Developer Guide

Team F14-3: RoomShare

Authors:

Teh Zi Huai
Tran Minh Duong, Harry
Tay Yee En, Ryan
Tay Tze-wei, Caleb

Table Of Contents

1. Introduction:

In this section, we will cover an overview of the RoomShare program, what it is, who it is for and how it works. It also includes basic instructions on how to use this developer guide.
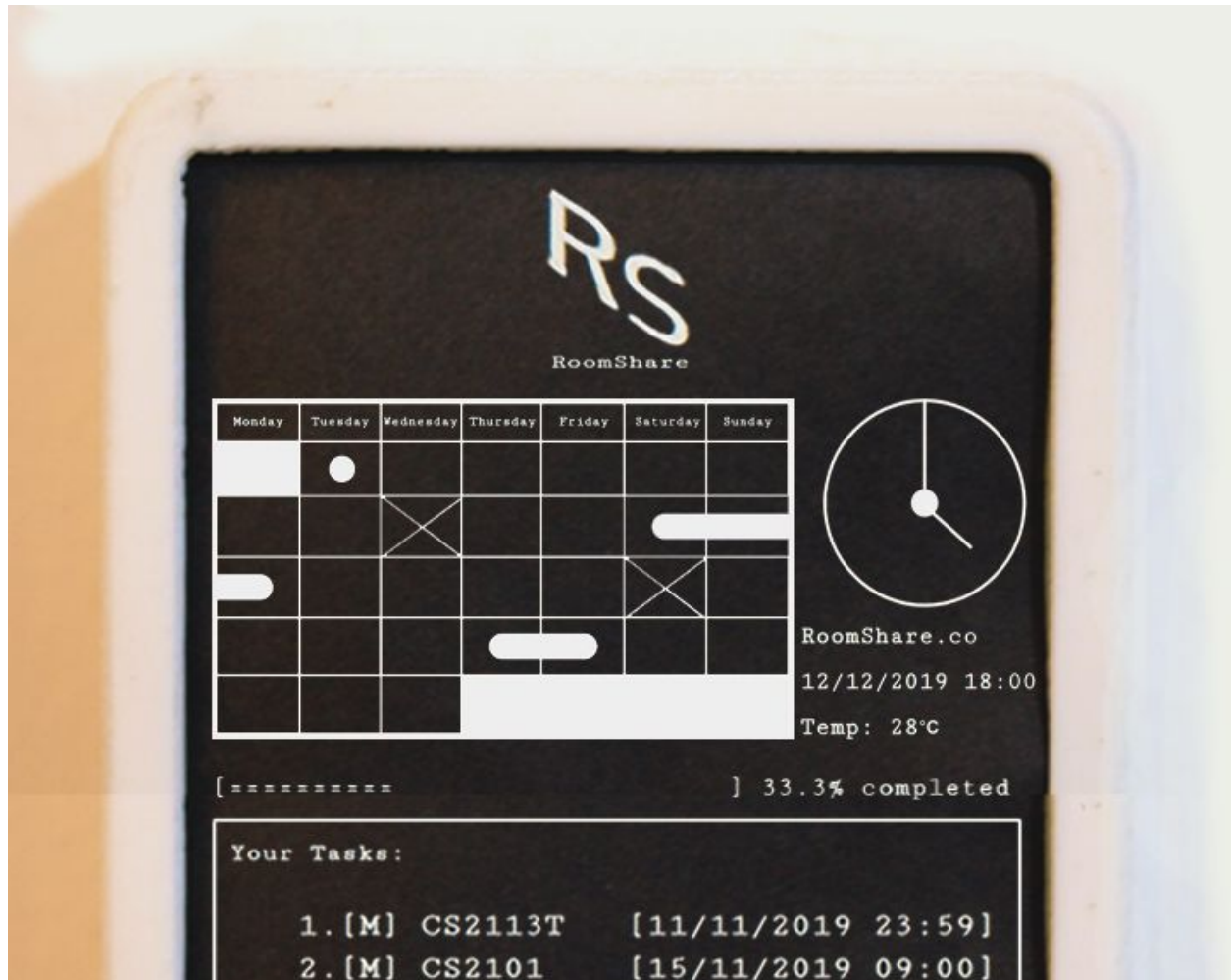
Fig 1.1 Mockup of RoomShare Device and Operating System

1.1 Overview

RoomShare is a program to help you keep track of your tasks in the office. You can keep track of things like assignments given to you (Assignment), meetings that are about to occur (Meeting), and also individuals who are taking a leave of absence from work (Leave). RoomShare has various functions to help keep track of your tasks, allowing you to focus on your work.

This developer guide is to provide programmers working on RoomShare add-ons and features with a better understanding of how RoomShare works. This includes how to set up the project on their workstation, software architecture as well as the various classes/functions used and how they work.

1.2 Target group

Our product is targeted at office workers who find it difficult to track tasks across many channels. In the office, there are various channels of communication, such as emails and chat applications. This creates a problem where tasks of different importance are spread out, resulting in unnecessary time being spent figuring out the allocated tasks for the day or week.

1.3 Value Proposition

Our project will provide five crucial points to aid in this target group we have selected.

We will be having a tagging system to allow names to be tagged to certain items in RoomShare, which helps in the organisation of the tasks.

Sub-tasks can be added to individual assignments, allowing users to see the agenda for the task at hand, or break the task down into smaller steps.

Priority management will be added to help users visualise the tasks that they need to do first, rather than just by the entry order. This reduces the time they would need to spend on figuring out which task has to be done first.

A progress bar showing percentage of task completion will aid in making the task list easier to read as users will be able to easily visualise how much they have completed. This will help them to keep track of their deadlines better.

Automation of updating recurring tasks (tasks part of their daily/weekly routine) will allow users to save time by eliminating the need to update these tasks repeatedly.

1.4 Using this Developer Guide

To use this developer guide most effectively, we recommend you reading the chapters in order. However if you want to check a specific topic, refer to the table below.

Chapter:	Summary:
2. Setting up (page 5)	How to install required software and setup the work environment to begin development on RoomShare.
3. Design (page 6)	Architecture of the software and how the different components interact with one another.
4. Implementation (page 8)	Various Classes used and how their functions work.
5. Appendix (page 30)	Additional information, developmental progress and future plans for the project.

2. Setting up:

Here we will run through step by step the process of setting up the work environment to start development on RoomShare.

2.1 Prerequisites

1. JDK 11
2. IntelliJ IDE

Fig 2.1 IntelliJ Icon

2.2 Setting up the project in your computer

1. Fork the repo at this website, and clone the fork to your computer
   https://github.com/RoomShare/

2. Open IntelliJ

3. Ensure that the correct JDk version is set up for Gradle
   I. Click Configure > Structure for New Projects
   II. Click New and find the directory of your JDK11

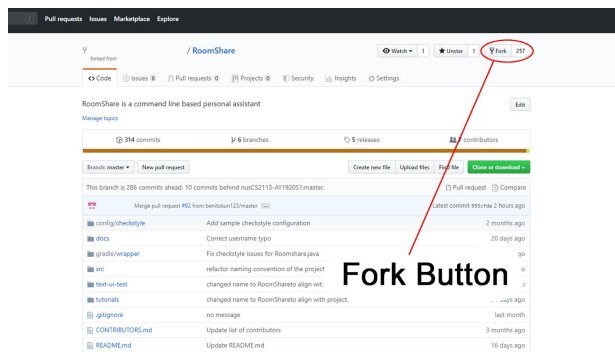4. Click import project, and import it as a gradle project

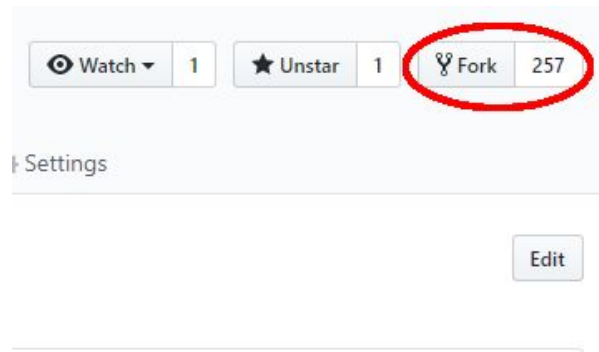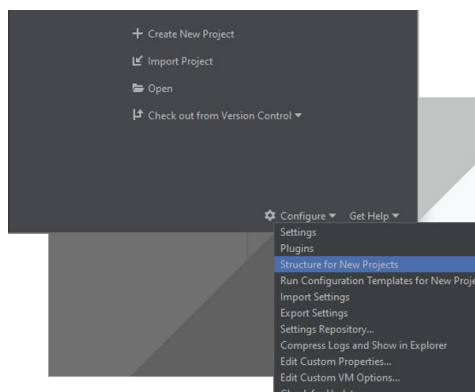Fig 2.2 RoomShare Github page

