

T A S U K E

The Developer’s guide

Table of Contents

[1. Preface 5](#_Toc383989054)

[ 1.1 Introduction to the *Tasuke* Developer Guide 5](#_Toc383989055)

[ 1.2 What is *Tasuke*? 5](#_Toc383989056)

[ 1.3 What must I know? 5](#_Toc383989057)

[ 1.4 Concept and Design Principles 6](#_Toc383989058)

[ 1.5 Conventions and Definitions 6](#_Toc383989059)

[ 1.6 Updates and Feedback 6](#_Toc383989060)

[2. Tasuke Build Environment 7](#_Toc383989061)

[ 2.1 Development Environment 7](#_Toc383989062)

[ 2.2 Execution Environment 7](#_Toc383989063)

[ 2.3 Application Files 8](#_Toc383989064)

[3. Software Architecture 9](#_Toc383989065)

[ 3.1 Software Architecture Overview 9](#_Toc383989066)

[ 3.2 Logic 10](#_Toc383989067)

[ 3.2 User Interface 11](#_Toc383989068)

[ 3.3 Interpreter 12](#_Toc383989069)

[ 3.4 Storage 13](#_Toc383989070)

[4. Developing The Application 14](#_Toc383989071)

[ 4.1 Add New Command 14](#_Toc383989072)

[5. Testing 16](#_Toc383989073)

[ 5.1 Testing Environment 16](#_Toc383989074)

[ 5.2 Sample Test – Setting Up 16](#_Toc383989075)

[ 5.3 Testing a Method 17](#_Toc383989076)

[6. future work 17](#_Toc383989077)

[7. Appendices 19](#_Toc383989078)

[ 7.1 Full Software Architecture Diagram 19](#_Toc383989079)

[ 7.2 Sequence Diagrams 20](#_Toc383989080)

[ 7.3 Application Programming Interface (API) 22](#_Toc383989081)

# 1. Preface

## 1.1 Introduction to the *Tasuke* Developer Guide

Welcome to the *Tasuke* Developer Guide! This guide is intended for developers and maintainers of *Tasuke*.

Here are the aims of the guide.

* To familiarize you with the design and implementation of *Tasuke*.
* To detail the organization of the software and its API.
* To provide solutions for future development.
* To assist you with troubleshooting the program.

## 1.2 What is *Tasuke*?

*Tasuke* is a desktop task manager aimed at individuals who are comfortable with keyboard-based commands for rapid data entry and retrieval. *Tasuke* will appeal to users who are familiar with the command-line-like style of calling and dismissing programs, and yet provides a simple but powerful GUI for clearer data organization and fine-tuning.

The basic functionality of *Tasuke* is as follows:

* Adding, editing and removing of tasks.
* Marking tasks as done or otherwise.
* Setting a starting time and deadline for tasks.
* Adding tags to tasks.
* Searching for tasks by date, completion and tags.
* Sorting the search result.

## 1.3 What must I know?

*Tasuke* is written in C++ using the Qt framework for the Windows platform. Hence, you should at least be familiar with C++. The design of the user interface of *Tasuke* requires knowledge of the QT framework.

## 1.4 Concept and Design Principles

*Tasuke* was developed with speed and simplicity as its goal. Keep these principles in mind when developing *Tasuke*!

* **Lightweight**: *Tasuke* is lightweight in terms of computer resource consumption.
* **Keyboard-based:** The most important functions of *Tasuke* must be keyboard-accessible.
* **Intuitive**: It is easy to learn to use. The command formats should follow natural language closely.
* **Unobtrusive**: *Tasuke* minimizes itself to the system tray when not needed.
* **Minimal**: The basic command-line interface of *Tasuke* should be simple.
* **Flexible**: Even though the basic interface is minimal, give users the option to add on features.

## 1.5 Conventions and Definitions

In this guide, a **task** refers to an object that users will create when they type the *add* command followed by a task description.

