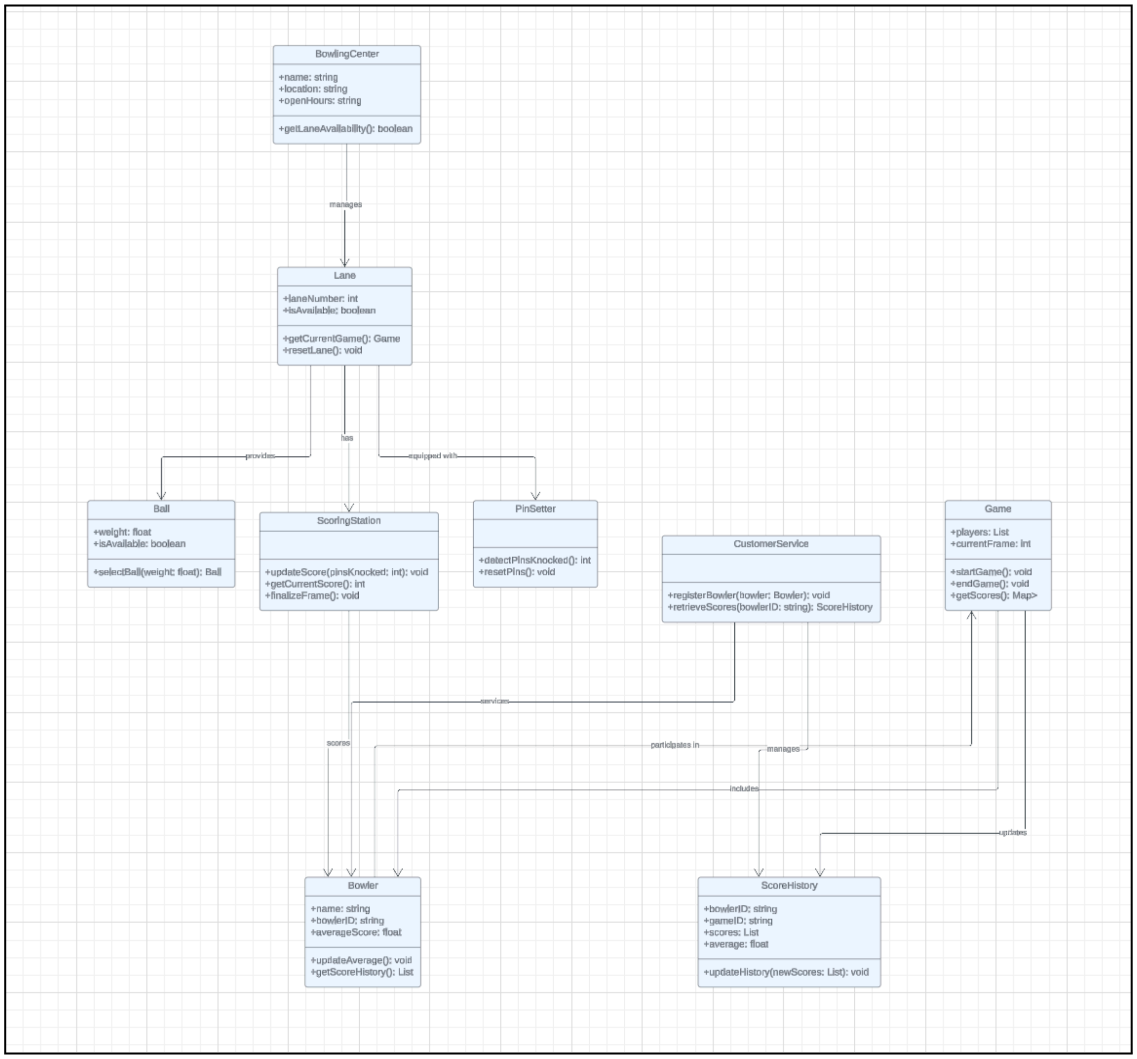
Refactoring Design Documentation

- **Yousaf Iqbal: yyi5708@g.rit.edu**
- **Devin Rhodie: dkr1418@g.rit.edu**
- **Sabrina O’Connor: sno9343@g.rit.edu**
- **Hunter Wells: haw2048@g.rit.edu**
- **Cameron Coleman: csc4122@g.rit.edu**

