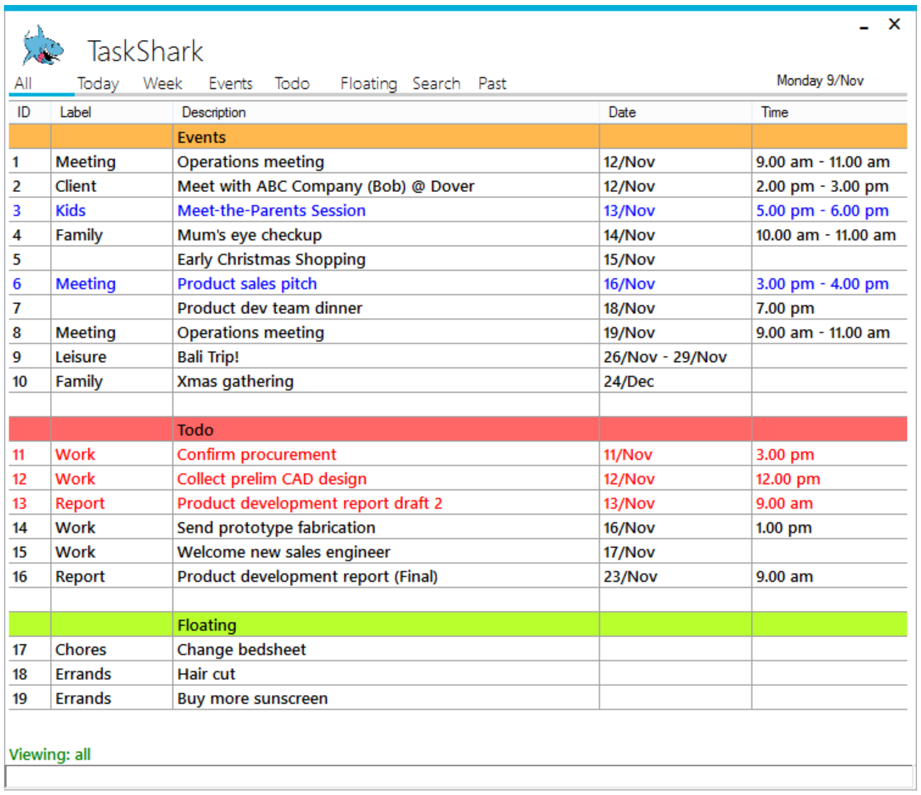
[T13-3C][V0.5]

TaskShark



ID	Label	Description	Date	Time
		Events		
1	Meeting	Operations meeting	12/Nov	9.00 am - 11.00 am
2	Client	Meet with ABC Company (Bob) @ Dover	12/Nov	2.00 pm - 3.00 pm
3	Kids	Meet-the-Parents Session	13/Nov	5.00 pm - 6.00 pm
4	Family	Mum's eye checkup	14/Nov	10.00 am - 11.00 am
5		Early Christmas Shopping	15/Nov	
6	Meeting	Product sales pitch	16/Nov	3.00 pm - 4.00 pm
7		Product dev team dinner	18/Nov	7.00 pm
8	Meeting	Operations meeting	19/Nov	9.00 am - 11.00 am
9	Leisure	Bali Trip!	26/Nov - 29/Nov	
10	Family	Xmas gathering	24/Dec	
		Todo		
11	Work	Confirm procurement	11/Nov	3.00 pm
12	Work	Collect prelim CAD design	12/Nov	12.00 pm
13	Report	Product development report draft 2	13/Nov	9.00 am
14	Work	Send prototype fabrication	16/Nov	1.00 pm
15	Work	Welcome new sales engineer	17/Nov	
16	Report	Product development report (Final)	23/Nov	9.00 am
		Floating		
17	Chores	Change bedsheet		
18	Errands	Hair cut		
19	Errands	Buy more sunscreen		

