Hong Kong Institute of Vocational Education
Department of Information Technology

Higher Diploma in Software Engineering

Module: Contemporary Topics in Software Engineering
Module Code: ITP4507

Report (Academic Year 2024/2025)

Supervisor: Mr. Pok Hong Cheng
Student: Karim Baba
Student ID: 240167470

1. Introduction

This report documents the design and implementation of the Musical Ensemble Management System (MEMS).
The system is a console-based Java application that allows users to manage musical ensembles, including creating ensembles, adding and updating musicians, and using undo/redo to navigate changes.

The assignment requires the use of the Command, Factory and Memento design patterns, and the design must be open for extension to support new ensemble types in future.

Development was done in stages:

- First, the system was implemented with the two ensemble types in the specification: **Orchestra** and **Jazz Band**.

- Then, a third ensemble type, **Rock Band**, was added without changing the core architecture, to demonstrate extensibility and conformance to the **Open/Closed Principle (OCP)**.

- After that, automated test cases were added (input files + scripts).

- Finally, the input echoing mechanism was introduced so that automated test outputs clearly show both prompts and the user commands.

---

2. Assumptions

Main assumptions:

- The specification initially required two ensemble types:
    - Orchestra ensemble
    - Jazz band ensemble
      A third type, Rock band ensemble, was added later as an extension using the same design.

- Each ensemble ID (e.g. E001, E102, E201) is unique within a single execution of the program.

- Musician IDs are assumed to be unique within the same ensemble.

- The system is in-memory and single-session. No data is persisted across runs.

- Interaction is entirely through the terminal. All input and output are textual.

- Commands are entered using fixed codes:
  `c, s, a, m, d, se, sa, cn, u, r, l, x`.
  Musician details follow the format `id, name`. Instrument choices are integers.

- Invalid inputs (unknown command, invalid ID, invalid role) result in an error message, and the internal state is not changed.

These assumptions keep the focus on object-oriented design and pattern usage rather than persistence or advanced UI.

---

3. System Design

3.1 Overview

The system is structured into several logical parts.

Domain layer

- `Musician`
- `Ensemble` (abstract)
- `OrchestraEnsemble`
- `JazzBandEnsemble`
- `RockBandEnsemble` (added later to test extensibility)

Core application / state

- `MEMSContext`
- `EnsembleFactory` (interface)
- `SimpleEnsembleFactory`

Command and history layer

- `Command` (interface)
- `AbstractCommand`
- Concrete command classes (e.g. `CreateEnsembleCommand`, `AddMusicianCommand`, etc.)

- `CommandManager`
- `CommandFactory` (interface)
- `SimpleCommandFactory`

Memento support

- `MusicianRoleMemento`
- `EnsembleNameMemento`

User interface

- `Main`

This separation makes the system easier to maintain and extend, and clearly shows where each design pattern is applied.

3.2 Domain Model

Musician
Represents one musician with:

- `musicianID`
- `mName`
- `role` (instrument) as an integer

Simple getters and setters are provided.

Ensemble (abstract)
Base class for all ensembles.

Fields:

- `ensembleID`

- `eName`
- `AbstractList<Musician> musicians`

Methods:

- Add / remove musicians
- Search for a musician by ID
- Get and set ensemble name
- `updateMusicianRole()` and `showEnsemble()` as abstract methods (per spec)

Concrete ensemble types extend this class.

OrchestraEnsemble / JazzBandEnsemble / RockBandEnsemble

- `OrchestraEnsemble`
    - Roles: violinist (1), cellist (2)
    - `showEnsemble()` prints separate lists for violinists and cellists.
- `JazzBandEnsemble`
    - Roles: pianist (1), saxophonist (2), drummer (3)
    - `showEnsemble()` prints separate lists for these roles.
- `RockBandEnsemble` (added later)
    - Roles: guitarist (1), bassist (2), drummer (3), vocalist (4)
    - `showEnsemble()` prints separate lists for these four sections.

Each subclass implements `showEnsemble()` according to its instruments.
`updateMusicianRole()` is also implemented, but actual changes in the running system are done via the command layer.

