

**Faculty of Aviation, Science and Technology**

**Assignment Cover Sheet**

CourseCode:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Course Title:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

AssignmentTitle: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Due Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date Submitted: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Lecturer Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**To be completed if this is an individual assignment**

I declare that this assignment is my individual work. I have not worked collaboratively nor have I copied from

any other student’s work or from any other source except where due acknowledgement is made explicitly in the text, nor has any part been written for me by another person.

StudentName:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ StudentID:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Signature:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**To be completed if this is a group assignment**

We declare that this is a group assignment and that no part of this submission has been copied from any other student's work or from any other source except where due acknowledgement is made explicitly in the text, nor has any part been written for us by another person.

Student ID Student Name Signature

\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_

Lecturer's comments: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Total Marks: \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Lecturer's Signature:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Feedback to Student:**

I/We acknowledged receiving feedback from the lecturer on this assignment.

Student’s Signature: \_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_

**Extension certification:**

This assignment has been given an extension and is now due on \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

Lecturer’s Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_

Table of Contents

[1.0 Introduction 3](#_Toc192336339)

[2.0 Task 1 3](#_Toc192336340)

[2.1 Design Pattern and Justification 3](#_Toc192336341)

[2.1.1 Design Pattern: Composite Pattern 3](#_Toc192336342)

[2.1.2 Justification for Using Composite Pattern 3](#_Toc192336343)

[2.1.3 Conclusion 4](#_Toc192336344)

[2.2 Class Diagram with Attributes and Methods 4](#_Toc192336345)

[2.2.1 Class Diagram Representation 4](#_Toc192336346)

[2.2.2 Explanation of Classes: 4](#_Toc192336347)

[2.3 Method implementations (Java) 5](#_Toc192336348)

[3.0 Task 2 8](#_Toc192336349)

[3.1 A diagram of the final set of objects and their references for the house graphic. 8](#_Toc192336350)

[4.0 Task 3 Design Pattern and Justification for Bounding Box Feature 9](#_Toc192336351)

[4.1 Chosen Design Pattern: Decorator Pattern 9](#_Toc192336352)

[4.2 Justification for Using the Decorator Pattern 9](#_Toc192336353)

[4.3 Class Diagram with Attributes and Methods 10](#_Toc192336354)

[4.3.1 Class Diagram Representation 10](#_Toc192336355)

[4.3.2 Explanation of Classes 10](#_Toc192336356)

[4.4 Method implementations (Java) 11](#_Toc192336357)

[5.0 Task 4 13](#_Toc192336358)

[5.1 A diagram of the set of objects and their references after the bounding box has been added. 13](#_Toc192336359)

[6.0 Conclusion 14](#_Toc192336360)

# 1.0 Introduction

In modern software development, designing scalable and maintainable graphical systems is crucial. This project focuses on implementing a **graphics rendering system** using **Object-Oriented Programming (OOP) principles** in Java. The system follows the **Composite Pattern**, which allows individual graphical objects (such as Points, Lines, Rectangles, etc.) to be treated uniformly with grouped objects (Composite Graphics).

# 2.0 Task 1

## 2.1 Design Pattern and Justification

### 2.1.1 Design Pattern: Composite Pattern

The **Composite Design Pattern** is used in this task to create a hierarchical structure of graphical objects, where both individual shapes (e.g., Point, Line, Rectangle) and groups of shapes (CompositeGraphic) can be treated uniformly.

