Part 1 - Assumptions

Assumptions Regarding the Problem Context

1. Ensemble Structure

- Each ensemble must have unique ensemble ID (String type)
- Ensemble can have a name that can be modified after creation
- Two concrete ensemble types exist: OrchestraEnsemble and JazzBandEnsemble
- OrchestraEnsemble support two roles: Violinist (role=1) and Cellist (role=2)
- JazzBandEnsemble support three roles: Pianist (role=1), Saxophonist (role=2), Drummer (role=3)
- Ensemble use LinkedList to store musicians internally
- Abstract class Ensemble define common behavior for all ensemble types
- System don't enforce unique ensemble ID globally, assume user provide unique ID

2. Musician Structure

- Each musician have unique musician ID (String type) within same ensemble
- Musician have a name (String) and role (int value)
- Role value represent the instrument musician play in the ensemble
- When musician move to different ensemble type, invalid roles are reset to 0
- Musician object store in ensemble's musician list
- If add musician with duplicate ID to same ensemble, old musician is removed first then new one added

3. Command Execution

- All operations implement Command interface with execute() and undo() methods
- Commands that modify data (create, add, delete, change) are executed through EnsembleService and stored in history
- Query commands (show, display) and control commands (undo, redo) execute directly in Main class
- Query commands still implement Command interface but undo() is empty (no-op)
- Each command provide description for display in history list

4. Current Ensemble Concept

- System maintain single "current ensemble" reference at any time
- Operations like add musician, delete musician work on current ensemble only
- User can switch current ensemble using set current ensemble command
- If current ensemble become null, operations that need it should show error
- When new ensemble created, Main class set it as current ensemble (not automatic by EnsembleService)

5. Undo/Redo Mechanism

- EnsembleService maintain two stacks: history stack and undoneCommands stack
- Only modification commands (create, add, delete, change name, change instrument) push to history stack
- When command execute through manager, it push to history stack
- When undo occur, command pop from history and push to undone stack

- When new modification command execute, undone stack is cleared
- Commands store necessary information (like previous state) to support undo

6. State Preservation

- Before modify musician role, save old role using MusicianState memento
- Before modify ensemble name, save old name using EnsembleState memento
- Memento objects store reference to original object and old values
- Restore operation apply saved values back to object
- Only ChangeInstrumentCmd and ChangeNameCmd use memento pattern
- Creation and deletion commands don't need memento (they restore by reverse operation)

7. Command Creation

- CommandParser factory class responsible for create all command objects
- Factory method take user input and construct appropriate command
- Each command type need different parameters from user input
- Factory also handle user input collection using Scanner (factory have dual responsibility)
- Main class don't directly instantiate command classes
- Factory use switch statement to determine which command to create

8. Collection Usage

- Ensemble use AbstractList (implemented as LinkedList)
- EnsembleService use LinkedList for storing ensembles
- Iterator pattern used to traverse musician list without exposing internal structure
- When add musician with duplicate ID, old musician is removed first then new one added
- Order of musicians preserved in insertion order

9. Polymorphism

- Different ensemble types override updateMusicianRole() with specific logic
- Different ensemble types override showEnsemble() with specific display format
- Commands implement common Command interface but have different execute logic
- Factory return Command interface type, actual type determined at runtime

10. Input/Output Assumptions

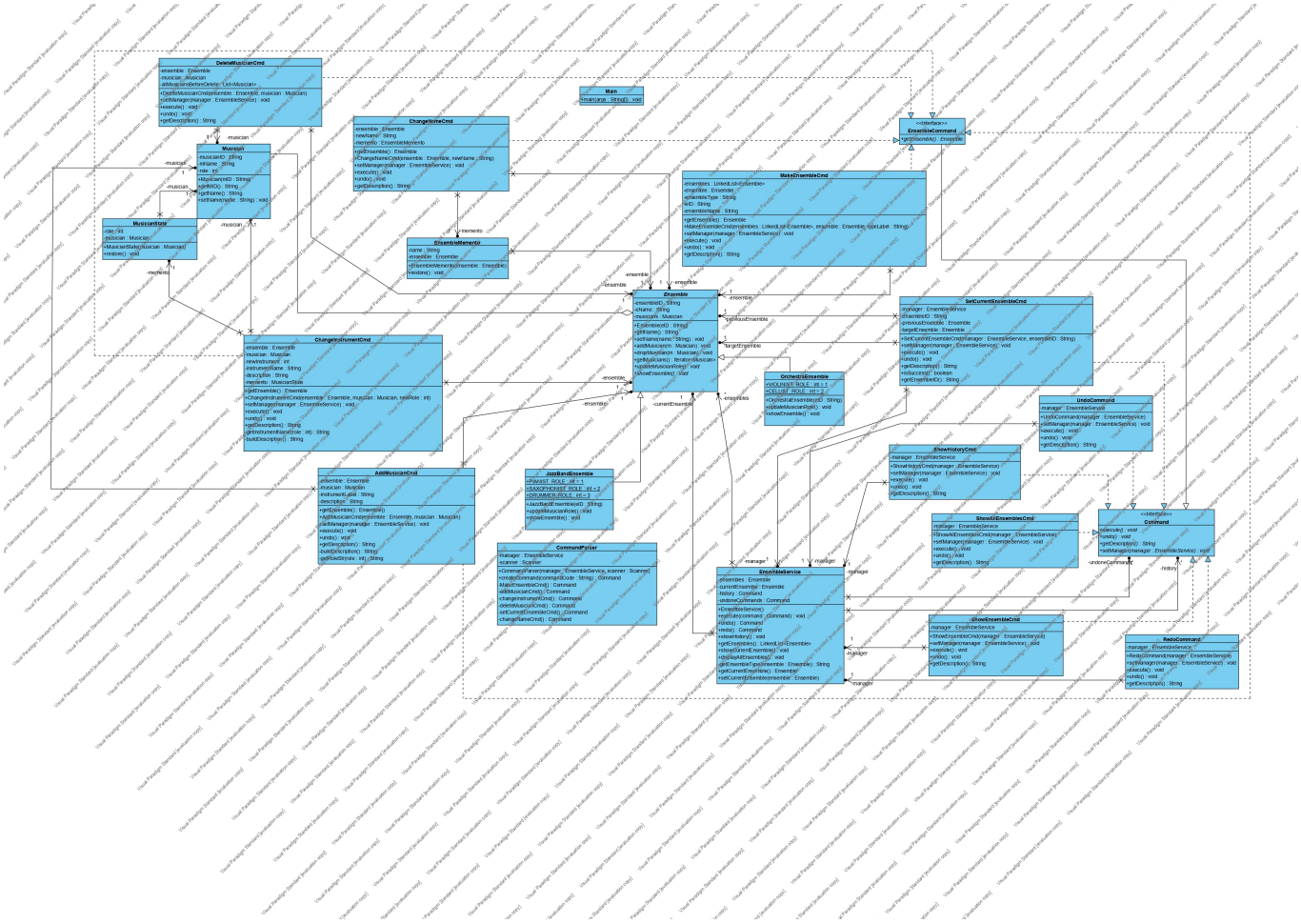
- Musician input format is "id, name" (comma and space separator)
- Instrument/role input is integer value (1, 2, or 3 depend on ensemble type)
- Invalid input should show error message and not execute command
- System output follow specific format for ensemble and musician display
- All interaction through console (System.out and Scanner)