3.3 Application Core

MEMSContext

Holds global state:

- A map of ensembles, keyed by ensemble ID.
- A reference to the current ensemble.

Provides methods to:

- Add and remove ensembles.
- Retrieve an ensemble by ID.
- Set and get the current ensemble.
- List all ensembles for display.

EnsembleFactory and SimpleEnsembleFactory

`EnsembleFactory` defines methods for creating ensembles and musicians.

`SimpleEnsembleFactory` implements this:

- Initially:
    - `"O"` → `OrchestraEnsemble`
    - `"J"` → `JazzBandEnsemble`
- After extension:
    - `"R"` → `RockBandEnsemble` (new)

`createMusician(id, name, role)` is shared for all types.

Because all ensemble creation goes through the factory, adding `RockBandEnsemble` only required:

- adding the new subclass, and
- updating `createEnsemble()` in `SimpleEnsembleFactory`

No changes were needed elsewhere in the core.

3.4 Command and History Layer

Command and AbstractCommand

- `Command` declares: `execute()`, `undo()`, `getDescription()`.
- `AbstractCommand` stores:
    - `MEMSContext ctx`
    - `CommandManager manager`
    - `description` text
    - an `undoable` flag

Concrete commands extend `AbstractCommand` and implement the actual behaviour.

CommandManager

Responsible for:

- Executing commands.
- Maintaining undo and redo stacks.
- Providing lists of commands for the “show history” function.

Concrete Command Classes

Examples:

- `CreateEnsembleCommand`

- `SetCurrentEnsembleCommand`
- `AddMusicianCommand`
- `ModifyMusicianInstrumentCommand`
- `DeleteMusicianCommand`
- `ChangeEnsembleNameCommand`
- `ShowCurrentEnsembleCommand`
- `ShowAllEnsemblesCommand`
- `ShowHistoryCommand`
- `UndoCommand`
- `RedoCommand`

Commands operate on `MEMSContext` and the ensemble/musician domain classes.
No structural changes in these classes were needed when `RockBandEnsemble` was added; they reuse the same context and factories.

CommandFactory and SimpleCommandFactory

`CommandFactory` defines `createCommand(String userCommand)`.

`SimpleCommandFactory`:

- Interprets input command codes (`c`, `a`, `m`, etc.).
- Reads additional data from the console (types, IDs, names, roles).
- Returns the appropriate command object.

Instrument prompting and validation:

- Initially:

    - Orchestra roles: 1 (violinist), 2 (cellist).
    - Jazz roles: 1–3 (pianist, saxophonist, drummer).
- After rock band was added:
    - For a `RockBandEnsemble`, prompts with:
      `1 = guitarist, 2 = bassist, 3 = drummer, 4 = vocalist`.
    - `readRoleForCurrentEnsemble()` and `isRoleValidForEnsemble()` were extended to handle these new roles.

3.5 Extensibility and Adding RockBandEnsemble

The specification requires the design to support new ensemble types. The development process reflected this explicitly.

Initial implementation (two ensemble types)

First version supported only:

- `OrchestraEnsemble`
- `JazzBandEnsemble`

Key points:

- Domain: only those two concrete classes.
- Factory: `SimpleEnsembleFactory.createEnsemble()` recognised `"O"` and `"J"`.
- UI: `SimpleCommandFactory` instrument prompts only for orchestra/jazz.
- `ShowAllEnsemblesCommand` labelled them as “Orchestra Ensemble…” and “Jazz Band Ensemble…”.

Extension: adding RockBandEnsemble

To add rock band support:

- Added **new domain class** `RockBandEnsemble` with four roles:
    - guitarist = 1
    - bassist = 2
    - drummer = 3
    - vocalist = 4
- Implemented `showEnsemble()` to display the four sections.
- Updated the factory:
    - `SimpleEnsembleFactory.createEnsemble()` now recognises `"R"` and returns `new RockBandEnsemble(id)`.
- Updated command factory:
    - `createCreateEnsembleCommand()` prompt updated to: `o = orchestra | j = jazz band | r = rock band`.
    - `readRoleForCurrentEnsemble()` extended with a branch for `RockBandEnsemble`.
    - `isRoleValidForEnsemble()` extended to validate roles 1–4 for rock bands.
- Updated `ShowAllEnsemblesCommand`:
    - Added a branch to label rock bands as “Rock Band Ensemble …”.

