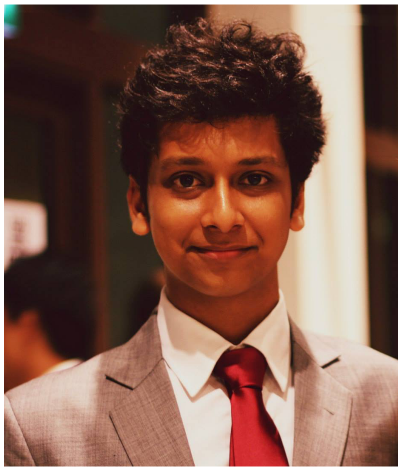
[t10-2j][V0.0]

TaskHackerPro

Supervisor: Wei Cheng Extra feature: FlexiCommands

		
YUNG MAN LEE Team lead, Scheduling and tracking	JEROME DERRICK Testing and Code Quality	ANSHUMAN MOHAN Integration and Documentation

TASKHACKERPRO

User Guide

1. Overview

TaskHackerPro is a quick and minimal desktop application that can be used to organize events, deadlines, and other to-dos. In the best case, our users will open the application as soon as they receive note of upcoming deadlines, events, or tasks. They will input details about their engagements into the app, and get on with their lives. Users are then expected to open up the application on a roughly daily basis to check on the events in the days to come. The core philosophy behind this program is that one should spend more time *doing* things, and less time *scheduling* them.

An important feature of the application is that it is sits on the user's local computer and can be run using a simple executable file. This essentially makes it a standalone application that works independently of the user's operating system, Internet coverage, etc. Further, the commands used to add and manipulate tasks are short, snappy and intuitive. This encourages users to faithfully enter all their engagements into the application.

TaskHackerPro is fuss-free, and maintains itself intelligently. As described below, users can create many different kinds of events by specifying different permutations of parameters and keywords. Users can also create events without deadlines, in which case the events go to a separate to-do list. This list can be viewed separately, so users can either assign deadlines to them, or delete them upon completion. Users also have the option of deleting a task in case it is either completed before the deadline, or no longer needs to be completed.

Advanced users may appreciate that the program maintains a log of its tasks in an accessible *.csv* file. Users are able to edit their tasks directly in this file. Once they open the program again, they can expect to see their changes reflected. If an incompatible change is made, the program automatically reverts to its last stable state. Users can share their tasks with their colleagues by sharing this *.csv* file, and requesting their coworkers to load the program using this file instead.

The developers hope that these features collectively make TaskHackerPro your schedule-manager of choice!

2. Getting Started

2.1. Launching the Program. To start using TaskHackerPro, users should run the execution file.

2.2. Viewing Tasks. When users boot up TaskHackerPro, they are treated to a view of the tasks and deadlines in the week to come. They can then navigate around their schedules by choosing simple number commands to view the next and previous weeks. Further, they can shift to *day view* (eg: *display 11/11/2015 or display this monday*) or *month view* (eg: *display jan*), and navigate similarly.

2.3. Altering Tasks. Once the tasks of interest have been brought on to the screen, users can make alterations of various forms using the task's ID number. These include:

• change time	*alter 1 as time 4pm this mon*
• change duration	*alter 1 as len 7.5 hrs*
• change location	*alter 1 as @ Pasir Ris*
• change description	*alter 1 as desc "don't forget passport this time"*
• change priority	*alter 1 as setPrior high*
• add more time to a task	*alter 1 as snooze 3 hrs*
• set a task as done	*done 1*
• delete a task	*delete 1*

Of course, many or all of the above alterations can be carried out at the same time in reasonable combinations. Some reasonable variations in the command language are also supported.

2.4. **Adding Tasks.** New tasks can be added very easily. The most basic task is one with a name and no other details. It can be added as: *add 2103 exam prep*. Since this task has no time set out for it, it is treated as a 'To Do' internally. Simple timed tasks may be added as: *add 2103 exam prep this monday for 3 hrs*. Further to this, deadlines may be added as *2103 exam prep due this monday*.

Tasks can be fleshed out further by supplying details for the following fields:

• date and time	*add biking this wed 5pm*
• duration	*add biking for 45 mins*
• location	*add biking @ clementi park connector*
• description	*add biking desc "Dover-Clementi-Ulu Pandan-· · · -Dover"*
• priority	*add biking setPrior medium*

Again, all these parameters may be filled out in one fell swoop. Many reasonable orderings and some alternate keywords are supported. For example, *add biking desc "Dover-Clementi-Ulu Pandan-· · · -Dover" this wed 5pm @ clementi park connector for 45 mins* is one of many reasonable inputs. The following is another smart feature: if the addition or alteration of a task causes two events to be scheduled simultaneously, then the program issues a gentle warning.

2.5. **Undo, Redo and History.** The latest operations may be cancelled by typing *undo*. This cancellation can be reversed by typing *redo*. A basic log of the latest commands can be viewed by typing *history*.

2.6. **Searching.** As flexible as the Display command is, it is still date-reliant. Users may run into trouble if they forget the approximate month for which a timed task was scheduled. In these cases, they can search for the task using any keywords that they do remember. Typing something like *search park connector* will pull up the event we added in the previous example.

2.7. **Saving; Storage Location.** Advanced users will appreciate that they are able to specify the location in which the program's data files are stored. These data files can be stored in a folder controlled by a cloud syncing service such as Dropbox, and then be shared across computers. This also allows users to effectively share their calendars with others. To specify a path (say /home/), users should type *save /home/* before exiting the program. If they are happy with the default location, they should just type *save*.

