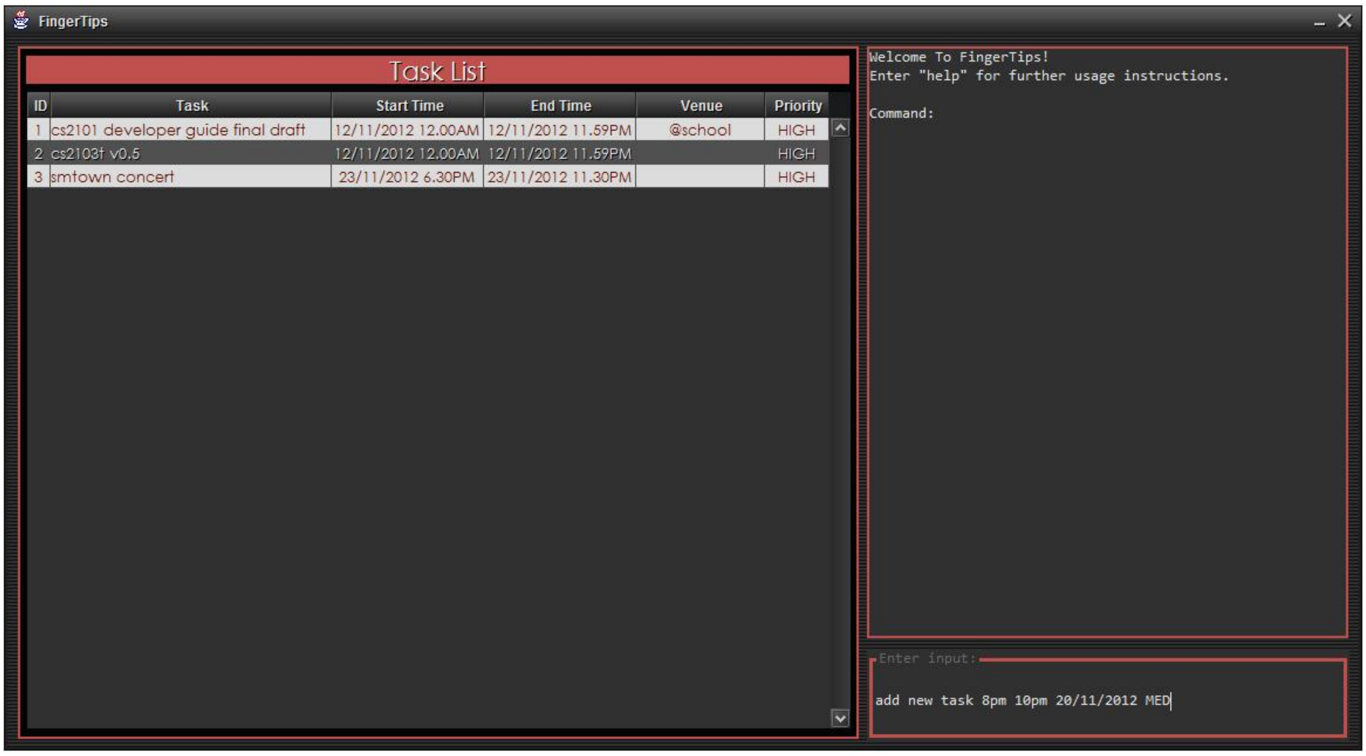
FingerTips

So Text Me Maybe!



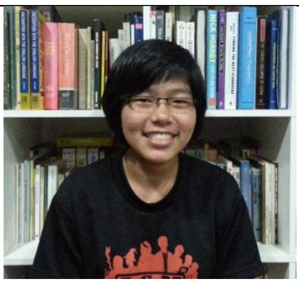
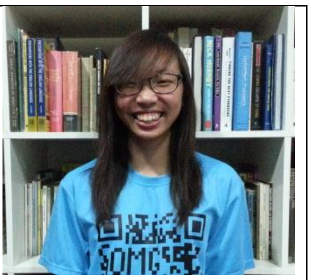
ID	Task	Start Time	End Time	Venue	Priority
1	cs2101 developer guide final draft	12/11/2012 12.00AM	12/11/2012 11.59PM	@school	HIGH
2	cs2103t v0.5	12/11/2012 12.00AM	12/11/2012 11.59PM		HIGH
3	smtown concert	23/11/2012 6.30PM	23/11/2012 11.30PM		HIGH

```
Welcome To FingerTips!
Enter "help" for further usage instructions.

Command:
```

Enter input:

```
add new task 8pm 10pm 20/11/2012 MED
```

Raymond Cheng Team lead, Coder, Product Tester	**Farhan** Coder, Documentation	 **Kathy Phua** Designer, Product Tester, Documentation	 **Koh Zhen Zhen** Designer, Product Tester, Documentation

Contents

1 Overview

What is **FingerTips**?

- **FingerTips** is a to-do task companion that accepts natural and simple but structured language commands via the keyboard.

Who is **FingerTips** designed for?

