RIMS Developer Guide

By: `Team W12-1` Since: `Aug 2019` Licence: `MIT`

1. Design

1.1. Architecture

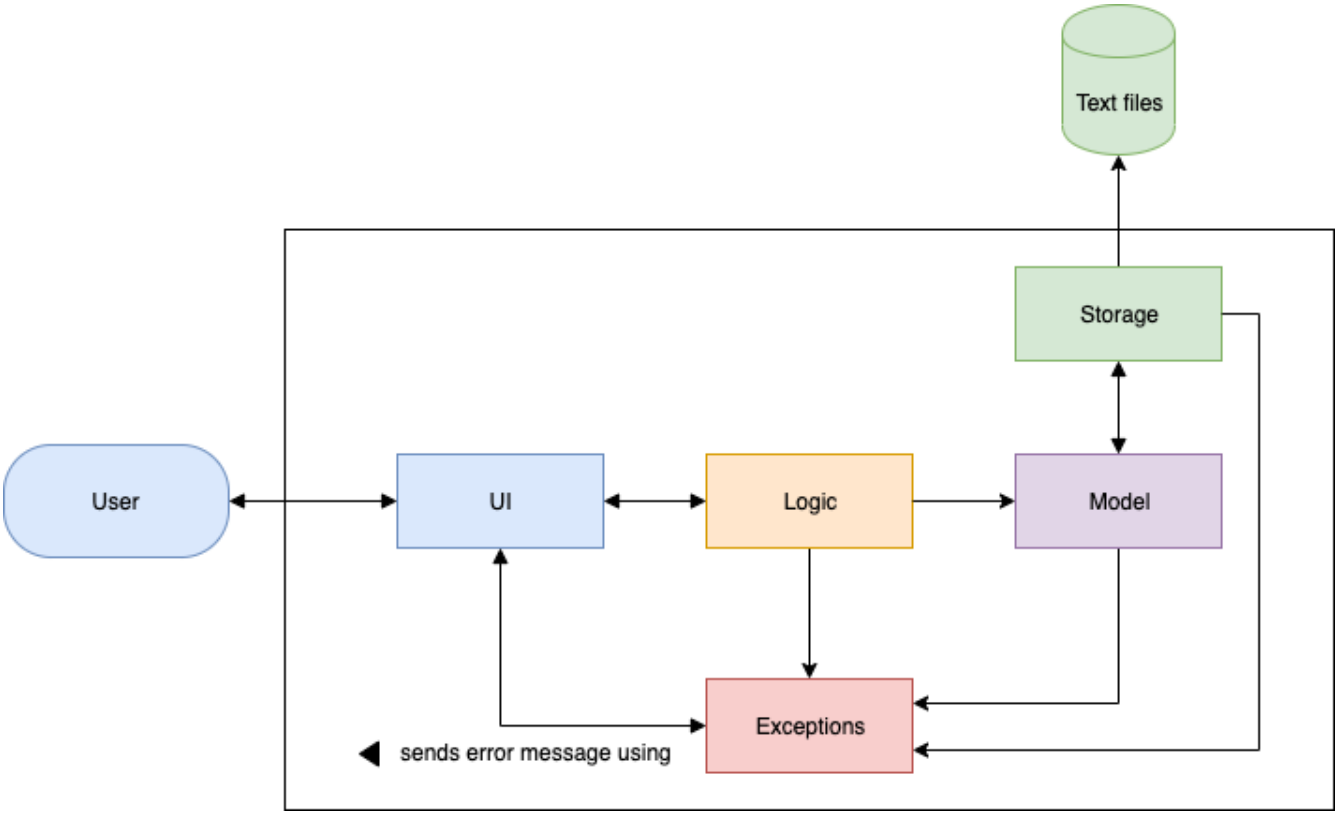

Figure 1.1: General Architecture Diagram for RIMS

The architecture diagram above is an abstracted high-level representation of the workings behind the RIMS software. There are a total of 5 main components within RIMS, and outside of RIMS, a single **User**, as well as **Text files** for data persistence.

The main components are listed below, with a brief description and overview of their scope:

- Model: The structure of RIMS, complete with resource and reservation details.
- Ui: The main user interface that the user interacts with to enter his/her input and receive updates and messages from RIMS itself.
- Logic: The mechanism which makes sense of the input given by Ui and subsequently executes the relevant commands within RIMS.
- Storage: Reads data from, and writes data to the external **Text files**.
- Exceptions: Custom RIMS exceptions that may be raised from contextual errors while executing commands in RIMS.

Note: In order to declutter the class diagrams shown below, CRUD (Create, Read, Update, Delete) methods for attributes in classes are omitted.

1.2. Model component

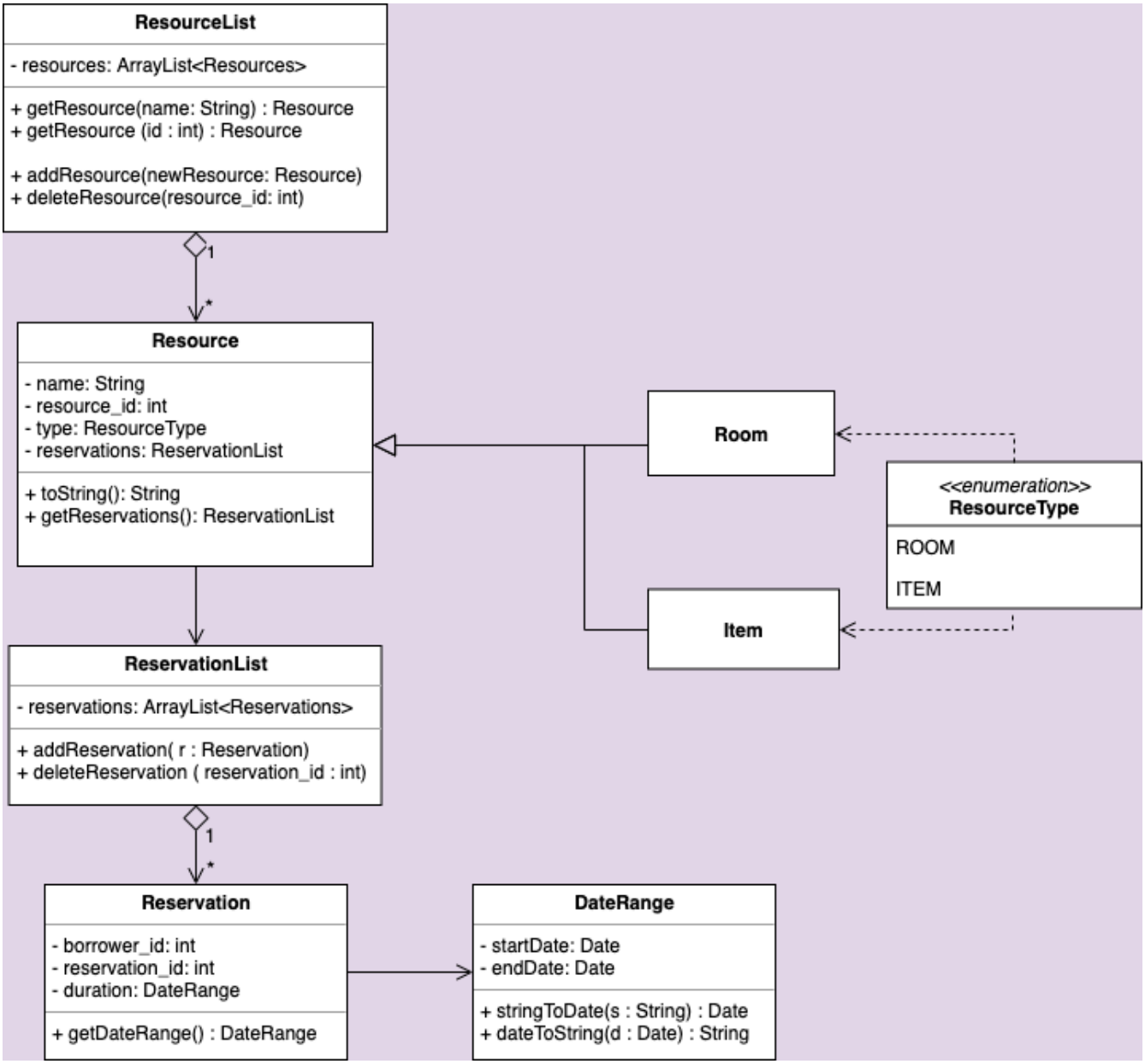

Figure 1.2: Class Diagram for Model component

Classes involved: Resource, Item, Room, ResourceList, ReservationList, Reservation, DateRange

The Model component forms the backbone of the RIMS software and is the structure to inventorise Resources and manage their loans. With reference to Figure 1.2,The RIMS structure comprises a single large ResourceList, which is an ArrayList of Resource objects with various custom methods to make changes in accordance to commands.

