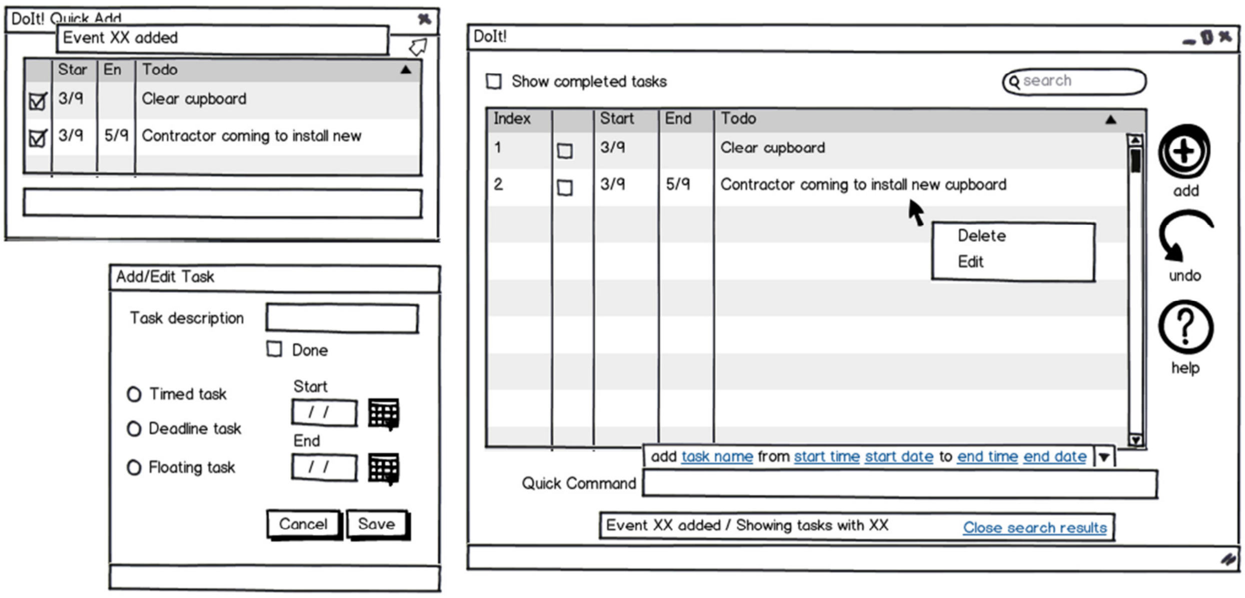
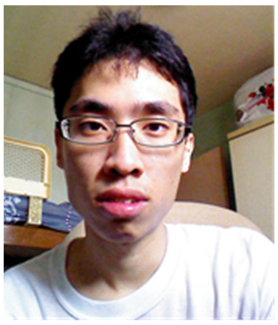
DoIt! Manual



	Star	En	Todo
☑	3/9		Clear cupboard
☑	3/9	5/9	Contractor coming to install new

Index		Start	End	Todo
1	☐	3/9		Clear cupboard
2	☐	3/9	5/9	Contractor coming to install new cupboard

Nguyen Hien Linh Team lead, architect, testing	 Yeo Kheng Meng Deadline watcher, lead programmer	Yeow Kai Yao GUI design, documentation

Credits

DoIt! would not be possible without the following software libraries:

- Joda-Time date and time handler http://joda-time.sourceforge.net/
- Natty natural language date parser http://natty.joestelmach.com/
- JLine console library https://github.com/jline/jline2

Contents

DoIt! User Manual

Welcome to DoIt!

DoIt! is an easy-to-use task organization program. Just type in what you have on your mind and DoIt! will put it on your schedule. Prefer using your mouse? DoIt! also provides an intuitive graphical interface, making it easy for you to manage your life!

1 System Requirements

DoIt! requires the following:

- Windows XP or later
- Java runtime environment 7. Download it from http://www.java.com/download

