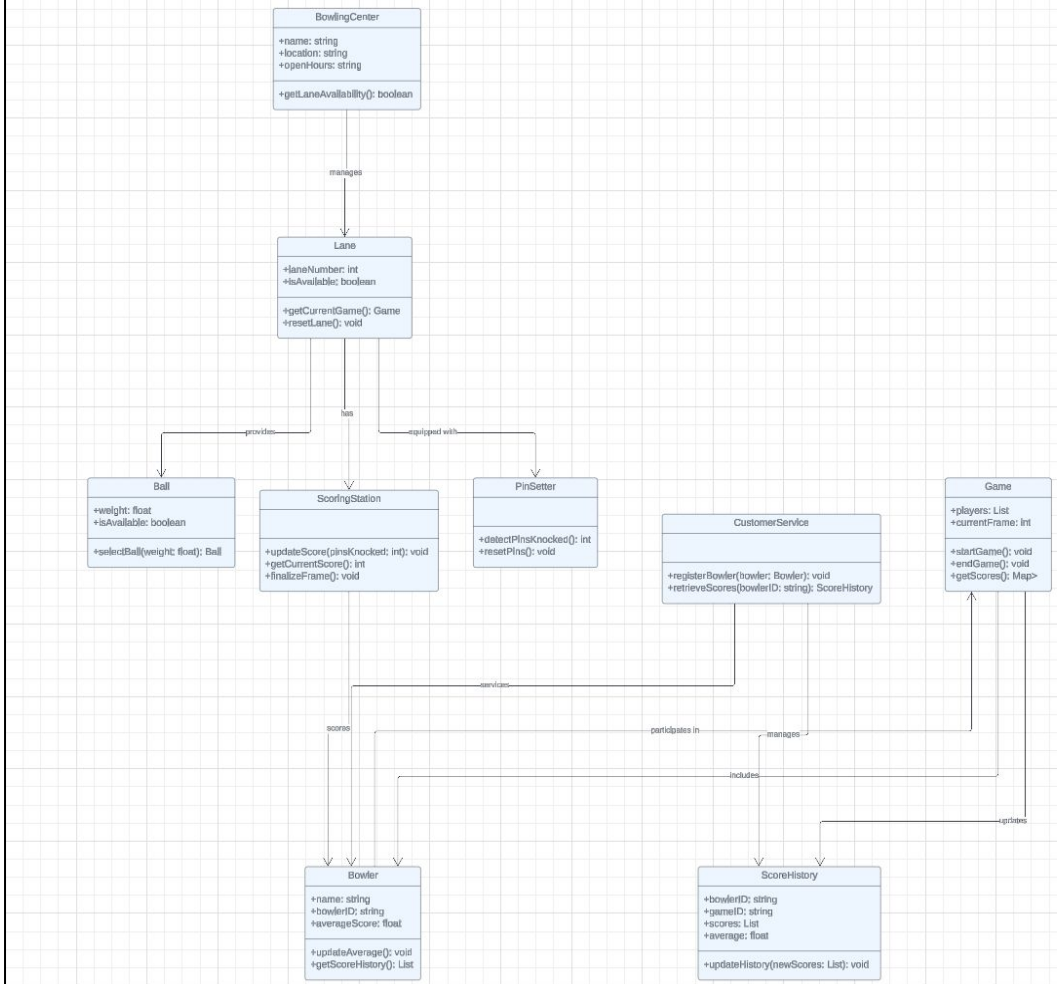
Refactoring Design Presentation

By: Yousaf, Devin, Sabrina, Hunter, and Cameron.

