Name: Kelven Lai Student ID: 1255199

1. Problem Context

The assignment is to design and implement a distributed shared whiteboard application. Shared whiteboards are dynamic, collaborative platforms enabling multiple users to simultaneously interact with a common canvas to draw, insert text, and add shapes. These features enhance the utility and flexibility of the whiteboard, making it a valuable resource for remote collaboration. The purpose of this assignment is to showcase an understanding and implementation of fundamental principles such as networked communication and concurrency, alongside proficiency in crafting a networked application. Additionally, this assignment encourages students to explore their own design approaches, including the selection of communication protocols, architectural design of the system, choice between using sockets or RMI for communication, GUI development, and strategies for error handling.

2. System Components

i. **Client Applications**:
   a. **CreateWhiteBoard**: Manages the session for the whiteboard creator or manager. It initiates the server, sends commands to other clients, and handles administrative tasks such as inviting users, approving join requests, and managing the session (including closing the session or kicking users).
   b. **JoinWhiteBoard**: Used by clients wanting to join an existing whiteboard session. It handles receiving drawing commands, participating in the shared canvas, and managing personal interactions like sending messages or leaving the session.

ii. **Server Application**:
   a. **UserHandler**: Acts as a centralized server handler for individual client connections. It manages user sessions, relays drawing commands and messages between clients, and maintains the state of the whiteboard session, including the list of connected users.

iii. **Utility Classes**:
   a. **DrawingUtils**: Provides static methods to handle the drawing operations on the canvas, interpreting **DrawingCommand** objects to render graphical representations.
   b. **ManagerUtils**: Contains methods that facilitate managerial actions, such as handling join requests, closing sessions, and kicking users, often involving decision dialogs.
   c. **AlertUtils**: Offers static methods to show different types of alerts and informational dialogs to the user.
   d. **DrawingTools**: Manages custom cursors used for different drawing tools within the application, enhancing the user interface.

iv. **Protocol Classes**:
   a. **DrawingCommand**: Encapsulates all the information necessary for drawing operations, such as drawing type, coordinates, color, and stroke width.
   b. **Message**: Used for creating and parsing messages that are sent and received over the network, supporting both command and control operations within the application.

v. **Constants Class**:
   a. **MessageConstants**: Defines constants used across the application for identifying types of messages and commands, ensuring consistency in the communication protocol.

3.1. Class Design

The following relationships and interactions are key to the system's design:

1. **Centralized Server Architecture:**

- The system is built on a centralized server model which coordinates all client interactions, data processing, and state synchronization.

2. **Client-Server Interaction:**
   - **CreateWhiteBoard** and **JoinWhiteBoard** manage communications with the server using Message class instances for real-time updates.
   - The **UserHandler** on the server processes these messages, updates session states, and broadcasts updates to ensure all clients have the latest canvas view.
3. **Utility Usage:**
   - **DrawingUtils** are employed by client classes to interpret and render **DrawingCommand** objects on the local canvas.
   - **ManagerUtils** and **AlertUtils** support administrative functions and user interactions within the **CreateWhiteBoard**.
   - **DrawingTools** adjust cursor styles according to the drawing tool selected, enhancing user interface responsiveness.
4. **Data Handling:**
   - The **DrawingCommand** and **Message** classes are crucial for encapsulating drawing actions and communication, respectively, ensuring consistent and efficient operations across the network.