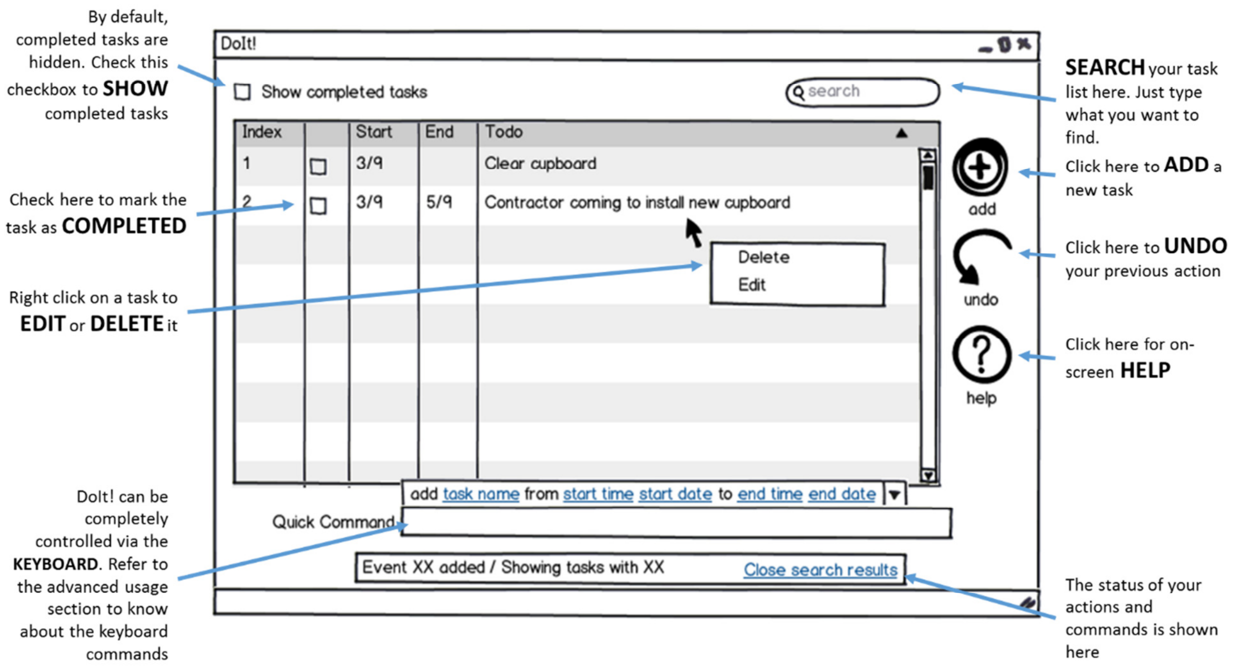
2 Running DoIt!

No installation is required. Just double click on the downloaded program's icon to launch the program.

DoIt also supports a complete command-line based interface. Run DoIt! with the –cli argument to do so.

3 Using DoIt!

3.1 DoIt! Main



Index		Start	End	Todo
1	☐	3/9		Clear cupboard
2	☐	3/9	5/9	Contractor coming to install new cupboard

Event XX added / Showing tasks with XX

By default, completed tasks are hidden. Check this checkbox to **SHOW** completed tasks

Check here to mark the task as **COMPLETED**

Right click on a task to **EDIT** or **DELETE** it

DoIt! can be completely controlled via the **KEYBOARD**. Refer to the advanced usage section to know about the keyboard commands

SEARCH your task list here. Just type what you want to find.

Click here to **ADD** a new task

Click here to **UNDO** your previous action

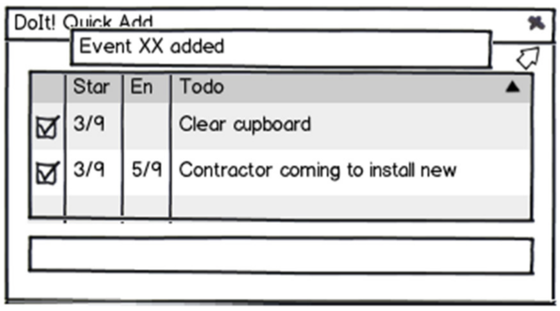
Click here for on-screen **HELP**

The status of your actions and commands is shown here

3.2 Quick Add

Pressing Ctrl-Alt-Z (you can change that) will launch the quick add dialog box. Simply type your new task into the box to add it to your task list.

The syntax follows the add command, but you don't have to specify the command word add. Type `full` to launch the full DoIt! user interface.



DoIt! Quick Add

Event XX added

	Star	En	Todo
☑	3/9		Clear cupboard
☑	3/9	5/9	Contractor coming to install new

3.3 Command-line Interface

Launch DoIt! in a terminal with the –cli argument. Then just type your commands into the prompts.

DoIt's command-line interface supports some handy shortcuts: press tab to have DoIt! complete your commands, and press up/down to view your command history.

4 Supported Task Types

DoIt! supports 3 distinct types of tasks:

1. Floating tasks: Tasks that have no specific time. E.g: Write report.
2. Deadline tasks: Tasks that have to be done by a specific time. Example: Write report by 2pm on 5 Sept 2012
3. Timed tasks: Tasks that have a specific start and end time. Example: Write report from 12pm 5 Sept 2012 to 2pm on 5 Sept 2012

5 Command Reference

DoIt! keyboard commands are easy to use! They start with a keyword for the action to take, following by some details about the action.