Viewing: all

Supervisor: Christopher Chak Hanrui Extra feature: GoodGUI

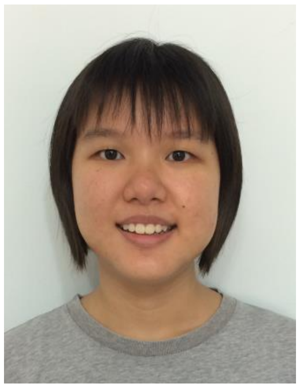
			
Chin Kiat Boon	Aaron Chong Jun Hao	Ng Ren Zhi	Soon Hao Ye
Team Lead Documentation Deliverables, Deadlines	Code Quality Scheduling and Tracking	Integration	Testing GUI

Table of Contents

Credits

The following external libraries are used in TaskShark:

1. **RapidJSON**	https://github.com/miloyip/rapidjson
2. **MetroFramework**	http://thielj.github.io/MetroFramework

The logo (shark.ico) belongs to **Tim van de Vall** at http://www.timvandevall.com.

Introduction & Motivation

On a daily basis, you may find yourself bombarded with ‘things to do’ or ‘events to attend’. Some things need to be done during specific times (e.g. attend meeting), some have deadlines (e.g. submit report), and others are simply ‘to be done someday’ (e.g. get haircut). Do you find these weighing heavily on your mind, or that you forget to do certain things on time?

If you spend most of your time near a computer and prefer typing over excessive use of a mouse, TaskShark was developed for you! TaskShark accepts natural language commands via keyboard, puts these tasks into your schedule and tracks them, so you no longer have so much on your mind.

After evaluating similar software in the market, TaskShark was created as a free software catered to students and office workers who frequently have new things to do or events that may be related.

General Information

Types of Tasks

1.	Task	with no deadline	“buy groceries”
2.	Todo	with a deadline	“buy groceries by tomorrow”
3.	Event	with a start and an end date/time	“Great Singapore Sale from 29 May to 26 July”

Flexible Date Formats

1. DD MM* “9 Nov”
   DD MM* YY* “9 Nov 15”
   Notes: Works with forward slash ‘/’ as delimiter
   MM* can be MM, MMM or MMMM
   YY* can be YY or YYYY (defaults to current year)

2. DD.MM*.YY* “9.11.15”
   Note: Does not default to current year, due to time signature

3. tmr “tmr”
   this DDD/DDDD “this Fri”
   next DDD/DDDD “next Friday”

Flexible Time Formats

1. HH AM/PM “8 AM”
   HH.MM AM/PM “8.00 AM”
   Note: Optional to specify AM or PM (defaults to AM)

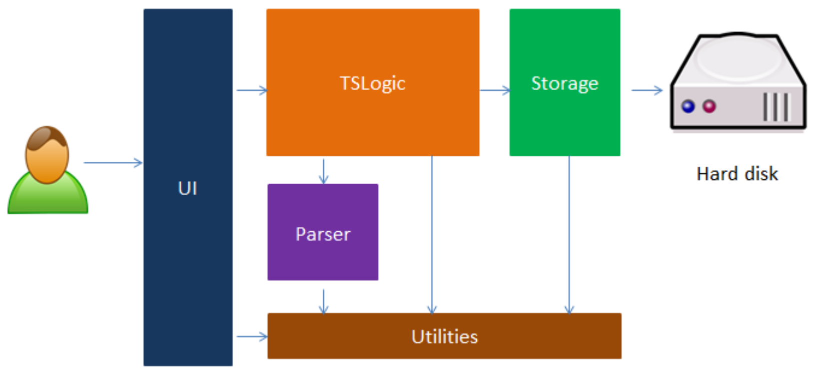
2. HHMM “0800”

Developer Guide

Introduction

Welcome to the developer guide for TaskShark! This guide orientates new team members for TaskShark by covering information such as architecture and design description. This provides a brief summary of the components within TaskShark, so you can start contributing immediately!

