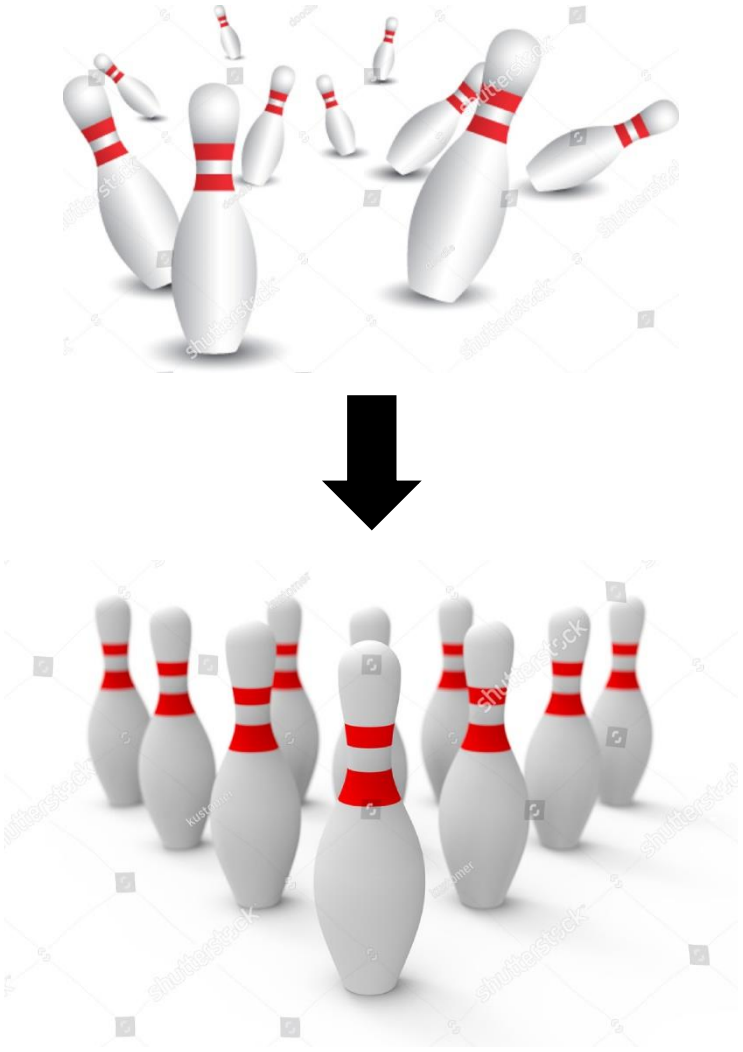
Software Engineering

Unit 2 Enhancement

Bowling Alley Simulation

Team 29



Contributors

1. Sudipta Halder (2021202011)
   - Refactored the code base and added new features.
   - 20 hours contributed to the project.

2. Sowmya Lahari Vajrala (2021202010)
   - Refactored the code base and added new features.
   - 20 hours contributed to the project.

3. Anjaneyulu Bairi (2021202008)
   - Refactored the code base and added new features.
   - 20 hours contributed to the project.

4. Josh Joy (2021204009)
   - Documentation and added new features
   - 20 hours contributed to the project.

Date Of Submission: 22-March-2021

GitHub Repository:

Link: github.com/RajaSudipta/BowlingAlleySimulation

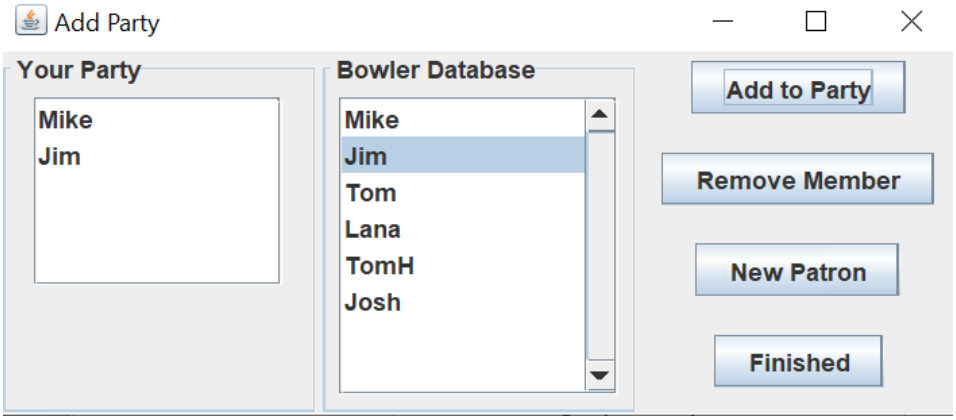
Link:https://gitlab.com/sowmyalahari.vajrala/BowlingAlleySimulation

Index

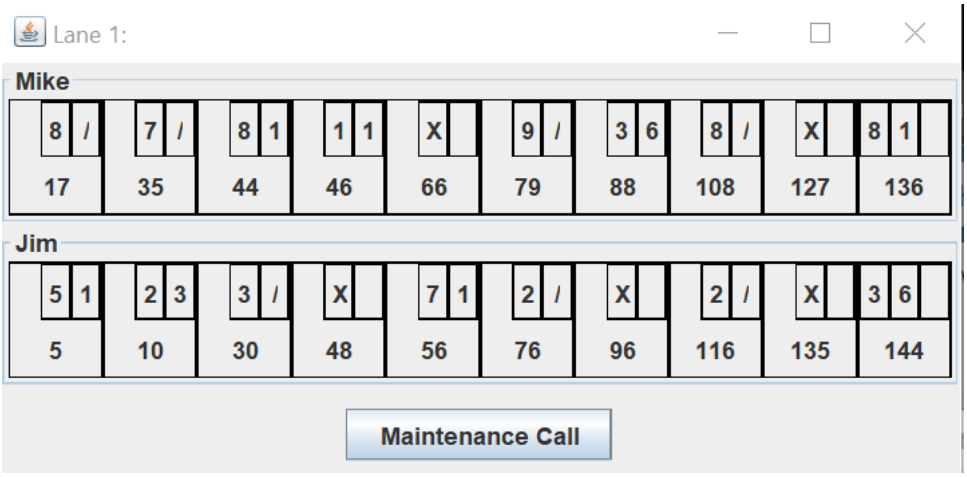
1. Introduction to Bowling Alley Simulation

The Software is a backend administrative tool for bowling alleys, this version simulates a bowling alley by adding virtual players. The Control Desk is the main window, which shows the status of the lanes and allows us to add new players.

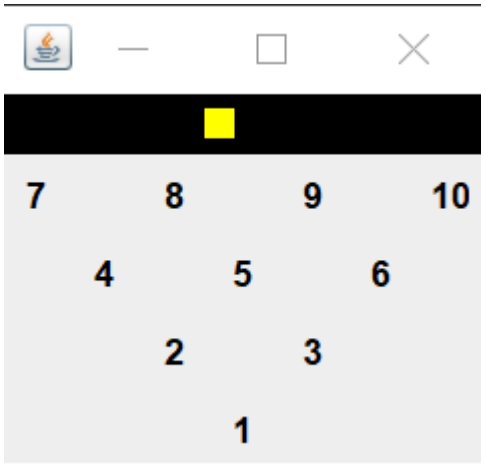
The Add Party window lets us assign existing players in the database to a lane and add new players to the database.



The Lane Window shows a live update of the player score as they bowl.

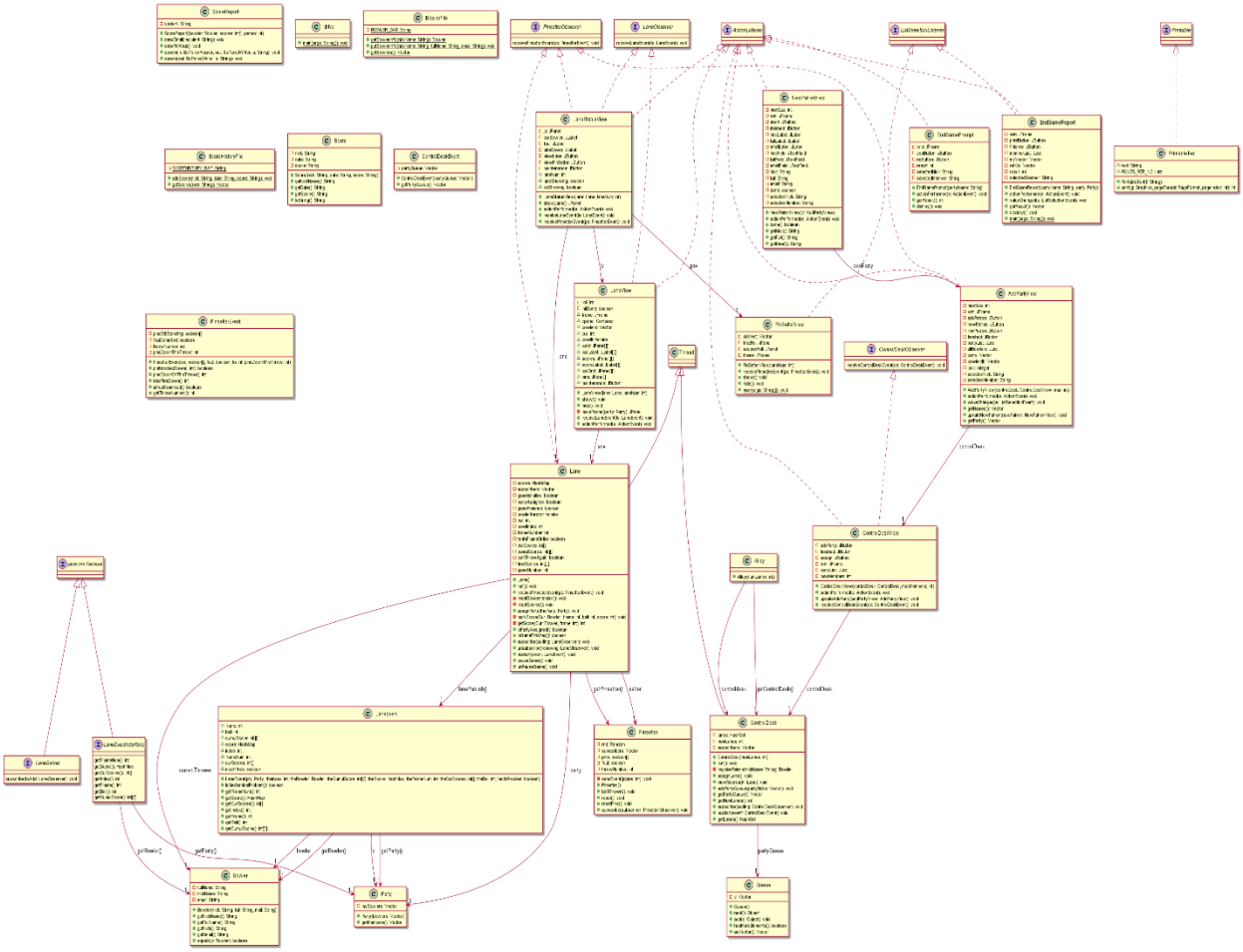


The Pin Setter Window displays whether a bowling pin has fallen down as the player bowls.



