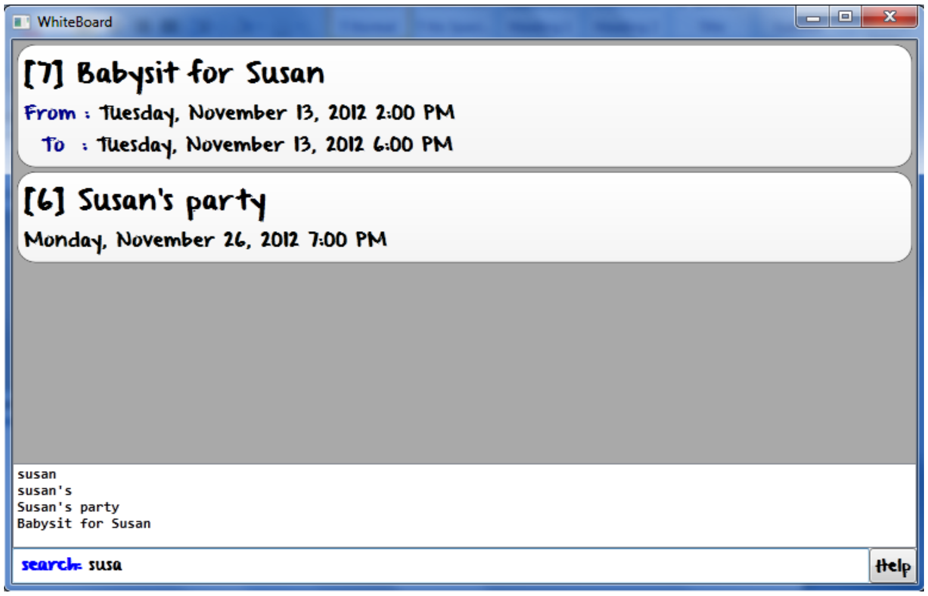
[T15-4S][V0.1]

WHITEBOARD™



[7] Babysit for Susan
From : Tuesday, November 13, 2012 2:00 PM
To : Tuesday, November 13, 2012 6:00 PM

[6] Susan's party
Monday, November 26, 2012 7:00 PM

```
susan
susan's
Susan's party
Babysit for Susan
```

search: susa

Salman Gadit Team lead, Developer, Bug fixer, External documentation	Varun Ganesh Developer, Tester, Product Documentation	Nizra Mohamed Nilufer Developer, Tester, Product Documentation	Saikrishnan Ranganathan Developer, Bug fixer, deadline watcher, External documentation

User Guide

Welcome to WhiteBoard, your own personal scheduler. Never miss a task again or lose track of your itinerary. It is extremely intuitive and easy to use, the application has been designed keeping in mind the user's experience as the utmost priority so that when using it and it behaves true to its name, like writing on a WhiteBoard.

The various functionalities of WhiteBoard and their usage has been explained and illustrated below.

So let's get started!

Adding Tasks:

Adding tasks in WhiteBoard is as simple as talking to it in a natural language. WhiteBoard supports tasks ranging between two dates or times, floating tasks (i.e. tasks with no date and time) and deadline tasks (i.e. tasks to be done before a specified date).

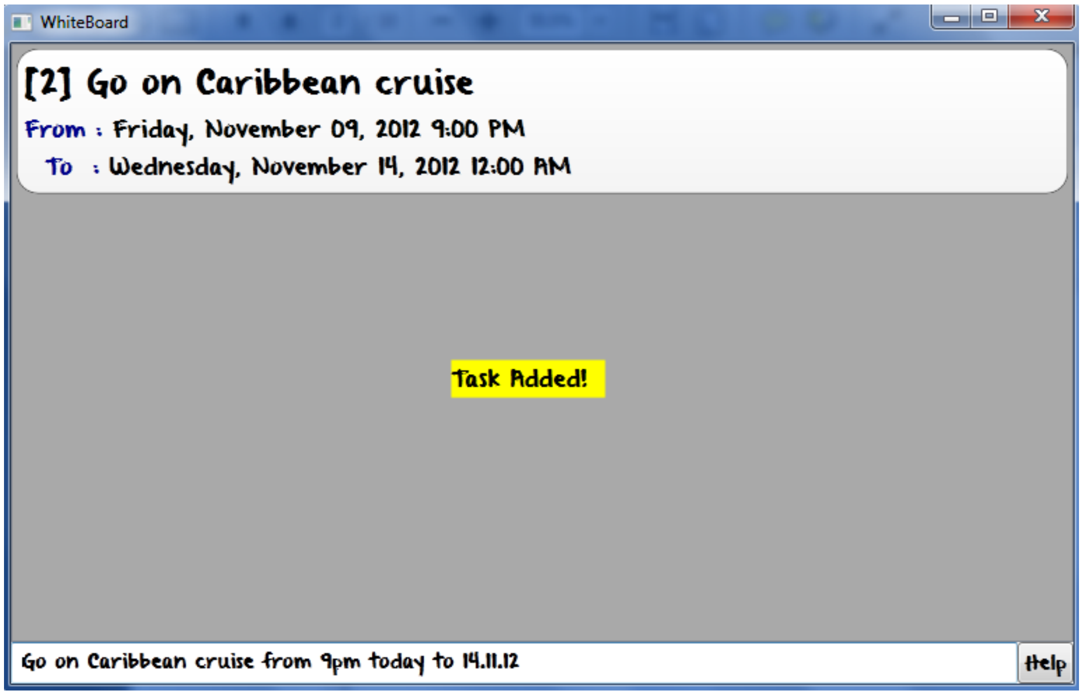
The general command format for adding tasks:

```
[DESCRIPTION OF TASK] BY/AT/ON/BEFORE [TIME AND/OR DATE]

[DESCRIPTION OF TASK]

[DESCRIPTION OF TASK] FROM/BETWEEN [TIME AND/OR DATE] TO/AND [TIME AND/OR DATE]
```

NOTE: Date could include any day of the week, words like today and tomorrow or in numerical date format.



[2] Go on Caribbean cruise
From : Friday, November 09, 2012 9:00 PM
To : Wednesday, November 14, 2012 12:00 AM

Task Added!

Go on Caribbean cruise from 9pm today to 14.11.12

In the above task the day Saturday could have been entered as a date form as well. Various date formats are supported - 09/11/2012, 9/11/12, 9-11-12 or 9.11.12 and so on. The time can also be entered in any way the user pleases. For example 730pm, 7:30pm, 7.30 pm or even as 24hr time i.e. 19:30

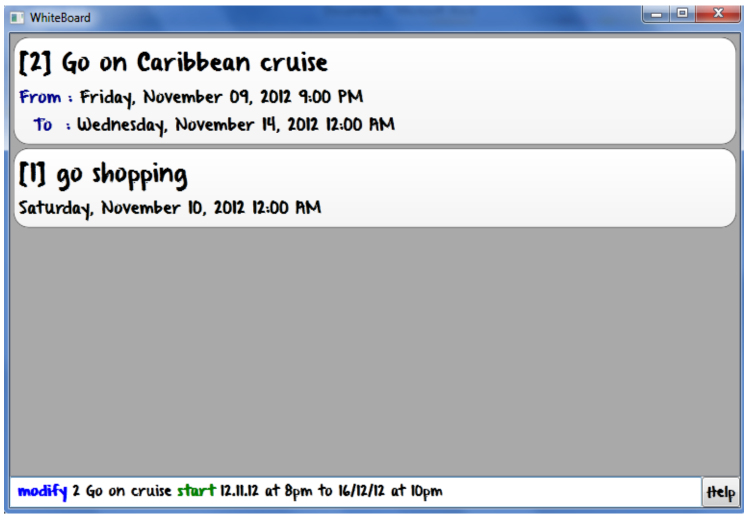
Modifying and Rescheduling Tasks:

No schedule assistant is complete without the ability to modify or change the timings and dates associated with a task. This process is greatly simplified in WhiteBoard. The general command format for modifying tasks:

MODIFY/UPDATE [TASK ID] [YOUR NEW TASK DESCRIPTION] START [NEW START DATE/TIME] END [NEW END DATE/TIME]