2.8. **Guide.** The user can pull up a list of helpful command tips by typing *help* at any time. Further to this, a detailed list of command possibilities has been provided as an appendix to this document.

Developer Guide

3. Architecture

The following figure provides a simple overview of the components of our program. While internal details are not being shown right now, it is important to take note of the way that components "talk to" each other. There is *limited* communication between components, and this is very important.

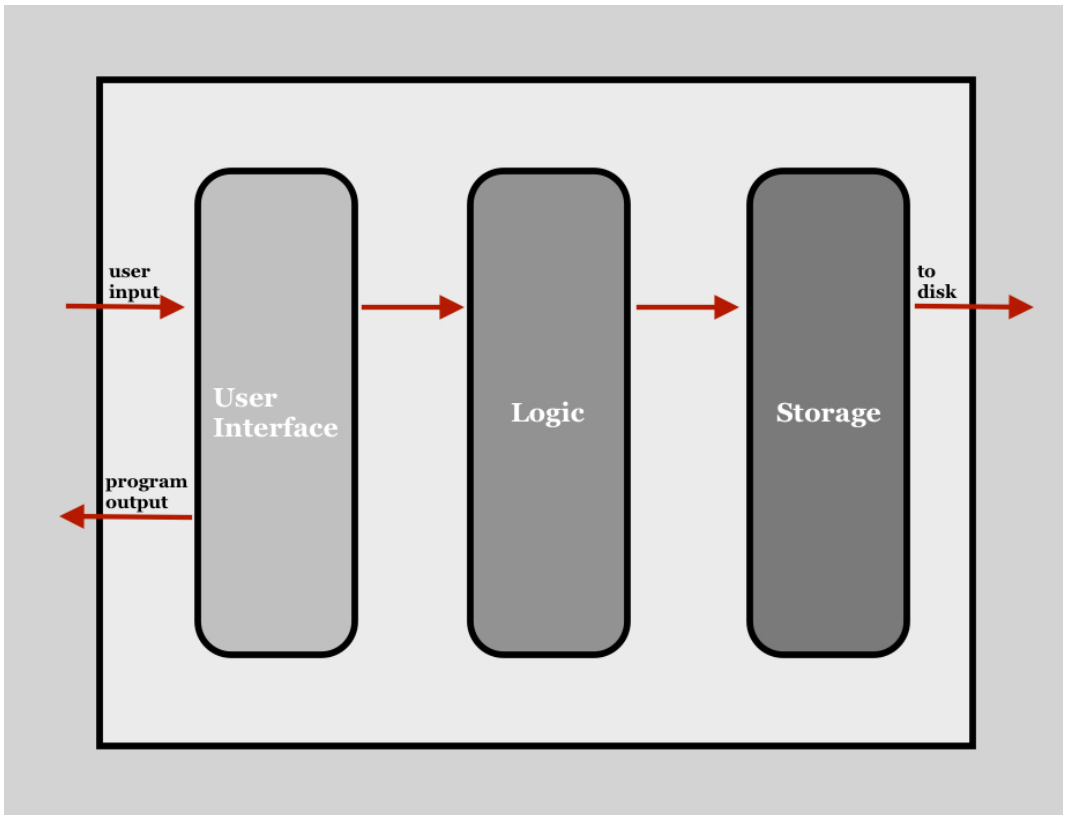


Figure 1. TaskHackerPro: An Architecture Diagram

Users interact with the UI only, the UI's various classes do little to no processing of the user's commands. Rather, they pass the job along to Logic. The meat of the work is done inside the logic-based classes: the command is checked for basic validity, following which it is parsed and interpreted. Further task-based work is handed off to baser classes, which make it their business to accept commands of a *particular type* and to do specialised work only. Once the command has been processed in this way, appropriate classes in the Storage component are invoked to help reflect these changes in the disk. Finally, confirmatory messages and/or error messages are shown to the user via the UI.

4. Architecture, In Action

In the previous section, we discussed the architecture of the program in a broad and zoomed-out way. Now we shall take a step closer, and breathe life into the plan. We provide some *activity diagrams* and use them to discuss the behavior of the system.