1.2.1. The Resource Class

A resource is represented by the Resource class, with basic attributes such as its name and its type (Room or Item represented by an enumeration) are inside. An enumeration ResourceType is used to represent types of resources so that more can be accomodated in the future. Classes Room and Item are inherited classes from Resource.

1.2.2. ReservationList and resource_id

The more important attributes of a Resource object is the resource_id and ReservationList. Firstly,

the resource_id is an integer to identify a single unique Resource. For example, if RIMS has two pens, which were borrowed for different durations by different users, I would need to be able to identify which one is available when a third user wants to borrow one again.

This leads to the second important attribute which is the ReservationList, which is an ArrayList of Reservation objects. A Reservation object contains details of a loan or reservation, such as the identity of the borrower (borrower_id), the unique identifier for that particular loan (reservation_id) to identify multiple unique Resources borrowed, and lastly a custom DateRange object to identify the duration for which the Resource is being borrowed.

The DateRange class contains functions to typecast the dates into a specified format of String type for passing to Ui as a message to the user.

1.3. UI component

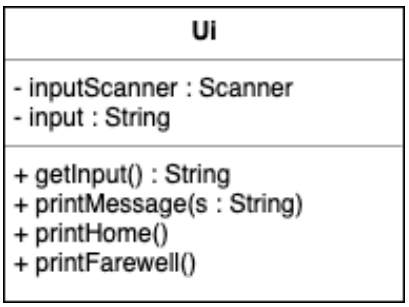

Figure 1.3 Class Diagram for Model component

Classes involved: Ui

The Ui class fulfills two purposes, as shown in Figure 1.3:

1. Taking in input given by the user.
2. Sending messages to notify the user.

For the first function, the Ui class contains a Scanner object to take in user input, as well as an input of String type to process it via the Logic component as a valid command/instruction.

For the second function, the Ui class has functions to print out user messages in a specific format. Examples include printHome(), which is the welcome message when launching RIMS, as well as printFarewell(), which is the message sent to the user upon termination of RIMS.

1.4. Logic component

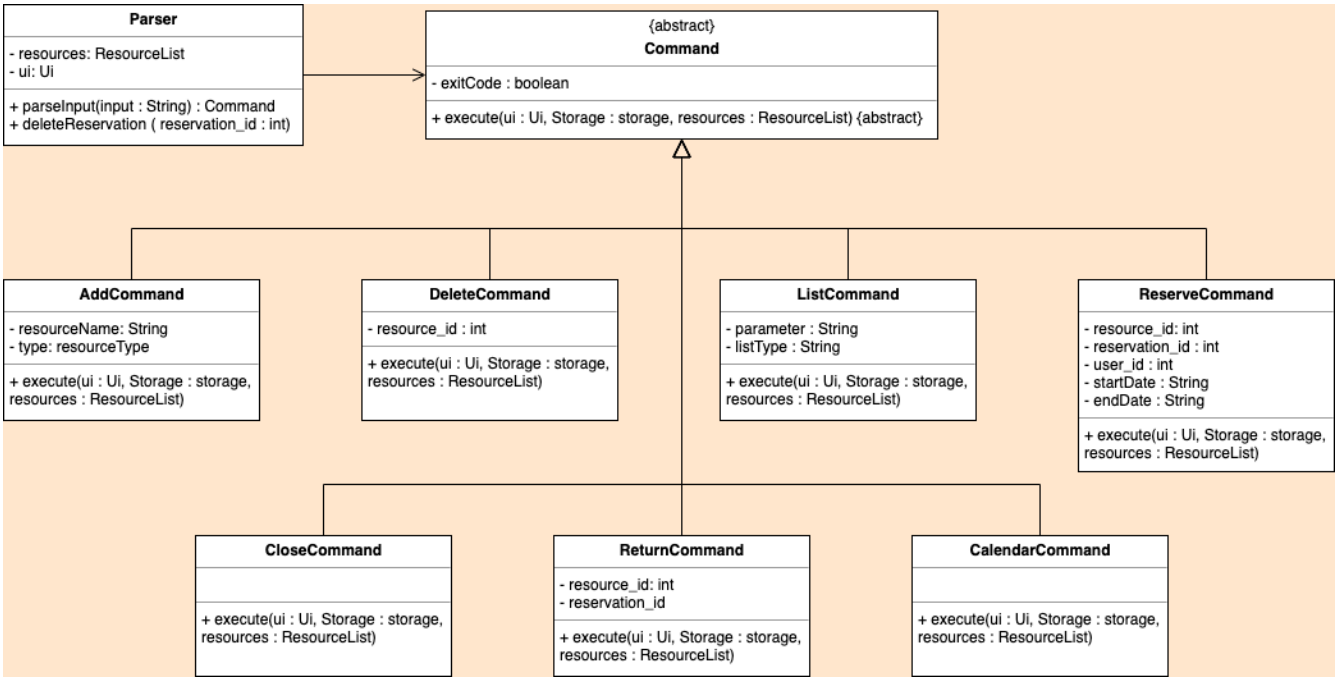

Figure 1.4: Class Diagram for Model component

Classes involved: Parser, Command and its inherited classes

The logic component is shows how user instructions are managed in RIMS. The Parser class receives input from the Ui, then parses it (using parseInput function) to identify the type of Command it is. After identifying the type of Command, Parser then evaluates the relevant parameters required and constructs the corresponding Command. Following that, the Command's execute method is called to carry out the instructions.

1.4.1. The Command class

Referring to Figure 1.4, the Command class is an abstract class with two main parts that are crucial for all inherited Command classes.

Firstly, since the main program runs on a loop to evaluate user inputs repeatedly for commands, it needs to know when to terminate. The Boolean attribute exitCode helps the main program to check if the Command is a CloseCommand, a special Command which instructs RIMS to terminate.

Secondly, the execute method is an abstract method that applies to all Commands in the RIMS system.

1. Using the parameters evaluated in Parser, it interacts with the Model component to make the necessary checks/updates to RIMS
2. A formatted message is to the user about the outcome via Ui
3. If necessary, data is saved to external files using the Storage component.

1.5. Storage component

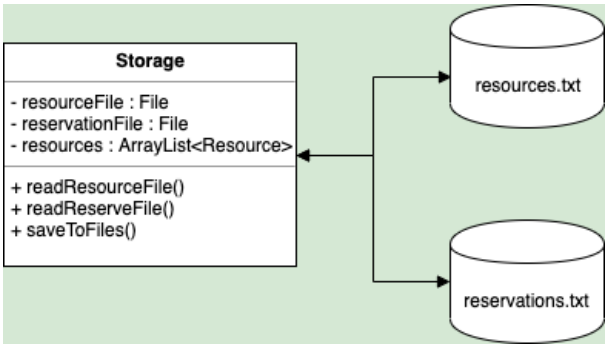

Figure 1.5: Class Diagram for Model component

Classes involved: Storage

The Storage class ensures data persistence upon termination of RIMS and is the intermediary between the external files and the Model component. Before discussing its attributes and methods, refer to Figure 1.5 to see where and how data in RIMS is saved and loaded.

