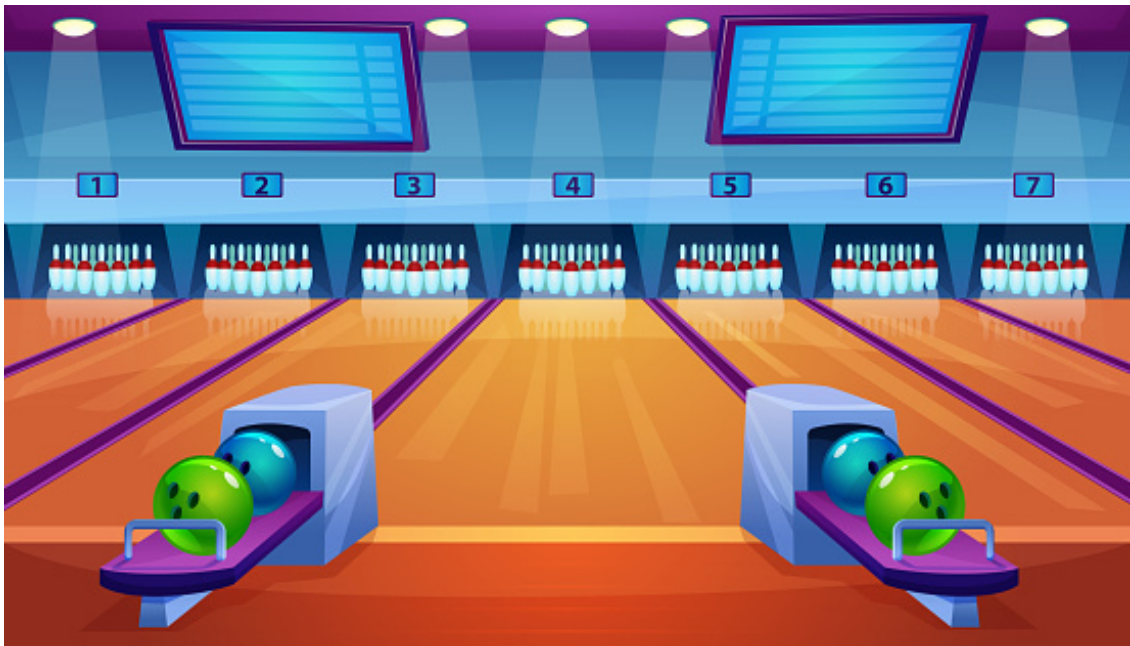


UNIT 2 PROJECT

Bowling Alley Enhancement

Design Document



TEAM-15

Anchal Soni 2020201099

Abhidha Jain 2021201038

Ayushi Maheshwari 2020201053

Karan Negi 2021201039

Date of submission

22 March, 2022

CONTENTS

1. Team member information	3
2. Overview of the project	4
3. Objective of our task	5
4. Description of each class in existing code	6
5. Analysis of original code design	7
7.1 Strength	7
7.2 Weakness	8
7.3 Fidelity to design document	8
7.4 Design Patterns used	9
7.5 Metric Analysis	9
6. UML Class Diagram before Enhancement	11
7. UML Sequence Diagram before Enhancement	14
8. New Functionalities Added	16
8.1 Making the UI interactive	16
8.2 Making the Code extensible and working for multiplayer	17
8.3 Adding the database layer and implementing the queries (Max, lowest, average score, top player, Number of Bowlers)	19
8.4 Implementing penalty for gutters	22
8.5 Declaring the Winner	22
8.6 Implementing emoticon	24
8.7 Making the code configurable	27

9. Description of modified classes	27
10. UML class diagram after enhancement	28
11. UML sequence diagram after	31
12. Code Metrics of Final Code	34
13. Analysis of Metric	37
14. Conclusion/Summary	39

1. TEAM MEMBER INFORMATION

Name and Roll Number	Working Hrs.	Roles
Abhidha Jain (2021201038)	30 hrs	Implemented a new UI to make the application interactive; Implemented the functionality to decide whether to give runner up a second chance and how
Anchal Soni (2020201099)	8 hrs	Contributed towards metrics generation and its analysis; Contributed to design documentation
Ayushi Maheshwari (2020201053)	25 hrs	Implemented emoji association with the scores according to pin down; Prepared class diagrams and sequence diagrams from final code, Contributed towards the documentation.
Karan Negi (2021201039)	30 hrs	Added functionality to read from config file; Implemented Show Scores view to query the database; Implemented penalty for gutters

2. OVERVIEW OF THE PROJECT

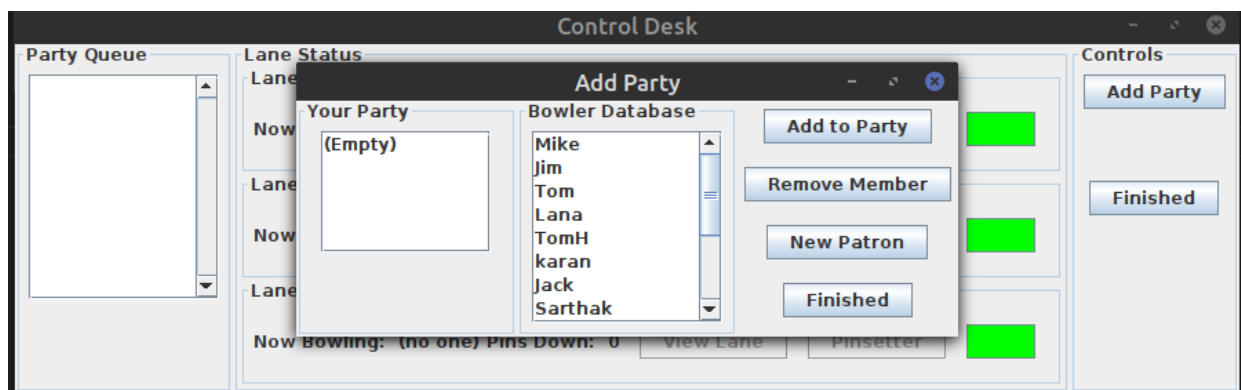
The project is based on simulating and managing a Bowling Alley. It is based on an earlier version of Java (Java 5). The project's main goal is to simulate and operate a bowling alley.

A bowling establishment is commonly referred to as a bowling alley. A bowling alley is composed of a number of bowling lanes. When a bowler enters a LSBC, he checks in at the control desk for a lane assignment. Bowlers may check in as a group or party so that they will be assigned to the same lane. Each lane can accommodate one to five bowlers. The order in which a party checks in determines the order in which they will bowl.

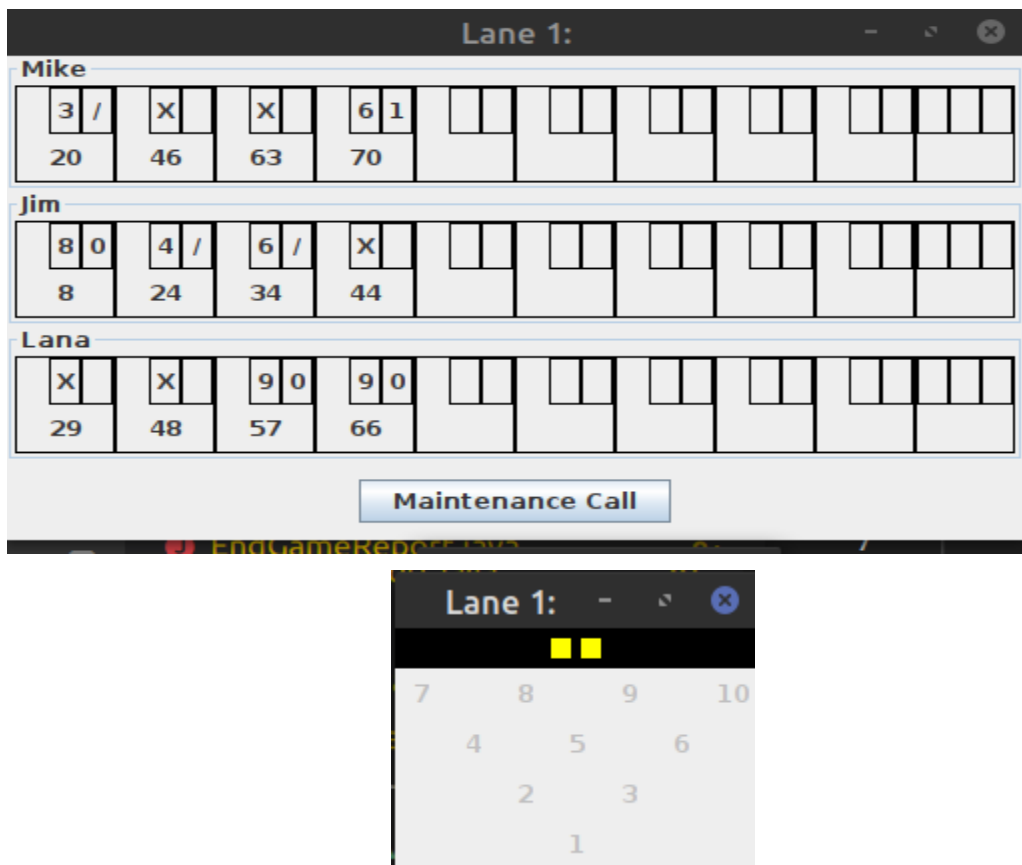
The user can select a party of his own with a number of players who can play the game, taking turns to knock over Bowling Pins.



A control desk may be utilized to monitor a set of lanes in this Bowling Alley project.



Those lanes can be assigned to the Bowler parties. At the end of the game, the pinsetter configuration may be reviewed, and scores can be generated.



3. OBJECTIVE OF OUR TASK

The purpose of this project is to enhance the functionalities of refactored code, there are a lot of new features which could be included in the bowling alley simulation. New enhancements which need to be implemented are:

1. New UI patterns
2. Cap for max players
3. Database layer & Search interface
4. Penalty for Gutters
5. Providing extra frames to the second highest scorer and declaring a winner.
6. Emoticons implementation
7. Configurable numbers

4. DESCRIPTION OF EACH CLASS IN EXISTING CODE

S.No	CLASS NAME	RESPONSIBILITIES
1.	Drive	This class begins the application and creates an alley class object and ControlDeskView class object initialized with necessary parameters <i>numLanes</i> and <i>maxPatronsPerParty</i> .
2.	BuildGeneralComponents	This class is responsible for building various UI components such as frames,panels & buttons.
3.	ControlDesk	It's a threaded class which initializes input number of lanes, assigns lanes, creates a party of nickname vectors and maintains a wait queue. It also maintains a list of subscribers and notifies all of them in case of an event.
4.	controlDeskEvent	return a Vector of Strings representing the names of the parties in the wait queue
5.	controlDeskObserver	Interface for classes that observe control desk events
6.	controlDeskView	Class for representation of the control desk. Makes use of GUI components to represent the control desk
7.	AddPartyView	The GUI used to Add Parties to the waiting party queue.
8.	Bowler	Class that holds all bowler info
9.	BowlerFile	Class for interfacing with Bowler database.Class for interfacing with Bowler database
10.	Lane	Classes for the lane object that is assigned for a party. It is a thread based class than extends <i>Thread</i> class and implements <i>PinsetterObserver</i> observer. It runs the simulation and assigns the score to the bowlers.

