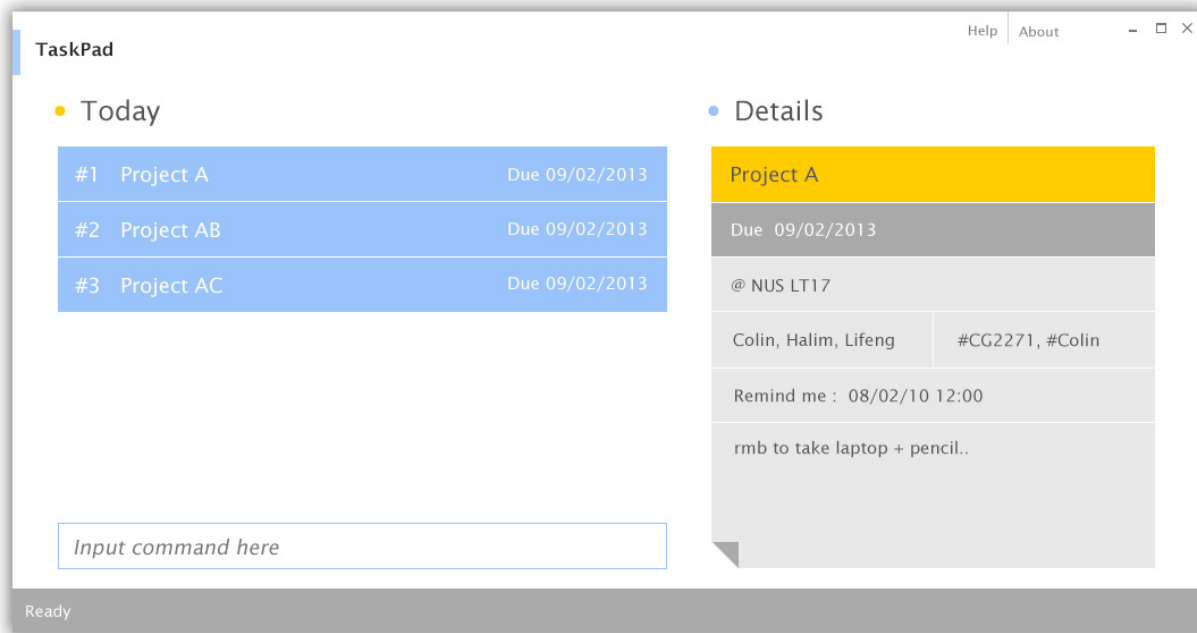


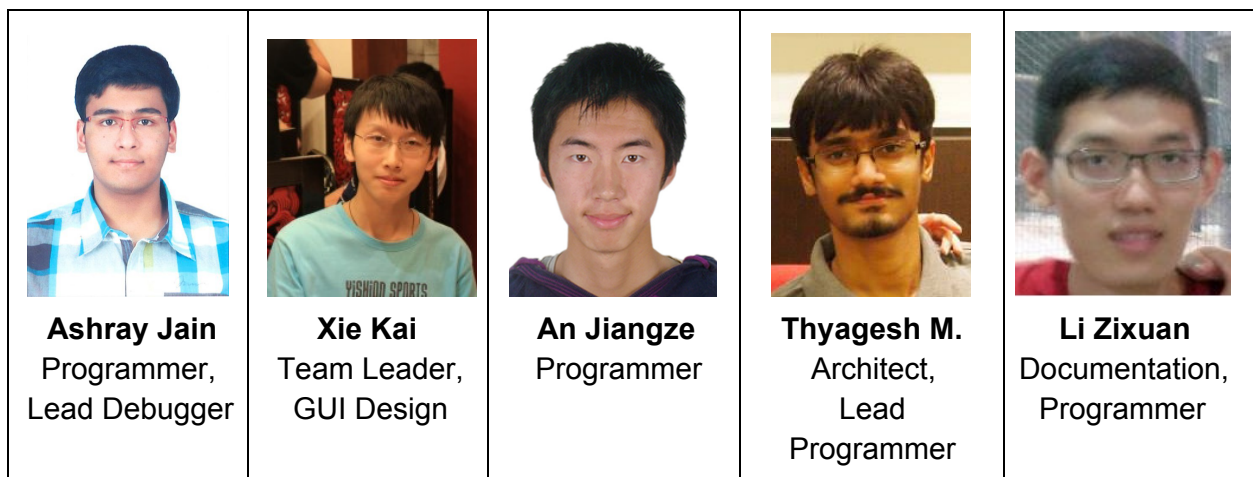
From the users of NotePad... comes a revolutionary new product...

TaskPad

By Team F12-1C



The Team



Acknowledgements

TaskPad is powered by [Qt 5 development framework](#), [libqxt](#) and [Natty](#)

User Guide

Introduction

TaskPad is a simple, elegant and revolutionary task organiser designed specifically for power users. It empowers the user with 4 basic yet comprehensive functions to **Add**, **Modify**, **Find** and **Delete** tasks. **Add** and **Modify** offer a comprehensive list of parameters that neatly catalogue the specifications of tasks while **Find** is a highly flexible command that can retrieve tasks according to any possible combination of parameters. **Delete**, on the other hand, just performs what it is meant to do, i.e., delete tasks as required.

TaskPad also has **Hot Key Templating**. All of the 4 functions above are made simple with intuitive hotkeys that call on the full command template for the particular function with clear instructions for the user to just fill in the blanks. Some additional features include **Date Navigation** to view tasks due on different days and **Date Parsing with Natural Language Processing** to allow the user to express himself in a vast variety of ways without being restricted to a format.

Tasks

TaskPad supports 3 types of task:

1. **Floating tasks** i.e *tasks with no/not yet specified deadline*
2. **Deadline tasks** i.e *tasks with specified deadline*
3. **Timed tasks** i.e *tasks with specified start and end time e.g meetings*

Standards

A function is performed by typing the command keywords (**add**, **mod**, **find**, **del**, **undo** and **redo**) followed by parameter keywords with the task details in a pair of **grave accent marks** (**due** ``<value>``, **ppl** ``<value>`` etc.). The complete list of possible parameters can be found in the Appendix TaskPad A

Functions

Add

To add a task, the keyword to use is **add** followed by the mandatory parameter - **name**. If only the name is given, the task is automatically considered to be a **Floating Task**. Typically the first parameter after the keyword add is taken as the name.

To make a **Deadline Task**, another parameter - the **due** date- has to be specified. In the following example a `Project A` is **due** on `13/12/13 23:59`, making it a deadline task.

```
add `Project A` due `13/12/13 23:59`
```

Task added successfully

Lastly to create a simple **Timed Task** three parameters have to be given - **name**, **from** start time, **to** end time.

```
add `Project A` from `09/02/90` to `09/02/13`
```

Task added successfully

Additional options include a **location (at)**, other participants or **members (ppl)**, an **importance (impt)** level, **tags (#)** to categorise them, **reminder times (rt)** for alarms and side **notes (note)** (if any).

The following example adds a deadline task `Project A` with a due date, a location, and an importance level. To understand the format, please read through the *Standards* section above.

```
add `Project A` due `12/12/13` at `NUS` impt `H`
```

Task added successfully

Complete **Add** Command Template:

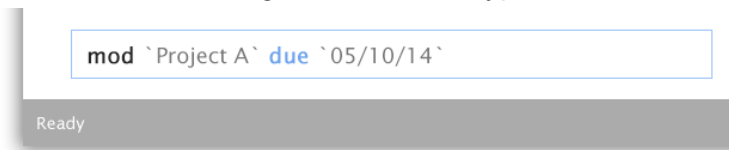
```
add `<name>` due `<due date>` from `<from date>` to `<to date>` at `<location>` ppl  
`<participants>` note `<notes>` impt `<H/M/L or HIGH/MEDIUM/LOW>` #<tags> rt `<remind  
time>`
```

Color code: **light blue** - compulsory, **gray** - optional

Modify

To modify a previously added task, the user can simply use the **mod** keyword followed by the name of the task to modify followed by the detail to edit and then the new value.

