Jimi – Task Manager Program

Han En Chou
Lead Programmer,
Documenter

Mah kai xin Amanda
Lead GUI Developer,
Documenter, Tester

Erin Teo Yi Ling
Team Leader, Programmer,
Documenter

Contents

User Guide

What is it?

Jimi is a simple and easy to use task manager program that is great for people who are swamped with things to do and need a way to organize their schedules. You interact with the program only with user commands in the form of text much like a command-line interface but you don't have to remember or follow complicated syntaxes, because the program recognizes natural language (but with minor limitations).

How it works

When Jimi starts up, the program will display "today's to-do list" which is a list of all of the tasks (in ascending order by time) that you have to do today. Important tasks are highlighted in red text. You can then proceed to giving the program commands to efficiently manage your tasks. To get started, simply install Jimi and hit ALT+SHIFT+J!

Overview of functionalities

- Adding a task/event
- Searching for a task/event
- Undoing/Redoing a user command
- Marking a task/event
- Editing a task/event

Entering the user command "help" or "?" on Jimi will also give you a quick guide on how you can get started! Also remember that Jimi is really flexible with whatever commands you may enter. The command formats specified below are just a guideline. Give it a shot!

Adding a task/event

- You can add a task to your existing list of tasks by using the "add" command.
- The basic add command follows this format:
  add <task name> by <deadline> at <location>
  add <task name> at <location> <date/time>

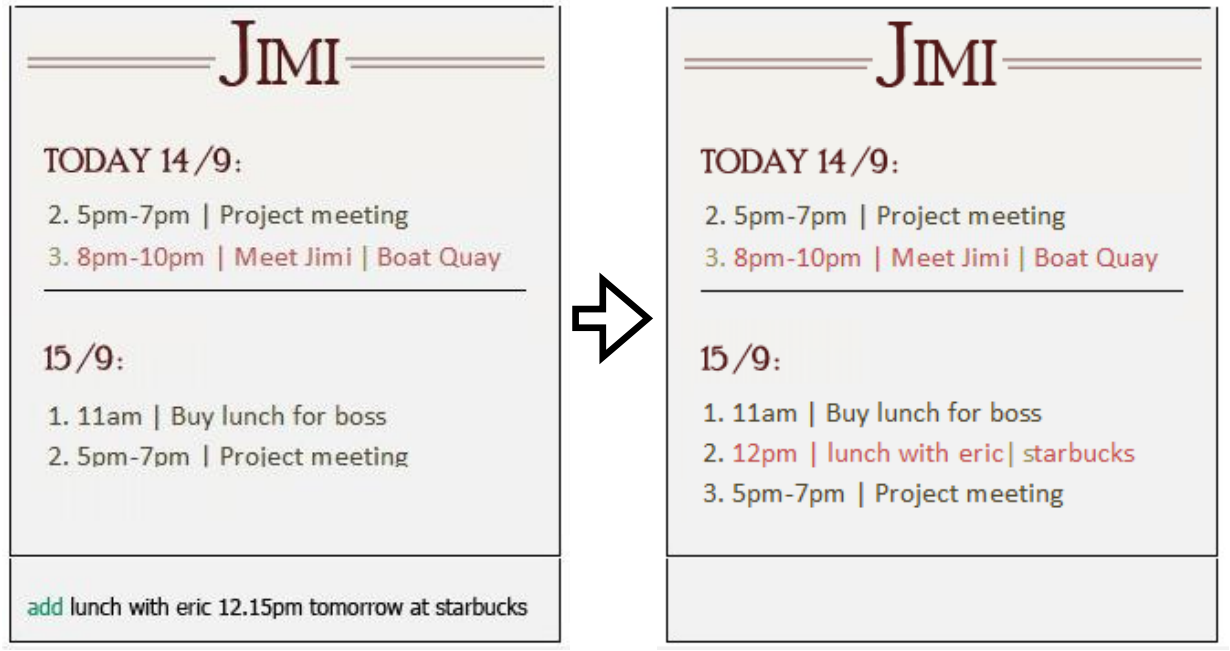
and many other variations. Keywords like "insert", "create", "put" can be used in place of "add", use whatever you like!

Examples would include:

- add lunch with boss 12.15pm tomorrow at starbucks
- add project meeting, 4pm-5pm on 7 Aug
- insert email mom by next Friday
- add skating practice wed 16:00

The add command recognizes different date and time formats and also takes note of the location of the task if specified. If the date is not defined then the program will put the task on the earliest date in the future with the specified time. If the time is not specified then the program will assign the task the time 12am.

- Additional add command features:
  - Adding "impt" or "important" to your command marks the task as important. For example: "add lunch with boss 11am tomorrow at starbucks impt". Important tasks appear as red text when displayed.



Searching for a task/event

- You can easily search for tasks based on its name, date or both by using the "search" command.
- The basic search command follows this format:
  search <query>

Keywords like "find", "display", "show" can be used in place of "search".