Fig 2.3 Fork button

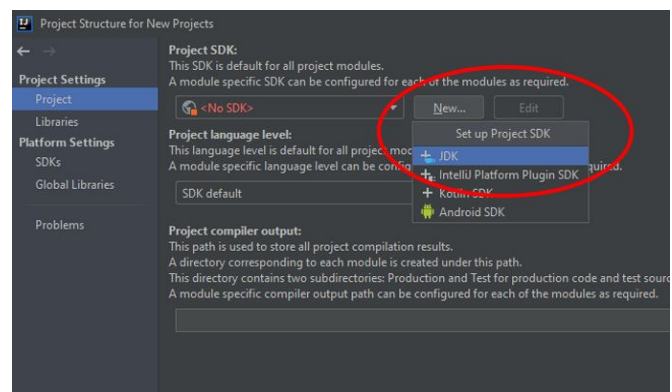Fig 2.4 Configure Structure from IntelliJ
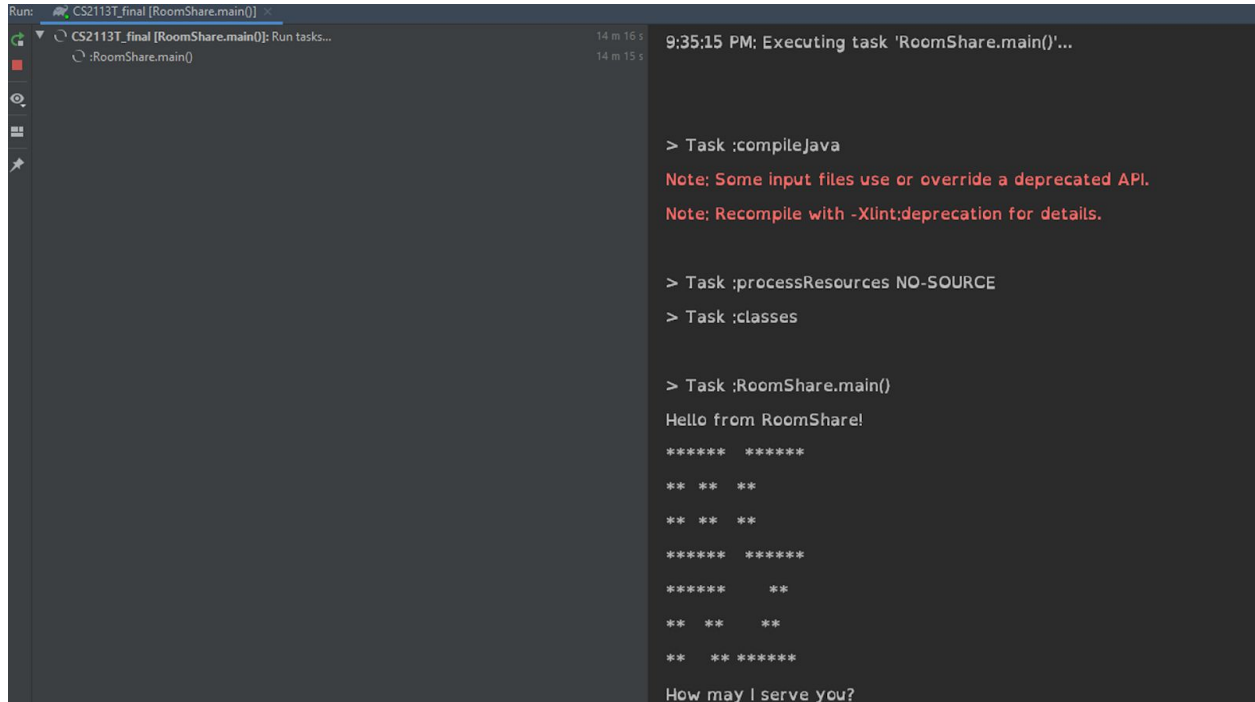
Fig 2.5 Add JDK11

4

2.3 Verifying the setup

1. Run the test by using gradle and ensure all the tests pass

After setting up the project, do take some time to read through the architecture of RoomShare. This will help you get a better idea of the design for RoomShare.

Fig 2.6 Successful Gradle Launch

3. Design

In this section we will discuss the Architectural design of RoomShare and explain how the various packages & classes used interact with one another.

3.1 Architecture
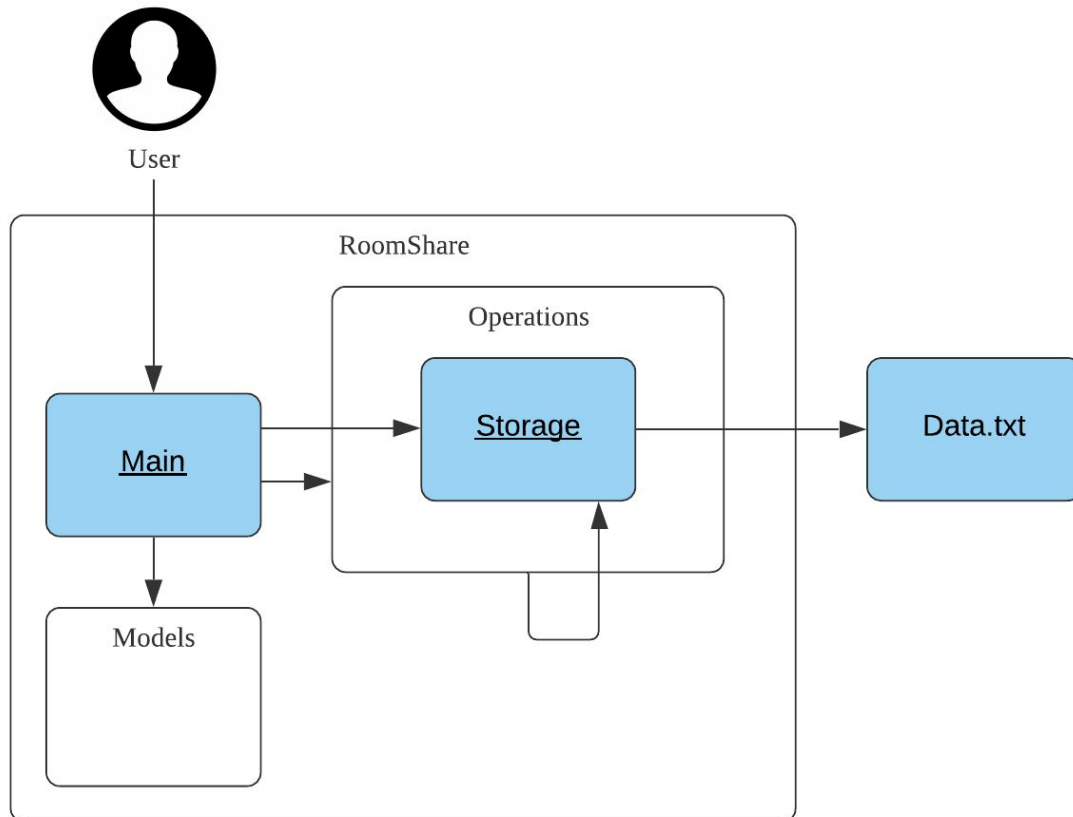
Fig 3.1 Architecture Diagram

The Architecture Diagram shown above explains the high-level design of the App.

Main has one class called RoomShare. It is responsible for:

- At app launch: Initializes the components in the correct sequence, and connects them up with each other
- At shut down: Shut down the components and saves whatever data necessary

Model contains the classes which are used to carry and store data in the app.
Operations contains the classes which handles the user inputs and data, with specific logic flow to carry out the instructions of the user. Storage, which is a subset of Operations, handles the writing of data to an external text file to store the user data.