For instance to change the due date type,

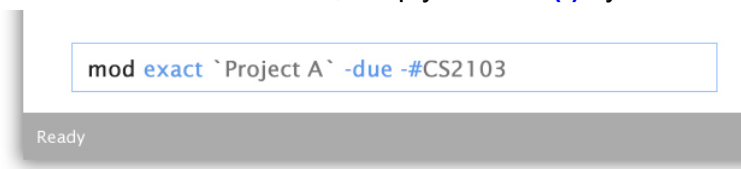


A screenshot of a terminal window. The command entered is `mod `Project A` due `05/10/14``. The word `due` is highlighted in blue. Below the command line, a gray bar displays the text "Ready".

When similar names are present, the system will return a list of all similarly named tasks allowing the user to choose the exact one he/she is referring to. The user can then, type in the index of the task that he/she wants to change and press the return key.

To skip this step, the user can specify **exact** in front of the name in which case if an exact match is found, the system executes the command directly. If there are more than one task with the exact same name, the above step is unavoidable.

To remove a certain detail, simply use the **(-)** symbol in front of the attribute's command word:



A screenshot of a terminal window. The command entered is `mod exact `Project A` -due -#CS2103`. The words `exact` and `-due` are highlighted in blue. Below the command line, a gray bar displays the text "Ready".

To add a certain detail, simply use the **(+)** symbol. E.g. `+rt `tomorrow 9pm``.

Complete **Modify** Command Template:

```
mod exact `<current name>` name `<name>` due|-due `<due date>` from|-from `<from this date>` to|-to `<to this date>` at `<location>` ppl|-ppl|-pplall `<participants>` note `<notes>` impt `<H/M/L or HIGH/MEDIUM/LOW>` #|+##|-#<tags> rt|-rt|-rtall `<remind time>` done|undone
```

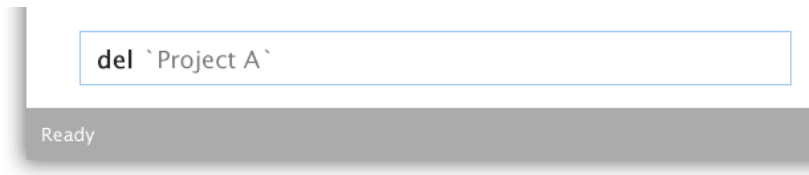
```
mod index name `<name>` due|-due `<due date>` from|-from `<from this date>` to|-to `<to this date>` at `<location>` +ppl|-ppl|-pplall `<participants>` note `<notes>` impt `<H/M/L or HIGH/MEDIUM/LOW>` #|+##|-#<tags> rt|-rt|-rtall `<remind time>` done|undone
```

Color code: **light blue** - compulsory, **gray** - optional

Delete

To delete a task, use the command **del**. Please be mindful that this command completely

deletes any memory of this task from the system. If similar task names are found, a workflow similar to that of modify will be followed.



Complete **Delete** Command Template:

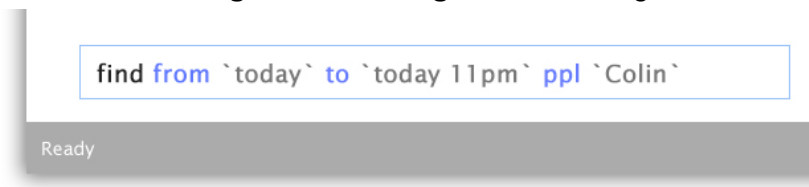
del exact ` <current name> `
del index

Color code: *light blue* - compulsory, *gray* - optional

Find

To find a task use the command **find**. A basic find operation involves just the name of the task with the optional keyword **exact**, which directly gives the details of a specific task that is given. It follows a workflow similar to that of the first step of **modify**.

There are other ways to search. For instance to search for tasks which are **with participants and due within a given date range**, the following command can be used:



Complete **Search** Command Template:

find exact name ` <name> ` timed deadline floating from ` <from this date> ` to ` <to this date> ` at ` <location> ` ppl ` <participants> ` note ` <notes> ` impt ` <H/M/L or HIGH/MEDIUM/LOW> ` #<tags> rt ` <remind time> ` done undone overdue

Color code: *light blue* - compulsory, *gray* - optional

Undo & Redo

To undo a command, simply type **undo** in the command bar.

To cancel undo, simply type **redo** in the command bar.

Hot Key Templating

Press special hotkeys to populate the command bar with command template. Then, using tab, input all required fields.

Example: **Ctrl-N** can be a hotkey for add command. Pressing this will populate the command bar with

add ` <name> ` due ` <due date> ` at ` <location> ` ppl ` <participants> ` note ` <notes> ` impt ` <H/M/L or HIGH/MEDIUM/LOW> ` #<tags> rt ` <remind time> `

and the user can then just start typing and use **<TAB>** key to navigate to and fill in all the desired fields. **<Shift> + <TAB>** helps navigate backwards. Some examples:

Function	Shortcut
Complete Add Template	Ctrl + N
Complete Modify Template	Ctrl + M
Complete Find Template	Ctrl + F
Delete Template	Ctrl + D
Jump between parameters in Template	Tab or Shift + Tab
Delete Word	Ctrl + Backspace

