

# **SOFTWARE ENGINEERING**

## Unit 2: Project Design Document

---

**Submitted By:**

**Team No. - 18**

Amanpreet Kaur (2020202005)

Kunal Kandhari (2020202016)

Purnima Grover (2021201014)

Shikha Raghuwanshi (2021201055)

### Index:

● Team Information and their responsibility .....	3
● Overview of the project .....	4
○ Summary of responsibilities of major classes .....	5
● Original Design .....	7
○ Analysis of original design .....	7
■ Strengths .....	7
■ Weaknesses .....	7
● Refactored Design .....	8
○ Analysis of refactored design .....	8
○ Architectural Patterns Used .....	9
○ Design Patterns Used .....	10
○ Summary of responsibilities of each new class .....	12
● Metrics used to refactor the code.....	13
● Initial and final Metrics of the code .....	14
● Screenshots of new requirements .....	15
● Questions on design and code metrics .....	19
● UML Diagrams.....	20
○ Class Diagram.....	20
○ Sequence Diagrams .....	23
○ Flowcharts .....	25
● Analysis of metrics obtained from CodeMR.....	27
● Analysis of metrics obtained from SonarCube .....	29

### Team Information and their responsibility:

- Team Number: 18
- Team Members:

Name	Roll Number	Role
Amanpreet Kaur - 35 hours	2020202005	Implemented penalty for gutters, two consecutive gutters cost half points from the highest score and if those 2 gutters were initial throws, it costs half points from the next frame.
Kunal Kandhari - 36 hours	2020202016	Implemented the database layer to support ad-hoc queries and added emoticons to express feelings based on player scores.
Purnima Grover - 38 hours	2021201014	Made the UI interactive and code extensible and working for multiple players.
Shikha Raghuwanshi - 35 hours	2021201055	Added rules to finalize the winner of the game between 1st highest and 2nd highest scorer.

- Submission Date: 22nd March, 2022

### Overview of the project:

The goal for this(second) iteration of the project was to enhance the bowling alley simulation software, the software that was given during the first iteration of the project. The Lucky Strikes Bowling Center (LSBC) is a software system that automates their chain of bowling establishments. It has bowling lanes and allows player(s) to register themselves at the control desk which monitors the number of frames completed by each bowler and shows the score of each player with the help of scoring station and pinsetter. In the first iteration we refactored the software system given to us to make it comply with the good software engineering design principles. The enhancements made during this iteration of the project are as follows:

- The UI is made interactive in the sense that every player first sets the angle of ball throw and the velocity with which it must move and then only after clicking on the button “throw ball” will the ball be thrown, previously it was all done randomly and the ball thrown was automatic in the code(it was not interactive).
- Before starting the game it first asks the admin for the number of lanes in the game and the number of players allowed per lane. Previously it was fixed, that is, only three lanes were there and only 5 players were allowed to play on a particular lane. Now even a single player can play on a lane or even multiple players as well.
- The database layer is enhanced as well. It stores the information of each player and the scores made by them. Also queries are allowed to extract the lowest/highest scores, top player etc.
- The penalization for gutter throws is also introduced in a way that if a player throws two consecutive gutters, he/she will be punished with half points off of the highest score obtained. And if the two consecutive gutters were the very first two throws of the player then he/she will be punished with half points that are scored in the next frame.
- At the end of 10 frames, a chance is given to the 2nd highest player to bowl. If the player becomes highest, he continues with 3 additional frames between 1st and 2nd highest till the winner is finalized. If there is a tie-break, the winner is declared as whoever had the most strikes.
- The emoji expressions are also introduced. Based on the player scores, the emojis are for either appreciation, envy or embarrassment.
- All the constant strings and variables which are declared in different files like numberOfFrame, numBowlers are kept in constants.java file so that if in future there is a need to change that we only have to change these things in one file only not in multiple files.

## **Unit 2: Project Design Document**

Along with the above enhancements, proper error handling is done to handle erroneous circumstances such as adding more than 'n' players to the game when only 'n' are allowed or not adding any player to the game or removing a player without selecting a player that is to be removed or adding a player with same information that is already present in the database and many more.

The current design is also made to comply with good software design principles such as low coupling, high cohesion, reusability, extensibility, very low code smells, removal of dead code, synchronization, separation of concern, information hiding, law of demeter, usage of good design patterns such as factory design pattern, singleton design pattern, memento design pattern, observer design pattern, state design pattern, adapter design pattern, MVC architecture etc.

### **Summary of responsibilities of major classes:**

Sr. no.	Class	Responsibility
1.	Drive	This is the main class, it starts the bowling game by setting up the control desk with parameters such as number of lanes and maximum number of players allowed in a party.
2.	BowlerFile	Retrieves the bowler's information from the database.
3.	ScoreHistoryFile	Maintains the history of scores of each bowler, that is, it adds scores of a bowler to a file and helps in retrieving the score history of a bowler.
4.	ControlDesk	Keeps track of all the players/parties, available or unavailable lanes in the system. If a lane is available, it is assigned to a party, otherwise the party or parties are put in a waiting queue.
5.	PinSetter	Contains code to decide which pins are left standing after a bowler completes a throw and also resets them after two consecutive throws.
6.	Lane	Corresponds to the representation of the lane, calculates the scores for each throw of the bowl using pinsetter. After the game is over, it calls EndGamePrompt to finish the game.

## Unit 2: Project Design Document

7.	Party	Maintains the database about the information of the players present in a party
8.	EndGamePrompt	After the game is over, it asks whether the party wants to continue playing with a new game.
9.	ScoreReport	Shows final scores of a bowlers, it can also send an email of the scores to the bowler or take a printout for that.
10.	Utility	Contains static method for creation of button which is used by all the view Classes and helped in removing the redundancy in code and provide high Cohesion in View classes.
11.	CreateControlDeskGUI	Separated Controls Panel GUI creation part to achieve higher cohesion.
12.	ControlDeskSubscribe	It contains the subscribe and publish part extracted from the ControlDesk Class to achieve higher cohesion.
13.	LaneSubscribe	It contains the subscribe and publish part extracted from the Lane Class to achieve higher cohesion.
14.	FindScore	This class will handle all the calculations of scores for party and bowlers.
15.	GameStart	All the methods related to games are managed in this class.
16.	GamePinSetting	All the pin settings which are required in the beginning of the game will be handled by this class.
17.	LaneVariables	This class contains the important data members which are shared with other classes which are associated with the Lane class.

### **Original Design:**

We began our analysis by creating various UML diagrams for the original design so that we can get to know the various dependencies between each class. Sonarcube is a Code Quality Assurance tool that provides reports for the code quality of the project. After analyzing below are our strengths and weaknesses of the original design:

### **Analysis of original design:**

#### **Strengths :**

- There is segregation of the code in different files with respect to the functionality it is performing like information of bowlers, Party and Score is in the pojo(plain old Java object) package where only information related attributes, getter and setter are used.
- A lot of repeated code was removed and a new class was created for that and whenever that code was needed the function of the new class was called. For eg - Code for creating labels and buttons is repeated multiple times, we write that code in Utility.java and whenever there is a need to create a button, we will call the function of Utility class.
- There is High Cohesion and Low Coupling amongst most of the classes.
- Followed camel-case syntax for naming the class, interface, method, and variable.
- Proper commenting in various classes and methods.
- The code smells in the overall project were very low.

#### **Weaknesses:**

- The interactivity with the user was missing. It was just a randomized simulation of the bowling alley game.
- The game only allowed three lanes and only five players per lane. If a single player wanted to play, he/she was not allowed to get a lane. So it was hard coded in that sense.
- The error handling part was missing such as it did not show proper error messages on adding more than 'n' players to the game when only 'n' are allowed or not adding any player to the game or removing a player without selecting a player that is to be removed or adding a player with same information that is already present in the database and many more.
- More good design patterns could have been explored and implemented such as factory design pattern, singleton design pattern, memento design pattern, observer design pattern, state design pattern, adapter design pattern etc.

### **Refactored Design:**

The original design was modified to make the game more interactive by making ball throws at a certain angle with certain velocity with a push of a button and not just randomized, various penalizations were also introduced for gutter throws and error handling was made better.

For refactoring, we focused on identifying and removing various bugs, code smells, security hotspots, dead codes, lengthy classes, and commented code, for which we have made use of SonarCude which is a Code Quality Assurance tool that collects and analyzes source code, and provides reports for the code quality of the project. It combines static and dynamic analysis tools and enables quality to be measured continually over time.

