Chronologer Developer's Guide

1) Target User Profile

The target user profile of this project are students from the National University of Singapore (NUS). We aim to create a user friendly task manager with a Graphical User Interface (GUI) that aids students like ourselves to have all their taskings under one application.

We aim to **enhance Duke (Optimize - option)** to be a tool that would eventually be used by future students to ensure that they are always on top of the taskings they have.

2) Value Proposition

Our value proposition is that we intend to fix the currently convoluted and rather unclear management of data and information that us students from NUS face by allowing a clear, concise and visual task manager.

3) Design and Architecture

3.1 Architecture

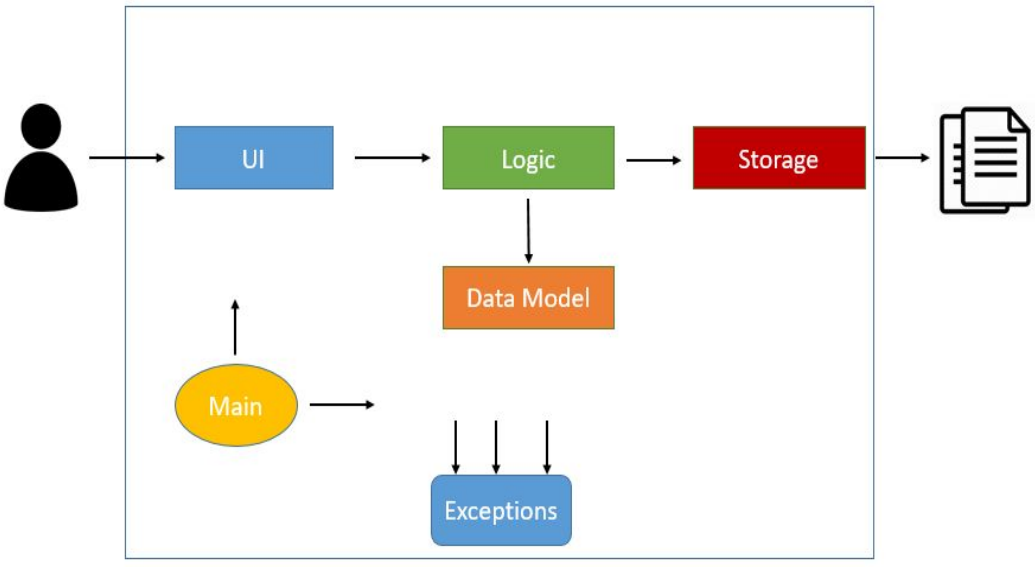


Figure 1. Architecture Diagram

The Architecture Diagram given above explains the high-level design of the App. Given below is a quick overview of each component.

Main consists of two classes called **ChronologerMain** and **ChronologerLauncher**. It is responsible for initializing the components required for the application.

The rest of the app consists of five components:

- **UI:** The UI components of the app
- **Logic:** The command processor and executor.
- **Data model:** Holds the data of the application.
- **Storage:** Save and loads the data to hard disk
- **Exceptions:** Collection of classes used to handle possible errors.

3.2 UI Ui.java

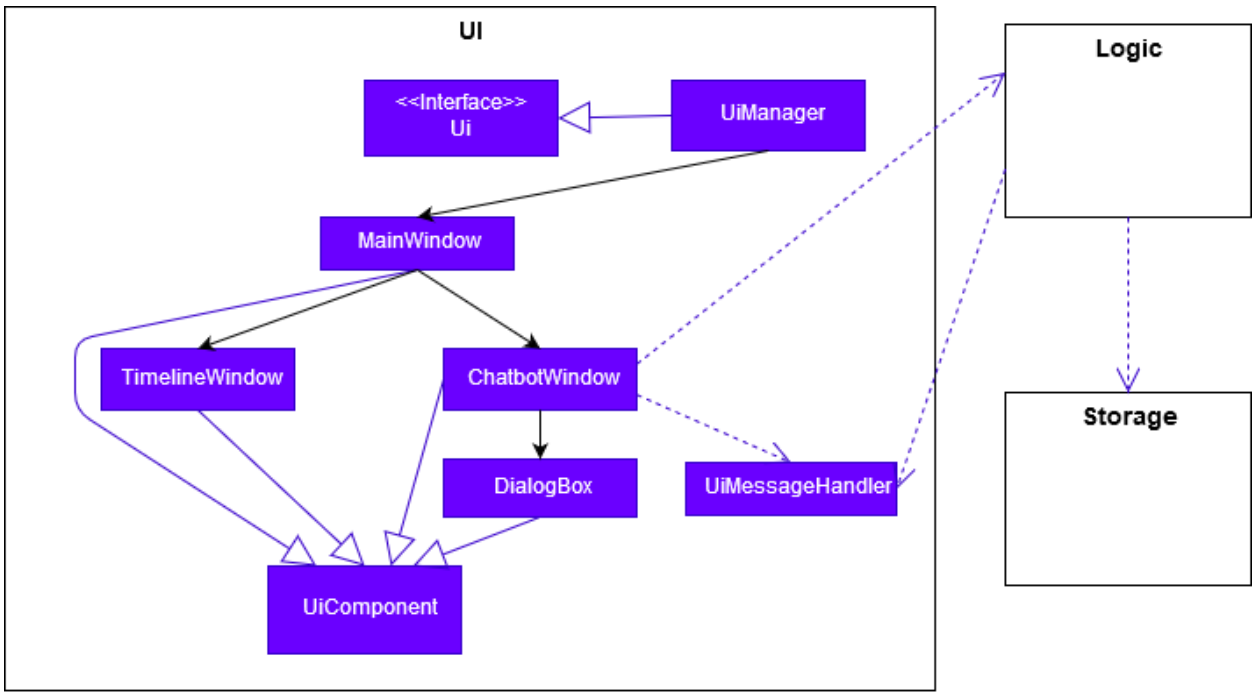


Figure 2. Structure of the UI Component.

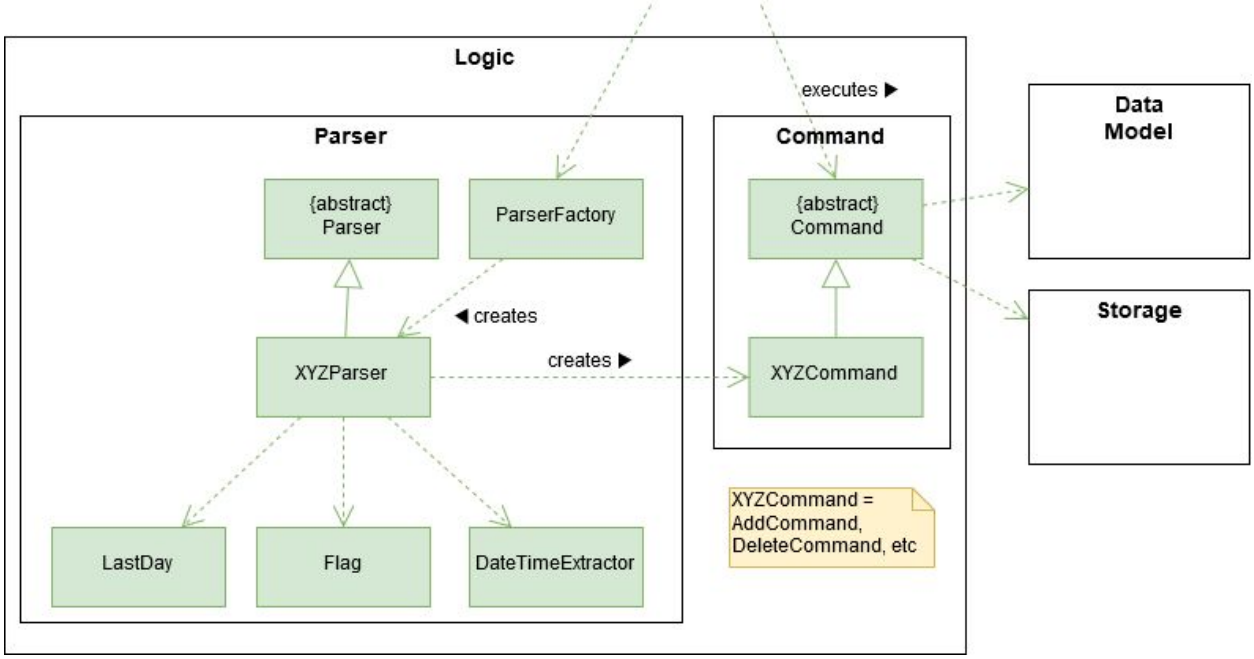
This class diagram of the UI above shows the relationships between the different classes in the UI.

The UI components are rendered via the JavaFX UI framework. The actual position, layout and properties are inscribed in the corresponding .fxml files under /resources/view.

The UI components such as the ChatbotWindow extend from a parent class called the UiComponent which defines the contract necessary to create a portion of the UI. Moreover this simplifies the process of building up the UI. This Ui interface is implemented by the UiManager which is the manager of all the UiComponents.

The entirety of the UI is built upon the MainWindow which will house the components such as the TimelineWindow. Hence the UiManager which holds a reference to the MainWindow will essentially create the UI. The features of the UI are predominantly view-based commands where a user can alter the state of the timeline by keying in various commands. Moreover, the UI also has background features such as the sieving and presenting of a particular day's priority tasks with just the time of the tasks, the highlight box to show the user the current day and date. (It also changes colours for the weekend).

3.3 Logic



Within Logic, the Parser class is utilised to interpret the incoming input from the UI and returns the Command class for execution of the user's instruction, interaction with the Data Model if needed and returning the result to the UI.

3.3.1 Parser

Parser model: Parser.java

The **Parser class**:

- Receives the user input from the **UI** component and extract the necessary components to create a **Command** component.
- Depending on the user input, either an object instance of abstract classes **IndexParser** or **DescriptionParser** is created to parse the user input.
- In addition, the **Parser** objects can also instruct the **UI** to perform certain actions such as displaying error messages.

3.3.2 Command

Command model: Command.java

The **Command class**:

- Stores a **Command** object that represents what Chronologer needs to execute.
- **Command** class is the abstract parent class of the different command sub-classes such as **addCommand, editCommand, searchCommand, etc.**

- Each **Command** object has various properties that will affect how it will execute and what it will create/modify the **Data Model** component.
- The result of the **Command** execution will be passed back to the **Ui** component where it will result in a message to the user or trigger a further action by the **Ui**.

Below is the Sequence Diagram for interactions of the Parser and Command with the Data Model when handleUserInput() is called for “deadline foo /by 05/11/2019 1500” input.

