Formating Instructions: Please use this provided LaTeX template to complete your homework. When including figures such as UMLs, it is highly encouraged to use tools (graphviz, drawio, tikz, etc..). Figures may be hand draw, however, these may not receive credit if the grader cannot read it. For ease of grading, when including code please have it as part of the same pdf as the question while also including correct formatting/indent, preferably syntax highlighting. Latex includes the minted, or lstlisting package as a helpful tool. For this assignment all code must be full code (no pseudo-code) and be written in either Java or C++. However, implements may ignore all logic not relevant to the design pattern with simple print out statements of "[BLANK] logic done here"

Question Instructions: In this homework assignment, you will apply one or more Behavioral patterns discussed in the lectures.
This is a group assignment that requires two students per group.
For each question:

1. Give the name of the design pattern(s) you are applying to the problem.
2. Present your reasons why this pattern will solve the problem. Please be specific to the problem and do not give general applicability statements. If there is an alternative pattern, explain why you preferred this one..
3. Show you design with a UML class diagram. If the pattern collaborations would be more visible with another diagram (e.g. sequence diagram), give that diagram as well.
   (a) Your diagram should show every participant in the pattern including the pattern related methods.
   (b) In pattern related classes, give the member (method and attribute) names that play a role in the pattern and effected by the pattern. Optionally, include the member names mentioned in the question. You are encouraged to omit the other methods and fields.
   (c) For the non-pattern related classes, you are not expected to give detailed class names etc. You may give a high-level component, like "UserInterface" or "DBManagement"
4. Give Java or C++ code for your design showing how you have implemented the pattern.
   (a) Pattern related methods and attributes should appear in the code
   (b) Client usage of the pattern should appear in the code
   (c) Non-pattern related parts of the methods could be a simple print. (e.g. "System.out.println()", "cout")
5. Explain how this design solves the problem.
6. Evaluate your design with respect to SOLID principles. Each principle should be addressed, if a principle is not applicable to the current pattern, say so.

1. (18 points) We have one Profile instance for each user of the application. The Profile class contains personal information such as name, picture, etc. In this application, users post their online meetings announcement, and they also post links to their online meetings.

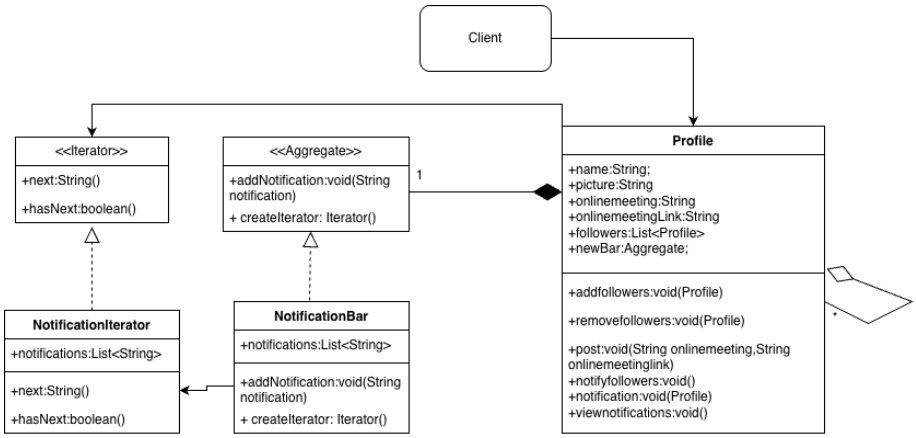
   Users may follow other users. Implement the **observer** pattern with **pull** method for this purpose. When a user posts an announcement or meeting link, we want all the followers to get informed.

   New functionality:
   The application maintains a collection of notifications. There is a notification bar for each user's page. This notification bar displays all the notifications it received from all of its followees. We want the user to iterate over the notifications independent of how the collection is represented.