11.	LaneStatusView	Class for representation of the lane view. Makes use of GUI components to represent the Lane view
12.	Party	It is a container that holds bowlers
13.	PinSetter	This class represents the pinsetter and is responsible for all pinsetter-related actions, such as building a pinsetter and transmitting pinsetter events to subscribers. It replicates all pin activities, such as the number of pins down, the number of pins left, and then resets them at the conclusion of each throw.
14.	PinSetterView	After each throw, this gui displays the status of the pins.
15.	ScoreCount	Calculates score of different ball strikes
16.	ScoreReport	It generates the score report buffer, and sends it to the user
17.	ScoreHistoryFile	It acts as a score database.

5. ANALYSIS OF ORIGINAL CODE DESIGN

We began our analysis by creating the UML diagram of the whole codebase , so that we get to know the interdependence of all the classes. So , after analysis we summarize the strength and weakness of the original design below :

5.1 STRENGTHS

- One of the major strengths about this project is that it uses **MVC pattern**. The view classes communicate with its respective model class and vice versa.
- **Proper Comments**: The whole code was properly commented, and practically every component of the system's purpose and operation were detailed.
- **Low Coupling**: The overall system had a low coupling metric, as did the subsystems.
- **Design Patterns**: A few design patterns can be observed in the code, such as the **Observer Pattern** and the **Adapter Pattern**. The Observer Pattern may be considered as a subscription-based configuration in which

various event-based states are broadcast to all subscriber objects. Additionally, the Printable Text Class module may identify Adapter Pattern during file database transactions.

5.2 WEAKNESSES

Describing some code smells and anti patterns:

- **Large class:** some classes like *Lane* class, methods can be potentially grouped into a separate class.
- **Long methods :** In some classes like *Lane.java*, *AddPartyView.java*, *PinSetterView.java*, *LaneStatusView.java* there are huge constructors and/or functions that can be split into separate functions.
- **Dead Code:** The original codebase had a lot of code that was not being used anywhere in the system. There were also multiple main methods in the system, which weren't being called/used anywhere and needed to be removed. Example of such classes is *ScoreReport.java*, *AddPartyView.java*, *Lane.java*, *LaneServer.java*
- **Duplicate Code :** Similar kind of job was being done in some methods by simply copy-pasting the previous code. This can be avoided by making methods and calling the method in the time of need to avoid repetition of code.
- **Lazy/Freeloader Class:** classes like *PinSetterEvent.java*, *Queue.java* and others Custom observer interfaces and classes were built for each type of observer, which might have been avoided if a single observer interface or the one given by Java had been used instead.
- The content of design document was very little or no implementation details.

5.3 FIDELITY TO DESIGN DOCUMENT

The original code has mostly followed the design documentation, which lists the characteristics that must be implemented.

- Each lane has its own scoring station that displays the bowlers' names as well as a visual depiction of their scores.
- The pinsetter interface transmits to the scoring station the pins that are standing after a bowler has completed a throw.
- A customizable display option will allow the operator to examine the score of a single scoring station or numerous scoring stations.

- This display feature is not visible on the window panel, lowering the design integrity.
- The operator of the control desk may view the results of any active lane.

5.4 DESIGN PATTERNS USED

- **Singleton Pattern**

Singleton is a creational design pattern that allows you to assure that a class has only one instance while also offering a global access point to this instance.

The drive class, which serves as the core function of this game, is created just once during its lifespan.

- **Proxy Pattern**

Proxy is a structural design pattern that allows you to give a replacement or placeholder for another object. A proxy manages access to the actual object, enabling you to do anything before or after the request reaches the real object.











Implemented within Lane so that LaneStatusView is able to generate the end-of-game routine originally found in Lane.

5.5 METRIC ANALYSIS



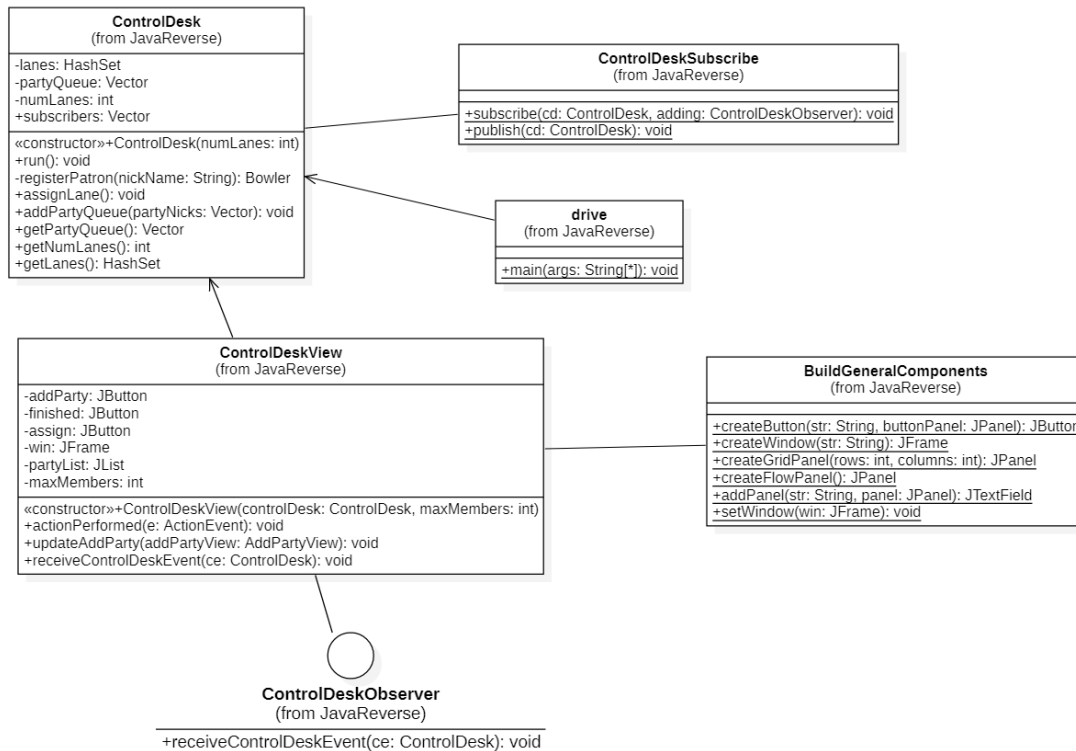
List of all classes (#29)

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	Lane	<div></div>	<div></div>	<div></div>	<div></div>	227	medium-high	low-medium	medium-high	low-medium
2	ControlDeskView	<div></div>	<div></div>	<div></div>	<div></div>	87	low-medium	low-medium	low-medium	low-medium
3	ControlDesk	<div></div>	<div></div>	<div></div>	<div></div>	68	low-medium	low-medium	medium-high	low-medium
4	LaneStatusView	<div></div>	<div></div>	<div></div>	<div></div>	93	low	low-medium	low-medium	low-medium
5	LaneView	<div></div>	<div></div>	<div></div>	<div></div>	140	low-medium	low	low-medium	low-medium
6	AddPartyView	<div></div>	<div></div>	<div></div>	<div></div>	127	low-medium	low	low-medium	low-medium
7	PinSetterView	<div></div>	<div></div>	<div></div>	<div></div>	111	low	low	low	low-medium
8	NewPatronView	<div></div>	<div></div>	<div></div>	<div></div>	85	low	low	low	low-medium
9	EndGameReport	<div></div>	<div></div>	<div></div>	<div></div>	79	low	low	low-medium	low-medium

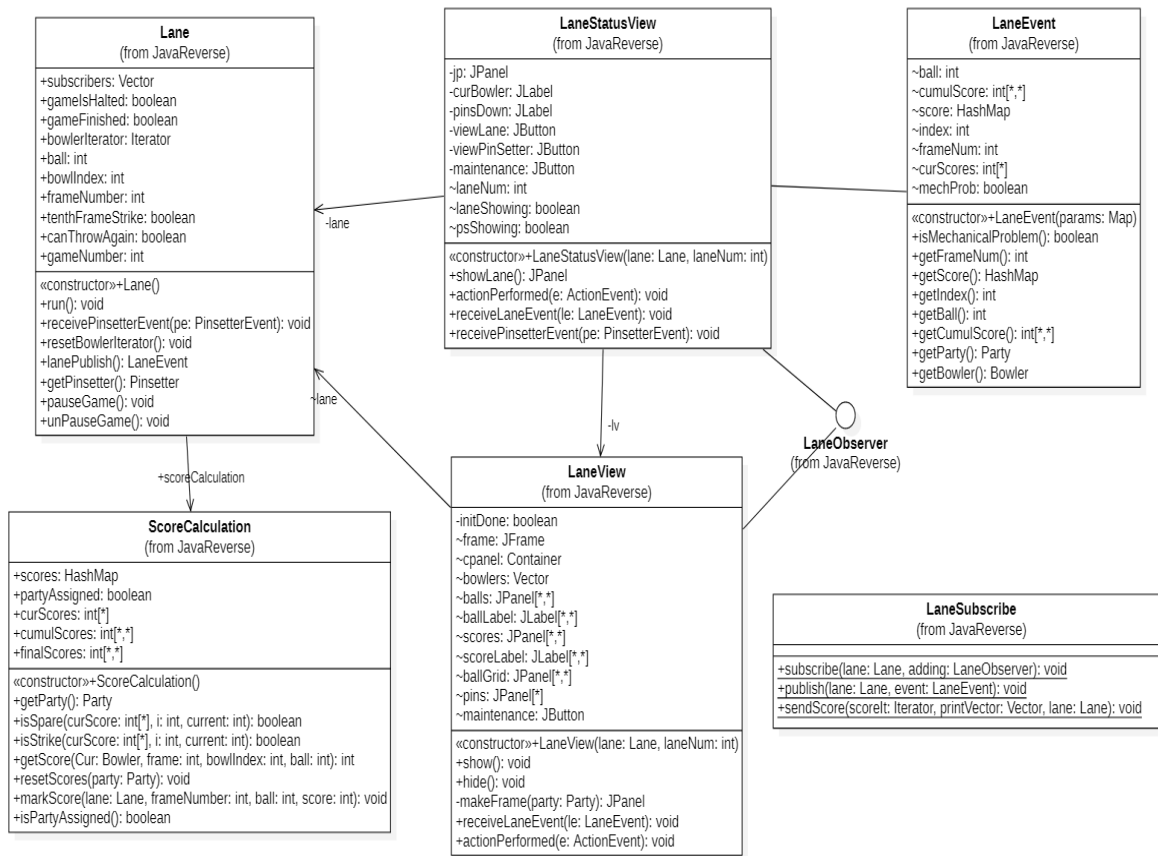
10	ScoreReport					76	low	low	low	low-medium
11	EndGamePrompt					55	low	low	low	low-medium
12	Pinsetter					47	low	low	low	low
13	LaneEvent					41	low	low	medium-high	low
14	BowlerFile					38	low	low	low	low
15	PinsetterEvent					26	low	low	low	low
16	Bowler					25	low	low	low	low
17	PrintableText					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

6. UML CLASS DIAGRAM BEFORE ENHANCEMENT

- **ControlDesk Refactored Classes:** UML Class Diagram for all the refactored classes and interfaces dealing with Control Desk, ControlDesk View & ControlDeskEvent.