### 2.1.2 Justification for Using Composite Pattern

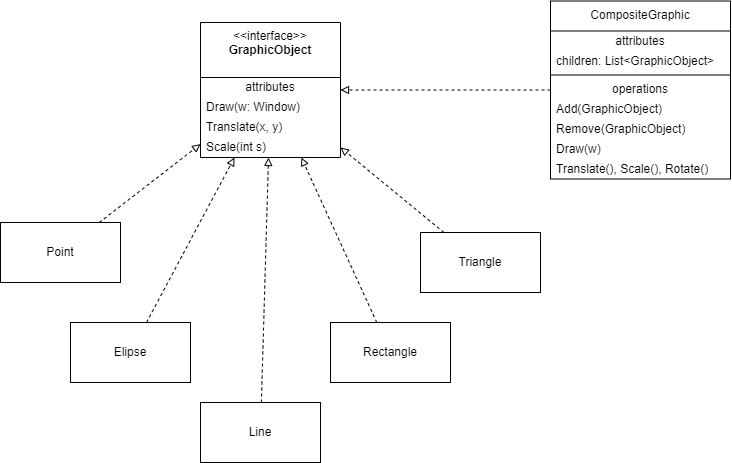
1. **Uniformity in Object Handling**
   * The pattern allows both individual shapes and groups of shapes to be treated as GraphicObject.
   * This simplifies operations like Draw(), Translate(), and Scale() because they apply equally to both single objects and composite objects.
2. **Supports Grouping and Hierarchy**
   * Using CompositeGraphic, multiple objects can be grouped together and manipulated as a single entity.
   * This is useful when creating complex graphics that contain multiple elements.
3. **Flexible and Scalable Design**
   * New graphical objects (e.g., Circle, Polygon) can be added without modifying the existing system.
   * The design is extensible, as additional transformations or graphical operations can be implemented in the future.
4. **Simplifies Client Code**
   * The client code does not need to check whether it is dealing with an individual shape or a composite shape.
   * It simply calls Draw(), Translate(), or Scale() on any GraphicObject, and the correct behavior is executed.

### **2.1.3 Conclusion**

By using the **Composite Pattern**, the system can efficiently manage both individual shapes and grouped objects, ensuring **code reusability, scalability, and maintainability**.

## 2.2 Class Diagram with Attributes and Methods

### 2.2.1 Class Diagram Representation



### 2.2.2 Explanation of Classes:

1. **GraphicObject (Interface)**

* Defines common methods: Draw(), Translate(), and Scale().
* Implemented by all graphical objects.

1. **Concrete Graphic Objects (Point, Line, Triangle, Ellipse, Rectangle)**

* Each class implements GraphicObject and provides specific implementations for Draw(), Translate(), and Scale().

1. **CompositeGraphic (Composite Class)**

* Stores multiple GraphicObject instances in a list (children).
* Implements Add(), Remove(), and calls Draw(), Translate(), and Scale() for all children.

## 2.3 Method implementations (Java)

1. import java.util.ArrayList;

2. import java.util.List;

3.

4. // Interface for all graphical objects

5. interface GraphicObject {

6.     void Draw(Window w);

7.

8.     void Translate(int x, int y);

9.

10.     void Scale(int s);

11. }

12.

13. // Concrete classes implementing GraphicObject

14. class Point implements GraphicObject {

15.     @Override

16.     public void Draw(Window w) {

17.         System.out.println("Drawing Point on window.");

18.     }

19.

20.     @Override

21.     public void Translate(int x, int y) {

22.         System.out.println("Translating Point by (" + x + ", " + y + ")");

23.     }

24.

25.     @Override

26.     public void Scale(int s) {

27.         System.out.println("Scaling Point by " + s);

28.     }

29. }

30.

31. class Line implements GraphicObject {

32.     @Override

33.     public void Draw(Window w) {

34.         System.out.println("Drawing Line on window.");

35.     }

36.

37.     @Override

38.     public void Translate(int x, int y) {

39.         System.out.println("Translating Line by (" + x + ", " + y + ")");

40.     }

41.

42.     @Override

43.     public void Scale(int s) {

44.         System.out.println("Scaling Line by " + s);

45.     }

46. }

47.

48. class Triangle implements GraphicObject {

49.     @Override

50.     public void Draw(Window w) {

51.         System.out.println("Drawing Triangle on window.");

52.     }

53.

54.     @Override

55.     public void Translate(int x, int y) {

56.         System.out.println("Translating Triangle by (" + x + ", " + y + ")");

57.     }

58.

59.     @Override

60.     public void Scale(int s) {

61.         System.out.println("Scaling Triangle by " + s);

62.     }

63. }

64.

65. class Ellipse implements GraphicObject {

66.     @Override

67.     public void Draw(Window w) {

68.         System.out.println("Drawing Ellipse on window.");

69.     }

70.