*Task ID is a unique integer associated with each individual task

*Apart from Task ID, the other fields are not compulsory, you may enter only the parameters you wish to change

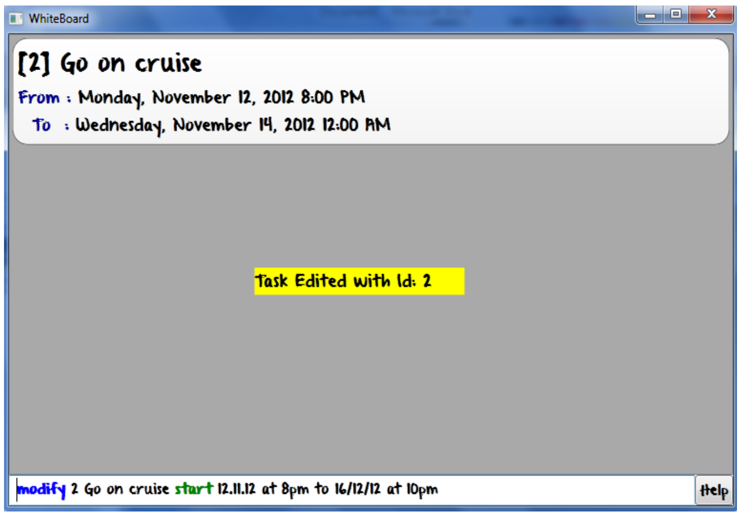


WhiteBoard

[2] Go on Caribbean cruise
From : Friday, November 09, 2012 9:00 PM
To : Wednesday, November 14, 2012 12:00 AM

[1] go shopping
Saturday, November 10, 2012 12:00 AM

modify 2 Go on cruise start 12.11.12 at 8pm to 16/12/12 at 10pm



WhiteBoard

[2] Go on cruise
From : Monday, November 12, 2012 8:00 PM
To : Wednesday, November 14, 2012 12:00 AM

Task Edited with Id: 2

modify 2 Go on cruise start 12.11.12 at 8pm to 16/12/12 at 10pm

Viewing Tasks:

View your itinerary with short and sweet commands. The general command format for viewing tasks:

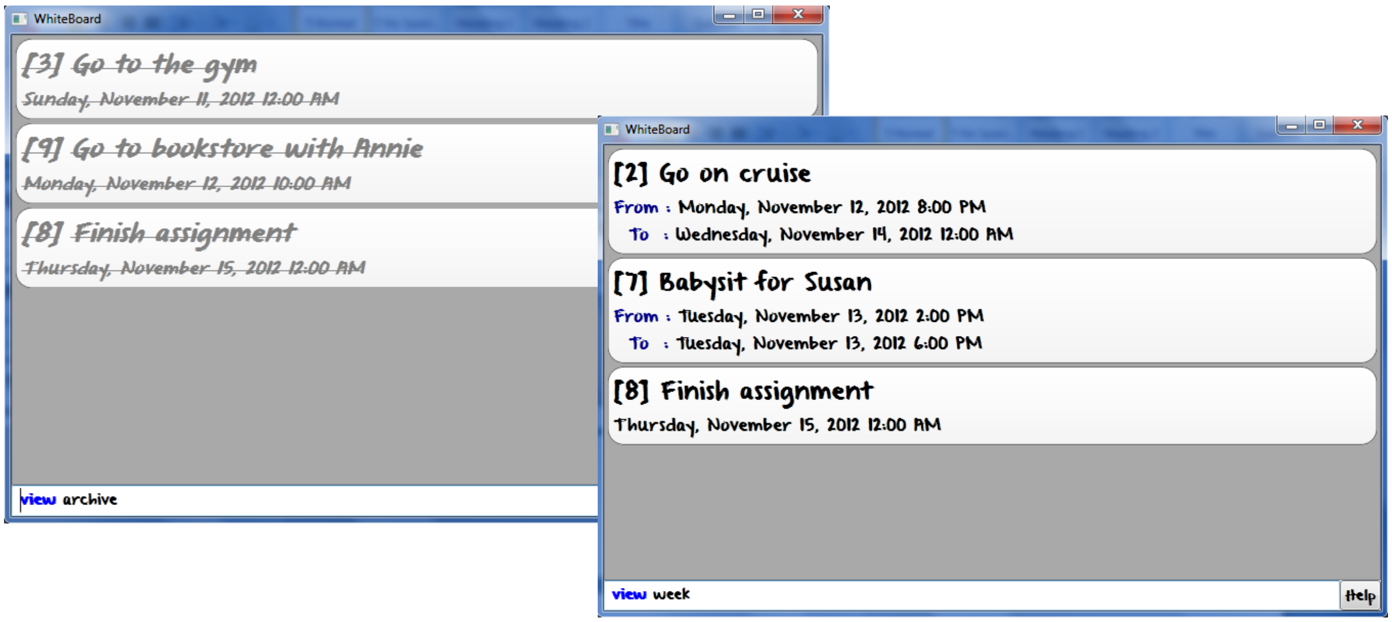
```
VIEW [TIME AND/OR DATE]

VIEW FROM [TIME AND/OR DATE] TO [TIME AND/OR DATE]

VIEW BY/ON/AT/BEFORE [TIME AND/OR DATE]
```

WhiteBoard offers the user some neat shortcuts to make viewing tasks a lot easier.

```
VIEW ALL – DISPLAYS ALL THE TASKS IN THE LIST SORTED IN ORDER OF DATE
VIEW TODAY – USER CAN LOOK AT THE SCHEDULE FOR THE CURRENT DAY
VIEW WEEK – SHOWS THE TASKS FOR THE CURRENT WEEK (I.E. MONDAY TO SUNDAY)
VIEW ARCHIVE – DISPLAYS ALL THE TASKS THAT THE USER HAS ARCHIVED AND MARKED AS DONE
```


WhiteBoard

~~[3] Go to the gym~~
~~Sunday, November 11, 2012 12:00 AM~~

~~[9] Go to bookstore with Annie~~
~~Monday, November 12, 2012 10:00 AM~~

~~[8] Finish assignment~~
~~Thursday, November 15, 2012 12:00 AM~~

view archive

WhiteBoard

[2] Go on cruise
From : Monday, November 12, 2012 8:00 PM
To : Wednesday, November 14, 2012 12:00 AM

[7] Babysit for Susan
From : Tuesday, November 13, 2012 2:00 PM
To : Tuesday, November 13, 2012 6:00 PM

[8] Finish assignment
Thursday, November 15, 2012 12:00 AM

view week

Deleting and Archiving Tasks:

Remove or delete unwanted tasks in a jiffy. Just specify the task ID:

```
DELETE/REMOVE [TASK ID]
```

Once a task is completed, if the user does not want to delete it forever, it can be archived as follows:

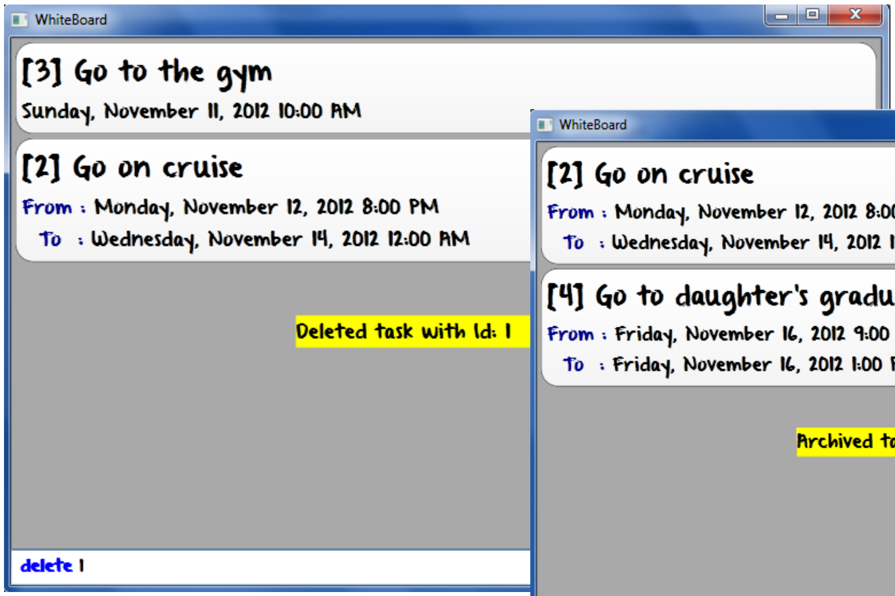
```
MARK [TASK ID] DONE/AS DONE
```

To un-archive a task you may simply mark it as not done, however this must follow a "VIEW ARCHIVE" command to first display all the archived tasks

```
MARK [TASK ID] AS NOT DONE
```