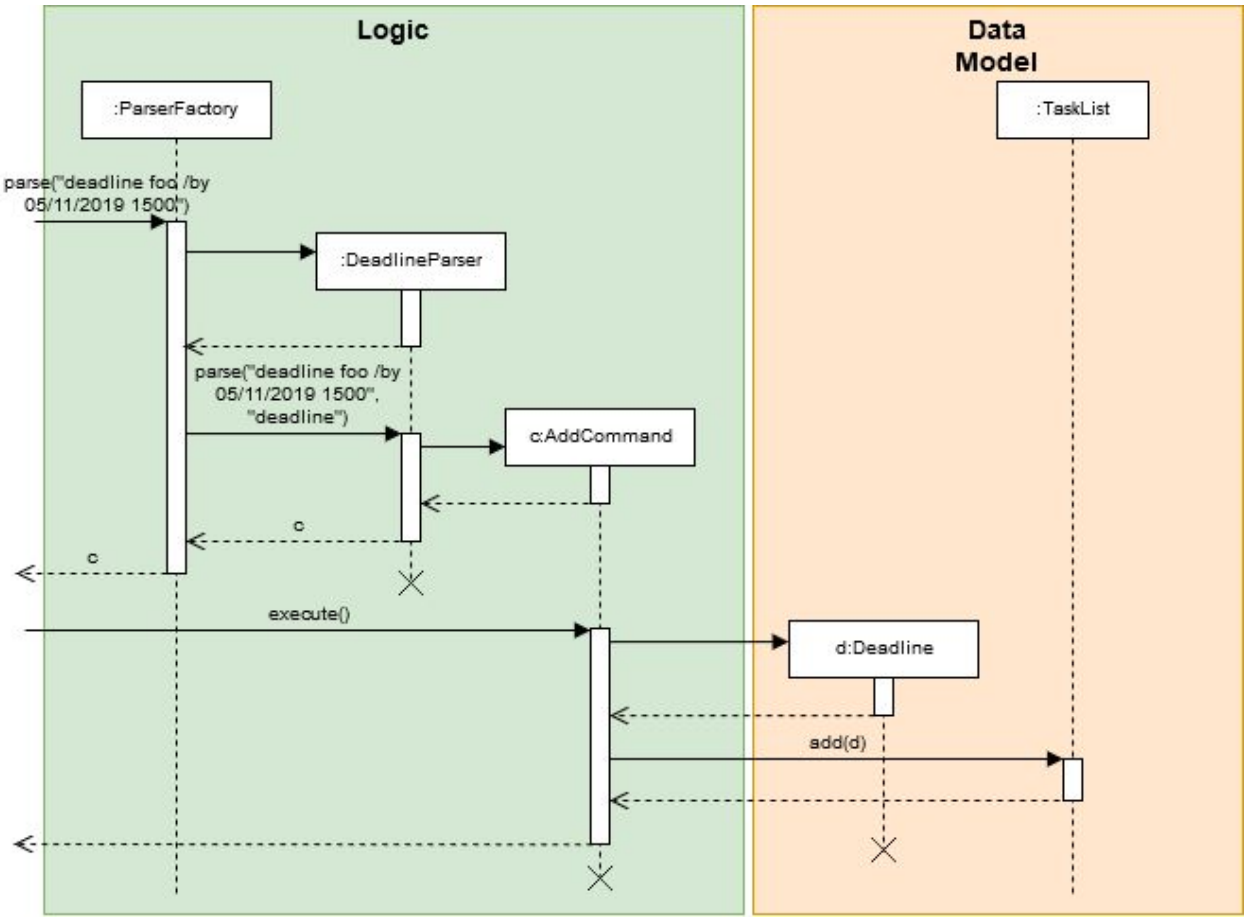


Figure 4. Interactions inside the Logic Component for the “deadline” command

3.4 Data Model

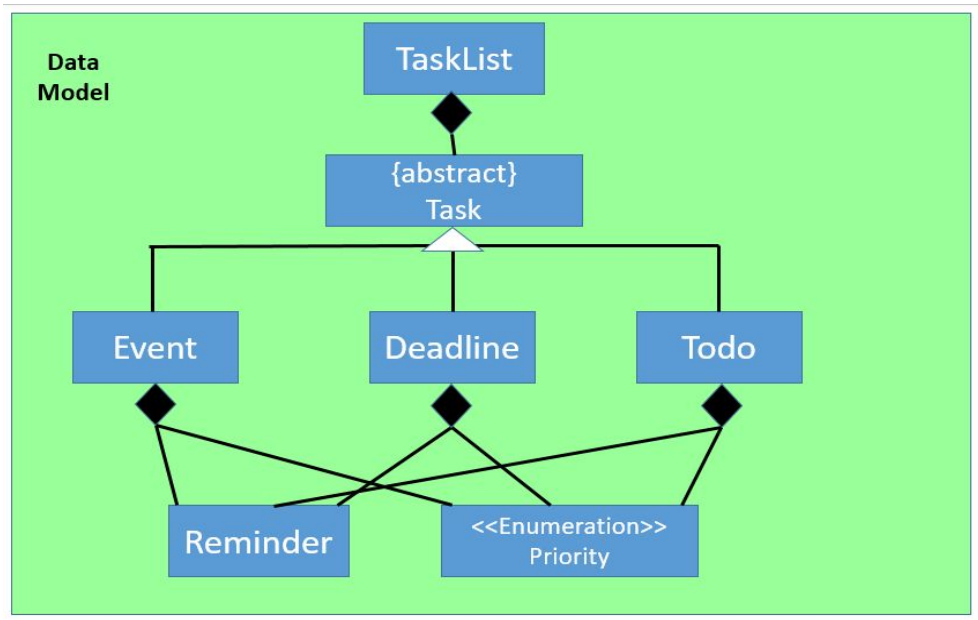


Figure 5. Structure of the Model Component.

Task model: Task.java

The **Data Model**:

- Stores a **Task** object that represents tasks in a user's daily life.
- **Task** object stored in a **TaskList** class.
- **TaskList** class handles all operations performed on the current list of tasks.
- **Task** class is the abstract parent class of **Deadline,Event** and **Todo.**
- Each **Task** have various properties that will be affected by the **Logic** component.
- Some properties eg: **Priority**, **Reminder** are their own class as they aren't primitive data types.

3.5 Storage

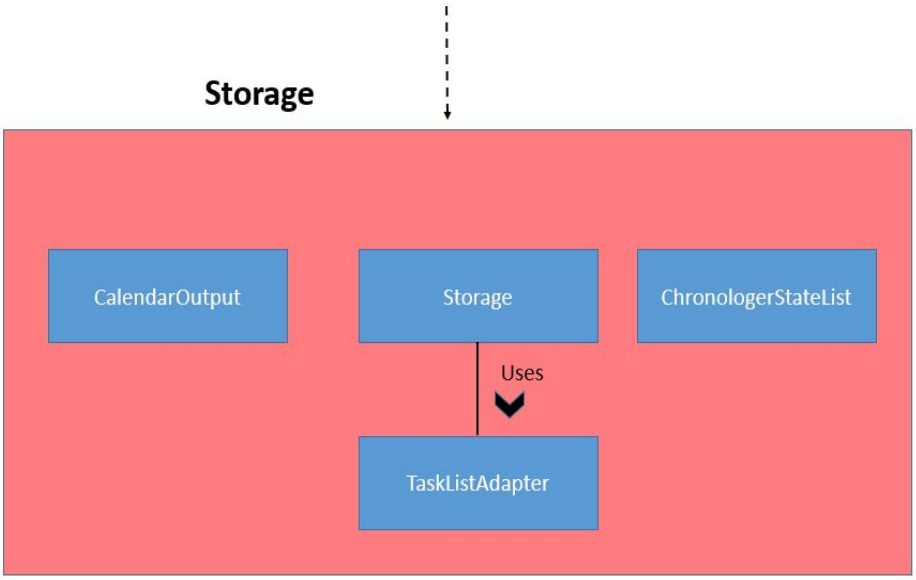


Figure 6. Structure of the Storage Component.

The **Storage Component**:

- Stores the tasklist in Json format and read it back.

- Save Calendar objects into ICS files.
- Maintains records of the tasklist for undo,redo and backup purposes.

3.6 Exceptions component

This component package consists of two classes, **ChronologerException** and **MyLogger**. **MyLogger** is used to write log messages to a file for easier debugging while **ChronologerException** is used to handle all of the expected and some unexpected exceptions to give users a better understanding of why the program crashes. Details of this package can be found here.

4) Implementation

This section describes some noteworthy details on how certain features are implemented.

4.1 Export timeline as ics file feature (Tan Yi Xiang)

4.1.1 Implementation

This feature utilizes iCal4j, a Java library used to read and write iCalendar data streams. This feature mechanism is facilitated by two classes, [ExportCommand] and [CalendarOutput]. [ExportCommand] is an element of the Command model in the logic component and extends from abstract superclass [Command].It also stores a Calendar object as [calendar]. It receives the file name and flags if any from the user input and use them to build an ics calendar according to Chronologer's [TaskList] which is a list of our data models.
Additionally, it implements the following major operations:

- **execute()**: Parse and build a calendar file based on the tasks in Chronologer's timeline.
- **initializeCalendar():** Initialize calendar with predefined settings.
- **extractDeadline():** Convert deadline tasks to calendar components.
- **extractEvent():** Convert deadline tasks to calendar components.
- **extractTodo():** Convert todo with period tasks to calendar components.

[CalendarOutput] is an element of the Storage component. It is a static class that receives the [calendar] built by [ExportCommand] and process it to an ics file. It will then store it to Chronologer's database directory.It only has one major operation:

- **outputCalendar():** Process the calendar into an ics file.

Given below is an example usage scenario and how the export feature behaves.

