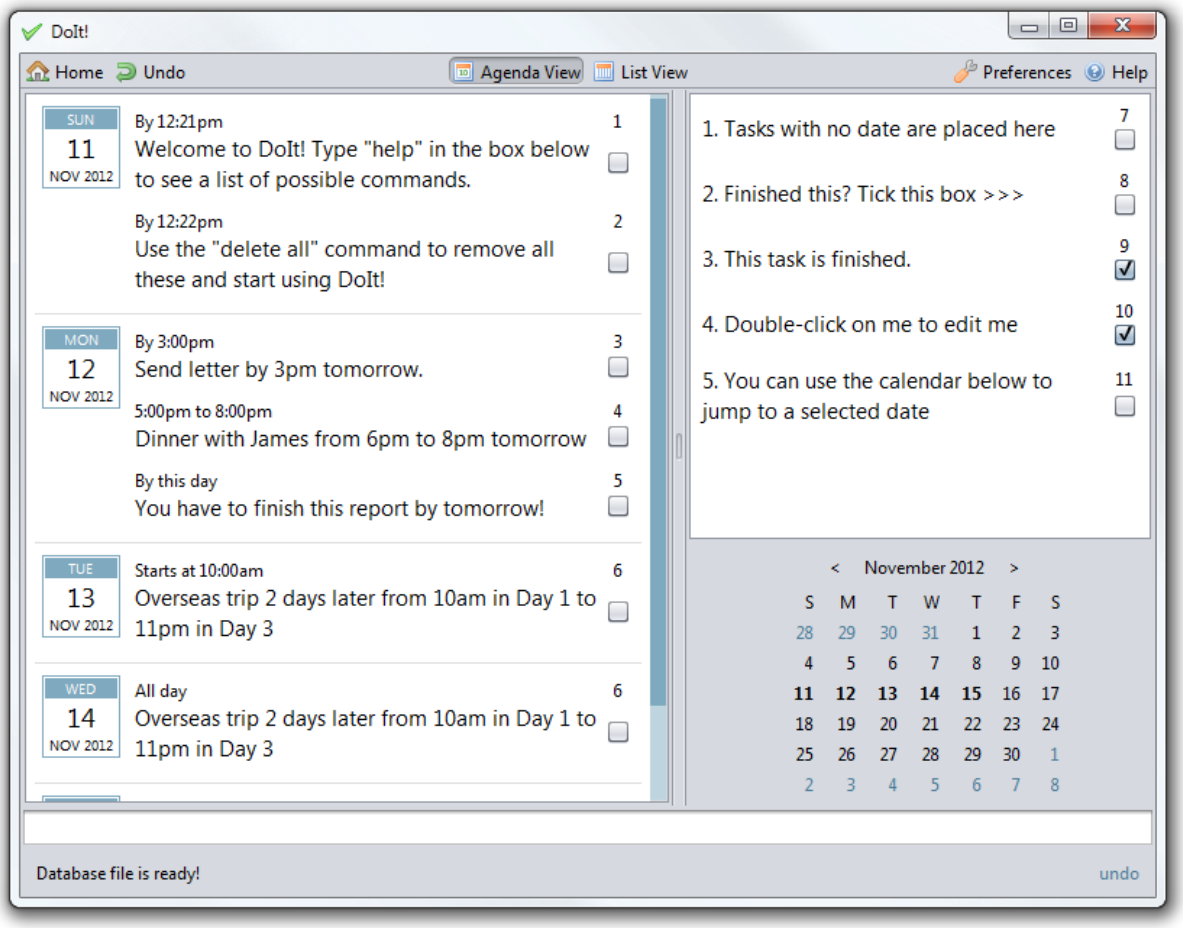
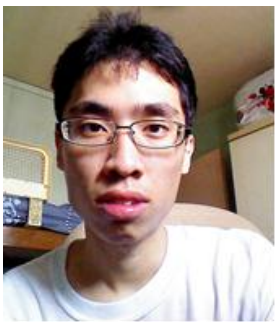
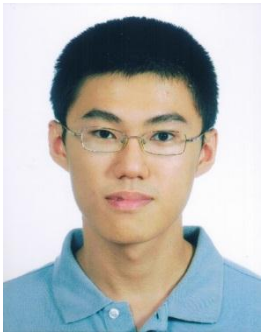
[w15-2j][V0.5]

DoIt! Manual



		
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/
- JIntellitype for global binding of keyboard shortcuts https://code.google.com/p/jintellitype/
- JLine console library https://github.com/jline/jline2

The DoIt! development team would also like to acknowledge the use of the following:

- Nuvola icons http://www.icon-king.com/projects/nuvola/ for DoIt's icon
- FamFamFam Silk icons http://www.famfamfam.com/lab/icons/silk/ for icons in DoIt's toolbar

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, 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

