iDO

				— 🔲 X

		iDO		

10th September,2012 (Monday) Upcoming Tasks:

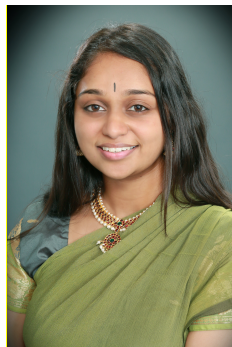| DESCRIPTION	VENUE	FROM/AT	END TIME	PRIORITY
Meeting with boss	Starbucks coffee	Monday 7pm	Monday 8pm	High
Submit CS2103 Project		Tuesday 6pm		High
Dinner with mom	Swensons	Tuesday 8pm	Tuesday 9pm	Medium

COMMANDS: add, delete, edit, undo, redo, restore, archive, done,

Enter command:

Aravindhan Duraisamy

Team Leader, Deadline watcher
Lead Tester

Spatika Narayanan

Documentation, Integration
Lead Architect

Supraja Bhavani Sekhar

Debugging, Research

Lead Developer

Contents

Quick Start User Guide

Product Overview

So many things to do, so little time, and so hard to keep track of them all! Looking for a desktop scheduler? Old school and prefer typing everything? Then iDo is the organizer for you – it vows to be your secretary, calendar, to-do list all rolled into one. iDo's can-do attitude can get your hectic work and home life sorted for you. All you need to do is just type in your tasks through simple commands and iDo will show your schedule for the day, keep track of deadlines, and even maintain your bucket list!

There are 3 kinds of tasks: those which have a fixed start and end time – **timed tasks**, those which only have deadlines – **deadline tasks**, and things that go which you want to do, but at no particular day or time – **floating tasks**. iDo command keywords are NOT case sensitive.

The key features of iDo

Major Features – *What can I do with a task?*

1. Add a task, keywords: add, priority

To create timed tasks,

add description,venue,start date,start time,end date,end time,**priority L/M/H**

Example for adding a timed task:

Add meeting with Boss,Starbucks,7 pm,8pm,**priority H**

Fields you type are **comma-separated**. The **start time and description are compulsory for timed tasks. Other fields are optional.**

If the dates are not entered then the task is by default scheduled at the earliest possible time on the same day, or for the earliest possible day. This holds for deadline tasks also.

If you don't enter priority, it is set to 'medium' by default for deadlines & timed tasks. Default priority for floating tasks is 'low'. Otherwise, it is set to whatever you enter.

2. Search for one (or many), keyword: search

User can search for tasks to view, by partial/full description and ALL relevant matches will be displayed. You can also search by time (tasks displayed are those which have the matching time in either start OR end time) as well as date.

3. Delete a task, keyword: delete

Enter the keyword **delete** followed by full or partial **description** of the task. You can also delete by time or date similar to the search feature.

Example:
delete description

The task is then moved to iDo's 'Recycle Bin'. The delete action can be undone using **undo**, but Recycle Bin cannot be viewed.

Wherever we say that you can enter the 'partial description', if there are multiple matches to the description you enter, then iDo will display all the matches in a numbered sequence. You can enter the numbers (separated by spaces) of the event you want to either **edit**/**delete**/view in **search** results.

4

For example, if you want to delete anything with 'boss' in the description:

delete boss

<Following event(s) & task(s) are displayed>

1. Meeting with boss at Starbucks, 7 pm- 8 pm
2. Submit presentation for boss, 9 pm

1 2

<both displayed events are deleted>

4. Edit a task, keyword: edit

User enters description of the task to be edited following the keyword **edit.**

```
****iDo****

No.  Task Description                    From/At          To
Valid Commands: add, delete, search, edit & exit.

Enter Command: edit mom
```

All the tasks matching the entered description are displayed. The user then enters the serial numbers of the task/s to be edited followed by the new task details.

```
****iDo****

No.  Task Description                    From/At          To
1       coffee with mom
2       meeting mom
3       mom birthday
Valid Commands: add, delete, search, edit & exit.

Enter Command: 1
Valid Commands: add, delete, search, edit & exit.

Enter Command: tea with mon_
```

When we enter another command 'search mom', the screenshot below shows the results displayed. As you can see, 'coffee with mom' has been modified to 'tea with mom'.

```
****iDo****

No.  Task Description                    From/At          To
1       tea with mom
2       meeting mom
3       mom birthday

The following tasks match the entered description:
Valid Commands: add, delete, search, edit & exit.

Enter Command:
```

Some Other Features – *What else can iDo do for me?*

1. Keyboard shortcut: ctrl + i

This opens up iDo for you, with your schedule for the day (timed tasks as well as deadlines to be met) displayed in chronological order.