Whenever this guide mentions **Windows**, this refers to the default build and execution environment, which is Microsoft Windows 7 and newer iterations of the operating system.

All code, filenames and commands are written using the Courier New font. When describing methods and functions, the parameters are written in *Calibri Italic* and the return values are written in **Calibri Bold**.

A list of technical terms may be found in the Glossary.

## 1.6 Updates and Feedback

The latest version of this guide may be obtained when pulling the tip of the repository from Google Code at <http://code.google.com/p/cs2103jan2014-w15-2c/> or it may be downloaded from [https://code.google.com/p/cs2103jan2014-w15-2c/source/browse/[W15-2C][V0.0]DevGuide.docx](https://code.google.com/p/cs2103jan2014-w15-2c/source/browse/%5bW15-2C%5d%5bV0.0%5dDevGuide.docx)

Feedback may be provided by raising an issue on the aforementioned Google Code website.

# 2. Tasuke Build Environment

## 2.1 Development Environment

*Tasuke* is written in the C++ programming language, using the Qt 5.2.1 library. It is compiled and written in Microsoft Visual Studio 2012, on Microsoft Windows 7 and Microsoft Windows 8. We recommend you to use the same development environment to prevent incompatibility issues. You should **avoid** using the Express edition of Visual Studio, as it does **not** support the QT Framework. Additionally, make sure that you install the Visual Studio add-in 1.2.3 Alpha for Qt 5 as well.

It is, in theory, possible to compile the source code of *Tasuke* on any platform, for any platform. However, *Tasuke* is written with developers and users of Microsoft Windows 7 and later in mind. Thus, we will only discuss issues arising from Windows-compiled *Tasuke*. When we write this guide, we assume that you are using a Windows environment to develop and maintain *Tasuke*.

Qt 5.2.1 and its Visual Studio add-in may be downloaded from <http://qt-project.org/downloads>

Visual Studio 2012 is a proprietary IDE by Microsoft. License keys are needed to activate it. The disc image for Visual Studio installation may be found at   
<http://www.microsoft.com/en-sg/download/details.aspx?id=30678>

Logging in *Tasuke* is done with glog, a library that implements application-level logging.  
glog can be obtained together with a fresh pull of *Tasuke*, but should you find the need to get a fresh copy of glog, it can be found on <https://code.google.com/p/google-glog/>  
Note that *Tasuke* uses glog-0.3.3, which is the latest version at the time of this writing.

The spelling check library used in *Tasuke* is hunspell. *Tasuke* uses hunspell version 1.3.2, last updated 16 February 2011. A copy of hunspell can be obtained together with a fresh pull of *Tasuke*, but should you find the need to obtain a fresh copy of hunspell, it can be found on http://hunspell.sourceforge.net/

For information on the testing environment, refer to Section 4.

## 2.2 Execution Environment

The recommended execution environment for the default build of *Tasuke* is Windows 7 and newer. No additional framework or library needs to be installed, and *Tasuke* itself does not require installation and may be run straight from the executable binary.

During its operation, *Tasuke* will create an .ini file in the %APPDATA% directory, which is where *Tasuke* stores and retrieves user data.

## 2.3 Application Files

Tasuke.sln is the Visual Studio Solution file and should be the entry point when developing *Tasuke*. Open this using Microsoft Visual Studio to begin browsing the code for *Tasuke*.

A full list of the files in the project can be found in §6.4, at page 11.

# 3. Software Architecture

## 3.1 Software Architecture Overview

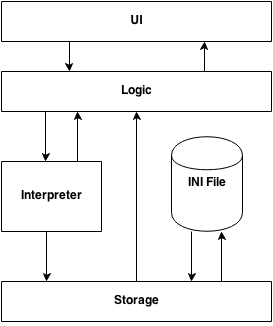


Figure 1 – High level logic of Tasuke

*For the full software architecture diagram, please go to Appendix 8.2.*

*Tasuke* uses four-layer architecture as shown above.