Product Overview

- **The Lucky Strikes Bowling Center (LSBC) is looking to automate their chain of bowling establishments located across the country. Like many modern bowling facilities, they will be installing new pin setting equipment which can detect the number of pins knocked down after a bowler has rolled his ball. This information can then be communicated to a scoring station that would be able to automatically score the bowler’s game. They would also like to establish a service for their customers that would maintain a history of a bowler’s scores, average and other related information.**

Domain Model

- 

Analysis of Original Design

- **The original design of the bowling alley simulation is modular, with components like PinGUI and LaneGUI handling visual updates, ControlDesk managing backend operations, and Lane acting as the central powerhouse for gameplay and scoring. Data management is handled by classes like Bowler, Party, and BowlerFile, while event-driven communication through LaneEvent and ControlDeskEvent ensures flexibility. The design is cohesive and functional, but some components, like Lane and ControlDesk, risk becoming overly centralized, creating potential challenges for scalability and maintenance. While the division of responsibilities is clear, careful attention is needed to minimize coupling and ensure efficient data synchronization.**

Design Weaknesses and Strengths

- **During our system design evaluation, several strengths were identified, including high cohesion, modularity, event-driven design, and the implementation of the Model-View-Controller (MVC) pattern. The design also incorporated multithreading, which initially appeared to be a strength. However, during implementation, our metric report revealed issues with multithreaded correctness, highlighting it as a challenge.**
- **On the other hand, some weaknesses were also noted. The system violates the Law of Demeter in certain areas, as some subclasses directly access classes they should not interact with. Additionally, the use of the Observer pattern, while effective in some respects, results in unwanted information being available to certain subscribers. This poses potential security risks, especially if the system were used in a larger or more sensitive environment than a bowling alley.**