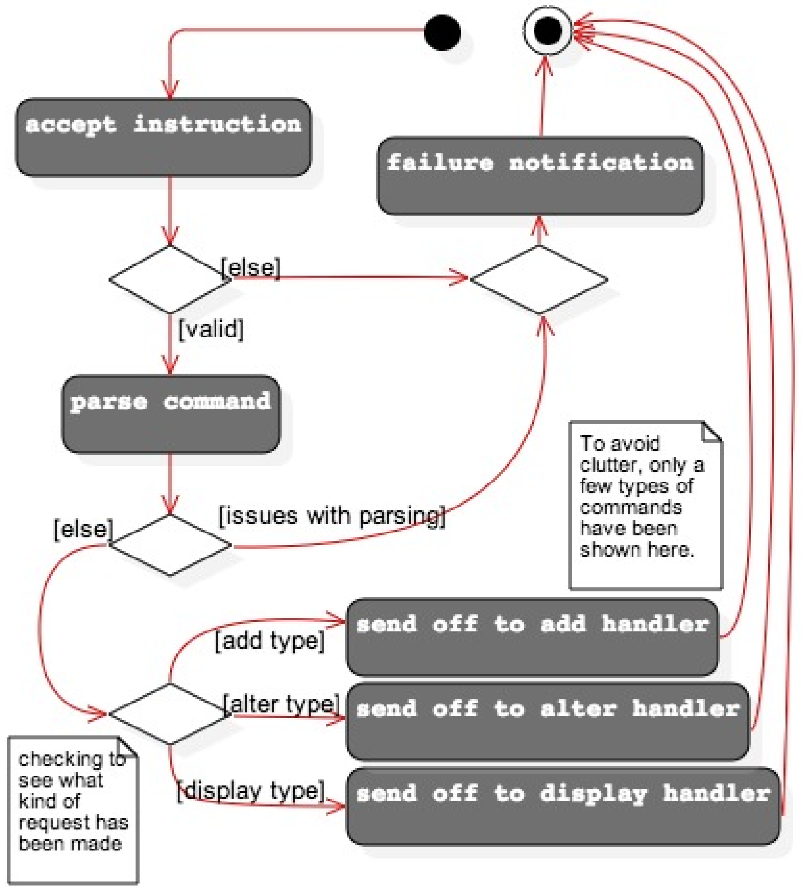
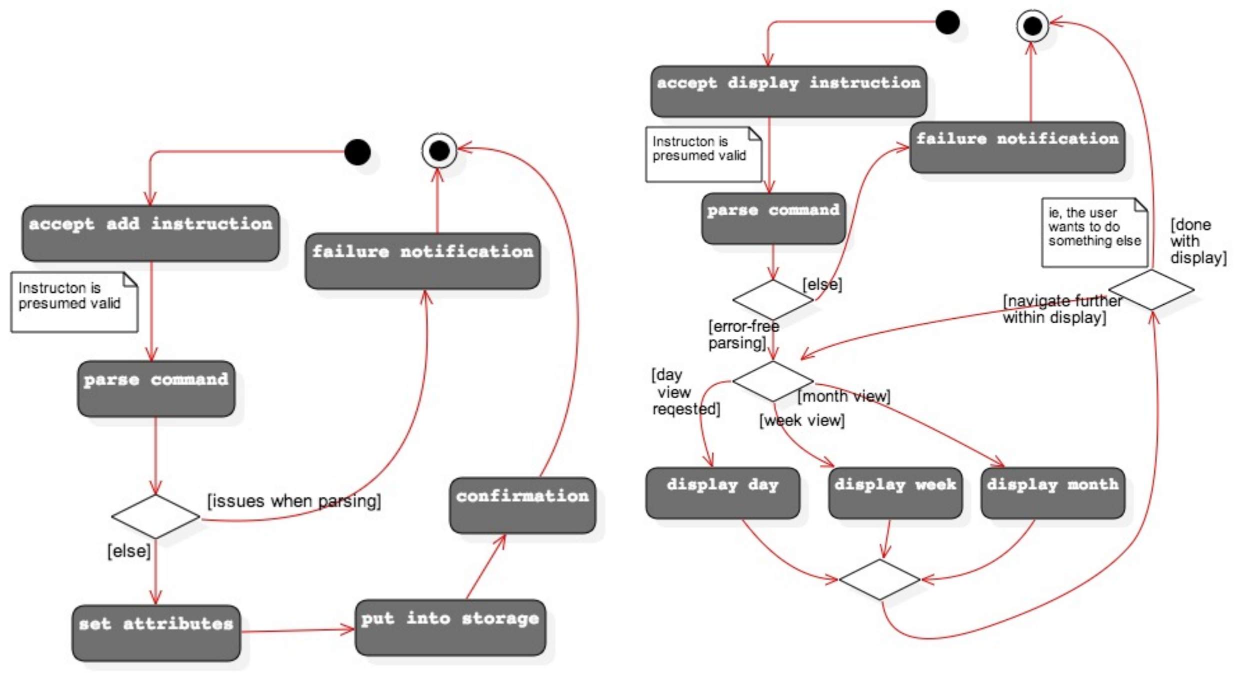

accept instruction
failure notification
[else]
[valid]
parse command
[issues with parsing]
[else]
To avoid clutter, only a few types of commands have been shown here.
[add type]
send off to add handler
[alter type]
send off to alter handler
[display type]
send off to display handler
checking to see what kind of request has been made

Figure 2. Activity Diagram for General Parsing

As may be apparent from the above diagram, our program is required to deal with several different kinds of commands, which can be attended to quite independently of each other. This makes our program a fine candidate for the Command-Handler pattern. To explain our program at a high level, any valid inputted command first undergoes primary parsing where we figure out what exactly is being requested. Then an object of the appropriate 'handler' class is created, and the information is passed along. The storage is then accessed separately, and the changes made are reflected as needed. Clearly, having multiple instances of the storage would be problematic. We prevent the instantiation of multiple objects from the storage by following the Singleton pattern in the relevant class (i.e. DataManager).

Since classes are accessed and objects are instantiated *only as and when needed*, individual objects cannot communicate unnecessarily. This is in tune with well-loved principles of software development such as Separation of Concerns and the Law of Demeter.



(A) Activity Diagram for Adding Events (B) Activity Diagram for Displaying Events

Figure 3. Going Deeper with Activity Diagrams

As the above discussion and diagrams have shown, our program relies on a 'storage' class called DataManager. Inside this class, we actually maintain a hashmap containing all the events. The rest of the program basically involves manipulating and playing around with this hashmap. When so many parts of the code interact with the same object, we obviously worry about having tight coupling. In our program, we have tried to reduce coupling by doing the following:

- By using the Singleton pattern when designing the DataManager class
- By minimizing the number of classes interacting with DataManager, so that only essential interactions occur.
- By providing a 'getter' function, so that other classes can safely request access to the internal hashmap, instead of just diving in and accessing DataManager's internal methods.