[Assume that the user has already added a number of tasks in Chronologer]
Step 1: User types in [**export MyCalendar -d**] command to only export deadline tasks to an ics file called MyCalendar.The [ExportCommand] will initialize [calendar] with some predefined properties eg:

```
Calendar calendar = new Calendar();
calendar.getProperties().add(new ProdId("-//Ben Fortuna//iCal4j 1.0//EN"));
calendar.getProperties().add(Version.VERSION_2_0);
calendar.getProperties().add(CalScale.GREGORIAN);
```

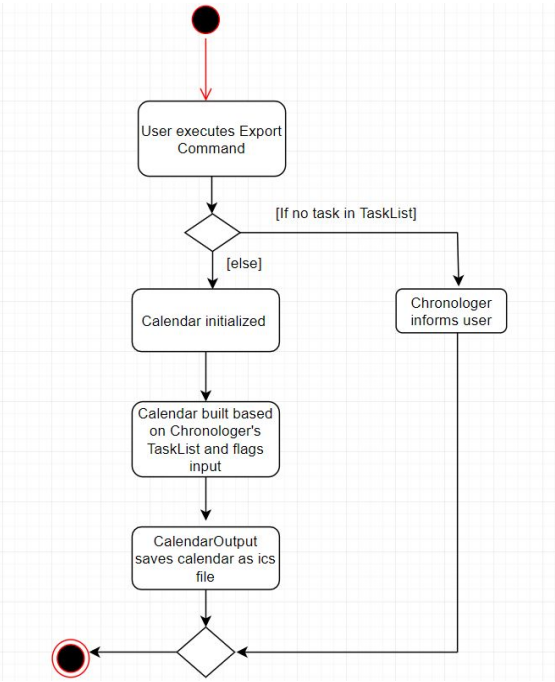
Step 2: The [ExportCommand] will then call **execute()** which is the main function to start the export functionality by calling one or more of the extract methods eg: **extractDeadline(), extractEvent()** etc.

Step 3: [ExportCommand] receives the flag value **-d.** Based on that flag, **execute()** will in turn call **extractDeadline()** to convert all deadline tasks in [TaskList] to calendar components.If no flags were found, [ExportCommand] would have process all tasks with dates in [TaskList].

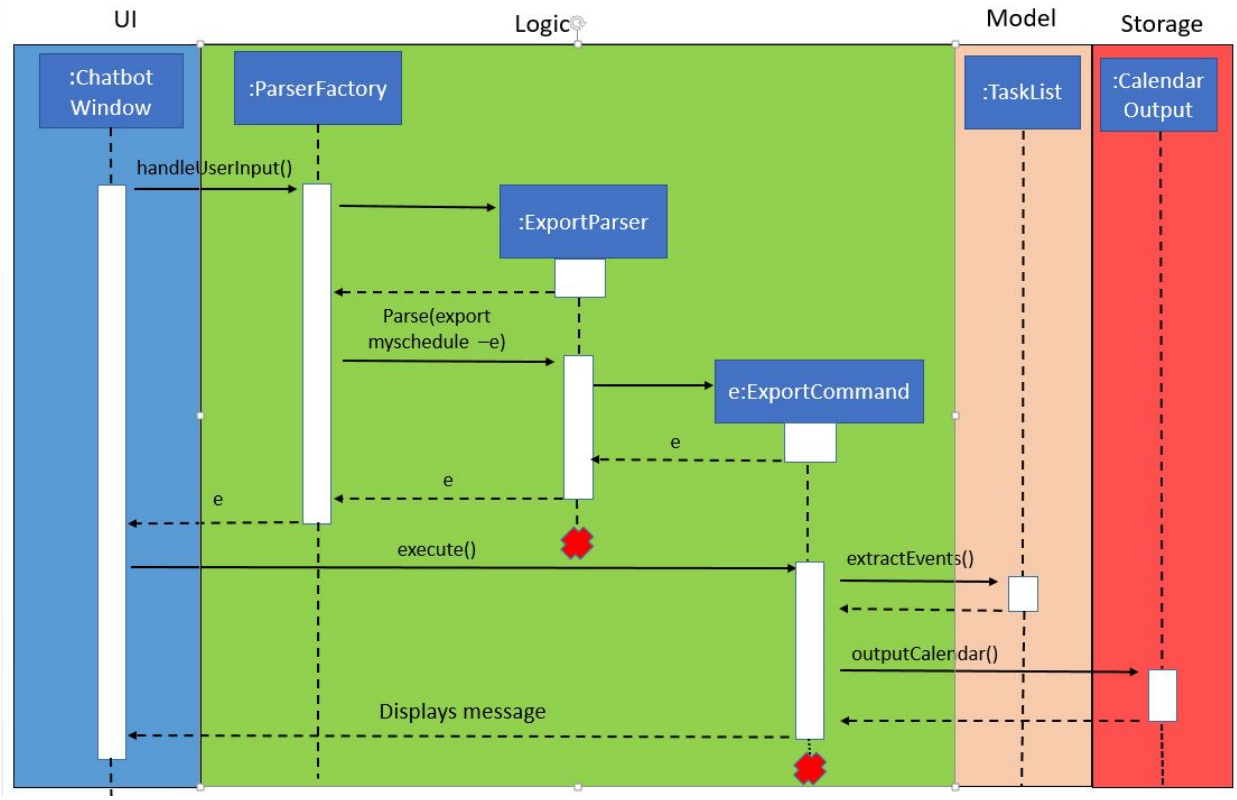
Step 4: **extractDeadline()** will iterate through the [TaskList] and check if each task is a deadline type task. If it is, **extractDeadline()** will convert the task details into valid iCal4j calendar properties.

Step 5: Once the iteration is done, **execute()** will pass the built [calendar] to [CalendarOutput] **outputCalendar()** method which will in turn process and save the calendar object as an Ics file called **MyCalendar.ics** in ChronologerDatabase directory . The file can then be opened by any supporting application.

The following activity diagram summarizes what happens when a user executes **export** command.



The following sequence diagram shows how the **export** operation works if a user decides to export event tasks.



4.2 History feature (Sai Ganesh Suresh)

4.2.1 Undo/Redo

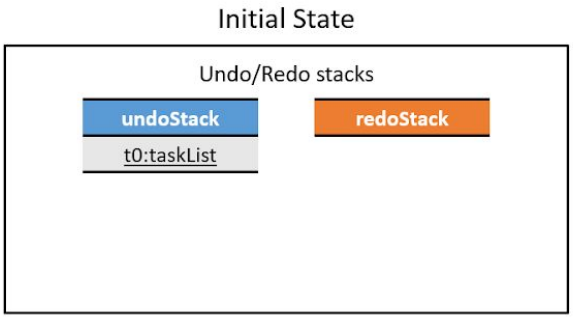
The undo/redo mechanism is facilitated by ChronologerStateList. It will be part of the History feature which will hold the undo/redo history, stored internally as two stacks: chronologerUndoStack and the chronologerUndoStack. Additionally, it implements the following operations:

- #undo() - Restores the core taskList to the previous state.
- #redo() - Restores the core taskList to the previous state.

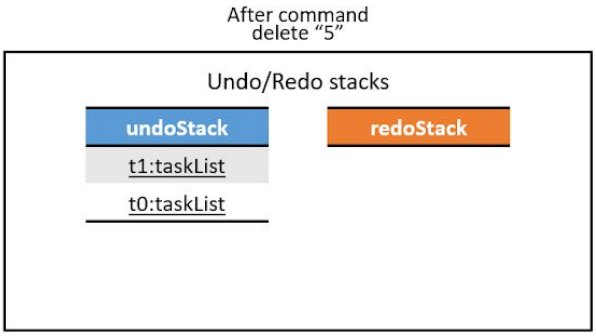
These operations are shown in the ChronologerStateList as ChronologerStateList#undo() and ChronologerStateList#redo() respectively.

Given below is an example usage scenario and how the undo/redo mechanism behaves at each step.

Step 1. The user launches the application for the first time. The chronologerUndoStack will be initialized with the initial tasklist upon load (if this is the very first launch, an empty tasklist becomes this base state).

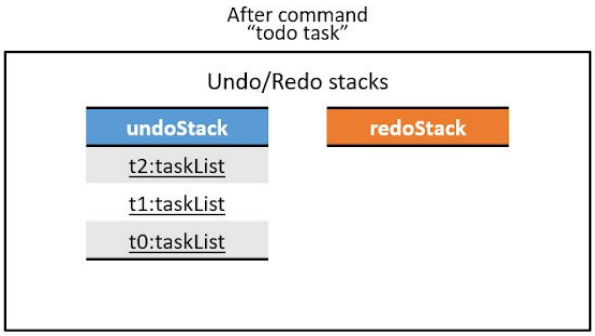


Step 2. The user executes delete 5 command to delete the 5th task in the tasklist which contains all the tasks in chronologer. The delete command calls ChronologerStateList#addState (), causing the modified taskList of the chronologer after the delete 5 command executes to be saved in the undoStack.



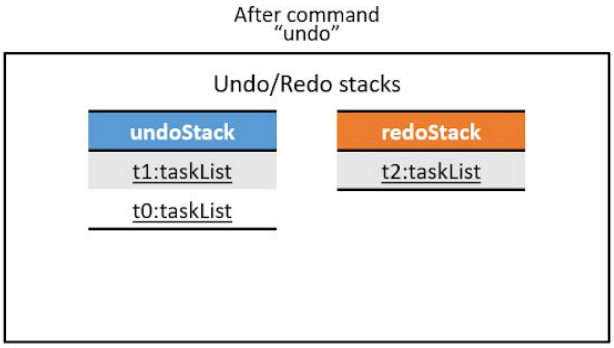
Step 3. The user now opts to add a task by performing “todo task”. The add command also calls ChronologerStateList#addState (), causing another modified taskList to be saved into the chronologerStateList.

i If a command fails its execution, it will not call ChronologerStateList#addState (), so the current taskList state will not be saved into the chronologerStateList.

