Jimi – Task Manager Program

Han En Chou	Mah kai xin Amanda	Erin Teo Yi Ling
Tester	Lead GUI Developer, Documenter	Team Leader, Program Designer, 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 "list of upcoming tasks" which is a list of all of the tasks (in ascending order by time) that you have to do from today onwards. Important tasks are highlighted in red text. You can then proceed to giving the program commands to efficiently manage your tasks.

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 "/?" 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>
    - add <task name> <deadline>
    - add <task name> <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 7 Aug 4pm - 5pm
- insert email mom next Friday
- add skating practice wed 16:00

The add command recognizes different date and time formats. If the time is not specified then the program will assign the task the time 12am. Dates should be entered in little-endian (i.e day, month, year)

- 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 gold text when displayed.
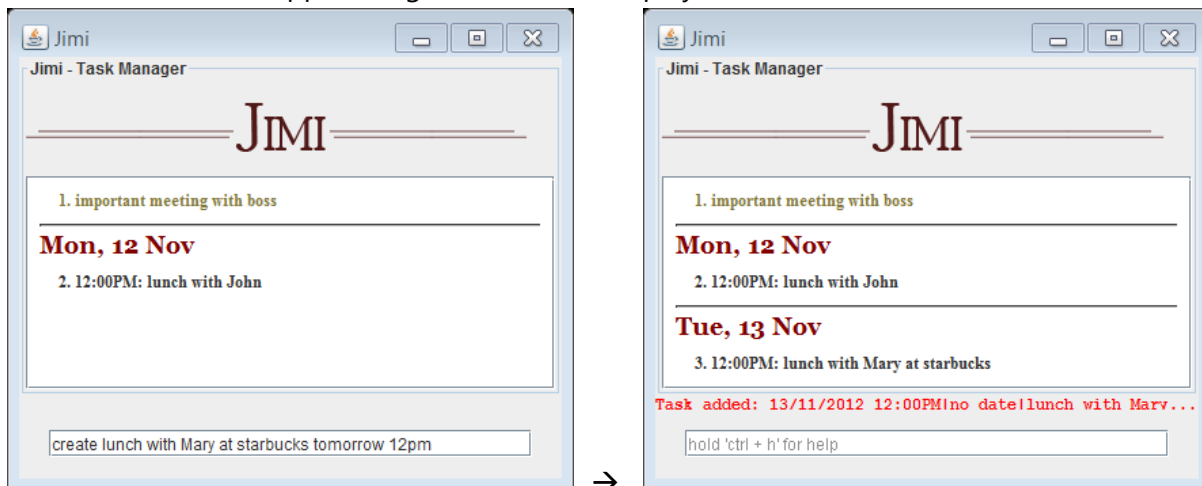
Searching for a task/event

- 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:
 - You 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 (start date inclusive and end date exclusive). 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>"
  - You can search for all tasks or upcoming tasks with the "all" or "upcoming" keyword.

Examples would include:

- search project
- search before 7 Aug
- find after today
- show tomorrow
- search project 7 Oct to 9 Oct
- search 7 Aug to 9 Aug
- search all
- search upcoming

After search command:

Jimi searches for dates really quickly! Program will display search results as a list. The list will be sorted in chronological order and tasks that are marked as important will be in gold text.
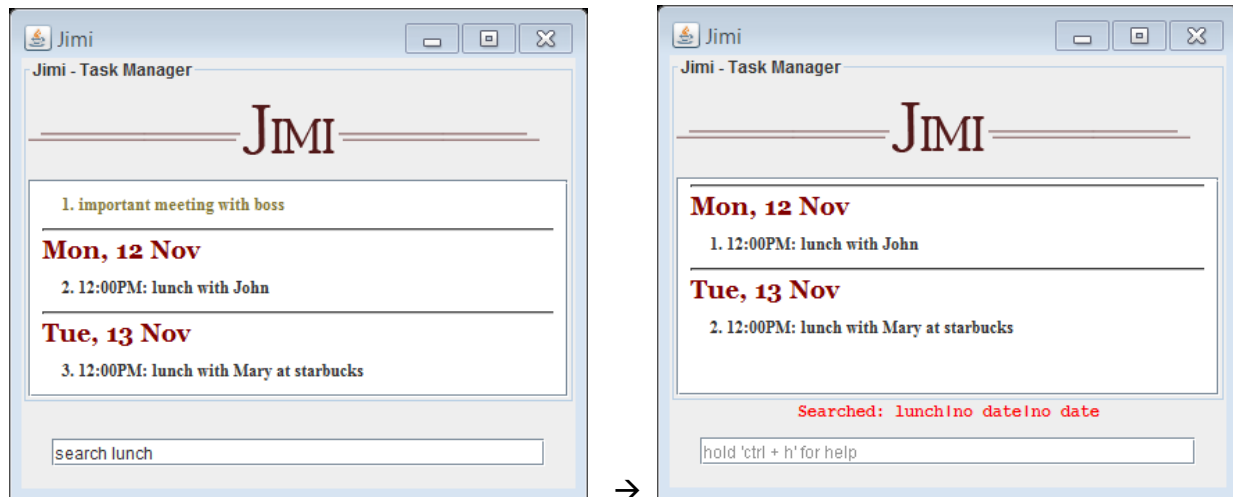
Undoing/Redoing a user command

