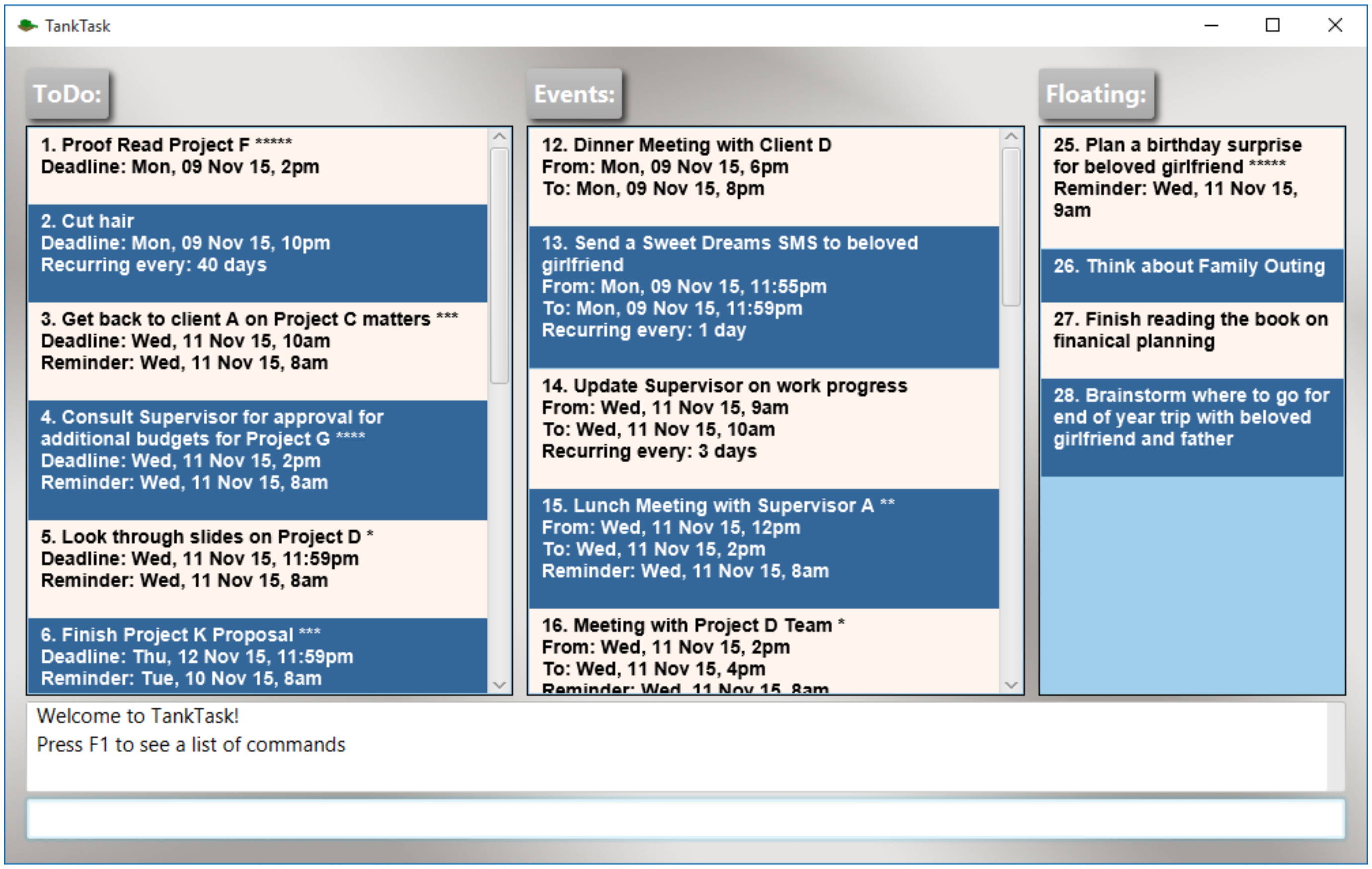
TankTask



TankTask

ToDo:

1. Proof Read Project F *****
Deadline: Mon, 09 Nov 15, 2pm

2. Cut hair
Deadline: Mon, 09 Nov 15, 10pm
Recurring every: 40 days

3. Get back to client A on Project C matters ***
Deadline: Wed, 11 Nov 15, 10am
Reminder: Wed, 11 Nov 15, 8am

4. Consult Supervisor for approval for additional budgets for Project G ****
Deadline: Wed, 11 Nov 15, 2pm
Reminder: Wed, 11 Nov 15, 8am

5. Look through slides on Project D *
Deadline: Wed, 11 Nov 15, 11:59pm
Reminder: Wed, 11 Nov 15, 8am

6. Finish Project K Proposal ***
Deadline: Thu, 12 Nov 15, 11:59pm
Reminder: Tue, 10 Nov 15, 8am

Events:

12. Dinner Meeting with Client D
From: Mon, 09 Nov 15, 6pm
To: Mon, 09 Nov 15, 8pm

13. Send a Sweet Dreams SMS to beloved girlfriend
From: Mon, 09 Nov 15, 11:55pm
To: Mon, 09 Nov 15, 11:59pm
Recurring every: 1 day

14. Update Supervisor on work progress
From: Wed, 11 Nov 15, 9am
To: Wed, 11 Nov 15, 10am
Recurring every: 3 days

15. Lunch Meeting with Supervisor A **
From: Wed, 11 Nov 15, 12pm
To: Wed, 11 Nov 15, 2pm
Reminder: Wed, 11 Nov 15, 8am

16. Meeting with Project D Team *
From: Wed, 11 Nov 15, 2pm
To: Wed, 11 Nov 15, 4pm
Reminder: Wed, 11 Nov 15, 8am

Floating:

25. Plan a birthday surprise for beloved girlfriend *****
Reminder: Wed, 11 Nov 15, 9am

26. Think about Family Outing

27. Finish reading the book on finanical planning

28. Brainstorm where to go for end of year trip with beloved girlfriend and father

Welcome to TankTask!
Press F1 to see a list of commands

Developed by: W15-1J (CS2103) / Group 6 Team 1 (CS2101)

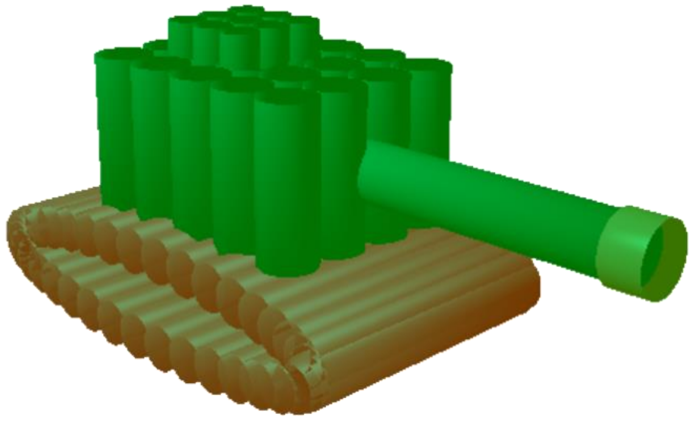
Supervisor: Chak Hanrui, Christopher

Extra features: Recurring Task

Khairul Rizqi B Mohd Shariff [Team lead, Main Tester, Git Expert]	**Koh Ling Ling Jean** [Graphics Designer]	**Luah Bao Jun** [Code Quality]	**Yu Ruofan** [Documentation]

[w15-1j][V0.5]

TankTask



User Guide

Content

1 About TankTask

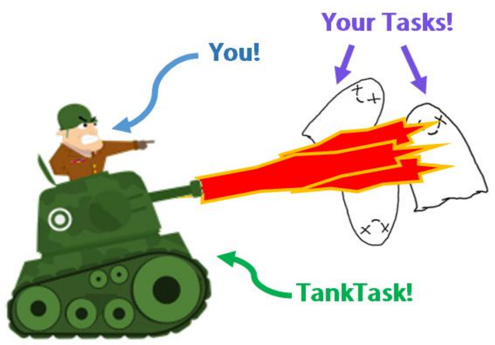
TankTask is a **task manager software** that helps you to keep track of your personal list of tasks. In **TankTask**, you can add and edit your tasks, and organise them neatly into lists. **TankTask** is quick, easy, and readily accessible from your desktop.

If you are constantly being bombarded with new tasks, or have problem remembering and sorting out your tasks. **TankTask** is for you. With **TankTask**, you will never forget a task or miss a deadline again!

TankTask is designed such that you can use it with only your keyboard. As the user, you can perform any action by simply typing your command into the program, or press a hotkey. **TankTask**'s commands are simple to learn as they resemble the natural English language that we speak.

Also, **TankTask** provides you with many useful functions. For example, you can switch between different views, set reminders for your tasks, make your task recur, search for a task, or sort your task list automatically.

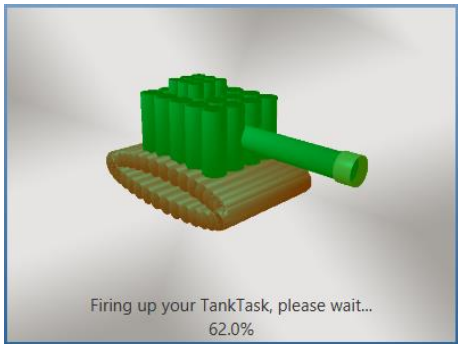
As such, **TankTask** is simple, intuitive, flexible and convenient to use. Our powerful yet friendly **TankTask** will assist you in defeating any formidable tasks ahead!



2 Getting Started

Let's get you started! To begin using **TankTask**, follow these simple steps:

1) Open your web browser, go to our git website: https://github.com/cs2103aug2015-w15-1j/main/releases
2) Click on the **latest version** (as of the time this guide is written, the latest stable version of **TankTask** is **V0.5**).
3) Click on **TankTask.zip** to download the .zip file
4) Extract the **TankTask folder** from the .zip file. In Windows, you can right click on the .zip file and select 'Extract All'. If your computer does not have an extract option, you can download 7-Zip or WinRAR to extract the folder.
5) Open the folder, double-click on **TankTask.jar** to run it
6) You should see a **loading screen** as shown on the right. Wait for **TankTask** to load.
7) Once loading is done, the loading screen will disappear and you should see the program's **default view**. Congratulations, you have successfully set up **TankTask** on your computer!


Firing up your TankTask, please wait...
62.0%

[Figure 1] Loading screen of TankTask

As you read the guide, you may want to keep **TankTask** open so that we can walk you through the program step-by-step.

3 Commands and Hotkeys

TankTask is able to perform actions by recognising certain commands that you enter or certain hotkeys that you press. Below is a list of commands and hotkeys available in **TankTask**. If you need help with the commands or hotkeys when you are using **TankTask**, you can also press **F1** to open the help list.

