

T A S U K E

The Developer’s guide

Table of Contents

[1. Preface 4](#_Toc385210720)

[ 1.1 Introduction to the *Tasuke* Developer Guide 4](#_Toc385210721)

[ 1.2 What is *Tasuke*? 4](#_Toc385210722)

[ 1.3 What must I know? 4](#_Toc385210723)

[ 1.4 Concept and Design Considerations 5](#_Toc385210724)

[ 1.5 Conventions and Definitions 5](#_Toc385210725)

[ 1.6 Updates and Feedback 5](#_Toc385210726)

[2. Tasuke Build Environment 6](#_Toc385210727)

[ 2.1 Development Environment 6](#_Toc385210728)

[ 2.2 Execution Environment 6](#_Toc385210729)

[ 2.3 Application Files 7](#_Toc385210730)

[3. Software Architecture 7](#_Toc385210731)

[ 3.1 Software Architecture Overview 7](#_Toc385210732)

[ 3.2 Design Principles and Patterns 8](#_Toc385210733)

[ 3.3 Logic 10](#_Toc385210734)

[ 3.4 User Interface 11](#_Toc385210735)

[ 3.5 Interpreter 12](#_Toc385210736)

[ 3.6 Storage 13](#_Toc385210737)

[4. Developing The Application 14](#_Toc385210738)

[ 4.1 Add New Command 14](#_Toc385210739)

[ 4.2 Add New Storage Format 17](#_Toc385210740)

[5. Testing 18](#_Toc385210741)

[ 5.1 Testing Environment 18](#_Toc385210742)

[ 5.2 Sample Test – Setting Up 18](#_Toc385210743)

[ 5.3 Testing a Method 18](#_Toc385210744)

[ 5.4 Running the Tests 19](#_Toc385210745)

[6. Future work and Known Issues 20](#_Toc385210746)

[7. Appendices 21](#_Toc385210747)

[ 7.1 Application Programming Interface (API) 21](#_Toc385210748)

[ 7.2 Files within the Project 26](#_Toc385210749)

# 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.

## 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 Considerations

*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

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 and glog libraries. 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.

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.

|  |  |
| --- | --- |
| Recommended tool(s) | Where to get this |
| Qt 5.2.1 for Windows 32-bit (VS 2012)  and  Visual Studio Add-in 1.2.2 for Qt5 | <http://qt-project.org/downloads> |
| Visual Studio 2012 (needs license key) | <http://www.microsoft.com/en-sg/download/details.aspx?id=30678> |
| glog-0.3.3 for logging | Simply pull from Tasuke’s repository or download it from <https://code.google.com/p/google-glog/> |
| hunspell 1.3.2 for spell checking | Simply pull from Tasuke’s repository or download it from  <http://hunspell.sourceforge.net/> |

For information on the testing environment, refer to *Section 5 Testing*.

## 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 partial list of the important files in the project can be found in Section 7.3.

# 3. Software Architecture

## 3.1 Software Architecture Overview

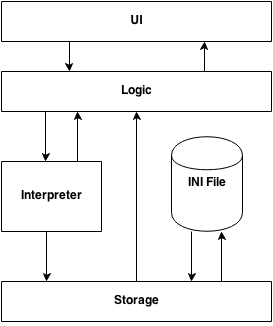


Figure 1 – High level logic of Tasuke

*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.

## 3.2 Design Principles and Patterns

The software architecture of *Tasuke* employs the following design principles and patterns. It is strongly recommended that developers conform to these design principles and patterns when writing code, for ease of maintenance, development and debugging.

**Single Responsibility Principle**

Each class in *Tasuke* has exactly one responsibility. Source files in the project should contain only one class, with the exception of interface classes such as IStorage and ICommand. For the user interface, each window and widget should be contained within its own class. Additionally, each thread created should be managed by a class with exactly one responsibility.

**Liskov Substitution Principle**

Each command class that extends ICommand should not have any more methods than the run() and undo() methods that already exist in ICommand. All command implementations should go into those those two methods, so that each command can be substituted for with ICommand.

