1. Question 1:

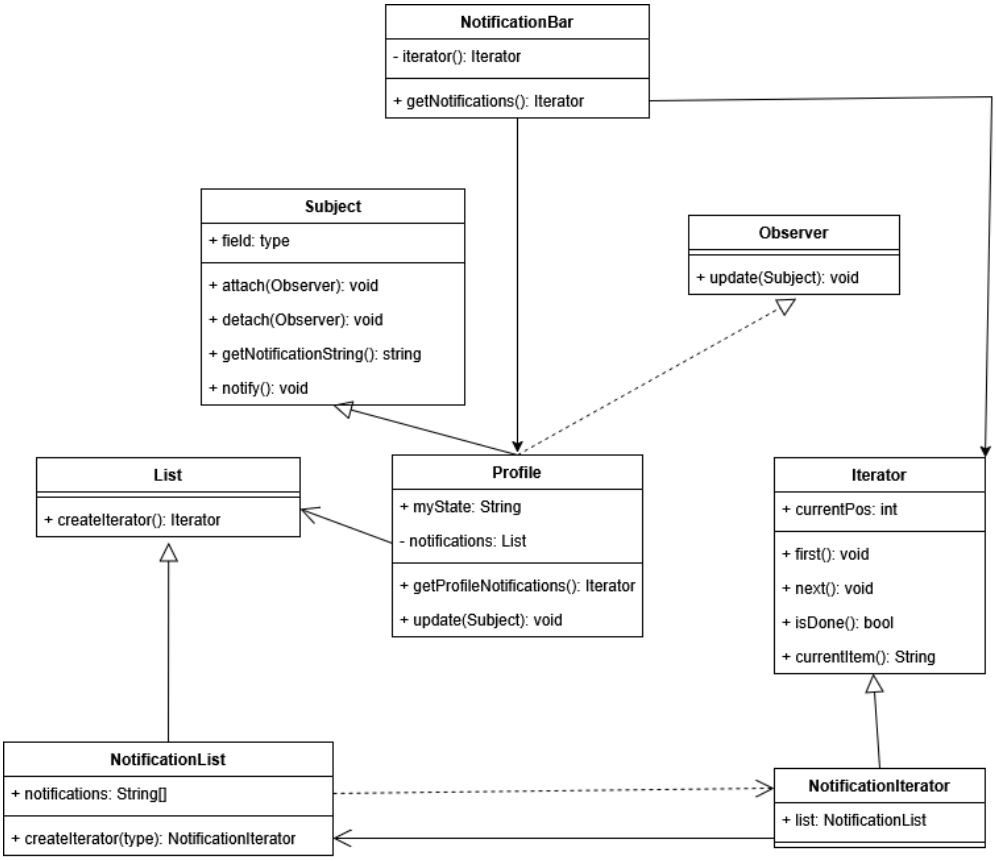
    1. Give the name of the design pattern(s) you are applying to the problem.

       Iterator

    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.

       The Iterator Method allows us to make a specific class focused on iterating through a data structure that way the user will not have to keep track of positioning or other matters related to getting data from the data structure.

    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.