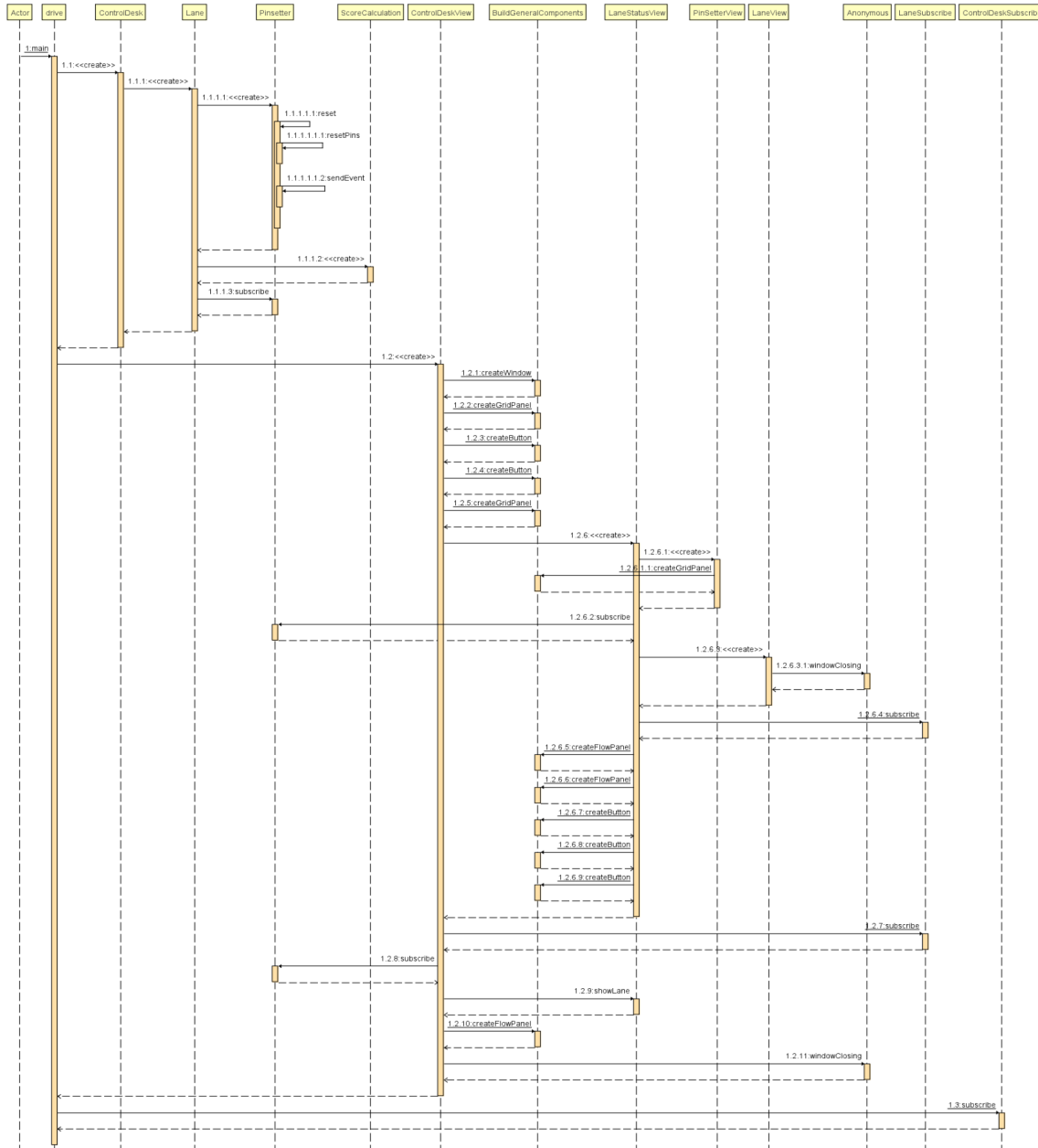


- **Lane Refactored Classes:** UML Class Diagram for all the refactored classes and interfaces dealing with Lane, LaneView & LaneEvent.

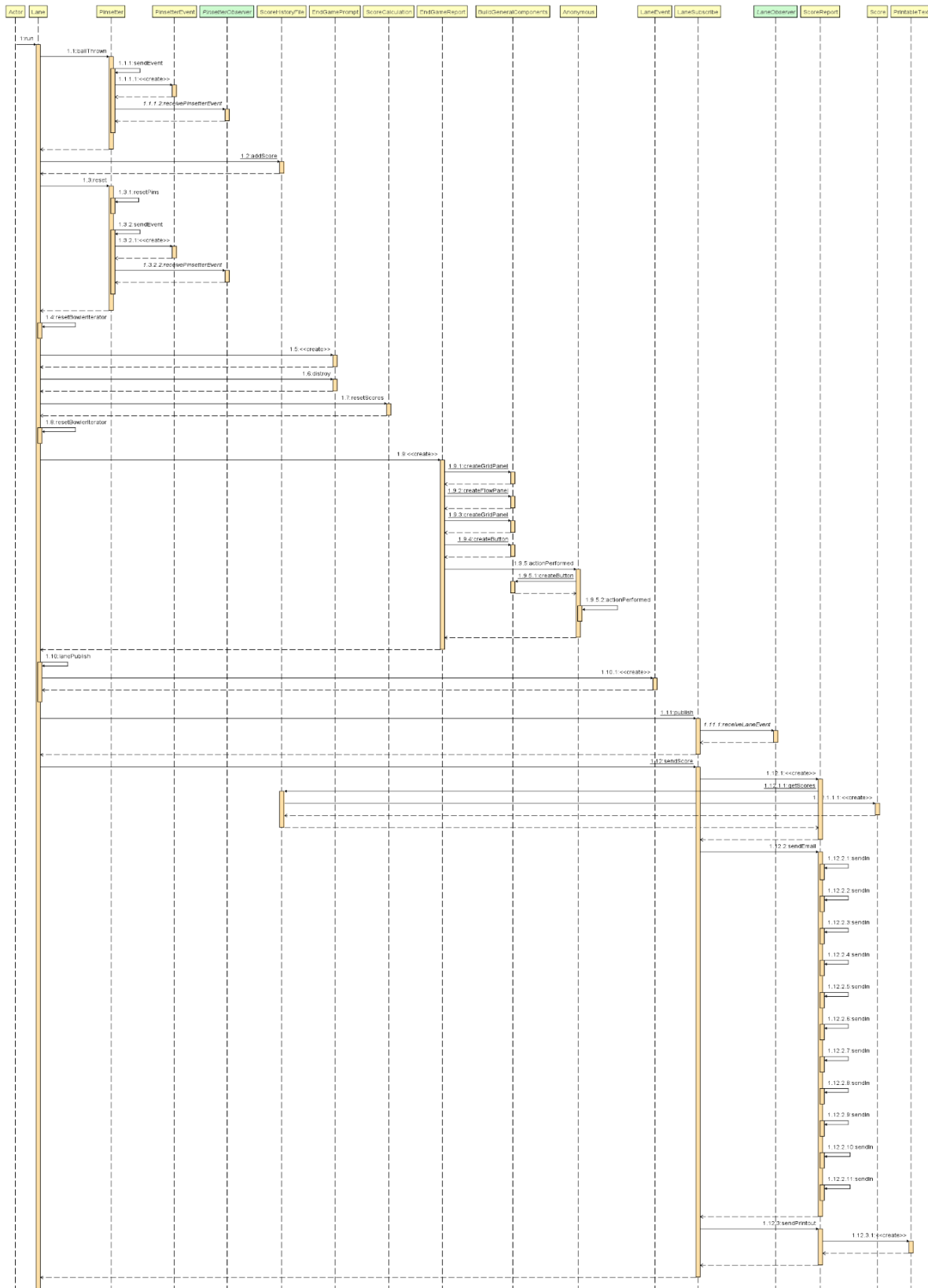


7. UML SEQUENCE DIAGRAM BEFORE ENHANCEMENT:

Drive Class:



Lane Class:

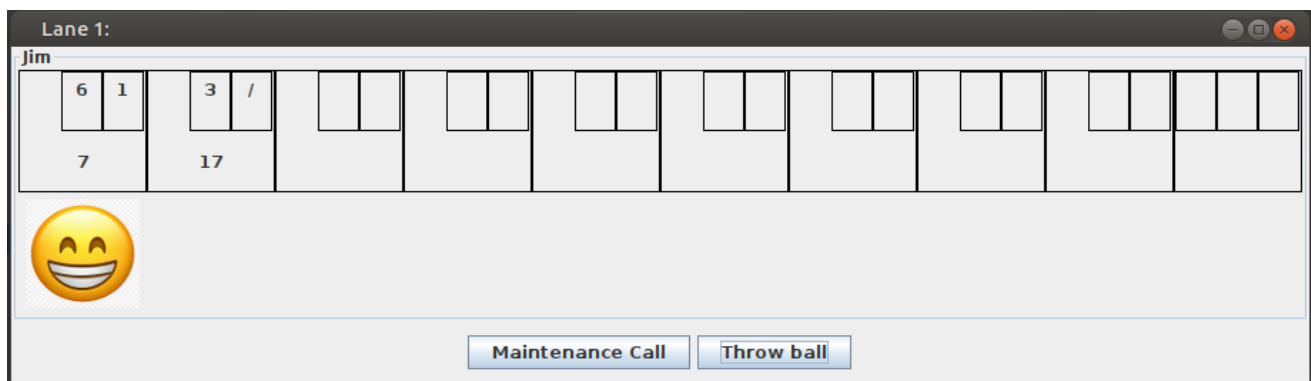


8. NEW FUNCTIONALTIES ADDED

The following functionalities have been added into the new design:

8.1 Making the UI Interactive

For making the UI interactive, instead of automatically throwing the ball we have created a button called Throw ball, and when the user clicks the button then only the ball will be thrown.