3.1 List of Hotkeys

F1	Press **F1** to open **Help**
F2	Press **F2** to toggle between **Floating** and **Overdue** list (if any)
F3	Press **F3** to get **Today's** Tasks
F4	Press **F4** see **Completed** Tasks
esc	Press **Esc** to **exit** the program
tab	Press **Tab** to toggle between **default** and **focus** view
↑ ↓	Use **Up** arrow or **Down** arrow to navigate list in focus view
← →	Use **Left** arrow key or **Right** arrow key to change list in focus view
ctrl + Z	Use **Ctrl + Z** to **undo**
ctrl + Y	Use **Ctrl + Y** to **redo**
shift + ↑	Use **Shift + Up** to navigate **previous commands**
shift + ↓	Use **Shift + Down** to navigate **previous commands**

3.2 List of Commands

*Note: You should replace the content in the brackets [ ] with your own content

COMMAND	FORMAT	EXAMPLES
`add`	⇨ **add** [Task Name] ⇨ **add** [Task Name] **by** [Deadline] ⇨ **add** [Task Name] **from** [Start] **to** [End]	✓ add Feed the fish ✓ add Feed the fish by today ✓ add Feed the fish from today 6pm to 6:15pm
`done`	⇨ [Task Index] **done**	✓ 1 done
`undone`	⇨ [Task Index] **undone**	✓ 1 undone
`delete / del`	⇨ [Task Index] **del**	✓ 1 del
`undo`	⇨ **undo**	✓ undo
`redo`	⇨ **redo**	✓ redo
`search`	⇨ **search** [query]	✓ search fish
`show`	⇨ **show** [Todo/Event/Floating/Overdue /Today/Complete] ⇨ **show**[X]	✓ show event ✓ showE
`sort / sortD / sortN / sortP`	⇨ **sort** [Field (Date/Name/Priority)] ⇨ **sort**[X]	✓ sort priority ✓ sortP
`every`	⇨ **every** [Interval] [Frequency (day/week/month/year)]	✓ every day ✓ every 2 weeks
`reset`	⇨ [Task Index] **reset** [Field (all/by/event/des/pri/rem/recur)]	✓ 1 reset all ✓ 1 reset des
`exit`	⇨ **exit**	✓ Exit
`rename`	⇨ [Task Index] **rename** [New Task Name]	✓ 1 rename Kill the fish
`deadline / by`	⇨ [Task Index] **by** [Deadline]	✓ 1 by 20 Oct 3pm
`event / from`	⇨ [Task Index] **from** [Start] **to** [End]	✓ 1 from 20 Oct 3pm ✓ 1 from 20 Oct 3pm to 5pm ✓ 1 from 20 Oct 3pm to 21 Oct 8am
`description / des`	⇨ [Task Index] **des** [Description]	✓ 1 des need to do this
`priority / pri`	⇨ [Task Index] **pri** [Priority (1-5)]	✓ 1 pri 5
`reminder /rem`	⇨ [Task Index] **rem** [Reminder Date]	✓ 1 rem 20 Oct 3pm

3.3 Date/Time Formats

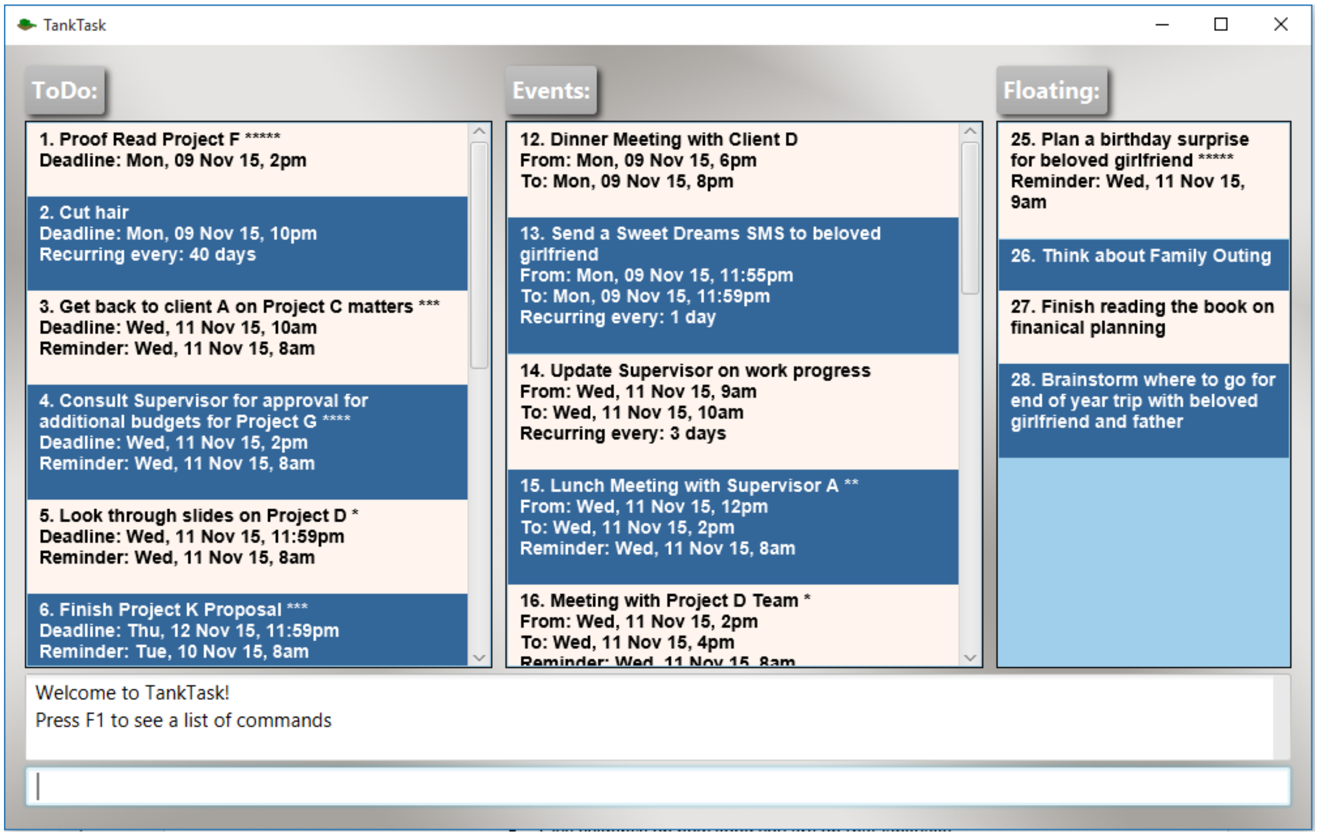
TankTask has a natural language parser that can recognise date/time in different format:

Date Formats	Time Format
dd/mm (e.g. 25/12)	**hh:mm [am/pm]** (e.g. 9am, 11:30pm)
dd-mm (e.g. 25-12)	**HH:mm** (e.g. 9:00, 23:30)
dd MMM (e.g. 25 Dec) or **MMM dd** (e.g. Dec 25)	
dd [month] (e.g. 25 December) or **[month] dd**	
Today/tomorrow/X days later (e.g. 2 days later)	
Day-of-week (e.g. Tuesday, next Sunday)	

4 UI Overview

Before you start adding tasks into the program, you should first familiarise yourself with the user interface (**UI**) by understanding the features that are being displayed on the screen. You should also learn about the different **views** in TankTask and how to switch between them.

4.1 Default View



1. Proof Read Project F *****
Deadline: Mon, 09 Nov 15, 2pm

2. Cut hair
Deadline: Mon, 09 Nov 15, 10pm
Recurring every: 40 days

3. Get back to client A on Project C matters ***
Deadline: Wed, 11 Nov 15, 10am
Reminder: Wed, 11 Nov 15, 8am

4. Consult Supervisor for approval for additional budgets for Project G ****
Deadline: Wed, 11 Nov 15, 2pm
Reminder: Wed, 11 Nov 15, 8am

5. Look through slides on Project D *
Deadline: Wed, 11 Nov 15, 11:59pm
Reminder: Wed, 11 Nov 15, 8am

6. Finish Project K Proposal ***
Deadline: Thu, 12 Nov 15, 11:59pm
Reminder: Tue, 10 Nov 15, 8am

12. Dinner Meeting with Client D
From: Mon, 09 Nov 15, 6pm
To: Mon, 09 Nov 15, 8pm

13. Send a Sweet Dreams SMS to beloved girlfriend
From: Mon, 09 Nov 15, 11:55pm
To: Mon, 09 Nov 15, 11:59pm
Recurring every: 1 day

14. Update Supervisor on work progress
From: Wed, 11 Nov 15, 9am
To: Wed, 11 Nov 15, 10am
Recurring every: 3 days

15. Lunch Meeting with Supervisor A **
From: Wed, 11 Nov 15, 12pm
To: Wed, 11 Nov 15, 2pm
Reminder: Wed, 11 Nov 15, 8am

16. Meeting with Project D Team *
From: Wed, 11 Nov 15, 2pm
To: Wed, 11 Nov 15, 4pm
Reminder: Wed, 11 Nov 15, 8am

25. Plan a birthday surprise for beloved girlfriend *****
Reminder: Wed, 11 Nov 15, 9am

26. Think about Family Outing

27. Finish reading the book on finanical planning

28. Brainstorm where to go for end of year trip with beloved girlfriend and father

Welcome to TankTask!
Press F1 to see a list of commands

[Figure 2] Default View of TankTask

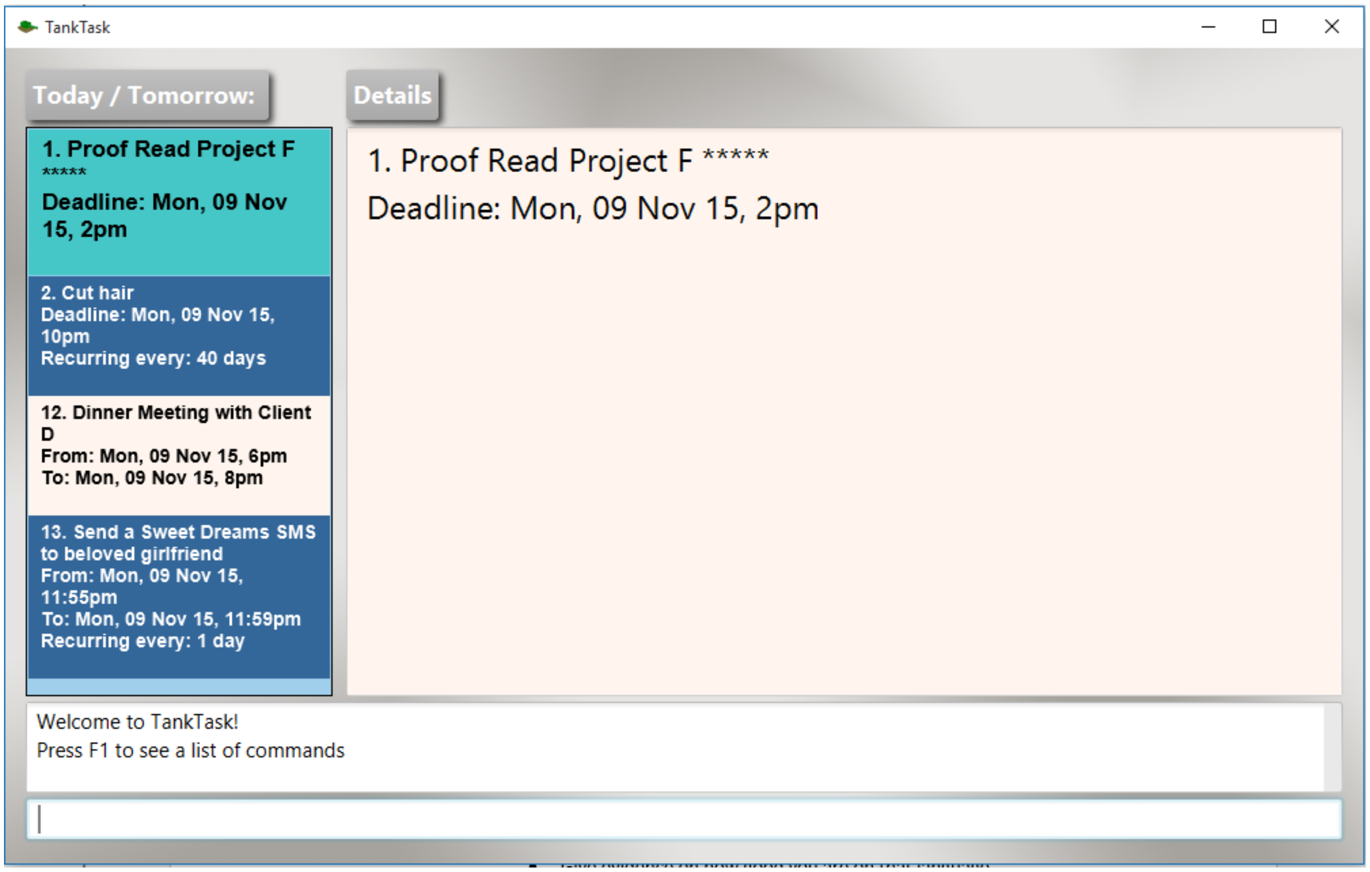
The **default view** is the first thing that you will see after TankTask is loaded. This is where you can see an overview of your tasks. It is separated into 3 panels:

- The **command line** at the bottom of the screen is where you type and enter your command
- The **feedback console** will display feedback messages to you after you enter a command
- The **task lists** display the tasks that you have added. There are 3 task lists, each containing a different type of task (ToDo, Event or Floating). A **ToDo** has a deadline, an **Event** has a start and end date/time, and a **Floating** has no specified date/time.

