CS2103 PROJECT MANUAL

Command line task manager

He Haocong

U099121H

Liu Jialong

U099122U

Wang Xiangyu

U099120W
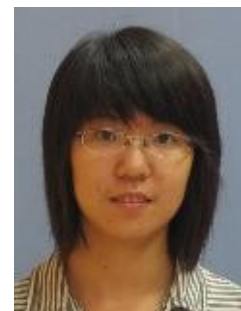
Zhou Biyan

U094837M

Group L11

November 15, 2010

Contents

1   Introduction

TaskManager (referred to as taskManager later in this document) is a multi-platform command-line application that manages one's to-do's.

It maintains a list of tasks and provides a rich collection of commands to manipulate them. It comes with a TaskManager shell (which features a command-line interface) and an interactive text-based user interface (TUI). Besides basic task manager functionalities, it supports command aliasing, piping and start up script. A decent set of command aliases are defined in the .tmrc file to enable intuitive command line task management. Finally, the auto-completion module adds on to the usability and `import`/`export` commands make sharing and visualizing tasks easy.

2 User Guide

2.1 Quick Start

This section introduces you the minimum amount of commands to get started.

1. Start taskManager shell:

   On Mac OS and GNU Linux (referred to as "*nix" later), taskManager can be started from shell by commands:

   ```
   $ cd the/folder/containing/taskManager
   $ ./taskManager¹
   ```

   On Microsoft Windows, taskManager can be launched in command prompt as well, or simply by double clicking "taskManager.exe".

   Once taskManager is launched, you will see a prompt like `> _`, and you will start typing commands!

2. Add some tasks:

   ```
   $ ./taskManager
   >  add "Sample task 1"
   >  add "Sample task 2"
   TaskManager:  This task is highly similiar to some existing task:
       1     Sample task 1
   Do you really want to add it?  y
   ```

   To add a task, simply use `add` command followed by the description of the task in a pair of quotation marks.

   If no error messages are shown, the task is successfully added. TaskManager may prompt for confirmation if the task to be added is highly similar to some existing task(s) to help prevent people forget adding tasks, which is the case in the example above.

3. List the existing tasks:

   ```
   >  ls
   1      Sample task 1
   2      Sample task 2
   ```

   To see the existing tasks, use `ls` command. By default, the taskManager shows the serial numbers and the descriptions of the tasks.

4. Mark a task as finished:

---

¹ *nix version can be run at any directory after installing taskManager – `"make install"`. See details in section `"Compilation and Installation"`.

```
> finish 2
> ls
1      Sample task 1
2 f    Sample task 2
```

To finish an existing task, use `finish` command followed by the serial number of the task to finish. Notice that for finished task, an 'f' is shown between serial number and task description.

5. Remove task(s):

```
> ls
1      Sample task 1
2 f    Sample task 2
> rm 1
TaskManager:  Do you really want to remove this task permanently?  y
> ls
2 f    Sample task 2
```

To remove an existing task, use `rm` command followed by the serial number of the task to remove. TaskManager will prompt for confirmation when removing tasks.

6. Exit from taskManager:

```
> exit
```

To quit from taskManager, use `exit` command. All changes to the existing tasks will be automatically saved.

2.2   More commands

2.2.1   read, import and export

TaskManager stores the tasks in an XML file which is by default ∼/`record.xml` on *nix, and `%USERPROFILE%\record.xml` on Windows.

TaskManager also supports importing/exporting the existing tasks from/to XML and HTML files. This is done by `read`, `import` and `export` commands.

read   reads an XML file, list all the tasks it contains without affecting the current task list.

This is helpful when you only want to peek the content of an xml file without really importing it.

```
> ls
1      Sample task 2
> read midterms.xml[2]
1 f   CS2103 midTerm Sep 29 06:30 - 07:30 pm MPSH 1B
2 f   CS3230 midTerm Oct 15 06:00 pm
```

```
                3 f   CS3241 midTerm Oct 07 lecture
                4 f   CS3244 midTerm Oct 04 lecture
                5 f   ST2132 midTerm Oct 08 LT33 12:15 - 1:30 pm
                >   ls
                1     Sample task 2
```

import   is similar to `read` command. It reads the content of the XML file and appends all the
         tasks in it to current task list.

```
                >   ls
                1     Sample task 2
                >   import mytasks.xml
                1     Sample task 2 ³
                2 f   CS2103 midTerm Sep 29 06:30 - 07:30 pm MPSH 1B
                3 f   CS3230 midTerm Oct 15 06:00 pm
                4 f   CS3241 midTerm Oct 07 lecture
                5 f   CS3244 midTerm Oct 04 lecture
                6 f   ST2132 midTerm Oct 08 LT33 12:15 - 1:30 pm
```

export   exports the current task list to an XML or HTML file.

```
                >   export sampletasks.xml⁴
                >   exit
                $   cat sampletasks.xml
                    <taskList>
                    <task>
                    <serialNumber> 1 </serialNumber>
                    <deadline> 1288473083 </deadline>
                    <priority> 0 </priority>
                    <description> Sample task 2 </description>
                    <group> default </group>
                    <isFinished> 0 </isFinished>
                    </task>
                    </taskList>
```

         `export` can also be used to generate an HTML file which is more visually pleasant in
         your favourate browser.

```
                >   export -html sampletasks.html⁵
                >   exit
```

---
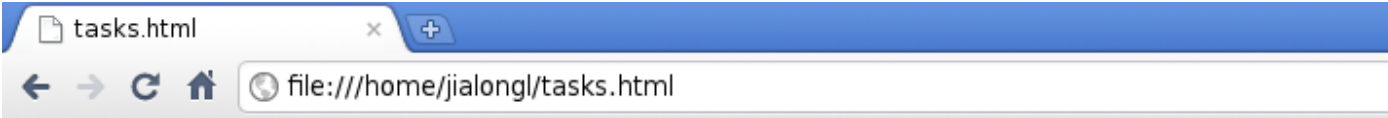
² The file name is not quoted. If the file name contains space, please quote it with a pair of quotation marks.

³ Task 1 is still in task list. Importing tasks will not erase existing tasks.

⁴ Currently `export` does not support environmental variables in path. i.e. `export ∼/abc.xml` will not export the file to user's home directory `/home/username/`.

⁵ If the name of the HTML file to export is not specified, the default file name `"record.html"` is used.