   Suggest a design pattern to support the new functionality. (parts 1 and 2 on the cover page)

   Give code and UML for both patterns. You may give them separately or combined. (rest of the parts on the cover page)

   1. Using Iterator and Observer
   2. The Observer is used to notify the followers whenever a new meeting information is posted. The Iterator is used to iterate each user's notification bar and display them.
   3. UML:
   4. Code:




```
import java.util.ArrayList;
import java.util.List;

interface Aggregate
{
    void addNotification(String notification);
    Iterator createIterator();
}
class NotificationBar implements Aggregate
{
    List<String> notifications = new ArrayList<>();
```

```
   @Override
    public void addNotification(String notification)
    {
        notifications.add(notification);
    }

    @Override
    public NotificationIterator createIterator()
    {
        return new NotificationIterator(notifications);
    }

}

interface Iterator{
boolean hasNext();
String next();
}

class NotificationIterator implements Iterator {

    List<String> notifications;
    int index = 0;

    public NotificationIterator(List<String> notifications) {
        this.notifications = notifications;
    }

  @Override
    public boolean hasNext() {
        return index < notifications.size();
    }

@Override
    public String next() {
        return notifications.get(index++);
    }
}



class Profile
{

String name, picture;
String onlinemeeting, onlinemeetingLink;
```

```
List<Profile> followers = new ArrayList<>();
Aggregate newbar = new NotificationBar();
public Profile(String name,String picture)
{

    this.name = name;
    this.picture = picture;

}

public void addfollowers(Profile person)
{

followers.add(person);
}

public void removefollowers(Profile person)
{
followers.remove(person);
}


public void post(String onlinemeeting, String onlinemeetingLink)
{

this.onlinemeeting=onlinemeeting;
this.onlinemeetingLink=onlinemeetingLink;
notifyfollowers();
}

public void notifyfollowers()
{
for(Profile person : followers){

person.notifications(this);

}
}

public void notifications(Profile person)
{


String msg = " New post from " + person.name+
" :" + person.onlinemeeting + " at " + person.onlinemeetingLink;
newbar.addNotification(msg);
/*System.out.println(
```

```
                "Notification for " + this.name +
                ": New post from " + person.name +
                " :" + person.onlinemeeting +
                " at " + person.onlinemeetingLink
            );*/
    }

    public void viewNotifications()
    {
    Iterator it = newbar.createIterator();
            System.out.println("\n=== " + name + "'s Notification Bar ===");
            while (it.hasNext()) {
                System.out.println("- " + it.next());
            }
    }

    public static void main(String[] args) {

        Profile Adam = new Profile("Adam","pic1.png");
        Profile James = new Profile("James","pic2.png");
        Profile Nathan = new Profile("Nathan","pic3.png");
        Adam.addfollowers(James);
        Adam.addfollowers(Nathan);
        Adam.post("scrumcall", "googlemeet.com");

        James.addfollowers(Adam);
        James.post("central park meetup", "googlemaps.com");


        James.viewNotifications();
        Adam.viewNotifications();

        James.addfollowers(Nathan);
        James.post("central park meetup2", "googlemaps.com");
        Nathan.viewNotifications();

    }

    }
```

5. The Observer pattern is used so that each Profile keeps a list of followers. When a user posts, the subject calls ‘notifyfollowers‘, which invokes ‘notifications(this)‘ on each follower. Followers then pulls the post details (meeting title and link) from the subject object, so they are always informed of new announcements or links without tight coupling.

   For the new functionality, each Profile has a ‘NotificationBar‘ that implements an ‘Aggregate‘ interface. It stores notifications internally but exposes a ‘createIterator‘ method. The Concrete Iterator walks the collection, letting the user iterate through notifications

without depending on the underlying representation.

6. SOLID Principles:
   - Single Responsibility Principle :
     NotificationBar and NotificationIterator each have a single responsibility : Iterating through the list and adding notifications. It does not apply to Profile, as it has multiple responsibilties.
   - Open/Closed Principle :
     The design is open for extension but closed for modification: The internal storage in NotificationBar can be changed without changing client code because iteration goes through the iterator
   - Liskov Substitution Principle:
     Any class implementing Iterator or Aggregate could be substituted.
   - Interface Segregation Principle:
     The interface only has essential methods and no unnecessary methods.
   - Dependency Inversion Principle :
     Profile depends on the both interface and not on the concerete class.

2. (18 points) We are implementing a digital painting application where the artist creates artwork using various digital brushes. Each brush has different painting techniques implemented as methods.