71.     @Override

72.     public void Translate(int x, int y) {

73.         System.out.println("Translating Ellipse by (" + x + ", " + y + ")");

74.     }

75.

76.     @Override

77.     public void Scale(int s) {

78.         System.out.println("Scaling Ellipse by " + s);

79.     }

80. }

81.

82. class Rectangle implements GraphicObject {

83.     @Override

84.     public void Draw(Window w) {

85.         System.out.println("Drawing Rectangle on window.");

86.     }

87.

88.     @Override

89.     public void Translate(int x, int y) {

90.         System.out.println("Translating Rectangle by (" + x + ", " + y + ")");

91.     }

92.

93.     @Override

94.     public void Scale(int s) {

95.         System.out.println("Scaling Rectangle by " + s);

96.     }

97. }

98.

99. // Composite class to group multiple GraphicObjects

100. class CompositeGraphic implements GraphicObject {

101.     private List<GraphicObject> children = new ArrayList<>();

102.

103.     public void Add(GraphicObject obj) {

104.         children.add(obj);

105.     }

106.

107.     public void Remove(GraphicObject obj) {

108.         children.remove(obj);

109.     }

110.

111.     @Override

112.     public void Draw(Window w) {

113.         System.out.println("Drawing CompositeGraphic:");

114.         for (GraphicObject obj : children) {

115.             obj.Draw(w);

116.         }

117.     }

118.

119.     @Override

120.     public void Translate(int x, int y) {

121.         for (GraphicObject obj : children) {

122.             obj.Translate(x, y);

123.         }

124.     }

125.

126.     @Override

127.     public void Scale(int s) {

128.         for (GraphicObject obj : children) {

129.             obj.Scale(s);

130.         }

131.     }

132. }

133.

134. // Mock Window class

135. class Window {

136. }

137.

138. // Example usage

139. public class Main {

140.     public static void main(String[] args) {

141.         Window window = new Window();

142.

143.         // Creating individual objects

144.         GraphicObject point = new Point();

145.         GraphicObject line = new Line();

146.         GraphicObject rectangle = new Rectangle();

147.

148.         // Creating a composite graphic

149.         CompositeGraphic group = new CompositeGraphic();

150.         group.Add(point);

151.         group.Add(line);

152.         group.Add(rectangle);

153.

154.         // Draw the composite graphic (draws all children)

155.         group.Draw(window);

156.

157.         // Apply transformations

158.         group.Translate(5, 10);

159.         group.Scale(2);

160.     }

161. }

162.

# 3.0 Task 2

## 3.1 A diagram of the final set of objects and their references for the house graphic.

A diagram of a house

AI-generated content may be incorrect.

# 4.0 Task 3 Design Pattern and Justification for Bounding Box Feature

## 4.1 Chosen Design Pattern: Decorator Pattern

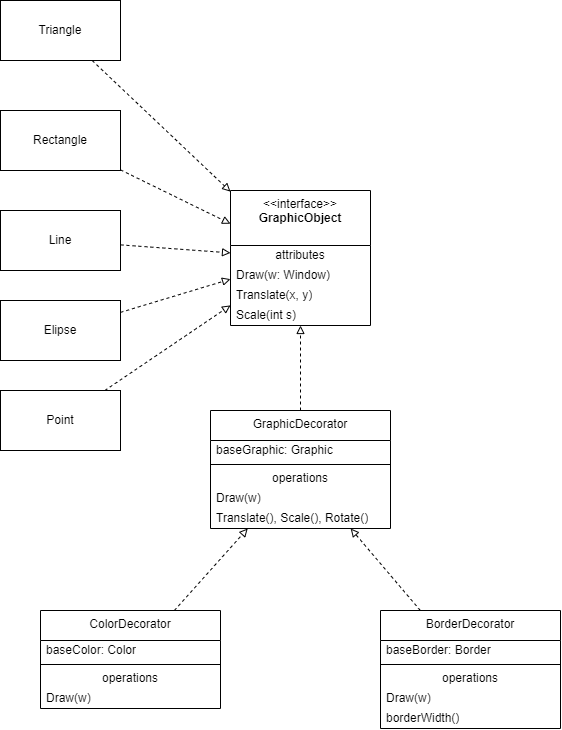
To implement the **bounding box feature** in the graphical system, we use the **Decorator Pattern**. This pattern allows us to extend the functionality of graphical objects **without modifying their existing classes**. Instead, we wrap graphical objects in a **BoundingBoxDecorator**, which calculates and draws a bounding box around the object.