Since WhiteBoard is all about convenience and making the user's life easier, we've included a way to delete or archive all the tasks currently displayed on the user's screen:

```
DELETE/REMOVE ALL
MARK ALL AS DONE
```

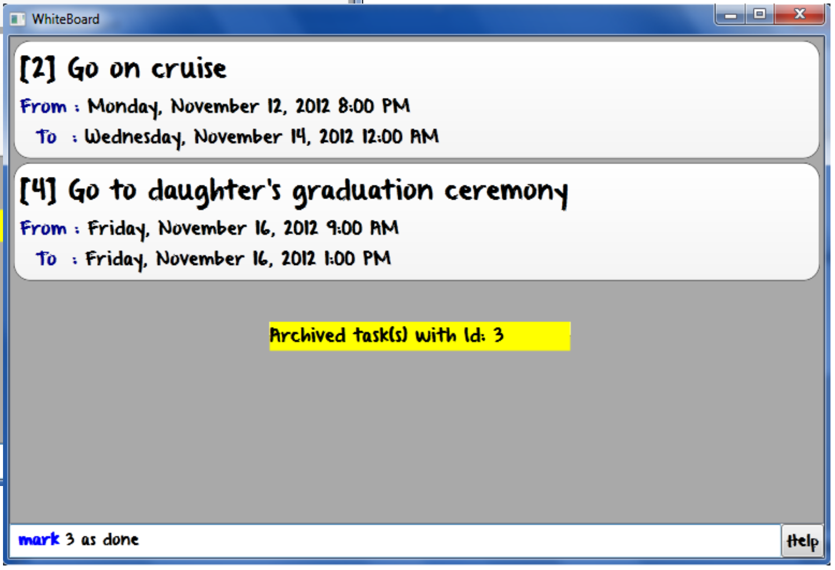

WhiteBoard

[3] Go to the gym
Sunday, November 11, 2012 10:00 AM

[2] Go on cruise
From : Monday, November 12, 2012 8:00 PM
To : Wednesday, November 14, 2012 12:00 AM

Deleted task with Id: 1

delete 1


WhiteBoard

[2] Go on cruise
From : Monday, November 12, 2012 8:00 PM
To : Wednesday, November 14, 2012 12:00 AM

[4] Go to daughter's graduation ceremony
From : Friday, November 16, 2012 9:00 AM
To : Friday, November 16, 2012 1:00 PM

Archived task(s) with Id: 3

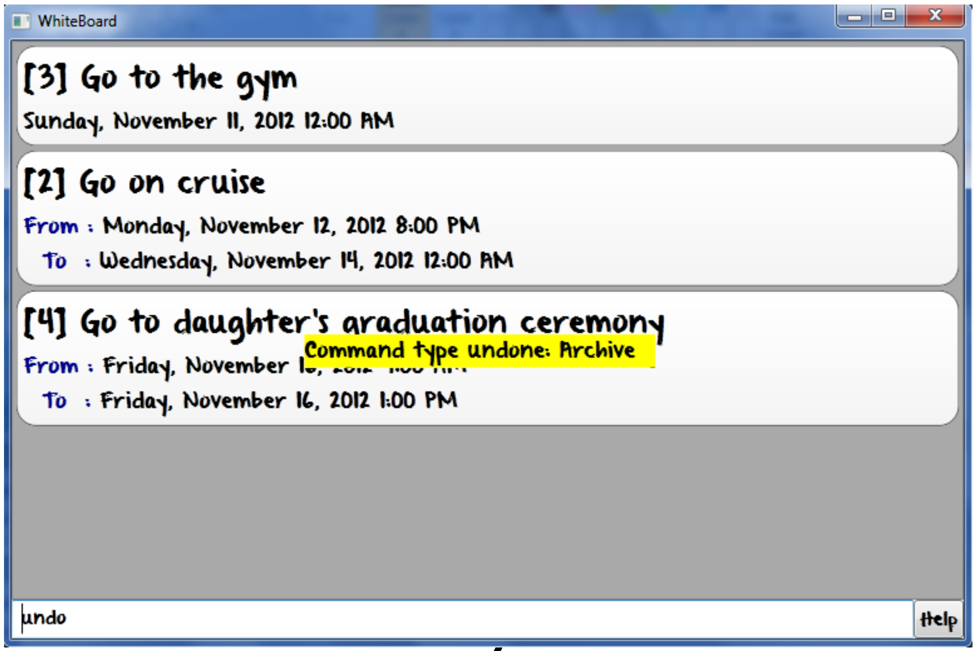
mark 3 as done

Undo:

Deleted a task by mistake? Added a wrong task?

```
UNDO
```

No problem just hit CTRL+Z or type UNDO. WhiteBoard's undo command works seamlessly with every action performed by the user. Be it adding, modifying, viewing or deleting tasks.


WhiteBoard

[3] Go to the gym
Sunday, November 11, 2012 12:00 AM

[2] Go on cruise
From : Monday, November 12, 2012 8:00 PM
To : Wednesday, November 14, 2012 12:00 AM

[4] Go to daughter's graduation ceremony
From : Friday, November 1
To : Friday, November 16, 2012 1:00 PM

Command type undone: Archive

undo

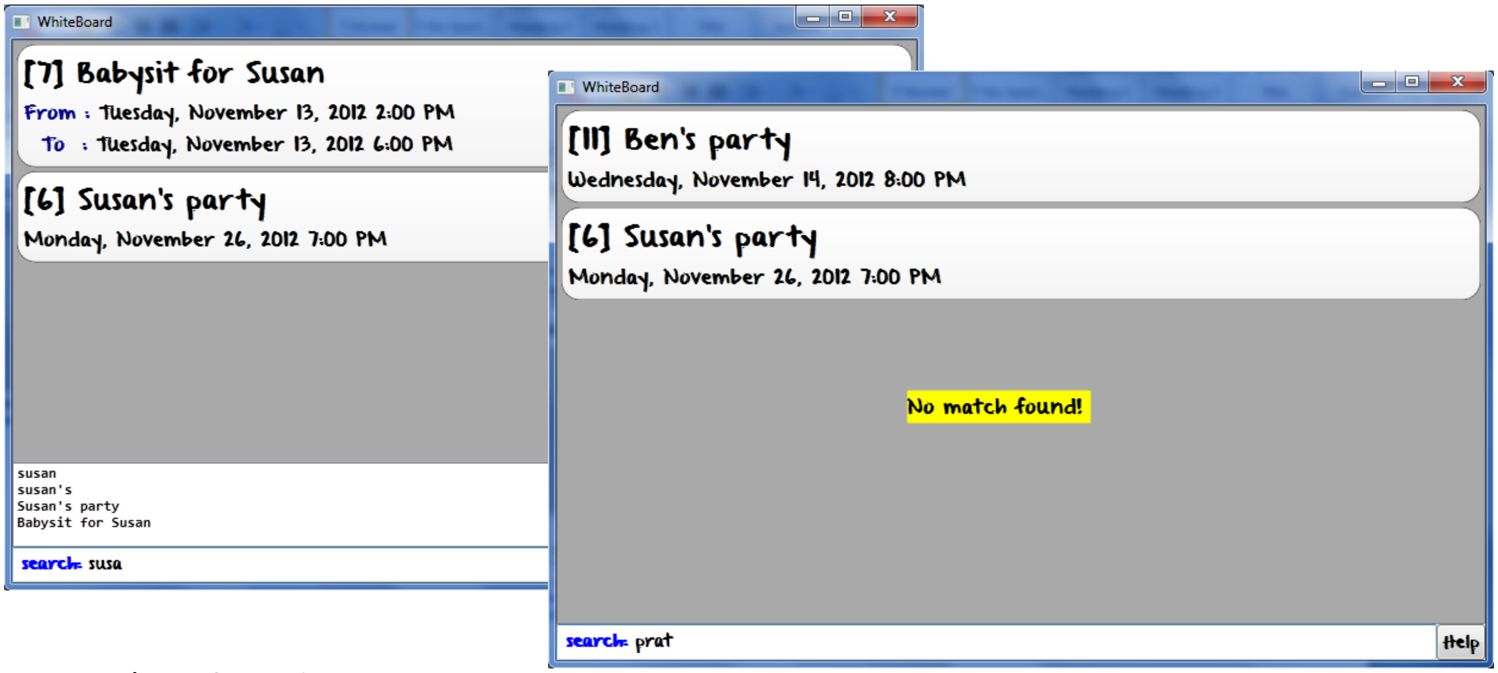
SuperSleuth Search:

Last but not least, users can take our SuperSleuth search for a spin.

```
SEARCH: [KEYWORD]
```

SuperSleuth will auto filter search results as the user types and display them on the screen.

What's more? SuperSleuth Search even pardons the user's spelling errors by providing near-miss results.


WhiteBoard

[7] Babysit for Susan
From : Tuesday, November 13, 2012 2:00 PM
To : Tuesday, November 13, 2012 6:00 PM

[6] Susan's party
Monday, November 26, 2012 7:00 PM

```
susan
susan's
Susan's party
Babysit for Susan
```

search: susa

WhiteBoard

[11] Ben's party
Wednesday, November 14, 2012 8:00 PM

[6] Susan's party
Monday, November 26, 2012 7:00 PM

No match found!

search: prat

Other Cool Features:

Command History:

In the command bar, the user can hit the up or down arrows to bring up the previous commands entered. This saves time and also helps the user remember action was performed last.

Syntax Highlighting:

Keywords are highlighted in different colours. This shows the user that it is a keyword and helps minimize syntactical errors.

Toasts:

Helpful toast messages appear briefly on screen after the user enters a command. This notifies the user about whether the action was successful or any errors were encountered.

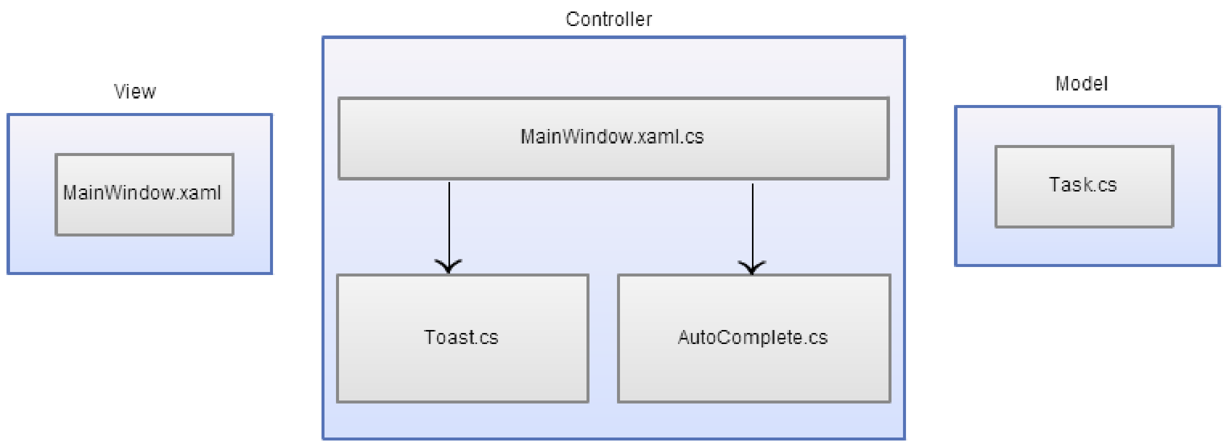
Developer Guide

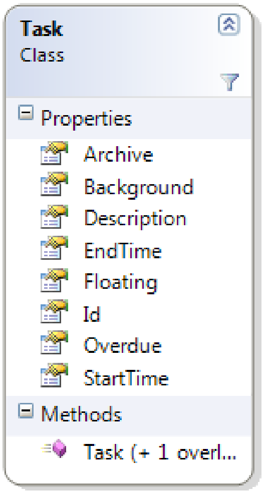
Welcome to the team! As a new developer this guide will help you understand the architecture, design patterns and the essential classes that make WHITEBOARD™ happen.

Architecture and Design Patterns

Architecture

WHITEBOARD™ follows an MVC architecture where the view (our beautiful UI) has been designed ground up using WPF. MainWindow.xaml plays the role of **View**, MainWindow.xaml.cs plays the role of Controller and Task.cs plays the role of **Model**.





Task
Class

Properties
- Archive
- Background
- Description
- EndTime
- Floating
- Id
- Overdue
- StartTime

Methods
- Task (+ 1 overl...

Model - Task Class

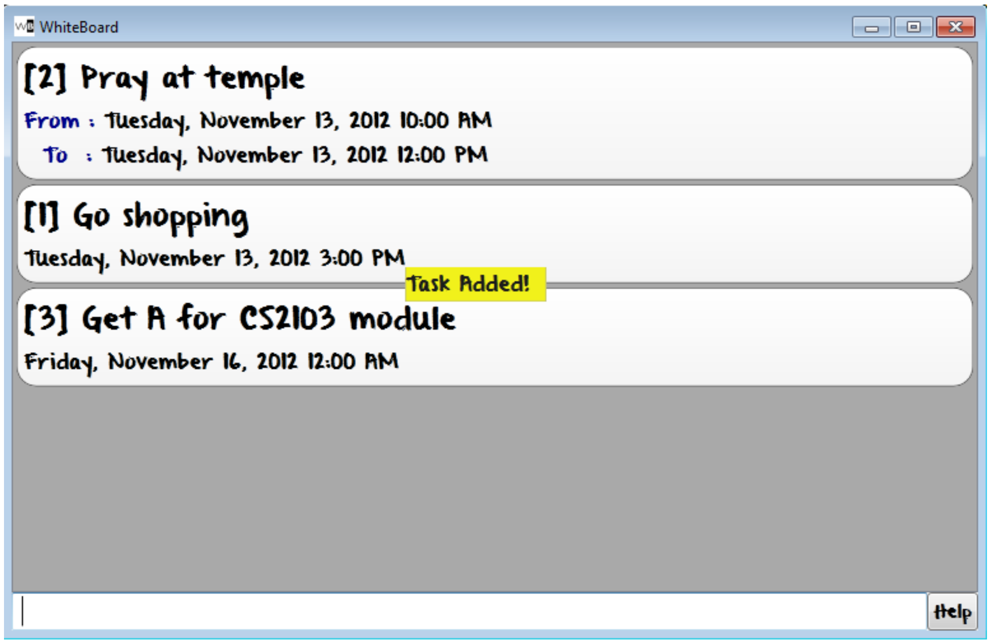
The Task class in WHITEBOARD™ is the core model of the application, it contains all the data about the users tasks/todos. The tasks are serialized and persisted to an XML file with the help of the FileHandler, which will be discussed in a later section.

View - UI

WHITEBOARD™ comes with a beautiful simple User Interface created in Windows Presentation Foundation (WPF 4.0), with no fancy interactions. It revolves around list of tasks and a text box to enter commands. Each list box item is data-templated with labels for the different fields that a are contained in the Task object.

We use data-binding to populate the list box with the required list of tasks. Data-binding makes it easy to bind a container on WPF with a List or Collection that implements the IEnumerable interface. Very simply, once the data context has been set to the list box, all it requires is a simple refresh to update the view.

Every action or command entered in the text box is responded with a small notification - called a *Toast* on the Android platform. Hence, we have also called it a toast notification.