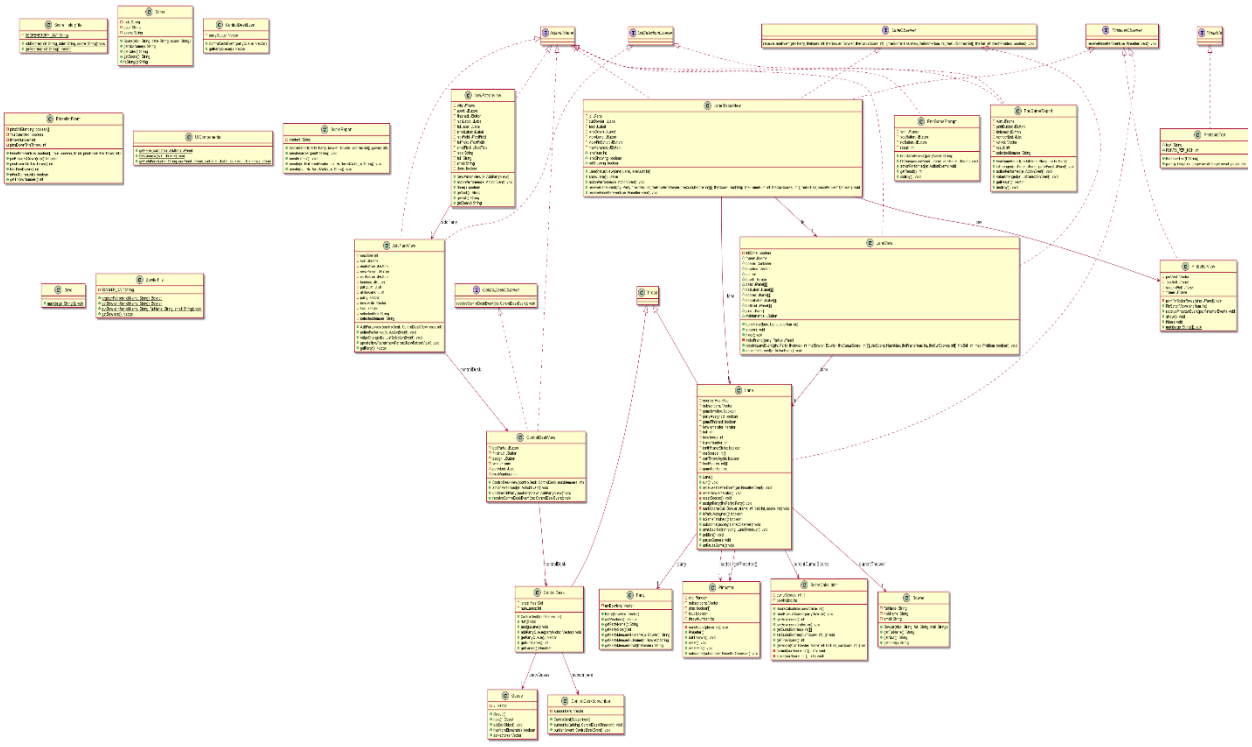
2. UML Class Diagram

Before Refactoring:



[Download the Diagram]

After Refactoring:

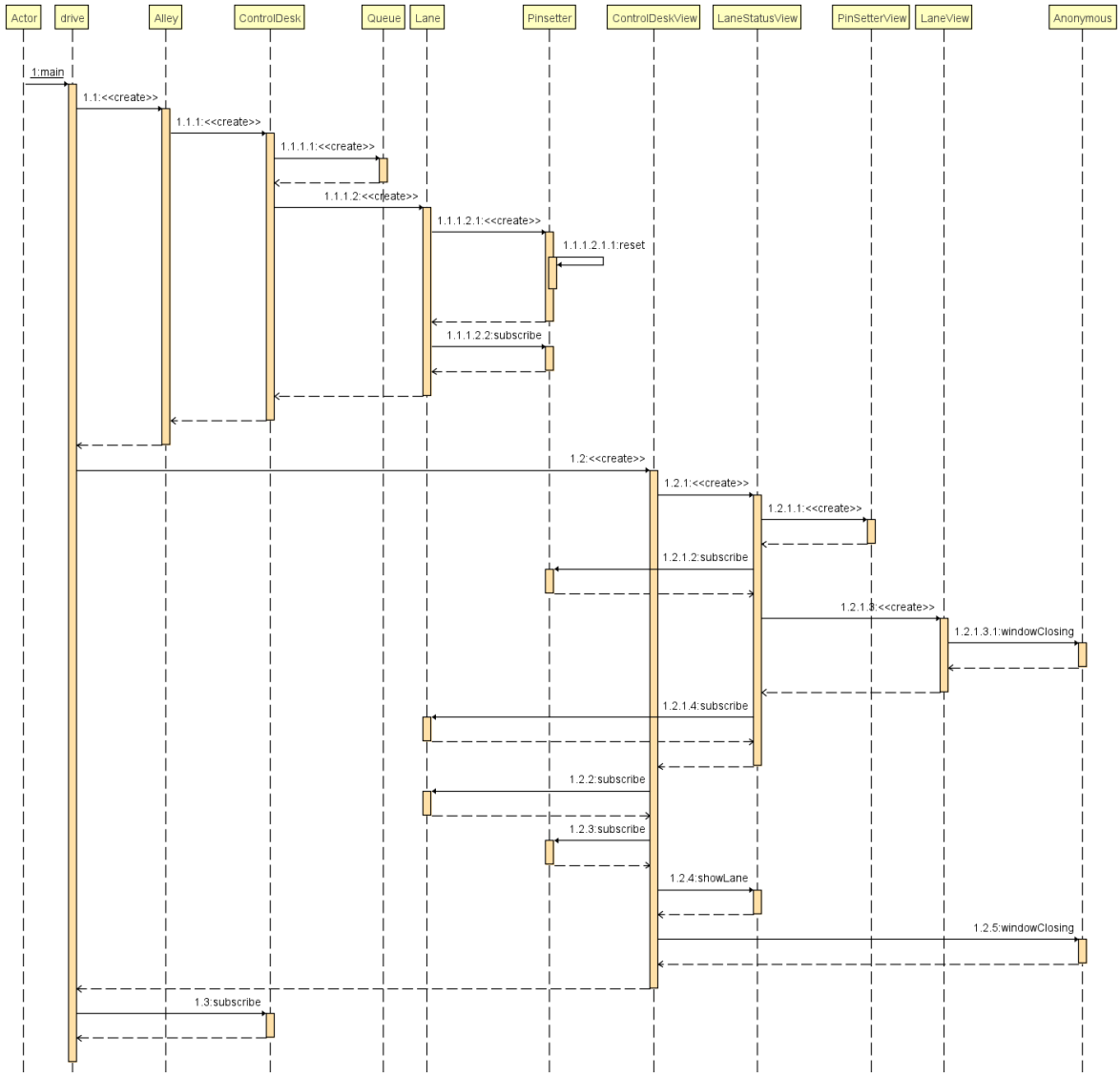


[Download the Diagram]

3. UML Sequence Diagram

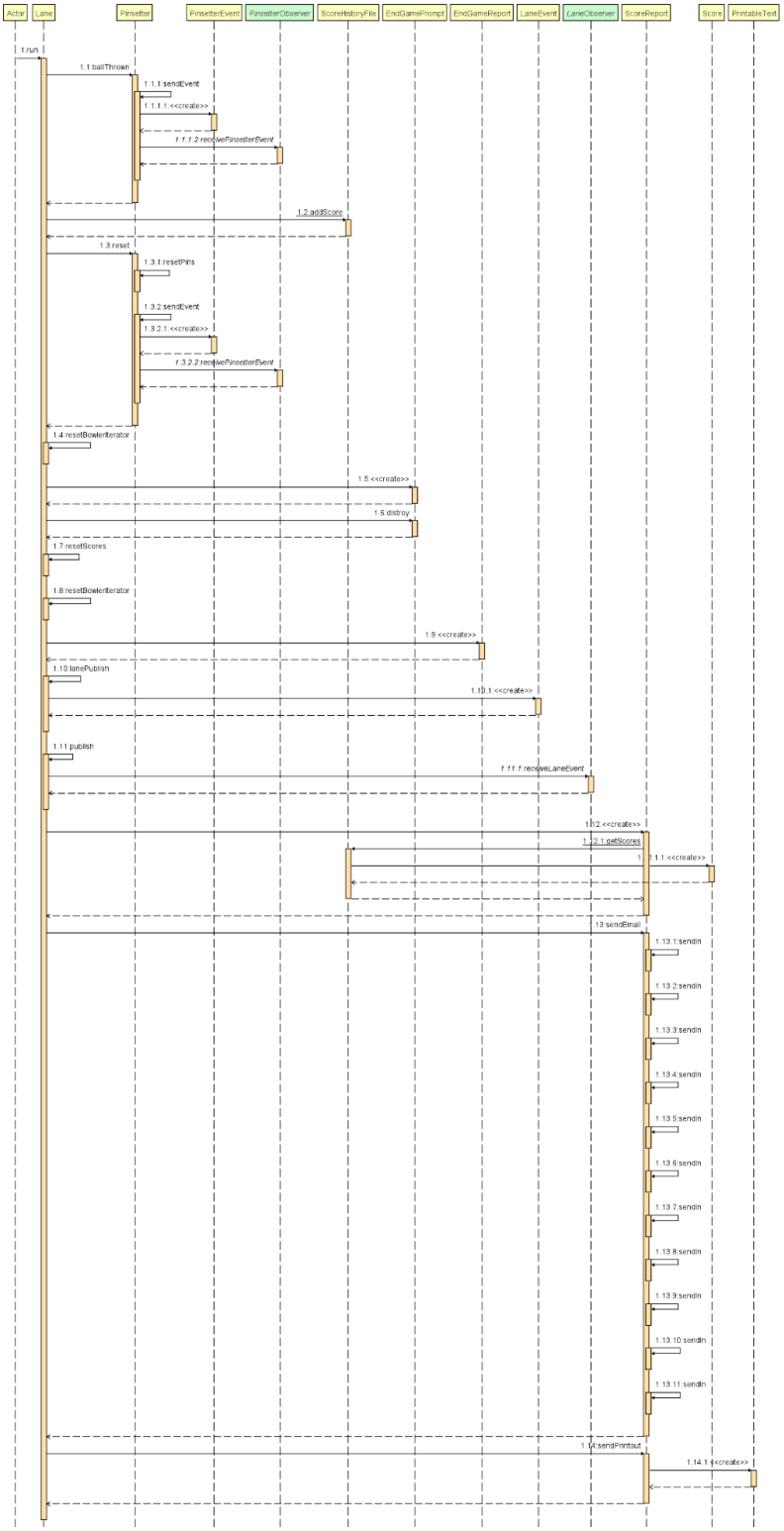
Before Refactoring:

Sequence Diagram of main() method in Drive.java.