No changes were required to:

- `MEMSContext`
- `CommandManager`
- the existing command types (they already worked with the new ensemble)

This shows the core is **open for extension** and **closed for modification**, consistent with the Open/Closed Principle.

3.6 Development Steps and Echoing Input

After the main features and Rock Band extension were complete, the next steps were:

1. **Adding automated tests**
    - Test input files (`tests/test01_...txt` etc.) were created for different usage scenarios.
    - Simple shell/batch scripts (`run_tests.sh`, `run_tests.bat`) run all tests by redirecting each file into `java Main` and storing the output in `test_outputs/*.out`.
2. **Issue noticed: inputs not visible in .out files**
    - When using input redirection (`java Main < testXX.txt`), the program reads from STDIN but does **not** echo the user input.
    - The `.out` files only showed prompts and responses, not the actual commands, which made it harder to read and review test results.
3. **Fix: echoing all user input**
   To make automated traces clearer, two small changes were made:
    - In `Main`:
        - Added a helper `readLineEcho(Scanner)` that reads a line and prints it.
        - The main loop now uses this helper instead of `scanner.nextLine()`, so each command code is echoed to the console and to the `.out` files.
    - In `SimpleCommandFactory`:
        - Added a helper `readLineEcho()` that wraps `scanner.nextLine()` and prints the line.
        - All interactive reads in that class (ensemble type, IDs, names, musician info, instrument roles) were changed to use this helper.
4. Result:

    - All test outputs now show both prompts and the actual input values, making the scripts and screenshots easier to understand.

These echoing changes do not affect the internal logic of the system; they only improve readability of automated tests and console sessions.

---

4. Design Patterns

4.1 Command Pattern

The Command pattern is used to encapsulate each user operation as a separate object.

- `Command` and `AbstractCommand` define the basic contract and shared data.
- Concrete commands implement:
    - Creation of ensembles.
    - Setting current ensemble.
    - Adding, modifying and deleting musicians.
    - Changing ensemble names.
    - Showing ensembles.
    - Showing history.
- `CommandManager` acts as invoker and history manager (maintains undo/redo stacks).
- `MEMSContext` and the ensemble/musician classes are the receivers.

Adding `RockBandEnsemble` did not require any changes to the command hierarchy; the same commands work with the new ensemble type without modification.

4.2 Factory Pattern

The Factory pattern centralises creation logic.

- `EnsembleFactory` defines creation methods for ensembles and musicians.

- `SimpleEnsembleFactory` decides which concrete ensemble class to instantiate:
    - Initially: `"O"` → `OrchestraEnsemble`, `"J"` → `JazzBandEnsemble`.
    - Later extended: `"R"` → `RockBandEnsemble`.

This allowed **adding a new ensemble type** by:

- implementing `RockBandEnsemble`, and
- extending `SimpleEnsembleFactory.createEnsemble()`.

No changes were needed in the command manager, context, or main loop.

4.3 Memento Pattern

The Memento pattern is used for undo/redo of particular changes:

- `MusicianRoleMemento` stores:
    - a reference to a `Musician`
    - old role
    - new role
- `EnsembleNameMemento` stores:
    - a reference to an `Ensemble`
    - old name
    - new name

These are used in:

- `ModifyMusicianInstrumentCommand`
- `ChangeEnsembleNameCommand`

The commands act as originators (they create and hold mementos), and `CommandManager` is the caretaker.
Because roles are integer-based, the same Memento mechanism works for Orchestra, Jazz Band and Rock Band without any change.

---

5. User Interface and Usage

The `Main` class provides the console UI:

- Creates core objects: `MEMSContext`, `CommandManager`, `SimpleEnsembleFactory`, `SimpleCommandFactory`.
- Displays the menu and the current ensemble.
- Reads the command code (now echoing it back to the console).
- Uses `SimpleCommandFactory.createCommand()` to build a `Command`.
- Executes the command via `CommandManager.execute()`.

