

Design Document

Unit 2 project

Bowling Alley Simulation Enhancement

TEAM 30

Date Of Submission : 20-March-2021

Table of Contents :

Team Information	4
Overview	4
UML Class Diagram	5
UML Before Refactoring	5
UML After Refactoring	9
Sequence Diagram Before Refactoring	12
Sequence Diagram After Refactoring	13
Responsibilities Of Classes	13
Analysis of Original Design	14
Cons	14
Pros	15
Fidelity to the Design Document	15
Design Patterns	15
Analysis of Refactored Design	16
Responsibilities Of Newly created Classes	16
How we achieved :	16
Design Patterns Used	17
Metric Analysis	18
Metric of Original Design	18
Metrics of Refactored Design	21
Discussion of metric	23
New Requirements	27
Stopping Simulation:	27
Design Pattern Used:	27
UI Pattern Used:	27
Penalty For Gutters:	28
Emoticons:	29
Tie Breaker	31

Team Information

Name (Roll)	Work hours	Roles (includes but not limited to)
Dipankar Saha (2020201084)	12	Provide a searchable view to make ad-hoc queries on the stored data. Highest / Lowest scores, Top Player, etc
Satyam Kumar Singh (2020201035)	12	Implement an emoticon for appreciate, envy, embarrass and other possible actions based on player score.
Sai Vishwak Gangam (2020202006)	12	Implement penalty for Gutters - On bowling 2 consecutive gutters or If the first 2 are at the start of the game, the player should be penalized.
Swaraj Renghe (20171119)	11	At the end of 10 frames, provide a chance to the 2nd highest player to bowl. If the player becomes the highest, continue with 3 additional frames to declare winner.

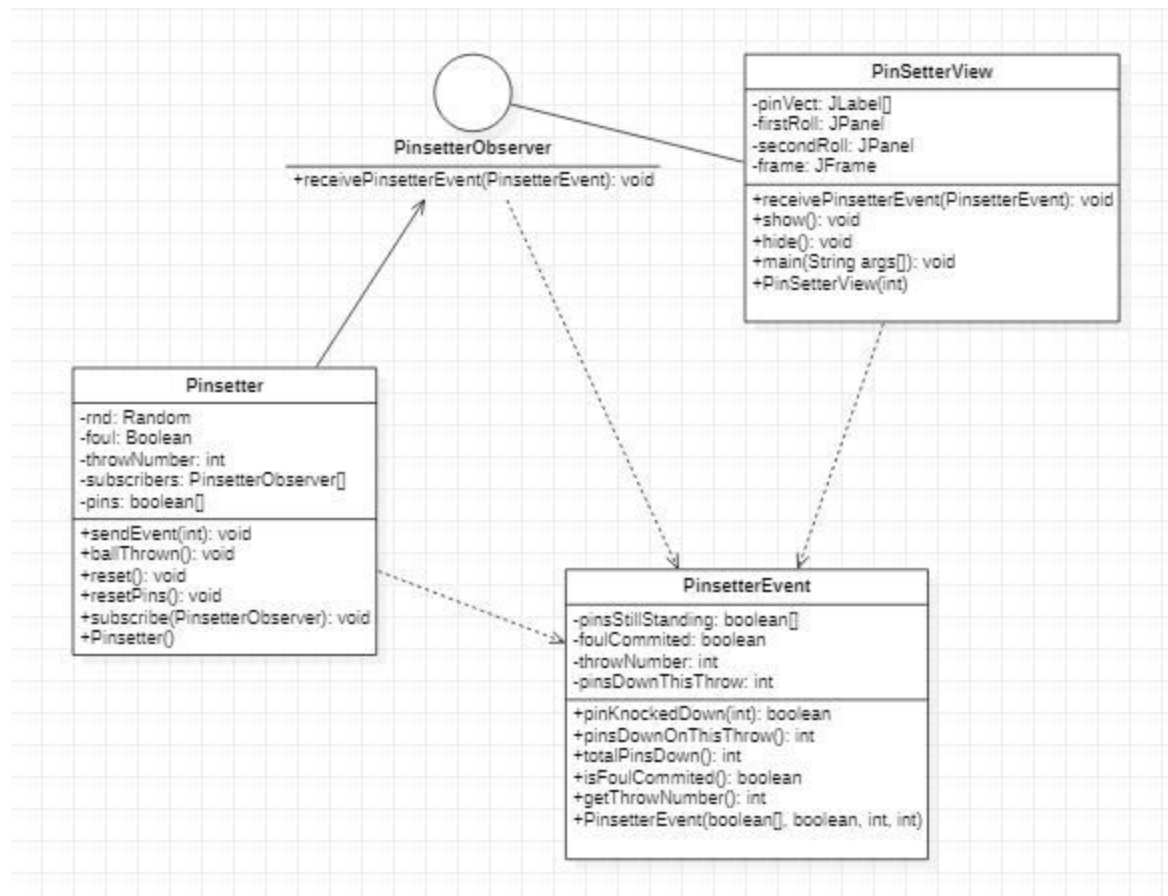
Overview

In this Document we present the Refactored Version of the original product (Bowling Management System) to automate the chain of bowling establishments located across the country. This original product had several designs and code reflecting poor software engineering design principles and some anti-patterns. We have made a fair attempt to recognize and solve those problems. Our refactored product caters to the requirement of installing new pin setting equipment which can detect the numbers 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. It also establishes a service for customers that would maintain a history of a bowler's scores, average and other related information.