- 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.
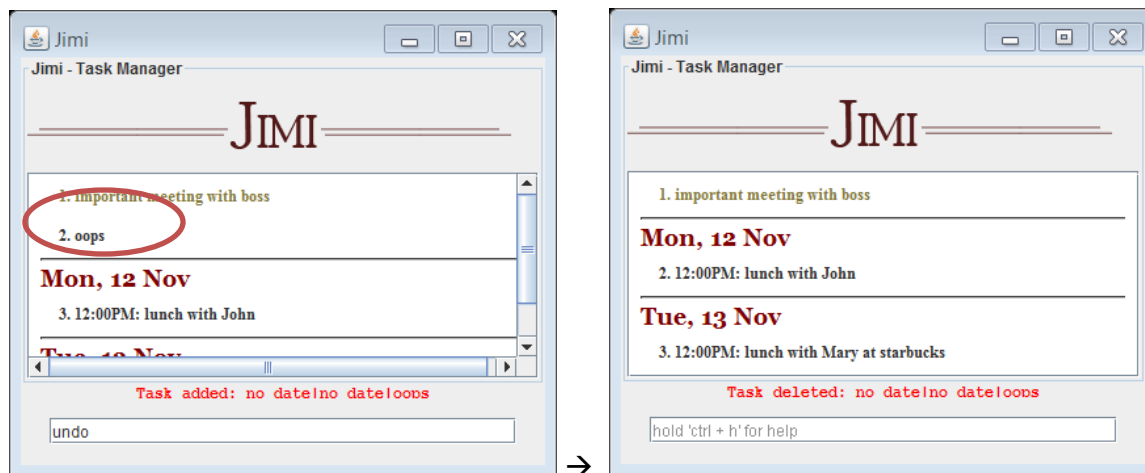
Marking a task/event

- 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 index>
    - mark all

The task index comes from the currently displayed list of tasks. Keywords like "clear", "delete" can be used in place of "mark". "mark all" marks all the tasks that are currently displayed. Examples would include:

- mark all
- mark 2

After mark command:

The program will not display this task subsequently.
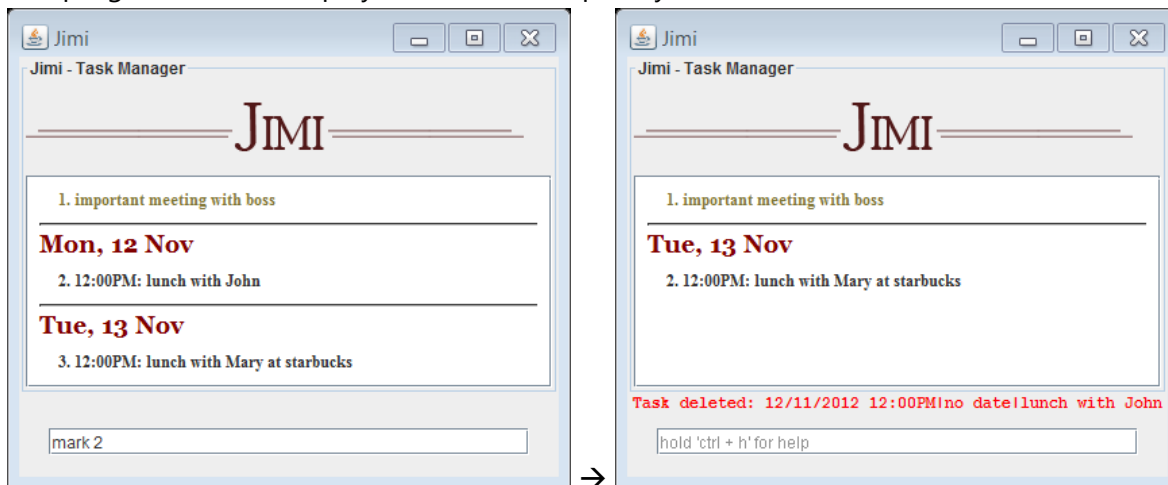
Editing a task/event