Command codes:

- `c` – create ensemble (type `o` / `j` / `r`)
- `s` – set current ensemble by ID
- `a` – add musician to current ensemble
- `m` – modify musician's instrument
- `d` – delete musician
- `se` – show current ensemble
- `sa` – display all ensembles
- `cn` – change ensemble's name
- `u` – undo

- `r` – redo
- `l` – show undo/redo lists
- `x` – exit

For rock bands, the prompts when adding or modifying musicians offer:

- `1 = guitarist`
- `2 = bassist`
- `3 = drummer`
- `4 = vocalist`

---

6. Test Plan and Test Cases

6.1 Test Plan

Testing uses black-box functional tests with predefined input files.

- Each test file (`tests/test0X_*.txt`) contains a sequence of commands.

Tests are run via input redirection, for example:

```
java Main < tests/test01_sample_basic.txt
```

- Shell and batch scripts (`run_tests.sh`, `run_tests.bat`) run all tests and store outputs in `test_outputs/*.out`.
- Because of the echoing functions added later, outputs include both prompts and inputs, making verification easier.

Objectives:

- Verify all required functions work as specified.
- Verify undo/redo behaves correctly.
- Verify invalid input handling.
- Verify that the Rock Band type integrates correctly and does not break existing behaviour.

6.2 Existing Test Cases (Orchestra & Jazz)

Summarised:

- **TC01 – Basic workflow (orchestra + jazz band)**
  Create ensembles, add musicians, show current ensemble and all ensembles.
- **TC02 – Undo / redo behaviour**
  Execute a sequence of commands, then perform multiple undo and redo operations.
- **TC03 – Invalid input handling**
  Test unknown commands, invalid formats, missing IDs, etc.
- **TC04 – Invalid ensemble ID for set-current**
  Attempt to set current ensemble to an ID that does not exist.
- **TC05 – Undo creation of ensemble**
  Undo a create operation and verify that the ensemble is removed and cannot be selected.

These tests were written for the initial two types and still pass after Rock Band was added.

6.3 New Test Cases for RockBandEnsemble

- **TC06 – Rock band basic creation and display**

  Purpose:
  Confirm that a rock band ensemble can be created, musicians can be added in all four roles, and the ensemble displays correctly.

  Input file: `tests/test06_rock_band_basic.txt`

  Key steps:

    - `c` → type `r`, ID `E201`, name `METAL HEADS`.
    - `a` → add vocalist (role 4).
    - `a` → add guitarist (role 1).
    - `a` → add bassist (role 2).
    - `a` → add drummer (role 3).
    - `se` to display the rock band.
    - `sa` to display all ensembles.
- Expected:
    - Rock band listed as `Rock Band Ensemble METAL HEADS (E201)`.
    - All four musicians appear under the correct headings.
- **TC07 – Rock band invalid role**

  Purpose:
  Verify that invalid role codes are rejected for rock bands.

  Input file: `tests/test07_rock_band_invalid_role.txt`

  Key steps:

    - `c` → type `r`, ID `E202`, name `ALT BAND`.
    - `a` → attempt to add musician with role `5`.
    - `a` → add valid guitarist with role `1`.
    - `se` to show ensemble.
- Expected:
    - First `a` with role `5` prints “Invalid instrument choice” and does not add the musician.

    - Second `a` adds `m302` correctly as guitarist.
    - `se` shows only the valid musician in the Guitarist section; other sections are empty.

For the submitted report, each test case can be accompanied by a screenshot of the console or an excerpt from the `.out` file.

---

7. Source Code Documentation

The Java source code is documented as follows:

- Each `.java` file starts with a header comment containing:
    - System name (Musical Ensemble Management System)
    - Module code
    - Student name
    - Student ID
    - File name
- Classes and key methods include short comments describing:
    - The responsibility of the class (e.g. `MEMSContext`, `CommandManager`).
    - The purpose of important methods (e.g. `execute()`, `undo()`, `showEnsemble()`).