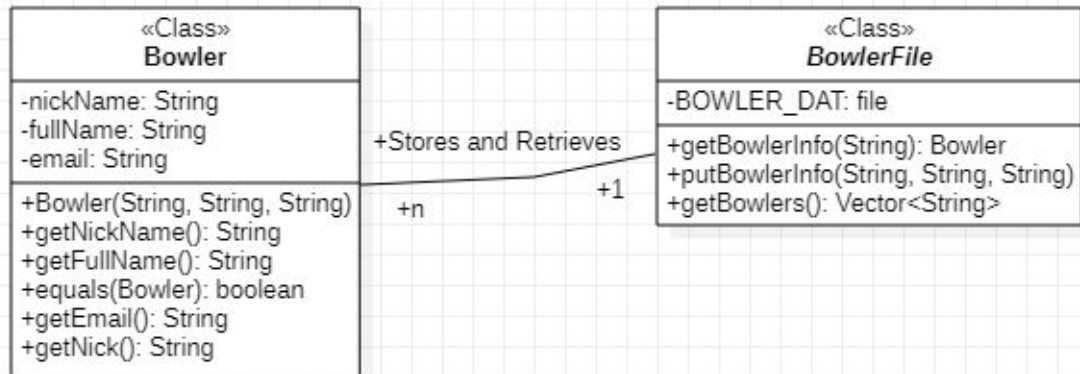
UML Class Diagram

UML Before Refactoring

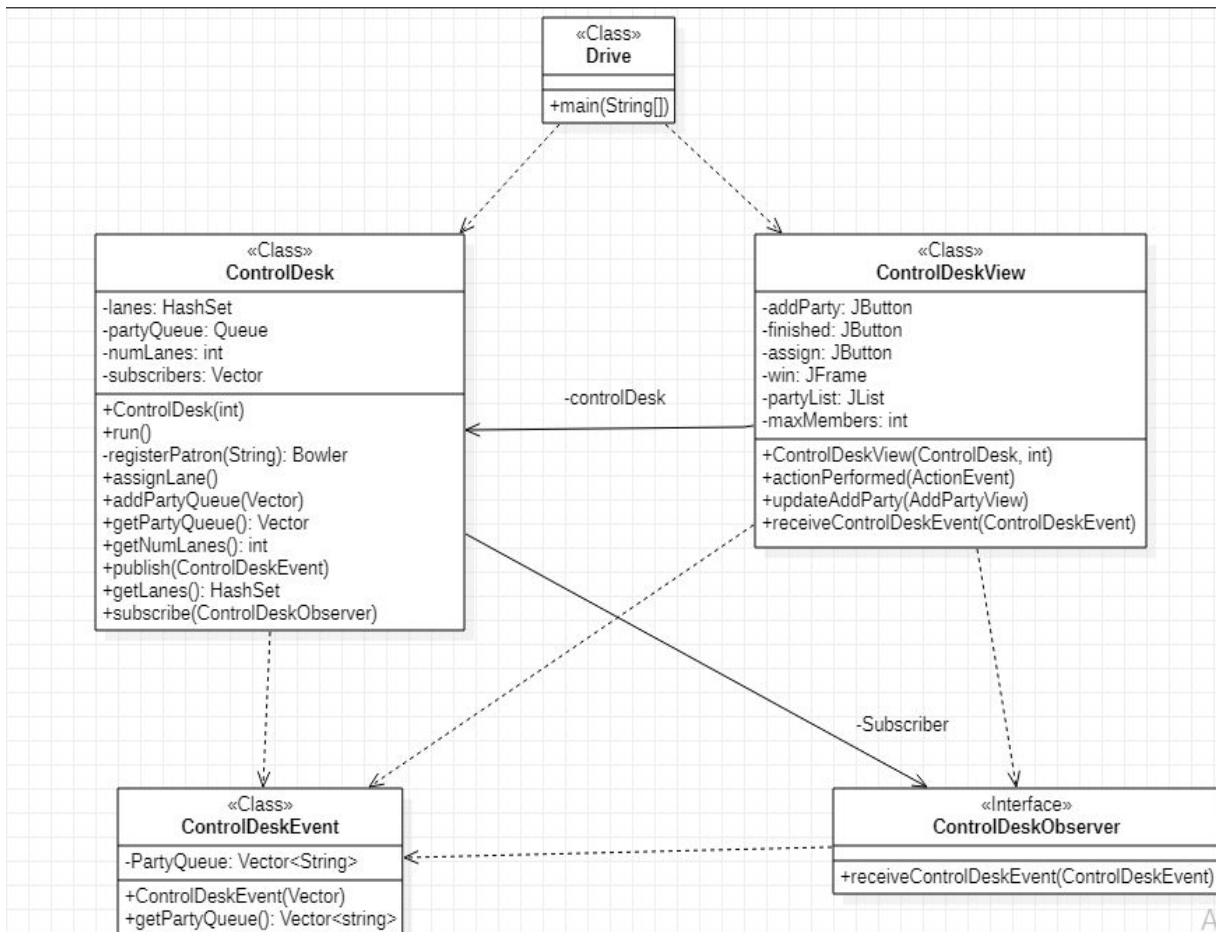
Pinsetter Classes



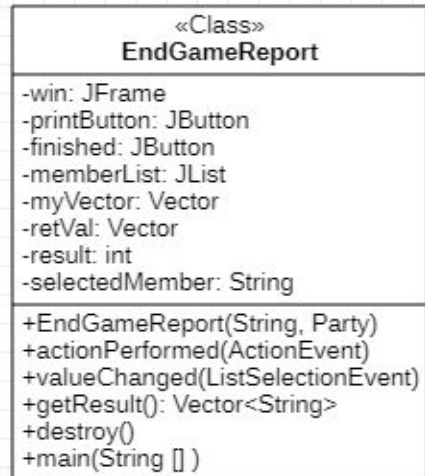
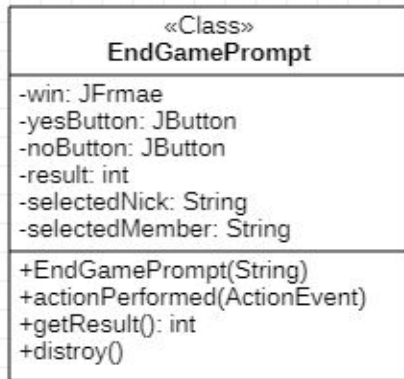
Bowler Classes