Product Overview

- **The Lucky Strikes Bowling Center (LSBC) aims to automate its nationwide bowling facilities by installing pin-setting equipment that detects pins knocked down and communicates this data to an automated scoring system.**
- **Additionally, LSBC plans to offer a service to track customers' score histories, averages, and related statistics for enhanced user experience.**

Domain Model



Analysis of Original Design

- **The bowling alley simulation features a modular design with distinct components like PinGUI and LaneGUI for visual updates, Lane for gameplay management, and ControlDesk for backend operations, supported by classes like Bowler and Party for data handling.**
- **While the system is cohesive and functional, the centralization of Lane and ControlDesk poses challenges for scalability and maintenance, necessitating efforts to reduce coupling and ensure efficient data synchronization.**

Design Weaknesses and Strengths

- **The system design demonstrates strengths such as high cohesion, modularity, an event-driven approach, adherence to the MVC pattern, and multithreading, though the latter revealed implementation challenges with correctness.**
- **Weaknesses include violations of the Law of Demeter and security concerns stemming from the Observer pattern, which allows unintended information access by certain subscribers.**

Use of Design Patterns

Name: ControlDesk		GoF Pattern: Observer
Participants:		
Class:	**Role:**	**Contribution:**
ControlDeskObserver	Observer	Defines an interface to be implemented by classes that want to be notified of updates
ControlDeskView	Concrete Observer	Implements the ControlDeskObserver interface to display the current status of the ControlDesk
AddPartyView	Concrete Observer	Receives updates about ControlDesk changes to manage party setup views
ControlDesk	Subject	Maintains a list of observers and notifies them of changes in the ControlDesk state
Deviations: None		

Use of Design Patterns

Name: Lane	GoF Pattern: Observer

Participants:

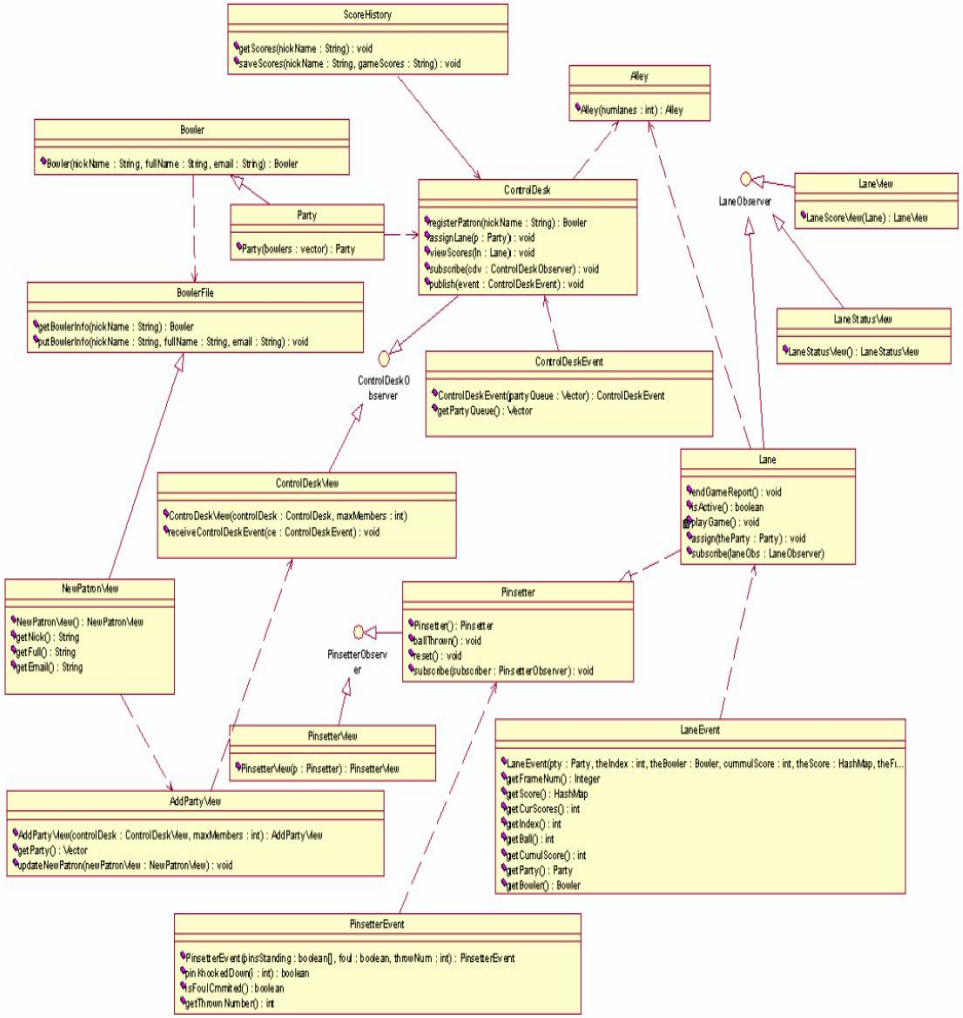
Class:	Role:	Contribution:
LaneObserver	Observer	Defines an interface for classes interested in updates from the Lane
LaneView	Concrete Observer	Implements the LaneObserver interface to display lane-specific information
LaneStatusView	Concrete Observer	Displays updates on the current status of the lane to users
ControlDesk	Subject	Notifies its observers about events like score updates, pin counts, or lane states