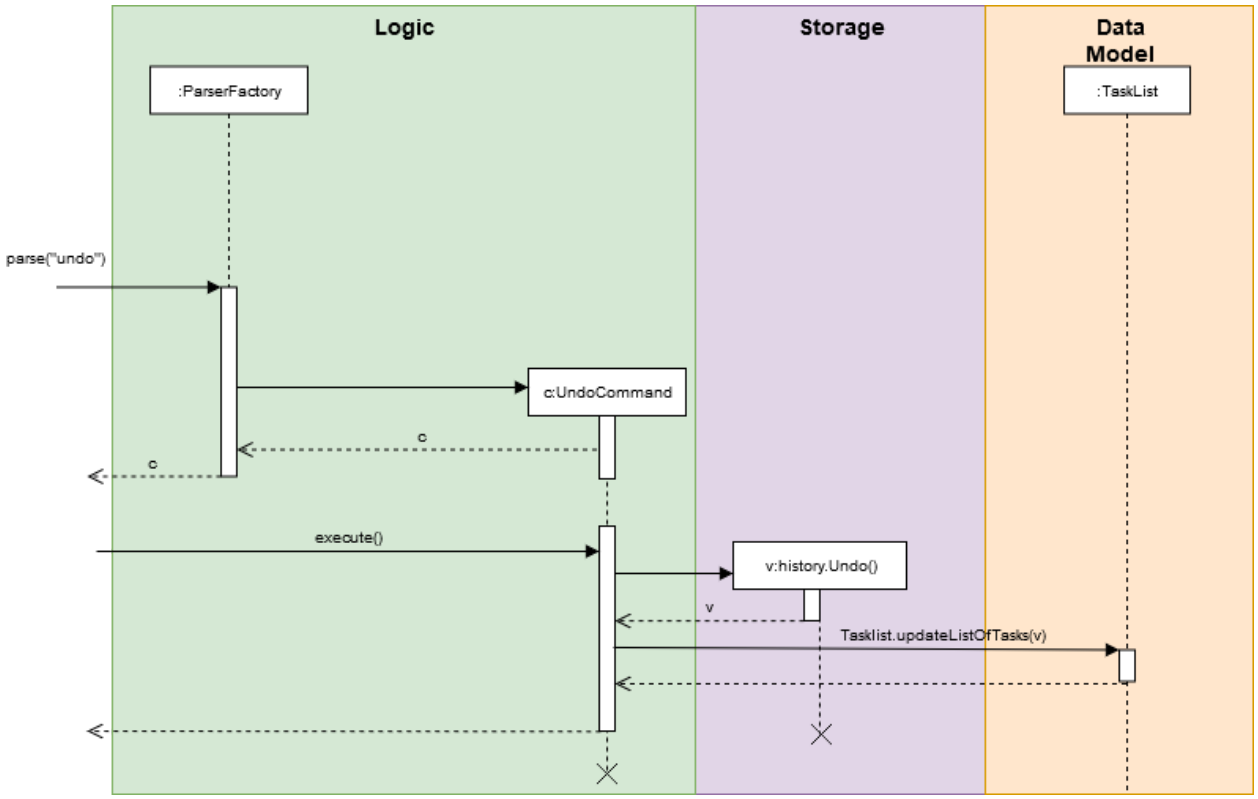


Step 4. The user now decides that adding the this task was a mistake, and decides to undo that action by executing the undo command. The undo command will call ChronologerStateList#undo (), which will pop the current head of the list t2 and push it into the redoStack while the head of the undoStack which is t1 will be used to populate the core taskList.

i If the size of the current undoStack is 1, then the stack only contains the initial taskList t0, then there are no previous taskLists to restore from. The undo command uses ChronologerStateList#checkUndoSizeInvalid() to check if this is the case. If so, it will return an error message to the user rather than attempting to perform the undo.



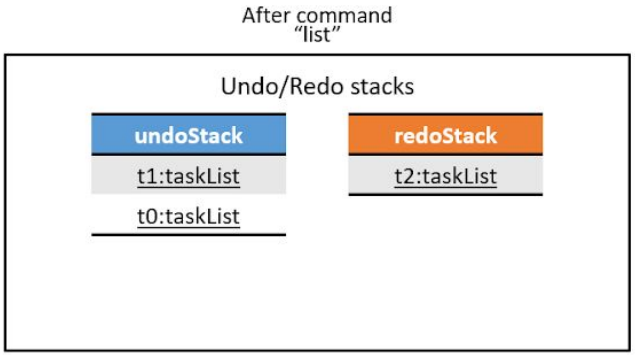
The following sequence diagram shows how the undo operation works:



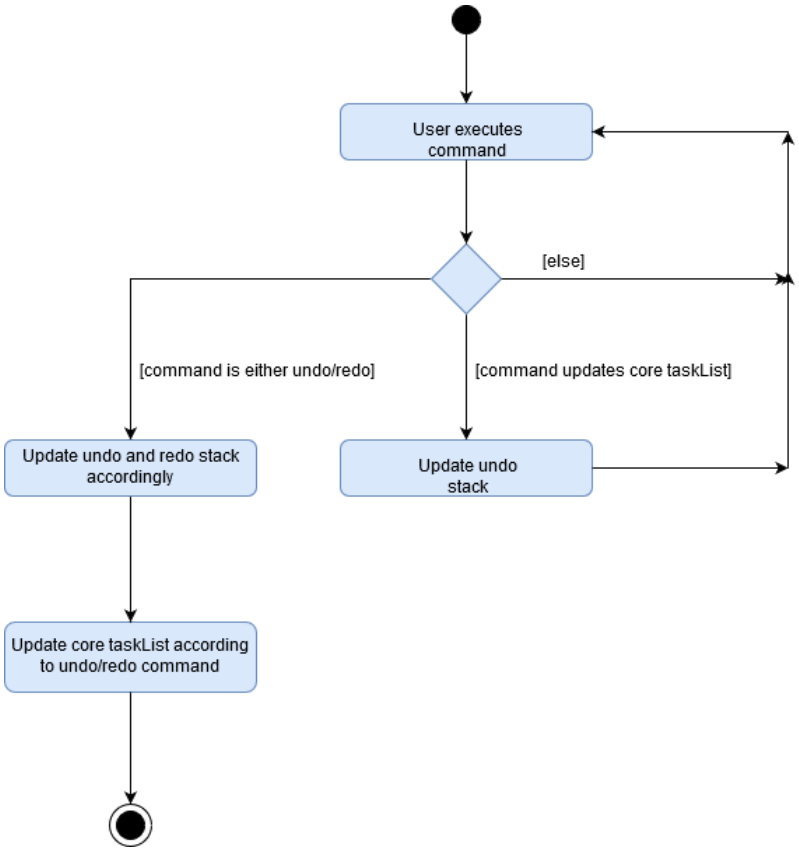
The redo command does the opposite — it calls ChronologerStateList#redo (), which will pop the current head of the redoStack t2 which will be used to populate the core taskList and also pushed into the undoStack.

i If the redoStack size is 0, it means the current version of the taskList is the latest taskList, then there are no undone taskLists to restore from. The redo command uses ChronologerStateList#checkRedoSizeInvalid() to check if this is the case. If so, it will return an error message to the user rather than attempting to perform the redo.

Step 5. The user then decides to execute the command “list”. Commands that do not modify the core taskList, such as list, will not call ChronologerStateList#undoChronologer() or ChronologerStateList#redoChronologer(). Thus, the undoStack and redoStack remain unchanged.



The following activity diagram summarizes what happens when a user executes a new command:



4.2.1 Design considerations

There are in general three methods in which undo/redo commands are usually performed in the applications we use daily. The three approaches are - Using Single Object Representing Change Approach, Using Command Pattern or lastly Using Memento Pattern. From these approaches, the Single Object approach seemed the best as our entire application depends on a centralised taskList or the core taskList.

Hence rather than having to worry about the accuracy of an inverse command which would have been the case for the command pattern, I decided to follow through with this method. Moreover in the proposed features, by limiting the number of actions that the user can perform on launch again allows memory usage to be controlled.

Moreover as a deep copy is needed to store these versions of the taskLists, it was essential to store objects and performing a deep copy using -

```
org.apache.commons.lang3.SerializationUtils
```

To perform a deep copy by

```
public static void addState(ArrayList<Task> listToStore) {
chronologerUndoStack.push(SerializationUtils.clone(listToStore));}
```

This is important as a copy functions in Java are predominantly shallow hence this again played into the decision to utilise the Single Object approach.

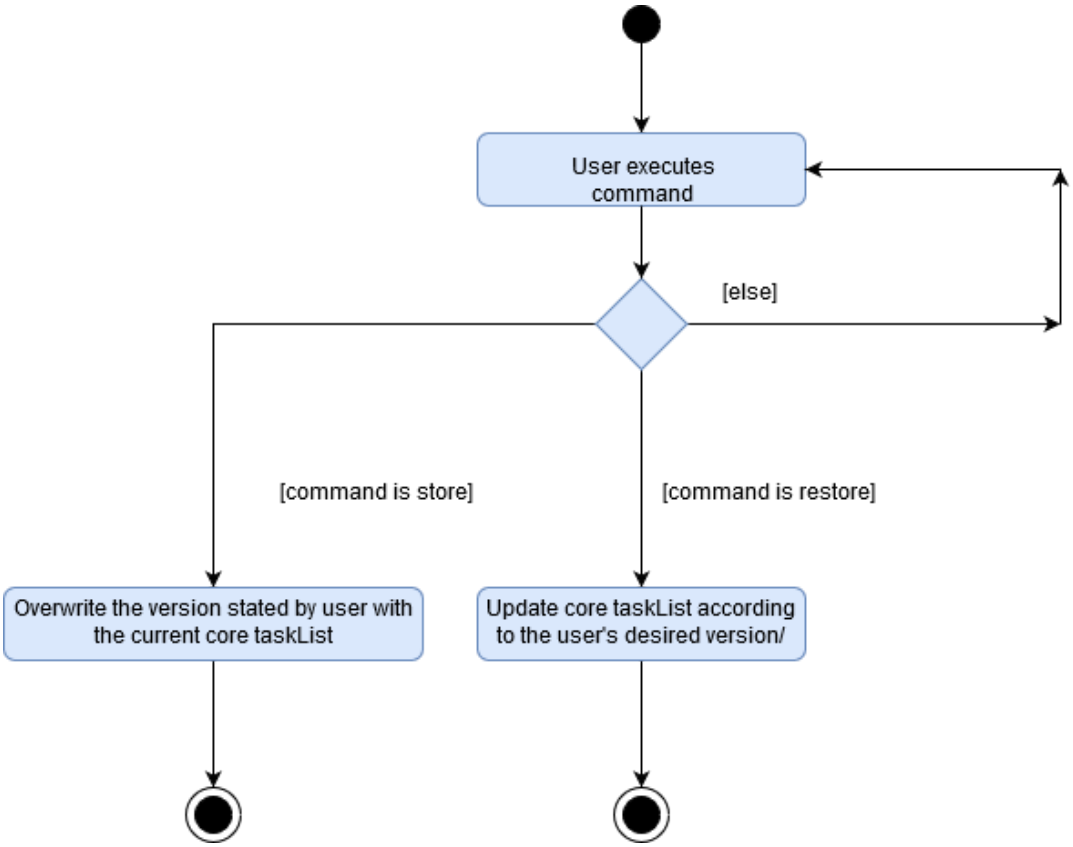
4.2.2 History features - States

This will allow the user to store a version of chronologer into persistent memory, where they can have three desired states stored. This will serve as a backup for the user as well as allowing the user to segregate his tasks however he likes. He can load from any of the three states and his current

This mechanism is facilitated by ChronologerStateList. There will be 3 files stored to persistent - Version1, Version2 and Version3. The user can then decide to store a particular state of their taskList using the #store and utilise #restore to obtain that list.

- #store() - Stores the core taskList as the version state.
- #restore() - Restores the core taskList from the pre-saved state.

The following activity diagram summarizes what happens when a user executes a new command:



4.2.3 History features - Command Line History

This feature was again designed to aid the user. By simply pressing the up arrow key and down arrow key, the user will be able to scroll through their past inputs to efficiently perform manipulations on the taskList.