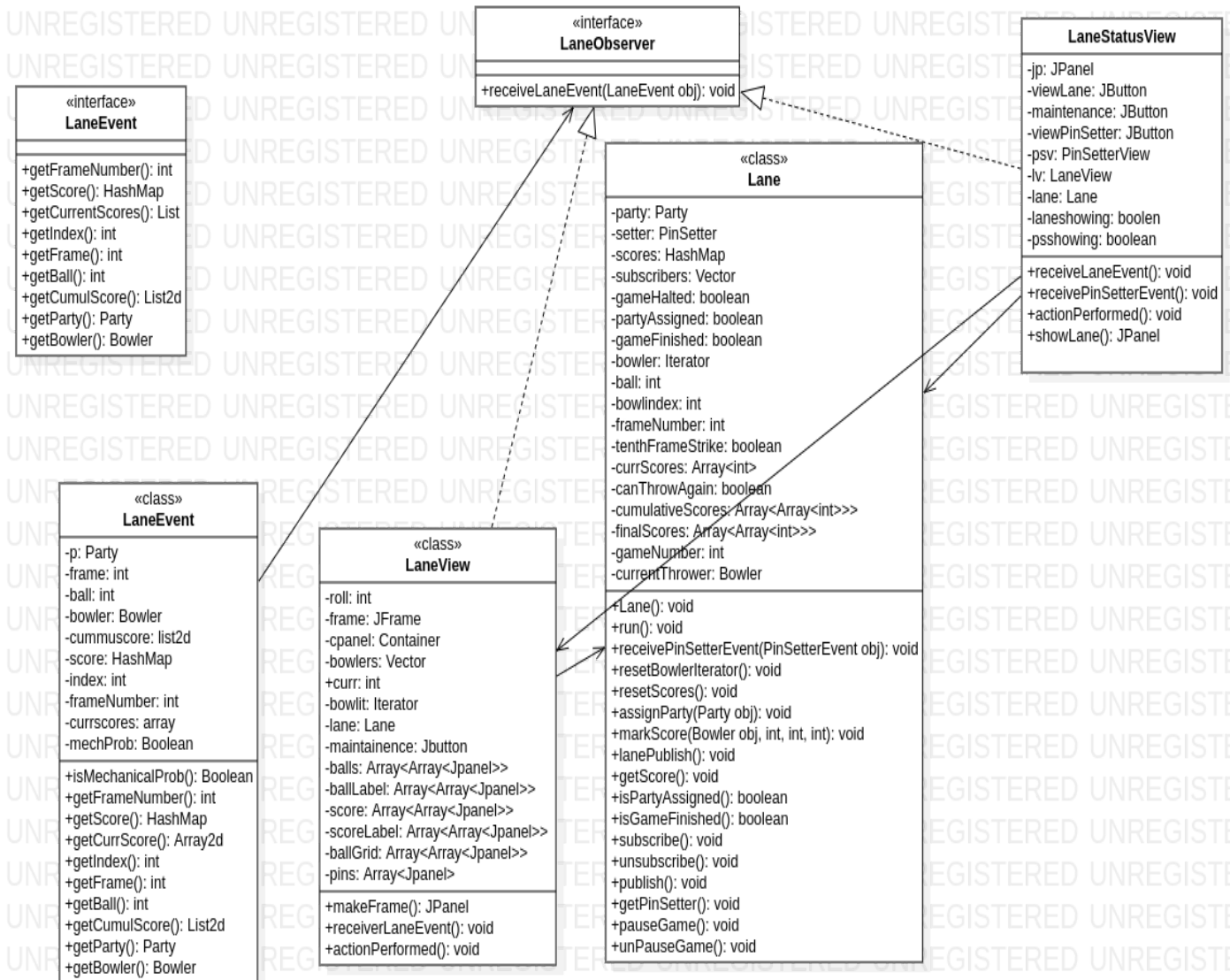
ControlDesk Classes



EndGame Classes

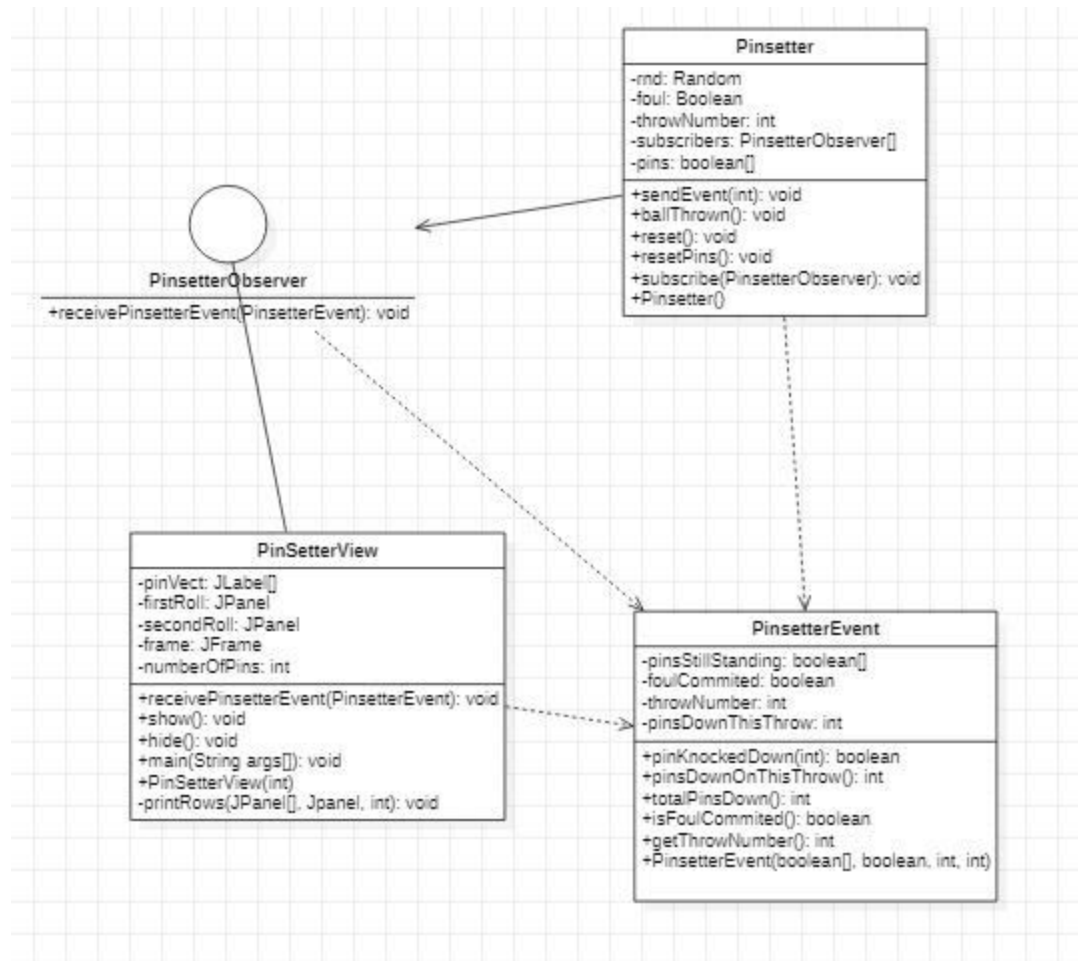


Lane Classes

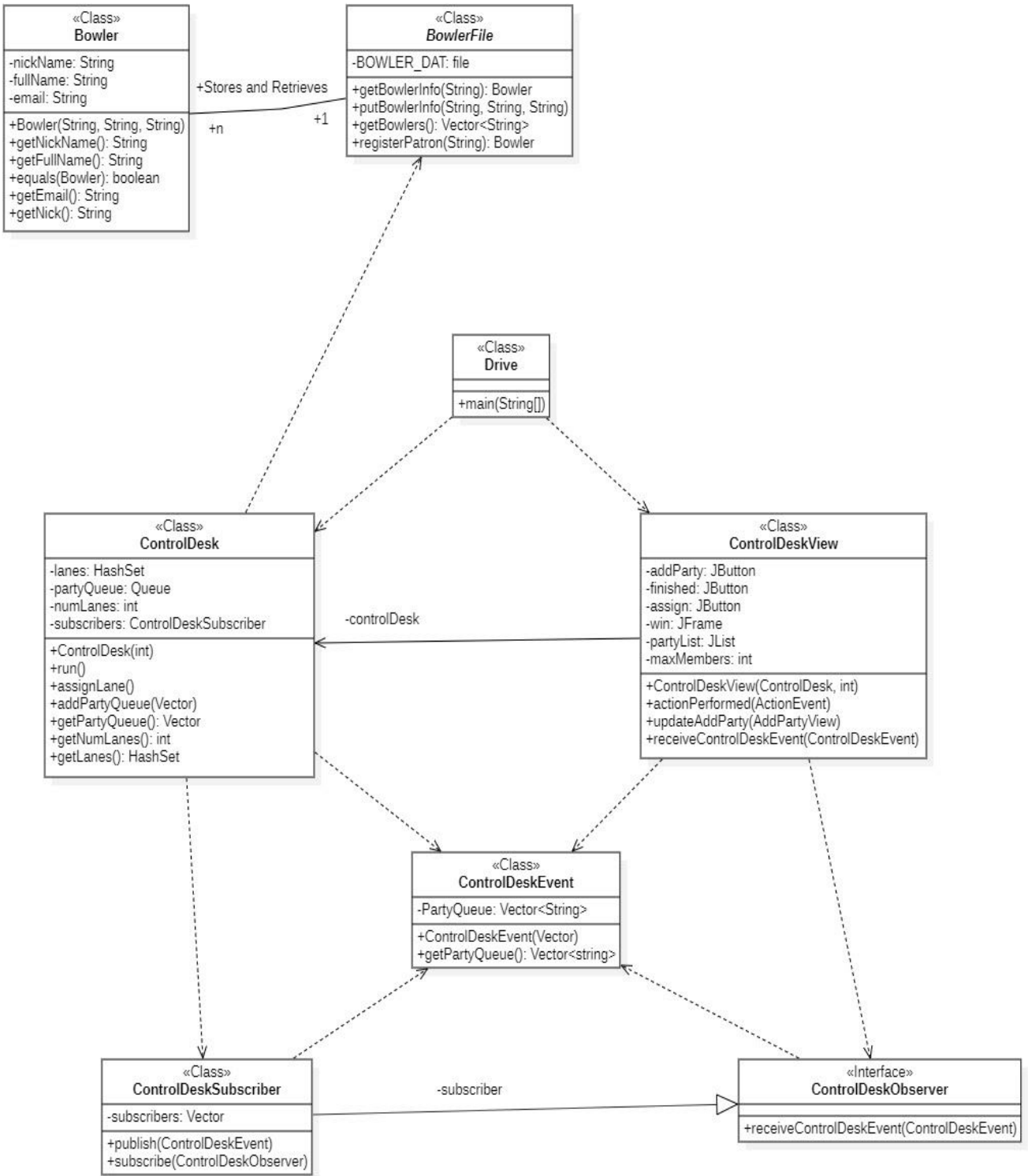


UML After Refactoring

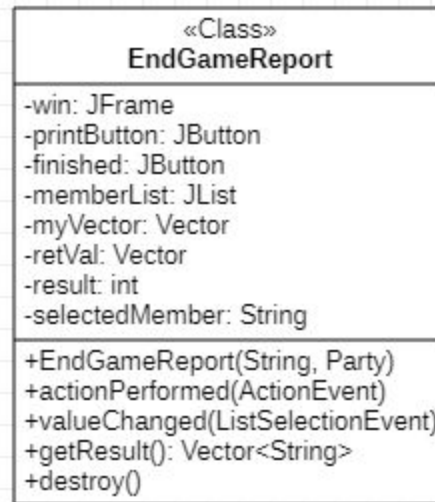
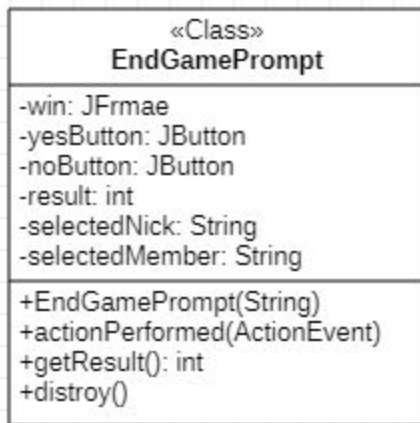
Pinsetter Classes