3.2. Interaction Diagrams

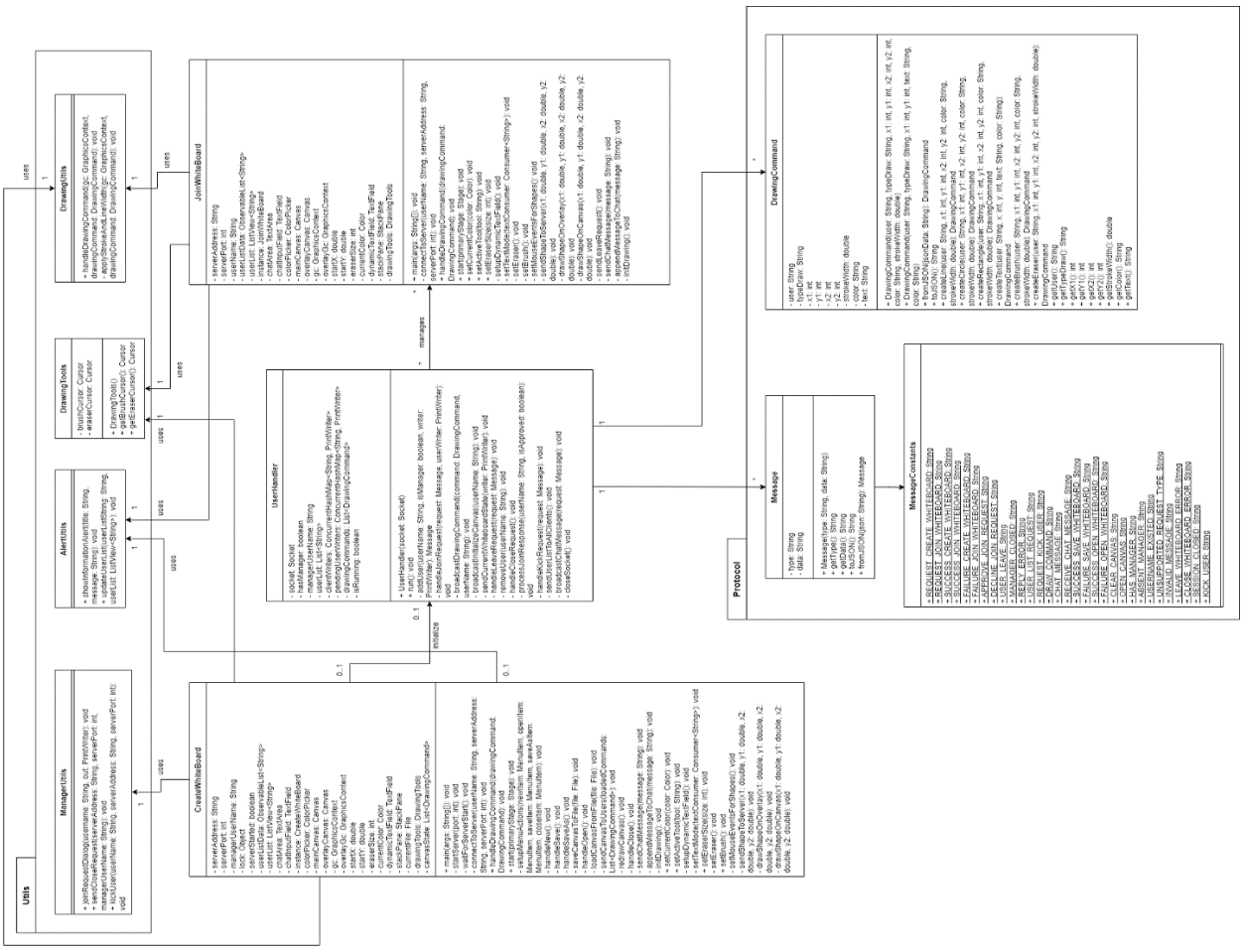


Figure 1: Design Class Diagram

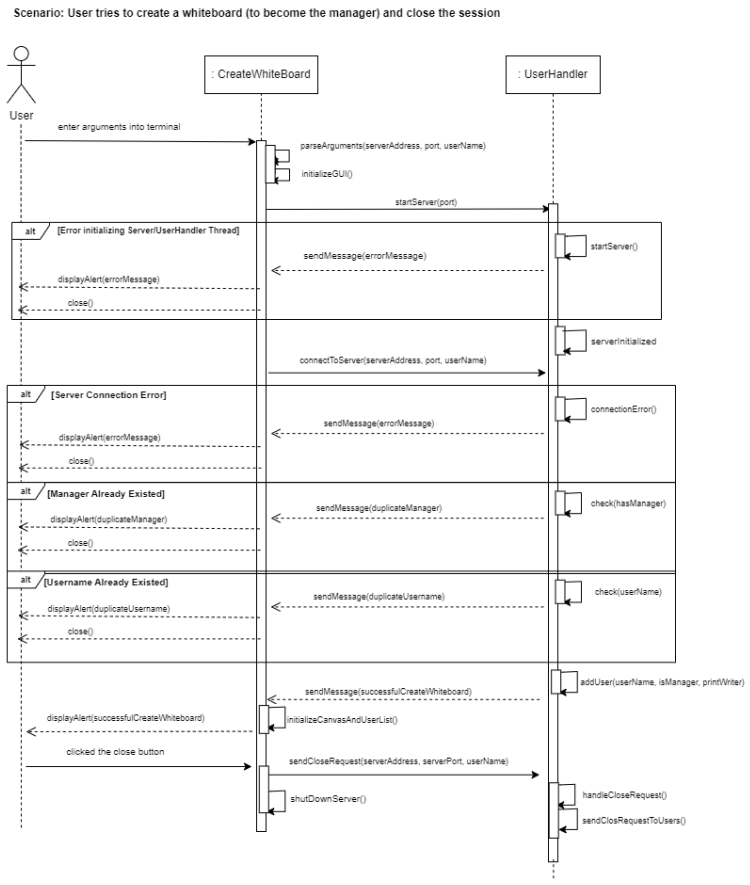


Figure 2: User tries to create a whiteboard

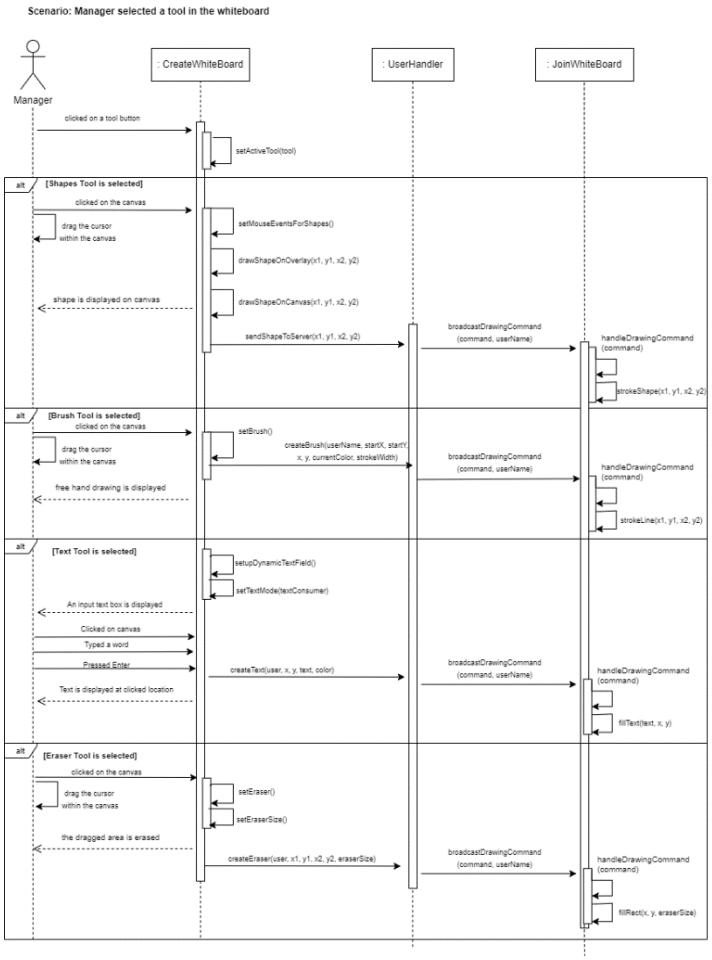


Figure 3: Manager tries to select a tool

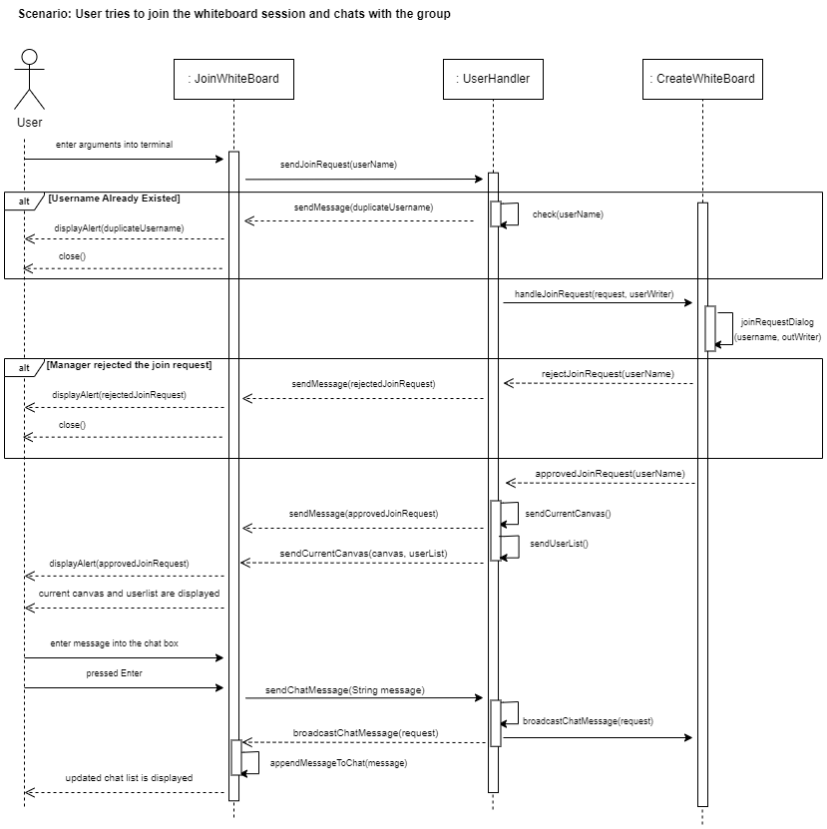


Figure 4: User tries to join whiteboard

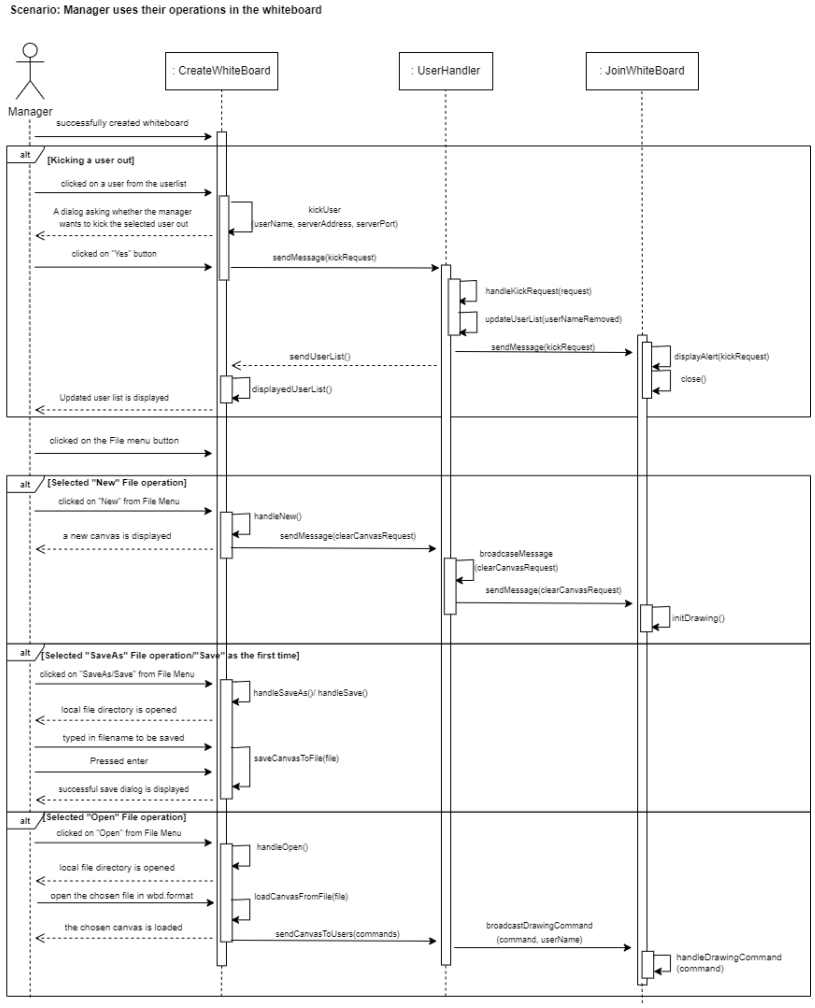


Figure 5: Manager uses their operations

4.1. Critical Analysis

4.1.1. Design Choices

1. **Server Architecture: Single Centralized Server with Client**
    - **Choice Rationale:** A centralized server architecture was chosen over multiple servers system to simplify state management and reduce the complexity of the synchronization mechanism. Utilizing a single server ensures all clients have a consistent view of the shared canvas, crucial for real-time collaborative drawing.
    - **Implementation:** Employing a **ConcurrentHashMap** for managing shared data ensures thread safety and efficient concurrency control, allowing simultaneous reads and updates by multiple client threads without performance degradation.
        - **Thread Safety: ConcurrentHashMap** is designed to handle concurrent operations efficiently, providing full concurrency of retrievals and adjustable expected concurrency for updates.