Architecture



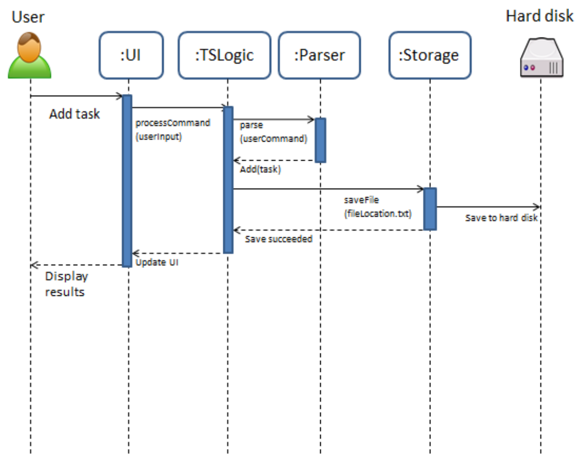
As shown above, TaskShark consists of 5 components, namely the **User Interface (UI)** component that the user would interact with, the **TSLogic** component that processes and updates TaskShark based on what the user wants, the **Parser** component that processes the user input into a list of instructions that is understood by TSLogic, the **Storage** component that enables TaskShark to save and load tasks from the hard disk previously input by the user, and finally the **Utilities** component that consists of common methods that are frequently used by the rest of the components.

Design Description

To provide a good idea of the interactions between the components, the following are the sequence diagrams, as well as specific use cases, to illustrate some of the general commands.

Adding a task

Sequence diagram



1. When the user inputs a command to add a new task onto the UI component, the UI will immediately pass the user input as a string over to the TSLogic component.
2. TSLogic then passes the string on to the Parser component, which will convert the user input into instructions that TSLogic understands. In this case, TSLogic would process the parsed input to add a task that the user intends to.
3. After adding the task, TSLogic calls the Storage component to save the input (together with any tasks that may have already been in TaskShark), as a list of tasks, into the hard disk.
4. Finally, TSLogic also passes the list of tasks to UI, which displays the list of tasks back to the user.

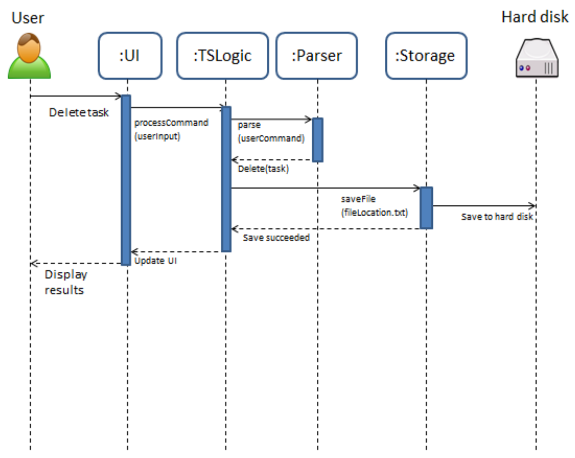
Use case

Software system: TaskShark
Use case: UC1 – Add a task
Actor: User

1. User types in "add **pull repo** at **11.59 pm**"
2. TaskShark displays "added **pull repo**"
3. TaskShark shows "**pull repo**" as its first element in the display list
   Use case ends.

Deleting a task

Sequence diagram



1. When the user inputs a command to delete a particular task onto the UI component, the UI will immediately pass the user input as a string over to the TSLogic component.
2. TSLogic then passes the string on to the Parser component, which will convert the user input into instructions that TSLogic understands. In this case, TSLogic would process the parsed input to delete a task that the user intends to.
3. After deleting the task, TSLogic then calls the Storage component to save the updated input into the hard disk.
4. Finally, TSLogic passes the list of tasks to UI, which displays the list of tasks back to the user.

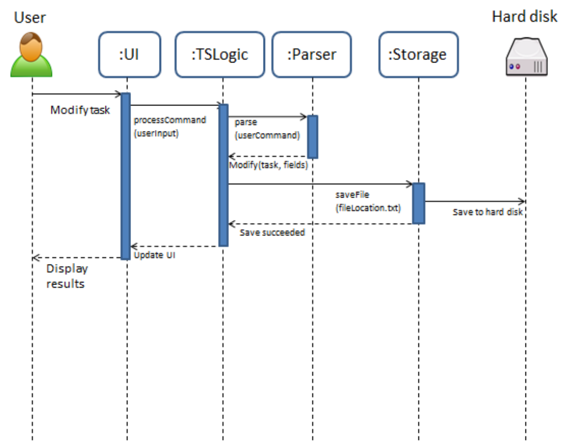
Use case

Software system: TaskShark
Use case: UC2 – Delete a task
Actor: User

1. User adds a task (UC1), “**pull repo** at **11.59 pm**”.
2. User adds another task (UC1), “**project meeting** at **2 pm**”.
3. User types in “Delete **1**”.
4. TaskShark shows “**project meeting**” as its first element in the display list.
   Use case ends.