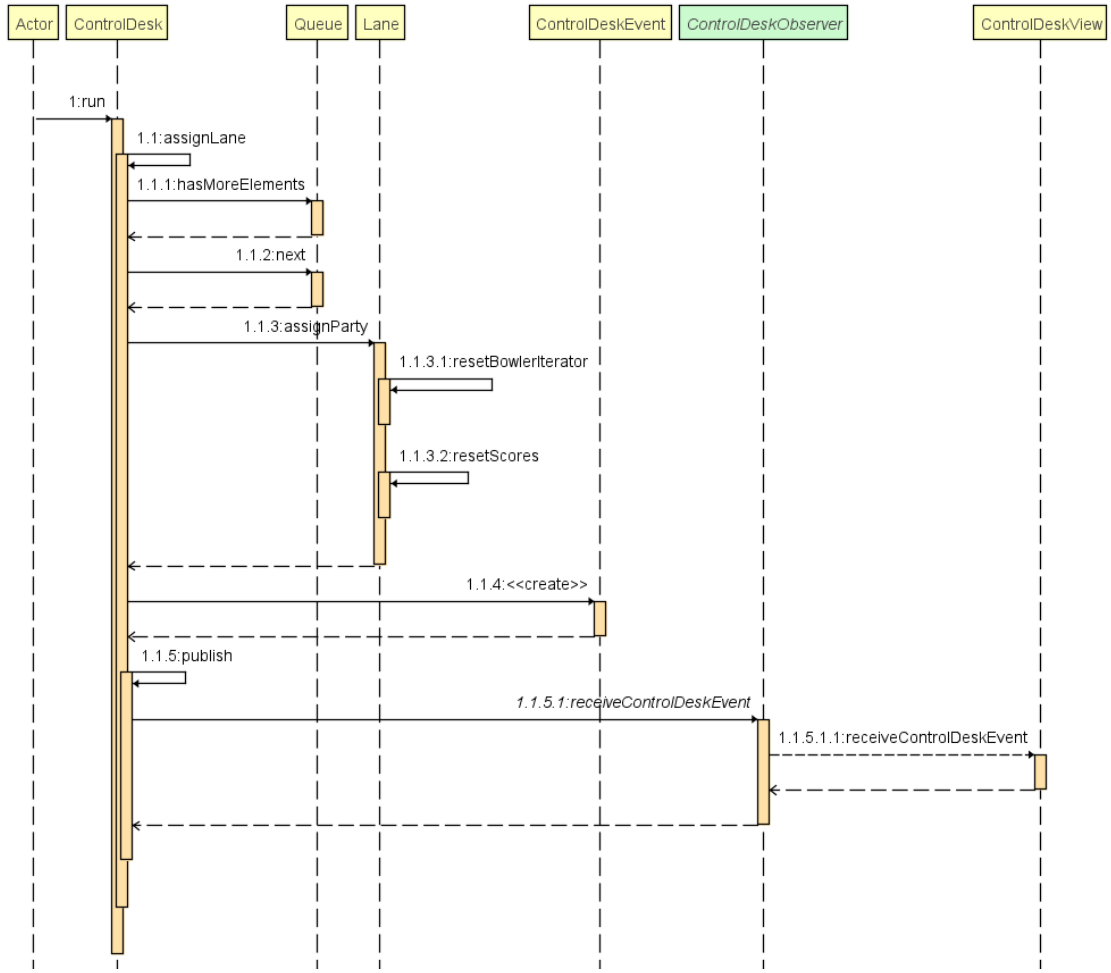
[Download the Diagram]

Sequence Diagram of related run() method in Lane.java. Sequence diagram is on the next page.



[Download the Diagram]

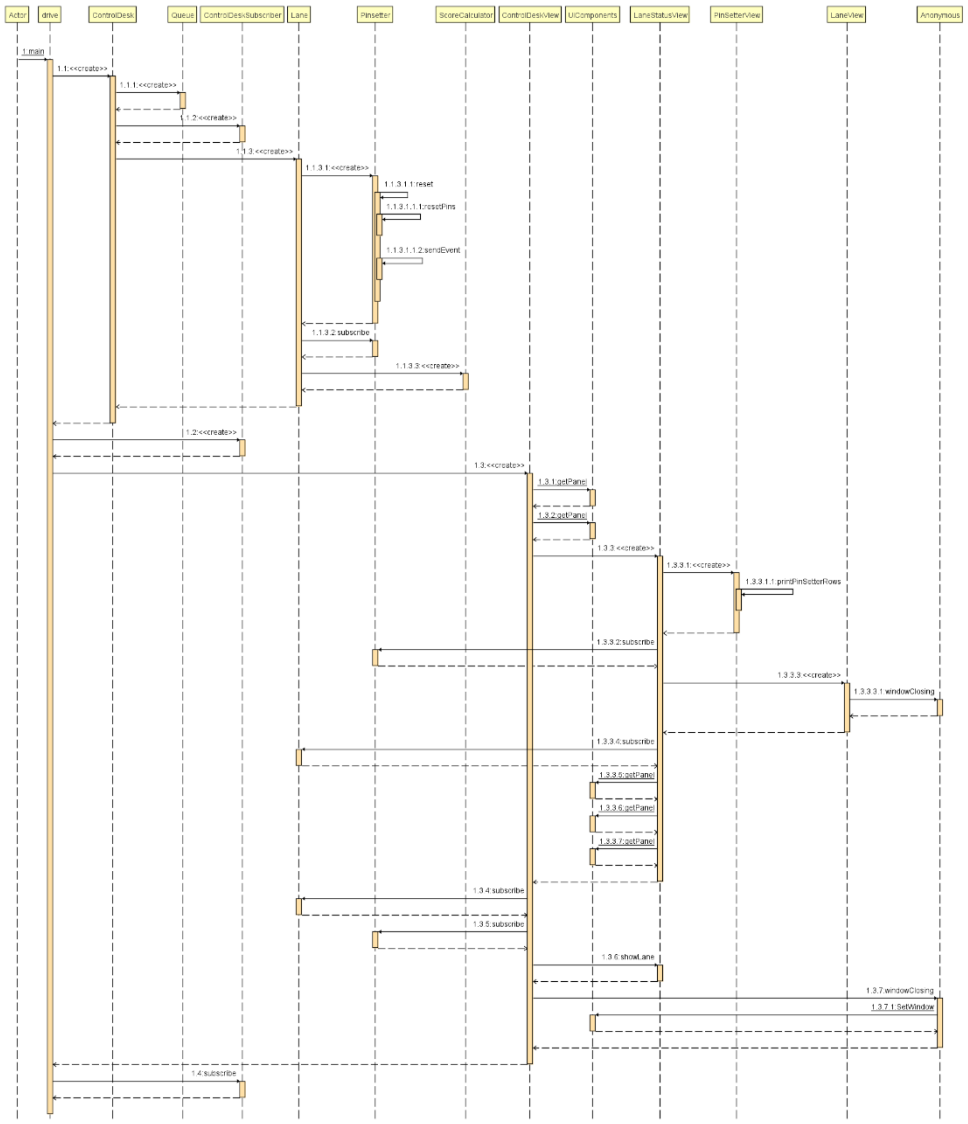
Sequence Diagram of related run() method in ControlDesk.java



[Download the Diagram]

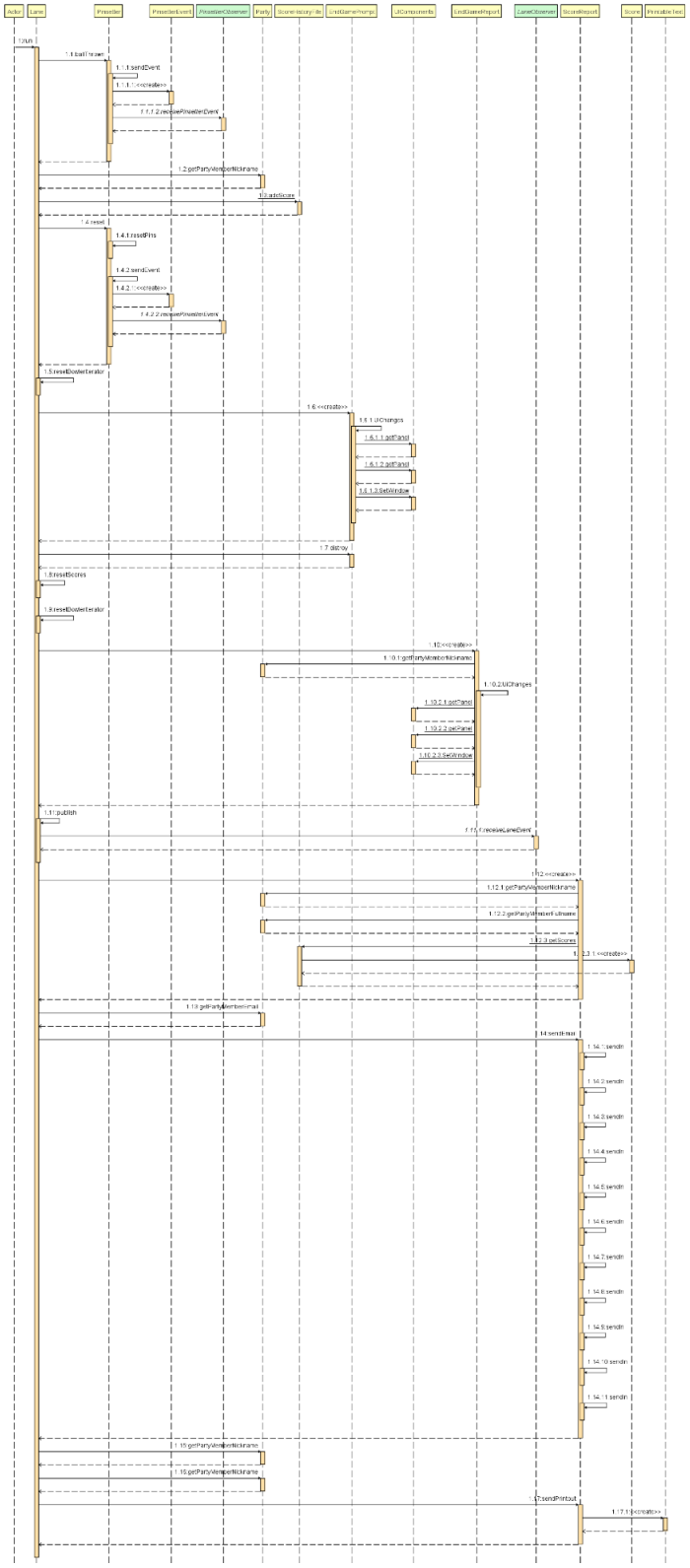
After Refactoring:

Sequence Diagram of main() method in Drive.java.