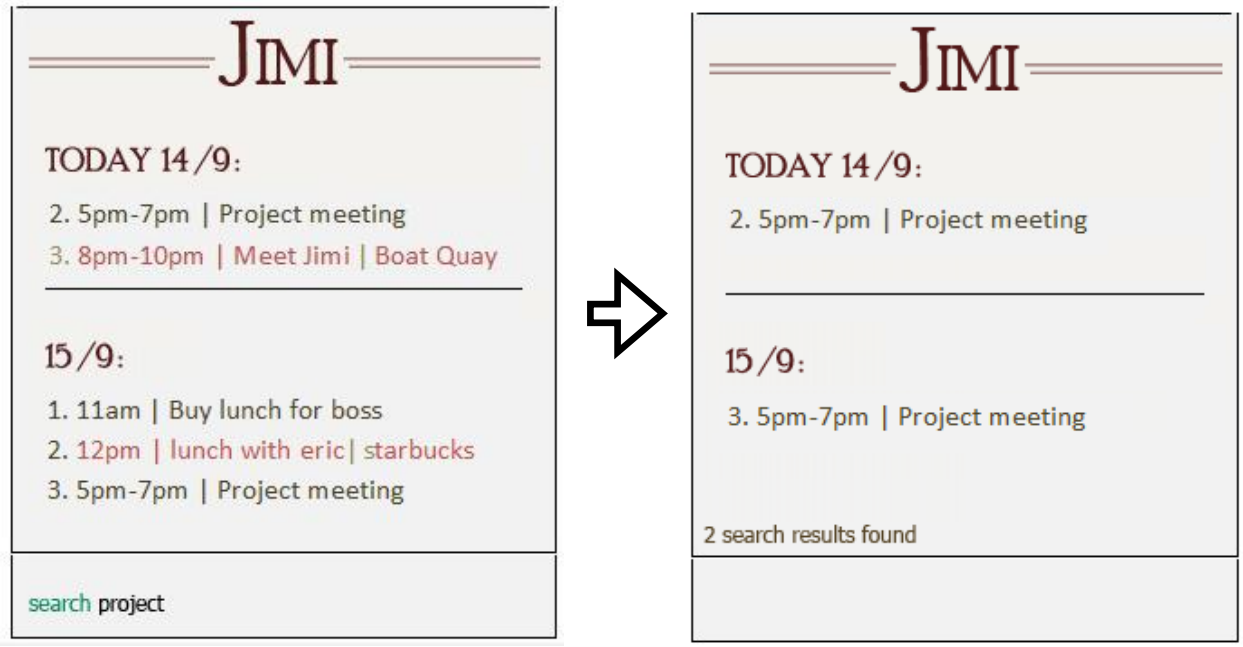
- Additional search command features:
  - Users can search for tasks within a certain timeline by adding a dates after the search command. Users should specify only a maximum of two dates: start date and end date. Jimi will search for all tasks between these dates. If only one date is specified, Jimi will return all tasks on that particular day. The date(s) can be accompanied by a query which also searches for tasks that match the dates and query specified. If only the query is specified, Jimi will search for all tasks that match the specified query. Using the keywords before or after users can search for tasks before or after a certain date. The command would be issued as such: "search <before/after > <date/time>"

Examples would include:

- search project
- search before 7 Aug
- find after today
- show tmr
- search project 7 Oct to 9 Oct
- search between 7 Aug to 9 Aug

After search command:

Jimi searches for dates really quickly! Program will display search results in the format <date> | <time> | <task name> as a list. The list will be sorted in chronological order and tasks that are marked as important will be in red text.



Undoing/Redoing a user command

- You may undo or redo a previously issued command using the "undo" or "redo" commands.
- If an undo/redo command is issued, program will undo or redo previously issued user commands. Commands that can be undone/redone are add, mark and edit.

Marking a task/event

- When you are done with certain tasks, you might want to mark it as done and get it off the list of tasks. You might do so using the "mark" command.
- The basic mark command follows this format:
  mark <task name>

mark <task index>

The task index comes from the most recently displayed list of tasks. Keywords like "check", "done" can be used in place of "mark". Task names have to more or less match the task with the name you want to mark.

Examples would include:

- mark special summer project
- mark 2

After mark command:

The program will not display this task subsequently.

Editing a task/event

- With the "edit" command, you can change certain details of a task.
- The basic edit command follows this format:
  edit <date><task name> <details you want to change>

edit <task index> <details you want to change>

The task index comes from the most recently displayed list of tasks. Keywords like "update", "change", "postpone", "delay", "advance" can be used in place of "edit".

Examples would include:

  - edit project to big project
  - edit 14/9 adventure time 8pm

- After the you update the task, the program will display the edited task in this format:
  "Task successfully edited: <date><time> <task name>"