- **FingerTips** is designed specifically for users who prefer typing over using touch-motion devices like the mouse and the trackpad.
- **FingerTips** is ideal for people who are always on the go, as it works as a stand-alone application without any internet connection required.
- **FingerTips** is as easy to use as ABC with simple and short commands that even a computer dummy can pick up easily.
- Best of all, for people who are on a tight budget, it's free.

What does **FingerTips** do?

- Similar to a normal to-do list, the program helps the user organise and manage present and future tasks, stores relevant details for easy reference, through simple keyboard commands.

How do I install **FingerTips**?

- **FingerTips** does not need to be installed. As the program is designed to be portable, you just have to start the application and it will be fully functional.

How do I start **FingerTips**?

- You can open **FingerTips** by double clicking the FingerTips.jar file.

2 User Guide

2.1 Basic Features

Function	Description
Add	Create a task and add it to the database.
Edit	Add/remove information related to your current task.
Remove	Removes a task from the database.
Display	Shows the details of a task.
Undo	Undo the previous action.
Redo	Undoes the undo command.
Mark Done	Updates a task status to completed
Clear	Deletes all entries within **FingerTips**.
Help	Displays a list of functions, and how to access them.
Quit	Exits **FingerTips**

Table 1: List of functions for **FingerTips**

Bonus Features:

- **Hash Tagging**: User can add keywords to task(s) to make them more searchable.
- **Priority ranking**: High priority tasks are displayed first to alert user to these tasks.

2.2 Basic Command Formats

2.2.1 Add Task

Sample structure:

a Meeting 21/09/2012 6pm @U-Town #school HIGH

General format:

a<space>description

or **add**<space>description

start time am/pm<space>end time am/pm<space>dd/mm/yyyy

<space>@venue<space>#hash tag<space>priority tag

Priority – **HIGH** / **MED** / **LOW**

Note: There should be no space between words in each field (except for the description field).

Example: @MarinaBaySands

The **add** function fits into the natural flow of how most people would save the data in their minds. The key field for this function to create a task would be the description.

2.2.2 Remove Task

Sample structure:

r 2

General format:

r or **rmv** or **remove**<space>number

The **remove** function would be similar to how people would update a task as completed on a pen and paper list. This would however, remove the task from **FingerTips**, permanently. Fortunately, this action is reversible (see **2.2.4**).

2.2.3 Edit Task

Sample structure:

e 2

General format:

e<space>number or **edit**

Each task on the list will be numbered with an index. The index entered by the user will call the corresponding task on the list to be edited.

2.2.4 Undo Last Action

Sample structure:

undo

General format:

u or **undo**

This function reverses the last action done. The **undo** function can be called until there are no more further actions found to be undone.

2.2.5 Redo Last Action

Sample structure:

redo

General format:

rd or **redo**

This function reverses the last undo action done. The **redo** function can be called until there are no more further actions found to be redone.

2.2.6 Display/Display+

Sample structure:

d meeting

General format:

d or **display**<space >keyword/hash tag

d+ or **display+**<space >keyword/hash tag

The **display** function lets you view the task details, by searching for specific keywords, or a hash tag. If you just type display, it would show all the tasks in the active list. Calling the **display+** function displays all tasks with the specified keywords or hash tag in the active and archive lists.

2.2.7 Mark Done

Sample structure:

done 3

General format:

done or **fin** or **finish**<space>number

The **done** function lets you mark a task as completed. The task will then be moved over to the archive list.

2.2.8 Clear

Sample structure:

clear

General format:

clear or **clr**

The **clear** function erases ALL data that is currently stored in **FingerTips**. Use this with extreme caution! This action is reversible by using the undo command (see 2.2.4).

2.2.9 Help

Sample structure:

help

General format:

help or **h**

Calling up the **help** function displays all the possible command syntaxes/formats within **FingerTips**, (as shown in Table 1), as well as the syntax/format for the edit mode of a task.

2.2.10 Quit

Sample structure:

quit

General format:

quit or **q** or **exit**

To close **FingerTips**, simply type the **quit** command. All data and any changes made will be saved.

2.3 Bonus Features Format

2.3.1 Hash Tag Tagging

Sample structure:

a Meeting 21/09/2012 6pm @U-Town **#school** HIGH

General format:

a<space>description or **add**<space>description

start time am/pm<space>end time am/pm<space>dd/mm/yyyy

<space>@venue<space>**#hash tag**<space>priority tag

The hash tag tagging can be done in **add** or **edit** functions. Similar to what is done in Twitter, users can group similar tasks together, i.e. *#school* or *#meeting*. Users can add only one hash tag per task, so choose carefully!

2.3.2 Priority Tagging

Sample structure:

a Meeting 21/09/2012 6pm @U-Town #school **HIGH**

General format:

a<space>description or **add**<space>description

start time am/pm<space>end time am/pm<space>dd/mm/yyyy

<space>@venue<space>#hash tag<space>**priority tag**

Priority – **HIGH** / **MED** / **LOW**

Tasks can be grouped according to their level of **priority** – High (Very Important), Medium (Important) and Low (Not So Important). In this way, you can keep your tasks organised, and better manage your time by prioritising the tasks that have a relatively higher priority.