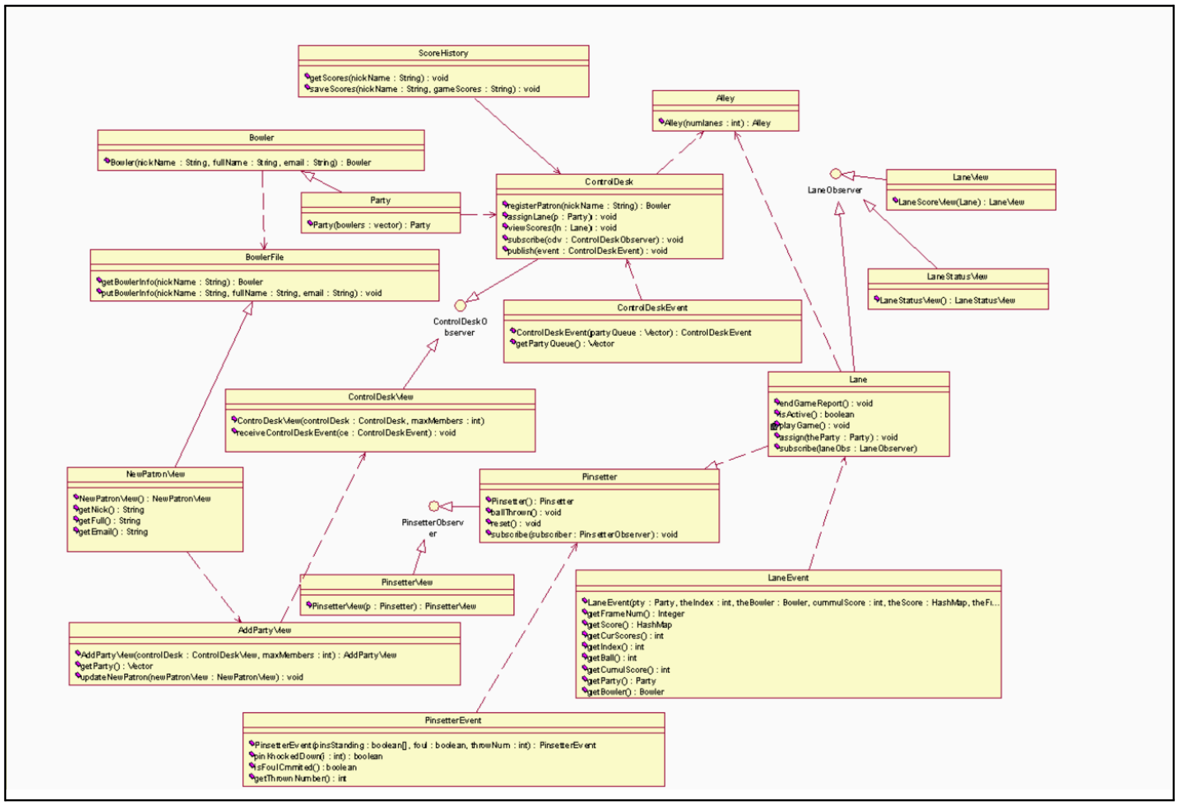
Use of Design Patterns

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		

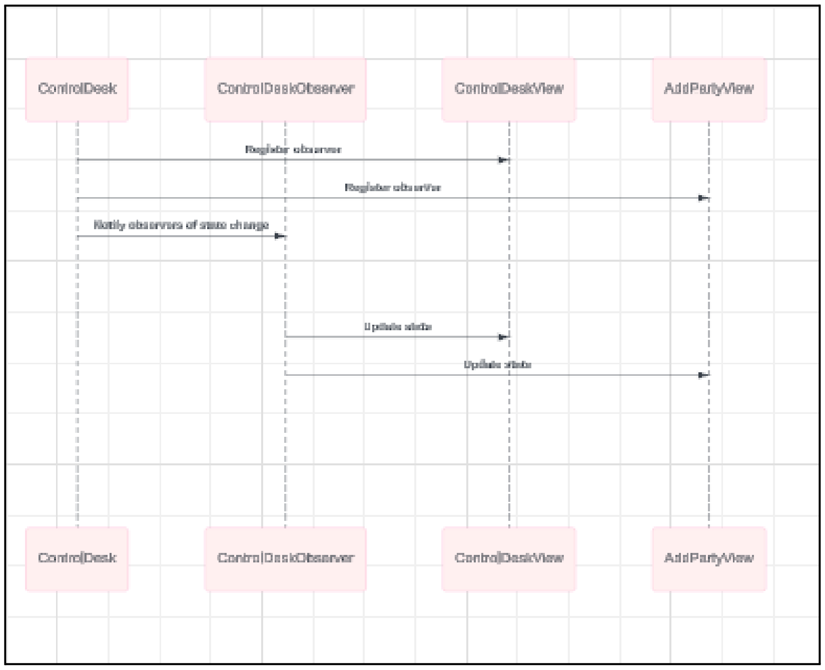
Name: Lane		**GoF Pattern: Observer**
Participants:		
Class:	**Role:**	**Contribution:**
LaneObserver	**Observer**	**Defines an interface for classes interested in updates from the Lane**
LaneView	**Concrete Observer**	**Implements the LaneObserver interface to display lane-specific information**
LaneStatusView	**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		

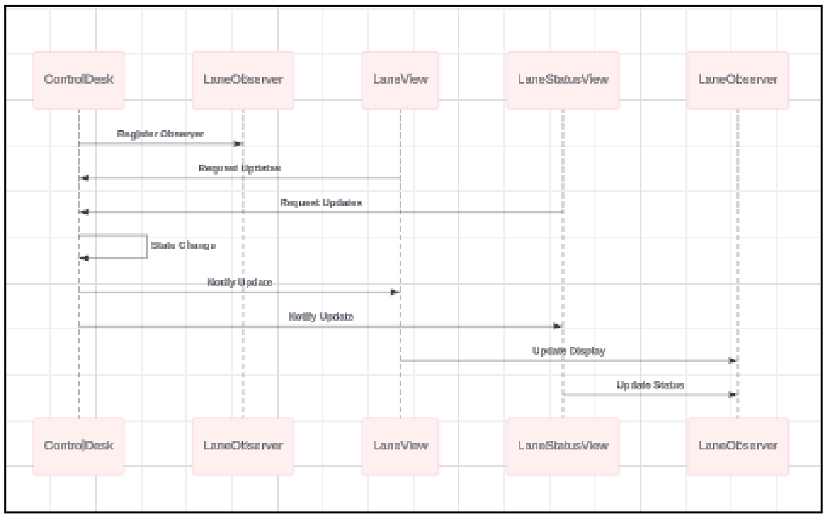
Name: Pinsetter		**GoF Pattern: Observer**
Participants:		
Class:	**Role:**	**Contribution:**
PinsetterObserver	**Observer**	**Defines an interface for receiving updates about pinsetter events**
PinsetterView	**Concrete Observer**	**Implements the PinsetterObserver interface to visually display pinsetter 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		