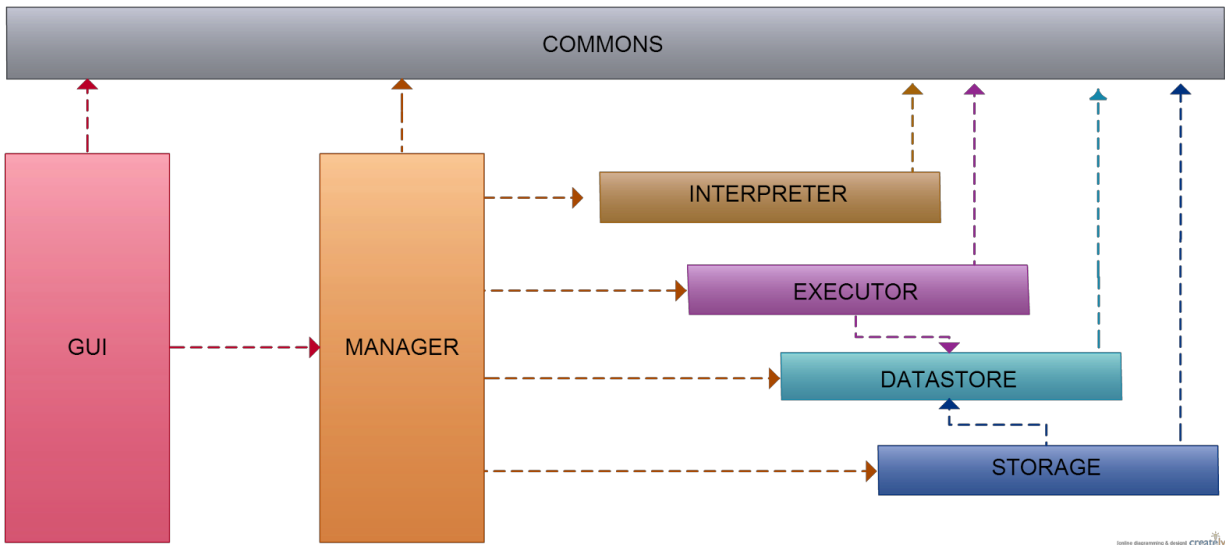
Arrow Keys (Up - Down) Navigation

Arrow Keys can cycle among entered commands' history

For more hotkey information, kindly refer to the Appendix Hotkey

Developer Guide

Architecture

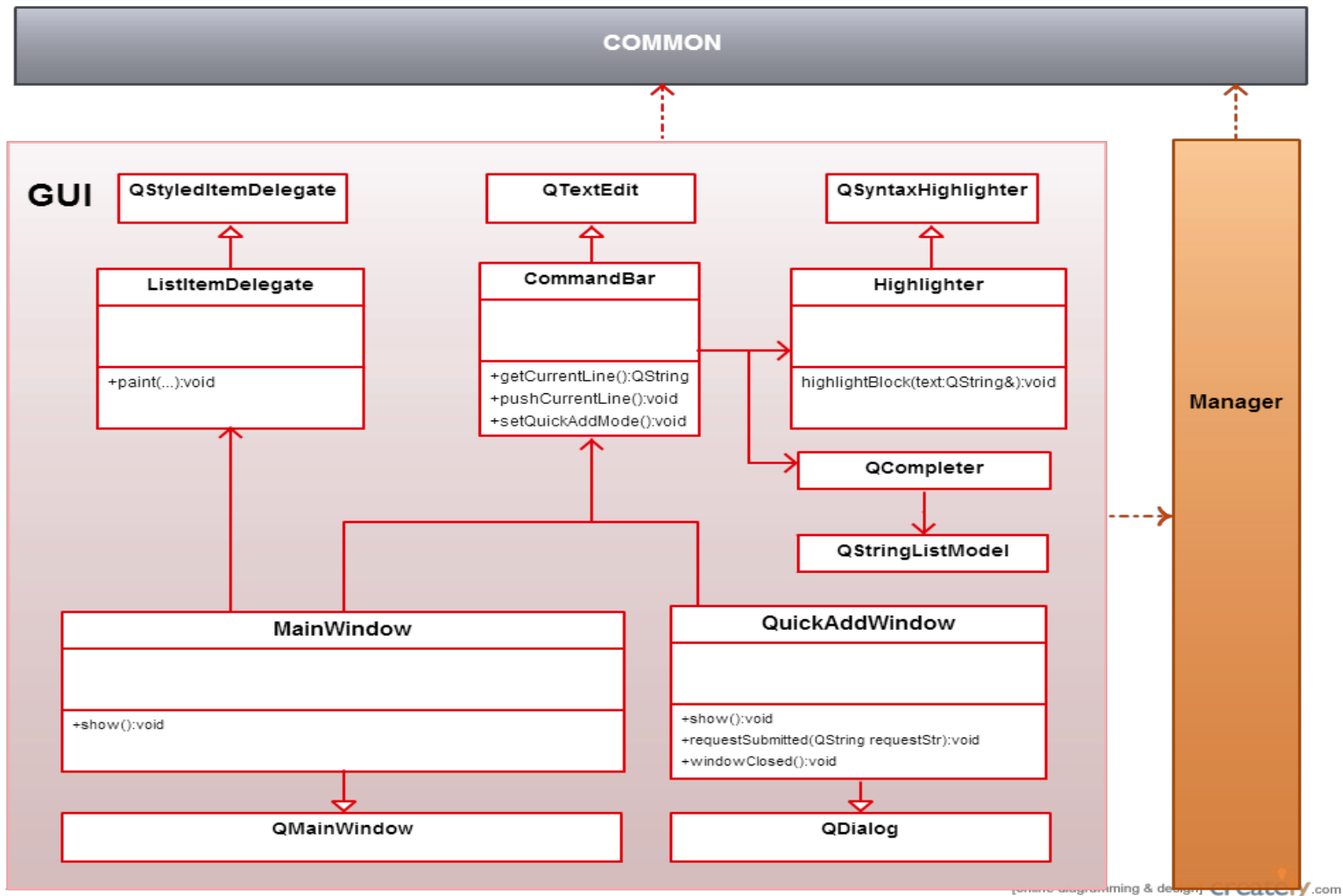


Given above is an overview of the TaskPad architecture.

- **GUI:** The GUI of TaskPad is developed under Qt 5.1.1 development framework (C++). The visual part of this GUI is generated by the file `mainwindow.ui` when compiled, and can be edited by Qt Designer.
- **Manager:** The Manager serves as a facade and synchronises the calls to the Interpreter, Executor and Storage for a given user command.
- **Storage:** The Storage component uses `std::fstream` to read from and write to the persistent data store in the form of a text file
- **Interpreter:** The core function of the Interpreter is to 'decipher' the user input. The Interpreter makes sense of the user input and creates a structured entity representing the command to be used for further processing.
- **Executor:** The Executor component determines the course of action that needs to be taken for a given command and instructs the Datastore appropriately, to carry it out.
- **Datastore:** The Datastore component is responsible for handling and operating on all data and the associated data structures used by TaskPad. Any changes to data in TaskPad has to be done by through this component.
- **Commons:** The Common component contains utility code used across the application.

GUI

Below shows the class diagram of the GUI component.



Class Overview

1. MainWindow

Contains the major GUI logic, acting as a presenter in MVP pattern.

2. QuickAddWindow

It is a window created by **MainWindow**, which helps user input a new task quickly. Connect to `requestSubmitted(QString)` in order to get the input string from user.

3. ListItemDelegate

It beautifies the GUI once its API `paint(...)` is called by **QWidget**. E.g., display a red bar for high-priority task, and orange bar for medium-priority task; display a strike-out for task

which is done.

4. **CommandBar**

Makes QTextEdit to support syntax-highlighting and auto-completion. Call getCurrentLine() to get user input. When it's in QuickAddWindow, calling of setQuickAddMode() is required.