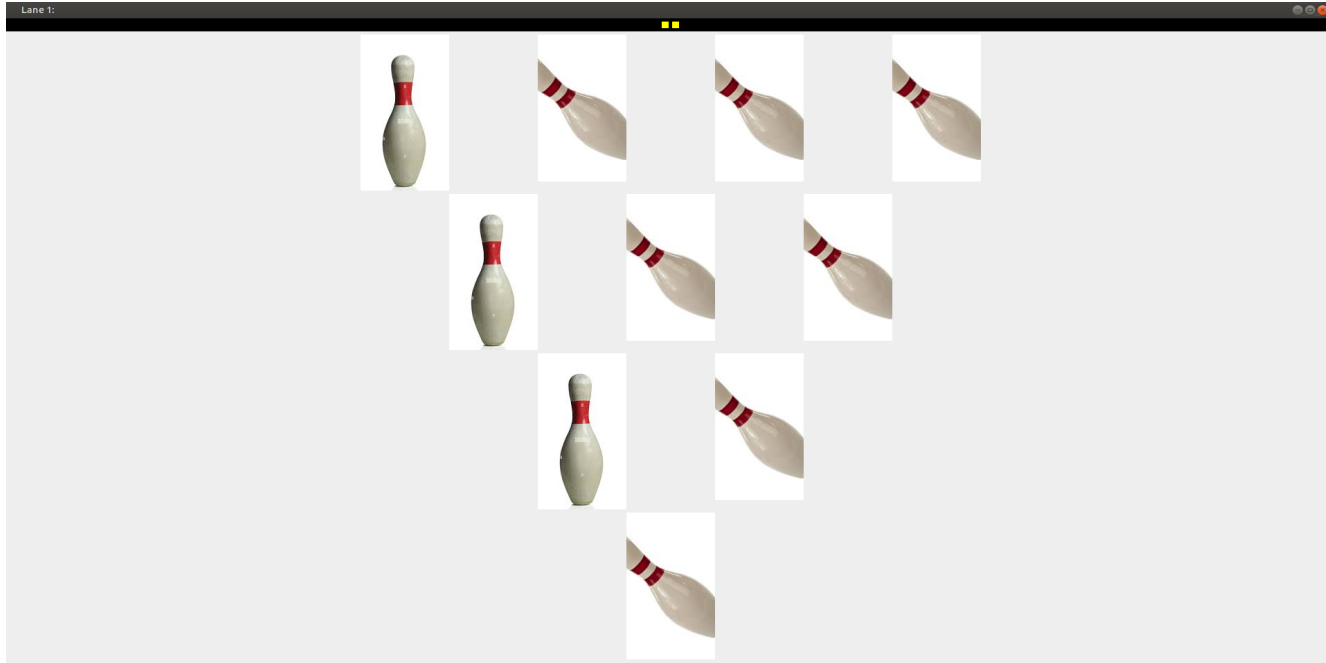


Also for making the UI creative in the pinsetter view we have changed the view and put images for the pins showing which pins are up and which pins are down. So we made the pinsetter view cooler.

```

var pic1 :ImageIcon = createImageIcon( path: "pin_s.jfif");
var pic2 :ImageIcon = createImageIcon( path: "pin_d.jfif");
if ( !(pe.isFoulCommitted()) ) {
    JLabel tempPin = new JLabel ( );
    for ( int c = 0; c < 10; c++ ) {
        boolean pin = pe.pinKnockedDown ( c );
        tempPin = (JLabel)pinVect.get ( c );
        if ( pin ) {
            tempPin.setIcon(pic2);
        }
        else {
            tempPin.setIcon(pic1);}
    }
}

```

8.2 Making the Code extensible and working for multiplayer

We created a UI which asks users for how many lanes can be present and what are the maximum numbers of patrons which can be present in each lane. This information is then passed to the ControlDeskView and initialized according to the input which was given by the user.

```

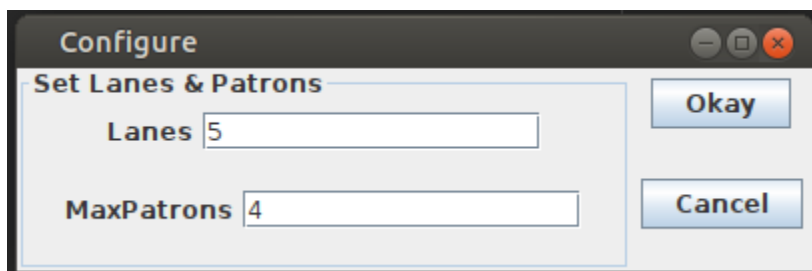
drive()
{
    wind = BuildGeneralComponents.createWindow( str: "Configure");
    JPanel configPanel = new JPanel();
    configPanel.setLayout(new BorderLayout());
    JPanel tempPanel = BuildGeneralComponents.createGridPanel( rows: 2, columns: 1);
    tempPanel.setBorder(new TitledBorder("Set Lanes & Patrons"));
    askLanes=BuildGeneralComponents.addPanel( str: "Lanes",tempPanel);
    askPatrons=BuildGeneralComponents.addPanel( str: "MaxPatrons",tempPanel);

    //button Panel
    JPanel buttonPanel = BuildGeneralComponents.createGridPanel( rows: 2, columns: 1);
    okButton = BuildGeneralComponents.createButton( str: "Okay",buttonPanel);
    okButton.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            nLanes = Integer.parseInt(askLanes.getText());
            mPatrons = Integer.parseInt(askPatrons.getText());
            wind.hide();
            int numLanes = nLanes;
            int maxPatronsPerParty=mPatrons;
            ControlDesk controlDesk = new ControlDesk(numLanes);

            ControlDeskView cdv = new ControlDeskView( controlDesk, maxPatronsPerParty);
            ControlDeskSubscribe.subscribe(controlDesk, cdv );
        }
    });

    cancelButton = BuildGeneralComponents.createButton( str: "Cancel",buttonPanel);
    cancelButton.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            wind.hide();
            exit( status: 0);
        }
    });
}
}

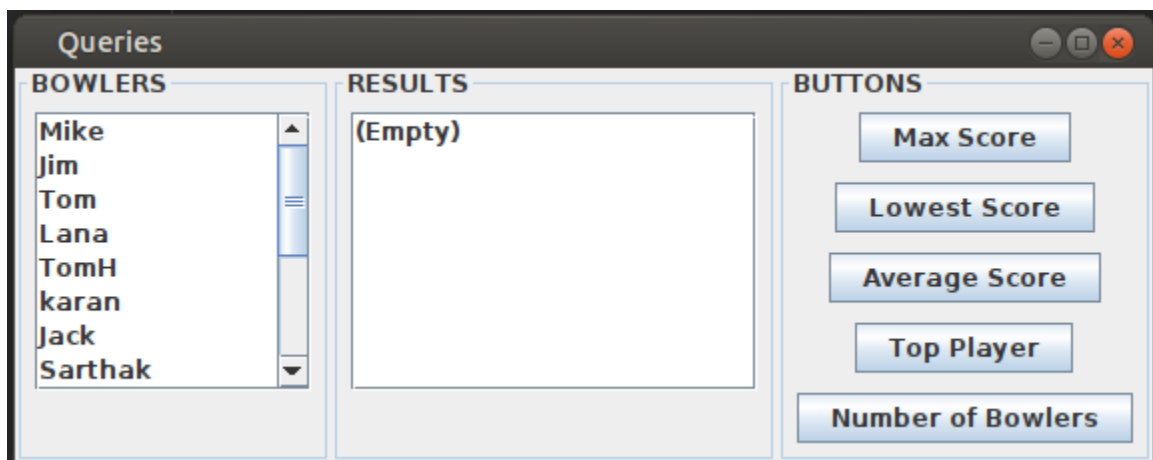
```





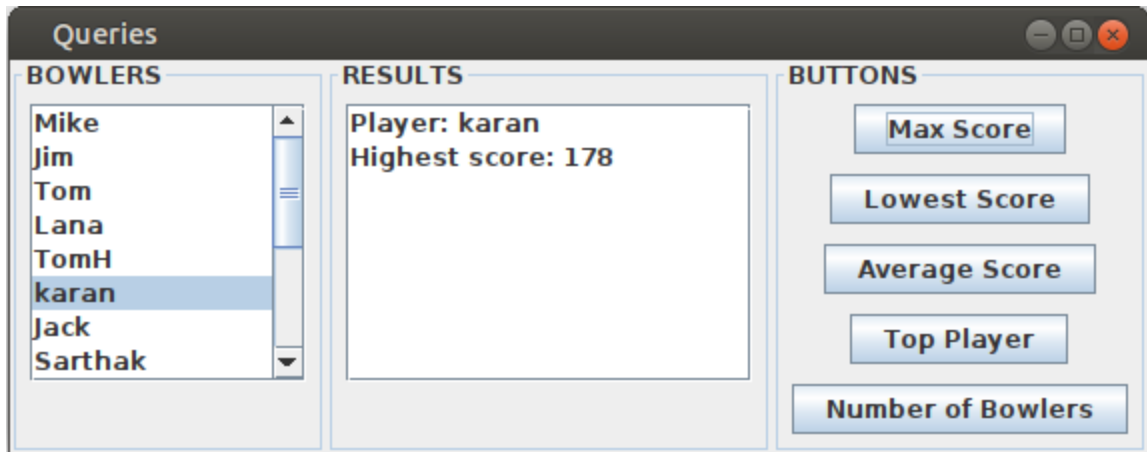
8.3 Adding the database layer and implementing the queries

The required document asked to add a database layer so that we can implement persistence and provide a searchable view for some of the ad-hoc queries. For this we have added a new button called Queries in the Controls Panel of ControlDesk which provides us with different set of queries which we can ask.