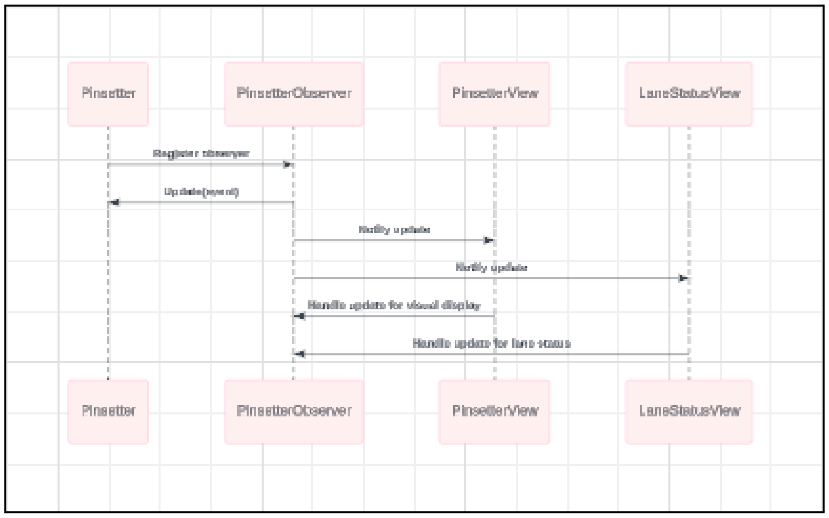
Subsystem and Class Structure


ScoreHistory
getScores(nickName : String) : void
saveScores(nickName : String, gameScores : String) : void
Alley
Alley(numlanes : int) : Alley
Bowler
Bowler(nickName : String, fullName : String, email : String) : Bowler
Party
Party(bowlers : vector) : Party
ControlDesk
registerPatron(nickName : String) : Bowler
assignLane(p : Party) : void
viewScores(ln : Lane) : void
subscribe(cdv : ControlDeskObserver) : void
publish(event : ControlDeskEvent) : void
LaneObserver
LaneView
LaneScoreView(Lane) : LaneView
LaneStatusView
LaneStatusView() : LaneStatusView
BowlerFile
getBowlerInfo(nickName : String) : Bowler
putBowlerInfo(nickName : String, fullName : String, email : String) : void
ControlDeskObserver
ControlDeskEvent
ControlDeskEvent(partyQueue : Vector) : ControlDeskEvent
getPartyQueue() : Vector
Lane
endGameReport() : void
isActive() : boolean
playGame() : void
assign(theParty : Party) : void
subscribe(laneObs : LaneObserver)
ControlDeskView
ControDeskView(controlDesk : ControlDesk, maxMembers : int)
receiveControlDeskEvent(ce : ControlDeskEvent) : void
NewPatronView
NewPatronView() : NewPatronView
getNick() : String
getFull() : String
getEmail() : String
Pinsetter
Pinsetter() : Pinsetter
ballThrown() : void
reset() : void
subscribe(subscriber : PinsetterObserver) : void
PinsetterObserver
PinsetterView
PinsetterView(p : Pinsetter) : PinsetterView
LaneEvent
LaneEvent(pty : Party, theIndex : int, theBowler : Bowler, cummulScore : int, theScore : HashMap, theFi...
getFrameNum() : Integer
getScore() : HashMap
getCurScores() : int
getIndex() : int
getBall() : int
getCumulScore() : int
getParty() : Party
getBowler() : Bowler
AddPartyView
AddPartyView(controlDesk : ControlDeskView, maxMembers : int) : AddPartyView
getParty() : Vector
updateNewPatron(newPatronView : NewPatronView) : void
PinsetterEvent
PinsetterEvent(pinsStanding : boolean[], foul : boolean, throwNum : int) : PinsetterEvent
pinKnockedDown(i : int) : boolean
isFoulCmmited() : boolean
getThrownNumber() : int

