Developer Guide

Table of Contents

1. Introduction

Type is a task manager that help user to decide what to do and when to do things. It increases the efficiency of the user by simplifying the process of adding, editing and archiving tasks. Type only accepts commands via keyboard.

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.

- Developing on the storage component of Type requires familiarity with disk-level file Input and Output (I/O).
- Developing on the Graphical User Interface (GUI) component requires further knowledge of Windows Presentation Foundation (WPF).
- Developing in the Model data logic require familiarity with regular expression.
- Developing Type's presentation logic requires some background in system programming.

2. 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.

2.1 Model-View-Presenter

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

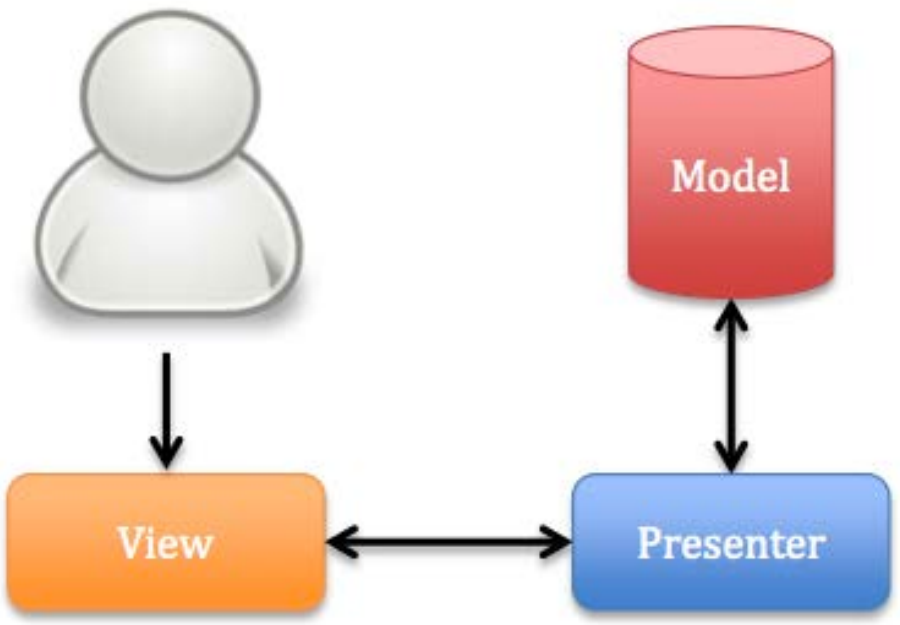

Figure 1. MVP Paradigm

The Model contains the data processing logic. The underlying storage is based on a Comma Separated Values (CSV) format text file. The View is the user interface which currently consists of a single WPF window displaying a minimalistic CLI for user interactions. The Presenter contains Type's presentation logic, which formats the data retrieved from the Model to what the user sees in its User Interface. Figure 2 shows Type's CLI.