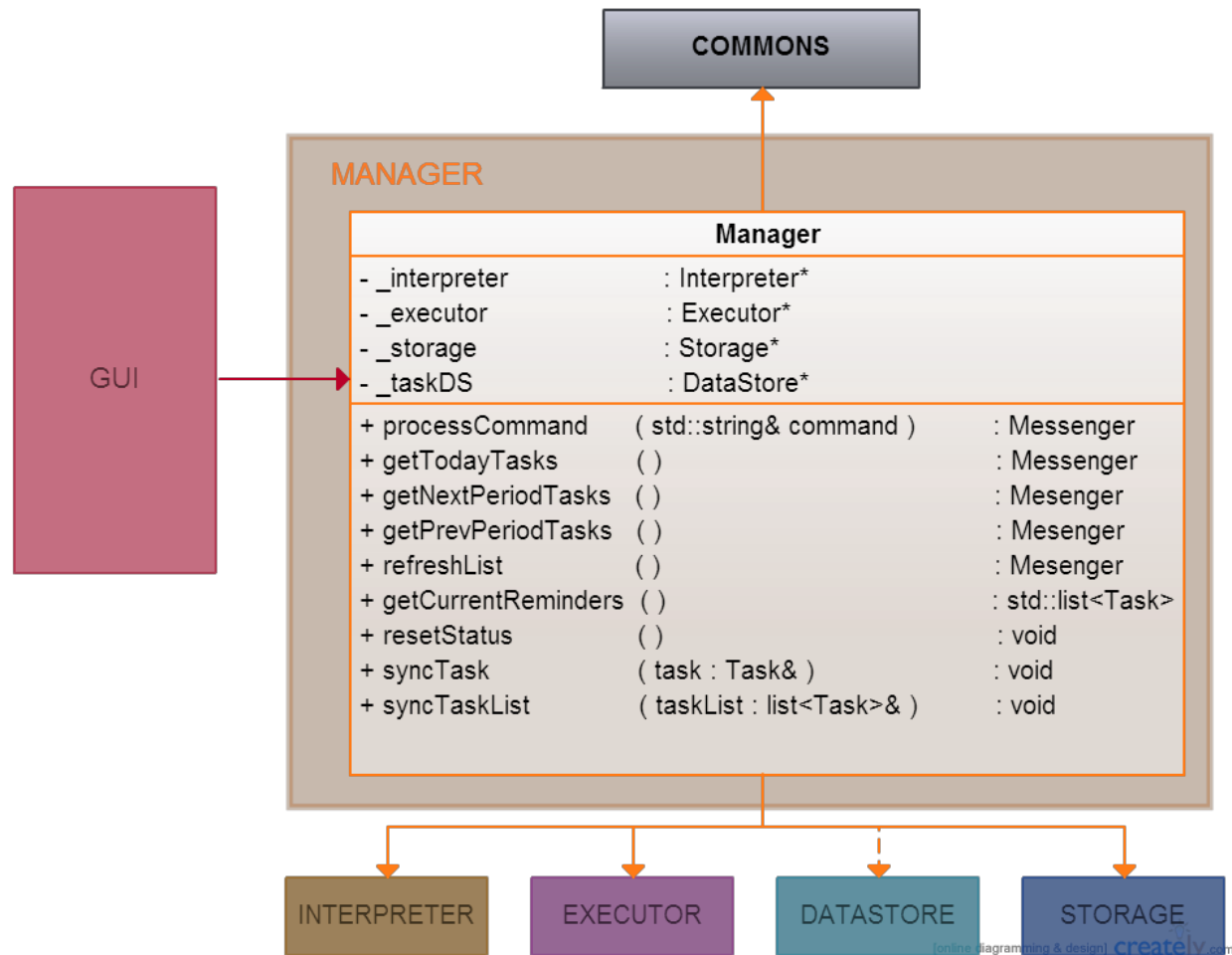
5. **Highlighter**

Provides syntax-highlighting ability for CommandBar. highlightBlock(QString) will be called by CommandBar's document automatically.

6. **Other classes (name starting with Q..)**

Refer to http://qt-project.org/wiki/Qt_5.0

Manager



The Manager class serves as a facade for all the other Logic related components (i.e. Interpreter, Executor and Storage). The GUI interacts with Manager to do all kinds of operations. Among other things, the Manager synchronises the calls to the Interpreter, Executor and the Storage to ensure that the life cycle of a given command completes successfully.

API

There are 5 kinds of API provided by the Manager to the GUI.

1. Command Processing - `processCommand(string)`

For a typical command the following steps are taken by the Manager

- The command is passed to the interpreter to determine the type and associated parameters of the given Command
- If there were no interpretation errors then the condensed Command object is passed to the Executor for execution
- If the Executor had no issues with the proceedings, then the operation is saved by calling

the save API of the Storage class with either the Task in question (for add command) or the entire list<Task> (the other commands like mod or del).

- A response object encapsulating a list of tasks (to be shown to user), the task that was edited and/or any error messages to be shown to user is then returned to the GUI

2. Date Navigation

- `getTodayTasks()`, `getNextPeriodTasks(PERIOD_TYPE)`, `getPrevPeriodTasks(PERIOD_TYPE)`

These are functions at higher level of abstraction for the convenience of the GUI to support the easy navigation between dates/periods. The following is done in these functions

- The new date range for which tasks are to be returned is calculated
- A new “find” command string is constructed
- The `processCommand()` function is called

3. Auto Update List - `refreshList()`

This function is called after any general command that the user executes to update the list of tasks he/she is currently viewing to include effects of the latest command that was executed.

4. Provide Reminder - `getCurrentReminders()`

This function is called every minute by the GUI to obtain a list of tasks for which the user needs to be informed about

5. Synchronise GUI - `resetStatus()`, `syncTask(Task&)`, `syncTaskList(list<Task>&)`

These functions are used to ensure that the GUI and the Manager are synchronised in terms of status and what is being displayed to the user currently. This is needed to allow the user to simply enter the index of a Task based on its placement in the current view to get its details.

State Dependence

The Manager is effectively the only segment other than the GUI that is aware of the current state of the program (for instance what the user is currently viewing on screen) and thus has to convert this data into stateless parameters to be passed into the Interpreter and Executor. In order to manage this, a few private attributes are used:-

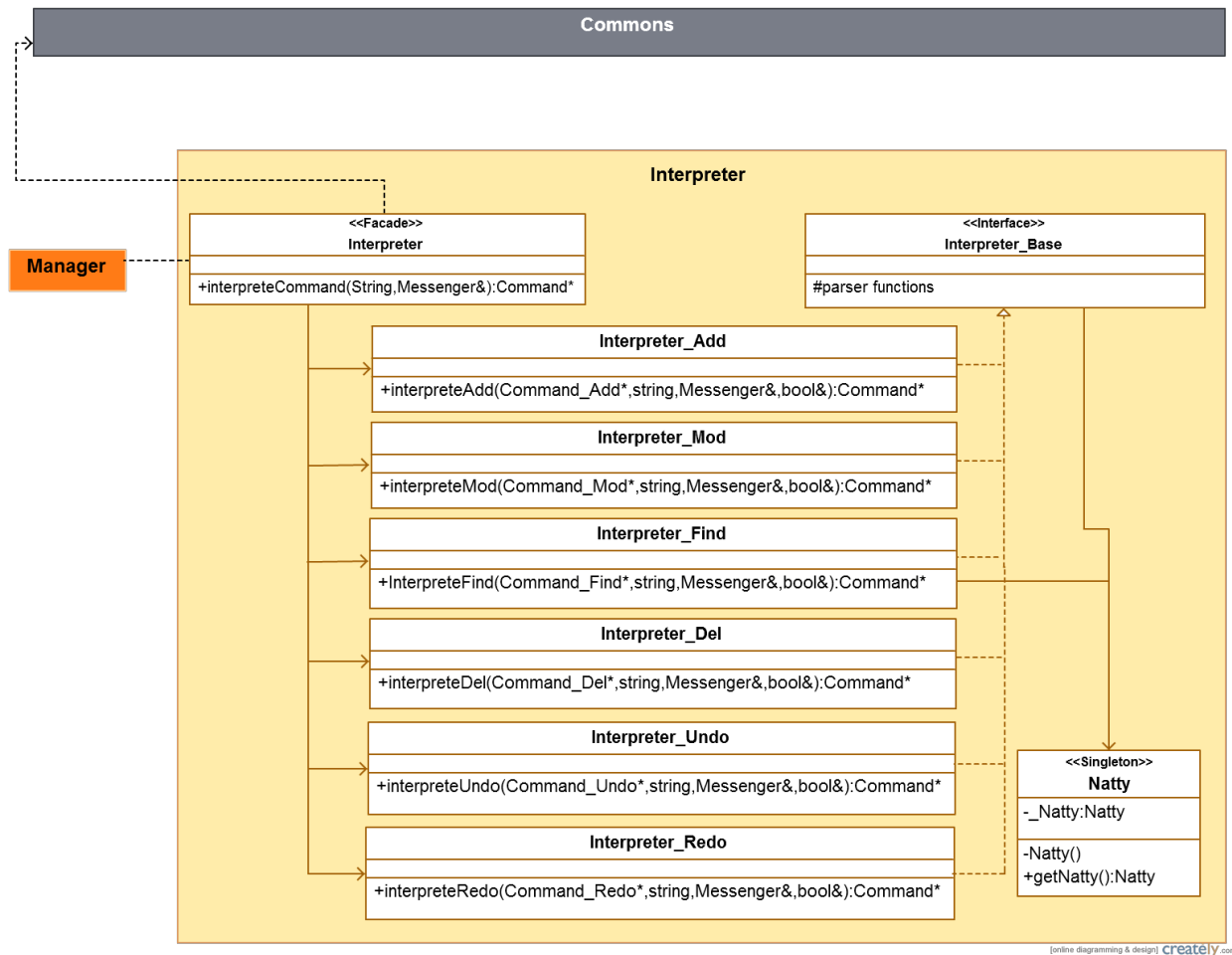
- The previous **Command** entered by the user is saved to allow for clarifications (if needed) without asking the user to retype the whole command again
- The previous **Messenger** object returned to the GUI is saved to be aware of what is being displayed to the user currently
- An **Integer** index value is stored to keep track of the specific Task whose details are being shown (if any)

- A ***pair<std:tm>*** (pair of timestamps) object that keeps track of the current period of time for which tasks are being displayed, example today's tasks or this month's tasks.