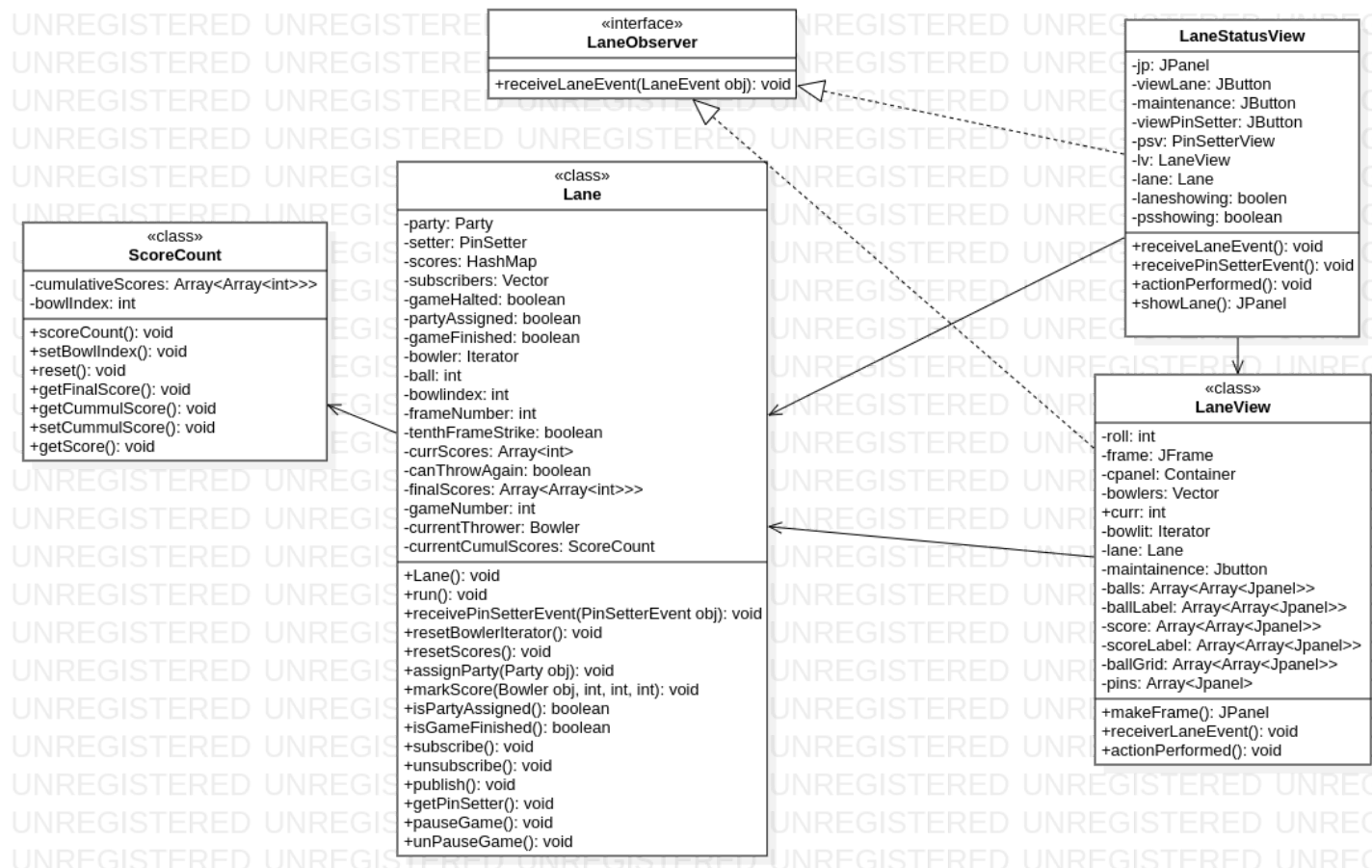
Bowler and ControlDesk classes



EndGame Classes

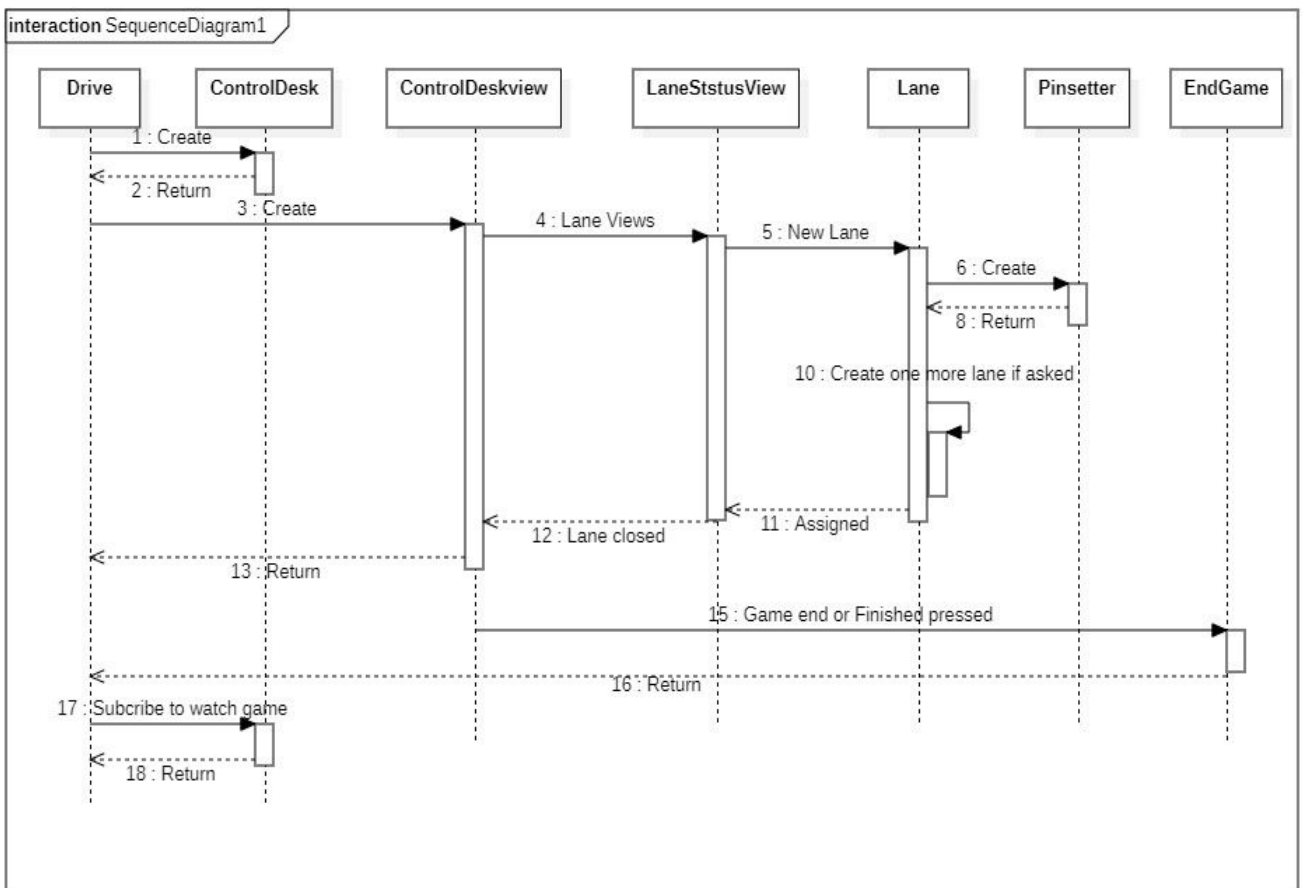


Lane Classes

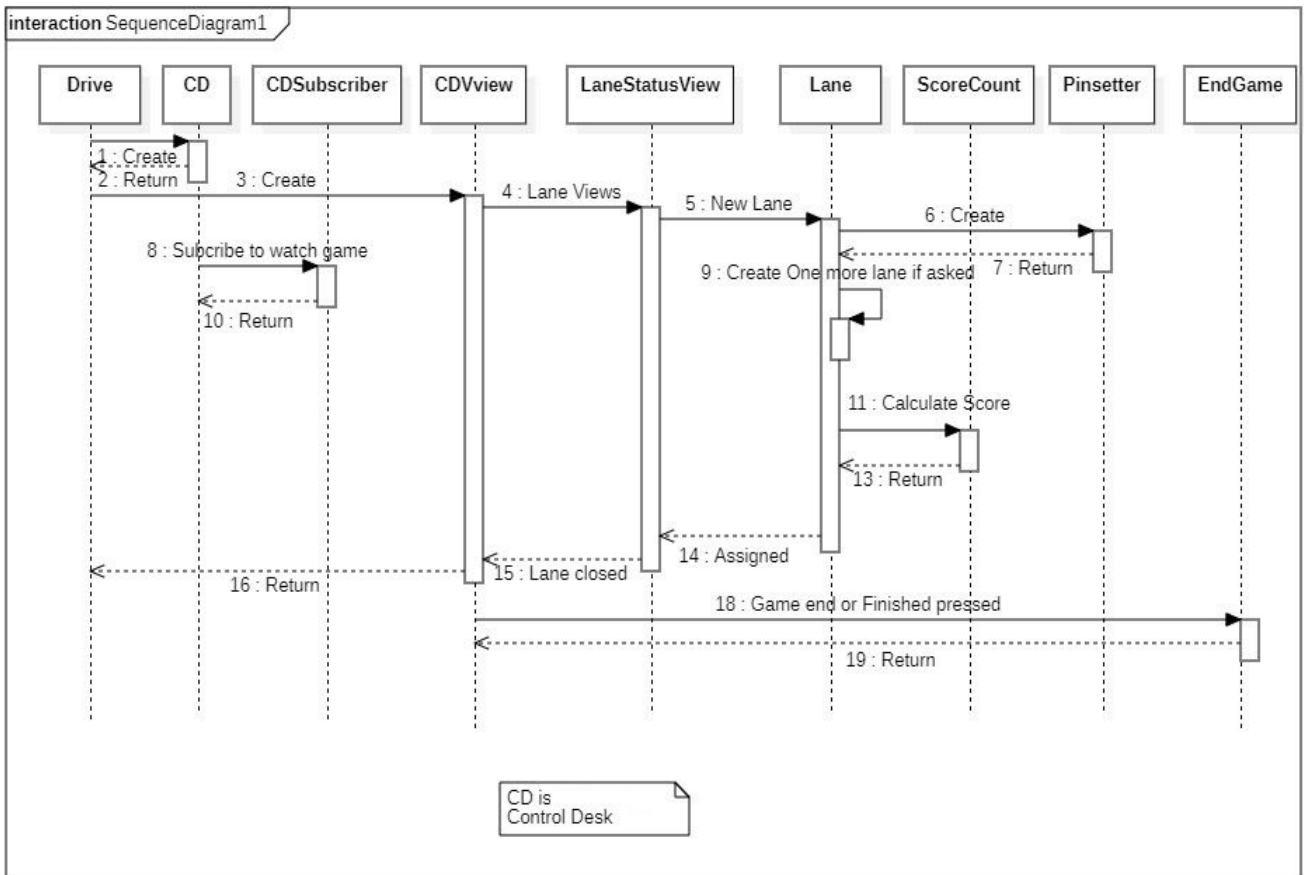


Sequence Diagram

Sequence Diagram Before Refactoring



Sequence Diagram After Refactoring



Responsibilities Of Classes

CLASS NAME	MAJOR RESPONSIBILITY
AddPartyView	GUI components need to add a party
Bowler	Holds all bowler information
BowlerFile	Interface to the Bowler database
ControlDesk	Initialize the main control desk and assigns Lanes to Parties
ControlDeskView	GUI components to represent of the control desk

Drive	Initialize the Game and create alley
Lane	Initialize lane, Simulate bowling game
LaneStatusView	GUI to Displays Lane Status, Lane's Pinsetter
Party	Container that holds bowlers
PinSetter	Represent the pinsetter that simulates dropping of pins on each throw
PinSetterView	GUI to show the status of pins after each throw
ScoreCount	Calculate scores got different ball types (normal/strike)
ScoreReport	Generate score report buffer, and send to user
ScoreHistoryFile	Manages Score database

Analysis of Original Design

Cons

1. **Dead Code** : The original design had a lot of code that was either commented out or not being used anywhere in the system. Those unwanted items were removed wherever necessary Including variables, methods in classes like : PinSetterView, LaneEvent. There were also multiple main methods in the system, which weren't being called/used anywhere.
2. **Single Large Method** : Most of the classes had very long methods trying to do too much, So we tried modularizing the code further in Classes like : LaneView broken into ScoreCount, PinSetterView, AddPartyView etc.
3. **Duplicate Code** : Similar kind of job was being done in some methods by simply copy-pasting the previous code. We reuse the code through creation of methods.

4. **Data Hiding** : Classes like the LaneEvent had its attributes not hidden from other classes using its objects, So we made it private and provided appropriate getters & setters.
5. **Use of Old/Deprecated Tools** : Functions like show(), hide() etc. and use of AWT instead of newer Swing, and avoiding Generics, etc. was observed.

Pros

1. **Proper Comments** : The code was well commented, and purpose and functionality of most of the part of the system was provided.
2. **Low Coupling** : The overall system as well as the subsystems were having low coupling metric.

Fidelity to the Design Document

The original codebase mostly fulfilled the design document requirements attached with the code. Except for the "Print Report" functionality presented at the end of a game, which was not working.

Design Patterns

- **Observer Pattern** : The event handling done in the system on button click is a good example of observer pattern. Here we wait on thread for an event like a button click by the user, and notify the corresponding event-handler which carries out a task corresponding to the given button click.
- **Adapter Pattern** : As we know that Adapter pattern is a structural design pattern that works as a bridge between two incompatible interfaces. So in the given system the ControlDesk Class acts as an Adapter. It joins Bowlers, Party and Queue subsystems.
- **Singleton Pattern** : A software design pattern that restricts the instantiation of a class to one "single" instance. This is clearly observed in the drive class, which acts as the main function in this program, and is instantiated only once in its lifetime.