2.4 Important Note

When **FingerTips** is first launched, it will create 5 files, namely *activelist*, *activetextfile*, *archivelist*, *archivetextfile*, and *runlog*. Please do not move, rename or delete these files as they are critical to the proper operation of **FingerTips**.

3 Developer Guide

3.1 Introduction

This developer guide helps the reader to familiarise the various aspects of **FingerTips** by walking through the development process of **FingerTips**. In addition to providing background information on the capabilities and core functions of **FingerTips**, this guide provides examples for interacting with the application by implementing specific features.

This guide is intended for programmers who want to develop **FingerTips**. *Java* is the main language used for **FingerTips** and this guide also assumes that the reader knows how to program in *Java*.

FingerTips is developed using Java Standard Edition 7 and it is cross-platform. The repository for **FingerTips** can be found at https://code.google.com/p/cs2103aug12-w09-3j.

3.2 Architecture Overview

The following *dependency diagram* captures the interactions between multiple components of **FingerTips**:

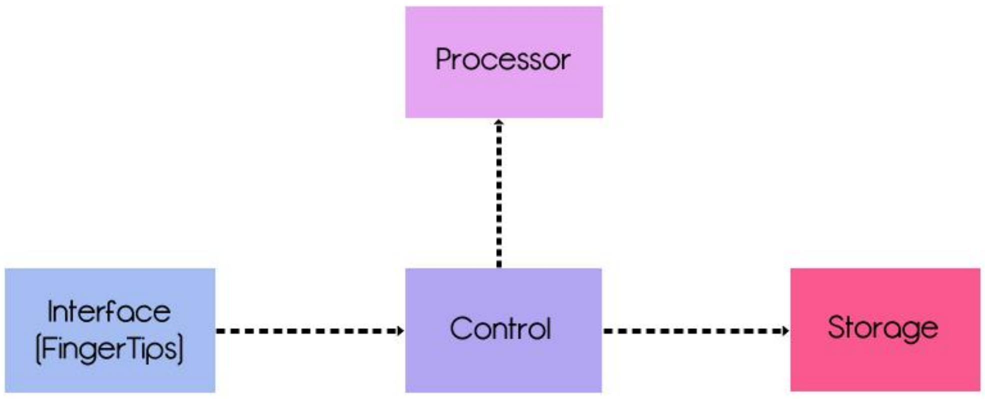


Figure 1: Dependency, Architectural Diagram for **FingerTips**

FingerTips makes use of the *n-tier* architectural style. Beginning with the **Interface** portion (GUI), text commands are typed into the input text field. Upon typing of a command in the input text field, the input will be parsed into the **Control**, which will be routed into the **Processor**. The **Processor** will tokenise the command into two parts: the *command string*, and the *argument string*. It will then invoke the relevant command handler (e.g. *add* function), based on the command string portion.

Upon invoking the respective command handler (e.g. *add* function), it will return a **COMMAND_TYPE** object to **Control**. If the command is a valid operation, the command and argument strings will be passed to the **Storage** component, where the necessary edits will be made. There are two main storage components, the *active list* (where current tasks are kept) and the *archive list* (where completed tasks are kept. This will be returned back to the user Interface as a success/failure message via **Control**, with the relevant data printed, if any.

3.3 Working With FingerTips

FingerTips is a hassle-free to-do list that helps the user to manage and store tasks for easy referencing in the future, through the use of simple keyboard commands. The User Interaction with **FingerTips** will be explained in the sub-sections below. All the **COMMAND_TYPE** Objects are called from the **Processor** component.

3.3.1 Determine Command Type Function

determineCommandType will return a **COMMAND_TYPE** object in the **Processor** that has a commandString attached to it. The current list of **COMMAND_TYPES** is as follows:

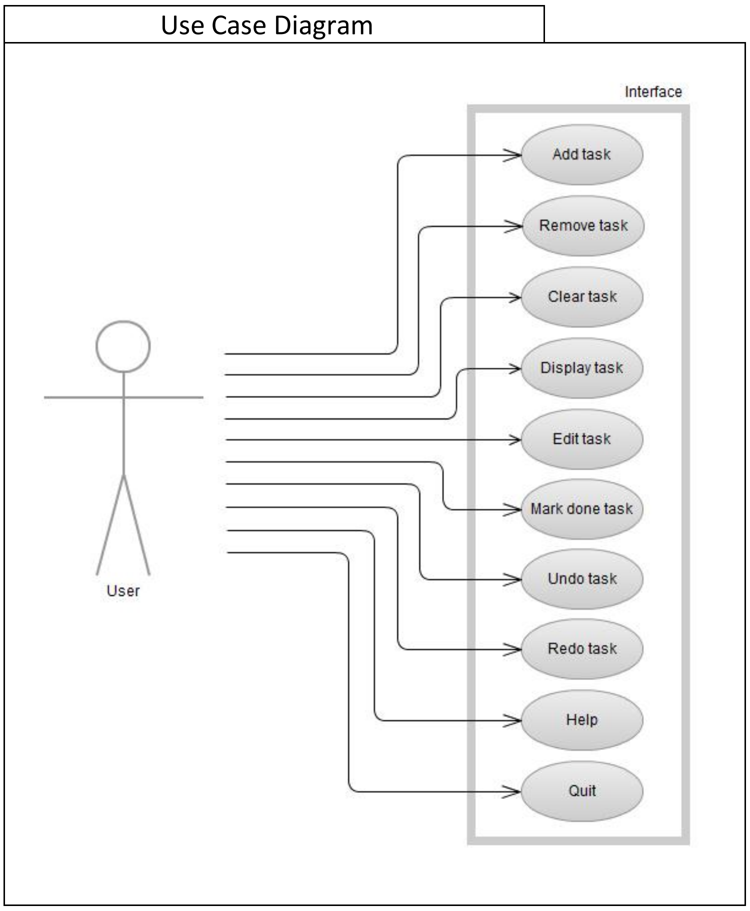


Figure 2: Use Case Diagram

The use case diagram above shows the interaction between the user and **FingerTips** for the main functionalities of the program.

3.3.2 Add Function

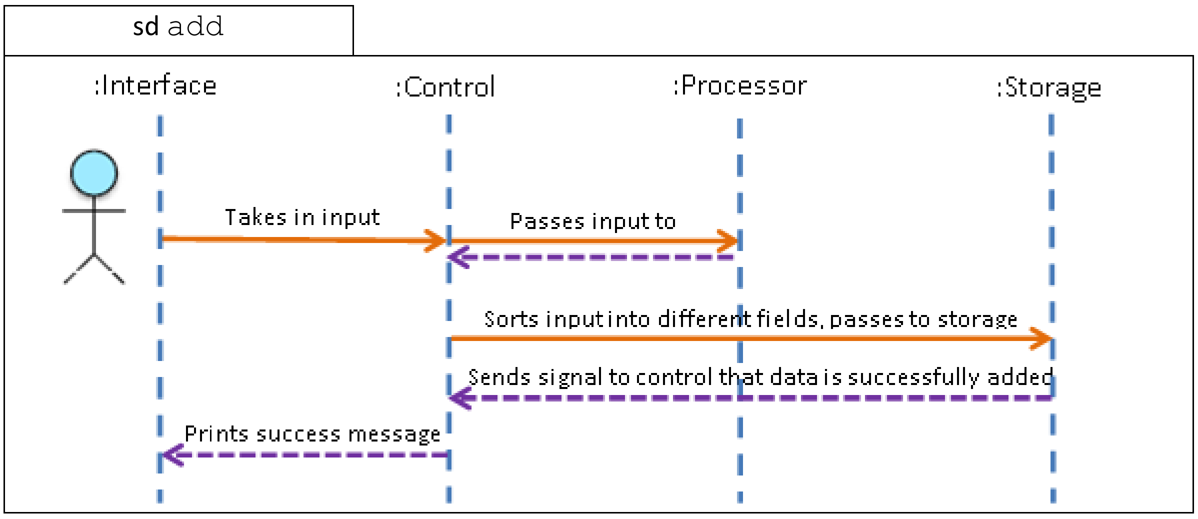

sd add
:Interface
:Control
:Processor
:Storage
Takes in input
Passes input to
Sorts input into different fields, passes to storage
Sends signal to control that data is successfully added
Prints success message

Figure 3: Sequence Diagram for add function

The `add` function takes in the input from the user and the **Processor** helps to sort the input into different fields before passing it to **Storage** via **Control**. When the input is stored, **Storage** will then return a signal to **Control** that the data is successfully added. The **Control** will then output the success message.

3.3.3 Display Function

The `display` function calls up the relevant task from the active list of the **Storage** component. The **Processor** first determines the data to be displayed, either based on **(1)** search string, or **(2)** hash tag. The data will then be passed back to **Control**, and printed out on the main display panel.

The `display+` function calls up the relevant tasks from the active and archive lists of the **Storage** component. The **Processor** first determines the tasks to be displayed, either based on **(1)** search string, or **(2)** hash tag. The data will then be passed back to **Control**, and printed out on the main display panel.

The `display` function also resets the data to be displayed if the `display+` function is called previously, which is the default view.