### **Analysis of refactored design:**

The enhancements made are as follows:

- The UI is made interactive in the sense that every player first sets the angle of ball throw and the velocity with which it must move and then only after clicking on the button “throw ball” will the ball be thrown, previously it was all done randomly and the ball thrown was automatic in the code(it was not interactive).
- Before starting the game it first asks the admin for the number of lanes in the game and the number of players allowed per lane. Previously it was fixed, that is, only three lanes were there and only 5 players were allowed to play on a particular lane. Now even a single player can play on a lane or even multiple players as well.
- The database layer is enhanced as well. It stores the information of each player and the scores made by them. Also queries are allowed to extract the lowest/highest scores, top player etc.
- The penalization for gutter throws is also introduced in a way that if a player throws two consecutive gutters, he/she will be punished with half points off of the highest score obtained. And if the two consecutive gutters were the very first two throws of the player then he/she will be punished with half points that are scored in the next frame.
- At the end of 10 frames, a chance is given to the 2nd highest player to bowl. If the player becomes highest, he continues with 3 additional frames between 1st and 2nd highest till the winner is finalized. If there is a tie-break, the winner is declared as whoever had the most strikes.
- The emoji expressions are also introduced. Based on the player scores, the emojis are for either appreciation, envy or embarrassment.

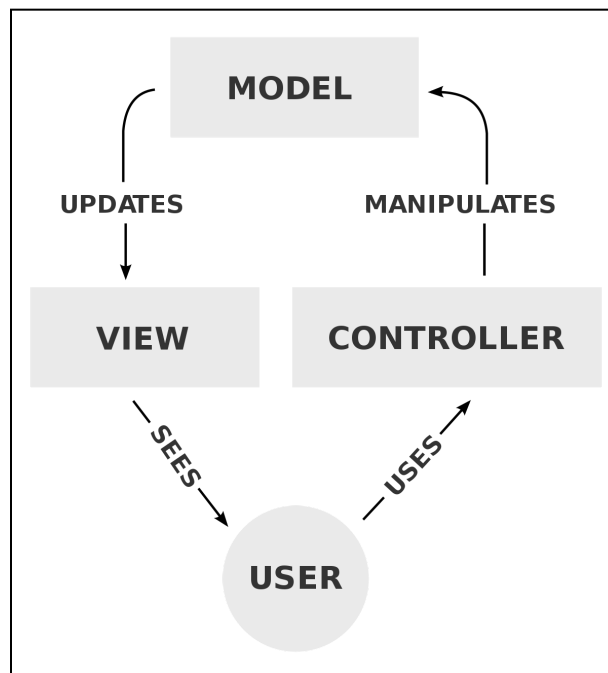


Along with the above enhancements, proper error handling is done to handle erroneous circumstances such as adding more than 'n' players to the game when only 'n' are allowed or not adding any player to the game or removing a player without selecting a player that is to be removed or adding a player with same information that is already present in the database and many more.

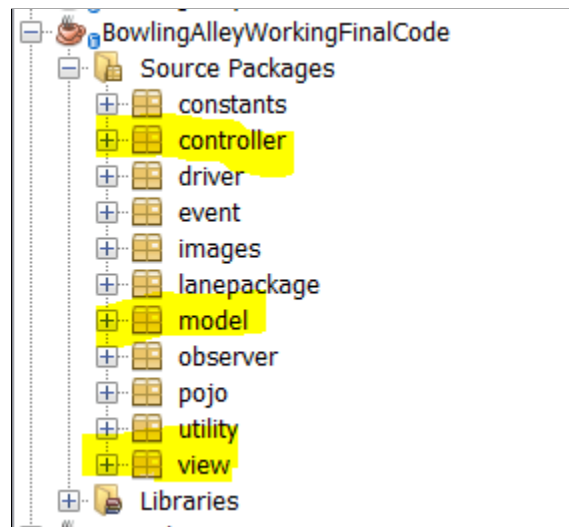
We also focused on identifying and improving the various code metrics like High Coupling between classes, Low Cohesion in classes, High Complexity, High Size and various other metrics for which we made use of CodeMR. We calculated the metrics, checked how our updates affected the metrics compared to what it was earlier. Based on the analysis we carried forward the refactoring.

### Architectural Used:

- **MVC architecture** is used, where MVC is an architectural pattern consisting of three parts: Model, View, Controller.
  - **Model:** Handles data logic.
  - **View:** It displays the information from the model to the user.
  - **Controller:** It controls the data flow into a model object and updates the view whenever data changes.

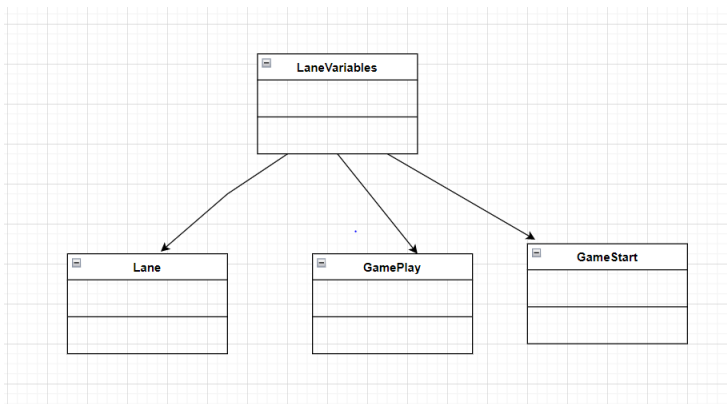


Screenshot of our code having model, view, controller packages:

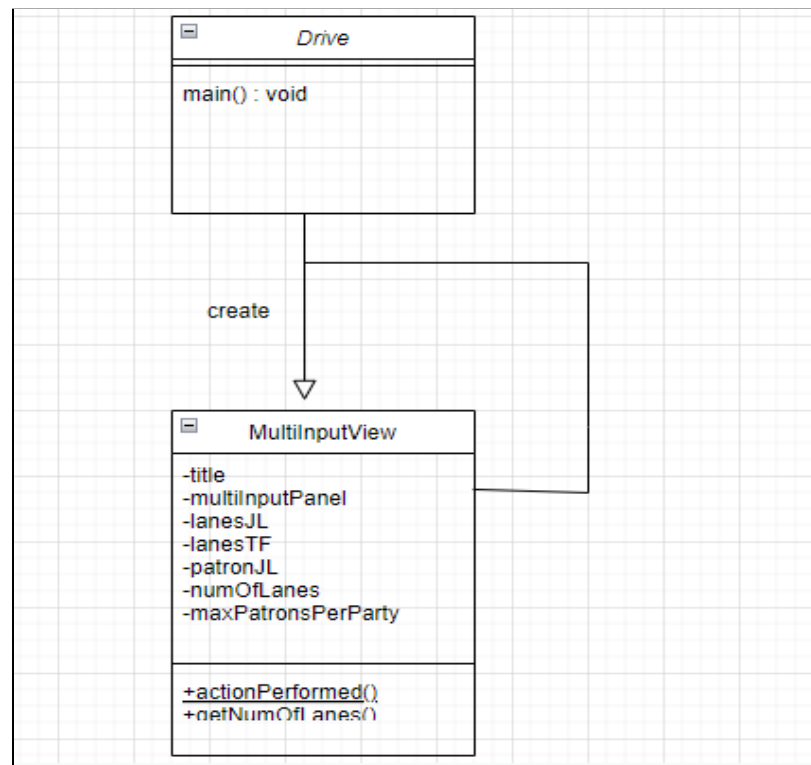


### Design Patterns Used:

- **Factory Design Pattern** - We created a superclass of LaneVariables where all the attributes of Lane class are stored and while splitting Lane Class into GamePlay, GameStart and CalculateScore, we inherit the class LaneVariables in all these classes and for accessing and updating all variables getters and setter of object of super class is used.



- **Singleton Design Pattern** - This pattern is reflected in the 'drive' class which is the main class. It is only called once in the whole software. So only a single object is created.