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

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

Examples would include:

  - edit 1 big project
  - edit 2 adventure time 8pm
  - edit 3 16/1/2012 1pm to 2pm
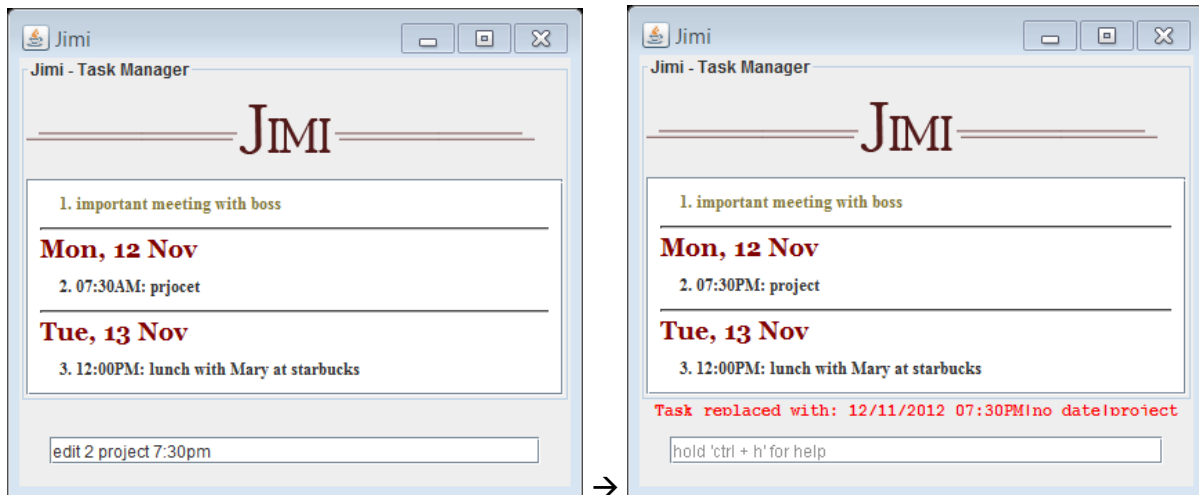
Developer Guide

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.
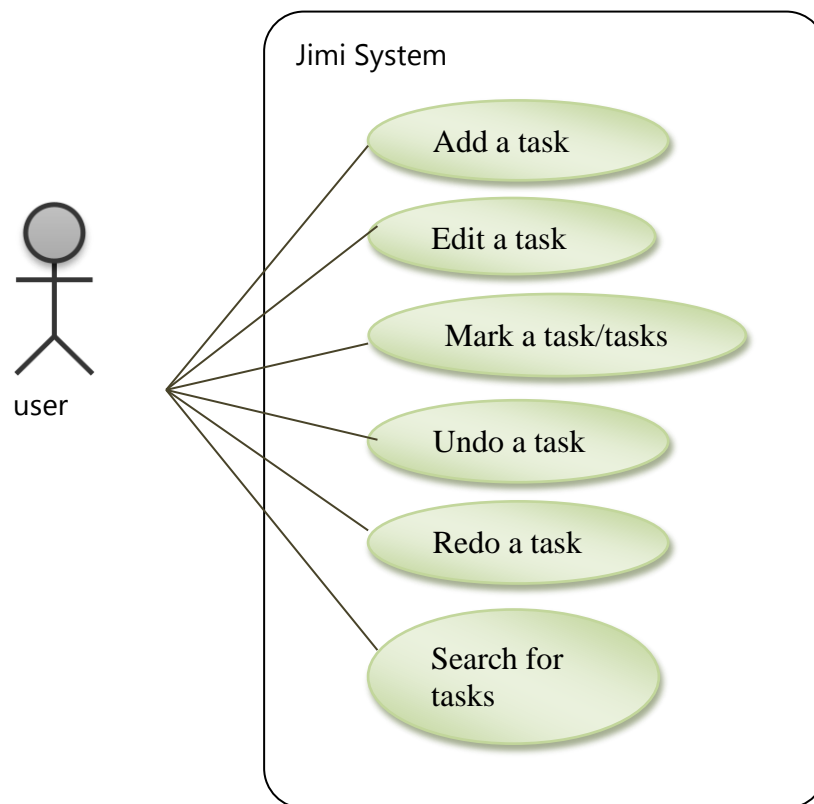
User Command Flow