We have tried to achieve high cohesion by implementing several classes that are quite focused and have methods geared only to a single tasks. Because these classes are so good at what they do, and because they provide such a thorough and comprehensive API, they can be reused by various classes in different ways, depending on what is needed. For example,

- The Event class instantiates the atomic building block of our program: the event! It also allows one to 'set' or 'get' the individual parameters of any event. Its different methods are used extensively both when adding events and when altering them.
- The CommandParser class provides an clever set of tools that is able to take an instruction and figure out what is needed to be done. It, too, is reused heavily when adding and altering events.

5. Looking Into Components

Having discussed the workings of the program as a whole, we shall now take a closer look at the individual components of our architecture. To make this more readable, only a representative selection of the classes inside each component have actually been shown. These diagrams are automatically generated, and will not be discussed. However, they should substantiate and explain the things discussed above.

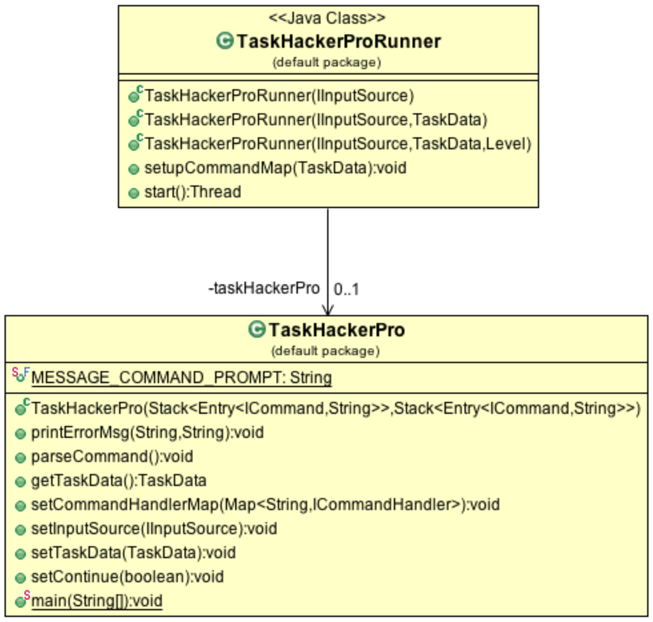

<<Java Class>>
TaskHackerProRunner
(default package)
TaskHackerProRunner(IInputSource)
TaskHackerProRunner(IInputSource,TaskData)
TaskHackerProRunner(IInputSource,TaskData,Level)
setupCommandMap(TaskData):void
start():Thread
-taskHackerPro
0..1
TaskHackerPro
(default package)
MESSAGE_COMMAND_PROMPT: String
TaskHackerPro(Stack<Entry<ICommand,String>>,Stack<Entry<ICommand,String>>)
printErrorMsg(String,String):void
parseCommand():void
getTaskData():TaskData
setCommandHandlerMap(Map<String,ICommandHandler>):void
setInputSource(IInputSource):void
setTaskData(TaskData):void
setContinue(boolean):void
main(String[]):void