1.5.1. External files

Data in RIMS is saved and loaded from two main text (.txt) files.

- resources.txt: Contains the current list of resources that are recorded by RIMS, regardless of their status (excluding the details of resources' reservations)
- reservations.txt: Contains the list of reservations and their details for every resource listed in RIMS.

1.5.2. Storage attributes and methods (Loading and Saving)

Loading and saving to/from the above mentioned external files are vital for Storage to function. In order to do both, Storage has attributes of File type which records the paths to the local external files. There is also a temporary ArrayList of Resources to retrieve and store data from the Model component.

For loading from files, we need to read both external files mentioned above, so dedicated methods for resources.txt and reservations.txt are present (as readResourceFile() and readReserveFile()).

As mentioned before in the Model component, Reservations are integrated into each Resource, therefore when saving into external files, we only need one method saveToFiles().

1.6. Exception component

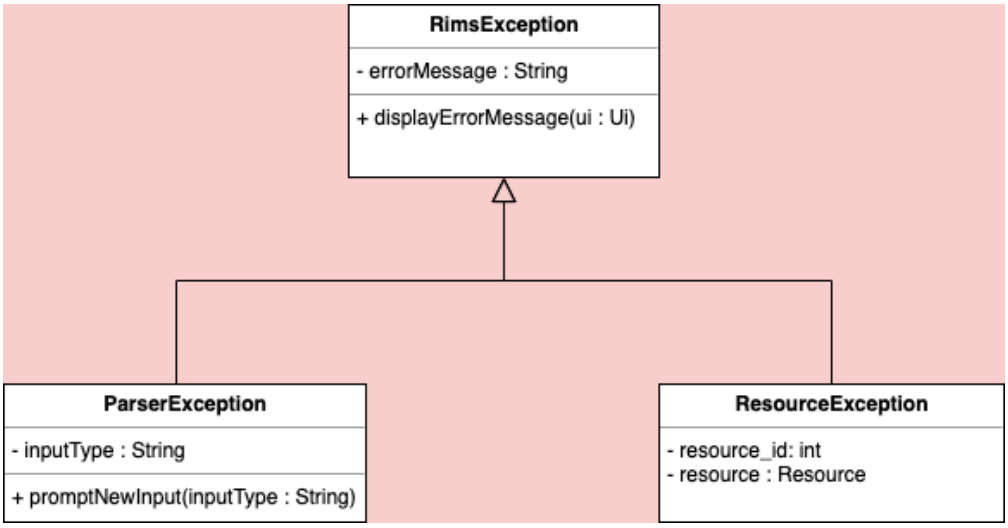

Figure 1.6: Class Diagram for Model component

Classes involved: RimsException and all inherited classes

RimsException is a custom Exception class to catch all errors presented by contextual problems occuring within RIMS that basic Exceptions will miss. The overarching idea behind this component is that RIMS should not terminate under any other circumstances other than by means of the CloseCommand (as mentioned in the Logic Component).

The basic RimsException class contains a String describing the type of error that has been occurred, and has a method to send a formatted output to the user notifying him/her about the error that has occurred.

Current subclasses of RimsException are ParserException and ResourceException, but are not limited to these as there may be other aspects which have errors that have not been identified.

1.6.1. ParserException

ParserExceptions are called when the Parser receives an invalid input. For instance, when the user is prompted for the type of resource specified, and he/she inputs "people", a ParserException should be raised. This is because the type of Resource specified does not belong to any that RIMS support, which are "item" and "room".

They are mainly handled by prompting the user to re-input a valid format of the required information.

1.6.2. ResourceException

These Exceptions are mainly handled when internal operations within the Model or Logic component have problems. For example, user inputs like the duration to borrow a resource is valid, but the resource might still be on loan. As such, a ResourceException is called.

They are mainly handled by sending a message to the user to notify them on the problem with the particular Resource.

2. Implementation

2.1. Add/Delete

Implemented by: Sean
Written by: Sean

2.1.1. Implementation

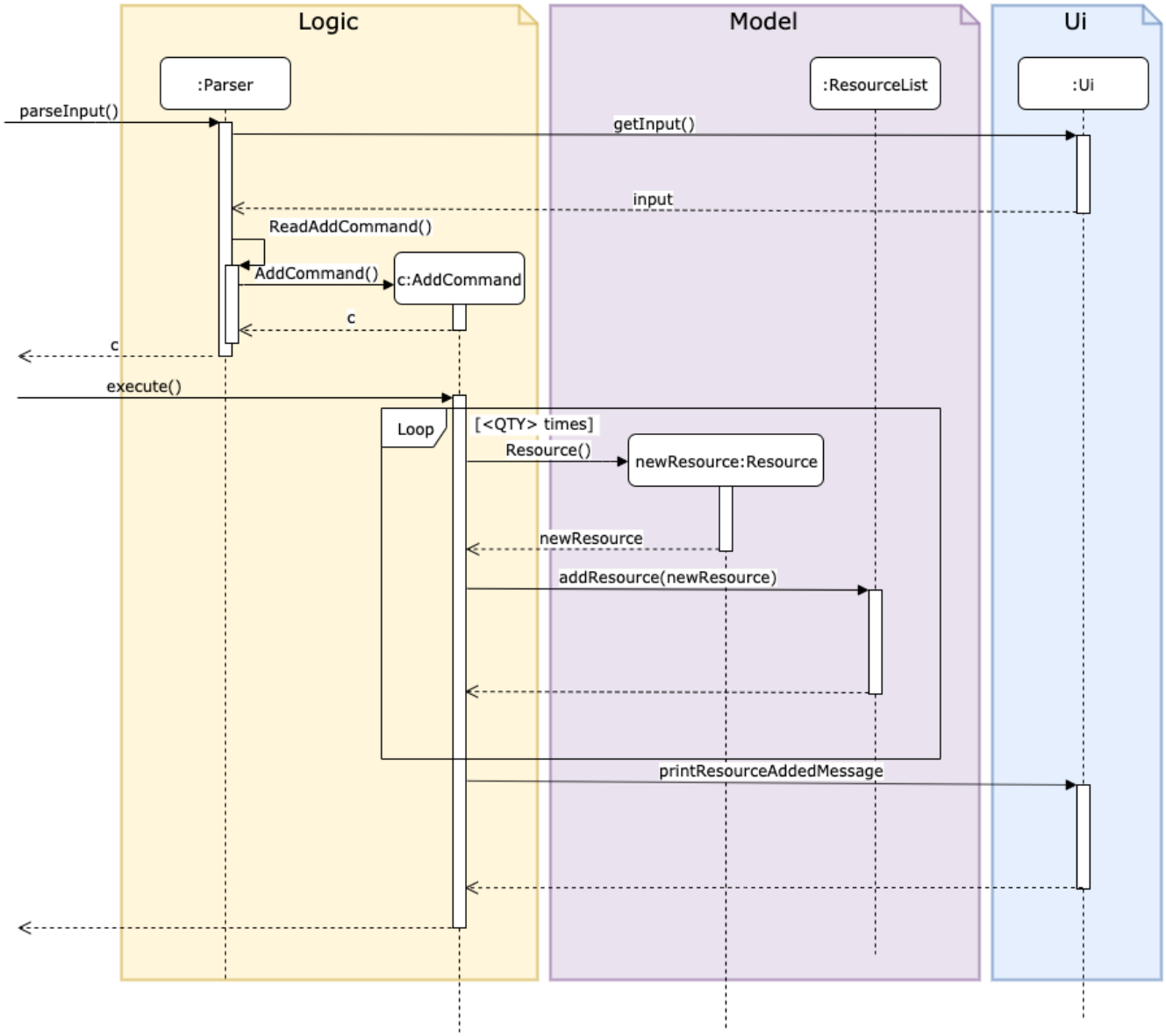

Figure 2.1.1: Sequence Diagram for Add

Adding and deleting resources from RIMS are facilitated by the AddCommand and DeleteCommand classes respectively, after receiving input from the user in the Ui and being passed through Parser. For adding resources, consider the case where the user wants to add a pen to RIMS.