How the architectural components interact with each other

The Sequence Diagram below show how the components interact with each other for the scenario where the user issues the command
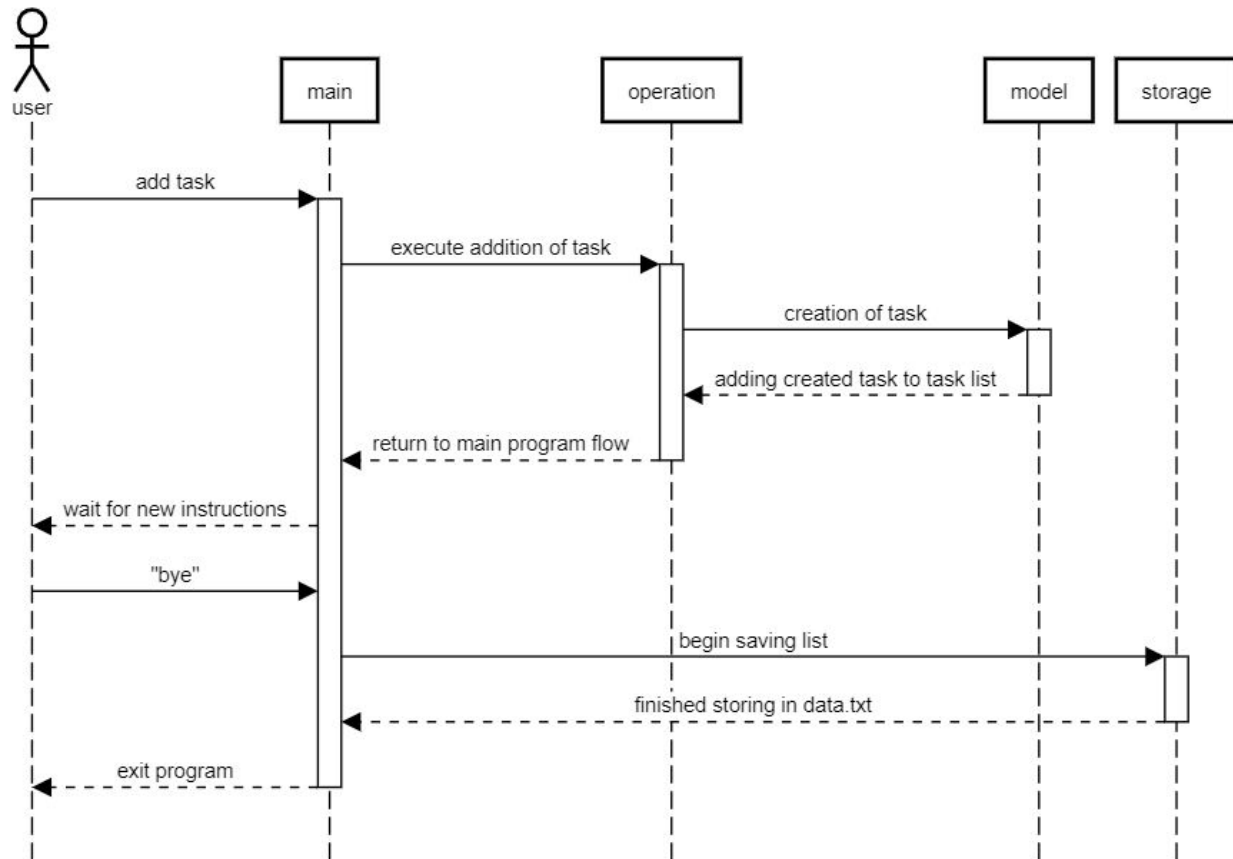
Fig 3.2: Component interactions for addition of task and exiting RoomShare

3.2 Packages

Below is a list of all the Classes found in the different packages

3.2.1 Model_Classes

Classes for the various tasks users can store. Task is the parent class and stores basic info such as the task's description, date, whether it is completed, priority & who it is assigned to. The type specific functions and variables are stored in the class extensions themselves, for example the Meeting class can store a fixed duration whereas the Assignment class cannot.

Classes in Model_Classes: Task, Assignment, Meeting, Leave, TaskReminder, ProgressBar

3.2.2 Enums

Enums contains all the enums (class type used to define) used in the development of this program. They are mainly used to define constants we want to check in the parser such as commands or error types in order to make the information easier to understand.

Classes in Enums: ExceptionType, HelpType, Priority, RecurranceSceduleType, ReplyType, ReportType, SaveType, SortType, TaskType, TimeUnit

3.2.3 Operations

Operations contains the classes used to carry out various commands in the program. These include classes to store and load information such as Storage and TaskList, the Parser class to process information, TaskCreator and SubTaskCreator to create tasks, CheckAnomaly for error detection and UI which prints information to interact with the user.

Classes in Operations: CheckAnomaly, Help, Parser, RecurHandler, Storage subTaskCreator, TaskCreator. TaskList, TempDeleteList, UI

3.2.4 CustomExceptions

Contains the RoomShareException class which manages exception handling within RoomShare.

Classes in CustomException: RoomShareException

3.3 Main
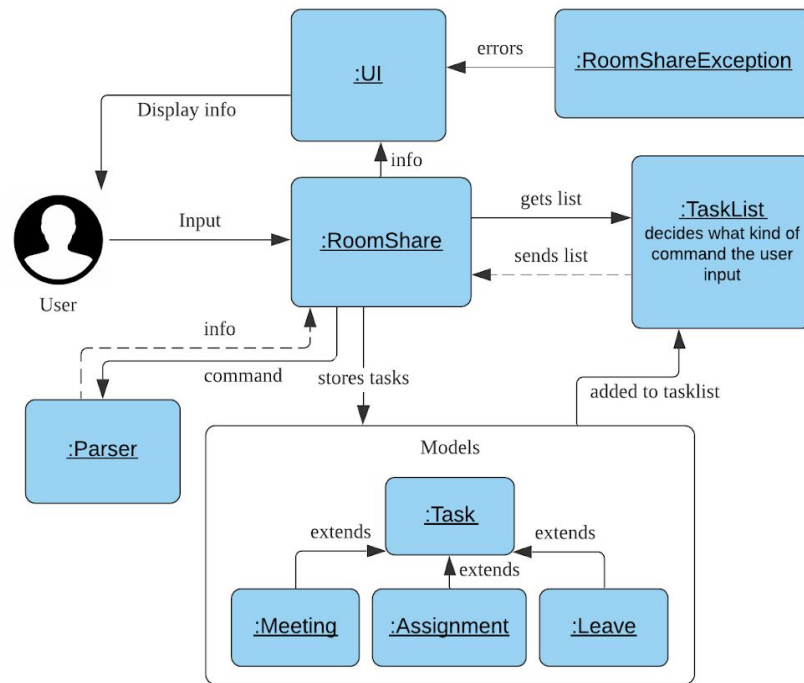
Fig. 3.3 User Interface Diagram

The main class contains instances of the model class which are split into different packages. These packages are: CustomException, Enums, ModelClasses and Operations. Each package has the following descriptions to identify what each package mainly handles, as seen in the table 3.1 shown below.

Additionally, task data is stored in text files which are loaded when the program boots up. This exit the program without worrying that their data will be lost.

To see more detailed information on the classes of each of the packages, move on to the next section on the implementation of RoomShare

4. Implementation

This section describes some of RoomShare's important features and how they work. These include command input, various sorting methods and subTasks.
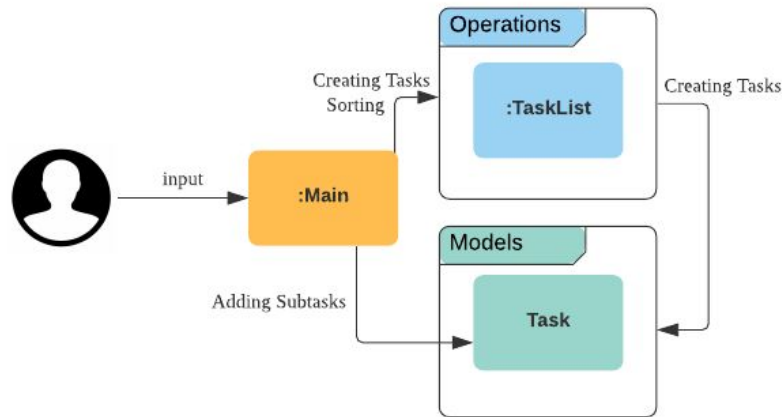
Fig 4: Diagram of how user input is implemented across packages

4.1: One line task creation

4.1.1 Current Implementation

The creation of tasks is handled by TaskCreator. It is a class in the Operations package that deals entirely with the process of creating task objects. These objects are from the ModelClasses package, and they are Assignment, Meeting and Leave. Each of these classes extend the Task class.

These objects have special conditions listed as so:
1. Assignment: Must have a deadline set
2. Meeting: Must have a date set, and can have a fixed duration specified
3. Leave: Must have both a date, fixed duration and user specified

These objects can also have various parameters associated with them, such as:
- Priority
- Recurrence
- Tagged name
- Fixed durations

During the task creation process, the user will enter "add", followed by the specifications of the task they wish to add. The specifications do not have to come in a particular order, and not all of these specifications must be set immediately on creation. The table below describes the list of all specifications, as well as the required formatting of each specification. The highlighted specifications are mandatory fields in task creation.