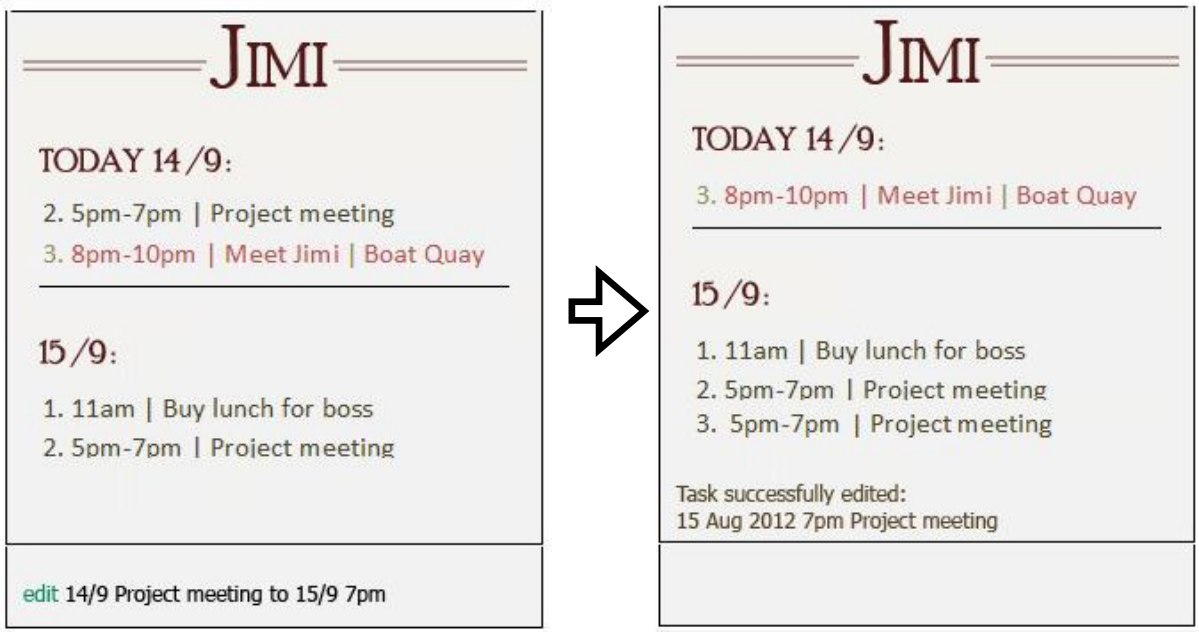
For example:

"Task successfully edited: 8 Sep 2012 10.00am Compile Project"

- Additional edit command features:
  - Postponing a task: Similar to the basic edit command but only involves only changing the time or deadline of a task. It follows this format: "edit <task index> <date><time>" or "edit <task name> <date><time>" ". Instead of putting the exact date and time, the user can also state the number of days or hours relative to the original date and time.

Examples would include:

- postpone 2 11am 7 Aug
- delay 3 two days later
- postpone special project 1 hr


JIMI
TODAY 14/9:
2. 5pm-7pm | Project meeting
3. 8pm-10pm | Meet Jimi | Boat Quay
15/9:
1. 11am | Buy lunch for boss
2. 5pm-7pm | Project meeting
edit 14/9 Project meeting to 15/9 7pm
JIMI
TODAY 14/9:
3. 8pm-10pm | Meet Jimi | Boat Quay
15/9:
1. 11am | Buy lunch for boss
2. 5pm-7pm | Project meeting
3. 5pm-7pm | Project meeting
Task successfully edited:
15 Aug 2012 7pm Project meeting

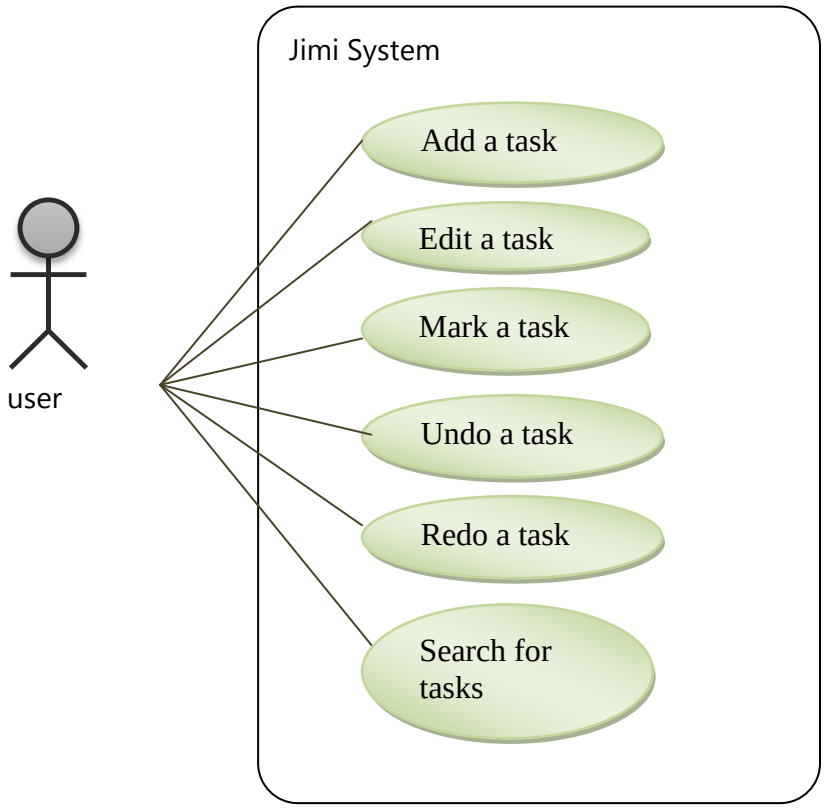
Developer Guide

Overview

- The development team uses Eclipse as our integrated development environment (IDE). Java is used as the primary programming language, for the development of this application. Swing is used for the development of the user interface (UI). We use TortoiseHg, which integrates into the Windows Shell for the Mercurial Revision Control System (RCS) to track changes.
- Our project can be found on Code Google at the following link: https://code.google.com/p/cs2103aug12-f12-2j/.