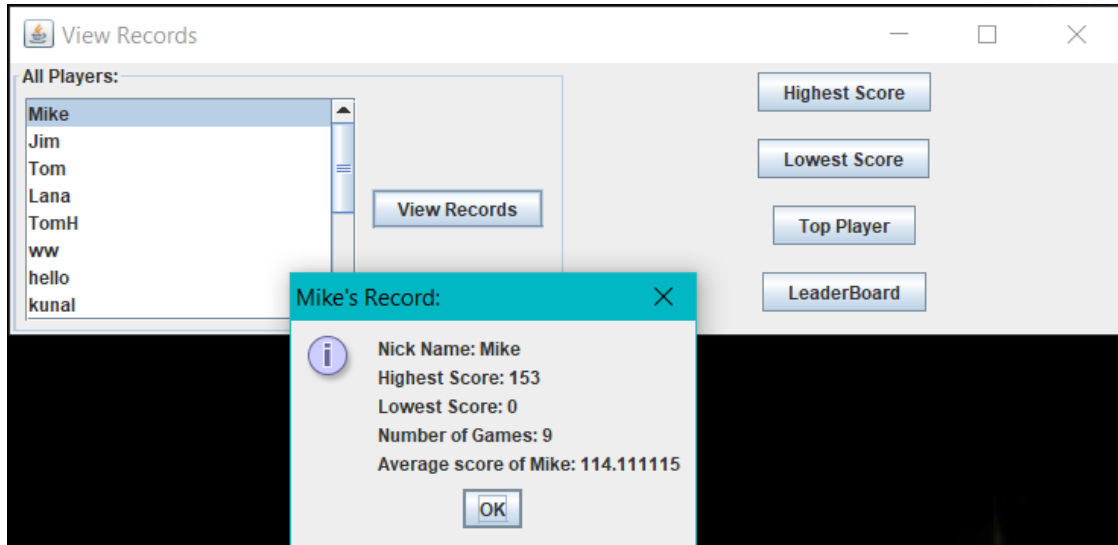


- **Memento Design Pattern** - This pattern allows us to maintain the previous state of an object. In our design, in the `score.java` class, the scores for each frame are stored in a 2D array cumulative score, and whenever the scores change, the previous scores remain intact.
- **Observer Design Pattern** - In the `ControlDesk` class, it notifies its subscribers when new party is added or when a lane assigned to a party; in the `PinSetter` class, it notifies its subscribers when a ball thrown knocks down the pins or when the pins are reset; in the `Lane` class, it notifies its subscribers when the players do not want to continue playing another game after their current game got over.
- **State Design Pattern** - This is a behavioral design pattern that allows the modification of the object behavior according to its internal state. In the current design, whenever the score of the player changes, it is reflected in other components as well such as the emoticon component which reacts according to the player score and it is also reflected on the UI of the lane view.
- **Adapter Design Pattern** - We used Adapter Design Pattern which connects two incompatible interfaces. Here, there is a class `CalculateScore.java`, where we separated

## Unit 2: Project Design Document

the code of calculation of score from LaneClass making it work as an adapter. It acts as a middle layer between Lane Class and Score.java.

- **Breadcrumbs pattern** is used for the UI part of this project. It helps when the user wants to query something, a query window opens and the user can ask various queries and get desired outputs.



### Summary of responsibilities of new classes:

We have added following new classes to the code:

Sr. no.	Class	Responsibility
1.	TieBreaker.java	This class is created to implement the logic for giving additional frames to 1st and 2nd highest players. The view to implement the tie break is done in this class and the whole logic for the same is implemented here.
2.	QueriesView.java	All the queries such as highest/lowest scores, top players are written here. It acts as a bridge between database and frontend.
3.	MultiInputView.java	This helps in the GUI implementation for the initial inputs for the game such as number of lanes and number of players allowed per lane.

## Unit 2: Project Design Document

4.	Utility.java	Contains static method for creation of button which is used by all the view Classes and helps in removing the redundancy in code and provide high Cohesion in View classes. Created various methods such as “showmessage” which will show all the warning messages and “getButtonPanel” which generates panels for buttons
5.	Constants.java	All the global constants used overall in the project are kept here.
6.	GamePinSetting	All the pin settings which are required in the beginning of the game will be handled by this class.

### Metrics used to refactor the code:

- **Coupling** - It refers to the dependency of one class on another in the system. It is desired to have loose coupling in the project as if we change any requirements in the future then there will be less changes as dependency will be less with other classes. Also, information flowing between the two classes is less. Loose Coupling improves the test ability of the system, flexibility and reusability of the code also increases.
- **Lack of Cohesion** - Cohesion measures the dependency of one method in class to another method. A good system should have methods which perform only one functionality. High cohesion is desirable in the systems as it provides encapsulation to the system. The classes with low cohesion depict inappropriate design and also increase complexity. The probability of errors also increases due to this. We have to divide that one class into more than two classes.
- **Line of Code** - It measures the size of the system, mainly it calculates the number of lines in a class and for a good system it is desirable to have classes which have less number of lines and also all the functions have functionality belonging to one class.
- **Complexity** - It tells how much the code is readable or understandable to another person. It also finds the interaction of one object to other objects of the system. The system needs to have a low level of complexity as it makes the software more readable and maintainable in the future. There will be high chances of errors in other classes also while making one minor change in the code if the system is high in complexity.
- **Size** - It is calculated by counting the number of lines of code or number of methods in the complete system. If a class has high size then that class needs to be split into multiple classes and the high size of a particular class depicts that it is not manageable at all.

## Unit 2: Project Design Document

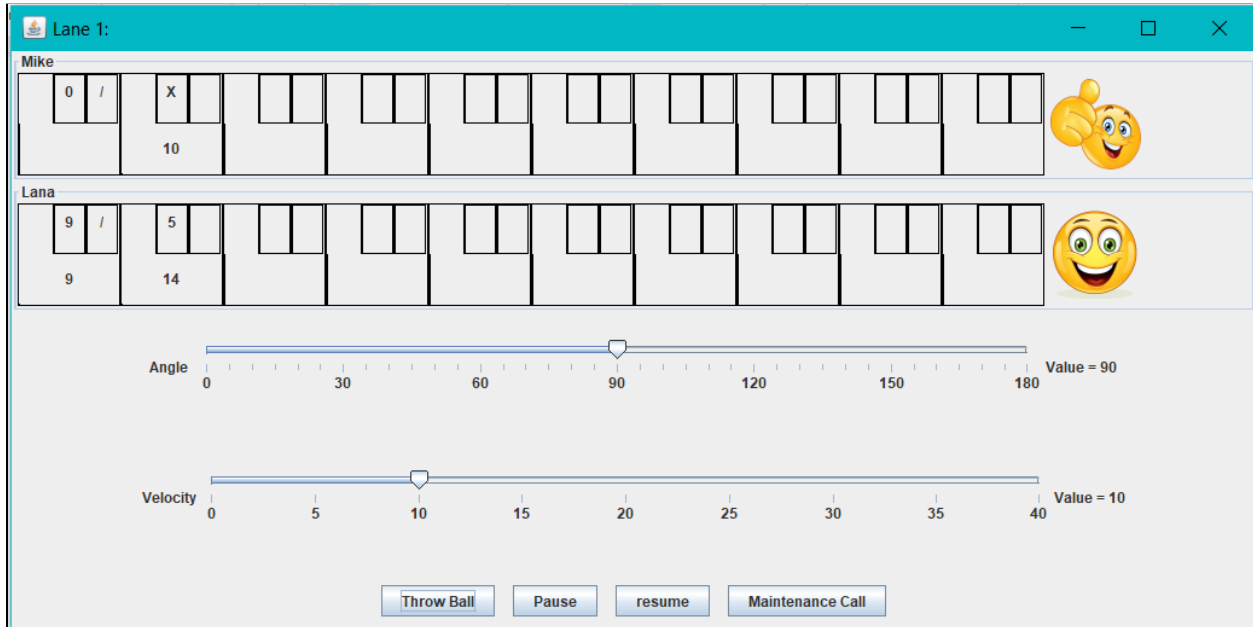
- **C3** - It stands for combination of Coupling, Cohesion and Complexity. It depicts the max value of Coupling, Cohesion and Complexity metrics.
- **Weighted method count** - It is related to complexity and size of the system. The number of possible execution paths of a class increases with the number of methods and their control flow complexity. Depending on the weight metric, it allows an assessment of the number of methods and their complexity.

