[F11C5][V0.2]

Tempus

 Hunar Khanna A0074454A UI coding, documentation, testing	 Raunak Rajpuria A0074900L Logic Coding, Documentation, Implementation	 Ishaan Anil Singal A0078549L Logic Coding, Documentation, Implementation	 Winnie Har A0074750E Deadline watcher, file process coding, testing

[F11C5][V0.2]

Tempus Developer Guide

Welcome to the *Tempus Dev Guide*! The Dev Guide provides a practical introduction to developing *Tempus* further or using *Tempus* to enhance other products. It explores the tools for developing, testing, and publishing the software further.

Please familiarize yourself with the *Tempus* software, by going through the User Manual or by viewing the video tutorial, before going through this guide.

This guide has been divided into several sections:

Tempus Basics
A description of what Tempus is, its functionalities and what it currently has to offer to the users.

Program Architecture
A description of the architecture of the whole program, and the interactions of different components

Component Architecture
A description of the functionality and interfaces of each component in the system

Testing
A description of how each component is tested, and how logging, assertions, exception handling and automated testing have been implemented

Change Log
A description of the changes made from the previous version.

Appendix
Contains the specifications of all the classes in the system.

Tempus Basics

What is Tempus?

Tempus is a free and user friendly organizer that helps the user to schedule activities and events. It has been written in C++.

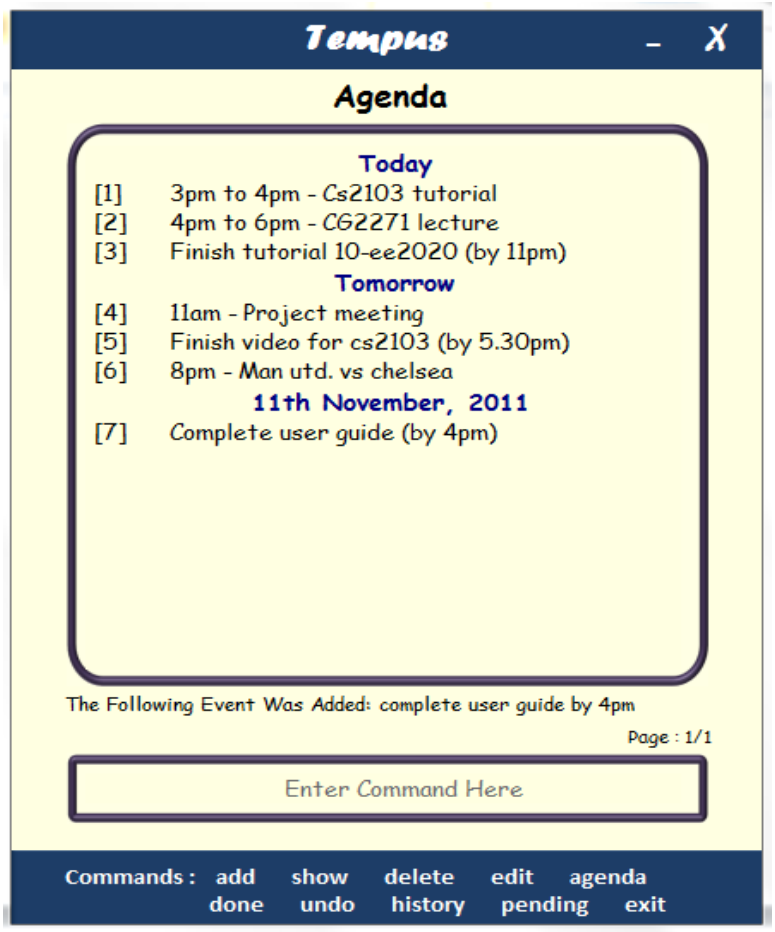
What's New in V0.2?

The GUI has been improved to make it more user-friendly. A number of features have been added. A brief summary of the features in Tempus V0.2 is given below.