Note the following formatting used for the command syntax descriptions below:

- **`Bold`** = keyword; type exactly
- *`Italics`* = replace with appropriate argument
- `[Item in square brackets]` = optional argument
- a|b = use a or b, you cannot use them together

Syntax	Description	Example
	Adding tasks	
`add` *`task name`*	Add new floating task	`add fix cupboard`
`add` *`task name`* `[by]` *`date/time`*	Add deadline task	`add fix cupboard by 2pm on 5 Sep`
`add` *`task name`* `[from]` *`start date`* `[to]` *`end date`*	Add timed task with specific start and end time and date	`add fix cupboard from 2pm on 5 Sept to 4pm on 6 Sept`
If DoIt! is getting your task names mistaken as dates (hey, even humans misunderstand each other sometimes!), just enclose the task name in double inverted commands e.g. add `"fix cupboard"`		
	Viewing tasks	
`list`	Lists all tasks in order of due date	`list`

list ***[complete] [incomplete] [done] [undone] [floating] [deadline] [timed] [today] [tomorrow] [overdue]***	List all tasks that meet the criteria specified. Note that you can use more than one criteria, only tasks that match all the criteria will be shown.	list complete floating
search *[date]*	Lists all tasks occurring on or due on *date*	search 10 Sept search Oct
search *[start date]* **to** *[end date]*	List all tasks occurring between the two dates (inclusive)	search 10 Sept to 15 Sept
search *[keyword] [keyword] ...*	List tasks with the keywords	search cupboard search cupboard shoe
refresh	List tasks based on previous list/search command	refresh
When viewing tasks, an index number is displayed next to each task. The index number changes when different arguments are used with the list or search command, and when tasks are added or removed. When required by a command, the index number used corresponds to the index number shown with the latest list command. In the command-line interface, the list is not automatically refreshed after each modification. If you have made any changes to the tasks, do a refresh or list to update the index numbers.		
Delete		
delete *index*	Deletes the task with number *index*.	delete 1
delete done|completed|finished	Deletes all tasks that has been marked as done	delete done
delete all	Deletes all tasks	delete all
delete over	Deletes tasks that ended before the current time	delete over
Edit tasks		
edit *index* **[-name|-n** *new name]* **[-start|-s** *new start time / start date]* **[-end|-e** *new end time / end date]*	Changes the name/start time/end time of the task specified by *index* to the new value. To get *index*, see the note under "viewing tasks"	edit 1 -name fix cupboard edit 2 -start 1800 on 10 Sep
	You can combine multiple things to edit in the same command. Also note that shortcuts (e.g. s) can be used	edit 3 -name fix shoe rack -s 1800 10 Sept
	If no date is specified, the previous date will be kept. This also applies to the time	edit 5 -e 2100
Postponing tasks		
postpone *index* **to** *new start time / start date*	Postpone the task to *new start time/start date* (ending time and date is shifted accordingly so that the task duration remains the same). To get *index*, see the note under "viewing tasks"	postpone 1 to 3pm
postpone *index* **by** *duration*	Postpones the task specified by index by *duration* (ending time and date is shifted accordingly so that the task duration remains the same)	postpone 1 by 1 hour
Marking tasks		

done *index*	Marks the task specified by *index* as done. To get *index*, see the note under “viewing tasks”	`done 1`
undone *index*	Marks the task specified by *index* as not yet done	`undone 1`
Undo changes`		
undo	Undo the last change you have made.	`undo`
Getting command help		
help	Show a list of all possible commands	`help`
help *[command]*	List the possible usage for *command*	`help add`

DoIt! Developer Guide

1 Introduction

Welcome to the developer guide for DoIt!

In case you do not know yet, DoIt! is a simple and easy-to-use task list management software. DoIt! is distinct from other task management software as it allows for complete control of it via the keyboard.

So let's get started!

2 Architecture

DoIt! is separated into various components: UI, Logic, Storage and some shared components, as illustrated in the overview diagram below. In the code, this is made distinct by having *one package* for each major component.

