This set of functionality consists of one GUI component and two commands, each of which rely on the same core components:

1. `HistoryPane` - GUI component that displays a numbered list of previous data-modifying commands that can be reverted by `app-undo`.
2. `app-undo` - Command that reverts until a specified data-modifying command in the history, or the previous data-modifying command if no argument is specified.
3. `app-redo` - Command that reverts the previous command if and only if it was an `app-undo`.

Overview of Changes

For undo/redo/history to function, there must be the following:

- A way to mark which `Command` s are data-modifying commands so that non-data-modifying commands can be ignored.
- A mechanism to clearly demarcate the model state before and after the execution of each data-modifying command.
- A way to record the state of the model before (or after) each command, together with the `Command` object responsible.
- Methods to switch the state of the model back and forth from the states stored in the history.

To achieve this, the following classes are introduced:

- `MutatorCommand` - An empty subclass of `Command` to indicate that a `Command` is a data-modifying command and should be considered for undo/redo/history.
- `HistoryRecord` - Record of the command and data of one atomic commit in the command history. It consists of the `MutatorCommand` responsible for the commit and the state of the model either before or after the commit. All recorded states are deep copies decoupled from their original references.
- `HistoryManager` - Represents the history of the application. Maintains data structures containing `HistoryRecord` s of each data state change.

Additionally, `LogicManager` and `Model` are modified to accommodate this functionality.

Shallow Copy vs Deep copy

- Performing a *shallow copy* of an object simply copies the values of the references of each object the original object is pointing to. This results in the copied object pointing to the same objects as the original. In the figure below, `patientBook` and `copiedPatientBook` are separate `PatientBook` objects but actually share the same `patient` object. Changes to `patient` through `PatientBook` would thus affect `copiedPatientBook` as well.

[shallow copy] | *shallow_copy.png*

Figure 1. Shallow copy

- Performing a *deep copy* creates duplicates of each object referenced by the original, the objects referenced by those duplicates, and so on. These duplicates are completely decoupled from their originals. In the figure below, `patient` is a different object from `copiedPatient`, thus changes to `patient` would not affect `copiedPatient`.

[deep copy] | *deep_copy.png*

Figure 2. Deep copy

The following methods are added to the `ModelManager` class:

- `Model#commit(MutatorCommand)` - Commits the changes made to the address book since the last call to this method, making them permanent and updating the UI data. Creates a new `HistoryRecord` containing the committing `MutatorCommand` and the `PatientBook` and `AppointmentBook` states before the execution of the command and pushes it to the `HistoryManager` for storage.
- `Model#getHistory()` - Returns an unmodifiable view of the history.
- `Model#undoTo(HistoryRecord)` - Reverts current model state to the that contained in the specified `HistoryRecord` (i.e. the state before the command was executed).
- `Model#redo()` - Redoes the previous `MutatorCommand` if it was an undo by popping the latest `HistoryRecord` from the `HistoryManager` 's redo stack.

Furthermore, in addition to `MutatorCommand` which was described earlier, the following logical classes are added:

1. `UndoCommand` - Undoes a designated command in the history, or the previous one if no argument is specified. The `COMMAND_WORD` for this command is `app-undo`.
2. `UndoCommandParser` - Parses input arguments and creates a new UndoCommand object.
3. `RedoCommand` - Redoes the previous command if it was an undo. The `COMMAND_WORD` for this command is `app-redo`.

`HistoryManager` checks the classes of commands pushed to the history and does not record them if they are instances of `UndoCommand` or `RedoCommand`. This ensures that successive `UndoCommand` s do not undo themselves instead of the desired data-modifying commands, requiring `RedoCommand` for the special case of undo reversion.

Example Usage Scenario

The following example usage scenario and the state of `historyManager` at each step.

Step 1: The user launches the application.

`HistoryManager` is initialized with empty `history` and `redoStack` objects.

[UndoRedoState0] | *UndoRedoState0.png*

Figure 3. Initial state of `historyManager`

Step 2: The user executes the command `pat-delete 3`.

After the `DeleteCommand` makes its changes on the model, `logicManager` calls `Model#commit(MutatorCommand)`, passing the command object into `modelManager`. In turn, `modelManager` passes the command object, `stagedPatientBook`, and `stagedAppointmentBook` into `historyManager` through `HistoryManager#pushRecord(MutatorCommand, PatientBook, AppointmentBook)`.

`historyManager` uses those objects to create a new `HistoryRecord` object which contains the model state **before** the command was executed (here labelled `hr0`) and pushes it into the `history`.

[UndoRedoState1] | *UndoRedoState1.png*

Figure 4. `historyManager` *after* `pat-delete 3` *is executed*

Step 3: The user executes the command `visit-start`.

The interaction between `logicManager`, `modelManager` and `historyManager` is the same as before.

`historyManager` creates a new `HistoryRecord` object (here labelled `hr1`) and pushes it into the `history`.

[UndoRedoState2] | *UndoRedoState2.png*

Figure 5. `historyManager` *after* `visit-start` *is executed*

Step 4: The user wants to revert to the first item in the history, so he executes the command `app-undo 1`.

First, the `UndoCommand` retrieves the `HistoryRecord` corresponding to the first item in the history by searching the list returned by `Model#getHistory()`. Then, `UndoCommand` calls `Model#undoTo(HistoryRecord)` passing in the target record. When this happens, `modelManager` calls `historyManager#popRecordsTo(HistoryRecord)` to pop all records after and including the target record from the history (`hr1` and `hr0` in the previous step). The `historyManager` uses these popped records to create new records of the model state **after** the commands were executed, and places these new records (`hr2` and `hr3`) into the `redoStack`.

