CSCI 498B/598B - HW 6 Solution Key

Your Name

November 26, 2025

Question 1: Profiles and Notifications

1. Name of Design Pattern(s)

Iterator and Observer

2. Justification

1. **Observer Pattern**: the follower/followee notification system.
2. **Iterator Pattern**: accessing the collection of notifications independently of the collection's representation.

   Iterator is providing a way for the `NotificationBar` (Client) to access the elements of the `Notification` collection sequentially **without exposing the collection's underlying structure** (e.g., `List` vs. `Array`). This satisfies the requirement for iteration independence.
3. **Alternative Pattern:** OPTIONAL as there is no good alternative...... While the **Visitor Pattern** can also traverse structures, it is overly complex for simple sequential access. The Iterator Pattern is a simpler and more direct solution for decoupling collection structure from traversal logic. Not using a pattern is another alternative, but then we will be exposing internal representation.

3. UML Class Diagram

We can implement our own iterator or use the existing iterator of a collection, like iterator of java.util.List.

Solution 1: Canonical Iterator: Diagram 1 introduces the `NotificationCollection` as the `Aggregate`, which is responsible for creating and returning the concrete `NotificationIterator`. This adheres strictly to the separation of concerns.

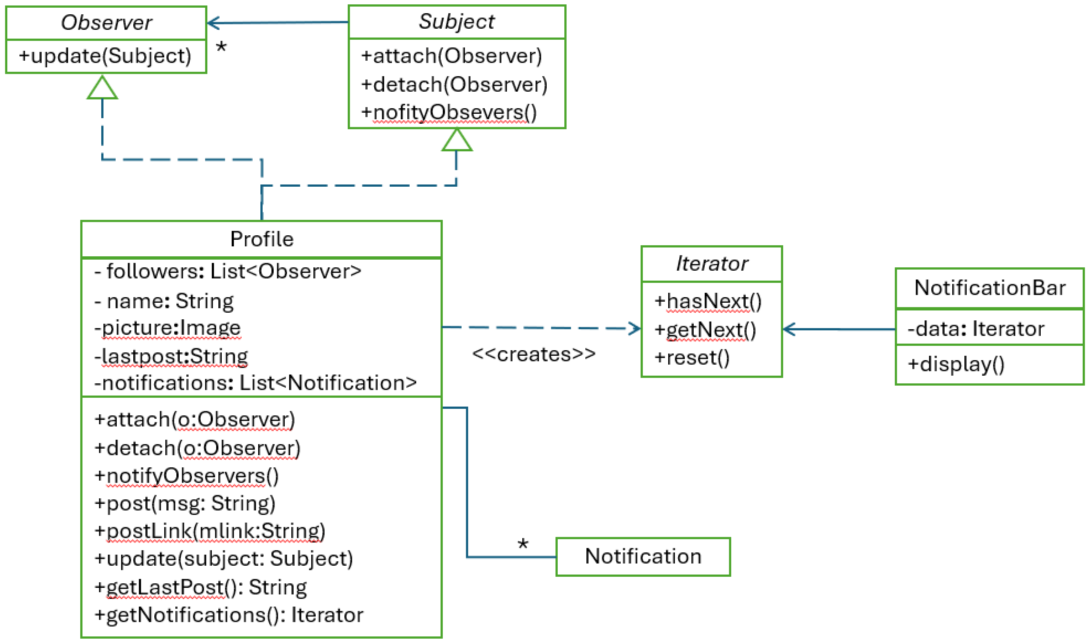


Figure 1: UML Diagram: Observer and Canonical Iterator (using Aggregate and Iterator classes).

Solution 2: Using the existing Iterator: Diagram 2 is simpler as it uses the existing iterator in the langauge. The `Profile` returns the built-in iterator of its internal `List<Notification>`.

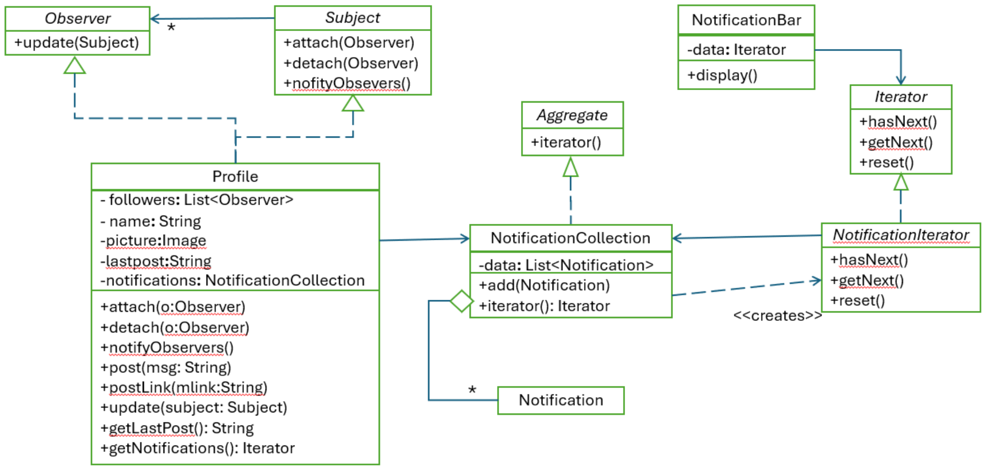


Figure 2: UML Diagram: Observer and existing Iterator.