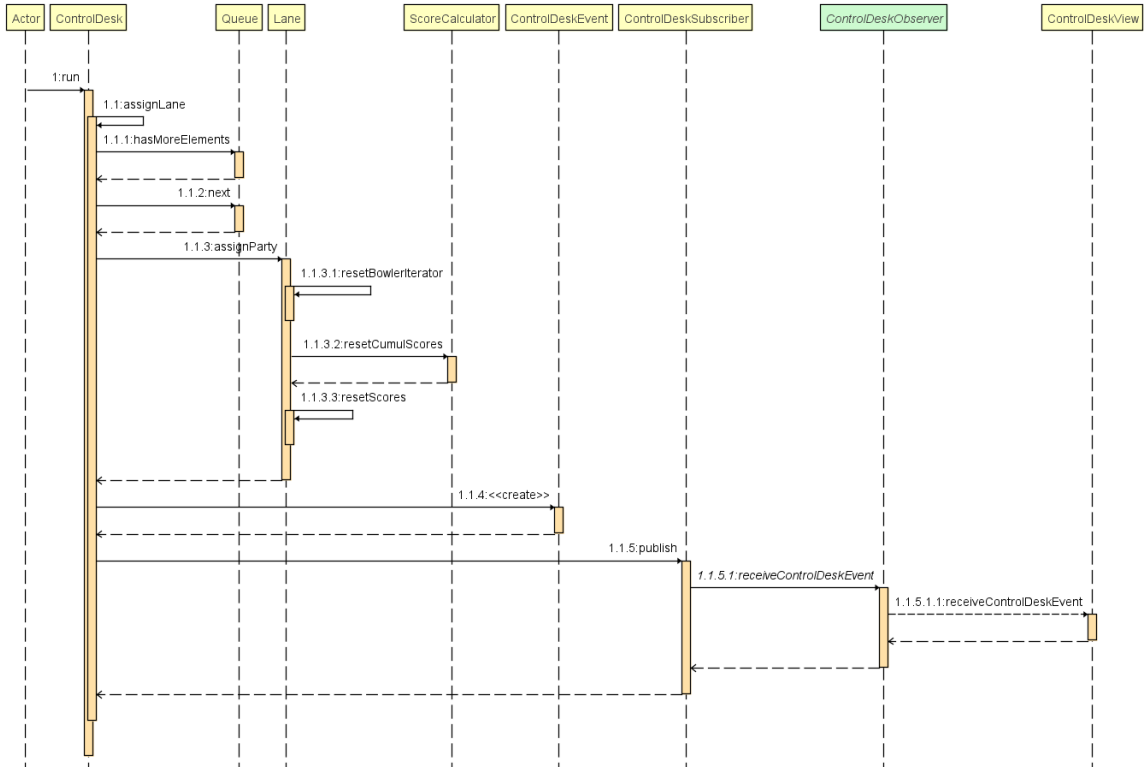
Download the Diagram

Sequence Diagram of related run() method in Lane.java. The Sequence diagram is on the next page.



[Download the Diagram]

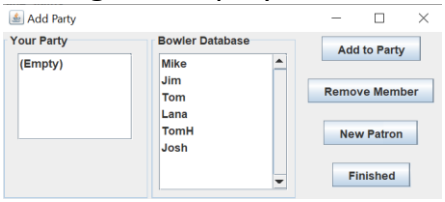
Sequence Diagram of related run() method in ControlDesk.java



[Download the Image]

4. Class Responsibility

1) AddPartyView

Variables	Methods	Responsibility
maxSize win addPatron remPatron finished partyList allBowlers Party Bowlerdb Lock	actionPerformed() valueChanged() getNames() updateNewPatron() getParty()	It takes care of the UI part in adding a new player. 

2) Bowler

Variables	Methods	Responsibility
FullName nickName email	getNickName() getFullName() getNick() getEmail() equals()	Contains Bowler/Player information.

3) BowlerFile

Variables	Methods	Responsibility
	getBowlerInfo() putBowlerInfo() getBowlers()	Access and update bowler details which are present in a file.

4) ControlDesk

Variables	Methods	Responsibility
Lanes partyQueue numLanes subscribers	run() registerPatron() assignLane() addPartyQueue() getPartyQueue() getNumLanes() subscribe() publish() getLanes()	It assigns a party(group of bowlers) to a lane. If all the lanes are occupied, then a new party is added to the PartyQueue.

5) ControlDeskEvent

Variables	Methods	Responsibility
partyQueue	getPartyQueue()	Handles the Party queue when lanes are occupied.

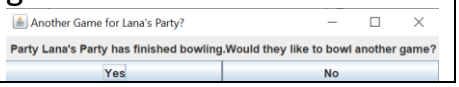
6) ControlDeskView

Variables	Methods	Responsibility
addPartyFinished Assign Win partyList maxMembers controldesk	actionPerformed() updateAddParty() receiveControlDeskEvent()	Displays the Control Desk. 

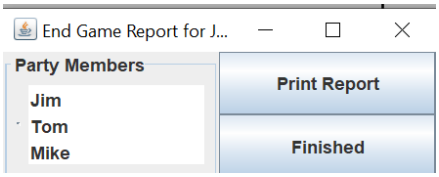
7)Drive

Variables	Methods	Responsibility
	main()	The main class where the game starts from. There are variables to set the number of Lane and the Maximum number of Players in Party. A ControlDeskObject is created here.

8) EndGamePrompt

Variables	Methods	Responsibility
Win yesButton noButton result selectedNick selectedMember	actionPerformed() getResult() destroy()	Displays a dialog box after a game completion, where you are asked whether you would like to play another game. 

9) EndGameReport

Variables	Methods	Responsibility
win printButton finished memberList myVector retVal Result selectedMember	actionPerformed() valueChanged() getResult() destroy()	Used to display the window below. 

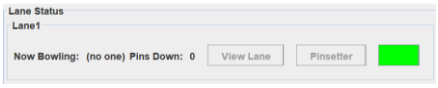
10) Lane

Variables	Methods	Responsibility
Party setter scores subscribers gameIsHalted gameFinished bowlerIterator ball bowlIndex frameNumber tenthFrameStrike curScores cumulScores canThrowAgain finalScores gameNumber CurrentThrower isFirstGame	run() receivePinsetterEvent() resetBowlerIterator() resetScores() assignParty() markScore() lanePublish() getScore() isPartyAssigned() isGameFinished() subscribe() unsubscribe() publish() getPinsetter() pauseGame() unPauseGame() handleEndGame()	This class implements a thread. The Lane Class performs the action of playing the game, players take turns and their score is updated to the frame, the pinsetter class decided which pins fall based on a random number generation.

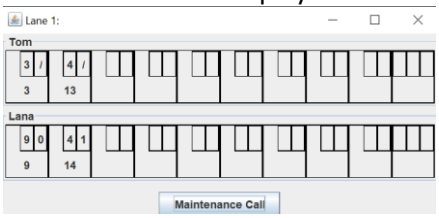
11) LaneEvent

Variables	Methods	Responsibility
P Frame ball bowler cumulScore score index frameNum curScores mechProb	isMechanicalProblem() getFrameNum() getScore() getCurScores() getIndex() getFrame() getBall() getCumulScore() getParty() getBowler()	Class is called when the game finishes, paused or unpaused.

12)LaneStatusView

Variables	Methods	Responsibility
jp curBowler foul pinsDown viewLane viewPinsetter maintenance psv lv lane laneNum laneShowing psShowing	showLane() actionPerformed() receiveLaneEvent() receivePinsetterEvent(	Class used to show the lane status section of control desk window. 

13)LaneView

Variables	Methods	Responsibility
roll initDone frame cpanel bowlers cur bowlIt balls ballLabel Scores scoreLabel ballGrid pins maintenance lane makeFrame() receiveLaneEvent() actionPerformed()	makeFrame() receiveLaneEvent() actionPerformed()	Displays the Lane Window which contains frames with players score. 

14) NewPatronView

Variables	Methods	Responsibility
maxSize win bbort finished nickLabel fullLabel emailLabel nickField fullField emailField nick full Email done selectedNick selectedMember addParty	actionPerformed() done() getNick() getFull() getEmail()	Displays the window to add new Patron/Player. 

15) Party

Variables	Methods	Responsibility
myBowlers	getMembers()	This class contains a vector of the bowlers in a Party. The getMembers() method returns the vector.

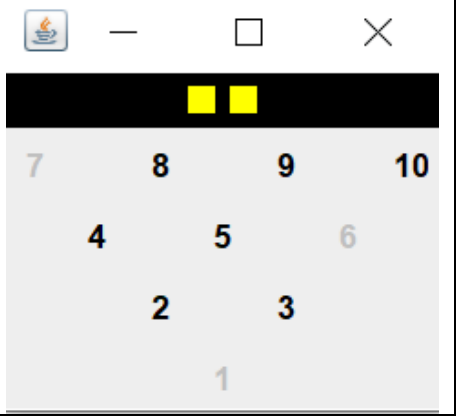
16)Pinsetter

Variables	Methods	Responsibility
rnd subscribers pins foul throwNumber sendEvent()	ballThrown() reset() resetPins() subscribe()	This class primarly calculates which pin to knock down, using the ballThrown() method.

17) PinsetterEvent

Variables	Methods	Responsibility
pinsStillStanding foulCommited throwNumber pinsDownThisThrow	pinKnockedDown() pinsDownOnThisThrow () totalPinsDown() isFoulComited() getThrowNumber()	Stores the details of Pinsetter.

18)PinsetterView

Variables	Methods	Responsibility
pinVect firstRoll secondRoll frame	receivePinsetterEvent() show() hide()	Displays the Pinsetter window. 

19) Queue

Variables	Methods	Responsibility
	next() add() hasMoreElements() asVector()	This class is used to queue a party when all the lanes are occupied.

20) Score

Variables	Methods	Responsibility
nick date score	getNickName() getDate() getScore() toString()	Class stores the scores of a player.

21) ScoreHistoryFile

Variables	Methods	Responsibility
	getScores()	Class is used to store the score in a file.

22) ScoreReport

Variables	Methods	Responsibility
content	sendEmail() sendPrintout() sendln()	This class displays the final score, previous score and sends score via email.

23) UIComponents

Variables	Methods	Responsibility
	getPanel() SetWindow() getFieldPanel()	Add and set properties of various UI Components

5. Analysis of Original Design

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 had low coupling metric.

Cons:

1. **Dead Code:** There was a lot of code in the initial design that was either commented out or not used anywhere in the system. Wherever possible, those undesired components were deleted in classes like PinsetterView and LaneEvent, variables and methods are included. There were also some key procedures in the system that were never called or used.
2. **Single Large Method:** We tried further modularizing the code in classes like LaneView separated into ScoreCount, PinsetterView, AddPartyView, and so on because some of the classes had very long methods and constuctors trying to do too much.
3. **Duplicate Code:** Some approaches did a similar thing by just copying and pasting the prior code. By creating methods, we may reuse the code.
4. **Use of old/deprecated tools:** Functions like show(), hide() etc. and use of AWT instead of newer Swing, and avoiding Generics, etc. was observed.
5. **Unused variables, methods, library imports:** In some places of the codebase there were unused variables, methods and unused library imports.
6. **Improper implementation of Observer pattern:** For subscribing and publishing various events, the Observer Pattern Design pattern was used in this project, which is an evidently suitable design pattern to apply. However, it was not fully implemented. The problem was that all of the tasks of subscribing and publishing were done in the same files as everything else, although according to proper coding standards, all of

these parts of publishing and subscribing should be in a centralised file that is utilised by all other files.

Fidelity to the Design Document:

The initial codebase mainly met the requirements of the design paper that came with the code. Except for the "Print Report" function, which was not operating at the end of a game.

Design Patterns

1. **Observer Pattern:** The system's event processing on a button click is a nice illustration of the observer pattern. Here, we wait on thread for a user-initiated event, such as a button click, and inform the related event-handler, which carries out a task corresponding to the button click.
2. **Adapter Pattern:** The Adapter pattern, as we all know, is a structural design pattern that acts as a link between two incompatible interfaces. So, in the given system the ControlDesk Class acts as an Adapter. It joins Bowlers, Party and Queue subsystems.
3. **Singleton Pattern:** A software design pattern that limits the number of "single" instances of a class. This is evident in the drive class, which serves as the program's primary function and is instantiated only once during its lifespan.