Figure 1: tasks exported as webpage [6]

2.2.2 task

```
> task 1
  Number:   1   Deadline:   Sun Oct 31 05:11:23 2010
  Priority: 0   Status:     Doing
  Group:    default
  Details:
    Sample task 2 7
```

To show detail information of a task, use `task` command followed by serial number of the task.

2.2.3 pri

```
>  pri 1 10
>  task 1
  Number:   1   Deadline:   Sun Oct 31 05:11:23 2010
  Priority: 10  Status:     Doing
  Group:    default
```

_____

[6] Page may not render correctly in IE 6 or its earlier versions.

[7] Adding tasks with detailed information is covered in section 2.3.1. In this example, default values are shown.

```
Details:
  Sample task 2
```

To change the priority of a task, use `pri` command followed by the serial number of a task and its new priority. Priority is typically a number between -20 and 20. By default, the priority of a newly added task is 0.

2.2.4 edit

```
>  edit 1 -d "Sample task 3" -p 12 -t 1d⁸ -g SampleGroup -f yes
>  task 1
   Number:   1   Deadline:   Mon Nov 1 05:47:59 2010
   Priority: 12  Status:     Finished
   Group:    SampleGroup
   Details:
     Sample task 3
```

To edit a task, `edit` command which has the form: `edit taskSerialNumber -d newDescription -p newPriority -t newDeadline -g newGroup -f finished_or_not`. Only `taskSerialNumber` is compulsory. Besides, to finish a task, `finish 1` is equivalent to `edit 1 -f yes`.

2.2.5 undo and redo

```
>  undo
```

would undo the last command. Note that it has no effect on commands like `ls`, `export`, `tui` and `undo`.

```
>  redo
```

re-does the last undo. It can be executed util all the undo's are re-done.

2.3 Using Options

Like `edit`, some of the commands come with options to support more functionality. In this section they are introduced in great detail.

2.3.1 add

`add` has three flags: `-t` to specify deadline, `-p` to set priority and `-g` to put the group name for a task.

-t adds a task with a deadline:

---

⁸ -t 1d means setting the deadline to be 1 day later. Time formats that taskManager accept are discussed in section 2.3.1.

```
>  add "74 hours from now" -t 3d2h
>  add "Tomorrow 2359" -t b2d
>  add "due at Jan 2, 1970 00:00 UTC" -t 86400
```

TaskManager support 3 types of time format:

"plus" format
: "Plus" format specifies the how much time left for the task, and it has form `?w?d?h?m`[9], where each question mark stands for a number (not a digit).
For example, 3d2h means the task will due after 3 days 2 hours the moment the command is executed – you have 3 days and 2 hours to finish it.

"by" format
: "By" format has a similar form of "plus" format. It is in the form of `b?w?d?h?m`[10], where each question mark stands for a number (not a digit). For example:

b0d22h	by 10:00pm today.
b2d	by 23:59 tomorrow.
b1w	by the end of this week. i.e. 23:59 of the Saturday[11]of this week.
b0w5d	by 23:59 on Friday of this week.
b2w3d8h	by 8:00am on the Wednesday of the next next week.

Unix timestamp
: "Unix timestamp" means the number of seconds elapsed since Jan 1, 1970 00:00:00. It is not recommended for users, but rather used as a lower-order procedure for developers.

-p adds a task with a priority:

```
>  add "some important task" -p 20
```

-g specifies a group for a task:

```
>  add "the task with group"[12] -g SampleGroup
```

Options are not compulsory. Different options can be used together. For example:

```
>  add "CS2103 final exam" -p 10 -g "finals" -t b3w2d
```

would add a task "CS2103 final exam" which has a priority of 10, belongs to group "finals" and is held on the Tuesday 3 weeks from now.

2.3.2 ls

`ls` has four flags: `-s` to sort tasks, `-k` to search tasks, `-f` to view only (un)finished tasks and `-g` to filter tasks by group name.

-s sorts the existing tasks:

---

[9] At least one of letters w/d/h/m should be specified.
[10] At least one of letters w/d/h/m should be specified.
[11] Sunday is the first day of a week.
[12] If group name contains spaces, use a pair of quotation marks to quote it.

```
>  ls -s "deadline priority"
```

A more general format is: `ls -s "keyword1 keyword2"` ...

The listed tasks will be sorted by keyword1 and then keyword2 ...

Available columns are: deadline, priority and serialnumber. Prefix of a keyword is also acceptable. e.g. `ls -s "p"` will sort the tasks by their priorities.

Examples:

```
>  ls -s "p d"
1    task 1 highest priorty.   3 Sun Oct 31 06:49:09 2010
2    task 2 high priority.     2 Mon Nov 1 06:54:42 2010
3    task 3 default priority.  0 Tue Nov 2 06:54:37 2010
```

-k filters tasks with a keyword[13] where `?` means any single character, `*` means a string of any length (including 0 length).

Take command `ls -k *Sam?le*task` as an example, `"This is a sample with an important task"` will match `*Sam?le*task` as the first `*` matches `"This is a "`, `?` matches 'p' and the second `*` matches `" with an important "`.

`"samqleTask"` will also match `*sam?le*task` as both `*` maps empty string and `?` to be 'q'.

-f shows finished/unfinished tasks:

`ls -f yes` shows only finished tasks.
`ls -f no` shows only unfinished tasks.

-g shows tasks of a specific group: `ls -g SampleTask` makes tasks only from SampleTask group shown.

Important

As said, different options can be used together when issuing commands. When more then one restrictive options are present, *conjuction* of these restrictions are used. e.g.

```
ls -g SampleTask -f y
```

will show tasks that are finished AND from "SampleTask" group.

2.3.3 rm

`rm` has basically two usages: remove tasks by a group name or serial number.

Use `-g` option to remove a group of tasks: `rm -g SampleTask` removes the entire SampleTask group.

`rm` can be used remove several tasks once as well. e.g. `rm 1 2 3` removes tasks 1, 2 and 3.

---

[13] keyword is case insensitive.

2.4 Getting inTUItive – The Text User Interface

The text-based user interface (referred to as TUI later) makes interaction more intuitive by visualizing the deadlines, groups and simplifying manipulation from issuing commands to moving cursors and pressing keys.