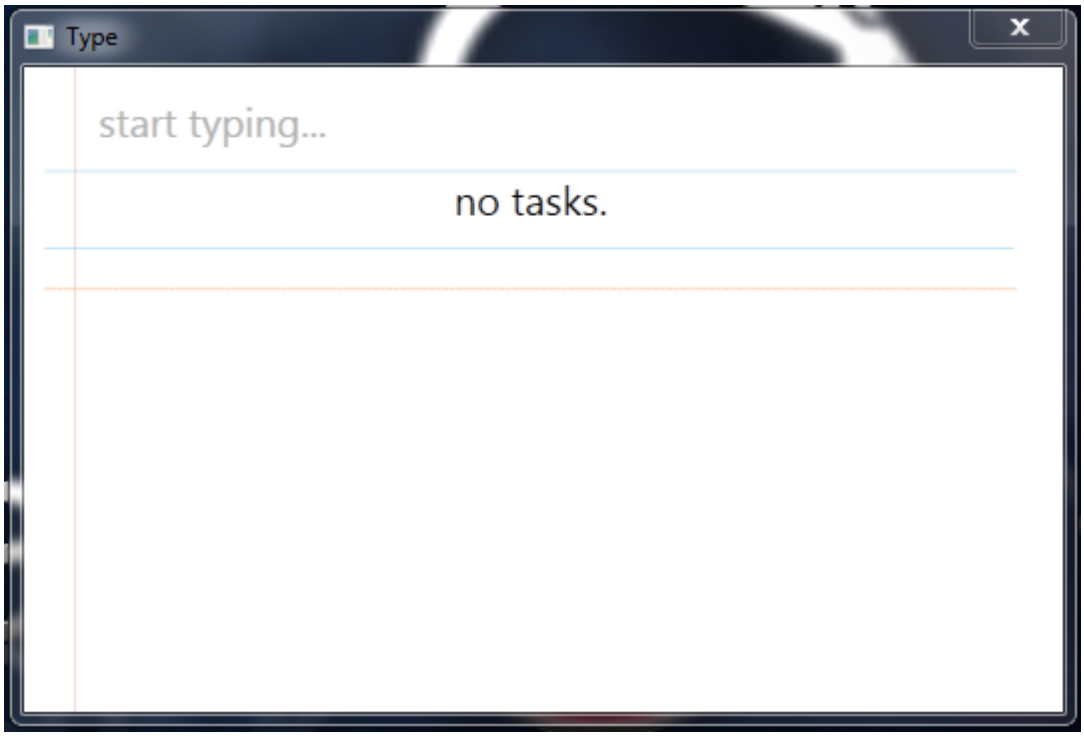

Figure 2. Minimalistic Command Line Interface

2.2 Task Record

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 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:

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.
`List<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
4. PRIORITYHIGH
5. PRIORITYLOW

3. 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.

- Sorting and display orders.
- Usability adjustments.

- Configuration files.

The class diagram below is a visualization of component dependencies in the current version of Type. The diagram is centred on the Presenter class, which is the entry point of the application.

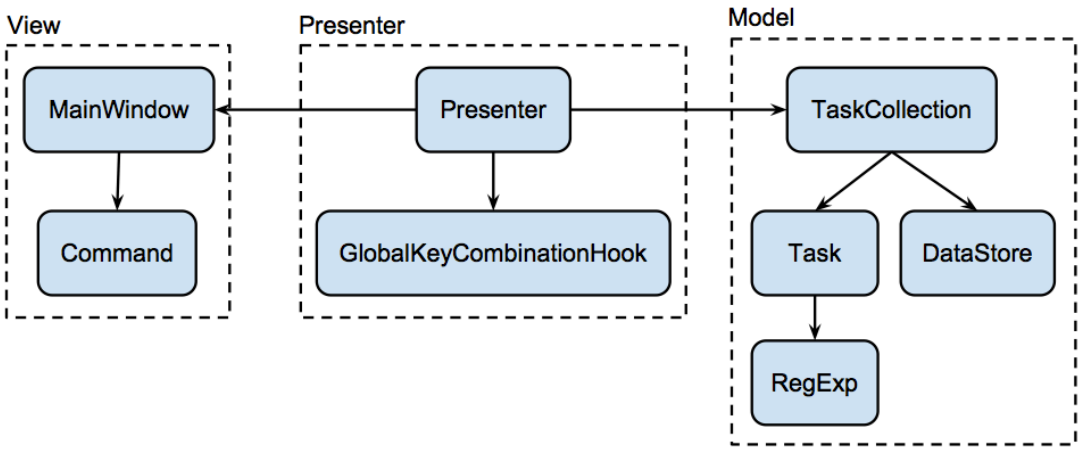

Figure 3. Type's Class Diagram

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.

4. Important API

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.