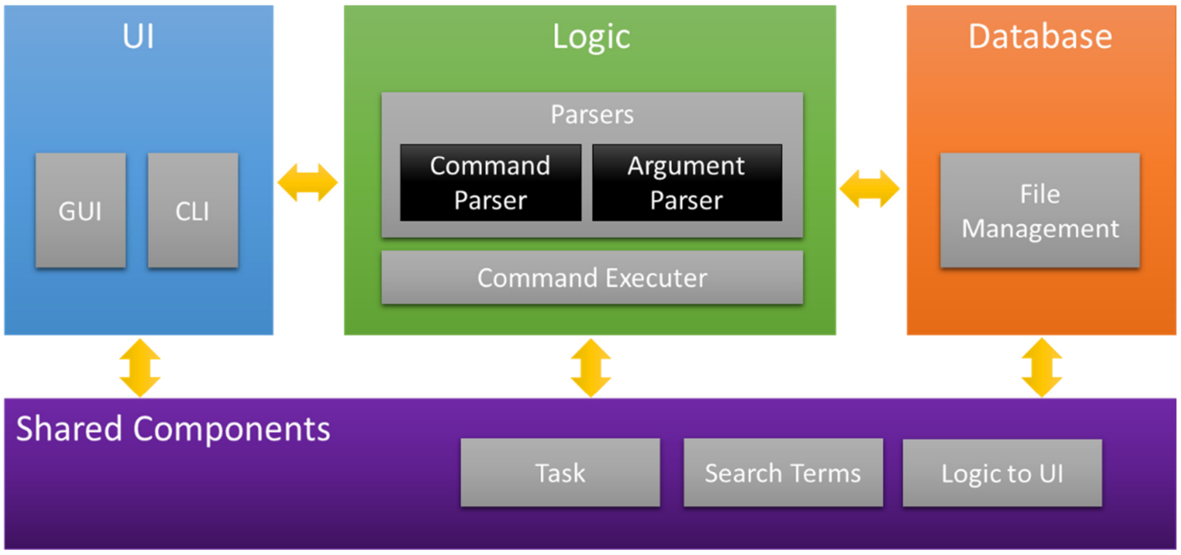


Figure 1: Architecture of DoIt!

3 Understanding the command flow

In DoIt!, all actions start from the user (and correspondingly, the user interface). That is, all actions start with the user doing something. The UI sends the command to the Logic component, which parses the command and then takes the appropriate action, usually invoking the Storage component. The response then goes back to the user the same way. This is reflected in the sequence diagrams:

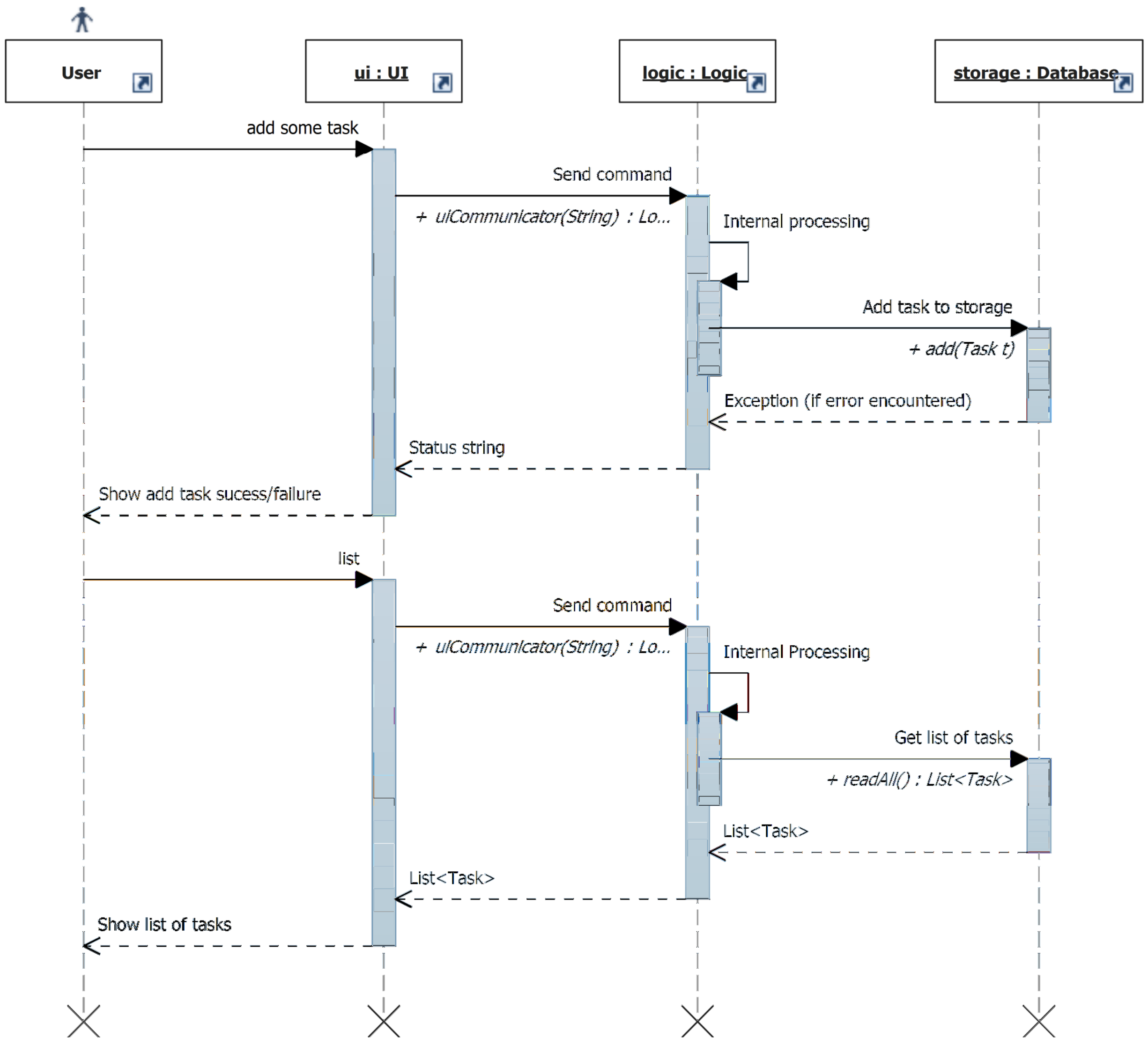


Figure 2: Sequence Diagram for adding a new task and viewing tasks. For details of "internal processing", refer to the sequence diagram under Logic.

