Software Engineering Assignment

Bowling Alley Simulation Refactoring

Design Document

(Updated)

Team - 15

Date of Submission: 20th March 2021

Overview of the Project

This project contains a java application that allows for the management of a bowling alley. In this application, we can print the bowler's score at the end of the game. We also have a control desk through which we can monitor the collection of lanes, i.e. a group of bowlers can be assigned to some lanes. We can also view the configuration of the pinsetter. We have been provided with code, designs and documentation. Our task is to reverse engineer the design from existing code and documentation(if any) and propose refactoring to improve the program's structure for future maintenance and evolution.

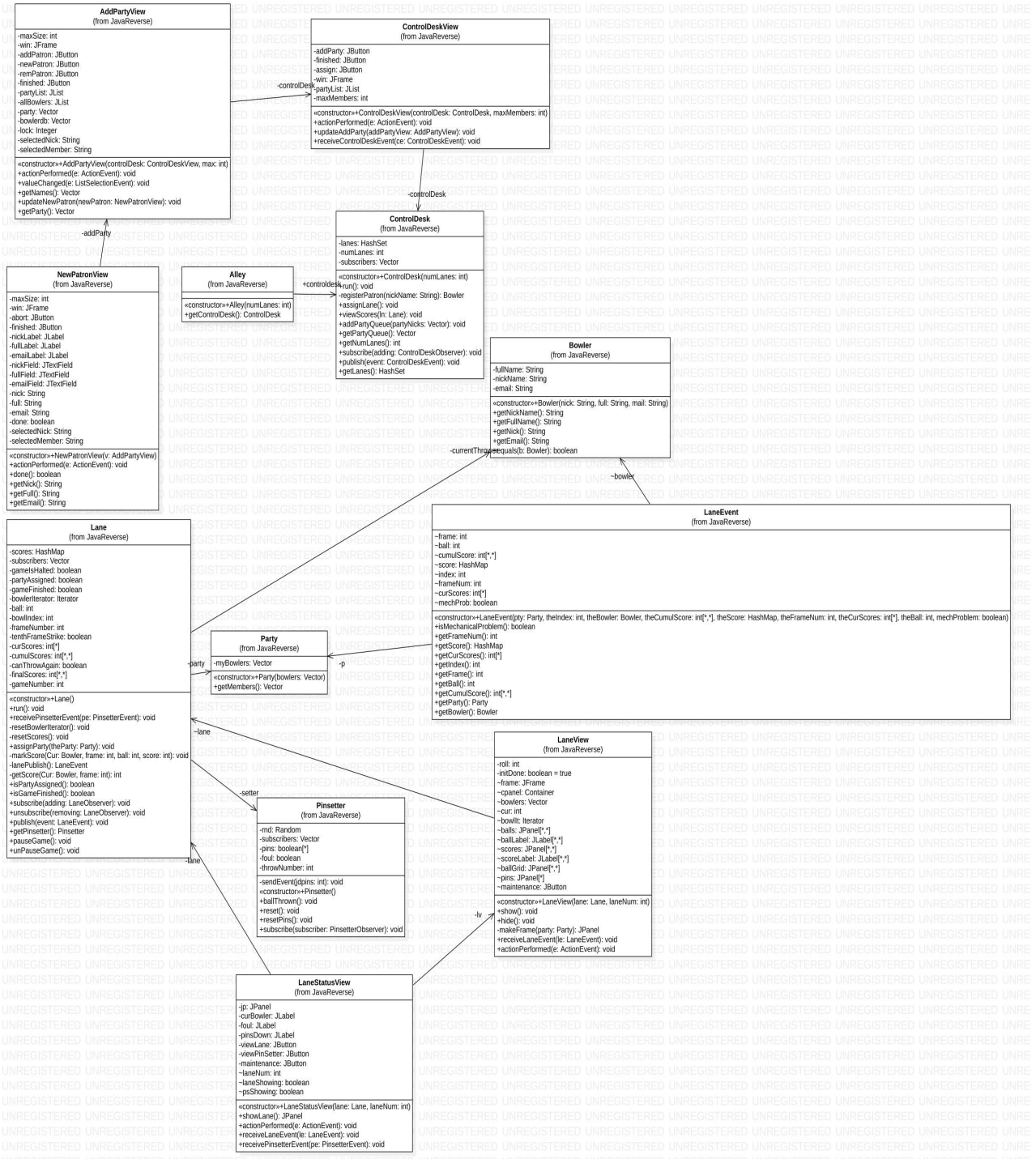
Technologies used:

StarUML	To create UML Class diagrams
Eclipse / Intellij	IDE for running code
Intellij/Eclipse - codeMR	Plugin for analysing metrics
Intellij - SequenceDiagram	To create UML Sequence diagrams
Eclipse - AutoRefactor	Plugin for making an initial set of refactoring to the code.

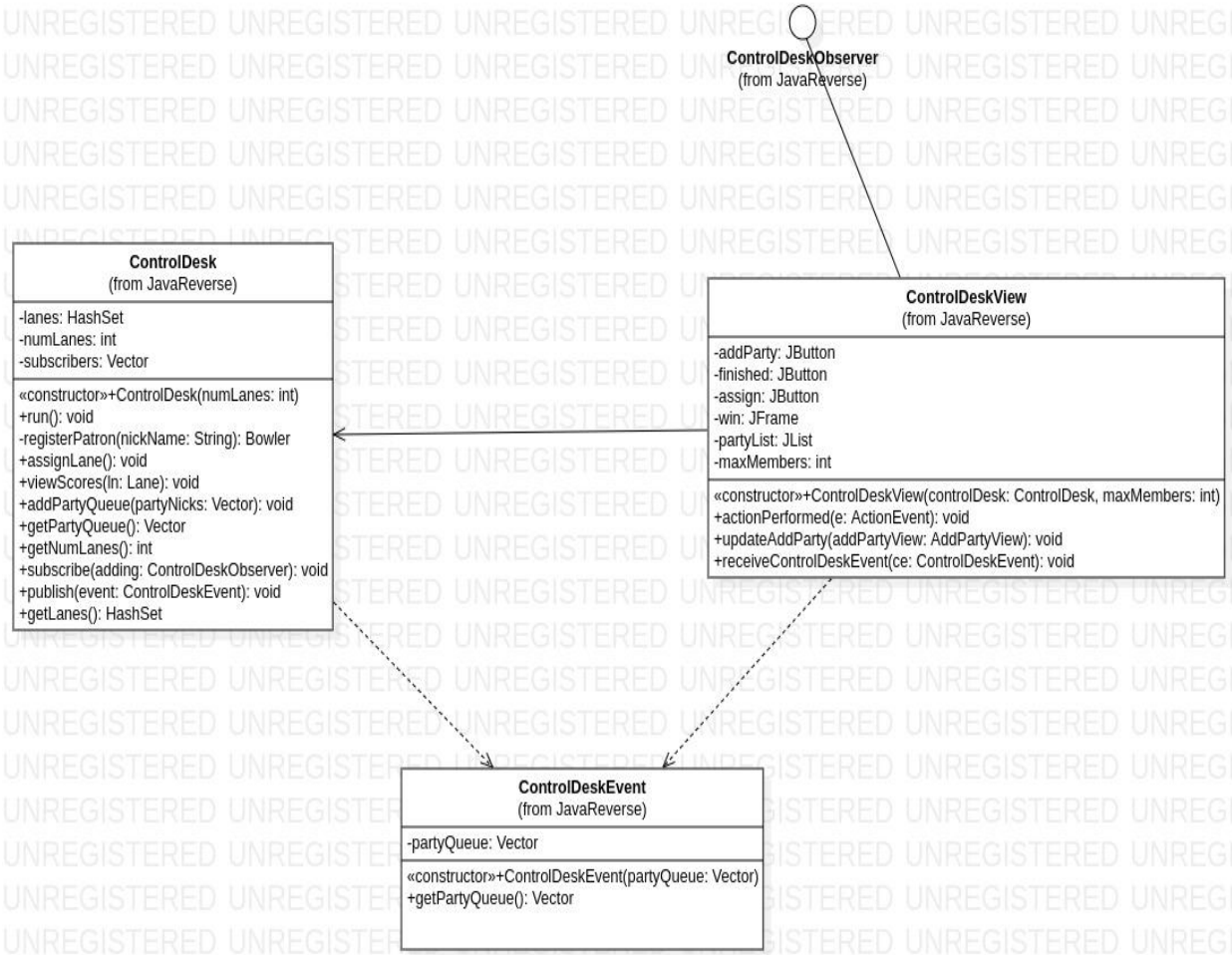
UML Diagrams for the Existing Code

UML Class Diagram:

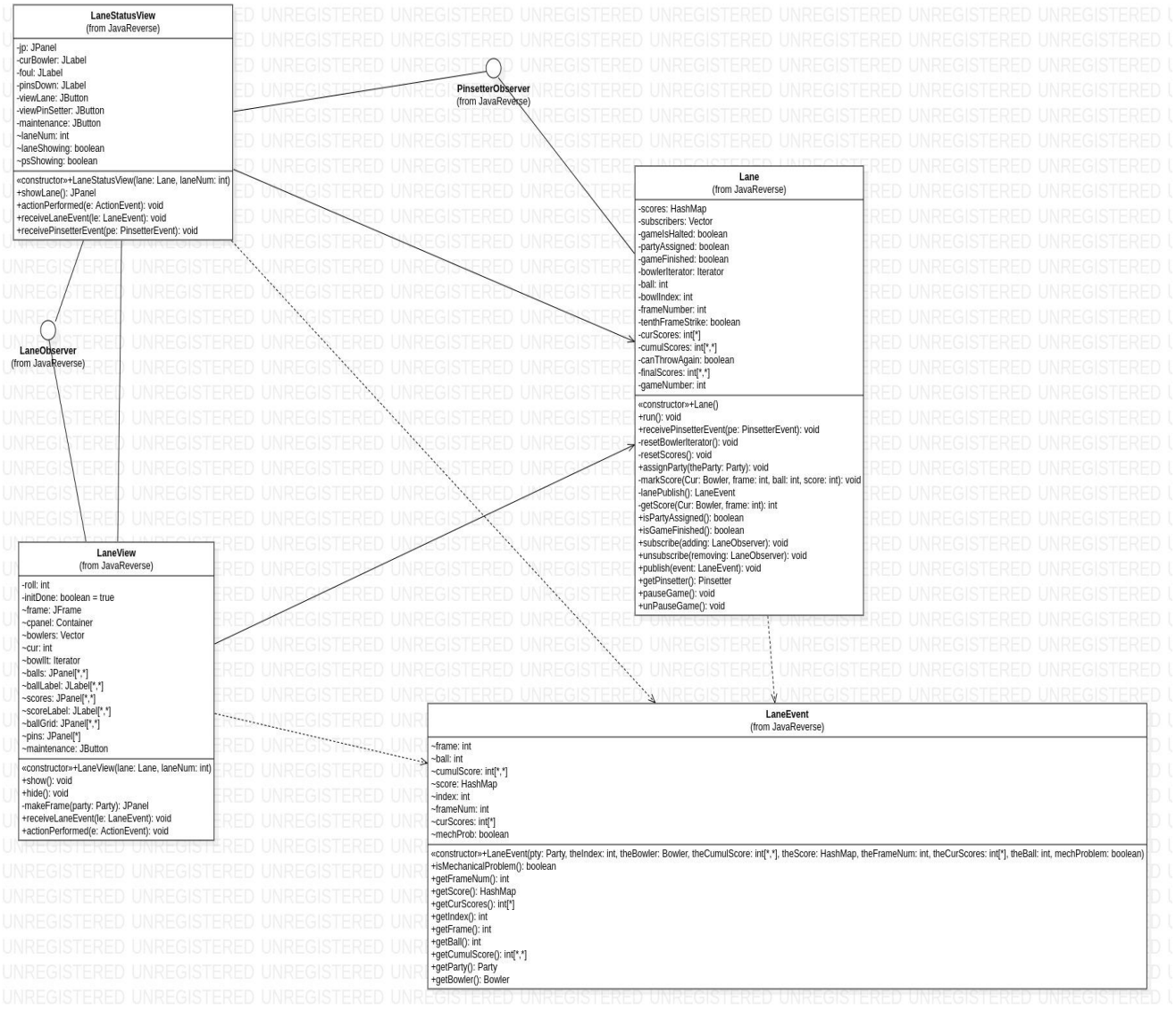
OVERVIEW OF BOWLING ALLEY-



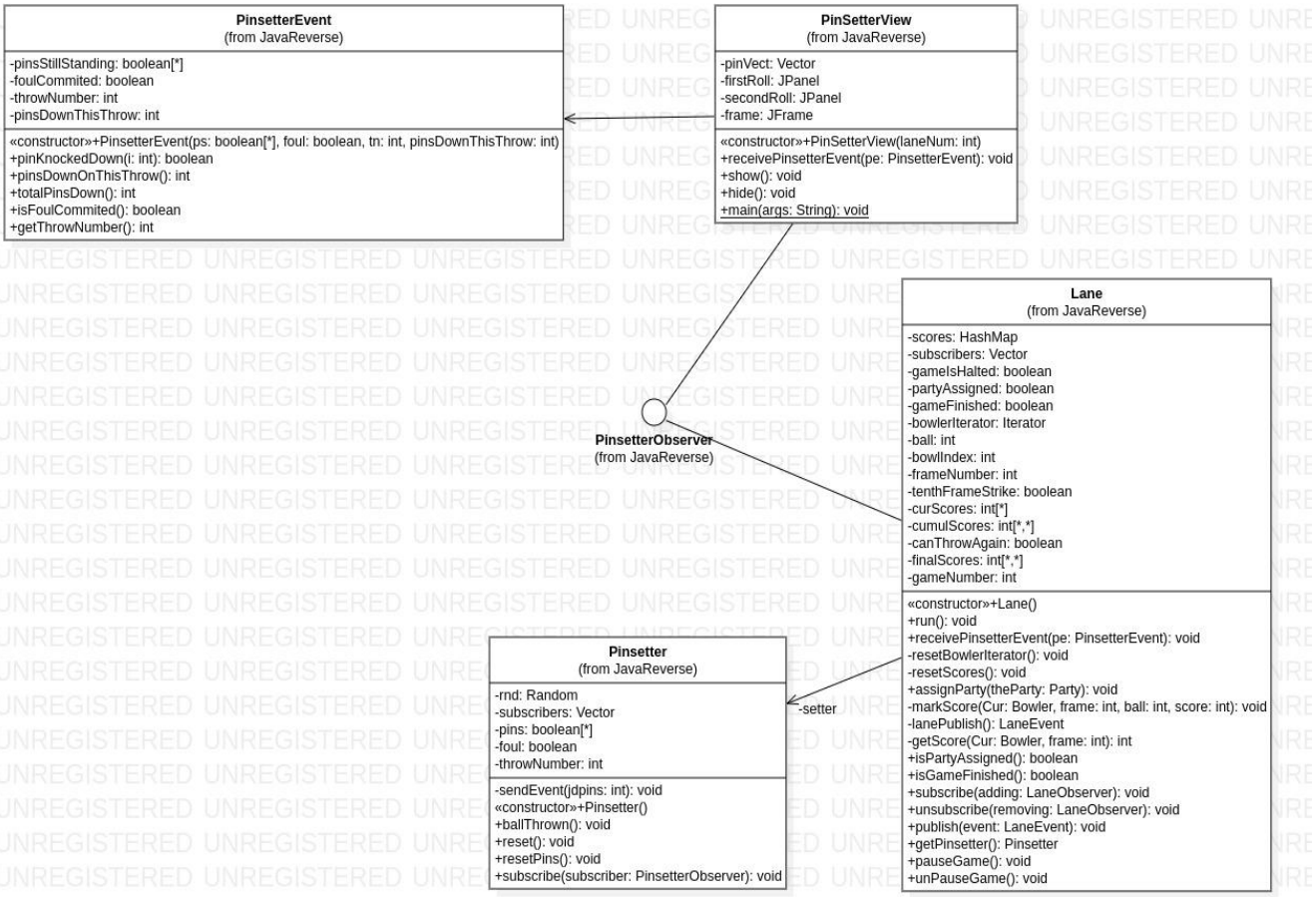
OVERVIEW OF CONTROL DESK AND ASSOCIATED CLASSES-