4. Java Code

Core Components (Common to Both Solutions):

```java
// Observer Pattern Interfaces (Pull Method)
interface Subject {
    void attach(Observer o);
    void detach(Observer o);
    void notifyObservers();
}

interface Observer {
    void update(Subject subject); // Pull method: subject passes itself
}

// Data class
class Notification {
    private String message;
    public Notification(String msg) { this.message = msg; }
    public String toString() { return message; }
}

// Iterator Interface
interface Iterator {
    boolean hasNext();
    Object getNext();
    void reset();
}

// Client Usage
class NotificationBar {
    public void display(Iterator iterator) {
        System.out.println("\n--- Notification Bar  ---");
        iterator.reset(); // Reset for a fresh display
        while (iterator.hasNext()) {
            System.out.println(" - " + iterator.getNext());
        }
    }
}
```

Solution 1: Canonical Iterator Code

This solution aligns with UML Diagram 1. It introduces the Aggregate interface for better separation.

```java
// Aggregate Interface
interface Aggregate {
    Iterator iterator();
}

// Concrete Aggregate
class NotificationCollection implements Aggregate {
    private List<Notification> data = new ArrayList<>();
    public void add(Notification n) { data.add(n); }

    public Iterator iterator() {
        return new NotificationIterator();
    }

    // Concrete Iterator
    public class NotificationIterator implements Iterator {
        private int position = 0;

        public boolean hasNext() { return position < data.size(); }
        public Object getNext() {
            if (hasNext()) return data.get(position++);
            return null;
        }
        public void reset() { position = 0; }
    }
}

// Profile Implementation (Delegates iteration to Aggregate)
class Profile implements Subject, Observer {
    private NotificationCollection notifications = new NotificationCollection();
    private List<Observer> followers = new ArrayList<>();
    private String name, lastPost;

    //Subject implementations
    public void attach(Observer o){followers.add(o);}
    public void dettach(Observer o){followers.remove(o);}
    public void notifyObserver(){
            for(Observer o: followers)
                    o.update(this);
    }

    public void post(String msg) {
        this.lastPost = msg;
        notifyObservers();
    }
    public String getLastPost() { return this.lastPost; }

    // Observer Implementation. Stores notification in the Aggregate
    public void update(Subject subject) {
        String post = subject.getLastPost(); // Pulls the state
        notifications.add(new Notification(subject + " posted: " + post));
    }

    // Iterator Factory Method (Delegates creation to the Aggregate)
    public Iterator getNotifications(){
        return notifications.iterator();
    }
}

```

Solution 2: Using existing Iterators

This solution aligns with UML Diagram 2. It relies on the concrete ‘Profile‘ class managing and iterating its own list.

```
class Profile implements Subject, Observer, Iterator {
    // Implements Iterator to iterate over its own list
    private List<Notification> notifications = new ArrayList<>();

    private List<Observer> followers = new ArrayList<>();
    private String name, lastPost;

    //Subject implementations
    public void attach(Observer o){followers.add(o);}
    public void dettach(Observer o){followers.remove(o);}
    public void notifyObserver(){
            for(Observer o: followers)
                    o.update(this);
    }

    public void post(String msg) {
        this.lastPost = msg;
        notifyObservers();
    }
    public String getLastPost() { return this.lastPost; }

    // Observer Implementation. Stores notification in the Aggregate
    public void update(Subject subject) {
        String post = subject.getLastPost(); // Pulls the state
        notifications.add(new Notification(subject + " posted: " + post));
    }

    public Iterator getNotifications(){ return notifications.iterator(); }

}

```

5. Explanation of Solution

- The Observer pattern solves the requirement: "When a user posts an announcement or meeting link, we want all the followers to get informed."

  Decoupling: The `Profile` class acts as both the Subject (when posting) and the Observer (when following). This decouples the act of posting from the mechanism of delivery. The posting user (`Subject`) does not need to know how many followers exist or how each follower processes the notification.

  Pull Method: By using the pull method (`update(Subject)`), the follower (`Observer`) is notified that a change occurred and must actively pull the specific data (`getLastPost()`) from the `Subject`. This is efficient as the `Observer` decides what data it needs, reducing the amount of information the `Subject` must push to all followers.

- The Iterator pattern solves the requirement: "We want the user to iterate over the notifications independent of how the collection is represented."

  Abstraction: The `Profile` exposes its notifications via a generic `Iterator` interface (obtained through `getNotifications()` or `NotificationCollection.iterator()`).

  Decoupling Traversal and Storage: The `NotificationBar` (the Client) only interacts with the `Iterator`. It is completely unaware of whether the underlying collection inside the `Profile` is an `ArrayList`, a linked list, or a custom array.

  Flexibility: This independence allows the application developers to change the internal data structure of the `NotificationCollection` (e.g., swapping a `List` for a fixed-size `Array`) simply by updating the `Aggregate` and `Concrete Iterator` classes, without requiring any changes to the `NotificationBar` client code.

6. SOLID Principles Evaluation

- **Single Responsibility Principle (SRP):** Adheres to. `Profile` manages user state and relationships. `NotificationCollection` manages the data structure. In one solution, `NotificationIterator` manages the traversal algorithm. In the other one `Iterator` manages the traversal algorithm.