Modifying a task

Sequence diagram



1. When the user inputs a command to modify a particular task onto the UI component, the UI will pass the user input as a string over to the TSLogic component.
2. TSLogic then passes the string on to the Parser component, which will convert the user input into instructions that TSLogic understands. In this case, Parser would return the specific task to modify, together with the fields to be modified.
3. After modifying the task, TSLogic then calls the Storage component to save the updated input into the hard disk.
4. Finally, TSLogic passes the list of tasks to UI, which displays the list of tasks back to the user.

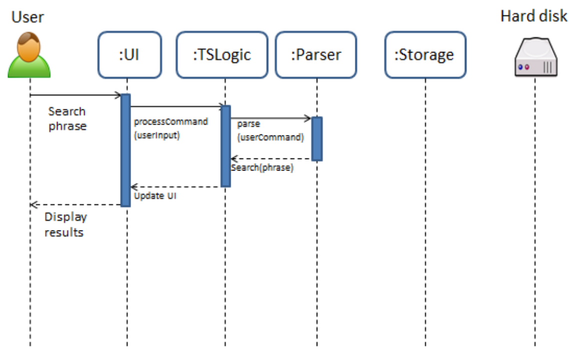
Use case

Software system: TaskShark
Use case: UC3 – Modify a task
Actor: User

1. User adds a task (UC1), “**pull repo** at **11.59 pm**”.
2. User adds another task (UC1), “**project meeting** at **2 pm**”.
3. User types in “modify **1 merge repo** at **11.59 pm**”.
4. TaskShark shows “**merge repo**” as its first element in the display list, and “**project meeting**” as its second element in the display list.
   Use case ends.

Searching tasks containing a phrase

Sequence diagram



1. When the user inputs a command to search tasks containing a certain phrase onto the UI component, the UI will pass the user input as a string over to the TSLogic component.
2. TSLogic then passes the string on to the Parser component, which will convert the user input into instructions that TSLogic understands. In this case, Parser would return the phrase to be searched, and TSLogic would match tasks containing the phrase.
3. TSLogic then passes the matching tasks to UI, which displays the list of tasks back to the user. Note that since there is already a copy of the tasks within TSLogic, there is no need to call the Storage component.

Use case

Software system: TaskShark
Use case: UC4 – Searching for a particular phrase
Actor: User

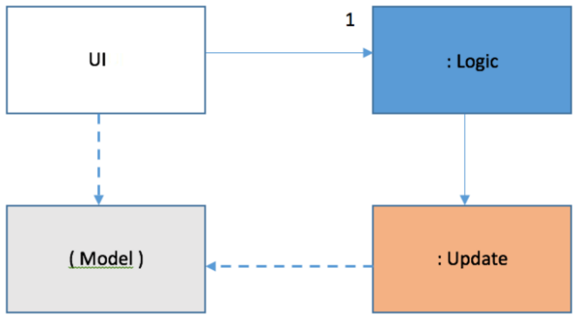
1. User adds a task (UC1), "**pull repo** at **11.59 pm**".
2. User adds another task (UC1), "**project meeting** at **2 pm**".
3. User types in "search **pull**"
4. TaskShark displays "**pull repo**" as its first element in the display list
5. User types in "search **p**"
6. TaskShark displays "**pull repo**" as its first element in the display list, and displays "**project meeting**" as its second element in the display list.
   Use case ends.

Component Structure

Below are the explanations of the major component classes and the important APIs in each, with the aim of helping new team members get a better understanding of the individual components.

UI

Class Diagram



The UI component uses the observer pattern to decouple itself from the Logic component. It calls namely:

1. `Logic::processCommand()` to perform TaskShark core functionalities,
2. `Logic::subscribe()` to pass in data structures/containers to be updated with displayed information, and
3. `Logic::getMode()` to get the state from logic, in order to determine what way to display tasks.

Important APIs

Below are the explanations of some of the important APIs in UI that aim to help new developers to have a better understanding of the component.