**Open-Close Principle**

Unless there is a bug in the original source code, developers and maintainers should not alter the source code.

Alternative storage methods can be achieved by extending the IStorage interface. Developers need only to implement a different way of saveFile() and loadFile().

New commands can be added by extending the ICommand interface. Developers need only to implement run() and undo(), and add an entry in the Interpreter class.

For more details on how to add a new command or storage option, see Chapter 4.

**Singleton Pattern**

Tasuke and Storage are classes that must only have exactly one instance at any time. Tasuke coordinates all windows and commands, while keeping track of alarms and signals. It also ensures that only one instance of *Tasuke* runs at any given time.

Storage interfaces with the filesystem of the host operating system, and there should not be more than one instance of Storage active to prevent read/write errors.

**Model-View-Controller (MVC) Pattern**

This pattern is used to implement Tasuke’s view of the task list in TaskWindow. The *model*, a Task, contains the data of a single task. This data is displayed in the form of a TaskEntry (the *view*) inside TaskWindow. The *controller* that manipulates the data in a Task is the Logic component.

**Command Pattern**

The commands AddCommand, RemoveCommand, and EditCommand exemplify the Command Pattern in Tasuke. The Tasuke class (Logic subsection) is the client of each of these commands, InputWindow is the invoker, and Storage and UI subsections are receivers.

**Façade Pattern**

*Tasuke* makes use of the façade pattern. The Logic subsection is the façade through which all other components interact. All input received from the User Interface should be passed to logic for handling. All task objects produced in the Storage subsection is passed to Logic for processing, and Logic makes use of the Interpreter subsection for deciding what to do. Each subsection does not communicate with other subsections directly unless through it is done through Logic.

## 3.3 Logic

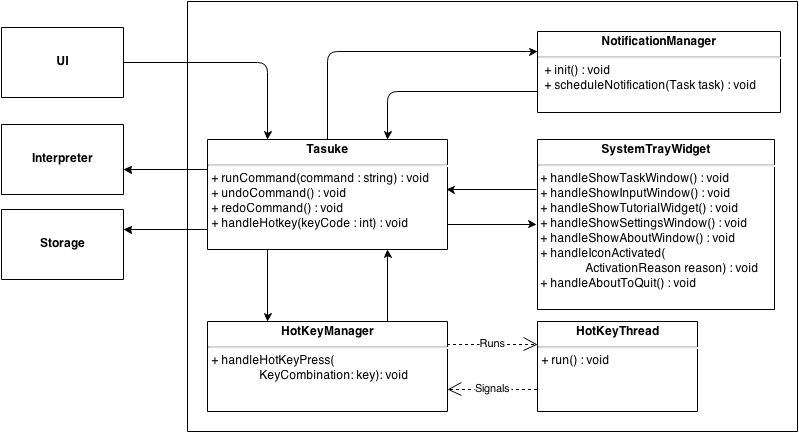


Figure 2 – Logic Subsystem

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 via HotKeyManager, which runs in the background to monitor for hotkey triggers. It reports any hotkey triggers to Tasuke, which then calls the appropriate UI window.
* Tasuke also maintains the SystemTrayWidget, which is responsible for the tray icon and the functions associated with it.
* Finally, Tasuke maintains NotificationManager which triggers the tooltip icon whenever a task is near its starting time.