- **Open/Closed Principle (OCP):** Adhered to. New ways to traverse the notifications (e.g., a reverse iterator) can be added by creating a new concrete `Iterator` class without modifying the `NotificationBar` client or the existing `Profile/Aggregate` classes.

Canonical Solution is problematic. Since the concrete iterator is inner class, adding a new iterator requires change in the code. In C++ implementation, the concrete iterator would be a **friend** and then we will not have such OCP problem.

- **Liskov Substitution Principle (LSP):** Adhered to. Any concrete `Iterator` implementation can be substituted where an `Iterator` is expected (in the `NotificationBar`'s `display` method) without breaking the program's functionality.
- **Interface Segregation Principle (ISP):** Adhered to. The interfaces are small and focused: `Subject` for observer management, `Observer` for receiving updates, `Iterator` for traversal. The `Profile` implements only the interfaces it needs.
- **Dependency Inversion Principle (DIP):** Adhered to. `NotificationBar` (Client, a high-level class) depends on the `Iterator` **abstraction**, not on the low-level concrete collection or concrete iterator implementation. This is the core benefit of the Iterator pattern.

Question 2: Configuring Buttons for painting

1. Name of Design Pattern(s)

Command pattern

2. Justification

I suggest the Command Pattern because it provides the necessary structure to decouple the actions (painting techniques) from the triggers (mouse clicks) and supports the dynamic setup required by the application. Since command turns the painting techniques, that are actually method calls, into objects, we can use them to configure the mouse event handling.

Command decouples the request for an action (the left or right mouse click, the **Invoker**) from the object that knows how to perform the action (the `DigitalBrush` or `TextureBrush`, the **Receiver**). Each painting technique (e.g., `smoothStroke()` or `stampPattern()`) on a specific brush instance is encapsulated within its own Concrete Command object (e.g., `SmoothStrokeCommand`). This fulfills the requirement that the artist "sets a primary painting technique as one of the methods of the selected brush."

Alternative Pattern Comparison:

- **Strategy Pattern:** While the Strategy pattern also involves interchangeable objects, its goal is to define a family of interchangeable algorithms. The Command pattern is preferred here because it specifically focuses on encapsulating an action request, which includes the Receiver (the Brush) and the method to be called (the Technique). The primary intent is to parameterize objects (the mouse buttons) with actions (the Commands).

3. UML Class Diagram

Key Participants:

- **Command (`Command`):** Declares the interface for executing an operation.
- **ConcreteCommand (such as `SmoothStrokeCommand`):** Implements the `execute()` method by calling a specific action on the `Receiver`. It holds the `Receiver` instance and a binding to one of its methods.
- **Receiver (`DigitalBrush`, `TextureBrush`):** The object that performs the actual operation when a Command's `execute()` is called. It contains the actual painting technique logic.
- **Invoker (`PaintingCanvas`):** Holds the Command objects and issues the request. It does not know the Command's Receiver or the specific method; it only calls `execute()`.
- **Client (`ArtistSetup`):** Creates the ConcreteCommand objects, setting their Receiver, and assigns them to the Invoker's slots.

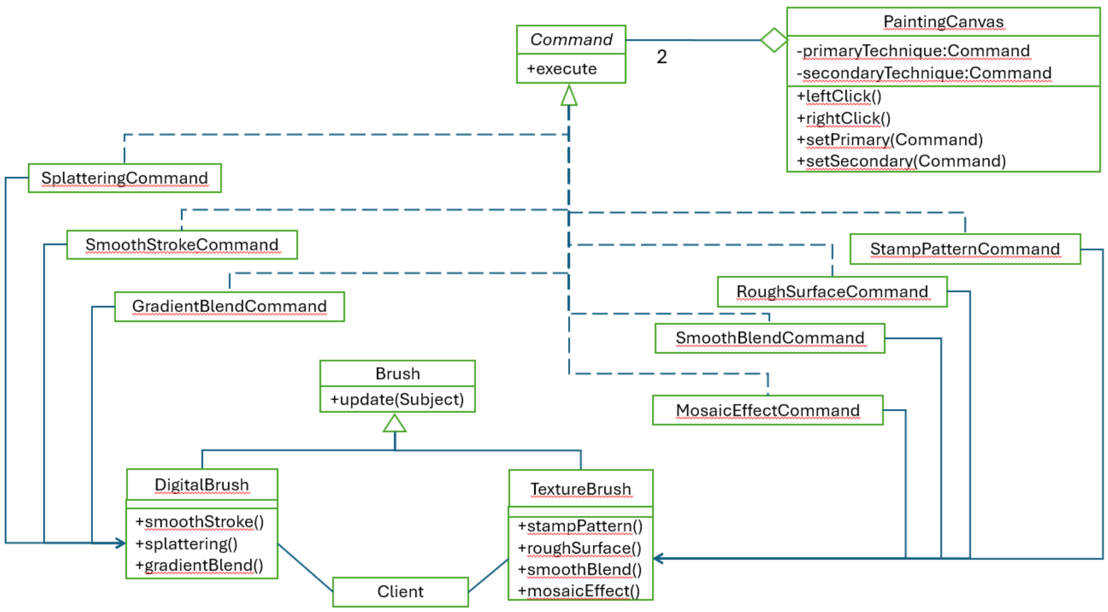


Figure 3: UML Class Diagram for the Command Pattern in the Digital Painting Application.