* The **UI layer** directly interacts with the user in the form of a graphical interface, and presents information to the user. It also fetches data from the user and passes it to Logic.
* The **Logic layer** handles user commands and manipulates data entered by the user.
* The **Interpreter layer** helps to organize the user command into a format that can be read by Logic and Storage.
* The **Storage layer** manages the file system which data is stored in. It also handles the INI File, where settings are stored.

For a full list of all methods provided by all the components, refer to the list of publicly accessible methods in section 7.3.

## 3.2 Logic

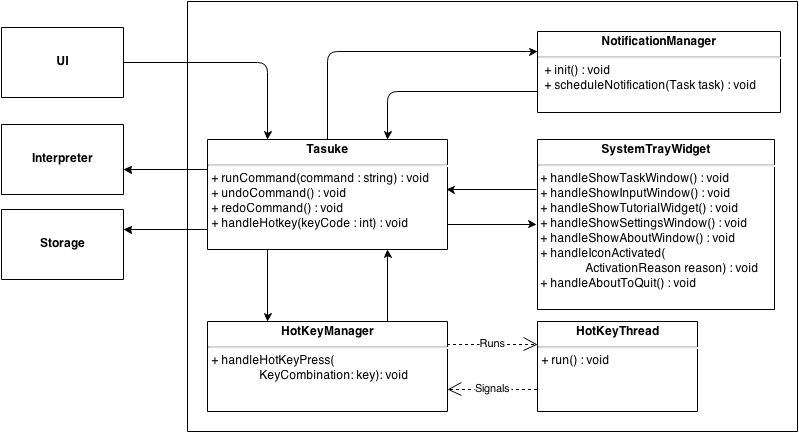


Figure 2 – Logic Subsystem

The **Logic** layer consists of the classes Tasuke, and HotKeyThread. Its main purpose is to maintain the state of the program, handle exceptions, and control the Interpreter and Storage.

* It calls **Interpreter** to interpret a string into a command, and then executes the command.
* It calls **Storage** to save or retrieve data. It also reacts to user actions by activating the UI classes in response to hotkey press.
* This layer also manages the HotKeyThread, which runs in the background to monitor for hotkey triggers. It reports any hotkey triggers to Tasuke, which then calls the appropriate UI window.

## 3.2 User Interface

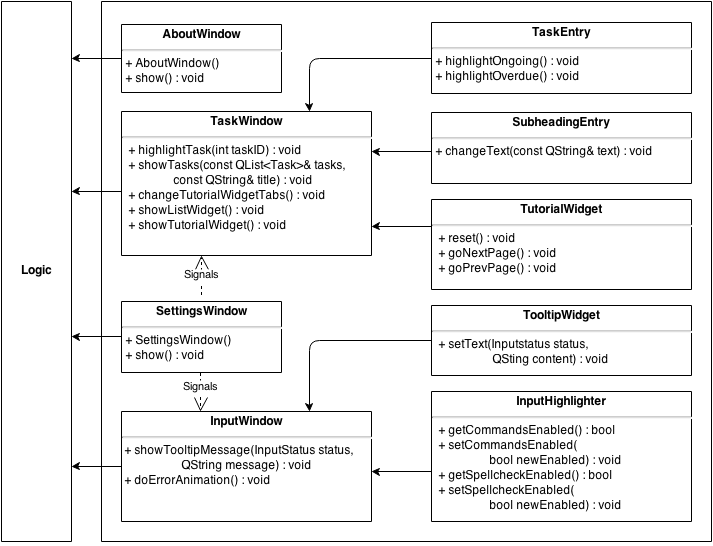


Figure 3 – User Interface

The **UI** layer consists of the classes InputWindow, SettingsWindow, AboutWindow, HelpWindow, and TaskWindow. Each class represents a window in *Tasuke*.