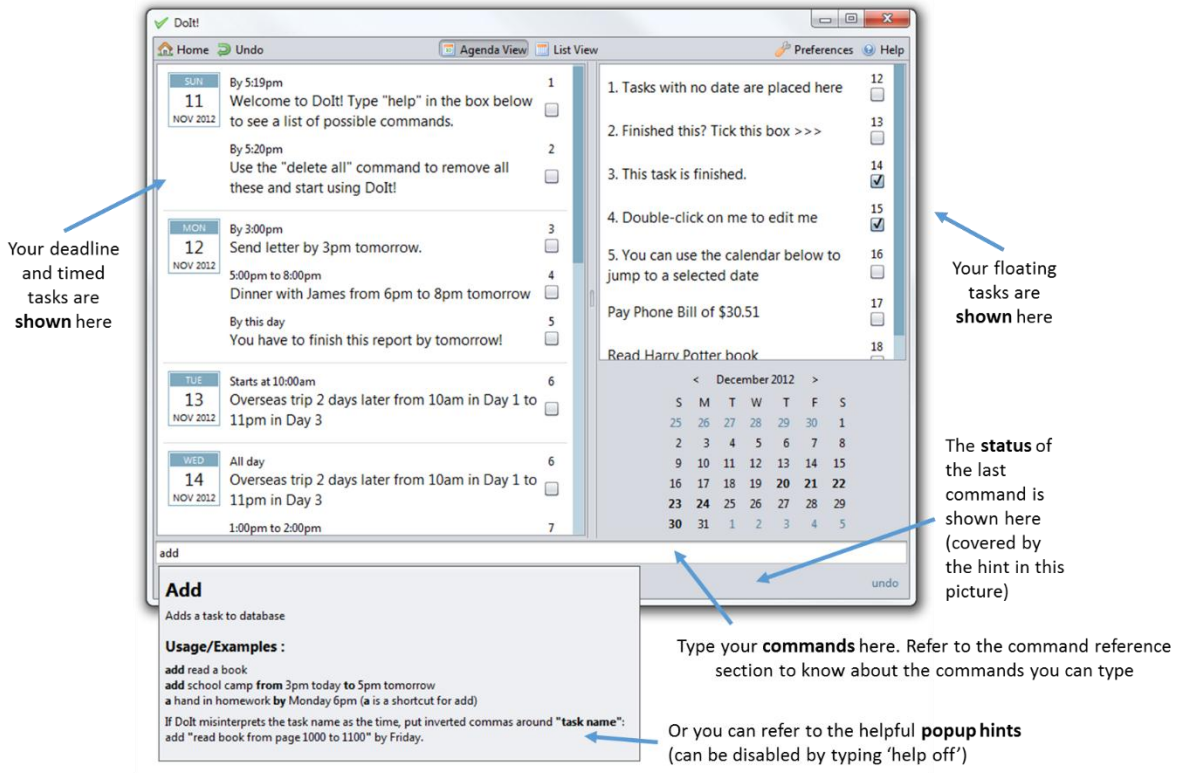
2 Running DoIt!

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.

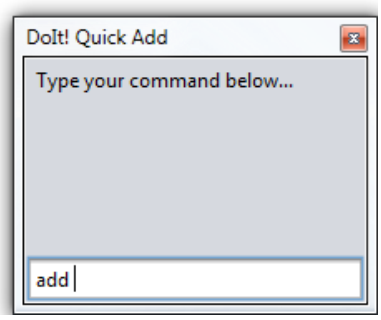
3 Using DoIt!

3.1 DoIt! Main



3.2 Quick Add

Pressing Win+A (you can change that in the preferences) 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 that of the main window.

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. It also supports different terminal sizes so feel free to resize your terminal window to whatever size you like!

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

5 Command Reference

DoIt's 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, 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`

sort *[type]|[done] [start]|[end]|[name] [descending][reverse]*	Sorts the tasks based on criteria specified. Only one criteria is allowed except for descending/reverse. If "descending" is not specified, the tasks are sorted in ascending order. If nothing is specified, the tasks are sorted by start date/deadline in ascending order.	`sort name`
		`sort start`
		`sort done descending`

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 | index*	Deletes the task(s) with number *index*.	`delete 1` `delete 2 3 4`
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. Both done and undone tasks will be deleted.	`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* **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 as done/undone		
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 complete control via the keyboard.

This document aims to help developers like you to understand DoIt!'s design and how you can extend it.

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.

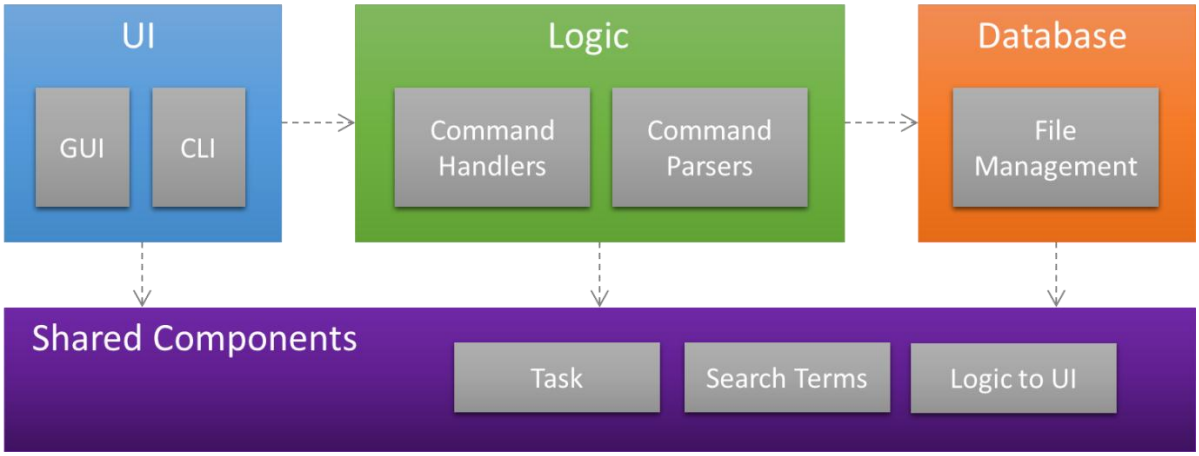


Figure 1: Architecture of DoIt!