4. Java Code

Receivers

```java
// Receiver Interface (The brush types)
public abstract class Brush { //Optional.
    protected String name;
    public Brush(String name) { this.name = name; }
}

public class DigitalBrush extends Brush {
    public DigitalBrush() { super("DigitalBrush"); }
    public void smoothStroke() { System.out.println(name + ": Performing smooth stroke."); }
    public void splattering() { System.out.println(name + ": Applying splattering effect."); }
    public void gradientBlend() { System.out.println(name + ": Blending gradient."); }
}

public class TextureBrush extends Brush {
    public TextureBrush() { super("TextureBrush"); }
    public void stampPattern() { System.out.println(name + ": Applying stamp pattern."); }
    public void roughSurface() { System.out.println(name + ": Performing rough surface."); }
    public void smoothBlend() { System.out.println(name + ": Performing smooth blend."); }
    public void mosaicEffect() { System.out.println(name + ": Creating mosaic effect."); }
}

```

Commands and Invoker

```java
// Command Interface
public interface Command {
    void execute();
}

class SmoothStrokeCommand implements Command {
    private DigitalBrush receiver;
    public SmoothStrokeCommand(DigitalBrush brush) { this.receiver = brush; }
    public void execute() {
        receiver.smoothStroke();
    }
}

class SplatterCommand implements Command {
    private DigitalBrush receiver; // The brush instance to operate on
    public SplatterCommand(DigitalBrush brush) { this.receiver = brush; }
    public void execute() {
        receiver.splattering(); // Calls the specific technique method
    }
}

class GradientBlendCommand implements Command {
    private DigitalBrush receiver; // The brush instance to operate on
    public GradientBlendCommand(DigitalBrush brush) { this.receiver = brush; }
    public void execute() {
        receiver.gradientBlend(); // Calls the specific technique method
    }
}

class StampPatternCommand implements Command {
    private TextureBrush receiver; // The brush instance to operate on
    public StampPatternCommand(TextureBrush brush) { this.receiver = brush; }
    public void execute() {
        receiver.stampPattern(); // Calls the specific technique method
    }
}

class RoughSurfaceCommand implements Command {
    private TextureBrush receiver; // The brush instance to operate on
    public RoughSurfaceCommand(TextureBrush brush) { this.receiver = brush; }
    public void execute() {
        receiver.roughSurface(); // Calls the specific technique method
    }
}

class SmoothBlendCommand implements Command {
    private TextureBrush receiver; // The brush instance to operate on
    public SmoothBlendCommand(TextureBrush brush) { this.receiver = brush; }
    public void execute() {
        receiver.smoothBlend(); // Calls the specific technique method
    }
}

class MosaicEffectCommand implements Command {
    private TextureBrush receiver; // The brush instance to operate on
    public MosaicEffectCommand(TextureBrush brush) { this.receiver = brush; }
    public void execute() {
        receiver.mosaicEffect(); // Calls the specific technique method
    }
}


// Invoker (The Canvas/Mouse Handler)
public class PaintingCanvas {
    // Slots to hold the assigned commands
    private Command primaryTechnique; // Left Click
    private Command secondaryTechnique; // Right Click

    // Client/Setup method to assign commands dynamically
    public void setPrimaryTechnique(Command command) {
        this.primaryTechnique = command;
        System.out.println("\n[SETUP] Primary technique (Left Click) set.");
    }

    public void setSecondaryTechnique(Command command) {
        this.secondaryTechnique = command;
        System.out.println("[SETUP] Secondary technique (Right Click) set.");
    }

    // Invoker logic: issues the request without knowing the receiver
    public void leftClick() {
        System.out.print("Left Click does ");
        if (primaryTechnique != null) {
            primaryTechnique.execute();
        } else {
            System.out.println("No primary technique assigned.");
        }
    }

    public void rightClick() {
        System.out.print("Right Click does ");
        if (secondaryTechnique != null) {
            secondaryTechnique.execute();
        } else {
            System.out.println("No secondary technique assigned.");
        }
    }
}
```

Client Usage :

```java
public class ArtistSetup {
    public static void main(String[] args) {
        PaintingCanvas canvas = new PaintingCanvas();
        DigitalBrush dBrush = new DigitalBrush();
        TextureBrush tBrush = new TextureBrush();

        // --- Session 1: Setup with DigitalBrush ---
        System.out.println("--- Session 1: DigitalBrush Setup ---");

        // 1. Create Concrete Commands, binding them to the dBrush (Receiver)
        Command smooth = new SmoothStrokeCommand(dBrush);
        Command splatter = new SplatterCommand(dBrush);

        // 2. Assign Commands to the Invoker slots
        canvas.setPrimaryTechnique(smooth);
        canvas.setSecondaryTechnique(splatter);

        // 3. Painting/Execution
        canvas.leftClick(); // Executes SmoothStrokeCommand on dBrush
        canvas.rightClick(); // Executes SplatterCommand on dBrush

        // --- Session 2: Setup with TextureBrush ---
        // The artist changes the brush, so the setup must be performed again
        System.out.println("\n--- Session 2: TextureBrush Setup (Must change Commands) ---");

        // 1. Create new Concrete Commands, binding them to the tBrush (Receiver)
        Command stamp = new StampPatternCommand(tBrush);
        Command mosaic = new MosaicEffectCommand(tBrush);

        // 2. Assign the new Commands to the Invoker slots
        canvas.setPrimaryTechnique(stamp);
        canvas.setSecondaryTechnique(mosaic);

        // 3. Painting/Execution
        canvas.leftClick(); // Executes StampPatternCommand on tBrush
        canvas.rightClick(); // Executes MosaicEffectCommand on tBrush
    }
}
```