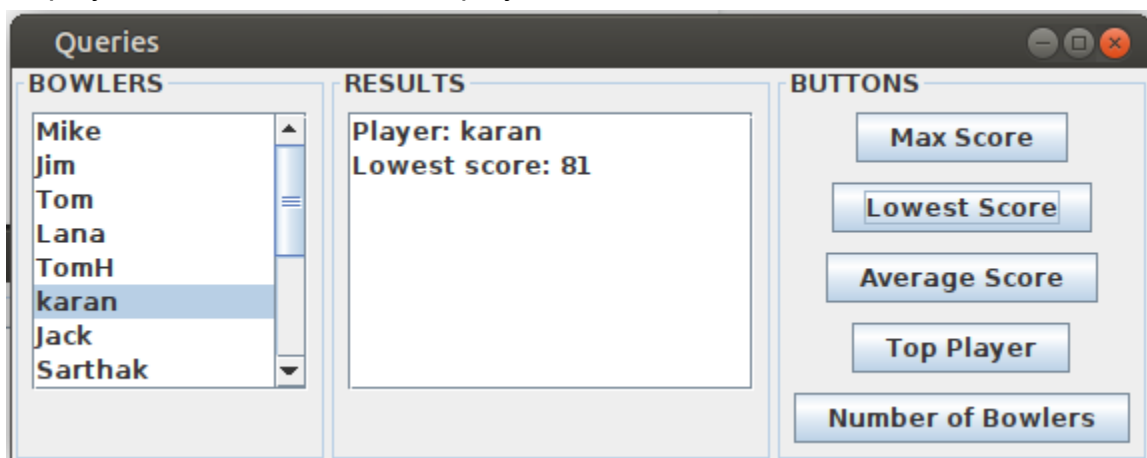
8.3.1 Max Score For Player Karan:

Displays the highest score of the player selected in the BOWLERS list.



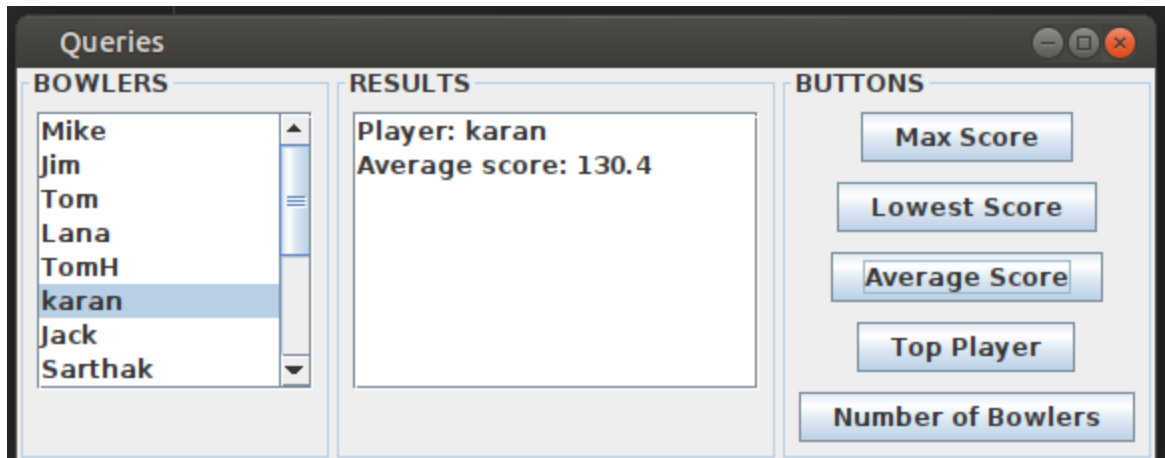
8.3.2 Lowest Score For Player Karan:

Displays the lowest score of the player selected in the BOWLERS list.



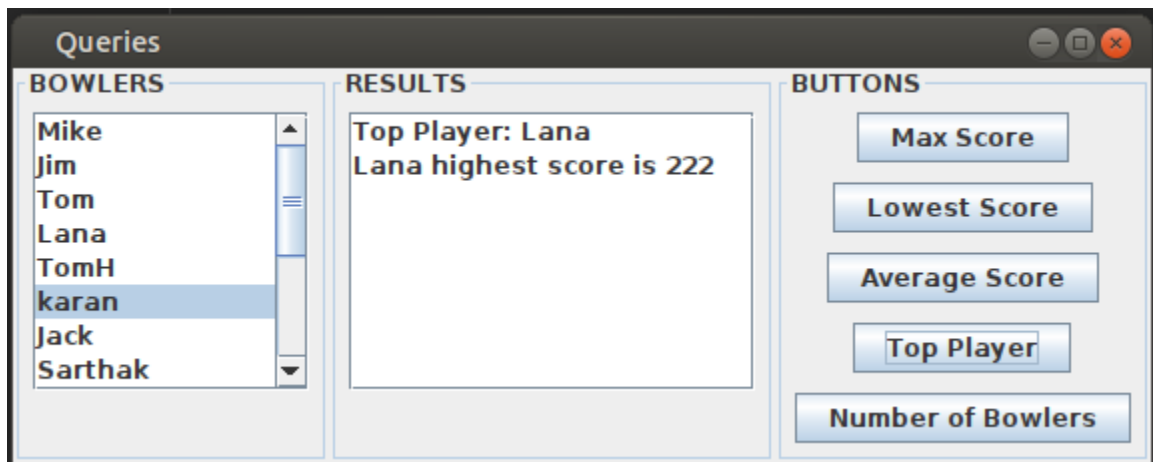
8.3.3 Average Score For Player Karan:

Displays the average score of the player selected in the BOWLERS list.



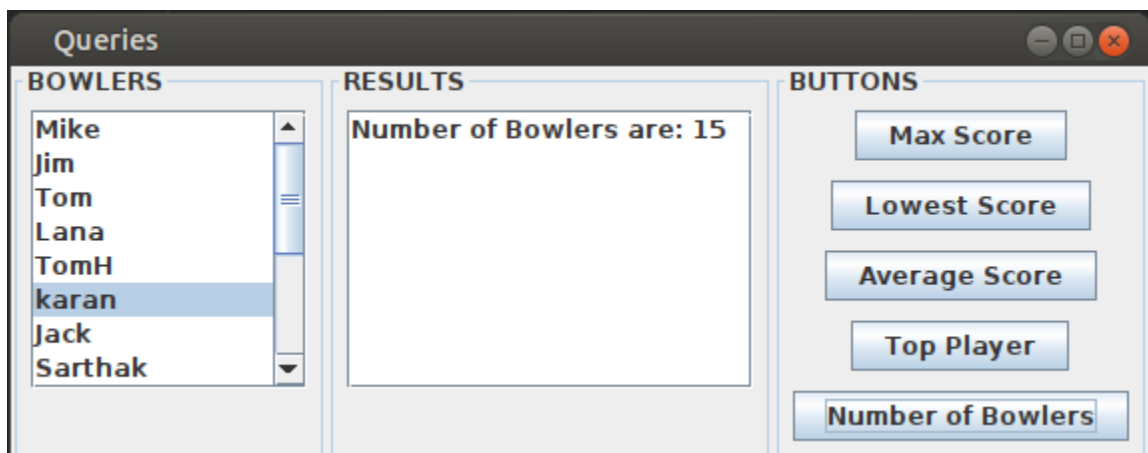
8.3.4 Top Player:

Displays the top player name along with his/her score in the results panel.



8.3.5 Number of Bowlers:

Displays the total number of bowlers registered with the Bowling Management.



8.4 Implementing penalty for gutters

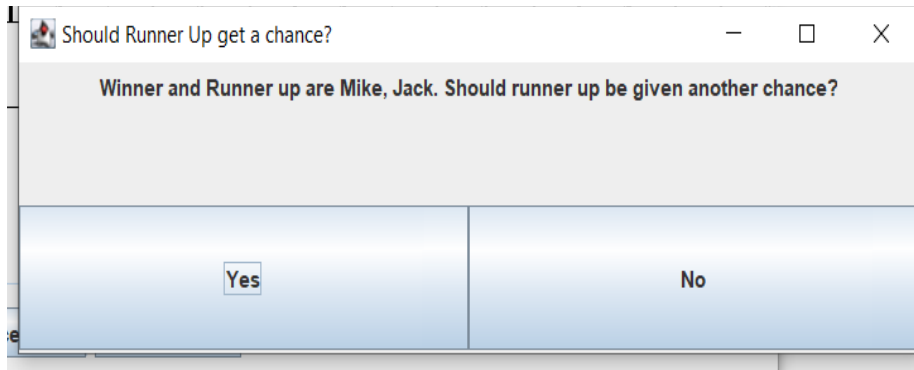
The logic for implementing the penalty for gutters can be categorized in two parts:

- If the two consecutive gutters are at the start of the game then the penalty is $\frac{1}{2}$ of the points which are scored in the next frame.
- On bowling two consecutive gutters, the player should be penalized $\frac{1}{2}$ of the highest score obtained till now.

We check for gutters if the cumulcore after a frame(1 frame=2 throws) remains same then both the throws were gutter, and then we check if it is at the start of the game else do the the necessary computation based on whether it is at the start of the game or at the middle.

```
if(i==2 && curScore[0]==0 && curScore[1]==0)
{
    System.out.println("Penalty for Gutters: " + curScore[2]/2);
    cumulScores[bowlIndex][(i / 2)] -= curScore[2]/2;
}
else
{
    int highScore = Integer.MIN_VALUE;
    System.out.println("Penalty for Gutter");
    for (int j=0; j<10; ++j){
        if (cumulScores[bowlIndex][j] > highScore) {
            highScore = cumulScores[bowlIndex][j];
        }
    }
    cumulScores[bowlIndex][(i / 2)] -= highScore/2;
}
```

8.5 Declaring the winner and providing extra frames



```

if (scoreCalculation.party.getPartySize() >= 2 && give_chance == false) {
    Vector<Bowler> top_2_players = scoreCalculation.get_Top2_players();
    first = top_2_players.firstElement();
    second = top_2_players.lastElement();

    ChanceToRunnerUpPrompt chance = new ChanceToRunnerUpPrompt(((Bowler) scoreCalculation.party.getMembers().get(0)), ((Bowler) scoreCalculation.party.getMembers().get(1)));
    int result = chance.getResult();
    chance.destroy();

    System.out.println("result was: " + result);

    if (result == 1) {
        // yes, Give second highest 1 chance,
        System.out.println("Giving chance\n");

        scoreCalculation.resetScores(scoreCalculation.party);
        gameFinished = false;
        gameNumber -= 1; // Reverting the gameNumber++ from checkGameFinished
        frameNumber = 0;
        give_chance = true;
        temp = true;
        scoreCalculation.party.resetBowlers(lane: this, top_2_players);
    }
}

if (!temp) {
    EndGamePrompt end = new EndGamePrompt("partyName: " + ((Bowler) scoreCalculation.party.getMembers().get(0)).getName() + " and " + ((Bowler) scoreCalculation.party.getMembers().get(1)).getName());
    end.show();
}

```

As per the requirement, at the end of 10 frames, a prompt is shown asking if runner up that is the second highest player should be provided a second chance or not. If chosen yes, 3 additional frames are provided between the winner and the runner up till the winner is finalized and in case of a tie, the player with most strikes is declared the winner.

Approach:

Cumulative scores of players is taken after their 10 chances and from that, the highest scorer and the second highest scorer are found and stored.

