Part 3: Design Patterns

This section explains the three design patterns that I used in MEMS system and why they are important.

3.1 Command Pattern

3.1.1 What is Command Pattern

Command Pattern is a behavioral design pattern that turns a request into a stand-alone object. This object contains all information about the request.

3.1.2 Why I Use It

In MEMS, I have 12 different functions (create ensemble, add musician, delete musician, etc.). Without Command Pattern, all these logic will be put in Main class which make the code very messy and hard to maintain. Also, I need undo/redo functionality which is very difficult to implement without Command Pattern.

3.1.3 How I Implement It

Command Interface:

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

All commands must implement these four methods. The `execute()` does the actual work, `undo()` reverses the operation, `getDescription()` returns description for undo/redo list.

Concrete Command Classes: I created 12 concrete command classes for 12 functions:

1. CreateEnsembleCommand - create new ensemble
2. SetCurrentEnsembleCommand - set which ensemble to work on
3. AddMusicianCommand - add musician to ensemble
4. ModifyInstrumentCommand - change musician's instrument
5. DeleteMusicianCommand - remove musician from ensemble
6. ChangeNameCommand - change ensemble name
7. ShowEnsembleCommand - display current ensemble info
8. DisplayAllEnsemblesCommand - show all ensembles
9. ListUndoRedoCommand - show undo/redo lists
10. UndoCommand - undo last operation
11. RedoCommand - redo undone operation
12. QuitCommand - exit program

Invoker (EnsembleManager): The EnsembleManager class acts as invoker. It maintains two stacks:

- `history` stack - store executed commands for undo
- `redoStack` - store undone commands for redo

When user execute command, manager push it to history stack. When user undo, manager pop from history and push to redo stack.

Benefits in Our System:

- Each command is a separate class, easy to understand and modify
- New commands can be added easily without changing existing code
- Undo/redo implementation becomes natural and simple
- Main class is clean, it just creates commands and executes them

3.1.4 Example: AddMusicianCommand

```
public class AddMusicianCommand implements EnsembleCommand {
    private final Ensemble ensemble;
    private final Musician musician;

    public void execute() {
        ensemble.addMusician(musician);
        // show success message
    }

    public void undo() {
        ensemble.removeMusician(musician);
        // show undo message
    }
}
```

When user add musician, execute() is called. If user want to undo, the undo() method will remove the musician back.

3.2 Factory Pattern

3.2.1 What is Factory Pattern

Factory Pattern is creational design pattern that provides a way to create objects without specifying the exact class of object that will be created.

3.2.2 Why I Use It

Creating command objects is complicated because:

- Different commands need different parameters
- Some commands need user input from scanner
- Main class should not know the details of how to create each command
- I want to keep Main class simple and clean

3.2.3 How I Implement It

CommandFactory Class:

```
public class CommandFactory {
    private final EnsembleManager manager;
    private final Scanner scanner;

    public Command createCommand(String commandCode) {
        switch (commandCode) {
            case "c": return createEnsembleCommand();
            case "a": return addMusicianCommand();
            case "d": return deleteMusicianCommand();
            // ... more cases
        }
    }
}
```

The factory has one main method `createCommand()` that takes command code (like "c", "a", "d") and returns appropriate Command object.

How It Works:

1. User types command code in Main
2. Main passes the code to CommandFactory
3. Factory creates the correct Command object
4. Factory handles all user input needed for that command
5. Factory returns ready-to-execute Command back to Main

Benefits in Our System:

- Main class don't need to know details of creating commands
- All creation logic is centralized in one place
- Easy to add new command types
- User input handling is separated from Main class

3.2.4 Example: Creating AddMusicianCommand

```
private Command addMusicianCommand() {
    if (manager.getCurrentEnsemble() == null) {
        // show error, return null
    }

    // ask user for musician name
    System.out.print("Enter musician name: ");
    String name = scanner.nextLine();

    // ask for instrument/role
    // create Musician object
    // create AddMusicianCommand with musician
```

```
        return new AddMusicianCommand(ensemble, musician);
    }
```

Factory handle all the interaction with user to collect information, then create and return the command.

3.3 Memento Pattern

3.3.1 What is Memento Pattern

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

3.3.2 Why I Use It

For undo functionality, I need to save the state before making changes. For example:

- Before changing musician's instrument, save the old instrument
- Before changing ensemble name, save the old name
- When user undo, restore the saved state

Without Memento Pattern, Command classes would need to know internal details of Musician and Ensemble classes, which breaks encapsulation.

3.3.3 How I Implement It

MusicianMemento Class:

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

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

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

This memento save musician's role/instrument. When need to restore, just call `restore()` method.

EnsembleMemento Class:

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

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

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

This memento save ensemble's name for undo operations.

How It's Used in Commands: In ModifyInstrumentCommand:

```
public class ModifyInstrumentCommand implements EnsembleCommand {
    private MusicianMemento memento;

    public void execute() {
        memento = new MusicianMemento(musician); // save state
        musician.setRole(newRole); // change role
    }

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

Benefits in Our System:

- Clean separation between command and domain objects
- Easy to save and restore state
- Domain classes (Musician, Ensemble) don't need to know about undo functionality
- Simple and clear code

3.4 How Patterns Work Together

All three patterns work together to make MEMS flexible and maintainable:

1. **User Input → Factory → Command**
    - User type command code
    - Factory create appropriate Command object
    - Command is ready to execute
2. **Command → Memento → Undo**
    - Before execute, Command create Memento to save state
    - Execute change the state
    - Undo use Memento to restore state

3. **Manager → Stack → History**
    - Manager execute commands and store in history
    - History stack enable undo/redo
    - Each command know how to undo itself

This combination makes the system:

- Easy to extend (add new commands)
- Easy to maintain (each pattern has clear responsibility)
- Easy to test (each command can be tested separately)
- Clean code structure (Main class is simple)

3.5 Summary

Pattern	Purpose	Main Classes
Command	Encapsulate operations as objects	Command, EnsembleCommand, 12 concrete commands
Factory	Create command objects	CommandFactory
Memento	Save and restore state	MusicianMemento, EnsembleMemento

These patterns are not just to fulfill assignment requirement, they actually solve real problems in the system design and make code better organized and easier to understand.