## 3.4 User Interface

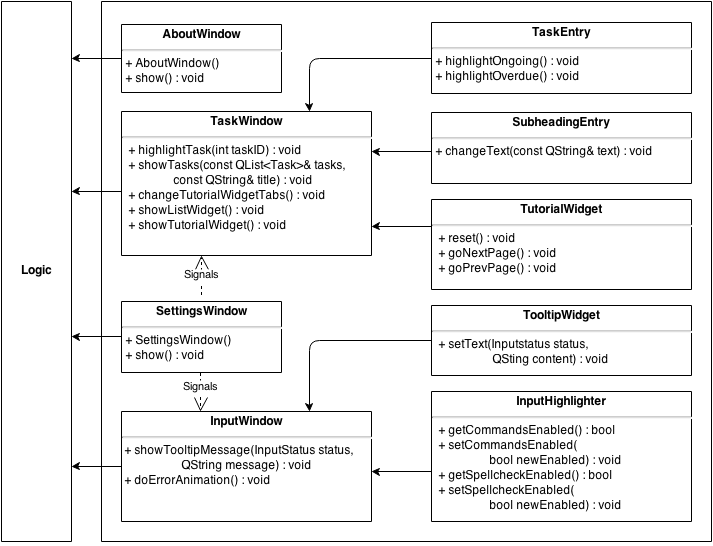


Figure 3 – User Interface

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

* InputWindow is where the command box resides. This component also passes user commands to **Logic**. InputWindow maintains the InputHighlighter which highlights specified keywords in text input, as well as TooltipWidget which displays the feedback guides under the command box.
* TaskWindow, also our main window, displays the list of tasks to the user. The list is provided by **Logic**. The TutorialWidget, maintained by TaskWindow, is shown on TaskWindow when the user enters ‘help’ as a command.
* TaskWindow also maintains TaskEntry and SubheadingEntry, whose objects are entries of the task list inside TaskWindow.
* SettingsWindow is the place users go to configure *Tasuke*.
* AboutWindow lists resource credits and brief information about *Tasuke*.

## 3.5 Interpreter

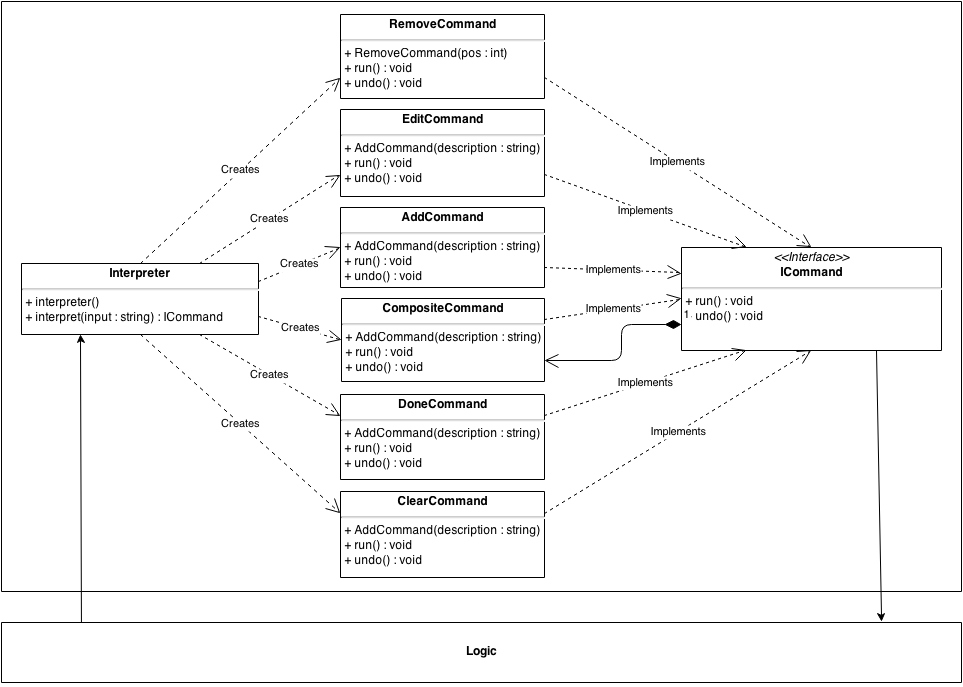


Figure 4 – Interpreter Subsystem

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 power 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. New commands should have a name that explains its purpose.

CompositeCommand is a special case. It is not only an implementation of ICommand, but it also contains one or more ICommand within itself.