1. When starting up RIMS, a Ui and Parser object is instantiated respectively.
2. Referring to Figure 1, a parseInput() function is called, prompting for input from the user. He/She types the relevant input, such as that the pen is of Item type, to the Ui object (getInput() function).

3. This input is received by the Parser object, which contains the relevant parameters (the Parser object also prompts for new input from user should the format be incorrect).
4. Following that, the Parser object identifies (through ReadAddCommand()) and constructs a new AddCommand with the evaluated parameters.
5. Afterwards, the AddCommand is executed with the execute() function, which constructs a new Resource based on the parameters provided in AddCommand, adds this Resource to the existing ResourceList in RIMS (through AddResource()). This step is repeated for the quantity (specified by <QTY>) of that particular Resource that is required to be added, as specified in AddCommand.
6. Lastly, the Ui object prints a message to the user to notify him/her of the resources that have been added into RIMS.

For deleting resources, the steps are largely the same, however, there are more errors to identify during execute() (refer to Figure 2.1.1). For instance, should the specified resource to be deleted not exist, Ui will print a message to notify the user. Also, should the specified resource to be deleted be on loan at the moment, UI will also send a message to the user to notify him/her of such.

2.1.2. Design Considerations

Aspect 1: How Resource objects are added

- **Alternative 1 (current choice)**: ResourceList adds a single Resource object created in AddCommand into list
  - Advantages: Follows contextual situation (recording object by object on an inventory book using paper and pen)
  - Disadvantages: Requires multiple calls to add multiple objects
- **Alternative 2**: ResourceList creates a single Resource object to add, given its details, and adds it into list
  - Advantages: Do not have to create Resource object in AddCommand
  - Disadvantages: ResourceList must create a new Resource to add to list
- **Alternative 3**: ResourceList has function to add/delete all Resources with a user specified quantity
  - Advantages: Only requires one function call in AddCommand.
  - Disadvantages: Since all resources are added in ResourceList, limited information about each unique resource added can be extracted for sending the message to the user

Aspect 2: How Resource objects are deleted

- **Alternative 1 (current choice)**: ResourceList deletes a single Resource object from the list using its resource ID
  - Pros: Identifies unique Resource to be deleted using its resource ID
  - Cons: May have unnecessary iteration by checking resource IDs of other Resources that are not of the same type

- **Alternative 2**: ResourceList deletes a single Resource object, given its name from the list
  - Pros: Delete any Resource that corresponds to the given name
  - Cons: Cannot pinpoint a unique Resource object to delete

2.2. Reserve/Loan

Implemented by: Bobby/Abhijit
Written by: Bobby/Sean

2.2.1. Implementation

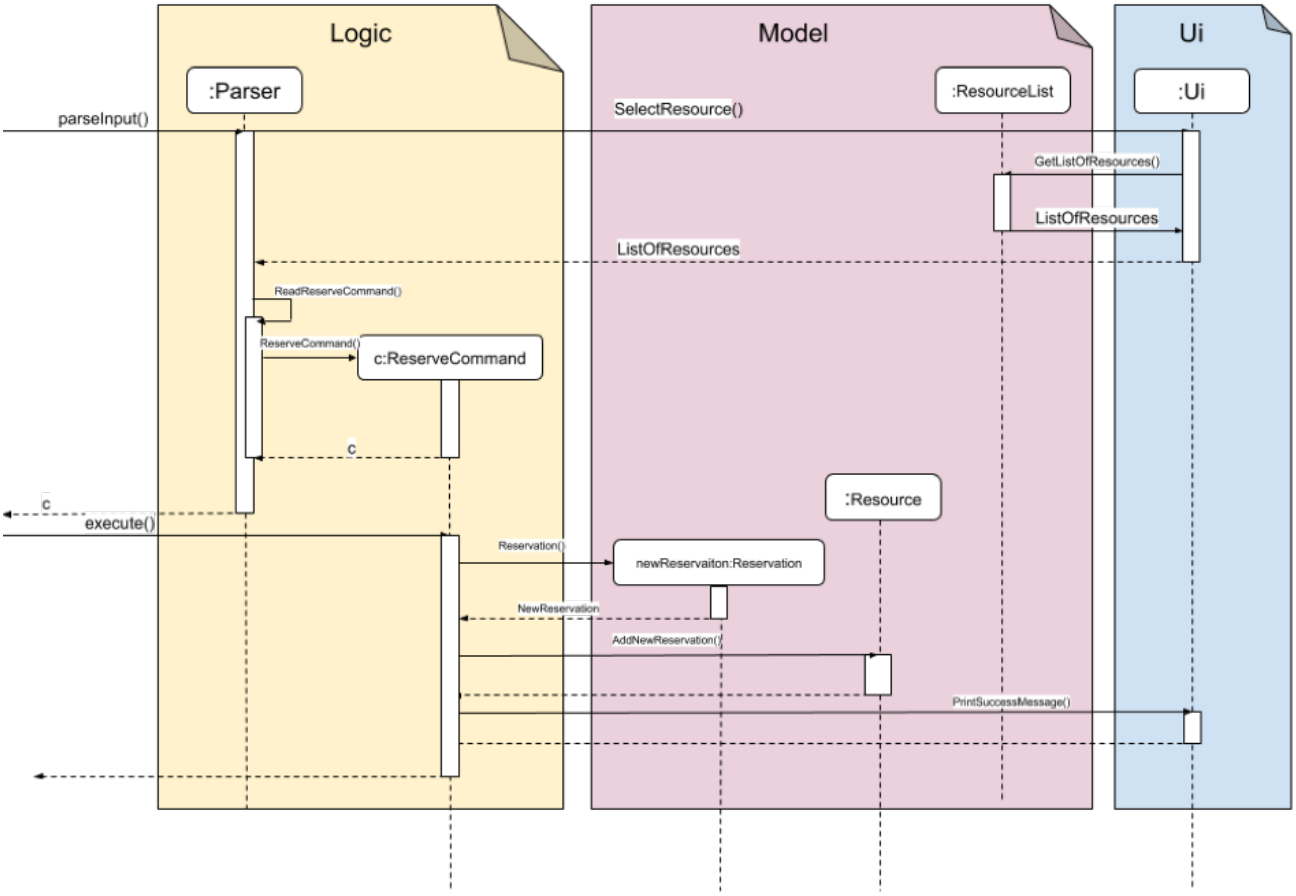

Figure 2.2.1: Sequence Diagram for Reserve

Making a reservation utilizes the following classes. The Parser and UI class will gather inputs from users. Then, these inputs will be used to create new reservation under the ReserveCommand class. New reservations are added to the ReservationList class under each Resource object.

For making a reservation, consider the case where a user wants to make reservation of an existing resource. When starting up RIMS, a Ui and Parser object is instantiated respectively.

1. Referring to Figure 2.2.1, a parseInput() function is called, prompting for input from the user. He/She enters the relevant input which is gathered by getinput() from Ui object.
2. After selecting a type of Resource, The Ui object will display a list of all Resource belong to that

type for the user to choose.

3. The user can then enter the name of the Resource they wish to make a reservation for.
4. Then, the user can choose a time period they wish to reserve.
5. Following that, the Parser object constructs a new ReserveCommand object with the evaluated parameters.
6. Afterwards, the ReserveCommand is executed with the execute() function, which constructs a new Reservation based on the parameters provided in ReserveCommand.
7. The execute() function checks if an object is available for loan. If no Resource object is available, then an exception is thrown.
8. If any Resource is available for loan, then a new reservation object will be instantiated and added to the ReservationList object belonging to this Resource.
9. Lastly, the Ui object prints a message to the user to notify him/her of the resources that have been added into RIMS.