   Here is an example brush class: DigitalBrush with 3 techniques.

```
public class DigitalBrush {
    public void smoothStroke() {.....}   // painting technique
     public void splattering() {.....}    // painting technique
    public void gradientBlend() {.....}  // painting technique
}
```

   Another example is TextureBrush with 4 techniques:

```
public class TextureBrush {
    public void stampPattern() {.....}    // applies a repeating pattern
    public void roughSurface() {.....}    // creates a textured, rough appearance
    public void smoothBlend() {.....}     // smoothly blends colors with existing artwork
    public void mosaicEffect() {.....}    // creates a tile-like mosaic effect
}
```

   At the beginning of the session, the artist selects a brush and does a setup for mouse keys. First, the artist sets a primary painting technique as one of the methods of the selected brush. The primary technique is activated with the left click on the mouse. Similarly, a secondary painting technique out of the remaining methods of the brush is set. The secondary technique is activated with the right click on the mouse.

   After this setup, the artist starts painting. For example: The artist selects the digital brush, sets smooth stroke as primary technique and splattering as secondary. During the session, when the artist clicks on the left mouse button, we see a smooth stroke with the digital brush. When the artist changes the brush, for example, to a TextureBrush, the setup must be performed again.

   Suggest a design pattern to realize the setting up and painting with the selected brush.

   1. Command

   2. Two reasons: Command pattern decouples Invoker from Receiver. Different "Invokers" (the left and right mouse clicks) need to perform without knowing anything about the operation (like `smoothStroke`) or the "Receiver" (the specific `DigitalBrush` or `TextureBrush` instance).

   It also encapsulates the requests as objects. The client can choose different techniques from the brush classes.

   Another possible design pattern is Strategy, but Strategy design patterns is about encapsulating interchangaeble algorithms for a specific task. In this case, there are many tasks (right click and left click) and many methods (two brushes and a couple of painting techniques)

   3.