As you can see from Figure 2, each task is displayed as a few lines of information. The first line shows the **task name** and the number on the left of the task name is the **task index**. The stars (*) on the right side of the task name represent the task's **priority level**. The remaining lines show the **deadline** or **date/time** of the task. If a task has a **reminder** set or is a **recurring** task, the information will also be shown.

4.2 Focus View

Want to examine a task in more detail? Simply press **Tab** to switch to **focus view**. In this view, you will be able to view all information of a particular task, displayed with a much larger font size.



1. Proof Read Project F *****
Deadline: Mon, 09 Nov 15, 2pm

2. Cut hair
Deadline: Mon, 09 Nov 15, 10pm
Recurring every: 40 days

12. Dinner Meeting with Client D
From: Mon, 09 Nov 15, 6pm
To: Mon, 09 Nov 15, 8pm

13. Send a Sweet Dreams SMS to beloved girlfriend
From: Mon, 09 Nov 15, 11:55pm
To: Mon, 09 Nov 15, 11:59pm
Recurring every: 1 day

1. Proof Read Project F *****
Deadline: Mon, 09 Nov 15, 2pm

```
Welcome to TankTask!
Press F1 to see a list of commands
```

[Figure 3] Focus View of TankTask

In focus view, only one list is displayed at a time in the left panel. The right panel displays all the information of the currently selected task. The selected task is highlighted in cyan, as shown in Figure 3.

To scroll through the task list to view different tasks, use the **up** and **down** arrow keys. To switch to a different list, use the **left** and **right** arrow keys. You can also use the show commanTo return to default view, just hit **Tab** again.

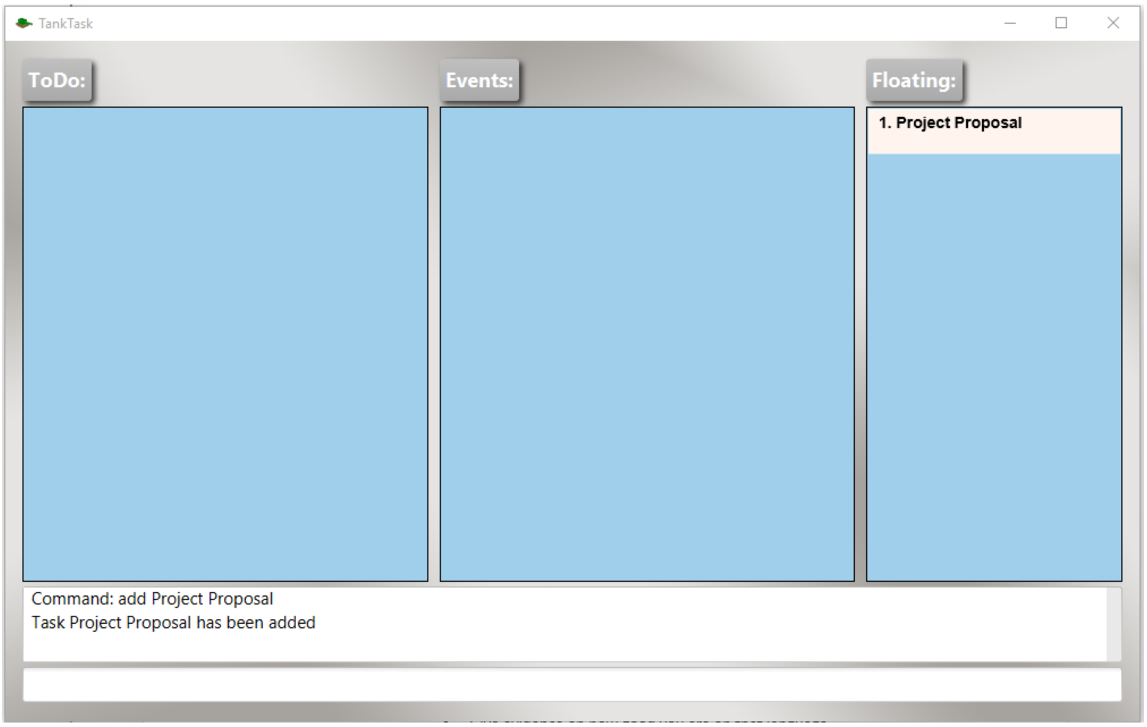
5 Adding a Task

5.1 Add a Floating

First, to add a new task, use the command word '**add**'.

e.g.
```
add Project Proposal
```

This will add a new task called 'Project Proposal'. By default, the new task will be a Floating, since it has no date/time. You should now see 'Project Proposal' in your Floating list, as shown in Figure 4 below:



Command: add Project Proposal
Task Project Proposal has been added

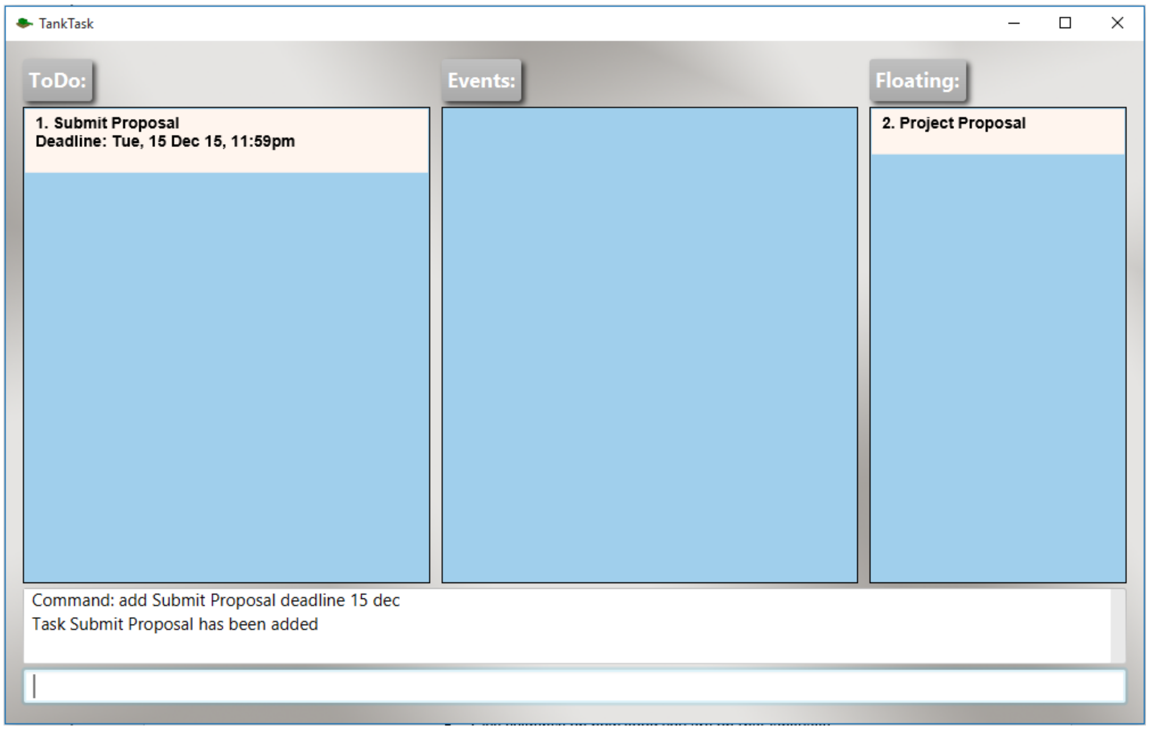
[Figure 4] Task ‘Project Proposal’ has been added

5.2 Add a ToDo

To add a **ToDo**, besides using **‘add’**, you need to also specify the deadline by typing the command word **‘deadline’** or **‘by’** after the task name.

e.g. `add Submit Proposal by 15 Dec`

You should now see a new **ToDo** called “Submit Proposal” in your ToDo list, as shown in Figure 5 below.



Command: add Submit Proposal deadline 15 dec
Task Submit Proposal has been added

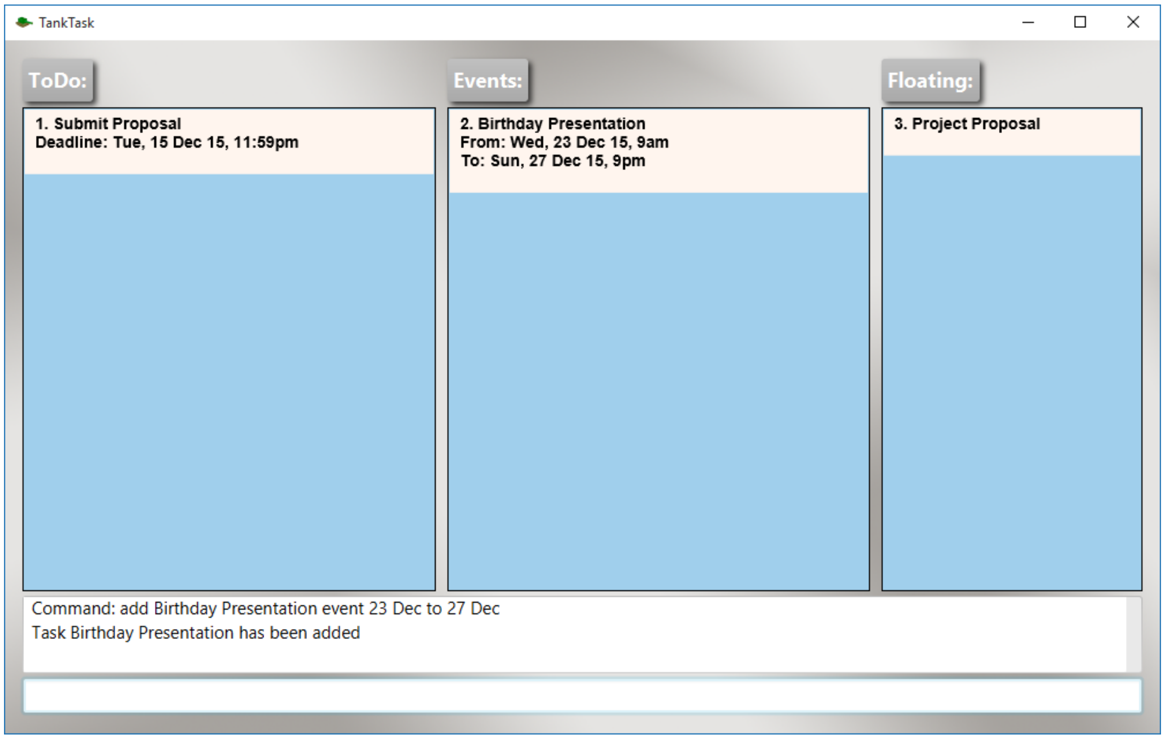
[Figure 5] Task ‘Submit Proposal’ has been added

5.3 Add an Event

Similarly, to add an event, type the command word **'event'** or **'from'** after the task name to indicate a start date/time for the task. If you want to indicate the end date/time too, use the command word **'to'**.

```
e.g.      add Birthday Preparation from 23 Dec to 27 Dec
```

The event "Birthday Preparation" should now appear, as shown in Fig 5 below:



1. Submit Proposal
Deadline: Tue, 15 Dec 15, 11:59pm

2. Birthday Presentation
From: Wed, 23 Dec 15, 9am
To: Sun, 27 Dec 15, 9pm

3. Project Proposal

```
Command: add Birthday Presentation event 23 Dec to 27 Dec
Task Birthday Presentation has been added
```

[Figure 6] Task 'Birthday Preparation' has been added

6 Editing a Task

To edit the information of a task, you will first type the **index** of the task that you want to edit, followed by the command word that corresponds to the **field** that you want to edit. The next few sections show how you can do this.