2.2.2. Design Considerations

Aspect 1: Selection of resource and quantity

- **Alternative 1 (Current choice)**: User select in terms of the following sequence - resource type, resource name, resource quantity. When each input is gathered, the ui class will feedback relevant information to help the user make decisions. In case of invalid input (such as invalid name), the parser will throw an exception and display an error message. Using this approach, users can only make one reservation for multiple resources of the same name at a time.
    - Advantages: More user friendly as user only requires one single command to make reservations for multiple resources of the same name.
    - Disadvantages: More difficult to catch exceptions. User also cannot select the exact resource they wish to borrow since resources are selected by a non-unique attribute.
- **Alternative 2 (Previous version)**: User select in terms of the following sequence - resource type, resource id. When parser starts gathering input, it will display a full list of all resources. User then select the resource to make a reservation for by entering a resource id. Using this approach, users can only make one reservation for one resource at a time.
    - Advantages: Easy to handle command and catch exceptions.
    - Disadvantages: Less user friendly in handling bulk reservation as user has to repeatedly enter the same command multiple times. Less user friendly when the resource list becomes long. Users then have to manually find a resource ID

Aspect 2: Selection of reservation dates

- **Alternative 1 (Current choice)**: User enters a single pair of start date followed by an end date. These pair of dates will be checked in the ReserveCommand class.
    - Advantages: -
    - Disadvantages: Users may need to key in repeated commands if they wish to make different reservations for a resource.

- **Alternative 2**: Users are able to enter a list of pairs of start date followed by an end date. These pair of dates will be checked in the ReserveCommand class
    - Advantages: More user friendly if users wish to make different reservations for a resource.
    - Disadvantages: -

2.3. Return

Implemented by: Bobby/Abhijit
Written by: Bobby/Sean

2.3.1. Implementation

Returning a resource utilizes the following classes: the Parser and UI class will gather inputs from users. Then, these inputs will be used to retrieve and delete the relevant reservation for the resource the user is returning.

For making a return, consider the case where a user wants to return a pen and record it in RIMS.

1. When RIMS is launched, a Ui and Parser Object is instantiated.
2. Referring to Figure 2.3.1, a parseInput() function is called, prompting the user for input. He/She enters the “return” keyword which is gathered by the Ui object.
3. The user is prompted to enter his/her user ID, and the Ui object will display a list of all Reservations made by the user.
4. The user then enters a the resources to return and the reservation ID corresponding to one of the reservations given in the list of Reservations displayed to be deleted from.
5. The Parser object constructs a new ReturnCommand object with the evaluated parameters.
6. The ReturnCommand is executed, which searches for the resources in the Reservation to be deleted. An exception is thrown if the Resources specified are not present in the Reservation, the reservation ID provided by the user is not valid, or the Resources are only reserved and not loaned out yet.
7. After the relevant Resources have been deleted from the Reservation, the Ui object sends a message to the user notifying him/her about the success return of the Resources.

2.3.2. Design Considerations

Aspect: How return is executed

- **Alternative 1 (Current choice)**: Choosing which Resource to return based on Reservation and User ID
    - Advantages: For multiple resources of the same instance borrowed, the user gets to decide which resource to return first (return a pen borrowed earlier, so as to return another pen due later).
    - Disadvantages: The user has to choose a Reservation rather than just specifying the Resources to return.

- **Alternative 2**: Choosing which Resource to return based on User ID only
  - Advantages: Less user input is required to return Resources.
  - Disadvantages: Returning Resources will take a longer period to search all Reservations, and the user has less flexibility in returning Resources.

2.4. List

Implemented by: Aarushi
Written by: Aarushi

Listing resources using RIMS is facilitated by the ListCommand, after receiving input from the user in the Ui and being passed through Parser. There are three different ways resources can be listed: all items listed, by room, or by item.

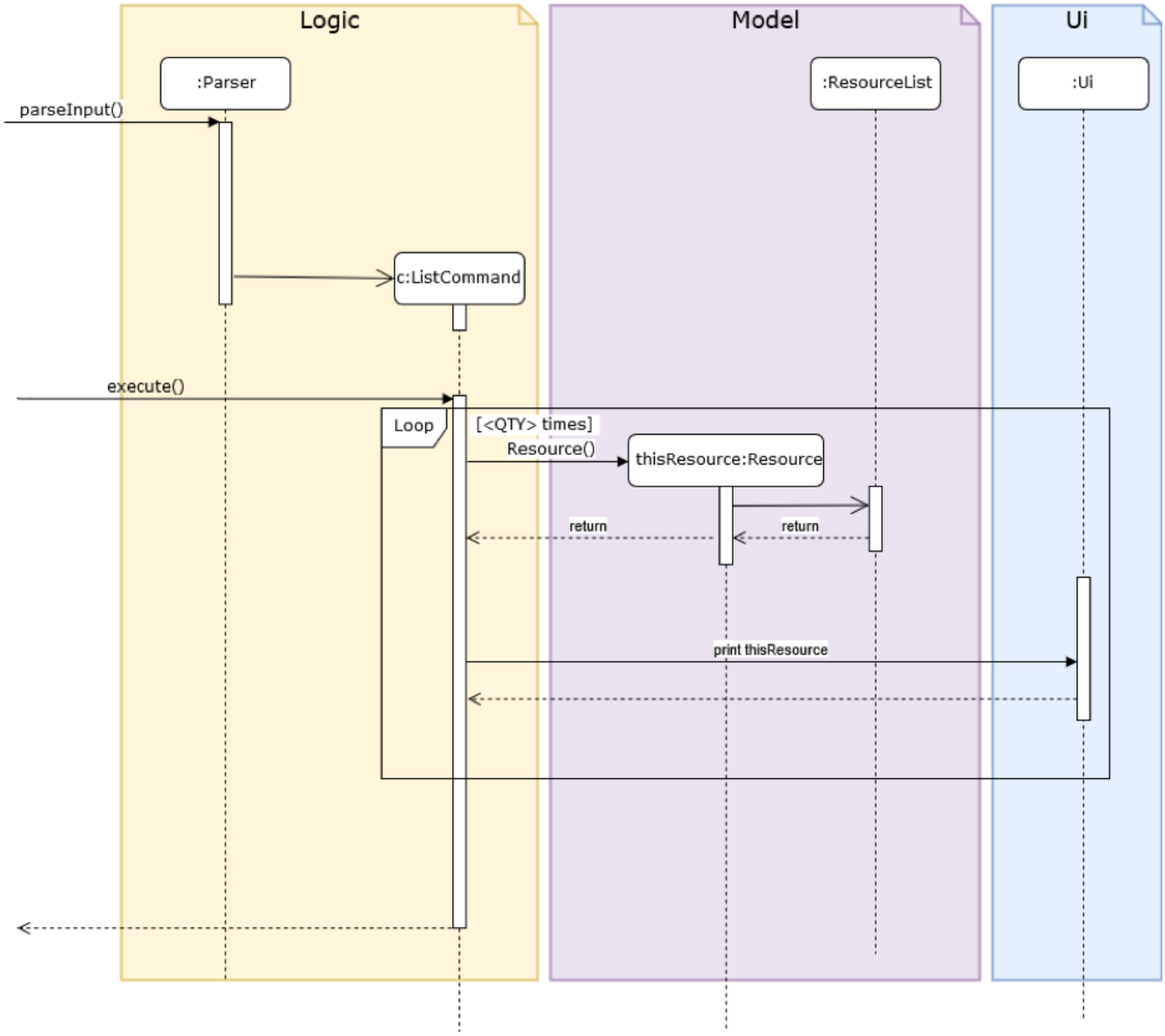

Figure 2.4.1: Sequence Diagram for Listing all Resources

For listing all resources(as shown in Figure 2.4.1), consider the case where the user wants a list of all the resources due in the inventory:

1. When starting up RIMS, a Ui and Parser object is instantiated respectively.
2. Referring to Figure 2.4.1, a parseInput() function is called, prompting for input from the user. He/She types the relevant input to the Ui object (getInput() function).
3. This input is received by the Parser object, which contains the relevant parameters.
4. Following that, the Parser object constructs a new ListCommand with the evaluated parameters.
5. Afterwards, the ListCommand is executed with the execute() function for QTY = resource.size(), which is the number of items in the ResourceList.
6. Lastly, the Ui object prints a list of all the resources in the ResourceList line by line within the loop.

