Developer Guide

Urgenda is a text-based task manager designed for users who are quick on the keyboard, or generally prefer using the keyboard over mouseclicks. It is a Java application that has a main GUI for users to interaction with mainly through typing.

This guide describes the implementation of Urgenda from a top-down approach, going from the big picture down to the small details, starting from the UI through which the user interacts with, to the Storage where the files are stored. This guide intends to allow easy assimilation of anyone who would like to contribute and add on to it.

Table of Contents

- Architecture
- UI component
    - Main class
    - MainController class
    - DisplayController class
    - TaskController class
    - TaskDetailsController class
- Logic component
    - Logic class
- Command component
- Parser component
    - Parser Class
- Storage component
    - Storage class

Architecture

Urgenda consists of 4 main components, with the interaction for the user through the UI

1. The `UI` component uses JavaFX with FXML files for displaying the UI to the user, with the controllers in java files to control the display
2. The `Logic` component is the main brainchild of Urgenda. Every other component interacts with `Logic` to provide information and details that are required for Urgenda to run smoothly

3. The `Parser` component is the parser of Urgenda, parsing natural language used by a typical user into processable variables and attributes for `Logic` to utilise for maximum effectiveness
4. The `Storage` component keeps all the Tasks, data and settings in textfiles on the user's computer, allowing Urgenda to instantly restore all the previous tasks of the user when he starts up Urgenda every time.

UI Component

The UI component is the solo component responsible for creating and maintaining Urgenda's graphical user interface. It is also the only component that directly handles any user interaction with Urgenda. User input is mainly through the command line input through the input bar at the bottom of the window. Minor click functionalities are also enabled to enhance the user experience, but do not provide any extra functions beyond that available through command line input.

JavaFX and CSS are used to setup the layout and design for the graphical user interface.

Main Class

The `Main` class acts as the intermediary between the controllers for all UI components with the back-end Logic component. This abstracts all command execution flow from the UI class.

MainController Class

The `MainController` class handles all user interaction with Urgenda, for both command line input as well as click inputs. Several event handlers set in this class, including `commandLineListener`, will call the UI class's handleCommandLine method as pass the command line String as the argument. Other event handlers include accelerated keyboard shortcuts for undo, redo, and help functions. This class also displays feedback from any user interaction to the user.

DisplayController Class

The `DisplayController` class handles the display area where relevant tasks are displayed to the user. The display area is set up using the `setDisplay` method, where the task panels are created and added to the display according to the task list argument passed. This class also handles the setup of the design for tasks with different task types and tasks to show details for.

TaskController Class

The `TaskController` class handles the set up of the panel for one single task. The task index as

ordered in the display area, description, prioritisation, as well as relevant dates and times, are shown on the panel. This class also applies the task design style for the task, and formats the dates and times with respect to the current date and time.

TaskDetailsController Class

The `TaskDetailsController` class is a extended class from `TaskController`. This class is invokes to create the task panel when the relevant task is required to show more details, such as location, date created and modified, and long descriptions.

FXML Files

Each controller class will load an FXML file, which sets the layout of the UI window according to the design set by these FXML files. There are a total of 5 FXML files: Main.fxml, DisplayView.fxml, TaskView.fxml, and DetailedTaskView.fxml for the main UI window, and HelpSplash.fxml for setting the help window. FXML files can be opened and edited using SceneBuilder 2.0, which is an official visual layout tool for JavaFX applications from Oracle. More information about SceneBuilder 2.0 can be found at: a link (http://www.oracle.com/technetwork/java/javase/downloads/sb2download-2177776.html)

Logic Component

The Logic component is accessible through the `Logic` class using the facade pattern, in which it is in charge of handling the execution of user inputs from the UI component. This component only relies on the Parser component and Storage component and works independently from the UI component. Furthermore, the `Command` component is part of the Logic of Urgenda which encompasses the functionalities of the different commands given by the user.

The table below shows the classes in Logic component and their functions:

Class	Function
`Logic` (Facade)	Main handler for external calls from other components. Also has the Singleton pattern as there should always be only one `Logic` handling the processes in Urgenda.
`LogicData`	Class in Logic component that stores the Tasks temporarily when Urgenda is running. Most data manipulation and edits are done through `LogicData`. It is also responsible for generation of the current state. Has the Singleton pattern as well to ensure that all data manipulation is done on the only LogicData.

Class	Function
`LogicCommand`	Class where the Commands are being stored in the Logic component. Execution of commands as well as undo/redo of these commands will be carried out by LogicCommand.