2. Undo a command, keyword: undo

This is used to undo the previous commands, if any. If the command was to add an item to the schedule, that item is removed after you type in: **undo**

3. Redo a command, keyword: redo

You can use this to redo whatever you've undone if you've used **undo**. Just type in: **redo**

4. Display all floating tasks, floating

5. Deadline alerts:

When a task deadline is reached, a message is displayed to the user as a reminder and he is given the option to postpone the deadline.

The user is notified about high priority deadlines starting from 15 days before the deadline, as soon as iDo is opened to home screen.

9. Exit iDo, keyword: exit

To exit iDo, simply type in the keyword **exit**.

Developer Guide

Our system is called iDo and is used to keep a schedule of tasks and events or maintain a detailed to-do list for the user. User has a set of command he can use to create, delete, edit and search for tasks. iDo processes these commands and organizes the user's agenda. The product should be tailored especially for people whose preferred mode of input is keyboard.

The design approach that we have used is Top-Down. First we break our system/product down into it's major parts. We find the **dependency** of these parts on each other – this gives us our architecture or the structure of our product at the highest level. The major components of iDo are explained below:

TextUI: Textual user interface

Logic: Logic of the organizer, iDo (CRUD, parsing user input, search, sort, etc.)

Storage: Database management, File I/O (retrieval of tasks, writing created and updated tasks)

The dotted arrows are dependency arrows. For example, Logic and it's component classes NEED to have the TextUI to operate (Logic has objects that represent TextUI). Logic depends on component TextUI. Similarly, since Logic components have objects that represent the Storage component of our system, Logic is **dependent** on Storage.
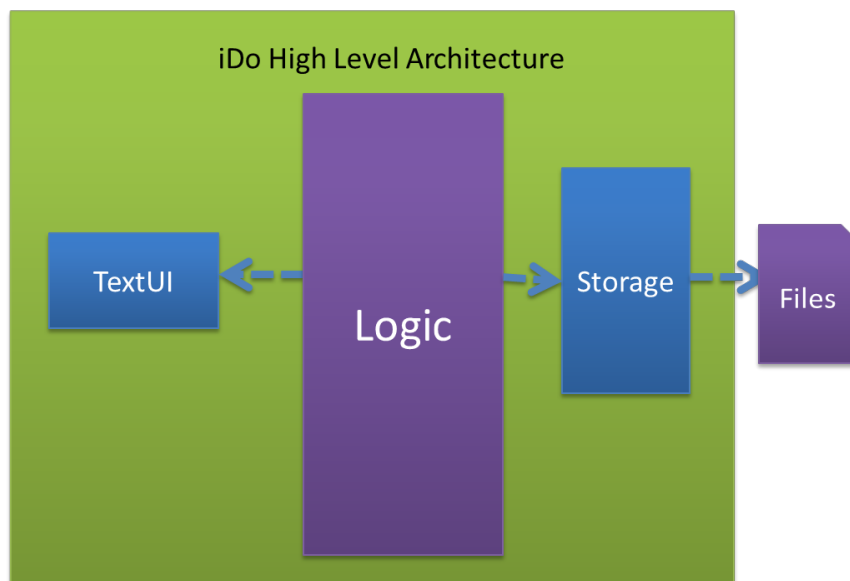
We can think of our system iDo, as 3 offices: front office (view), middle office (controller& back office. UI represents the front office, Logic represents middle and back office is the Storage component.

The **function of UI** or user interface is just to take input from the user (the commands that he/she enters), pass it on to Logic. It is not concerned with what goes on in the other two 'offices', only with the result of those operations. It then gives appropriate feedback to the user, depending on the response it gets from Logic. Of course, UI is also concerned with properly formatting output, making it presentable and user friendly.

The **function of Logic:** We can think of Logic component as the coordinator or **controller** of the system. Most communications within the system need to go through the Logic, to facilitate easy testing, debugging and error handling in later stages of product development. Logic is responsible for getting the user input and then converting it (parsing) into 'Tasks' that can be stored in a vector.

The **function of Storage:** Since we need to maintain a database of the user's schedule, deadlines, etc. we need to have persistent storage in the form of text files. The storage component takes care of writing the appropriate events, tasks, etc. to relevant files, and reading from those files when Logic component requires.