Analysis of Refactored Design

Responsibilities Of Newly created Classes

CLASS NAME	MAJOR RESPONSIBILITY
ControlDeskSubscriber	Maintain the Subscribers
ScoreCount	Calculate the score for every throw

How we achieved :

- **Low Coupling :**

The dependencies between the classes were moderate and we have tried to make it low by passing the parameters locally and removing the redundant ones wherever possible.

We have extended our class list to break down large files such as Lane into different subclasses. We have made sure that these classes were mostly independent and did not require too many other dependencies to increase the coupling.

- **High Cohesion**

Cohesion tells about the consistency and organization of different units. The more tasks a single class tries to perform, we have a problem with cohesion there. The long classes had numerous methods which often became unrelated and too broad. We split such classes eg. ControlDesk to two Classes(ControlDesk and ControlDeskSubscriber) to divide the task and achieve cohesion.

- **Separation of Concerns**

Separation of Concerns is a design principle for separating a system into distinct sections such that each section addresses a separate concern. An example of how we achieved in the refactored design is by creating a separate score calculating class. Previously Lane Class had a method `getScore()` which calculates the score but we have created a separate class `ScoreCount` for calculating the score and the updated score is sent to Lane Class to mark.

- **Dead Code Elimination**

An optimization that removes code which does not affect the program results. We removed the multiple main methods in the system that weren't being called/used anywhere else. We also eliminated extra classes like, LaneEvent and PinsetterEvent. The implementations of these classes were unnecessary, and their functioning could be handled by existing classes without taking a hit on code complexity or cohesiveness.

- **Information Hiding**

Data Abstraction or Information hiding is a crucial aspect of OOPS. Classes like the LaneEvent had its attributes exposed to the other classes through its objects, So we made it hidden by making it private. Appropriate getters & setters were provided.

- **The Law of Demeter**

The fundamental notion of the LoD is that a given object should assume as little as possible about the structure or properties of anything else (including its subcomponents). That we achieved by ensuring low coupling and high cohesion. We have tried to make the classes as independent as possible so they have very few neighbours.

- **Extensibility**

As we ensured low coupling, we made sure that it was easier to introduce new module. As we wrote basic code for the UI, we could easily extend it to add specific features in the respective classes.

- **Reusability**

To ensure the code reusability several methods were written. Wherever in the original code we found that a similar kind of task was done by copy-pasting we modularized the code.

Design Patterns Used

- **Observer Pattern**

The three observer classes present in the project are ControlDeskObserver, LaneObserver and PinSetterObserver any change in the event will be notified to all the subscribers of above classes and the respective action is done.