## 3.6 Storage

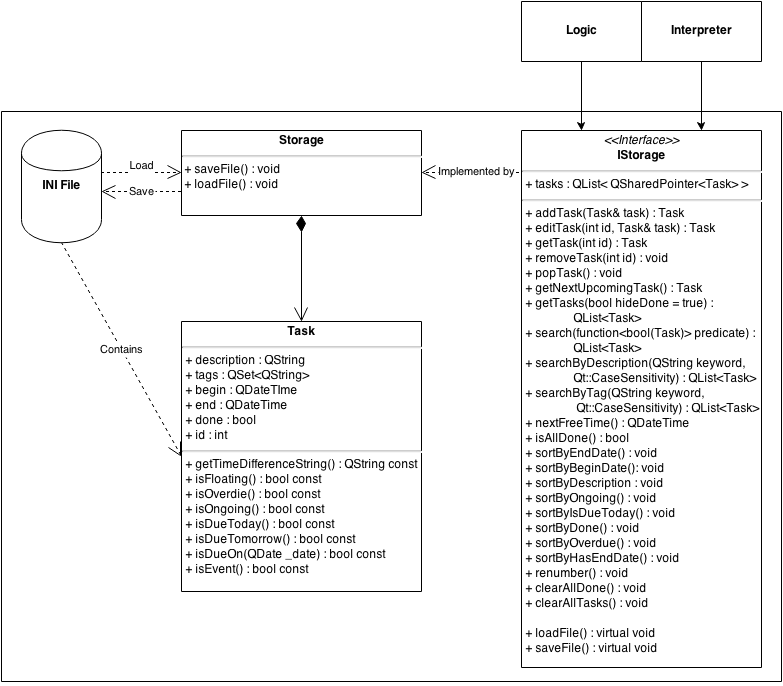


Figure 5 – Storage Subsystem

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 serialize and deserialize Task objects, and must be able to respond to all public methods specified in Storage.cpp.

# 4. Developing The Application

## 4.1 Add New Command

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. First, let’s try to understand how commands are executed.

This is the sequence diagram for the add command.

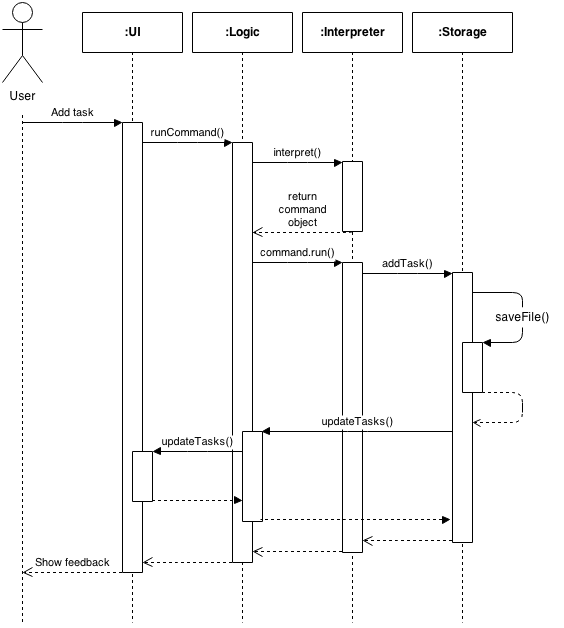


Figure 6 – Sequence Diagram for Add command

And this is for the show command.

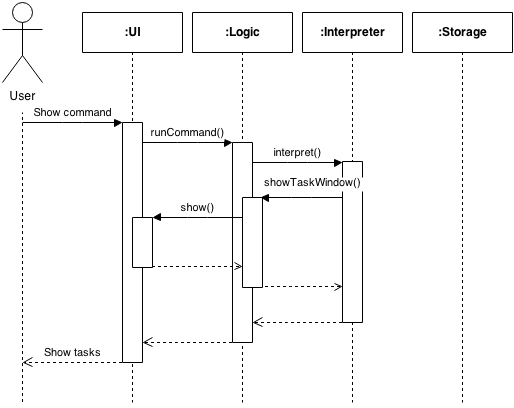


Figure 7 – Sequence Diagram for Show Command

As you can see from the diagrams above, **Logic** will pass the command to **Interpreter** to identify and carry out the command. **Storage** will save any manipulation to the data.