5. Explanation of how this design solves the problem

The Command Pattern solves the problem by providing the necessary layers of indirection:

- **Encapsulation:** Every painting technique (`smoothStroke`, `splattering`, etc.) paired with a specific brush instance (`DigitalBrush`) is wrapped into a concrete `Command` object. This turns an action request (a method call on a `Receiver`) into an object that can be stored, passed around, and executed later.
- **Parameterization, i.e. Setup:** The `PaintingCanvas` (the Invoker) has slots (`primaryTechnique` and `secondaryTechnique`) that hold generic Command objects. The client (`ArtistSetup`) creates the desired command objects binds them to the currently selected brush (`Receiver`), and then **dynamically assigns** them to the canvas slots using the `setPrimaryTechnique()` or the `setSecondTechnique()` methods. This process realizes the dynamic setup required: the artist selects the technique and assigns it to a mouse button.
- **Execution:** When the artist clicks the left button, the `PaintingCanvas` only needs to call `primaryTechnique.execute()`. The `Canvas` does not know or care if this executes `smoothStroke()` on a `DigitalBrush` or `stampPattern()` on a `TextureBrush`.

  This structure allows the application to dynamically change the entire painting mechanism (the brush and the method) by swapping out the Command objects, while the Invoker (`PaintingCanvas`) remains unchanged and unaware of the details.

6. SOLID Principles Evaluation

- **Single Responsibility Principle (SRP): Adhered to.** `Brush` and its subclasses (`Receiver`) are responsible only for implementing the techniques. The `Command` objects are responsible only for binding a technique to a receiver. The `PaintingCanvas` (`Invoker`) is responsible only for invoking the `execute()` method.
- **Open/Closed Principle (OCP): Adhered to.** Adding a new painting technique (e.g., Airbrush) only requires creating a new `AirbrushCommand` class; the `PaintingCanvas` (Invoker) remains closed to modification.
- **Liskov Substitution Principle (LSP): Adhered to.** Any `Command` implementation (for example, `SmoothStrokeCommand` or `SplatterCommand`) can be substituted in the `PaintingCanvas`'s command slots without altering the canvas's functionality.
- **Interface Segregation Principle (ISP): Adhered to.** The `Command` interface is minimal, defining only the `execute()` method. There are no unnecessary methods forced upon the implementers.
- **Dependency Inversion Principle (DIP): Adhered to.** The high-level `PaintingCanvas` (Invoker) depends on the `Command` **abstraction**, not on low-level concrete commands (like `SplatterCommand`).

Question 3: Brush Undo/Redo

1. Name of Design Pattern(s)

Memento

2. Justification

Memento Pattern captures and externalizing the internal state of the `PaintingCanvas` object (`byte[]`) without violating encapsulation. The `Memento` object stores the `byte[]` array, which can be restored later.

This pattern addressed the desing constraint of saving only what is necessary. Memento allows us to capture *o*nly the necessary state (`byte[]`) without cloning the complex `PaintingCanvas` object. Saving previous image as memento will help the command pattern to perform undo operations seamlessly.

Encapsulation: The `Memento` pattern separates the state management logic into a dedicated `Memento` object (`CanvasMemento`). Only the `PaintingCanvas` (`Originator`) can access the `Memento`'s state, preventing other objects from tampering with the stored data.

Alternative Pattern Comparison

- **Prototype Pattern:** This was explicitly disallowed for a good reason. While cloning is simple, it would copy unnecessary state (caches, UI references) and violate the constraint of focusing only on the `byte[]`. Memento targets specific, necessary state saving.
- **Command Pattern (without memento):** (OPTIONAL...) Using only the `Command` pattern is not sufficient. It is used to drive the undo/redo mechanism, the `Memento` is necessary to actually store the state. The `Command` pattern alone cannot restore the object's previous state.

3. UML Class Diagram

PaintingCanvas is the originator that creates a Memento and an external caretaker saves them.

There are many alternatives. Here are two.