This way, the Executor and the Interpreter can be kept with minimal dependence on the TaskPad system's state, reducing coupling.

Interpreter

Below is the diagram for Interpreter class.



API

Command* interpretCommand(string command, Messenger& response)

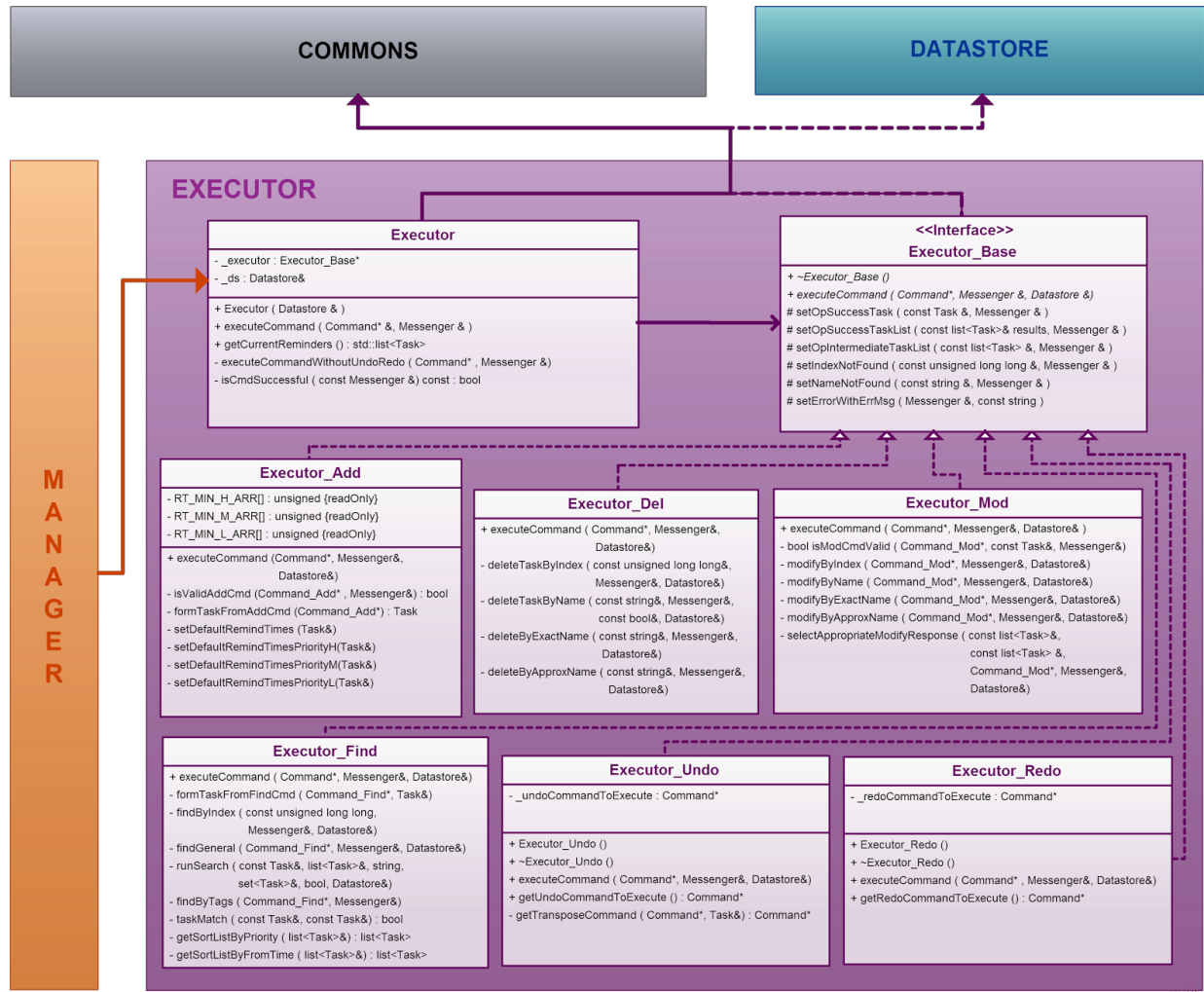
- This function gets the command string and Messenger reference from Manager. Then it will 'decipher' the command string and segments it into different parts according to the input key words. The segmented user inputs will then be allocated to the respective components in Command object, At the end of the function call, it returns the Command pointer and updates the 'SUCCESS' status to Messenger reference. In case of error, it returns NULL pointer and sets the status to 'ERROR'.

void interpretCommand(unsigned ActualIndex, Command* prevCommand)

- This function is only called when an "intermediate" state is encountered. The user input index, ActualIndex is parsed and set into the previous command object, prevCommand.

Executor

The Class diagram of the Executor component is as follows:



The Executor component is the heart of TaskPad. All inputs by the user or otherwise are “executed” and processed in this component. This component comprises of a facade class, Executor, an abstract base class Executor_Base, and 6 other classes that derive from the Executor_Base class and represent the 6 different operations that TaskPad currently supports. Additional types of operations can be easily added to TaskPad with minimal cost by simply deriving the new class from Executor_Base and changing the facade to handle the new class.

The Executor facade contains a Datastore object and an Executor_Base pointer. The Datastore object is passed in by the Manager on creation and is used for all changes to be made to data. The Executor_Base pointer is used for creating objects of one of the 6 Executor child classes depending on the desired command type and invoking their API for it.

API

Executor class provides two functions as the API.

Executor (Datastore &)

This is the constructor for the Executor facade class. It expects reference Datastore object, i.e., the Datastore on which to execute all the commands it receives.

void executeCommand (Command * cmd , Messenger & response)

This is the method of Executor facade that actually determines which type of Command it has received and creates an appropriate Executor object for that Command type. It then passes on control to this newly created object for further processing. The function can handle Add, Modify, Delete, Find, Undo and Redo commands. It returns four types of responses through the Messenger object :

1. STATUS:

This can be *SUCCESS*, *ERROR* or *INTERMEDIATE*.

- *SUCCESS* is returned when the given command was executed successfully.
- *ERROR* is returned when the given command could not be executed successfully due to reasons provided in the errMsg [string] property of the Messenger object.
- *INTERMEDIATE* is returned when the given command was executed partially due to the command not being specific enough and multiple Tasks specify the command parameters.

2. ERRMSG:

This property is used to provide an error message when an error is encountered while executing a given command.

3. TASK:

This property is used to return a singular Task object when a command was successful. The Task returned is the Task that the given command acted upon, after being updated.