3.3.4 Remove Function

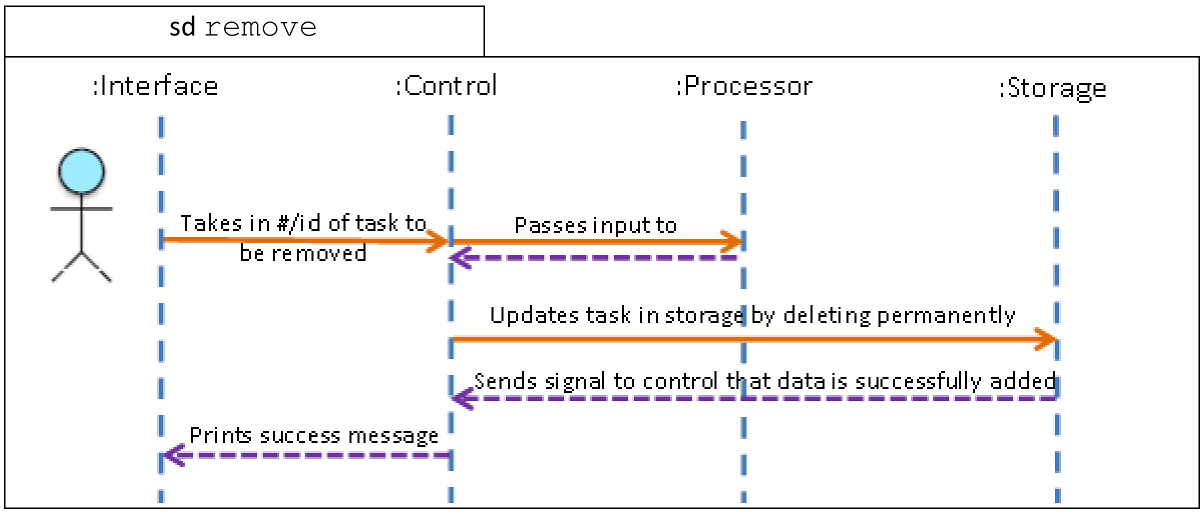

sd remove
:Interface
:Control
:Processor
:Storage
Takes in #/id of task to be removed
Passes input to
Updates task in storage by deleting permanently
Sends signal to control that data is successfully added
Prints success message

Figure 4: Sequence Diagram for remove function

The `remove` function takes in the index of the task to be removed, based on the index of tasks shown in the **Interface**. **FingerTips** will then retrieve the relevant data from **Storage**, returning the task details.

The input is passed to **Storage** via **Control** which will update the file. The **Storage** then sends a signal back to **Control** that the task is successfully removed and the **Control** displays the success message to the user.

3.3.5 Edit Function

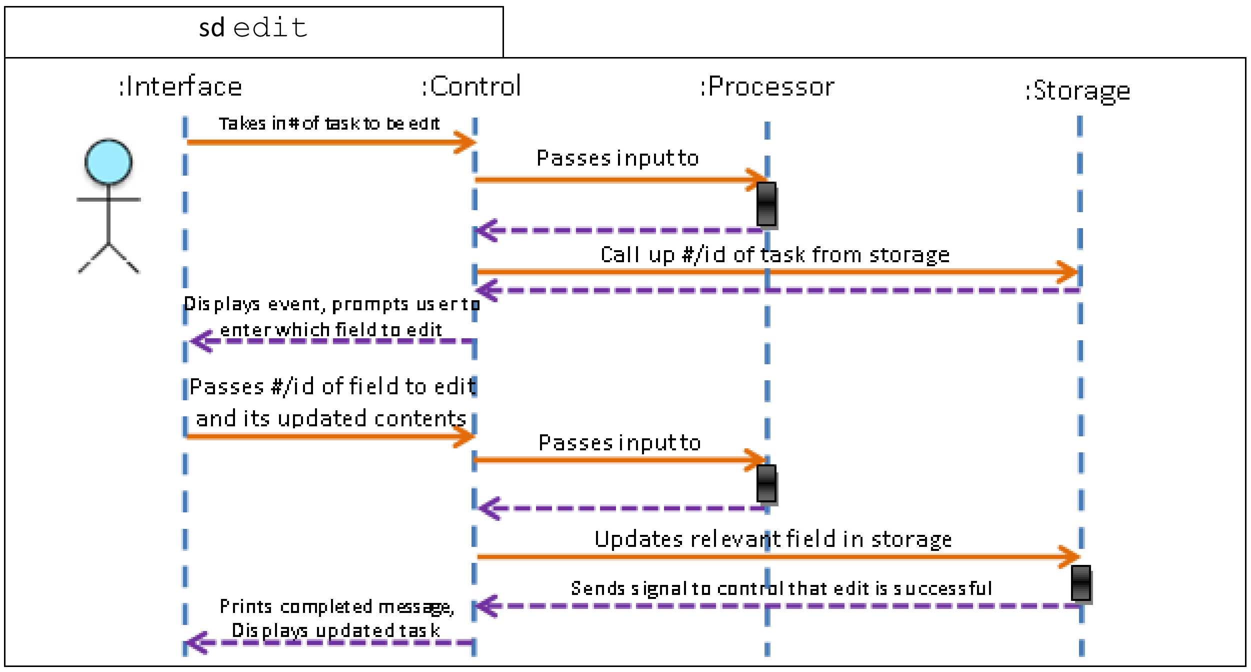

sd edit
:Interface
:Control
:Processor
:Storage
Takes in # of task to be edit
Passes input to
Call up #/id of task from storage
Displays event, prompts user to enter which field to edit
Passes #/id of field to edit and its updated contents
Passes input to
Updates relevant field in storage
Sends signal to control that edit is successful
Prints completed message, Displays updated task

Figure 5: Sequence Diagram for edit function

The `edit` function takes in the index of the task to be edited, based on the index of tasks shown in the **Interface**. **FingerTips** will then retrieve the data from **Storage**, returning the task details.