## 4.2 Justification for Using the Decorator Pattern

1. **Extending Functionality Without Modifying Existing Code**
   * Instead of modifying the GraphicObject class to include bounding box logic, we create a **BoundingBoxDecorator** that wraps any GraphicObject.
   * This keeps the original classes **clean and focused** on their primary responsibilities.
2. **Dynamic and Flexible Application**
   * The bounding box feature can be added to any graphical object dynamically.
   * If we want a **bounding box around a single shape**, we wrap it with BoundingBoxDecorator.
   * If we want a **bounding box around an entire composite object**, we wrap the CompositeGraphic.
3. **Supports Future Enhancements**
   * We can **easily add new decorators** (e.g., ColorDecorator, BorderDecorator) without modifying the base object.
   * This ensures the system is **scalable and open to new features**.
4. **Separates Concerns**
   * The **decorator only handles bounding boxes**, while the original objects remain responsible for their own shape rendering.
   * This promotes **modularity and maintainability** in the system.

## 4.3 Class Diagram with Attributes and Methods

### 4.3.1 Class Diagram Representation



### 4.3.2 Explanation of Classes

1. `GraphicObject` (Interface)

- Defines common methods: `Draw()`, `Translate()`, `Scale()`, and `Rotate()`.

2. `SimpleGraphic` (Concrete Class)

- Represents \*\*basic graphical objects\*\* like `Point`, `Line`, `Rectangle`, etc.

3. `GraphicDecorator` (Abstract Class)

- Wraps a `GraphicObject` and delegates method calls to it.

- Acts as a base class for decorators like `BoundingBoxDecorator`.

4. `BoundingBoxDecorator` (Concrete Class)

- Extends `GraphicDecorator` to add a bounding box feature.

- Implements `Draw()` by first drawing the bounding box, then calling the original `Draw()`.

## 4.4 Method implementations (Java)

1. // Interface for all graphical objects

2. interface GraphicObject {

3.     void Draw(Window w);

4.     void Translate(int x, int y);

5.     void Scale(int s);

6.     void Rotate(Point p, double angle);

7. }

8.

9. // Concrete classes implementing GraphicObject

10. class Point implements GraphicObject {

11.     @Override

12.     public void Draw(Window w) {

13.         System.out.println("Drawing Point on window.");

14.     }

15.

16.     @Override

17.     public void Translate(int x, int y) {

18.         System.out.println("Translating Point by (" + x + ", " + y + ")");

19.     }

20.

21.     @Override

22.     public void Scale(int s) {

23.         System.out.println("Scaling Point by " + s);

24.     }

25.

26.     @Override

27.     public void Rotate(Point p, double angle) {

28.         System.out.println("Rotating Point around " + p + " by " + angle + " degrees.");

29.     }

30. }

31.

32. class Line implements GraphicObject {

33.     @Override

34.     public void Draw(Window w) {

35.         System.out.println("Drawing Line on window.");

36.     }

37.

38.     @Override

39.     public void Translate(int x, int y) {

40.         System.out.println("Translating Line by (" + x + ", " + y + ")");

41.     }

42.

43.     @Override

44.     public void Scale(int s) {

45.         System.out.println("Scaling Line by " + s);

46.     }

47.

48.     @Override

49.     public void Rotate(Point p, double angle) {

50.         System.out.println("Rotating Line around " + p + " by " + angle + " degrees.");

51.     }

52. }

53.

54. // Other shapes (Ellipse, Rectangle, Triangle) can be implemented similarly...

55.

