International Institute of Technology in Sfax

PROJECT REPORT

For the validation of the course Advanced Modeling for Information Systems (MASI)

Title :
Drawingf App

PREPARED BY: :

Slim Makni

Mohamed Gharbi

INSTRUCTOR :

M. Mohamed Zayeni

Academic Year : 2024-2025

Contents

1 Introduction

Modern software development demands clean, maintainable, and extensible code. Design patterns provide established solutions to recurring software design challenges, serving as templates that can be applied across diverse contexts. In this project, we have developed a JavaFX-based drawing application that exemplifies the practical implementation of several classic design patterns.

Our Drawing Studio application goes beyond simple shape drawing to include graph visualization and shortest path calculations, making it a useful educational tool for understanding graph algorithms. Throughout its development, we applied design patterns strategically to address specific challenges and ensure the application's structure remains cohesive and extensible.

The application features:

- Interactive drawing of geometric shapes (rectangles, circles, lines, triangles)
- Graph construction with labeled nodes and connecting edges
- Shortest path calculation and visualization using multiple algorithms
- Dynamic shape decoration with borders and shadows
- Multiple logging strategies for tracking user actions
- Data persistence through file storage
- A modern, intuitive user interface

This report discusses each of these features in depth, explaining how the underlying design patterns facilitate their implementation and how different components interact to create a cohesive application.

2 System Architecture Overview

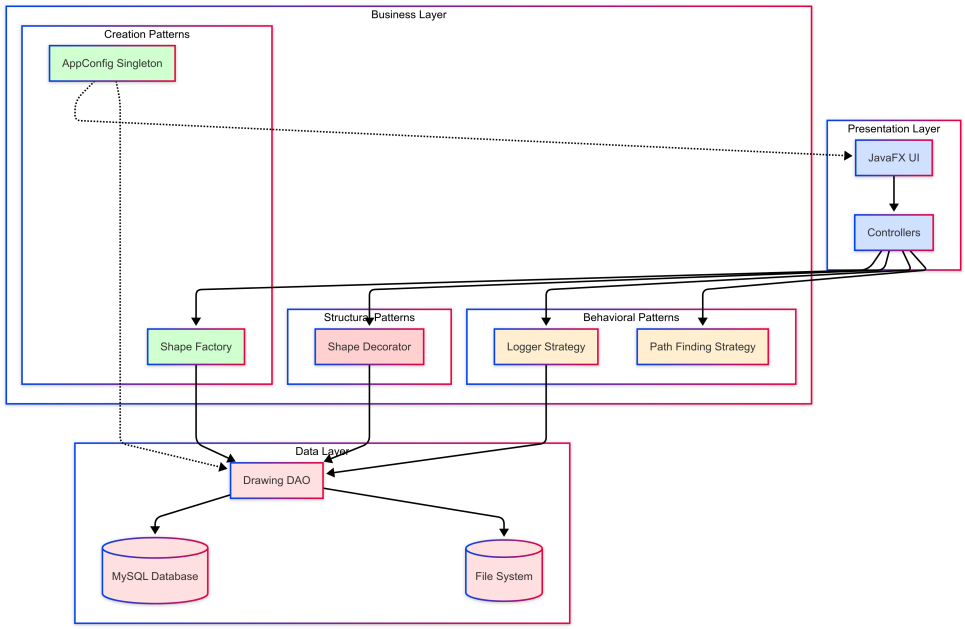

Figure 1: System Design Overview showing the architecture and pattern implementations

Our application follows a modular architecture organized around separate concerns and connected through well-defined interfaces. The system architecture is built on:

- **Presentation Layer:** JavaFX components for rendering and user interaction
- **Business Logic Layer:** Shape creation, graph manipulation, and algorithm implementations
- **Persistence Layer:** Data storage and retrieval operations

Design patterns are applied throughout these layers to create clean boundaries and facilitate extensibility. The architecture ensures that adding new shapes, algorithms, or storage mechanisms requires minimal changes to the existing codebase.

3 Features in Detail

3.1 Interactive Shape Drawing

Drawing shapes is the core functionality of our application. Users can select from different shape tools and draw them directly on the canvas by clicking and dragging.