4 Components

With the knowledge of how the different components in DoIt! and how they interact, you can get more details about your component of interest in this section.

4.1 UI

This is the place to look for if you want to modify the user interface or develop an alternative user interface.

4.1.1 Classes

The UI in DoIt! is designed as a UI abstract class from which actual concrete user interfaces can be extended from. It is thus easy to add an alternative user interface, for example one that works with mobile devices.

There are currently four concrete UIs: the GuiMain subclass, the GuiQuick subclass, the Cli subclass and the CliWithJline subclass.

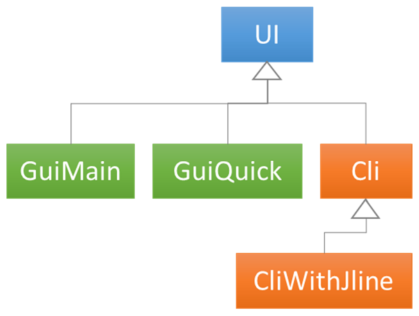


Figure 3: Class diagram for UI

4.1.2 Important APIs of the UI Abstract Class

`abstract void`	**`runUI`**`()` This method is the starting point of every UI; it is called when the UI is to be run.
`LogicToUi`	**`sendCommandToLogic`**`(String command)` Passes `command` to logic to be processed. Returns a `LogicToUi` object comprising a String message, or a String and a List<Task> depending on `command`.
`String`	**`checkFilePermissions`**`()` Sends a command to logic to check for the database file permissions. Returns a string containing the status of the file. **It is important to call this method before allowing any user interaction with tasks so that the user will be warned of a read-only situation.**

4.1.3 GuiMain Subclass

The GuiMain subclass holds the code for the main window of DoIt's graphical user interface. It is developed in Java Swing, and uses (mainly) a JTable and a JTextField. Pop-up boxes are implemented using a JEditorPane in a JPopupMenu, a technique used in [1].

4.1.4 GuiQuick Subclass

This subclass implements the quick view/quick add window of DoIt!

GUI Coding and WindowBuilder

The GUI for DoIt! has been designed with Eclipse's WindowBuilder. If you are changing the code, periodically check that WindowBuilder is still able to display the GUI in "design view" as WindowBuilder has been noted to give parsing errors for code that is still valid. This will ensure that development of the GUI in WindowBuilder continues to work.

4.1.5 Cli Subclass

This subclass implements a plain-Jane command-line interface that works using standard Java I/O.

[1] http://www.jroller.com/santhosh/date/20050620#file_path_autocompletion

4.1.6 CliWithJline Subclass

This subclass extends the Cli subclass and implements an enhanced command-line interface that uses the Jline console library to implement features such as a command history. Note that the Jline library interfaces natively with the terminal of the underlying operating system, and as such does not work when a full terminal is not available, for instance in the Eclipse console.

4.2 Logic

This is where the commands from the user interface are processed and the relevant action taken.