* InputWindow is where the command box resides. This component also passes user commands to **Logic**.
* TaskWindow, also our main window, displays the list of tasks to the user. The list is provided by **Logic**.
* SettingsWindow is the place users go to configure *Tasuke*.
* HelpWindow is opened by the user when they need help.
* AboutWindow provides brief information about *Tasuke* for the user.

## 3.3 Interpreter

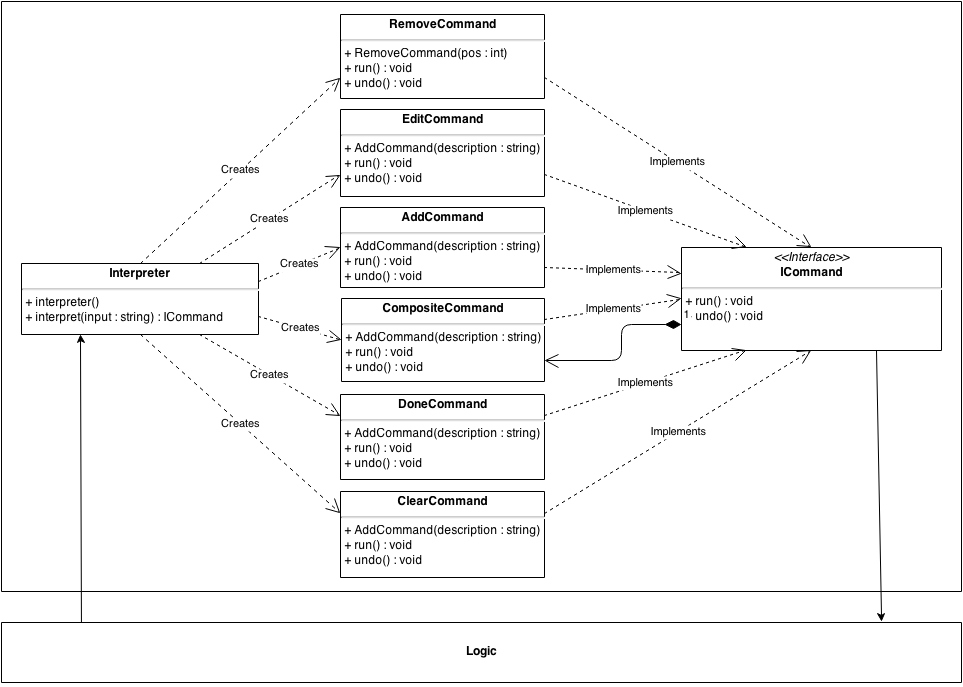


Figure 4 – Interpreter Subsystem

The **Interpreter** layer does most of the heavy lifting by interpreting the text that users enter into *Tasuke*, and translates them into a form that is understandable by Logic and Storage. It contains the functions that powers the fast, powerful and flexible command structure of *Tasuke*.

New commands can easily be added by creating another command that conforms to the ICommand interface in Commands class inside Commands.cpp.

## 3.4 Storage

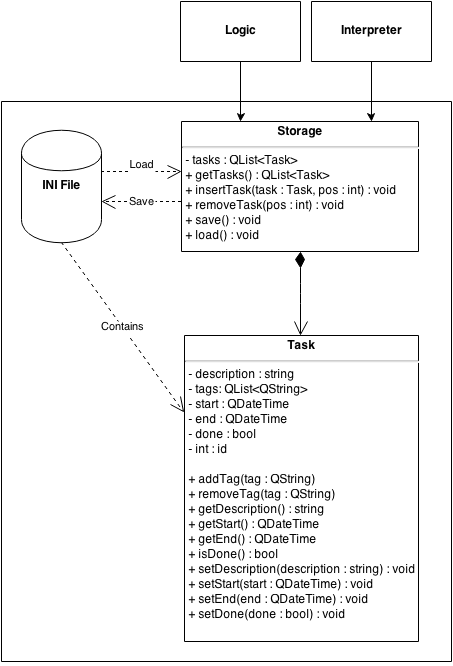


Figure 5 – Storage Subsystem

The **Storage** layer handles interactions with the file system that *Tasuke* resides in. its main purpose is to write to the .ini file, as well as to read from it. It is responsible for transforming plaintext to task objects, which it then passes to **Logic** through **Storage** for further action, and vice versa.