- up - Move up your previous inputs till the very first.
- down - Move down your previous inputs till current.
-

4.3 [Proposed] School tasks Management feature (Hans Kurnia)

4.3.1 Proposed Implementation

The school tasks management feature facilitates the management of school related tasks that students will encounter during their time in school by modifying and integrating together existing classes.

It implements the following major operations:

- **AddCommand#execute()**:Adds a new task to the tasklist and checks for clashes
- **AddRecurringCommand():** Adds recurring tasks to the tasklist
- **parse():** parses user input and returns a command to be executed
- **extractModCode():** Extracts the module code from the user input

These tasks will include

- ***Lectures*** - Weekly recurring task with a start and end timing until the end of the current semester.

- ***Assignments*** - Tasks that have a due date.

- ***Tutorials*** - Weekly recurring task with a start and end timing until the end of the current semester.

- ***Examinations*** - Tasks that have a start and end timing. These tasks are high priority and are created as such.

Each of these tasks will have a module code associated with them at creation, this module code will be used for filtering and sorting these tasks accordingly in the future. Specifically for lectures and Tutorials, during creation they will be created as weekly recurring tasks set as the same “time-slot” of each week till the date specified by the user.

Below is an example usage scenario for Module management when it attempts to create a lecture and how the feature behaves.

Step 1.

The user launches the application for the first time. It is initialised with an empty schedule.

Step 2.

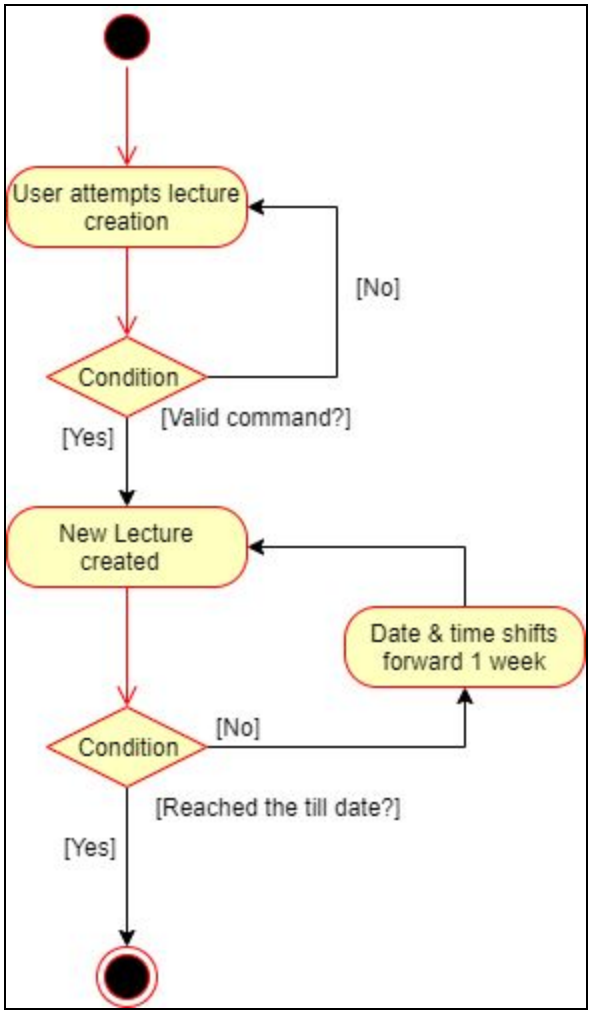
The user adds in a Lecture using the command ‘lecture /m cs2040c /at mondays 1000-1200 /till 01/01/2020”’ as a lecture that is scheduled every monday at 1000am to 1200pm till the 1st of january 2020.

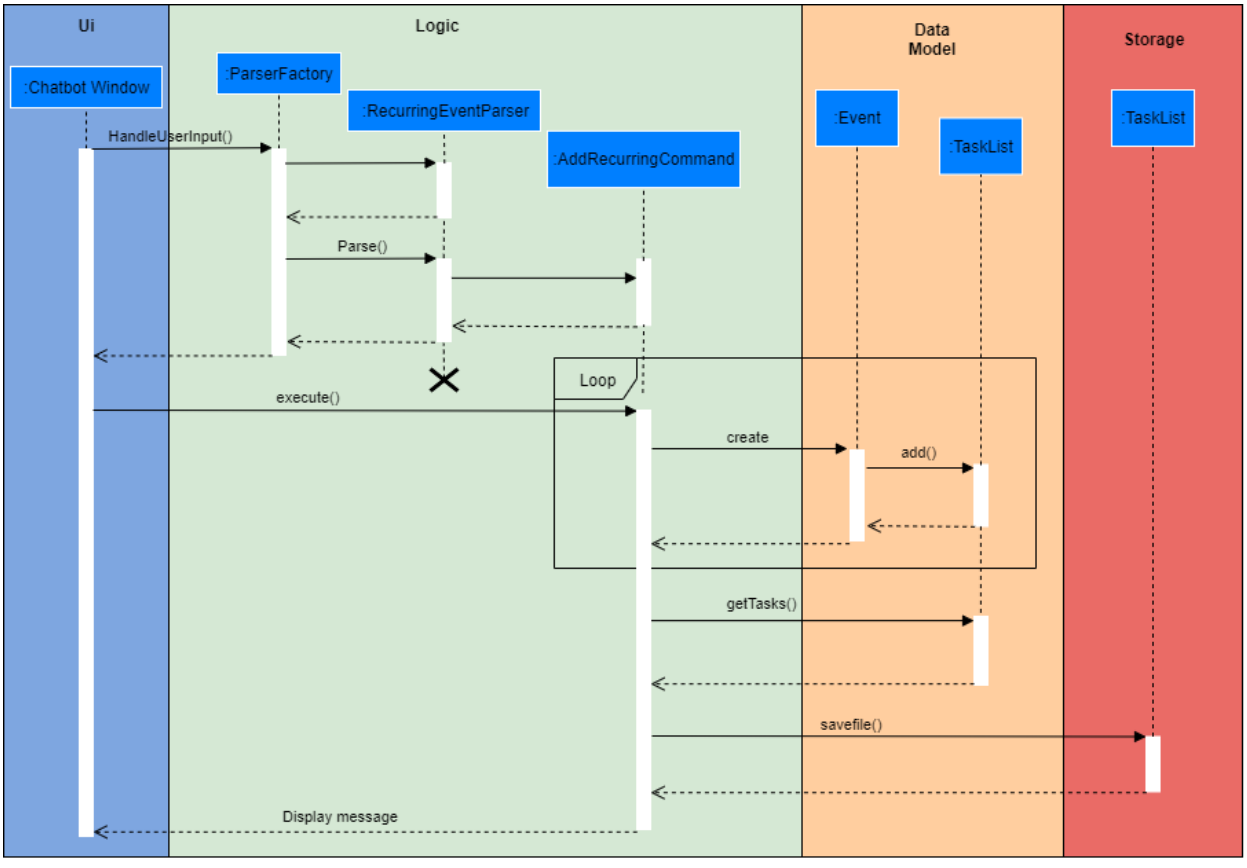
Step 3.

Chronologer will **parse()** the user input, calling **extractModcode()** to extract the given module code as CS2040C. After **parse()** is completed, it will return a **AddRecurringCommand** to be executed.

Step 4.

Chronologer will **execute()** this command which will add the new lecture tasks to the tasklist for every monday on 10am to 12pm till 01/01/2020.


User attempts lecture creation
[No]
Condition
[Valid command?]
[Yes]
New Lecture created
Date & time shifts forward 1 week
Condition
[No]
[Reached the till date?]
[Yes]


Ui
Logic
Data Model
Storage
:Chatbot Window
:ParserFactory
:RecurringEventParser
:AddRecurringCommand
:Event
:TaskList
:TaskList
HandleUserInput()
Parse()
Loop
execute()
create
add()
getTasks()
savefile()
Display message

This sequence diagram above shows how the **lecture** operation works if a user decides to add lectures.

4.3.2 Design Considerations

Aspect: How creation of lectures & tutorial executes

- Alternative 1 (current choice): Add to the list continuously till last week of semester
    - *Pros:*
        - Users will be able to modify individual lectures rather than as a set
        - Easy to implement.
    - *Cons:*
        - May have performance issues in terms of memory usage and overcrowding of index list (list command).
- Alternative 2: Add the next recurring task only when the current week's task has passed.
    - *Pros:*
        - Will use less memory and will clear up the index list (list command).
    - *Cons:*
        - View command to change timeline will not be able to see the lectures when we switch weeks.
        - Chronologer have to keep checking for time, making it resource expensive.

Aspect: Data/Logic structure for module management

- Alternative 1 (current choice): Modify/extend existing Data/Logic classes
    - Pros:
        - Minimises new classes created to handle module management.
        - Existing commands will still be usable for these tasks with minor modifications.
    - Cons:
        - Multiple method/constructor overloading to handle these task types.
- Alternative 2: Create new classes to handle these tasks
    - Pros:
        - Clear distinction in data/logic classes for these task types.
    - Cons:
        - Many new classes will have to be created with some having only minor changes to existing classes.
        - Existing commands will not work with these new classes, requiring major modifications to accommodate these task types.

4.4 Task Scheduling feature (Fauzan)

4.4.1 Implementation

The scheduling feature is facilitated by [TaskScheduler]. It checks the [TaskList] and search through all events to find free time slots that can accommodate either a user-inputted duration or a chosen task's duration.

It implements the following operations:

TaskScheduler#scheduleTask() – search for a free period of time that can fit the chosen task's duration value from now

TaskScheduler#scheduleByDeadline() – search for a free period of time that can fit the chosen task's duration value from now till the chosen deadline's date

TaskScheduler#scheduleWithinPeriod() - search within the chosen timespan period all uninterrupted period of times that can fit the chosen task

Below is an example usage scenario for TaskScheduler#scheduleByDeadline() and how the feature behaves.

Step 1.