How these components work together: When the user enters a command, the UI simply reads this and passes it on to Logic. Logic processes this user input into parts that can be easily stored (as a Task) and changed. Based on the user input, the Logic has figured out what type of command the user would like to perform. The command is performed and feedback is given to user through UI. This sequence is shown by a **sequence diagram on page 16** (due to space constraints). Dependencies are shown below. As you can see, similar to our architecture diagram from page 1 of the developer's guide, Logic depends on most classes of UI and Storage.
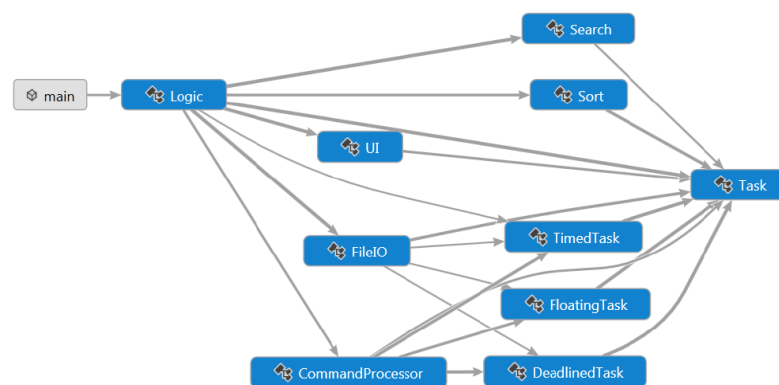
Components of Front Office – UI:

Since our user interface is currently textual, we have one class – UI to represent the 'front office' component. The important sub-components of this are: void displayHomeScreen(vector<Task*>), void feedback(bool,string), string getUserInput(). Whenever displayHomeScreen is invoked (this is done by the Logic), it displays the list of task given to it. The getUserInput component is invoked by the Logic component whenever Logic is ready to read the next user command. The feedback component is to interact with the user after each operation is performed.
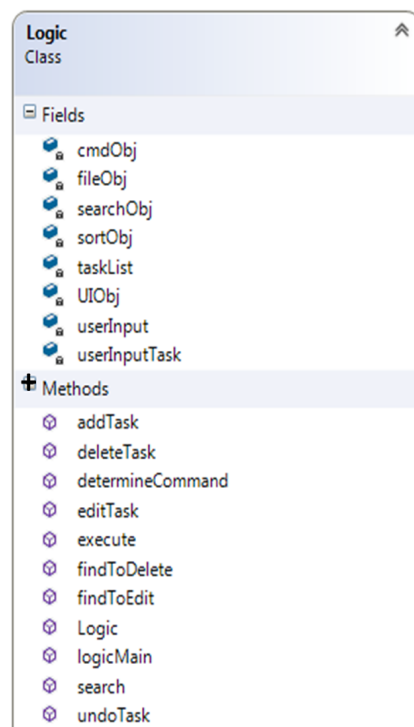
Components of Middle Office – Logic:

The 4 major components in this office are represented by: **Logic class, CommandProcessor class, Search class, Sort class**.

1. Logic class

Below is the class diagram for the class Logic, of iDo, to demonstrate its main components;

+ public fields/methods, - private fields/methods.

logicMain – in **Logic** class controls the flow of control through the logic component of the project. Uses a UI object to read and display data from/to the user. Uses a commandProcessor object to parse the user input to give you the command type (add/delete/edit/search/undo, etc.). Calls the **execute()** function of **Logic** class in order to execute the command. Returns the feedback to the UI.

execute - executes the new task as per the command entered by the user. It determines the command type by calling determineCommandType. Based on the command type, it calls the relevant function within a switch case. Returns feedback to logicMain depending on the feedback from the function it called.

addTask - adds the task entered by the user into the task list (data member of the Logic class). Gets the existing list of tasks by using the file obj. Appends the task to the list. Calls the sort function using the sort object by passing the new list of tasks as argument. Sends the sorted list back to file handling class to be stored in the relevant file. Returns feedback to the execute function.