3 Understanding the Command Flow

In DoIt!, all actions are initiated by 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. The Logic component parses the command and then takes the appropriate action, usually invoking the Storage component. The response then goes back to the user via the method's return style. This is reflected in the sequence diagrams:

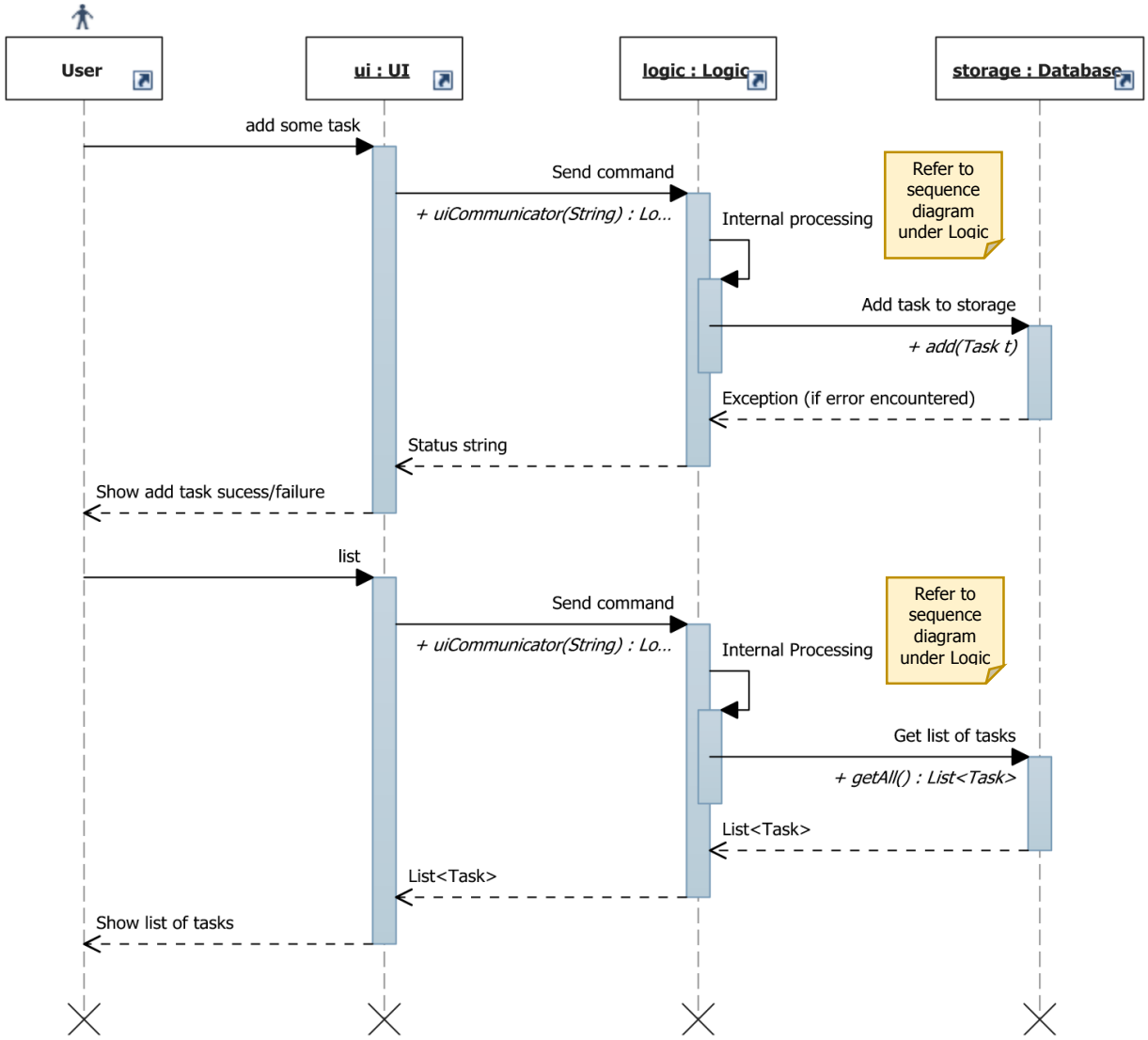


Figure 2: Sequence diagram for adding a new task and viewing tasks.

4 Components

Now that you have seen an overview of the various components in DoIt!, as well as how they interact, you are probably interested in some more details of a component or two… 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. Several classes inherit from `UI`, including `GuiCommandBox` that implements a GUI command box with hints and command history support, as well as `Cli` that implements a terminal interface to DoIt!