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.
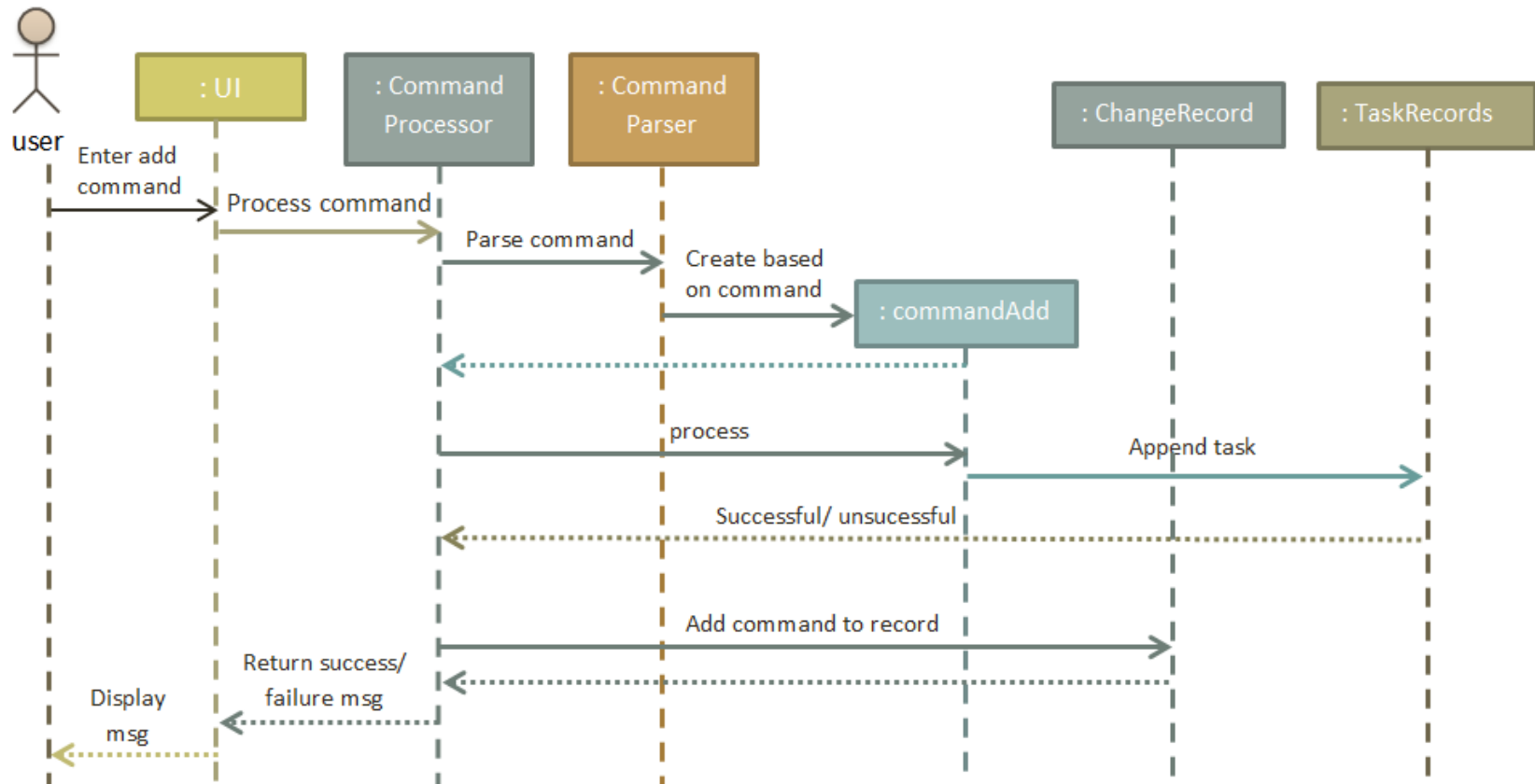7) The CommandProcessor adds this command to the ChangeRecord. This is so that a user can undo/redo previous operations.
8) CommandProcessor returns a success or failure message based on what is returned.
9) The UI displays the success or failure message to the user.