1. Command is caretaker. Each Command object saves a Memento and redo is restoring the PaintingCanvas.
   - This design is not optimal since Command objects need a reference to PaintingCanvas to call `saveState()` before `receiver.action()` which is the painting technique. Also, `undo` will call the `PaintingCanvas.restore(memento)`, which means PaintingCanvas and Command are tightly coupled. (does not adhere to SOLID)
   - An alternative is, HistoryManager has a reference to PaintingCanvas. Its `execute(Command)` method will take a Command object, first calls `canvas.saveState()` to get the memento, than put it into the Command. This design is not ideal either as it puts one more method to Command interface. Then, the command object should return that memento back to the HistoryManager so that it can call PaintingCanvas to restore. (consider SOLID again)
2. HistoryManager is the caretaker of Memento and has a reference to PaintingCanvas. Command is decoupled from PaintingCanvas completely and its interface is not polluted. We can implement the history manager with two stacks or an array with an index.
   - Two-stack:
     The `execute(Command)` method asks for memento before executing the command, and then saves the Memento, not the command, in its `undostack`. We need to clear the `redostack` since this new command invalidates that. The `undo` method of the history object will first get a current memento from PaintingCanvas and saves it into `redostack`. Then it pops a memento from `undostack` and calls `PaintingCanvas.restore`. [NOTE: this is different than having commands in the stacks, we need to save the current memento before doing undo.] The `redo` is similar to execute: first

get a memento from PaintingCanvas and push it into `undostack` (this is the current state before redo).Then pop a memento from `redostack` and restore the canvas with that memento.

- Array and index :
  Before any command executes, the history manager asks PaintingCanvas for a memento and stores it at index 0 of the array.
  The `execute(Command)` method asks for memento to save it at ++index. This design uses post-action mementos. The `undo` method of the history object decrements the index and restores the PaintingCanvas with the current memento. The `redo` is incrementing the index, gets that memento and restores the canvas with that memento. (do the index checks in all)

Making the HistoryManager as the caretaker is a better choice in terms of SOLID and dependencies. Among these solutions, using array and index is more effective.

Here I give a UML Class diagram, which cannot show the dynamic flow. A better diagram would be a sequence or collaboration diagram. The dynamic behavior is given in the code in the next section.

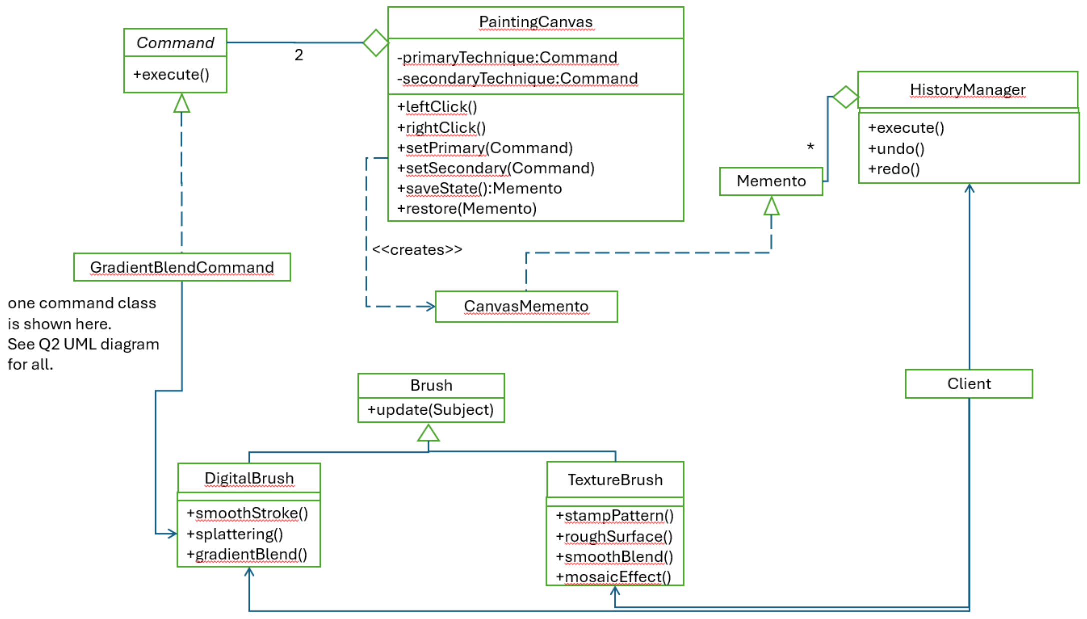
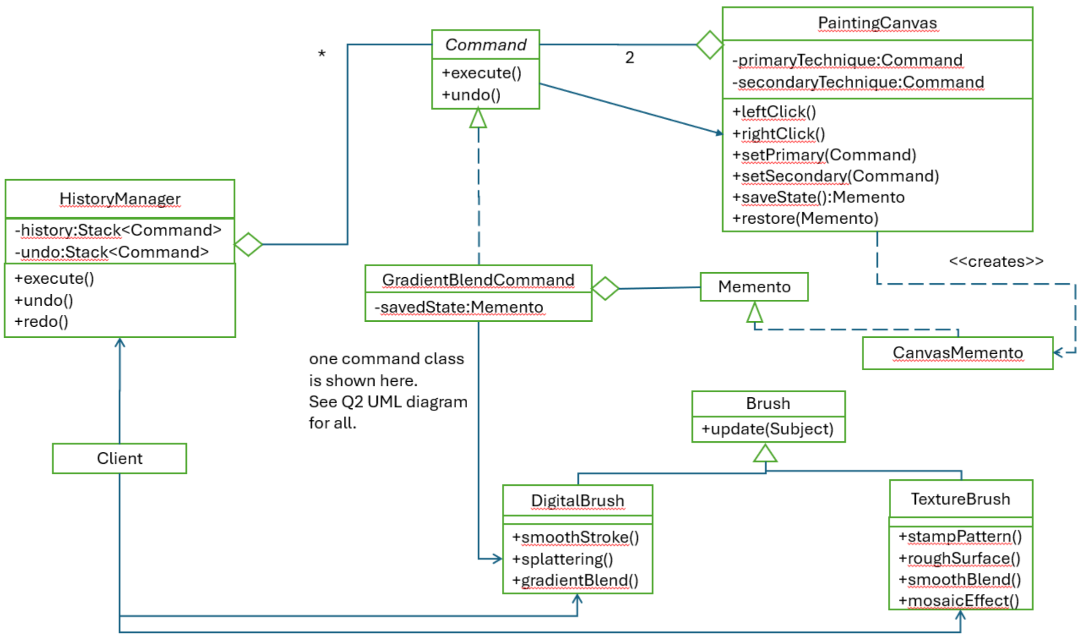