4.2.1 Class Diagram

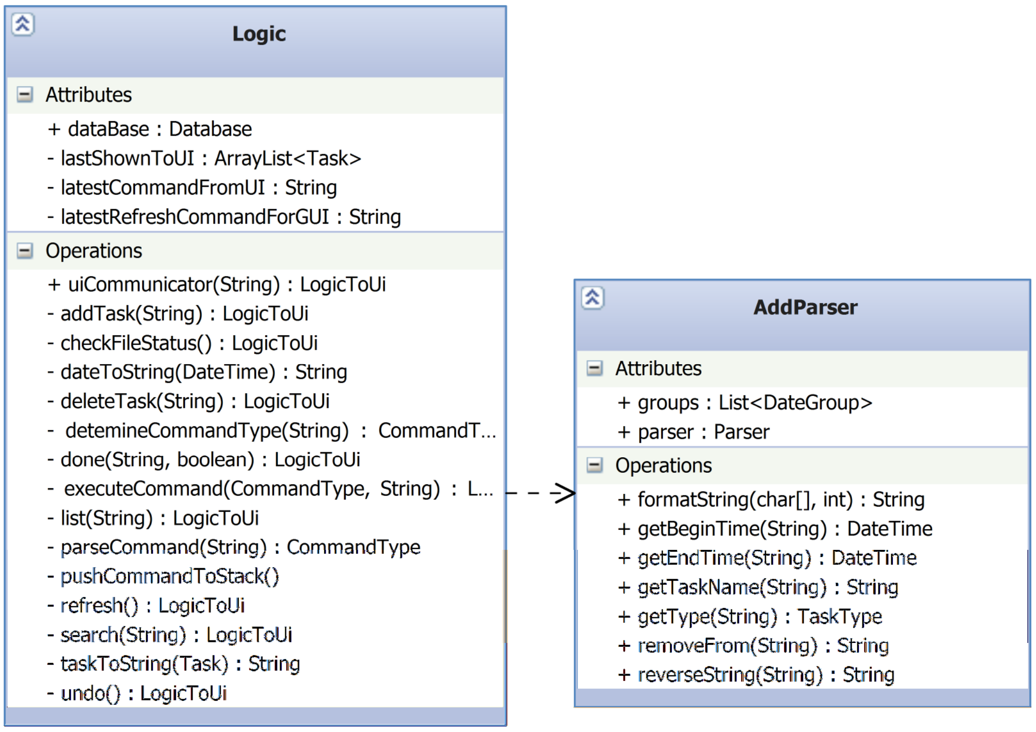

Logic
Attributes
+ dataBase : Database
- lastShownToUI : ArrayList<Task>
- latestCommandFromUI : String
- latestRefreshCommandForGUI : String
Operations
+ uiCommunicator(String) : LogicToUi
- addTask(String) : LogicToUi
- checkFileStatus() : LogicToUi
- dateToString(DateTime) : String
- deleteTask(String) : LogicToUi
- detemineCommandType(String) : CommandT...
- done(String, boolean) : LogicToUi
- executeCommand(CommandType, String) : L...
- list(String) : LogicToUi
- parseCommand(String) : CommandType
- pushCommandToStack()
- refresh() : LogicToUi
- search(String) : LogicToUi
- taskToString(Task) : String
- undo() : LogicToUi
AddParser
Attributes
+ groups : List<DateGroup>
+ parser : Parser
Operations
+ formatString(char[], int) : String
+ getBeginTime(String) : DateTime
+ getEndTime(String) : DateTime
+ getTaskName(String) : String
+ getType(String) : TaskType
+ removeFrom(String) : String
+ reverseString(String) : String

Figure 4: Class diagram for Logic

4.2.2 Important API

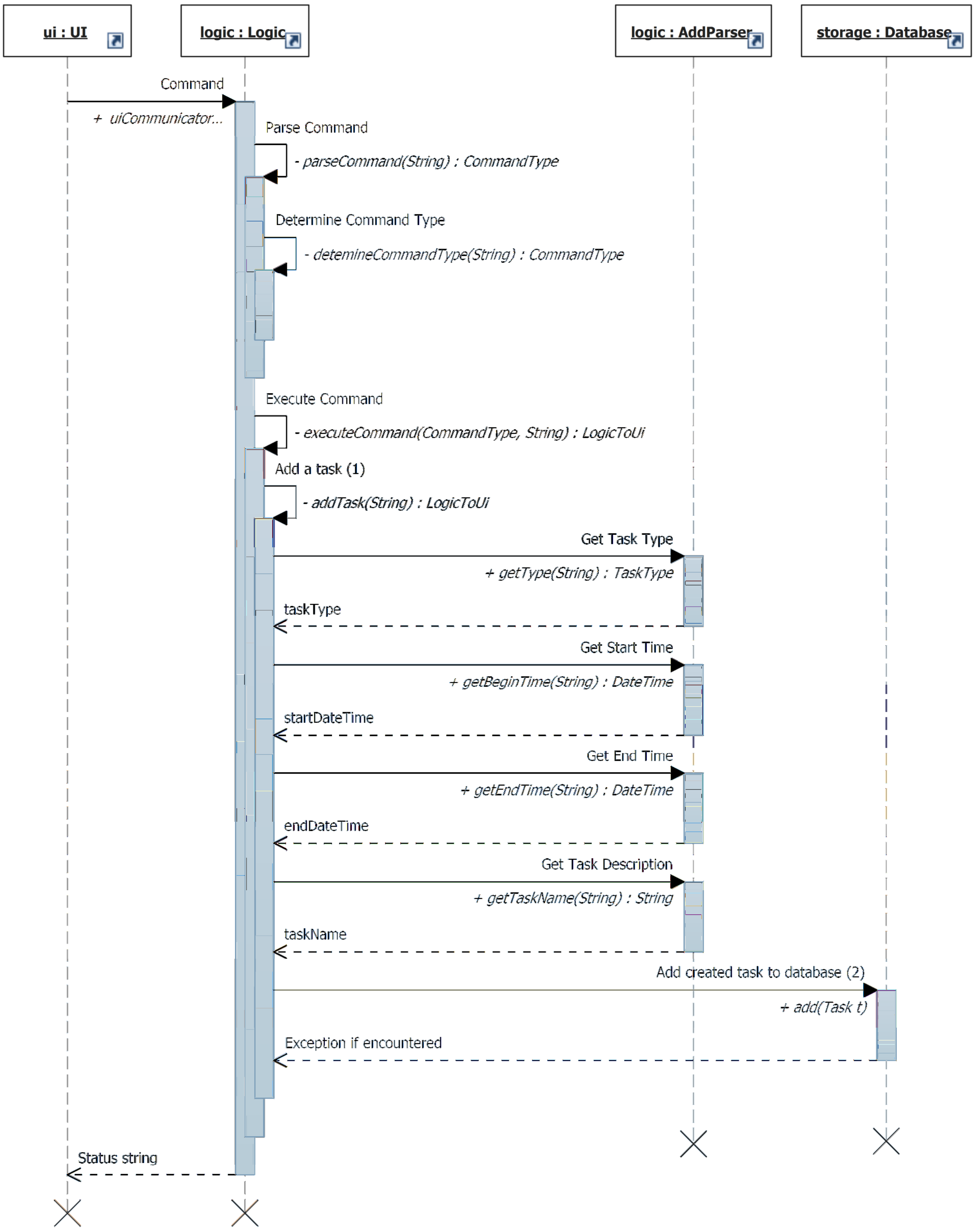
The most important interface to Logic is the uiCommunicator(String) method, which serves as the entry point to the entire Logic component.