3.1.1 Implementation Details

The shape creation process utilizes the Factory pattern, with a dedicated factory for each shape type:

Listing 1: Shape factory initialization in HelloFX.java

```
// Set up factories
factories.put(ShapeType.RECTANGLE, new RectangleFactory());
factories.put(ShapeType.CIRCLE, new CircleFactory());
factories.put(ShapeType.LINE, new LineFactory());
factories.put(ShapeType.TRIANGLE, new TriangleFactory());
```

When a user selects a shape type and starts drawing, the application:

1. Identifies the appropriate factory based on the selected shape type
2. Uses the factory to instantiate the shape at the starting coordinates
3. Sets appearance properties (fill color, border color) based on user selections
4. Updates the shape dimensions during mouse drag events
5. Finalizes the shape when the mouse is released

This process encapsulates shape creation logic within dedicated factory classes, allowing clean separation of concerns and making it straightforward to add new shape types without modifying the core drawing logic.

3.2 Graph Construction and Visualization

The graph functionality allows users to place nodes and connect them with edges, forming a directed graph structure that can be analyzed and visualized.

3.2.1 Node Placement

Users can place labeled nodes (N1, N2, etc.) by selecting the Node tool and clicking on the canvas. Each node is visually represented as a circle with a text label showing its identifier.

Listing 2: Node creation in HelloFX.java

```
Node modelNode = new Node("N" + (++nodeCounter));
graph.addNode(modelNode);

Circle circle = new Circle(x, y, 25, fillColorPicker.getValue());
circle.setStroke(strokeColorPicker.getValue());
circle.setStrokeWidth(2);

Text label = new Text(modelNode.getId());
label.setFill(Color.BLACK);
label.setStyle("-fx-font-weight: bold; -fx-font-size: 12px;");

nodeCircles.put(modelNode, circle);
nodeLabels.put(modelNode, label);
drawingPane.getChildren().addAll(circle, label);
```

3.2.2 Edge Creation

To create an edge, users select the Edge tool and click on two nodes sequentially. The application:

1. Identifies the first selected node and highlights it
2. Waits for the second node selection
3. Creates a model edge with calculated weight (based on distance)
4. Adds a visual line representation connecting the nodes
5. Updates the internal graph model for algorithm processing

Listing 3: Edge creation in HelloFX.java

```
double w = Math.hypot(
    nodeCircles.get(firstSelectedNode).getCenterX() - nodeCircles.get(targe
    nodeCircles.get(firstSelectedNode).getCenterY() - nodeCircles.get(targe
);
graph.addEdge(firstSelectedNode, target, w);
Line line = new Line(
    nodeCircles.get(firstSelectedNode).getCenterX(), nodeCircles.get(firstS
    nodeCircles.get(target).getCenterX(), nodeCircles.get(target).getCenter
);
line.setStroke(strokeColorPicker.getValue());
Edge modelEdge = new Edge(firstSelectedNode, target, w);
edgeLines.put(modelEdge, line);
drawingPane.getChildren().add(0, line);
```

This implementation maintains a clean separation between the graph's data model and its visual representation, allowing algorithms to work with the model while updates reflect in the UI.

3.3 Shortest Path Algorithms

A key feature of our application is the ability to compute and visualize the shortest path between any two nodes in the graph. Users can:

1. Select a source node from a dropdown
2. Select a goal node by clicking on it after activating the "Select Goal Node" button
3. Choose between different path-finding algorithms (Dijkstra or BFS)
4. Compute and visualize the shortest path with highlighted edges

3.3.1 Algorithm Implementation

The shortest path calculation is implemented using the Strategy pattern, with different algorithms sharing a common interface:

Listing 4: ShortestPathStrategy interface

```
public interface ShortestPathStrategy {
    List<Node> findShortestPath(Graph graph, Node source, Node target);
}
```

Two concrete implementations are provided:

- **DijkstraStrategy**: A weighted shortest path algorithm ideal for graphs where edges have different costs
- **BFSStrategy**: An unweighted shortest path algorithm suitable for finding the path with the fewest edges

The algorithm selection uses the Factory pattern through the `ShortestPathStrategyFactory`:

Listing 5: ShortestPathStrategyFactory.java

```
public class ShortestPathStrategyFactory {
    public enum AlgorithmType { DIJKSTRA, BFS }

    public static ShortestPathStrategy getStrategy(AlgorithmType type) {
        switch (type) {
            case DIJKSTRA:
                return new DijkstraStrategy();
            case BFS:
                return new BFSStrategy();
            default:
                throw new IllegalArgumentException("Unknown algorithm type"
        }
    }
}
```

3.3.2 Path Visualization

When a path is found, it is visualized by:

- Highlighting relevant edges in red
- Adding directional arrows showing the path direction
- Visually distinguishing the goal node in orange

Listing 6: Path visualization in HelloFX.java

```
private void highlightPath(java.util.List<Node> path) {
    // Reset all edges
    edgeLines.values().forEach(l->l.setStroke(Color.GRAY));

    // Remove old arrows
    drawingPane.getChildren().removeIf(n -> n.getUserData() != null &&
                                            n.getUserData().equals("arrow")
```

```
        // Highlight and add arrow for each segment
        for (int i = 0; i < path.size() - 1; i++) {
            Node u = path.get(i), v = path.get(i + 1);
            edgeLines.entrySet().stream()
                .filter(en -> en.getKey().getSource().equals(u) &&
                              en.getKey().getTarget().equals(v))
                .forEach(en -> {
                    Line l = en.getValue();
                    l.setStroke(Color.RED);

                    // Create arrow head
                    double ex = l.getEndX(), ey = l.getEndY();
                    double sx = l.getStartX(), sy = l.getStartY();
                    double angle = Math.atan2(ey - sy, ex - sx) - Math.PI / 2;
                    Polygon arrow = new Polygon(
                        ex, ey,
                        ex + 5 * Math.cos(angle - .3), ey + 5 * Math.sin(angle
                        ex + 5 * Math.cos(angle + .3), ey + 5 * Math.sin(angle
                    );
                    arrow.setFill(Color.RED);
                    arrow.setUserData("arrow");
                    drawingPane.getChildren().add(arrow);
                });
        }
}
```

The Strategy pattern here enables users to switch between algorithms without changing the main application logic, while the Factory pattern simplifies the creation of the appropriate algorithm instances.

3.4 Shape Decoration

The Decorator pattern is employed to enhance shapes with visual effects like borders and shadows, without modifying the core shape classes.

3.4.1 Implementation Details

Decoration is implemented through a hierarchy of decorator classes:

- `ShapeDecorator`: Abstract base decorator

- `BorderShapeDecorator`: Adds customizable borders to shapes
- `ShadowShapeDecorator`: Adds shadow effects to shapes

Users can select decoration options from a dropdown, and the application applies them when creating shapes:

Listing 7: Decorator application in HelloFX.java

```
// Apply decorator based on user selection
String decoratorChoice = decoratorSelector.getValue();
if ("Border".equals(decoratorChoice)) {
    BorderShapeDecorator decorator = new BorderShapeDecorator(previewShape,
                                        strokeColorPicker.getValue(), 3);
    previewShape = decorator.getDecoratedShape();
} else if ("Shadow".equals(decoratorChoice)) {
    ShadowShapeDecorator decorator = new ShadowShapeDecorator(previewShape,
                                        Color.GRAY, 10);
    previewShape = decorator.getDecoratedShape();
} else if ("Border + Shadow".equals(decoratorChoice)) {
    BorderShapeDecorator borderDecorator = new BorderShapeDecorator(preview
                                        strokeColorPicker.getValue(), 3);
    ShadowShapeDecorator shadowDecorator = new ShadowShapeDecorator(
                                        borderDecorator.getDecoratedShape
                                        Color.GRAY, 10);
    previewShape = shadowDecorator.getDecoratedShape();
}
```

The Decorator pattern allows effects to be combined (like "Border + Shadow") and applied dynamically at runtime, while maintaining a clean separation of concerns in the codebase.

3.5 Logging System

The application tracks user actions through a flexible logging system implemented using the Strategy pattern.