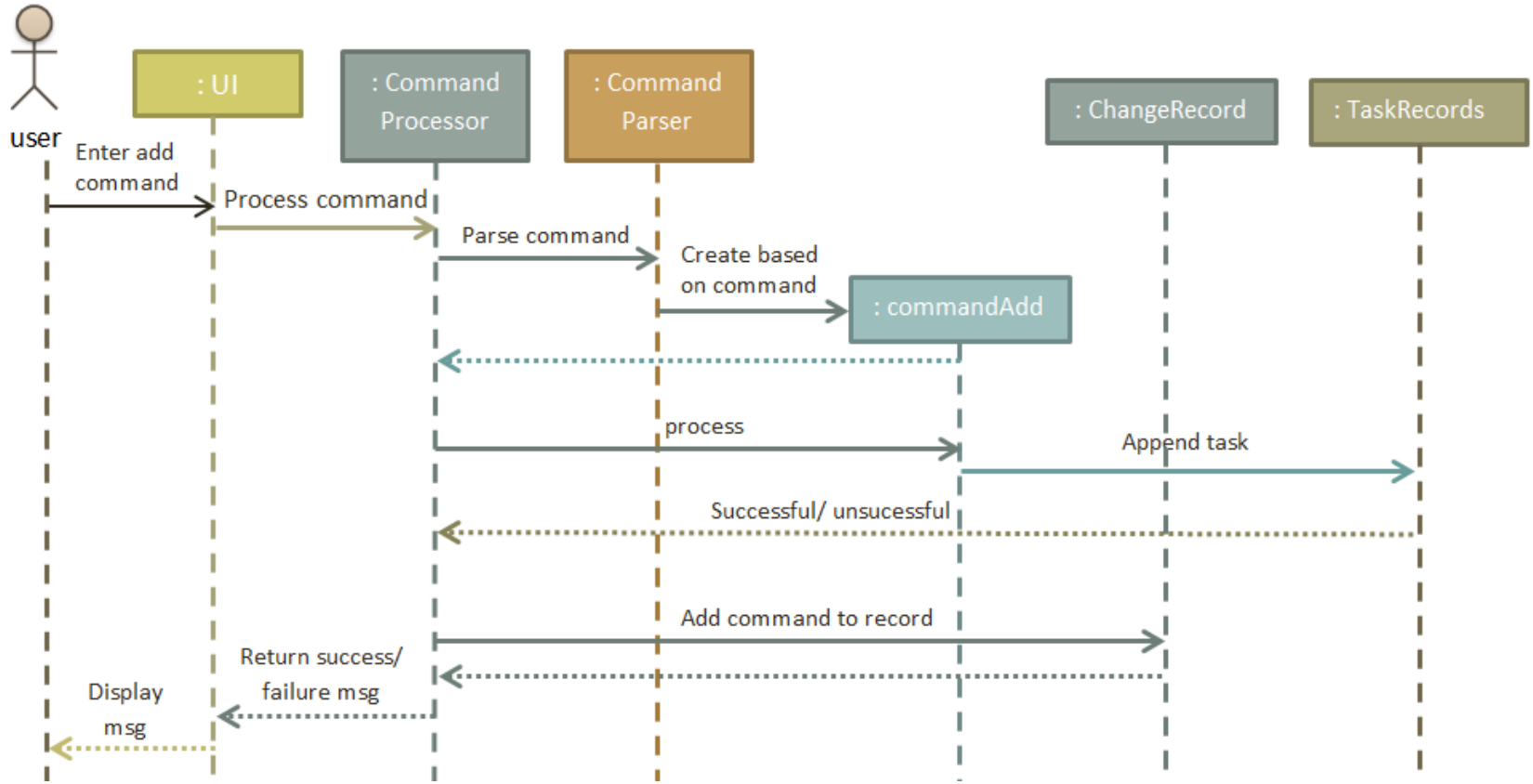
Use case diagram

The program allows the user to issue textual commands to fulfill a certain task. Please refer to the use case diagram below to see what the user can do with the system.



User Command Flow

What happens when the user enters a command (in this example, an add command is issued):