The task is displayed and the user is prompted to enter the field to be edited. The updated data is sent to **Storage**. **Control** sends a signal that the task is edited successfully and a success message is displayed.

3.3.6 Undo Last Action Function

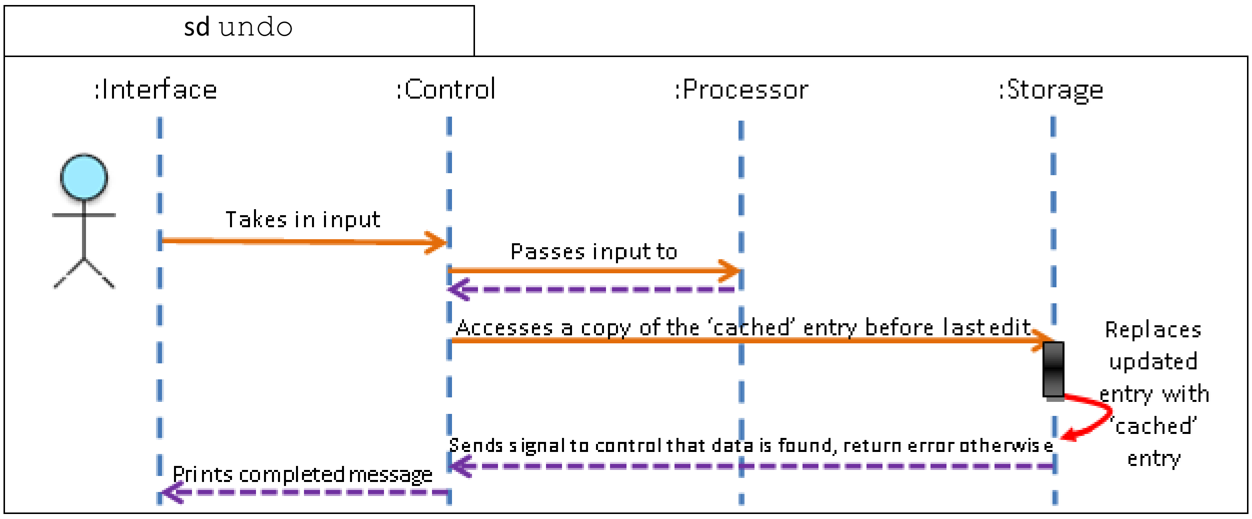

sd undo
:Interface
:Control
:Processor
:Storage
Takes in input
Passes input to
Accesses a copy of the 'cached' entry before last edit
Replaces updated entry with 'cached' entry
Sends signal to control that data is found, return error otherwise
Prints completed message

Figure 6: Sequence Diagram for undo function

The `undo` function can take place in the following four scenarios: **(1)** adding a task **(2)** editing a task details **(3)** removing a task and **(4)** using the clear function (see 3.3.9). In each case, the following end results will appear respectively: **(1)** task is removed **(2)** task details are back to its prior state before editing **(3)** task is re-added **(4)** all tasks are restored in the active and archive lists.

The **Control** then returns a message to the user via the **Interface**, to indicate if the **undo** function has been successful, and if the data has been restored (or removed) to its prior state.

3.3.7 Redo Last Action Function

The redo function reverses the last undo action committed (for more details on the undo function, see 3.3.6).

3.3.8 Mark Done Function

The `done` function takes in the index of the task that is to be mark as done by the user. The index is then passed to **Storage** where the data of the item marked as done is moved to the archive list.

3.3.9 Clear Function

The `clear` function regenerates new files for the active list. The `clear+` function regenerates new files for the archive list.

3.4 Important APIs

CMD.java

```
public CMD(Processor.COMMAND_TYPE command, Object data)
```
Receives a command from COMMAND_TYPE list from Processor and returns a Data String to be displayed as output.

```
public Processor.COMMAND_TYPE getCommandType()
```
Receives a cmd String input from Interface and returns a commandType Object from Processor.

```
public void setCommandType(Processor.COMMAND_TYPE commandType)
```
Receives commandType Object from Processor.

Control.java

```
public CMD performAction(String userInput)
```
Receives a cmd String input from Interface and returns a CMD Object from Processor.

```
public CMD undo(CMD command)
```
Receives a cmd String input from Interface, returns a CMD Object to invoke the necessary change in Storage.

```
public String[] processEditMode(String userInput)
```
Receives a cmd String input from Interface and returns a Data String from Processor.

Processor.java

```
public CMD translateToCMD(String userInput)
```
Receives userInput String from Control and returns userCMD Object and String output.