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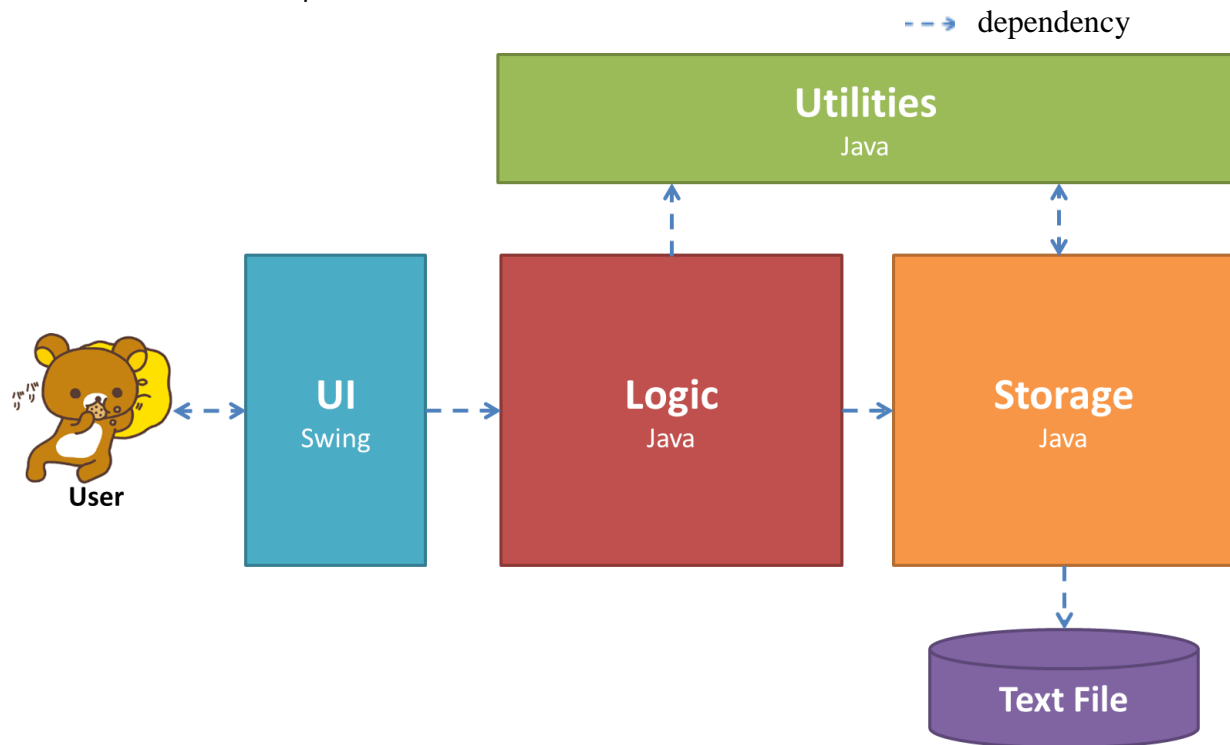
Jimi is an easy to use task manager java desktop application.