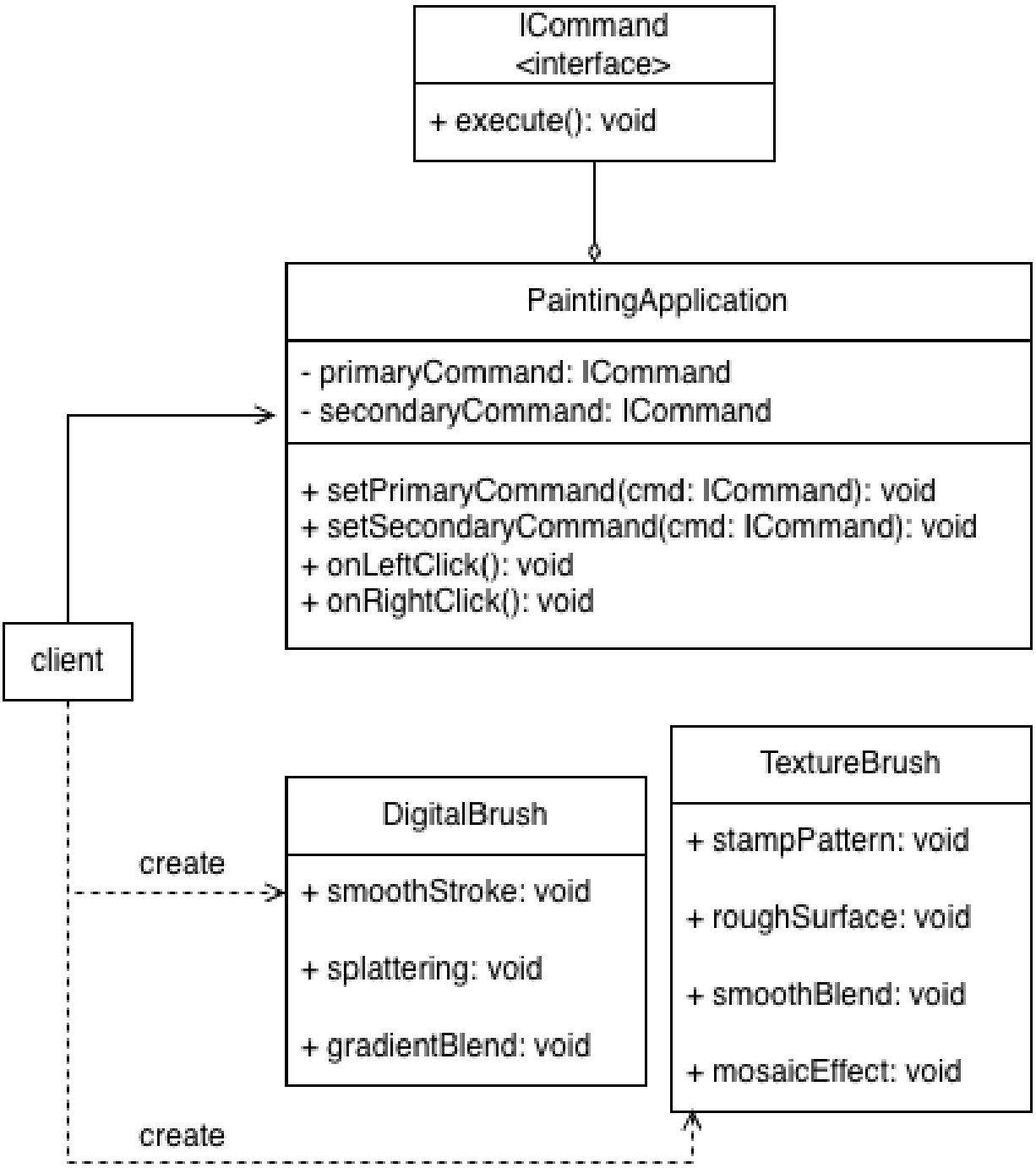


Figure 1: Enter Caption

4.

```
// enable lambda & force single method
@FunctionalInterface
interface ICommand {
    void execute();
}

// Receiver Class 1
// signature must match ICommand.execute
class DigitalBrush {
    public void smoothStroke() {
        System.out.println("DigitalBrush: Painting a smooth stroke.");
    }

    public void splattering() {
        System.out.println("DigitalBrush: Creating a splatter effect.");
    }

    public void gradientBlend() {
        System.out.println("DigitalBrush: Blending with a gradient.");
    }
}

// Receiver Class 2
// signature must match ICommand.execute
class TextureBrush {
    public void stampPattern() {
        System.out.println("TextureBrush: Stamping a repeating pattern.");
    }

    public void roughSurface() {
        System.out.println("TextureBrush: Creating a rough surface texture.");
    }

    public void smoothBlend() {
        System.out.println("TextureBrush: Blending colors smoothly.");
    }

    public void mosaicEffect() {
        System.out.println("TextureBrush: Applying a mosaic tile effect.");
    }
}

// invoker
class PaintingApplication {
    private ICommand primaryCommand;
```

```
    private ICommand secondaryCommand;

    public void setPrimaryCommand(ICommand command) {
        this.primaryCommand = command;
    }

    public void setSecondaryCommand(ICommand command) {
        this.secondaryCommand = command;
    }

    public void onLeftClick() {
        if (primaryCommand != null)
            primaryCommand.execute();
    }

    public void onRightClick() {
        if (secondaryCommand != null)
            secondaryCommand.execute();
    }
}

// Client (Using Lambdas and Method References)
public class Client2 {
    public static void main(String[] args) {
        PaintingApplication app = new PaintingApplication();

        DigitalBrush digitalBrush = new DigitalBrush();

        // pass method references directly
        app.setPrimaryCommand(digitalBrush::smoothStroke); // app.setPrimaryCommand(()->dig
        app.setSecondaryCommand(digitalBrush::splattering);

        app.onLeftClick();
        app.onRightClick();

        System.out.println("Artist selects TextureBrush...");
        TextureBrush textureBrush = new TextureBrush();

        // re-configure with new method references
        app.setPrimaryCommand(textureBrush::stampPattern);
        app.setSecondaryCommand(textureBrush::mosaicEffect);

        app.onLeftClick();
        app.onRightClick();
    }
}
```

5. Invokers (`paintingApplication`) issue requests without knowing anything about the operation (brushes or techniques) being requested. All the configuration is done in client code. For example, when the artist wants to change brushes, the `Client` just creates a *new* `TextureBrush` receiver and *new* commands, then calls `app.setPrimaryCommand()` again. The `PaintingApplication` is unchanged.

