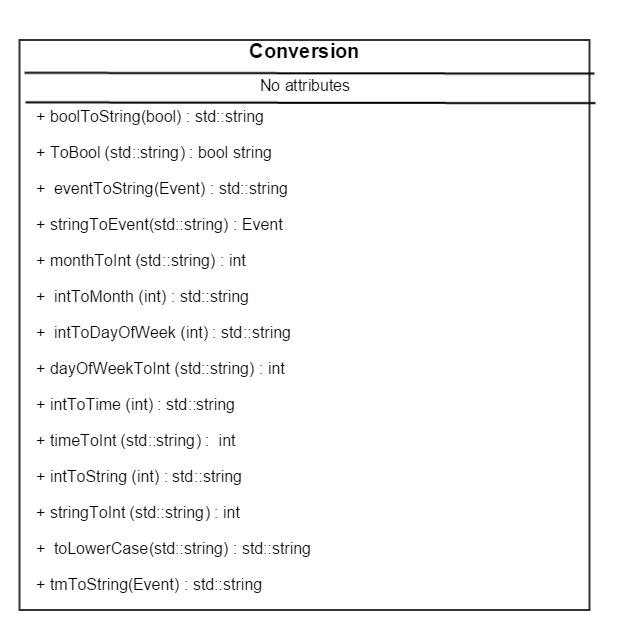
**5.2 Search**

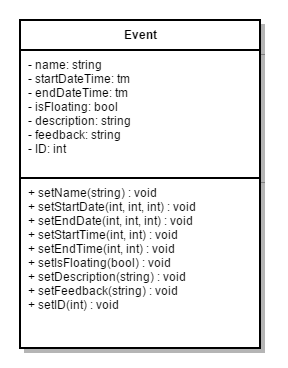
****

**5.3 EventArchive**

****

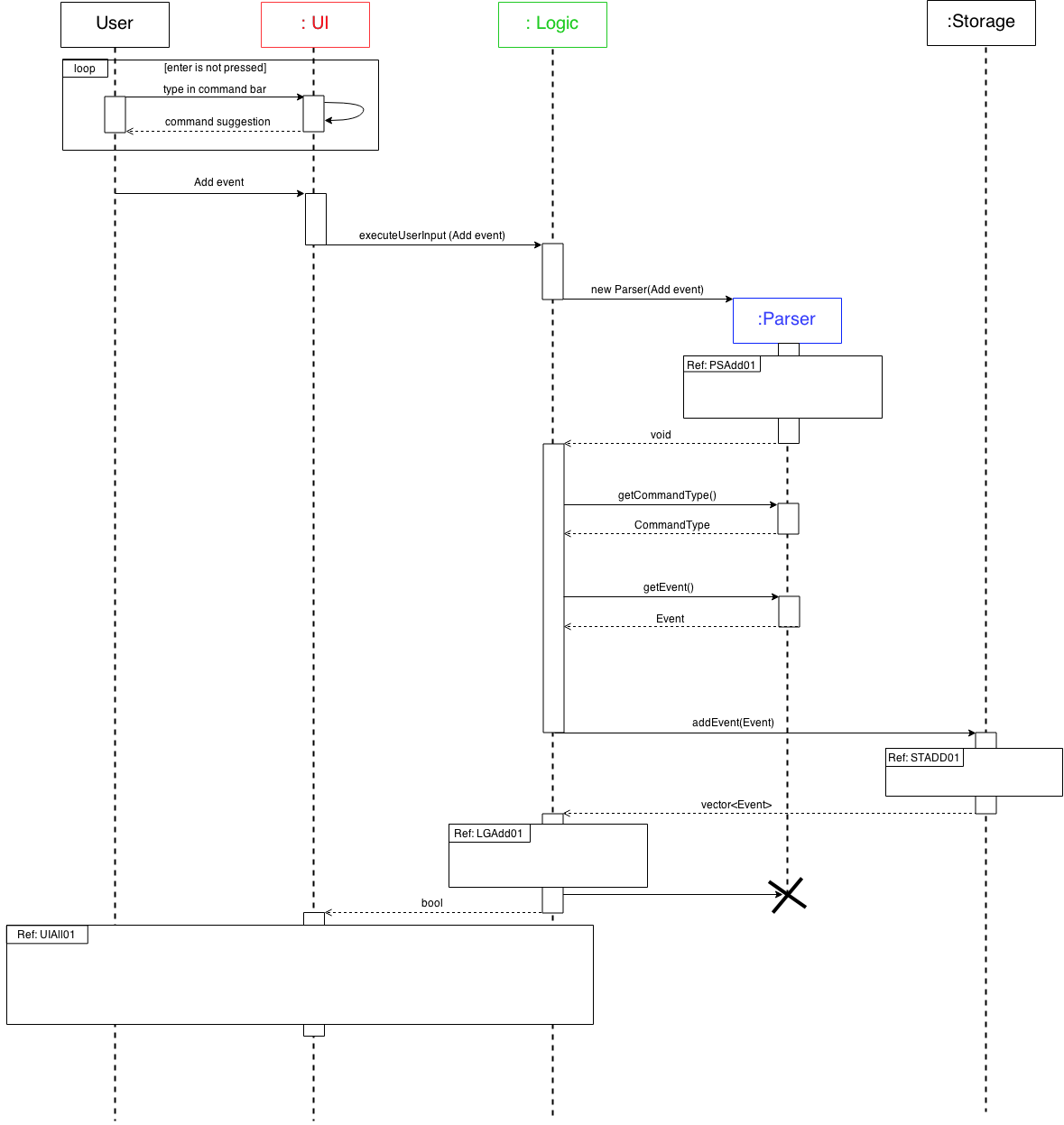
**6. Global Classes**

**6.1 Conversion 6.2 Event**

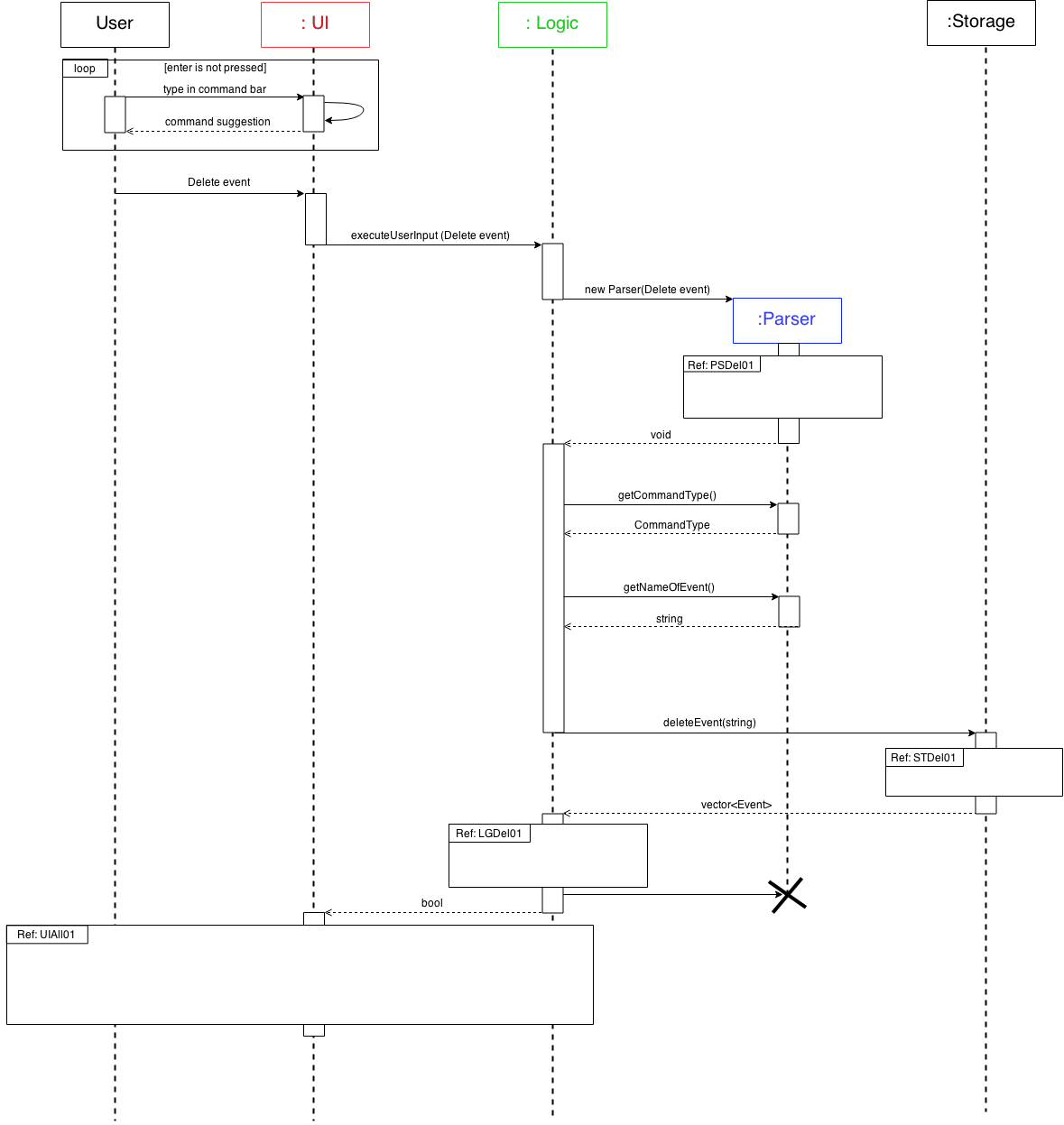
****

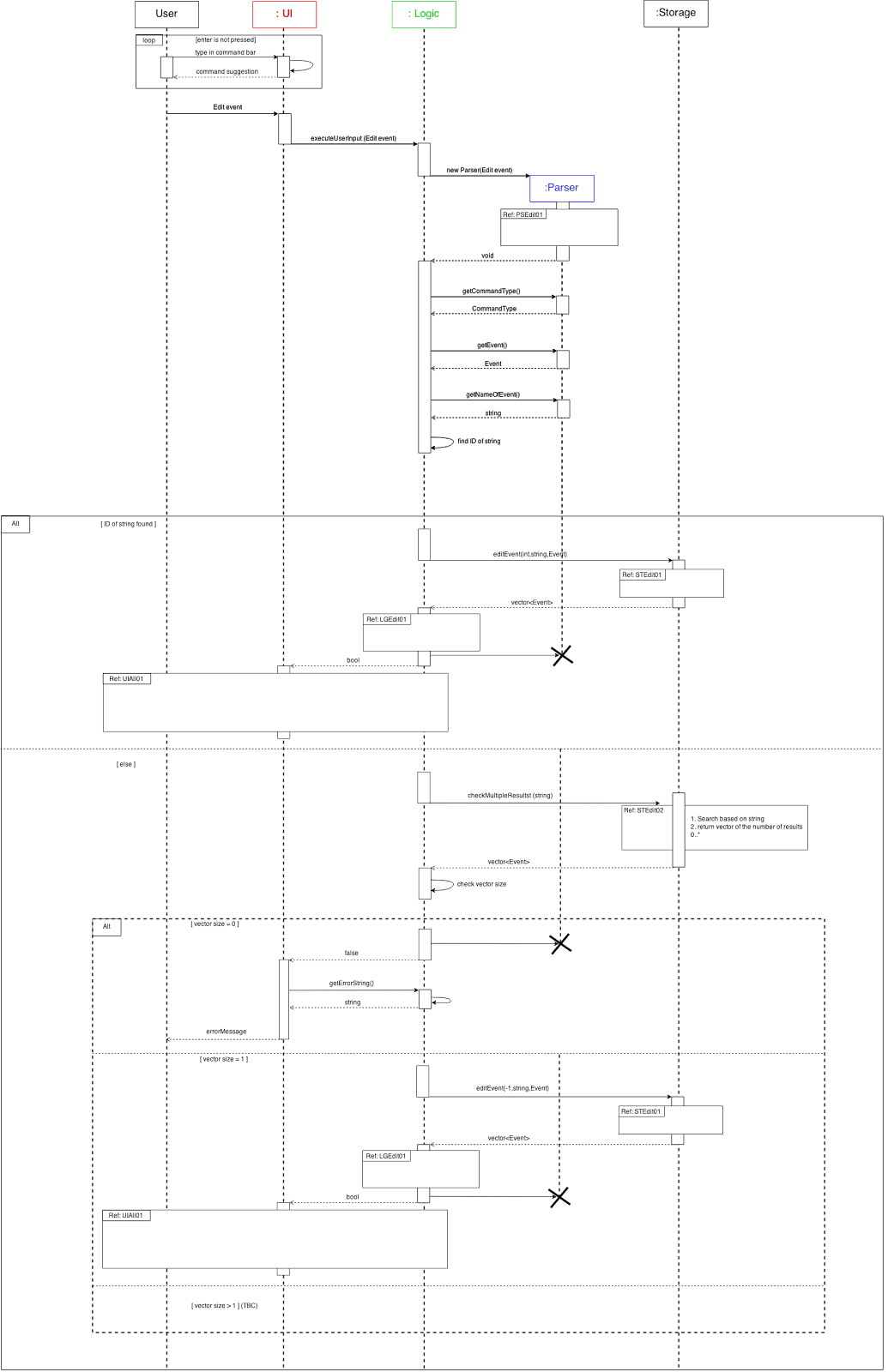
**7. Sequence Diagrams**

**7.1 ADD**

****

**7.2 DELETE**

****

**7.3 EDIT8. 8. Important APIs**

**8.1 GUI**

void executeUserInput(string input): Centralises all calls from various parts/event handlers from the UI to Logic for execution. Thereafter, based on the boolean variable received from Logic::executeUserInput(), it proceeds to call functions to display the relevant information to the various displays on the GUI.

bool displayToAllDisplays(): Gets display vectors from Logic when invoked by function executeUserInput(), then displays these vectors to main display, floating tasks display and feedback box. Returns true upon successful display.