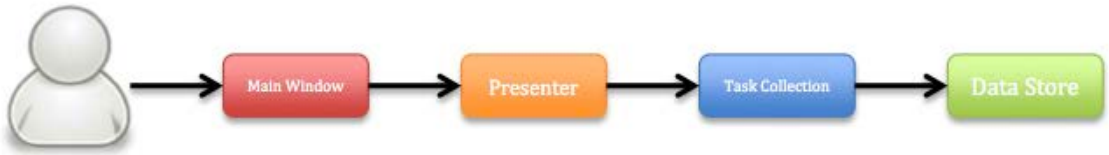

Figure 4. Data Flow in Type

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

4.1 View

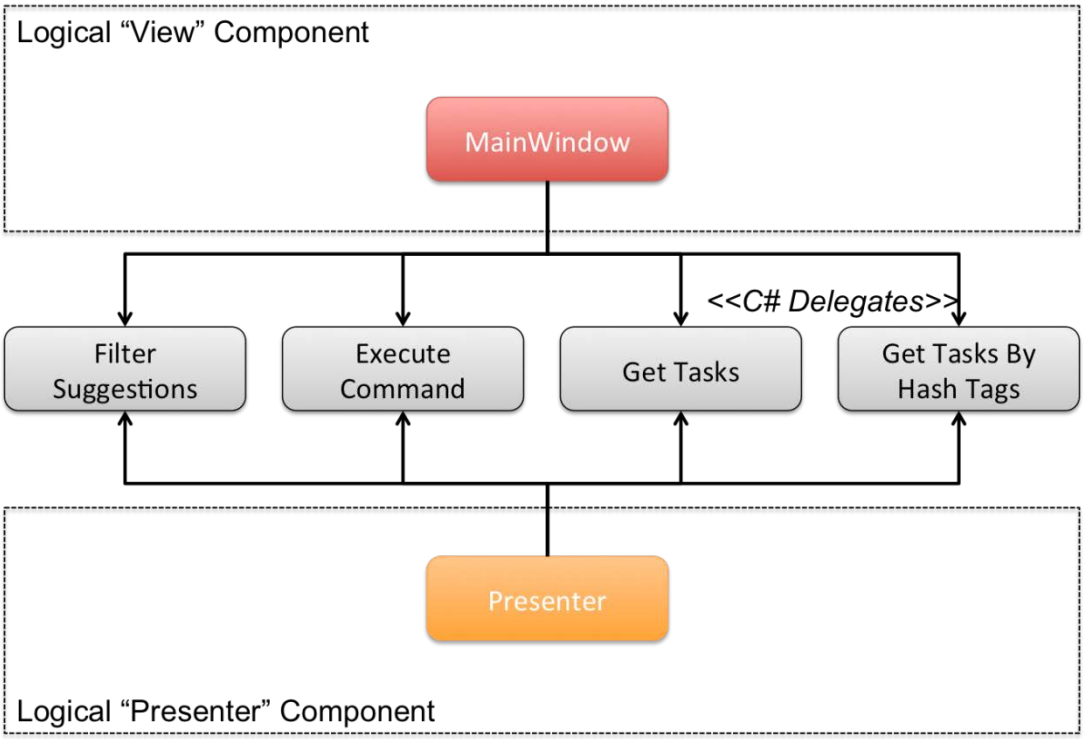

Figure 5. Interaction between View and Presenter

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

Prototype	Description
`IList<Task> FilterSuggestionsCallback(string partialText)`	Retrieves a list of suggestions that begin with a specified prefix.
`ExecuteCommandCallback(string cmd, string content, Task selectedTask = null)`	Parses a raw string and executes its command, if valid. If no valid command is found, this method does nothing.

`IList<Task> GetTasksCallback(int num)`	Retrieves a list of tasks to be displayed.
`IList<Task> GetTasksByHashTagCallback(string content)`	Retrieves a list of tasks tagged with at least one hash tag.