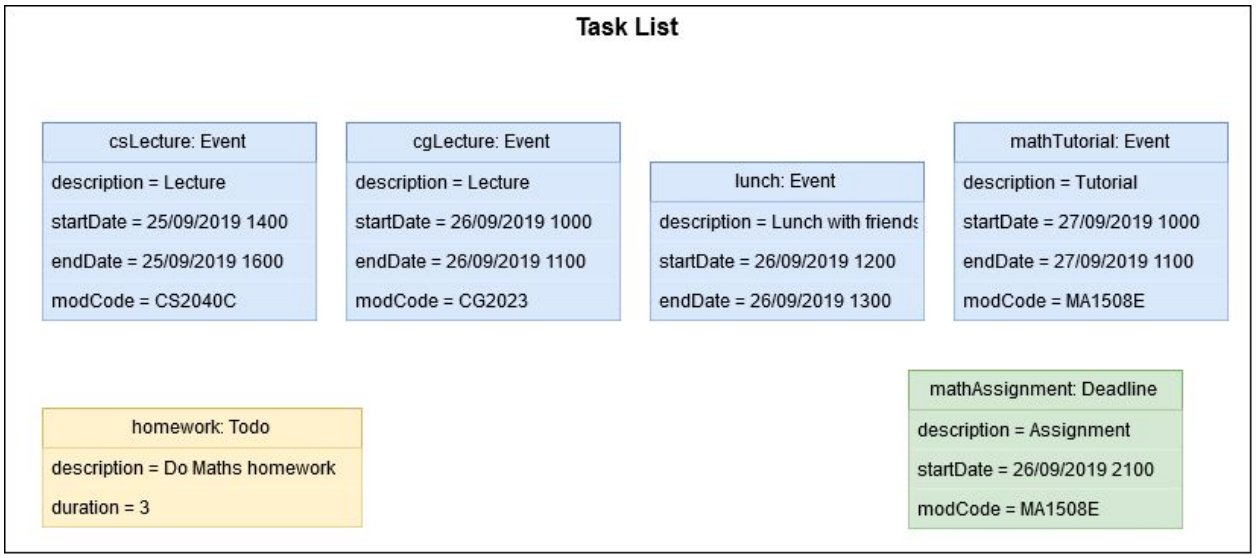
The user launches the application for the first time. It is initialised with an empty schedule.

Step 2.

The user populate his schedule with various event tasks. An Assignment task is also added in. The 1-indexed Task List now has 5 tasks with mathAssignment at index 5.

Step 3.

The user adds in a Todo task using the command 'todo Do Maths homework /for 3' as a planned time investment to settle an MA1508E Assignment with deadline 26/09/2019 2100. The added task is now located at index 6.



Step 4.

The user executes ‘schedule 6 /by 5’. This calls the ScheduleCommand to look up the task at index 6 of the TaskList and get its duration value, 3 hours. The command also retrieves the date from the assignment at index 5. This is used as a cut-off timing when finding a free time slot to accomplish the task. These parameters are then passed to TaskScheduler, which executes TaskScheduler#scheduleByDeadline().

Alternatively, a raw deadline date can be inputted instead (ex. ‘schedule 6 /by 26/09/2019 2100’) to be processed by TaskScheduler. A raw duration value can also be inputted instead (ex. “Schedule 3 -r /by 26/09/2019 2100”).

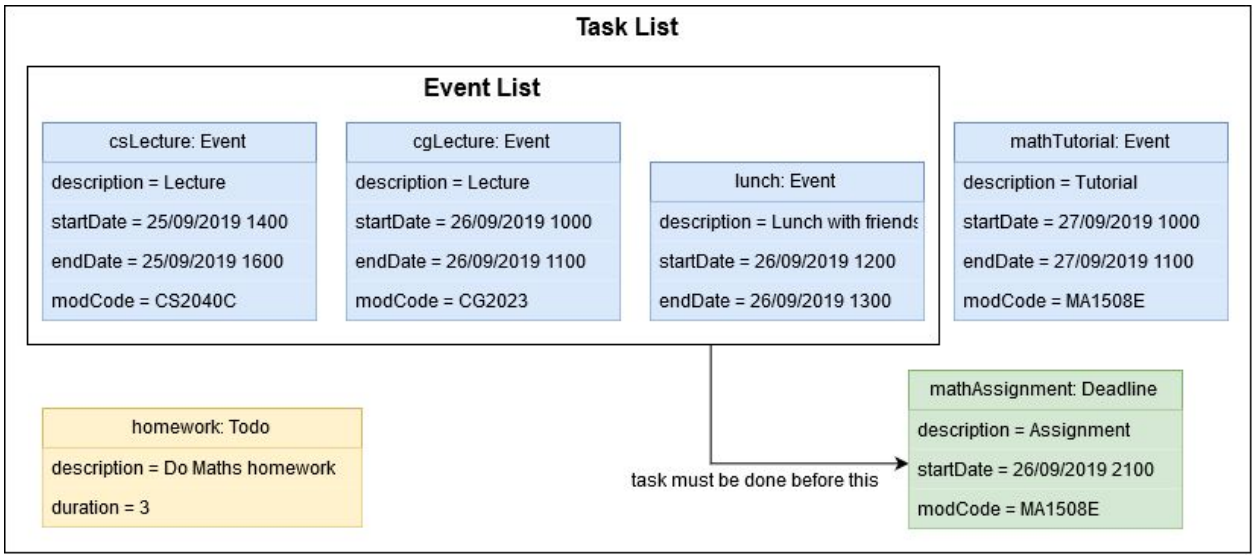
*Additional info: If the task at the 1st index is not a Todo or the task at the 2nd index is not a Deadline, the command will throw an exception, informing the user that they have selected the wrong index number.

Step 5.

TaskScheduler retrieves all events before the assignment and sort them chronologically in a list. Any events before the current time or after the assignment deadline is ignored and not factored in.

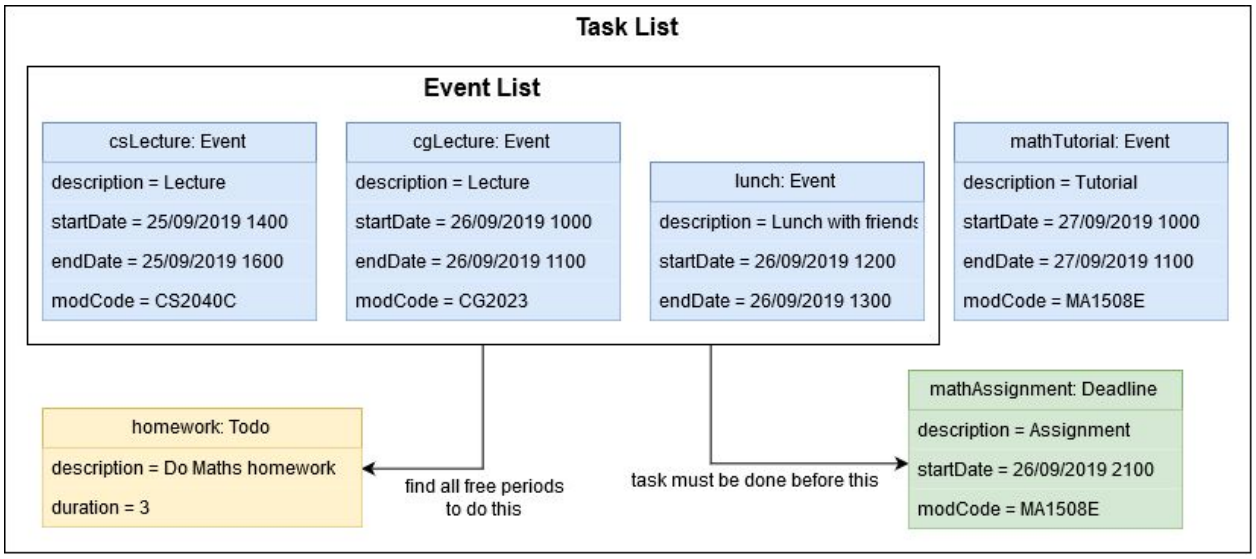
*If the duration is greater than the period between the present time and the deadline date, Chronologer informs the user that it is not feasible to accomplish the task by the date.

*If no deadline was selected, TaskScheduler calls scheduleTask() instead and retrieves all the events within a hard limit of 30 days.



Step 6.

TaskScheduler iterates through the list to find a period between events that is large enough to accomodate the task duration. Whenever it has found such a period, it retrieves the period start date and end date to be placed within the display result message that will be sent to the UI.

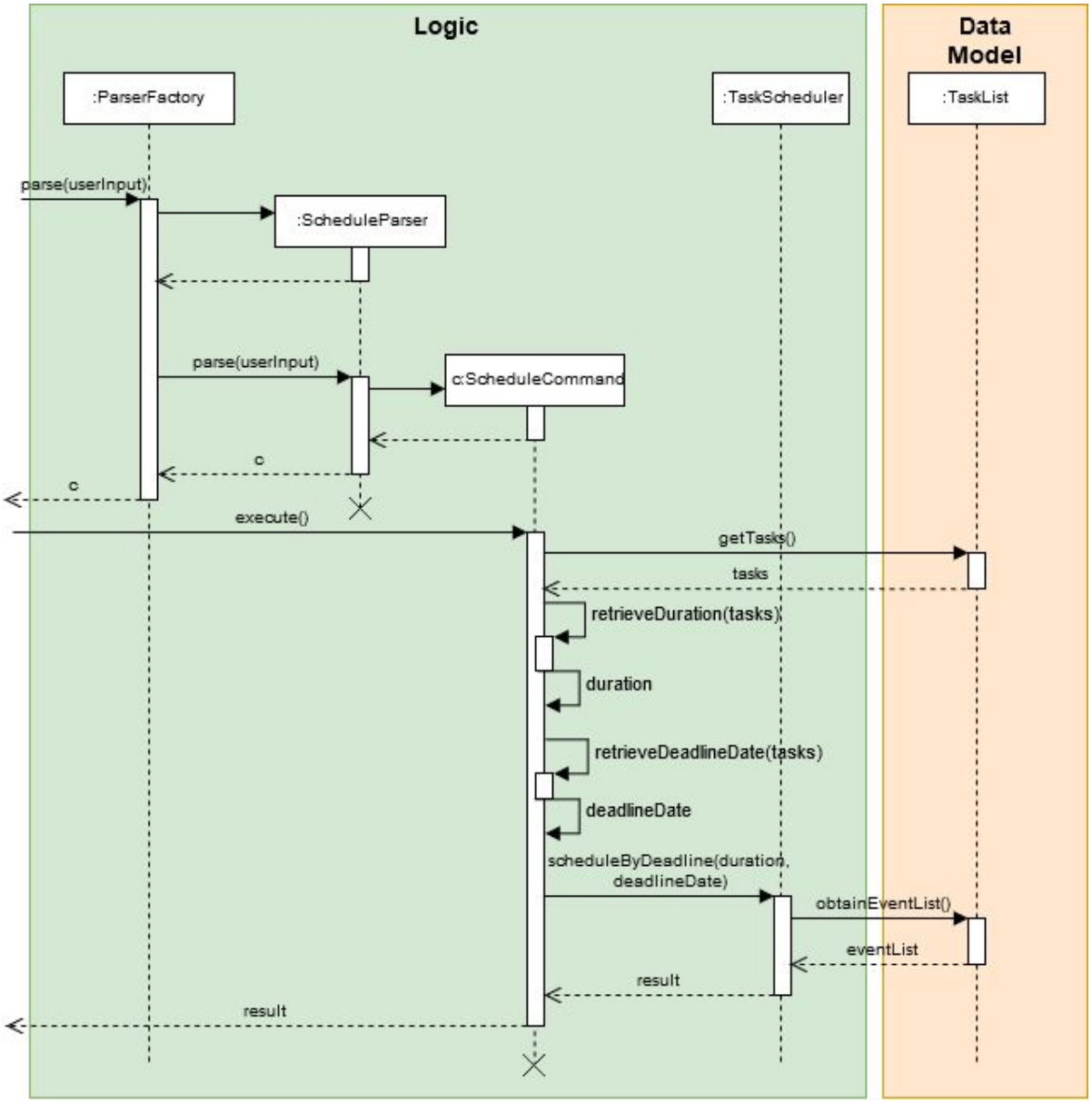


Step 7.

