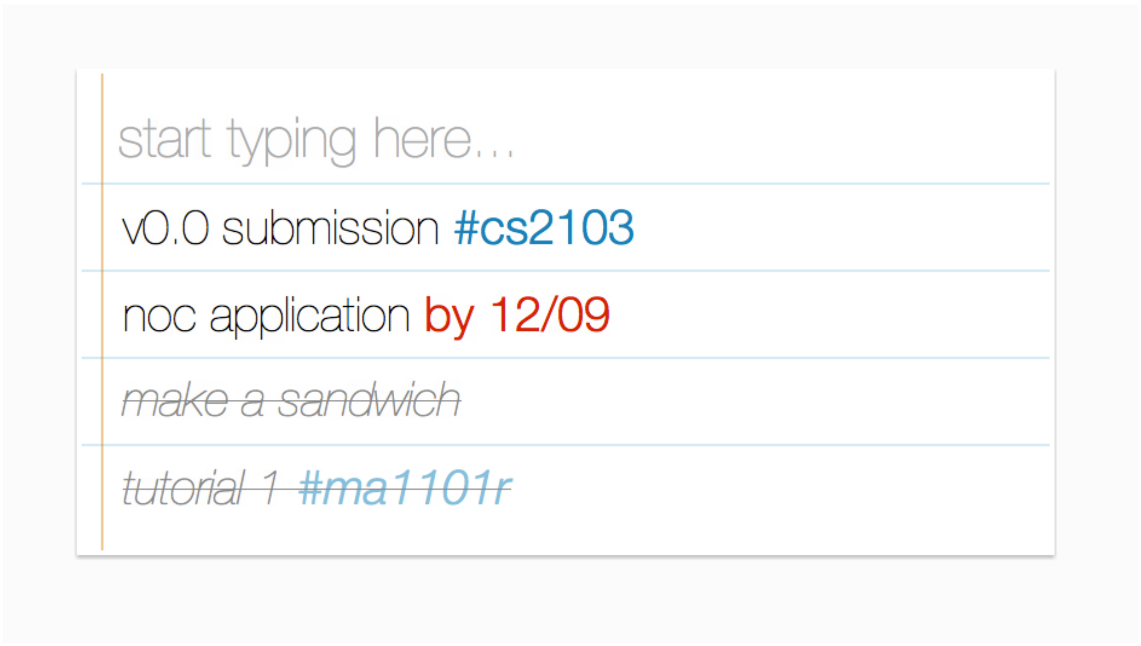
type. *Y*our *P*roductivity *E*nhancer

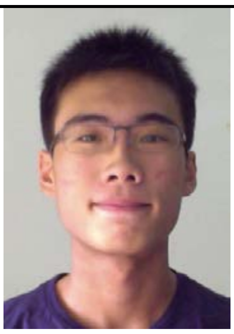

start typing here...

v0.0 submission #cs2103

noc application by 12/09

~~make a sandwich~~

~~tutorial 1 #ma1101r~~

			
Camillus Gerard Cai *Controller, System and Integration Testing*	**Michael Yong** *Model, Presentation Logic and Language Processing*	**Ang Civics** *Data Model and Storage*	**Yan Rong** *Presenter, User Experience and Design*

User Guide

Introduction

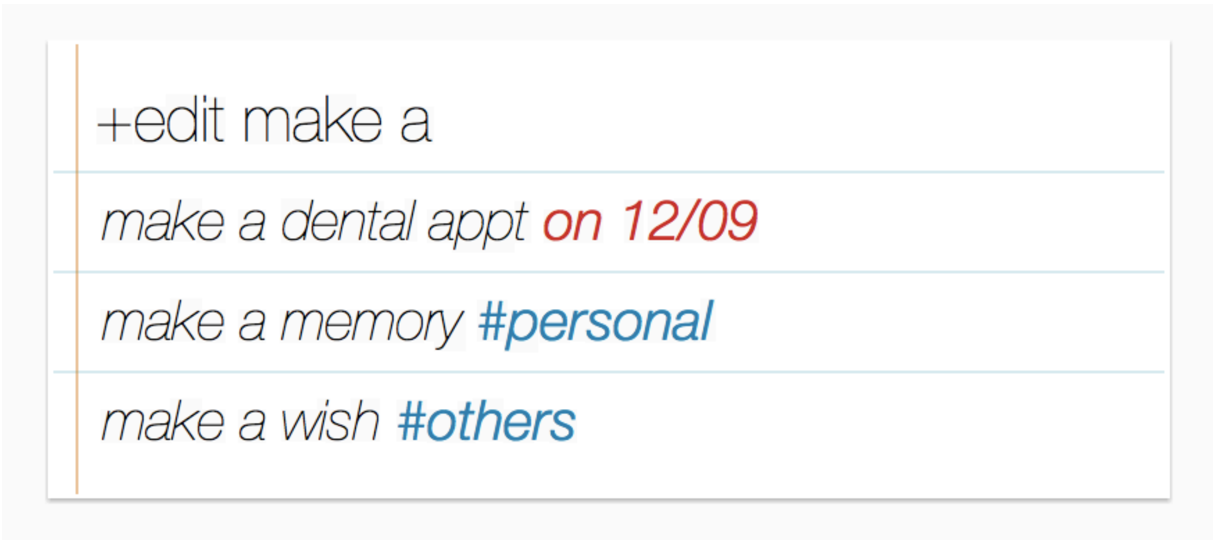
type breaks down all barriers to organizing tasks – it gets out of your way and allows you to focus on what you have to do.