Furthermore, the table below shows the notable API for usage of Logic:

Method	Return type and function
`executeCommand(String command, int index)`	Returns a `StateFeedback` object which consists of the execution feedback, as well as the current state of the task objects relevant to display for UI. The method will be used to process all inputs by the user.
`retrieveStartupState()`	Returns a `StateFeedback` object of the system. This method is for the initial startup of Urgenda in setting up the components as well as retrieval of previously saved tasks.
`displayHelp()`	Returns a `String` which consists of the help manual of Urgenda. This method is used for the request of Urgenda's help manual.
`getCurrentSaveDirectory()`	Returns a `String` of the current location where the data is being saved on the user's computer.

Logic Class

The `Logic` class contains the methods that handle the core functionality of Urgenda. It can be thought of as the "processor" of Urgenda. User inputs are passed to the `executeCommand(String, int)` to determine the corresponding command object based on the user input by the `Parser` component.

After knowing the type of command, `Logic`retrieves the updated state and data per launch time from `LogicData` via the `UpdateState()` method call. After which the command object will be passed to `LogicCommand` for process through the `processCommand(Command)` method call. The command will then be executed, and `LogicData` will update its relevant fields. In the case of adding a task, the task will be added task list via the `addTask(Task)` method call and the display state will be updated correspondingly. `LogicData` maintains a temporary set of data same as that displayed to the user per launch time so as to facillitate number pointing of task and reduce dependency with `Storage` component (e.g. when user inputs delete 4, `Logic` is able to determine which is task 4 without having to call `Storage`). `Storage` component will then store the data to ensure no loss of user data upon unintentional early termination of Urgenda Program. More details of the storing procedure are mentioned in the `Storage` section.

A generic example of the process flow in Logic can be seen below:

After knowing the type of command, `Logic`retrieves the updated state and data per launch

time from `LogicData` via the `UpdateSate()` method call. After which the command object will be passed to `LogicCommand` for process through the `processCommand(Command)` method call. The command will then be executed, and `LogicData` will update its relevant fields. In the case of adding a task, the task will be added to task list via the `addTask(Task)` method call and the display state will be updated correspondingly. `LogicData` maintains a temporary set of data same as that displayed to the user per launch time so as to facilitate number pointing of task and reduce dependency with `Storage` component (e.g. when user inputs delete 4, `Logic` is able to determine which is task 4 without having to call `Storage`). `Storage` component will then store the data to ensure no loss of user data upon unintentional early termination of Urgenda Program. More details of the storing procedure are mentioned in the `Storage` section.

The executeCommand(String) method will then return the appropriate feedback to its caller method. The caller method can then decide how to update the user interface.

Command Component

Here is the abstract method that is present in `Command` class.

Method	Return type and function
`execute()`	Returns a `String` which represents the feedback of the command being executed. This is the abstract method for the generic execution of Commands.

Additionally, here are the abstract methods present in the `TaskCommand` class.

Method	Return type and function
`undo()`	Returns a `String` which represents the feedback of the command being undone. This is the abstract method of the generic undo execution of each TaskCommand.
`redo()`	Returns a `String` which represents the feedback of the command being done again. This is the abstract method of the generic redo execution of each TaskCommand.

The Command component is part of the Logic processing in Urgenda, where the specific commands are being executed by the program. It mainly consists the specific execution instructions of the individual command types.

`Command` is an abstract class that uses the Command Pattern and holds the `execute()` method where the generic execution of `Command.execute()` can be used. Classes that extends from it will have their own implementation of the `execute()` method. `TaskCommand` is another abstract class which extends `Command` and is for commands that deal with manipulation of Task objects. `TaskCommand` has two abstract functions which are `Undo()` and `Redo()` which are also implemented separately by the child classes to revert the changes made by that command.

The structure of the Command component allows the flexibility of adding new command

types to Urgenda by simply extending one of the two abstract classes (`Command` and `TaskCommand`). The abstraction of the Command class allows new Commands to be added by just extending and implementing their unique `execute()` command.

Parser Component

The Parser component is accessible through the `Parser` class using the interface pattern. This component is invoked by the `Logic` component, and has the function of parsing a passed in user command string and return an appropriate Command Object. In order to do this, Parser will access different classes, each having its unique functions, as listed in the next section.