6.1 Set Deadline

If the task is a **Floating**, you can convert the task to a **ToDo** by setting a deadline for the task.
If the task is a **ToDo**, you can change the current deadline to a new one.

To set a deadline, type the task **index** followed by the command word **'deadline'** or **'by'**. For example, referring to Figure 6, if you want to set a deadline for 'Submit Proposal', the task index is **1**.

```
e.g.      1 by 30 Dec 12pm
```

6.2 Set Start and End Date/Time

Similar to setting a deadline, you can convert a **Floating** to an **Event** by setting the date/time, or change the current date/time of an **Event**.

To set the start and end date/time, type the task **index** followed by the command word **‘event’** or **‘from’**. Use the command word **‘to’** if you also want to indicate the end date/time. If the end date/time is not specified, TankTask will set the end date/time for you, depending on what you have entered.

e.g.	`2 event 20 Sep 10am to 27 Sep 8pm`	
e.g.	`2 event 20 Sep 10am to 8pm`	[End date will be same as start date, which is 20 Sep]
e.g.	`2 event 20 Sep 10am`	[End time will default to 9pm]
e.g.	`2 event 20 Sep`	[Start time will default to 9am]

6.3 Set Priority

Sometimes, you might want to give each task a **priority** level to differentiate between important and trivial tasks. To set the priority of a task, type the command word **‘priority’**, followed by the priority level, which has to be a number from **1 to 5**.

e.g. `3 priority 5`

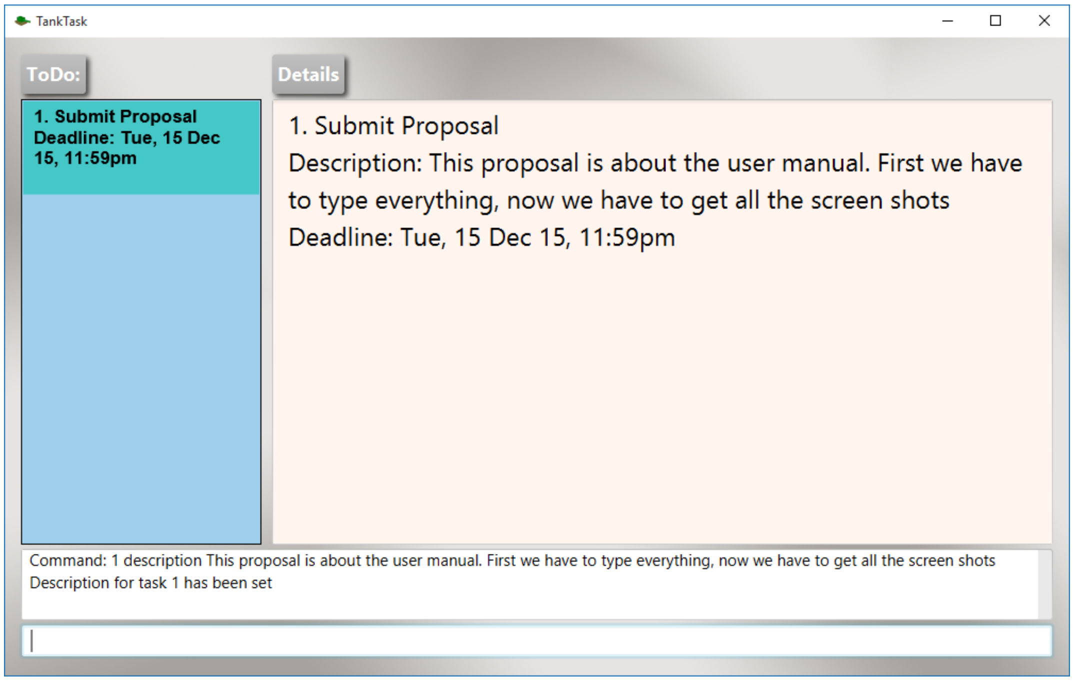
After you entered the command, the priority level will appear as the number of stars beside the task name. Note that 5 being the highest level, 1 being the lowest.

6.4 Set Description

Any extra details that you may want to add to your task should be put under the task’s **description**. To set a description, use the command keyword **‘description’.** You may enter as many words as you want in the task description.

e.g. `4 description This proposal is about the user manual. First we have to...`

After you entered the command, you can switch to focus view to see the description of the task.



1. Submit Proposal
Deadline: Tue, 15 Dec 15, 11:59pm

1. Submit Proposal
Description: This proposal is about the user manual. First we have to type everything, now we have to get all the screen shots
Deadline: Tue, 15 Dec 15, 11:59pm

```
Command: 1 description This proposal is about the user manual. First we have to type everything, now we have to get all the screen shots
Description for task 1 has been set
```

[Figure 7] Description for Task 1 has been set

6.5 Rename a Task

You can simply use the command word **'rename'** to rename a task.

e.g.
```
5 rename New proposal
```

Task 5 will be renamed to 'New Proposal'.

6.6 One-Shot Command

As you become more familiar with **TankTask**'s commands, you can start using **one-shot command** to set multiple fields of a task at the same time. A one-shot command can include more than 1 command word, such as **'deadline'**, **'priority'**, and **'description'**. Note that you can type the command words in any order.

e.g.
```
6 deadline 1 Sep 11:30pm priority 3 description I need to do this
```

7 Managing the Tasks

7.1 View a Specific List

In the focus view, as mentioned earlier, you can use the **left** and **right** arrow keys to switch between different lists,

One way of switching to a specific list directly is to use the **'show'** command word. Simply type show followed by the name of the list (ToDo, Event, Floating, Today, Overdue or Complete). Alternatively you can type '**show**X**'**, where X is replaced by the letter that represents the list (e.g. T for ToDo, E for Event, etc).

You can also press **F3** to view the Today/Tomorrow list or **F4** to view the Completed list.

e.g.
```
show todo
```

e.g.
```
showT
```

7.2 Mark Task as Done

Once you have completed a task, you can mark a task as **done**, use the command word **'done'**.

e.g.
```
1 done
```

Once the task is marked as done, you should see it disappear from the task list.

To view your completed tasks, enter **'showC'** or '**show completed'**. If you want to unmark a task, you can use the command word **'undone'**, which will put the task back into your task list.

7.3 Delete a Task

If a task gets cancelled or abandoned, you can **delete** a task, using the command word **'delete'**. This will remove the task from your task list.

e.g.
```
2 delete
```

7.4 Undo/Redo a Command

To undo an action, enter the command **'undo'**, or simply press **Ctrl+Z**. For example, if you have deleted a task just now, you can simply undo. The task should reappear in the task list.

Conversely, you can enter **'redo'** or press **Ctrl+Y** to reverse the effect of undo.

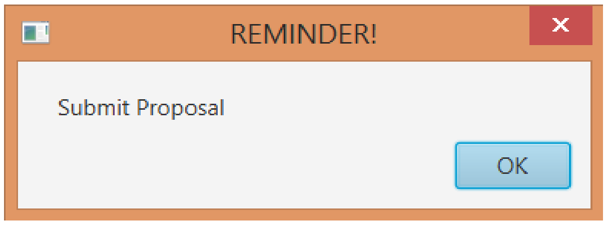
7.5 Set Reminder

To make sure that you will remember to do a task, you can ask TankTask to remind you about the task at a later date/time.

To set the reminder, use the command word **'reminder'**. Take note that the reminder has to happen before the task's deadline or start date/time.

```
e.g.      4 reminder 20 Dec 12pm
```

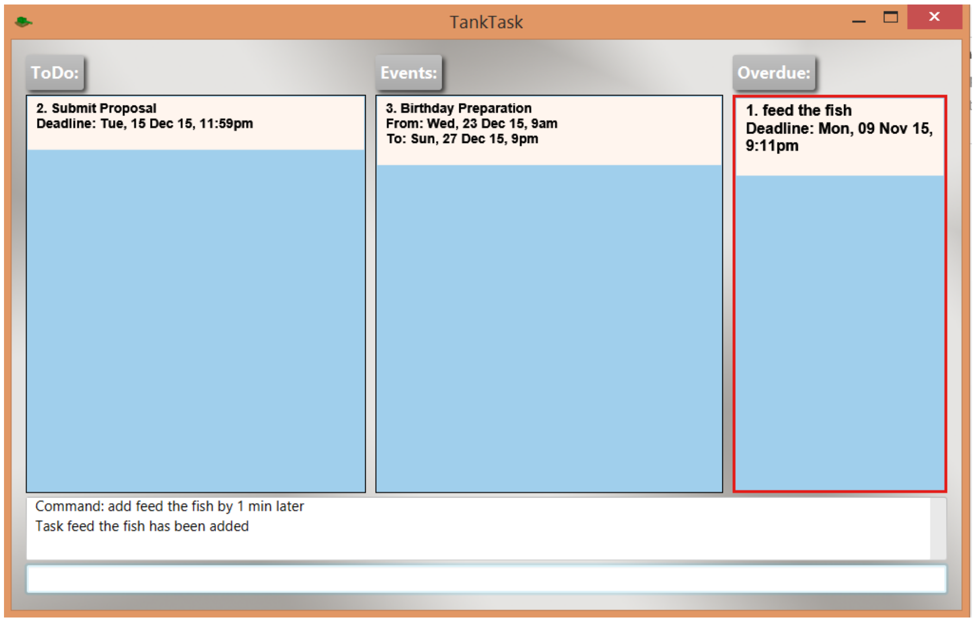
When the reminder date/time arrives, a reminder pop-up will appear on your screen, like the one shown below. Note that TankTask has to be running for the pop-up to appear. If you open TankTask after the reminder date/time, the pop-up will appear right after TankTask has loaded.



[Figure 8] Reminder for 'Submit Proposal'

7.6 Overdue Tasks

Once the deadline or start time of a task has passed, the task will be put into the **Overdue** list. When you have an overdue task, **TankTask** will remind you about the overdue task by displaying the Overdue list. The Overdue list will replace the Floating list in the default view, as shown in Figure 9 below. You can press **F2** to toggle between Floating and Overdue list.



2. Submit Proposal
Deadline: Tue, 15 Dec 15, 11:59pm

3. Birthday Preparation
From: Wed, 23 Dec 15, 9am
To: Sun, 27 Dec 15, 9pm

1. feed the fish
Deadline: Mon, 09 Nov 15, 9:11pm

```
Command: add feed the fish by 1 min later
Task feed the fish has been added
```

[Figure 9] The task 'feed the fish' is overdue

7.7 Search for a Task

If you want to find a task, you can use the **'search'** command word to locate the task quickly.

```
e.g.        search project proposal
```

TankTask will search through the task lists for tasks that contain both the word 'project' and 'proposal' in its information. Once search is completed, **TankTask** will switch to focus view and display a list of matches.

You can also search for dates to get a list of tasks that match the date. For example, if you search '11 Nov', all the tasks that have a deadline or start date on that day will be displayed in the search result.

7.8 Sort Tasks

By default, the tasks are listed in order of date. You can also choose to sort the tasks by priority or name instead.