4. TASKLIST:

This property is used to return a list of Tasks when the **STATUS** is *INTERMEDIATE* and more information is required from the user in order to successfully execute the command. The list contains all the Tasks that satisfy the given command parameters.

std::list<Task> getCurrentReminders()

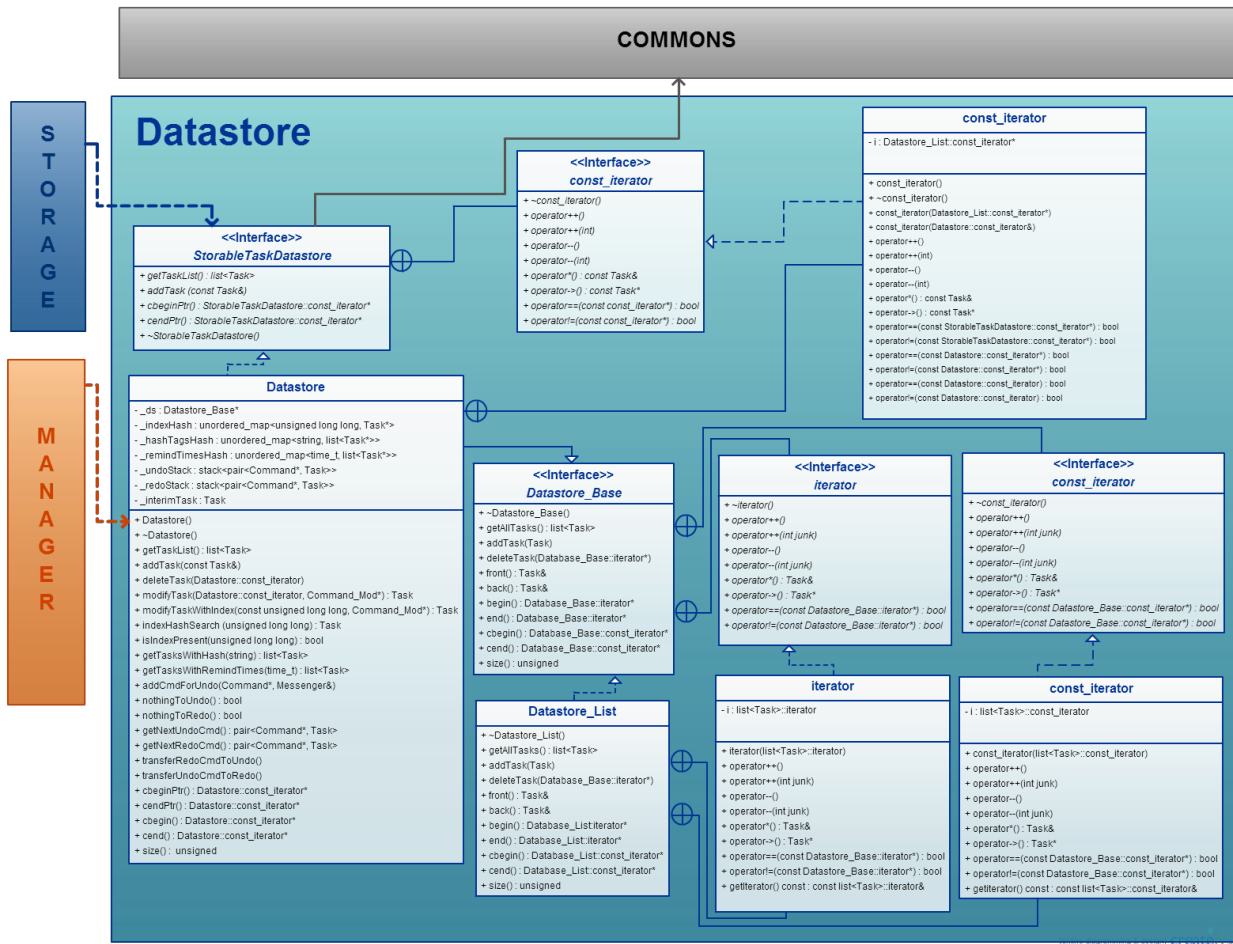
This method returns the list of Tasks that have a remind time which is within one minute of the current time.

The Executor component is fairly cohesive in that it handles all the command execution and that alone.

In terms of coupling, Executor has a dependency on the Commons [Task, Command, Messenger] classes and the Datastore component for handling any changes to the data. Other than these, it is not coupled with any other class in the entire system.

Datastore

The Class diagram of the Datastore component is as follows:



The Datastore component is, as its name suggests, a component where all the data of used by TaskPad is stored. It is highly flexible and supports very easy and cost effective addition and change of the underlying data structures being used for storing various data. Any and all changes to the data of the program must be done through this component.

This component comprises of an abstract `Datastore_Base` class that acts as the base class for any type of `Datastore` that is needed in the program. The current structure has only one type of `Datastore` implemented, namely, `Datastore_List`, that uses a `List` as the underlying data structure. All these classes come with their own iterator classes to enable easy traversal of data. Apart from these classes, there is the main `Datastore` facade. This facade derives from another specialized facade, `StorableTaskDatastore`. This base facade class has restricted functionality and hides other API from anyone who uses an object of this type, for example, the `Storage` component, which needs access to basic read/write functionality but not the full set of APIs that are utilized by the `Executor Component`. An important point to note is that these facade classes only provide constant iterators to prevent any external modification of the data.

API

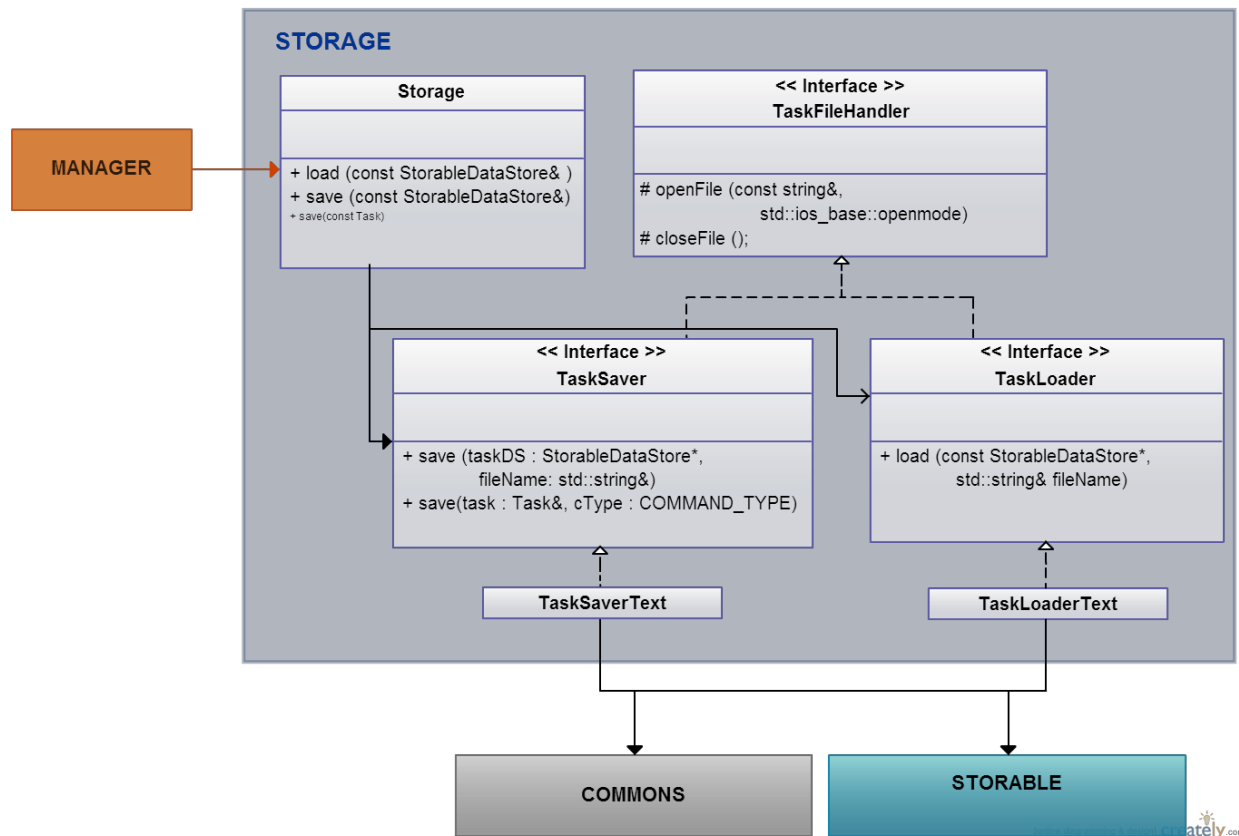
Datastore class provides API for doing all kinds of operations on the database. Moreover, it handles all the secondary tasks related with these changes to the data, like maintaining and updating the hashes and stacks being used for fast searches and for providing undo and redo functionality respectively.