To begin, you simply start typing!

Getting Started

Pressing `Shift + Space` will show type onscreen.

Smart algorithms intelligently predict your command before you finish it. type continuously shows suggestions as you are typing. The more of a command you type, the more accurate suggestions become.



+edit make a

make a dental appt on 12/09

make a memory #personal

make a wish #others

Pressing `Tab` will try to auto-complete your command, task, or #tag. If there are multiple choices, suggestions instantly appear in the list. You can navigate and select these suggestions using the up/down arrow keys, or continue typing to narrow down the possibilities. When there is only a single choice, pressing `Enter/Return or Tab` will select the suggestion.

type keeps a journal of all your tasks to date. There is no need to delete anything. Archiving is functionally similar to deleting in permanently hiding completed or unwanted tasks, but still allows you to find them should you ever need to.

Pressing `Esc` will hide type but it will still be running in the background.

Your Task List – Reimagined

Active tasks are displayed in a list. Active tasks include all tasks except archived tasks and completed tasks.

Completed tasks appear with a strikethrough. Dates, #tag, and priority are color coded for easy viewing. Choose how you want your tasks to be displayed – by date, priority or `type`'s default display.

Archiving a task removes it from the list, but does not permanently delete it. You can search through all archived tasks since the beginning of time.

Use the up and down arrow keys to scroll. Holding `shift` and pressing the `up/down` arrow key will scroll the list one page at a time.

Have no fear of losing your tasks. `type` automatically saves a copy of all your tasks, saving you the hassle of manually saving it. What's more, your tasks will be displayed exactly the same way you left it the next time `type` is launched.

Adding a Task

Quick Add

Simply start typing and hit `Enter/Return` to create a new task.

```
>> Make a sandwich
```

`#<tag>` to group tasks. Tags can appear anywhere in the task. You can use more than a single tag per task.

```
>> Do #cs2010r tutorial 4
>> Book #JetStar tickets for #Thailand holiday
```

`+<value>` or **`-<value>`** to give a particular task additional priority. Priorities can only appear at the end of a task. The default priority will be 0.

```
>> Uni application deadline tmr +5
>> #ma1101r tutorial 1 -99
```

Dates and Durations

`type` supports an amazing variety of time and date formats. Sophisticated algorithms understand what you mean when you enter a task, without you having to change the way you say things.

Start times and due dates can be attached to tasks. Ex:

```
>> 12pm make a sandwich
```

would infer that the due date for making a sandwich is 12pm today.

Keywords like "**`by`**", "**`at`**", and "**`deadline`**" suggest deadlines.

```
>> report by 4pm
```

Keywords like "**`from`**", and "**`since`**" suggest start times.

```
>> submit income tax returns from 1/5/2013
```

Tasks can have both start times and deadlines.

```
>> #cs2105 sockets assignment from monday to 14 sep
```

Completing a Task

`:done <text>` completes the first task that start with that text. Ex:

```
>> :done make a
```

would complete "make a bicycle", from a list containing "make a bicycle", "make a sandwich", etc. Completing a task marks it with a strikethrough on the display.

`:done #<hash tag>` completes all tasks with a particular tag.

```
>> :done #cs2010r
```

Undo Last Action

`:undo` rolls back last action. `type` supports unlimited levels of undo.

```
>> :undo
```

Archiving Tasks

`:archive` archives all completed tasks.

```
>> :archive
```

:**`archive <text>`** archives a specific task that start with that text. You can archive an active task using this command. Ex:

```
>> :archive make a
```

would archive the first task starting with "make a".