Table 4.1.1: list of specifications and the required formatting

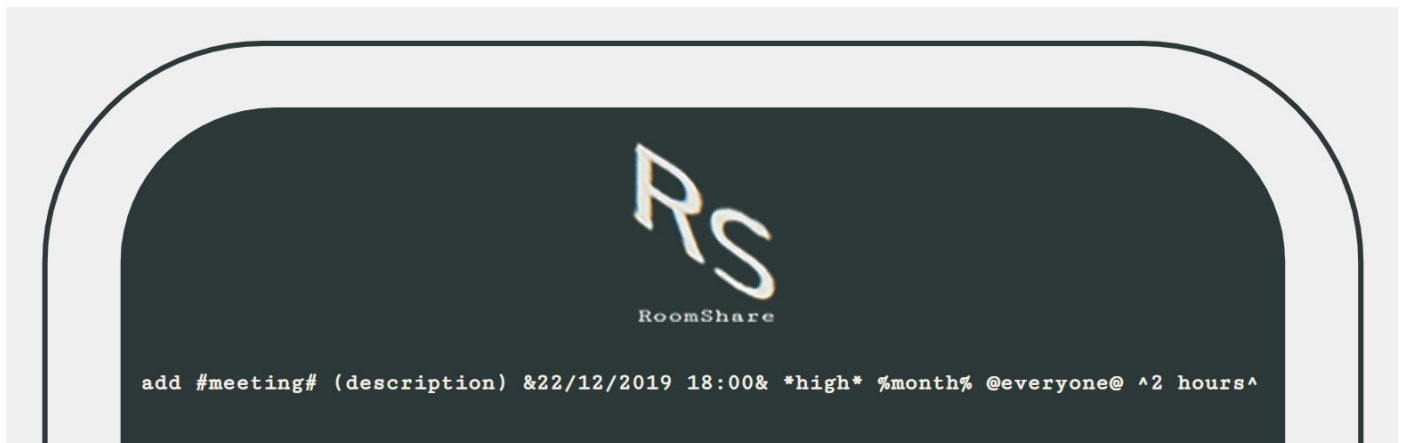| specification	delimiter	example
Task type	#	#meeting#
Description	()	(progress report)
Date	&	&22/12/2019 18:00&, or &22/12/2019 09:00&23/12/2019 18:00&
Priority	*	*high*
Recurrence	%	%month%
Tagged name	@	@everyone@
Fixed duration	^	^2 hours^

A sample entry would be as such:

Fig 4.1.1 sample entry input

Additionally, there are some additional formats for dates, with a simpler date scheduling for timing that we already know. Date specifications can also be input as:
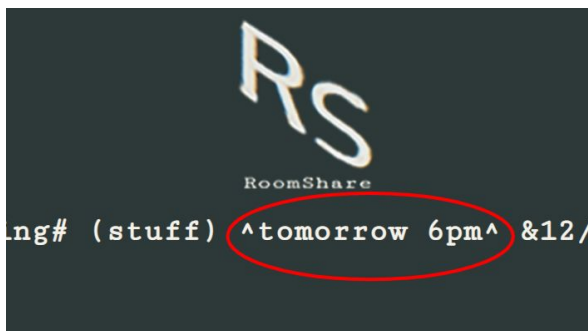
Fig 4.1.2 alternate date style 1
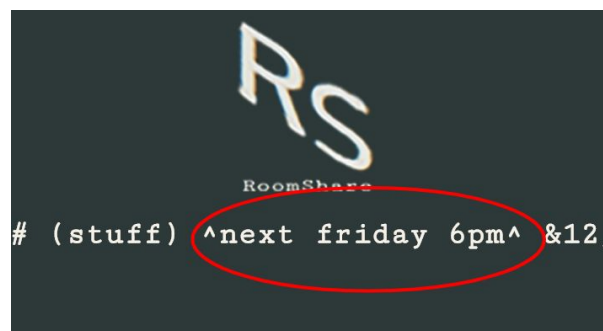
Fig 4.1.3 alternate date style 2

This eliminates the need for lengthy typing of the date field during task creation.

Do take note that Leave will require a different type of formatting for the date, and will ignore all fixed duration specifications in the input string. This facilitates the handling of leave of absence, which can possibly be lengthy.

The creation of tasks will now be shown below:

Step 1:
The user inputs the command and the task specifications. This input string will be read by RoomShare through the use of the Parser class. The input string is then passed into TaskCreator.

Step 2:
TaskCreator will be invoked with its create() method. The input string will be broken based on each of the mentioned flags, and the relevant information will be formatted for creation of tasks. Absent or incorrect information will be set to a default value instead.

Step 3:
TaskCreator returns the task, and the main class then adds the task into the task list. The main class then waits for a new command.

The sequence diagram for this operation is shown below in figure 4.1.1
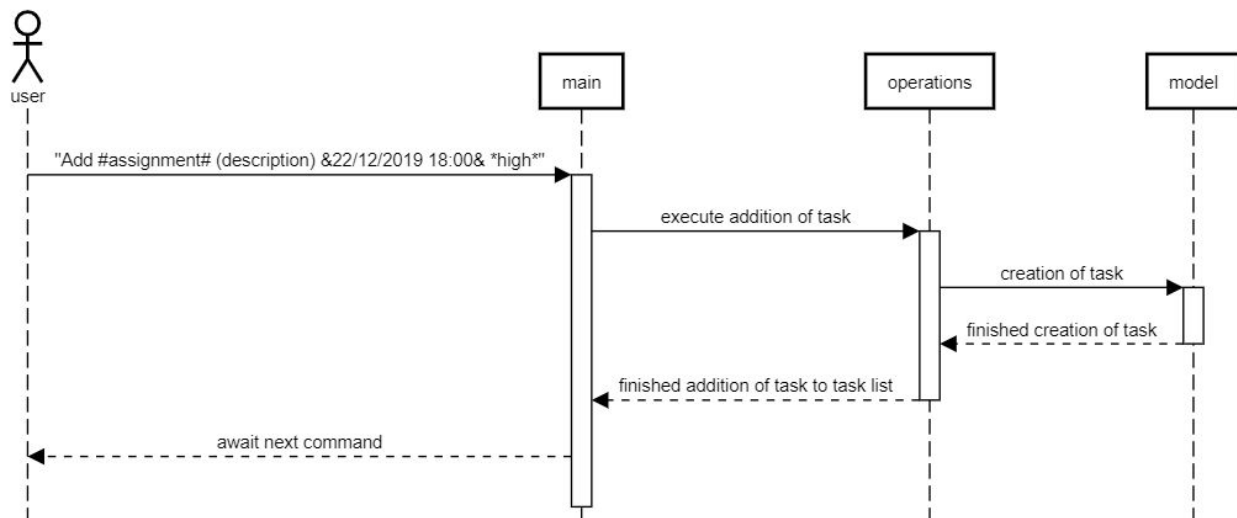
Fig 4.1.4: Sequence diagram of creating a task

4.1.2 Design considerations

Aspect: Input string
This area was designed with fast typers in mind. Users need to be used to the formatting of the input.

Aspect: specific format using only one type of flag
Pros:
- Only one special character needed
Cons:
- Requires memorising of exact placement of each field
- Empty fields also require the special character, and multiple empty fields can become hard to differentiate

Aspect: Using different flags for each field (Current choice)
Pros:
- Fields do not need to be in specific placements
- Empty fields can simply be omitted from the input
Cons:
- Multiple flags need to memorised
- Special characters used cannot be used in the input string, limiting user's choice of characters.

Evaluation:
Using different flags is more favourable, due to the increase in flexibility of the input. The special characters used in each flag are not commonly used when describing an Assignment, Meeting or Leave.

4.2 Priority

4.2.1 Current Implementation

The priority function is done through the sorting of the task list, which is an ArrayList by structure. The sorting is done through the Comparator class, and the order of sorting is done in descending order of importance, denoted by the SortType of the task list.
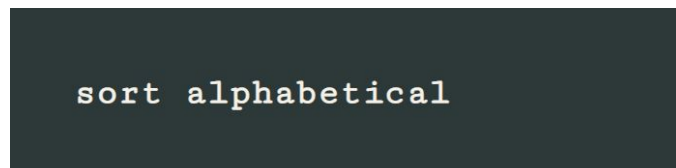
Sorting will be invoked upon any addition of tasks to the task list, or when the list method is called.

The sorting of the task list can be changed using the sort command, where the user specifies the parameter that they wish to sort the task list by. The parameters to sort by are:
- Priority, which is the default setting
- By alphabetical order
- By the closest deadline to the current date.

Additionally, all the tasks that have been completed are automatically sorted into the bottom of the task list, regardless of the current sorting type of the task list.

The user can change the sort type by calling the sort command, followed by the parameter they wish to sort by. This is shown in figure~ below.

Fig4.2.1: example of changing the sort type

If there are any errors in the changing of sort type, the sort type will automatically be set to the default of by priority of the tasks. This is done through exception handling, where the wrongly entered sort type causes a RoomShareException to be thrown, and the exception is then caught and the program then sets the sort type to the default of priority.