WhiteBoard

[2] Pray at temple
From : Tuesday, November 13, 2012 10:00 AM
To : Tuesday, November 13, 2012 12:00 PM

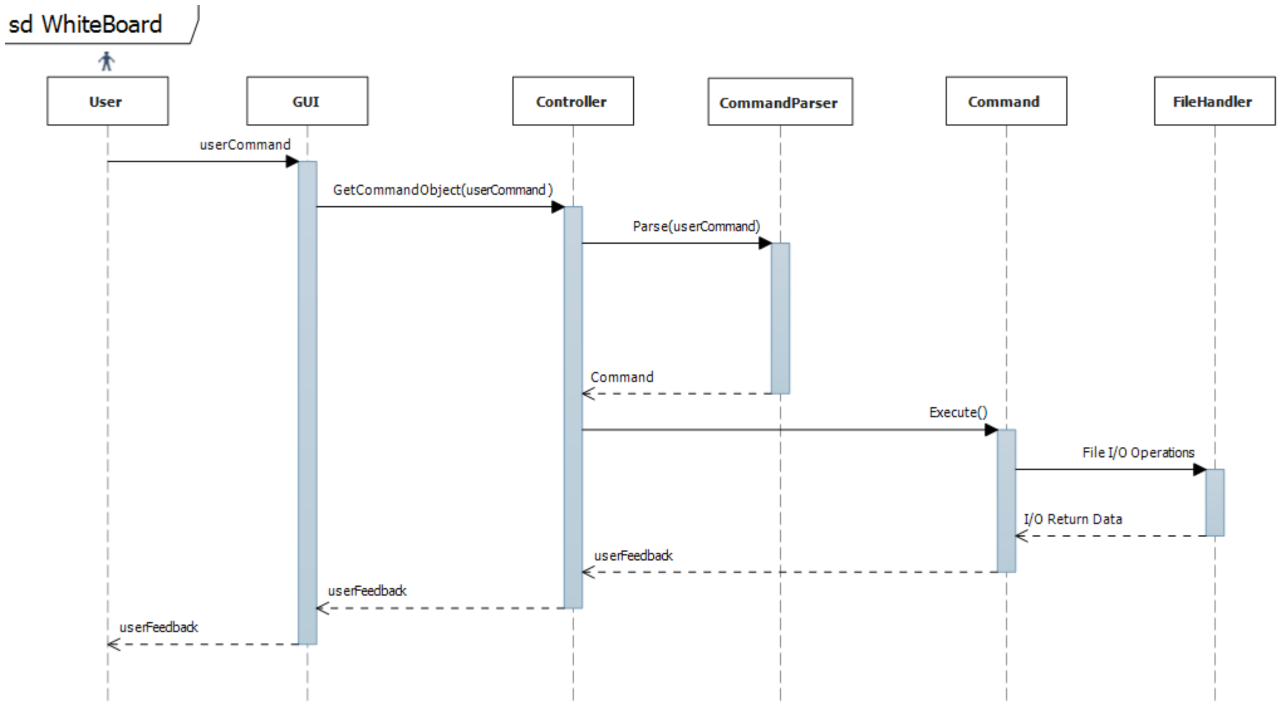
[1] Go shopping
Tuesday, November 13, 2012 3:00 PM

Task Added!

[3] Get A for CS2103 module
Friday, November 16, 2012 12:00 AM

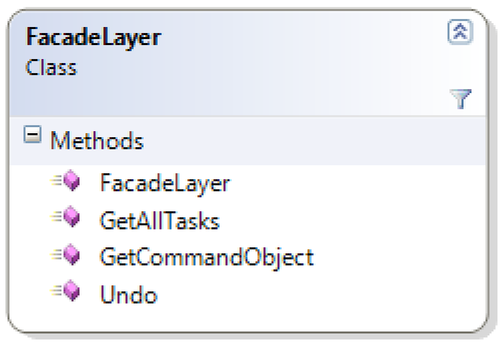
The toast is implemented as a hidden label with dynamically adjustable width. When required, the text item for the toast is set after every command action and the toast label is made visible with an animation. The animation is timed control of its Opacity, giving it the fade in and fade out effect like an actual toast notification.

General Program Flow



WHITEBOARD™ takes in command from the user through its elegant UI and sends it over to the FacadeLayer which then calls CommandParser to process the natural language like command. The CommandParser decides which Command object to instantiate and returns it back to the FacadeLayer, the FacadeLayer then called the abstract method Execute() on the Command which returns the feedback to be given back to the user. Thus the FacadeLayer class effectively creates a single point of entry and exit for the GUI.

Know Thy Classes



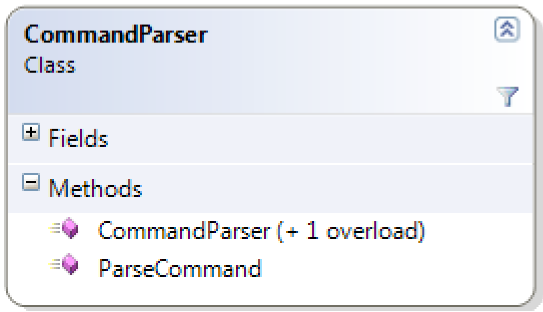
FacadeLayer

The FacadeLayer class uses the Facade pattern and and provides a single point of entry to the User Interface(View) to interact with the Command Parser.

The GetCommandObject() returns a Command object to execute for the given user String and the return value of executing that Command object is the feedback to be displayed to the user.

The GetAllTasks() method is used at startup to display all of the users tasks.

The Undo() method will undo the last user command. If you are intrigued about the Undo algorithm, please refer to the section Algorithms below.



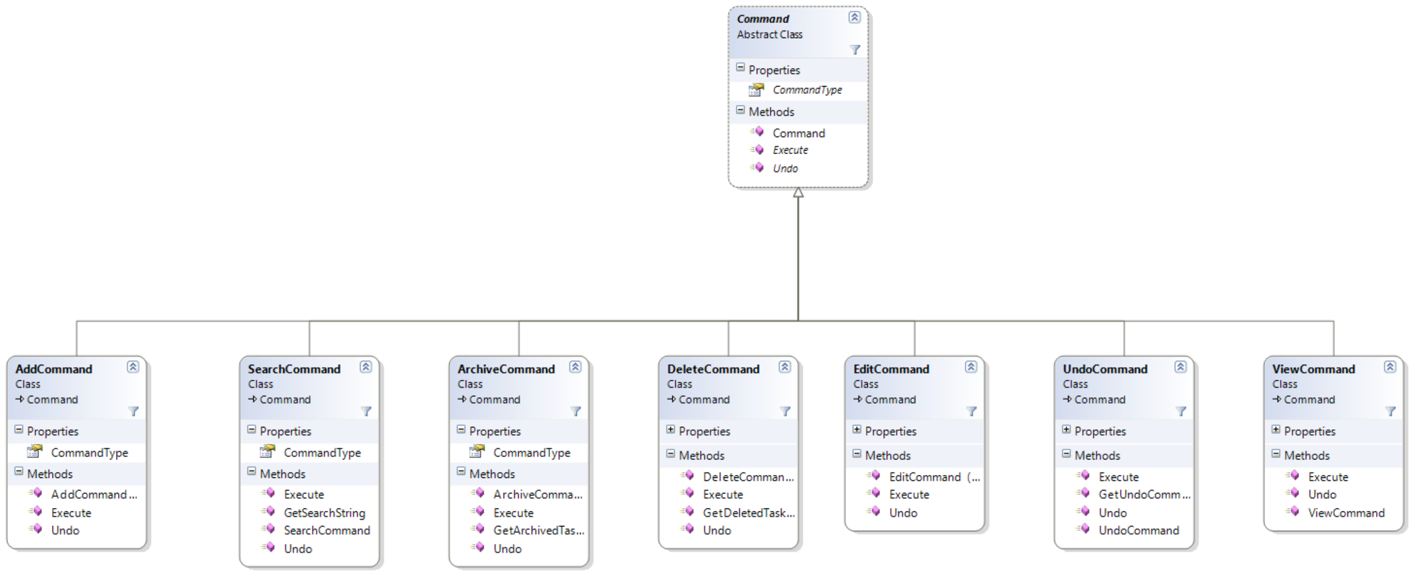
Command Parser

The CommandParser class is responsible for parsing the highly intuitive and user friendly commands that WHITEBOARD™ can understand.