`:archive #<hash tag>` archives all completed tasks marked with that hash tag.

```
>> :archive #cs1020
```

Editing Tasks

`:edit <task>` allows you to edit an active task. Ex:

```
>> :edit make a sandwich
```

would copy "make a sandwich" into the command box. You can then edit your task. Hit `Enter/Return` to add the task back into your list. **Emptying the command box will result in the task being deleted.**

Sorting

`:sort <field>` sorts tasks by the chosen field. Completed tasks are always displayed at the bottom.

```
>> :sort date
```

- **`date`** sorts tasks by due date. Floating tasks appear at the bottom.

- `priority` sorts tasks by priority.
- `normal` reverts to `type`'s default behavior: Overdue tasks on top, floating tasks displayed next, and remaining tasks sorted by due date and priority.

Filtering

`/` shows all pending tasks. This is the default view.

`/archive` shows all archived tasks.

`/<tag>` filters tasks by tags.

```
>> /ma1101r
>> /school
```

You can filter by more than one tag. Ex:

```
>> /archive school
```

Shows archived school tasks.

Help

```
>> ?
```

Brings up the help screen. `type` speaks natural language. We trust you will not be using this much.

Summary

Commands

Create new task	`<task>`
Complete a Task	`:done <task>`
Complete all tasks with a tag	`:done #<tag name>`
Archive all complete tasks	`:archive`
Archive a single task	`:archive <task>`
Archive all tasks with a tag	`:archive #<tag name>`
Edit a Task	`:edit <task>`
Undo last action	`:undo`
Filter by hash tags	`/<tag name> [<tag name>] ...`
Show archive tasks	`/archive <tag name> [<tag name>] ...`
Sorting the display	`:sort <field>`
Help (Displays this list)	`?`

Task Attributes

Tagging	`#<tag name>`
Priority	`+/-<value>`

Date Formats (Not Exhaustive)

14 October 2012	`14 oct 12`
The next occurrence of 13 February	`13/2`
15 August 2013	`150813`
The next occurrence of 18 November	`18 nov`

Time Formats (Not Exhaustive)

4pm today	`4pm`
7pm today	`19:00`
5pm today	`1700hrs`
5.45pm today	`17:45`

Duration Formats (Not Exhaustive)

45 minutes from start time	`45 mins`
5 hours from start time	`5 hours`
2 hours and 30 minutes from start time	`2 h 30 min`
One hour from start time	`1 hr`

Keywords (Not Exhaustive)

`...` **`from`** `<date/time>` **`to`** `<date/time>`
`...` **`by`** `<date/time>`
`...` **`due`** `<date/time>`
`...` **`since`** `<date/time>`
`...` **`on`** `<date>`

Developer Guide

Table of Contents

Introduction

Type is a task manager that increases the efficiency of the user by simplifying the process of adding and editing tasks.

This guide will give you an overview of Type's architecture and key APIs. Reading only this document will not enable you to jump straight into development. You should further familiarize yourself with the documentation and source code of the components you would like to work on.

This developer guide is designed for people already well acquainted with Client-side Development, Microsoft Windows, C#, and the .NET framework.

- If you intend to work on the storage component of Type, you should also be familiar with disk-level file Input and Output (I/O).
- Working on the Graphical User Interface (GUI) component requires further knowledge of Windows Presentation Foundation (WPF).
- Developing Type's core application logic requires some background in system programming.

Architecture

Type is an Offline, Native, Command Line Interface (CLI), .NET Framework 4, and Microsoft Windows application. It is designed to launch the moment a user logs in and remain in the background until logout. The GUI of the application can be called up at any time by pressing a global hotkey, currently defined to be `Shift + Space`.

Model-View-Presenter

Type's logical architecture comprises three discrete components following the Model-View-Presenter (MVP) paradigm[1], a standard design pattern commonly seen in WPF applications (Figure 1).

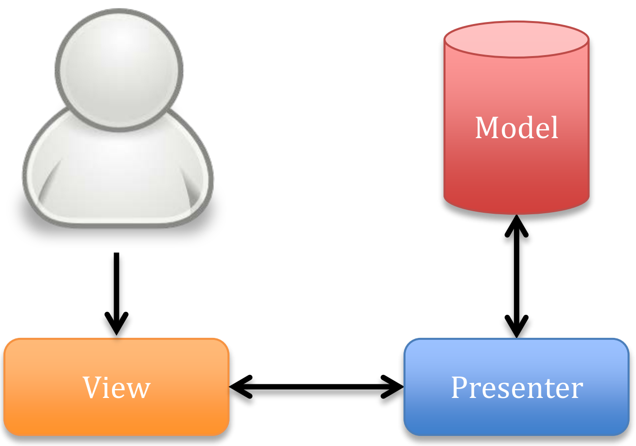


Figure 1 MVP Paradigm. The key difference from Model-View-Controller (MVC) is that the View has no notion of the Controller.