In the code above, we used lambda as command, so we don't have concrete command classes like `smoothStrokeCommand, splatteringCommand`, and so on. The client is responsible for creating different commands and pass those command objects into `PaintingApplication` (the invoker)

6. Single Responsibility Principle: Followed.

DigitalBrush (Receiver) is only responsible for painting logic.

PaintingApplication (Invoker) is only responsible for invoking a command.

Open/Closed Principle: Followed.

The PaintingApplication is closed for modification as long as the operation (left click and right click) reamins the same.

The system is open to extension. You can add a new brush class that follows the same function signature (return void and take no parameter) without touching any existing code.

Liskov Substitution Principle: N/A. Since we use lambda to create concrete command object in the client code, there's no concrete command subclasses.

Interface Segregation Principle: Followed.

The ICommand interface is minimal and has only one method, execute().

Dependency Inversion Principle: Followed.

The high-level PaintingApplication (Invoker) does not depend on the low-level DigitalBrush (Receiver).

3. (16 points) In Question 2, you designed a system to map brush techniques (like smoothStroke() or stampPattern()) to mouse clicks.

   Every brush stroke modifies a central Canvas object, which holds the entire state of the artwork in a single byte[] (a byte array). When smoothStroke() is executed, it modifies this byte[].

   Extend your design to support multi-level undo and redo operations (e.g., for the last 10 strokes). Your solution must be able to restore the Canvas's byte[] to the exact state it was in before an operation was executed.

   Note: A "prototype-like" solution, such as simply cloning the entire Canvas object, is not an acceptable design. The Canvas object itself is complex and may contain many other elements (like caches or UI references). Your solution should focus on explicitly managing the byte[] state.

   1. Memento

   2. The Memento design pattern often paris up with the Command pattern to capture an application's state without violating encapsulation.

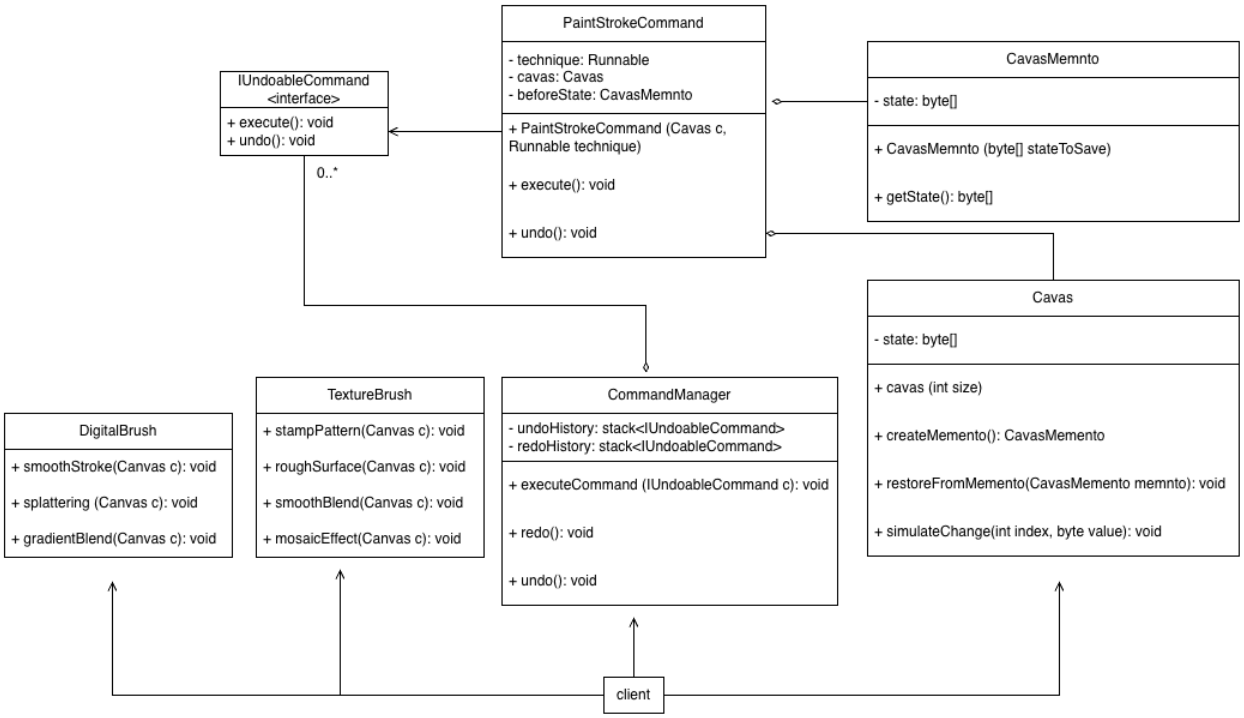
   The Canvas will be the Originator. It will have a method (createMemento()) that creates a CanvasMemento object and copies its private byte[] state into it.