The sequence diagram to illustrate the process of changing sort type is shown below:

sequence diagram for changing sort type

Fig4.2.2: sequence diagram of changing sort type

The following section now describes the sorting mechanism:

Step 1:
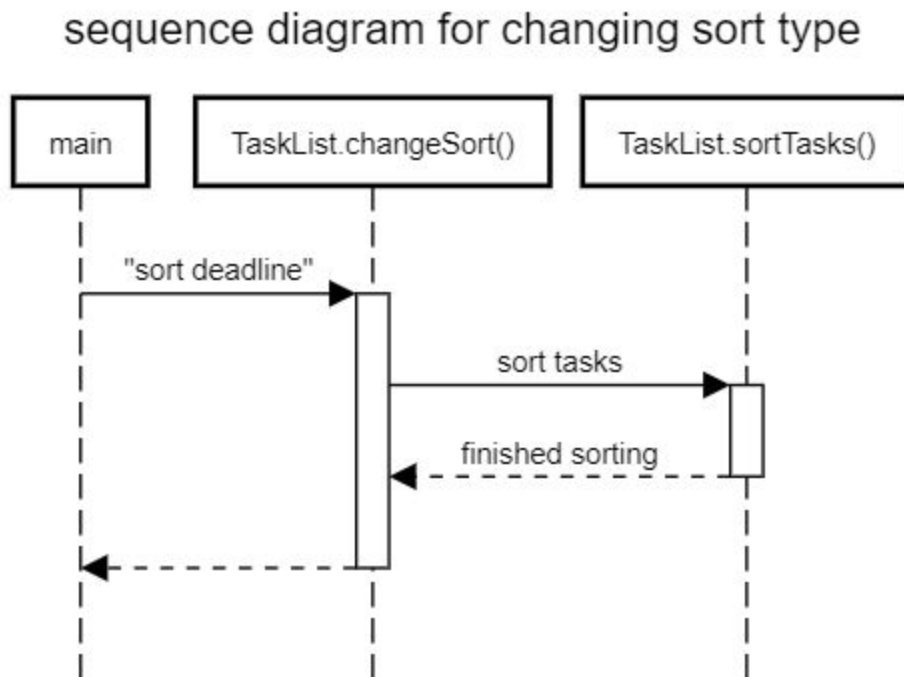The task list calls the sortTask() method, which checks the sort type of the task list and will call the method associated with the respective sort types. This relationship is described in the table below.

Table 4.2.1: sort types and the associated sort method

Sort type	method
priority	comparePriority()
alphabetical	compareAlphabetical()
deadline	compareDeadline()

Step 2:
The Comparator class compares the tasks based on the specified sort type. Each sorting method has its own individual implementation. However, all completed tasks will be sorted into the bottom of the list. This is accomplished by a logical check of the completion status of the task by the getDone() method.

comparePriority():
The tasks are sorted based on their priority that is given by getValue(). The values of each priority is given below in Table 4.2.2

Table 4.2.2:

Priority	value
high	0
medium	1
low	2

The task list is then sorted based on Comparator.comparingInt().

The resultant task list will then be sorted based on descending order of priority. The sequence diagram of the sorting operation is shown below

compareAlphabetical():
The tasks are sorted based on the alphabetical order of their description. The two Strings representing the descriptions are compared using the compareTo() method of Strings.

compareDeadline():

The tasks are sorted based on the date parameter. The dates are compared using the compareTo() method, and the closer date is sorted into the top of the list.

The sequence diagram of the sorting function is shown in the figure below
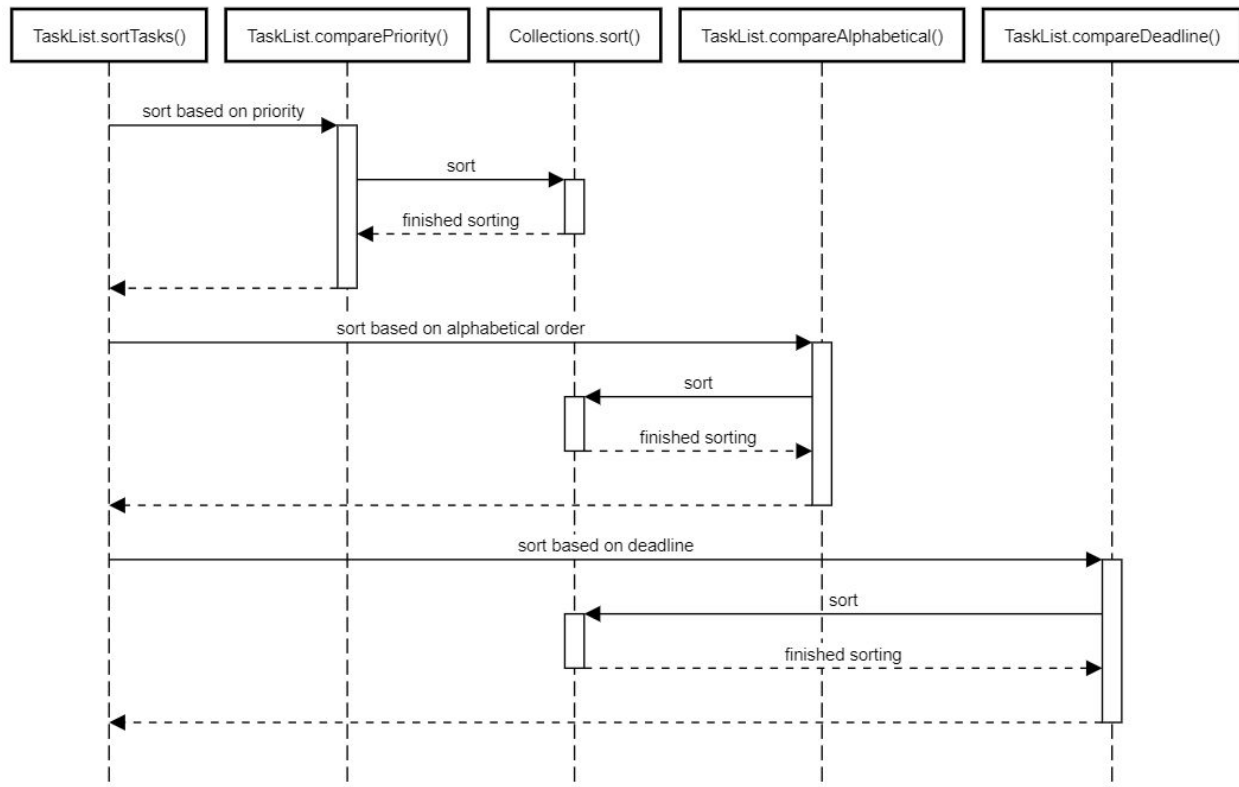
Fig4.2.3: sequence diagram of the sorting

4.2.2 Design considerations

Aspect: Data structure to support the sorting function

Alternative 1: use an array
- Pros: simple implementation for starting programmers who are unfamiliar with list data structures
- Cons: Has a fixed storage size, not good when task list becomes large

Alternative 2 (current choice): use an ArrayList
- Pros: relatively simple implementation; supports many functions that are integral to a task list

4.3.1 Current implementation

Subtasks are strings that are appended to classes such as Assignment and Meeting. The list of subtasks are stored as an ArrayList of Strings. These subtasks will be listed when the list command is called by the user.

On calling the subtasks command, the subtask subroutine will be run in main. The user must have a valid input string of like the example given below:

subtask 1 task1,task2,task3

This creates a subtask for the task at the front of the task list, with the three separate subtasks of task1, task2, task3. To define different subtasks in one line, the user must split the subtasks with a comma (,).

The steps involved in creating a subtask is shown below

Step 1:
User inputs the command to add a subtask, as given in the example shown above.

Step 2:
The subtasks will be set if the index given is valid. If the index given is invalid or there are no Assignment type objects, then an error message will be displayed to the user.

The sequence diagram for the subtask operation is shown below
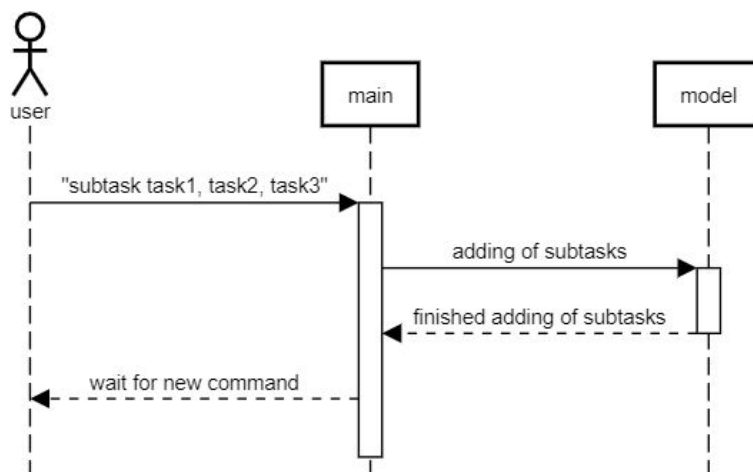
Figure 4.3.1: sequence diagram of subtask command

4.3.2: Design Considerations

Aspect: All task types support sub tasks

Pros:

- Implementation will be in abstract class Task
- Reduced checking of task types when manipulating information

Cons:

- Unnecessary for Meetings and Leave
- May cause confusion for users when task type which do not need subtasks get a subtask entry by accident due to human error

Aspect: Only Assignment supports subtasks (Current Choice)

- Most pertinent to assignments, since assignments usually have large amount of work to be done.

4.4.1 Current implementation

Once the tasks have been added into the task list, users can update the task via the update command. The update command uses the TaskCreator class to check for the updated fields sets the information in the tasks to the new value specified by the user.

All information in the tasks can be changed with the setter methods in their parent Task class. The TaskCreator class will scan the input string similar to how the addition of tasks. Upon encountering the same flags, an update will be carried out on the task.
Upon encountering errors in the update of description, TaskCreator will set the information back to the default values.

The steps in updating a task are shown below:

Step 1:
The user inputs the update command. TaskCreator receives the command via the Parser class, and TaskCreator calls the update() method on the specified task.