TUI can be launched using command `tui` in command-line mode:
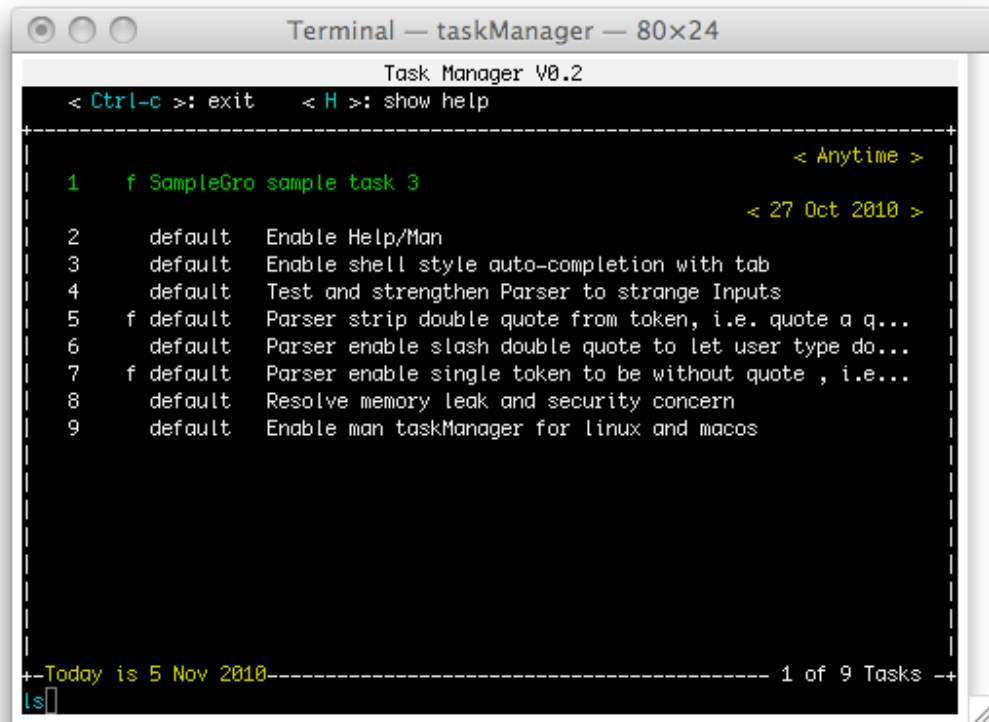
Figure 2: The first screen of TUI

As seen, the tasks are grouped according to deadlines. If a task's deadline is "Anytime", it means the task has no specific deadline – you can finish it at ease.

Besides, TUI provides some features that are not available in the command line mode:

1. auto-completion

   TUI helps you complete words (including built-in commands and words you have entered) using `TAB` key. For example, after you typed `"un"`, the word `"do"` will appear right after `"un"` in cyan, and now you can complete the word `"undo"` if you press `TAB`.

2. summary of tasks in a calendar

   Press `c` in TUI, a small calendar would appear at the bottom right corner. The dates that has tasks due have a highlighted background. You can navigate dates using arrow keys to see what exactly are the tasks due on that date.

3. incremental search of tasks by keyword.

Press `s` in TUI to start incremental search – tasks become fewer and fewer as you type. Press `ENTER` to end searching and do whatever thing you like on the tasks!

This is a summary of the key bindings:

`SPACE` or `ENTER`	show details of the selected task
↓ or `j`	select the task below
↑ or `k`	select the task above
`g`	select the first task
`G`	select the last task
`number g`	select the `number`th task displayed in the list
`number G`	select the task whose serial number is `number`
`number selectionKey`[14]	select `number` tasks above/below
`p`	previous page/page up
`n`	next page/page down
`u`	undo
`R`	redo
`c`	show the small calendar
`C`	issue a command without leaving TUI
`a`	add a task
`e`	edit a task
`d`	remove a task
`f`	finish a task
`s`	search tasks by keywords

You can always press `H` to read the key bindings.

2.5   Advanced Usages

2.5.1   command piping

```
>  ls | rm
```

TaskManager supports command piping for most commands though it is a bit different from traditional Unix pipe. Piping means if one command selects some tasks, then the selected tasks will be passed to the next command as input. The tasks after the last command will be shown as output. Piping in taskManager is done with symbol |. When a pipe signs appear in a command, the broken-up commands (separated by pipes) are executed one by one from left to right. For example:

1. finish all tasks:

```
>  ls | finish
```

2. remove all finished tasks:

---

[14] It means pressing a number key and then press ↑/↓/`j`/`k` key.

```
>  ls -f yes | rm
```

3. import from a file and replace current task list:

```
>  ls | rm | import newTasks.xml
```

4. import all CS2103 group tasks from a file:

```
>  read newTasks.xml | ls -g CS2103 | add
```

5. export all CS2103 related tasks to a html file:

```
>  ls -k *CS2103* | export -html cs2103tasks.html
```

6. show details of CS2103 tasks, sort by priority:

```
>  ls -g CS2103 | sort "pri" | task
```

2.5.2 command mapping

```
>  map "ls" "ls -f no"
```

TaskManager supports custom command mapping/aliasing so you can alias the taskManager commands to the format you are more comfortable with.

General format of `map` is: map "new command" "original command"

A simple mapping is like the previous example, which maps `ls` to `ls -f no`, which effectively hides finished tasks when doing `ls`. To retain the original `ls` command, use . The command that `ls` maps to will not be executed then.

When several `map` commands are executed in sequence :

```
>  map "lsa" "ls"
>  map "ls" "ls -f no"
```

Then `ls` will show the unfinished tasks only, while `lsa` lists all the tasks.

The order of mapping matters as commands are executed one by one. Thus reversing the order of `map` commands MIGHT NOT work.

More complex mapping makes use of symbol `$`. There are two kinds of `$` symbols:

$0 matches all characters from the current position.
$1, $2, $3 ...matches one token (i.e. a word or several words .

Examples:

1.

```
>  map "tomorrow $1" "add $1 -t 1d"
>  tomorrow "Finish user guide"
```

The latter command will be parsed as `add "Finish user guide" -t 1d`, and a new task "Finish user guide" will be added with the deadline of 24 hours.

2.

```
>  map "do $1 at $2" "add $1 -t $2"
>  do "Laundry" at 4h
```

The latter command will be parsed as `add "Laundry" -t 4h`, and a new task called "Laundry" will be added with the deadline of 4hours.

3.

```
>  map "ls $0" "ls -f no $0"
>  ls
>  ls -g cs2103
```

The second command will be parsed as `ls -f no`, and will list out all unfinished tasks. The third command will be parsed as `ls -f no -g cs2103`, and will list out all unfinished cs2103 tasks.

Important

TUI uses `ls` to retrieve tasks. Mapping `ls` to something else will affect behaviour of TUI, which may or may not be the case you want.