3.5.1 Implementation Details

Logging is managed through the `LoggerStrategy` interface with three concrete implementations:

- `ConsoleLogger`: Outputs messages to the console

- `FileLogger`: Writes messages to a log file
- `DatabaseLogger`: Stores messages in a MySQL database

Users can switch between logging strategies at runtime through the UI:

Listing 8: Logging strategy selection in HelloFX.java

```
logSelector.setOnAction(e -> {
    switch (logSelector.getValue()) {
        case "Console":
            logger = new ConsoleLogger();
            break;
        case "File":
            logger = new FileLogger();
            break;
        case "Database":
            logger = new DatabaseLogger();
            break;
    }
    logger.log("Log strategy changed: " + logSelector.getValue());
    statusLabel.setText("Log strategy changed: " + logSelector.getValue());
});
```

This implementation demonstrates how the Strategy pattern can effectively decouple the way a feature operates from the context where it's used.

3.6 Data Persistence

The application supports saving and loading drawings using the DAO (Data Access Object) pattern.

3.6.1 Implementation Details

Persistence is handled through the `DrawingDAO` interface with a `FileDrawingDAO` implementation:

Listing 9: DAO interface

```
public interface DrawingDAO {
    void save(List<Shape> shapes) throws Exception;
    List<Shape> load() throws Exception;
    List<Shape> load(String path) throws Exception;
}
```

This abstraction allows for different storage mechanisms to be implemented without changing the application's business logic.

Listing 10: DAO usage in HelloFX.java

```
saveBtn.setOnAction(e -> {
    try {
        drawingDAO.save(shapes);
        logger.log("Drawing saved.");
        statusLabel.setText("Drawing saved");
    } catch (Exception ex) {
        logger.log("Error during save: " + ex.getMessage());
        statusLabel.setText("Error during save");
    }
});
```

The DAO pattern abstracts away the data storage details, making it possible to switch between storage mechanisms (file-based, database, cloud storage) by implementing new DAO classes without modifying the application logic.

4 Design Patterns Analysis

4.1 Factory Pattern

4.1.1 Overview

The Factory pattern creates objects without specifying the exact class to be created. It provides an interface for creating objects, but lets subclasses decide which classes to instantiate.

4.1.2 Implementation in Our Project

We've implemented the Factory pattern in two key areas:

1. **Shape Creation:** Through `ShapeFactory` interface and concrete factories (`RectangleFactory`, `CircleFactory`, etc.)
2. **Algorithm Creation:** Through `ShortestPathStrategyFactory` for instantiating path-finding algorithms

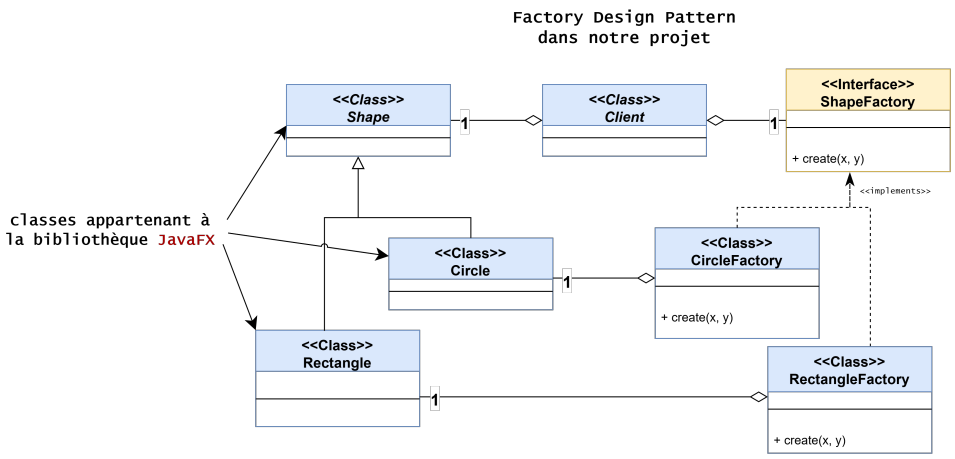

Figure 2: Factory UML diagram

4.1.3 Benefits Realized

- **Decoupling:** The application doesn't need to know how shapes are constructed
- **Extensibility:** New shapes can be added by creating a new factory without modifying existing code
- **Centralization:** Creation logic is centralized in specialized classes

4.2 Strategy Pattern

4.2.1 Overview

The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm vary independently from clients that use it.