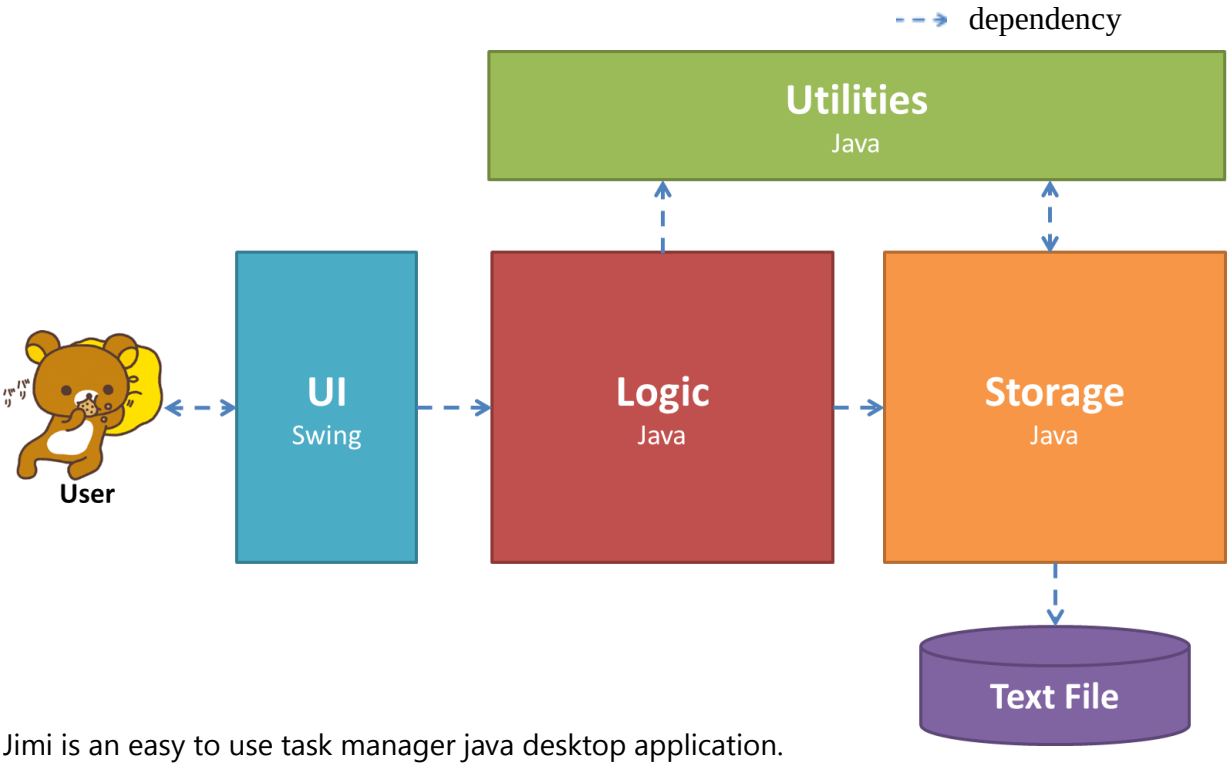
1) The user issues a command using the textbox in the UI.
2) The UI passes the command to the CommandProcessor as a String object.
3) CommandProcessor uses the CommandParser to parse the String as a command. The CommandParser parses the command entered as a String and finds out the command type (in this case, the command type is add). Based on the command type, the CommandParser parses the String and extracts Task information from it. It creates a new Command (in this case CommandAdd) from the task information and command type.
4) The CommandProcessor then tells the CommandAdd to process the command.
5) CommandAdd uses TaskRecords to append the task to the text file storage.
6) TaskRecords returns whether the task has been successfully appended to the text file storage.
7) The CommandProcessor adds this command to the ChangeRecord. This is so that a user can undo/redo previous operations.
8) CommandProcessor returns a success or failure message based on what is returned.
9) The UI displays the success or failure message to the user.

Notes:

- The flow is more or less the same for each operation the user does.
- When the search command is issued, the command processor does as extra step of getting the current list of tasks from the task records to display to the user after processing the search command.
- For undo and redo commands, an appropriate command (command that is to be undone/redone) is taken out from the change record and then processed as per normal.

Architecture

Overview of main components:



Jimi is an easy to use task manager java desktop application.

Main components:

- UI: The UI that is seen by users is a single java swing class (GUI.java) with only a few components (such as a textbox for users to enter commands).
- Logic: The logic of the program is implemented using ordinary java classes.
- Storage: A text file is used to store tasks and we use the java objects Scanner and FileWriter to read from and write to the file. A single java class is used to handle reading and writing from the file (TaskRecords.java)
- Utilities: The utilities component contains common utility classes that are shared by components in the system.

Tools and technologies

Eclipse

- Learning resources:
    - Setting up project and debugging: Getting started [Slides]
    - Keyboard shortcuts: Eclipse tips and tricks [Article]

Hg Tortoise and Mercurial

- Learning resources:
    - Basics of Hg: Hg Init by Joel Spolsky [Tutorial]
    - Basics of Mercurial: Mercurial tutorial [Tutorial]
    - More on Mercurial: Mercurial: The Definitive Guide by Bryan O'Sullivan
    - [Tutorial]

GoogleCode

- Learning resources:
    - Googlecode and mercurial: Google code and mercurial tutorial for eclipse users [Tutorial]
    - Googlecode and mercurial: 4 Priceless Tips in 4 Minutes for Google Code, Mercurial and Eclipse [Video]

External libraries (Credits)