Step 2:
TaskCreator scans the input string for the flags used in the task creation. Upon encountering the flags, the appropriate extraction methods are called to update the information of the task.

Although the ArrayList Collections class supports swapping of objects, there will not be any swapping of object classes by creating a new task and swapping it in at the correct index. This is to prevent any formatting errors. Another side consideration for this is that it is simply easier to update the tasks with their respective getter methods.

The sequence diagram for this process is shown in the figure below:
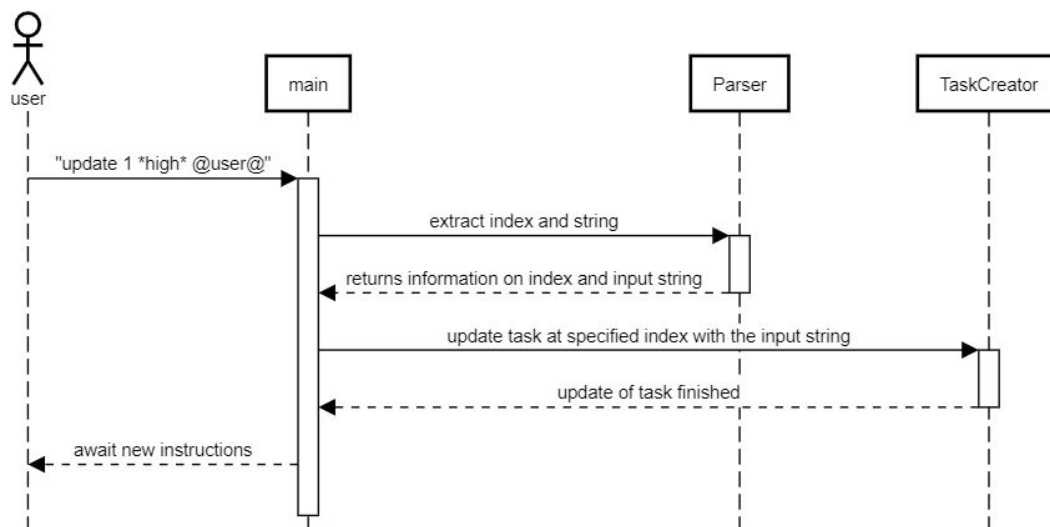
Fig4.4.1: sequence diagram for task update.

<u>4.4.2: Design Considerations</u>

Aspect: Create new task entirely, and swap it for the existing task

Not a favourable way to implement, as it requires extracting the information from the old task, ensuring the updated information is used in the creation of the new task, then creating and swapping the new task with the old task. Many complicated steps are involved in this method

Aspect: Use setters of the Task parent class to update the tasks (Current Choice)
Pros:
- Simple implementation
Cons:
- Tasks that do not share the same kind of information details may end up having their information updated at the same time

4.5: Storage

<u>4.5.1: Current Implementation</u>

Storage uses a BufferedReader to read from text files, and BufferedWriter to write to text files. The information in the tasks are extracted and formatted to an appropriate form in Storage, and written into a file named data.txt for saving. When loading a file, data.txt is read and Storage converts the formatted information into parameters for creating tasks, and creates tasks from that information. Storage also handles the writing of log files for the task list.

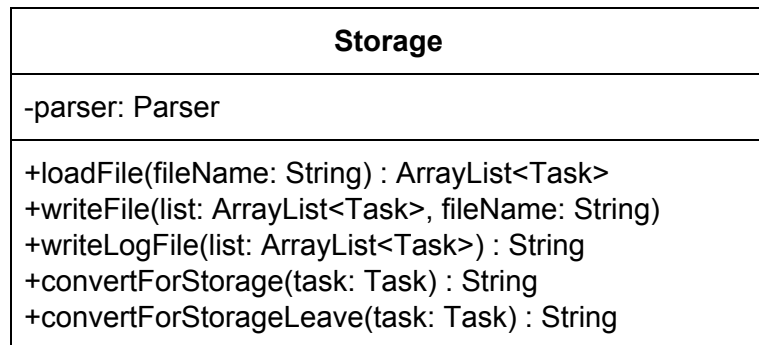The following figure shows the structure of the Storage Class.

Storage
-parser: Parser
+loadFile(fileName: String) : ArrayList<Task> +writeFile(list: ArrayList<Task>, fileName: String) +writeLogFile(list: ArrayList<Task>) : String +convertForStorage(task: Task) : String +convertForStorageLeave(task: Task) : String

Fig4.5.1: class diagram of Storage Class

The steps below will describe the saving process of the Storage Class

Step 1:
Storage uses the writeFile() method to begin the file writing process. A BufferedWriter is opened

Step 2:
Storage extracts the information from the tasks in the list iteratively. Each task will have its information formatted in the following manner as described by the table below.

Table 4.5.1: format of storage

entry	field	sample
0	type	A for Assignment, M for Meeting, L for Leave
1	done	y for done, n for not done
2	priority	high, med, low
3	description	Progress update
4	date	22/12/2019 18:00
5	recurrence	day, week, month, none
6	assignee	everyone

7	isFixedDuration	F for fixed duration, N for not fixed duration
8	duration	2 (must be a number)
9	Time unit	Minutes, hours, days, weeks, months, unDefined
10	subtasks	Task1, task2, task3 (each subtask must be split by a comma)

Each field must be separated from the previous by a #. A sample formatted information may look like the following:

M#n#high#progress update#22/12/2019 18:00#week#everyone#F#2#hours##

The date field is formatted by using the convertForStorage() and convertForStorageLeave() methods.

The sequence diagram of the file writing is shown below
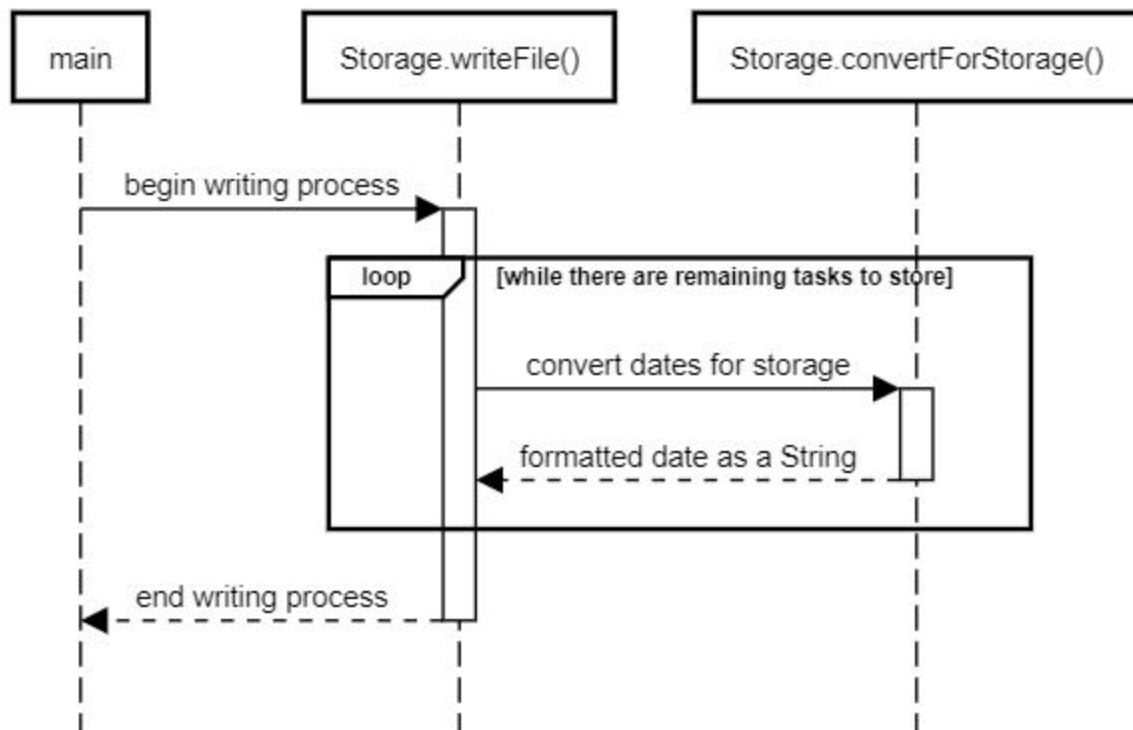

Fig4.5.2: sequence diagram of writing process

Additionally, the writeLogFile() process uses a similar method of storage, but the main difference is that the log file contains no formatting as it is meant for the average user to read. The information will be stored as plain English with no other formatting

When reading from the file, the following steps are carried out

Step 1:
Storage.loadFile() is called, and it opens a BufferedReader to read from the file. The information is extracted in order and used to create Tasks

Step 2:
An ArrayList is populated with the Tasks, and returned to the main program flow.