- **Singleton Class**

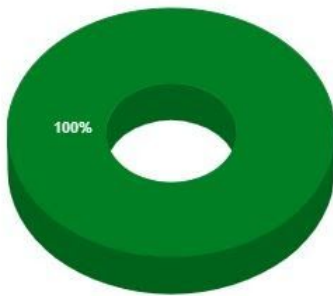
This is a software design pattern that restricts the instantiation of a class to one single instance. In our design for example, the drive class is the main function of this game and is instantiated only once in a lifetime.

Metric Analysis

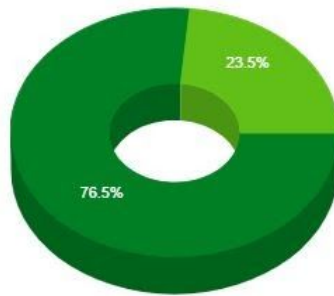
Metric of Original Design

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

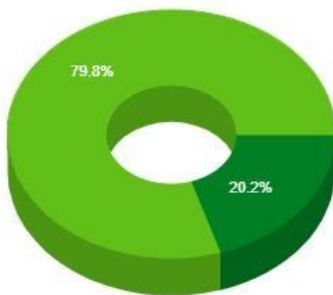
14 Bowler	■	■	■	■	30	low	low	low	low
15 PinsetterEvent	■	■	■	■	26	low	low	low	low
16 Bowler	■	■	■	■	25	low	low	low	low
17 PrintableText	■	■	■	■	21	low	low	low	low
18 ScoreHistoryFile	■	■	■	■	20	low	low	low	low
19 Score	■	■	■	■	16	low	low	low	low
20 Queue	■	■	■	■	12	low	low	low	low
21 LaneEventInterface	■	■	■	■	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 PinsetterObserver	■	■	■	■	2	low	low	low	low
27 ControlDeskObserver	■	■	■	■	2	low	low	low	low
28 LaneServer	■	■	■	■	2	low	low	low	low
29 LaneObserver	■	■	■	■	2	low	low	low	low



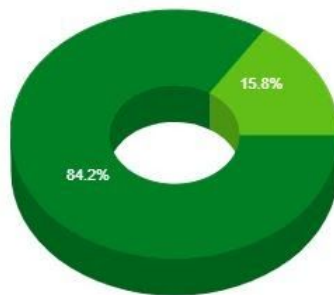
Number of Methods ▼



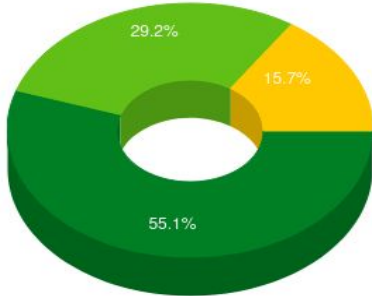
Class-Methods Lines of Code ▼



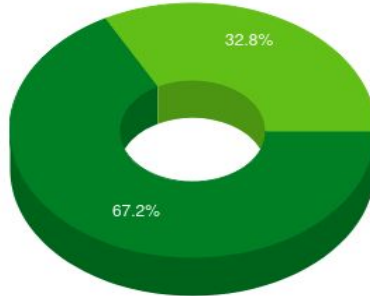
Class Lines of Code ▼



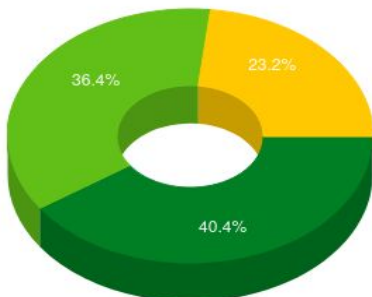
C3 ▼



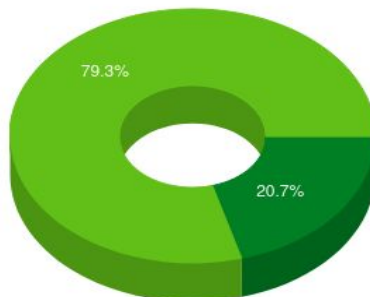
Complexity



Coupling



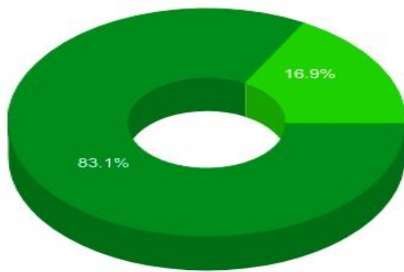
Lack of Cohesion



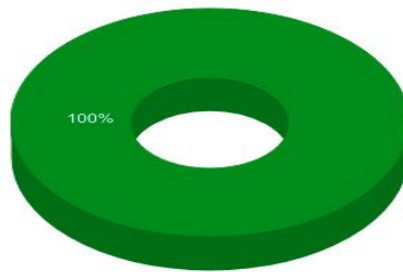
Size

Metrics of Refactored Design

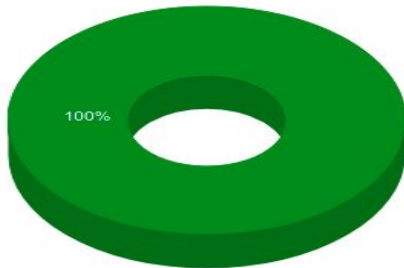
List of all classes (#29)										
ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	Lane	■	■	■	■	167	low-medium	low	low	low-medium
2	ScoreCount	■	■	■	■	60	low-medium	low	low	low-medium
3	LaneView	■	■	■	■	140	low	low	low	low-medium
4	AddPartyView	■	■	■	■	130	low	low	low	low-medium
5	LaneStatusView	■	■	■	■	93	low	low	low	low-medium
6	ControlDeskView	■	■	■	■	87	low	low	low	low-medium
7	NewPatronView	■	■	■	■	85	low	low	low	low-medium
8	ScoreReport	■	■	■	■	76	low	low	low	low-medium
9	PinSetterView	■	■	■	■	74	low	low	low	low-medium
10	EndGameReport	■	■	■	■	72	low	low	low	low-medium
11	EndGamePrompt	■	■	■	■	55	low	low	low	low-medium
12	ControlDesk	■	■	■	■	48	low	low	low	low
13	Pinsetter	■	■	■	■	47	low	low	low	low
14	BowlerFile	■	■	■	■	47	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	ControlDeskSubscr...	■	■	■	■	9	low	low	low	low
22	drive	■	■	■	■	7	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	PinsetterObserver	■	■	■	■	2	low	low	low	low
27	ControlDeskObserver	■	■	■	■	2	low	low	low	low
28	LaneServer	■	■	■	■	2	low	low	low	low



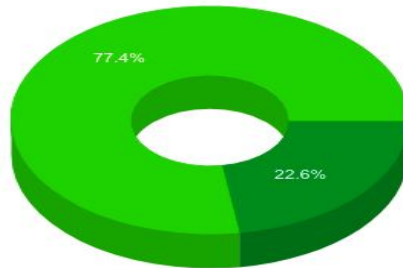
Complexity



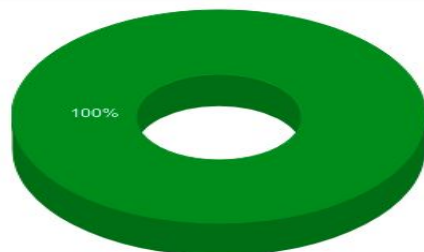
Coupling



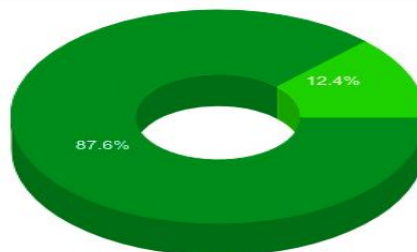
Lack of Cohesion



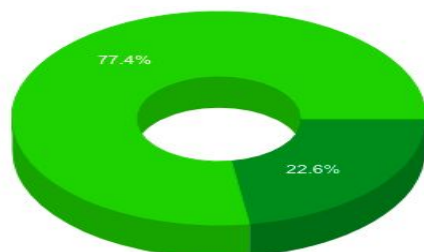
Size



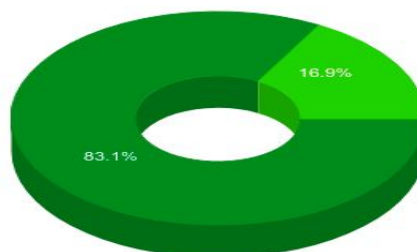
Number of Methods



Class-Methods Lines of Code



Class Lines of Code



C3

Discussion of metric

1. *What were the metrics for the code base? What did these initial measurements tell you about the system?*

There were several metrics for the code base used for analysis of the original and the refactored code like : Coupling, Cohesion, Lines Of Code, Cyclomatic Complexity, Modularity, Data Hiding, Size of classes & methods, extensibility, reusability etc. as shown in the graphs and tables [here](#).