deleteTask - deletes a task from the task list. Obtains the index of task to be deleted from the function- findToDelete Deletes the task existing in the above obtained index. Removes the task from the vector list of tasks.

```cpp
bool Logic::findToDelete(Task * userInputTask)
{
    int i;
    //based on description, use search obj to retrieve vector list of matching tasks
    vector<int> searchResults;
    //fileObj.setFileName("task.txt");
    //fileObj.readList();
    //Logic::taskList = fileObj.getTaskList();
    searchObj.setInputList(fileObj.getTaskList());
    searchObj.executeSearch(userInputTask->getDesc());
    searchResults=searchObj.getIndices();
    vector<Task*> tempList;
    for(i=0;i<searchResults.size();i++)
    tempList.push_back(taskList[searchResults[i]]);
    UIObj.displayHomeScreen(tempList);
    userInput=UIObj.getUserInput();
    vector<int> userIndex= cmdObj.intProcessor(userInput);

    for(i=0;i<userIndex.size();i++)
    {
        Logic::deleteTask(searchResults[userIndex[i]-1]);
    }
    fileObj.setTaskList(taskList);
    fileObj.writeList();
    searchObj.clearSearchResults();
    return true;
    //getUserInput() and call intProcessor of command processor
```

```
        //now u have int vector use that to find matching task * from retrieved list (from
search)
        //matching task * shld be deleted
}
```

editTask - edits a task in the task list. Obtains the index of task to be edited from the function-findToEdit. Deletes the task existing in the above obtained index. Adds the new task into the relevant index.

search - returns the search results based on the user input. Takes the description entered by the user as argument. Calls the search function of search class using search object. Calls the display in UI using UI object. Returns feedback to execute.

2. CommandProcessor class: used to manipulate the string that the user enters by parsing it into the correct command type and the fields of the task to be created. Has 3 MAJOR sub-components: cmdProcessor, intProcessor & descProcessor.

```
CommandProcessor
Class

⊞ Methods
    ◉  cmdProcessor
    ◉  descProcessor
    ◉  intProcessor
    ◉  stringToTime
```
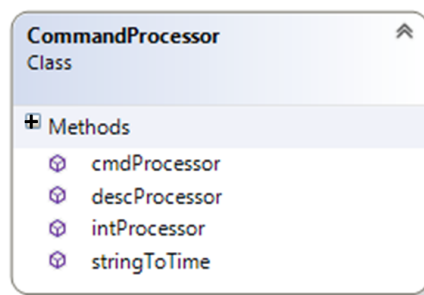
cmdProcessor is invoked when the preliminary user command needs to be processed.

```cpp
string CommandProcessor::cmdProcessor (string userInput, Task*& newTask)
{
        int i, j;
        tm* sTime = NULL;
        tm* eTime = NULL;
        char description[MAX_DESC_SIZE], startTime[MAX_TIME_SIZE], endTime[MAX_TIME_SIZE],
cmd[MAX_COMMAND_SIZE], singleWord[MAX_COMMAND_SIZE];
        bool start = false, end = false;
        int state = 3;
        strcpy(description, "");
        strcpy(startTime, "");
        strcpy(endTime, "");
        strcpy(cmd, "");

        for(i=0; i<userInput.size()&&userInput[i]!=' ';i++)
        {
                cmd[i]=userInput[i];
        }

        cmd[i]='\0';
```

```cpp
bool validCmd = CommandProcessor::actualKeyWord(cmd);
if(validCmd == false){
            strcpy (description ,cmd);
             strcat(description, " ");
            strcpy(cmd, "add");
      }
if(i==userInput.size()){
      Task *ft = new FloatingTask();
            ft->setDesc(description);
            newTask=ft;
}

else{
      i++;
      while(i < userInput.size()){

            for(j = 0; i < userInput.size() && userInput[i] != ' ' ;i++, j++){
                  singleWord[j] = userInput[i];
            }
            singleWord[j] = '\0';
            strcat(singleWord, " ");
            i++;

            if(start == true && state == 1){
                  sTime = CommandProcessor::stringToTime(singleWord);
                  state = 2;
            }
            else if(end == true && state == 1){
                  eTime = CommandProcessor::stringToTime(singleWord);
                  state = 2;
            }
            if(singleWord[0] == '[')
                  CommandProcessor::trim(singleWord);
            start = isStart(singleWord);
            end = isEnd(singleWord);

            if(start != false || end != false){
                  state = 1;
            }

            if(state == 2){
                  state++;
            }
            else if(state == 3){
                  strcat(description, singleWord);
            }

      }
}

if(eTime == NULL && sTime == NULL){
      newTask = new FloatingTask(description);

}
else if(sTime == NULL){
      newTask = new DeadlinedTask(description, eTime);
```

```
        }
        else if(eTime == NULL){
                newTask = new DeadlinedTask(description, sTime);

        }
        else{
                newTask = new TimedTask(description, sTime, eTime);
        }
        return cmd;
}
```

intProcessor is invoked when subsequent user commands needs to be processed. That is, if the user wants to delete tasks 5, 6, 7 of the displayed matches, the string "5 6 7" needs to be converted to an int vector of the positions of those tasks corresponding to tasks 5, 6 and 7 of the display, in the original taskList.

descProcessor is invoked during the follow up user commands for 'edit' command.