bool displayErrorString(): Get error string from Logic when invoked by function executeUserInput(), then displays the string on the GUI’s main display. Upon successful display, it returns true to caller.

**8.2 Logic**

bool Logic::executeUserInput(string input): Called with the exact user input string. Creates Parser object to determine the correct action to take, then calls another Logic::executeCommand to execute this action. Returns true by default; will be able to return false when future error cases are implemented.

void Logic::executeCommand(Parser::commandType command, Event userEvent): Calls APIs from EventStorage to execute the desired command based on the input parameters (e.g. if command = ADDFLOAT, it will call EventStorage to add the floating event denoted by userEvent).

**8.3 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.

**8.4 InputStringSplit**

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

**8.5 ParserProcessor**

Event ParserProcessor::processAddEvent(vector<string> fragmentedWords): 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.

**8.6 EventStorage**

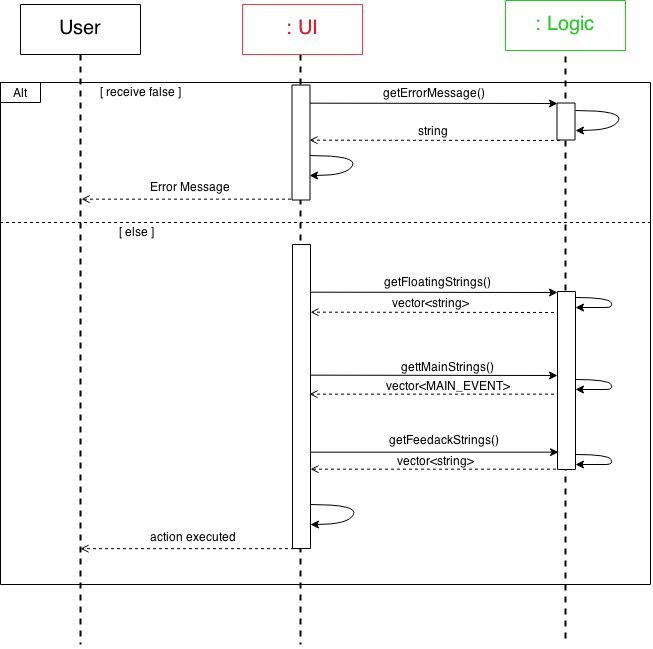
void EventStorage::writeToCurrentFile(): Processes events and stores them in an external storage as strings. Implements commands from Logic (eg, add, delete, edit) using internal vector<Event> (currentContent and currentFloatingContent) followed by saving the new data in the external txt file.

void EventStorage::readToCurrentContent(): Used when EventStorage object is created to read all data from external txt file into internal vector<Event>. To do this, event components that were saved, as strings of text will be individual transferred into an event format and saved in the internal storage (currentContent or currentFloatingContent).

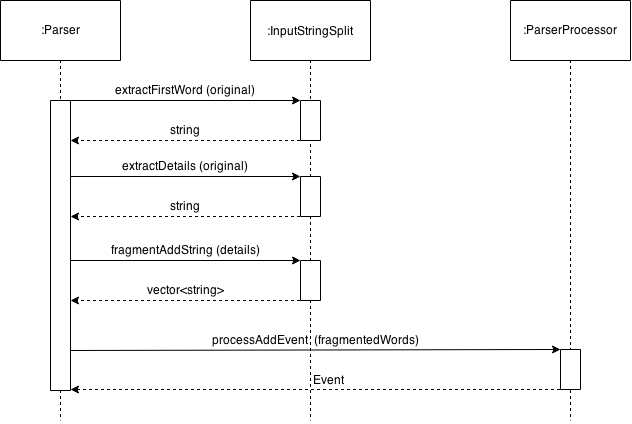
**9. Appendix**

**9.1 Sequence Diagrams**

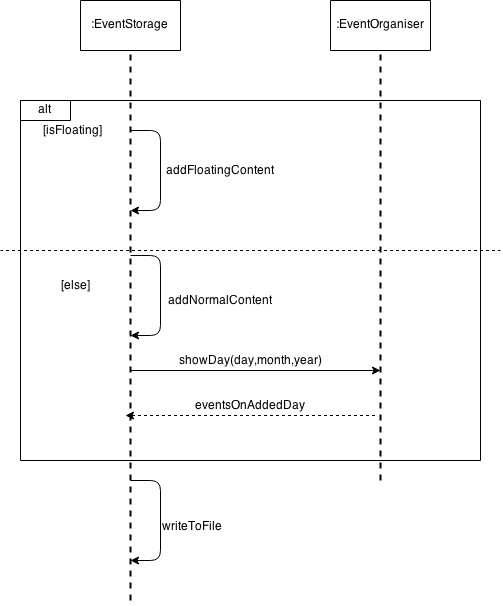
**UIAll01**

****

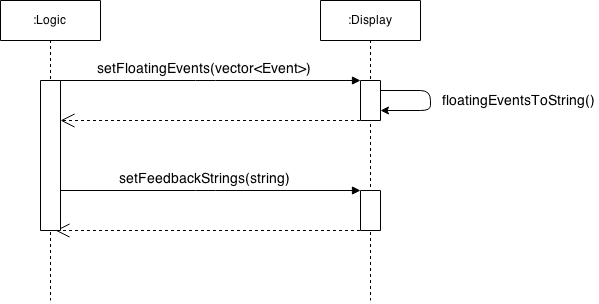
**PSAdd01**



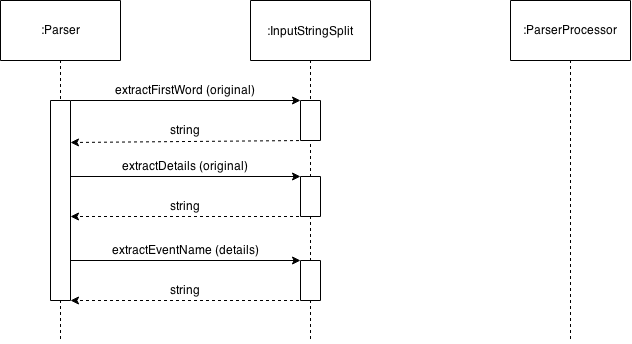
**STADD01**



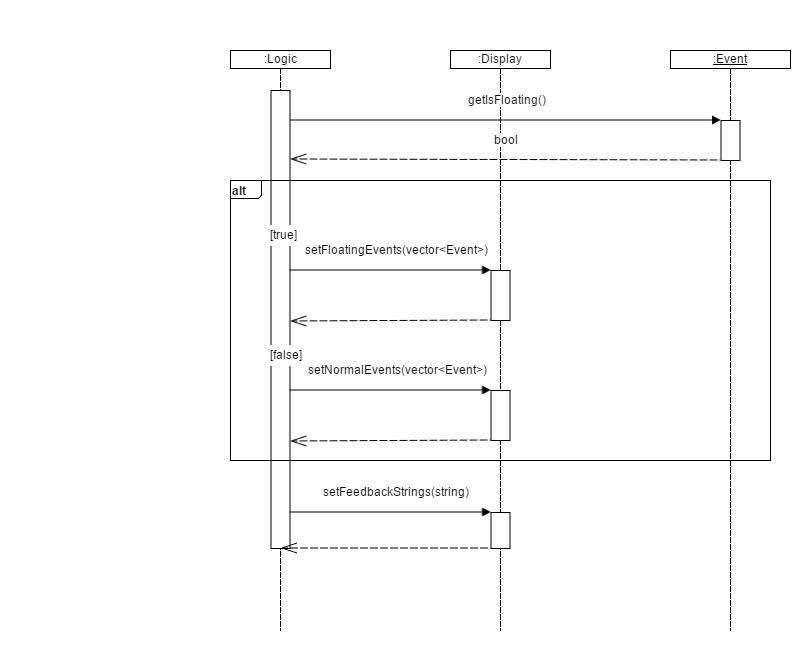
**LGAdd01**



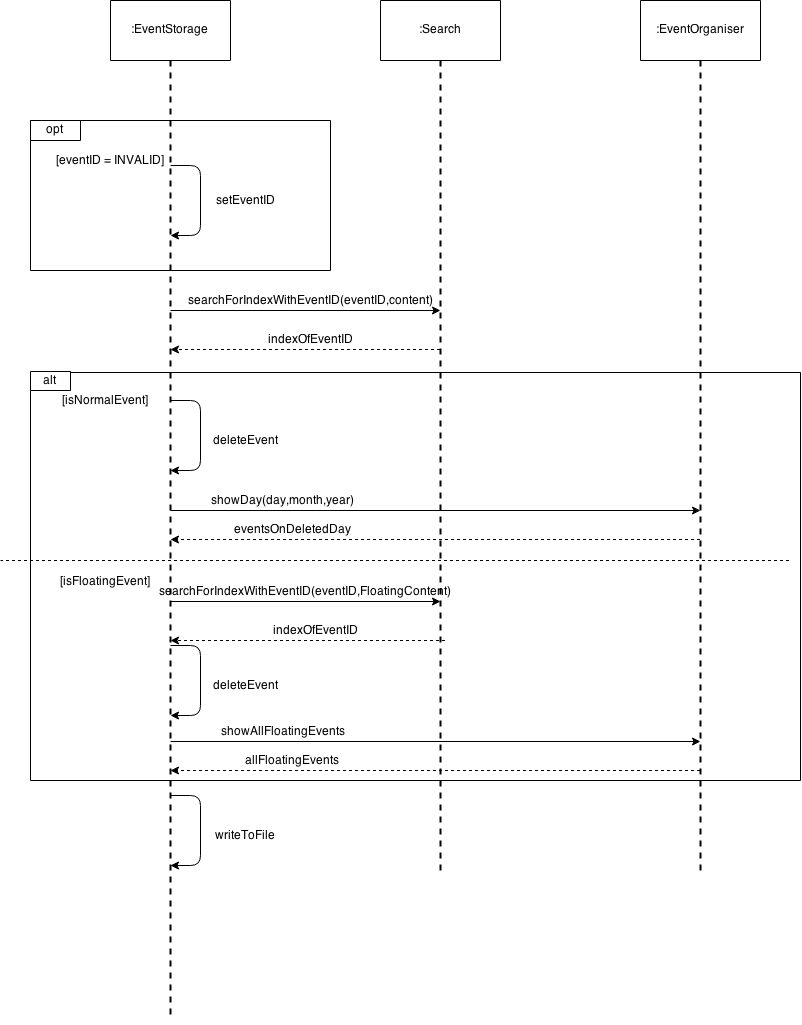
**PSDel01**

****

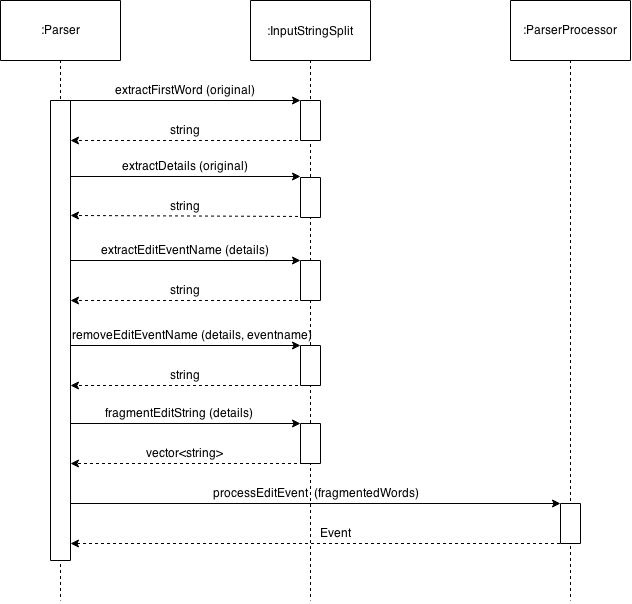
**LGDel01**

****

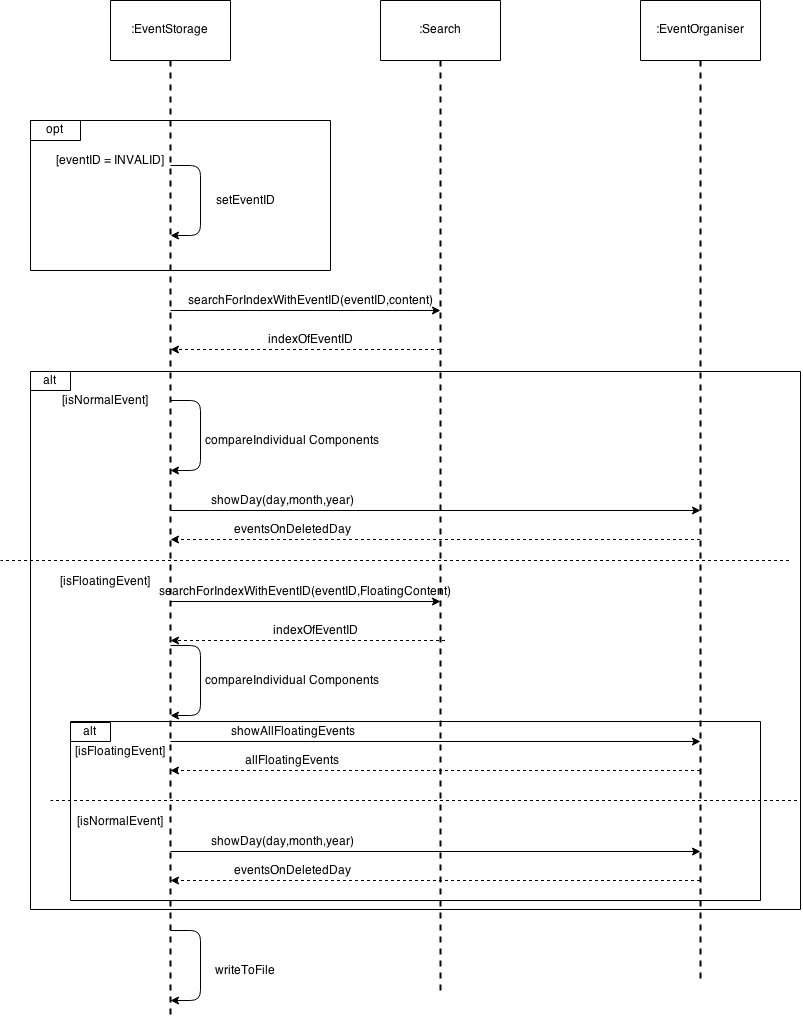
**STDel01**

****

**PSEdit01**



**STEdit01**

****

**9.2 APIs**

**UI**

void executeUserInput(std::string input){

   bool isExecuted = lGPtr->executeUserInput(input);

   bool isAllDisplayed;

   bool isErrorStringDisplayed;

   if(isExecuted){

          isAllDisplayed = displayToAllDisplays();

   } else {

          isErrorStringDisplayed = displayErrorString();

   }

   // if (isAllDisplayed && isErrorStringDisplayed), Error

   //Else carry on

}

bool displayToAllDisplays(){

   vector<std::string> displayToFloating = lGPtr->getFloatingStrings();

   vector<Display::MAIN\_EVENT> displayToMain = lGPtr->getMainStrings();

   vector<std::string> displayToFeedback = lGPtr-> getFeedbackStrings();

   bool checkAllDisplayed;

   bool displayFeedback = displayToFeedbackBox (displayToFeedback);

   bool displayFloating = displayToFloatingDisplay (displayToFloating);

   bool displayMain = displayToMainDisplay (displayToMain);

   if (!displayFeedback || !displayFloating || !displayMain){

          checkAllDisplayed = false;

   } else {

          checkAllDisplayed = true;

   }

   return checkAllDisplayed;

}

bool displayErrorString(){

   bool isErrorStringShown;

   std::string tempErrorString = lGPtr->getErrorString();

   String^ errorString = convertToSys(tempErrorString);

   display->Text = errorString;

   //if there is error, isErrorStringShown = false;

   isErrorStringShown = true;

   return isErrorStringShown;

}

**Logic**

bool Logic::executeUserInput(string input) {

   parserPtr = new Parser(input);

   Parser::commandType command = getCommand();

   Event userEvent = getEvent();

   executeCommand(command, userEvent);

   deleteParserPtr();

   return true;

}

void Logic::executeCommand(Parser::commandType command, Event userEvent) {

   string eventName = parserPtr->getNameOfEvent();//can be index in integer form (e.g. "1") or actual name of event (e.g. "lunch")

   int index, id;

   vector<Event> tempEvents;

   switch (command) {

   case Parser::ADDFLOAT: {

          display.setFloatingEvents(eventStore.addEvent(userEvent));

          display.setFeedbackStrings(userEvent.getName() + ADDED\_MESSAGE);

          break;

                                         }

   case Parser::DELETE\_: {

          if (isNumber(eventName)) {

                  index = std::stoi(eventName);

                  id = display.getID(index);

                  eventName = display.getEventName(index);

          } else {

                  id = INVALID\_NUMBER;

          }

          tempEvents = eventStore.deleteEvent(id, eventName);

          bool isFloat = tempEvents[0].getIsFloating();

          if (isFloat) {

                  display.setFloatingEvents(tempEvents);

          } else {

                  display.setNormalEvents(tempEvents);

          }

          display.setFeedbackStrings(eventName + DELETED\_MESSAGE);

          break;

                                        }

   case Parser::EDIT: {

          Event tempEvent = parserPtr->getEvent();

          if (isNumber(eventName)) {

                  index = std::stoi(eventName);

                  id = display.getID(index);

                  eventName = display.getEventName(index);

          } else {

                  id = INVALID\_NUMBER;

          }

          tempEvents = eventStore.editEvent(id, eventName, tempEvent);

          bool isFloat = tempEvents[0].getIsFloating();

          if (isFloat) {

                  display.setFloatingEvents(tempEvents);

          } else {

                  display.setNormalEvents(tempEvents);

          }

          display.setFeedbackStrings(userEvent.getName() + EDITED\_MESSAGE);

          break;

                                  }

   default:

          break;

   }

}

**Parser**

void Parser::tokenizeOriginalString(){

   command = splitter.extractFirstWord(original);

   details = splitter.extractDetails(original);

   std::vector<std::string> fragmentedWords;

   if(command == "add"){

          fragmentedWords = splitter.fragmentAddString(details);

          tempEventStore = processor.processAddEvent(fragmentedWords);

          if(tempEventStore.getIsFloating() == true){

                  typeOfCommand = Parser::ADDFLOAT;

          } else {

                  typeOfCommand = Parser::ADD;

          }

   }

   else if(command == "delete"){

          nameOfEvent = splitter.extractEventName(details);

          typeOfCommand = Parser::DELETE\_;

   }

   else if(command == "edit"){

          nameOfEvent = splitter.extractEditEventName(details);

          details = splitter.removeEditEventName(details,nameOfEvent);

          fragmentedWords = splitter.fragmentEditString(details);

          tempEventStore = processor.processEditEvent(fragmentedWords);

          typeOfCommand = Parser::EDIT;

   }

   return;

}

**InputStringSplit**

std::vector<std::string> InputStringSplit::fragmentAddString(std::string input){

   std::string::size\_type strCutIndex;

   std::vector<std::string> fragmentedWords;

   std::string tempString;

   bool endOfString = false;

   strCutIndex = input.find\_first\_of(";"); // ; indicates end of event name

   tempString = input.substr(0,strCutIndex);

   fragmentedWords.push\_back(tempString + ";");

   strCutIndex = input.find\_first\_not\_of(" -.;",strCutIndex);

   if(strCutIndex == std::string::npos){

          endOfString = true;

   }

   while(!endOfString){

          input = input.substr(strCutIndex);

          strCutIndex = input.find\_first\_of(" -.");

          fragmentedWords.push\_back(input.substr(0,strCutIndex));

          strCutIndex = input.find\_first\_not\_of(" -.",strCutIndex);

          if(strCutIndex == std::string::npos){

                  endOfString = true;

          }

   }

   return fragmentedWords;

}

**ParserProcessor**

Event ParserProcessor::processAddEvent(std::vector<std::string> fragmentedWords){

   Conversion convertor;

   Event tempEventStore;

   std::string strMonth;

   int tempInt;

   int day = 0, month = 0, year = 0, hour = 0, minute = 0;

   bool matchFound = false;

   bool startDayFound = false;

   bool endDayFound = false;

   bool startTimeFound = false;

   bool endTimeFound = false;

   bool afterTwelve = false;

   bool nameFound = false;

   int tempi = 0;

   unsigned int i;

   //finding all the names of event

   for(i = 0; i < fragmentedWords.size() && !nameFound; i++){

          if(fragmentedWords[i].find(";") != std::string::npos){

                  tempEventStore.setName(fragmentedWords[i].substr(0,fragmentedWords[i].find\_last\_of(";")));

                  tempi++;

                  nameFound = true;

          }

   }

   int j;

   for(i = tempi; i < fragmentedWords.size(); i++){

          //finding date

          for (j = 0; j < NUMBER\_OF\_KEYWORDS\_MONTHS && !matchFound; j++){

                  if(fragmentedWords[i].find(keywordMonths[j]) != std::string::npos){

                         matchFound = true;

                         tempi = i;

                         strMonth = keywordMonths[j];

                  }

          }

          if(matchFound){

                  try {

                         auto tempStoi = std::stoi(fragmentedWords[tempi]);

                         fragmentedWords[tempi] = LOCKUP\_USED\_INFORMATION;

                         tempInt = tempStoi;

                  } catch (const std::invalid\_argument& e){

                         tempi--;

                         auto tempStoi = std::stoi(fragmentedWords[tempi]);

                         fragmentedWords[tempi] = LOCKUP\_USED\_INFORMATION;

                         tempInt = tempStoi;

                  }

                  day = tempInt;

                  month = convertor.monthToInt(strMonth);

                  year = 2015-1900;

                  if(!startDayFound){

                         startDayFound = true;

                         tempEventStore.setStartDate(day,month,year);

                  } else {

                         endDayFound = true;

                         tempEventStore.setEndDate(day,month,year);

                  }

          }

          matchFound = false;

          //finding time

          for (j = 0; j < NUMBER\_OF\_KEYWORDS\_TIME && !matchFound; j++){

                  if(fragmentedWords[i].find(keywordTime[j]) != std::string::npos){

                         matchFound = true;

                         tempi = i;

                         if(keywordTime[j] == "pm"){

                                afterTwelve = true;

                         }

                  }

          }

          if(matchFound){

                  try {

                         auto tempStoi = std::stoi(fragmentedWords[tempi]);

                         fragmentedWords[tempi] = LOCKUP\_USED\_INFORMATION;

                         tempInt = tempStoi;

                  } catch (const std::invalid\_argument& e){

                         tempi--;

                         auto tempStoi = std::stoi(fragmentedWords[tempi]);

                         fragmentedWords[tempi] = LOCKUP\_USED\_INFORMATION;

                         tempInt = tempStoi;

                  }

                  if(tempInt >= 100){

                         minute = tempInt%100;

                         if(afterTwelve){

                                hour = tempInt/100 + 12;

                         } else {

                                hour = tempInt/100;

                         }

                         if(!startTimeFound){

                                tempEventStore.setStartTime(hour,minute);

                                startTimeFound = true;

                         }

                         else {

                                tempEventStore.setEndTime(hour,minute);

                                endTimeFound = true;

                         }

                  } else if(tempInt < 100){

                         tempi--;

                         try {

                                hour = std::stoi(fragmentedWords[tempi]);

                                fragmentedWords[tempi] = LOCKUP\_USED\_INFORMATION;

                                minute = tempInt;

                         } catch (const std::invalid\_argument& e){

                                hour = tempInt;

                                minute = 0;

                         }

                         if(afterTwelve){

                                hour = hour + 12;

                         }

                         if(!startTimeFound){

                                startTimeFound = true;

                                tempEventStore.setStartTime(hour,minute);

                         }

                         else {

                                endTimeFound = true;

                                tempEventStore.setEndTime(hour,minute);

                         }

                  }

          }

          matchFound = false;

   }

   if(!startDayFound && !startTimeFound){

          tempEventStore.setIsFloating(true);

   }

   if(startDayFound && !startTimeFound){

          tempEventStore.setStartTime(0,0);

          tempEventStore.setEndTime(23,59);

   }

   if(!startDayFound && startTimeFound){

          time\_t t = time(0);

          struct tm\* now = localtime(&t);

      day = now->tm\_mday;

          month = now->tm\_mon;

          year = now->tm\_year;

          tempEventStore.setStartDate(day,month,year);

          tempEventStore.setEndDate(day,month,year);

   }

   if(!endDayFound){

          tempEventStore.setEndDate(day,month,year);

   }

   if(!endTimeFound){

          tempEventStore.setEndTime(hour+1,minute);

   }

   if(endDayFound && !endTimeFound){

          tempEventStore.setEndTime(23,59);

   }

   struct tm\* temptmPtr;

   temptmPtr = &tempEventStore.getStartDate();

   mktime(temptmPtr);

   tempEventStore.setStartDate(temptmPtr->tm\_mday,temptmPtr->tm\_mon,temptmPtr->tm\_year);

   tempEventStore.setStartTime(temptmPtr->tm\_hour,temptmPtr->tm\_min);

   temptmPtr = &tempEventStore.getEndDate();

   mktime(temptmPtr);

   tempEventStore.setEndDate(temptmPtr->tm\_mday,temptmPtr->tm\_mon,temptmPtr->tm\_year);

   tempEventStore.setEndTime(temptmPtr->tm\_hour,temptmPtr->tm\_min);

   return tempEventStore;

}

**EventStorage**

void EventStorage::writeToCurrentFile(){

   std::ofstream writeFile(currentFile);

   for(auto i=0;i<currentContent.size();i++){

          writeFile

                  << conversion.boolToString(currentContent[i].getIsFloating()) << std::endl

                  << currentContent[i].getName() << std::endl

                  << conversion.tmToString(currentContent[i]) << std::endl

                  << currentContent[i].getDescription() << std::endl

                  << currentContent[i].getFeedback() << std::endl

                  << currentContent[i].getID() << std::endl;

   }

   for(auto i=0;i<currentFloatingContent.size();i++){

          writeFile

                  << conversion.boolToString(currentFloatingContent[i].getIsFloating()) << std::endl

                  << currentFloatingContent[i].getName() << std::endl

                  << '\n' << '\n' << '\n' << '\n' << '\n' << '\n' << '\n' << '\n' << '\n' << std::endl

                  << currentFloatingContent[i].getDescription() << std::endl

                  << currentFloatingContent[i].getFeedback() << std::endl

                  << currentFloatingContent[i].getID() << std::endl;

   }

   writeFile.close();

}

void EventStorage::readToCurrentContent(){

   std::ifstream readFile(currentFile);

   std::string textLine, name, description, feedback, tags, startDateYear, startDateMonth, startDateDay, startDateHour, startDateMin, endDateYear, endDateMonth, endDateDay, endDateHour, endDateMin, id;

   getline(readFile, textLine);

   while(!readFile.eof()){

          Event\* tempEvent = new Event;

          if(textLine == "0"){      //Normal case

                  //getinfo from textfile;

                  getline(readFile, name);

                  //cin event times

                  getline(readFile, startDateYear);

                  getline(readFile, startDateMonth);

                  getline(readFile, startDateDay);

                  getline(readFile, startDateHour);

                  getline(readFile, startDateMin);

                  getline(readFile, endDateYear);

                  getline(readFile, endDateMonth);

                  getline(readFile, endDateDay);

                  getline(readFile, endDateHour);

                  getline(readFile, endDateMin);

                  getline(readFile, description);

                  getline(readFile, feedback);

                  getline(readFile, id);

                  //createEvent

                  tempEvent->setIsFloating(false); //stringToBool

                  tempEvent->setName(name);

                  tempEvent->setDescription(description);

                  tempEvent->setFeedback(feedback);

                  tempEvent->setID(atoi(id.c\_str()));

                  tempEvent->setStartTime(atoi(startDateHour.c\_str()),atoi(startDateMin.c\_str()));

                  tempEvent->setStartDate(atoi(startDateDay.c\_str()),atoi(startDateMonth.c\_str()),atoi(startDateYear.c\_str()));

                  tempEvent->setEndTime(atoi(endDateHour.c\_str()),atoi(endDateMin.c\_str()));

                  tempEvent->setEndDate(atoi(endDateDay.c\_str()),atoi(endDateMonth.c\_str()),atoi(endDateYear.c\_str()));

                  currentContent.push\_back(\*tempEvent);

          } else if(textLine == "1"){      //floatingEvent

                  //getinfo from textfile

                  getline(readFile, name);

                  getline(readFile, startDateYear);

                  getline(readFile, startDateMonth);

                  getline(readFile, startDateDay);

                  getline(readFile, startDateHour);

                  getline(readFile, startDateMin);

                  getline(readFile, endDateYear);

                  getline(readFile, endDateMonth);

                  getline(readFile, endDateDay);

                  getline(readFile, endDateHour);

                  getline(readFile, endDateMin);

                  getline(readFile, description);

                  getline(readFile, feedback);

                  getline(readFile, id);

                  //createEvent

                  tempEvent->setIsFloating(true);

                  tempEvent->setName(name);

                  tempEvent->setDescription(description);

                  tempEvent->setFeedback(feedback);

                  tempEvent->setID(atoi(id.c\_str()));

                  currentFloatingContent.push\_back(\*tempEvent);

          }

          delete tempEvent;

          getline(readFile, textLine);//takes in 0/1 for isFloating check

   }

   readFile.close();

}