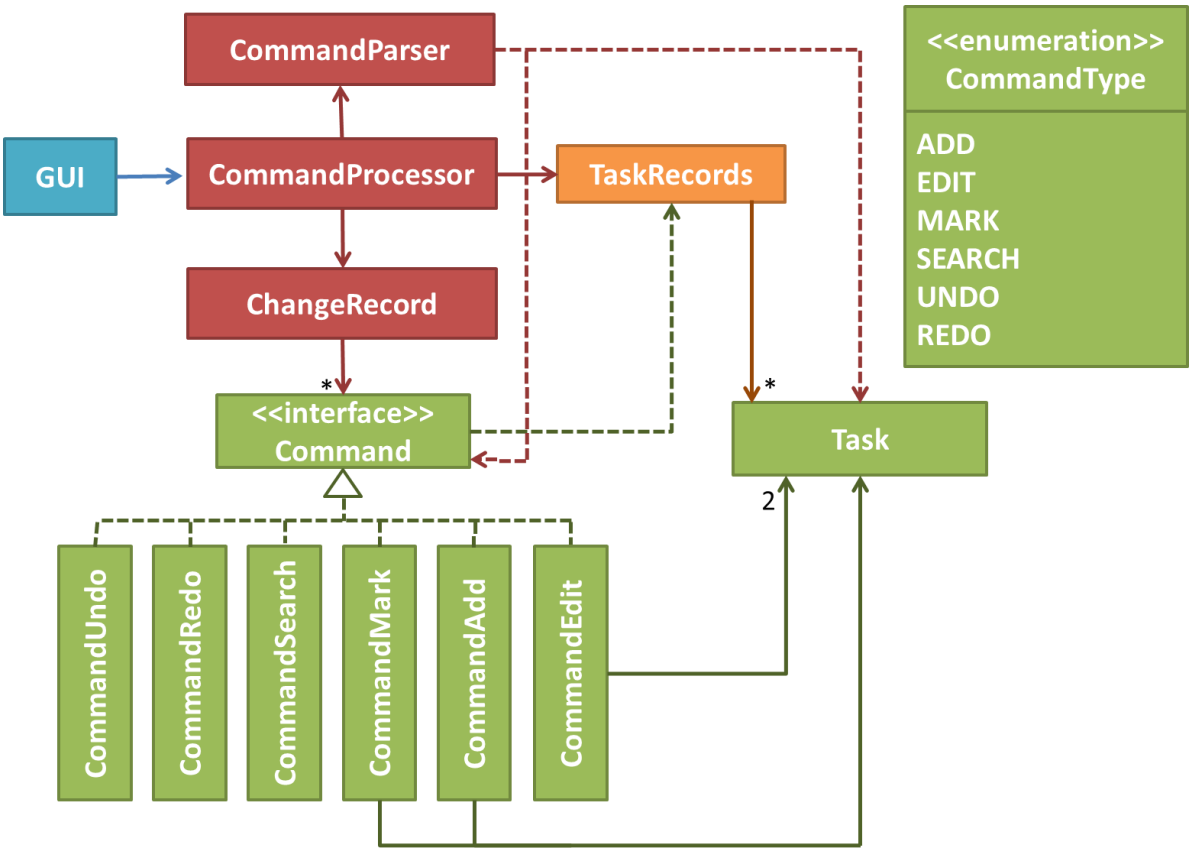
Our team uses the following third-party resources for development:

- Jodatime (Free)
  Website: http://joda-time.sourceforge.net/
  To replace Java date and time classes as Java JDK contains bugs. In addition, it provides more functionality such as the DateTimeFormatter which is widely used in our project.

- Natty (Free)
  Website: http://natty.joestelmach.com/
  Natty uses natural language recognition and translation techniques to generate dates with optional parse and syntax information. This is to provide high flexibility in command format, almost as if the program understands natural language.

Class Diagram

The diagram below is a simple class diagram showing how classes in the system interact.



Note: Colour of the class represents the part of the architecture it is a part of (see architecture).

Overview of classes:

- GUI.java (Package: ui): Interface that the user sees. It interacts directly with the user.
- CommandProcessor.java (Package: commandLogic): This is the main logic of the program. It processes user commands and decides what messages to show the user. Messages that are to be shown to the user are passed to the GUI. It maintains an instance of CommandParser, ChangeRecord and TaskRecords. This is so these instances are common and constant throughout the running of the program.
- ChangeRecord.java (Package: commandLogic): An instance of this is maintained by CommandProcessor. It keeps track of user commands issued and can tell the CommandProcessor which command to undo/redo when the undo/redo command is issued.
- CommandParser.java (Package: commandLogic): An instance of this is maintained by CommandProcessor. It parses commands that are passed in as Strings to appropriate Command objects (CommandAdd/CommandEdit/CommandMark etc).

- TaskRecords.java (Package: storage): An instance of this is maintained by CommandProcessor. This reads from and writes to the text file storage directly. It maintains a list of tasks representing the tasks that are in the text file storage and also a list of tasks representing the current list of tasks being displayed to the user.
- Command.java (Package: utilities): interface for CommandAdd, CommandEdit, CommandMark, CommandSearch, CommandUndo and CommandRedo.
- CommandAdd.java/CommandMark.java (Package: utilities): represents the add/mark command respectively. Maintains a task associated with the given command (task to be added/task to be deleted). These commands call methods in TaskRecords to perform their intended operations.
- CommandEdit.java (Package: utilities): represents the edit command. Maintains two tasks associated with the given command (task to be replaced and new task). This command calls methods in TaskRecords to perform its intended operation.
- CommandUndo.java/CommandRedo.java: represents the undo/redo/ commands respectively.
- CommandSearch.java (Package: utilities): represents the search command. This command calls methods in TaskRecords to perform its intended operation.
- Task.java (Package: utilities): Data structure for task object. Contains task information such as task name, start time, end time and whether or not the task is important.

Classes in more detail

Constructors/Methods that are self-explanatory will not be explained in the Constructor/Method Summary.