Storage.java

```
private void loadFromFile(File source, Vector<Entry> destination)
```
Receives source File and returns a hash table of entries.

```
public void saveToFile(File objDest, Vector<Entry> list, File txtDest)
```
Opens storage Files and updates the user converted inputs.

```
public Vector<Entry> display(boolean arc)
```
Clears the ArrayList used to store Entries to be displayed (displayEntries) and add activeEntries and/or archiveEntries to the ArrayList of DisplayEntries, depending on the boolean arc that is passed in.

UI.java

```
public void actionPerformed(ActionEvent e)
```
Implements a Listener Class and passes data input from text field in a string format to Control.

```
private TableCellRenderer setTableDesign()
```
Sets up the main task display panel, formats text, row and column according to the type of data to be displayed.

3.5 Code Samples

```
public static Control getInstance(){
       if (control == null) {
              control = new Control();
       }
return control;
}
```

In **FingerTips**, we have adhered to the *singleton pattern principle*. This involves restricting to a single instance of Control, Processor, and Storage, through checking on start to see if any other instances exist.

We have also followed the principle of *single responsibility principle*. This involves ensuring that each class has a single responsibility, and that responsibility should be encapsulated within that class. *Coupling* is also reduced by limiting the interaction between variables and objects of different classes. This prevents major changes when only a small code of the section is changed. In this way, the *Law of Demeter* is upheld.

3.6 Testing

In developing or expanding the current functions of **FingerTips** (hereby referred to in this section as *Software Under Test*, or *SUT*), the developer is advised to update the test cases as well. This is to prevent regressions, and ensure that most, if not all, the functions/classes are working well. We strive to have at least 50% code coverage testing in our Unit Testing and Automated Test Cases, and they are explained in the below two sub-sections.

We have made use of the in-built *JUnit* in *Eclipse* to carry out unit testing for **FingerTips**. The test cases are as written in the test folder. Currently, we have built testing for the five main functions (*add, remove, edit, display, undo*), as well as handled error exception during file reading and user command input.

Use Case	Test Case No.	Test Case Description	Expected Results
Add	a01	Add normally	New task added
Add	a02	Add with invalid date	Returns invalid date error
Add	a03	Add with no description	Prompts user to re-enter command string again
Remove	r01	Remove normally	Existing task is removed
Remove	r02	Remove out-of-range task	Task to be removed is not found; Prompts user to re-enter command string again
Remove	r03	Remove a specific hash tag	All entries containing that specified hash tag(s) are removed
Remove	r04	Remove invalid hash tag	Entries containing hash tag is not found; prompts user to re-enter command string again
Edit	e01	Edit normally	Existing task to be edited is displayed
Edit	e02	Edit invalid numbering input	Task to be edited is not found
Display	d01	Display normally	All entries are displayed
Display	d02	Display hash tag	All entries with specified hashtag are displayed
Display	d03	Display invalid hash tag	If hash tag is not found, an error message will appear, and no entries will be displayed
Display	d04	Display both active and archive list	
Undo	u01	Undo previous action	Last action is undone
Clear	c01	Clear all existing entries	All entries are cleared

Table 3: Summary of Use-Test Cases

3.6.1 Unit Testing

Items tested under this section would cover:

- The Main Use Cases/Features of the SUT – Add, Remove, Edit, Undo, Display, Clear, Mark Done
- Valid output messages are displayed
- Commands are executed correctly by each method in the respective classes, i.e. the method(s) functions as intended
- Valid changes to the Storage component are made
- Validation undertaken by methods i.e.*FileNotFoundException* Error

```
public void testIsDate() {

           for (int i=0; i<actualOutput.length; i++) {
                actualOutput[i] = "true";
                expectedOutput[i] = "true";
           }

           input[0] = "12May2012";     // false - not a valid format
           input[1] = "12 10 12";// false - not a valid format
           input[2] = "12122012";// false - not a valid format
           input[3] = "0/10/2012";     // true
           input[4] = "1/12/12/12";// true
           input[5] = "1-13-2012";     // true
           input[6] = "14.10.12";      // true

           for (int i=0; i<3; i++) {
                expectedOutput[i] = "false";
           }

           for (int i=0; i<input.length; i++) {
                if (p.isDate(input[i]) != null) {
                     actualOutput[i] = "true";
                }
                else {
                     actualOutput[i] = "false";
                }
           }

           assertArrayEquals(expectedOutput, actualOutput);
     }
```

Code Fragment 1

This code snippet above shows the unit testing done to check and validate the dates parsed from the user input. Both correct and incorrect test cases are set. A set of *expectedOutput* results for valid and invalid date inputs is also added. The test cases are passed into the test to be run, where the *actualOutput* results are compared with the set of *expectedOutput* results using the *assertArrayEquals* method.

```
public void testDetermineCmdEditMode() {

            input[0] = "descriptions"; // invalid, returns error
            input[1] = "due";                // invalid, returns error
            input[2] = "hasheS";       // invalid, returns error
            input[3] = "nothing";      // invalid, returns error
            input[4] = "test";               // invalid, returns error
            input[5] = "endTIME";      // valid, returns endtime
            input[6] = "q";                  // valid, returns end
            input[7] = "#";                  // valid, returns hash

            for (int i=0; i<5; i++) {
                  expectedOutput[i] = "error";
            }
            expectedOutput[5] = "duetime";
            expectedOutput[6] = "end";
            expectedOutput[7] = "hash";


            for (int i=0; i<input.length; i++) {
                  answer = p.determineCmdEditMode(input[i]);
                  actualOutput[i] = answer[0];
            }

            assertArrayEquals(expectedOutput, actualOutput);
      }
```