- To sort by priority, type **'sortP'** or **'sort priority'**. This will list the tasks in order from highest to lowest priority.
- To sort by name, type **'sortN'** or **'sort name'**. This will list the tasks in alphabetical order.
- To sort by date, type **'sortD'** or **'sort date'**. This will put the tasks with the nearest approaching dates at the top of the lists

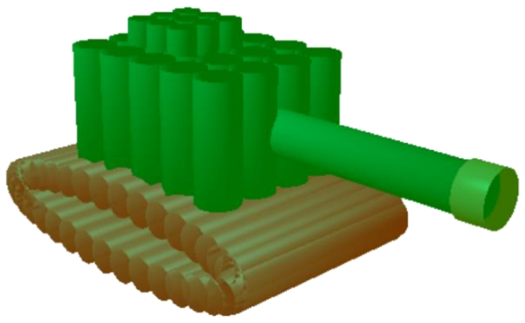
7.9 Recurring Tasks

Recurring tasks are tasks that are set to repeat on a fixed basis. You can set a task to recur every X number of days, weeks, months or even years. To make a task recur, type the command word **'every'**, followed by the interval and frequency of recurrence. This will allow **TankTask** to add the same task into the task list every time the task recur.

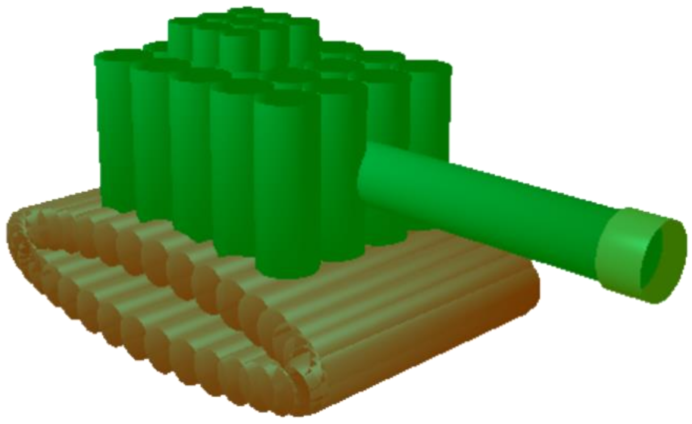
```
e.g.        8 every day

e.g.        8 every 2 weeks
```

However, do note that only tasks with dates (**ToDos** or **Events**) can recur. Before you can make the task recur, you have to first set either a deadline or a start date/time for the task.

You have come to the end of the user guide. We hope that you will make good use of **TankTask** to manage your tasks efficiently. So, go ahead and start tanking your day!



TankTask



Developer Guide

Content

1 Introduction

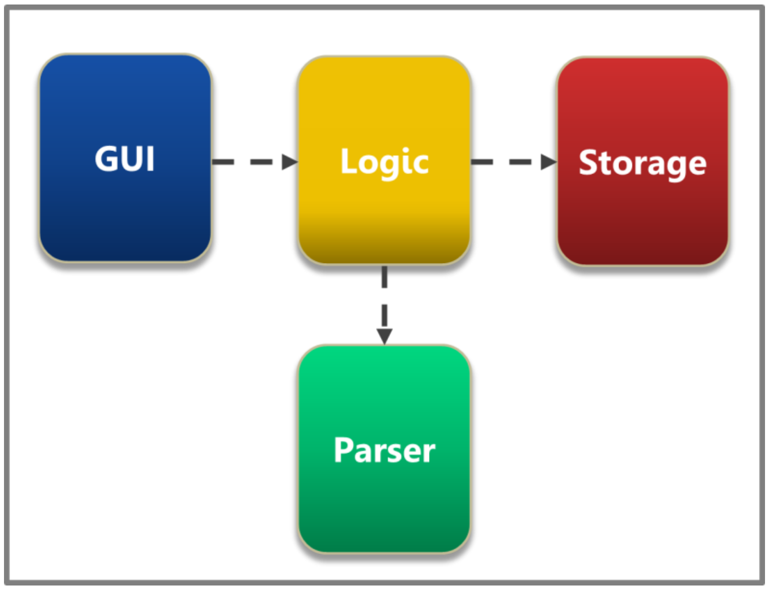
TankTask is a task manager software that allows the user to organise and keep track of tasks. To know more information about the program's features, you may refer to the user guide.

TankTask is designed to be used with mainly the keyboard. To perform an action, such as adding a new task, the user types and enters a command into the program. The user can also use hotkeys to perform certain actions instantly. For a list of all the available commands and hotkeys in **TankTask**, you may refer to the appendix.

The purpose of this developer guide is to walk you through the implementation of **TankTask**. The guide will focus on explaining how **TankTask** behaves as a whole when handling a user command, and how its components interact and work together to execute the command. By the end of this developer guide, you should be clear of the structure and internal working of **TankTask**, and be able to continue the development of this program.

2 Architecture

This developer guide will use a top-down approach to present the structure of **TankTask**, starting from the high-level organisation, and then going into each program component. Thus, we first introduce to you the architecture of **TankTask**:



[Figure 1] Architecture of TankTask

As seen in [Figure 1], the architecture consists of 4 main components: **GUI**, **Logic**, **Parser** and **Storage**. The arrows indicate the dependency of the components on one another. For example, Logic is dependent on Parser, but the reverse is not true.

When the user runs **TankTask**, the first component that will be initialized is the Graphic User Interface (**GUI**). **GUI** controls how the program is being displayed, in terms of both visual appearance and the information that is displayed to the user. It also controls input and output, providing an interface for interaction between the user and the program.

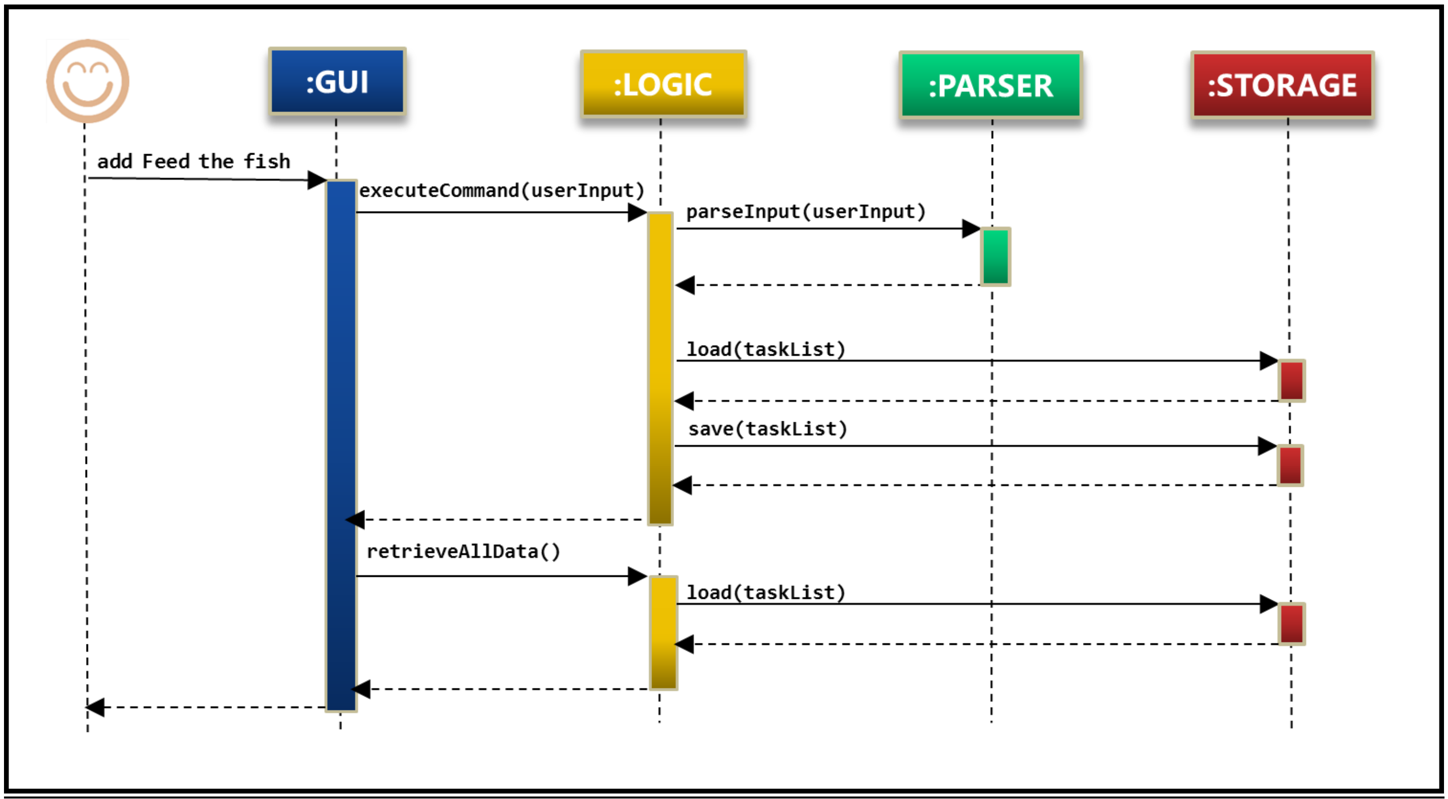
The next component to be initialized will be **Logic**. Located in the centre of the architecture, **Logic** essentially serves as a 'middleman' between the components, by determining what operation should be carried out next, and directing the flow of information from one component to another.

Parser is a specialized component that interprets the user input and converts it into a format that can be understood and handled by the other components. Parsing the input involves splitting the input into tokens, identifying what type of command or information each token represents, and assembling the tokens into a logical order.

The last component in the architecture is **Storage**, which is in charge of managing data and storing the data in the computer memory. To store or modify the data, **Storage** will write the information to a text file and save the file. When data needs to be accessed, **Storage** will read from the file and returns the information.

3 Behaviour

To understand how **TankTask** work during execution, it is easier to visualize the process and flow of information using a sequence diagram. Figure 2 illustrates what happens when the program executes the user command "add Feed the fish":



[Figure 2] Sequence diagram for the command "add Feed the Fish"

Basically, what happens is that the command is executed by the backend components, and **GUI** will then get and display the updated task lists. The steps involved are described in detail below:

1) **GUI** uses the method `runEvents` to take in the user input.
2) **GUI** sends user input to **Logic** by calling the method `executeCommand`.
3) **Logic** sends input to **Parser** by calling the method `parseInput`.
4) **Parser** parses the user input and returns the content to **Logic**.
5) Based on the parsed content, **Logic** decides what command to execute (what command object to create).
6) **Logic** calls **Storage** to load the existing task list
7) **Logic** updates the task list with the new task, and calls **Storage** to save the modified task list.

8) Logic returns a feedback string to **GUI** to inform **GUI** of the result of the execution.
9) If execution is successful, **GUI** needs to refresh its display. It uses `retrieveAllData` to get the updated task list.
10) Again, Logic will call Storage to load the task list.
11) Logic returns the updated task list to **GUI**.
12) **GUI** displays the updated task list to the user.

In general, these 12 steps apply to every type of user command. The only differences are the type of command object that Logic creates and how the command object creates or modifies a task. Depending on the currently active view of TankTask, **GUI** may also invoke a different method to retrieve different data for the particular task list that it is displaying.

4 Components

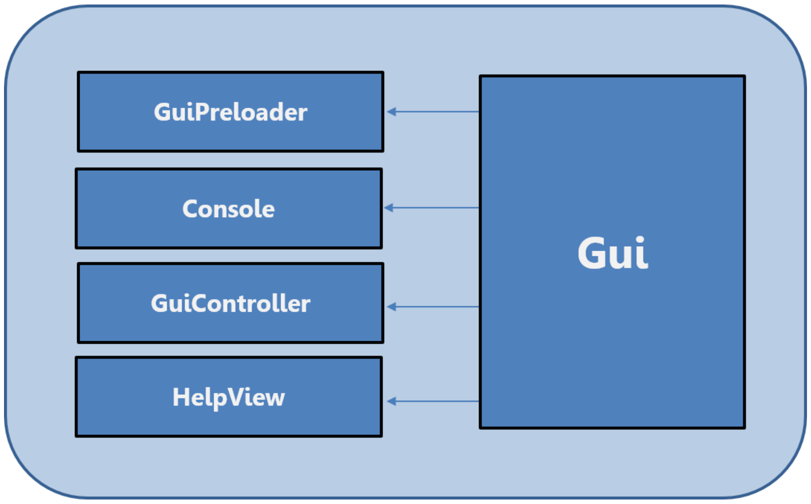
4.1 GUI

The **GUI** component's role is to act as a platform for interaction between the user and the program. It accepts inputs (commands) from the user, and displays the outputs (task lists and feedback messages) to the user. This means that GUI is in charge of everything that the user sees and how the user uses the program.