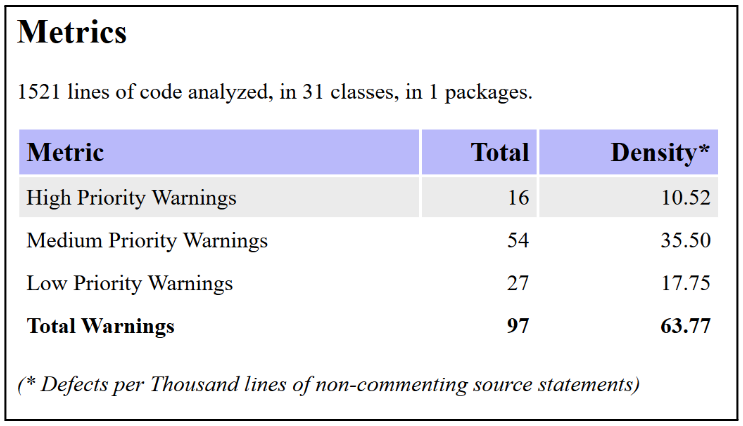
Sequence Diagrams

- 

- 

- 

Metric Analysis

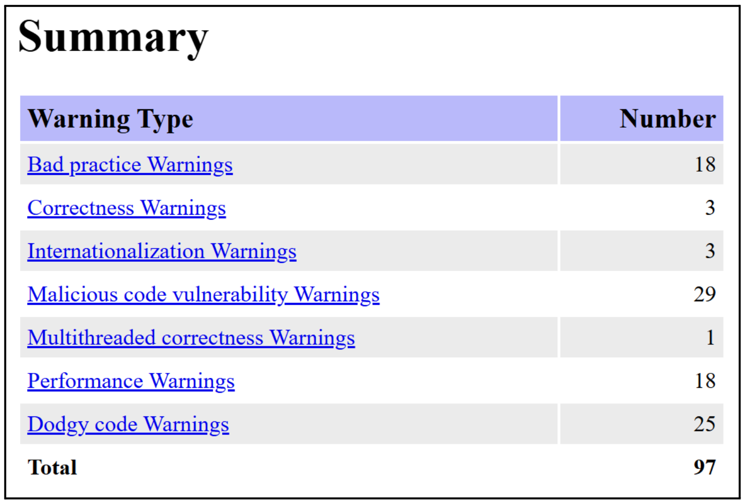
- 

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**

- **The initial design included numerous medium-priority warnings, resulting in a very high density of issues. Our efforts focused on reducing this density in a manageable way, given the time constraints. We prioritized addressing the three types of warnings: correctness warnings, internationalization warnings, and multithreaded correctness warnings. These were chosen because they provided the best opportunity to improve the system significantly without risking major disruptions. Each selected warning listed was relatively straightforward to fix or critical to the system, making them logical targets for our efforts.**

The Refactored Design

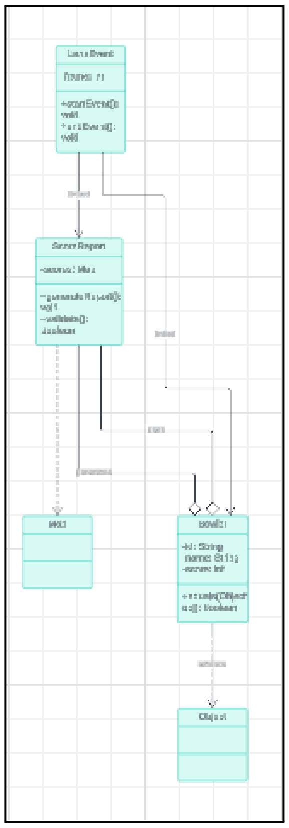
- **The refactored design improves maintainability and robustness by focusing on low coupling, high cohesion, and proper encapsulation. Correctness issues, such as null pointer dereferences and improper initialization, were addressed to ensure reliable functionality. Internationalization was enhanced by specifying consistent encoding practices, making the system platform-independent and adaptable to different locales. Multithreading safety was improved by delaying thread starts until object initialization is complete, preventing race conditions. Overall, the design adheres to key principles like separation of concerns, the Law of Demeter, and extensibility, creating a more modular, reusable, and future-proof system.**