Deviations: None

Use of Design Patterns

Name: Pinsetter		GoF Pattern: Observer
Participants:		
Class:	**Role:**	**Contribution:**
PinsetterObserver	Observer	Defines an interface for receiving updates about pinsetter events
PinsetterView	Concrete Observer	Implements the PinsetterObserver interface to visually display pinsetter updates
LaneStatusView	Concrete Observer	Receives pinsetter updates to reflect the lane status in real time
Pinsetter	Subject	Notifies observers when pins are reset or events occur in the pinsetter
Deviations: None		

Subsystem and Class Structure



Metric Analysis

Metrics

1521 lines of code analyzed, in 31 classes, in 1 packages.

Metric	Total	Density*
High Priority Warnings	16	10.52
Medium Priority Warnings	54	35.50
Low Priority Warnings	27	17.75
Total Warnings	**97**	**63.77**

(Defects per Thousand lines of non-commenting source statements)*

Summary

Warning Type	Number
Bad practice Warnings	18
Correctness Warnings	3
Internationalization Warnings	3
Malicious code vulnerability Warnings	29
Multithreaded correctness Warnings	1
Performance Warnings	18
Dodgy code Warnings	25
Total	**97**

Metric Analysis

- **The initial design had a high density of medium-priority warnings, which we addressed strategically to improve the system within time constraints.**
- **Prioritizing correctness, internationalization, and multithreaded correctness warnings allowed us to target critical or manageable issues that offered significant improvements without major disruptions.**

The Refactored Design

- **The refactored design enhances maintainability and robustness by prioritizing low coupling, high cohesion, proper encapsulation, and adherence to principles like separation of concerns and the Law of Demeter.**
- **Key improvements include resolving correctness issues, ensuring multithreading safety, and standardizing internationalization practices, resulting in a modular, reusable, and extensible system.**

The Refactored Design

Refactoring Identification	**Correctness Warnings**
Metric Evidence	**3**
Standard Refactoring Pattern	**To address correctness warnings, the standard refactoring patterns focus on ensuring that the code behaves as intended by eliminating potential errors, such as null pointer dereferences and incorrect method calls. Common strategies include adding proper null checks, refining exception handling, and using more appropriate data types or logic to prevent edge cases from causing issues.**
Description Of The Refactoring	**The Bowler class's custom equals method must follow the standard Object.equals(Object) signature to avoid incorrect behavior, and potential null pointer dereference in ScoreReport needs to be safeguarded with null checks. Additionally, ensure the LaneEvent.frame field is properly initialized to avoid unwritten field issues.**
Classes Involved	● **Bowler.java** ● **ScopeReport.java** ● **LaneEvent.java**