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.
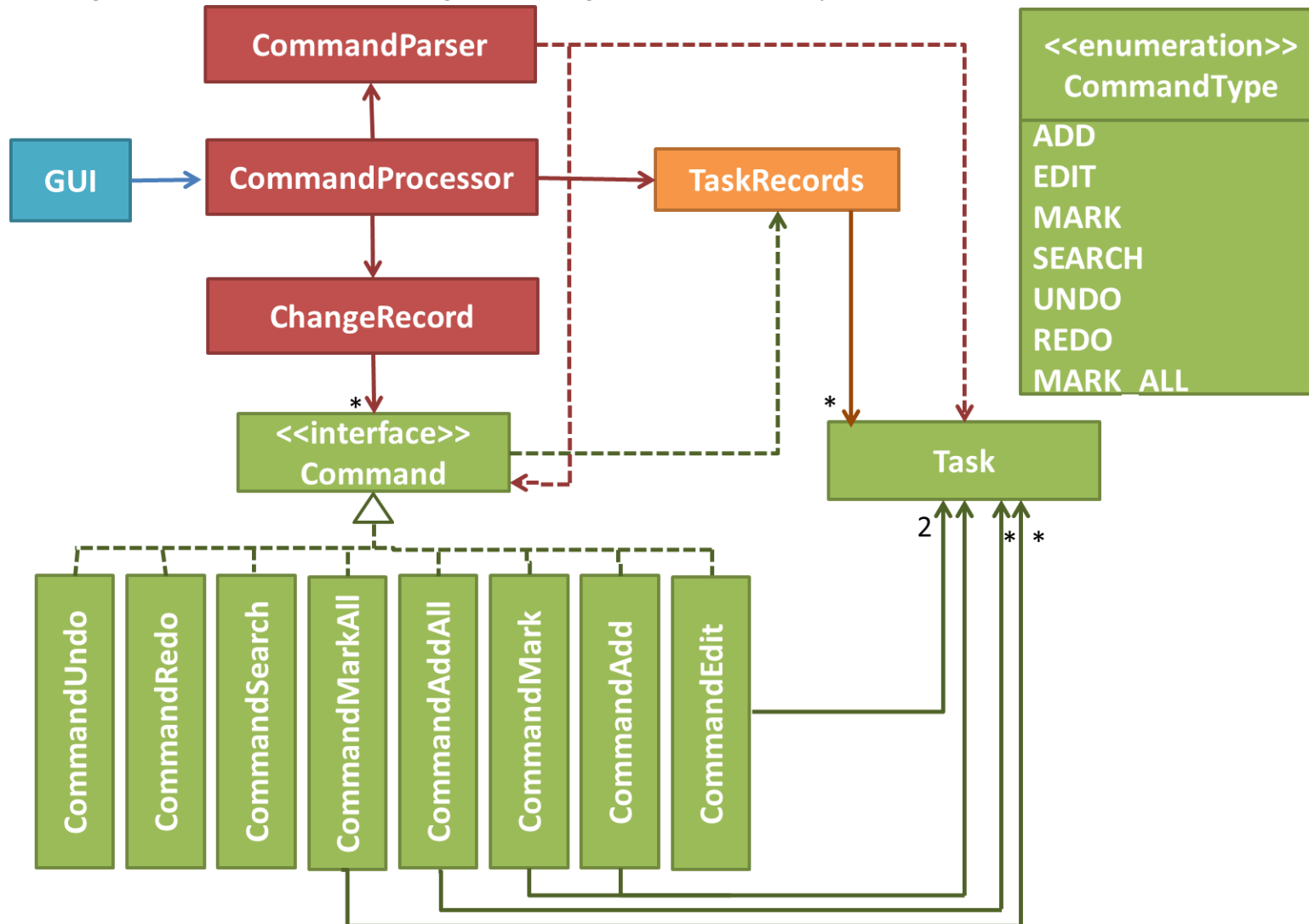
Note: Colour of the class represents the part of the architecture it is a part of (see architecture). Overview of main classes (helper classes not shown, see class details for more):

- 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, CommandMarkAll, Command AddAll, 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.
- CommandMarkAll.java/CommandAddAll.java (Package: utilities): represents the mark all and add all command. Add all is not a real user command; it facilitates reversing of the mark all command (so that it can be undone). Maintains list of tasks associated with the given command. 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.

16

Classes in more detail:

Main classes:

CommandProcessor

CommandProcessor
-commandParser: CommandParser
-changeRecord: ChangeRecord
-taskRecords: TaskRecords
-latestSearch: Command
+processCommand(command: String):String
+getCurrentListModelOfTasks():DefaultListModel<String>
+getPreviouslyIssued(): String
+getLaterIssued(): String

Note: latestSearch is to facilitate refreshing of currently displayed list of tasks.

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.

getCurrentListModelOfTasks():DefaultListModel<String>:

  Returns a ListModel containing tasks formatted as Strings. Tasks with different dates are
  separated.

getPreviouslyIssued(): String

  Gets previously issued command from the change record.

getLaterIssued(): String

  Gets later issued command from the change record.

CommandParser

CommandParser
-commandTypeKeywordsDict: HashMap<String, CommandType>
-specialKeywordsDict: HashMap<String, CommandType>
-relativeTimeKeywordsDict: HashMap<String, RelativeType>
-allDateVariantsDict: HashSet<String>
-disallowedStartCharsDict: HashSet<Character>
-commandToParse: String
+parseCommand(command: String): Command

Notes:

- HashMaps contains all keywords for the different command types and keywords. HashSets are mostly used to check for unparsable words.
- Java file also contains a enum for RelativeType.