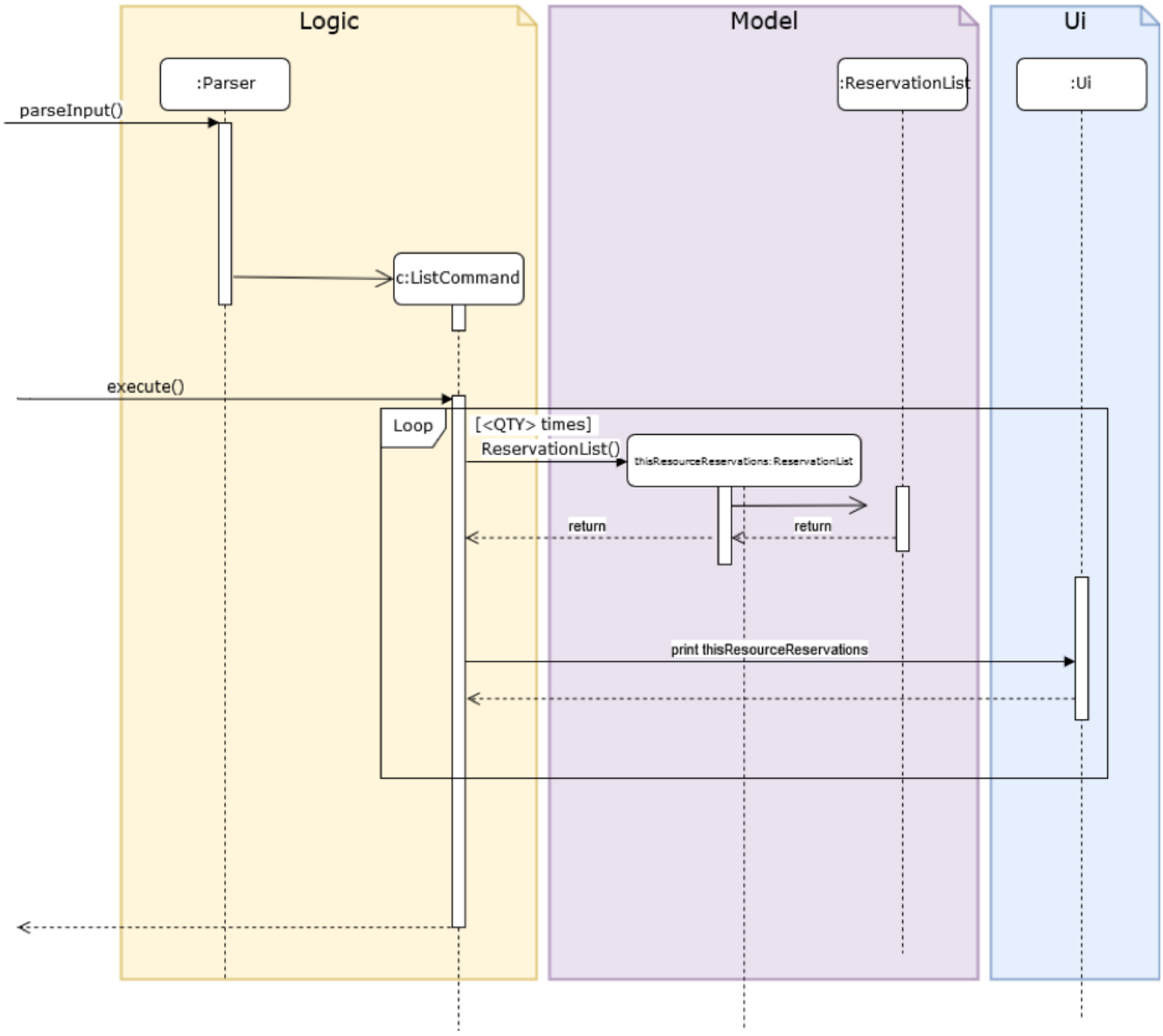

Figure 2.4.2: Sequence Diagram for Listing one type of Resource

For listing resources by item/room(as shown in Figure 2.4.2), consider the case where the user wants a list of all the dates a specific room/item is booked on:

1. When starting up RIMS, a Ui and Parser object is instantiated respectively.
2. Referring to Figure 2.4.2, a parseInput() function is called, prompting for input from the user. He/She types the relevant input to the Ui object (getInput() function).

3. This input is received by the Parser object, which contains the relevant parameters, such as whether the resource the user wants to check is an Item or a Room(the Parser object also prompts for new input from user should the format be incorrect or if the resource does not exist).
4. Following that, the Parser object constructs a new ListCommand with the evaluated parameters.
5. Afterwards, the ListCommand is executed with the execute() function for QTY times, which is the number of that particular item/room in the ReservationList.
6. Lastly, the Ui object prints a list of all the dates that resource is booked on in the ReservationList line by line within the loop.