If you wish to implement a storage subsystem that implements another form of storage such as JSON or databases, please note that the new subsystem must be able to wrap and unwrap Task objects, and must be able to respond to all public methods specified in Storage.cpp.

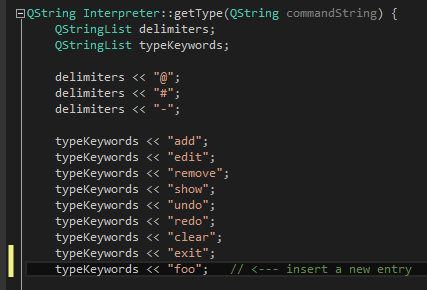
# 4. Developing The Application

This section will walk you through the process of writing code for adding a new command to *Tasuke*. This way, you will get a feel of how the major elements of *Tasuke* interact with each other. The command this guide is teaching will enable users to type the “foo” command that prints a “bar” in the UI.

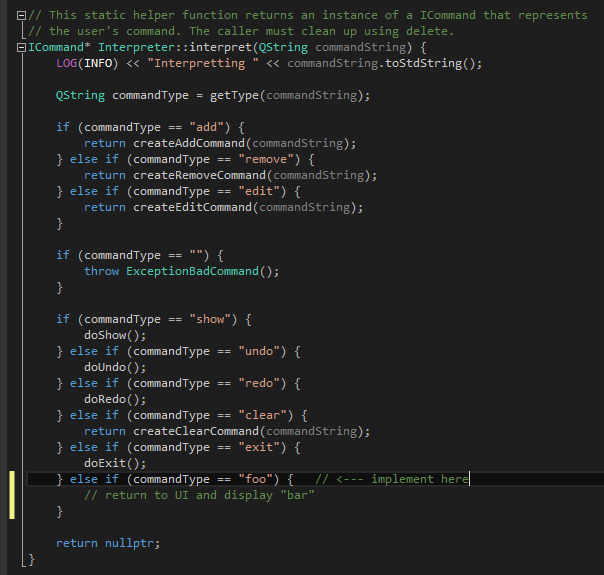
Foo is a one-off command; that is, trying to undo it does nothing since it merely posts a result to the user interface. However, for the sake of this exercise, we will pretend that foo is undoable.

## 4.1 Add New Command

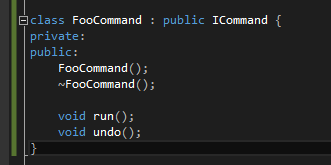
Firstly, you have to get *Tasuke* to recognize the command when it sees it. This is achieved by going into Interpreter.cpp and going to the Interpreter::getType() method, and adding a new typeKeywords entry:



Next, go to Interpreter::interpret() to add some logic when the command matches the user input and return a Command class:



Finally, for commands that is undoable, go to Commands.cpp and add a new entry there. Note that all undoable commands must implement the ICommand interface. This step should be skipped if the new command does not need undo().



Finally, in the concrete implementation of the methods, the run and undo methods should do what the name describes. run() is where the actual implementation of the command goes, while undo() is the reverse.

Remember to throw an ExceptionBadCommand if the user inputs the wrong syntax as well.

# 5. Testing

## 5.1 Testing Environment

The *Tasuke* testing framework is developed alongside *Tasuke*, therefore all development requirements and limitations will also apply to the testing framework. As with most programs written in C++, *Tasuke* uses the CppUnit Unit Testing Library for its unit testing.