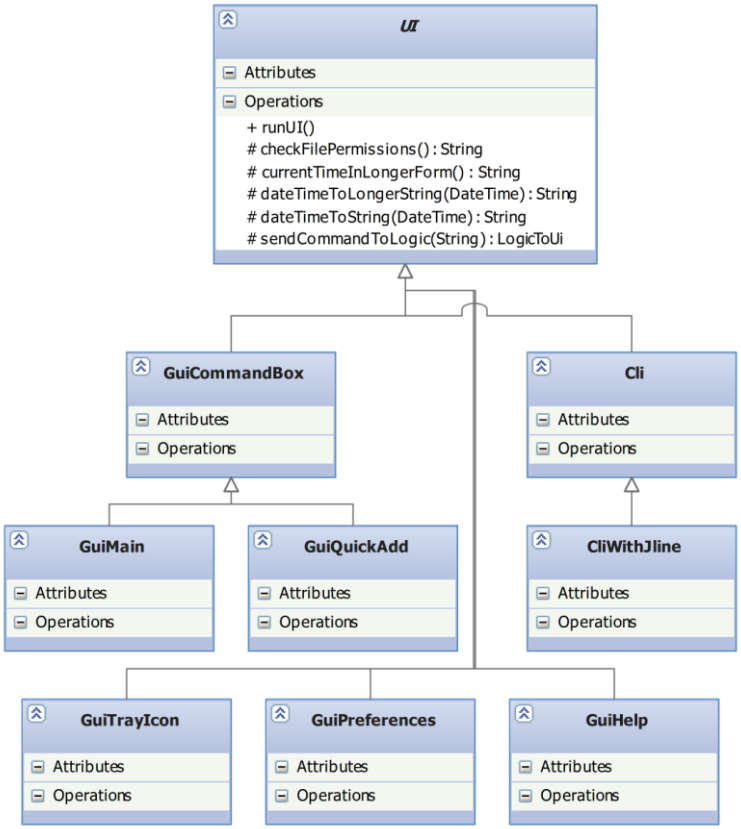


Figure 3: Class diagram for UI Component

4.1.2 Important Methods 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 "`filestatus`" command to logic to check for the database file permissions. Returns a string containing the status of the file. **It is important for the UI class to call this method before beginning any user interaction so that the user will be warned of a read-only or file-lock situation.**
`String`	**`getHTMLHelp(String command), getNoHTMLHelp(String command)`** To get command specific help for display to the user. See Hint Class for more details.

4.1.3 GuiCommandBox

This class implements a basic command box in a GUI with pop-up hints and command history. Pop-up boxes are implemented using a `JEditorPane` in a `JPopupMenu`, a technique used in [1].

4.1.4 GuiMain

The `GuiMain` subclass holds the code for the main window of DoIt's graphical user interface.

4.1.5 GuiQuick Subclass

This subclass implements the quick add window of DoIt!

GUI Coding and WindowBuilder
WindowBuilder has been occasionally noted to give parsing errors for code that is still valid. So check the "design view" often so that development of the GUI in WindowBuilder continues to work.

4.1.6 Cli Subclass

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

4.1.7 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 Tab Completion and support for a resizable terminal window. 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.

4.1.8 Hint Class

This class provides some help text for each of the commands. Its methods are not accessible directly; to get the help text, use `getHTMLHelp()` or `getNoHTMLHelp()` in the UI class.

Help Text File: help.xml

The help text is stored in the XML format at `/src/main/resource/help.xml`.

The hint class will accept only one name and summary field. The name field should be enclosed in `<h1>` HTML tags to allow for custom styling by the GUI. Usage of `<b>` tags is encouraged to emphasise a specific aspect of a command. Multiple usage and extra tags are allowed.

The hint class will strip away the HTML encoding and return a normal help string if it is called through `getNoHTMLHelp()`.

Refer to appendix for example snippets of the format.