The ParseCommand() method accepts the string as a parameter, which is the command input by the user. The method parses this string for information and decides which operation to perform.

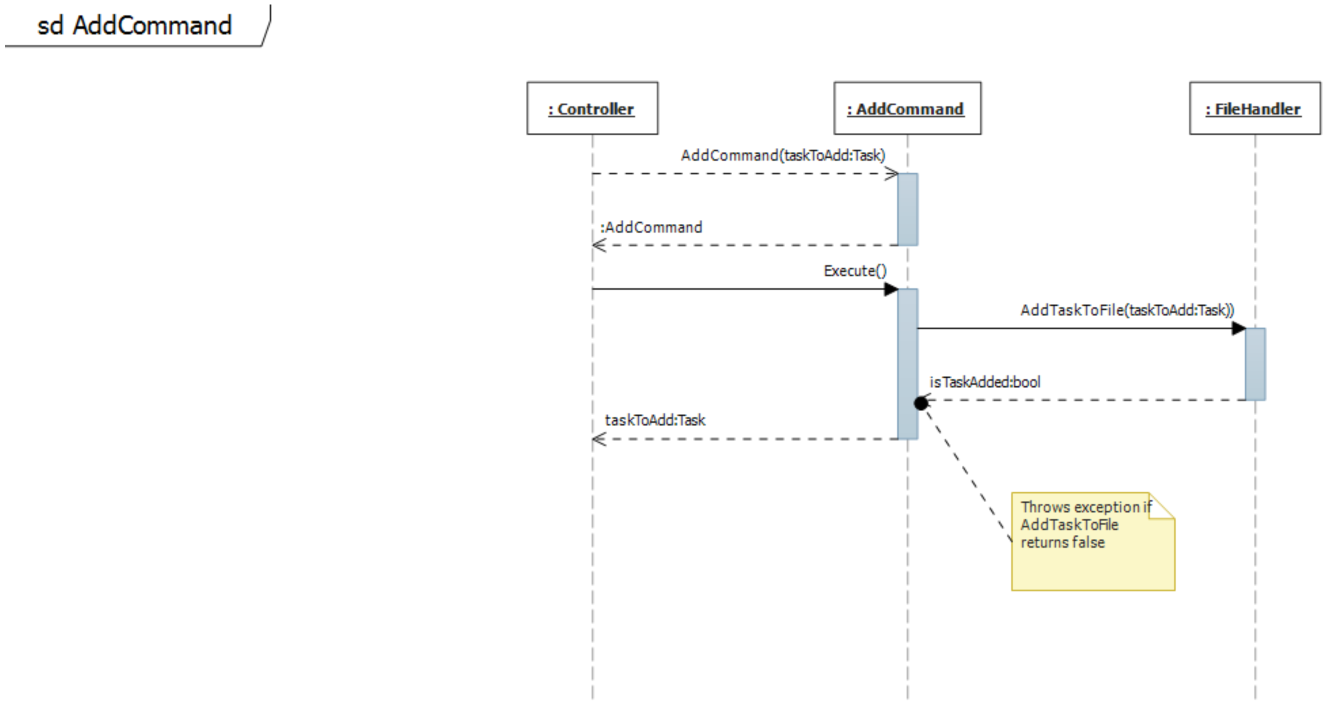
It consequently returns the appropriate Command object containing the fields necessary for performing the operation (i.e. AddCommand, ArchiveCommand, DeleteCommand, EditCommand, SearchCommand, UnarchiveCommand, UndoCommand or ViewCommand).

Command and its children

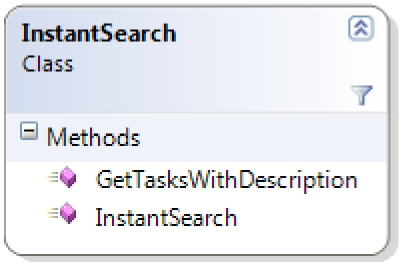


All user initiated actions in WHITEBOARD™ trigger the Command Parser to create Command objects such as AddCommand, EditCommand etc. and return them back to the FacadeLayer which then calls the abstract method Execute() that contains all the Command specific business logic. The Execute command returns the feedback in the form of a list of Task objects to the FacadeLayer class which is then feedbacked to the user through the View.

An example sequence diagram for AddCommand is shown below, the other Command objects follow a similar pattern.


sd AddCommand
: Controller
: AddCommand
: FileHandler
AddCommand(taskToAdd:Task)
:AddCommand
Execute()
AddTaskToFile(taskToAdd:Task))
isTaskAdded:bool
taskToAdd:Task
Throws exception if AddTaskToFile returns false

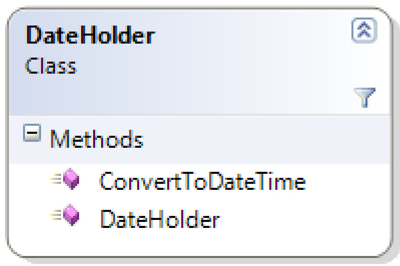
Toast



The Toast class contains methods to display the toast notification at the end of a task or response from the back-end of the system. The Toast notification flashes for about 2- 3 seconds before fading out. The toast can be called by accessing the void method ShowToast by sending in the message to display as the parameter. This ensures that even this component of the system remains Object Oriented.

Instant Search

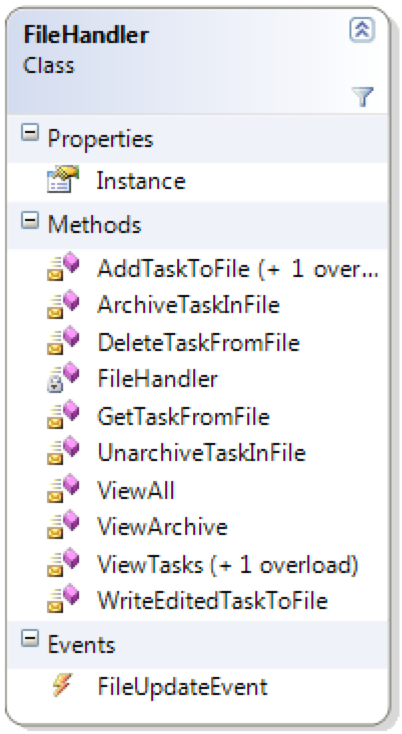
The Instant Search class is responsible for the filtered results on screen as you search. It implements the Observer pattern where the FileHandler class plays the role of the subject and notifies InstantSearch whenever a Task is added/deleted/edited/archived/unarchived, Instant Search then updates its query set accordingly. The GetTasksWithDescription() method returns a list of Tasks which contains the given search string.



Date Holder

The DateHolder class was designed to act as a helper class to CommandParser. Essentially it consists of a string, containing the date information entered by the user and a date id denoting whether the date string is in the form of a numerical date or as a day of the week.

The public method ConvertToDateTIme() is then used to parse the date string according to its id and return the correspoding DateTime object.



FileHandler

Data Storage in WHITEBOARD™ is managed by the process of serialisation. The list of tasks are stored into an XML file in the backend, by the process of serialisation through C#. To retrieve the XML object and convert back into the list of Tasks, a process of deserialisation is used. The file handler manages the actions of add, edit, remove, archive and delete. The general sequence of doing any of these tasks is:

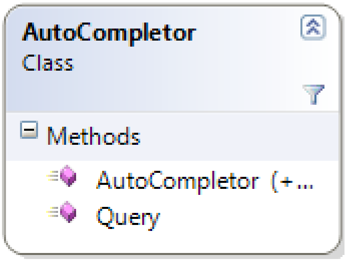
1. Deserialise XML file into a List of Task objects
2. Perform required action on the list
3. Serialise and overwrite with the new XML file

FileHandler is the subject of the Observer pattern implemented in the InstantSearch and Autocompletor class, it triggers the FileUpdateEvent whenever a Task is added/deleted/edited/archived/unarchived. The Task object and the type of update is passed to the listeners in the observing classes.