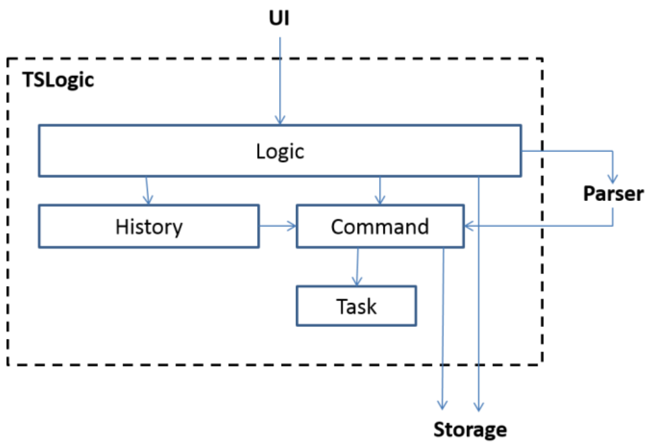
1. `void UI::processAndExecute()`
   Receives user input from the input textbox and passes to `std::string Logic::processCommand()`, a string is returned to be the user feedback, and it will be displayed in the feedback textbox in green. If an exception is caught, the exception message is displayed in red.

2. `System::Void UI::input_KeyDown()`
   This event handler is called whenever a key is pressed down. It is flow controller for all other functionalities such as selecting from autoSuggest , scrolling and receiving input.

3. `System::Void UI::input_KeyUp()`
   This event handler is called whenever a key is released. It is the flow controller for all other functionalities such as autosuggest.

Logic

<u>Class Diagram</u>



TSLogic is a loose adoption of the Command pattern, with Logic as the CommandCreator and History as the CommandQueue. Logic accepts user input from the UI component and creates commands by using TSParser. Logic then performs execute on the command. All action commands (e.g. CRUD, markdone) are added to History, while view/search commands are not added to the stack. If an Undo or Redo Command is executed, History will execute the latest Command in its Undo or redo stack, before updating the two stacks accordingly. After command has executed, Logic accesses the updated vector of tasks in Command, and passes it on to the Storage component to save the file.

Command has a total of 16 subclasses that each performs its respective functions. They are namely:

1. **Add:** Adds a new task to TaskShark
2. **Delete:** Deletes an existing task in TaskShark
3. **Modify:** Modify an existing task in TaskShark
4. **Pick:** Choose between two reserved time slots for a single event
5. **Search:** Search all tasks in TaskShark according to time, date, name, label
6. **PowerSearch:** Search and allows for free time-slot search
7. **MarkDone:** Mark a task as "done"
8. **UnmarkDone:** Mark a completed task as "not done"
9. **Undo:** Undo the previous action
10. **Redo:** Redo the most recent undone action
11. **View:** Change display according to: home, all, today, week, events, todo, floating, search, past
12. **ClearAll:** Remove all tasks currently in TaskShark
13. **DisplayAll:** Display all tasks
14. **Load:** Load tasks from another TaskShark text file
15. **Save:** Save tasks in TaskShark to a specified file path (automatically saves after every action)
16. **Exit:** Exits TaskShark

Any additional functionality in TaskShark should thus be coded as another subclass of Command.

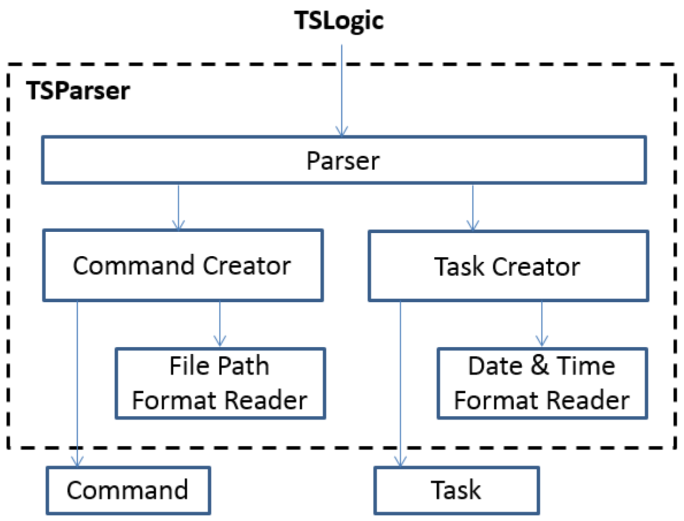
Important APIs

Logic provides these APIs to process and execute user input through Parser, as well as link Storage.