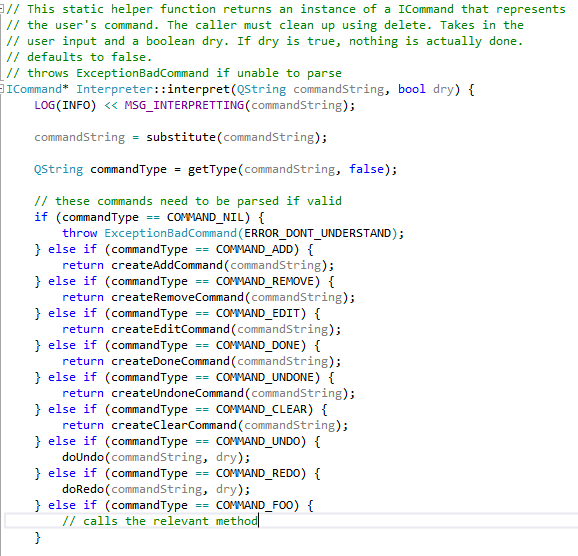
The command this guide is teaching will enable users to type the “foo” command that prints a “bar” in the UI.

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.

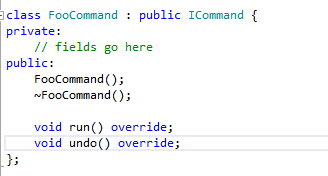
Firstly, you have to get *Tasuke* to recognize the command when it sees it. This is achieved by going into Constants.h and going to the Interpreter::getType() method, and adding a new typeKeywords entry:



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 are undoable, go to Commands.h and add a new entry there. Then go to Commands.cpp and write your implementation. Note that all undoable commands must implement the ICommand interface. This step should be skipped if the new command does not need undo().



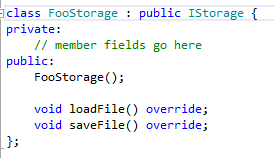
Finally, in the concrete implementation of the methods, the run and undo methods should do what the name describes. run() is where the actual implementation of the command goes, while undo() is the reverse.

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

## 4.2 Add New Storage Format

This section will walk you through the code to add a new storage format for *Tasuke*. Currently, *Tasuke* supports only storing of user data in .ini files. However, sometimes you may wish to save the file in a JSON-compatible format, for example. This section will guide you on writing a new way of storing user data called FooStorage.

Firstly, go to Storage.h and write a new storage class that extends the IStorage interface:



Next, go to Storage.cpp and write your implementation there. It is highly encouraged that your new storage class employs existing methods from IStorage rather than writing your own to ensure compatibility. Your new storage class simply needs to provide a way for IStorage to read from, and write to a file on the local disk, or over the Internet, and IStorage will handle the rest.

# 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. *Tasuke* uses the **CppUnit Unit Testing Library** for its unit testing. Visual Studio 2012 and newer contains native unit testing functionality which we make use of.

## 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

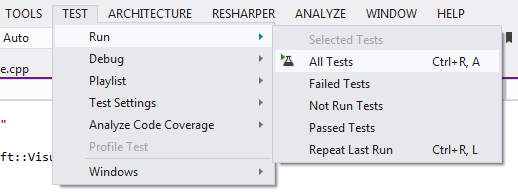
Adding a new method to test another method is fairly straightforward. In this example you will write a method that tests the previously written Foo command. The basic objective is to test that a method returns the right output when given a particular input.

If you need to simulate a command entry, use Tasuke::instance().runCommand(“command”); to simulate a command as though it were typed from the user interface.

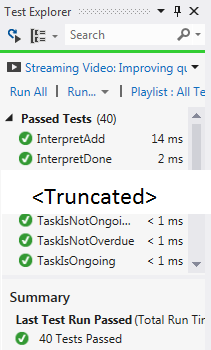
All tests should be written with Asserts provided for by the namespace, and all tests should be given a name that describes the purpose of the task, prefixed with the name of the class being tested.