Features

- Add, edit, delete, reschedule and mark certain events as done
- Search for events by name and location
- Show events for a particular date in future/history
- Show pending, history and agenda events explicitly
- Show free times for particular date
- Quick - activate the program by pressing Ctrl + Alt + S
- Pop up reminders for upcoming events at specific times
- Undo all actions

A screenshot of the main display window of Tempus is given below:



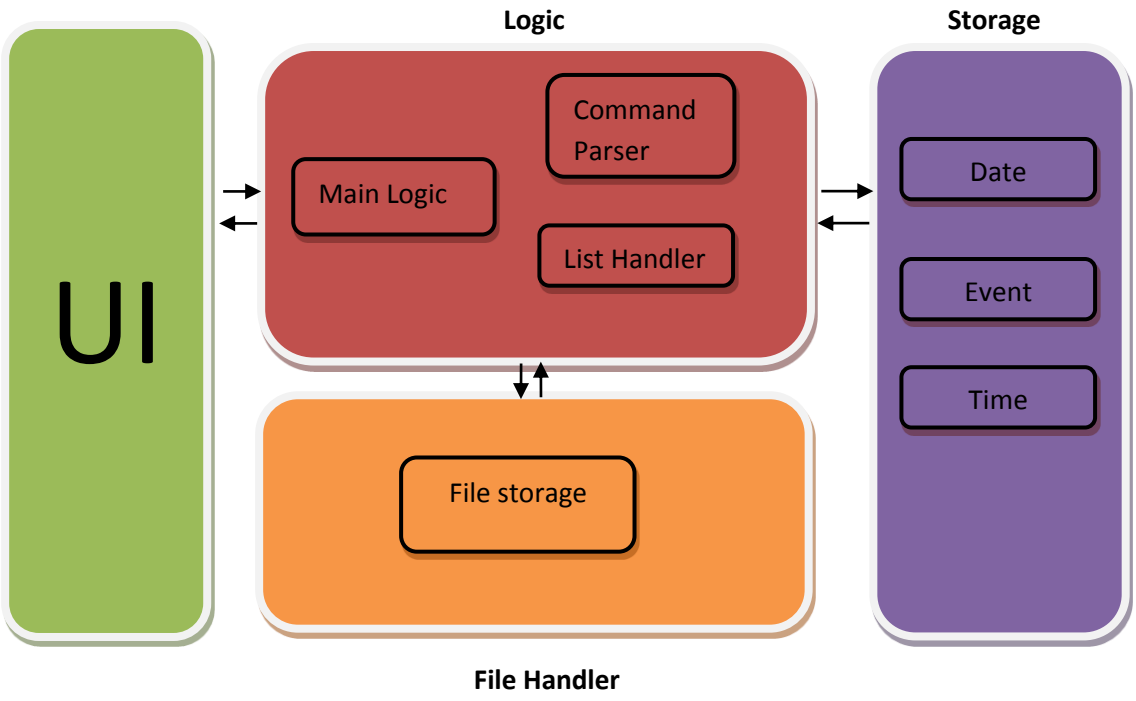
Program Architecture

This section describes the overall structural organization of the system. *Tempus* has been designed in an n-tier architectural style where the higher layers make use of services provided by lower layers, and the functions of the lower layers are independent of layers above them.