   The CanvasMemento is the Memento. It is a simple, dumb object whose only job is to hold a copy of that byte[] state.

   The PaintStrokeCommand (our ConcreteCommand) will act as the Caretaker. In its execute() method, it will first ask the Canvas for a memento (the before state) and store it as a private field . It will then proceed with the stroke.

   The command's undo() method will then use this stored memento to restore the Canvas to its previous state. This respects the constraint of only managing the byte[] and not cloning the Canvas itself.

3.


PaintStrokeCommand
- technique: Runnable
- cavas: Cavas
- beforeState: CavasMemnto
+ PaintStrokeCommand (Cavas c, Runnable technique)
+ execute(): void
+ undo(): void
CavasMemnto
- state: byte[]
+ CavasMemnto (byte[] stateToSave)
+ getState(): byte[]
IUndoableCommand <interface>
+ execute(): void
+ undo(): void
0..*
Cavas
- state: byte[]
+ cavas (int size)
+ createMemento(): CavasMemento
+ restoreFromMemento(CavasMemento memnto): void
+ simulateChange(int index, byte value): void
TextureBrush
+ stampPattern(Canvas c): void
+ roughSurface(Canvas c): void
+ smoothBlend(Canvas c): void
+ mosaicEffect(Canvas c): void
CommandManager
- undoHistory: stack<IUndoableCommand>
- redoHistory: stack<IUndoableCommand>
+ executeCommand (IUndoableCommand c): void
+ redo(): void
+ undo(): void
DigitalBrush
+ smoothStroke(Canvas c): void
+ splattering (Canvas c): void
+ gradientBlend(Canvas c): void
client