Part 2 - Application Design

System Architecture

The Music Ensembles Management System (MEMS) is design using object-oriented principles with several design patterns to make code more flexible and maintainable.

Class Diagram



Main Components

1. Main Application Layer

- **Main.java**: Entry point of program, handle user interface and main loop
- **EnsembleService.java**: Central manager that coordinate all operations and maintain system state

2. Ensemble Layer

- **Ensemble.java**: Abstract base class for all ensemble types
- **OrchestraEnsemble.java**: Concrete class for orchestra ensemble with violinist and cellist
- **JazzBandEnsemble.java**: Concrete class for jazz band with pianist, saxophonist, and drummer

3. Musician Layer

- **Musician.java**: Represent individual musician with ID, name and role/instrument

4. Command Layer

- **Command.java**: Interface define command operations
- **EnsembleCommand.java**: Interface for commands that operate on ensemble
- **CommandParser.java**: Factory class create command objects from user input

- Concrete command classes:
    - **MakeEnsembleCmd.java**: Create new ensemble
    - **AddMusicianCmd.java**: Add musician to ensemble
    - **ChangeInstrumentCmd.java**: Change musician instrument
    - **ChangeNameCmd.java**: Change ensemble name
    - **DeleteMusicianCmd.java**: Remove musician from ensemble
    - **SetCurrentEnsembleCmd.java**: Switch current ensemble
    - **ShowEnsembleCmd.java**: Display current ensemble
    - **ShowAllEnsemblesCmd.java**: Display all ensembles
    - **ShowHistoryCmd.java**: Show undo/redo history
    - **UndoCommand.java**: Undo last operation
    - **RedoCommand.java**: Redo undone operation

5. State Management Layer

- **EnsembleState.java**: Store ensemble state for memento pattern
- **MusicianState.java**: Store musician state for memento pattern

Part 3 - Design Patterns

Discussion and Explanation on Design Patterns

This section discuss and explain each of the design patterns applied to this Music Ensembles Management System.

---

1. Command Pattern

What is Command Pattern

Command Pattern is behavioral design pattern that encapsulate a request as an object. This pattern turn operations into stand-alone objects that contain all information about the operation.

Implementation in MEMS

Command Interface:

```
public interface Command {
    void execute();
    void undo();
    String getDescription();
    void setManager(EnsembleService manager);
}
```

The system implement 11 concrete command classes:

- `MakeEnsembleCmd`, `AddMusicianCmd`, `DeleteMusicianCmd`, `ChangeInstrumentCmd`, `ChangeNameCmd`
- `SetCurrentEnsembleCmd`, `ShowEnsembleCmd`, `ShowAllEnsemblesCmd`, `ShowHistoryCmd`

- `UndoCommand`, `RedoCommand`

Some commands also implement `EnsembleCommand` interface which extend Command and add getEnsemble() method.

Invoker (EnsembleService):

```
public class EnsembleService {
    private final Stack<Command> history;
    private final Stack<Command> undoneCommands;

    public void execute(Command command) {
        undoneCommands.clear();
        command.execute();
        history.push(command);
    }
}
```

EnsembleService maintain two stacks for undo/redo functionality.

How It Work

1. CommandParser create appropriate Command object from user input
2. For modification commands (create, add, delete, change), Main call manager.execute() which store command in history
3. For query commands (se, sa, l) and control commands (u, r), Main call command.execute() directly without storing in history
4. When undo is called, command pop from history and execute undo()
5. Undone commands can be redone

This design allow modification commands to be undone while query commands don't affect undo/redo history.

Benefits

- Each operation is separate class with own logic
- Easy to implement undo/redo functionality for modification commands
- New commands can add but require modify CommandParser switch statement
- Main class don't need know implementation details of each operation

---

2. Factory Pattern

What is Factory Pattern

Factory Pattern is creational design pattern that provide interface for creating objects without specify their exact classes.