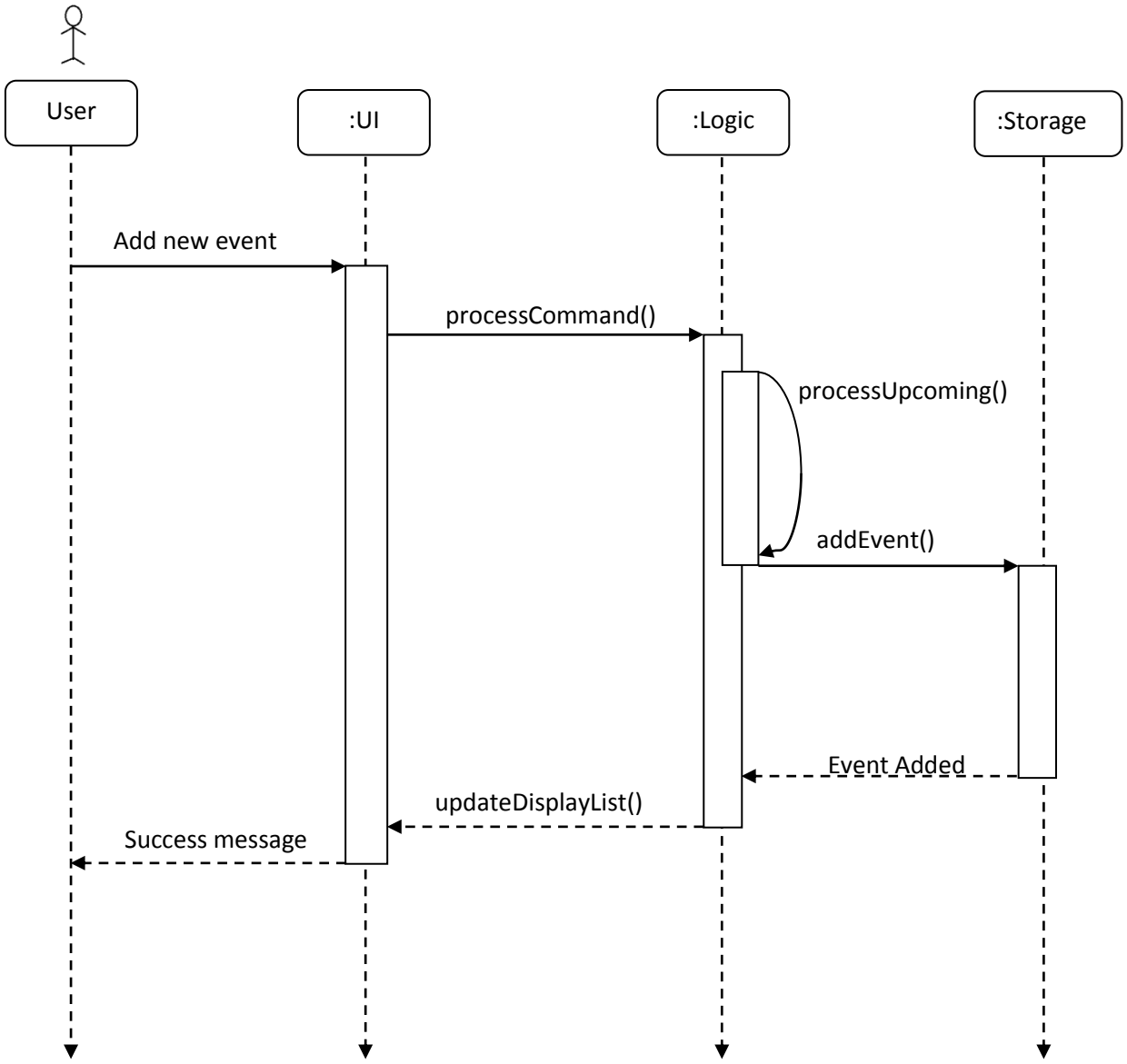
Tempus consists of 4 major components:

- **UI:** Comprises of a powerful GUI. This component interacts with the user, informs the Logic component about the user's request, and displays the formatted output.
- **Logic:** Accepts the user's request from the UI component, validates the command entered, processes the validated command and sends the appropriate result to the UI.
- **Storage:** Stores the list of events internally in system memory. This component is used for data storage during run-time only.
- **File Handler:** Stores the user data in an external file. The data is written to file after every user command, and is retrieved when the application is restarted.