OVERVIEW OF LANE AND ITS ASSOCIATED CLASSES

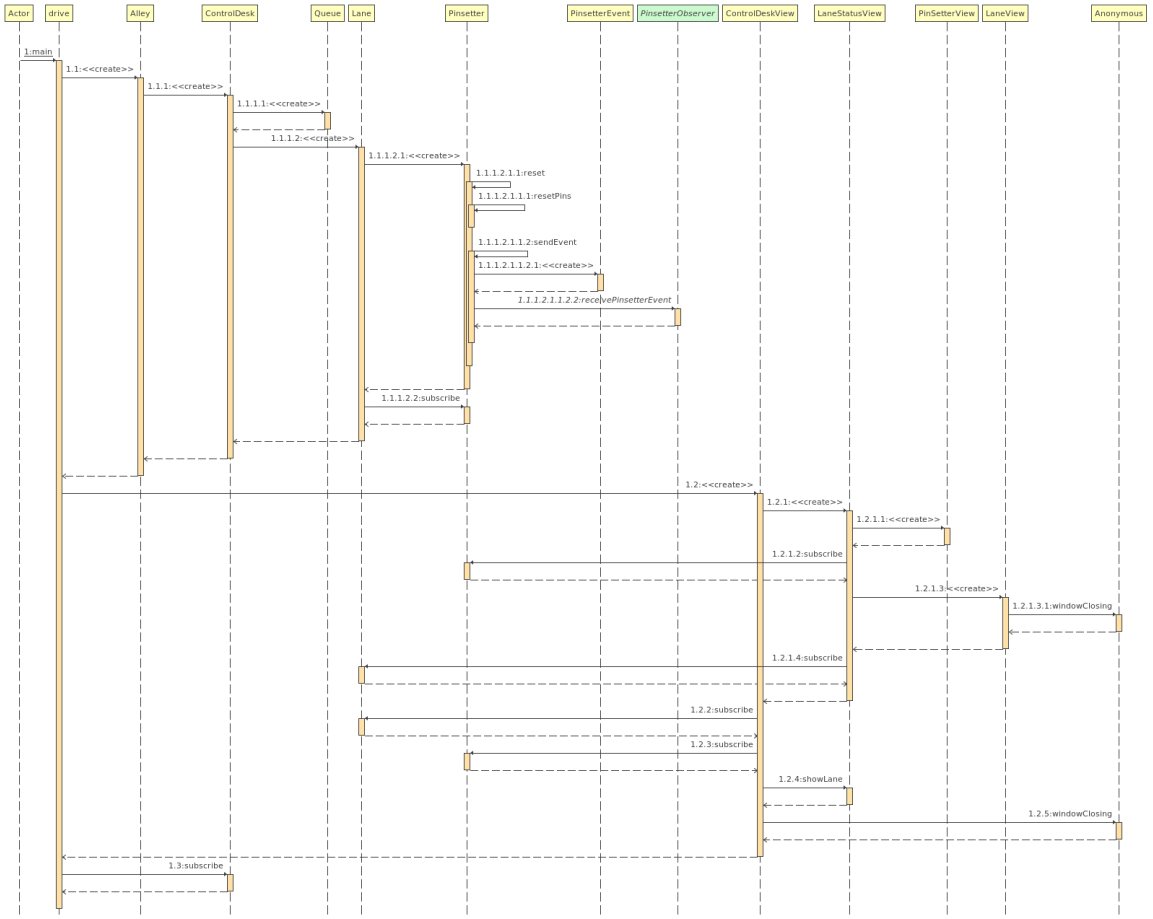


OVERVIEW OF PINSETTER AND ASSOCIATED CLASSES

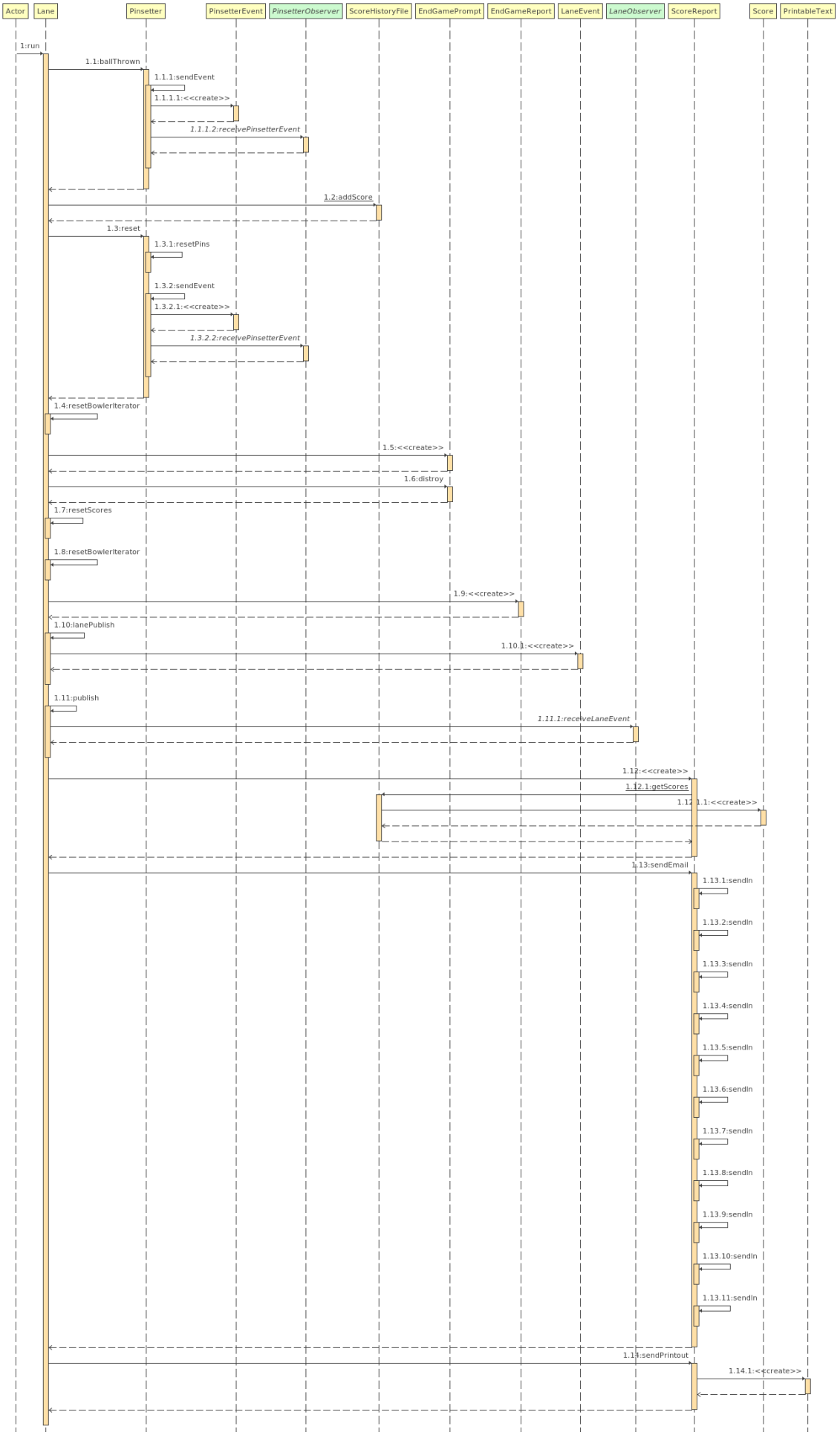


UML Sequence Diagram

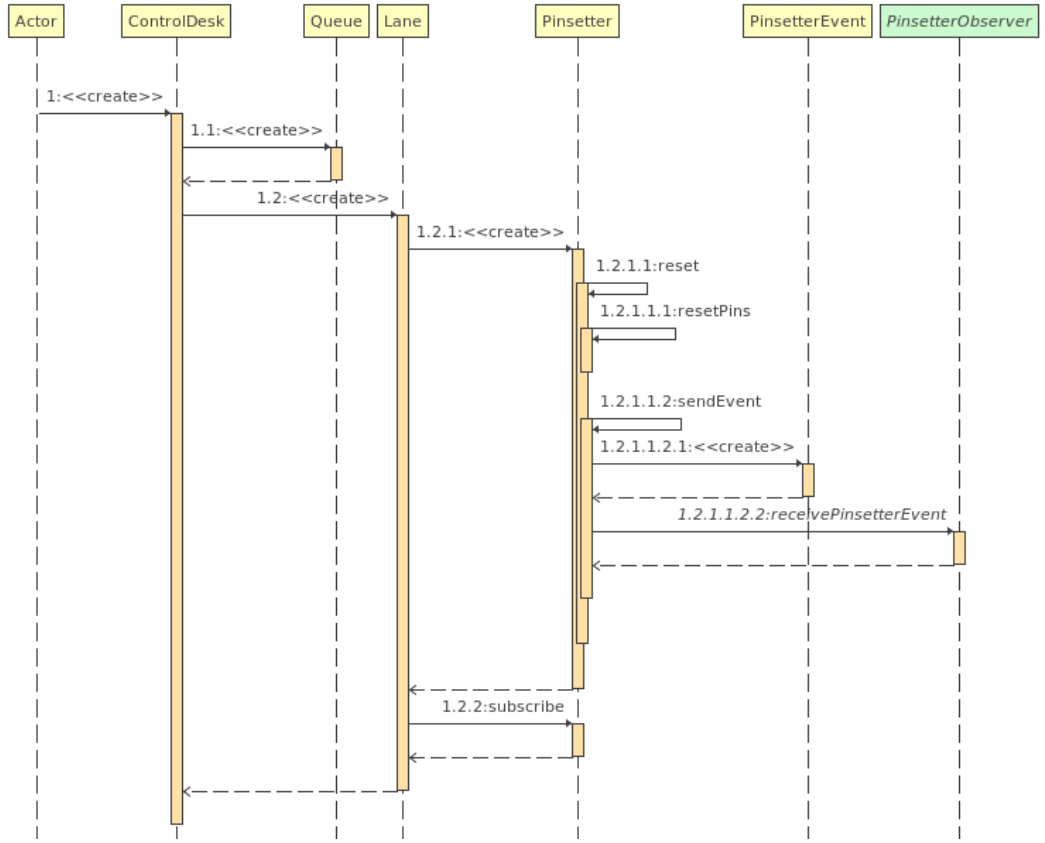
Full Sequence Diagram of the OLD CODE:



SEQUENCE DIAGRAM OF LANE CLASS (WITHOUT REFACTORING):