2.5.3   taskManager script

Task manager commands can be saved in a text file and be executed using run command.

```
$ cat tmscript
ls
map "ls" "ls -f no"
ls
$ ./taskManager
Task Manager V 0.2
type " exit" to quit, "help" for more instructions
================================================
>  run tmscript
1 f   Sample task 1.  This also has high priorty
2     Sample task 2.  This has high priority
3     Sample task 3.  This is the latest

2     Sample task 2.  This has high priority
3     Sample task 3.  This is the latest
```

The first 3 tasks are the result of the first `ls` in the script. The last 2 tasks are the result of the second `ls` in the script. Because `ls` is mapped to `ls -f no`, finished tasks are not shown by the second `ls`.

2.5.4 startup script

By default, taskManager executes a special script everytime when it is started. This script is ~/.tmrc on *nix and `%USERPROFILE%\tmrc.txt` on Windows. This file can be editted to include customized settings.

Examples:

1. To switch to the interactive user interface by default, add this line into tmrc:

   ```
   tui
   ```

2. To save a backup file when taskManager is started:

   ```
   export /tmp/backupTasklist.xml
   ```

3. To show tasks when taskManager is started:

   ```
   ls
   ```

4. To remove finished tasks when taskManager is started:

   ```
   ls -f yes | rm
   ```

5. To run a script with all self-defined mappings when taskManager is started:

   ```
   run /home/myusername/mymappings
   ```

2.5.5 talk to taskManager

For all inputs that cannot be recognized by taskManager as a command, it will be treated as natural language sentence. TaskManager will try its best to recognize it and give correct response. For example:

```
> what do I do today?
```

will list all the tasks due today.

2.6   Compilation and Installation

This section discusses compilation and installation of taskManager. One thing to note is that the TUI is build by default, which requires PDcurses library. It is free and can be downloaded here: http://sourceforge.net/projects/pdcurses/files/.

Unix-like Operating Systems are likely to have curses (its cross-platform version is PD-curses) library installed already. For Windows' users convenience, the necessary library files for compilation are included in the zip.

2.6.1   Microsoft Windows

On Windows, taskManager can be built with Visual Studio 2008® [15] in the following steps:

1. Start Visual Studio with "C++ Development Settings".

2. Create an *empty win32 console project* called "taskManager".

3. Drag all the .h and .cpp files (including those in the subdirectories) into the solution folder. Files should be automatically categorized into header files and source files.

4. Press `Alt + F7` to edit the project properties. Under `"Configuration Properties"` => `"General"`, set `"Character Set"` to be `"Use Multi-Byte Character Set"`; Under `"Linker"` => `"Input"`, add pdcurses.lib to `"Additional Dependencies"` and add the folder containing `pdcurses.lib` to `"Additional Library Directories"`.

5. Copy `pdcurses.dll` to `%WINDIR%\system32\` or the directory your executable will be generated (which would probably be "Debug" folder).

6. Build the solution.

7. Copy `tmrc` to your home directory (e.g. `C:\Documents and Settings\John Doe\` on Windows XP) and rename it to `tmrc.txt` in order to use the commands we pre-customized. This step is optional.

2.6.2   Unix-like Operating Systems

On Unix-like operating systems such as GNU Linux and Mac OS X: start a shell; unzip the zip archive; change directory properly and type:

```
$ make
$ sudo make install
```

---

[15] Menu names mentioned below may vary among different versions of Visual Studio.

"make install" is optional. It just makes taskManager available system wide by copying the executable and man page to cooresponding directories.

Another option is to copy the `tmrc` to your home directory and rename it to `.tmrc` in order to use the commands we pre-customized.

The curses library should ship with most Linux distributions and Mac OS. If not, it can be installed with the package manager (`apt-get`, `yum`, `pacman` on various Linux distributions and `port` on Mac).

3 Developer Guide

This part of the manual is mainly meant for developers. Users who are interested in finding out the actual structure behind are also welcome to read this part.

3.1 Development Process

3.1.1 Overview

We strictly followed a top-down approach designing the software. As for the software development life cycle (SDLC), our team is following the iterative approach so that every version of our software is a complete and working product that satisfies our target users. In each iteration, we are following the waterfall approach to save time and energy.

3.1.2 Domain Analysis

1. Requirement Study

   This project aims to build up a task manager program that helps user to manager his/her to-do's. This program should provide the user with a CLI (command line interface), which means a user is able to type `add "write developer's guide" -t 1d` at the prompt to add a new task "write developer's guide" which has a deadline of 1 day.

   We did some surveys to find out what are the common features that existing products have – adding/displaying/editting/removing tasks, setting priorities, search tasks by keywords... We then brainstormed together to come up with other features that could possibly fill in the blanks of existing products and provide users with software of better quality and user experience, for example System alert/notification, integration with system calendar (though they are not implemented), natural commands and auto-completion of commands...

   As the product should cater Windows users, we have also decided to make a TUI (text user interface) in order to create a more user friendly environment after conducting interviews with typical Windows users. (user guide for TUI is given in section 2.4)

2. OO Domain Model

   A condensed object-oriented domain model is given in Figure 3. This model is an overview of the problem domain and ignores most of the details. Basically, the problem is modeled as a user interacting with a task manager that manages his/her tasks. More detailed descriptions about sub-problems like how commands are parsed, how tasks are modified and stored, how messages are shown to the screen, as discussed in the following.

3.1.3 Design

We chose a multi-tier architecture, in fact the most widely used three-tier architecture to implement the program. Figure 4 shows the diagram of the architecture. The user can

Figure 3: A condensed OO domain model

interact with the software through either CLI or TUI. The user interfaces will pass whatever command it receives from the user to the Logic. The Logic of the program will then perform its tasks accordingly and manipulate the storage files which store all the user data.
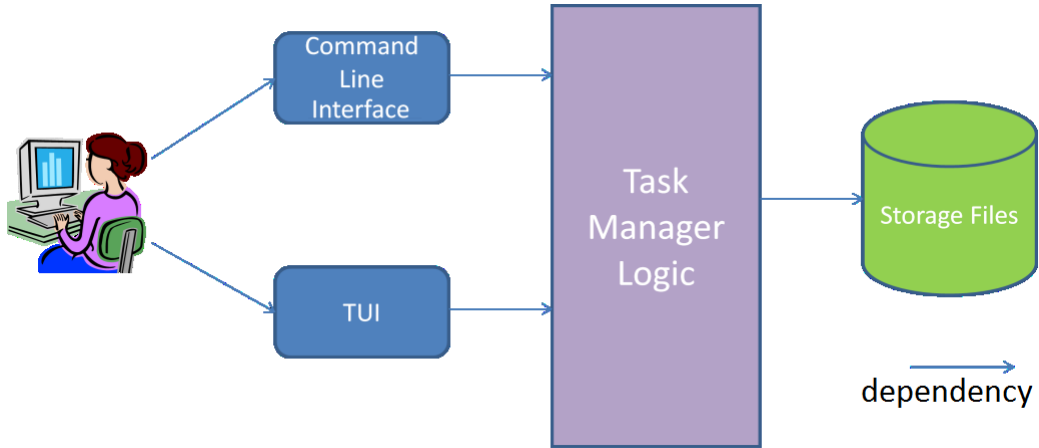

Figure 4: Three-tier Architecture

3.1.4 Implementation

1. Overview of the taskManager workflow:

   (a) The program starts with a `shell`.