CommandProcessor

CommandProcessor
-commandParser: CommandParser -changeRecord: ChangeRecord -taskRecords: TaskRecords
+processCommand(command: String):String -processAdd(command: Command): String -processEdit(command: Command): String -processMark(command: Command): String -processSearch(command: Command): String -processUndo(): String -processRedo(): String

Constructor Summary:

CommandProcessor()

Initialises a new instance of CommandParser, ChangeRecord and TaskRecords

Method Summary:

processCommand(command: String):String

Parses the command using the command parser and processes it. It returns a String message that is to be displayed to the user. The new command is also added to the change record.

processAdd/processEdit/processMark/processSearch(command: Command): String

These process the add/edit/mark/search command respectively. It returns a String message that is to be displayed to the user.

processUndo/processRedo(): String

Gets the command to undo/redo from the change record and processes it. It returns a String message that is to be displayed to the user.

CommandParser

CommandParser
-dictionary: HashMap<String, CommandType>
-initialiseDictionary(): void +parseCommand(command: String): Command -removeFirstWord(inputString: String): String -removeExtraWhiteSpaces(inputString: String): String -removeTrailingWhiteSpace(inputString: String) -removeLeadingWhiteSpace(inputString: String) -parseAdd(command: String): CommandAdd -parseEdit(command: String): CommandEdit -parseMark(command: String): CommandMark -parseSearch(command: String): CommandSearch -getCommandType(command: String): CommandType

Notes:

- HashMap contains all keywords for the different command types

Constructor Summary:

CommandProcessor()

Initializes the dictionary by calling the initaliseDictionary() method

Method Summary:

initialiseDictionary(): void

Loads all keywords into the dictionary. The keywords help to identify command types.

ChangeRecord

ChangeRecord
-toRedoStack: Stack<Command> -toUndostack: Stack<Command>
+add(newCommand: Command): void +undo(): Command +redo(): Command

Notes:

- When a new command is added to the Change Record, the toRedoStack is cleared.
- Only reversible commands are added to the stack.

Constructor Summary:

ChangeRecord()

Initialises a new instance of toRedoStack and toUndoStack.

Method Summary:

add(newCommand: Command): void

Push the new command (if it is reversible) issued by the user into the undo stack. The redo stack is cleared.

undo()/redo(): Command

Command is popped from undo stack/redo stack and returned to caller to be processed.

TaskRecords

TaskRecords
-currentListOfTasks: Task[] -allTaskRecords: TreeSet<Task> -myFile: File
-initialiseCurrentListOfTasks(): void -initialiseAllTaskRecords(): void - convertStringToDate(stringDate: String): DateTime +getCurrentListOfTasks(): Task[] +getTaskByIndex(index: int): Task +getTaskByName(taskName: String): Task +appendTask(taskToBeAdded: Task): boolean -rewriteFile(): void +deleteTask(taskToBeDeleted: Task): boolean +clearAllTasks(): boolean +replaceTask(taskToBeReplaced: Task, newTask: Task): Boolean +setCurrentListOfTasks(query: String): void +setCurrentListOfTasks(fromDate: DateTime, toDate: DateTime): void +setCurrentListOfTasks(fromDate: DateTime): void +setCurrentListOfTasks(query: String, fromDate: DateTime, toDate: DateTime): void +setCurrentListOfTasks(query: String, fromDate: DateTime): void -findMatchesFromSetOfTasks(setOfTasks: Set<Task>, query: String): Task[]

Note: We use a TreeSet to maintain the list of all task records so that it is fast to search, add, edit and delete. You can see the java API here:
http://docs.oracle.com/javase/6/docs/api/java/util/TreeSet.html

- rewriteFile() rewrites all content from the file to show all content that is in the TreeSet (allTaskRecords). This is so changes in the TreeSet are reflected in the file.
- setCurrentListOfTasks sets the current list of tasks to contain tasks that fulfill the parameters input. This is usually used to facilitate the search command.

Constructor Summary:

TaskRecords()

Initalises the file object and all task records and the current list of tasks.

Method Summary:

initialiseCurrentListOfTasks(): void

Sets the current list of tasks to the set of tasks between the current time and 24 hours later.

initialiseAllTaskRecords(): void

Reads all task records from the file into a TreeSet.

getTaskByIndex(index: int): Task

Returns the task from the specified index in the current list of tasks.

getTaskByName(taskName: String): Task

Returns the task with the specified name. The whole task records is searched and the first matching name is returned.

rewriteFile(): void

rewrites all content from the file to show all content that is in the TreeSet (allTaskRecords). This is so changes in the TreeSet are reflected in the file.

setCurrentListOfTasks(all variants)

Sets the current list of tasks to contain tasks that fulfill the parameters input. This is usually used to facilitate the search command.

findMatchesFromSetOfTasks(setOfTasks: Set<Task>, query: String): Task[]

Looks through the specified set of tasks and returns a list of tasks that match the query specified.

Command

Command
+processCommand(taskRecords: TaskRecords): String +reverseCommand(): Command +isReversible(): Boolean +getCommandType(): CommandType

Notes:

- reverseCommand() returns null if the command is not reversible.
- Only the commands add, edit and mark are reversible.
- When processed a command returns an appropriate String message that is supposed to be shown to the user.
- If the add, edit or mark command cannot be properly processed, it will be set to not reversible.