The following diagram shows the Software Architecture and the interaction between components in the system. Each component has been explained in further detail later in this guide



The following diagram shows how the System Components interact when the User adds a new event in Tempus:



Interaction between the components can be described as below:

- **UI** receives the command entered by the user and passes it to **Logic**
- **Logic** interprets the user command and accordingly updates the run-time storage by interacting with **Storage**
- **Logic** also updates the external storage by interacting with the **File Handler**
- **Logic** passes the result of the operation to the **UI**
- **UI** displays the result to the user

Libraries
Only system libraries have been used for the implementation of *Tempus.*
'Gtest' library was used for automated testing of the Logic and Storage components (explained in more detail later).

Component Framework

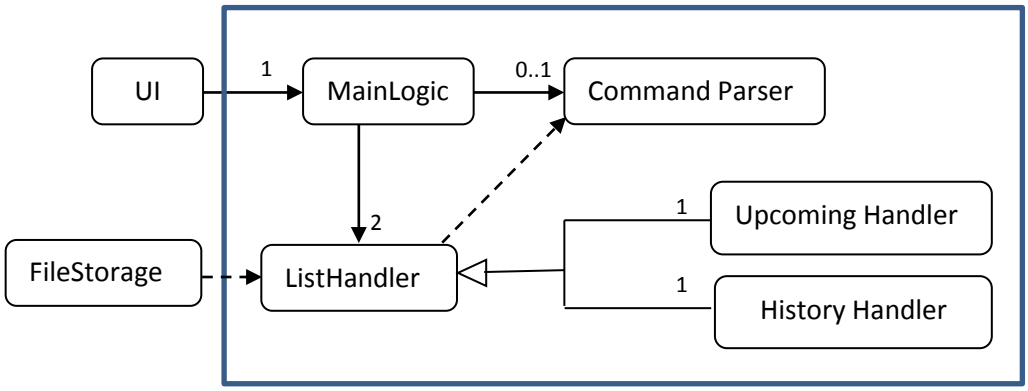
UI

The user's interaction is limited to the UI (class 'Form1'), which is responsible to pass on the user's request to the 'Logic' Layer.

- Class 'Form1.h' generates an interface between the user and the Logic. It also contains the 'main' driver method.

Logic

Class diagram for Logic component



The classes in the Logic component and their functionalities are as follows:

- **MainLogic:** Main Delegator
  - Receives user input from the **UI.**
  - Passes this input to **CommandParser.**
  - Receives the deconstructed command from **CommandParser.**
  - Passes valid command to **ListHandler.**
  - Receives result from **ListHandler.**
  - Formats result to be displayed.
  - Writes to external storage.
  - Returns formatted result to **UI.**

- **CommandParser:**
  - Receives user command from **MainLogic.**
  - Tokenizes command into individual details according to keywords.
  - Stores individual details as attributes.
  - Checks validity of details entered.
  - Returns result of validity check to **MainLogic**

- **ListHandler:**
  - Contains a list that stores all the events, along with a list that keeps track of what events are to be displayed on the screen.
  - Receives details of user command from **MainLogic.**
  - Performs the actions required for different tasks (add, delete, edit, search etc.) by making appropriate changes in the internal storage classes and updating its lists.

- **UpcomingHandler/HistoryHandler:**
  - Subclasses of ListHandler class. They handle past events and upcoming events respectively.
  - They are **singleton** classes. Only one instance of these classes can be created. This instance represents the respective lists.
  - Apart from inheriting the attributes and methods of the parent class, both have certain methods of their own to act on their type of list (i.e History or Upcoming).

Interaction between the classes in the Logic component can be described as below:

- **MainLogic** has two objects of **ListHandler** class. One of them is substituted as **UpcomingHandler** type and the other as **HistoryHandler** type.

- **MainLogic** creates an object of **CommandParser** and passes the user input to it.