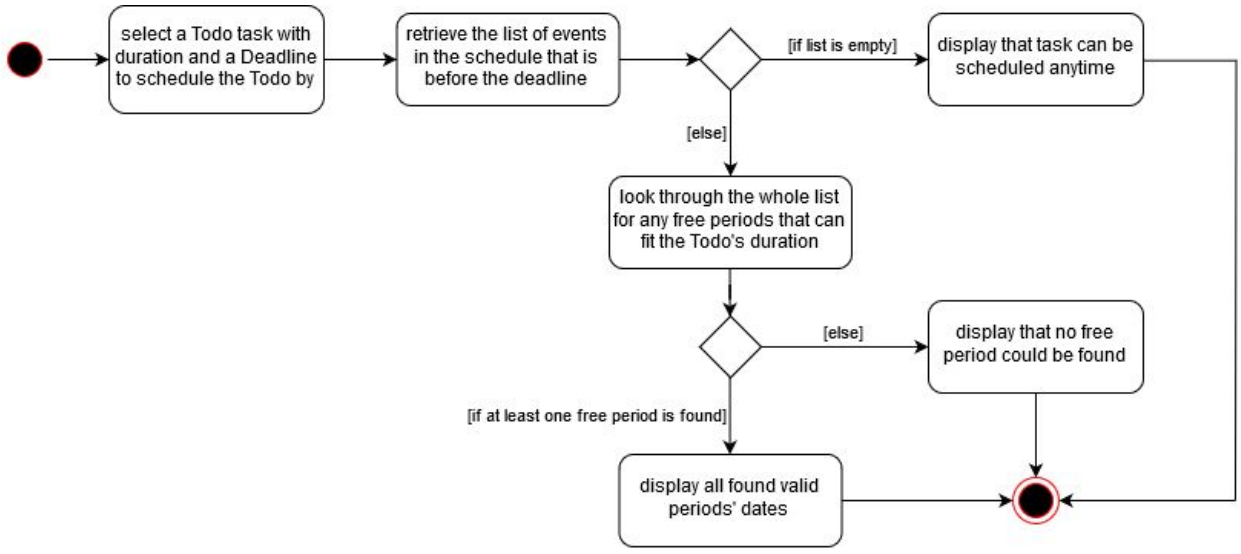
Once TaskScheduler iterates through the whole list, Chronologer displays a message to the user that it has found at least one slots that they can work on the task before the deadline. A list of periods is then presented below the message.

Additional info: If a free period could not be found, Chronologer would instead inform the user that a free period could not be found and ask them to consider freeing up their schedule. Additionally, if low-priority events were included in the list, Chronologer would suggest freeing them up as considerations.

Below is the sequence diagram which portrays how the feature works:



Below is the activity diagram for a general flow of the feature.



4.4.2 Design considerations

1) Using multiple task lists to maintain different types of tasks

Pros:

User don't need to look through the whole masterlist to find the desired Todo and Deadline. Eg. Inputting '1' as the parameter for Todo selects the 1st Todo and '1' for Deadline parameter selects the 1st Deadline.

Cons:

Need to maintain multiple task lists and ensure consistency for all of them is maintained after every command

2) Automatically convert Todo with duration to a Todo with period when a free period is found

Pros:

Automatic scheduling and insertion of tasks within the user's schedule can be convenient to the user who are not picky where the task ends up in.

Cons:

Need to implement some logic to determine the best timing to place the Todo in. Additionally, the user may find it troublesome to manually edit the Todo if it's placed at a time that the user does not exactly like.

Appendix A: Instructions for manual testing

Step 0:

A. Type in "tester" to load the schedule with a pre-filled tasks for testing purpose.
B. Type in "sudo-clear" to clear the schedule (WARNING: irreversible)

a) Export timeline as ics feature test guide

a1) Export all supported task

1. For best result, initialise with an empty list. Type in "sudo-clear" to do so if list not already empty. (WARNING: This action is non-reversible)

2. Add a deadline task, an event task and a todo with period task to the tasklist eg.
    A. deadline submit homework /by 12/12/2019 1900
    B. Event circus /at 13/12/2019 1900 - 14/12/2019 1600
    C. todo housework /between 15/12/2019 1600 - 15/12/2019 1700

3. Type in export <file name> to convert the tasklist to an ics format.
   Eg:
   - Export mySchedule
4. Go to **src/ChronologerDatabase** directory and open the ics file (mySchedule.ics in this case) with any supporting application eg: Calendar, online ics converter etc.

5. Ensure that the tasks in the tasklist are reflected correctly in the calendar application.

a2) Export specific tasks types

1. For best result, initialise with an empty list. Type in “sudo-clear” to do so. (WARNING: This action is non-reversible)

2. Add a deadline task, an event task and a todo with period task to the tasklist.You may reuse the same example above.

3. Type in export <file name> -d to only convert deadline type tasks into an ics file. (Note that deadline type tasks includes exams and assignments due to their nature,essentially all tasks marked with **[D]** in list).
   Eg:
   - Export deadlineSchedule -d

4. Go to **src/ChronologerDatabase** directory and open the ics file (mySchedule.ics in this case) with any supporting application eg: Calendar, online ics converter etc.

5. Ensure that only the deadline tasks are converted into the calendar (Tasks in **list** marked with **[D]**).

6. Repeat step 2-4 but using different flags each time (-d,-e,-t) and ensure that only the corresponding task types are converted. (You may use multiple flags in the command)
   Eg:
   - Export eventSchedule -e (Tasks in list marked with **[E]** converted)
   - Export todoSchedule -t (Tasks in list marked with **[T]** converted but only those with date range)
   - Export deadlineWithEvent -d -e (Tasks in list marked with **[D] + [E]** converted)

b) School Tasks management feature test guide

b1) Adding assignments

1. Type “Assignment /m <Module Code> /by <dd/mm/yyyy HHmm>” to add an assignment with a module code attached to that date and time.
   Conditions:
   1. Input is not case-sensitive
   2. Module code must be a *valid* module code
   3. Time must in 24-hour format

   Eg:
   - Assignment /m cs2040C /by 17/11/2019 2359

2. Chronologer will give a confirmation message.

3. Type “list” to output the tasklist and ensure the assignment has been added to the list and reflected in the timeline on the GUI.

b2) Adding Examinations

1. Type “exam /m <Module Code> </at> <start_datetime - end_datetime>” to add an examination with a module code attached to that date and time.
   Conditions:
   1. Input is not case-sensitive
   2. Module code must be a *valid* module code
   3. Time must be in 24-hour format, start/end timing can be swapped

   Eg:
   - exam /m MA1508E /at 16/11/2019 1000 - 16/11/2019 1200
   - exam /m MA1508E /at 17/11/2019 1200 - 17/11/2019 1000

2. Chronologer will give a confirmation message.

3. Type “list” to output the tasklist and ensure the exam has been added to the list and reflected in the timeline on the GUI.

b3) Adding lectures

1. Type “lecture </m> <module code of examination> </at> <day_of_week start_time-end_time> </till> <end_date>” to add weekly lectures with a module code attached on the same time slot every week till the date given..
   Conditions:
   1. Input is not case-sensitive
   2. Module code must be given and a *valid* module code
   3. Time must be in 24-hour format, start/end timing can be swapped
   4. Day_of_week allowed to be misspelled as long as the first 3 letters are correct

   Eg:
   - lecture /m Ec1301 /at mondays 1200-1400 /till 01/01/2020
   - Lecture /m ma1508E /at wednezdais 1000-0800 /till 12/12/2019

2. Chronologer will give a confirmation message and inform how many tasks are now in the list.

3. Type “list” to output the tasklist and ensure the lectures has been added to the list and reflected in the timeline on the GUI.

b4) Adding tutorials

1. Type “tutorial </m> <module code of examination> </at> <day_of_week start_time-end_time> </till> <end_date>” to add weekly tutorials with a module code attached on the same time slot every week till the date given.
   Conditions:
   a. Input is not case-sensitive
   b. Module code must be given and a *valid* module code
   c. Time must be in 24-hour format, start/end timing can be swapped
   d. Day_of_week allowed to be misspelled as long as the first 3 letters are correct

   Eg:
   - tutorial /m cs2040c /at tuesdays 0800-1000 /till 25/12/2020
   - tuTorial /m cs2040c /at thurzdai 1000-1200 /till 25/12/2020

2. Chronologer will give a confirmation message and inform how many tasks are now in the list.

3. Type “list” to output the tasklist and ensure the exam has been added to the list and reflected in the timeline on the GUI.

c) Check Scheduling feature is finding the correct free periods

1. For best result, initialise with an empty list. Type in “sudo-clear” to do so. (WARNING: This action is non-reversible)
2. Add a deadline 24 hours from now. (eg. deadline test /by 16/11/2019 1600)
   - Assuming Step 1 is followed, this deadline is at index 1 of the viewable task list
3. Add a todo with a duration of 4 hours. (eg. todo something /for 4)
4. Schedule the todo by the existing deadline by typing “schedule 2 /by 1”
5. The output would be “You can schedule this task from now till the deadline.”
   - If there are existing events between the present time and the deadline, the output would be “You can schedule this task from <DATE_TIME> till <DATE_TIME> instead
6. To test raw date input, type “schedule 2 /by 16/11/2019 1800”
7. The display message should be the same in Step 5.

8. To test raw duration input for Todo, include the “-r” flag after the duration value in the input. Type “schedule 4 -r /by 2” to find a free period of 4 hours by 16/11/2019 1600.
9. The same can be done for raw date input. Type “schedule 4 -r /by 16/11/2019 1800”
10. Add in events by typing in “event PE run /at 15/11/2019 1600 - 15/11/2019 1800” or similar. Type in the same schedule command inputs as above, Chronologer will display a different result message based on the events inputted.
    - If there’s no free period found, the result message will inform that there’s no free period.

d) Checking history features.