6. Analysis of Refactored Design

Responsibilities of newly created classes:

Class Name	Major Responsibility
ControlDeskSubscriber	Maintain the Subscribers
ScoreCalculator	Calculate the score for every throw
UIComponents	Add and set properties of various UI Components

1. Code Repetition:

Repetition of same code instead of creating one common method and using it is a major code smell. This makes the code very bulky, lengthy and decreases the code quality. If there is a bug in the repeated code, then it needs to be fixed in multiple places. This makes maintaining the code extremely hard.

Issue:

In all view classes code repetition can be observed as below.

```
addPatron = new JButton("Add to Party");
JPanel addPatronPanel = new JPanel();
addPatronPanel.setLayout(new FlowLayout());
addPatron.addActionListener(this);
addPatronPanel.add(addPatron);

remPatron = new JButton("Remove Member");
JPanel remPatronPanel = new JPanel();
remPatronPanel.setLayout(new FlowLayout());
remPatron.addActionListener(this);
remPatronPanel.add(remPatron);

newPatron = new JButton("New Patron");
JPanel newPatronPanel = new JPanel();
newPatronPanel.setLayout(new FlowLayout());
newPatron.addActionListener(this);
newPatronPanel.add(newPatron);
```

Fix:

A new class is added to handle adding and setting the properties of UI components. This has reduced the code repetition.

```
import java.awt.Dimension;

public class UIComponents {

    public static JPanel getPanel(JButton curbutton)
    {
        JPanel curbuttonPanel = new JPanel();
        curbuttonPanel.setLayout(new FlowLayout());
        curbuttonPanel.add(curbutton);
        return curbuttonPanel;
    }

    public static void SetWindow(JFrame win){
        Dimension screenSize = (Toolkit.getDefaultToolkit()).getScreenSize();
        win.setLocation(
                ((screenSize.width) / 2) - ((win.getSize().width) / 2),
                ((screenSize.height) / 2) - ((win.getSize().height) / 2));
```

As the same code is repeated in multiple classes, adding these methods in a new class allowed all those classes to access the methods.

2. Unused Variables:

Variables that were declared for a specific purpose but were not used later in the code. When utilized correctly, they can be quite effective, but when used incorrectly, they can reveal a lack of adequate design. **Example:** selectedNick, selectedMember, maxSize in NewPatronView.java. Every file was cleaned from unused variables.

3. Removed unnecessary imports: The Integrated Development Environment (IDE) should handle the imports section of a file, not the developer. If this is the case, imports that aren't used or aren't useful should be avoided. Leaving them in affects the readability of the code since their existence might be perplexing. Removed all unnecessary imports from java files.

4.Removed Redundant Casting to various fields: Numerous fields were cast to other fields needlessly in many files. Casting expressions that are not needed make the code more difficult to read and understand.

5.Redundant manual array Copy: Instead of writing a manual for loop in the file PinsetterEvent.java, a copy method should be used to copy an array since it helps to prevent errors.

Before Refactoring:

```
for (int i=0; i <= 9; i++) {
        pinsStillStanding[i] = ps[i];
}
```

After Refactoring:

```
System.arraycopy(ps, 0, pinsStillStanding, 0, 10);
```

6. Empty Catch statements:

Catch statements were missing from a number of files. It may be problematic if the captured code did not output anything, making it impossible to debug. As a result, the inaccuracies were printed in the right places. **Example:** Pinsetter.java

Before Refactoring:

```
-                } catch (Exception e) {}
```

After Refactoring:

```
+               } catch (Exception e) {
+                       e.printStackTrace();
+               }
```

7. Law of Demeter:

The Law of Demeter principle states that a module should not have knowledge of the inner details of the objects it manipulates. In the old design every class accesses the data members of Bowler object directly which is unnecessary as the details can be accessed through Party class because a Bowler's details are needed only when the Bowler is a part of a Party.

To abstract Bowler class from all the other classes, new methods to access the details of a particular Bowler are added to Party class. The added methods are

1. getPartyMemberNickname()
2. getPartyMemberFullname()
3. getPartyMemberEmail()

Wherever the Bowler class's methods are referenced, those methods are replaced by corresponding methods of Party class.

8. Removing unused methods/ dead codes: There were some functions in the codebase which remained uninvoked. So, we removed them. **Example:** equals() method in Bowler.java, getNickName(); getNick() both methods were returning same thing. So, deleted getNickName().

```
public String getNickName() {

    return nickName;

}
```

```
public String getNick ( ) {
    return nickName;
}
```

```
public boolean equals ( Bowler b) {
        boolean retval = true;
        if ( !(nickName.equals(b.getNickName())) ) {
                        retval = false;
        }
        if ( !(fullName.equals(b.getFullName())) ) {
                        retval = false;
        }
        if ( !(email.equals(b.getEmail())) ) {
                        retval = false;
        }
        return retval;
}
```

```
public boolean equals ( Bowler b) {
        boolean retval = true;
        if ( !(nickName.equals(b.getNickName())) ) {
                        retval = false;
        }
        if ( !(fullName.equals(b.getFullName())) ) {
                        retval = false;
        }
        if ( !(email.equals(b.getEmail())) ) {
                        retval = false;
        }
        return retval;
}
```

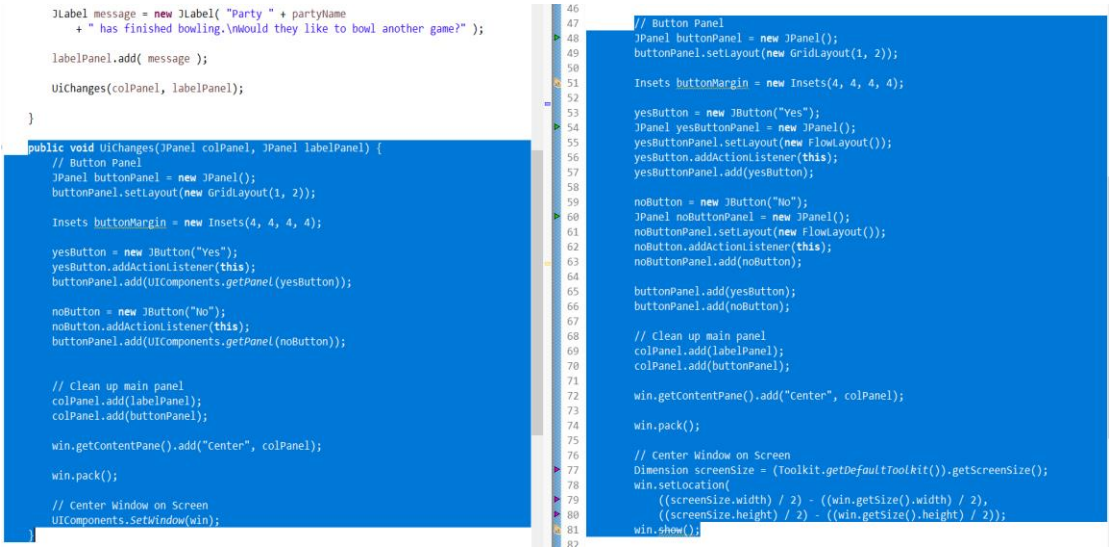
9. Removing totally redundant classes: Alley class was just a wrapper class which was returning ControlDesk object whereas we can get a ControlDesk object from ControlDesk class itself by calling ControlDesk cds = new ControlDesk(numLane). So, we removed the Alley class, basically whole Alley.java file.

```
Alley a = new Alley( numLanes );
ControlDesk controlDesk = a.getControlDesk();
```

```
ControlDesk controlDesk = new ControlDesk(numLanes);
```

10. Dividing long methods and constructors into sub-methods:

Some methods and constructors were too long and they were very hard to understand and not at all readable. So, we divided them into sub-methods.
Example: EndGamePrompt constructor, getScore() method in ScoreCalculator.java.



```
        JLabel message = new JLabel( "Party " + partyName
            + " has finished bowling.\nWould they like to bowl another game?" );

        labelPanel.add( message );

        UiChanges(colPanel, labelPanel);

    }

    public void UiChanges(JPanel colPanel, JPanel labelPanel) {
        // Button Panel
        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new GridLayout(1, 2));

        Insets buttonMargin = new Insets(4, 4, 4, 4);

        yesButton = new JButton("Yes");
        yesButton.addActionListener(this);
        buttonPanel.add(UIComponents.getPanel(yesButton));

        noButton = new JButton("No");
        noButton.addActionListener(this);
        buttonPanel.add(UIComponents.getPanel(noButton));


        // Clean up main panel
        colPanel.add(labelPanel);
        colPanel.add(buttonPanel);

        win.getContentPane().add("Center", colPanel);

        win.pack();

        // Center Window on Screen
        UIComponents.SetWindow(win);
    }
```

```
        // Button Panel
        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new GridLayout(1, 2));

        Insets buttonMargin = new Insets(4, 4, 4, 4);

        yesButton = new JButton("Yes");
        JPanel yesButtonPanel = new JPanel();
        yesButtonPanel.setLayout(new FlowLayout());
        yesButton.addActionListener(this);
        yesButtonPanel.add(yesButton);

        noButton = new JButton("No");
        JPanel noButtonPanel = new JPanel();
        noButtonPanel.setLayout(new FlowLayout());
        noButton.addActionListener(this);
        noButtonPanel.add(noButton);

        buttonPanel.add(yesButton);
        buttonPanel.add(noButton);

        // Clean up main panel
        colPanel.add(labelPanel);
        colPanel.add(buttonPanel);

        win.getContentPane().add("Center", colPanel);

        win.pack();

        // Center Window on Screen
        Dimension screenSize = (Toolkit.getDefaultToolkit()).getScreenSize();
        win.setLocation(
            ((screenSize.width) / 2) - ((win.getSize().width) / 2),
            ((screenSize.height) / 2) - ((win.getSize().height) / 2));
        win.show();
```

11.Assigning appropriate functions to appropriate files:

Many files had a wide array of functions, although good coding practices dictate that a file should contain all relevant functions. This results in an issue of:

Lack of cohesion: Cohesion metrics assess how effectively a class's methods are connected to one another. A coherent class serves a single purpose. A non-cohesive class performs two or more functions that are not connected. It may be necessary to reorganize a non-cohesive class into two or more smaller courses. The following cohesiveness metrics are based on the idea that methods are connected if they function on the same class-level variables.
If two methods work on distinct variables, they are unconnected. Methods in a cohesive class use the same set of variables. There are certain methods in a non-cohesive class that function on separate data.

We relocated functions to suitable files or established new files for that purpose since many functions in code in files were not even connected to each other. **Example:** 1. getScore() method in Lane.java. Nobody can claim that a file called Lane.java was computing a bowler's score, which it was. This functioned in a variety of other ways, such as doing all score-related tasks by itself. As a result, we created a new class ScoreCalculator.java and relocated all associated functions to that file with the required arguments. 2. registerPatron() method was in ControlDesk.java, moved it to BowelFile.java.

Advantages:

- Module complexity has been reduced (they are simpler, having fewer operations).
- Increased module reusability, since application developers will be able to discover the component, they require more readily within the module's coherent collection of activities.
- Increased system maintainability, as logical domain changes touch fewer modules and changes in one module necessitate fewer changes in others.