## 5.2 Sample Test – Setting Up

The class that creates a test environment has already been set up. All test cases for a particular class should be written in a file named <ClassName>Tests.cpp. If you are testing a new method written in an existing class, skip to Section 4.3.

The test class should be set up similarly to TasukeTests.cpp, which we will use as a sample. Ensure that your new test class includes the same header files and uses the same namespace as that. It should also have the same namespace UnitTest. Replace references to the Tasuke class with the class you want to test as appropriate. Your test class should also have TEST\_MODULE\_INITIALIZE and TEST\_MODULE\_CLEANUP.

Note that StorageStub is for simulating a storage environment during testing without interfering with any actual stored user data written by the actual Storage class.

All method testing code should go under public within TEST\_CLASS().

## 5.3 Testing a Method

Adding a new method to test another method is fairly straightforward. In this example you will write a method that tests the previously written Foo command. The basic objective is to test that a method returns the right output when given a particular input.

TEST\_METHOD(TestFooCommand) {

}

# 6. future work

For future versions of *Tasuke*, here is what we plan to add:

* Users should be able to append tags to tasks whether during creation or after creation. Tasks can then be configured to display differently depending on whether or not it has a particular tag. The tags can also be used to indicate urgency and importance.
* Better natural language command emulation. We plan to do this by expanding the dictionary size used to detect an appropriate English-like command, such as synonyms for existing commands.
* User reminders. We plan to implement some form of notification to the users when a task is nearing its due time. This may be anything from a tray icon message to a pop-up. It may be disabled.
* Add a window where settings and configurations can be fine-tuned by users to their liking. This includes the possibility for users to substitute commands with custom ones. For example, users may replace the keyword for Add command with a word of his or her choice.