These measurements highlighted several aspects of the original system that we have discussed in detail in the analysis section of the original design [here](#).

2. *How did you use these measurements to guide your refactoring?*

The insights that we gathered from these measurement helped to to refactor the original design in an organized and well targeted manner. The detailed discussion of how these measurement helped us to identify the anti-patterns etc. and our approach to solve these problems to an acceptable extent is [here](#).

3. *How did your refactoring affect the metrics? Did your refactoring improve the metrics? In all areas? In some areas? What contributed to these results?*

As observed by looking at the metric measurements shown [here](#). We can conclude that we have decreased the complexity and cohesion of Lane Class by deleting the Redundant classes like LaneEventInterface and LaneEvent. We have also created Separate class for score calculation which helped us to increase the cohesiveness of Lane class and the newly created class. The same way we have created a separate class for subscribers and improved the cohesiveness of ControlDesk class. Data encapsulation has been increased by making the attribute private to LaneEvent class. Apart from this we have tried to improve other areas of overall code by various means which can be found [here](#).

New Requirements

Max Player:

This was a simple change of the 'maxPatrons' variable to 6. This allows a maximum of 6 players to play the game. The UML diagrams hence remain the same.

Add and Store Player:

This was already there in the code as an Add Patron option in the AddParty window. It will simply Add and store a Player details in the database.

Ad-hoc Queries:

Queries such as Top Score, Top Scorer, Player Max score , Player Min score , Last scores of Players have been implemented using a new class ShowScores.java. ShowScore button has been created in the ControlDeskView window screen and by clicking on that a new window will pop up where various queries will come along with a player list.

- **Design Pattern Used:**

The **Decorator Pattern** has been used while implementing this requirement. Decorator pattern attaches additional responsibilities to an object dynamically and provides a flexible alternative to subclassing for extending functionality. In this case, score is the object that we are decorating everytime to get the desired output as a query result. Everytime a game is finished, corresponding query results are updated dynamically.

- **UI Pattern Used:**

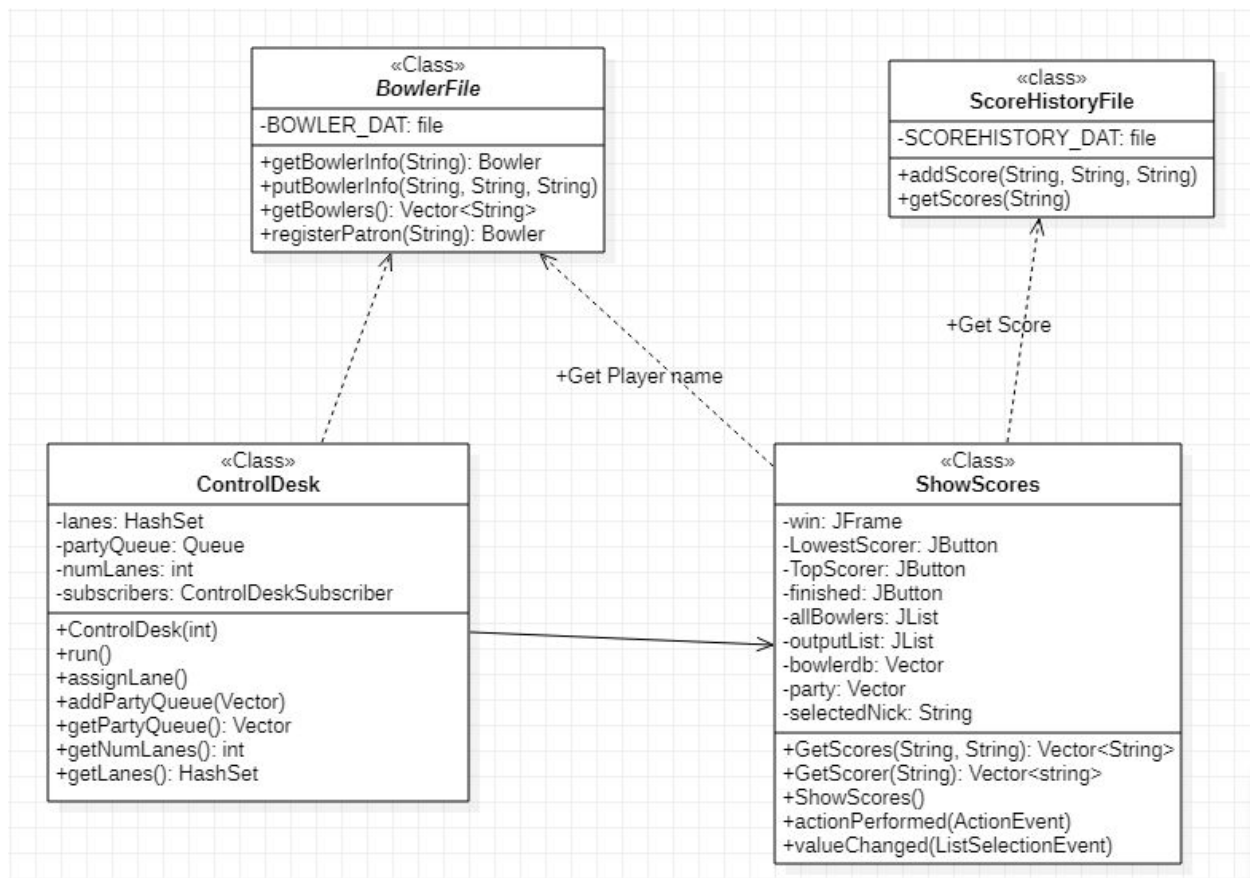
Breadcrumbs pattern has been used while designing the UI part of this requirement. As part of this where the query window will pop up for better visibility whenever requested and the user will quickly run some query to get the desired information. Also **Clear Primary Actions** pattern has been used to clearly and separately give out the query that a user could possibly run.



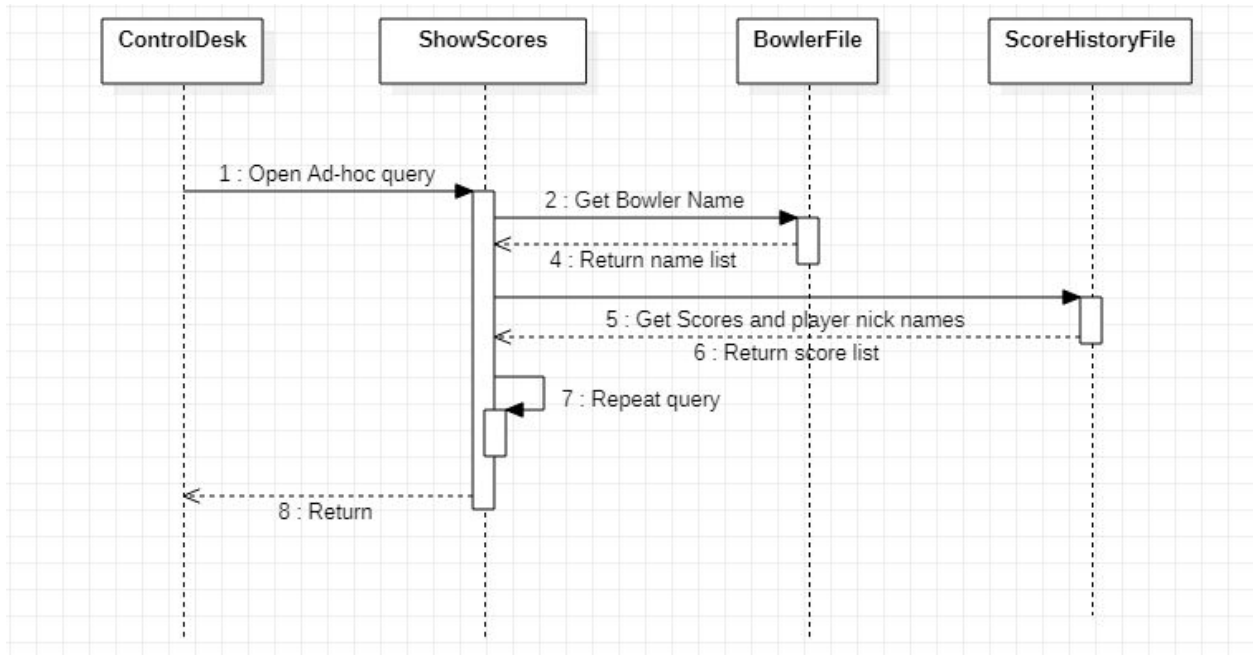
- New Class Description:

Class Name	Description
ShowScores	Returns result for various queries like Top score, Lowest score, last score.

- Class diagram



- Sequence Diagram:



Stopping Simulation:

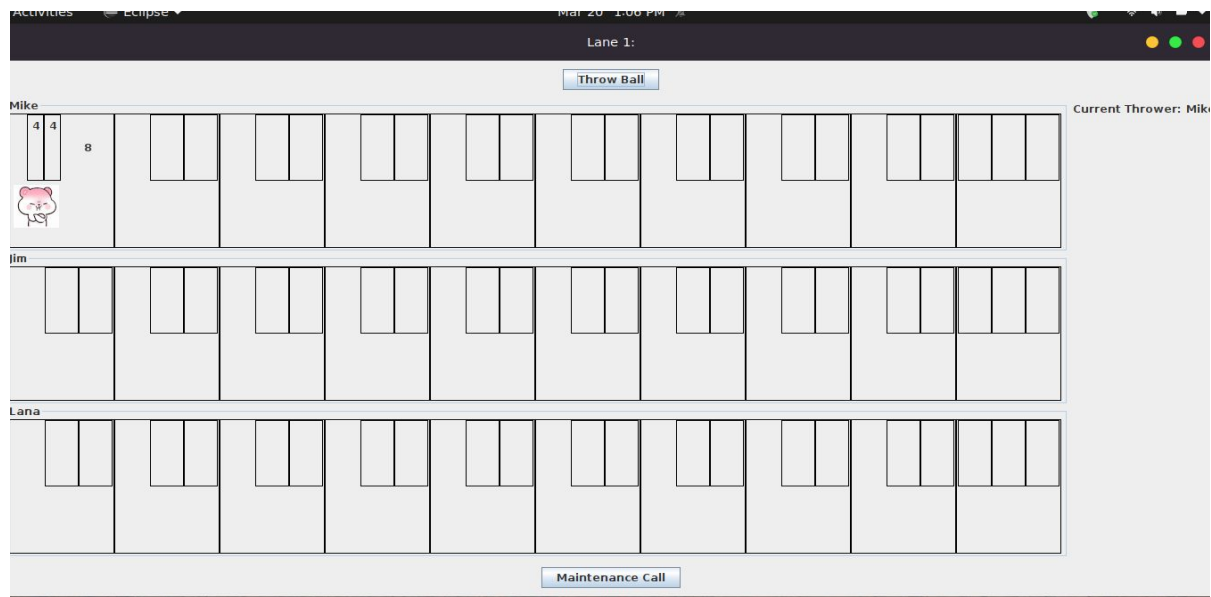
In the previous code throwing the ball is done in continuous fashion without the user intervention so in order to make the game interactive we have implemented a new button named “Throw Ball” when the particular user clicks on the button the ball will be thrown.

Design Pattern Used:

Here we have used an **Observer Pattern**, whenever the throw ball button is clicked then LaneEvent Interface will notify LaneView Class eventually the ball will be thrown.

UI Pattern Used:

Here we have used a **Clear Primary Actions** UI pattern where the player in the party can see who is the current thrower in order to get ready when his chance comes to throw.



Penalty For Gutters:





The previous code was designed in such a way that there was no option to commit a foul so eventually the code has not handled the score calculation part in case of foul. So we added the option of committing a foul by increasing the probability of making a foul. The score calculation incase of a foul is as follows:

1. Whenever there is a single foul the current score is treated as zero.
2. Whenever there are two consecutive fouls if the two fouls are done at the starting of the game then the player would be penalized $1/2$ of the points that is scored in the next frame. Otherwise a player will be penalized $1/2$ points of the highest score obtained by him in the previous throws.

Emoticons:

Implemented emoticons for several possible actions based on each player's score on the Laneview. After each frame in the game The cumulative score is updated and the user is presented with an emoticon expressing their performance for the bowl thrown in the current frame.

We have analyzed the distribution of the player scores and accordingly divided the score and corresponding emoticon as given below.

1.	Low score (<7)	
2.	Medium_Low (<9)	
3.	Medium (<11)	
4.	High (>=11)	

Throw Ball																														
Mike																														
X		20	4	/	40	X		69	X		89	9	/	101	2	3	106	X		125	9	0	134	3	3	140	2	2	144	
																														
Lana																														
3	1	4	6	2	12	X		30	X		44	F	4	48	9	/	60	2	6	68	4	3	75	6	1	82	X	7	1	100
																														
TomH																														
8	0	8	F	6	14	9	0	23	8	/	35	2	7	44	4	2	50	3	/	68	8	/	84	6	/	103	9	/	6	119
																														

We have also added a **logo** for our product as shown below.





UI Design Patterns Used :

We have a tendency to pay more attention and give more weight to negative than positive experiences or other kinds of information. We have tried to benefit from it by using negative emotions using emojis and ensured a proper impact is made on the player by studying the score distribution pattern. We have used Persuasive UI Design Patterns like Loss Aversion, Negativity bias, Optimism Bias.

The proper history of cumulative score and corresponding emoticons is not removed from screen benefitting from the cognitive design pattern known as **Self-Monitoring**.

The user is also given extra points for continuous strikes as per **Fixed-Reward** pattern, **Competition** and **Praise** design patterns.

Tie Breaker

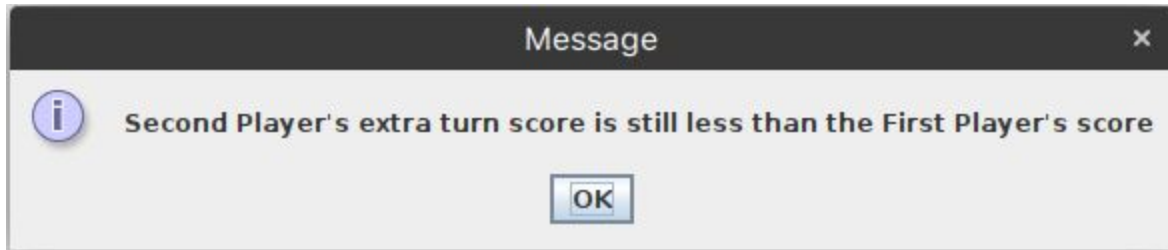
At the end of 10 frames, the second highest player is given a chance to create a tiebreaker scenario. He/She is given one extra chance to bowl, and if the player becomes the highest, a mini game between the first and the second player is created, for three additional frames.

At the end of this, a winner is finalised. If the score is tied at the end of this extra mini game, the winner is the player with most strikes.

If the second highest player fails the highest score even with the extra turn, the game ends and the player with the highest score is declared the winner.

For implementing this particular feature, we used the advantage that the **observer pattern** presented us (We refactored the *Lane* class, *LaneEvent* class, *LaneView* class and the

LaneObserver class to form an observer design pattern), and we extended the advantages offered to us by this refactor. The *LaneEvent* class notifies the *Lane* about the current scores of the players at the end of the first 10 frames, and the *LaneView* is accordingly altered. The top two players are selected, and there is a careful coordination between the *Lane* and its events and the views reflecting the change.



Our intention in redesigning the *LaneView* was to provide the top two players a feeling of competition, independent from the rest of the players. We utilise the **Competition UI Design Pattern** with **gamification**, with complimentary goals, which is a great mechanism to provide incentives for self-improvement, engagement and a feeling of satisfaction and team-play. We also created multiple alert boxes, as displayed above, to increase engagement with the user and to provide context in the happenings of the game.

Below is a representation of the algorithm used to decide the outcome of the game.