CommandAdd

CommandAdd
-taskToBeAdded: Task
+processCommand(taskRecords: TaskRecords): String +reverseCommand(): Command +isReversible(): Boolean +getCommandType(): CommandType

CommandEdit

CommandEdit
-taskToBeReplaced: Task -newTask: Task
+processCommand(taskRecords: TaskRecords): String +reverseCommand(): Command +isReversible(): Boolean +getCommandType(): CommandType

CommandMark

CommandMark
-taskToBeDeleted: Task
+processCommand(taskRecords: TaskRecords): String +reverseCommand(): Command +isReversible(): Boolean +getCommandType(): CommandType

CommandRedo

CommandRedo
+processCommand(taskRecords: TaskRecords): String +reverseCommand(): Command +isReversible(): Boolean +getCommandType(): CommandType

Note: processCommand() here does not do anything.

CommandUndo

CommandUndo
+processCommand(taskRecords: TaskRecords): String +reverseCommand(): Command +isReversible(): Boolean +getCommandType(): CommandType

Note: processCommand() here does not do anything.

CommandSearch

CommandSearch
-query: String -fromDate: DateTime -toDate: DateTime
+processCommand(taskRecords: TaskRecords): String +reverseCommand(): Command +isReversible(): Boolean +getCommandType(): CommandType

Task

Task
-taskName: String -startTime: DateTime -endTime: DateTime -isImportant: boolean
+toString(): String +isMatch(query: String): Boolean +getTaskName(): String +setTaskName(taskName: String): void +getStartTime(): DateTime +getEndTime(): DateTime +setStartTime(startTime: DateTime): void +setEndTime(endTime: DateTime): void +isImportant(): Boolean +setImportant(isImportant: Boolean): void +compareTo(otherTask: Task): int

Note: if the start time is after the end time, swap their values.

Constructor Summary:

Task(startTime: DateTime)

This constructor is usually used to create dummy tasks for searching.

Method Summary:

compareTo(otherTask: Task): int

Tasks are compared by their start dates followed by their names. A negative value is returned if the other task has a later date or a name that is lexicographically greater. 0 is returned if the other task has the same start date and name. A positive value is returned otherwise.

Testing

Doing unit testing is highly recommended in our project. You can use the JUnit test cases in the testing package to carry tests for various components. Create your own if you like. Always test the project before committing it to the parent repository.

System Specification

Users: No specific types of users. Anyone can use the program.

Focus: Provide natural language recognizing capabilities to user typed commands.

Task Information

A task should contain the following information:

- Task name
- Start Time
- End Time
- Importance

Features

Program first startup

- Current list of tasks will contain tasks that date from the current time to 24 hours later are shown.

Displaying current list of tasks

- List of tasks are displayed in a neat and readable format [Pending, still experimenting]
- Tasks that are important should be displayed in red or bold [Pending]

Parsing a command

- Parsing a command is case-insensitive.
- Commands allow recognition of natural language to a large extend (Flexicommands). [Pending]
- A command can be of any length as long as it is readable by the parser.
- A command that is read should be parsed into an appropriate Command object. It should be able to extract task information as well:
    - Task name, Start Time, End Time, Importance [Pending]

Refreshing the current list of tasks

- Every time a command is issued and processed the current list of tasks refreshes itself to reflect changes [Pending]

Adding a task

- A start time must be specified. If none is specified by the user, use the current time.
- An end time need not be specified.
- If the importance is not specified, the task's importance should be set to not important.
- If the start time is somehow entered such that it is chronologically after the end time, swap the two.
- Displays the message:
    - "<*Task*> was added" if adding the task to the records was successful

- "Task could not be added" if adding the task to the records was not successful

Marking a task

- A task can be deleted by specifying its index in the current list of tasks.
- A task can be deleted by specifying its name. [Pending]
  - If multiple tasks have the same name, delete the one that is chronologically earliest.
- Displays the message:
  - "<*Task*> was deleted" if deleting the task from the records was successful
  - "Task could not be deleted" if deleting the task from the records was not successful

Editing a task

- A task can be edited by specifying its index in the current list of tasks followed by the details that the user wants to change. Details the user can change: task name, start time, end time [Pending allow users to edit end time without having to specify start time] and importance [pending].
- A task can be edited by specifying its name followed by the details that the user wants to change. [Pending]
  - If multiple tasks have the same name, edit the one that is chronologically earliest.
- Displays the message:
  - "Replaced with <*Task*>" (Where <Task> is the updated task) if editing the task was successful
  - "Task could not be edited" if editing the task was not successful

Searching for tasks

- Searching for tasks should be case-insensitive.
- Provide searching for tasks by task name and start time or a combination. A range can be put for the start time (from date, to date). This allows searching for tasks that occur during a certain time period. Combinations:
  - Task name
  - From date
  - Task name and From date
  - Task name, from date and to date
  - From date and to date

Undoing a previous command/Redoing a previous command

- Whenever a command is issued, it should be recorded so that it can be undone later.
- Only reversible commands should be recorded (i.e. add, edit and mark). If the command issued failed when processed, do not record it as well.
- We implemented this function using two stacks an undo stack which contains commands to undo and a redo stack which contains commands to redo.

- Whenever a new command is added, clear the redo stack.
- If there are no commands to undo/redo, display the message:
    - "There are no commands to undo/redo"
- If the undo/redo was successful, display the message that is associated with that command that was undone/redone (see adding a task, editing a task, marking a task).