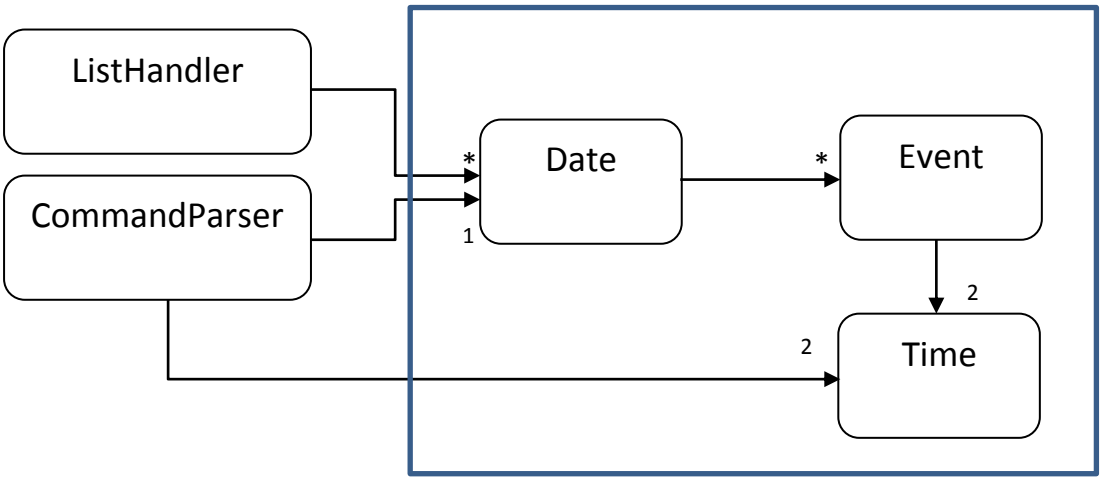
- **CommandParser** tokenizes and validates the input.

- **MainLogic** passes the **CommandParser** object(which now contains the command details) to the **ListHandler** class using the appropriate object(**UpcomingHandler** or **HistoryHandler)**, which performs necessary actions on the internal storage.

- Depending on success or failure, **MainLogic** writes to external storage and passes the appropriate result to the **UI.**

Storage

Class Diagram for Storage



The classes in the Logic component and their functionalities are as follows:

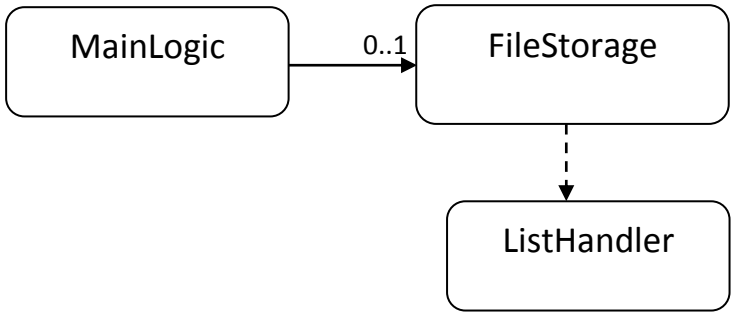
- **Date**:
  - Contains information about a particular date (day, month, year), and stores all the events for that day (in a vector of **Event** objects -> eventList)
  - Receives a call from the **ListHandler** to perform necessary functions on the vector of events
  - Modifies the content of an **Event** object in the event vector based on the inputs from **ListHandler**
  - Contains some helper functions used by itself, and other classes in the upper layer (to get the largest eventId for that day, compare two dates, increment dates etc)
- **Event:**
  - Contains all the information about each event (name, time, location etc)
  - Contains two **Time** objects, startTime and endTime
- **Time:**
  - Contains all the information about a specific time (hour, minutes, am/pm)
  - Contains certain helper functions used by other classes in the upper layer(such as to format the time, get the time in a string form, check whether one time is greater than the other, increment time)

Interaction between the classes in the Logic component can be described as below:

- **Date** can have any number of **Event** objects.
- **Event** has two **Time** objects.
- The classes in this component and their methods are used by the **Logic** component to store the user data internally.

FileHandler

Class Diagram for Database



The classes in the Logic component and their functionalities are as follows:

- **FileStorage:**
  - Contains the methods to read from and write into the external storage file.
  - The method to read the external file is called by **MainLogic** during the startup of the program, and when the current day changes (i.e. when the current time is 12.00am).
  - The method to write to the external file is called when the user exits the program, and every time the user enters a successful command.