    4. Give Java or C++ code for your design showing how you have implemented the pattern.

```
import java.awt.*;
import java.util.ArrayList;
//CLIENT
public class NotificationBar {
    private Iterator iterator;

    public Iterator getNotifications(Profile p) {
        iterator = p.getProfileNotifications();
        return iterator;
    }

    public static void main(String[] args) {
        Profile p1 = new Profile();
        Profile p2 = new Profile();
        p1.attach(p2);
        p1.notifyObservers();
        p2.notifyObservers();
        NotificationBar nb = new NotificationBar();
        nb.getNotifications(p2);
        while (!nb.iterator.isDone()){
            System.out.println(nb.iterator.getCurrentItem());
            nb.iterator.next();
        }
    }
}

abstract class Subject {
    ArrayList<Observer> observers;

    public Subject() {
        observers = new ArrayList<>();
    }
    public void attach(Observer o) {
        observers.add(o);
    }
    public void detach(Observer o) {
        observers.remove(o);
    }
    public void notifyObservers() {
        for (Observer o: observers) {
            o.update(this);
        }
    }
    public String getNotificationString(){
```

```
        return "Notification";
    }
}

interface Observer {
    public void update(Subject s);
}

class Profile extends Subject implements Observer{
    private String name;
    private Image picture;
    private List notifications;

    public Profile() {
        notifications = new NotificationList(10);
        name = "Default";
        picture = null;
        notifications.addNotification("New Profile created!");
    }

    public void update(Subject s) {
        notifications.addNotification(s.getNotificationString());
    }

    public Iterator getProfileNotifications() {
        return notifications.createIterator();
    }
}

interface Iterator {
    public void first();
    public void next();
    public String getCurrentItem();
    public boolean isDone();
}

class NotificationIterator implements Iterator {
    private int currentPos = 0;
    private NotificationList collection;

    public NotificationIterator(NotificationList nl) {
        collection = nl;
    }
```

```
    public void first() {
        currentPos = 0;
    }

    public void next() {
        currentPos += 1;
    }

    public String getCurrentItem() {
        return collection.getItemAt(currentPos);
    }

    public boolean isDone() {
        return currentPos < collection.size();
    }
}

abstract class List {
    public abstract Iterator createIterator();

    public abstract void addNotification(String p);
}

class NotificationList extends List {
    private String[] notifications;
    private int count = 0;

    public NotificationList(int size) {
        notifications = new String[size];
    }

    public void addNotification(String p) {
        if (count < notifications.length) {
            notifications[count] = p;
            count++;
        }
    }

    public int size() {
        return count;
    }

    public String getItemAt(int pos) {
        if (pos >= 0 && pos < count) {
```

```
            return notifications[pos];
        }
        return null;
    }

    public Iterator createIterator() {
        return new NotificationIterator(this);
    }
}
```

5. Explain how this design solves the problem.
   This design allows us to iterate over the notifications no matter how that collection is represented.

6. Evaluate your design with respect to SOLID principles. Each principle should be address, if a principle is not applicable to the current pattern, say so.
   (a) Single Responsibility Principle

   This design does follow the SRP because every class only has one job. Iterator defines the logic for traversing the collection, NotificationIterator implements that logic specifically for a NotificationList, List defines the logic for creating the Iterator, and NotificationList holds the actual collection and creates its specific Iterator.

   (b) Open-Closed Principle
   The design follows the open-closed principle because we could add different types of Iterators by creating new classes that implement the Iterator interface and/or List interface without having to change any existing classes.

   (c) Liskov Substitution Principle

   The LSP is followed as all children can be a substitute for the Iterator or List class in this implementation and no functionality is lost.

   (d) Interface Segregation Principle

   This design follows the ISP because all subclasses of Iterator and List use all methods.

   (e) Dependency Inversion Principle

   This design follows the DIP because the client only relies on the high-level abstract class Iterator instead of the specific implementations.

2. Question 2:

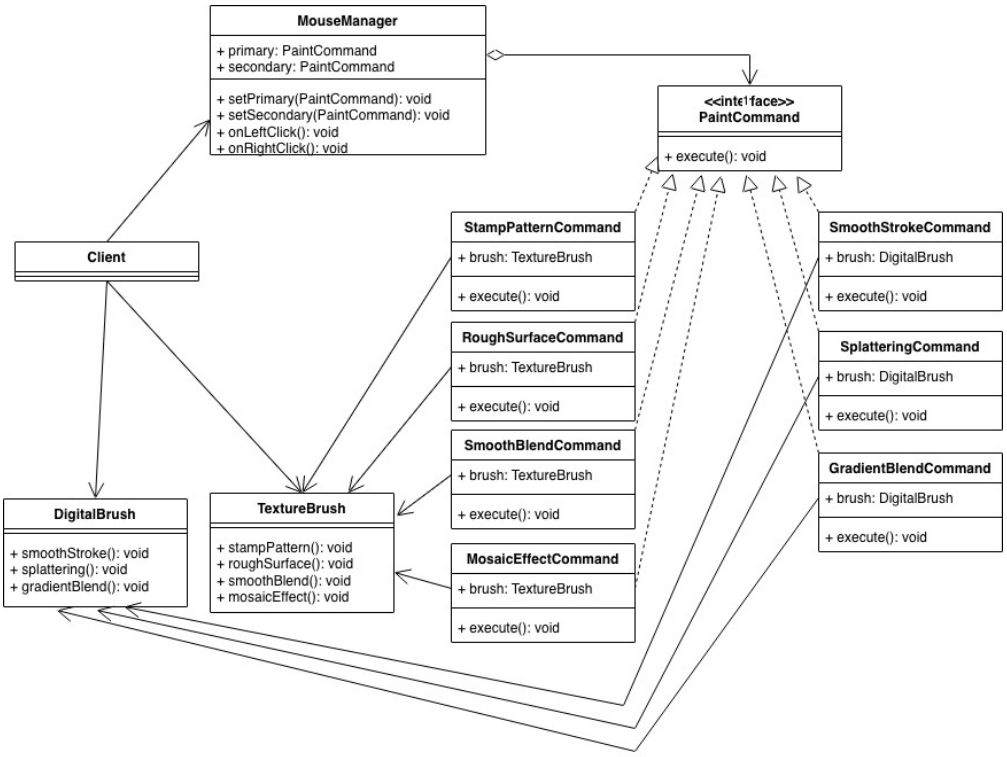
    1. Give the name of the design pattern(s) you are applying to the problem.