3. Search class: contains the list to be searched on, the search results, and the indices of the search result in the main vector. There are two search functions (or rather one overloaded function) to enable description based search and task based search.

```
        bool Search::executeSearch(Task* query){

                for(int i =  0; i < Search::inputList.size(); i++){
                        if((Search::inputList[i]) == query){
                                Search::searchResults.push_back(Search::inputList[i]);
                                Search::resultIndices.push_back(i);
                        }
        }
                return true;
        }
```
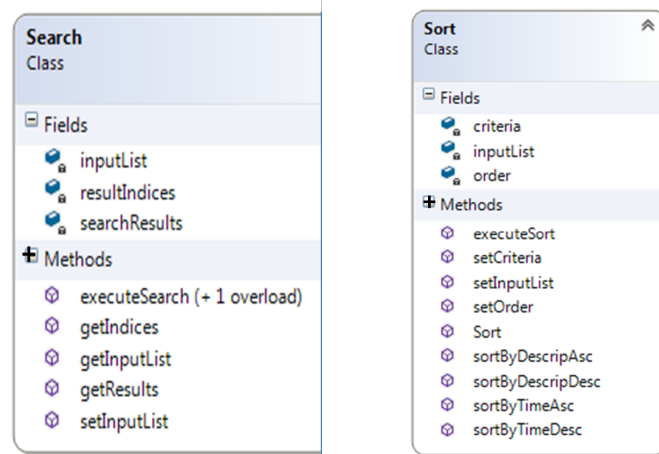
4. Sort class: This class sorts the main list based on time or descriptions in ascending or descending order.

```
    void Sort::sortByTimeDesc(){
        for(int i = 0; i < Sort::inputList.size() - 1; i++){
                for(int j = 0; j < Sort::inputList.size() - 1 - i; j++){
                        if(Sort::inputList[j]->getEnd() < Sort::inputList[j+1]->getEnd()) {
                                Task *temp = Sort::inputList[j];
                                Sort::inputList[j] = Sort::inputList[j + 1];
                                Sort::inputList[j + 1] = temp;
                        }
                }
        }
    }
```

The last page of this guide contains a sequence diagram explaining the sequence of events & interaction between the components for 'add' command. This involves front, back and middle office working together.

Components of Back Office – Storage:

The major components of Storage are the classes: Task (an abstract classes which is concretized through 3 classes – DeadlinedTask, TimedTask, FloatingTask) and FileIO.