Before you start developing the **GUI**, we recommend that you get a good understanding of **JavaFX**, as it is used heavily in **GUI**'s codes. Also, you will have to understand how **GUI** displays the information, how it controls the input and output, and how it communicates with the other components.

4.1.1 Sub-components

The **GUI** component is made up of 5 sub-components, as shown in Figure 3 below. As you can see, the **GUI** component has a main class (with the same name **Gui**) that depends on four other more specialized sub-components.



[Figure 3] Class diagram of GUI

The sub-components are described in the following list:

1) The **Gui class** controls the main display of the program. It creates and handles the JavaFX components that are shown to the users. This includes everything from displaying the task lists to the switching of views.

2) **GuiPreloader** sets up the pre-loader for the program. The pre-loader is the small loading screen that is displayed to the user as the components of **TankTask** are being set up. Having a pre-loader is necessary so that the user is aware that the program is still loading. Once the program is successfully loaded, the pre-loader will disappear.
3) **Console** is used to direct the default output stream of the program into the feedback console of **GUI**. This is so that GUI can print messages (such as feedback or error messages) and display them to the user.
4) **GuiController** is in charge of interacting with the **Logic** component. It passes user input to **Logic**, and in turn receives information on the success of the command execution and the updated task lists from Logic. The information is then sent to the **Gui class** to be displayed.
5) **HelpView** contains all the data that is related to the help window. It controls the way in which the help window is displayed, as well as the help content that is shown to the user.

4.1.2 Display

GUI allows the user to switch between different **views**. As the GUI developer, you need to learn how to display information for each type of view.

The **default view** shows an overview of the tasks and displays the tasks in three separate task lists that are based on the type of the tasks (ToDo, Event or Floating). For each task, essential information such as the task name, index, and dates (if any) are shown. You may examine the methods `setUpHeadings` and `setUpContents` in the GUI class to learn how the task lists' headings and contents are being displayed.

The display format for a task is determined by the **Storage** component, more specifically the **Task** class, as the information shown for each task is dependent on its data. For example, if the user sets a reminder for a particular task, the line 'Reminder has been set' will be shown at the end of the task.

The **focus view** is used to display more details of a task. It has a column on the left that displays only one task list at a time, while details of the selected task is displayed in the panel on the right. Similar to default view, you may examine the methods `setUpFocusHeadings` and `setUpFocusHeadings` in the GUI class to learn about display in focus view.

Whenever the user scrolls through the task list, the right panel should be refreshed with the information of the currently selected task. When the user switches to another list, the left column should be refreshed to show the current list.

Each time a command is executed, the views need to be refreshed and redisplayed so that the user can see a live update of the changes made by the command. You can look to the method `refresh` to see how this is done.

4.1.3 Interaction between GUI and User

In **TankTask**, the main interaction between the user and **GUI** will be from the commands entered by the user. The execution of the commands is mainly handled by the backend components (**Logic**, **Parser** and **Storage**). Thus, as the **GUI** developer, your job here is simply to send the command to the **Logic** component in the method `executeCommand`.

However, to make our program more convenient for the user, we also allow the user to perform certain actions using **hotkeys**. Pressing of hotkeys are instantaneous and do not require any form of parsing or processing. Hence, hotkeys are mainly handled by the **GUI** component.

Currently, the different hotkeys that can be used include 'enter', 'tab', 'esc', 'shift' , 'ctrl-z', 'ctrl-y', the function keys (F1-F4) and the arrow keys. You may refer to the help list to learn about all the available hotkeys.

Some of these hotkeys are straightforward, for example 'enter' simply takes in the user input and sends it to **Logic**. The hotkeys that are worth taking note of are those that change the views and display of the program. One example is 'tab', which causes **TankTask** to switch between default and focus view. The focus view also allows the users to navigate through the use of arrow keys – Up & Down to traverse through the current list, and Left & Right to change the list. Your job will be to keep track and retrieve the corresponding list and details when these keys are pressed. You can refer to methods in the **GUI class**, such as `eventUp` and `eventDown`, to see what **GUI** does when these hotkeys are being pressed.

4.1.4 Communication with Other Components

Although the data that are displayed by **GUI** mostly comes from **Storage**, **GUI** only interacts with **Logic** directly, and **Logic** will be the component that obtains the data from **Storage**.

Every time the user enters a command, **GUI** takes the command and passes it to Logic. After command execution, Logic will return a string to **GUI**, which may be a feedback message (to show that the command execution is successful), an error message, or a command for **GUI** to execute (for example, if the command is 'exit', **GUI** will cause the program to exit). If the string is a message, **GUI** will display it to the user via the feedback console.

After a command is executed, **GUI** will have to get the updated state of each task list and displays them. To get the updated states, the **Gui class** will call **GuiController** to retrieve the updated contents from **Logic**. **GuiController** will retrieve the data for the default task lists using the method `retrieveAllData`. If the program is in focus view, **GUIController** will use other variants of the method to retrieve data for the specific task list.

4.1.5 Notable Methods

As the main component of **TankTask**, **GUI** is essentially at the 'front' of the program. It does not need to be called by other components of the program, as it is the one that is giving out requests to the other components. Thus, **GUI** does not have any method in its API.

To summarize, the **GUI** component essentially has 3 jobs – 1) It takes in the **user input** and passes it to **Logic**, 2) It determines what happens when the user presses a **hotkey**, and 3) It displays the task lists and **refreshes** them frequently by obtaining the updated data from **Logic**. Thus, the methods below are important in ensuring that GUI can fulfil its jobs:

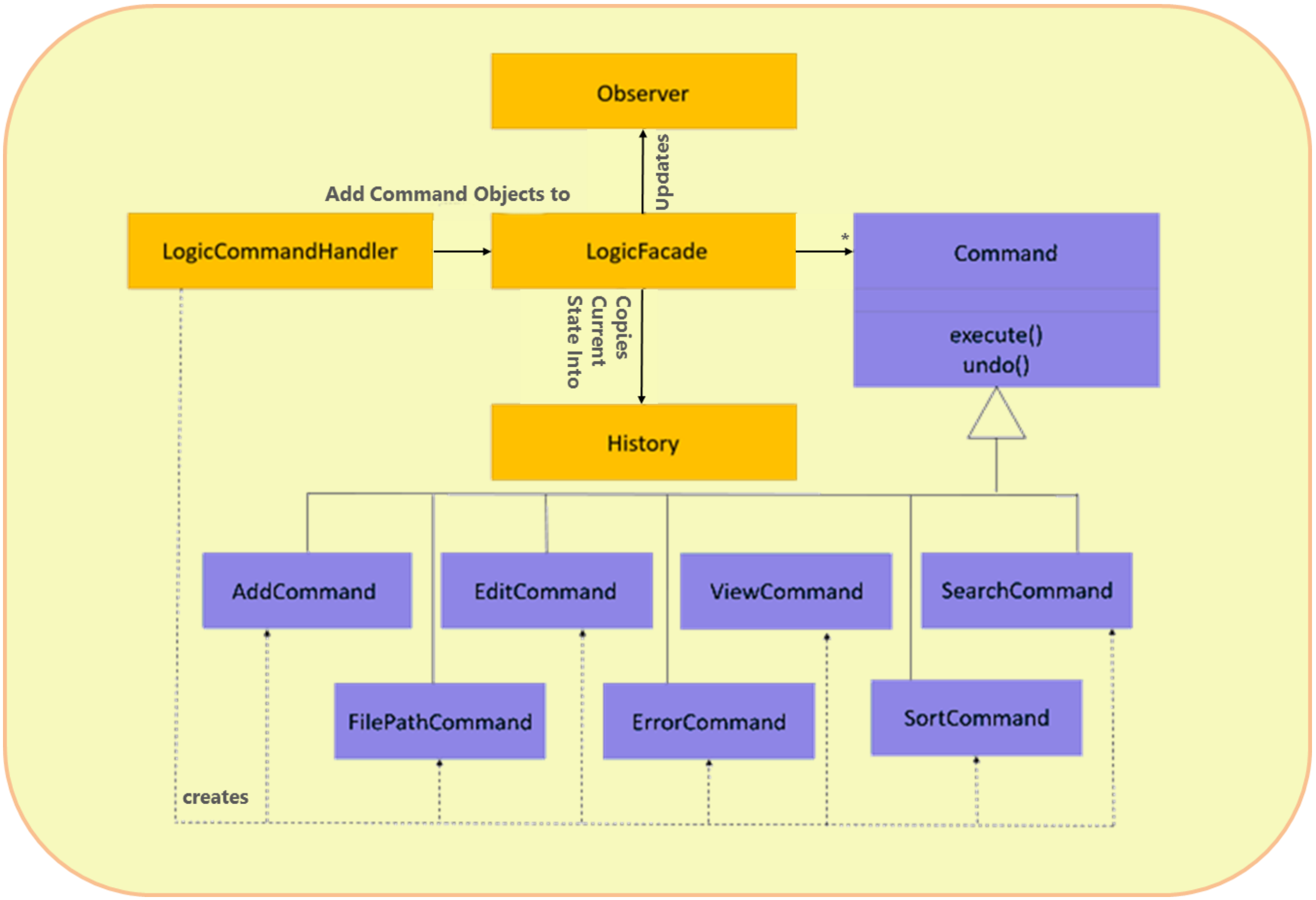
Return Type	Method & Description
`void`	`runEvents()` Retrieves the input from the user, clears the text field and passes it to **Logic**. Receives a string value from **Logic**, and determines if any follow up actions is required.
`void`	`determineEvents()` Determines how the user and GUI interact through the use of hotkeys. For example, when the user press 'Enter', GUI will run the `userInputCommands` method.
`void`	`retrieveAllData()` Obtains all necessary task lists from Logic component.

[Figure 4] Notable Methods of GUI

4.2 Logic

The Logic component handles the core functions of **TankTask**. It determines the commands to be executed, and transfers data between different components. It is also the only component that interacts with every component of the program.

Figure 5 below shows Logic's class diagram. Logic implements both the **Command** and **Observer** pattern. The next few sections will explain how these patterns are applied in Logic.



[Figure 5] Class diagram of Logic

4.2.1 API

Although Logic interacts with every TankTask component, **GUI** is the only component that actively calls Logic. Hence, the methods in Logic's API are meant for transferring data between **GUI** and Logic.

Return Type	Method and Description
String	`execute(String userInput)` Handles the execution of user inputs
String	`retrieveTaskData(String dataType)` Retrieves data to display the updated task list(s)
String	`retrieveSearchData()` Retrieves data to display the search result

[Figure 6] API of Logic

4.2.2 Logic Command Structure

The **LogicFacade** class plays the role of the 'façade', as it is the class that directly interacts with both the **GUI** and **Parser**. When a user input is passed to Logic, it is received by **LogicFacade**. **LogicFacade** will first send the user input to **Parser** for parsing using the method `parseInput`, and receive the parsed input as an ArrayList<String> object.

The ArrayList<String> object will then be forwarded to the **LogicCommandHandler** class, to be constructed into a **Command** object containing the type of command and the variables needed to execute the command. As a Logic developer, what you need to know is that **LogicCommandHandler** will determine the specific type of **Command** to create depending on the command type ascertained in the ArrayList<String> object. The following list shows the current variations (subclasses) of the **Command** class and their uses.