Parser Class

Class	Function
`CommandTypeParser`	In charge of parsing the passed in user command string to determine the type of expected returned Command object, as well as returning the string of arguments for further processing by other respective classes, such as `DateTimeParser` or `TaskDetailsParser`.
`Collection of CommandParsers`	Includes various classes that are in charge of generating and returned a specific type of Command object, such as `AddTask` or `DeleteTask`. The type and details of the returned Command objects depend on the private attributes String _argsString and int _index present in each of these classes, which will be set upon calling of its constructor. Each classes have its own method that invokes different functions in `PublicFunctions` to perform the correct parsing depending on the type of returned Command.
`DateTimeParser`	In charge of recognizing and parsing date and time values in the argument string. DateTimeParser relies on PrettyTimeParser, which is a external open source project that parses date and time values in natural language flexibly. The role of DateTimeParser is to make modifications to the pick-up patterns of PrettyTimeParser, as well as handling user keywords in the command. The parsing result is made directly on the variables stored in PublicVariables, and DateTimeParser returned the argument string already trimmed of date time expressions.
`TaskDetailParser`	In charge of recognizing and parsing other relevant details in the argument string, including task index, description, location and hashtags.
`PublicFunctions`	This class contains all the public functions shared between the command parser classes to perform their role.

Class	Function
`PublicVariables`	This class contains all the public variables accessible to the different command parser classes and needed to generate the appropriate type of returned Command object and correct details.

The API of parser is:

Method	Return type and function
`ParseCommand(String commandString)`	Returns a Command object with specific type and details, as parsed by parser.

Storage Component

The Storage component is accessible through the `Storage` class using the facade pattern, where it handles and directs file manipulation using the respective classes. Gestalt's Principle is used in this component to enhance the cohesiveness of each class and reduce the coupling, where only necessary dependencies are utilized. The functions of each class are grouped accordingly to the very meaning that each class name suggest.

Urgenda primarily has 3 files:

- `data.txt` stores all the tasks, either completed or uncompleted, in JSON format. Each line, or each string, represents one task. This file is able to be renamed or moved to other directories as per user's desire.
- `settings.txt` stores the settings of the user, such as the file name and file location, so that on start-up Urgenda can retrieve these settings, retrieve the datafile and set preferences for the user. This file cannot be moved or renamed.
- `help.txt` is where the documentation for the user is stored. If the user ever require any form of assistance in entering commands, he can bring up the help panel, which retrieves the text from this file. This file cannot be moved or renamed.

Storage Class

In every case where `LogicData` needs to access or edit the data in the file or the datafile itself, it goes through `Storage` class, which then dispatches the corresponding method using the respective classes within the `Storage` component.

As mentioned above, apart from the `Storage` class which acts as the facade, all other classes have their own specific functions:

Class	Function

Class	Function
`FileEditor`	Contains all the file manipulation methods that is required in the `Storage` component - retrieving from file, writing to file, renaming, moving to other directories, clearing the file. Essentially, only this class can access and manipulate the actual file itself
`JsonCipher`	The primary ciphering tool. `Storage` uses the external library Gson that allows conversion of objects to a string. In this class, instead of converting directly from a `Task` to a `String`, a `Task` is converted to a `LinkedHashMap<String, String>`, then converted into a `String`. This allows for easier conversion back into a `Task` from a `String`. `JsonCipher` provides the tools required for converting from `Task` <=> `LinkedHashMap<String, String>` <=> `String`
`Encryptor`	A subclass of `JsonCipher`, the role of `Encryptor` is to encrypt all `Task` into a `String` using `JsonCipher` as a means of doing so.
`Decryptor`	A subclass of `JsonCipher`, the role of `Decryptor` is to decrypt all `String` into a `Task` using `JsonCipher` as a means of doing so.
`SettingsEditor`	This class handles all matters related to the settings of the user, in order not to mix it with the actual data file. File manipulation and encryption/decryption is done using the `FileEditor` class and through `JsonCipher` directly, since there is no need `Task` involved with the settings.

The main APIs of the `Storage` class include:

Method	Return type and function
`updateArrayList()`	Returns `ArrayList<Task>`. This method is used during startup to retrieve all tasks in the datafile and pack it into an ArrayList.
`save(ArrayList<Task> tasks, ArrayList<Task> archives)`	Void function. This method is used to store all tasks in the datafile, for easy retrieval, relocation to another computer.
`changeFileSettings(String path, String name)`	Void function. This method allows the datafile to be renamed and move to other directories/folders through Urgenda itself, with no need to enter File Explorer

Sequence diagram `updateArrayList`

`updateArrayList()` is the generic method for `updateCurrentTaskList()` and `updateArchiveTaskList()`.

Sequence diagram `save(ArrayList<Task> tasks, ArrayList<Task> archives)`

`save(ArrayList<Task> tasks, ArrayList<Task> archives)` saves the current list of tasks into the specified file by writing onto it.

Sequence diagram `changeFileSettings(String path, String name)`

`changeFilePath(String path)` and `changeFileName(String name)` are similar methods to `changeFileSettings(String path, String name)`, whereby the latter changes both the name and the directory the datafile is saved in. `changeFileSettings(String path, String name)` has two parts to it:

- edit the preferred file name and file location in `settings.txt`
- rename/relocate the actual datafile with the preferred name to the preferred location