Testing

Logging

A log file (logger.txt) is generated as soon as the program starts and the MainLogic writes each action it does onto that file. In case the program crashes or there is an unexpected reaction, the developer can check the logger file and observe which command was successfully executed and which after command the program crashed.

Exception Handling

All the invalid, but possible values that the user can enter are thrown as an error to the respective function calling this method. Hence, all the commands that are executed have been implemented through a try-catch method, as seen from the code fragment below. The test.isValidInput() statement checks the validity of the statement entered by the user and throws and exception if it invalid. Also, the two statements in the 'if' case ie. processPending & processUpcoming also throw errors if there was any discrepancy in the execution of the command. If an error is caught, the statusMessage is modified and displayed accordingly.

```
try{
	test.setInputDetails(command);
	valid=test.isValidInput();

	if(valid)
	{
		if (dtype==PENDING)
			processPending(test);

		else
			processUpcoming(test);
	}
}
catch ( TempusException obj){
	statusMessage = obj.getMessage();
	valid=false;
}
```

In order to throw the exceptions, a helper class has been created with the name "TempusExceptions". This class contains all the possible strings to be thrown.

Assertions

As the architecture has been made in a n-tier top-down style, the lower layers don't test the range of values the layers above passes to them, and assumes that the values are in the right range. In order to make this assumption valid, certain assertions have been places, that check whether the value passed is the same as expected or not. Certain assertions are also placed in the higher layers to make sure that the value returned by lower layers match the expected range.

Defensive Coding

The classes **UpcomingHandler** and **HistoryHandler** have been implemented as Singleton classes. Only one object of these classes can be created. This object is passed around to perform the various operations on the internal lists.

Automated Testing

Although an automated test for the GUI cannot be implemented at this stage, an automated test for the API has been implemented through the use of the 'Gtest' library (that is open source and available for download). A few test methods have been used to check if that actual logic of setting values and their validation is correct or not. Please refer to the other project in the solution, with the name TempusTest to run the tests (or to further add tests).
The code fragment attached below shows a test (tests the time incremental function) that was implemented. Here, the file passTimeincrement.txt contains 3 words on one line: one is the test input, one is the incremental value, and one is the expected output. The EXPECT_EQ function takes two parameters, one is the test value and the other is the expected value. Since only Boolean values can be entered in this EXPECT_EQ function, it is always compared with either true or false.

```
TEST (testing_time_increment, time_sample)
{
	ifstream ifs("passTimeincrement.txt");
	Time timeObj, timeObj2;
	int minutes;
	string input, time1, time2;
	stringstream str;

	while(getline(ifs,input))
	{
		str<<input;
		str>>time1;
		str>>minutes;
		str>>time2;
		timeObj.timeStringToObject(time1);
		timeObj2.timeStringToObject(time2);
		timeObj.incrementTime(minutes);
		EXPECT_EQ(timeObj.isTimeEqual(timeObj2), true);
	}

}
```

Change Log

A number of changes have been implemented in Tempus V0.2 as a part of our constant endeavor to develop a great product. The major ones are listed below:

- The **format** in which the user can enter a command is new. **No symbols** have to be used by the user for any command.
- The **Help Bar** has been improved to make it more user-friendly.
- The user can **undo** all actions.
- A larger number of events can be displayed per page.
- The **Auto-complete** feature has been improved.
- The user can search for **free time** on a particular day
- The concept of priority has been removed.
- The **Reminder** functionality was added.
- The **History** functionality was added. All events before the current date are now moved to a separate list for past events.
- The user can **mark an event as done.**
- The user is now allowed to **edit the date of an event** as well.
- The system writes the content of the internal storage to the external file after every valid user command.