2.5. Calendar

Implemented by: Daniel
Written by: Daniel

2.5.1. Implementation

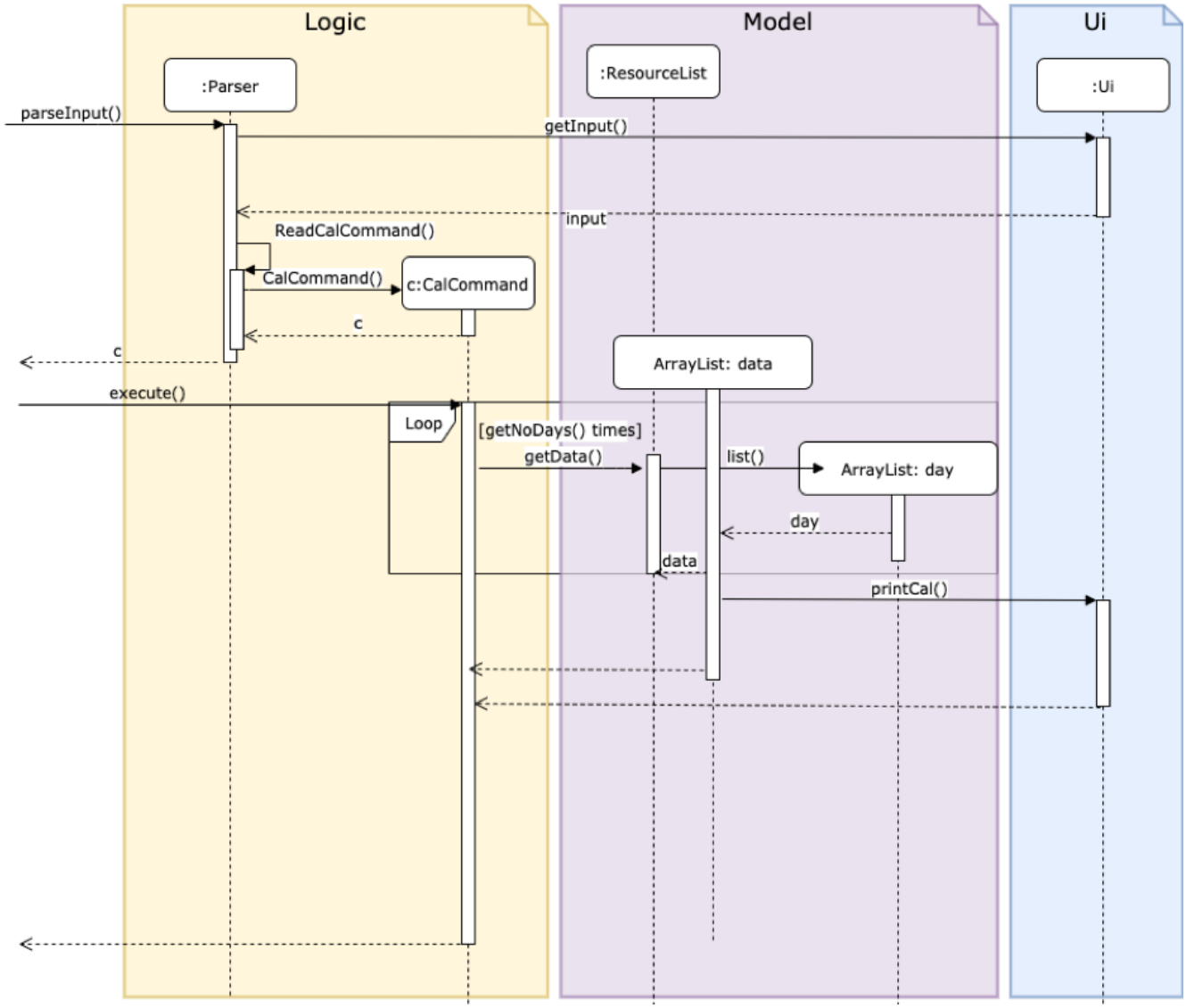

Figure 2.5: Sequence Diagram for Calendar

Viewing the resources on loan in a particular month, in the form of a calendar visualisation, is

achieved using CalendarCommand. It extends Command and stores the necessary attributes and methods necessary to print to screen, a visualisation of the data.

The user will input the command: cal in the UI. This is then parsed by Parser to invoke the constructor of CalendarCommand.

Given below is an example usage scenario and how CalendarCommand works:

1. When RIMS launches, a UI and Parser object is instantiated.
2. Referring to Figure 2.5, a parseInput() function is called, prompting the user to input a command. The user enters a desired command, such as cal, into the UI.
3. This input is received by the Parser and it will call the constructor in CalendarCommand.
4. CalendarCommand is executed where getData() and printCal() is called.
5. getData() will fetch data from ResourceList using the List() function. It will iterate through all the days of the current month and store it in an array.
6. printCal() will draw the grid with dates and entries using the data obtained from getData().

2.5.2. Design Considerations

Aspect: How entries is obtained

- **Alternative 1 (current choice)**: A 2-dimensional array is created to store all entries of the month to be called later.
    - Advantages: Performance is optimised as memoization will allow faster access of entries when the calendar is printed line by line in future
    - Disadvantages: Requires more space to store entries
- **Alternative 2**: The function list() is called every time an entry is printed
    - Advantages: Does not require as much space at the first alternative as the data is discarded after every row of a cell is printed
    - Disadvantages: Will result in taking more time to print out the whole calendar if the data is big. This is because list() has to be called for every line on every cell(day) of the calendar as opposed to calling list() once for every cell(day).

Appendix A: RIMS

A.1. Product Scope

A.1.1. Target User Profile

- needs to manage various facilities, items and resources efficiently
- needs to track loaning and borrowing of items and facilities
- wants to digitalise his inventory

- is comfortable using Command Line Interface (CLI) apps
- community centres
- school/tertiary institutions
- people who manage dorms – Office of Housing Services / hostels' facilities management

A.1.2. Value proposition

Manage facilities more quickly and efficiently by maintaining a digital ledger of his inventory

A.2. User Stories

Priorities: High (must have) - * * *, Medium (nice to have) - * *, Low (unlikely to have) - *

Priority	As a …	I want to …	So that I can…
* * *	logistics head	have a digital ledger of my assets	better track my assets
* * *	logistics manager	have a record of all items in my inventory	keep track of items the status of my items (whether rented, borrowed or reserved, location, date due, etc.)
* * *	facilities manager	have a record of all my rooms and facilities under my charge	better manage the people booking my facilities
* * *	logistics head	record items that have been lent out and the name of the person who borrowed the item	keep track of items that I have lent out and to whom
* * *	house/CSC/JCRC committee member	record the names of those who I have borrowed items from	keep track of the items I have borrowed and who I have borrowed those items from
* * *	house committee member	view if deadlines are approaching for me to return items	return items that I have borrowed promptly
* * *	CSC member	check which items that I lent out are overdue for return	remind users who have not returned my items to do so
* * *	logistics manager	add and delete items from my inventory	keep my digital ledger up-to-date with my actual inventory
* *	orientation committee member	record the names of those who have reserved items in advance from me	keep track of the items that have been reserved in advance and who they have been reserved for
* *	logistics committee member	view in a calendar format when an item/facility has been and when it will next be available	keep track of my items/facilities
* *	procurement executive	extend return-by dates for items I have lent out	allow others who still need the items to use them for longer

Priority	As a …	I want to …	So that I can…
* *	logistics head	extend return-by dates for items I have borrowed	allow others who still need the items to use them for longer
*	facilities manager	suggest alternatives to my clients if the item/facility they requested is already booked	still have business / be helpful
*	CSC member	access my items in a GUI	have a more user-friendly experience

{More to be added}

A.3. Use Cases

(For all use cases below, the **System** is the `RIMS` and the **Actor** is the `user`, unless specified otherwise)

Use case: Add asset

MSS

1. User enters command `add /item ITEM /qty QUANTITY` or `add /room ROOM /cap CAPACITY`.
2. RIMS appends asset to asset list.

   Use case ends.

Extensions

1a. `ITEM`/`ROOM` is empty.

   1a1. RIMS re-prompts for user to key in command again.

   Use case resumes at step 1.

1b. `QUANTITY`/`CAPACITY` is less than or equals to 0 or not of `int` type.

   1b1. RIMS re-prompts for user to key in command again.

   Use case resumes at step 1.

1c. Any keywords in command is misspelt or in wrong format.

   1c1. RIMS re-prompts for user to key in command in proper format.

   Use case resumes at step 1.

Use case: Add a borrowed asset (from third party)

MSS

1. User enters command `add /item ITEM /qty QUANTITY /by DEADLINE` or ` add /room ROOM /cap

CAPACITY /by DEADLINE`.

2. RIMS appends asset to asset list.

   Use case ends.

Extensions

1a. `ITEM`/`ROOM` is empty.

   1a1. RIMS re-prompts for user to key in command again.

   Use case resumes at step 1.

1b. `QUANTITY`/`CAPACITY` is less than or equals to 0 or not of `int` type.

   1b1. RIMS re-prompts for user to key in command again.

   Use case resumes at step 1.

1c. Any keywords in command is misspelt or in wrong format.

   1c1. RIMS re-prompts for user to key in command in proper format.

   Use case resumes at step 1.

1d. DEADLINE is empty or an invalid due date.

   1d1. RIMS displays error message and re-prompts user for valid due date.

   1d2. User enters a valid due date.

   Use case resumes at step 2.

Use case: Delete asset

MSS

1. User enters command `delete /item ITEM /qty QUANTITY` or `delete /room ROOM`.
2. RIMS removes the asset from asset list.

   Use case ends.

Extensions

1a. `ITEM`/`ROOM` is left unspecified by the user.

   Use case ends.

1b. `QUANTITY` is less than or equals to 0 or not of `int` type.

   1b1. RIMS re-prompts for user to key in command.

   Use case resumes at step 1.

1c. Any keywords in command is misspelt or in wrong format.

1c1. RIMS re-prompts for user to key in command in proper format.

Use case resumes at step 1.

2a. `ITEM`/`ROOM` not found in asset list.