12. Properly Implementing ObserverPattern Design:

The Observer Pattern establishes a one-to-many relationship between objects, ensuring that when one object changes state, all of its dependents are immediately alerted and updated.
Note the following:

- Between Subject(One) and Observer(Many), there is a one-to-many interdependence (Many).
- Because Observers do not have access to data, there is a dependence. Subject is the only source of data for them.

For subscribing and publishing various events, the ObserverPattern Design pattern was used in this project, which is an evident suitable design pattern to apply. However, it was not fully implemented. The problem was that all of the tasks of subscribing and publishing were done in the same files as everything else, although according to proper coding standards, all of these parts of

publishing and subscribing should be in a centralised file that is utilised by all other files.

Example: we shifted the subscribe() and publish() method from ControlDesk.java to newly created ControlDeskSubscriber.java which will only be used for subscribing and publishing.

```
import java.util.Iterator;

public class ControlDeskSubscriber {

    private Vector subscribers;

    public ControlDeskSubscriber() {
        subscribers = new Vector();
    }


     /**
     * Allows objects to subscribe as observers
     *
     * @param adding    the ControlDeskObserver that will be subscribed
     *
     */
     public static void subscribe(ControlDesk controlDesk, ControlDeskObserver adding) {
        controlDesk.subscribers.add(adding);
    }

    /**
     * Broadcast an event to subscribing objects.
     *
     * @param event the ControlDeskEvent to broadcast
     *
     */
    public static void publish(ControlDesk controlDesk, ControlDeskEvent event) {
        for (Object subscriber : controlDesk.subscribers) {
            ((ControlDeskObserver) subscriber).receiveControlDeskEvent(event);
        }
    }

}
```

We also did the same thing for Lane.java, made a new dedicated file LaneSubscriber.java. But since it was increasing coupling, we restored it back.

Advantages: Interacting items are given a loosely connected design. Objects that are loosely connected are adaptable to changing needs. In this case, loose coupling implies that the interacting objects should know less about one another.

This loose connection is provided via the observer pattern as follows:

- The only thing the subject is aware of is that the observer implements the Observer interface. There's nothing else.
- For adding or removing observers we don’t need to change Subject.
- Subject and observer classes can be reused independently of one another.

13. Removing redundant Classes, Interfaces: We removed the LaneEvent.java because this class was having only constructor to initialize variables and getter methods. It was having low cohesion. So, we removed it and in some places LaneEvent objects were passes in functions.We directly passed the relevant variables in place of LaneEvent object. Similarly, we also removed LaneEventInterface because after removal of LaneEvent class, it was also useless.

```
public interface LaneObserver {
   public void receiveLaneEvent(Party pty, int theIndex, Bowler theBowler, int[][] theCumulScore, HashMap theScore, int theFrameNum, int[] theCurS
};
```

```
public interface LaneObserver {
    public void receiveLaneEvent(LaneEvent le);
};
```

14. Low Coupling: We tried to keep the dependencies between the classes as low as possible by sending parameters locally and deleting duplicate ones whenever possible. We've expanded our class list to allow us to divide down huge files like Lane into subclasses. We made sure that these classes were mainly self-contained and that they didn't require too many other dependencies to increase coupling.

15. 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.

16. Reusability: Several methods were built to ensure code reusability. We modularized the code wherever we observed a similar operation being done via copy-pasting in the original code.

7. Metric Analysis:

1) Number of classes:
Original Design: 29
Refactored: 28 (removed Alley, LaneEvent, added ControlDeskSubscriber)

2) Number of methods:
Original Design: 141
Refactored: 133 (removed LaneEvent and Alley classes so their methods were removed)

3) Weighted methods per class
The number of methods per class calculated in a weighted fashion for the overall code decreased.
Original Design: 230
Refactored: 227
Technique: Smartly decomposing the functions in Lane, we were able to bring it down.

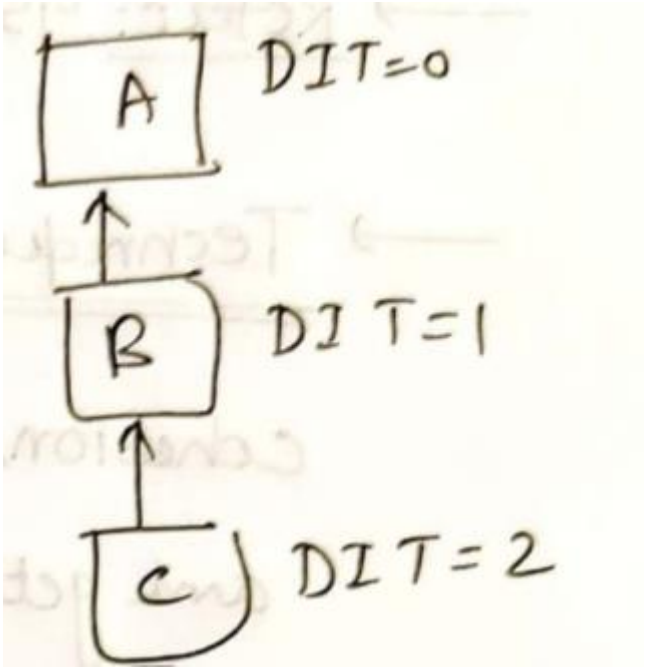
4) Lines of Code
Original Design: 1793
Refactored: 1637
The following was done to reduce the no of lines of code:
Code cleaning, removal of redundant classes, unused methods, libraries and variables.

5) Response for Class
The Response for class metric is the total number of methods that can potentially be executed in response to a message received by an object of a class. This number is the sum of the methods of the class, and all distinct methods are invoked directly within the class methods.

Original Design: 202
Refactored: 184

Technique: Cleaning Lane.java, deleting Alley.java and LaneEvent.java

6) Depth of Inheritance (DIT)
Code metric that measures the maximum length between a node and the root node in a class hierarchy.
Higher DIT means greater level of complexity, also means a longer number of attributes and methods being inherited.



Original Design: 27
Refactored:26

Explanation: Deleted LaneEvent.java, so the depth came down because some classes were inheriting LaneEvent.java

7) Coupling between Objects

Original Design: 38
Refactored: 34

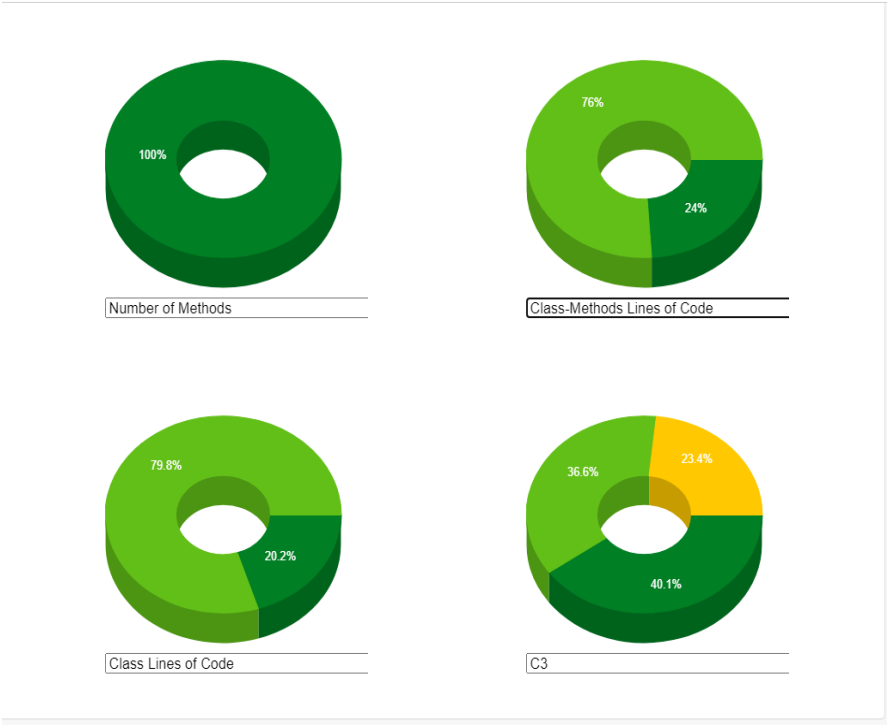
Explanation: Tried to keep the dependencies between the classes as low as possible by sending parameters locally (deleting LaneEvent.java and sending its parameters locally). Dividing huge files like Lane into subclasses such as ScoreCalculator.java which helped to achieve low coupling

8) Lack of Cohesion
In lack of cohesion lower is better.
Original Design: 100
Refactored: 95

Technique: LaneEvent.java had low cohesion, since it only had constructors and getter methods, which were not at all connected. So, we deleted the class and that increased the overall cohesion.

Metric of original design:


Analysis of BowlingAlleyEclipse
General Information
Total lines of code: 1438
Number of classes: 29
Number of packages: 1
Number of external packages: 19
Number of external classes: 96
Number of problematic classes: 0
Number of highly problematic classes: 0
36.6%
23.4%
40.1%
C3
Very High
High
Medium-high
Low-medium
Low
Distribution of Quality Attributes
Complexity, Coupling, Cohesion, and Size
29.3%
15.8%
54.9%
Complexity
33%
67%
Coupling
36.6%
23.4%
40.1%
Lack of Cohesion
79.8%
20.2%
Size


100%
Number of Methods
76%
24%
Class-Methods Lines of Code
79.8%
20.2%
Class Lines of Code
36.6%
23.4%
40.1%
C3

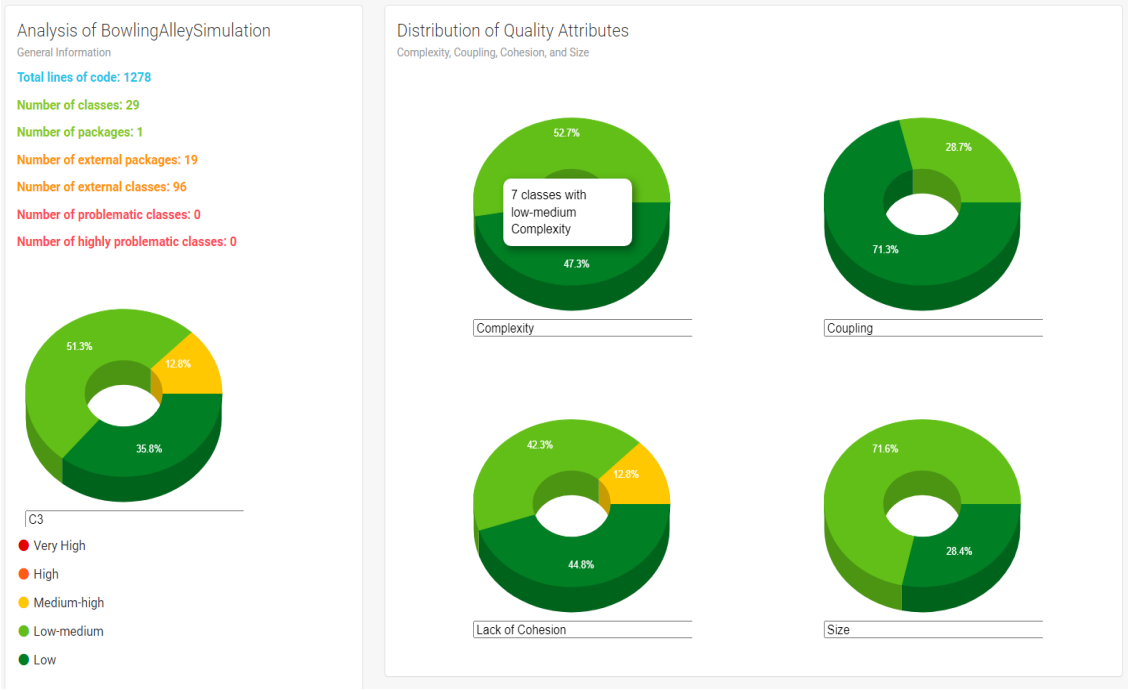
List of all classes (#29)