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

4.2 Logic

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

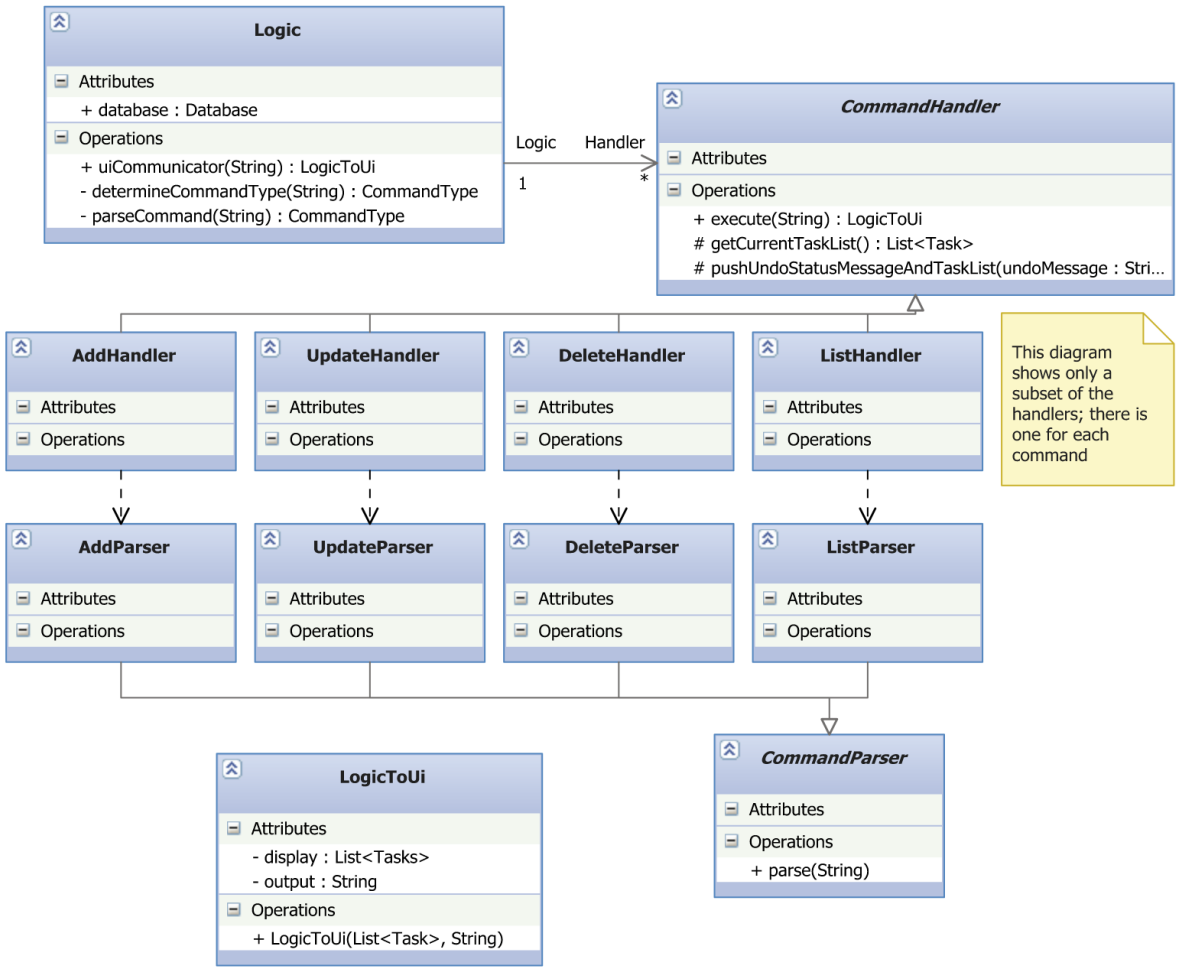


Figure 4: Class diagram for Logic Component

4.2.1 Important APIs

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

Logic and serial numbers in tasks
The Logic component holds a complete representation of the tasks currently shown to the user. This is needed because the index number shown in the UI is different from the serial number in the tasks. The Logic is responsible for holding the mapping between them.

4.2.2 LastShownToUi

This class holds the last list of tasks that was sent to the UI to be shown to the user.

The use of certain commands requires the user to provide the index number of the task they wish to work with. However, a mapping between the index number and the actual task it is referring to is needed for the command to work. Therefore, the Logic component will maintain the latest list of tasks given to the UI. This last shown list is stored in this class for easy accessibility by the command handlers/parsers.

This class applies the Singleton pattern to ensure consistency among the handlers/parsers. An instance of LastShownToUI class has been pre-acquired and referenced in the handlers/parsers via this protected variable:

```
protected static LastShownToUI lastShownObject
```

To retrieve the list of last shown tasks, use this:

```
LastShownToUI.getLastShownList()
```

To update the LastShownToUI instance with the new list, use this:

```
LastShownToUI.setLastShownList(List<Task> newList)
```

4.2.3 Command Handlers

Command handlers act on a particular command they are designed to handle.

Every command handler must extend the `CommandHandler` abstract class, which specifies that the handler must implement an `execute()` method. **Note that each command handler is expected to repeat the same action if the `execute()` method of a handler object is called again.** Its constructor must also accept a `String` parameter which is the arguments given for that command.

The instantiation of its associated parser should also be done in the constructor. See the `CommandParser` section for more details.

Only Command Handlers will have the view over Database object. This view has been obtained referenced via this variable:

```
protected static Database dataBase
```

Input/Output Requirements

The return type of the `execute()` command is a `LogicToUi` object. All exceptions are expected to be caught by the handlers. An appropriate return message must be generated after by every handler and stored to the `LogicToUi` object.

A possible list of return messages are stored as protected constants in the CommandHandler.

Refer to the `LogicToUi` section for more details.

Supporting undo operations

Support for the undo operation is only required for handlers which modify the database like `add` and `delete`. Commands like `list` and `search` should not be added to the undo history.

In order to provide the undo functionality, the CommandHandler will maintain a stack (implemented as a `LinkedList`) of database clones and their associated undo messages.

They are only accessible through the protected methods in the `CommandHandler`.

How to support undo in the handler?

To ensure the consistent saving of every undo step, follow the following steps. Place the following statements into a method named `updateDatabaseNSendToUndoStack()` – refer to Appendix for a sample. Then call this method just before `LogicToUi` is returned.

1. At the top of the method, place the following line to create a copy of the current database state before your modifications.
   `List<task> currentTaskList = super.getCurrentTaskList();`
2. Perform the necessary modification operations on the database.
3. At the bottom of the method, place the line below. The `undoMessage` is the message shown to the user when your command is undone, and will be displayed in this manner: "The `undoMessage` has been undone" so phrase your message accordingly.
   `super.pushUndoStatusMessageAndTaskList(undoMessage, currentTaskList);`

If any exceptions are thrown by the database during the modification of the database in statement (2), statement (3) will not be executed and thus nothing will be pushed to the undo stack.

When the database throws an exception, your operation will not be committed and thus there is no need to rewrite to the database.

Supporting refresh operations

The `CommandHandler` will store the last view and sort operations with the help of these respective variables.

```
protected static CommandHandler latestRefreshHandlerForUI
protected static CommandHandler latestSortHandlerForUI
```

All viewing/filter commands are required to store a reference of themselves into one of these variables should they require the GUI to maintain that view after a refresh.

The refresh command will execute the latest sort handler which will in turn execute the latest list/search handler. This is to ensure a seamless viewing experience as the sorting will be kept across different view commands.

During program launch, these variables will be initialised with default list and sort handlers. The former will list all tasks and the latter will sort the tasks by start dates/deadline with floating tasks at the back.

4.2.4 Command Parsers

Command Parsers exist to aid command Handlers in processing each command. They parse the arguments provided by the user. Since each command is unique, you have considerable leeway in the implementation, but the following are the basic guidelines:

1. All parsers must extend the `CommandParser` abstract class and implement the `parse()` method. Similar to the `CommandHandler`, its constructor must accept a `String` parameter which is the argument for that command.
2. The parser must **not** modify the database in any way. For example, if a task is to be added to the database, the parser should only generate the required fields of the task and not add the task itself to the database.
3. For arguments which have an index, the parser should obtain the relevant `Task` object from the `lastShownToUi` list to extract its serial number. This serial number is then returned to its associated handler.
4. Exceptions can be thrown from the parser and are expected to be caught by its associated handler.