- **AddCommand** – For creation of tasks.
- **EditCommand** – For editing existing tasks
- **ViewCommand** – For retrieval of specific types of tasks.
- **SearchCommand** – For searching of tasks using keywords and then storing them in the search result list.
- **SortCommand** – For sorting of tasks according to the user's preference
- **ErrorCommand** – For displaying specified input errors determined by **Parser**
- **FilePathCommand** – For changing of the location of storage data according to the user's preference

Commands such as **AddCommand** and **EditCommand** will be able to retrieve the list of all **Task** objects available in the **Storage** component. Details of the retrieval process will be explained further in the **Storage** section. What you need to know is that to manipulate the **Tasks**, Logic simply has to execute the **Command**, and the rest is handled by **Storage**.

Also, by extending from the **Command** superclass, it is easy to add new types of commands to Logic. Applying the **Command** pattern here, the **LogicFacade** class can simply run the `execute` or `undo` method on each **Command** object without knowing the exact type of the command or what it does. Once execution is done, the **Command** object will be stored in a **Stack** Object in **LogicFacade** for easier retrieval, if they need to be executed again.

4.2.3 Adding New Tasks

Adding a new task is done using **AddCommand**. **LogicCommandHandler** will decipher the ArrayList<String> object returned from **Parser**. If it contains the command word `'add'`, **LogicCommandHandler** will create a new **AddCommand** object. Once the object is created, it will be passed back to **LogicFacade** for execution. The new **Task** object created will then be stored in **Storage**. This process applies to any type of task, whether is it a **ToDo**, **Event** or **Floating**. In the future, should there be a new type of **Task**, you can simply include the new task type in the **AddCommand** class.

4.2.4 Editing Existing Tasks

All operations for updating existing tasks are done using **EditCommand**. Similar to **AddCommand**, **LogicFacade** will receive the Command Object from **LogicCommandHandler** and execute it, except now the type of **Command** object received is **EditCommand**.

EditCommand is used for updating the various fields in a **Task** object. Take note that depending on the fields updated, a **Task** object's task type may change. The following list shows examples as to how a **Task** may change type.

- **Floating**, when set with a deadline, will become a **ToDo**.
- **Floating**, when set with a start and end date/time will become an **Event**.
- Removal of all date/time will revert an **Event** or **ToDo** back into a **Floating**.

In the future, should you want to add a new particular attribute or field to the **Task** class, simply expand the **EditCommand** class by including getter and setter methods for the new attribute/field.

4.2.5 Retrieving Existing Tasks

The **ViewCommand** class has a different purpose compared to the previous two variations of the **Command** class. **ViewCommand** relays the parsed user inputs back to the **GUI** so that **GUI** can display a specific list of tasks to the user. For example, the user may want to see only tasks that are due today. **GUI** will then have to dynamically change its view to show the user what he wants to see.

This is a good time to introduce the **Observer** sub-component. As seen in Figure 5. **Observer** will observe **LogicFacade** and, depending on the user inputs, update the data to fit what the user wants to see. By applying the **Observer** pattern, **LogicFacade** will update **Observer** on changes in the task lists, so that **Observer** does not have to call **LogicFacade** purposely to get updates. After every command execution, **GUI** can simply retrieve the modified data from **Observer**.

Always remember that all interactions within the **Logic** component must be made through the **LogicFacade** Class. In the future, should you wish to have a new way of displaying data, such as displaying all tasks for the current month, simply expand **Observer** with the new method of data retrieval.

4.2.6 Reverting to an Earlier State

This method `undo` is only available for **Command** objects that make permanent changes to the data, such as **AddCommand** and **EditCommand**. This is a good time to introduce the **History** sub-component. There are two types of stack in the **History** sub-component, the `stateUndo` and `stateRedo` stack. Before any add or edit commands are executed, the current data state is stored in **History**'s `stateUndo` stack. After the command has been executed, the modified state will also be put into the stateUndo stack. This allows the program's state to revert should the user chooses to undo his commands. To undo, **LogicFacade** will simply restore the old data from previous states.

For example, state B is the modified state A. If the user executes 'undo', state B will be stored in `stateRedo` (so that the user can execute 'redo' if he changes his mind again), while state A, which is taken from `stateUndo`, will be used to overwrite the current state. When 'redo' is executed, state B will be transferred back `stateUndo` and be used to overwrite.

4.2.7 Searching for Tasks

SearchCommand uses keywords in the user input to search for tasks. The user input is tokenized and search results are matched using the "AND" search method (meaning the task has to contain every word in the search query). This allows the user to obtain search results as closest as possible to what the users wants.

At present, the search algorithm will search through all the fields in each particular task. If they are of an exact match, they will be included in the search results. In the future, if you want to implement an improved search algorithm, simply expand this class and include your new search algorithm.

4.2.8 Sorting for Tasks

SortCommand contains different sorting methods that allow the user to sort the tasks by priority, name or date. By default, **TankTask** sorts the tasks by dates. If the user were to use the sort function, the effect will last until the program closes, as **TankTask** will sort the tasks by dates again every time the program boots up.

In the future, should you want to use a new sorting algorithm or make a permanent sort, then expand this class with your new methods and you are good to go.

4.2.9 Changing File Location

FilePathCommand is used to move the task data file of TankTask to the new location that is specified by the user. Here, your job as the **Logic** developer is to simply supply **Storage** with the correct target address. The file and data transfer operations can be left to the Storage developer.

4.2.10 Displaying Errors

ErrorCommand takes the errors that **Parser** has identified in the user input, and transfer the errors to **GUI**, so that GUI can display the error message to the user. Your job as the **Logic** developer is to simply ensure that the error messages are transferred properly.

4.3 Parser

The **Parser** component's job is to parse the user input. It splits, interprets, and processes the user input into an ArrayList<String> of strings that can be handled by the other components.

As the **Parser** developer, you must understand the workings of the various sub-components in the Parser and how they interact with each other. You also need to understand the different stages that take place during parsing.

4.3.1 API

Parser is essentially an independent component, as it does not call methods from other components. The only external interaction **Parser** has is with **Logic**. Every time the user enters a command, **Logic** will call the method `parseInput` to send the command to **Parser** for parsing.

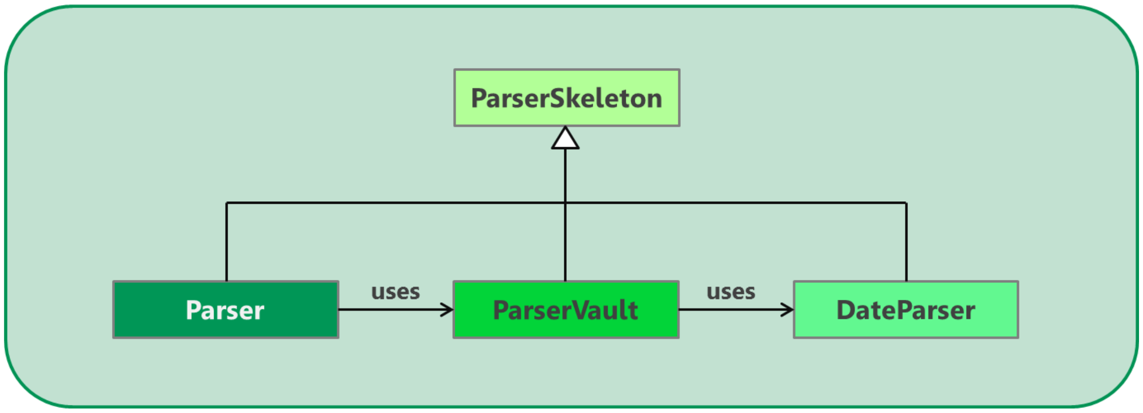
As such, the only method in **Parser**'s API is `parseInput`, and it should remain as the only public method of **Parser**. The other components do not need to know or access the specific functions of **Parser**. Every input that needs to be parsed can simply be sent via `parseInput`, and **Parser** will handle the rest.

Return Type	Method and Description
`ArrayList<String>`	`parseInput(String userInput)` Splits the input into tokens and starts the parsing process.

[Figure 6] API of Parser

4.3.2 Sub-components

The **Parser** component is made up of 3 sub-components: the **Parser class**, the **ParserVault** class and the **DateParser** class. All 3 classes are sub-classes of the abstract class **ParserSkeleton**, and their relationships are illustrated below:



[Figure 7] Class diagram of Parser

- The **Parser class**, which shares the same name as the component itself, is the main class of the **Parser** component. Its main job is to split the user input and scans the input for command words and information.
- **ParserVault** is like a 'vault' that stores information that has been parsed by the **Parser class**. **ParserVault** will then process the stored content, and generates the result by assembling the content into an ArrayList<String>..
- **DateParser** checks if a given string can be interpreted a date, and parses the date string into a standardized format. It is usually invoked when the user command includes a date (e.g. when setting a deadline or a reminder).
- **ParserSkeleton** is an abstract class that contains data shared between all sub-components of the **Parser**, such as a list of all command words. It also contains basic methods that are useful to all sub-components, such as `isNumber` (checks if a token is a number) and `removeEndSpacesOrBrackets`.

The entire parsing process can be split into two main stages, as you will see in the next few sections.

4.3.3 Parser Class

To understand how parsing works, we will walk you through the process, starting from **Parser class**, in the steps below:

1) **Parser class** receives the input from **Logic** with the method `parseInput`, and splits the input into tokens, so that the input can be parsed word by word.
2) **Parser class** will first parse the first two words, using `parseFirstTwoWords`. If the first 2 words contain a command with a dominating effect (e.g. 'exit' or 'delete'), **Parser class** is done parsing, and will return the result made by **ParserVault**. If not, **Parser class** continues to parse the rest of the input using `parseRemaining`.
3) `parseRemaining` scans the remaining tokens from start to end for command keywords. When a command word is encountered, it is stored into **ParserVault**.
4) The tokens that come after the command word will be used to 'grow' (i.e. accumulate in) a `growingToken`, until another command word is encountered, or when the end of the input is reached. The `growingToken` will then be stored in **ParserVault** under the field of the last command word.
5) Steps 3 and 4 are repeated until all tokens are parsed. **Parser class** also checks for invalidity in the process. Any error encountered in the input will cause **Parser class** to return an error.
6) **Parser class** invokes **ParserVault** to generate the result using the stored content.

You have just gone through the first stage of the parsing process. To sum up, what happens in this stage is that **Parser class** parses the input into command words and contents, and stores them into **ParserVault** for further processing later. You should examine the code to learn about other small but important operations that are omitted in the steps above.

4.3.4 ParserVault

Now, we move on to the second stage, where the stored contents are processed and assembled by **ParserVault**.

To develop **ParserVault**, you should be aware that **ParserVault** has 4 tasks:

1) Stores the contents that are parsed by the **Parser class**.
2) Decides on the type of result to create, based on the command words found in the input.
3) Checks that the content of each field is valid
4) Assembles the contents into an ArrayList<String> and returns the result.

Task 1 takes place concurrently with the first stage. This is where **ParserVault** keeps track of command words seen by storing them in `seenCommands`. The information of command word will be stored in `fieldContent` under the corresponding field (e.g. the information of command word 'description' will be stored under field `'description'`).

Task 2 happens after **Parser class** is done parsing. **ParserVault** uses different methods to generate different results. For example, for single word input like 'exit', the method `makeCommandOnlyResult` is invoked. Other times, if the user input only has one command word, the result is made based on that command word (e.g. if command word is 'show', **ParserVault** uses `makeShowResult`). Also, `makeMultiFieldResult` is used for inputs with more than 1 command word.

Next, in Task 3, **ParserVault** checks that each relevant field has valid content. For example in `makePriorityResult`, **ParserVault** will ensure that the priority is within 1 to 5. If content is invalid, **ParserVault** will generate an error result.

Task 4 happens after the validity check. If another developer is developing **Logic**, it is important that he/she knows about the fields that are included in the result. For example, for `'delete'`, **Parser** will return `['delete', <Task Index>]`. For `'search'`, the result is `['search', <search query>]`. Refer to the appendix for the full list of result types.