### Initial and final Metrics of the code:

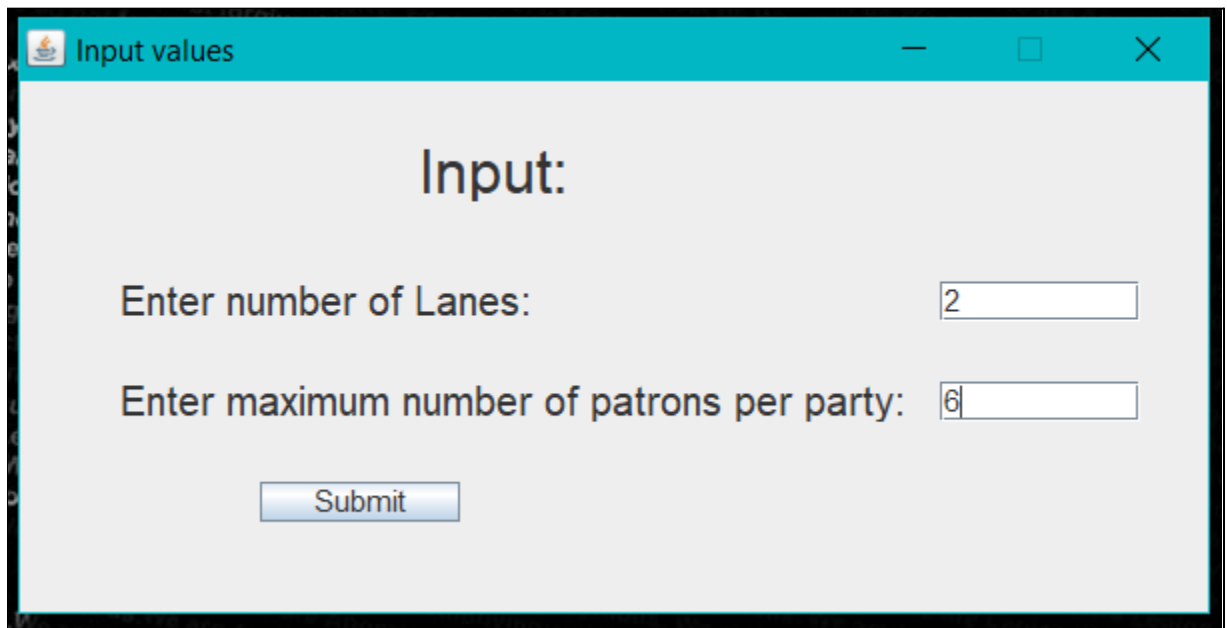
Metrics	Old Code (in %)			Refactored Code (in %)		
	Low	Low-Medium	Medium-High	Low	Low-Medium	Medium-High
<b>Coupling</b>	67	33	0	94.6	5.4	0
<b>Lack of Cohesion</b>	40.1	36.6	23.4	60.9	39.1	0
<b>Size</b>	20.2	79.8	0	19.8	80.2	0
<b>Complexity</b>	54.9	29.3	15.8	53.9	44.6	1.5
<b>C3</b>	40.1	36.6	23.4	46.6	52	1.4
<b>Weighted Method Count</b>	60.9	23.3	15.8	61.5	38.5	0

### Screenshot of New Requirements:

- Simulation replaced by an actual set of players playing the game:

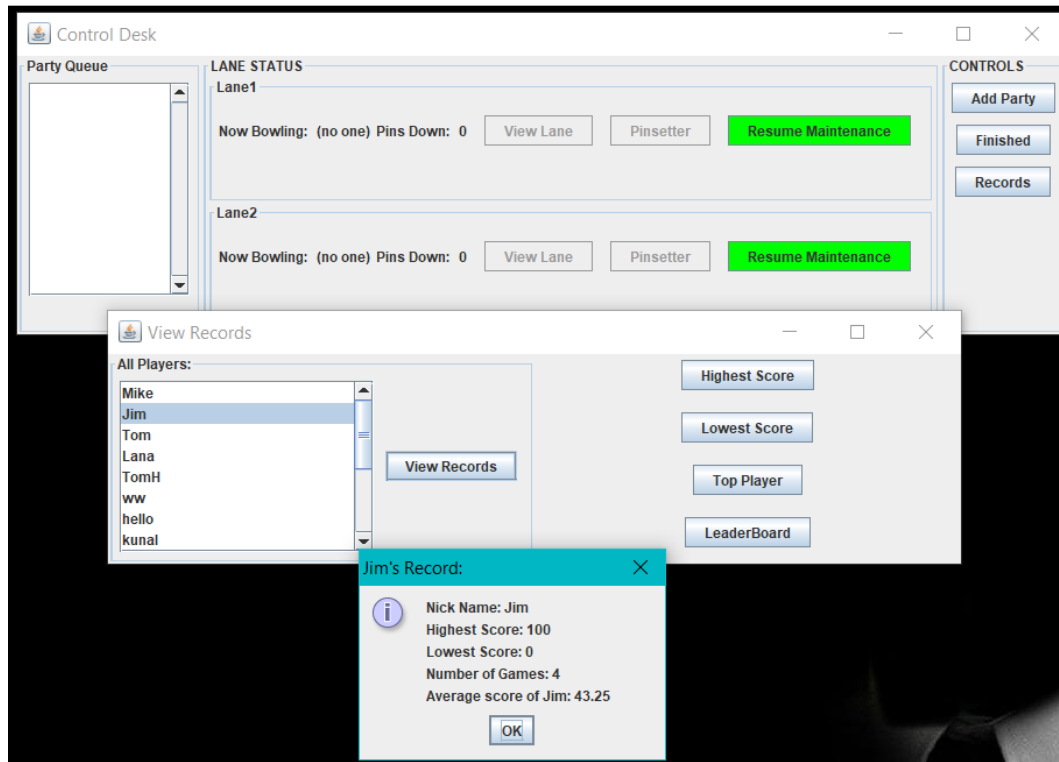


- Code made extensible and working for multiplayer:

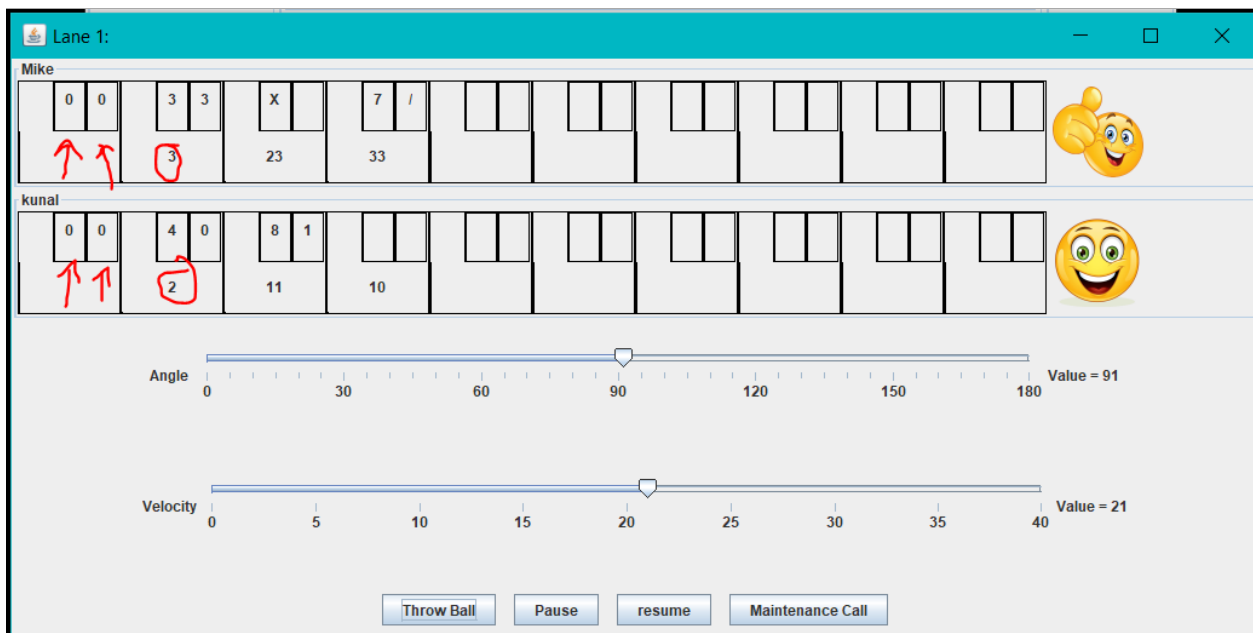


## Unit 2: Project Design Document

- Some possible queries - highest/lowest scores, Top Player:



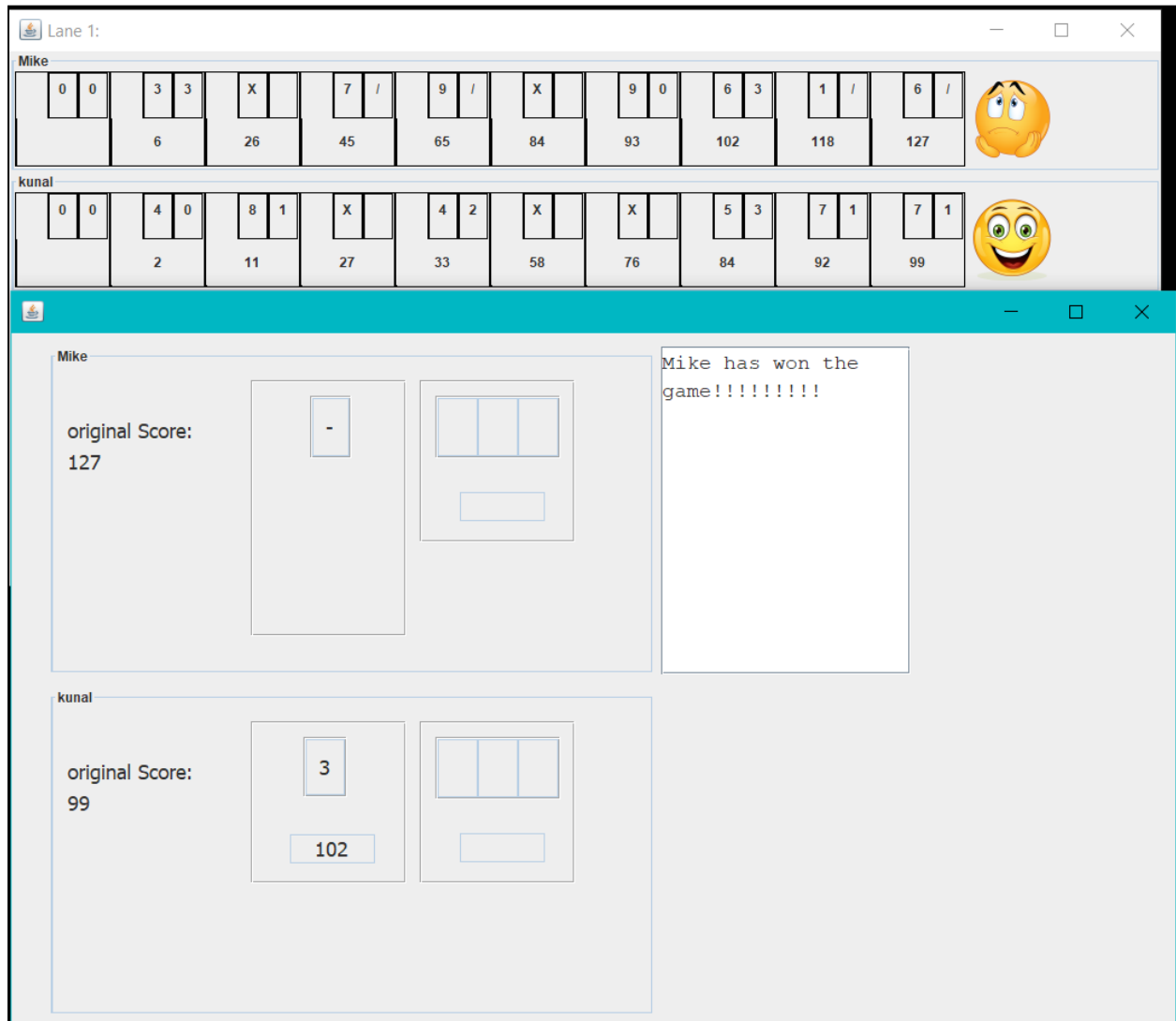
- Penalty for Gutters:








## Unit 2: Project Design Document

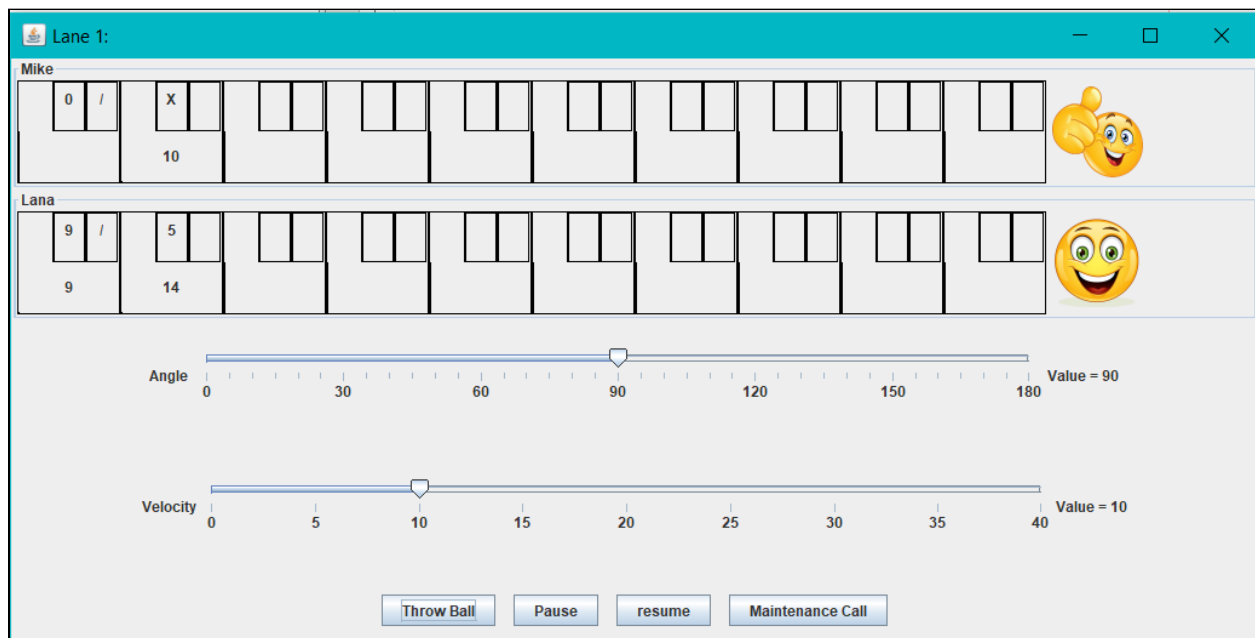
- Providing a chance to the 2nd highest player to bowl:



## Unit 2: Project Design Document

- **Emoticons for appreciation, envy, embarrassment based on player score:**

	All pins down
	0 pins down
	Any other number of pins down



### **Ques-1: What were the metrics for the code base? What did these initial measurements tell you about the system?**

#### **Ans:-**

The code metrics considered for the analysis of both the original and refactored code are coupling, cohesion, cyclomatic complexity, LOC(lines of code), size, C3, weighted method count etc. Various graphs included in this report show the values of these code metrics for the given project. The complete analysis of these code metrics is done in the initial parts of this report which thoroughly explains the analysis of the original design.

### **Ques-2: How did you use these measurements to guide your refactoring?**

#### **Ans:-**

The code metrics gathered from the original design helped us to better refactor the original design. The patterns used in the refactored design and the whole thorough design analysis is done in the initial section of this report which highlights the refactored design analysis. Even after adding the new functionality and interactivity in the original design, the code analysis was done and further refactoring was done in order to ensure low coupling, high cohesion and very low code smells in the final design.

### **Ques-3: How did your refactoring after the metrics? Did your refactoring improve the metrics? In all areas? What contributed to these results?**

#### **Ans:-**

- The refactored code has improved in quality of code, lack of cohesion is also improved, coupling and cohesion also changed.
- Few classes are divided into multiple classes to increase readability and decrease coupling.
- Repetitive code will be added in one class and called from there again and again so that there will be no redundancy in the code.
- Removed the code smells.
- Cognitive complexity of the classes are decreased in many classes.

So, making all these changes decreases the overall complexity of the code.

## Unit 2: Project Design Document

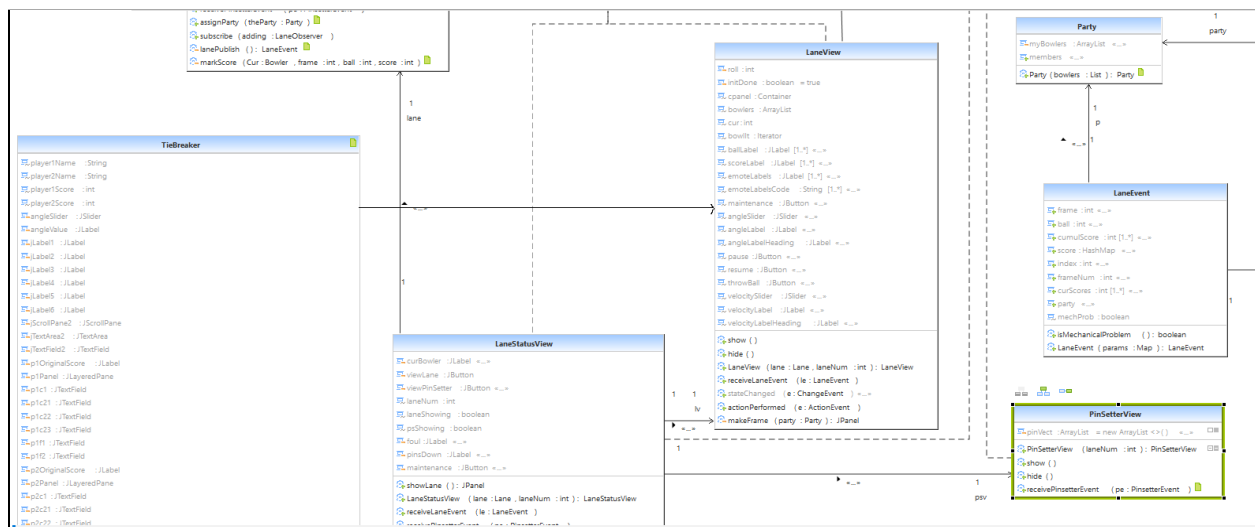
The comparison which is done above in the pie charts and tables shows how the new code decreased the complexity of the system with respect to the original code.