Appendix

Class Specifications- Logic

Class MainLogic

Important Attributes	
ListHandler* upcomingObj	A pointer to object of **ListHandler** class which is initialized as **UpcomingHandler** type. The upcoming list is maintained here.
ListHandler* historyObj	A pointer to object of **ListHandler** class which is initialized as **HistoryHandler** type. The history list is maintained here.
stack<DISPLAYTYPE> dtypeOrder	Stack used for undo, to keep track of the display string user sees
Stack<string> commandOrder	Stack used for undo, to keep track of the commands the user enters
String statusMessage	Stores the success or error message. It is returned to the **UI**
String outputDisplay	Stores the string to be displayed on the screen, used by **UI**
String label	Stores the title string above the output display, used by **UI**
Important Methods	
processCommand (string)	Takes in the user statement from the **UI** as a parameter, calls the relevant functions to test the validity, and calls other methods for further processing
checkIfReminder()	Checks if there are any upcoming events in the next 10 minutes, and returns a bool value to the **UI** accordingly
getReminderTest()	Returns a string to the **UI** that contains the details of the events scheduled for 10 minutes later
readIfNextDay()	Used to check whether the day has changed(ie the current time has just crossed 12am) and updates the history and upcoming lists(as each list stores events from the current date)

Class CommandParser

Important Attributes	
string inputCommand, inputName, inputDate, inputIndex…etc Time startTime, endTime Date dateObj	Attributes that will contain individual details about the statement the user enters. Certain attributes may remain NULL(“ ”) as the user might not enter all details such as location
Important Methods	
setInputDetails (string)	Takes in the user command from **MainLogic**, and sets the details in its attributes (along with identifying the type of command: add, delete etc)
isValidInput()	Tests for the validity of the details that were set from the user statement, and informs the **MainLogic** whether to process the statement further, or show an error message.

Class ListHandler

Important Attributes	
vector<Date>masterList	List that stores all the events in the internal memory, based on the dates (each vector location contains a **Date** object)
vector<Event>displayList	List that stores the events to be displayed on the screen
stack<string>commandOrder	Stack that stores the string of commands entered by the user for undo purposes (like add, delete etc)
stack<Event>eventOrder	Stack that pushes the object each time an operation on an event is done, for undo purposes
Important Methods	
virtual addEvent(CommandParser) virtual deleteEvent(CommandParser) virtual editEvent(CommandParser) virtual undoCommand() virtual showEvents(CommandParser) virtual checkIfClash() virtual showFreeTime(CommandParser)	Methods that are virtual and have not been implemented in the parent class, but in the sub-classes that inherit it (i.e. upcomingHandler and historyHandler in this case)
markAsDone(CommandParser)	Marks a specific event as done based on the index given by a user, or all events as done, if no index is specified
defaultList()	Pushes all the events stored in the masterList to the displayList. Used for displaying Agenda/Archive in upcomingHandler/historyHandler respectively
formatDisplayList()	Formats the details stored in the displayList and returns the string to the **MainLogic**

Class UpcomingHandler

All attributes are inherited from the parent class (**ListHandler**)

Important Methods	
addEvent(CommandParser)	Adds an event in the masterList by: - calculating the index (difference of the date entered from the current system date) - generating a unique eventId - passing the individual information to the **Date** class, to be inserted in the vector
deleteEvent(CommandParser)	Deletes an event from the masterList by: - obtaining the event information from the index user entered - sending the vector index information to delete the event object from the vector to **Date**
editEvent(CommandParser)	Edits an event in the masterList by: - obtaining the event information from the index user entered - checking the parameter that the user wants to edit - sending the vector index information, to edit a specific parameter to the **Date** class
showEvents(CommandParser)	Checks what the user wants to see and modifies displayList accordingly: - Searches for name/location in all events - Shows a particular date if user enters a date - Checks for all events with deadline

undoCommand()	Checks the stack for the last command and event object pushed. Based on the last command, processes the opposite of the command (if an event was added, undo will delete the last event added) and pops the respective elements from the stack
showFreeTime(CommandParser)	Shows free time by: - calculating the index (difference of the date entered from the current system date) - Going through all the events/deadlines present for the day and checks for the one hour slots that are empty - Displays the freetime in a range format (from 7am to 12am frame)