   (b) `Shell` asks `IO Module` to get user input.

   (c) `IO Module` requests the `Parser` to parse the text input to a `CommandList` object for it. Then `IO Module` returns the `CommandList` to `Shell`.

   (d) `Shell` calls `MainCommandExecutor` which will execute the command accordingly.

   (e) The `MainCommandExecutor` returns a `Result` object to the `Shell`.

   (f) `Shell` requests `IO Module` again to handle screen output using the `Result` object.

   (g) `IO Module` uses `Parser` to parse the `Result` to a string and print that string.

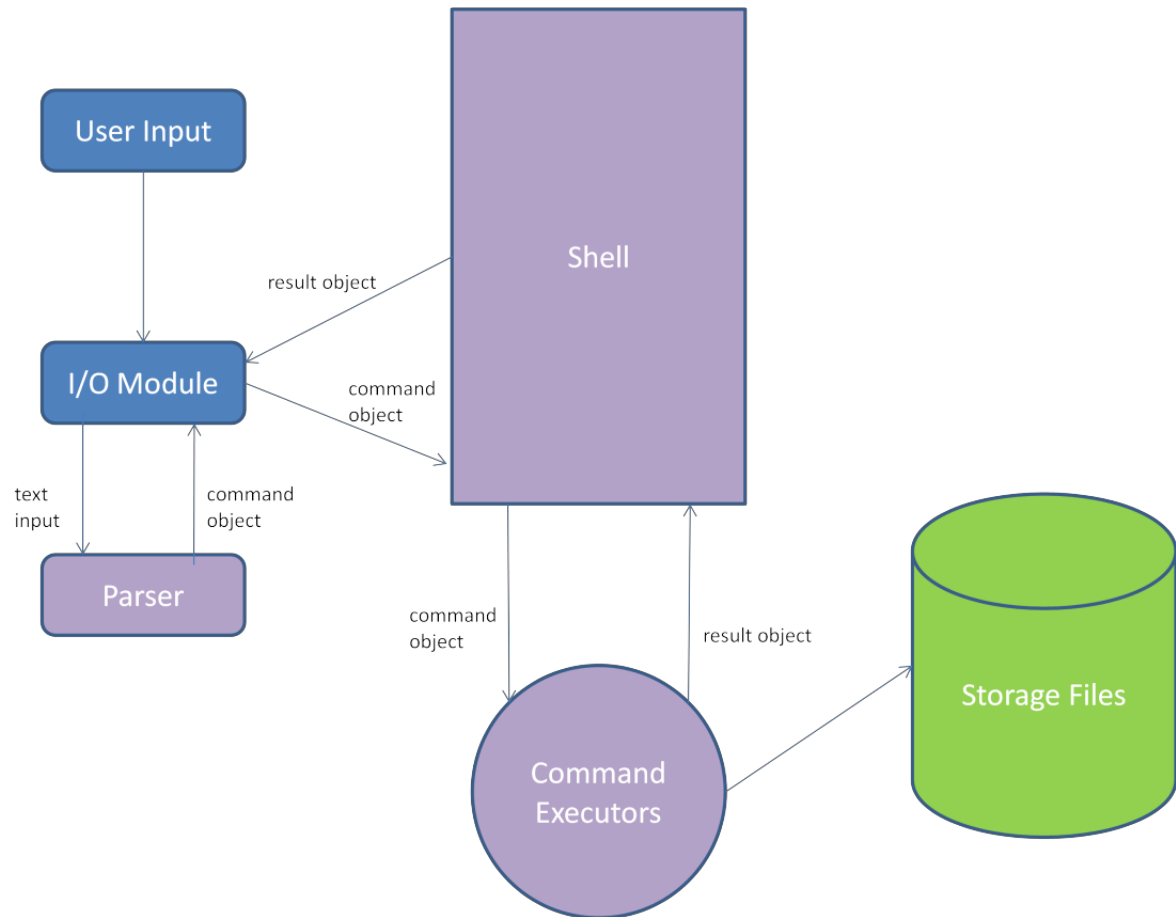   The flow chart of the above process is given as:

Figure 5: Workflow of taskManager

2. UI Implementation

We have implemented two user interfaces – CLI and TUI. As user interfaces, they basically reads text input (be it a string or a keystroke) from the user and passes it to the `Logic` and receive `Result` objects and print to screen.

Thus in our implementation, they are both `IO Modules` – The CLI is implemented as `Keyboard IO Module` and the TUI is implemented as `PdcIO`. The advantages of this polymorphism are:

(a) the `Logic` does not need to know what is exactly (`Keyboard IO Module` or `PdcIO`) it is talking to it. For example, `Logic` can just call `getCommand()` method and the `Command` will be generated.

(b) it will be easy when you want to implement, say a GUI, for taskManager in the future.

(c) decreases the level of coupling in the software – the `Logic` does not need to be aware of each `IO Module`.

19

```
    virtual void showWelcomMessage() {};
    virtual CommandList getCommand() {return vector<Command *>();};
    virtual void showOutput (Result* result) {};
    virtual void handleException(exception_e except) {};
    virtual void confirm(string prompt) {return true;};
    virtual void echo (string s) {};
```

Listing 1: abstract methods in `TM_IO_Module.h` to apply polymorphism

3. Logic Implementation

   The Logic part of taskManager is in charge of getting Command

   (a) Shell

   The `shell` is the controller of the whole Logic. Shell is able to start a loop that in each iteration:

      i. asks `IO Module` for a `CommandList` object. Why a `CommandList` object instead of a `Command` object? It is because a line read by `IO Module` could contain several `Command`s separated by pipes (which are denoted as '|', see section 2.5.1, command piping for details).

      ii. calls the `MainCommandExecutor` to handle each `Command` in the `CommandList`. A `Result` object will be returned to the `Shell`.

      iii. passes the `Result` to `IO Module` and asks it to print things to screen using the `Result`.