Implementation in MEMS

CommandParser Factory:

```
public class CommandParser {
    private final EnsembleService manager;
    private final Scanner scanner;

    public Command createCommand(String commandCode) {
        switch (commandCode) {
            case "c": return makeEnsembleCmd();
            case "a": return addMusicianCmd();
            case "m": return changeInstrumentCmd();
            case "d": return deleteMusicianCmd();
            // ... other cases
            default: return null;
        }
    }
}
```

Factory have helper methods to create complex commands and collect user input. This is simple factory pattern implementation where factory handle both object creation and user input collection.

How It Work

1. Main class receive command code from user
2. CommandParser's createCommand() is called
3. Factory use switch statement to determine which command to create
4. Factory collect necessary parameters from user (using Scanner)
5. Factory return ready-to-execute Command object

Benefits and Limitations

Benefits:

- All object creation logic in one place
- Main class don't need know how to create each command
- User input handling separated from Main class

Limitations:

- Factory use switch statement, so adding new command require modify factory code (not fully follow Open-Closed Principle)
- Factory have dual responsibility: create objects and collect user input (coupling UI and creation logic)
- This is acceptable for assignment simplification but can be improved by separate concerns

---

3. Memento Pattern

What is Memento Pattern

Memento Pattern is behavioral design pattern that let you save and restore previous state of an object without reveal implementation details.

Implementation in MEMS

Memento Classes:

```
public class MusicianState {
    private final int role;
    private final Musician musician;

    public MusicianState(Musician musician) {
        this.musician = musician;
        this.role = musician.getRole();
    }

    public void restore() {
        musician.setRole(role);
    }
}
```

```
public class EnsembleState {
    private final String name;
    private final Ensemble ensemble;

    public EnsembleState(Ensemble ensemble) {
        this.ensemble = ensemble;
        this.name = ensemble.getName();
    }

    public void restore() {
        ensemble.setName(name);
    }
}
```

Usage in Commands:

```
public class ChangeInstrumentCmd implements EnsembleCommand {
    private MusicianState memento;

    public void execute() {
        memento = new MusicianState(musician);  // save state
        musician.setRole(newInstrument);         // change state
    }

    public void undo() {
        memento.restore();  // restore old state
    }
}
```

How It Work

1. Before modify object, command create memento to save current state

2. Memento store reference to original object and old values
3. Command execute and change object state
4. When undo called, memento restore old state

Benefits

- Object internal state remain private
- Easy to restore previous state
- Undo logic separate from business logic
- Don't need expose object internals

How Patterns Work Together

The three patterns work together in this workflow:

For Modification Commands (create, add, delete, change):

```
User Input → Factory create Command → Command save state with Memento (if needed)
          → Main call manager.execute(command)
          → Command execute and store in history stack
          → User type "u" → Undo pop command from history
          → Command use Memento to restore old state (for change operations)
```

For Query Commands (show ensemble, show all, list history):

```
User Input → Factory create Command → Main call command.execute() directly
          → Command execute (no history storage)
          → Command undo() is empty (no-op)
```

For Control Commands (undo, redo):

```
User Input "u" or "r" → Factory create UndoCommand or RedoCommand
                     → Main call command.execute() directly
                     → Command manipulate history stacks in EnsembleService
```

Summary:

Pattern	Role	Key Classes
Command	Encapsulate operations, enable undo/redo for modification commands	Command interface, EnsembleCommand interface, 11 concrete commands, EnsembleService
Factory	Create command objects and collect user input	CommandParser

Pattern	Role	Key Classes
Memento	Save and restore state for change operations	MusicianState, EnsembleState

Key Points:

- Not all commands go through EnsembleService.execute() - only modification commands do
- Query and control commands execute directly but still implement Command interface
- Factory use switch statement which require modification when add new command type
- Factory handle both object creation and user input (dual responsibility for simplification)

These three design patterns was applied according to assignment requirements. They solve specific problems: Factory solve object creation, Command solve undo/redo for modifications, Memento solve state preservation for change operations.

Part 4 - Test Cases

Test Cases Overview

This document contains test cases covering all system commands including create, modify, delete, query operations, and Undo/Redo functionality.

Test Case 1: Create Orchestra Ensemble

Input:

```
c
o
E001
Vienna Orchestra
```

Expected Output:

```
Orchestra ensemble is created.
Current ensemble is changed to E001.
```

Explanation: Create an orchestra ensemble with ID E001 and name Vienna Orchestra

Screenshot:

```
Music Ensembles Management System (MEMS)
c = create ensemble, s = set current ensemble, a = add musician, m = modify musician's instrument,
d = delete musician, se = show ensemble, sa = display all ensembles, cn = change ensemble's name,
u = undo, r = redo, l = list undo/redo, x = exit system
No current ensemble set.
Please enter command [ c | s | a | m | d | se | sa | cn | u | r | l | x ] :- c
Enter music type (o = orchestra | j = jazz band) :- o
Ensemble ID:- E001
Ensemble Name:- Vienna Orchestra
Orchestra ensemble is created.
Current ensemble is changed to E001.
```

Test Case 2: Create Jazz Band Ensemble

Input:

```
c
j
J001
Blue Note Band
```

Expected Output:

```
Jazz band ensemble is created.
Current ensemble is changed to J001.
```