## 5.4 Running the Tests

To run the tests after you have written them, go to Test -> Run -> All Tests or simply press Ctrl+R followed by the A key.



If the tests pass, you should see this:



And if any of the tests fail, the green bar will change to red and the right panel will notify you of the tests that failed.

# 6. Future work and Known Issues

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

* Support for Linux and other UNIX-like operating systems.
* Run commands in other threads, in order to prevent UI event loops from being blocked.
* Date parser should be run in a new thread, so that it is faster.
* Compile dictionaries right into *Tasuke* so everything is in one neat executable.
* Enhance support for more intelligent datetime inputs.
* Automatic updates of the *Tasuke* program.
* Animate the main Task window when new entries are added or updated for better feedback
* A timeline view so that users can easily see the current organization of their tasks.
* Priority rating system for tasks.
* Allow users to set a custom background image within the main Task window.
* Support for tasks that repeat on a regular basis.

These are the known issues for the current version of *Tasuke*:

* Commands must be typed in lowercase. This is because parsing support is case sensitive.
* Settings must make use of a mouse despite the keyboard-only philosophy if the design. This may be corrected in future versions when fields in the Settings window can be selected with the tab key.
* Users have to hover over the task object with a mouse in the main Task window in order for the tooltip with detailed information to show up. This may be corrected in future versions when the tooltip will show up via selection from the keyboard.
* Tooltip must be disabled on Mac, otherwise the input box is blocked.

# 7. Appendices

## 7.1 Application Programming Interface (API)

This is a list of API for *Tasuke*. Only exposed, essential classes, with methods that can be expanded upon for custom builds, are displayed for brevity.

INTERPRETER

|  |  |
| --- | --- |
| **static void** | setLast(int \_last)  Sets the task ID of the last task to \_last. |
| **static QString** | getType(QString commandString, bool doSub = true)  This method tries to guess the type of the command from commandString. If it cannot be determined it returns COMMAND\_NIL. This method also defaults to perform substitution, but can be disabled |
| **static ICommand\*** | interpret(QString commandString, bool dry = false)  This static helper function returns an instance of an ICommand that represents the user’s command. The caller must clean up using delete. If the Boolean dry is true, there is no action taken, and defaults to false. This method throws ExceptionBadCommand if it cannot parse the input. |
| **static void** | initFormats()  This method will generate all date formats.  WARNING: This method blocks for a very long time. Run this method on another thread at startup if this method is thread-safe. |