   We applied the top-down approach when designing this loop so that `Shell` does not need to care about:

   • how `IO Module` generates that `CommandList`;
   • how commands are executed – it just calls `MainCommandExecutor`;
   • how `IO Module` prints to screen.

   `Shell` can be considered as the top-level class in the taskManager Logic. It applies the Façade pattern. It is aware of every other component in the Logic and hides the complexity of the Logic from classes outside.

   (b) Parser

   By intuition, `Parser` is a class that parse things – `Command`s and `Result`s. It is used by instances of `IO Module` to do conversion from user input to `Command`s. That is, deduce the input text and return a `Command` object to the caller. Why `IO Module` does not parse user input itself? It is because we want `IO Module`s to do their deeds only and do them well. Parsing user input is not the job of UI. The UI is only responsible for getting the user input and printing messages to screen. This is also an application of achieving high cohesion.

   By the same argument, converting a `Result` object to text for output is doing by `Parser` as well.

(c) Command Executors

Command Executors is a collection of classes that directly manipulates underlying data structures, as its name implies. Command Executor is an abstract class that is extended by MainCommandExecutor and SubCommandExecutors such as AddCommandExecutor, LsCommandExecutor[16]... The relationship between command executors is demonstrated in the class diagram in Figure 6.
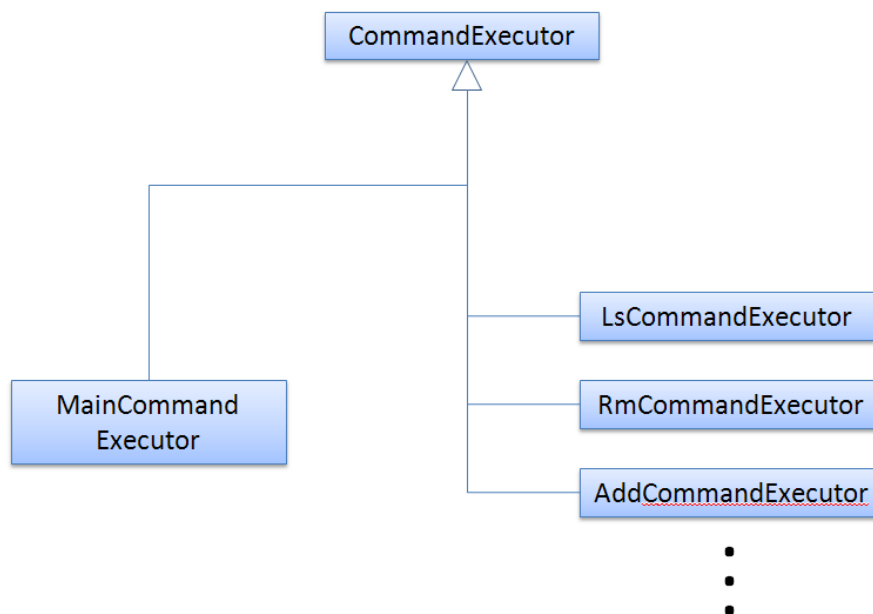

Figure 6: MainCommandExecutor and SubCommandExecutors

MainCommandExecutor takes in a Command object, figures out the type of the command and distributes the work to corresponding SubCommandExecutor. which manipulates (be it add/set/get/remove items in) the underlying data structure and return a Result object accordingly.

For example, if the Command object passed to main command executor has "ls" as its command field. The main command executor will infer this and invoke the lsCommandExecutor object and asks it to do the actual work. lsCommandExecutor will scan through all the tasks and then a result object containing all the tasks will be returned.

(d) Comparer

Comparer is implemented so as to compare different tasks according to deadline/priority/serial number... In fact sorting the tasks is made possible by Comparer. When a new comparer object is initialized, the keywords will be given to the comparer as arguments. The comparer is then able to sort some given tasks residing inside a task list according to the keywords (e.g. sort according to deadline, priority etc).

---

[16] Note that we only use SubCommandExecutor as a name to refer to these command executors, SubCommandExecutor is never in the code.

21

```
vector<Task*> TaskList::sort(Comparer* comp) {};
```

Listing 2: sorting a task list requires a Comparer. basicstyle

The code fragment above shows that sorting a task list objects requires a `Comparer` object.

(e) Filters

`Filter` is an abstract class that is defined for selecting tasks that satisfy some condition from the main task list which stores all the tasks.
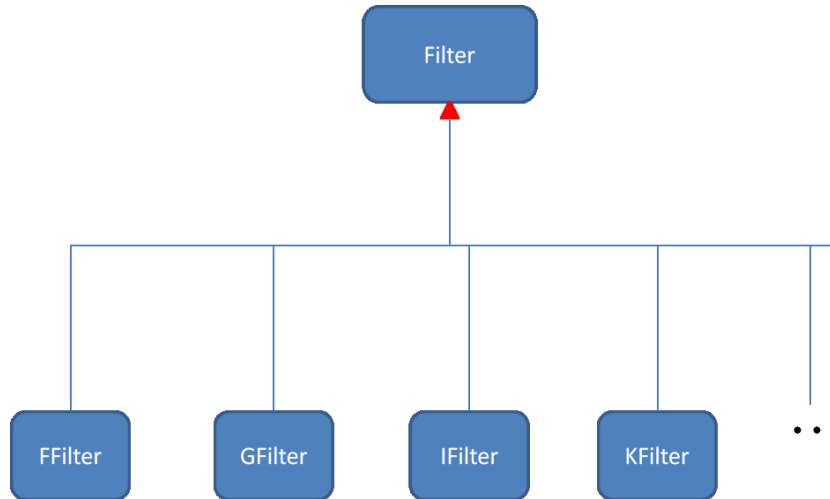

Figure 7: The Filter classes

It has subclasses like: The filter super class is basically an interface and the subclass filters are the filters that do the actual work. For example, a GFilter uses a string as its keyword. It will filter out the tasks that do not have this keyword as its group name.