Core application logic is contained within Type's Presenter, which binds data in the Model to what the user sees in its User Interface.

The user interface currently consists of a single WPF window displaying a minimalistic CLI for user interactions:

[1] See Wikipedia's article on MVP at http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter.

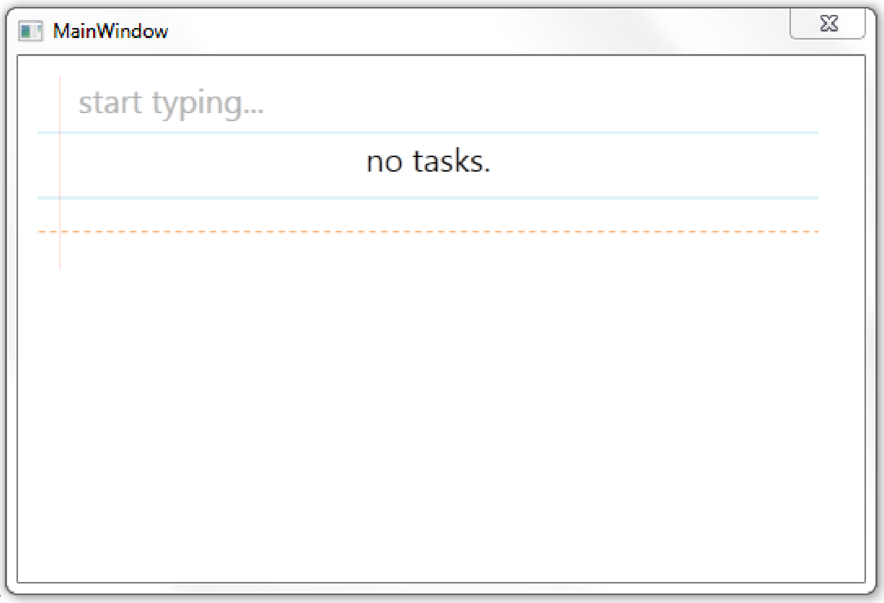


Figure 2 Minimalistic Command-Line User Interface

The underlying storage of the Model is based on a Comma Separated Values (CSV) format text file. CSV is human-readable and lightweight, making it ideal for an application like Type that aims to consume as few resources on the host system as possible.

Each Windows user has an individual data file with its own set of tasks stored in a folder in his/her own User Profile[2]. Type can be completely erased by deleting this folder.

Active Records

Data in computer systems can be broadly categorized into 'active' and 'passive' records.

Passive records are structures containing a set of fields that represent the data to be stored. Programs that use passive records contain methods that manipulate these structures individually, as well as in collections.

Type prefers active records. Task records in Type contain methods to parse and manage themselves, in addition to the underlying data that they store. This allows program logic to focus solely on the manipulation of collections of Tasks.

"Task" Record

Task records must be initialized with a string containing raw text to parse:

```
public Task(string rawText);
```

Upon initialization, the constructor parses times and hash tags, and populates the relevant fields in the record's data structure.

The following table describes the structure of a Task record in Type.

[2] "%APPDATA%\Type"

Type	Identifier	Description
`int`	`Id`	A globally unique identifier that serves as the Primary Key of a record.
`string`	`RawText`	The unparsed text of a record.
`bool`	`Done`	A flag representing the completion status of the task.
`bool`	`Archive`	A flag representing the archived status of the task.
`DateTime`	`Start`	The Date Time value at which the task becomes active.
`DateTime`	`End`	The Date Time value at which the task becomes overdue.
`List<string>`	`Tags`	A collection of all hash tags associated with the task as strings.
`IList<Tuple<string, ParsedType>>`	`Tokens`	A collection of tokens, stored as Tuples, representing consecutive substrings of `RawText` that are rendered by the GUI based on the `ParsedType` enumeration. Each substring corresponds to a section of raw text that was parsed by the constructor

Table 1 Structure of a Task Record.

The `ParsedType` enumeration provides substring-specific styling information to users of the Task record. The enumeration is defined to contain:

1. `STRING`
2. `HASHTAG`
3. `DATETIME`

State of Work

Type's MVP architecture allows each component to be developed independently of the others. You do not need to understand Type in its entirety to begin contributing code – It is only necessary to appreciate how your code interacts with adjacent components.