### UML Diagrams

#### Class Diagrams:

#### Tie-Breaker UML

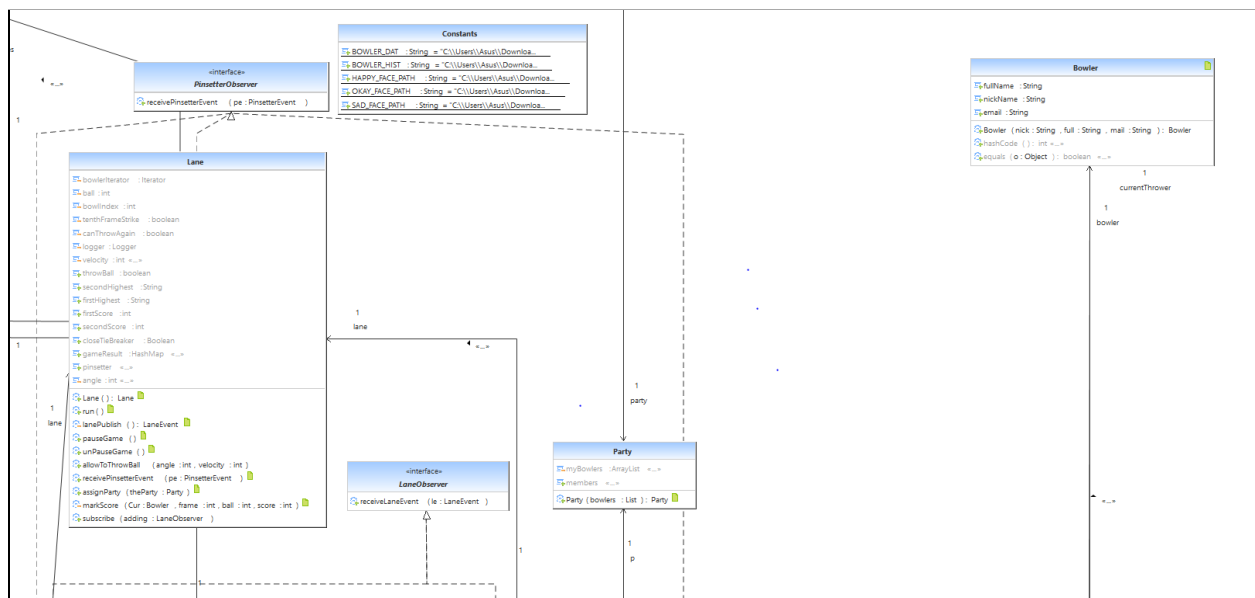
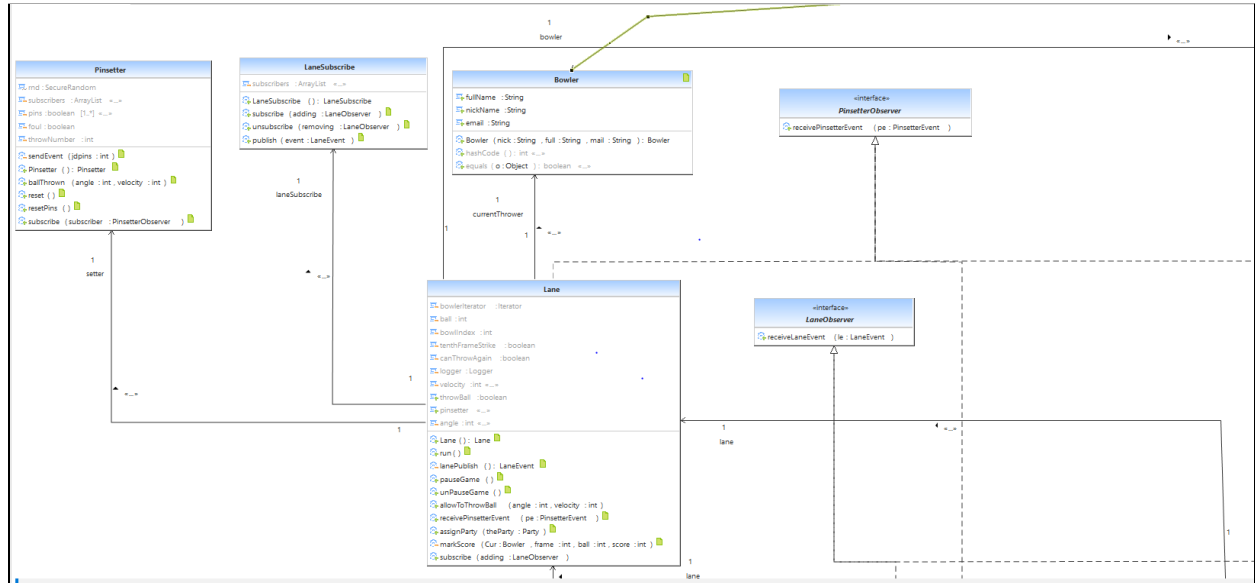
UML class diagram of tiebreaker class and all its related components.



## Unit 2: Project Design Document

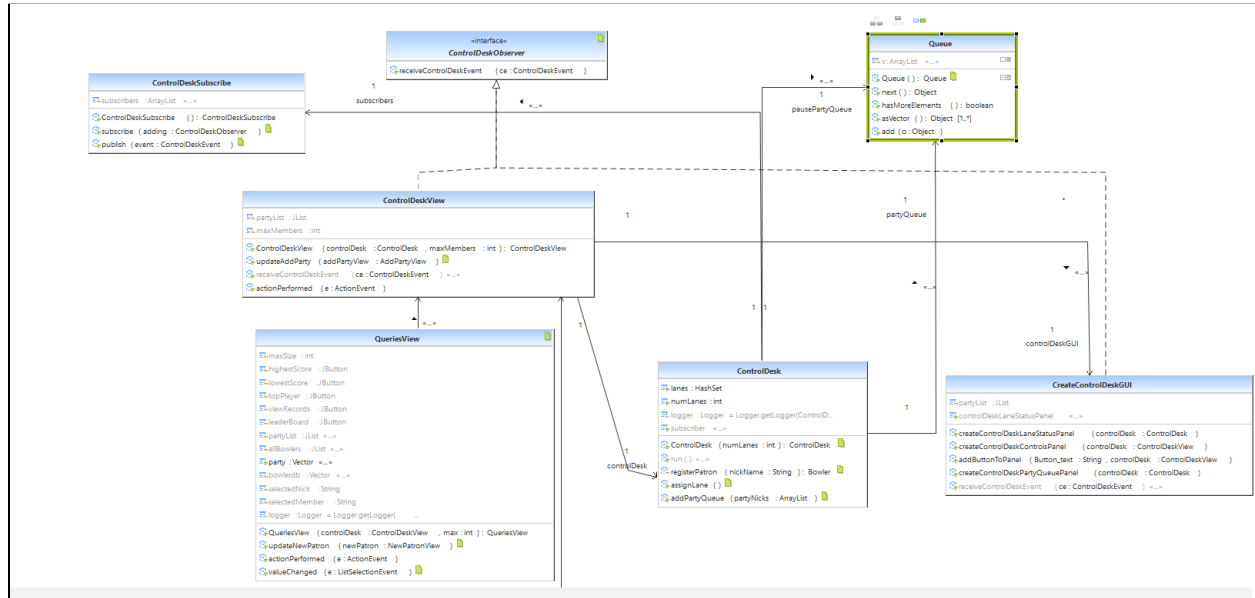
## Lane UML

UML class diagram of lane class and all its related components.