Figure 4. Class Diagram for UI's Most Important Classes

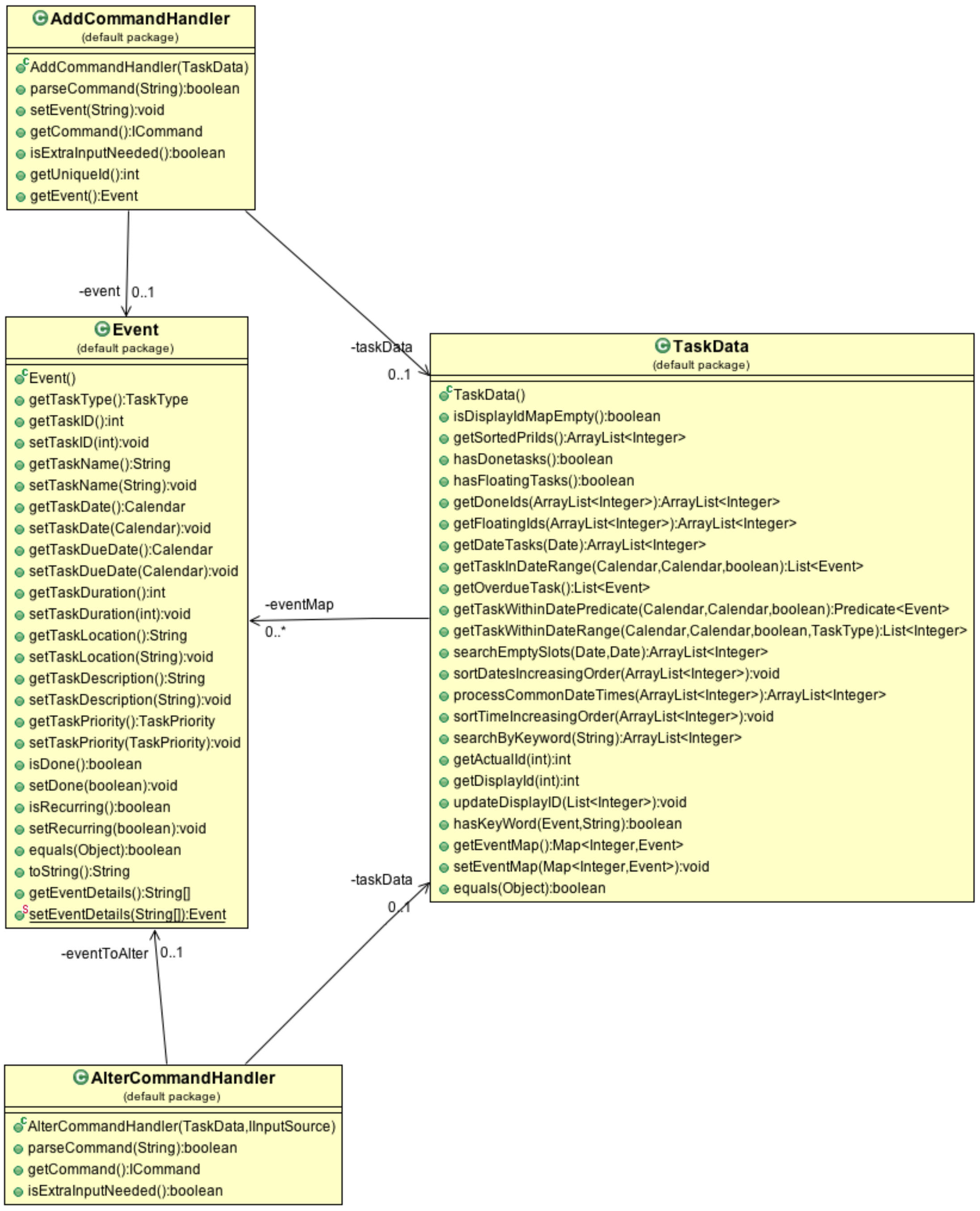
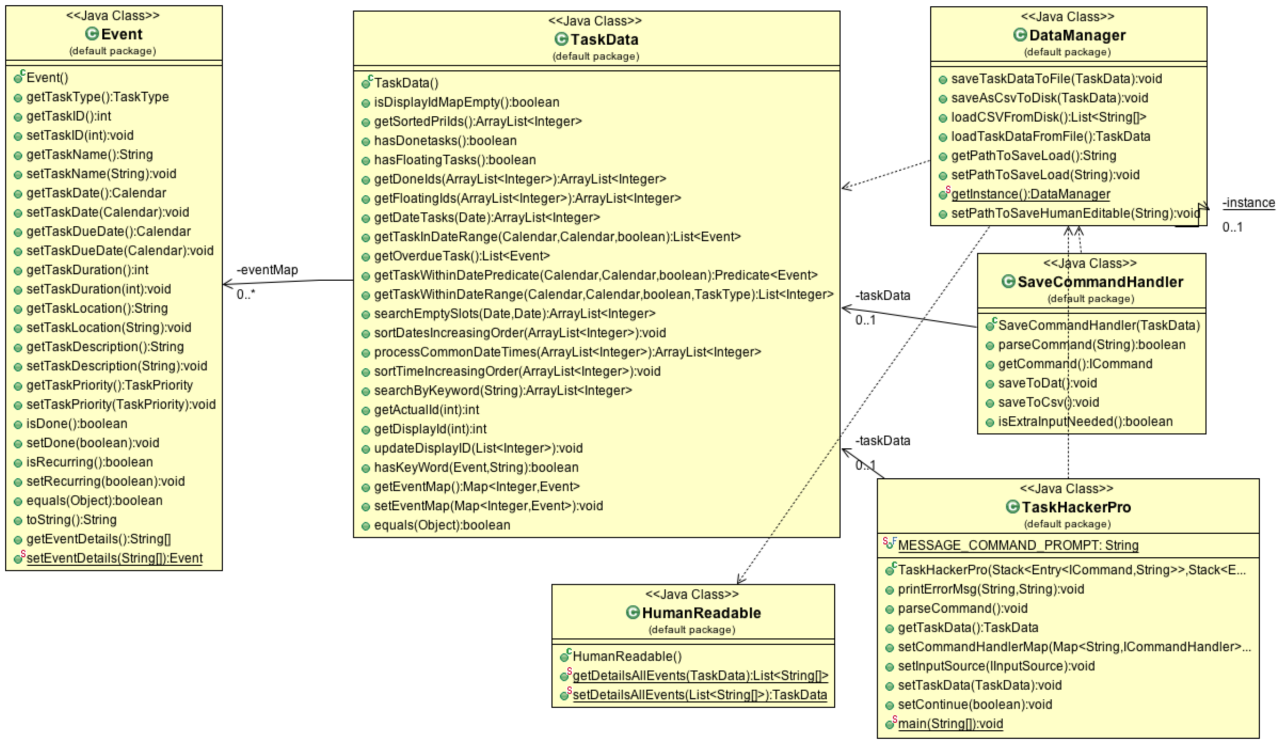