Note about Logic and serial numbers in tasks

The Logic and UI components are designed such that the Logic component holds a complete representation of the tasks currently shown to the user. This is needed as the index number for each task shown in the UI is based on the current "view" of the tasks, which makes the index number different from each task's serial number, the latter which is used for some commands to the database such as update and delete. As such, a mapping to the internal serial number for each task is required and by having a complete representation of the tasks shown, this mapping can be done by Logic.

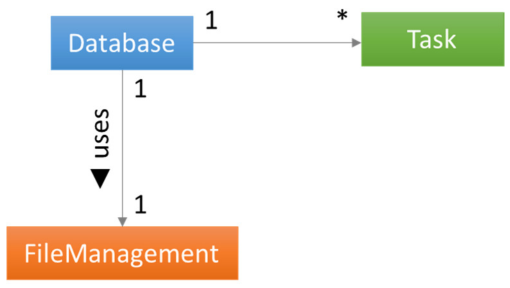
4.2.3 Sequence Diagram

The following is a sequence diagram of the process in the Logic class when **adding a task**. Note that the diagram is largely similar for other commands, except **(1)** *Add a task (1)* is replaced by the appropriate method call in Logic, **(2)** *Add created task to database (2)* is replaced by the corresponding call to Database, and **(3)** calls to AddParser are replaced by calls to the corresponding parser class. For some commands with simpler parsing, you would not find an associated parser class as the parsing is done in the method itself.


ui : UI
logic : Logic
logic : AddParser
storage : Database
Command
+ uiCommunicator...
Parse Command
- parseCommand(String) : CommandType
Determine Command Type
- deteminеCommandType(String) : CommandType
Execute Command
- executeCommand(CommandType, String) : LogicToUi
Add a task (1)
- addTask(String) : LogicToUi
Get Task Type
+ getType(String) : TaskType
taskType
Get Start Time
+ getBeginTime(String) : DateTime
startDateTime
Get End Time
+ getEndTime(String) : DateTime
endDateTime
Get Task Description
+ getTaskName(String) : String
taskName
Add created task to database (2)
+ add(Task t)
Exception if encountered
Status string

4.3 Storage

Figure 5: Class diagram for Storage



The storage component handles the writing and reading of tasks to the text file storage component on the disk.

This component is implemented with two classes: Database and FileManagement.

4.3.1 Database class

Upon instantiation, the Database class reads in the text file of the tasks through the FileManagement class and stores the tasks in a `List<Task>`. The tasks in the list are stored in order of the start date of the tasks.

4.3.1.1 Important APIs

`void`	**`add`**`(Task newTask)` To add a new task to database
`void`	**`delete`**`(int serial)` To delete an existing task in database. The serial number can be obtained from the Task object you want to delete. Will throw `NoSuchElementException` if existing Task by serial number cannot be found

`void`	**`deleteAll`**`()`
	`deleteDone``()`
	`deleteOver``()`
	To delete ALL tasks/completed tasks/tasks that are past their deadlines or end times in database, respectively.

`Database.DB_File_Status`	**`getFileAttributes`**`()`
	To get file permissions of the database file. One of the following values will be returned:
	`FILE_ALL_OK, FILE_READ_ONLY, FILE_UNUSABLE, FILE_PERMISSIONS_UNKNOWN, FILE_IS_CORRUPT`
	You are strongly suggested to call this method at the earliest opportunity to verify read and write permissions.