4.

```
import java.util.Stack;

interface IUndoableCommand {
    void execute();

    void undo(); // New method
}

/**
 * The Originator (Canvas)
 * Holds the state and knows how to save/restore itself.
 */
class Canvas {
    private byte[] state;

    public Canvas(int size) {
        this.state = new byte[size];
        System.out.println("Canvas initialized with " + size + " bytes.");
    }

    // Creates a Memento and copies the current state into it.
    public CanvasMemento createMemento() {
        // Return a memento with a DEEP COPY of the state.
        return new CanvasMemento(this.state.clone());
    }

    // Restores its state from a Memento.
    public void restoreFromMemento(CanvasMemento memento) {
        // Get the state from the memento and make a DEEP COPY.
        this.state = memento.getState().clone();
        System.out.println("Canvas restored from Memento.");
    }

    // Helper to simulate a change
    public void simulateChange(int index, byte value) {
        this.state[index] = value;
    }
}

// The Memento
class CanvasMemento {
    private final byte[] state;

    public CanvasMemento(byte[] stateToSave) {
        this.state = stateToSave;
    }
```

```
    public byte[] getState() {
        return this.state;
    }
}

/**
 * The Receiver Classes (The Brushes)
 * Their methods are passed as lambda functions
 */
class DigitalBrush {
    // The technique now modifies the canvas state it is given.
    public void smoothStroke(Canvas canvas) {
        System.out.println("DigitalBrush: Painting a smooth stroke.");
        canvas.simulateChange(0, (byte) 1);
    }

    public void splattering(Canvas canvas) {
        System.out.println("DigitalBrush: Creating a splatter effect.");
        canvas.simulateChange(1, (byte) 2);
    }

    public void gradientBlend(Canvas canvas) {
        System.out.println("DigitalBrush: Blending with a gradient.");
        canvas.simulateChange(1, (byte) 3);

    }
}

class TextureBrush {
    public void stampPattern(Canvas canvas) {
        System.out.println("TextureBrush: Stamping a repeating pattern.");
        canvas.simulateChange(2, (byte) 1);
    }

    public void roughSurface(Canvas canvas) {
        System.out.println("TextureBrush: Creating a rough surface texture.");
        canvas.simulateChange(2, (byte) 2);
    }

    public void smoothBlend(Canvas canvas) {
        System.out.println("TextureBrush: Blending colors smoothly.");
        canvas.simulateChange(2, (byte) 3);

    }

    public void mosaicEffect(Canvas canvas) {
```

```
        System.out.println("TextureBrush: Applying a mosaic tile effect.");
        canvas.simulateChange(2, (byte) 4);
    }
}

/**
 * The ConcreteCommand
 * also acts as the Memento's Caretaker.
 */
class PaintStrokeCommand implements IUndoableCommand {
    private Runnable technique; // The lambda (e.g., () -> digitalBrush.smoothStroke(canvas
    private Canvas canvas; // The Originator
    private CanvasMemento beforeState; // The Memento

    public PaintStrokeCommand(Canvas canvas, Runnable technique) {
        this.canvas = canvas;
        this.technique = technique;
    }

    @Override
    public void execute() {
        // step 1: Save the "before" state *before* executing.
        this.beforeState = canvas.createMemento();

        // step 2: Execute the command (run the lambda).
        this.technique.run();
    }

    @Override
    public void undo() {
        if (this.beforeState != null) {
            canvas.restoreFromMemento(this.beforeState);
        }
    }
}

/**
 * The Invoker (CommandManager)
 * Manages the undo/redo history stacks.
 */
class CommandManager {
    private Stack<IUndoableCommand> undoHistory = new Stack<>();
    private Stack<IUndoableCommand> redoHistory = new Stack<>();

    public void executeCommand(IUndoableCommand command) {
        System.out.println("--- Executing Command ---");
        command.execute();
```

```
        undoHistory.push(command);
        redoHistory.clear();
    }

    public void undo() {
        System.out.println("--- Undoing ---");
        if (undoHistory.isEmpty()) {
            System.out.println("Nothing to undo.");
            return;
        }
        IUndoableCommand command = undoHistory.pop();
        command.undo();
        redoHistory.push(command);
    }

    public void redo() {
        System.out.println("--- Redoing ---");
        if (redoHistory.isEmpty()) {
            System.out.println("Nothing to redo.");
            return;
        }
        IUndoableCommand command = redoHistory.pop(); // Get the last undone command

        // Re-executing the command will save a *new* 'before' memento
        // (of the state just undid to) and then re-apply the stroke.
        command.execute();

        undoHistory.push(command);
    }
}

public class Memento2 {
    public static void main(String[] args) {
        // Setup all the main objects
        Canvas canvas = new Canvas(1024);
        DigitalBrush digitalBrush = new DigitalBrush();
        TextureBrush textureBrush = new TextureBrush();
        CommandManager manager = new CommandManager();

        // case 1: Smooth Stroke (DigitalBrush) ---
        // Create the lambda for the technique
        Runnable smoothStrokeLambda = () -> digitalBrush.smoothStroke(canvas);
        // Create the command object, passing the lambda and canvas
        IUndoableCommand cmd1 = new PaintStrokeCommand(canvas, smoothStrokeLambda);
        manager.executeCommand(cmd1);

        // case 2: Stamp Pattern (TextureBrush) ---
```

```
        Runnable stampPatternLambda = () -> textureBrush.stampPattern(canvas);
        IUndoableCommand cmd2 = new PaintStrokeCommand(canvas, stampPatternLambda);
        manager.executeCommand(cmd2);

        // case 3: Splatter (DigitalBrush) ---
        Runnable splatterLambda = () -> digitalBrush.splattering(canvas);
        IUndoableCommand cmd3 = new PaintStrokeCommand(canvas, splatterLambda);
        manager.executeCommand(cmd3);

        manager.undo(); // undo "Splatter". Canvas restored to after "Stamp".
        manager.undo(); // undo "Stamp". Canvas restored to after "Smooth".

        manager.redo(); // redo "Stamp".

        manager.undo(); // undo "Stamp" again.

    }
}
```

5.