[UndoRedoState3] | *UndoRedoState3.png*

Figure 6. `historyManager` *after* `app-undo 1` *is executed*

Finally, `modelManager` calls `ModelManager#changeBaseTo(PatientBook, AppointmentBook)` using the state objects in the target record. This performs the actual reversion of the state.

The action of the `UndoCommand` is summarised in the sequence diagram below:

[UndoSequenceDiagram] | *UndoSequenceDiagram.png*

Figure 7. Sequence diagram for `app-undo`

Step 5: The user wants to redo `pat-delete 3`, so he executes the command `app-redo`.

The `RedoCommand` calls `Model#redo()`. The `modelManager` calls `HistoryManager#popRedo(PatientBook, AppointmentBook)` passing it the current `stagedPatientBook` and `stagedAppointmentBook`.

`historyManager` pops the record at the top of the `redoStack` (`hr3`) and uses its `command`, together with the `PatientBook` and `AppointmentBook` just passed in by the `modelManager`, to create a new `HistoryRecord` (`hr4`) describing the model state **before** that command was executed. It then pushes `hr4` into the history.

[UndoRedoState4] | *UndoRedoState4.png*

Figure 8. `historyManager` *after* `app-redo` *is executed*

`historyManager#popRedo()` returns the `HistoryRecord` (`hr3`) containing the state after the redo. `modelManager` can now call `ModelManager#changeBaseTo()` to change the state to it.

Step 6: The user executes the command `pat-clear`.

The action of the `logicManager` and `modelManager` is similar what is described in **Step 2**; only this time, when `modelManager` calls `HistoryManager#pushRecord(MutatorCommand, PatientBook, AppointmentBook)`, the `historyManager` sees that the committing command is not an `UndoCommand` or a `RedoCommand` and clears the `redoStack` to avoid branching.

[UndoRedoState5] | *UndoRedoState5.png*

Figure 9. `historyManager` *after* `pat-clear` *is executed*

Model and Logic Design Considerations

Aspect: How to undo and redo between states

- **Alternative 1 (current choice)**: Save the entire `PatientBook` and `AppointmentBook` objects to record each model state.
  - Pros: Easier to implement.
  - Cons: Consumes more memory.
- Alternative 2: Only save the `MutatorCommand` objects but implement an `undo()` method for each `MutatorCommand` which does exactly the reverse of its `execute()` method.
  - Pros: Consumes much less memory.
  - Cons: Difficult to implement - doubles the amount of work needed for each command.

Aspect: How to record the PatientBook and AppointmentBook states in the history

- Alternative 1: Simply store references to `PatientBook` and `AppointmentBook`.
  - Pros: Easier to implement.
  - Cons: Relies on the assumption that the objects in `PatientBook` and `AppointmentBook` are

immutable; if they are not truly immutable, changes to the current model’s `PatientBook` and `AppointmentBook` state may leak and affect the states stored in the history.

- **Alternative 2 (current choice):** Defensively store deep copies of the `PatientBook` and `AppointmentBook`.
  - Pros: Prevents improperly coded `Patient` or `Appointment` (or their associated classes) from breaking undo/redo/history functionality. Can reuse JSON serialization code for persistent storage of `PatientBook` and `AppointmentBook` to create deep copies by serializing then immediately deserializing them.
  - Cons: Consumes more memory and CPU time. More difficult to implement - MVC pattern between UI view and models is broken in two. This is because each time the current state is swapped with a state in the history by `ModelManager`, the `ObservableList` viewed by the UI must also be updated by the `ModelManager` instead of the `PatientBook` as the current `PatientBook` is completely decoupled and placed into the history.

Aspect: Which class to place the HistoryManager in

- **Alternative 1 (current choice):** Make `HistoryManager` a field of `ModelManager`.
  - Pros: Ensures atomicity of the records in the history as pushing a transaction to the `HistoryManager` can only be (and is always) done by `Model#commit()` itself - records in the history are guaranteed to be products of complete command execution rather than intermediate states.
  - Cons: More difficult to test `ModelManager` as two `ModelManager` objects may have the same current state but differing `HistoryManager` objects. May violate Single Responsibility Principle as `ModelManager` now has to manage both its current state and its previous states.
- Alternative 2: Make `HistoryManager` a field of `LogicManager`.
  - Pros: Higher cohesion as `ModelManager` only represents the model’s current state. Easier to test `ModelManager` as only its current state matters.
  - Cons: It is possible for intermediate model states to be pushed to the `HistoryManager` - trusts `LogicManager` to push the transaction to history after (and only after) calling `Model#commit()`. Requires `Command#execute()` to accept `HistoryManager` as a parameter just so `UndoCommand` and `RedoCommand` can work even though the vast majority of commands do not require it.

UI

The command history is constantly displayed in a panel on the right side of the app. This `HistoryPanel` uses `HistoryRecordCard` s to display the user-input text that invoked each command. It is a view of the `ObservableList<HistoryRecord>` returned by `HistoryManager#asUnmodifiableObservableList()`.

UI Design Considerations

Aspect: Where to display the history

- **Alternative 1 (current choice):** Permanently display it in a dedicated panel.
  - Pros: User does not have to execute a 'history' command to view the history, making it much

easier to use the multiple undo function.

    - Cons: Takes up more space in the UI.

- Alternative 2: Display it as a tab in the `TabPane`.
    - Pros: Saves space in the UI.
    - Cons: User has to switch to the history tab to view it. Less intuitive UX as the other tabs in the `TabPane` all display actual data such as `Patient`, `Visit`, and `Appointment` info, whereas history is app metadata.