4.3.5 DateParser

DateParser is the most specialized sub-component of **Parser**. Having this sub-component is necessary as date parsing is rather complex. It is built to be able to work for different date formats, including dates typed in natural languages.

To ease your burden as the developer, **DateParser** makes use of an external natural language date parser called **Natty**, which can recognise date keywords like 'today' or 'Tuesday', and generates a formatted date string. However, due to a few restrictions of Natty and specific requirements of **TankTask**, the date parsing needs to be enhanced in several ways.

To begin, you can refer to the method `parseDate`, which contains all the important steps in the date parsing process:

1) `swapDayAndMonth` swaps the positions of day and month in the string. This is because **Natty** only recognises the MDY (month-day-year) format, while we recommend users to enter the dates in the more common DMY format.
2) `addDigitsToYear` adds the first two digits of the current year to the string, in the case where the user only types the last two digits (e.g. '15' instead of '2015'). This is needed as **Natty** has problem deciphering two-digit years.
3) `parseDateWithNatty` uses **Natty** to parse the string into a formatted date string.
4) `standardizeDateFormat` converts the date string into the format that will be displayed in **TankTask**

5) `confirmDateIsInFuture` ensures that the date generated is in the future (e.g. when the user types 'Jan 15', if the current month is already past January, the year will be set to the next year rather than the current year).
6) `removeMinuteIfZero` removes unneeded zeroes (e.g. 9:00pm will be converted to 9pm).

Hence, to adjust or improve the date parsing, you may simply add more methods to `parseDate` or edit these methods.

DateParser also does other date-related functions. For example, `isDayOfWeek` checks if the given token is a day of the week (e.g. Friday), and `hasNoTime` checks if the given string contains time in any acceptable format (e.g. 9pm or 21:00). You should be able to learn and use these methods by yourself as they are quite self-explanatory.

4.3.6 Parser Expansion

As the **Parser** developer, it is very likely that you will have to implement new command words as the program becomes more complex. To do so, follow these simple steps:

1) In **ParserSkeleton**, create the new command word, and categorise the command word by putting it in the correct lists (e.g. Is the command word a dominating command? Does the command require content?)
2) In **ParserVault**, create a field for the new command if necessary. Add the new field to `fieldContent`.
3) In **ParserVault**, create a new method to generate result for the new field, or reuse an existing method.
4) Process the content of the field as required (e.g. Does it need to be parsed to date? Does it have any limitations?)

For example, if you want to create a new feature that allows user to set category for a task, you can create the new command word 'category'. If the user can specify the category with words, then the behaviour of the 'category' will be similar to 'description'. Otherwise, you can create a new method for category to make its content more restrictive.

You may notice in **ParserVault** a list called `command_families`, which contains the variants or abbreviations for certain command words. This is where you can create alternative forms for an existing command (e.g. 'cat' for 'category').

Besides adding command words, you are also free to enhance or modify the **Parser** in any way you deemed fit. But as far as possible, you should consider reusing the existing methods and structures rather than creating new ones. You should also keep the number of methods in **ParserSkeleton** to a minimum, as it is an abstract class that should contain only methods used among all sub-components. When you need to make a new sub-component, it should be a subclass of **ParserSkeleton**. It should also be unnecessary to have any bi-directional dependencies between the sub-components (i.e. a specialized sub-component like **DateParser** should not depend on a less specialized one like the **Parser class**).

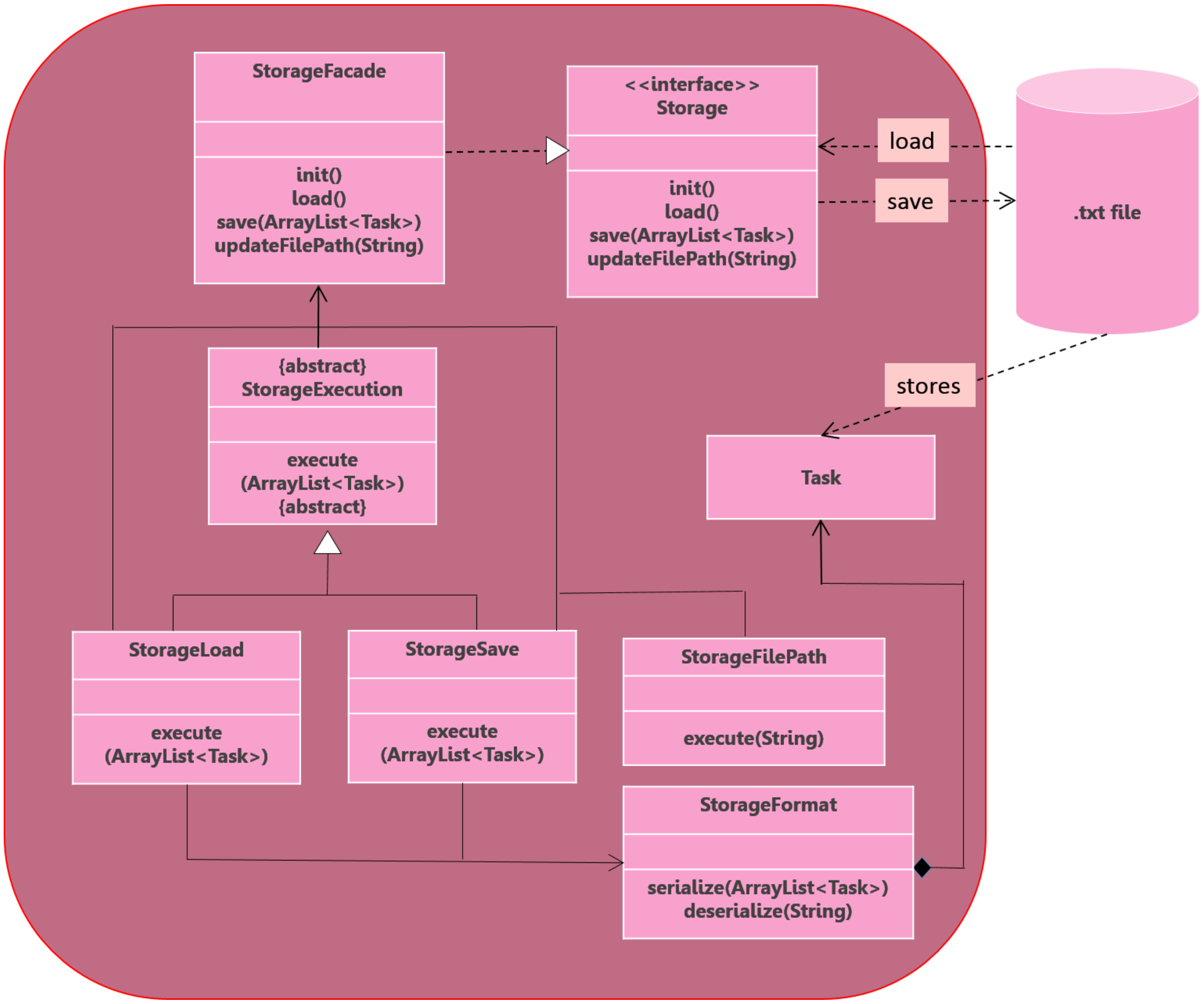
4.4 Storage

The **Storage** component is in charge of loading and saving tasks into a database. When the tasks are being used by the program, they are handled in the form of **Task** objects. When the tasks are stored on the hard disk, they are stored in a **.txt file** as plaintext in the **JSON** (JavaScript Object Notation) format. Currently, TankTask only supports storing data in **JSON** text format, so as the **Storage** developer, you will first need to know how to use **JSON**.

Thus, accessing the tasks data mainly involves reading data from and writing data to the **.txt file**. To retrieve a task's information, the data in the **.txt file** will be converted back to a **Task** object and passed to **Logic** for further action.

4.4.1 Sub-components

Figure 8 below gives an overview of how the sub-components of **Storage** work together to load and save tasks:



[Figure 8] Class Diagram of Storage

The following list describes the sub-components in more detail:

1) The **Storage interface** contains all public methods used by **Logic** to load from or save data to the **.txt file**. It also contains the method updateFilePath which update the file path when the user decides to change the file location of the **.txt file**

2) **StorageFacade** implements **Storage interface**, where each of its methods (load, save and updateFilePath) calls the execute method from **StorageLoad**, **StorageSave** and **StorageFilePath** respectively. Thus, it acts as a façade as the jobs of loading, saving and updating file paths are actually being done by the internal classes.
3) **StorageExecution** is an abstract class that provides the abstract operation of execute for its two sub-classes, **StorageLoad** and **StorageSave**. The sub-classes each implement their own version of the execute operation (one for loading, one for saving).
4) **StorageLoad** loads the task data into the program by reading the .txt file.
5) **StorageSave** saves the task data on to the hard disk by writing to the .txt file.
6) **StorageFilePath** updates the file path by creating a new file in the new file location specified by the user and transferring all task data from the old file to the new file.
7) **StorageFormat** is used for the conversion of task data from plaintext to **Task** objects and vice versa. It is used by **StorageLoad** to deserialize text to **Task** objects, and used by **StorageSave** to serialize **Task** objects to text.

4.4.2 API

All the core methods of Storage (load, save and updateFilePath) are public methods, and are all hence part of Storage's API. load and save are frequently invoked by **Logic** when updating task data. updateFilePath is invoked by **Logic** only when the user enters the 'filepath' command, which will change the file location of the **.txt file**.

Return Type	Method and Description
ArrayList<Task>	`load()` Loads all data from file and deserialize text into Task objects.
void	`save(ArrayList<Task> allData)` Serialize all Task objects into human-readable text and writes into file.
boolean	`updateFilePath(String newFilepath)` Updates new text file location for data storage.

[Figure 9] API of Storage

5 Testing

TankTask uses **JUnit4** to perform system level testing on the backend components. You are able to obtain **JUnit4** at http://junit.org/. In addition if you are using **Eclipse IDE**, You can get **EclEmma Java Code Coverage 2.3.2** from Eclipse Marketplace. Together, these tools will help check how comprehensive your test cases are by determining the percentage of lines of code that has been executed by your various test cases.

We employ white box testing so that each test case is more targeted. Try to make your test cases cover as many lines of code in **TankTask** as you can. Do take note that as designed by the JUnit developers, each test case should run independently of each other. Hence, the orders at which your test cases run are not pre-determined. It would be useful to clear all entries in your **TankTask** text file before you run your testing operations. This is so that the numbering or listing of tasks can be accurately pre-determined and compared with the test results.

All the system-level test cases can be found in the src/test/main.java.gui in the **TankTaskSystemTest.java**. All the component-level test cases can be found in the respective packages in the src/test folder.

When adding new features, be sure to add new test cases frequently to check and maintain the integrity of the components and sub-components. Ensure that they are still functioning normally before doing a merge request. It is the responsibility of you, the **TankTask** developer to ensure that your newly modified codes are fully functional before merging to the git repository.

6 Future Development

Here are some recommendations for you to develop this project further.

You can implement a new feature that organises tasks into **categories**. We recognise the possible need for our users to separate tasks of different purposes. For example, a user may have task originating from his work or personal life. By assigning categories to the tasks, a user would be able to display all the tasks within a specific category, so that it is easier for the user to tackle tasks from one category at a time.

You can also choose to implement a **sub-tasking** mechanism within each task. Based on prior research, we identified that some tasks can be are too big and intimidating for a user to handle, which deters the user from getting started on the task. With sub-tasks, **TankTask** can help the user break down each task into smaller bite-sized pieces, so that the user would find them more manageable. It would also prevent the user from procrastinating on their tasks.

Another possible area for improvement is the **search** algorithm. The current search requires exact matches before the task is included in the search results. But there are times where the user may have spelling errors or trouble remembering the exact keywords. Hence, you can attempt to make our search algorithm work even in these scenarios.

You have come to the end of this developer guide. We hope that with our guidance, you will now able to contribute regularly to the development of **TankTask**. Welcome aboard! ☺