(f) Underlying Data Structures

i. Task

`Task` is the class that stores various information of one task.
One thing to note is the `clone()` function. It is designed for the sake of security during development. It returns a copy of the `Task` object thus protects it from modification outside the class.

ii. Task List

`TaskList` is a container of `Task` for the ease of manipulation.
The following diagram illustrates the structure of Task List class. For the same reason above, a method `clone()` was implemented. All manipulations except for "sort" operation are all done on duplications of the task list object passed in. We want to make sure that the task list is never screwed up by manipulations in other classes.

22

(g) Auto-completion Agent

Auto-completion function is only available in the TUI. The auto-completion agent is a knowledge-based agent. Its knowledge base is initialized to all tokens in existing task descriptions and group names. Tokens are automatically added when a word is typed (only in TUI) or when a tasks is added or edited. When querying the agent with a string, the agent will answer with the shortest suffix that forms a know token with that string. If no such suffix exists, the agent will answer an empty string.

3.1.5 Testing

In this section, we are going to present some bug reports produced during the development of this project. In the later development cycles we did not choose to code an ATD as constrained by time and the complexity of our project. However, we did have carried out systematic tests in order to discover bugs.

The following is a part of the bug reports we kept:

Bug Number	Description	Fixed by
1	Most exception will still cause memory leak.	He Haocong
2	Auto completion display bug.	He Haocong
3	Debug assertion failed when trying to edit in TUI.	He Haocong
4	Parser is adding extra spaces after $0 replacement.	He Haocong
5	possible accessing string index out of range in Parser.	He Haocong
6	map bug: $0 does not match arbitray number of tokens.	Liu Jialong
7	map bug: misbehavior of "ls -g default" after map "ls" "ls -f yes".	Liu Jialong

3.1.6 Possible Extensions

Currently the product is a console application. Listed are some possible directions for extension of this project.

1. User and Authentication controls

   One possible feature that could be implemented is to make this program support multi-user environment. In order to make sure the privacy of each user, it is better to include password features.

2. GUI (Graphical User Interface) development

   The current product has a command line based user interface together with a TUI. Developers can make use of the core of this program to develop a graphical user interface. It will possibly make this product more user-friendly and easier to use.

3. Windows notification application

   Currently the product is able to help the user keep track of the tasks. Future improvement could involve some alarm-like notifications. The project team has attempted to tackle this feature. However, due to the limitation of time and knowledge, we are not able to integrate this functionality into current v0.2 product.

23

4. Remote control

The current product is only working on independent terminals. In future extensions, it is possible to extend this product to a network-support. Users are able to update their tasks lists at any terminal with Internet access.

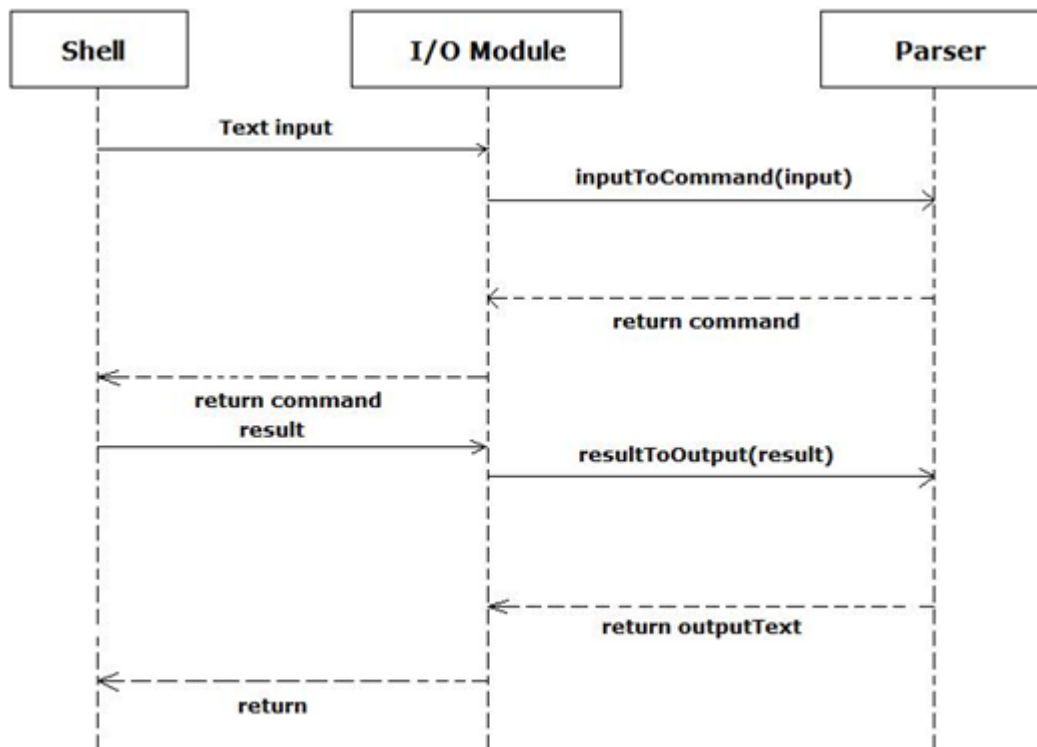## 3.2 A Typical Use Cases Explained in Sequence Diagrams