# 7. Appendices

## 7.1 Full Software Architecture Diagram

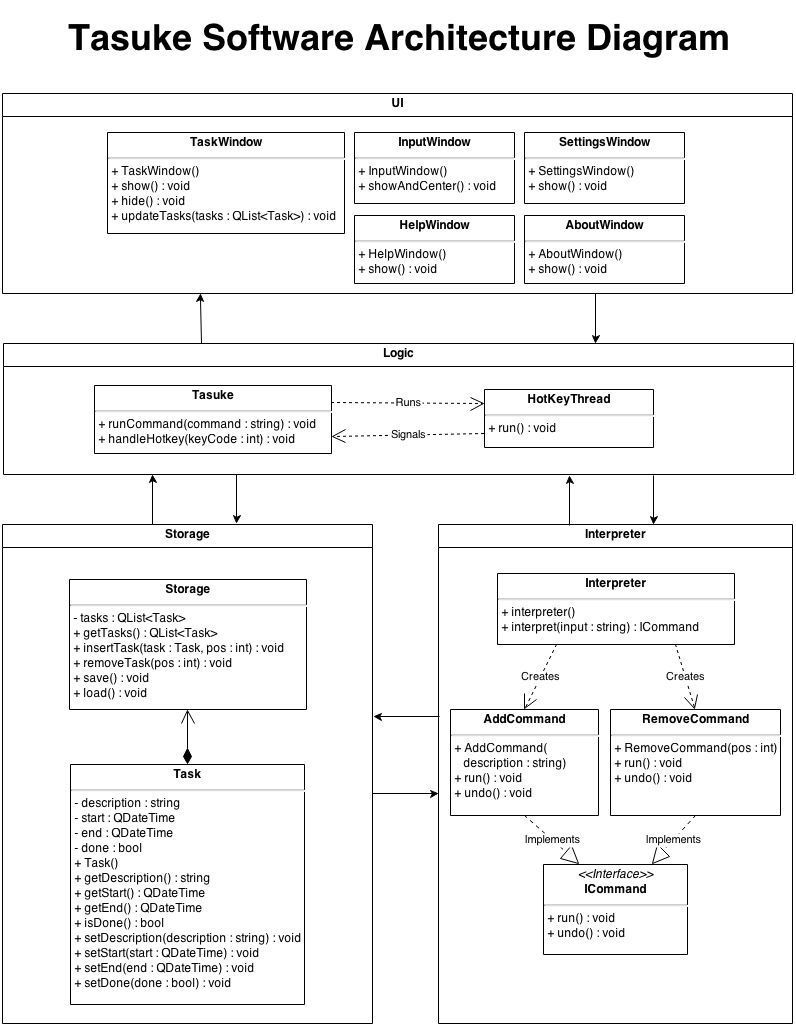


Figure 6 – Full Software Architecture Diagram

## 7.2 Sequence Diagrams

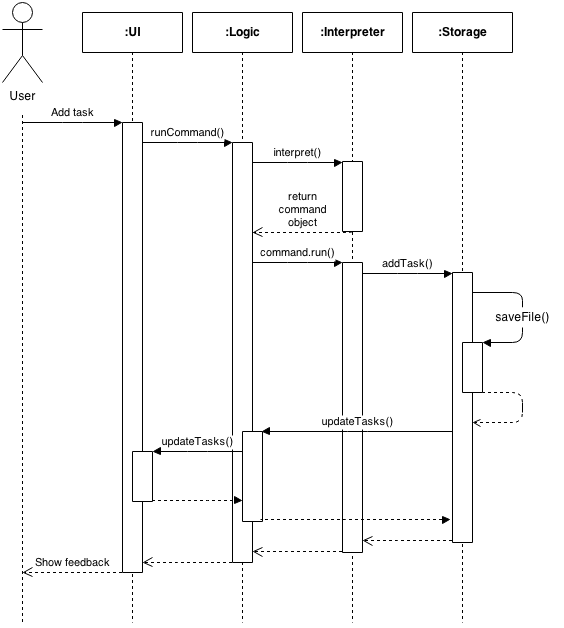


Figure 7 – Sequence Diagram for Add command

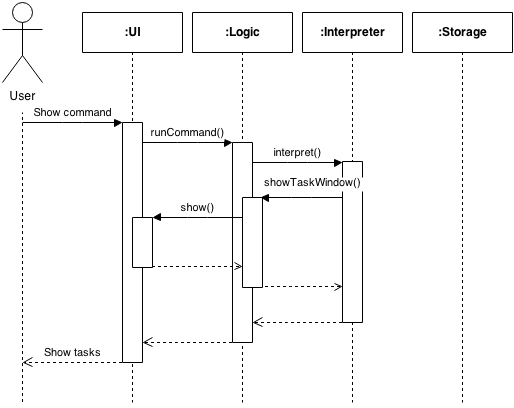


Figure 8 – Sequence Diagram for Show Command

## 7.3 Application Programming Interface (API)

This is a list of API for *Tasuke*. Only exposed, essential classes are displayed for brevity.  
  
Storage

|  |  |
| --- | --- |
| Modifier and Type | Function and Summary Description |
| void | loadFile()  Opens a stored .ini file at %appdata% and loads it to memory. |
| void | saveFile()  Writes the contents in memory to an .ini file in %appdata%. Append only. |
| void | addTask(Task& task)  Adds a single task to the list of tasks in memory. |
| void | addTask(Task& task, int pos)  Adds a single task to the list of tasks in memory at position pos. |
| Task& | getTask(int pos)  Returns a single Task at position pos in the list of tasks in memory. |
| void | removeTask(int pos)  Removes the task located at position pos in the list of tasks in memory. |
| Task& | popTask()  Returns the next task in the list of tasks in memory. |
| QList<Task> | getTasks()  Returns the whole list of tasks |
| int | totalTasks()  Returns the total number of tasks in the list of tasks in memory. |