4.2.2 Implementation in Our Project

We've applied the Strategy pattern in two main areas:

1. **Logging:** `LoggerStrategy` interface with `ConsoleLogger`, `FileLogger`, and `DatabaseLogger` implementations
2. **Shortest Path Algorithms:** `ShortestPathStrategy` interface with `DijkstraStrategy` and `BFSStrategy` implementations

4.2.3 Benefits Realized

- **Runtime Flexibility:** Algorithms can be switched at runtime through the UI

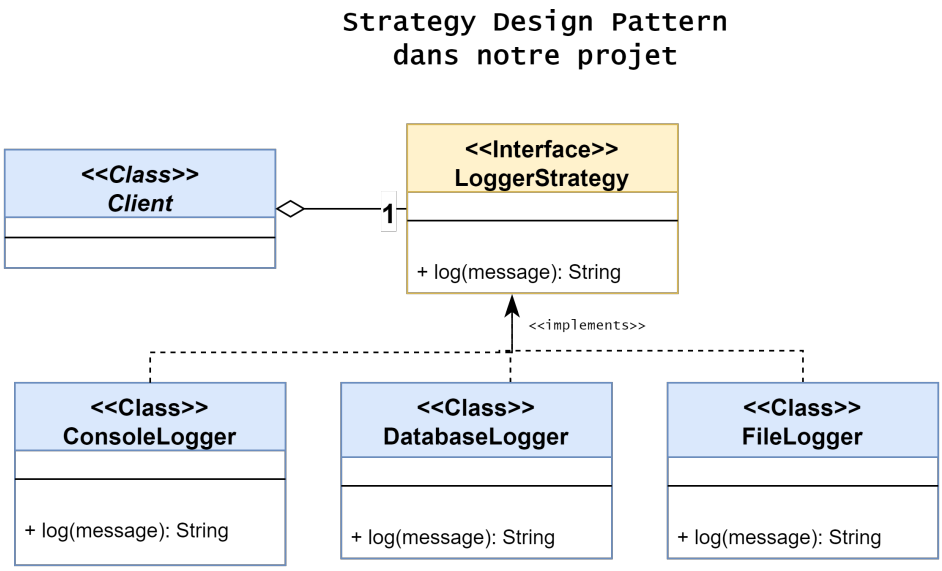

Figure 3: Strategy UML diagram about logger

- **Clean Abstraction:** The context (main application) doesn't need to know the specifics of each algorithm
- **Easy Extension:** New algorithms can be added by creating new strategy classes

4.3 Singleton Pattern

4.3.1 Overview

The Singleton pattern ensures a class has only one instance and provides a global point of access to it.

4.3.2 Implementation in Our Project

We use the Singleton pattern for global application configuration through the `AppConfig` class, ensuring consistent settings across the application.

4.3.3 Benefits Realized

- **Controlled Access:** Provides a single access point to configuration
- **Reduced Memory Footprint:** Avoids multiple instances of configuration data
- **Consistent State:** Ensures all components work with the same configuration