The Refactored Design

Refactoring Identification	**Internationalization Warnings**
Metric Evidence	**3**
Standard Refactoring Pattern	**To address internationalization warnings, the standard refactoring patterns involve ensuring that the code is independent of system-specific settings, such as default encodings, and supports multiple languages and regions. This typically includes replacing hard-coded strings with resource bundles, ensuring consistent encoding practices, and removing dependencies on locale-specific defaults.**
Description Of The Refactoring	**The methods getBowlerInfo, getBowlers, and getScores rely on the platform's default encoding, which can lead to inconsistent behavior. Specify a charset (e.g., UTF-8) when reading files to ensure consistent encoding across all platforms.**
Classes Involved	● **BowlerFile.java** ● **ScoreHistoryFile.java**

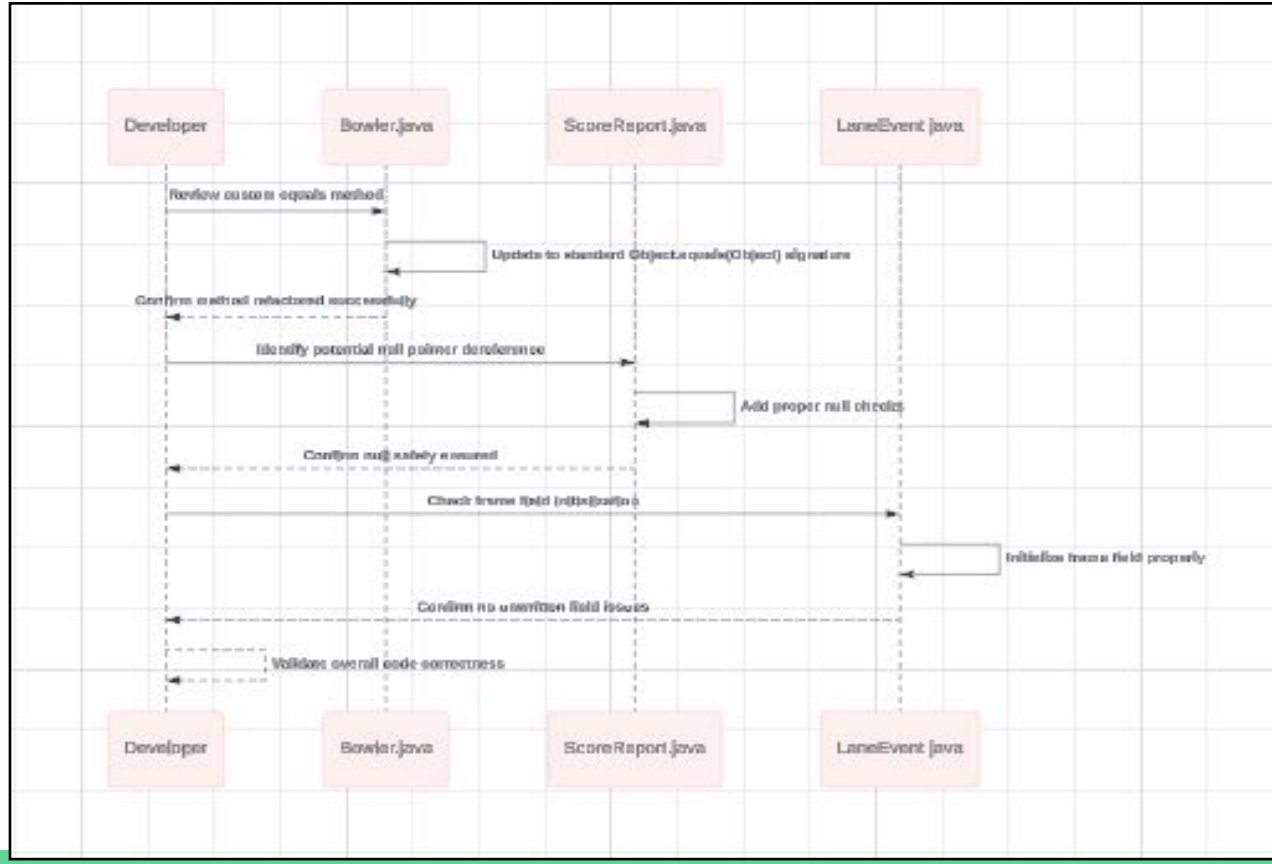
The Refactored Design

Refactoring Identification	**Multithreaded Correctness Warnings**
Metric Evidence	**1**