Assume we are adding a task "Write developers guide" with deadline "1d":
The top level sequence diagram looks like:
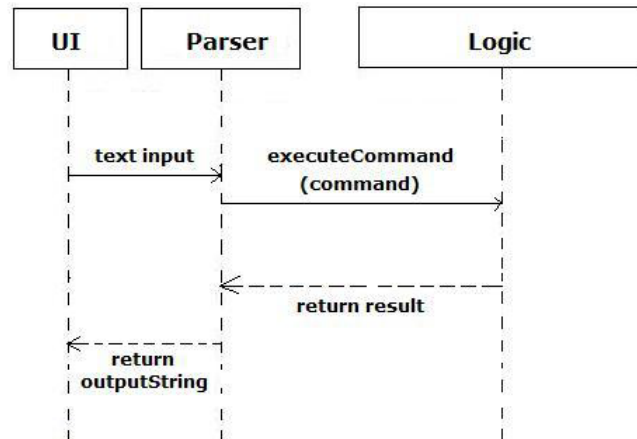
A more detailed sequence diagram inside the Logic:
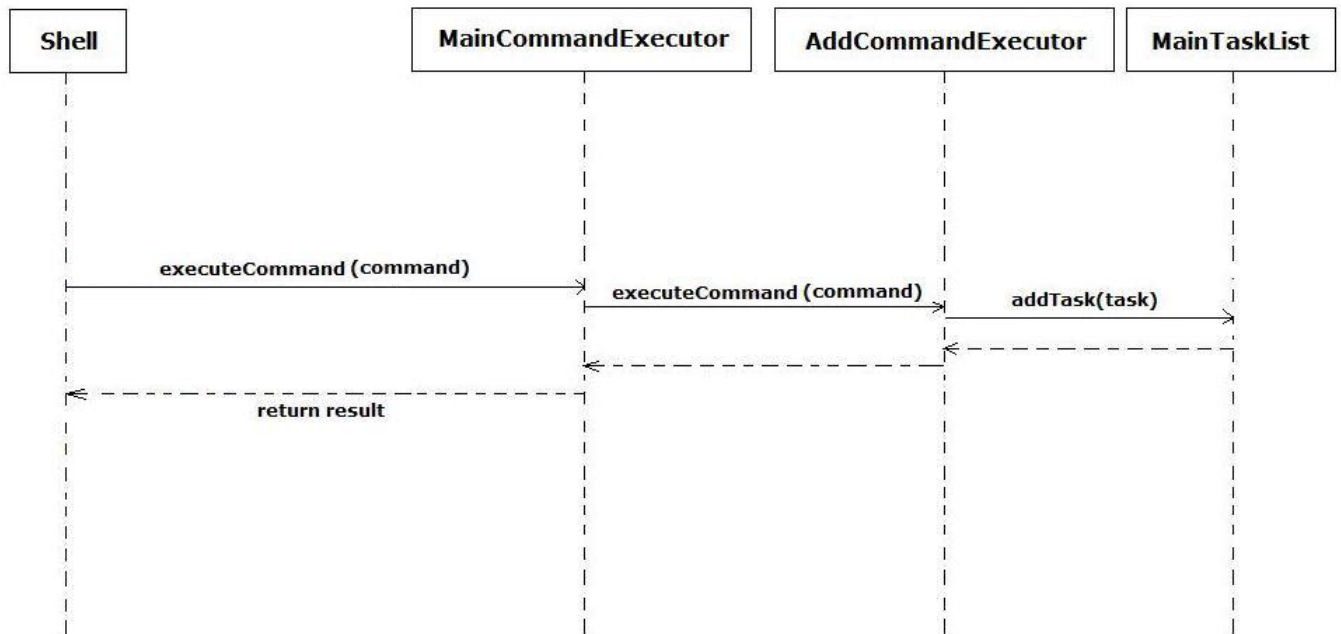
What happens inside the UI part:

3.3   Miscellaneous

1. util.cpp

   util.cpp, as its name implies, provides utility functions for different classes. These functions are not specifically related to certain classes thus we did not wrap it in a "Util" class as it increases coupling of the software otherwise.

4 Milestones and Individual Work

1. Decide project topic and brainstorming.

   Date: Sept 17
   Members attended: All

   We picked task manager as the project topic. There were conflicts among opinions as some of us do not like the GUI board game as it probably means development on Windows, while others want to have a taste of GUI programming. We finally came to agreement on the point that we were more confident to build a useful task manager that ourselves will use even after the project than a game that we are addicted to.

   So we chose task manager.

   We then came up with "cool" features like `import`, `export` and `cron` (set a regular task) and ended the meet before agreeing to meet tomorrow.

2. Design detailed architecture and develop prototype.

   Date: Sept 18
   Members attended: All

   We designed the Shell–Parser–CommmandExecutor–TaskList architecture as we thought it was an intuitive design and easy to split up work.

   We sat down together in a quiet lab and created the source files in to just print a welcome message, which is the very first (and rough) prototype. Then we added functions like adding, displaying, editting and removing tasks.

   He Haocong was responsible for the `Shell`, which acted as a UI, and the `add` and `ls` command executors.
   Liu Jialong was doing the parsing methods for the above commands.
   Zhou Biyan was taking down notes for the design — including the architecture and class definitions.

   All members participated equally.

3. Adding more commands to "taskManager".

   Date: Sept 19
   Members attended: All

   We found nothing to do so we decided to meet again to further develop the prototype. This time we finished `import` and `export` commands. We decided to store the user's tasks in an XML file. Immediately we came to the problem of storing tasks with characters like `<`, `>` and `/`. We learnt from the way web browsers handles HTML – store the special characters as "entities".

   He Haocong coded command executors `edit` and `rm`.
   Wang Xiangyu did the `import` command executor.
   Zhou Biyan was responsible for the `export` command executor.
   Liu Jialong was doing the parsing methods for the above commands.

4. Prepare for v0.1 demo

Date: Oct 7
Members attended: All

It was just a short meet before the demo for v0.1. We were there devising and rehersing a senario for demo and checking if there are any other bugs.

5. After v0.1 demo. . .

Date: Oct 9
Methods attended: All

We decided to:

   (a) support `undo`, `redo`, `run`, TUI, command piping and mapping based on the feedback by the evaluators.

   (b) give up `cron` considering the workload.

Then the details of the implementation were discussed. The architecture was not changed.

Liu Jialong was allocated to code the prasing methods.
He Haocong was responsible for the TUI and `run`, `undo` and `redo` command executors.
Wang Xiangyu was asked to find out how notifications work on Windows.

6. Rearrange the code to compile on Windows (overnight session)

Date: Oct 18 - 19
Members attended: He Haocong, Liu Jialong

The purpose of this session is to extract the method declaration to .h and definition to .cpp files to make the code standard.

As we were developing on *nix, we realized that squeezing all the classes in the .h files will fail to compile on Windows using Visual Stdio. After reconstrcuting the codes, we are able to compile taskManager on Windows and write a better Makefile by the way – the previous Makefile contains only one rule and one command, which is something like

```
g++ -o taskManager main.cpp Shell.h Parser.h MainCommandExecutor.h ...
```

We got to know better of C++ compilation process by writing the new Makefile and solving the linking problems and it was worth the stay-up.

7. Feature close shop, testing and documenting. (overnight session)

Date: Nov 4 - 5
Members attended: He Haocong, Liu Jialong

Things done:

(a) Memory leak was fixed.

(b) User guide was compiled.

(c) We went throught the user guide to make sure the commands work.

(d) The taskManager startup script (`tmrc`) was enriched to provide some more natural commands.

8. Finish the last requirements for submission.

Date: Nov 6
Members attended: All

We met up to:

(a) compile the developer guide

(b) usability testing

(c) brainstorming for demo video

9. Finish the last requirements for submission.

Date: Nov 8
Members attended: He Haocong, Liu Jialong, Zhou Biyan

Fine tuning the software and proj manual. Do video.