Class HistoryHandler

All attributes are inherited from the parent class (ListHandler)

Important Methods	
addEvent(CommandParser)	Adds an event in the masterList by directly calling the add method of **Date** and passing in the individual details (as the details have already been set in each object while reading from the file)
deleteEvent(CommandParser)	This method is used by editEvent() when the event is rescheduled. It deletes an event by checking the unique EventId, finding it in the vector, and sending the info to **Date** to delete it from the vector
editEvent(CommandParser)	An event from the Archive can only be edited if the user reschedules it (ie edits the date and/or time). It edits an event in the masterList by: - obtaining the event information from the index user entered - deleting the event from that date's event vector and adding it into another date's event vector - checking if the user wants to edit the time as well - sending the vector index information to the **Date** class, to edit the time
showEvents(CommandParser)	Checks what the user wants to see and modifies displayList accordingly: - Searches for name/location in all events if user enters - Shows a particular date if user enters a date
undoCommand()	Checks the stack for the last command and event object pushed. Based on the last command, processes the opposite of the command (if an event was added, undo will delete the last event added) and pops the respective elements from the stack
showPending()	Displays all the events that have not been marked as done but are in history (have been set to display events from the past 3 days, as the user might not want to see the whole history)

Class Specifications- Storage

Class Date

Important Attributes	
Int day, month, year	Stores the information about each date, that can be set explicitly only by **ListHandler**
vector<Event>eventList	Stores all the events for that day in a vector form chronologically based on the time
Important Methods	
addEvent (string, string…,..)	Adds an **Event** object in the vector of **Event** objects
deleteEvent(int)	Deletes an **Event** object from the vector at the specified index
editEventName(int,string) editEventLocation(int,string) editEventTime(int, Time,Time)	Edits the name/location/time of the **Event** object in the vector at the specified index
markEventAsDone(int)	Changes the status of the event at the specified index to done (“d”)
getLargestId()	Returns the largest Id in the **Event** vector
dateStringToObject(string)	Converts a string into its attributes by settings the values of day, month, year (set as 0,0,0 if invalid)
compareDays(Date) dateDifference(Date)	Takes in another **Date** object as a parameter and performs arithmetic operations on it (compare and difference)

Class Event

Important Attributes	
string Id	Stores the unique Id of an event
string name	Stores the name of the event
string location	Stores the location of the event (optional)
string eventType	Stores whether the event is a deadline(“d”) or a normal event (“se”)
string eventStatus	Stores whether the event has been marked as done or not. “d” for done, and “nd” for not done.
Time startTime	Stores the start time of the event as a **Time** object
Time endTime	Stores the end time of the event as a **Time** object
Important Methods	
getEventDetails ()	Returns a string that contains the necessary information about the Event in a formatted manner

Class Time

Important Attributes	
int hour, min	Stores the hour and minutes
string am_pm	Stores whether the time is am/pm as the user cannot enter time in the military format
Important Methods	
timeStringToObject(string)	Takes in a string, breaks it down, and allocates its attributes

	(hours and minutes). 0,0 is allocated if the input is invalid
isGreaterThan(Time), isTimeEqual(Time)	Takes in another **Time** object, and calculates whether they are equal or if one is greater than the other
formatCompleteTime(Time)	Takes in the end **Time** object as a parameter, and is called by the start **Time** object, where this method returns a formatted string of the complete time (7pm or 7.30pm to 8pm)

Class Specifications- FileHandler

Class FileStorage

Important Methods	
readFromFile(ListHandler*, ListHandler*)	Takes in two ListHandler object pointers (for the two lists: historylist and upcoming list) as parameters and uses the ListHandler's methods to add in the read objects to the list
writeToFile(ListHandler*, ListHandler*)	Takes in two ListHandler object pointers (for the two lists: historylist and upcoming list) and uses the ListHandler's methods to retrieve information about each event to be written