AddCommandHandler
(default package)
AddCommandHandler(TaskData)
parseCommand(String):boolean
setEvent(String):void
getCommand():ICommand
isExtraInputNeeded():boolean
getUniqueId():int
getEvent():Event
-event
0..1
-taskData
0..1
Event
(default package)
Event()
getTaskType():TaskType
getTaskID():int
setTaskID(int):void
getTaskName():String
setTaskName(String):void
getTaskDate():Calendar
setTaskDate(Calendar):void
getTaskDueDate():Calendar
setTaskDueDate(Calendar):void
getTaskDuration():int
setTaskDuration(int):void
getTaskLocation():String
setTaskLocation(String):void
getTaskDescription():String
setTaskDescription(String):void
getTaskPriority():TaskPriority
setTaskPriority(TaskPriority):void
isDone():boolean
setDone(boolean):void
isRecurring():boolean
setRecurring(boolean):void
equals(Object):boolean
toString():String
getEventDetails():String[]
setEventDetails(String[]):Event
TaskData
(default package)
TaskData()
isDisplayIdMapEmpty():boolean
getSortedPriIds():ArrayList<Integer>
hasDonetasks():boolean
hasFloatingTasks():boolean
getDoneIds(ArrayList<Integer>):ArrayList<Integer>
getFloatingIds(ArrayList<Integer>):ArrayList<Integer>
getDateTasks(Date):ArrayList<Integer>
getTaskInDateRange(Calendar,Calendar,boolean):List<Event>
getOverdueTask():List<Event>
getTaskWithinDatePredicate(Calendar,Calendar,boolean):Predicate<Event>
getTaskWithinDateRange(Calendar,Calendar,boolean,TaskType):List<Integer>
searchEmptySlots(Date,Date):ArrayList<Integer>
sortDatesIncreasingOrder(ArrayList<Integer>):void
processCommonDateTimes(ArrayList<Integer>):ArrayList<Integer>
sortTimeIncreasingOrder(ArrayList<Integer>):void
searchByKeyword(String):ArrayList<Integer>
getActualId(int):int
getDisplayId(int):int
updateDisplayID(List<Integer>):void
hasKeyWord(Event,String):boolean
getEventMap():Map<Integer,Event>
setEventMap(Map<Integer,Event>):void
equals(Object):boolean
-eventMap
0..*
-taskData
0..1
-eventToAlter
0..1
AlterCommandHandler
(default package)
AlterCommandHandler(TaskData,IInputSource)
parseCommand(String):boolean
getCommand():ICommand
isExtraInputNeeded():boolean

Figure 5. Class Diagram for Logic's Most Important Classes



<<Java Class>>
Event
(default package)
Event()
getTaskType():TaskType
getTaskID():int
setTaskID(int):void
getTaskName():String
setTaskName(String):void
getTaskDate():Calendar
setTaskDate(Calendar):void
getTaskDueDate():Calendar
setTaskDueDate(Calendar):void
getTaskDuration():int
setTaskDuration(int):void
getTaskLocation():String
setTaskLocation(String):void
getTaskDescription():String
setTaskDescription(String):void
getTaskPriority():TaskPriority
setTaskPriority(TaskPriority):void
isDone():boolean
setDone(boolean):void
isRecurring():boolean
setRecurring(boolean):void
equals(Object):boolean
toString():String
getEventDetails():String[]
setEventDetails(String[]):Event
-eventMap
0..*
<<Java Class>>
TaskData
(default package)
TaskData()
isDisplayIdMapEmpty():boolean
getSortedPriIds():ArrayList<Integer>
hasDonetasks():boolean
hasFloatingTasks():boolean
getDoneIds(ArrayList<Integer>):ArrayList<Integer>
getFloatingIds(ArrayList<Integer>):ArrayList<Integer>
getDateTasks(Date):ArrayList<Integer>
getTaskInDateRange(Calendar,Calendar,boolean):List<Event>
getOverdueTask():List<Event>
getTaskWithinDatePredicate(Calendar,Calendar,boolean):Predicate<Event>
getTaskWithinDateRange(Calendar,Calendar,boolean,TaskType):List<Integer>
searchEmptySlots(Date,Date):ArrayList<Integer>
sortDatesIncreasingOrder(ArrayList<Integer>):void
processCommonDateTimes(ArrayList<Integer>):ArrayList<Integer>
sortTimeIncreasingOrder(ArrayList<Integer>):void
searchByKeyword(String):ArrayList<Integer>
getActualId(int):int
getDisplayId(int):int
updateDisplayID(List<Integer>):void
hasKeyWord(Event,String):boolean
getEventMap():Map<Integer,Event>
setEventMap(Map<Integer,Event>):void
equals(Object):boolean
<<Java Class>>
DataManager
(default package)
saveTaskDataToFile(TaskData):void
saveAsCsvToDisk(TaskData):void
loadCSVFromDisk():List<String[]>
loadTaskDataFromFile():TaskData
getPathToSaveLoad():String
setPathToSaveLoad(String):void
getInstance():DataManager
setPathToSaveHumanEditable(String):void
-instance
0..1
-taskData
0..1
<<Java Class>>
SaveCommandHandler
(default package)
SaveCommandHandler(TaskData)
parseCommand(String):boolean
getCommand():ICommand
saveToDat():void
saveToCsv():void
isExtraInputNeeded():boolean
-taskData
0..1
<<Java Class>>
TaskHackerPro
(default package)
MESSAGE_COMMAND_PROMPT: String
TaskHackerPro(Stack<Entry<ICommand,String>>,Stack<E...
printErrorMsg(String,String):void
parseCommand():void
getTaskData():TaskData
setCommandHandlerMap(Map<String,ICommandHandler>...
setInputSource(IInputSource):void
setTaskData(TaskData):void
setContinue(boolean):void
main(String[]):void
<<Java Class>>
HumanReadable
(default package)
HumanReadable()
getDetailsAllEvents(TaskData):List<String[]>
setDetailsAllEvents(List<String[]>):TaskData

Figure 6. Class Diagram for Storage's Most Important Classes

6. The Armory: API

Below, we note the key aspects of each component's API.