1. For optimal testing, utilise the store command by typing “store 1”. Now you have the initial state stored as version 1.
2. Then proceed to initialise with an empty list. Type in “sudo-clear” to do so. (WARNING: This action is non-reversible)
3. Add a deadline task. (eg. deadline test /by 15/11/2019 1600)
   - Assuming Step 2 is followed, this deadline is at index 1 (to check type “list”)
4. Now add another todo task by simply typing “todo Test”.
5. Now you can test the undo command by typing “undo”. This should give you an output “Undo successful!”.
6. Now you can test the redo command as you have performed the undo command prior in Step 5. Simply type “redo” and the output should be “Redo successful!”.
7. Now you may also test the restore command by typing in “restore 1” which should revert the task manager to the state it was when you performed Step 1.
8. Another feature you could now test having entered multiple commands is the command line history feature. Simply use the arrow keys “up” and “down” to navigate through your previous inputs.
    - If you attempt to undo further than possible or redo further than possible you will be alerted. Accordingly.
    - If you attempt to store or restore version number apart from 1-3 you will be alerted.

e) Checking GUI features.

1. To test and see different weeks of the preloaded tasklist, simply key in “week 2”. (the week numbers range from 0-18, do take note that recess week is week 7, the week after is 8 etc.)
2. If you’d like to see your current week simply type “week current”
3. If you’d like to change themes simply enter “theme light” or “theme dark”
    - For repeat inputs you will be notified that the GUI will not change.

Appendix B: Requirements

a) User stories

Priority: 1 - Must-Have, 2 - Nice-To-have, 3 - Not-Useful

Priority	Users	Function	Benefits
1	Student who likes to travel	See schedule on a GUI such that it resembles a timeline.	Possible to check for a free time slot to help the user plan trips during times without classes
1	Student	See a more visual representation (eg. GUI) of all upcoming task deadlines.	Allows the user to prioritise with ease.
1	Student with a hobby	Set a task that can recur at weekly intervals and appear on a GUI	Will prevent the user from having to add a weekly task repeatedly
1	Student with a lot of Co-Curricular Activities	Ability to rate the priority of a particular task with a visual icon or symbol	Allows the user to focus on certain tasks
1	Student	Ability to store the schedule in the form of a human readable file to make it portable	Allows the user to see the schedule without using software all the time
1	Student	Allows the user to mark certain tasks as skippable/ignorable	Allows the user to move around or allocate more time for more crucial tasks
1	Student	Ability to add comments and description to a task which can be seen in a GUI	Allows the user to be able to easily identify tasks etc.
1	Forgetful Student	Reminder options	Remember tasks
1	Busy student	Warns of time clashes	Better planning
1	Student who likes to	View task	Organise his/her

	organise	deadlines in one place	schedule better
1	Student	Ability to give me a reminder from the task deadline or at the time of the task like 3 days earlier	I do not have to actively remember and keep track of important task
1	Student	Can include durations to task	I'm reminded how long it will last
1	Student	Can have a location listed with the task.	I can plan my route
1	Student	Modify tasks easily	I can make last minute changes
1	Student	Search tasks by module code	Keep track of module's workload
1	Student	Add module-specific tasks (Lecture, assignments etc.)	Able to build up timelines for different modules
2	Student	A warning if schedule have no rest period	Avoid overwork
2	Minimalist student	Minimalistic software	Easy to read and interact
2	Student who likes to take shortcuts	Use CLI macros	Efficient and save time typing and navigating software
2	Student Mobile User	Import schedule to other devices	Refer to them on the go
2	Student who likes to plan ahead	See semester timeline	Plan my semester better
2	Student	Change timeline granularity	Adjust my planning scope
2	Student who likes to play video games	Check longest break period	Make most of gaming sessions
2	Student	Export ics to other calendar	Save onto cloud calendars
2	Student who also likes to work Part time	Plan schedule	To earn pocket money during

			semester
3	Social Student	Add participants to planned task	Keep track of outings with friends
3	Student who uses Luminus	Offline and lightweight alternative	Keep track of schedule without internet
3	Student	Daily reports/summary of the day's timeline	Prepare myself for the day
3	Student who likes to customise to one's liking	Customisable UI	App feels more personal and looks better in my eyes
3	Student	Ability to easily update the GUI	I don't become lazy and stop doing this
3	Student	Wants a software that works reliably	I can relax more instead worrying about petty stuff
3	Student	Open task manager automatically at certain times of the day	I can automate my life and be constantly reminded of my tasks

b) Use cases

(For all use cases,system is Chronologer and Actor is a Student unless specified)

Use case: UC1-Add a task

MSS:

1. Student keys in addition command for particular task.
2. Chronologer adds the task.
   Use case ends

Extensions:

- ❏ 2a) Chronologer detects that the student didn't fill in the required task info.
  - 2ai) Chronologer requests for the task input.
  - 2aii) Task keys in new lecture info
  - Steps 2ai - 2aii are repeated until all info are correct.

- ❏ 2b) Chronologer detects that the task clashes with another on the same time slot.
  - 2bi) Chronologer notifies user of the clash
  - 2bii) Student can either cancel task input or reenter with different time input.

- Steps 2bi - 2bii are repeated if clashes still occur.

Use case: UC02-Deleting a task

Precondition: The task is present in Chronologer.

MSS:

1. Student keys in delete command for that specific task.
2. Chronologer deletes the task.
   Use case ends.

Use case: UC03-Adding priority to task

Precondition: The task is present in Chronologer.

MSS:

1. Student inputs priority command for the task.
2. Chronologer adds the priority to the task & displays confirmation message of change.
   Use case ends

Extensions:

❏ 1a) Chronologer detects that the priority level is invalid
- 1ai) Chronologer requests for a valid priority input.
- 1aii) Student keys in new priority level.
- Steps 1ai - 1aii are repeated until all info are correct.
  Use case resume from step 2.

Use case: UC04-Adding reminder to task

Precondition: The task exists in the task manager

MSS:

1. Student keys in reminder command.
2. Chronologer sets reminder date to the task & will display reminder every subsequent time it's activated..
   Use case ends.

Extensions:

❏ 1a)Chronologer detects invalid date format.
- 1ai) Chronologer request for a valid date.
- 1aii) Student keys in new date.
- Steps 1ai - 1aii are repeated until all info are correct.
  Use case resume from step 2.

Use case: UC05-Search for nearest free time-slot

Guarantees: The search only includes time period after current system time.

MSS:

1. Student keys in a command to search for a free time slot.
2. Chronologer finds and displays the nearest free time slot.
   Use case ends.

Use case: UC06-Display schedule for certain date
Guarantees: Schedule shown sorted by time.
MSS:
1. Student keys in a command to display schedule.
2. Chronologer displays schedule of the day.
   Use case ends.

Extensions:
- ❏ 1a) Chronologer detects that there's no task on that particular date.
    - 1ai) Chronologer notifies user that the date has no tasks.

Use case: UC07-Edit task description
Preconditions: Task present in Chronologer.
MSS:
1. Student keys in a command to edit task description.
2. Chronologer edits task description and displays a confirmation message.
   Use case ends.

Use case: UC08-Postpone task to another date
Precondition: Task present in Chronologer.
MSS:
1. Students keys in command to postpone task.
2. Chronologer postpones that task to a new date and displays a confirmation message
   Use case ends

Extensions:
- ❏ 1a) Chronologer detects that new date clashes with another task.
    - 1ai) Chronologer notifies user of the clash.
    - 1aii) User can either retry with another date or cancel operation.

- ❏ 1b) Chronologer detects incorrect date format in student input.
    - 1bi) Chronologer informs user about date format mistakes and request valid date.
    - 1bii)User keys in new date.
    - Steps 1bi - 1bii are repeated until all info are correct.
      Use case resumes from step 2.

Use case: UC09-Export schedule as .ics file
MSS:
1. Student keys in export command.

2. Chronologer process and export system schedule in .ics format that can be opened by any supporting application in the computer.
   Use case ends.

Extensions:

- ❏ 1a) Chronologer detects that there's no task in its system.
    - 1ai) Chronologer notifies user that it's unable to create a .ics file due to empty schedule.

Use case: UC10-Add location details to task

Preconditions: Task present in Chronologer.

MSS:

1. Students keys in command to add location to certain task.
2. Chronologer adds location to task and displays a confirmation message.
   Use case ends.

Extensions:

- ❏ 1a) User forgot to add location by mistake.
    - 1ai) Chronologer notifies user that location is empty.
    - 1aii)User reenters location command.
    - Step 1ai - 1aii is repeated until all info correct or user cancels input. Use case resumes from step 2.

Use case: UC11-Add a recurring task

MSS:

1. Student keys in additional command for recurring task.
2. Chronologer adds the lecture.
   Use case ends.

Extensions:

- ❏ 2a) Chronologer detects that the student didn't fill in the required task info.
    - 2ai) Chronologer requests for the task input.
    - 2aii) Student keys in new task info
      Steps 2ai - 2aii are repeated until all info are correct.

- ❏ 2b) Chronologer detects that the task clashes with another task on the same time slot.
    - 2bi) Chronologer notifies user of the clash
    - 2bii) Student can either cancel task input or reenter with different time input.
    - Steps 2bi - 2bii are repeated if clashes still occur.

Use case: UC12-Schedule a task before a deadline

MSS:

1. Student keys in command to schedule a task before a specified deadline.
2. Chronologer displays all valid periods that can accommodate the task.
   Use case ends.

Extensions:

❏ 2a) Chronologer detects that the selected tasks are incompatible.
- 2ai) Chronologer notifies user of the error.
  Use case ends.

❏ 2b) Chronologer detects that there's no free period for the task.
- 2bi) Chronologer notifies user of the lack of a free period.
  Use case ends.

c) Non functional requirements

1. The commands to be inputted in the CLI should be simple and easy to remember.

2. The response to any user action should become visible within a few seconds.

3. The software should be functional without internet connection.

4. Software should be easily testable.

5. Software should be able to operate on any mainstream OS with Java 11 or above installed.

6. Software should have adequate user and developer documentation .

D: Glossary

a) Priority: Level of importance for a certain lesson.

b) Timeline: Graphical representation for a daily schedule.

c) Clashes/Collision : Lessons that share the same time period .

d) ICS: Calendar file format.

e) GUI: Graphical user interface.

f) iCalendar: Common data format used to store information about calendar specific data such as events, appointments etc.