These task classes maintain the various fields like task description, start time/date, end time/date. We have 3 concrete classes for each type of task to take care of differences that might arise during later implementations.
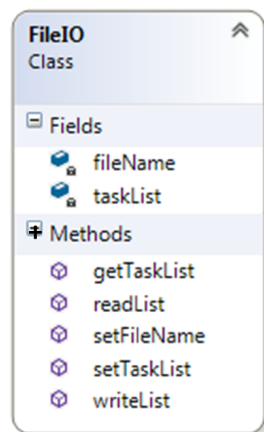
Below is a class diagram to explain this inheritance. 'Task' is in italics to show that it is an abstract class (UML notation). # - private fields/methods; + - public fields
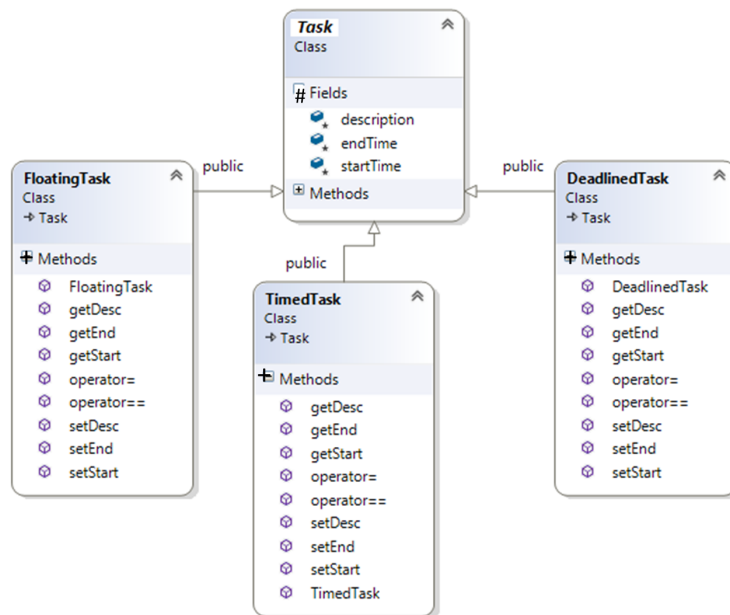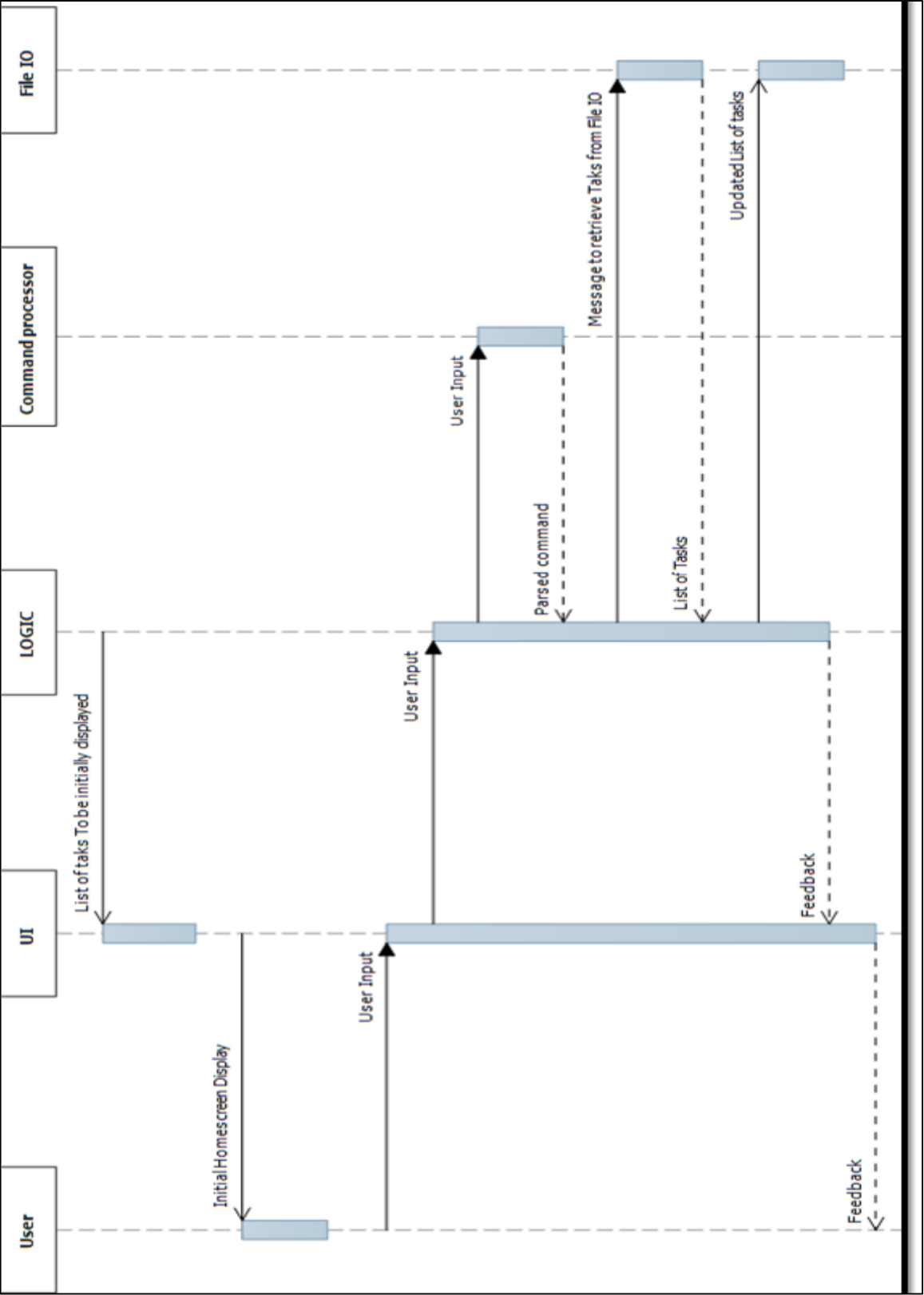
FileIO

It has a task list which stores the tasks to be read from files and the tasks to be written on to files and getters and setters for the same.

[T10-2C][V0.2]