The creation of a parser for each handler is not mandated but highly recommended for possible sharing of similar parsers in future.

4.2.5 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)** the *AddHandler* class is replaced by the appropriate command handler, **(2)** *Add created task to database* is replaced by the corresponding call to Database, and **(3)** calls to *AddParser* are replaced by calls to the corresponding command parser.

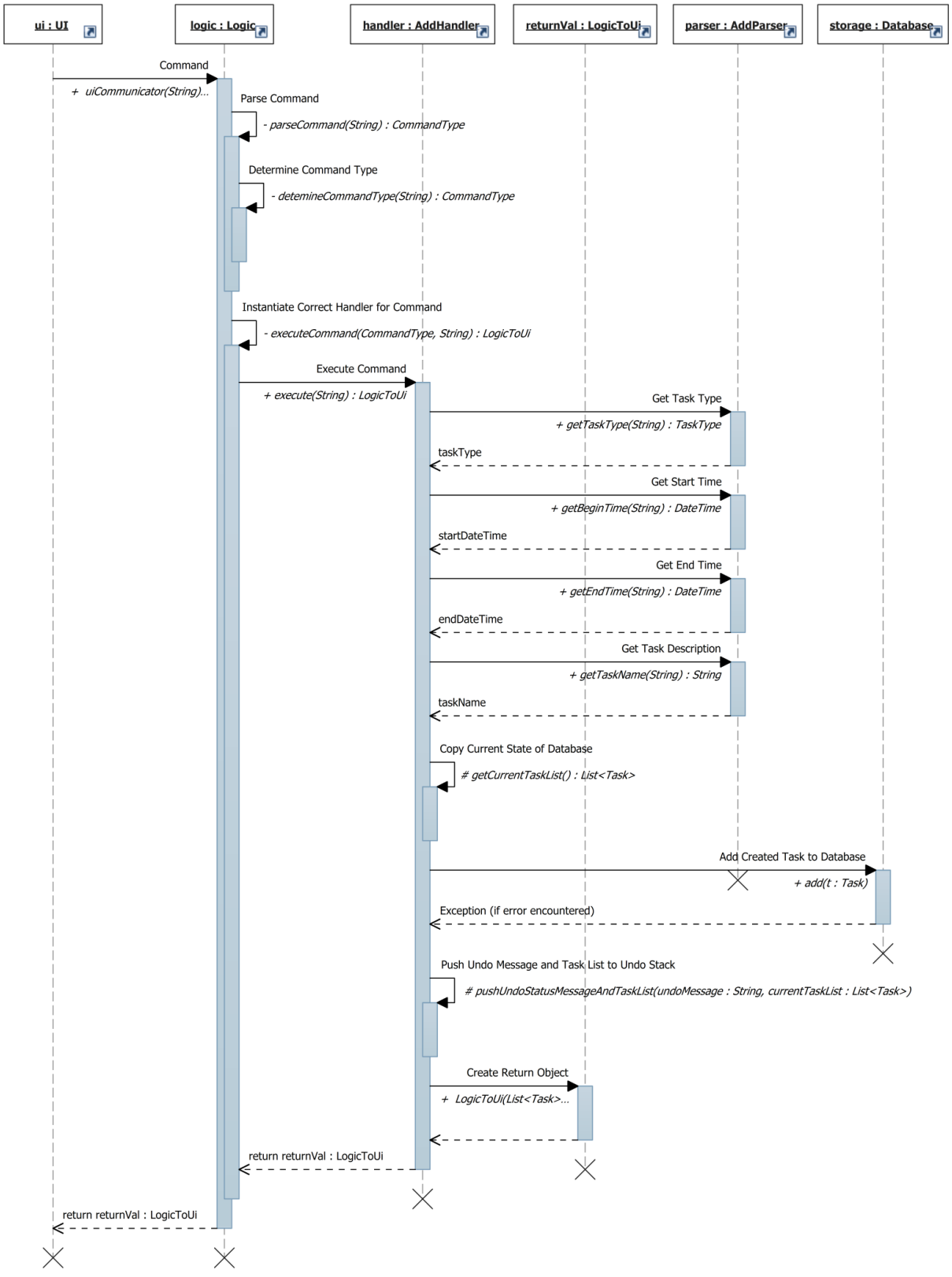


Figure 5: Sequence Diagram showing the process of adding a task

4.3 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`.