1. `std::string Logic::processCommand(std::string userCommand);`
   Allows TSLogic to call Parser::parse to understand the user input and employs the following API.
   a. `Command* parse(std::string userInput);`
      Called by Logic::processCommand, which parses the input command and creates a corresponding Command object.
   b. `virtual void Command::execute();`
      Called by Logic::processCommand, which allows the command class to perform its respective functionality.
   c. `bool saveFile(std::string filePath, std::vector<Task> taskVector);`
      Called by Logic::processCommand, which saves the current list of Tasks in TaskShark into its specified text file.

Parser

Class Diagram



As shown above, TSParser relies on a façade (Parser) called by Logic to parse every user input. Parser determines if the user input is a valid Command keyword, then decides if to create a Task. Command Creator may require File Path Format Reader to decipher into a suitable file path name. Task Creator may require Date & Time Format Reader to process date and time based on heuristics. Command object may contain a Task object. Parser then passes the Command object back to Logic.

Important APIs

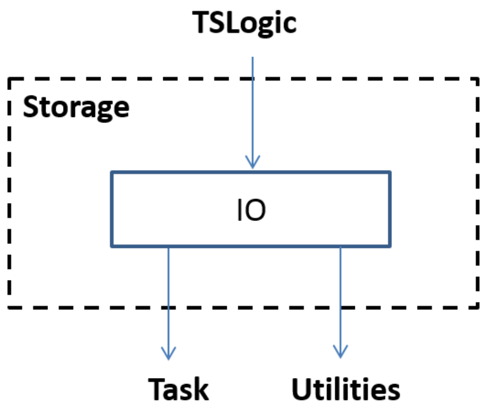
TSParser provides these APIs to convert flexible natural language into parameters for TaskShark.

1. `Command Parser::parse(std::string userInput);`
   Decides the intention of the user and returns a Command object with relevant parameters.

2. `std::string Parser::parseFileName(std::string fileName);`
   Ensures that the file name has the correct extension (.txt) by appending the extension if missing.

Storage

Class Diagram



The Storage component consists mainly of the IO class. TSLogic component calls Storage when it needs to interact with the external text file storage for TaskShark's tasks. Storage makes use of helper functions from the Utilities Class, and deals with Task objects. IO processes data to external text files in JSON format.

Important APIs

IO reads and writes data to and from a text file saved in the system in JSON format, so that TaskShark can retrieve Task data from previous sessions after you close TaskShark.

1. `std::vector<Task> IO::loadFile(std::string fileName);`
   Loads a .txt file previously written by TaskShark in JSON format to access saved data

2. `bool setFilePath(std::string newFilePath, std::vector<Task> taskVector, bool isRemovePrevFile);`
   Saves all current TaskShark Task data into a .txt file according to the specified file path in JSON format. Previous file path can be removed if specified.

Instructions for Setting Up and Testing

Setting Up

1. On GitHub, navigate to the main page of the repository.
2. In the right sidebar of the repository page, click to copy the clone URL for the repository.
3. Open Terminal (for Mac and Linux users) or the command prompt (for Windows users).
4. Change the current working directory to the location where you want the cloned directory.
5. Type ***git clone***, and then paste the URL you copied in Step 2.
6. Press Enter. Your local clone will be created.
7. Open the File Explorer and navigate to the cloned directory.
8. Double-click TaskShark.sln.
9. Build using F7 hotkey.

Testing (Unit tests in TaskSharkTests project)

1. Create a ***Component*****Test.cpp** file for the desired ***Component***.
2. Create a **TEST_CLASS(*****Method*****Test) {}** for the desired ***Method***.
3. Create a **TEST_METHOD(*****Component_method_type*****) {}** for each ***type*** of test.
4. Include sufficient test cases in each TEST_METHOD.
5. Build using F7 Hotkey.
6. Run All.
7. Inform other developers if any of their tests fail.
8. Rectify your own test or code.