       Command

    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..

       The command pattern turns the actions of setting up and painting with a brush into a standalone object that contains all the details about which brush type was selected.

    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.



    4. Give Java or C++ code for your design showing how you have implemented the pattern.

```
interface PaintCommand {
    void execute();
```

```
}

//RECEIVER
class DigitalBrush {
    public void smoothStroke() {
        System.out.println("Applying smooth stroke with digital brush");
    }

    public void splattering() {
        System.out.println("Applying splattering effect with digital brush");
    }

    public void gradientBlend() {
        System.out.println("Applying gradient blend with digital brush");
    }
}

//RECEIVER
class TextureBrush {
    public void stampPattern() {
        System.out.println("Applying stamp pattern with texture brush");
    }
    public void roughSurface() {
        System.out.println("Creating rough surface with texture brush");
    }
    public void smoothBlend() {
        System.out.println("Applying smooth blend with texture brush");
    }
    public void mosaicEffect() {
        System.out.println("Creating mosaic effect with texture brush");
    }
}

//DIGITAL BRUSH COMMANDS
class SmoothStrokeCommand implements PaintCommand {
    private DigitalBrush brush;

    public SmoothStrokeCommand(DigitalBrush brush) {
        this.brush = brush;
    }
    public void execute() {
        brush.smoothStroke();
    }
}
```

```
class SplatteringCommand implements PaintCommand {
    private DigitalBrush brush;

    public SplatteringCommand(DigitalBrush brush) {
        this.brush = brush;
    }
    public void execute() {
        brush.splattering();
    }
}

class GradientBlendCommand implements PaintCommand {
    private DigitalBrush brush;

    public GradientBlendCommand(DigitalBrush brush) {
        this.brush = brush;
    }
    public void execute() {
        brush.gradientBlend();
    }
}

//TEXTURE BRUSH COMMANDS
class StampPatternCommand implements PaintCommand {
    private TextureBrush brush;

    public StampPatternCommand(TextureBrush brush) {
        this.brush = brush;
    }
    public void execute() {
        brush.stampPattern();
    }
}

class RoughSurfaceCommand implements PaintCommand {
    private TextureBrush brush;

    public RoughSurfaceCommand(TextureBrush brush) {
        this.brush = brush;
    }
    public void execute() {
        brush.roughSurface();
    }
```

```
}

class SmoothBlendCommand implements PaintCommand {
    private TextureBrush brush;

    public SmoothBlendCommand(TextureBrush brush) {
        this.brush = brush;
    }
    public void execute() {
        brush.smoothBlend();
    }
}

class MosaicEffectCommand implements PaintCommand {
    private TextureBrush brush;

    public MosaicEffectCommand(TextureBrush brush) {
        this.brush = brush;
    }
    public void execute() {
        brush.mosaicEffect();
    }
}

//INVOKER
class MouseManager {
    private PaintCommand primary;
    private PaintCommand secondary;

    public void setPrimary(PaintCommand pc) {
        this.primary = pc;
    }
    public void setSecondary(PaintCommand pc) {
        this.secondary = pc;
    }

    public void onLeftClick() {
        if (primary != null) {
            primary.execute();
        }
    }
    public void onRightClick() {
        if (secondary != null) {
            secondary.execute();
```

```
        }
    }
}

public class Client {
    public static void main(String[] args) {
        DigitalBrush db = new DigitalBrush();
        MouseManager mm = new MouseManager();
        mm.setPrimary(new SmoothStrokeCommand(db));
        mm.setSecondary(new SplatteringCommand(db));
        mm.onLeftClick();
        mm.onRightClick();
        TextureBrush tb = new TextureBrush();
        mm.setPrimary(new MosaicEffectCommand(tb));
        mm.setSecondary(new StampPatternCommand(tb));
        mm.onLeftClick();
        mm.onRightClick();
    }
}
```

5. Explain how this design solves the problem.

   Command solves this problem because it enables decoupling between the sender (mouse) and receiver (specific brush in the paint application). The mouse doesn't need to know how to actually paint, it just needs to send along the information about left vs. right clicks.