- **Multi-Level Undo:** The `CommandManager` uses a `Stack<IUndoableCommand>` for its `undoHistory`. Every time `executeCommand` is called, the command is pushed onto this stack. This allows for a virtually unlimited (up to the stack size) number of undo levels.
- **Redo:** The `CommandManager` uses a `redoHistory` stack. When an item is undone, it's moved from the `undoHistory` to the `redoHistory` . When `redo()` is called, the command is moved back .
- **No Canvas Cloning:** This design fulfills the constraint. We did not clone the `Canvas` object. We only ask the `Canvas` to give us its state, which it does by cloning its internal `byte[]` in a `CanvasMemento`. The command and manager only ever see the `Canvas` and `CanvasMemento`, never the complex internals.

6.

Single Responsibility Principle: Followed.

Canvas: Manages its own byte[] state.

CanvasMemento: Only holds state.

PaintStrokeCommand: Only knows when to save/restore state and what technique to execute.

CommandManager: Only manages the undo/redo history.

Open/Closed Principle: Followed.

The CommandManager is completely closed for modification, and at the same time open to extension by creating new classes that implement IUndoableCommand.

Liskov Substitution Principle: Followed.

The CommandManager's stacks hold IUndoableCommand. Any object that implements this interface (like PaintStrokeCommand) can be substituted, and the Manager's undo/redo logic will work.

Interface Segregation Principle: Followed.

The IUndoableCommand interface is small and specific (execute, undo). Clients (the CommandManager) depend only on what they need.

Dependency Inversion Principle: Followed.

The high-level CommandManager does not depend on the low-level PaintStrokeCommand. Both depend on the abstraction IUndoableCommand.

Table 1: Grading Rubric for **16** points questions

1 (1 point)	0	missing or incorrect
	+1	correct pattern
2 (1 point)	0	missing
	+1	the reason provided correctly describes an advantage of the pattern and is specifically beneficial to this scenario
3 (5 points)	0	missing
	+2	includes all participants (including client) that play a role in the pattern
	+2	all class relations are correct
	+1	includes all class members that are related to the pattern
4 (5 points)	0	missing
	+1	**Completeness**: Includes all pattern-related methods and attributes from the design.
	+2	**Client Usage**: Includes code that correctly demonstrates the client's use of the pattern.
	+2	**Correctness**: (This sub-criterion is worth 0, 1, or 2 points) **0:** Fundamentally incorrect or missing. **+1:** Implements and uses some pattern-related methods and/or with some mistakes. **+2:** Correctly implements and uses all pattern-related methods.
5 (2 points)	0	missing or not enough detail
	+2	uses the specific names and roles of their designed classes and their collaborations to demonstrate how the problem is systematically solved.
6 (2 points)		**SOLID Principles**: (This criterion is worth 0, 1, or 2 points)
	0	missing.
	+1	correctly identifies at least two relevant SOLID principles and accurately assesses whether the design adheres to them.
	+2	correctly identifies all of the relevant SOLID principles and accurately assesses whether the design adheres to them.

Table 2: Grading Rubric for **18** points questions

1 (1 point)	0	missing or incorrect
	+1	correct pattern
2 (1 point)	0	missing
	+1	the reason provided correctly describes an advantage of the pattern and is specifically beneficial to this scenario
3 (5 points)	0	missing
	+2	includes all participants (including client) that play a role in the pattern
	+2	all class relations are correct
	+1	includes all class members that are related to the pattern
4 (7 points)	0	missing
	+1	**Completeness**: Includes all pattern-related methods and attributes from the design.
	+2	**Client Usage**: Includes code that correctly demonstrates the client's use of the pattern.
	+4	**Correctness**: (This sub-criterion is worth 0, 2, or 4 points) **0:** Fundamentally incorrect or missing. **+2:** Implements and uses some pattern-related methods and/or with some mistakes. **+4:** Correctly implements and uses all pattern-related methods.
5 (2 points)	0	missing or not enough detail
	+2	uses the specific names and roles of their designed classes and their collaborations to demonstrate how the problem is systematically solved.
6 (2 points)		**SOLID Principles**: (This criterion is worth 0, 1, or 2 points)
	0	missing.
	+1	correctly identifies at least two relevant SOLID principles and accurately assesses whether the design adheres to them.
	+2	correctly identifies all of the relevant SOLID principles and accurately assesses whether the design adheres to them.