UI

(1) parseCommand() [sends the commmand to Logic]
(2) showWelcomeMessage()

Logic

(1) getDetailFromCommand() [does some very detailed breakdown and matching]
(2) parseCommandSegment() [similar to above]

Storage

(1) saveTaskDataToFile() [writes event information to disk]
(2) loadTaskDataFromFile() [finds and returns a all the event information from the disk]
(3) saveAsCsvToDisk() [saves the information to disk in human readable form]
(4) loadCSVFromDisk() [loads the human-readable file from the disk]

7. Code Samples

Please refer to the appendix for a sample of our code, along with an explanation.

8. Testing

Given that TaskHackerPro is a product in development, its developers are excited to bring new software engineers on board. That said, however, the following instructions, laid down by the current team, are to be followed strictly by all programmers, both old and new.

First, install the tools required by the project. These include *Eclipse* and *Gradle*. (See the appendix for details.) Before planning or writing new code, study the UML diagrams above and make sure you understand how your new code will interact with the other existing parts. Speak to the other authors if you need.

Once you have written code, run the JUnit test suite to make sure that your changes have not affected or broken the program in unexpected ways. This is an example of regression testing. Please note that our existing testing suite also places great emphasis on integration testing. As such, all commands are tested via their respective command handlers, and the test run for a single command often involves many different parts of the program. This is a step up from naïvely invoking the methods of a single class, and simulates actual usage a little better. Once your code passes the existing tests, use the existing unit- and integration-tests as templates to add new test cases that rigorously examine the functionality of your new code. An obvious alternative to the above is to follow TDD. In that case, *only* your newly added test cases should be failing, and nothing else. Optionally and additionally, you may also run a similar testing suite using Gradle. Regardless of the method and the software used, you must fix any and all bugs before your code is considered safe.

Once you have the green signal from the testing suite, you should refactor your code, test again, and then push your commits to a new branch in the remote repository. If your code is meant to close an issue, then close your issues via a commit message in your last commit. In the remote repository, merge your newly created branch to the *develop* branch. (Never to *master*) Ensure that the newly-merged *develop* is passed by Travis CI, which is another testing framework we are using (see appendix for more on this). If you break the build on Travis CI, you must work on the code some more and fix it. Once you get the green signal from Travis CI, create a new pull request to *master* and request a code review from the team leader. Never close any issues manually; issues will be closed automatically once the relevant commit is successfully merged to *master*. Once your code is merged to *master*, delete the new branch you created.

Code Sample

```
private static Map<String, String> parseCommandSegment(Pattern pattern,
            Set<String> groups, List<Entry<String, String>> list) {

        Matcher patternMatcher;
        Map<String, String> returnMap = new HashMap<String, String>();

        Iterator<Entry<String, String>> it = list.iterator();
        while (it.hasNext()) {
            Entry<String, String> entry = it.next();
            String keyword = entry.getKey();
            String value = entry.getValue();

            patternMatcher = pattern.matcher(keyword + CHAR_SPACE + value);
            if (patternMatcher.find()) {
                for (String group : groups) {
                    String matchedString = patternMatcher.group(group);
                    if (matchedString != null) {
                        returnMap.put(group, matchedString);
                    }
                }
                int start = Math.max(0, patternMatcher.start() - keyword.length() - 1);
                int end = patternMatcher.end() - keyword.length() - 1;
                String newValue = value.substring(0, start) + value.substring(end);

                if (STRING_NULL.equals(newValue)) {
                    it.remove();
                } else {
                    entry.setValue(newValue);
                }
            }
        }
        return returnMap;
    }
```

This function is the first parsing step of the natural language input. It is used to separate the commands followed by keywords in add. The function receives the pattern needed to be checked, the list of keywords, and the group names in regular expressions. It parses the commands by identifying keywords and values which matches the pattern, then put the pair keyword and value into a map for further process.

Non-Functional requirements

The developers decided to prioritize the following non-functional requirements before they started designing this product:

Accessibility
Ability to access with ease and benefit from the system. User can achieve specified goals with effectiveness and efficiency in a specified context of use.

Extensibility
Ability to have function extended,in which the system's internal structure and data flow are minimally or not affected, particularly that recompiling or changing the original source code is unnecessary when changing a system's behavior.

Fault tolerance
The system should able to continue to work properly if the user makes mistakes when typing. Examples of this include multiple faulty commands, misspellings, etc.

Maintainability
Learning from the past in order to improve the ability to maintain systems, or improve the ability to maintain systems, or improve reliability of systems based on maintenance experience.

Portability
Able to use the software in different environments. When the software with the same functionality is produced for several computing platforms, portability is the key issue for development cost reduction.