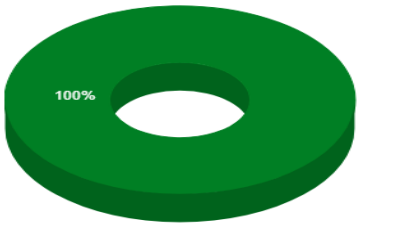
ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	Lane					227	medium-high	low-medium	medium-high	low-medium
2	ControlDeskView					87	low-medium	low-medium	low-medium	low-medium
3	ControlDesk					68	low-medium	low-medium	medium-high	low-medium
4	LaneStatusView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

Item	Value	Mean Value...	Min Value	Max Value	Resource with Max V...	Description
Number of Classes	29	0	1	1	NewPatronView.java	Return the number of classes and inner classes of a class in a project.
Lines of Code	1793	61.828	3	319	Lane.java	Number of the lines of the code in a project.
Number of Methods	141	4.862	1	17	Lane.java	The number of methods in a project.
Number of Attributes	118	4.069	1	18	Lane.java	The number of attributes in a project.
Cyclomatic Complexity	192	6.621	2	4	LaneView.java	It is calculated based on the number of different possible paths through the source code.
Weight Methods per Class	230	7.931	5	7	ScoreReport.java	It is the sum of the complexities of all class methods.
Depth of Inheritance Tree	26	0.897	1	2	Lane.java	Provides the position of the class in the inheritance tree.
Number of Children	6	0.207	0	3	PinsetterObserver.java	It is the number of direct descendants (subclasses) for each class.
Coupling between Objects	38	1.31	1	4	ControlDesk.java	Total of the number of classes that a class referenced plus the number of classes that referenced the class.
Fan-out	29	1	1	1	NewPatronView.java	Defined as the number of other classes referenced by a class.
Response for Class	202	6.966	1	37	Lane.java	Measures the complexity of the class in terms of method calls. It is calculated by adding the number of methods in the class (not including inherited methods)...
Lack of Cohesion of Methods	100	3.448	0	17	Lane.java	LCOM defined by CK.
Lack of Cohesion of Methods 2	10.87	0.375	0	0.91	LaneEvent.java	It is the percentage of methods that do not access a specific attribute averaged over all attributes in the class. If the number of methods or attributes is zero, L...
Lack of Cohesion of Methods 4	133	4.586	0	18	Lane.java	LCOM4 measures the number of 'connected components' in a class. A connected component is a set of related methods and fields. There should be only one s...
Tight Class Cohesion	10.476	0.361	0.1	2	PrintableText.java	Measures the 'connection density', so to speak (while LCC is only affected by whether the methods are connected at all).
Loose Class Cohesion	10.479	0.361	0.101	2	PrintableText.java	Measures the overall connectedness. It depends on the number of methods and how they group together.

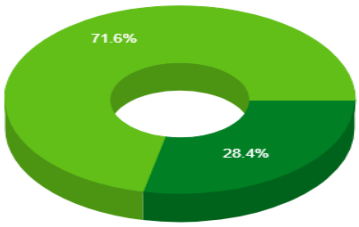
Download the Table

Metrics of refactored design:

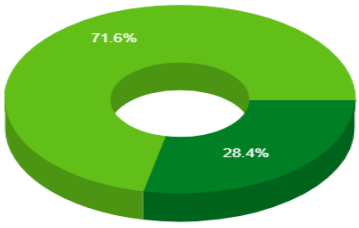




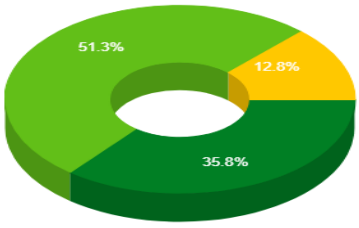
Number of Methods



Class-Methods Lines of Code



Class Lines of Code



C3

List of all classes (#29)

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	Lane					164	low-medium	low-medium	medium-high	low-medium
2	ControlDeskView					74	low-medium	low-medium	low-medium	low-medium
3	ControlDesk					45	low-medium	low-medium	low	low
4	LaneStatusView					84	low	low-medium	low-medium	low-medium
5	LaneView					137	low-medium	low	low-medium	low-medium
6	AddPartyView					109	low-medium	low	low-medium	low-medium
7	PinSetterView					74	low-medium	low	low-medium	low-medium
8	ScoreCalculator					70	low-medium	low	low	low-medium
9	ScoreReport					76	low	low	low	low-medium
10	NewPatronView					64	low	low	low	low-medium
11	EndGameReport					63	low	low	low-medium	low-medium
12	BowlerFile					47	low	low	low	low
13	Pinsetter					47	low	low	low	low
14	EndGamePrompt					46	low	low	low	low

15	PinsetterEvent	■	■	■	■	26	low	low	low	low
16	PrintableText	■	■	■	■	21	low	low	low	low
17	ScoreHistoryFile	■	■	■	■	20	low	low	low	low
18	UIComponents	■	■	■	■	17	low	low	low	low
19	Party	■	■	■	■	16	low	low	low	low
20	Score	■	■	■	■	16	low	low	low	low
21	ControlDeskSubscr...	■	■	■	■	14	low	low	low	low
22	Bowler	■	■	■	■	14	low	low	low	low
23	Queue	■	■	■	■	12	low	low	low	low
24	drive	■	■	■	■	8	low	low	low	low
25	ControlDeskEvent	■	■	■	■	6	low	low	low	low
26	ControlDeskObserver	■	■	■	■	2	low	low	low	low
27	LaneObserver	■	■	■	■	2	low	low	low	low
28	LaneServer	■	■	■	■	2	low	low	low	low
29	PinsetterObserver	■	■	■	■	2	low	low	low	low

o3smeasures Main Measures Diagnostic View ×

Project: BowlingAlleySimulation

Item	Value	Mean Value...	Min Value	Max Value	Resource with Max V...	Description
Number of Classes	28	0	1	1	PinsetterObserver.java	Return the number of classes and inner classes of a class in a project.
Lines of Code	1637	58.464	3	220	Lane.java	Number of the lines of the code in a project.
Number of Methods	133	4.75	1	15	Lane.java	The number of methods in a project.
Number of Attributes	105	3.75	0	18	Lane.java	The number of attributes in a project.
Cyclomatic Complexity	194	6.929	1	8	PinSetterView.java	It is calculated based on the number of different possible paths through the source code.
Weight Methods per Class	227	8.107	1	8	PinSetterView.java	It is the sum of the complexities of all class methods.
Depth of Inheritance Tree	27	0.964	0	2	ControlDesk.java	Provides the position of the class in the inheritance tree.
Number of Children	6	0.214	2	3	PinsetterObserver.java	It is the number of direct descendants (subclasses) for each class.
Coupling between Objects	34	1.214	1	3	ControlDesk.java	Total of the number of classes that a class referenced plus the number of classes that referenced the class.
Fan-out	28	1	1	1	PinsetterObserver.java	Defined as the number of other classes referenced by a class.
Response for Class	184	6.571	1	29	Lane.java	Measures the complexity of the class in terms of method calls. It is calculated by adding the number of methods in the class (not including inherited methods) pl
Lack of Cohesion of Methods	95	3.393	0	15	Lane.java	LCOM defined by CK.
Lack of Cohesion of Methods 2	10.127	0.362	0	0.857	LaneView.java	It is the percentage of methods that do not access a specific attribute averaged over all attributes in the class. If the number of methods or attributes is zero, LCC
Lack of Cohesion of Methods 4	118	4.214	0	18	Lane.java	LCOM4 measures the number of 'connected components' in a class. A connected component is a set of related methods and fields. There should be only one suc
Tight Class Cohesion	8.634	0.308	0	2	PrintableText.java	Measures the 'connection density', so to speak (while LCC is only affected by whether the methods are connected at all).
Loose Class Cohesion	8.64	0.309	0	2	PrintableText.java	Measures the overall connectedness. It depends on the number of methods and how they group together.

[Download the Table]

Discussion of Metric:

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

As illustrated in the graphs and tables above, numerous metrics for the code base were utilized for examination of the original and refactored code, including coupling, cohesion, lines of code, cyclomatic complexity, modularity, data hiding, size of classes and methods, extensibility, and reusability as shown in the graphs and tables here.

These data brought to light various elements of the original system that we covered in depth in the analysis section of the original design here.

How did you use these measurements to guide your refactoring?

We were able to rework the original design in an orderly and well-targeted manner thanks to the insights gained from these measurements. Here is a full overview of how these measurements assisted us in solving these problems to an acceptable level.

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 can be seen from the metric measurements presented here. By removing redundant classes like LaneEventInterface and LaneEvent, we were able to reduce the complexity and cohesion of Lane Class. We also developed a separate class for score calculation, which helped us to increase the cohesiveness of between Lane and the newly created class. Similarly, we created a separate class for subscribers and improved the ControlDesk class's cohesion.

Apart from this we have tried to improve other areas of overall code by various means which can be found here.

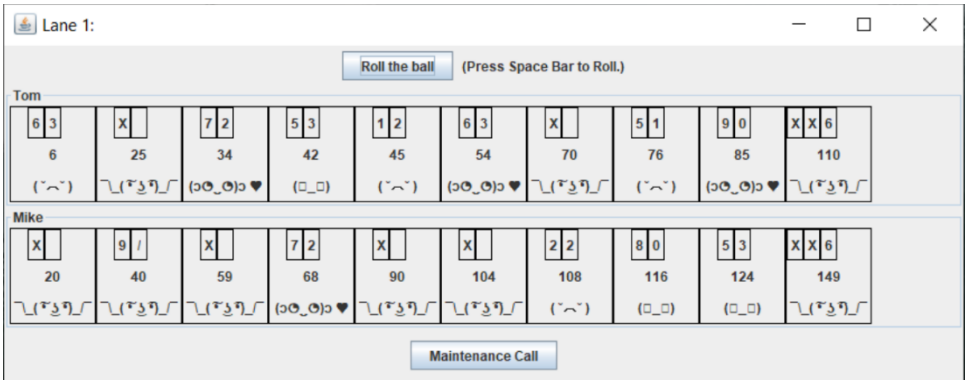
8. New Requirements:

1. Making the code interactive (Throw Ball Button):

The original code simulates “rolling the ball” continuously without taking any input from the user. To make the game more interactive “roll the ball” button has been added, the player must click on “roll the ball” or press space bar to generate scores for each frame.