Standard Refactoring Pattern	**To address multithreaded correctness warnings, the standard refactoring patterns focus on ensuring proper synchronization and thread safety by eliminating race conditions and ensuring thread-safe access to shared resources. This often involves using synchronization mechanisms like synchronized blocks, locks, or atomic variables to prevent concurrent modification issues.**
Description Of The Refactoring	**The new bowling.Lane() constructor calls start(), potentially starting threads before the object is fully constructed, which can cause issues in a multithreaded environment. Refactor the code to delay starting threads until after the object is fully initialized.**
Classes Involved	• **Lane.java**

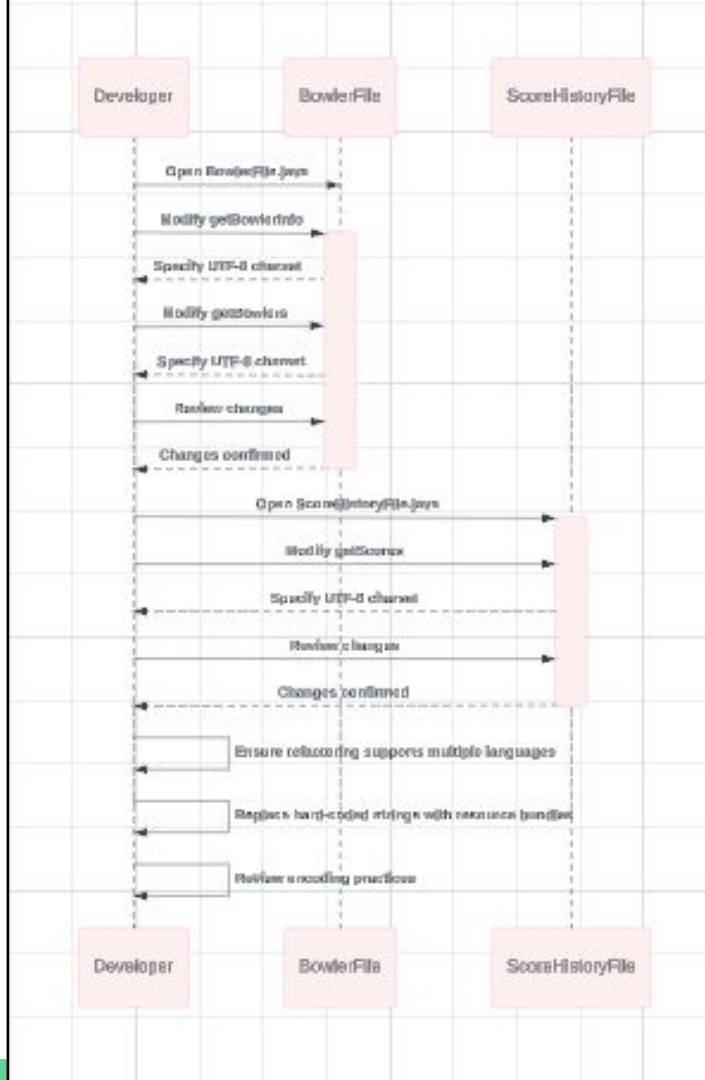
Design Patterns

- **The refactored design retained and optimized the Observer pattern to enhance modularity and maintainability, ensuring components like ControlDesk, Lane, and Pinsetter efficiently notify their observers of state changes.**
- **By refining observer interfaces and notification mechanisms, the system achieved clearer separation of concerns, improved extensibility, and consistent dynamic behavior across views like ControlDeskView, LaneStatusView, and PinsetterView.**