2. **Communication Protocol: TCP Sockets**
    - **Choice Rationale:** TCP was selected due to its reliability in ensuring data is delivered in order and without loss, crucial for maintaining the integrity of drawing commands and updates in a real-time collaborative environment.
    - **Implementation:** Each client connects to the server via a TCP socket, through which all drawing commands and canvas updates are communicated, ensuring all participants' views are synchronized without errors.

3. **Message Protocol: JSON for Message and Drawing Command Classes**
    - **Choice Rationale:** JSON was chosen for its ease of use, human-readability, and widespread adoption in data interchange. It supports complex data structures necessary for conveying drawing commands and messages efficiently.
    - **Implementation:** JSON is used to serialize **Message** and **DrawingCommand** classes for network transmission, simplifying the parsing and generation of complex data structures across different client platforms.
        - **Flexibility:** JSON's format allows easy modification and extension of the data model, accommodating future enhancements such as new drawing tools or collaborative features.

4. **Data Storage: File-Based Storage**
    - **Choice Rationale:** File-based storage provides a simple, reliable way to save and retrieve whiteboard sessions without the complexity and overhead of a database system. This choice benefits from ease of backup and portability.
    - **Implementation:** Drawing sessions are serialized to JSON and saved to disk, allowing users to save their work and load previous sessions. This approach also facilitates easy sharing and archiving of whiteboard content.

5. **GUI Design: JavaFX**
    - **Choice Rationale:** JavaFX was selected for its rich set of features and capabilities in building sophisticated graphical user interfaces. It provides extensive support for customizing UI components and animations, enhancing the user experience in a collaborative drawing application.
    - **Implementation:** JavaFX is used to create a responsive and intuitive user interface for the whiteboard application. Features such as dynamic tool selection, color pickers, and shape options are implemented to give users control over their drawing environment.

- **Customization:** JavaFX supports CSS styling, allowing for detailed customization of the UI, including themes and styles that can improve accessibility and user engagement.

6. **Error Handling: Try-Catch Blocks**
   - **Choice Rationale:**
   - **Granular Control:** Try-catch blocks allow for detailed isolation and management of specific error conditions, which facilitates easier debugging and issue resolution during development and testing.
   - **Code Readability:** Implementing try-catch blocks enhances code readability by clearly separating the error-handling sections from the regular flow of the program, which improves both maintainability and understandability.

7. **Thread Management:**
   - **Server Thread**
     - **Purpose:** Handles listening for incoming connection requests from clients.
     - **Context:** Utilized in the CreateWhiteBoard application when the manager initiates the server. It constantly listens on a socket for new clients and manages their connection requests.
   - **Client Communication Threads**
     - **Purpose**: Manages network communications between the server and each connected client.
     - **Context**: In the UserHandler class, a new thread is created for each connected client to manage incoming messages, including drawing commands, chat messages, and control commands such as join requests or session terminations.
   - **UI Update Threads**
     - **Purpose**: Updates the graphical user interface in response to received data without disrupting user interactions.
     - **Context**: Used in both CreateWhiteBoard and JoinWhiteBoard applications. These threads refresh the UI based on data received from the server, employing Platform.runLater() to ensure thread safety while updating UI components.
   - **Drawing Command Processing Thread**
     - **Purpose**: Processes and disseminates drawing commands from one client to all others to maintain synchronized state across the whiteboard.
     - **Context**: Typically managed within the UserHandler or a dedicated service, this thread processes received drawing commands and updates all clients to ensure the whiteboard reflects all user inputs.
   - **Asynchronous Task Threads**
     - **Purpose**: Executes long-running operations that are non-disruptive to the main application workflow.
     - **Context**: These tasks might include saving the current whiteboard state, loading sessions, or handling other resource-intensive operations in the background.
   - **Connection Monitoring Threads**
     - **Purpose**: Monitors the status of client-server connections to manage disconnections or network issues effectively.
     - **Context**: These threads periodically assess the health of client connections, handling reconnections or alerting users to connectivity problems as necessary.

4.1.2. Challenges Encountered

- **Proper Thread Handling:** Managing concurrency effectively in a multi-threaded environment to ensure smooth operation and avoid common pitfalls such as thread interference and memory consistency errors. This involves careful structuring of thread synchronization and

coordination, particularly when threads are drawing on the whiteboard simultaneously or updating shared resources like the user list or chat messages.