Figure 4: UML Class Diagram for Memento Pattern, HistoryManager is the caretaker.

Diagram 5 is not a good design. Preferred one is the Diagram 4.
Key Participants in Memento:

- **Originator (`PaintingCanvas`):** The object whose state is saved. It creates the `Memento` (`createMemento()`) and uses a `Memento` to restore its state (`restoreMemento()`). Holds the `byte[]` state.
- **Memento (`CanvasMemento`):** Stores the internal state of the `Originator` (`byte[]`). It has a restricted interface; only the `Originator` can access its contents.
- **Caretaker (`HistoryManager`):** Responsible for keeping the `Memento`s safe. It never operates on the `Memento`'s contents. Manages the undo/redo history stacks or the history array.



BAD!

Figure 5: Bad solution: Command is the caretaker for Memento Pattern introduces many dependencies.

4. Java Code

The code will focus on the Memento implementation and its interaction with the Command and the `PaintingCanvas` (Originator).

```java
// Public marker interface for Memento (visible to Caretaker)
interface Memento {}


// --- 1. Originator (PaintingCanvas) and Private Memento ---

// Originator: The object whose state is saved
class PaintingCanvas {
    private Command primaryTechnique;
    private Command secondaryTechnique;

    // The critical state that must be saved for undo/redo
    private byte[] artworkState;
    // Other complex fields that should NOT be cloned
    private int uiReferenceID = 12345;

    public PaintingCanvas(int size) {
        artworkState = new byte[size];
    }

    // Private Inner Memento Class: Enforces encapsulation
    private class CanvasMemento implements Memento {
        private final byte[] state;

        // Private constructor for encapsulation
        private CanvasMemento(byte[] state) {
            // Deep copy for immutability
            this.state = Arrays.copyOf(state, state.length);
        }

        // Internal getter used only by the outer class
        private byte[] getSavedState() {
            return Arrays.copyOf(this.state, this.state.length);
        }
    }

    // creating the memento
    public Memento saveState() {
        System.out.println("[PaintingCanvas] Saving state." );
        return new CanvasMemento(artworkState);
    }

    // restore state using the memento
    public void restore(Memento memento) {
        if(!(memento instanceof CanvasMemento)) {
                System.err.println("wrong memento, cannot restore previous state");
                return;
        }
        // Must cast to the private inner class type to access its internal state
        CanvasMemento canvasMemento = (CanvasMemento) memento;
        artworkState = canvasMemento.getSavedState();
        System.out.println("[PaintingCanvas] Restored state from Memento. Current size: " + artworkState.length);
    }

    // Business logic method (modifies byte[])
    public void applyStroke() {
        System.out.println(" A Brush instance is calling this method to make a stroke");
        //simulating the state change via random assignment
        int newLength = artworkState.length + (int)(Math.random() * 5 + 1);
        this.artworkState = new byte[newLength];
        this.artworkState[0] = (byte)newLength;
    }

    public void setPrimary(Command command) {
        this.primaryTechnique = command;
    }

    //using command via manager
    public void leftClick(HistoryManager manager) {
        if (primaryTechnique != null) {
            System.out.print("[Left Click] -> ");
            // Use manager to execute and save the command
            manager.execute(primaryTechnique);
        } else {
            System.out.println("No primary technique assigned.");
        }
    }

    public void setSecondary(Command command) {
        this.secondaryTechnique = command;
    }

    //using command via manager
    public void rightClick(HistoryManager manager) {
        if (secondaryTechnique != null) {
            System.out.print("[Right Click] -> ");
            // Use manager to execute and save the command
            manager.execute(secondaryTechnique);
        } else {
            System.out.println("No secondary technique assigned.");
        }
    }

}

// --- 2.  Caretaker: Manages the Command history----
// HistoryManager with shift-left array
class HistoryManager {
    private Memento[] history;
    private int capacity;
    private int size;   // number of valid entries
    private int index;  // current position

    private PaintingCanvas canvas;

    public HistoryManager(PaintingCanvas canvas, int capacity) {
        this.canvas = canvas;
        this.capacity = capacity;
        history = new Memento[capacity];
        size = 1;
        index = 0;
        // seed initial state
        history[0] = canvas.saveState();
    }

    public void execute(Command cmd) {
        // run command
        cmd.execute(canvas);

        // capture new state
        Memento newState = canvas.saveState();

        if (size < capacity) {
            history[++index] = newState;
            size = index + 1;
        } else {
            // shift left to drop oldest
            for (int i = 1; i < capacity; i++) {
                history[i - 1] = history[i];
            }
            history[capacity - 1] = newState;
            index = capacity - 1;
            size = capacity;
        }
    }

    public void undo() {
        if (index > 0) {
            canvas.restore(history[--index]);
        }
    }

    public void redo() {
        if (index < size - 1) {
            canvas.restore(history[++index]);
        }
    }
}



// --- 3. Command Components and the Receivers are the same as Q2 implementation ---


// --- 4. Client Demonstration ---
class Client {
    public static void main(String[] args) {
        PaintingCanvas canvas = new PaintingCanvas(500);
        DigitalBrush brush = new DigitalBrush();
        HistoryManager manager = new HistoryManager();

        Command cmd1 = new GradientBlendCommand( brush);
        Command cmd2 = new SplatteringCommand( brush);

        // Set up the primary technique and stroke
        System.out.println("\n--- Performing Stroke 1 ---");
        canvas.setPrimary(cmd1);
        canvas.leftClick(manager);

        // Assign Stroke 2 to the primary technique and execute
        System.out.println("\n--- Performing Stroke 2 ---");
        canvas.setPrimary(cmd2);
        canvas.leftClick(manager);

        // --- Performing Undo ---
        System.out.println("\n--- Performing Undo (Stroke 2) ---");
        manager.undo();

        // --- Performing Redo ---
        System.out.println("\n--- Performing Redo (Stroke 2) ---");
        manager.redo();

        // --- Performing Undo (Stroke 2 again) ---
        System.out.println("\n--- Performing Undo (Stroke 2 again) ---");
        manager.undo();

        // --- Performing Undo (Stroke 1) ---
        System.out.println("\n--- Performing Undo (Stroke 1) ---");
        manager.undo();
    }
}