In general, the areas listed below require further development.

- Parsing of Time formats
- Persisting data between sessions

The following dependency diagram is a visualization of component dependencies in the current version of Type (Figure 3). The diagram is centered on the Presenter class, which is the entry point of the application.

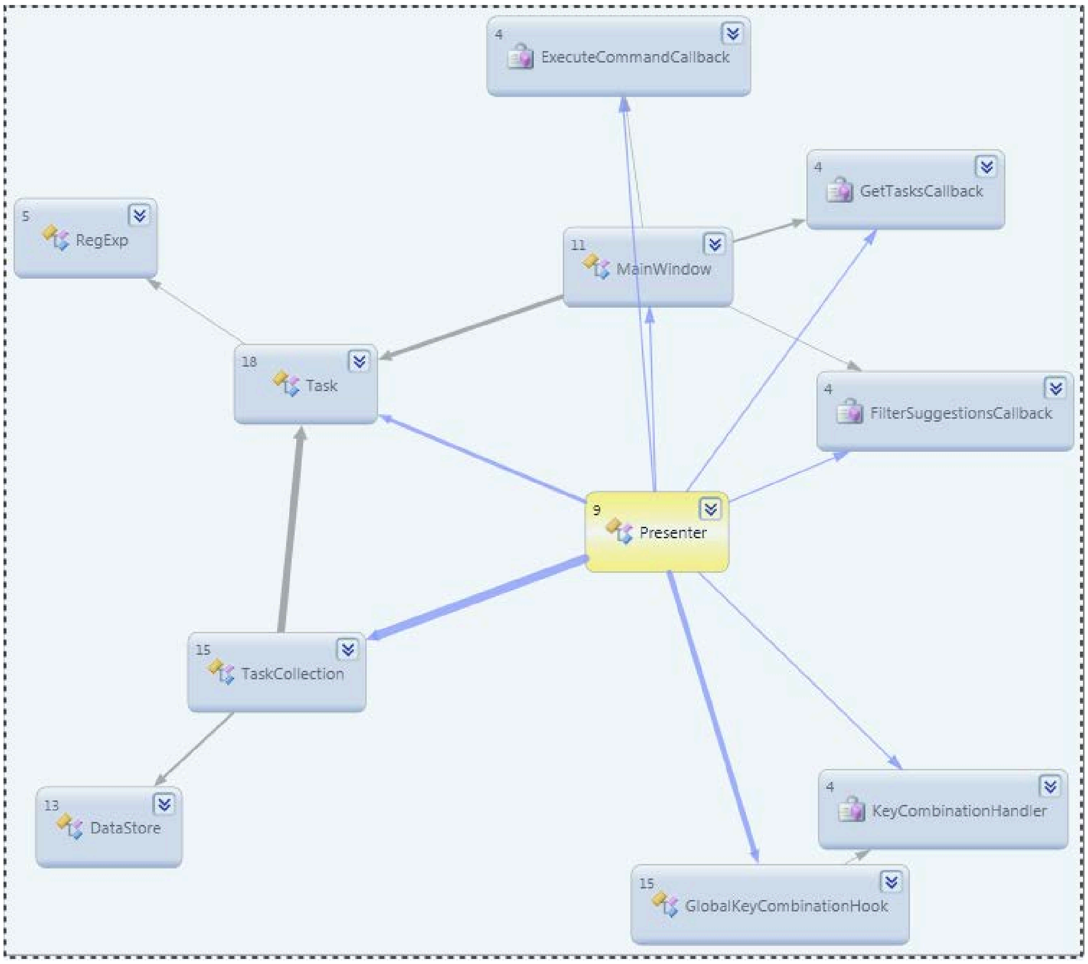


Figure 3 Dependency Diagram centered on the Presenter class.

In the remainder of this developer guide, we will only introduce the core classes directly involved in Type's data flow. Before starting to contribute, you should read the documentation of the classes your code is expected to interact with.

API Overview

Each of Type's logical components contains one or more classes. These classes manipulate a single collection of Task records to support Create, Retrieve, Update, and Delete (CRUD) functions.

The following diagram is a visual summary of important classes that deal with the flow of data through the application (Figure 4).