Constructor Summary:

CommandProcessor()

Initializes the dictionary by calling the initaliseDictionary() method

Method Summary:

parseCommand(command: String): Command

Parses the command according to keywords and creates appropriate Command object.

ChangeRecord

ChangeRecord
-toRedoStack: Stack<Command>
-toUndostack: Stack<Command>
-prevCommandsStack: Stack <String>
-laterCommandsStack: Stack <String>
+add(newCommand: Command): void
+add(newCommand: String): void
+undo(): Command
+redo(): Command
+getLaterCommand(): String
+getPrevCommand(): Stirng

Notes:

- Two stacks are used to record which commands to undo/redo and previously/later issued commands.
- Only reversible commands are added to the undo/redo 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.

add(newCommand: String): void

    Adds new command issued into the previous commands stack. Later commands stack is cleared.

undo()/redo(): Command

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

getLaterCommand()/getPrevCommand(): String

    Returns top of the later/previous commands stack. If no later/previous commands were issued, returns an empty string.

TaskRecords

TaskRecords
-currentListOfTasks: Task[]
-allTaskRecords: TreeSet<Task>
-myFile: File
+getInstance(): TaskRecords
+getInstance(fileName: String): TaskRecords
+getCurrentListOfTasks(): Task[]
+getTaskByIndex(index: int): Task
+appendTask(taskToBeAdded: Task): boolean
+deleteTask(taskToBeDeleted: Task): Boolean
+addAll(tasksToBeAdded: Task[]): void
+removeAll(tasksToBeAdded: Task[]): void
+clearAllTasks(): boolean
+replaceTask(taskToBeReplaced: Task, newTask: Task): Boolean
+setCurrentListOfTasks(): void
+setCurrentListOfTasks(query: String): void
+setCurrentListOfTasks(query: String, fromDate: DateTime, toDate: DateTime): void
+setCurrentListOfTasks(query: String, fromDate: DateTime): void

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

- When a task is deleted or edited, program will rewrite 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:

getTaskByIndex(index: int): Task

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

getCurrentListOfTasks(): Task[]

     Returns the current list of tasks.

appendTask(taskToBeAdded: Task): Boolean

     Appends the task specified to the text file.

deleteTask(taskToBeDeleted: Task): Boolean

20

Removes the specified task from the text file.

addAll(tasksToBeAdded: Task[]): void

Adds all tasks in array to the text file.

removeAll(tasksToBeAdded: Task[]): void

Deletes all tasks in array from the text file.

clearAllTasks(): Boolean

Clears all tasks in the task records. Method is used for testing.

replaceTask(taskToBeReplaced: Task, newTask: Task): Boolean

Replaces the task specified with the second task specified in the text 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.

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, mark all 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
-isReversible: boolean
+processCommand(taskRecords: TaskRecords): String
+reverseCommand(): Command
+isReversible(): boolean
+getCommandType(): CommandType

Note: The default for isReversible is true but changes to false if processCommand fails.

CommandEdit

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

Note: The default for isReversible is true but changes to false if processCommand fails.

CommandMark

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

Note: The default for isReversible is true but changes to false if processCommand fails.

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

22

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

Method Summary:

getTimesAsString(): String

　　　Returns a task's time in hh:mma format. If 2 times then format is "hh:mma to hh:mma".

toString(): String

　　　Returns Task formatted as String. Format is "startTime|endTime|taskName".

isMatch(query: String): Boolean

　　　True if task name matches query. False otherwise.

compareTo(otherTask: Task): int

23

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. A task with a null date is smaller than a task with a proper date.

DateComparator

This is a helper classes that contains methods that compare dates.

DateComparator
+isNow(dateTimeToCheck: DateTime): boolean
+isSameTimeOfDayAsNow(dateTimeToCheck: DateTime): boolean
+compareNullDatesLast(firstDate: DateTime, secondDate: DateTime): int
+isSameDay(firstDate: DateTime, secondDate: DateTime): Boolean

Method Summary:

isNow(dateTimeToCheck: DateTime): Boolean

Returns true if the datetime specified is the current date and time. Returns false otherwise. Has some allowance for a small difference in time.

isSameTimeOfDayAsNow(dateTimeToCheck: DateTime): boolean

Returns true if the time specified is the same time of the day as the current time. Returns false otherwise. Has some allowance for a small difference in time.

compareNullDatesLast(firstDate: DateTime, secondDate: DateTime): int

A null date is always bigger than any other date. Returns positive number if firstDate is greater than secondDate, returns negative number if firstDate is smaller than secondDate and returns zero if they are the same.

isSameDay(firstDate: DateTime, secondDate: DateTime): boolean

Returns true if firstDate and secondDate fall on the same day. Returns false otherwise.