Actor
Lane
Pinsetter
PinsetterEvent
PinsetterObserver
ScoreHistoryFile
EndGamePrompt
EndGameReport
LaneEvent
LaneObserver
ScoreReport
Score
PrintableText
1:run
1.1:ballThrown
1.1.1:sendEvent
1.1.1.1:<<create>>
1.1.1.2:receivePinsetterEvent
1.2:addScore
1.3:reset
1.3.1:resetPins
1.3.2:sendEvent
1.3.2.1:<<create>>
1.3.2.2:receivePinsetterEvent
1.4:resetBowlerIterator
1.5:<<create>>
1.6:distroy
1.7:resetScores
1.8:resetBowlerIterator
1.9:<<create>>
1.10:lanePublish
1.10.1:<<create>>
1.11:publish
1.11.1:receiveLaneEvent
1.12:<<create>>
1.12.1:getScores
1.12.1.1:<<create>>
1.13:sendEmail
1.13.1:sendln
1.13.2:sendln
1.13.3:sendln
1.13.4:sendln
1.13.5:sendln
1.13.6:sendln
1.13.7:sendln
1.13.8:sendln
1.13.9:sendln
1.13.10:sendln
1.13.11:sendln
1.14:sendPrintout
1.14.1:<<create>>

SEQUENCE DIAGRAM OF CONTROL DESK(Without Refactoring):



Class Responsibility Table

CLASS	RESPONSIBILITY
drive	Class that starts the program by initialising parameters and calling other relevant class objects
ControlDesk	Assigns lanes to arriving parties and maintains the wait queue
ControlDeskView	Launches the controlDesk window to add parties and start the game.
ControlDeskEvent	Maintains queue of parties waiting
Lane	Creates a thread for every lane, where the game happens. Simulates the game of a player and calculates score.
LaneView	Displays lane window where the game is happening
LaneEvent	Maintains various events occurring in the lane and reporting them to relevant classes for action
LaneStatusView	Handles display of lane activities
Bowler	Class to store bowler's information - full name, nickname and email id
BowlerFile	Class to retrieve information from the bowler database.
Score	Class to store score information of the bowler. Encapsulates bowler's nickname, date of game and its score
ScoreHistoryFile	Class to retrieve score related information of bowler from database
ScoreReport	Class to display score related information

Alley	Container class called by driver that simulates a bowling alley and instantiates ControlDesk
Pinsetter	Simulates the action of ball hitting the pins during the game
PinSetterView	Creates pinsetter GUI displaying which roll it is
PinsetterEvent	Observes the pins and reports the actions back to relevant classes
Party	Class that holds bowlers
Queue	Class to create queue data structure
AddPartyView	Class to handle GUI of adding parties to play
EndGamePrompt	Display a prompt that shows up when game ends, and handles input provided
EndGameReport	Displays the report at the end of the game

Analyzing Original Code

Weakness

The source material that consists of code, documentation and designs exhibits poor software engineering design principles and anti-patterns.

Anti-Patterns

1.) Cut and paste Programming

Weakness	Fixed
Dead code in many modules	Refactored and eliminated using the DRY principle

2.)Functional Decomposition

Weakness	Fixed
Single method classes with no significant use	Merged into existing classes

3.)The Blob

Weakness	Fixed
It's just opposite to the above weakness i.e. single controller class in which multiple simple data classes can be seen throughout such as Lane.java	Created appropriate classes and distributed accordingly

4.)Lava Flow

Weakness	Fixed
Commented codes with no explanation with leftovers i.e. to be done codes	Removed Dead code and checked if any bugs present and wrote necessary code.

5.)Golden Hammer

Weakness	Fixed
Some deprecated functions were used all over the code	Fixed using Intellij

Fidelity to Design Documents:

- On each lane, the bowling alley consists of a total number of bowling lanes and bowl classes.
- Each lane consists of an autonomous scoring system that records the names of the bowlers, their scores, and then displays a graphical representation of their scores.
- The scoring station now receives an update of the remaining positions from the pinsetter interface after each bowler completes a pass.

Code Smells

Filename	Problem	Solution
Lane.java	Just a large class from which all subclasses are inheriting properties.	Reconstructed a comparatively smaller class and removed unnecessary inheritance.
ControlDesk.java	Just a large class from which all subclasses are inheriting properties.	Restructured the Class to reduce complexity.
All .java files	In terms of syntax and formatting, there are many Javadoc errors	Improvised and added new comments explaining the code

Laneserver.java	A loop in which an empty statement which is not a part of it.	Removed empty statement
LaneEvent.java	A loop in which an empty statement which is not a part of it.	Removed empty statement
LaneView.java	New Integer() and show() are Deprecated function	Used win.setVisible(True) instead of show()
LaneStatusView.java	Commented code statements New Integer() is an Deprecated function	Used Integer.valueof( ) instead of Integer
NewPatronView.java	hide() and show() are Deprecated function	Used win.setVisible(True/False) instead of show() and hide()

LaneStatusView.java	Redundant conditional statements
	Ex:-
	if (lane.isPartyAssigned())
	{
	if (laneShowing == false)
	{
	lv.show();
	
	laneShowing=true;
		All unnecessary conditional statements combined.

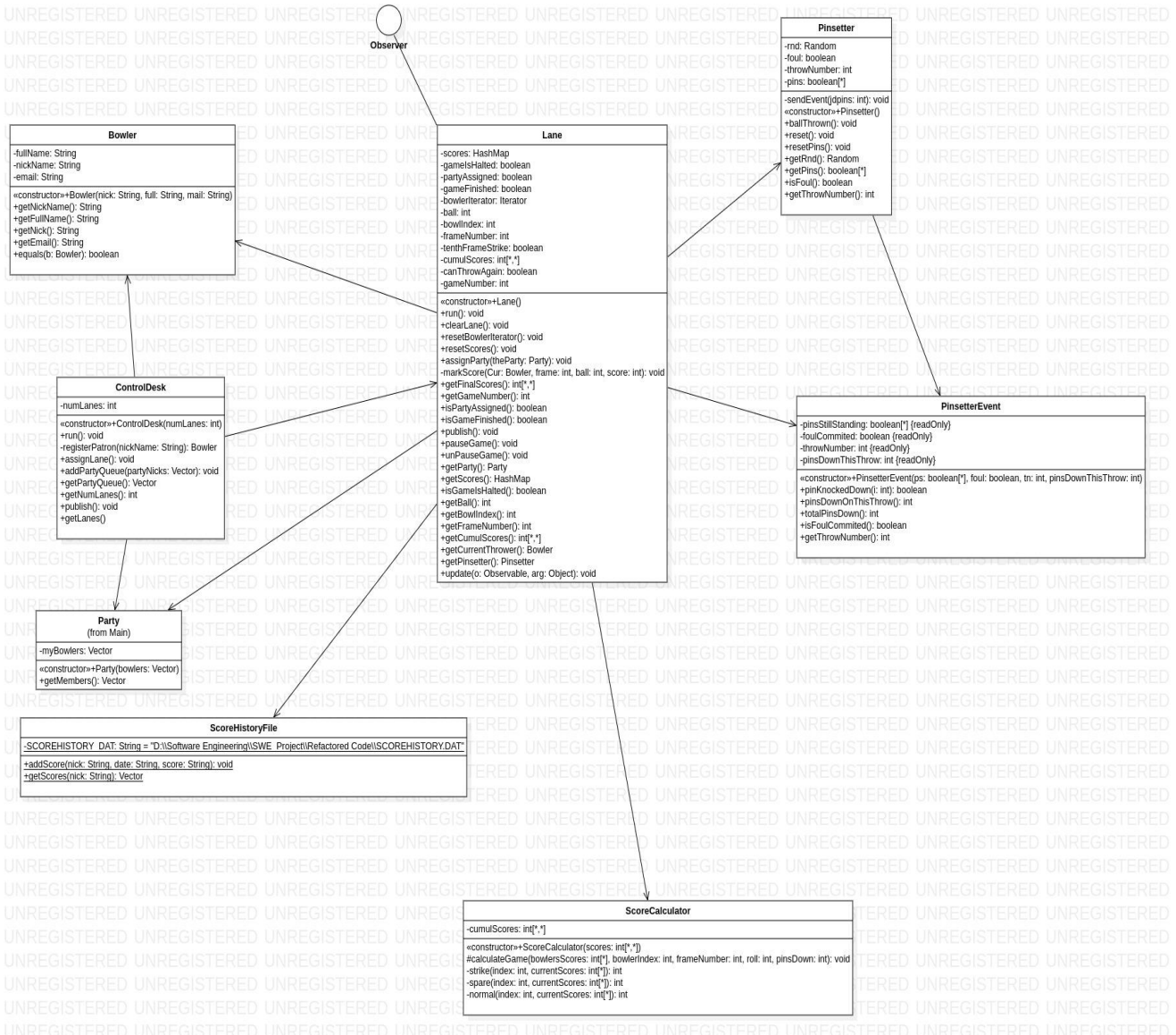
All .java files	Redundant code	Refactored

-Button.java -New PatronView.java -EndGameprompt.java -Addpartyview.java -ControlDeskview.java -LaneStatusview.java -ViewLane.java -ViewPinSetter	There are several functions who are utilizing the same code for implementing buttons and windows	Removing the redundancy use of the same line of code by calling it via object.