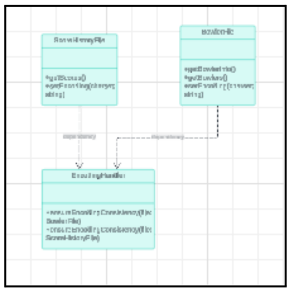
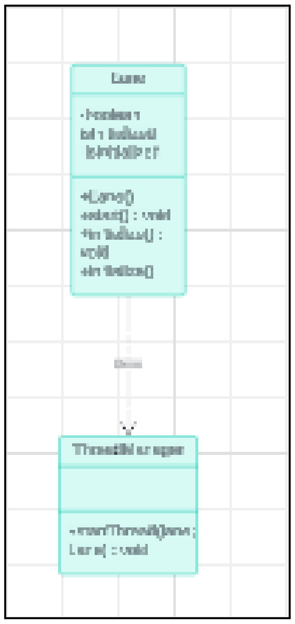
Refactoring Identification	Correctness Warnings
Metric Evidence	3
Standard Refactoring Pattern	To address correctness warnings, the standard refactoring patterns focus on ensuring that the code behaves as intended by eliminating potential errors, such as null pointer dereferences and incorrect method 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

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

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

Refactored Class Structure

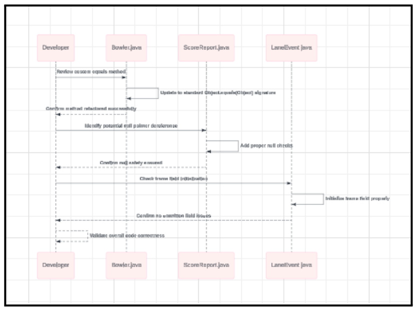
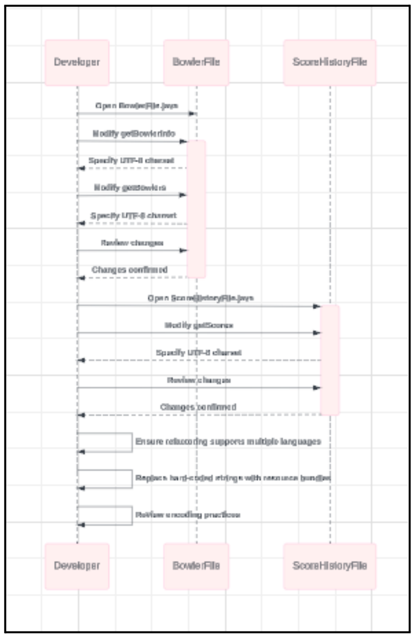
- 

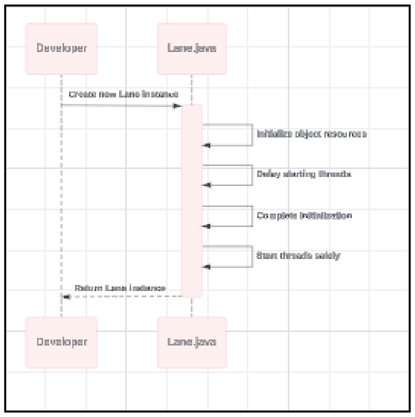
- 
- 

Design Patterns

- **In the refactored design, we continued to utilize the Observer pattern, enhancing its implementation for improved modularity and maintainability. The pattern was already integral to components like ControlDesk, Lane, and Pinsetter, where subjects notify their respective observers of state changes. By adhering to the Observer pattern, we ensured a clear separation of concerns, allowing views like ControlDeskView, LaneStatusView, and PinsetterView to react dynamically to updates without directly coupling to the core logic. The refactoring focused on optimizing the pattern's use by refining observer interfaces and ensuring proper notification mechanisms. This approach strengthened the design's extensibility and reusability while maintaining consistent behavior across the system.**