Explanation: Create a jazz band ensemble with ID J001 and name Blue Note Band

Screenshot:

```
Music Ensembles Management System (MEMS)
c = create ensemble, s = set current ensemble, a = add musician, m = modify musician's instrument,
d = delete musician, se = show ensemble, sa = display all ensembles, cn = change ensemble's name,
u = undo, r = redo, l = list undo/redo, x = exit system
The current ensemble is E001 Vienna Orchestra.
Please enter command [ c | s | a | m | d | se | sa | cn | u | r | l | x ] :- c
Enter music type (o = orchestra | j = jazz band) :- j
Ensemble ID:- J001
Ensemble Name:- Blue Note Band
Jazz band ensemble is created.
Current ensemble is changed to J001.
```

Test Case 3: Display All Ensembles

Input:

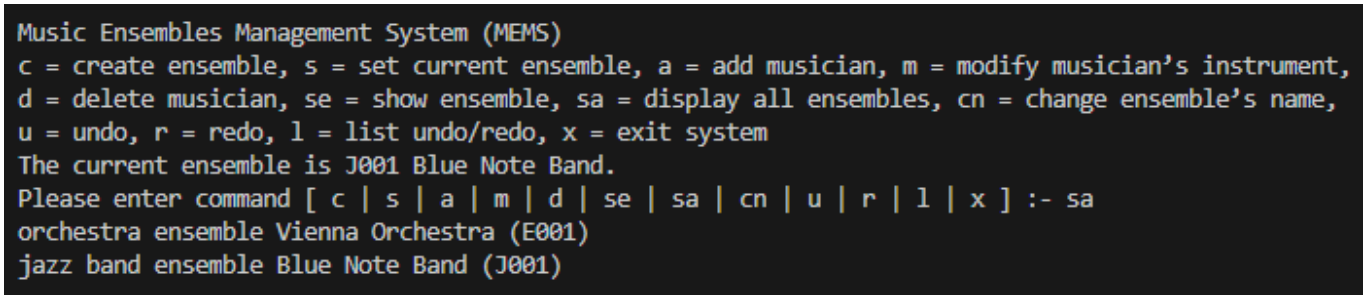
```
sa
```

Expected Output:

```
orchestra ensemble Vienna Orchestra (E001)
jazz band ensemble Blue Note Band (J001)
```

Explanation: List all ensembles in the system

Screenshot:



Test Case 4: Set Current Ensemble

Input:

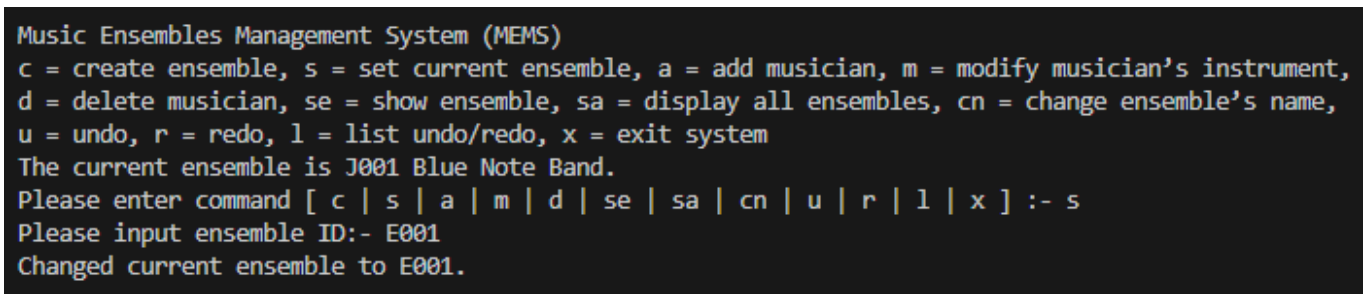
```
s
E001
```

Expected Output:

```
Changed current ensemble to E001.
```

Explanation: Switch current working ensemble to E001

Screenshot:



Test Case 5: Add Musician to Orchestra

Precondition: Current ensemble is E001 (Orchestra)

Input:

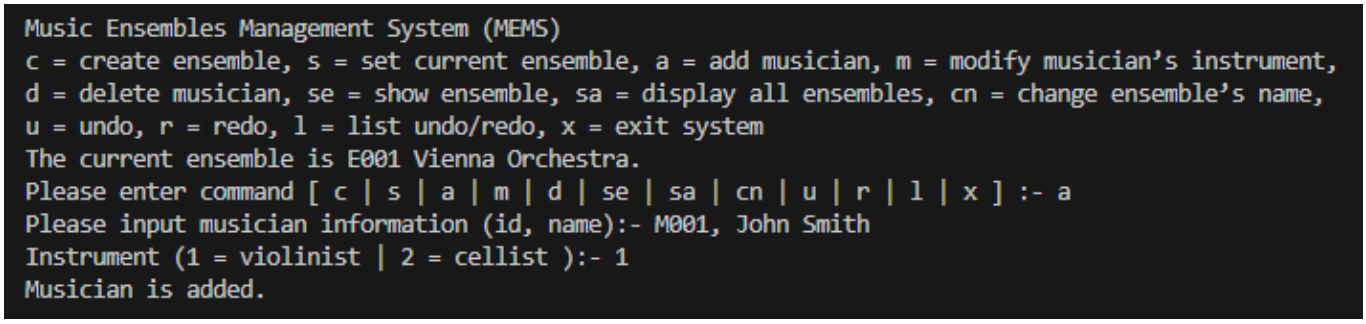
```
a
M001, John Smith
1
```

Expected Output:

```
Musician is added.
```

Explanation: Add violinist John Smith (ID: M001) to orchestra, role code 1 represents violinist