2a1. RIMS re-prompts user to key in a valid asset.

2a2. User enters a valid asset name in the list.

Use case resumes at step 2

2b. `QUANTITY` is more than the number of assets available in asset list.

2b1. RIMS re-prompts for user to key in command.

Use case resumes at step 1.

Use case: Lend asset

MSS

1. User enters command `lend /item ITEM /qty QUANTITY /by DEADLINE` or `lend /room ROOM /by DEADLINE`
2. RIMS updates the status of the asset and appends asset to the 'on loan' list

Use case ends.

Extensions

1a. `ITEM`/`ROOM` is empty.

Use case ends.

1b. `DEADLINE` is invalid or empty.

1b1. RIMS displays error message and re-prompts user for valid due date.

1b2. User enters a valid due date

Use case resumes at step 2.

1c. `QUANTITY` is less than or equals to 0 or not of `int` type.

1c1. RIMS re-prompts for user to key in command.

Use case resumes at step 1.

1d. Any keywords in command is misspelt or in wrong format.

1d1. RIMS re-prompts for user to key in command in proper format.

Use case resumes at step 1.

2a. `ITEM`/`ROOM` not found in asset list.

2a1. RIMS re-prompts user to key in a valid asset.

2a2. User enters a valid asset name in the list.

Use case resumes at step 2.

2b. `QUANTITY` is more than the number of assets available in asset list.

2b1. RIMS re-prompts for user to key in command.

Use case resumes at step 1.

Use case: Return asset

MSS

1. User enters command `return /item ITEM /qty QUANTITY` or `return /room ROOM`
2. RIMS updates the status of the asset and removes the asset from the 'on loan' list

Use case ends.

Extensions

1a. `ITEM`/`ROOM` is empty.

Use case ends.

1b. `QUANTITY` is less than or equals to 0 or not of `int` type.

1b1. RIMS re-prompts for user to key in command.

Use case resumes at step 1.

1c. Any keywords in command is misspelt or in wrong format.

1c1. RIMS re-prompts for user to key in command in proper format.

Use case resumes at step 1.

2a. `ITEM`/`ROOM` not found in asset list.

2a1. RIMS re-prompts user to key in a valid asset.

2a2. User enters a valid asset name in the list.

Use case resumes at step 2.

`NEW` 2b. `QUANTITY` is more than the number of assets borrowed.

2b1. RIMS re-prompts for user to key in command.

Use case resumes at step 1.

`NEW` 2c. `ITEM`/`ROOM` not found in 'on loan' list.

2c1. RIMS re-prompts for user to key in command.

Use case resumes at step 1.

Use case: Reserve asset

MSS

1. User enters command `reserve /item ITEM /qty QUANTITY /from DATE /to DEADLINE` or `reserve /room ROOM /from DATE /to DEADLINE`.
2. RIMS updates the status of the asset and appends asset to the 'reserved' list.

   Use case ends.

Extensions

1a. `ITEM`/`ROOM` is empty.

Use case ends.

1b. `QUANTITY` is less than or equals to 0 or not of `int` type.

1b1. RIMS re-prompts for user to key in command.

Use case resumes at step 1.

1c. `DATE` is invalid or empty.

1c1. RIMS displays error message and re-prompts user for valid start date.

1c2. User enters a valid start date.

Use case resumes at step 2.

1d. `DEADLINE` is invalid or empty.

1d1. RIMS displays error message and re-prompts user for valid due date.

1d2. User enters a valid due date.

Use case resumes at step 2.

1e. Any keywords in command is misspelt or in wrong format.

1e1. RIMS re-prompts for user to key in command in proper format.

Use case resumes at step 1.

2a. `ITEM`/`ROOM` not found in asset list.

2a1. RIMS re-prompts user to key in a valid asset.

2a2. User enters a valid asset name in the list.

Use case resumes at step 2.

2b. `QUANTITY` is more than the number of assets available in asset list.

2b1. RIMS re-prompts for user to key in command.

Use case resumes at step 1.

`NEW` 2c. `ITEM`/`ROOM` not found in 'on loan' list.

2c1. RIMS re-prompts for user to key in command.

Use case resumes at step 1.

Use case: Extend existing loan

MSS

1. User enters command `extend /item ITEM /qty QUANTITY /by DEADLINE` or `extend /room ROOM /by DEADLINE`.
2. RIMS updates the status of the asset and updates asset on the ‘on-loan’ list.

Use case ends.

Extensions

1a. `ITEM`/`ROOM` is empty.

Use case ends.

1b. `QUANTITY` is less than or equals to 0 or not of `int` type.

1b1. RIMS re-prompts for user to key in command.

Use case resumes at step 1.

Use case resumes at step 2.

1c. `DEADLINE` is invalid or empty.

1c1. RIMS displays error message and re-prompts user for valid due date.

1c2. User enters a valid due date.

Use case resumes at step 2.

1d. Any keywords in command is misspelt or in wrong format.

1d1. RIMS re-prompts for user to key in command in proper format.

Use case resumes at step 1.

2a. `ITEM`/`ROOM` not found in asset list.

2a1. RIMS re-prompts user to key in a valid asset.

2a2. User enters a valid asset name in the list.

Use case resumes at step 2.

2b. `ITEM`/`ROOM` not available for extension.

2b1. RIMS re-prompts for user to key in command.

Use case resumes at step 1.

2c. `DEADLINE` is not available for extension due to reservation.

2c1. RIMS displays message that asset has been reserved.

Use case ends.

Use case: View assets on loan with a specified deadine

MSS

1. User enters command `due DATE`.
2. RIMS displays assets due for return on specified date.

Use case ends.

Extensions

- 1a. `DATE` is invalid or empty.

1a1. RIMS displays error message and re-prompts user for valid date.

1a2. User enters a valid date.

Use case resumes at step 2.

1b. Command `due` is misspelt.

1b1. RIMS displays error message.

Use case ends.

2a. Asset list is empty.

2a1. RIMS informs user that no assets are owned at the moment.

Use case ends.

Use case: List assets and their statuses

MSS

1. User enters command `list`.
2. RIMS displays all assets on asset list and their statuses.

Use case ends.

Extensions

1a. Command `list` is misspelt.

1a1. RIMS displays error message.

Use case ends.

2a. Asset list is empty.

2a1. RIMS informs user that no assets are owned at the moment.

Use case ends.

Use case: View calendar

MSS

1. User enters command `calendar`.
2. RIMS displays all assets that are loaned out/reserved in a table form.

Use case ends.

Extensions

1a. Command `calendar` is misspelt.

1a1. RIMS displays error message.

Use case ends.

2a. Asset list is empty.

2a1. RIMS informs user that no assets are owned at the moment.

Use case ends.

Use case: Exit RIMS

MSS

1. User enters command `bye`.
2. RIMS saves and terminates.

Use case ends.

Extensions

1a. Command `bye` is misspelt.

1a1. RIMS displays error message.

Use case ends.

{More to be added}

A.4. Non Functional Requirements

1. **Computer environment**: should work on any mainstream OS as long as it has Java 11 or above installed.
2. **Persistence**: the total logistics list should be retained upon termination of RIMS.
3. **Ease of use**: Program functions should be easy and convenient to use.
4. **Quality requirement**: The system should be efficient enough for the user to quickly update and keep track of the facilities and items under their care.
5. **Speed**: Program should execute commands efficiently and without lag.

A.5. Glossary

Inventory

The complete collection of items, rooms and resources as maintained by the user.

Item

An object in the user's inventory, that is not a room or facility, that can be lent out to others. There can be multiple instances of a certain item in a user's inventory.

Facility

A room under the user's purview. Rooms are uniquely defined and there cannot be multiple instances of a certain room.

Deadline

The specified date and time by which an object that has been lent out by the user, must be returned to the user.

Mainstream OS

Windows, Linux, Unix, OS-X