Product Survey

The developers studied other existing products before designing TaskHackerPro.
Some positive and negative aspects of existing products are as under:

Product: Google Calendar **Documented by**: Yung Man Lee

Strengths:
- Syncs with email
- Quick add function with examples and documentations
- Locations link to Google Map
- Has agenda view, day view, week view and month view
- Allows reminders for events
- Can separate events to different task list
- Notifications allows to be postponed / snoozed
- Allow recurring events
- Can add events according to time zone
- Can add details for the events
- Can highlight an event with colors
- Allow to use off-line
- Can share the calendar with other user
- Can Drag-and-Pull to change to date of an event (Easy management)
- Can undo a recent operations

Weaknesses:
- No manipulations for expired events
- No warnings if time clashed
- Can not add a floating task
- Cannot mark a task as done
- Errors in location easily to be occurs in quick add mode

Product: Windows 8.1 calender **Documented by**: Jerome Derrick

Strengths:
- Synchronise with email
- Invite someone to an event or meeting
- Events can be color coded
- Easily switch between month, week, day and agenda view
- Built in task list that can be edited by clicking the View button near the top-right of the screen and selecting Task.
- Recurring events can be set with ease.
- Has keyboard shortcuts for creating task, editing etc.
- Windows 8 calender Color coding helps user to see overlapping events
- Prevents user from over scheduling
- Can set recurring task or schedules easily

- Has the ability to synchronous its calendar with other accounts
- can still function in offline mode

Weaknesses:
- There is no outstanding list to alert the user if a task has passed its due date.
- Important tasks cannot be highlighted with colors
- Complicated with too many key strokes for a simple function
- Interface is not user friendly when navigating, editing etc.
- Not able to add in a note for a task
- No customization of keyboard shortcuts
- Has no command line feature
- No summary of upcoming task displayed to the user after starting up calender app
- Unable to swap tasks instantly. The user has to delete and create the same task again because of manual swap
- Insufficient GUI makes it very hard for the user to use Windows 8 calendar

Product: Outlook Calendar **Documented by**: Anshuman Mohan

Strengths:
- Can enter the time needed to complete a task
- Automatically signals clashes in scheduling, but allows both events to be stored
- Allows recurring tasks to be set in one go
- Allows tasks to be dropped/deleted without completing
- Supports month view, week view, day view
- Allows the addition of tags such as location, notes, etc
- Synchronises with with email and proposes the addition of events to the calendar
- Allows others to add events to calendar (subject to the user's approval)
- Allows tasks to be "snoozed"
- Very easy to use; aesthetically pleasing

Weaknesses:
- Has no command line / quick add feature. The user must click and type in order to add anything.
- Completed items are not deleted or marked as completed. I guess they can just be ignored
- The user cannot add tasks without specifying a date and a time
- It does no check to see if a task was completed or not. No "overdue" feature.
- Does not feature advanced quick shortcut
- The user cannot teach it any new shortcuts
- It has no user guide, but is very easy to understand even by itself
- One cannot set "priority" for an event
- It provides no automatic summaries
- Cannot use offline

Appendices

1: Development Tools

The developers currently use the following tools:

(1) Java SE Development Kit 8 (1.8.0.25+)
(2) Eclipse IDE for Java Developers (Luna SR2 4.4.2)
(3) SourceTree 2.0.3
(4) Gradle 2.2.1

New developers should install the Gradle plug-in for Eclipse from the Eclipse Marketplace. They should clone the project from *https://github.com/cs2103jan2015-t10-2j/main.git*, and use Gradle to build a model locally. They are then good to go.

2: Non-Functional Requirements

The developers decided to prioritize the following non-functional requirements before they started designing this product:

(1) Accessibility: Ability to access with ease and benefit from the system. User can achieve specified goals with effectiveness and efficiency in a specified context of use.
(2) Extensibility: Ability to have function extended,in which the system's internal structure and data flow are minimally or not affected, particularly that recompiling or changing the original source code is unnecessary when changing a system's behavior.
(3) Fault tolerance: The system should able to continue to work properly if the user makes mistakes when typing. Examples of this include multiple faulty commands, misspellings, etc.
(4) Maintainability: Learning from the past in order to improve the ability to maintain systems, or improve the ability to maintain systems, or improve reliability of systems based on maintenance experience.
(5) Portability: Able to use the software in different environments. When the software with the same functionality is produced for several computing platforms, portability is the key issue for development cost reduction.

3: Detailed List of Commands Supported

If the user wishes to specify the location to which the program saves output files, (in order to sync with a cloud sharing device, for example) they should type "save" followed by the desired location before exiting the program.

When adding or altering tasks, the following keywords may be used to specify various fields:

(1) Specify the task's date:
   (a) Just enter a date in the dd/mm/yyyy format.
   (b) Enter the next/previous day by saying "yesterday" or "tomorrow".
   (c) Enter a day by saying the day's name. For example,"this monday", "next monday", etc.
   (d) Enter a date exactly seven days later by saying "next week".
   (e) Enter a date exactly a month later by saying "next month".
(2) Specify the task's time:
   (a) Enter a time in the 24-hour format
   (b) Enter a time by saying "at 5pm", "@ 5:05pm", etc.
(3) Give the duration in hours or in minutes: "for 60 minutes", "for 2 hours", etc.
(4) Set a description by typing "desc" followed by a description in quotes.
(5) Set a due date by saying "due" followed by a valid date, day, etc.

(6) Set a location by saying “@” followed by a location
(7) Set a priority by saying “setPrior” followed by high, medium, or low.

When displaying tasks, the following options are available:

(1) month view: “display month” (can then toggle to next and previous months)
(2) week view: “display week” (can then toggle to next and previous weeks)
(3) day view: “display today”, “display tomorrow”, “display mon”, “display tues”, etc. (can then toggle to next and previous days)
(4) tasks that have been completed: display done
(5) tasks that are unscheduled (no time has been set): display checklist
(6) display items sorted by priority: display priority

A task can be set as Done by pulling it up in display, and then simply saying “done” followed by its index number.

A clever and robust search has been made available, to allow users to search for events by keyword.