Screenshot:



Test Case 6: Add Musician to Jazz Band

Precondition: Current ensemble is J001 (Jazz Band)

Input:

```
s
J001
a
M002, Mary Johnson
1
```

Expected Output:

```
Changed current ensemble to J001.
Musician is added.
```

Explanation: Add saxophonist Mary Johnson (ID: M002) to jazz band, role code 1 represents pianist

Screenshot:

```
Music Ensembles Management System (MEMS)
c = create ensemble, s = set current ensemble, a = add musician, m = modify musician's instrument,
d = delete musician, se = show ensemble, sa = display all ensembles, cn = change ensemble's name,
u = undo, r = redo, l = list undo/redo, x = exit system
The current ensemble is E001 Vienna Orchestra.
Please enter command [ c | s | a | m | d | se | sa | cn | u | r | l | x ] :- s
Please input ensemble ID:- J001
Changed current ensemble to J001.

Music Ensembles Management System (MEMS)
c = create ensemble, s = set current ensemble, a = add musician, m = modify musician's instrument,
d = delete musician, se = show ensemble, sa = display all ensembles, cn = change ensemble's name,
u = undo, r = redo, l = list undo/redo, x = exit system
The current ensemble is J001 Blue Note Band.
Please enter command [ c | s | a | m | d | se | sa | cn | u | r | l | x ] :- a
Please input musician information (id, name):- M002, Mary Johnson
Instrument (1 = pianist | 2 = saxophonist | 3 = drummer):- 1
Musician is added.
```

Test Case 7: Show Ensemble Details

Precondition: Current ensemble has musicians

Input:

```
se
```

Expected Output:

```
Jazz Band Ensemble Blue Note Band (J001)
Pianist:
M002, Mary Johnson
Saxophonist:
NIL
Drummer:
NIL
```

Explanation: Display current ensemble details including all musicians and their roles

Screenshot:

```
Music Ensembles Management System (MEMS)
c = create ensemble, s = set current ensemble, a = add musician, m = modify musician's instrument,
d = delete musician, se = show ensemble, sa = display all ensembles, cn = change ensemble's name,
u = undo, r = redo, l = list undo/redo, x = exit system
The current ensemble is J001 Blue Note Band.
Please enter command [ c | s | a | m | d | se | sa | cn | u | r | l | x ] :- se
Jazz Band Ensemble Blue Note Band (J001)
Pianist:
M002, Mary Johnson
Saxophonist:
NIL
Drummer:
NIL
```

Test Case 8: Modify Musician's Instrument

Precondition: Current ensemble is J001, has musician M002

Input:

```
m
M002
2
```

Expected Output:

```
instrument is updated.
```

Explanation: Change musician M002's instrument from pianist to saxophonist (role code 2)

Screenshot:

```
Music Ensembles Management System (MEMS)
c = create ensemble, s = set current ensemble, a = add musician, m = modify musician's instrument,
d = delete musician, se = show ensemble, sa = display all ensembles, cn = change ensemble's name,
u = undo, r = redo, l = list undo/redo, x = exit system
The current ensemble is J001 Blue Note Band.
Please enter command [ c | s | a | m | d | se | sa | cn | u | r | l | x ] :- m
Please input musician ID:- M002
Instrument (1 = pianist | 2 = saxophonist | 3 = drummer):- 2
instrument is updated.
```

Test Case 9: Change Ensemble Name

Precondition: Current ensemble is J001

Input:

```
cn
Blue Note Jazz Ensemble
```

Expected Output:

```
Ensemble's name is updated.
```

Explanation: Change ensemble J001's name from "Blue Note Band" to "Blue Note Jazz Ensemble"

Screenshot:

```
Music Ensembles Management System (MEMS)
c = create ensemble, s = set current ensemble, a = add musician, m = modify musician's instrument,
d = delete musician, se = show ensemble, sa = display all ensembles, cn = change ensemble's name,
u = undo, r = redo, l = list undo/redo, x = exit system
The current ensemble is J001 Blue Note Band.
Please enter command [ c | s | a | m | d | se | sa | cn | u | r | l | x ] :- cn
Please input new name of the current ensemble:- Blue Note Jazz
Ensemble's name is updated.
```

Test Case 10: Delete Musician

Precondition: Current ensemble has musician M002

Input:

```
d
M002
```

Expected Output:

```
Musician is deleted.
```

Explanation: Delete musician with ID M002 from current ensemble

Screenshot:

```
Music Ensembles Management System (MEMS)
c = create ensemble, s = set current ensemble, a = add musician, m = modify musician's instrument,
d = delete musician, se = show ensemble, sa = display all ensembles, cn = change ensemble's name,
u = undo, r = redo, l = list undo/redo, x = exit system
The current ensemble is J001 Blue Note Jazz.
Please enter command [ c | s | a | m | d | se | sa | cn | u | r | l | x ] :- d
Please input musician ID:- M002
Musician is deleted.
```