After that it is prompted to ask if runner up is to be given a second chance and if yes then 3 more chances are given to both the players and their cumulative scores are updated and then again they are compared to declare the winner with the higher score.

In case of a tie after the 3 chances, strikes of the two players are stored and compared and the player with the larger number of strikes is declared the winner.

8.6 Implementing emoticon

We implemented the emoticon based on the score of the player like if a player gets a strike then a star emoji face is shown , for spare throws we use happy emoji ,else if the throw is a normal throw then it will be okay emoji and for gutters we use sad emoji. For winning we have a star emoji,etc.

Lane 1:

Tom

X		9	/	3	1	X											
20	33	37	47														

🌟

Lana

4	3	6	3	6	/	1	0										
7	16	27	28														

😊

karan

X		4	0	4	0	2	/										
14	18	22	32														

😄

Maintenance Call Throw ball

As we can see for a strike Tom reacts with a star emoji ,for a normal throw Lana reacts with an okay emoji and for a spare throw karan reacts with a happy emoji.


```

void setEmojiIconOnLabel(JLabel imageLabel, String emojiName) throws IOException {
    Image emojiImg;
    emojiImg = ImageIO.read(new File("emojis/" + emojiName ));
    Image scaledImg = emojiImg.getScaledInstance(60, 60, Image.SCALE_SMOOTH);
    ImageIcon emojiImgIcon = new ImageIcon(scaledImg);
    imageLabel.setIcon(emojiImgIcon);
}

```

```

try {
    JPanel emojiPanel = new JPanel();
    JLabel emojiLabel = new JLabel();
    setEmojiIconOnLabel(emojiLabel, emojiName: "happy.jpg");
    emojiLabels[i]=emojiLabel;
    emojiPanel.add(emojiLabel);
    pins[i].add(emojiPanel, BorderLayout.EAST);
} catch (IOException e) {
    e.printStackTrace();
}
panel.add(pins[i]);

```

```

for (int i = 0; i < 21; i++) {
    if (((int[]) ((HashMap) le.getScore()))
        .get(bowlers.get(k))[i]
        != -1)
        if (((int[]) ((HashMap) le.getScore()))
            .get(bowlers.get(k))[i]
            == 10
            && (i % 2 == 0 || i == 19))
            //happy face
            {
                ballLabel[k][i].setText("X");
                try {
                    setEmojiIconOnLabel(emojiLabels[k], emojiName: "star.png");
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
}

```

```

else if (
    i > 0
        && ((int[]) ((HashMap) le.getScore())
            .get(bowlers.get(k)))[i]
        + ((int[]) ((HashMap) le.getScore())
            .get(bowlers.get(k)))[i
            - 1]
        == 10
        && i % 2 == 1)
    //okay face
{
    ballLabel[k][i].setText("/");
    try {
        setEmojiIconOnLabel(emojiLabels[k], emojiName: "happy.jpg");
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

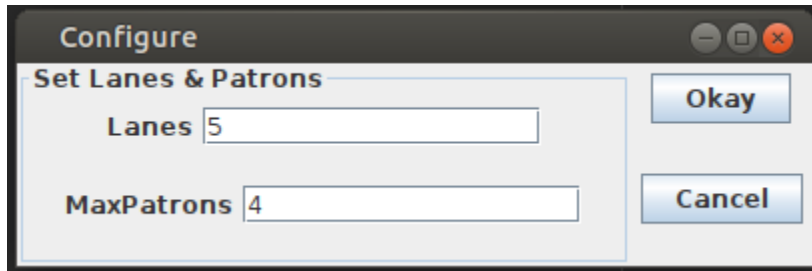
```

else if ( ((int[])((HashMap) le.getScore()).get(bowlers.get(k)))[i] == -2 )
    //sad face
{
    ballLabel[k][i].setText("F");
    try {
        setEmojiIconOnLabel(emojiLabels[k], emojiName: "cry.png");
    } catch (IOException e) {
        e.printStackTrace();
    }
} else
    //shook face
{
    ballLabel[k][i].setText(
        (new Integer(((int[]) ((HashMap) le.getScore())
            .get(bowlers.get(k)))[i]))
            .toString());
    try {
        setEmojiIconOnLabel(emojiLabels[k], emojiName: "confused.png");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

8.7 Making the code configurable

We have made the number quotes configurable as the user can configure the maxpatrons and numLanes according to him by using the UI interface which we provide him.



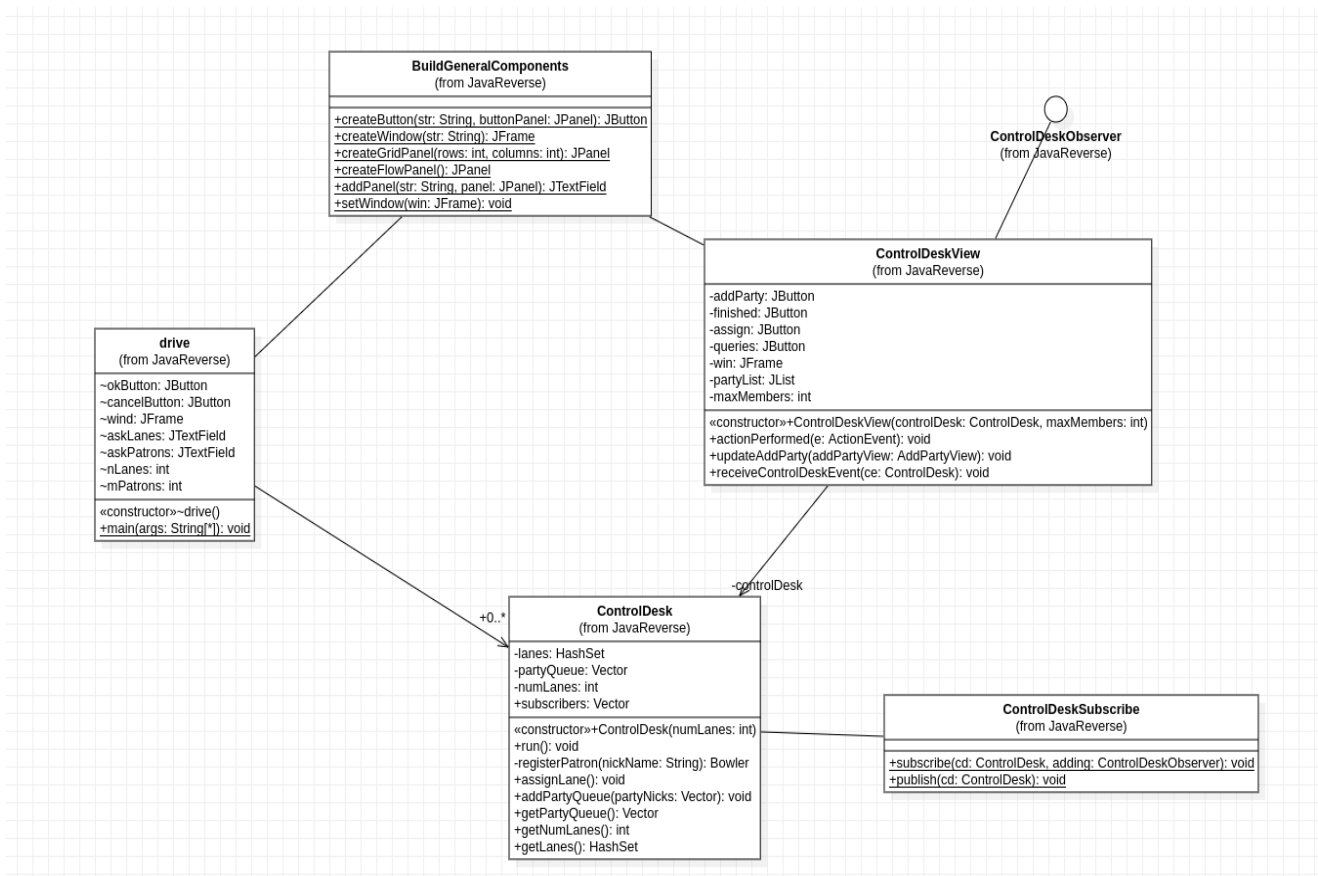
9. DESCRIPTION OF MODIFIED CLASSES

S.No	CLASS NAME	RESPONSIBILITIES
1.	Laneview	Added Button for throw ball functionality. Added functionalities to show emoticon as per the score of the player.
2.	QueryView	We created this class whose responsibility is to provide a searchable view to the users for the adhoc queries such as Top player, Highest,lowest and average score of a player,etc.
3.	ScoreCalculation	The ScoreCalculation class whose initial responsibility was to calculate the scores is added with more functionality in order to calculate the penalty for gutters and to calculate and find the top two scorers.
4.	ScoreHistoryFile	More functionalities were added to ScoreHistoryFile. Apart from functions which return the scores, ScoreHistoryFile provides functions which give the maximum and lowest score of a particular player and function which returns the player which has the maximum score.