The second kind of API provided is query API's that allow querying the various hashes for getting information about Tasks with particular attributes.

The third kind of API provided is that for manipulating the undo and redo functionality, for example, adding an executed command to the Undo Stack and retrieving a previously executed Command as part of an undo.

NOTE: The `StorableTaskDatastore` contains only a very small subset of the above APIs like getting a list of all task in the system and adding a Task to the system. This is because this class is intended for use by the Storage Component only.

Storage



The Storage Component does all the file handling and is the only segment in the whole program that edits the computer's file system. The facade class handles external API calls and reroutes it to the appropriate Class that is derived from TaskSaver (for saving) and TaskLoader (for loading). The above structure allows for easy expanding into handling files of other formats such as json/xml. The new classes simply need to inherit from TaskSaver/TaskLoader accordingly and implement the APIs defined in parent classes.

API

`bool save (const task&, const COMMAND_TYPE cType)`

This method creates a new `.task` file for add and modify commands and a `.deltask` file for delete commands. This is used as an intermediate quick storage to file as modifying the main may prove to be too slow when the program is actively used by the user. This serves as a quick backup of changes and also allows for recovery when the system crashes before changes are saved to the main storage file.

`bool save (StorableDataStore* ds, std::string& fileName)`

This method is used to dump the entire local list of tasks to the main storage file.

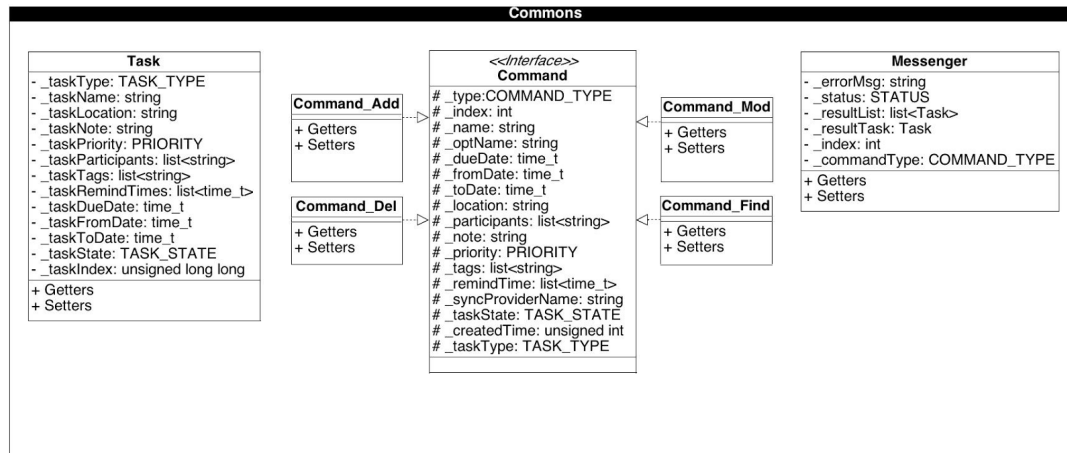
Principles

The following rules apply when a task is saved to file:-

- ❖ Every task starts with the label “StartOfTask” in a new line and ends with the label “EndOfTask” in a new line
- ❖ In between, all the attributes are saved in one line each with the corresponding label at the beginning of each line.
- ❖ For recurring attributes like ppl or reminder times, multiple line starting with the same label are entered.

An example file and available labels are in Appendix Storage A.

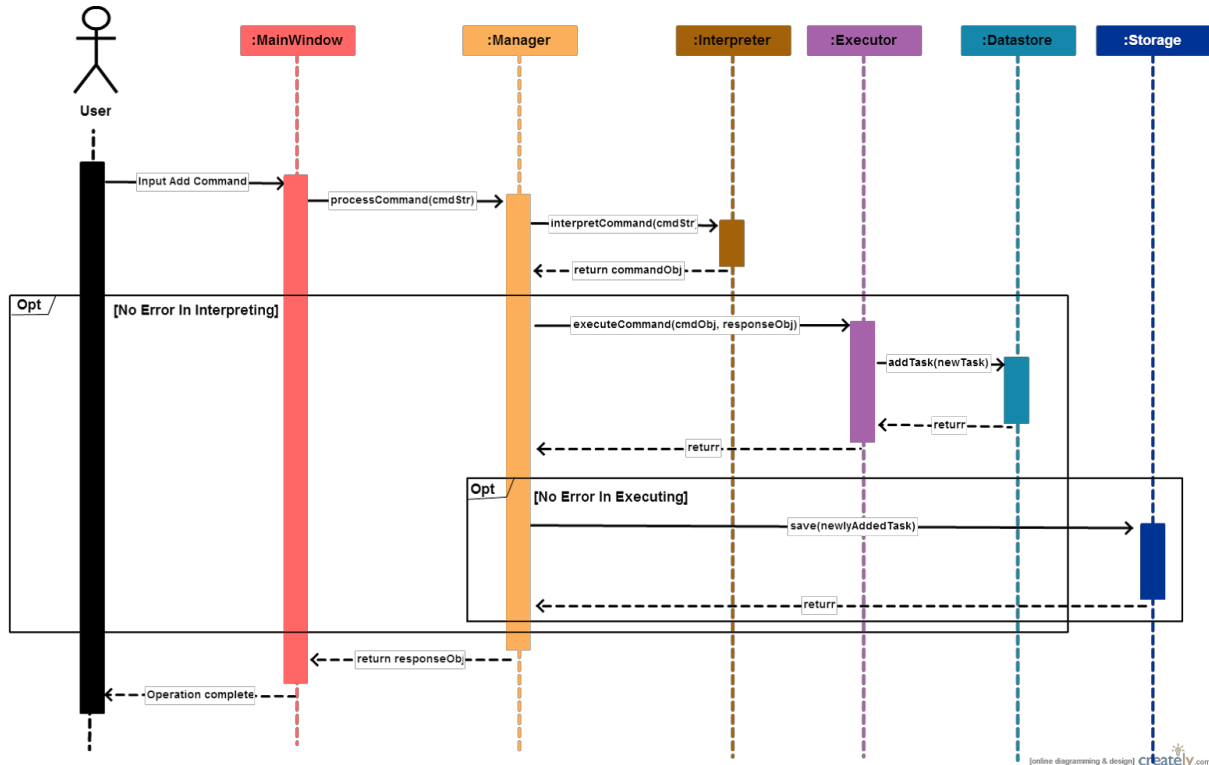
Commons



Common Component Class Overview

- **Command**: Provides the normal API for the Command component.
 - **Command_Add**: Provides specific APIs for the Add command.
 - **Command_Mod**: Provides specific APIs for the Modify command.
 - **Command_Del**: Provides specific APIs for the Delete command.
 - **Command_Find**: Provides specific APIs for the Find command.
- **Task**: Provides an abstraction for a Task in the system. It contains all the attributes required for a given task along with the associated getter and setter functions. It features automatic index generation for each Task that is created and type handling (DEADLINE, FLOATING, TIMED) depending on the appropriate
- **Messenger**: Provides an encapsulation for all the different data that need to be returned after processing a command
 - a list of Tasks to display
 - an integer specifying the Task in the list to be displayed in detail
 - a Task object that has just been edited/added/removed
 - an errorMsg explaining the issue if any
 - a `COMMAND_TYPE` to convey the type of command just executed
 - a `STATUS` object to keep track of the current Status of the system

Sequence Diagram for A Typical Command (*add a task*)



1. User types in an **Add** command, and it's captured by the MainWindow (UI) class.
2. MainWindow class uses the API provided by the Manager class, processCommand(cmdStr), to process the user input.
3. The Manager class then forwards the request to the Interpreter class using its API, interpretCommand(cmdStr).
4. The Interpreter class parses the user input into a Command object. This is then returned to the Manager class.
5. The Manager class receives the Command object. If the Interpreter class did not report any errors while interpreting, the Manager class forwards the Command object to the Executor class.
6. The Executor class then tries to execute the command described by the Command object.
7. The Executor figures out what Task corresponds to the Add Command object and class the Datastore class to actually perform the addition of the task to the data structure.
8. The Datastore class, in addition to adding the provided Task to the main data structure, updates the associated data structures like hashes for the new Task.
9. If the Executor succeeds in executing the command, the Manager class calls the save(newlyAddedTask) method provided by the Storage class to save the changes.
10. The Manager class then returns a responseObj that contains the appropriate feedback for user's input.
11. The MainWindow (UI) class receives this responseObj and feeds back to the user, appropriately.