Test Case 11: List Undo/Redo History

Input:

```
l
```

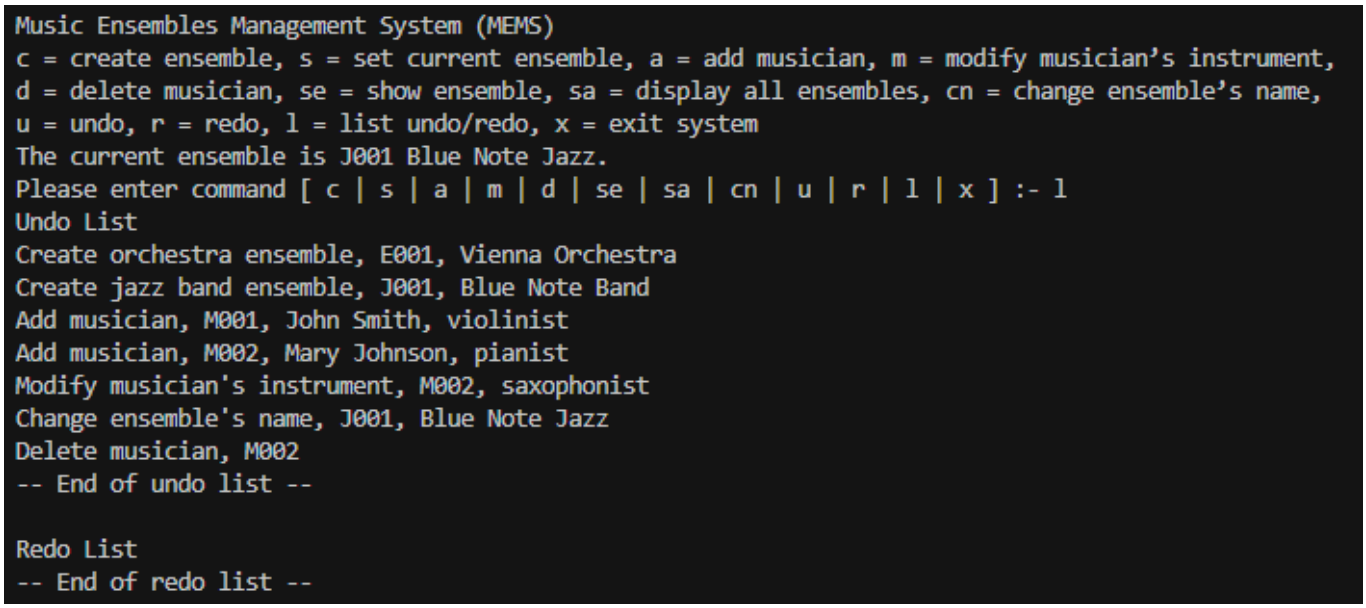
Expected Output:

```
Undo List
Create orchestra ensemble, E001, Vienna Orchestra
Create jazz band ensemble, J001, Blue Note Band
Add musician, M001, John Smith, violinist
Add musician, M002, Mary Johnson, pianist
Modify musician's instrument, M002, saxophonist
Change ensemble's name, J001, Blue Note Jazz
Delete musician, M002
-- End of undo list --

Redo List
-- End of redo list --
```

Explanation: Display all undoable and redoable commands

Screenshot:



Test Case 12: Undo Command

Precondition: At least one undoable command executed

Input:

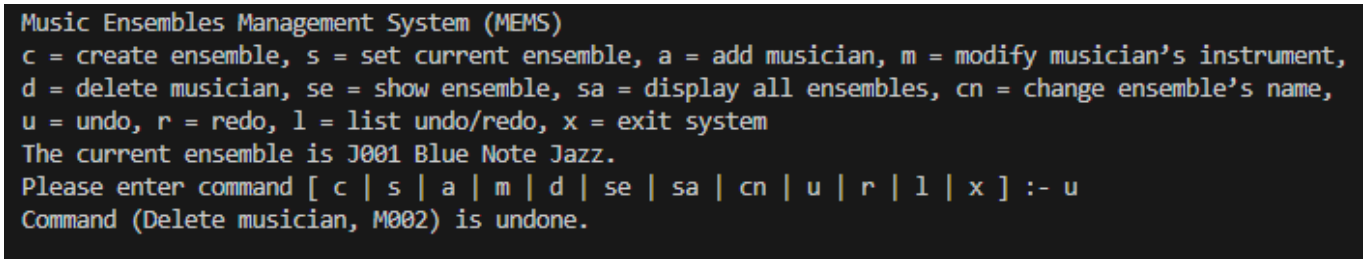
```
u
```

Expected Output:

```
Command (Delete musician, M002) is undone.
```

Explanation: Undo the last executed command (delete musician)

Screenshot:



Test Case 13: Redo Command

Precondition: Undo has been executed

Input:

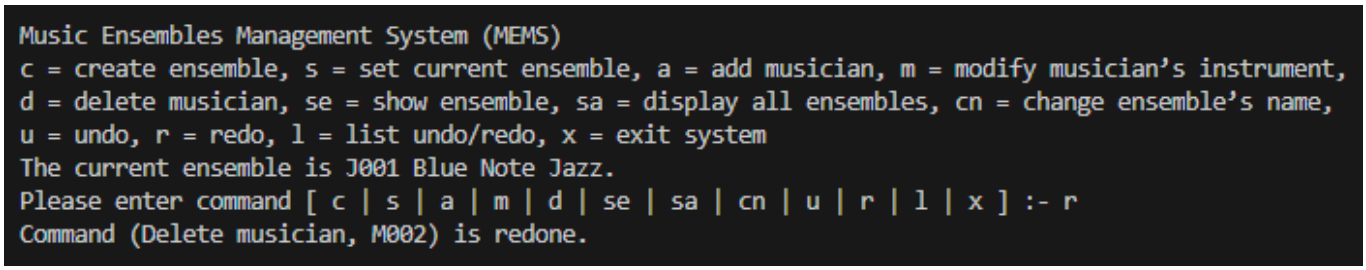
```
r
```

Expected Output:

```
Command (Delete musician, M002) is redone.
```

Explanation: Redo the previously undone command

Screenshot:



Test Case 14: Multiple Undo Operations

Input:

```
u
u
```

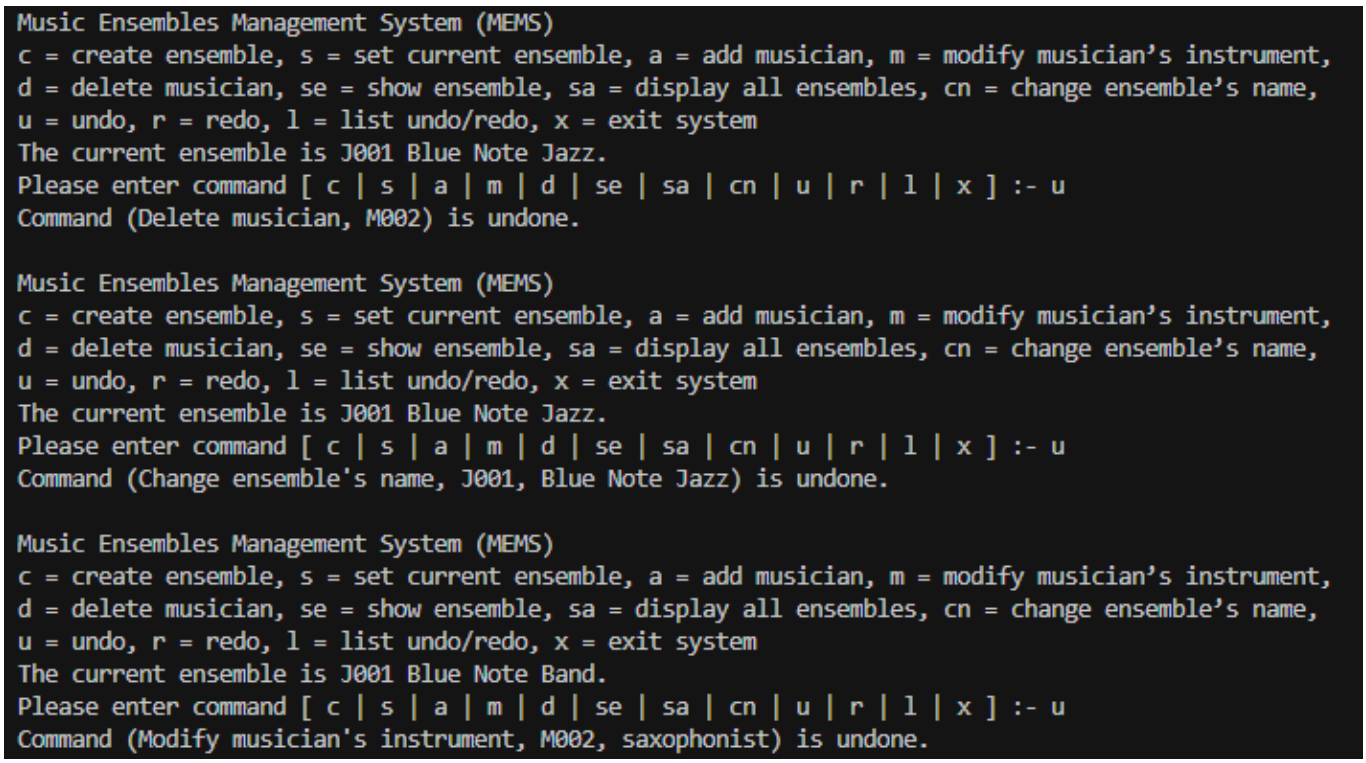
```
u
```

Expected Output:

```
Command (Delete musician, M002) is undone.
Command (Change ensemble's name, J001, Blue Note Jazz) is undone.
Command (Modify musician's instrument, M002, saxophonist) is undone.
```

Explanation: Undo multiple operations consecutively

Screenshot:



Test Case 15: Error Handling - Add Musician Without Current Ensemble

Precondition: System just started, no current ensemble set

Input:

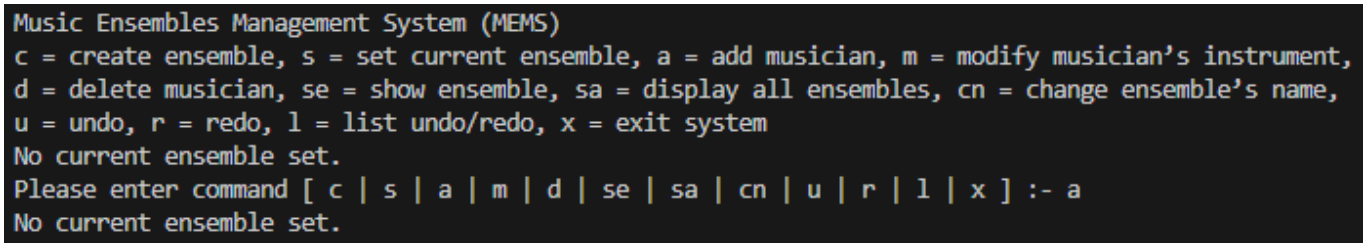
```
a
```

Expected Output:

```
No current ensemble set.
```

Explanation: Test error handling when trying to add musician without current ensemble

Screenshot:



Test Case 16: Error Handling - Set Non-existent Ensemble

Input:

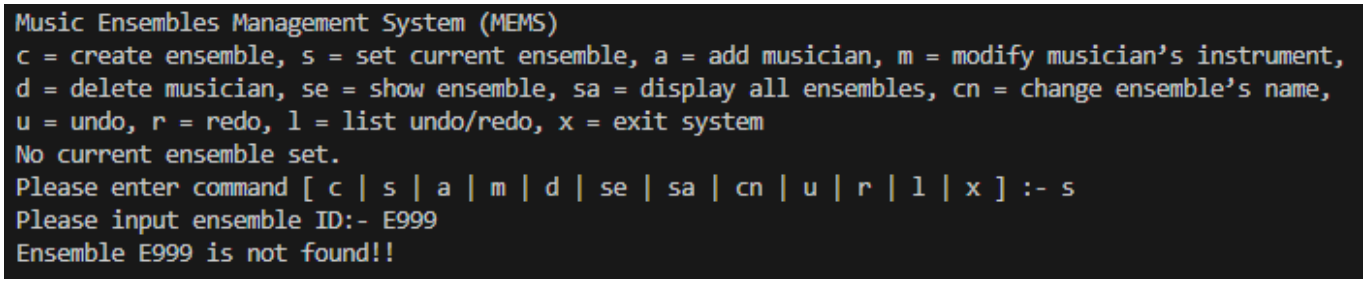
```
s
E999
```

Expected Output:

```
Ensemble E999 is not found!!
```

Explanation: Test error handling when trying to set a non-existent ensemble ID

Screenshot:



Test Case 17: Error Handling - Undo with Empty List

Precondition: System just started or undo list is empty

Input:

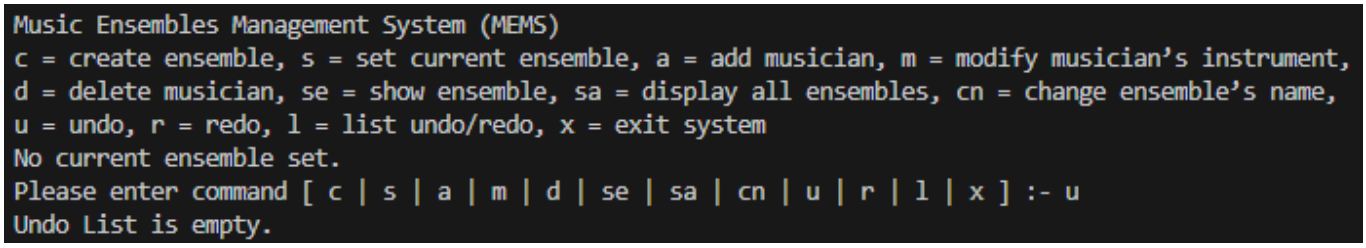
```
u
```

Expected Output:

```
Undo List is empty.
```

Explanation: Test error handling when undo list is empty

Screenshot:



Test Case 18: Error Handling - Redo with Empty List

Precondition: No undo has been executed

Input:

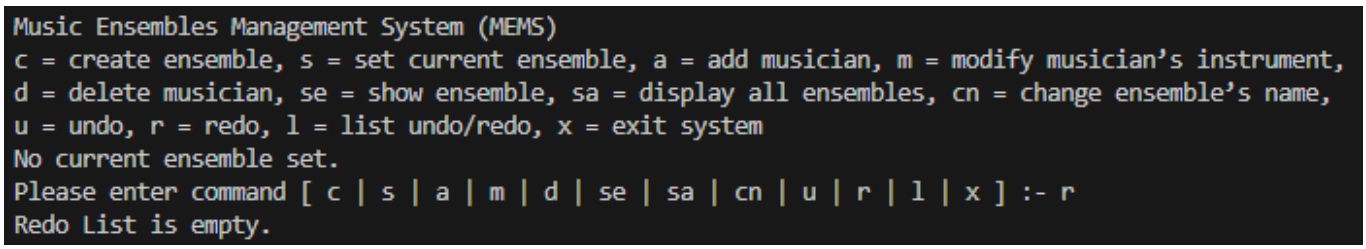
```
r
```

Expected Output:

```
Redo List is empty.
```

Explanation: Test error handling when redo list is empty

Screenshot:



Test Case 19: Exit System

Input:

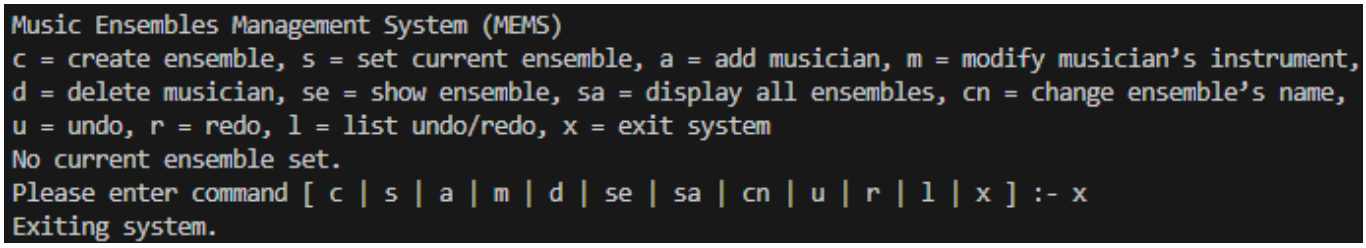
```
x
```

Expected Output:

```
Exiting system.
```

Explanation: Normal system exit

Screenshot:


Music Ensembles Management System (MEMS)
c = create ensemble, s = set current ensemble, a = add musician, m = modify musician's instrument,
d = delete musician, se = show ensemble, sa = display all ensembles, cn = change ensemble's name,
u = undo, r = redo, l = list undo/redo, x = exit system
No current ensemble set.
Please enter command [c | s | a | m | d | se | sa | cn | u | r | l | x] :- x
Exiting system.