`int`	**`getUndoStepsLeft`**`()` Get the number of undo operations remaining. Only commands that involve a database change will increment the count (commands like “list” do not increment the count). **Logic is strongly advised to check this before attempting an undo operation.**
`Task`	**`locateATask`**`(int serial)` To locate existing task in database
`java.util.ArrayList<Task>`	**`readAll`**`()` To return all the tasks in database
`java.util.ArrayList<Task>`	**`search`**`(shared.SearchTerms terms)` To give the results based on a search term.
`void`	**`undo`**`()` To undo the last write operation in database. Only commands that involve a database change will be added in the undo stack. **Throws NoMoreUndoStepsException if no more steps left. Also see getUndoStepsLeft().**
`void`	**`update`**`(int originalSerial, Task updated)` To update existing task in database Will throw `NoSuchElementException` if existing Task by serial number cannot be found

4.3.1.2 Exceptions

Any method that involves a change in the database may throw the following exceptions:

1. `java.io.IOException` is thrown if the Database class is unable to write to the file due to some permission issues.
2. `storage.WillNotWriteToCorruptFileException` is thrown if the file can be accessed but is corrupt. As the name implies, the Database class will not write to the file to prevent corrupting the text file further. The user should be shown the list of tasks that can still be read.

The `List<Task>` in the Database class will not be modified if any of the above exceptions occur.

4.4 Shared Components

4.4.1 Task

The task object is the main data type used by DoIt! Each task object holds one task.

4.4.1.1 Constructors

The class has 6 constructors, allowing you to create the 3 different types of tasks (floating, deadline and timed) and 3 more that allow you to set the (boolean) done status of the task when creating the object.

To instantiate an undone floating task:

```
Task(java.lang.String name)

Example: new Task(“This is an undone floating task”);
```

To instantiate an undone deadline task:

```
Task(java.lang.String name, org.joda.time.DateTime deadline)

Example: new Task(“This is an undone deadline task”, new DateTime(2011, 9, 5,
23,59))
```

To instantiate an undone timed task:

```
Task(java.lang.String name, org.joda.time.DateTime startTime,
org.joda.time.DateTime endTime)

Example: new Task(“This is a timed task”, new DateTime(2011, 9, 5, 23,59), new
DateTime(2013, 12, 31, 00, 00))
```

4.4.1.2 Important APIs

`void`	**`done`**`(boolean newDoneStatus)` Sets done status for the task
`DateTime`	**`getDeadline`**`()` Get deadline of deadline task. **If this field is not applicable, the Task class will return a public constant INVALID_DATE_FIELD DateTime object.**
`DateTime`	**`getEndTime`**`()` Get end time of timed task **If this field is not applicable, the Task class will return a public constant INVALID_DATE_FIELD DateTime object.**

`int`	**`getSerial`**`()` For the purposes of deletion and update commands. This is so as the index number the user sees varies according the UI filters.
`DateTime`	**`getStartTime`**`()` Get start time of timed task **If this field is not applicable, the Task class will return a public constant INVALID_DATE_FIELD DateTime object.**
`String`	**`getTaskName`**`()`
`Task.TaskType`	**`getType`**`()` Returns the type of task
`boolean`	**`isDeadlineTask`**`()`
`boolean`	**`isDone`**`()` To check if the Task is completed
`boolean`	**`isFloatingTask`**`()`
`boolean`	**`isTimedTask`**`()`
`boolean`	**`searchDateRange`**`(DateTime startRange, DateTime endRange)` For ease of searching by the database, the task will do the comparison by itself
`boolean`	**`searchName`**`(String term)`
`String`	**`showInfo`**`()` Returns the entire contents of the Task object in the same format as database.txt. **For debugging purposes only.**
`void`	**`updateOrClone`**`(Task updated)` To clone the current task given to this object. All fields including serial number will be copied

4.4.2 SearchTerms

The SearchTerms object is a container for passing search terms from Logic to Storage. It is used by the list and search commands.

It holds the following attributes that can be set in the constructor:

Attribute	**Description**
`completedTasks : boolean`	If true, matches only completed tasks
`incompleteTasks : boolean`	If true, matches only incomplete tasks
`timedTasks : boolean`	If true, matches only timed tasks
`deadlineTasks : boolean`	If true, matches only deadline tasks
`floatingTasks : boolean`	If true, matches only floating tasks
`startRange : DateTime and endRange : DateTime`	If specified, matches only between `startRange` and `endRange` inclusive
`keywords : String[]`	If specified, matches only tasks that contain ALL the strings in the array. Keywords are case-insensitive.

Attributes can be combined. For example:

- Get incomplete timed tasks: incomplete = true, timed = true
- Get deadline tasks with "market" and "home" keywords: deadlineTask = true, String[] keywords = { "home", "market" }
- Get tasks that happen on 27 Dec 2012: startRange = new DateTime(2012, 12, 27, 00,00), endRange = new DateTime(2012, 12, 27, 23, 59)