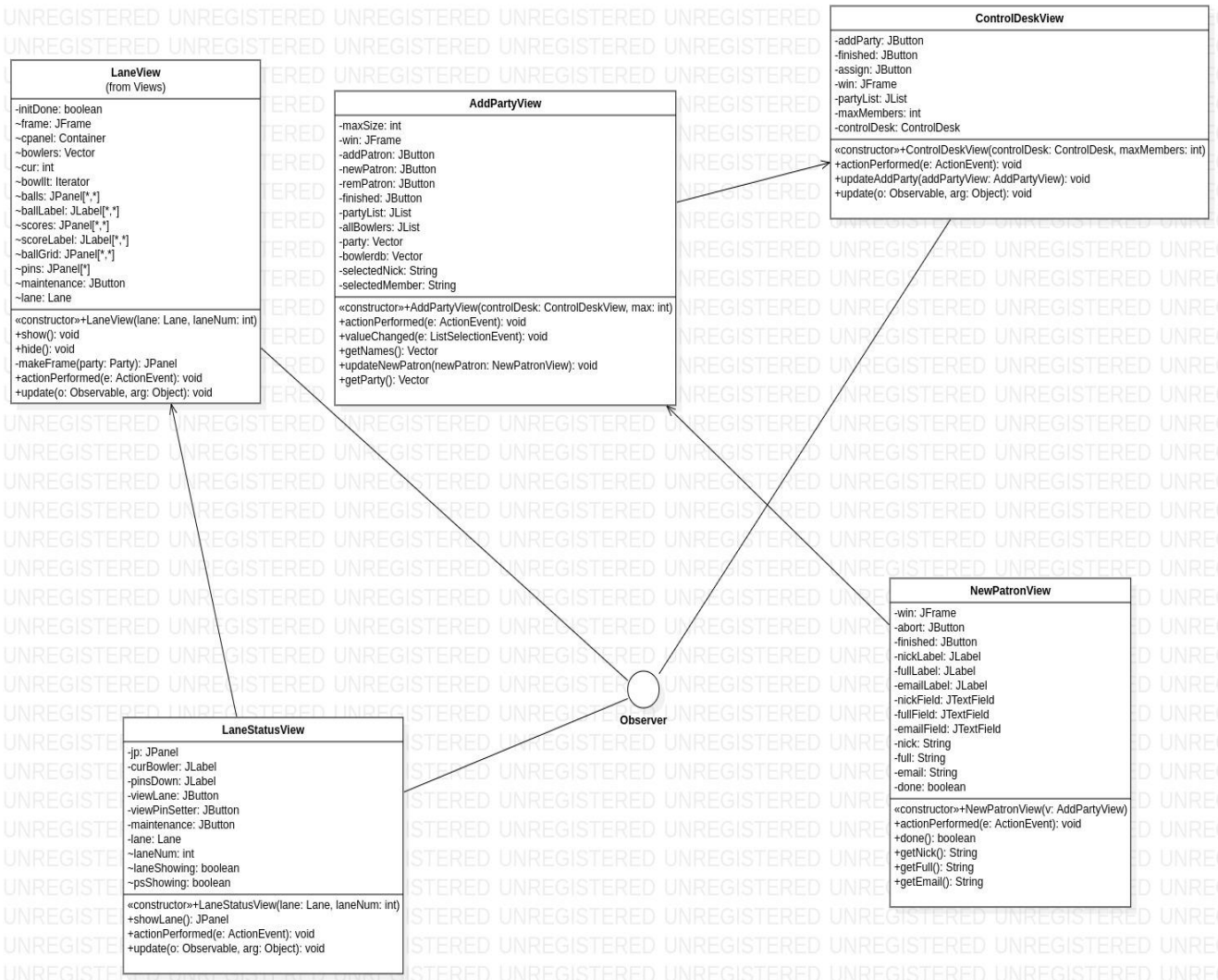
UML Diagrams for the refactored code

UML Class Diagram

MAIN PACKAGE CLASSES

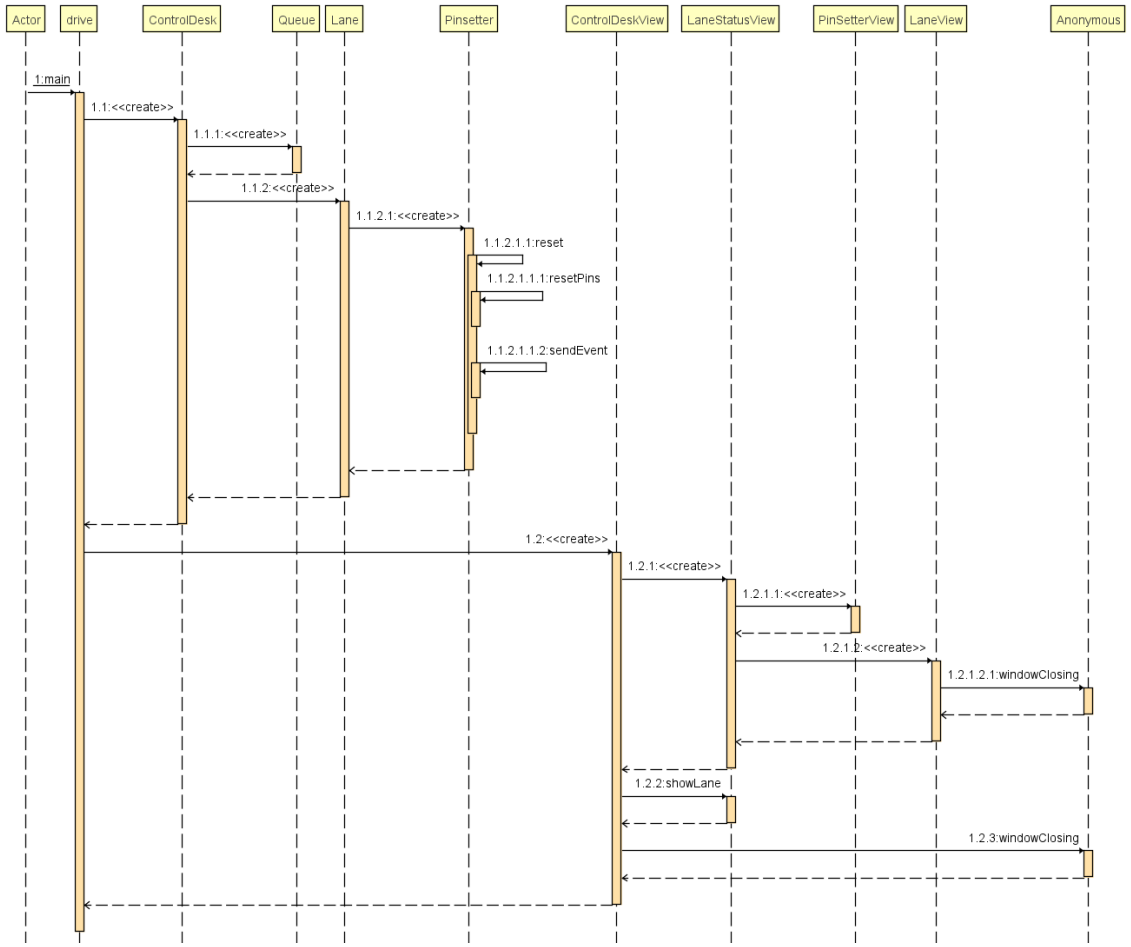


VIEW PACKAGE CLASSES

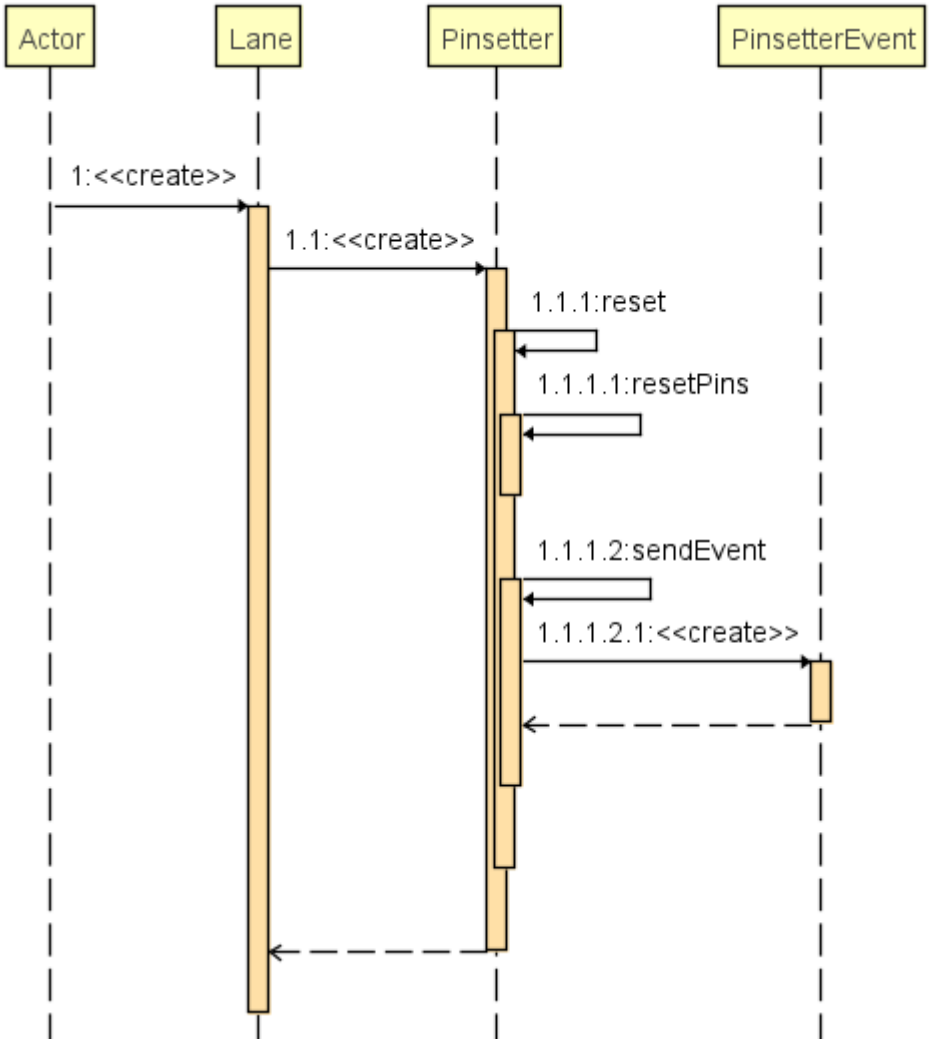


UML Sequence Diagram

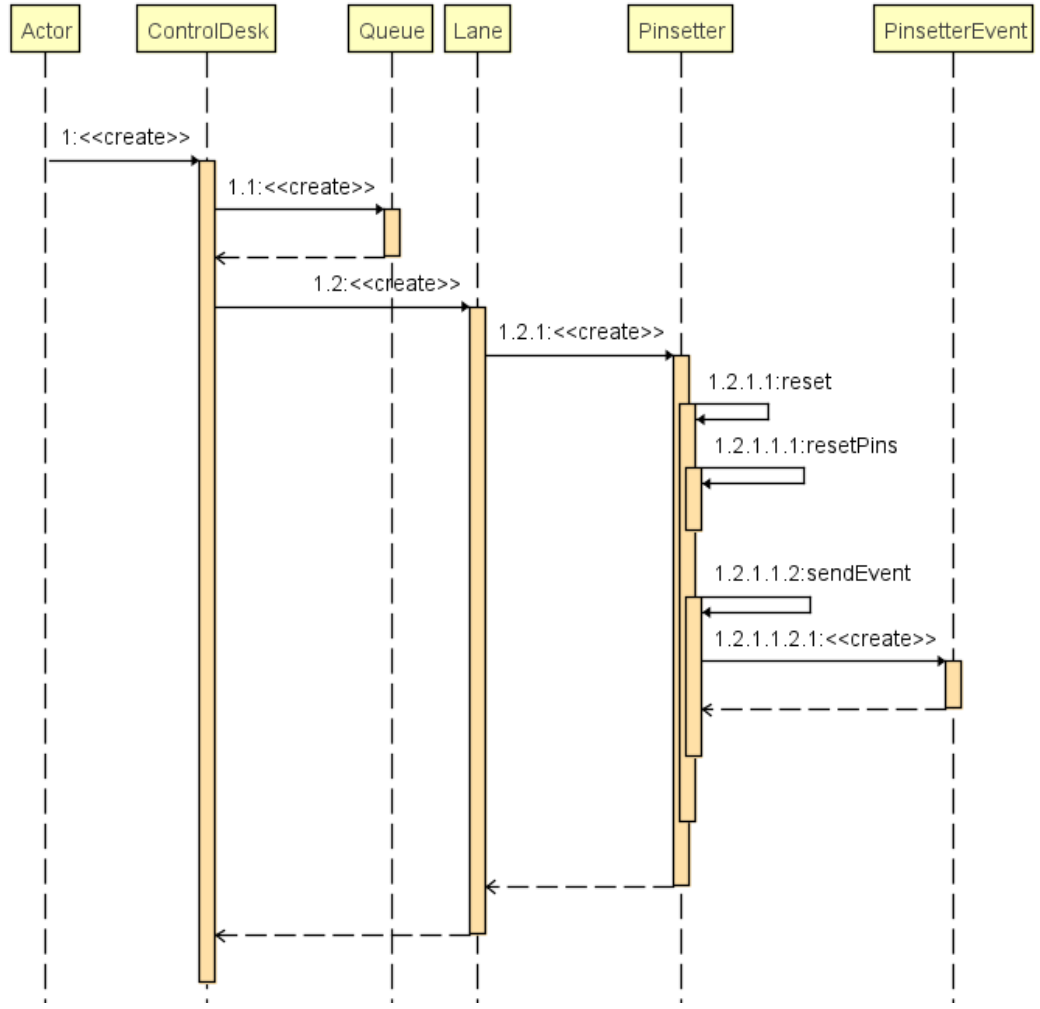
Drive Sequence Diagram:



Lane Sequence Diagram:



Control Desk



Class Responsibility Table After Refactoring

Classes added	Responsibility
ScoreCalculator	Calculates score of the current player
SearchDataBase	Returns various queries by fetching relevant information from BowlerFile
SearchView	Class created to display database query option

Analyzing Refactored Code

Strengths

1. Low Coupling:
    a. Low coupling refers to a relationship in which one module interacts with another module through a simple and stable interface and does not need to be concerned with the other module's internal implementation.
    b. Modules should be as independent as possible from other modules so that changes to module don't heavily impact other modules.
    c. By aiming for low coupling, you can easily make changes to the internals of modules without worrying about their impact on other modules in the system. Low coupling also makes it easier to design, write, and test code since our modules are not interdependent on each other.
    Changes made in Code:
    d. In order to decouple the Main files from the Views, we shifted the end-of-game code in Lane.java to LaneStatusView.java.
2. High Cohesion:
    a. Cohesion often refers to how the elements of a module belong together. Related code should be close to each other to make it highly cohesive.
    b. The elements within the module are directly related to the functionality that the module is meant to provide. By keeping high cohesion within our code, we end up trying DRY code and reduce duplication of knowledge in our modules. We can easily design, write, and test our code since the code for a module is all located together and works together.

c. Low cohesion would mean that the code that makes up some functionality is spread out all over your code-base. Not only is it hard to discover what code is related to your module, but it is also difficult to jump between different modules and keep track of all the code in your head.
Changes made in Code:
d. Replaced the extended Thread call in Lane to an implemented Runnable. Then in the constructor we created a new thread object, passing itself (as a Runnable object) as a parameter), and then started it. This effectively allows us to extend the Observable class in Lane.