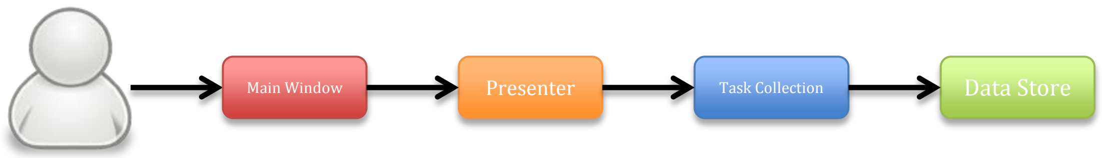


Figure 4 Data Flow Components of Type.

The next sections describe these classes in relation to their position within the logical architecture of Type.

View

The View mediates interaction with the user. The following delegates specify the way in which other components must communicate with the View:

Type	**Prototype**	**Description**
Delegate	`IList<Task> FilterSuggestionsCallback(string partialText)`	Retrieves a list of suggestions that begin with a specified prefix.
Delegate	`void ExecuteCommandCallback(string rawText, Task selectedTask = null)`	Parses a raw string and executes its command, if valid. If no valid command is found, this method does nothing.
Delegate	`IList<Task> GetTasksCallback(int num)`	Retrieves a list of tasks to be displayed.

Table 2 External API of logical View component accessed via the `MainWindow` class.

The View consists of a single WPF form “`MainWindow`”. Its constructor expects the following to be immediately specified:

```
public MainWindow(
     FilterSuggestionsCallback GetFilterSuggestions,
     ExecuteCommandCallback ExecuteCommand,
     GetTasksCallback GetTasks
);
```

After initialization, all API delegates will have already been set to the appropriate callbacks. It is advisable to retrieve an initial list of tasks to display using `GetTasksCallback`.

GUI in WPF is event driven. The logic of `MainWindow` revolves around listening for two main events: `TextChange` and `KeyUp`.

The `TextChange` event is triggered every time the text in the input box changes. When the event is triggered, `MainWindow` sends user input to the Controller via `FilterSuggestionsCallback`, which returns a list of matches that start with the input.

The `KeyUp` event is triggered every time a previously depressed key is released. However, the event handler only processes the following special keys:

- **Enter/Return**. Sends the contents of the input box back to the Presenter via `ExecuteCommandCallback`. After the command is processed, `MainWindow` gets an updated list of Tasks to display from the Controller with `GetTasksCallback`. The following sequence diagram presents a visualization of this interaction (Figure 5).

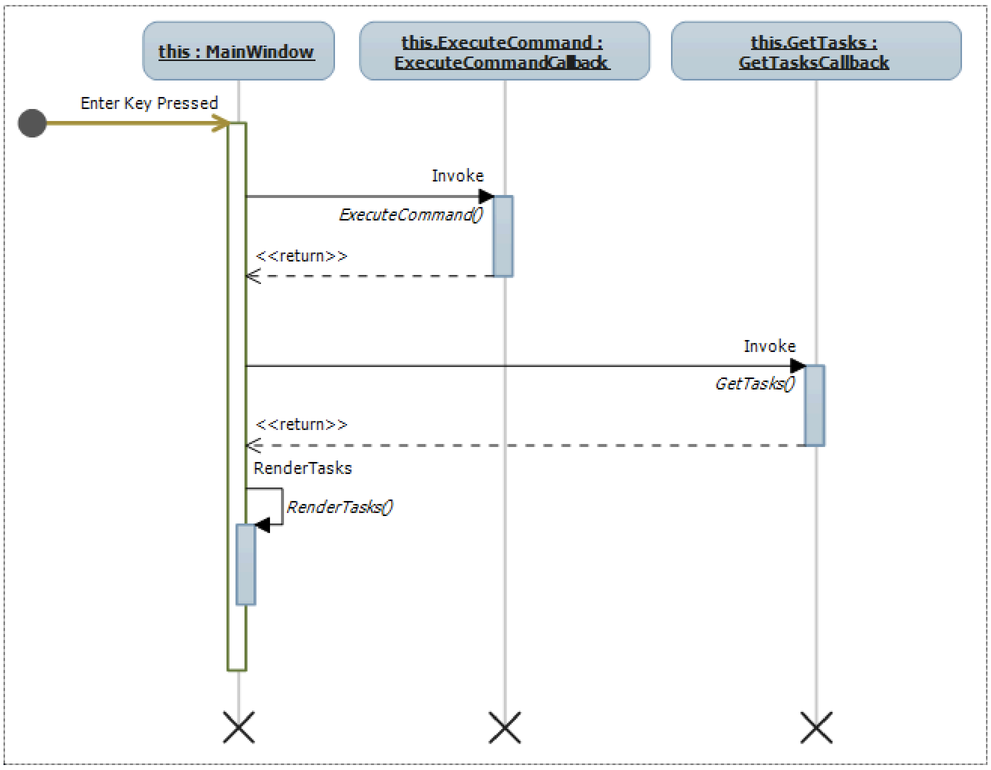


Figure 5 Sequence Diagram showing method calls when the "Enter/Return" key is pressed.