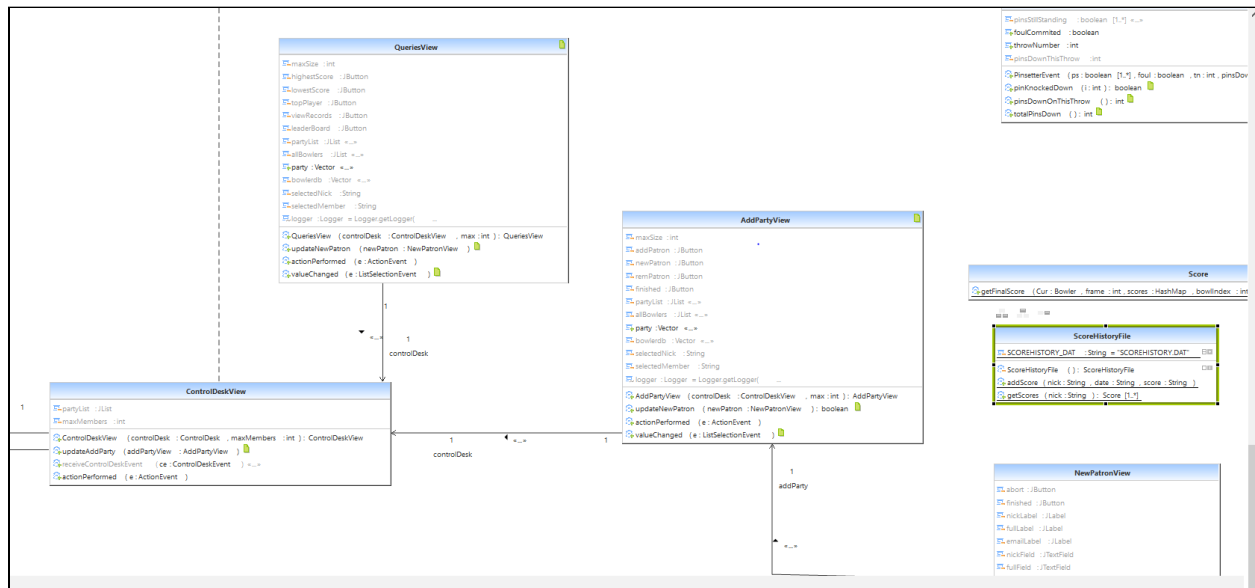
## QueriesView UML:

UML class diagram of QueryView class and all its related components.



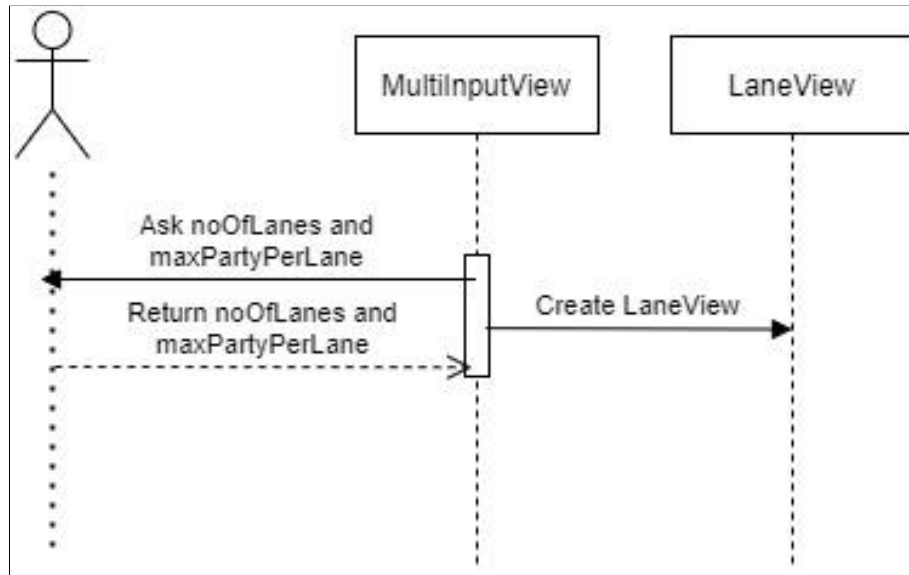
## Control Desk UML :

UML class diagram of control desk class and all its related components.

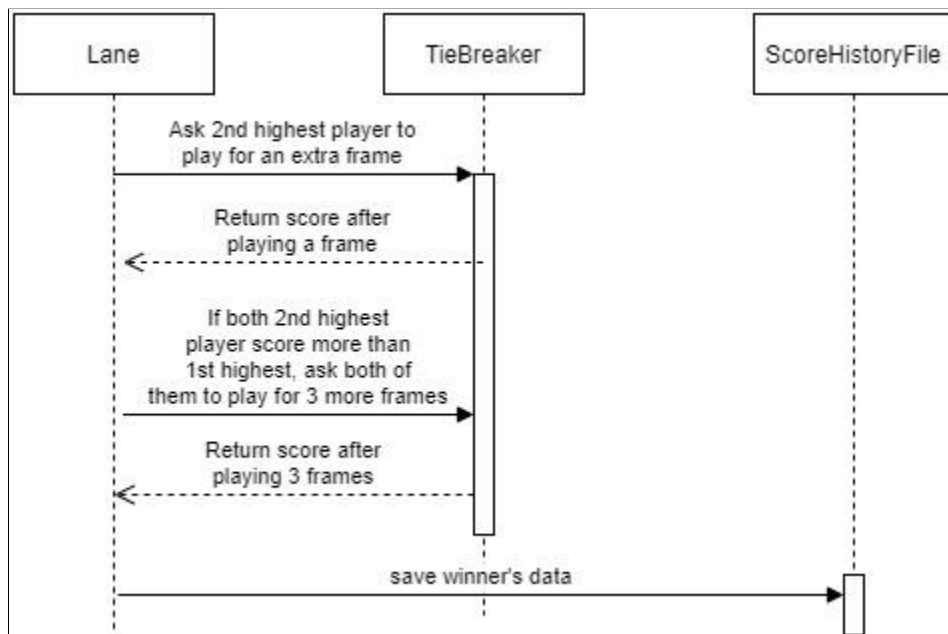


### Sequence Diagrams:

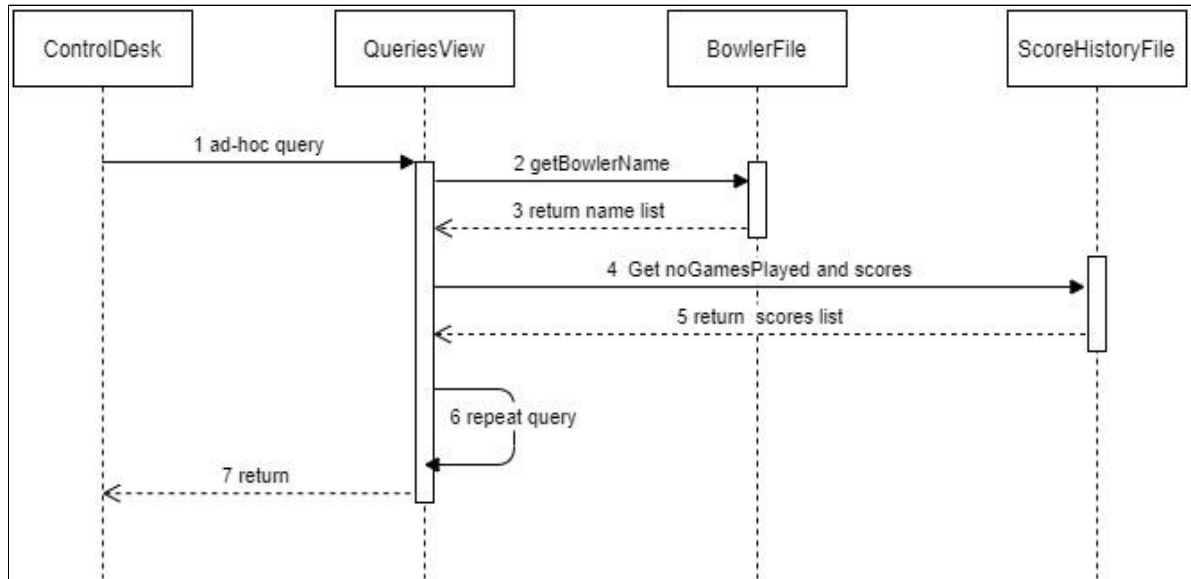
- Sequence Diagram for multi-player input from user