- **Communication via Sockets:** Developing a reliable communication system using TCP sockets that supports the continuous and simultaneous exchange of drawing commands and updates between clients. This includes challenges in handling socket connections, managing data flow without loss or corruption, and ensuring that all clients are synchronized in real-time, especially when implementing dynamic tools like the brush and eraser, which require frequent and rapid updates.
- **Robust Error Handling:** Implementing a sophisticated error handling strategy to catch and respond to various exceptions that may occur during network communication, file operations, or user interactions, which is critical for maintaining the reliability and usability of the application.

4.2. Conclusion

In conclusion, the collaborative whiteboard application was developed through strategic design choices aimed at optimizing performance and user experience. These included a centralized server architecture, reliable TCP socket communication, JSON for data handling, and a JavaFX-designed GUI. Challenges such as effective multi-threading, robust client communication, error handling, and GUI responsiveness were addressed with detailed system design and careful implementation. Overall, the project not only showcased technical proficiency in real-time collaboration tools but also emphasized the critical nature of thoughtful system architecture and user-centric design in developing stable and efficient applications.

5.1. Excellence Elements

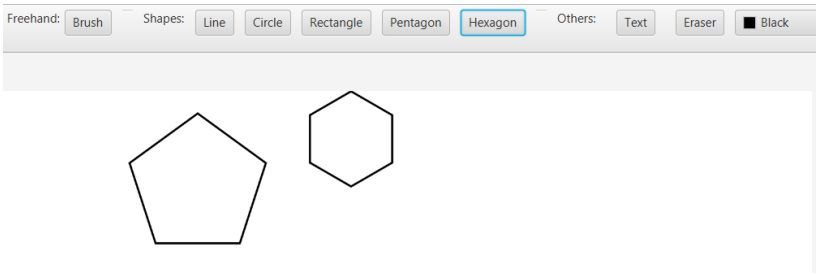
1. **Robust Error Handling**
    - **Description**: The application includes comprehensive error handling mechanisms that ensure any issues during the execution, such as network failures, file access errors, or data format issues, are caught and managed gracefully.
    - **Discussion**: For instance, if there is a failure in network communication, the system captures the exception and notifies the user via a clear, understandable error message. This not only prevents the application from crashing but also provides feedback that helps users understand what went wrong, improving reliability and user experience.

2. **Proper Thread Handling**
    - **Description**: The application employs sophisticated thread management strategies to handle multiple users interacting with the whiteboard simultaneously. This includes the use of separate threads for handling client requests and UI updates to ensure smooth performance and responsiveness.
    - **Discussion**: By using dedicated threads for network operations and UI management (via **Platform.runLater()** in JavaFX), the system maintains responsiveness even under load. Proper synchronization mechanisms are used to avoid race conditions and deadlocks when multiple threads update shared resources like the whiteboard canvas or user lists.

3. **Clear Feedback Mechanisms**
    - **Description**: The system provides immediate and clear feedback to users through dialog boxes and status messages for various actions, such as joining or leaving a session, errors, or successful operations.
    - **Discussion**: Whenever an action is taken (e.g., saving a session, attempting to connect to a server), the outcome is immediately communicated through a popup dialog or a status bar update. This feedback is crucial for a good user experience, ensuring that users are always informed about the state of their actions.

5.2. Creativity Elements

1. **Added New Shapes like Pentagon and Hexagon**
    - **Description**: Beyond standard shapes like circles and rectangles, the application includes options to draw pentagons and hexagons, which enhances the creative tools available to users.
    - **Illustration/Discussion**: These shapes can be selected from the toolbar and drawn on the canvas with adjustable size and color. Implementing these shapes involved calculating the vertices based on user input points, demonstrating an advanced level of geometric computation in the application.



2. **Draggable Shapes with Mouse Cursor**
    - **Description**: Shapes on the canvas are not only static but can be repositioned while dragging. This is achieved by implementing draggable functionality, where users can select and move shapes around the canvas.
    - **Illustration/Discussion**: This feature enhances the interactive experience, allowing for dynamic adjustments to drawings. It was implemented using mouse event listeners that update the position of the shape in response to user drag actions.

3. **Cursor Icons for Tools like Brush and Eraser**
    - **Description**: To enhance the user interface and make the application more intuitive, custom cursor icons are used for different tools. For instance, selecting the brush tool changes the cursor to a brush icon, and the eraser tool changes it to an eraser icon.
    - **Discussion**: This visual cue helps users quickly identify the active tool, improving usability. The custom cursors are implemented using the **Cursor** class in JavaFX, which allows for the setting of image-based cursors.