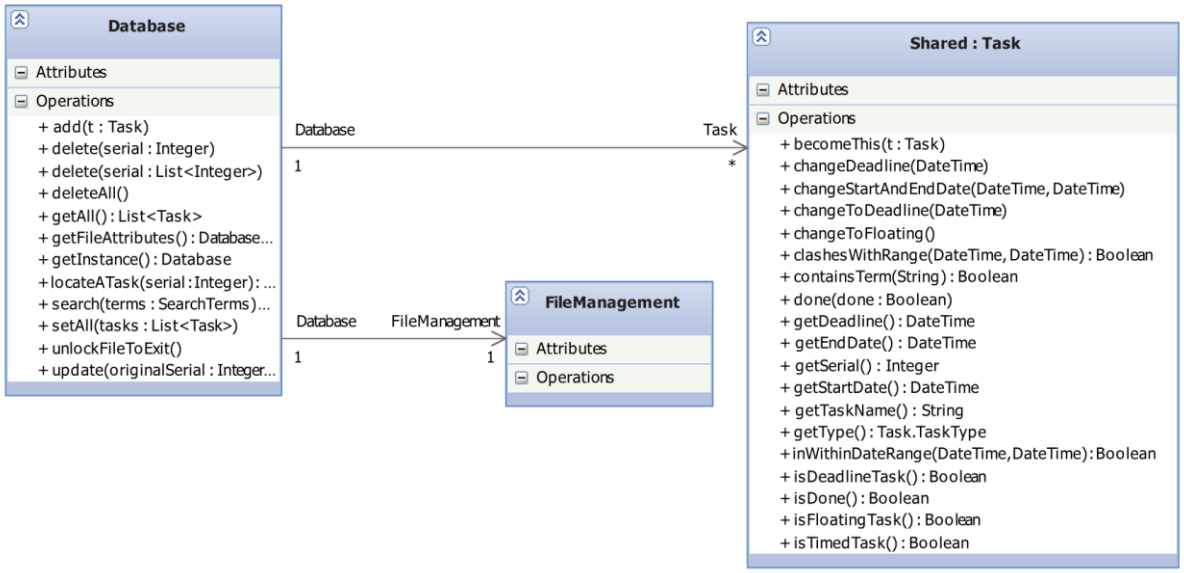


Figure 6: Class Diagram for Storage Component

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 `HashMap<Integer, Task>` where `Integer` is the serial number of the Task.

The database will do a deep copy (clone) for every List or Task object that travels in or out of its public APIs. This is to prevent modification of the tasks stored internally via external references.

Important APIs

`void`	**`add`**`(Task newTask)` Adds a new task to database.
`void`	**`delete(int serial)`** **`delete(int[] serial)`** Deletes an existing task(s) in database. The serial number can be obtained from the Task object you want to delete.
`void`	**`deleteAll`**`()` Deletes all tasks in database.
`Database.DB_File_Status`	**`getFileAttributes`**`()` Gets the file permissions of the database file. One of the following values will be returned: `FILE_ALL_OK, FILE_READ_ONLY, FILE_IS_LOCKED, FILE_PERMISSIONS_UNKNOWN, FILE_IS_CORRUPT` **You are strongly suggested to call this method at the earliest opportunity to verify read and write permissions.**
`Task`	**`locateATask`**`(int serial)` Locates an existing task in the database that has `serial`.

`java.util.List<Task>`	**`readAll`**`()` Returns a `List<Task>` of all the tasks in the database.
`java.util.List<Task>`	**`search`**`(shared.SearchTerms terms)` Returns a `List<Task>` of all the tasks that match the search term.
`void`	**`update`**`(int originalSerial, Task updated)` Updates an existing task in the database.

File Permissions

A private method `verifyFileWritingAbility()` exists to help check that the database has write permissions for the database file. It will throw the exceptions listed below.

Every public facing method that involves a change to the database content must call this method at the earliest opportunity before proceeding with its job. This is to ensure that the database operation can be performed successfully.

Exceptions

Any methods above that involve 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.
3. `java.util.NoSuchElementException` is thrown if the database cannot locate one or more tasks it is required to read/modify.

To preserve database integrity, the `HashMap<Integer, 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.

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))`

Important APIs

`void`	**`done(boolean newDoneStatus)`** Marks the task as done or undone.
`DateTime`	**`getDeadline(), getStartTime(), getEndTime()`** Gets the relevant times of the task. **If the field is not applicable, a public constant Task.INVALID_DATE_FIELD DateTime object will be returned.**
`int`	**`getSerial()`** Gets the serial number of the task, for the purposes of deletion and updating.
`String`	**`getTaskName()`**
`Task.TaskType`	**`getType()`** Returns the type of task.
`boolean`	**`isDone()`** Returns true if the task has been marked as done.
`boolean`	**`isFloatingTask(), isDeadlineTask(), isTimedTask()`** Returns true if the task is of the type requested.
`boolean`	**`clashesWithRange(DateTime startRange, DateTime endRange)`** Returns true if the time of the task overlaps the date range provided. Floating tasks will always return false to this.
`boolean`	**`containsTerm(String term)`** Returns true if the task contains the term provided. Case-insensitive.
`boolean`	**`isEqualTo(Task toCompare)`** Returns true if the given task toCompare has exactly the same attributes as the current Task. The serial number is also compared.
`String`	**`showInfo()`** Returns the entire contents of the Task object as a String. **For debugging and testing purposes only.**
`void`	**`becomeThis(Task updated)`** Copies all the attributes including the serial number from the *`updated`* Task object to *`this`* Task object.
`int`	**`compareTo(Task t)`** Compares the current task with another task. Ordering is by start time for timed tasks, and by deadline for deadline tasks. Floating tasks are always considered bigger than timed or deadline tasks. If the tasks have the same start time, they are ordered lexicographically by the task name. Returns 0 is they are equal, -1 if the current task is smaller than the task provided, and if it is larger.

4.4.2 SearchTerms

The `SearchTerms` object is a container for passing search terms. It is used by the list and search commands to pass search terms from Logic to Storage. It can also be used to pass search terms from Logic to UI (through the `LogicToUi` class) for display purposes – for example, to notify the user on the search terms used in the current view.

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

Attribute	Description
	If the attribute is set to true,
`completedTasks : boolean`	Only completed tasks will be matched
`incompleteTasks : boolean`	Only incomplete tasks will be matched
`timedTasks : boolean`	Only timed tasks will be matched
`deadlineTasks : boolean`	Only deadline tasks will be matched
`floatingTasks : boolean`	Only floating tasks will be matched
`startRange : DateTime` and `endRange : DateTime`	If specified, matches only tasks that overlaps the time period of `startRange` to `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)`

4.4.3 LogicToUi

The LogicToUi object is a container for passing results from the CommandHandlers to the UI. It is required because handlers will return a multitude of different things.