- AddTaskToFile method: The task is added to xml file by deserializing the xml file into a list of task objects, adding the task to be added to the list, and serializing the task object list into the xml file.
- GetTaskFromFile method: Task object is retrieved from file for editing by deserializing the entire xml file into a list of task objects and then matching the task id of the required task. The required task object is then returned.

- WriteEditedTaskToFile method: Edited task is written back to the file by deserializing the entire xml file into a list of objects, looking for the task object with matching task id, replacing it with the edited task object and then serializing the task object list into the xml file.
- DeleteTaskFromFile method: Task is deleted from the xml file by deserializing the entire file into a list of task objects and then removing it from the list. The list of objects is then serialized into the xml file.
- ArchiveTaskInFile method: Task is archived in file by deserializing the xml file into a list of task objects, setting the archive flag of the task to be archived to true and serializing the list to the xml file.
- UnarchiveTaskInFile method: Task is unarchived in the file by deserializing the xml file into a list of task objects, setting the archive flag of the task to be archived to false and serializing the list to the xml file.
- ViewAll method: Deserializes the entire xml file into a list of task objects. Puts the unarchived tasks into a separate list and returns it.
- ViewArchive method: Deserializes the entire xml file into a list of task objects. Puts the archived tasks into a separate list and returns it.
- ViewTasks method: Deserializes the entire xml file into a list of task objects. Puts the tasks for the date or date range in a separate list and returns it.

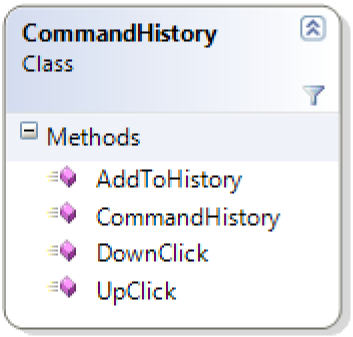
AutoCompletor



The AutoCompletor class is responsible for suggesting words and sentences as you search. It also implements the Observer pattern where the FileHandler class plays the role of the subject and notifies AutoCompletor whenever a Task is added/deleted/edited/archived/unarchived, AutoCompletor then updates its query set accordingly.

The Query() method returns suggested words and sentences for a given search string.

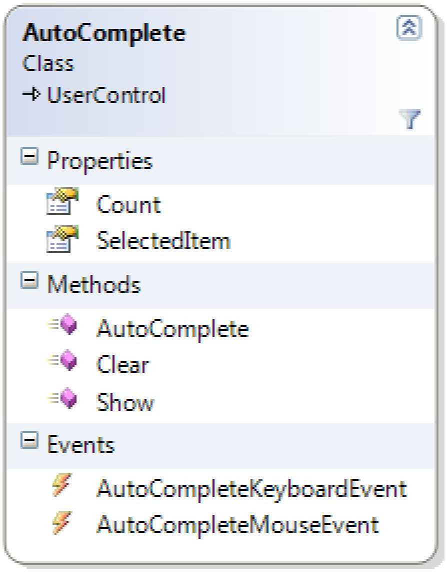
Command History



The Command History class stores the history of commands typed by the user in one session. Each command is stored after it has been entered into the system and is inserted into the Command History object using the AddToHistory method. UpClick() and DownClick() are public methods that help the GUI layer call these methods when the user presses the respective arrow key. These methods then return the appropriate string for the command history.

AutoComplete

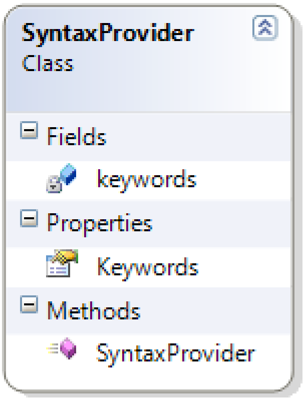
AutoComplete is a custom WPF user control that we created for the purpose of making search easier. It comprises of a listbox which is dynamically bound to a list in the AutoComplete.xaml.cs file and refreshes dynamically as the list is updated.



The public method Show() populates the user control with the correct information and positions it correctly in the main UI. The Clear() method is an api to clear the list. This user control also has public properties that can be accessed by the instantiator. Count returns a count of the number of elements currently in the list while SelectedItem returns the string that is selected by the user and null if nothing.

There are also two delegate methods which help in the implementation of an Observer pattern for this user control. But handling these events in the UI we place listeners for when the user selects an item on the list, whether by keyboard or selection by mouse. This maintains a seamless separation between the parent layer of the user control, and the user control itself.

SyntaxProvider



Syntax Provider is a simple central point to store all the keywords that the command parser and the syntax highlighting need. By making it a central class it removes the need of having to triple check multiple declarations across the code. The list of keywords can be accessed by calling the Keywords property.

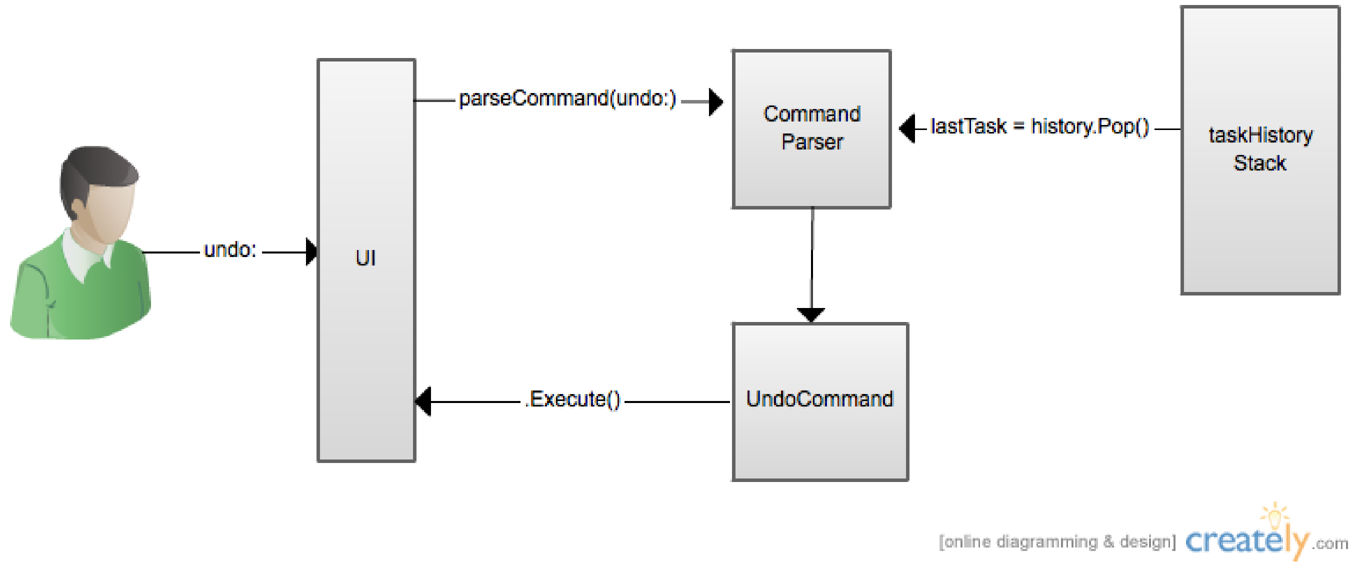
Important Algorithms

Undo function

The undo functionality for WHITEBOARD™ is postulated under two expectations:

1. Every operation has a screen-state, i.e. whatever tasks that the user is seeing on screen. The operation can modify this screen state, so this screen state should be stored.
2. Every undo operation should, therefore, perform the *opposite* task that the previous command did, and at the same time time, restore the screen state to the previous version.