- Sequence Diagram for TieBreaker event

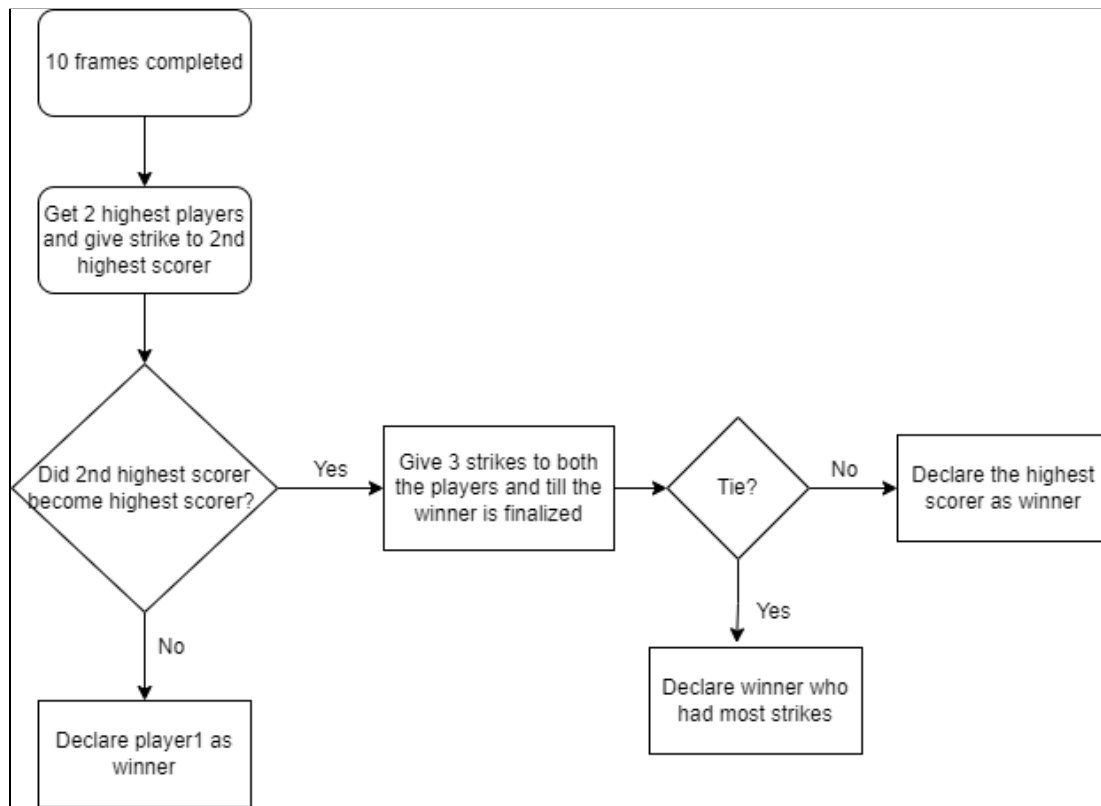


- Sequence Diagram for Ad-Hoc queries





### Flow Diagrams:



### The design pattern reflects on various criteria.

#### Low Coupling:

Larger classes like Lane.java were broken down into few classes which will be the subclasses such that the dependencies remained minimum overall. Various unnecessary relationships and dependencies were removed. Low coupling improves the test ability of the system, flexibility and reusability of the code also increases.

#### High Cohesion:

Cohesion measures the dependency of one method in class to another method. A good system should have methods which perform only one functionality. The classes with low cohesion depict inappropriate design and also increase complexity. The probability of errors also increases due to this. We have to divide that one class into more than two classes. We have separated the score calculation part from some classes and removed some redundancy in the functionalities from

these methods. We have also created an Utility class which contains a static method for creation of buttons which is used by all the view classes in order to maintain high cohesion.

### **Separation of concern:**

It is a design principle in which the different sections of the application are separated and the separated section represents the different concerns. We have separated various java files according to the functionalities and related tasks into different packages. A proper package structure is maintained. The score calculation part is separated from the original files.

### **Information hiding:**

Information hiding is maintained by properly implementing the OOPS concepts, mainly the abstraction. There are various interfaces which are implemented, the interfaces are the means of communicating with the other objects. We have also maintained the LaneVariable class which is having all the data members which are private and getter setters are the behaviors by which these private data members can be accessed.

### **Law of Demeter:**

According to the LoD principle, few classes should also be interacting with a lesser number of other classes to some extent. This is achieved by having a low coupling among the classes. Higher coupling will not be easy to handle and programs become difficult to maintain. The communication is done by passing objects or by creating the instance variables. The classes are independent to each and low coupling is maintained satisfying the LoD.

### **Extensibility:**

Some base features are set and with time some new features can be extended to the existing feature; the older features are not affected; this is the concept of extensibility. We are working properly by maintaining different packages which contain classes with similar functionalities. And it is easy to introduce more features to the existing system. For all the new features different methods are created for the desired functionalities.

## Unit 2: Project Design Document

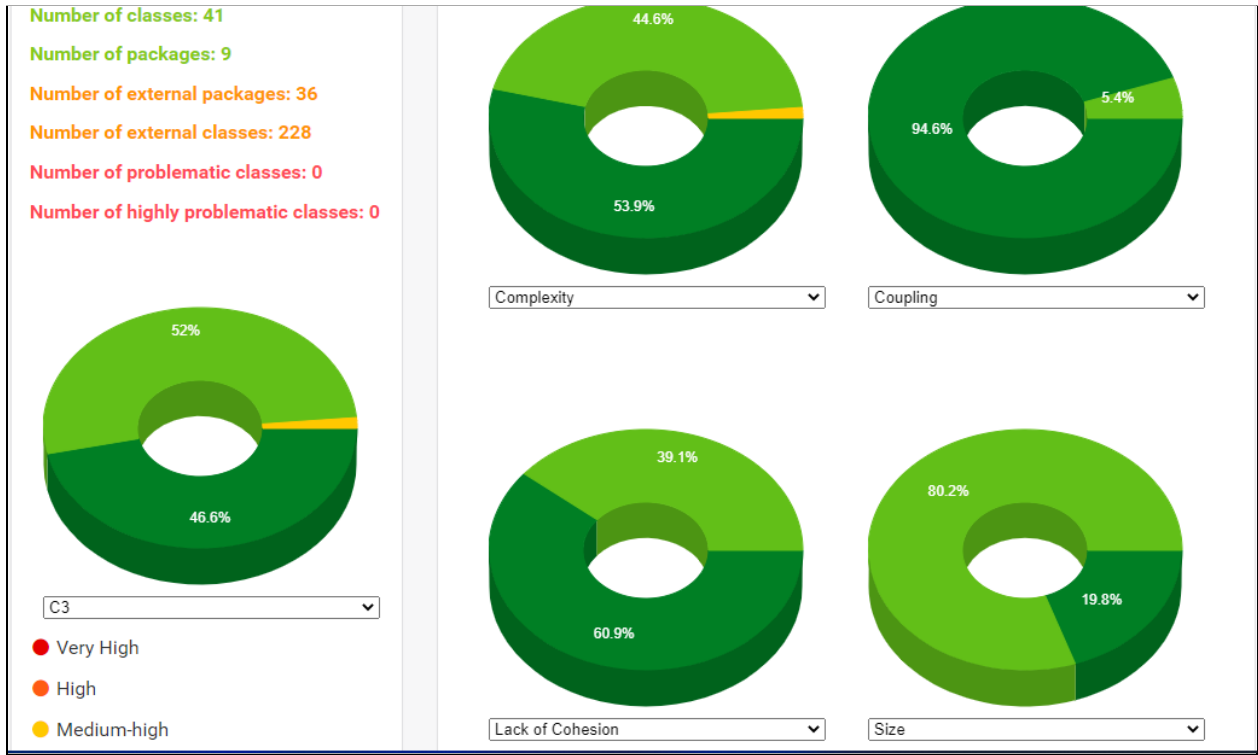
### Reusability:

Proper code reusability is maintained by creating different functions and classes. The code redundancy is removed from the original code. There are various static methods which are created for performing the different tasks, and they can be called anywhere when required.

### CODEMR:

1	Lane					110	low-medium	low-medium	low	low-medium
2	MultInputView					29	medium-high	low	low	low
3	LaneView					254	low-medium	low	low-medium	low-medium
4	QueriesView					138	low-medium	low	low-medium	low-medium
5	AddPartyView					129	low-medium	low	low-medium	low-medium
6	Score					78	low-medium	low	low	low-medium
7	CalculateScore					74	low-medium	low	low	low-medium
8	ControlDesk					65	low-medium	low	low-medium	low-medium
9	ControlDeskView					59	low-medium	low	low-medium	low-medium
10	PinSetterView					109	low	low	low	low-medium
11	NewPatronView					99	low	low	low	low-medium

METRIC ANALYSIS



### SONARQUBE ANALYSIS:

