**Important APIs**

**Logic**

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

bool Logic::executeUserInput(string input) {

parserPtr = new Parser(input);

Parser::commandType command = getCommand();

Event userEvent = getEvent();

executeCommand(command, userEvent);

deleteParserPtr();

return true;

}

void 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). Since there are many possible commands, a switch case is implemented. Only relevant cases are shown below.

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**

Parser component:-

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

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 object:-

std::vector<std::string> InputStringSplit::fragmentAddString(std::string input): Takes in additional details as a complete string from Parser and splits them up into its components by removing spaces and “.-“ symbols, and stores them in a vector of strings. Returns the vector holding the separated components of the input string.

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 object:-

Event ParserProcessor::processAddEvent(std::vector<std::string> fragmentedWords): Using the vector of strings that is the information of the event, it will identify event names, dates and time in their respective formats and store them into an Event class object. Dates and time will be converted from string to integer form. Missing details of the event will be determined and assigned accordingly, and the completed Event will be returned.

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**