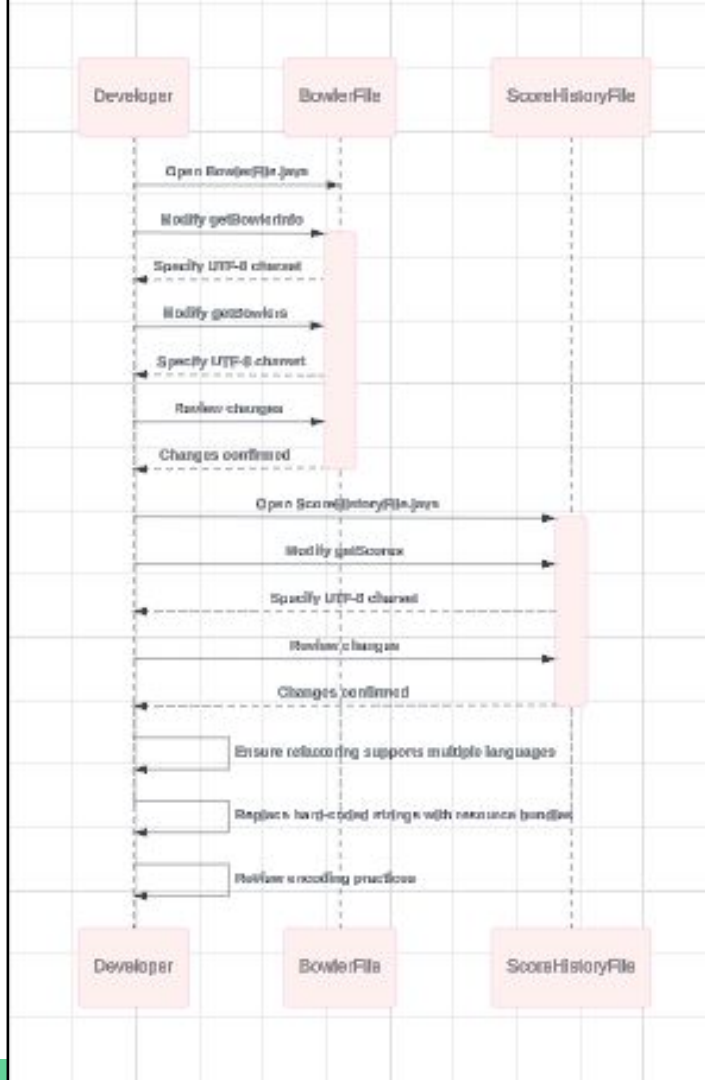
Sequence Diagram



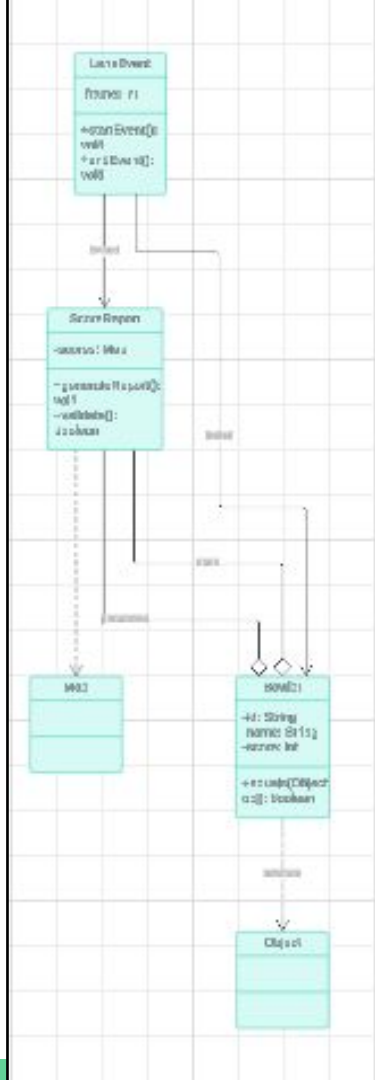
Sequence Diagram



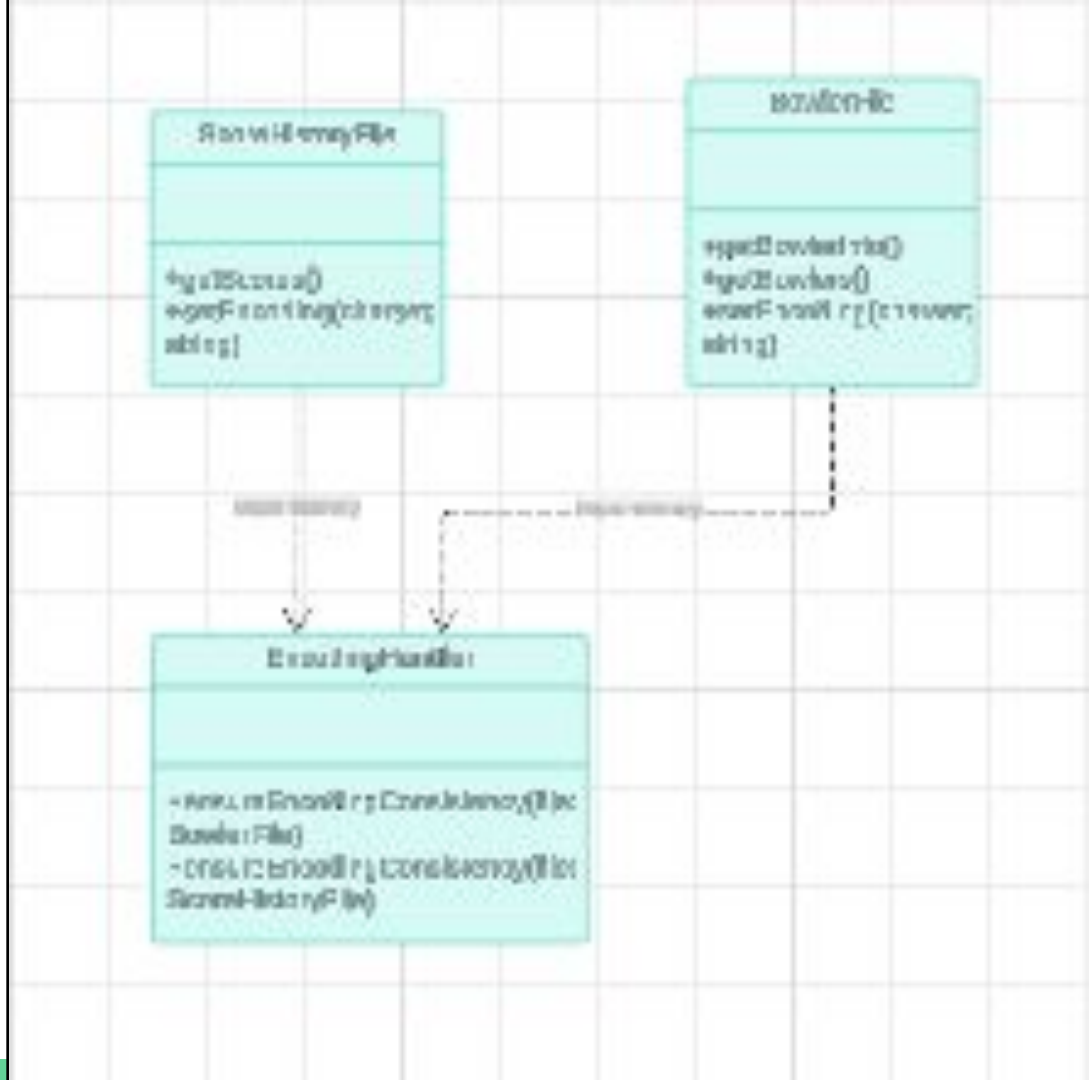
Sequence Diagram



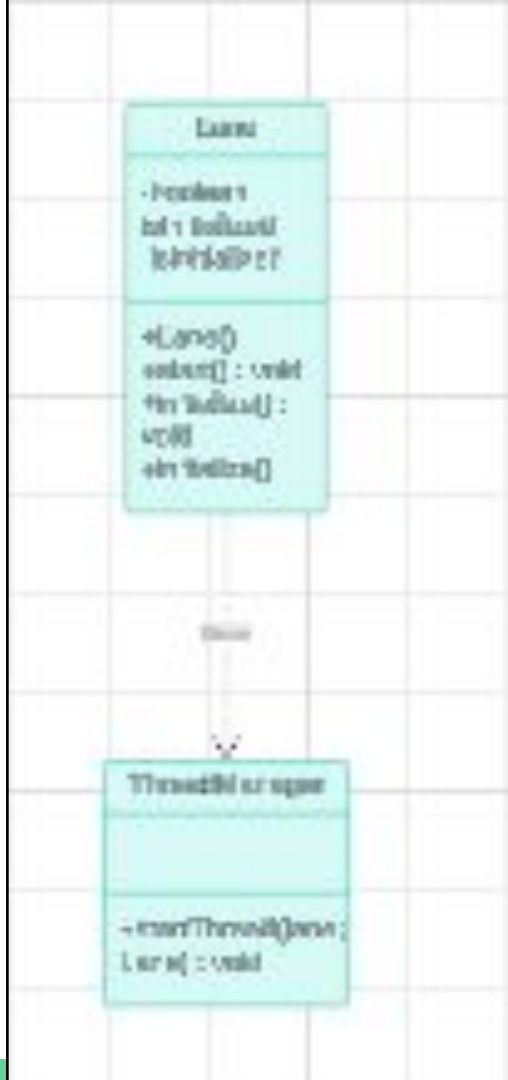
Class Diagram



Class Diagram



Class Diagram



Implementation

- **The team implemented refactorings to improve reliability and maintainability by addressing correctness issues, internationalization, and multithreaded safety.**
- **Key updates included resolving null pointer exceptions, ensuring platform-independent file encoding, and delaying thread starts in Lane until full object initialization, effectively reducing technical debt and aligning the system with robust design principles for future extensibility.**

Metric Analysis

Metrics

1523 lines of code analyzed, in 31 classes, in 1 packages.

Metric	Total	Density*
High Priority Warnings	13	8.54
Medium Priority Warnings	45	29.55
Low Priority Warnings	25	16.41
Total Warnings	**83**	**54.50**

(Defects per Thousand lines of non-commenting source statements)*

Summary

Warning Type	Number
Bad practice Warnings	12
Malicious code vulnerability Warnings	29
Multithreaded correctness Warnings	1
Performance Warnings	18
Dodgy code Warnings	23
Total	**83**

Metric Analysis

- **The refactored codebase reduced total warnings from 97 to 83, with high-priority warnings dropping from 16 to 13 and correctness issues fully resolved, reflecting improved accuracy and maintainability.**
- **While performance and malicious code vulnerability warnings were unchanged, the refactoring significantly improved critical areas, laying a foundation for future optimization.**

Reflection

- **Discovering the design in the existing software underscored the need to understand both functionality and structure, with metric analysis revealing issues like high coupling and correctness flaws that guided refactoring.**
- **Targeted changes, such as refining observers and improving thread safety, highlighted the value of iterative improvements and design patterns in enhancing modularity, maintainability, and system robustness.**

Thank You!

Any Questions?