Appendix TaskPad

The complete list of all the Attributes a task can have and its corresponding key words

Attribute	Keyword
Name	name
Due Date (and Time)	due, by
From Date (and Time)	from
To Date (and Time)	to
Location	at, place, location
Participants	ppl, with
Note	note
Priority	impt, priority
Tags	#
Reminder Times	rt, remind
Remove Due/From/To Date	-due, -from, -to
Modify Participants (add, remove, remove all, override)	+ppl, -ppl, -pplall, ppl
Modify Reminder Times (add, remove, remove all, override)	+rt, -rt, -rtall, rt
Modify Tags (add, remove, remove all, override)	+#xxx, -#xxx, -#, #xxx

Appendix Hotkey

Global Hotkeys

Alt + `	-- open quick add window
Ctrl + Alt + T	-- open main window

Local Hotkeys

Quick Add window

ESC	-- exit window
-----	----------------

Mainwindow

Ctrl + H	-- hide main window
Ctrl + T	-- show Today's view
Ctrl + I	-- show Inbox's view
Alt + 1	-- show Today's view
Alt + 2	-- show Inbox's view

// Date Navigation

Alt + D	-- show next day
Alt + Shift + D	-- show prev. day
Alt + W	-- show next week
Alt + Shift + W	-- show prev. day
Alt + M	-- show next month
Alt + Shift + M	-- show prev. month

CommandBar

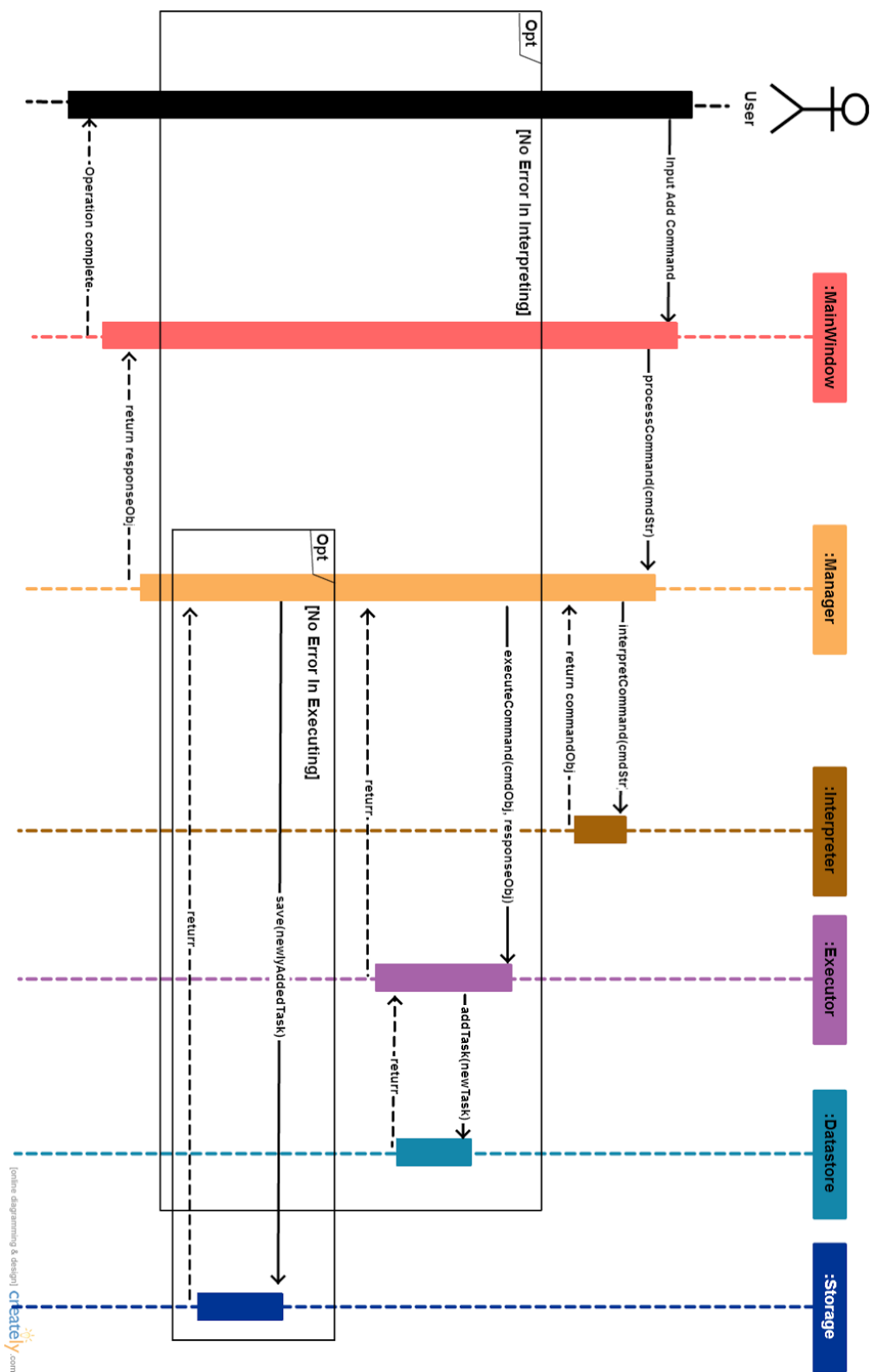
// Hotkey Template

Ctrl + N	-- new deadline task
Ctrl + Shift + N	-- new timed task
Ctrl + M	-- modify task as done
Ctrl + Shift + M	-- modify task by index
Ctrl + Alt + Shift + M	-- modify task by name
Ctrl + D	-- delete task by index
Ctrl + Shift + D	-- delete task by name
Ctrl + F	-- find task
Ctrl + U	-- undo
Ctrl + R	-- redo
Tab/Space	-- auto complete

/ in HotKey Template mode

Tab	-- jump to next blank
Shift + Tab	-- jump to prev. blank

Appendix Sequence Diagram



Appendix Storage

The labels used in the file are the same as the ones in Appendix TaskPad

-- Start of File --

```
StartOfTask
index: 101
name: newDeadline Task
due: 1382543146
at: somewhere
ppl: ppl 1
ppl: ppl 2
ppl: ppl 3
ppl: ppl 4
note: this is a sample note
impt: HIGH
rt: 1382111146
rt: 1382370346
state: UNDONE
EndOfTask
```

```
StartOfTask
index: 201
name: task 2
impt: MEDIUM
#: test1
#: test2
state: DONE
EndOfTask
```

--- End of File ---

The above format is used due to the following advantages:

- The user can easily edit the file and add/edit/remove attributes of a Task.
- In addition the labels are the same as the ones used in the command entered in the program and hence no new vocabulary is needed.
- This form of “label: data” structure gives tremendous stability.
 - It allows the user to enter the attributes with no specific order requirement. In fact the two reminder times in the above example file can be separated.
 - Empty lines in the file will not affect it
 - Unexpected strings will not cause the system to break. At worst, data for certain tasks may be lost due to mishandling of the file by the user.