3. Separation of Concerns:
    a. The separation of concerns (SoC) is one of the most fundamental principles in software development.
    b. The principle is simple: don't write your program as one solid block, instead, break up the code into chunks that are finalized tiny pieces of the system each able to complete a simple distinct job.
    Changes in Code:
    c. Moved the drive.java file outside the Main and Views package. Now the final refactored code is split into three main packages each of which addresses the three main issues:
        i. Main: Contains the main code of the bowling alley code.
        ii. Views: Contains the code related to the GUI of the project.
        iii. Drive.java: main file which is responsible to run the code and acts as a connection between Main and Views.
4. Information Hiding:
    a. Information hiding is the principle of segregation of the design decisions in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed.
    Changes made in Code:
    b. Used the concepts of Encapsulation and abstraction to achieve information hiding.
5. Law of Demeter
    a. The Law of Demeter principle reduces dependencies and helps build components that are loosely coupled for code reuse, easier maintenance, and testability.
    b. The Law of Demeter principle states that a module should not have the knowledge on the inner details of the objects it manipulates. In other words, a software component or an object

should not have the knowledge of the internal working of other objects or components.

Changes made in Code:

c. Separation of the Lane and Observer functionality classes.
d. We also removed the implements LaneObserver class in the Views Package. We replaced it with implements Observer.
e. Changed the logic for adding an Observer to Lane's list of observers to be in line with the java implementation in the appropriate GUI classes.

Design Patterns Used

1. Singleton Pattern:
    a. Ensure a class has only one instance, and provide a global point of access to it.
    b. This is useful when exactly one object is needed to coordinate actions across the system.

    Example in Code:

    c. The **drive.java** class is a class that is instantiated once and is responsible for initializing all objects and calling the corresponding methods.

2. Facade Pattern:
    a. Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to an existing system to hide its complexities.
    b. This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.

    Code Example:

    c. The whole code has been divided into three separate packages.