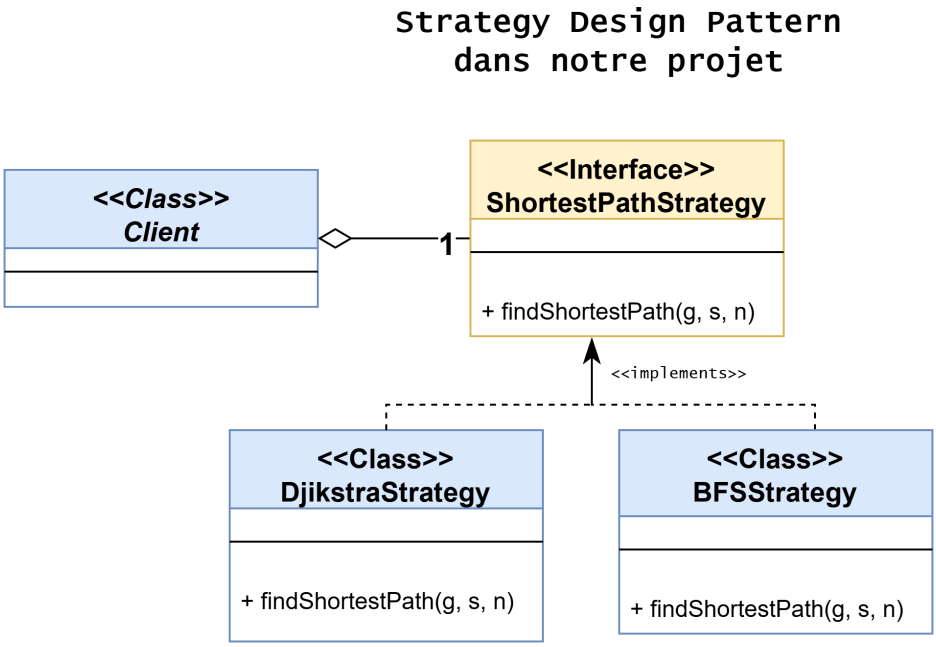

Figure 4: Strategy UML diagram about Shortest path algorithm

4.4 Decorator Pattern

4.4.1 Overview

The Decorator pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

4.4.2 Implementation in Our Project

We implement the Decorator pattern through the `ShapeDecorator` abstract class with `BorderShapeDecorator` and `ShadowShapeDecorator` concrete decorators.

4.4.3 Benefits Realized

- **Dynamic Composition:** Visual effects can be added at runtime
- **Combinable Effects:** Multiple decorators can be stacked (e.g., border + shadow)
- **Open/Closed Principle:** New decorators can be added without modifying existing shapes

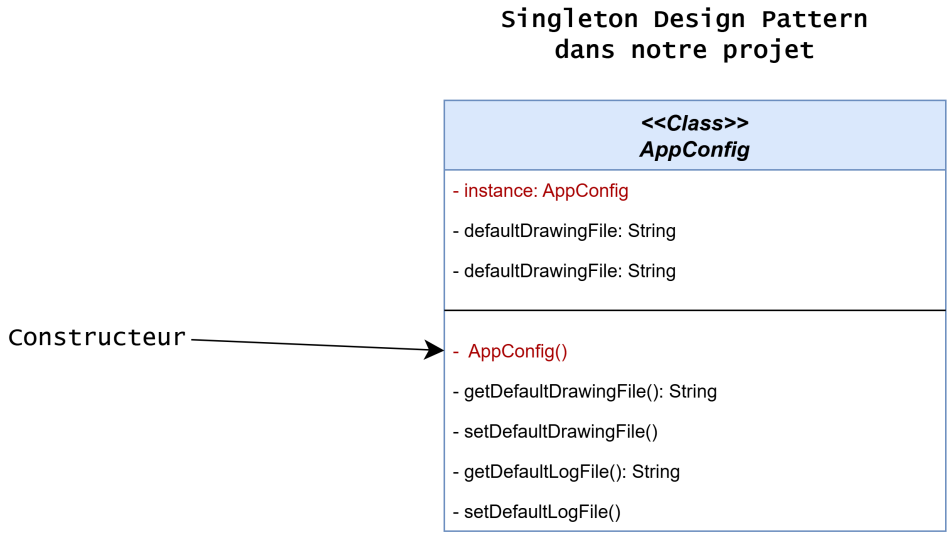

Figure 5: Singleton UML diagram

4.5 DAO Pattern

4.5.1 Overview

The Data Access Object (DAO) pattern provides an abstract interface to a database or other persistence mechanism.

4.5.2 Implementation in Our Project

We implement the DAO pattern through the `DrawingDAO` interface and the `FileDrawingDAO` concrete implementation.

4.5.3 Benefits Realized

- **Abstraction:** The application doesn't need to know how data is stored
- **Separation of Concerns:** Persistence logic is separated from business logic
- **Flexibility:** Storage mechanisms can be changed by implementing new DAO classes

5 Pattern Integration and Synergies

The true power of design patterns emerges when they work together. In our application, we've identified several key synergies:

- **Factory + Strategy:** The Factory pattern (`ShortestPathStrategyFactory`) creates instances of Strategy pattern implementations (`DijkstraStrategy`, `BFSStrategy`), simplifying algorithm creation and selection.