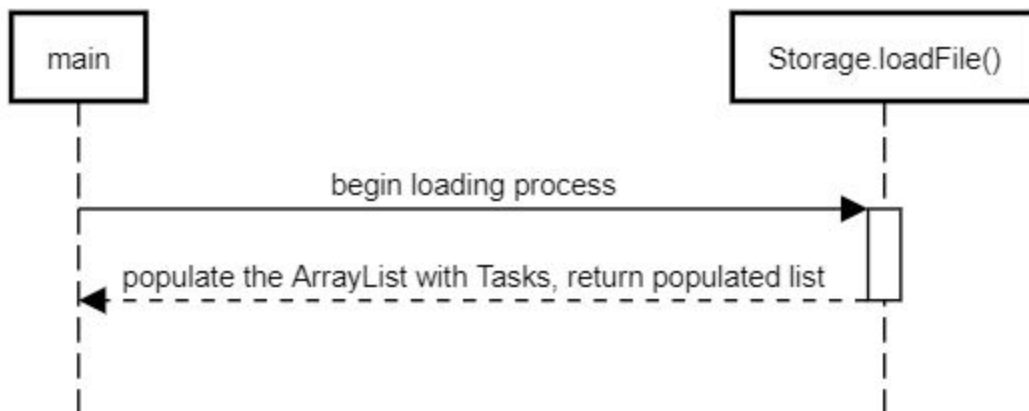The sequence diagram is as follows for the loading of files

Fig 4.5.3: sequence diagram for loading process

Due to the use of # as a flag for extracting information, it is imperative that # do not appear in the Tasks' information

5 Testing

This section will explain the steps to setup the test environment for RoomShare, how to run test classes and how to integrate your own test classes in the future.

5.1 Setting Up Test Environment

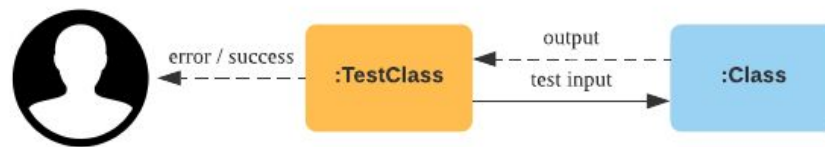Now we will run through the steps to set up the test environment in IntelliJ.

Fig 5.1 Test Class diagram

Steps:

1. In IntelliJ, create a test folder in the same directory as the main folder.
2. Create another folder named "java". This will be where we store the test classes.
3. Open the Project Structure window - File > Project Structure
4. Click on Modules on the left panel and navigate to your "java" folder in the middle portion of the screen
5. Click on the "java' folder and press the green "Tests" icon in the "Mark as:" bar
6. Press "Apply" and then "ok" to save the changes. The java folder should be highlighted in green now.
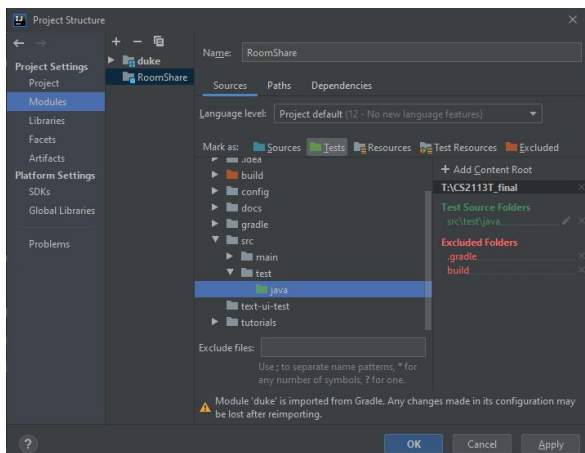
Fig 5.2 Project Structure window
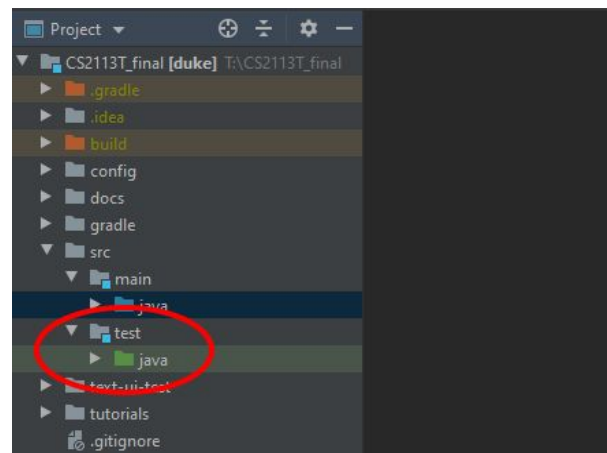
Fig 5.3 "java" folder highlighted in green

Now that we have setup the test environment, we will be able to create test classes inside.

Testing of RoomShare is mainly done using gradle. To run tests, you can use the terminal in IntelliJ and type in the command "gradlew test". The tests should run automatically.

As you can see from the diagram below, the terminal will print "BUILD SUCCESSFUL" if all the tests pass.

Fig 5.4 Using the Terminal to run Test code

Alternatively, you may access this option quickly through the gradle panel on the right of the screen under Source Sets > test

Fig 5.5 Test through Gradle Panel

In order to properly create test classes, we recommend that you follow our template below.
As an example we will run through how our AssigmentTest class works and the steps taken to write it. We will be testing the Assignment class's getDescription() function to see if it returns the correct String.
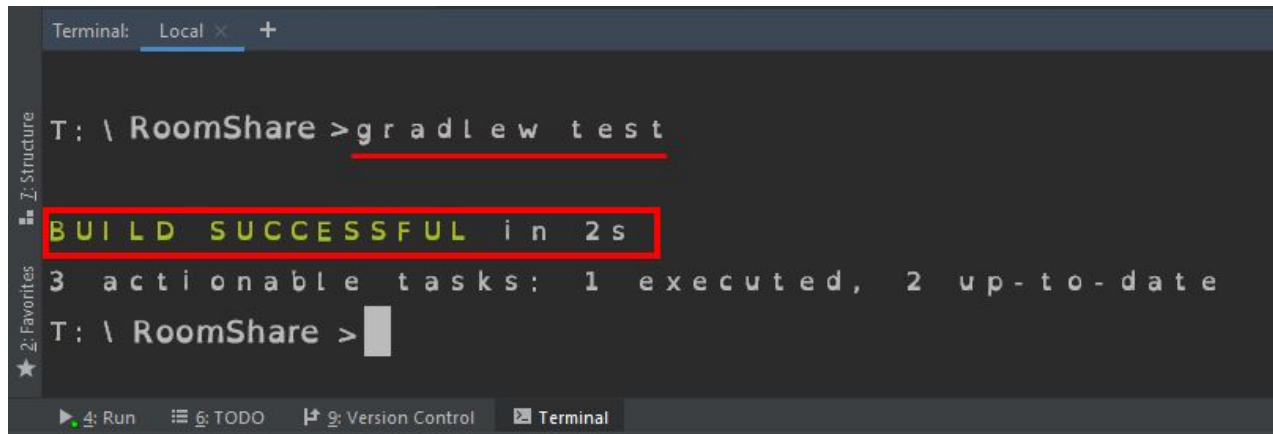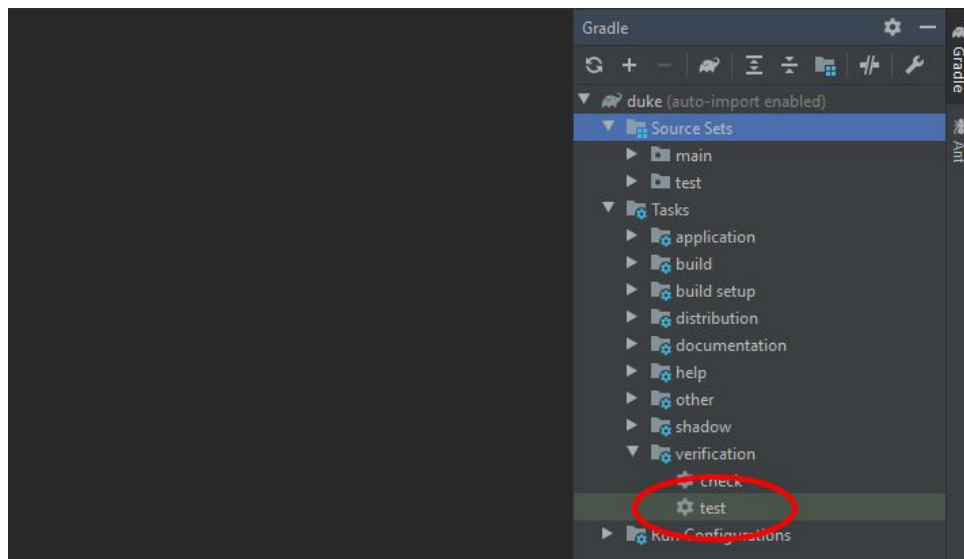
Steps:
1. Firstly in our includes portion, make sure to include the "AssertionEquals", "AssertFalse" and Test packages in order to use Test functions.

```
1    import Enums.Priority;
2    import Enums.RecurrenceScheduleType;
3    import Model_Classes.Assignment;
4    import org.junit.jupiter.api.Test;
5
6    import java.text.ParseException;
7    import java.text.SimpleDateFormat;
8    import java.util.Date;
9
10   import static org.junit.jupiter.api.Assertions.assertEquals;
11   import static org.junit.jupiter.api.Assertions.assertFalse;
```

Fig 5.6 packages required

2. Now we will create the class and initialise some variables. Since we are checking the description of an assignment object, we need to create one first.

```
public class AssignmentTest {
    private static SimpleDateFormat format = new SimpleDateFormat( pattern: "dd/MM/yyyy HH:mm");
    private static Date date;

    static {
        try {
            date = format.parse( source: "22/12/2019 18:00");
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }

    private Assignment assignment = new Assignment( description: "description", date);
```

Fig 5.7 initialising Assignment object

3. Next we will create the test function. We put a @Test on top of the function to indicate to IntelliJ that this is a test function. Here we are testing that the description output by the getDescription() function is "description". If the test function is recognised by the IntelliJ it will show a green triangle at the left hand side of the function.