Metric Analysis

- Initial Measurements

The initial metrics as seen on Eclipse Metrics is as follows:

Git Staging | Metrics - code - Method Lines of Code (avg/max per method)

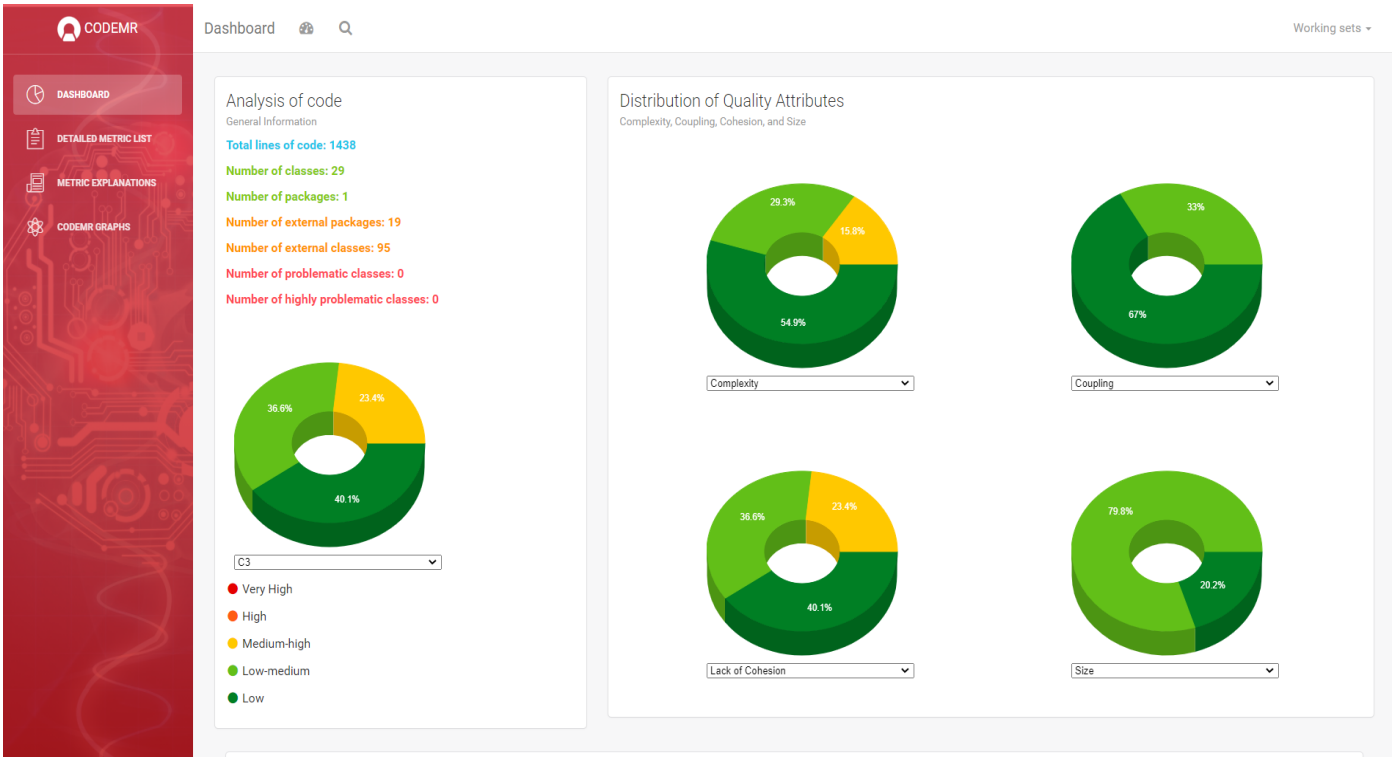
Metric	Total	Mean	Std. ...	Maxi...	Resource causing Maximum	Method
McCabe Cyclomatic Complexity (a		2.319	4.062	38	/code/Lane.java	getScore
Number of Parameters (avg/max p		0.723	1.131	9	/code/LaneEvent.java	LaneEvent
Nested Block Depth (avg/max per		1.511	1.177	7	/code/Lane.java	run
Afferent Coupling (avg/max per p		0	0	0	/code	
Efferent Coupling (avg/max per pa		0	0	0	/code	
Instability (avg/max per packageF		1	0	1	/code	
Abstractness (avg/max per packag		0.172	0	0.172	/code	
Normalized Distance (avg/max pe		0.172	0	0.172	/code	
Depth of Inheritance Tree (avg/ma		0.897	0.48	2	/code/ControlDesk.java	
Weighted methods per Class (avg/	327	11.2...	15.991	87	/code/Lane.java	
Number of Children (avg/max per	6	0.207	0.663	3	/code/PinsetterObserver.java	
Number of Overridden Methods (a	3	0.103	0.305	1	/code/Score.java	
Lack of Cohesion of Methods (avg		0.375	0.374	0.91	/code/LaneEvent.java	
Number of Attributes (avg/max pe	138	4.759	5.556	18	/code/Lane.java	
Number of Static Attributes (avg/r	2	0.069	0.253	1	/code/BowlerFile.java	
Number of Methods (avg/max per	133	4.586	3.765	17	/code/Lane.java	
Number of Static Methods (avg/m	8	0.276	0.69	3	/code/BowlerFile.java	
Specialization Index (avg/max per		0.017	0.052	0.2	/code/Score.java	
Number of Classes (avg/max per p	29	29	0	29	/code	
Number of Interfaces (avg/max pe	5	5	0	5	/code	
Number of Packages	1					
Total Lines of Code	1814					
Method Lines of Code (avg/max p	1284	9.106	17.201	88	/code/Lane.java	getScore

The following metrics were calculated by the plugin:

- McCabe Cyclomatic Complexity
    - Indicate the Complexity of the program.
    - A quantitative measure of the number of linearly independent paths through a program's source code.
    - Functions that violates the constraint are (along with their respective cyclomatic complexity):
        - Lane.java
            - getScore - 38
            - run - 19
            - receivePinsetterEvent - 12
        - LaneView.java
            - receiveLaneEvent - 19
        - LaneStatusView.java
            - actionPerformed - 11
        - AddPartyView.java
            - actionPerformed - 11

    - The highest Cyclomatic Complexity is shown by the getScore function in Lane.java, which was fixed by moving it to ScoreCalculator.java.
- Number of parameters
    - It should be kept low for a simpler understanding of code.
    - Functions that violated our constraints are:
        - LaneEvent.java
            - LaneEvent - 9
    - We decided to use the inbuilt observer in java, thus getting rid of this class altogether.
- Nested Block Depth
    - Functions that violated our constraints are (along with their respective nested block depth):
        - Lane.java:
            - run - 7
            - getScore - 7
        - We moved the getScore function thereby reducing nested block depth by a little bit.

Given below is the CodeMr analysis of the original code



List of all classes (#29)

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	Lane					227	medium-high	low-medium	medium-high	low-medium
2	ControlDeskView					87	low-medium	low-medium	low-medium	low-medium
3	ControlDesk					68	low-medium	low-medium	medium-high	low-medium
4	LaneStatusView					93	low	low-medium	low-medium	low-medium
5	LaneView					140	low-medium	low	low-medium	low-medium
6	AddPartyView					127	low-medium	low	low-medium	low-medium
7	PinSetterView					111	low	low	low	low-medium
8	NewPatronView					85	low	low	low	low-medium
9	EndGameReport					79	low	low	low-medium	low-medium
10	ScoreReport					76	low	low	low	low-medium
11	EndGamePrompt					55	low	low	low	low-medium
12	Pinsetter					47	low	low	low	low
13	LaneEvent					41	low	low	medium-high	low
14	BowlerFile					38	low	low	low	low
15	PinsetterEvent					26	low	low	low	low
15	PinsetterEvent					26	low	low	low	low
16	Bowler					25	low	low	low	low
17	PrintableText					21	low	low	low	low
18	ScoreHistoryFile					20	low	low	low	low
19	Score					16	low	low	low	low
20	Queue					12	low	low	low	low
21	LaneEventInterface					10	low	low	low	low
22	drive					8	low	low	low	low
23	Alley					6	low	low	low	low
24	ControlDeskEvent					6	low	low	low	low
25	Party					6	low	low	low	low
26	ControlDeskObserver					2	low	low	low	low
27	LaneObserver					2	low	low	low	low
28	LaneServer					2	low	low	low	low
29	PinsetterObserver					2	low	low	low	low

- Analysis of how to refactor based on the above measurements
    1. The analysis of each parameter and code smells has been done in the above sections.
    2. From our analysis of the metrics it was evident that we need to reduce the complexity, coupling and lack of cohesion in the files with large numbers of lines of code and in general large classes.
    3. We applied several design patterns and removed code smells as documented above in order to achieve the following final results.