6. Evaluate your design with respect to SOLID principles. Each principle should be address, if a principle is not applicable to the current pattern, say so.

   (a) Single Responsibility Principle

   This design follows the SRP because each class only has one job. The PaintCommand interface sets up the actions for commands. The concrete commands each delegate the specific action to the appropriate brush. The Brush classes both do the actual execution of the painting action. The Mouse Manager handles the mouse clicks. The client sets up the brush type and primary/secondary actions.

   (b) Open-Closed Principle

   Any new brushes or brushstrokes can be added by creating new Concrete Command or Brush type classes and no changes would have to be made to any other existing classes.

(c) Liskov Substitution Principle

Every class that implements PaintCommand can replace any PaintCommand class without affecting the correctness of the code or the MouseManager and correctly implements the execute() method.

(d) Interface Segregation Principle

No class is forced to depend on interfaces or methods that they don't use; all subclasses of PaintCommand use the single execute() method.

(e) Dependency Inversion Principle

This design follows the DIP because the Client class only relies on the high-level MouseManager, and the MouseManager only relies on the high-level interface PaintCommand instead of the lower-level implementations. We use dependency injection to setup each specific brush and brushstroke using the MouseManager at runtime.

3. Question 3:

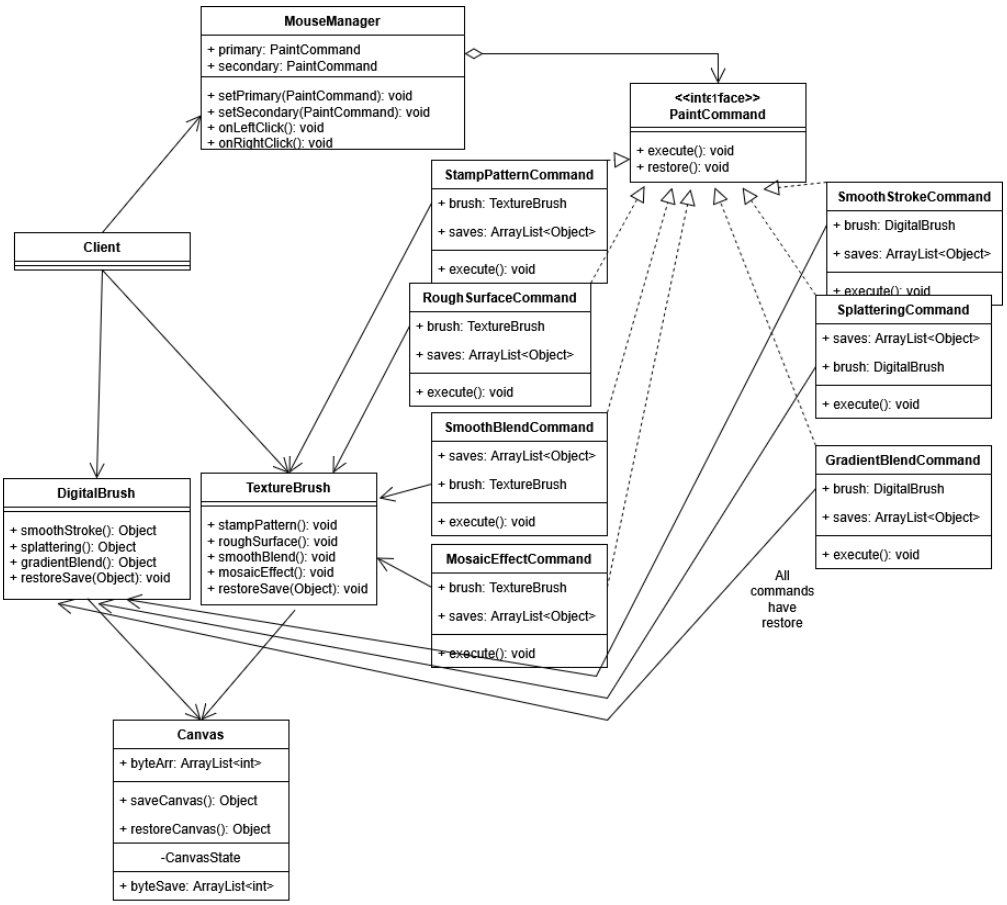
1. Give the name of the design pattern(s) you are applying to the problem.