```

5. Explanation of how this design solves the problem

Memento gives a safe and encapsulated way to store and restore the `PaintingCanvas` state.

- The Memento only stores a deep copy of the `byte[]` array, ignoring complex, non-essential data (caches, UI references), thus satisfying the design constraint. There is no access to the `byte [ ]` array, a sensitive information, to any object other than the memento and the originator.
- `HistoryManager` is the caretaker of `Memento`. Even though it has a reference to `PaintingCanvas`, history manager cannot see the internal representation of the canvas.

(Either explain this part in the design or here. Since this is the place to explain my design, I put it here as well. –antipattern copy/paste programming:)

We can implement the history manager with two stacks or an array with an index.

- Two-stack:
  The `execute(Command)` method asks for memento before executing the command, and then saves the Memento, not the command, in its `undostack`. We need to clear the `redostack` since this new command invalidates that. The `undo` method of the history object will first get a current memento from PaintingCanvas and saves it into `redostack`. Then it pops a memento from `undostack` and calls `PaintingCanvas.restore`. [NOTE: this is different than having commands in the stacks, we need to save the current memento before doing undo.] The `redo` is similar to execute: first get a memento from PaintingCanvas and push it into `undostack` (this is the current state before redo).Then pop a memento from `redostack` and restore the canvas with that memento.
- Array and index :
  Before any command executes, the history manager asks PaintingCanvas for a memento and stores it at index 0 of the array.
  The `execute(Command)` method asks for memento to save it at ++index. This design uses post-action mementos. The `undo` method of the history object decrements the index and restores the PaintingCanvas with the current memento. The `redo` is incrementing the index, gets that memento and restores the canvas with that memento. (do the index checks in all)

This design achieves complete decoupling: the `HistoryManager` knows *when* to save and restore, but not *what* is being saved, and the `PaintingCanvas` maintains control over its state.

6. SOLID Principles Evaluation

- **Single Responsibility Principle (SRP): Adhered to.**
  - The **PaintingCanvas** (`Originator`) manages its current state (`byte[]`) and knows how to create and restore `Mementos`.
  - The `CanvasMemento` (`Memento`) is a passive object whose sole responsibility is to hold the saved state.
  - The `HistoryManager` (`Caretaker`) is responsible for managing the memento collection and be a caretaker.
- **Open/Closed Principle (OCP): Adhered to.** The `HistoryManager` (`Caretaker`) is closed to modification. Adding a new state component to the `PaintingCanvas` (e.g., a ”color history”) only requires changes to the `PaintingCanvas` and the `CanvasMemento`. The core history management logic remains unchanged.
- **Liskov Substitution Principle (LSP): Less Applicable.** The Memento pattern is defined by the three roles (`Originator`, `Memento`, `Caretaker`) and encapsulation, not primarily by polymorphic substitution through inheritance. We rely on the Command pattern in the execution chain for LSP adherence, not on the Memento participants themselves.
- **Interface Segregation Principle (ISP): Adhered to.** The pattern enforces segregation through **restricted access**. The `CanvasMemento`'s state access methods are typically restricted (e.g., package-private or internal access) to the `PaintingCanvas` (`Originator`). This ensures the `HistoryManager` (`Caretaker`) does not depend on methods for state access that it neither needs nor should use.
- **Dependency Inversion Principle (DIP): Adhered to.** The high-level `HistoryManager` (`Caretaker`) depends on the abstract `CanvasMemento` abstraction to hold state. It is decoupled from the low-level `PaintingCanvas` implementation and its specific data structure (`byte[]`), ensuring the history logic is portable and robust.