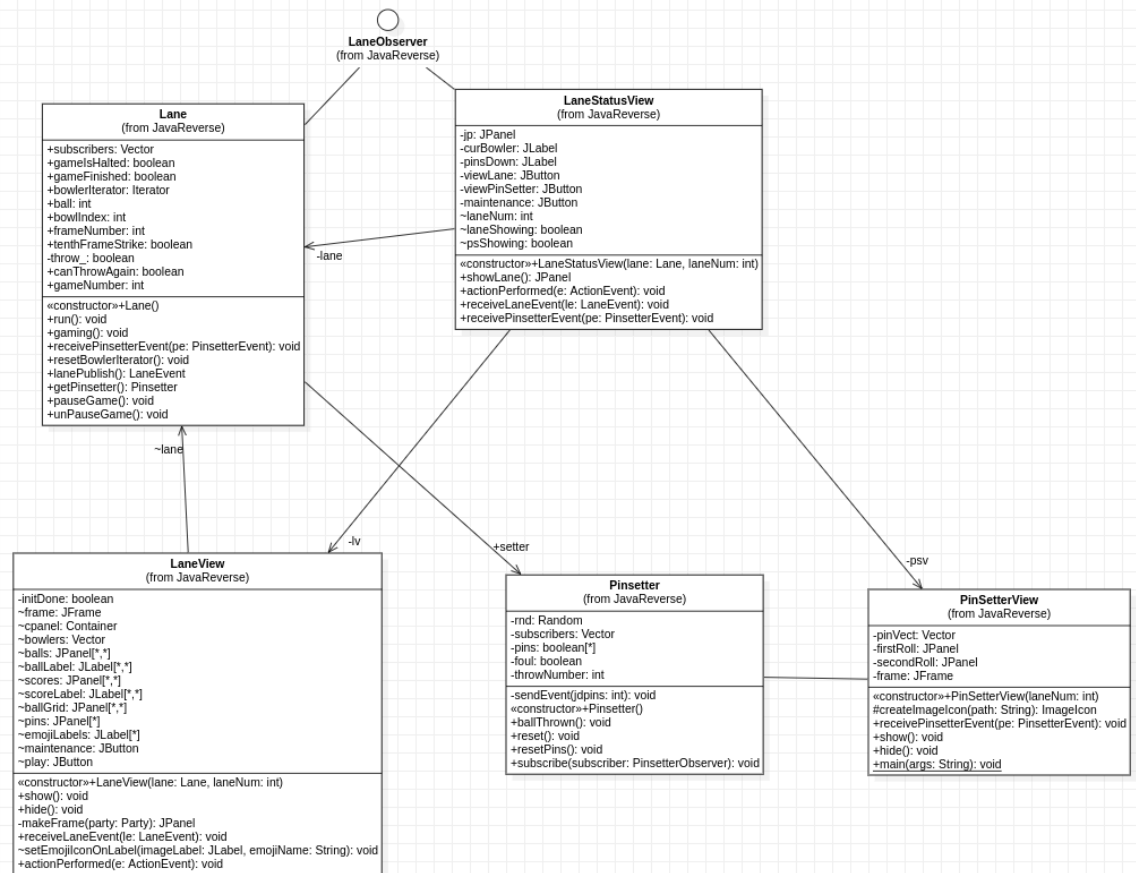
5.	controlDeskView	ControlDeskView provides an additional functionality called queries in the Controls Panel which directs you to QueryView where we can get answers for some predefined queries.
7.	PinSetterView	The PinSetterView now instead of providing the dot view of pins provides a view using the images of pins, uses different images for up and down pins.
8.	Lane	Added functionality to throw ball only when clicked on throw ball button. Added functionality to provide the runner up, a second chance at winning.
9.	ChanceToRunnerUpPrompt	Added panels and buttons for the functionality of providing runner up a second chance at winning.

10. UML CLASS DIAGRAM AFTER ENHANCEMENT

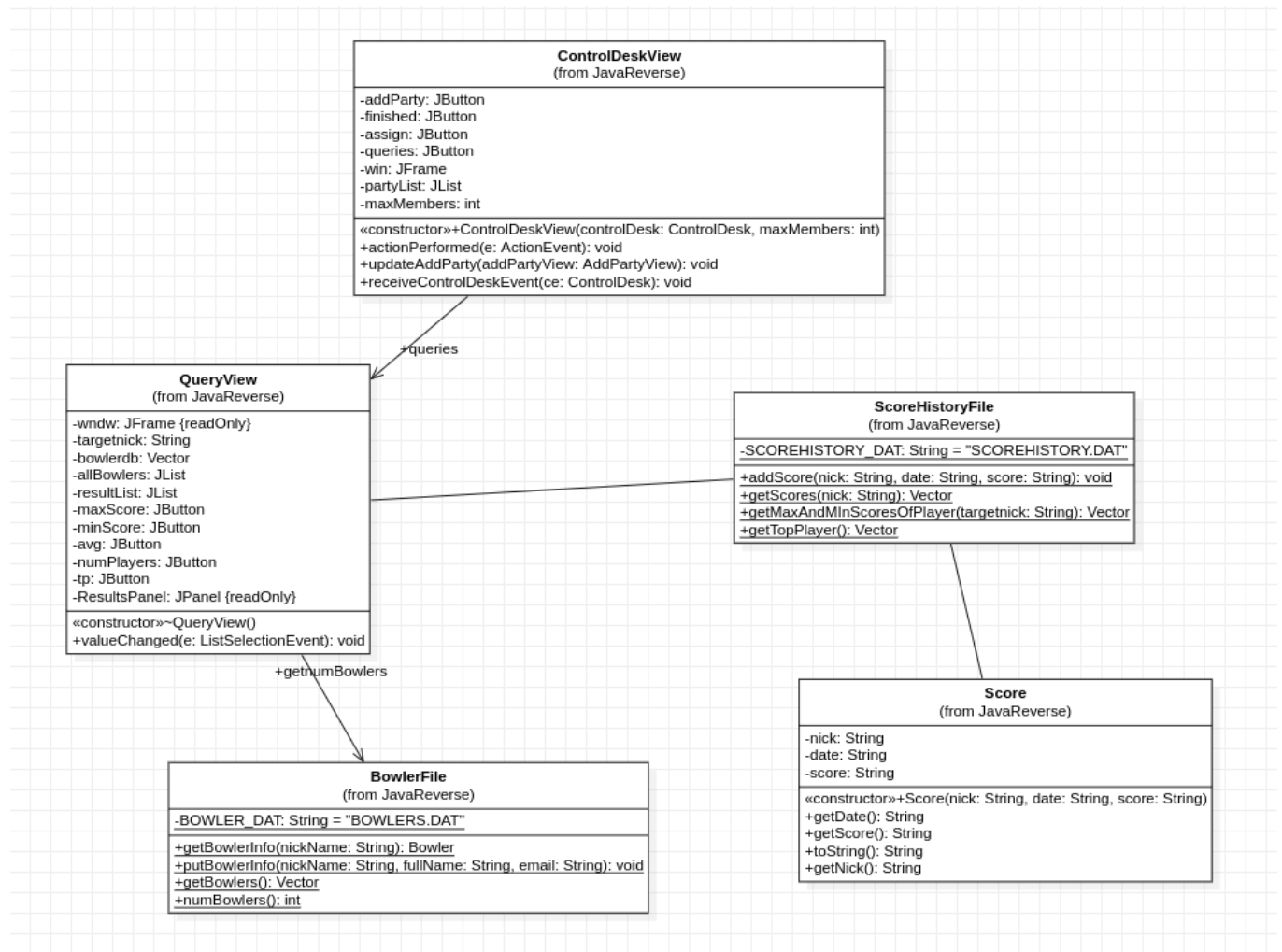
- ControlDesk Enhanced Classes:



- Lane Enhanced Classes:

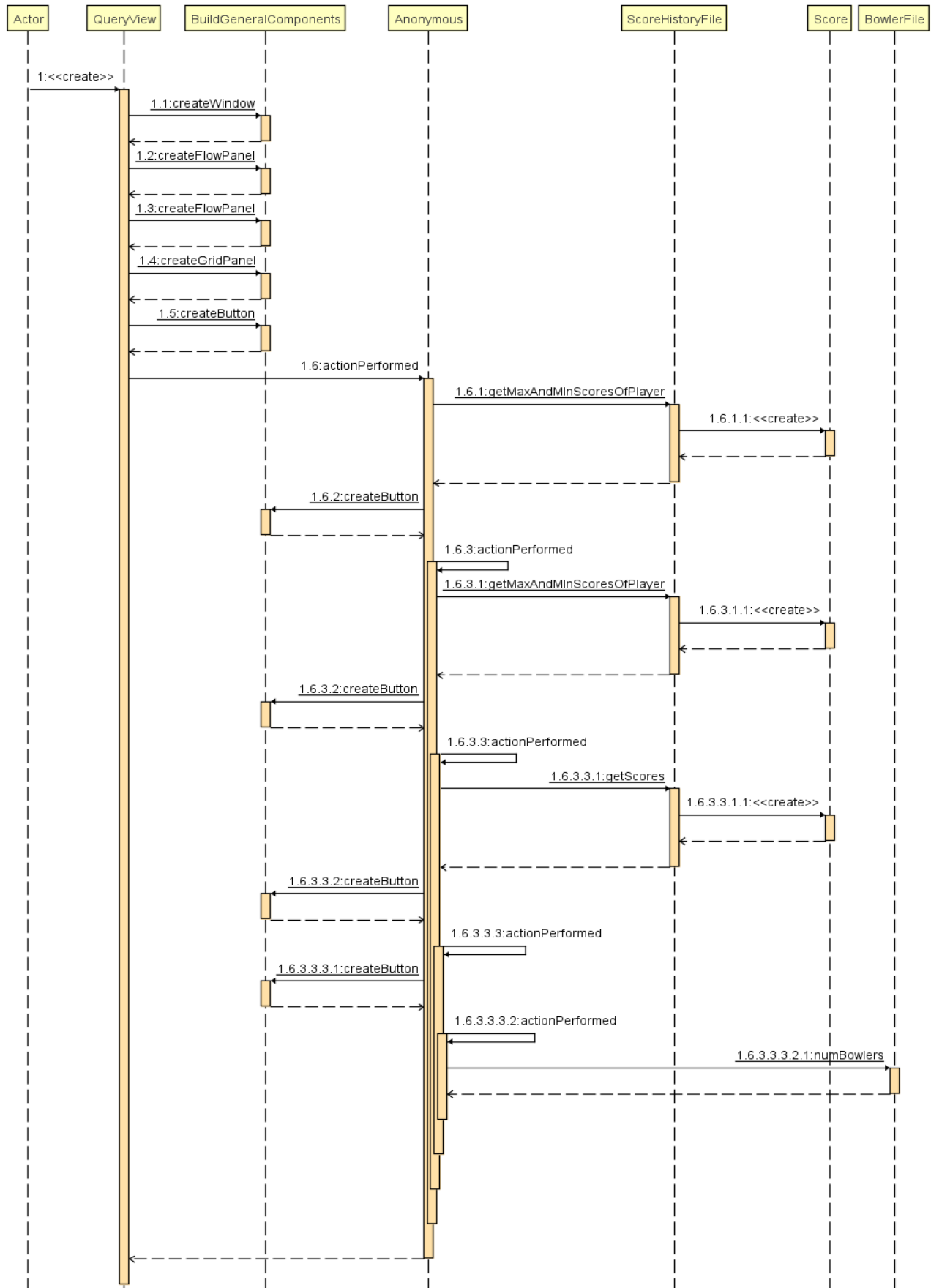


- **QueryView Classes:**

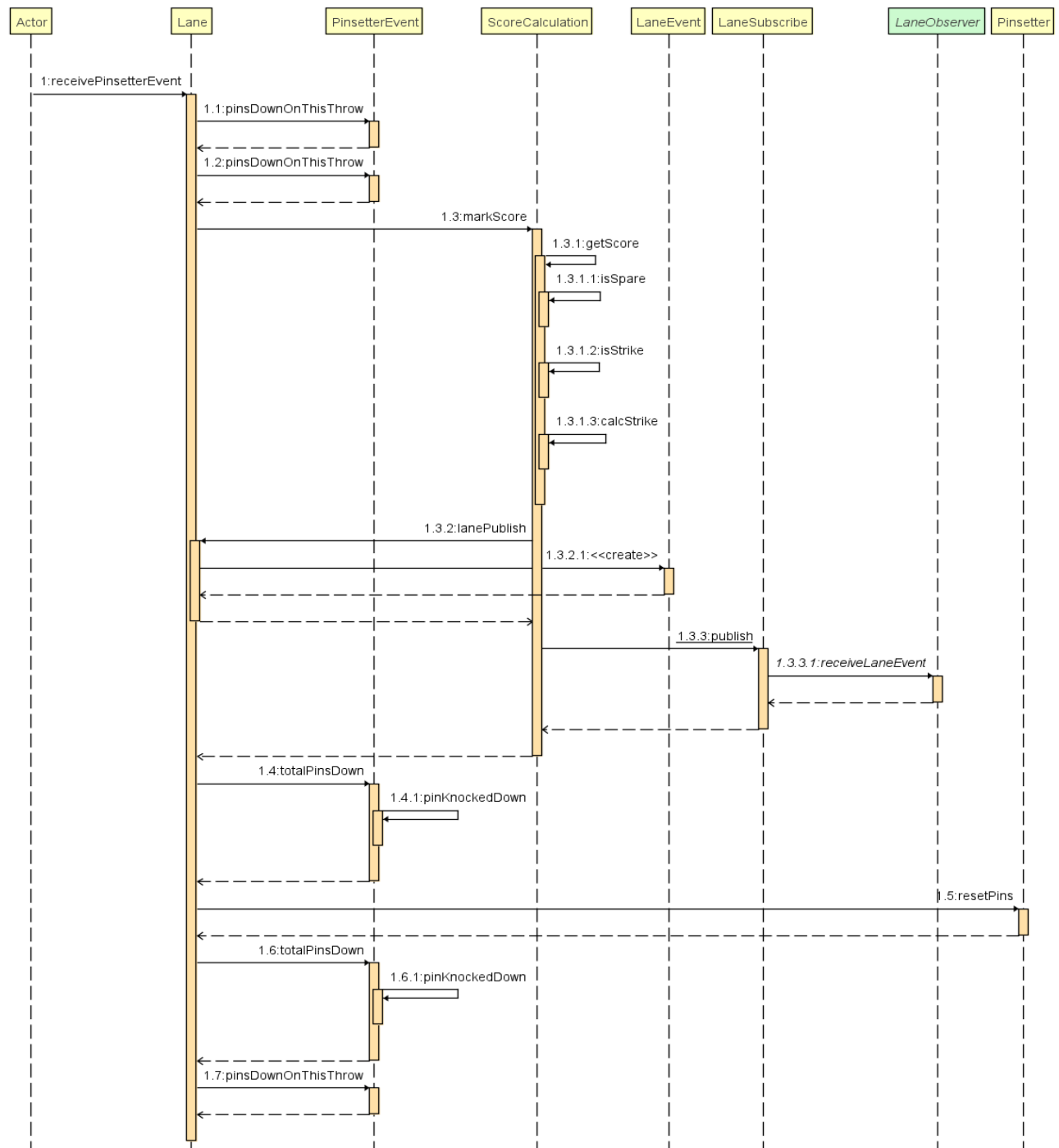




Query View Classes:



Lane Enhanced Classes:



12. CODE METRICS OF FINAL CODE

Analysis of src_unit2

General Information

Total lines of code: 1658

Number of classes: 33

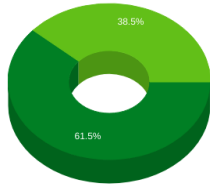
Number of packages: 1

Number of external packages: 21

Number of external classes: 101

Number of problematic classes: 0

Number of highly problematic classes: 0



Coupling

Very High

High

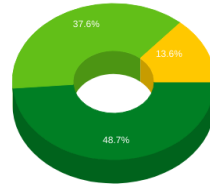
Medium-high

Low-medium

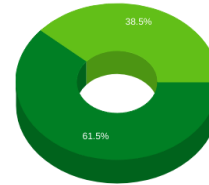
Low

Distribution of Quality Attributes

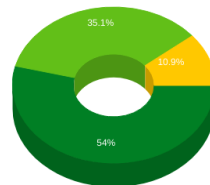
Complexity, Coupling, Cohesion, and Size



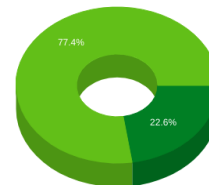
Complexity



Coupling



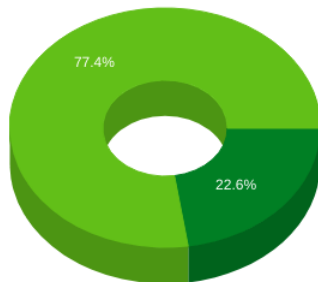
Lack of Cohesion



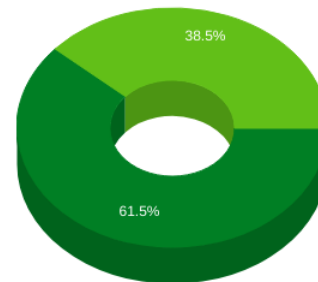
Size

Distribution of Quality Attributes

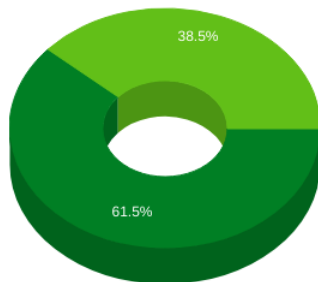
Complexity, Coupling, Cohesion, and Size



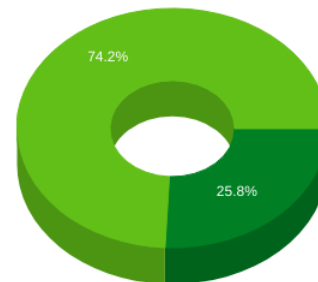
Class Lines of Code



Coupling Between Object Classes



CBO App



Class-Methods Lines of Code

List of all classes (#33)

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	Lane					226	medium-high	low-medium	low	low-medium
2	LaneView					181	low-medium	low-medium	medium-high	low-medium
3	ControlDeskView					78	low-medium	low-medium	low-medium	low-medium
4	ControlDesk					57	low-medium	low-medium	low-medium	low-medium
5	LaneStatusView					78	low	low-medium	low-medium	low-medium
6	SubscriberLane					19	low	low-medium	low	low
7	Database					104	low-medium	low	low-medium	low-medium
8	CalculateScore					102	low-medium	low	low-medium	low-medium
9	AddPartyView					102	low-medium	low	low-medium	low-medium
10	PinSetterView					111	low	low	low	low-medium
11	ScoreReport					71	low	low	low	low-medium
12	EndGameReport					61	low	low	low-medium	low-medium
13	ScoreHistoryFile					60	low	low	low	low-medium
14	NewPatronView					52	low	low	low	low-medium
15	Pinsetter					47	low	low	low	low
16	LaneEvent					42	low	low	low	low
17	EndGamePrompt					39	low	low	low	low

18	CreateElements	■	■	■	■	39	low	low	low	low
19	BowlerFile	■	■	■	■	29	low	low	low	low
20	Bowler	■	■	■	■	25	low	low	low	low
21	PinsetterEvent	■	■	■	■	24	low	low	low	low
22	PrintableText	■	■	■	■	21	low	low	low	low
23	Party	■	■	■	■	17	low	low	low	low
24	Score	■	■	■	■	16	low	low	low	low
25	Queue	■	■	■	■	12	low	low	low	low
26	SubscribeControlDesk	■	■	■	■	11	low	low	low	low
27	LaneEventInterface	■	■	■	■	10	low	low	low	low
28	drive	■	■	■	■	10	low	low	low	low
29	ControlDeskEvent	■	■	■	■	6	low	low	low	low
30	PinsetterObserver	■	■	■	■	2	low	low	low	low
31	ControlDeskObserver	■	■	■	■	2	low	low	low	low
32	LaneServer	■	■	■	■	2	low	low	low	low
33	LaneObserver	■	■	■	■	2	low	low	low	low

13. ANALYSIS OF METRIC

- Coupling: Coupling refers to the degree to which the various modules/classes depend on each other. We prefer low Coupling for our designs.

- Line of code: A measure for size. Literally means how many lines of code is present.
- Complexity: It implies the code being difficult to understand and it also describes the interactions between a number of entities. Higher levels of complexity in software increase the risk of unintentionally interfering with interactions and so increases the chance of introducing defects when making changes.
- Class-Methods line of code: Total number of all nonempty, non-commented lines of methods inside a class.
- Lack of Cohesion: Cohesion measures how well the methods of a class are related to each other. High cohesion (low lack of cohesion) tends to be preferable.
- 1-CAM: CAM metric is the measure of cohesion based on parameter types of methods. LCAM = 1-CAM
- C3: Mixture of major attributes that start with C, related Quality Attributes: Coupling, Cohesion, Complexity
- Size: Size is one of the oldest and most common forms of software measurement. Measured by the number of lines or methods in the code. A very high count might indicate that a class or method is trying to do too much work and should be split up. It might also indicate that the class might be hard to maintain.

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

- Several metrics were considered for analyzing original and refactored code like - Coupling, Cohesion, Lines of Code, Number of Children etc.
- The metrics for original and enhanced code are present in the document.
- These measurements highlighted several aspects of the original code and provided information about the code.
- The metrics of the old code demonstrate how the classes in the system are tightly connected, particularly classes like Lane. That is, attributes, methods, subclasses, local variables, and so on in the Lane class refer to many other classes, making the code more complicated and difficult to comprehend.
- Few classes' code also lacks internal modularity, which means that code from one class seems to be dispersed among many classes and methods, particularly in classes like Lane, LaneEvent, and ControlDesk.
- Some view classes had repetitive code for creating Jpanels and buttons.

2. How did you use these measurements and your understanding to guide implementation of new requirements?

- The metrics showed us various aspects of the code, and hence helped us in identifying classes where coupling is high, cohesion is low etc.
 - High Complexity suggested that we need to make the code less complicated, more readable and to decrease the interaction between separate entities , so we refactored in the same way. (Lane class)
 - Lane class was most populated, hence broke it down to subclasses.
 - Metrics helped us identify anti-patterns and after looking at those we came up with strategies to solve these issues.
1. After adding functionalities to the code most of the metrics remained same, they are either low or medium-low for most of the metrics.
 2. Complexity of the code increased to medium-high for the lane class, after adding new code to the class it became more complex.
 3. Lack of Cohesion increased to medium-high from low after adding code to laneview class.

14. CONCLUSION/SUMMARY

The experience of refactoring code in both Unit 1 and Unit 2 is incredibly enriching for the Team. As part of Unit:1, we concentrated on code refactoring; as part of the current project (Unit:2), we focused on UI while maintaining a good balance with code reworking.