   Memento

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..

   The memento pattern allows us to store the previous states of variables before they are changed so we can restore them at a later point which would allow us to save and restore actions.

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.



4. Give Java or C++ code for your design showing how you have implemented the pattern.

```
interface PaintCommand {
    void execute();
    void restore();
}

//RECEIVER
class DigitalBrush {
    Canvas c;
    DigitalBrush(Canvas cin){
        c = cin;
    }
    public Object smoothStroke() {
        Object save = c.saveCanvas();
        System.out.println("Applying smooth stroke with digital brush");
        return save;
    }

    public Object splattering() {
```

```
        Object save = c.saveCanvas();
        System.out.println("Applying splattering effect with digital brush");
        return save;
    }

    public Object gradientBlend() {
        Object save = c.saveCanvas();
        System.out.println("Applying gradient blend with digital brush");
        return save;
    }

    public restoreSave(Object save){
        Canvas.restoreCanvas(save);
    }
}

//RECEIVER
class TextureBrush {
    Canvas c;
    TextureBrush(Canvas cin){
        c = cin;
    }
    public void stampPattern() {
        Object save = c.saveCanvas();
        System.out.println("Applying stamp pattern with texture brush");
        return save;
    }
    public void roughSurface() {
        Object save = c.saveCanvas();
        System.out.println("Creating rough surface with texture brush");
        return save;
    }
    public void smoothBlend() {
        Object save = c.saveCanvas();
        System.out.println("Applying smooth blend with texture brush");
        return save;
    }
    public void mosaicEffect() {
        Object save = c.saveCanvas();
        System.out.println("Creating mosaic effect with texture brush");
        return save;
    }
    public restoreSave(Object save){
        Canvas.restoreCanvas(save);
```

```
    }
}

//DIGITAL BRUSH COMMANDS
class SmoothStrokeCommand implements PaintCommand {
    private DigitalBrush brush;
    private ArrayList<Object> saves;

    public SmoothStrokeCommand(DigitalBrush brush) {
        this.brush = brush;
    }
    public void execute() {
        Object save = brush.smoothStroke();
        saves.add(save);
    }
    public void restore(){
        brush.restoreSave(saves.get(saves.size() - 1));
        saves.remove(saves.size()-1);
    }
}

class SplatteringCommand implements PaintCommand {
    private DigitalBrush brush;
    private ArrayList<Object> saves;

    public SplatteringCommand(DigitalBrush brush) {
        this.brush = brush;
        saves = new ArrayList();
    }
    public void execute() {
        Object save = brush.splattering();
        saves.add(save);
    }
    public void restore(){
        brush.restoreSave(saves.get(saves.size() - 1));
        saves.remove(saves.size()-1);
    }
}

class GradientBlendCommand implements PaintCommand {
    private DigitalBrush brush;
    private ArrayList<Object> saves;

    public GradientBlendCommand(DigitalBrush brush) {
```

```
        this.brush = brush;
        saves = new ArrayList();
    }
    public void execute() {
        Object save = brush.gradientBlend();
        saves.add(save);
    }
    public void restore(){
        brush.restoreSave(saves.get(saves.size() - 1));
        saves.remove(saves.size()-1);
    }
}

//TEXTURE BRUSH COMMANDS
class StampPatternCommand implements PaintCommand {
    private TextureBrush brush;
    private ArrayList<Object> saves;

    public StampPatternCommand(TextureBrush brush) {
        this.brush = brush;
        saves = new ArrayList();
    }
    public void execute() {
        Object save = brush.stampPattern();
        saves.add(save);
    }
    public void restore(){
        brush.restoreSave(saves.get(saves.size() - 1));
        saves.remove(saves.size()-1);
    }
}

class RoughSurfaceCommand implements PaintCommand {
    private TextureBrush brush;
    private ArrayList<Object> saves;

    public RoughSurfaceCommand(TextureBrush brush) {
        this.brush = brush;
        saves = new ArrayList();
    }
    public void execute() {
        Object save = brush.roughSurface();
        saves.add(save);
    }
```

```
    public void restore(){
        brush.restoreSave(saves.get(saves.size() - 1));
        saves.remove(saves.size()-1);
    }
}

class SmoothBlendCommand implements PaintCommand {
    private TextureBrush brush;
    private ArrayList<Object> saves;

    public SmoothBlendCommand(TextureBrush brush) {
        this.brush = brush;
        saves = new ArrayList();
    }
    public void execute() {
        Object save = brush.smoothBlend();
        saves.add(save);
    }
    public void restore(){
        brush.restoreSave(saves.get(saves.size() - 1));
        saves.remove(saves.size()-1);
    }
}

class MosaicEffectCommand implements PaintCommand {
    private TextureBrush brush;
    private ArrayList<Object> saves;

    public MosaicEffectCommand(TextureBrush brush) {
        this.brush = brush;
        saves = new ArrayList();
    }
    public void execute() {
        brush.mosaicEffect();
    }
    public void restore(){
        brush.restoreSave(saves.get(saves.size() - 1));
        saves.remove(saves.size()-1);
    }
}

//INVOKER
class MouseManager {
    private PaintCommand primary;
```

```
    private PaintCommand secondary;

    public void setPrimary(PaintCommand pc) {
        this.primary = pc;
    }
    public void setSecondary(PaintCommand pc) {
        this.secondary = pc;
    }

    public void onLeftClick() {
        if (primary != null) {
            primary.execute();
        }
    }
    public void onRightClick() {
        if (secondary != null) {
            secondary.execute();
        }
    }
}

public class Canvas{
    ArrayList<int> byteArr;
    public Canvas(){
        byteArr = new ArrayList<>();
    }
    public Object saveCanvas(){
        return new CanvasState();
    }
    public void restoreCanvas(Object o){
        if(o !instanceof CanvasState){
            System.out.println("State could not be restored");
        }
        Memento m = (Memento) o;
        byteArr = m.byteSave;
    }
    private class CanvasState{
        ArrayList<int> byteSave;
        CanvasState() {
            byteSave = byteArr;
        }
    }
}
```

```
public class Client {
    public static void main(String[] args) {
        Canvas c = new Canvas();
        DigitalBrush db = new DigitalBrush(c);
        MouseManager mm = new MouseManager();
        mm.setPrimary(new SmoothStrokeCommand(db));
        mm.setSecondary(new SplatteringCommand(db));
        mm.onLeftClick();
        mm.onRightClick();
        TextureBrush tb = new TextureBrush(c);
        mm.setPrimary(new MosaicEffectCommand(tb));
        mm.setSecondary(new StampPatternCommand(tb));
        mm.onLeftClick();
        mm.onRightClick();
        mm.onLeftClick();
    }
}
```

5. Explain how this design solves the problem.
   By having a memento we are able to save the byte ArrayList and pass it to the command which called the action. From this command we are able to call restore and restore the last save made in the canvas. All classes but the canvas class will not be able to reference the class as the class is private to canvas and can only be interpreted in that class. So saves cannot be altered unintentionally.
6. Evaluate your design with respect to SOLID principles. Each principle should be address, if a principle is not applicable to the current pattern, say so.
   (a) Single Responsibility Principle

   While the original follows single responsibility it could be argued that this version does not as all commands also have to be the CareTaker for the saves to the canvas.

   (b) Open-Closed Principle
   This design follows OCP as new commands or brushes will only need to add the ability to handle mementos and will not have to redo logic in previous classes or in the canvas.

   (c) Liskov Substitution Principle

   Every class that implements PaintCommand can replace any PaintCommand class without affecting the correctness of the code or the MouseManager and correctly implements the execute() method. No changes made from question 2 will affect this answer.

(d) Interface Segregation Principle

The ISP is followed as all implementations of paintCommand will need to override the execute and the restore function. This could be violated if a command does not allow the user to restore in which case this would be violated.

(e) Dependency Inversion Principle

This design follows the DIP because the Client class only relies on the high-level MouseManager, and the MouseManager only relies on the high-level interface PaintCommand instead of the lower-level implementations. We use dependency injection to setup each specific brush and brushstroke using the MouseManager at runtime. No changes made from question 2 will affect this answer.