Sequence Diagrams

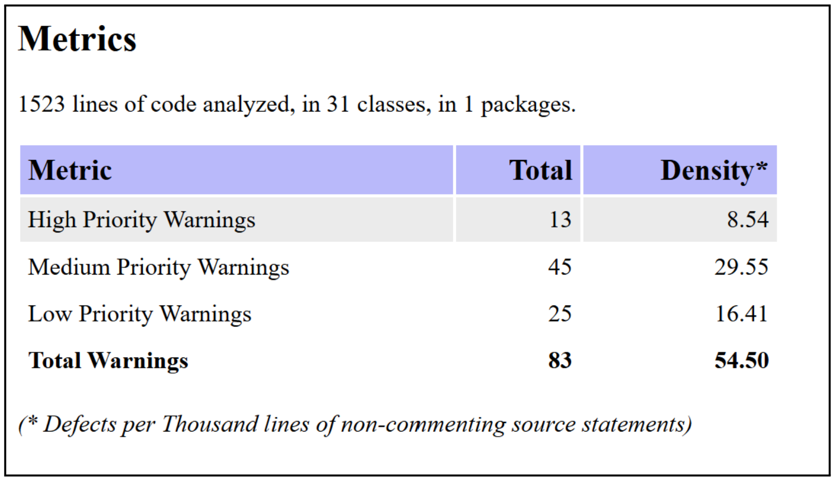
- 
Developer
Bowler.java
ScoreReport.java
LaneEvent.java
Review custom equals method
Update to standard Object.equals(Object) signature
Confirm method refactored successfully
Identify potential null pointer dereference
Add proper null checks
Confirm null safety ensured
Check frame field initialization
Initialize frame field properly
Confirm no unwritten field issues
Validate overall code correctness
- 

- 

Implementation

- **The team implemented refactorings to address correctness, internationalization, and multithreaded safety, improving the system’s reliability and maintainability. Correctness issues, like null pointer exceptions and improper initialization, were resolved in classes such as Bowler, ScoreReport, and LaneEvent. The potential null pointer dereference was resolved by asserting that the variable being dereferenced is not null. To ensure platform-independent encoding and prepare for future internationalization, file-handling methods in BowlerFile and ScoreHistoryFile were updated. Multithreaded safety was improved by delaying thread starts in Lane until objects were fully initialized, preventing race conditions. These refactorings addressed critical technical debt and aligned the system with robust design principles for future extensibility.**

Metric Analysis

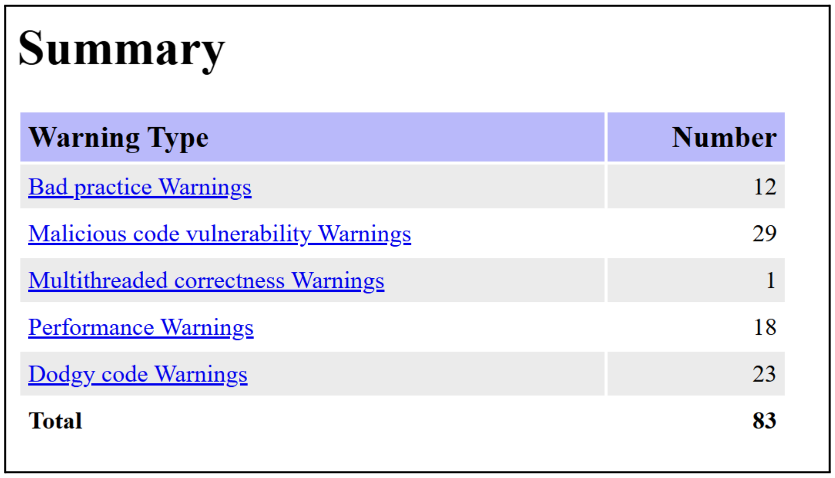
- 

  ### 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**

-
- **The refactored codebase shows a reduction in total warnings, improving from 97 to 83, with a lower density of issues per 1,000 lines. High-priority warnings decreased from 16 to 13, and correctness issues were fully resolved, reflecting improved logical accuracy. Bad practice and dodgy code warnings also saw reductions, indicating better adherence to coding standards and improved code maintainability. However, performance and malicious code vulnerability warnings remained unchanged, as these areas were not a primary focus of the refactoring. Overall, the refactoring significantly enhanced code quality in critical areas while leaving some opportunities for future optimization.**

Reflection

- **Discovering the design in the existing software system highlighted the importance of understanding both the intended functionality and the underlying structure. Metric analysis proved invaluable in identifying problem areas, such as high coupling, lack of cohesion, and potential correctness issues, guiding our refactoring efforts. Refactoring reinforced the benefits of adhering to design principles, such as improving modularity, enhancing maintainability, and reducing technical debt. The team observed that small, targeted changes, like refining observer implementations and ensuring thread safety, can significantly improve system robustness. Overall, the process emphasized the value of iterative improvements and the role of design patterns in creating extensible and reliable software.**