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

Attribute	Description
`output : String`	A message for the UI to display to the user
`display : List<Task>`	A list of tasks for the UI to display to the user
`filters : SearchTerms`	The search terms used to obtain the list of tasks
`currentSorting : SortStatus`	The parameter used for sorting the list of tasks
`sortReverse: boolean`	If true, indicates that sorting is in descending order
`lastChangedSerial : int`	The serial number for the last changed task. The UI can use this to highlight the last change made.

Every `LogicToUi` object is expected to carry a return message to be displayed by the UI after every command. This is enforced by the constructors. The other attributes are optional.

4.4.4 NattyParserWrapper

DoIt uses the open source date parser library Natty written primarily by Joe Stelmach.

Instantiating Natty may incur a noticeable delay. Since the Natty API is used many times, the delay can be frustrating. Therefore, the Natty library is enclosed in this wrapper as a singleton object. A dummy command is sent to it at the beginning to perform the instantiation only once.

We have made some modifications to this library. For example, the date format interpreted by Natty has been changed from the MM/DD/YY format to the DD/MM/YY format.

5 Testing

Testing is an integral part of the development of DoIt!. You are encouraged to employ test-driven development (TDD) whenever feasible. Automated testing is carried out using test cases based on the `JUnit4` framework. The entire code base must pass all test cases by the `AllTests` test suite before you commit to the repository.

These are the conventions set out for the development of DoIt!

Where to put my test files?	`/src/test/[Package Name]` [**Package Name]** has to be identical to the package of the class under test.
How to name test classes?	The test classes have to be named **`[Class Name]`**`Test`. For example, to test the Database class, name the test class `DatabaseTest`.
How to name test methods?	The test methods should ideally be named `test`**`[name of method under test]`**. However, this is not strictly mandated as testing of a particular behaviour may require several methods.
Package-level Test Suite	Every package has a Test Suite named **[Package name]**Tests which allows automated testing of the entire package. You should update this with any new test cases.
Main Test Suite	The root of the test directory contains the `AllTests` test suite which will run all of these package-level test suites. This will in effect, be testing everything. You should update this with any new package test suites.
Updates to test cases	You should update the test cases whenever a bug has been detected and fixed. Simply add your test case to the relevant test classes.
Exceptional Cases	1. Classes in the `Logic` package are tested differently. To avoid duplicate testing of the handlers/parsers and the `Logic` class itself, the test command should be sent directly to the `Logic.uiCommunicator()` method and the result retrieved from the `LogicToUI` object. 2. `FileManagementTest` is excluded from the suites and has to be run independently. This is due to the corruption test which will render the file unusable during the duration of the test run.

Possible Loss of Database data during and after tests
The text file database on disk may be irreversibly modified during the automated testing procedure. You should do a backup of the text file before the commencement of any test if you want to keep the contents of the database.

6 Known Issues

6.1 Command line issues

If the user has >= 1000 tasks, the CLI index column will shift. A 3 digit limit for the index number is hardcoded on the assumption that few users have 1000 tasks and to balance against the limited console width.

CliWithJline requires at least a terminal width of 67 characters. Display behaviour on consoles smaller than this minimum is not defined.

6.2 GUI issues

Agenda view can be quite slow if a very large number (thousands) of tasks are to be shown.

6.3 Keyboard shortcuts

Keyboard shortcuts are currently only available if DoIt! for the Windows platform. This feature is still not supported in Mac OS and Linux.

7 Future Work

7.1 Google Calendar and Tasks integration

DoIt! currently does not integrate with Google services. However, there exist some users that would prefer this functionality so they can use Google features like SMS and email reminders.

7.2 Password-protected database

The database file is stored in plaintext format. A password-lock with a suitable encryption scheme is envisioned to cater to more security conscious users.

7.3 Internationalisation

The interface of DoIt! and the commands it accepts is currently only limited to the English language. To cater to a greater group of users, support for more languages is intended.

8 Appendix

8.1 Help.xml sample

```
<?xml version="1.0"?>
<hint>
    <command>
        <name><![CDATA[<h1>Delete</h1>]]></name>
        <summary>Deletes task(s) from your schedule</summary>
        <usage><![CDATA[<b>delete</b> [index] (Deletes the task at index) ]]></usage>
        <usage><![CDATA[<b>delete done</b> (Delete all completed tasks) ]]></usage>
        <usage><![CDATA[<b>del</b> over (Delete all tasks before this moment) ]]></usage>
        <usage><![CDATA[<b>d</b> all (Deletes <b>EACH AND EVERY</b> task in your
schedule) ]]></usage>
    </command>
    <command>
        <name><![CDATA[<h1>Postpone</h1>]]></name>
        <summary>Postpones a task</summary>
        <usage><![CDATA[<b>postpone</b> [index] [time parameter] ]]></usage>
        <usage><![CDATA[<b>postpone</b> 1 3 hours (postpones task at index 1 by 3
hours) ]]></usage>
        <extra>This command has no effect on floating tasks. Start and End times for timed
tasks will shift together.</extra>
    </command>
</hint>
```

8.2 Code example for Undo Support

```
protected void updateDatabaseNSendToUndoStack()
        throws NoSuchElementException, IOException, WillNotWriteToCorruptFileException {

        List<Task> copyCurrentTaskList = super.getCurrentTaskList();
        dataBase.update(toBeDoneSerial, copy);
        String taskDetails = taskToString(copy);
        String undoMessage = "marking of task \"" + taskDetails + "\" as done";
        super.pushUndoStatMesNTaskList(undoMessage, copyCurrentTaskList);
}
```