- **Strategy + Singleton:** Strategy implementations can access configuration through the Singleton `AppConfig`, ensuring consistent behavior across algorithms.
- **Decorator + Factory:** Shape factories create base shapes that are then enhanced by decorators, providing a clean separation between shape creation and decoration.
- **DAO + Strategy:** The logging strategy implementations can utilize the DAO pattern for database logging, demonstrating how patterns can be nested.

These integrations create a cohesive system where components work together while maintaining clean boundaries and responsibilities.

6 Conclusion

The Drawing Studio application demonstrates how design patterns can be applied effectively in a real-world JavaFX application. By leveraging Factory, Strategy, Singleton, Decorator, and DAO patterns, we've created a modular, extensible, and maintainable system.

The integration of these patterns has enabled us to build a feature-rich application that allows users to draw shapes, construct graphs, visualize shortest paths, and persist their work. Each pattern addresses specific design challenges, resulting in clean separation of concerns and enhanced flexibility.

Key takeaways from this project include:

- Design patterns provide tested solutions to common software design problems
- Patterns can work together synergistically to enhance maintainability and extensibility
- Strategic application of patterns leads to more modular, flexible code
- Understanding when and how to apply patterns is as important as knowing their structure

Through careful pattern selection and implementation, we've created an application that not only serves its functional purpose but also exemplifies good software design principles.

7 References

1. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
2. Oracle. (2023). JavaFX Documentation. `https://openjfx.io/javadoc/19/`
3. Project source code and documentation files.

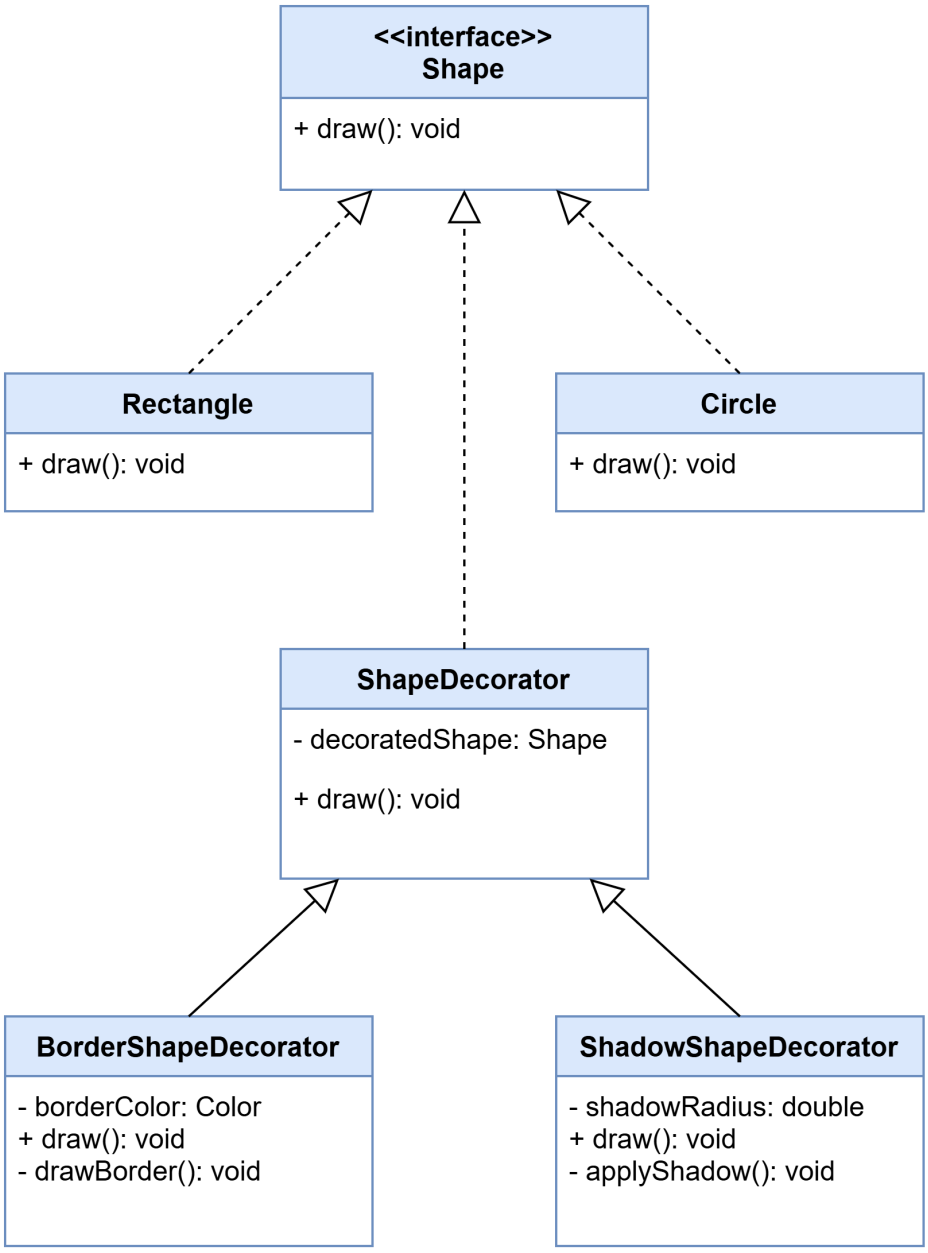

Figure 6: Decorator UML diagram

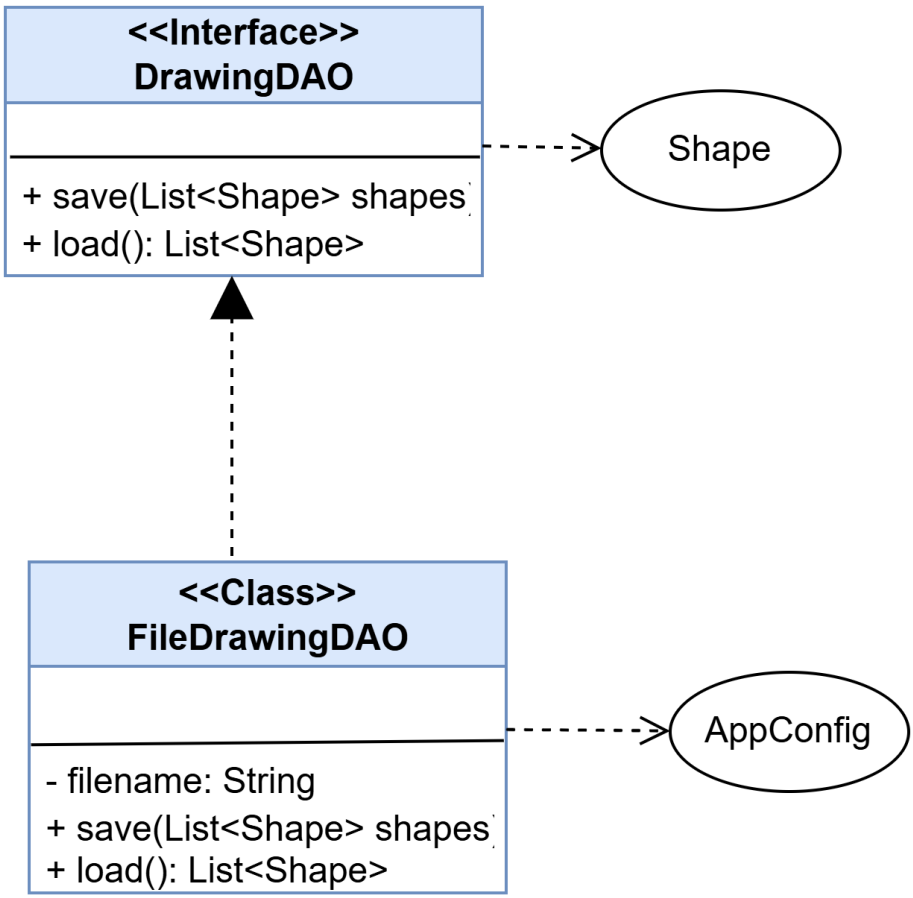

Figure 7: DAO UML diagram