IStorage

|  |  |
| --- | --- |
| Modifier and Type | Function and Summary Description |
| void | addTask(Task& task)  Adds a single task to the list of tasks in memory. |
| void | editTask(int id, Task& task)  Replaces the task at id with task. |
| Task& | getTask(int id)  Returns a single Task with identification id in the list of tasks in memory. |
| void | removeTask(int id)  Removes the task with identification id in the list of tasks in memory. |
| void | popTask()  Removes the next task in the list of tasks in memory. |
| Task | getNextUpcomingTask()  Gets the next task in the list in memory. |
| QList<Task> const | getTasks(bool hideDone = true)  Gets a list of tasks and defaults to skipping tasks which are done. |
| int | totalTasks()  Returns the total number of tasks in memory. |
| QList<Task> const | search(std::function<bool(Task)> predicate)  Searches the list of tasks in memory for those with a particular status specified in function. |
| QList<Task> const | searchByDescription(QString keyword, Qt::CaseSensitivity = Qt::CaseSensitive)  Searches the list of tasks in memory and returns a list of tasks that contains the keyword in its description. Case sensitivity optional. |
| QList<Task>  const | searchByTag(QString keyword, Qt::CaseSensitivity = Qt::CaseSensitive)  Searches the list of tasks in memory and returns a list of tasks that contains the keyword in its tag. Case sensitivity optional. |
| QDateTime | nextFreeTime()  Returns the next unallocated time. |
| bool | isAllDone()  Returns TRUE if all tasks in memory are done. |
| void | sortByEndDate()  Sorts the list of tasks in memory by the existence of an end date-time, with the earliest ones at the front and the latest ones at the back. |
| void | sortByBeginDate()  Sorts the list of tasks in memory by the existence of a begin date-time, with the earliest ones at the front and the latest ones at the back. |
| void | sortByDescription()  Sorts the list of tasks in memory alphabetically by their descriptions. |
| void | sortByOngoing()  Sorts the list of tasks in memory by its ongoing status. Tasks which are ongoing are at the front. |
| void | sortByIsDueToday()  Sorts the list of tasks in memory by whether or not it is due on the current day. Those that are will be sorted to the front. |
| void | sortByDone()  Sorts the list of tasks in memory by whether or not it is done. Tasks which are done are sorted to the back. |
| void | sortByOverdue()  Sorts the list of tasks in memory by whether or not it is overdue. Tasks which are overdue will be sorted to the front. |
| void | sortByHasEndDate()  Sorts the list of tasks in memory by the existence of an end date-time. This method produces a different sort than sortByEndDate(). Tasks with no end date-time are sorted to the back. |
| void | renumber()  Enforces a default sorting pattern for the list of tasks in memory, and renumbers the ID of each task. |
| void | clearAllDone()  Removes all done tasks in memory. |
| void | clearAllTasks()  Removes all tasks in memory without discrimination. |
| virtual void | loadFile() = 0  Virtual method for opening a file, reading its contents and serializing the data into Task objects for storage in memory. |
| virtual void | saveFile() = 0  Virtual method for opening a file, deserializing the Task objects in memory into human-readable text and writing it to the file. |

tASK

|  |  |
| --- | --- |
| Modifier and Type | Function and Description |
| Constructor | Task(QString \_description)  Constructs a task object with a 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. |
| bool | removeTag(QString& tag)  Searches for tags that matches tag and removes it from the task. Returns true if a tag was removed, false if otherwise. |
| QList<String> | getTags()  Retrieves all the tags of the task. Tags in task are stored as a QSet, thus the QSet is transformed into a QList before returning to the caller. |
| QSet<QString> | getTagsSet()  Retrieves all tags of the task, without changes. |
| void | setBegin(QDateTime \_begin) Sets or changes the starting date-time of a task to \_begin. It is the caller’s responsibility to ensure that the date-time passed in is complete and valid, as this method makes no assumptions about the date-time. |
| void | setBeginDate(QDate \_date)  Sets or changes the starting date of a task to \_date. If the resulting date-time is invalid, this method will append the time as 12 a.m. of that day. |
| void | setBeginTime(QTime \_time)  Sets or changes the starting time of a task to \_time. If the resulting date-time is invalid, this method will append the date as the current day. |
| QDateTime const | getBegin()  Retrieves the starting time of the task. |
| void | setEnd(QDateTime& \_end)  Sets or changes the ending date-time of a task to \_begin. It is the caller’s responsibility to ensure that the date-time passed in is complete and valid, as this method makes no assumptions about the date-time. |
| void | setEndDate(QDate \_date)  Sets or changes the ending date of a task to \_date. If the resulting date-time is invalid, this method will append the time as 12 a.m. of that day. |
| void | setEndTime(QTime\_time)  Sets or changes the ending time of a task to \_time. If the resulting date-time is invalid, this method will append the date as the current day. |
| QDateTime const | getEnd()  Retrieves the due time of the task. |
| QString const | getTimeDifferenceString()  Returns the QString representation of a countdown from now to the end of a task. This method also returns the countup if the task has already expired. This method does NOT guarantee the accuracy of dates, and assumes that there are 12 identical months in a year, 4 week in each month, and each month has exactly 30 days. |
| void | setDone(bool \_done)  Sets the “done” status of the task to \_done. |
| void | markDone()  Sets the “done” status of the task to TRUE. |
| void | markUndone()  Sets the “done” status of the task to FALSE. |
| bool const | isDone()  Retrieves the “done” status of the task. |
| void | setId(int \_id)  Sets the list ID of this task to \_id.  This ID is used in sorting and display of tasks. |
| int const | getId()  Retrieves the ID of this task. |
| bool const | isFloating()  Returns TRUE if task does not have a valid start AND end date-time, and false if otherwise. |
| bool const | isOverdue()  Returns FALSE if there is no end date-time for this task, or it is not valid, or it is not overdue, and true if otherwise. |
| bool const | isOngoing()  Returns TRUE if the task has a start date-time earlier than the current date-time, and is not overdue. Returns FALSE if task is overdue, has no valid start date-time, or has not yet started. |
| bool const | isDueOn(QDate \_date)  Returns FALSE if task has no valid end date-time. Returns TRUE if task is overdue, and has an end date-time within the specified date. |
| bool const | isDueToday()  Returns FALSE if this task has no valid end date, is already overdue, or has an end date-time that does not fall within the current day, and TRUE if otherwise. |
| bool const | isDueTomorrow()  Returns FALSE if this task has no valid end date, is already overdue, or has an end date-time that does not fall within the current day plus one day, and TRUE if otherwise. |
| bool const | isEvent()  Returns TRUE if task has a valid start and end date-time, and false if otherwise. |