- Final Measurements
  Given below are the final metrics using Eclipse:

Metrics - Refactored Code - McCabe Cyclomatic Complexity (avg/max per method)

Metric	Total	Mean	Std. ...	Maxi...	Resource causing Maximum	Method
McCabe Cyclomatic Complexity (avg/max per method)		2.35	2.988	20	/Refactored Code/Views/LaneView.java	update
Number of Parameters (avg/max per method)		0.756	1.062	5	/Refactored Code/Main/ScoreCalculator.ja...	calculateGame
Nested Block Depth (avg/max per method)		1.667	1.079	6	/Refactored Code/Main/Lane.java	run
Afferent Coupling (avg/max per packageFragment)		3	3.559	8	/Refactored Code/Main	
Efferent Coupling (avg/max per packageFragment)		2.667	3.091	7	/Refactored Code/Views	
Instability (avg/max per packageFragment)		0.625	0.445	1	/Refactored Code	
Abstractness (avg/max per packageFragment)		0	0	0	/Refactored Code	
Normalized Distance (avg/max per packageFragment)		0.375	0.445	1	/Refactored Code/Main	
Depth of Inheritance Tree (avg/max per type)		1.136	0.343	2	/Refactored Code/Main/ControlDesk.java	
Weighted methods per Class (avg/max per type)	289	13.1...	11.343	49	/Refactored Code/Main/Lane.java	
Number of Children (avg/max per type)	0	0	0	0	/Refactored Code/drive.java	
Number of Overridden Methods (avg/max per type)	1	0.045	0.208	1	/Refactored Code/Main/Score.java	
Lack of Cohesion of Methods (avg/max per type)		0.426	0.335	0.886	/Refactored Code/Main/Lane.java	
Number of Attributes (avg/max per type)	114	5.182	5.078	16	/Refactored Code/Main/Lane.java	
Number of Static Attributes (avg/max per type)	2	0.091	0.287	1	/Refactored Code/Main/ScoreHistoryFile.j...	
Number of Methods (avg/max per type)	117	5.318	4.733	24	/Refactored Code/Main/Lane.java	
Number of Static Methods (avg/max per type)	6	0.273	0.75	3	/Refactored Code/Main/BowlerFile.java	
Specialization Index (avg/max per type)		0.009	0.042	0.2	/Refactored Code/Main/Score.java	
Number of Classes (avg/max per packageFragment)	22	7.333	4.497	11	/Refactored Code/Main	
Number of Interfaces (avg/max per packageFragment)	0	0	0	0	/Refactored Code	
Number of Packages	3					
Total Lines of Code	1713					
Method Lines of Code (avg/max per method)	1201	9.764	16.247	86	/Refactored Code/Views/PinSetterView.java	PinSetterView

We can see from the above table that Cyclomatic Complexity, Number of Parameters per method, and Nested Block Depth have improved significantly.

Given below is the CodeMr analysis for the refactored Code

Dashboard

Working sets

Analysis of Refactored Code

General Information

Total lines of code: 1339

Number of classes: 22

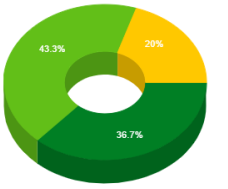
Number of packages: 3

Number of external packages: 14

Number of external classes: 94

Number of problematic classes: 0

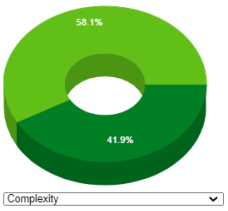
Number of highly problematic classes: 0

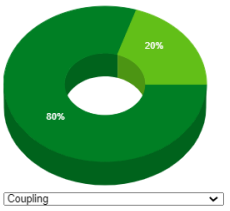


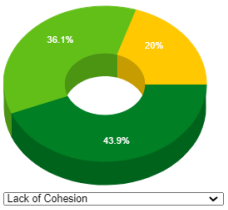
- Very High
- High
- Medium-high
- Low-medium
- Low

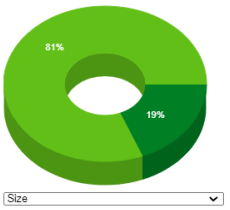
Distribution of Quality Attributes

Complexity, Coupling, Cohesion, and Size



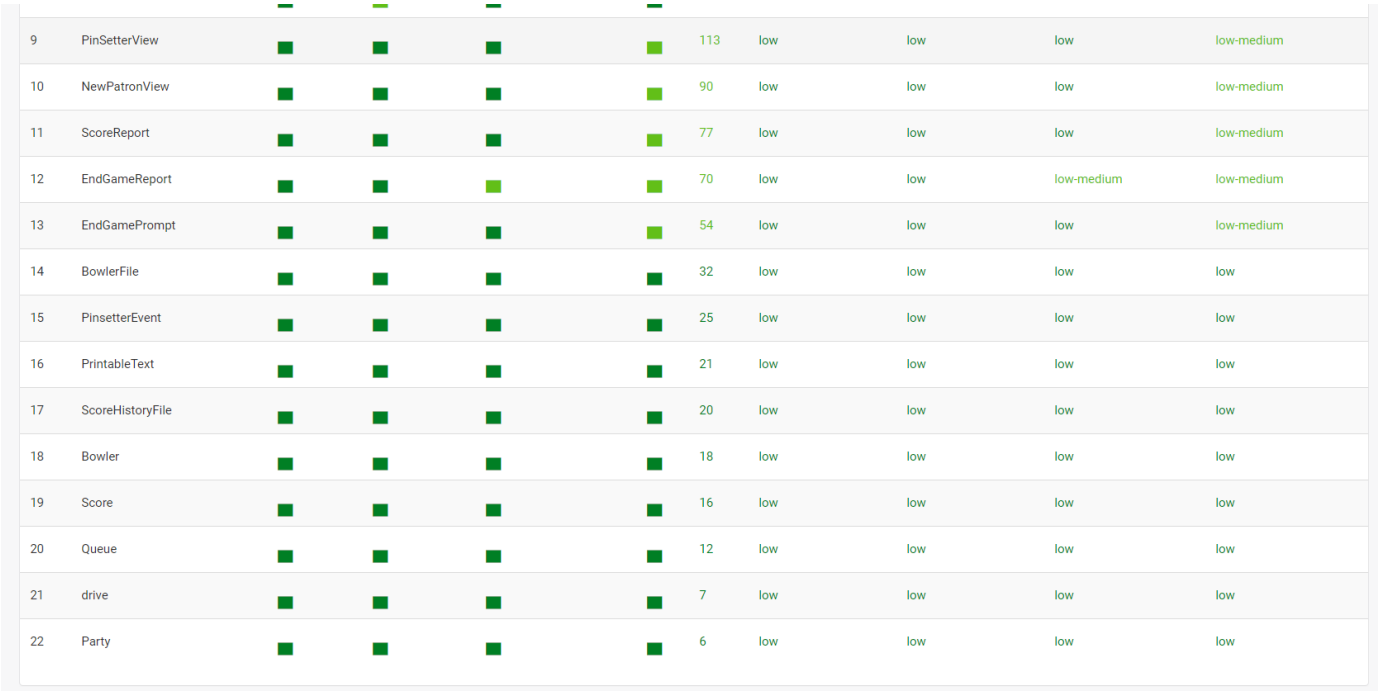






List of all classes (#22)

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	Lane					141	low-medium	low-medium	medium-high	low-medium
2	LaneStatusView					127	low-medium	low-medium	low-medium	low-medium
3	AddPartyView					128	low-medium	low	low-medium	low-medium
4	LaneView					127	low-medium	low	medium-high	low-medium
5	ControlDeskView					97	low-medium	low	low-medium	low-medium
6	ControlDesk					61	low-medium	low	low-medium	low-medium
7	Pinsetter					50	low-medium	low	low	low
8	ScoreCalculator					47	low-medium	low	low	low
9	PinSetterView					113	low	low	low	low-medium
10	NewPatronView					90	low	low	low	low-medium
11	ScoreReport					77	low	low	low	low-medium
12	EndGameReport					70	low	low	low-medium	low-medium
13	EndGamePrompt					54	low	low	low	low-medium
14	BowlerFile					32	low	low	low	low
15	PinsetterEvent					25	low	low	low	low



9	PinSetterView					113	low	low	low	low-medium
10	NewPatronView					90	low	low	low	low-medium
11	ScoreReport					77	low	low	low	low-medium
12	EndGameReport					70	low	low	low-medium	low-medium
13	EndGamePrompt					54	low	low	low	low-medium
14	BowlerFile					32	low	low	low	low
15	PinsetterEvent					25	low	low	low	low
16	PrintableText					21	low	low	low	low
17	ScoreHistoryFile					20	low	low	low	low
18	Bowler					18	low	low	low	low
19	Score					16	low	low	low	low
20	Queue					12	low	low	low	low
21	drive					7	low	low	low	low
22	Party					6	low	low	low	low

As can be seen in the final measurements, all greens have been obtained for all the files. Here, the coupling and lack of cohesion in the files with large numbers of lines of code and in general large classes has been reduced to low in every file.

It can be seen that the code base has significantly improved in terms of code quality due to the refactoring. This has been achieved by creating a balance among competing criteria such as low coupling and high cohesion so as to develop efficient classes with no dead or duplicate code.