- **Tab**. Complete the current sentence in the input box to the best possible match. This feature has not been implemented, and we encourage you to explore it when joining our team.
- **Escape**. Hides the GUI. Type continues to run in the background, and can be invoked again by pressing `Shift + Space`.

Tasks are displayed by modifying the WPF document representing the GUI. The host OS handles redrawing the window natively.

Presenter

The Presenter acts as a bridge between Type's User Interface and Model. It does not expose an external API. It is also the first class initialized when the application loads, and is responsible for creating resources and setting them to their initial states.

Initialization order is important. An instance of Type's Model must always be initialized before the View. The last component that should be initialized is `GlobalKeyCombinationHook`. Once every component has been initialized, Presenter starts listening to key combinations, and the application begins to accept user input (Figure 6).

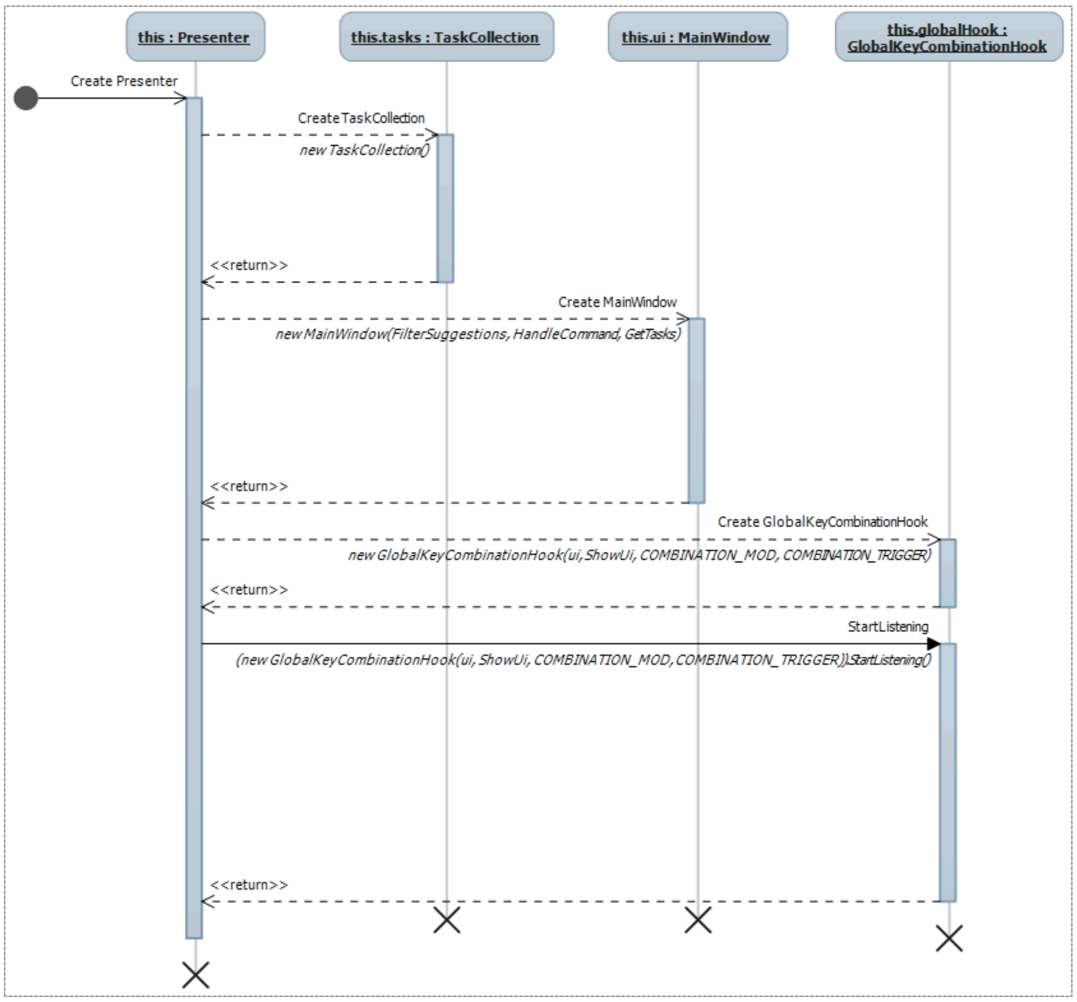

this : Presenter
this.tasks : TaskCollection
this.ui : MainWindow
this.globalHook : GlobalKeyCombinationHook
Create Presenter
Create TaskCollection
new TaskCollection()
<<return>>
Create MainWindow
new MainWindow(FilterSuggestions, HandleCommand, GetTasks)
<<return>>
Create GlobalKeyCombinationHook
new GlobalKeyCombinationHook(ui, ShowUi, COMBINATION_MOD, COMBINATION_TRIGGER)
<<return>>
StartListening
(new GlobalKeyCombinationHook(ui, ShowUi, COMBINATION_MOD, COMBINATION_TRIGGER)).StartListening()
<<return>>

Figure 6 Sequence Diagram showing the initialization sequence of the `Presenter` class.

Presenter also contains three methods that handle data from `MainWindow`. These methods correspond to the View API previously described. References to these methods are passed to `MainWindow` when the User Interface is initialized. The prototypes for these methods are as follows. Implementation details and documentation can be found in the source code.

```
private IList<Task> FilterSuggestions(string partialText)

private void HandleCommand(string rawText, Task selected
= null)

private IList<Task> GetTasks(int num)
```

Listening to the `Shift + Space` keyboard shortcut is done by setting a callback with the host OS. This is achieved through various functions in the Win32 API outside the scope of this developer guide.

Type encapsulates these external calls in a class called `GlobalKeyCombinationHook`. Further documentation can be found in the source code, and information about the Win32 API can be found on Microsoft's developer resource library MSDN[3].

Model

Type works by manipulating collections of Tasks. A collection, and its CRUD operations, is encapsulated in a `TaskCollection`.

A `TaskCollection` exposes the following API:

Type	Prototype	Description