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,
GetTasksByHashTagCallback GetTasksByHashTag)
```

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`.

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

`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 Presenter with `GetTasksCallback`.

- **`Tab:`** Complete the current sentence in the input box to the best possible match.
- **`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.

4.2 Presenter

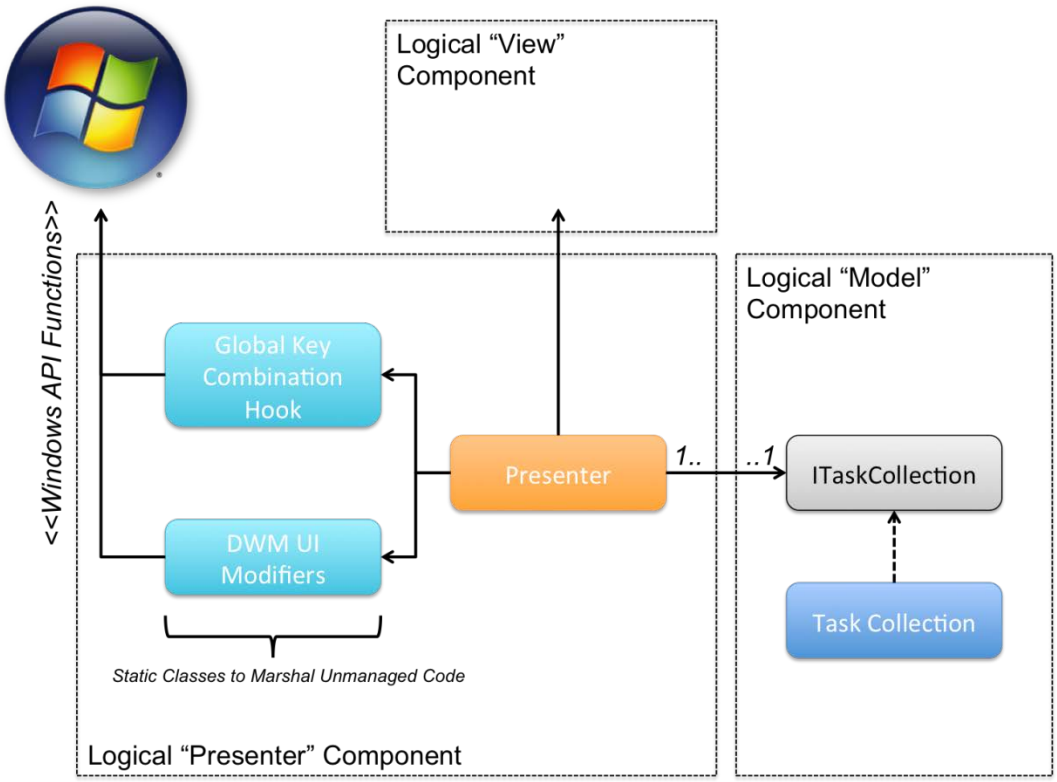

Figure 6. Interactions among Presenter, View, and Model

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.

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> GetTasksWithPartialText(string
partialText)

private void HandleCommand(string cmd, string content,
Task selected = null)

private IList<Task> GetTasksNoFilter(int num)

private IList<Task> GetTasksByHashTags(string content)
```

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].

4.3 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:

Prototype	Description
`Task Create(string input)`	Creates a new Task record and returns a reference to it.
`Task GetTask(int id)`	Retrieves a reference to a Task record based on its Primary Key.
`IList<Task> Get()`	Returns a list of Task records in the collection.
`List<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.
`Task UpdateDone(int id, bool done)`	Sets the done flag of a Task record to 'done' according to its Primary Key.
`UpdateArchive(int id, bool archive)`	Sets the archive flag of a Task record to 'archive' according to its Primary Key.
`Task UpdateRawText(int id, string str)`	Modifies the contents of a Task record according to its Primary Key by parsing a new string 'str'.
`List<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.

4.3.1 DataStore

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

A DataStore exposes the following API:

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

Table 4. Internal API exposed by the DataStore class.

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

5. Build Instructions

`hg clone` https://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`.

6. 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 presentation logic or storage component, unit testing is recommended. There are currently several test projects used to test these aforementioned areas:

HashTagsTest and DateTest	-- `RegExp` class
DataStoreTest	-- `DataStore` class
TaskCollectionTest	-- `TaskCollection` class

You can always contribute more test cases.

[2] "%APPDATA%\Type"
[3] See Microsoft's Reference at http://msdn.microsoft.com/en-us/library/windows/desktop/ff818516.