tASK

|  |  |
| --- | --- |
| Modifier and Type | Function and Description |
| void | setDescription(QString& \_description)  Sets the description of a task to \_description. |
| QString const | getDescription()  Retrieves the description of the task. |
| void | addTag(QString& tag)  Appends a tag tag to the task. |
| void | removeTag(QString& tag)  Searches for tags that matches tag and removes it from the task. |
| QList<String> | getTags()  Retrieves all the tags of the task. |
| void | setBegin(QDateTime& \_begin) Sets or changes the starting time of a task to \_begin. |
| QDateTime const | getBegin()  Retrieves the starting time of the task. |
| void | setEnd(QDateTime& \_end)  Sets or changes the due time of a task to \_end. |
| QDateTime const | getEnd()  Retrieves the due time of the task. |
| void | setDone(bool \_done)  Sets the “done” status of the task to \_done. |
| void | markDone()  Sets the “done” status of the task to TRUE. |
| bool const | isDone()  Retrieves the “done” status of the task. |

Commands

|  |  |
| --- | --- |
| Modifier and Type | Function and Description |
| Constructor | AddCommand(Task& task) Constructs a command that has the ICommand interface. |
| Destructor | ~AddCommand() |
| void | run()  Executes the current command. |
| void | undo()  Undoes the command. |

* 7.4 Files within the Project

This is what the working directory should contain when it is freshly cloned from the repository, sorted by alphabetical order.

|  |  |  |
| --- | --- | --- |
| **Directory Root** | | |
| .hgIgnore  AssemblyDependencies.dgml  Tasuke.sln  [W15-2C][V0.1]DevGuide.docx | | |
|  | **Folder: Tasuke** | |
|  | **Folder: fonts** |
| PrintBold.otf  PrintClearly.otf  Quicksand\_Bold.otf  Quicksand\_Bold\_Oblique.otf  Quicksand\_Book.otf  Quicksand\_Book\_Oblique.otf  Quicksand\_Light.otf  Quicksand\_Light\_Oblique.otf |
| **.h Header Files** | |
| AboutWindow.h  CommandFactorry.h  Commands.h  Constants.h  Exceptions.h  HotKeyThread.h  InputWindow.h  Storage.h  Task.h  TaskEntry.  TaskWindow.h  Tasuke.h | |
| **.cpp Source Files** | |
| AboutWindow.cpp  CommandFactorry.cpp  Commands.cpp  Exceptions.cpp  HotKeyThread.cpp  InputWindow.cpp  main.cpp  Storage.cpp  Task.cpp  TaskEntry.cpp  TaskWindow.cpp  Tasuke.cpp | |
| **.ui UI Files** | |
| AboutWindow.ui  InputWindow.ui  SHAWindow.ui  TaskWindow.ui  TutorialWindow.ui | |
| **PNG Image Files** | |
| about.png  closebButton.png  closeButtoHover.png  InputWindowMask.png  minButton.png  minButtonHover.png  roundedEntryMask.png  roundedMask.png  TaskWindowBorderGrey.png  TaskWindowMaskWhite.png  Tasuke.png  Tasuke\_OLD.png  Traysuke.png | |
| **Misc** | |
| Resources.qrc  Tasuke.vcxproj  Tasuke.vcxproj.filters | |
|  | **Folder: UnitTests** | |
| stdafx.cpp  stdafx.h  StorageStub.cpp  StorageStub.h targetver.h TasukeTests.cpp UnitTests.vcxproj  UnitTests.vcxproj.filters | |
|  | Folder: glog-0.3.3 | |
| <omitted for brevity. An updated list of files that should be in this folder can be obtained by visiting the Google Code project page for glog: <https://code.google.com/p/google-glog/> > | |
|  | Folder: hunspell-1.3.2 | |
|  | <omitted for brevity. An updated list of files that should be in this folder can be obtained by visiting the Sourceforge project page for hunspell: http://hunspell.sourceforge.net/ > | |