Method	`Task Create(string input)`	Creates a new Task record and returns a reference to it.
Method	`Task GetTask(int id)`	Retrieves a reference to a Task record based on its Primary Key.
Method	`IList<Task> Get()`	Returns a list of Task records in the collection.
Method	`IList<Task> Get(int number, int skip = 0)`	Returns the first 'number' number of Task records, starting at index 'skip'. The default value of 'skip' is zero, the first record.
Method	`Task UpdateDone(int id, bool done)`	Sets the done flag of a Task record to 'done' according to its Primary Key.
Method	`Task UpdateArchive(int id, bool archive)`	Sets the archive flag of a Task record to 'archive' according to its Primary Key.

[3] See Microsoft's Reference at http://msdn.microsoft.com/en-us/library/windows/desktop/ff818516.

Method	`Task UpdateRawText(int id, string str)`	Modifies the contents of a Task record according to its Primary Key by parsing a new string 'str'.
Method	`IList<Task> FilterAll(string input)`	Returns a list of Task records with text that begins with 'input'.

Table 3 External API of logical Model component accessed via the `TaskCollection` class.

`TaskCollection` is responsible for persisting its associated data through a `DataStore` object.

The `DataStore` abstracts low-level file I/O. Data is stored in the comma-separated values (CSV) format.

A CSV file consists of multiple lines, with each line representing a single record. A record consists of a series of values separated by commas. In Type, the format of a record is as follows:

```
UniqueID,Data,DoneFlag,ArchiveFlag
```

A sample record could resemble:

```
0,Make a sandwich #lunch at 3pm,false,false
```

The `DataStore` object exposes the following API to its clients:

Type	**Prototype**	**Description**
Method	`void ChangeRow(int index, List<string> row)`	Change the value of a row
Method	`List<string> Get(int index)`	Returns row given index
Method	`Dictionary<int, List<string>> Get()`	Get all rows in the data store
Method	`int InsertRow(List<string> row)`	Insert New Row to file

Table 4 Internal API exposed by the `DataStore` class.

Build Instructions

```
hg clone
https://cgcai%40comp.nus.edu.sg@code.google.com/p/cs2103aug12-w10-1s/
```

The GUI of Type does not immediately show up when the application is first launched in Visual Studio. To show the GUI, press `Shift + Space`.

Testing Methodology

Both exploratory testing and unit testing are used in the development of Type.

If you are working on the GUI component, you are encouraged to do exploratory testing.

For the development of Type's core application logic or storage component, unit testing is recommended.

There are currently several test projects used to test these aforementioned areas:

- `AutoCompleteTests`
- `LanguageTests`
- `ModelTests`
- `DataStorageTest`

You are welcome to contribute more test cases.