Code Fragment 2

This code snippet shows the unit testing done to check and validate the edit mode commands passed from the user input. Both correct and incorrect test cases are set. A set of *expectedOutput* results for valid and invalid edit mode commands is also added. The test cases are passed into the test to be run, where the *actualOutput* results are compared with the set of *expectedOutput* results using the *assertArrayEquals* method.

3.6.2 Integration Testing

By using the top-down approach, the higher-level components are integrated first and tested as a group before the lower-level components are brought in for further testing. Integration testing occurs after the unit testing of input modules and before user acceptance testing. This ensures that the components are working normally as indicated from the tests.

3.6.3 Automated Testing

This involves running **FingerTips** through a wide variety of data inputs to simulate a user using the SUT.

Test Case Input	Expected Output	Actual Output
add Project meeting	Added.	Same as expected output.
a Shopping	Added.	
display	1 Project Meeting 2 Shopping	
remove 1	Removed.	
display	1 Shopping	
undo	Undo completed.	
display	1 Project Meeting 2 Shopping	
e 1	Task: Project Meeting Enter the field you wish to modify, and the new data to replace with. Type "end" to exit edit mode and "help" for further assistance.	
ddate 12/12/2012	1 Project Meeting 12/12/2012 2 Shopping	
quit	Edit mode ended.	
clear	All active tasks deleted.	
display	There is nothing to print.	
quit		

Table 4: A code snippet of Automated Testing

Note: *The data in the lighter blue areas is the expected output to be displayed in the table. The data in the darker blue areas is the expected output to be displayed in the text box above the user input field.*

3.6.4 User Acceptance Test

Since the software is designed with the Users in mind, what better way to test this software than having a sample of users doing the User Acceptance Test (UAT)? This involves conducting a field trial to verify the working integrity of the software through daily and rigorous usage. This allows us to find flaws in the SUT's design or architecture, which may not be found via the two testing methods above (see **3.6.1**, and **3.6.3**). Users will be given a set of instructions on current limitations, as well as what to look out for.

Users may then log issues via our group's **Google Code repository** as linked above.

3.6.5 Logging

FingerTips uses the Java built-in logging API in the *java.util.logging* package. Logging produces a system events log to aid us in troubleshooting problems. Logging is done though a *Logger* instance.

Logger gathers the data to be logged into a LogRecord which is then forwarded to a Handler. The Handler will then determine the next action to be taken by LogRecord. LogRecord is passed through a Filter to determine if it should be forwarded or not.

3.7 Possible Developments for FingerTips

3.7.1 Known Issues

Although **FingerTips** can handle and manage tasks competently, there are still a number of known issues found in the program:

- Updates in edit mode can only be updated one field at a time
- Redo function cannot undo marking a task as done

For detailed information, please refer to **FingerTips**' issue tracker here: https://code.google.com/p/cs2103aug12-w09-3j/issues/list

3.7.2 Future Development

At the current point of time where this project manual is written, **FingerTips** runs on a simple GUI with basic but essential commands of a to-do list. In the future, **FingerTips** can be upgraded to include a calendar view pop-up window, an add-your-own-command function for the user to personalize his/her shortcut keys. Also, a sync function can be added to allow **FingerTips** to be more portable.

4 Change Log

V0.0 (10 Sep 2012):

- Added in User Guide
    - Listed Main Functions (Add, Remove, Edit, Search, Undo)

V0.1 (16 Oct 2012):

- Updated User Guide with the three additional functions listed below
- Added in Developer Guide
    - Updated Core Functions
        - Search function updated to Display Function
        - Added in new functions: Done, Quit, Help
- First Working iteration of **FingerTips**
- Added in basic Unit Testing

V0.2 (24 Oct 2012):

- Fixed bugs and problematic functions
- Aligned text layout for display
- Added new function: Clear, Undo

V0.3 (31 Oct 2012):

- Added partial implementation of GUI
    - **FingerTips** will still run via CLI as this stage since not all functions are working in the GUI
- Added in Partial Unit Testing and Automated Testing
- Streamlined the codes for Processor, Control and Storage
- Applied several patterns and principles that help to improve **FingerTips** workflow

V0.4 (7 Nov 2012): - Release Candidate

- Bug fixes for GUI
- Refactored code from v0.3
- Added in first working version of exportable jar file

V0.5 (12 Nov 2012): - First Working Release

- Updated GUI
- Updated Test Case Samples
- Handled error exception caused by user input
- Added Product Demonstration Video