To make this work, we had to ensure that each command object stores the previous screen state by default. We also had to provide a unique behaviour for each command object's undo operation and therefore use the abstract method undo, which performs the undo task and returns the previous screen state. This result is executed via the UndoCommand object which then data binds the screen state to the main UI. All actions are stored on a global 'taskHistory' stack, and every undo operation retrieves the latest command object through that.



Near Miss Search

The SearchCommand in WHITEBOARD™ uses a very efficient near miss search algorithm that compares the likeness of two strings by a factor known as edit distance. Edit distance is the number of edits required to convert one string to another, these include insertion, deletion and substitution of characters.

While converting "mee" to "meet" we insert "t" at the end of the string, giving us an edit distance of 1. Similarly while converting "meem" to "meet", we are substituting "m" with a “t”, giving us an edit distance of 1 and finally by converting “meeetin” to “meeting” we delete a single character “e” and add a “g” to the end, giving us an edit distance of 2.

Let us see how we implement this algorithm, say S is the array containing the source string and T is the array containing the target string, then NS is the number of characters in S and NT is the number of characters in T. Let us number the indices of the array from 1 to ‘ns’ and 1 to ‘nt’ respectively.

Suppose we are examining T[i] and S[j] chars, we already "know" the edit distance for substrings S[1..j] and T[1..i-1], and for substrings S[1..j-1] and T[1..i-1], and for S[1..j-1] and T[1..i]. Then we consider the following cases:

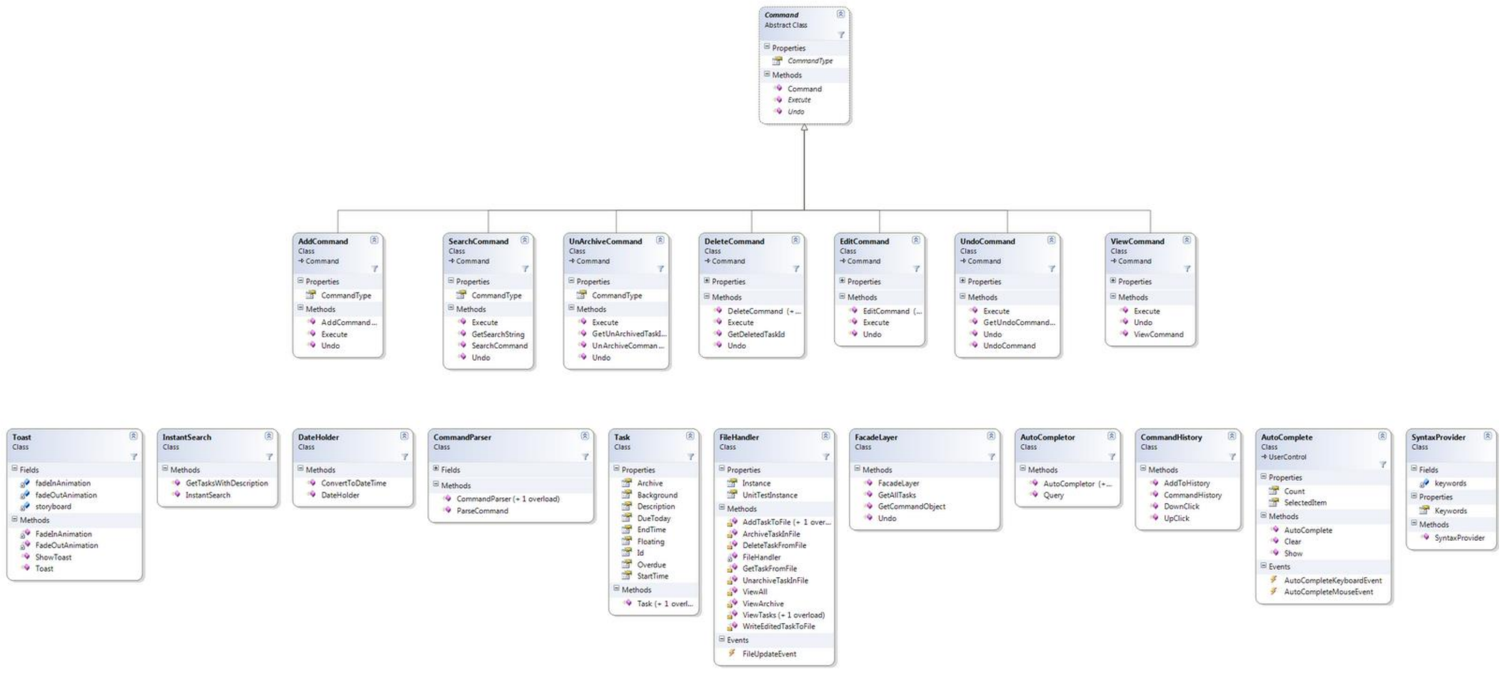
1. Suppose, S[j] is the same as T[i], then edit distance involving S[1..j] and T[1..i] will be same as with S[1..j-1] and T[1..i-1].

2. Suppose S[j] is not equal to T[i], then let
e1 = edit distance of substrings involving S[1..j-1] and T[1..i]
e2 = edit distance of substrings involving S[1..j] and T[1..i-1]
e3 = edit distance of substrings involving S[1..j-1] and T[1..i-1]

Then amongst e1,e2,e3:

- if e1 is the smallest, then edit distance of substrings S[1..j] and T[1..i] = e1 + 1. Here it is equivalent of deletion of char S[j] in the source string.
- if e2 is the smallest, then edit distance of substrings S[1..j] and T[1..i] = e2 + 1. Here it is equivalent of insertion of char T[i] in the source string to form the target string.
- if e3 is the smallest, then edit distance of substrings S[1..j] and T[1..i] = e3 + 1. Here it is equivalent of substituting S[j] with T[i] to form the target string.

Project Class Diagram


Command
Abstract Class
Properties
CommandType
Methods
Command
Execute
Undo
AddCommand
Class
→ Command
Properties
CommandType
Methods
AddCommand...
Execute
Undo
SearchCommand
Class
→ Command
Properties
CommandType
Methods
Execute
GetSearchString
SearchCommand
Undo
UnArchiveCommand
Class
→ Command
Properties
CommandType
Methods
Execute
GetUnArchivedTaskI...
UnArchiveComman...
Undo
DeleteCommand
Class
→ Command
Properties
Methods
DeleteCommand (+...
Execute
GetDeletedTaskId
Undo
EditCommand
Class
→ Command
Properties
Methods
EditCommand (...
Execute
Undo
UndoCommand
Class
→ Command
Properties
Methods
Execute
GetUndoCommand...
Undo
UndoCommand
ViewCommand
Class
→ Command
Properties
Methods
Execute
Undo
ViewCommand
Toast
Class
Fields
fadeInAnimation
fadeOutAnimation
storyboard
Methods
FadeInAnimation
FadeOutAnimation
ShowToast
Toast
InstantSearch
Class
Methods
GetTasksWithDescription
InstantSearch
DateHolder
Class
Methods
ConvertToDateTime
DateHolder
CommandParser
Class
Fields
Methods
CommandParser (+ 1 overload)
ParseCommand
Task
Class
Properties
Archive
Background
Description
DueToday
EndTime
Floating
Id
Overdue
StartTime
Methods
Task (+ 1 overl...
FileHandler
Class
Properties
Instance
UnitTestInstance
Methods
AddTaskToFile (+ 1 over...
ArchiveTaskInFile
DeleteTaskFromFile
FileHandler
GetTaskFromFile
UnarchiveTaskInFile
ViewAll
ViewArchive
ViewTasks (+ 1 overload)
WriteEditedTaskToFile
Events
FileUpdateEvent
FacadeLayer
Class
Methods
FacadeLayer
GetAllTasks
GetCommandObject
Undo
AutoCompletor
Class
Methods
AutoCompletor (+...
Query
CommandHistory
Class
Methods
AddToHistory
CommandHistory
DownClick
UpClick
AutoComplete
Class
→ UserControl
Properties
Count
SelectedItem
Methods
AutoComplete
Clear
Show
Events
AutoCompleteKeyboardEvent
AutoCompleteMouseEvent
SyntaxProvider
Class
Fields
keywords
Properties
Keywords
Methods
SyntaxProvider