ICommand (INTERFACE)

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

Exceptions

These are the custom exceptions that have been defined.

|  |
| --- |
| ExceptionNullPtr : public std::Exception  This exception is thrown only during programming errors. If this happens, then it indicates a bug in the code. |
| ExceptionBadCommand : public std::Exception  This exception is thrown when an invalid command is entered by the user. |
| ExceptionNotImplemented : public std::Exception  This exception is used as a scaffolding placeholder for when the user tries to access a feature still under testing and should normally be hidden. |
| ExceptionNoMoreTasks : public std::Exception  This exception is thrown when the user requests for tasks when there is no more. |
| ExceptionIconsetOutOfRange: public std::Exception  This exception is used when the Icon Set stored in settings is corrupt and out of range. |
| ExceptionThemeOutOfRange: public std::Exception  This exception is used when the Theme stored in settings is corrupt and out of range. |

## 7.2 Files within the Project

This is what the working directory should contain when it is freshly cloned from the repository, sorted by alphabetical order. Do note that only the source files, headers and UI files are included for brevity; folders for the images and fonts used in the application are not listed due to the volume.

|  |  |
| --- | --- |
| **Directory Root** | |
| .hgIgnore  AssemblyDependencies.dgml  Tasuke.sln  [W15-2C][V0.5]DevGuide.docx | |
|  | **Folder: Tasuke** |
| **.h Header Files** |
| AboutWindow.h  Commands.h  Constants.h  Exceptions.h  HotKeyManager.h  HotKeyThread.h  InputHighlighter.h  InputWindow.h  Interpreter.h  MacWindowActivator.h  NotificationManager.h  SettingsWindow.h  SlidingStackedWidget.h  Storage.h  SubHeadingEntry.h  SystemTrayWidget.h  Task.h  TaskEntry.h  TaskWindow.h  Tasuke.h  ThemeStyleSheets.h  TooltipWidget.h  TutorialWidget.h |
| **.cpp Source Files** |
| AboutWindow.cpp  Commands.cpp  Exceptions.cpp  HotKeyManager.cpp  HotKeyThread.cpp  InputHighlighter.cpp  InputWindow.cpp  Interpreter.cpp  main.cpp  NotificationManager.cpp  SettingsWindow.cpp  SlidingStackedWidget.cpp  Storage.cpp  SubheadingEntry.cpp  SystemTrayWidget.cpp  Task.cpp  TaskEntry.cpp  TaskWindow.cpp  Tasuke.cpp  ThemeStyleSheets.cpp  TooltipWidget.cpp  TutorialWidget.cpp |
| **.ui UI Files** |
| AboutWindow.ui  InputWindow.ui  SHAWindow.ui  TaskWindow.ui  TutorialWindow.ui |
| **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/ > |