User Testing

Jimi System was sent to a few students who are Keyboard-Savvy and are asked to answer the *Test Feedback Form*.

Table of Feedback Results*

	1	2	3	4	5	6	7	8	Avg
Overall User Interface	4	3	5	4	2	5	3	3	3.625
Flexibility of command Parser	5	5	5	5	2	5	5	4	4.5

* Rate the Jimi from 1 (not at all well) to 5 (extremely well).

User Interface: The UI is described as simple, small and uncluttered. However, there is room to grow especially in terms of alignment and user-friendliness.

Flexibility of command Parser: The flexibility of Jimi is described as simply easy to use as one can just naturally enter an input without worrying about syntax of input.

Conclusion:
Gaining insight from the feedback of results, we were able to improve portions of our program, though not extensively. For instance, the user interface gained a facelift for ease of use: shortcut keys such as 'ctrl-h' is included to further enhance our system's user-friendliness. Some users also mentioned the alignment in Jimi is a little messy, which we have also fixed in v0.5.

As for the Flexibility of command Parser, many users are impressed by how the system is able to capture task and dates from random syntaxes that the users used. However there were a couple of bugs, for example, 'fried chicken friday' was parsed as 'ed' as 'fri' in 'fried' was parsed as 'Friday'. These bugs in fixed for v0.5.

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.
Learning resource: http://www.vogella.com/articles/JUnit/article.html

Reporting Bugs and Issues

Please report bugs at the issue tracker: http://code.google.com/p/cs2103aug12-f12-2j/issues/list

Patterns And Principles

We love patterns! These patterns are used in Jimi:
Abstraction Occurrence pattern

- This pattern is implemented in the TaskRecords class where a list of all tasks is maintained. Duplicate tasks are ignored because adding to a TreeSet ignores duplicates.

Singleton

- Singleton class is implements in TaskRecords class as well. We only want one instance of TaskRecords running because we don't want 2 instances of TaskRecords to change the text file at the same time.

Façade

- CommandProcessor acts as the façade class. GUI only communicates with the CommandProcessor.

Command pattern

- Command pattern is implemented (see class diagrams). ChangeRecord treats all commands as command objects and does not know the specific type of command. Command class is an interface.

MVC

- See architecture.
- View: GUI
- Controller: Logic/GUI (GUI detects UI events, Logic takes follow up action)
- Model: Storage

Principles:

- Single responsibility principle: A class should be highly cohesive.
- Open-closed principle: New Command objects can easily be added to extend the functionality of Jimi

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

When displaying a list of tasks, always sort in chronological order of the start time followed by lexicographical order of the task name.

Features

Program first startup

- Current list of tasks will contain tasks with no date and tasks with date from the start of today onwards.

Displaying current list of tasks

- List of tasks are displayed in a neat and readable format.

Parsing a command

- Parsing a command is case-insensitive.
- Commands allow recognition of natural language to a large extent (Flexicommands).
- 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

Refreshing the current list of tasks

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

Adding a task

- Start and end times need not be specified.
- If the start time is somehow entered such that it is chronologically after the end time, throw StartTimeAfterEndTime exception.
- Displays the message:
    - "Task added: <Task>" 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.
- List of currently displayed tasks can be deleted if the keyword "all" is specified.

- Displays the message:
  - "Task Deleted: <Task>" 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.
- Displays the message:
  - "Task 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.
- When keywords "before" or "after" specified, show tasks that come before/after the date specified.
- When the keyword "all" is specified, show all tasks.
- When the keyword "upcoming" is specified, show all upcoming tasks (floating tasks and today's tasks onwards)
- 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).

Change log

<u>V0.1</u>

- Basic GUI
- Basic Command Parsing

<u>V0.2</u>

- Rough implementation of Flexicommands
- Some tests written
- Support for floating tasks

<u>V0.3</u>

- Refined Command Parsing
- Refined GUI
- Some exceptions

<u>V0.4</u>

- Practiced defensive coding, exceptions, assertions and logging
- Refined Command Parsing

<u>V0.5</u>

- Bug fixes after extensive testing
- Make code not smelly
- Refined Command Parsing
- Added help menu
- More tests