Fig 5.8 Test function

Appendix A

1.1 User Stories

Here we have a list of feature/requirements that the users we interviewed suggested our product have. We organised the suggestions into 3 groups based on how important we felt they were or from the frequency of the same suggestion.

Must Do:
- As a user, I would like to have an authentication system so that I can just track tasks that are specific to me and have privacy for my tasks
- As a user, I would like to have a way to broadcast certain tasks to all users of the task tracker
- As a user, I would like to prioritise my tasks so that I know which are the important tasks I need to handle immediately
- As a user, I would like a way to have a timer and alarm that I can set so that I can be alerted of tasks without having to keep track of RoomShare all the time
- As a user, I would like a way to assign tasks to other users who are sharing the system, so that I can tell other users to do certain tasks without leaving the system.

Good to Do:
- As a user, I would like to have sub tasks under my main task so that my list looks more readable. Example buying groceries
- As a user, I would like to group my tasks according to their nature, so that it is easier to visualise
- As a user, I would like to have ranged based operations like deleting based on range, so that I can easily organise my task list
- As a user, I would like to have a temporary storage for my deleted tasks, so that I can reverse my mistakes in deleting certain tasks.

Not useful:
- As a user, I would like to have a chat system in RoomShare, so I can talk to my flatmates in the system
- As a user, I would like to be able to make a poll for my roommates to decide on things such as what to eat for dinner, so that we can come to decisions quickly
- As a user, I would like to be able to see who is at home from the program

In this section we document the different use cases RoomShare is currently at. This includes how the program process different user inputs and the output it will give.

System: RoomShare
Actor: User
Use case: CheckAnomaly
1. User creates an event type task
2. RoomShare checks if the event has any clashes with the current task list

Extension:
2.a: Event clashes with another other task
      2a.1: RoomShare tells user that there is a clash in timings, and doesn't add the task
      2a.2: RoomShare returns to normal program flow
2.b: Event does not clash with another task
      2b.1: RoomShare adds the task into the task list
      2b.2: RoomShare returns to normal program flow

System: RoomShare
Actor: User
Use case: Snooze
1. User enters 'snooze *n*', where n is the index of the task the user wants to snooze
2. RoomShare asks the user how long they wish to snooze the task for
3. User inputs the amount of time
4. RoomShare asks the user what time unit they wish to snooze the task
5. User inputs the time unit
6. RoomShare snoozes the task for that specified time unit and amount

System RoomShare
Actor: User
Use case: fixed duration
1. User enters an Event type task
2. RoomShare asks if the event has a fixed duration
3. User enters 'yes' or 'no'

Extension:
3a: User enters 'yes'
      3a.1: RoomShare asks for the duration, and show an example of a valid input
      3a.2: User inputs the duration
      3a.3: RoomShare creates an event with a fixed duration, and adds it to the task list
      3a.4: RoomShare returns to main program flow

3b: User enters 'no'
      3b.1: RoomShare creates an Event type task and adds it to the task list
      3b.2: RoomShare returns to main program flow

System: RoomShare
Actor: User
Use case: recurring tasks
1. User inputs 'recur'
2. RoomShare enters recurring mode, and shows the user the list of possible commands
3. User enters either 'add', 'find x' (where x is a keyword to search for), 'list' or 'exit'

Extension:
3.a User enters 'add'
    3a.1: RoomShare prompts to choose a recurrence schedule ('day', 'week' or 'month')
    3a.2: User enters one of the recurrence schedules
    3a.3: RoomShare prompts user to enter a task as per normal, and shows some
examples
    3a.4: User enters the task description
    3a.5: RoomShare creates a recurring task of the type, and adds it into the task list
    3a.6: RoomShare returns to main flow of recurring mode

3.b User enters 'find x'
    3b.1: RoomShare iterates through the task list for recurring tasks that contains the
keyword x
    3b.2: RoomShare prints out all recurring tasks that contain the keyword
    3b.3: RoomShare returns to main flow of recurring mode

3.c User enters 'list'
    3c.1: RoomShare prints out all the recurring tasks
    3c.2: RoomShare returns to main flow of recurring mode

3.d User enters 'exit'
    3.d.1 RoomShare returns to main program flow, and tells user that they have exited
recurring
        Mode.
3* User enters invalid command
- RoomShare tells user that they have input a wrong command and prompts the correct
  commands

This portion explains future plans for the project such as features we have yet to implement as well as possible changes to program architecture.

System: RoomShare
Actor: User
Use case: Logging In
1. RoomShare prompts the user to enter their username
2. User inputs their username
3. RoomShare prompts the user to enter their password
4. User inputs their password
5. Repeat step 1 to 4 until user gets both username and password correct
6. RoomShare greets the user and user gets to perform actions on their task list

Extension:
1.a User can choose to quit by entering "quit" end RoomShare's processes

Use case End

System: RoomShare
Actor: User
Use case: Broadcast
1. User enters "broadcast" when in their account
2. RoomShare prompts for an action to add or delete to be performed on the broadcast task list, or to cancel the process.
3. RoomShare performs the process
4. RoomShare then tells the user that the process has been completed
5. RoomShare returns to normal program flow

Extension:
2.a User wishes to cancel the process
        2a.1 User inputs cancel instead of a task description
        2a.2 RoomShare cancels the process notifies the user

2.b User wishes to add a task
        2b.1 User inputs add
        2b.2 RoomShare prompts to add a task
        2b.3 User inputs the information to create a task
        2b.4 RoomShare creates the task and adds it to the broadcast list
        2b.5 RoomShare notifies the user the task is added to the user

2.c User wishes to delete a task

2c.1 User inputs delete

2c.2 RoomShare lists out the tasks and prompts for an index to delete

2c.3 User inputs the index to delete

2c.4 RoomShare deletes the specified task, then notifies the user the task is deleted

Use case End

System: RoomShare

Actor: User

Use case:Prioritising Tasks

1. User enters "priority *n*", where n is the task index
2. RoomShare retrieves the task and prints out the task description along with it current set priority
3. RoomShare prompts the user if they would like to set its priority
4. User enters "yes" or "no"
5. RoomShare completes the appropriate action, and returns to normal program flow

Extension:

3.a User inputs "yes"

      3a.1: RoomShare shows the available tasks priority levels (low, medium, high, utmost urgency)

      3a.2: RoomShare prompts the user to indicate which level of priority they would like to set

      3a.3: User inputs the level of priority

      3b.4: RoomShare updates the task's priority level, sorts the task list based on priority

      3b.5: RoomShare tells the user the task's priority has been updated

3.b User inputs "no"

      3a.1: RoomShare tells the user that the priority will not be updated.

Use case End

System: RoomShare

Actor: User

Use case: Alarm system

1. User performs the creation of a task
2. RoomShare prompts the user if they would like to be alerted of any updates of the task
3. User inputs their choice of alert
4. RoomShare stores the choice and alerts if the task has been updated

Extension:

3.a: User inputs "yes"

   3a.1: RoomShare adds an alarm to the task

   3a.2: RoomShare adds the task into the task list

3.b: User inputs "no"

   3b.1: RoomShare adds the task to the task list

<u>System: RoomShare</u>

<u>Actor: User</u>

<u>Use case: Task Assigning</u>

1. User inputs "assign" when RoomShare is waiting for a command
2. RoomShare prompts user to specify which username to assign a task to
3. User inputs the specified person's username
4. RoomShare prompts to enter a task
5. User inputs a task into RoomShare
6. RoomShare adds the task to the specified user's task list
7. RoomShare tells the user that the task has been assigned to the user, and returns back to normal program flow

Extension:

*1. User inputs "cancel"

   *1.1: RoomShare cancels the assignment process

   *1.2: RoomShare returns to normal program flow

<u>1.4 Non-functional requirements:</u>

RoomShare should tell the user what they can input, to reduce the number of wrong inputs

RoomShare should have a relatively fast response time, of no more than 5 seconds

RoomShare should attempt to reduce the memory required to store information

The interface should be intuitive to first time users

The system should be run on Java 11.

The project is not required to support online databases