56. // Decorator class implementing GraphicObject

57. abstract class GraphicDecorator implements GraphicObject {

58.     protected GraphicObject baseGraphic;

59.

60.     public GraphicDecorator(GraphicObject baseGraphic) {

61.         this.baseGraphic = baseGraphic;

62.     }

63.

64.     @Override

65.     public void Draw(Window w) {

66.         baseGraphic.Draw(w);

67.     }

68.

69.     @Override

70.     public void Translate(int x, int y) {

71.         baseGraphic.Translate(x, y);

72.     }

73.

74.     @Override

75.     public void Scale(int s) {

76.         baseGraphic.Scale(s);

77.     }

78.

79.     @Override

80.     public void Rotate(Point p, double angle) {

81.         baseGraphic.Rotate(p, angle);

82.     }

83. }

84.

85. // Concrete decorator - Adds color

86. class ColorDecorator extends GraphicDecorator {

87.     private String color;

88.

89.     public ColorDecorator(GraphicObject baseGraphic, String color) {

90.         super(baseGraphic);

91.         this.color = color;

92.     }

93.

94.     @Override

95.     public void Draw(Window w) {

96.         System.out.println("Applying color: " + color);

97.         super.Draw(w);

98.     }

99. }

100.

101. // Concrete decorator - Adds border

102. class BorderDecorator extends GraphicDecorator {

103.     private int borderWidth;

104.

105.     public BorderDecorator(GraphicObject baseGraphic, int borderWidth) {

106.         super(baseGraphic);

107.         this.borderWidth = borderWidth;

108.     }

109.

110.     @Override

111.     public void Draw(Window w) {

112.         super.Draw(w);

113.         System.out.println("Applying border width: " + borderWidth);

114.     }

115. }

116.

117. // Mock Window class

118. class Window {}

119.

120. // Example usage

121. public class Main {

122.     public static void main(String[] args) {

123.         Window window = new Window();

124.         GraphicObject rectangle = new Line();  // Assuming Line represents a simple drawable shape

125.

126.         // Apply decorators

127.         GraphicObject coloredRectangle = new ColorDecorator(rectangle, "Red");

128.         GraphicObject borderedRectangle = new BorderDecorator(coloredRectangle, 3);

129.

130.         // Draw with applied decorations

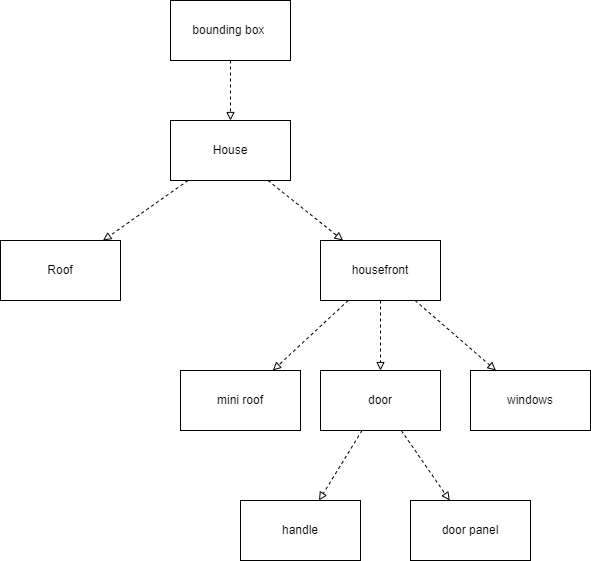
131.         borderedRectangle.Draw(window);

132.     }

133. }

# 5.0 Task 4

## 5.1 A diagram of the set of objects and their references after the bounding box has been added.



# 6.0 Conclusion

In this assignment, we designed a graphical system using **design patterns** to keep it flexible and reusable. The **Composite Pattern** helped us handle both simple and grouped objects the same way, making it easy to apply transformations like drawing, scaling, and rotating.

To add a **bounding box**, we used the **Decorator Pattern**, allowing us to extend object features without changing their core structure. This made the system more modular and easy to update.

Overall, these patterns helped create a **well-structured, scalable, and maintainable** system that meets the client’s needs. This assignment showed the importance of using the right design patterns to build efficient software.