1.1. Design Patterns used**:** The **observer pattern** has been used to implement the “roll the ball” button functionality. When the “roll the ball” button is clicked laneView Class notifies Lane to roll the ball.

1.2. UI Patterns used: The **Clear Primary Actions** user interface design pattern has been used, “the roll button” is the primary button on top and there is also added information “Press Space Bar to Roll” to show visually guide the user on the course of action.



2. Configurable Number of lanes and maximum players:

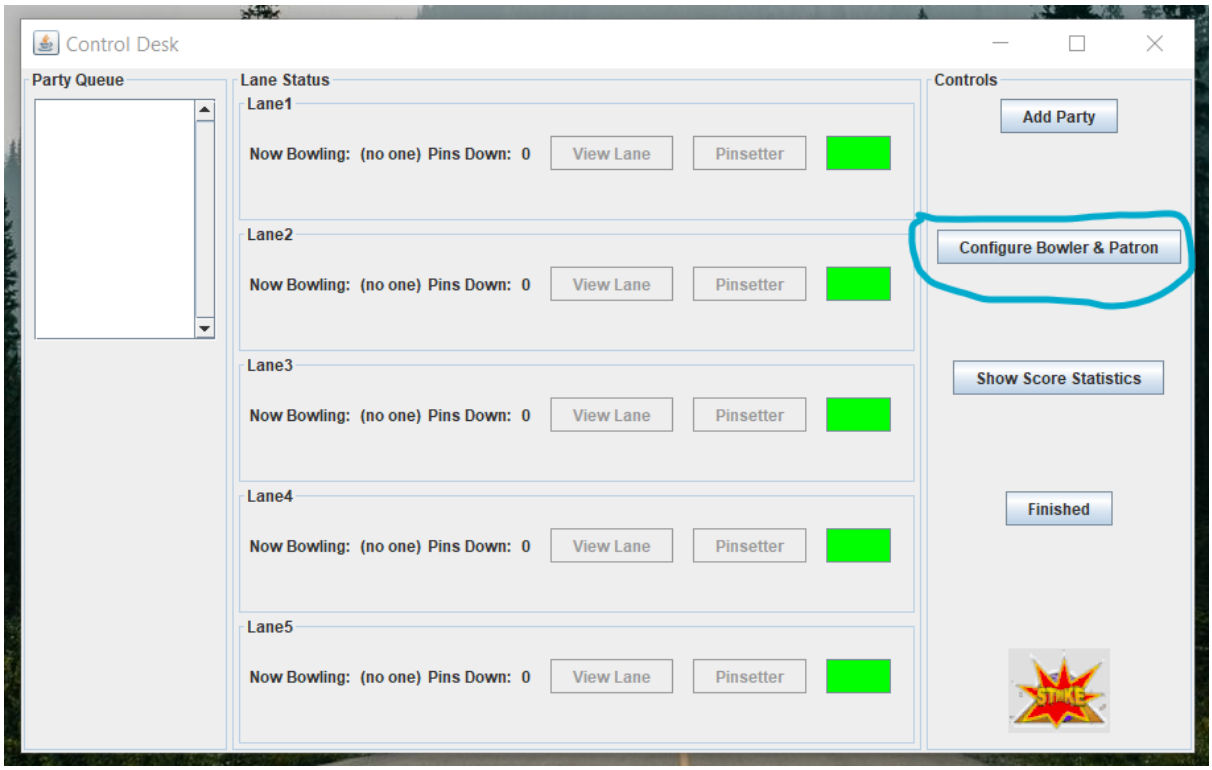
We have made the number of lanes and maximum number of players in each game configurable. A JButton named `Configure Bowler & Patron` is given to perform this operation.

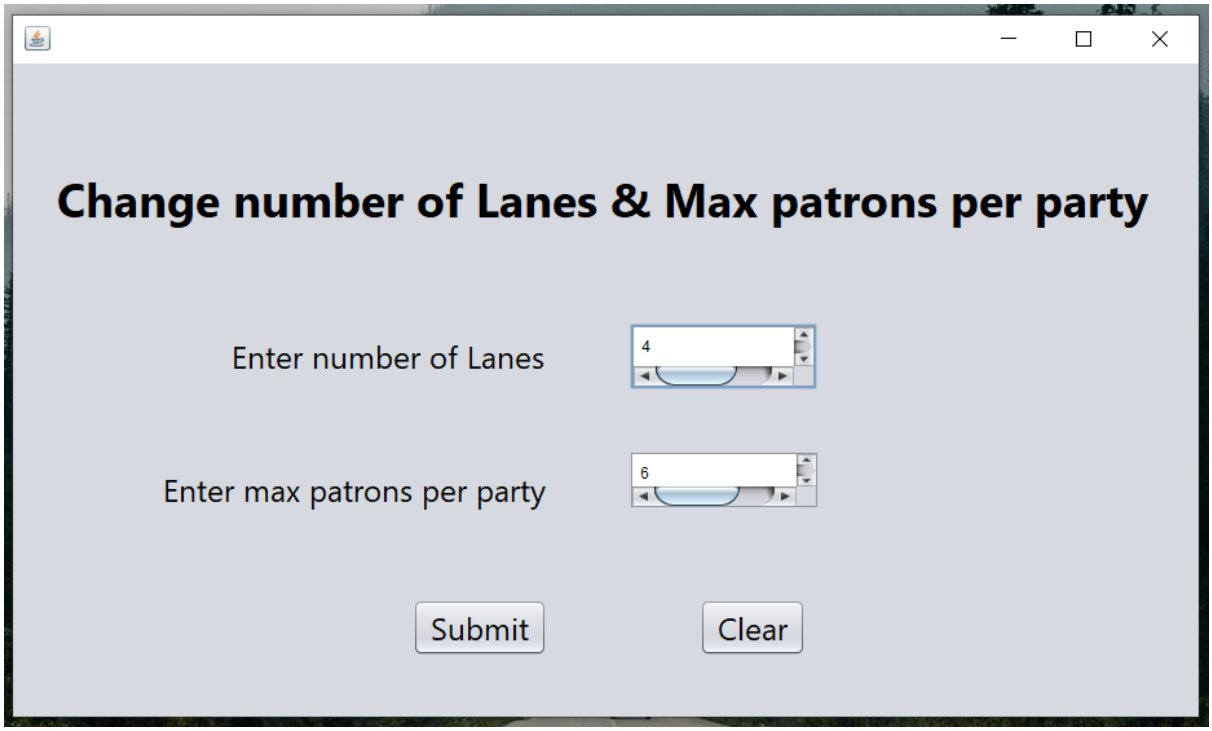
Code Snippets:

```
public LanePatronJFrame() {
    initComponents();
      start();
}

/**
 * This method is called from within the constructor to initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is always
 * regenerated by the Form Editor.
 */
@SuppressWarnings("unchecked")
// <editor-fold defaultstate="collapsed" desc="Generated Code">
private void initComponents() {

    jLabel1 = new javax.swing.JLabel();
    jLabel2 = new javax.swing.JLabel();
    jLabel3 = new javax.swing.JLabel();
    jScrollPane1 = new javax.swing.JScrollPane();
    patrons = new javax.swing.JTextArea();
    jScrollPane2 = new javax.swing.JScrollPane();
    lanes = new javax.swing.JTextArea();
    submit = new javax.swing.JButton();
    clear = new javax.swing.JButton();

    setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);

    jLabel1.setFont(new java.awt.Font("Segoe UI", 1, 36)); // NOI18N
    jLabel1.setText("Change number of Lanes & Max patrons per party");

    jLabel2.setFont(new java.awt.Font("Segoe UI", 0, 24)); // NOI18N
    jLabel2.setText("Enter number of Lanes");

    jLabel3.setFont(new java.awt.Font("Segoe UI", 0, 24)); // NOI18N
    jLabel3.setText("Enter max patrons per party");

    patrons.setColumns(15);
    patrons.setRows(3);
    jScrollPane1.setViewportView(patrons);
```

```
private void submitActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    int numOfLanes = Integer.parseInt(lanes.getText());
    int numOfPatrons = Integer.parseInt(patrons.getText());

    CreateDB.updateLanePatronTable(numOfLanes, numOfPatrons);
}

private void clearActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    lanes.setText("");
    patrons.setText("");
}
```

Working Samples:





When clicked on that button a new window will pop up giving the option to enter number of lanes and enter max patrons per party. After submitting the desired values, the changes will be stored in the database and the changes would be reflected in the next game.

3. Adding database layer to implement persistence of the scores and players:

Previously, the codebase was implemented using file operations. It was also persistent. But we removed all those file operations and replaced them with database operations, making it more robust and persistent. We have used SQLite for implementing the database part. We have created a separate class called `CreateDB` to implement all the database related operations. Basically, this class handles all the database related operations.

Code Snippets:

```
public class CreateDB {
    public static void connectDB()
    {
        Connection c = null;
        try {
            Class.forName("org.sqlite.JDBC");
//          c = DriverManager.getConnection("jdbc:sqlite:myBlog.sqlite");
            c = DriverManager.getConnection("jdbc:sqlite:bowlingAlley.db");
        } catch ( Exception e ) {
            System.err.println( e.getClass().getName() + ": " + e.getMessage() );
            System.exit(0);
        }
        System.out.println("Opened database successfully");
    }

    public static void createBowlerTable()
    {
        Connection c = null;
        Statement stmt = null;
        try {
            Class.forName("org.sqlite.JDBC");
//          c = DriverManager.getConnection("jdbc:sqlite:myBlog.sqlite");
            c = DriverManager.getConnection("jdbc:sqlite:bowlingAlley.db");
            System.out.println("Opened database successfully");

            stmt = c.createStatement();
            String sql = "CREATE TABLE IF NOT EXISTS bowler " +
                    "(nickname varchar(20) PRIMARY KEY," +
                    " fullname varchar(40) NOT NULL, " +
                    " email varchar(60) NOT NULL)";
            stmt.executeUpdate(sql);
            stmt.close();
            c.close();
        } catch ( Exception e ) {
            System.err.println( e.getClass().getName() + ": " + e.getMessage() );
            System.exit(0);
        }
```

```java
public static void insertIntoBowlerTable()
{
    Connection c = null;
    Statement stmt = null;
    try {
        Class.forName("org.sqlite.JDBC");
        c = DriverManager.getConnection("jdbc:sqlite:myBlog.sqlite");
        c = DriverManager.getConnection("jdbc:sqlite:bowlingAlley.db");
        c.setAutoCommit(false);
        System.out.println("Opened database successfully");

        stmt = c.createStatement();
        String sql = "INSERT INTO bowler (nickname, fullname, email) VALUES " +
                "('Mike', 'M. J. Lutz', 'ml@nowhere.net'), " +
                "('Jim', 'J. R. Vallino', 'jv@nowhere.net'), " +
                "('Tom', 'T. R. Reichmayr', 'tr@nowhere.net'), " +
                "('Lana', 'L. R. Verschage', 'lv@nowhere.net');";
        stmt.executeUpdate(sql);
        stmt.close();
        c.commit();
        c.close();
    } catch ( Exception e ) {
        System.err.println( e.getClass().getName() + ": " + e.getMessage() );
        System.exit(0);
    }
    System.out.println("bowler Records created successfully");
}
```

4. Implementing searchable view to make ad-hoc queries on the stored data:

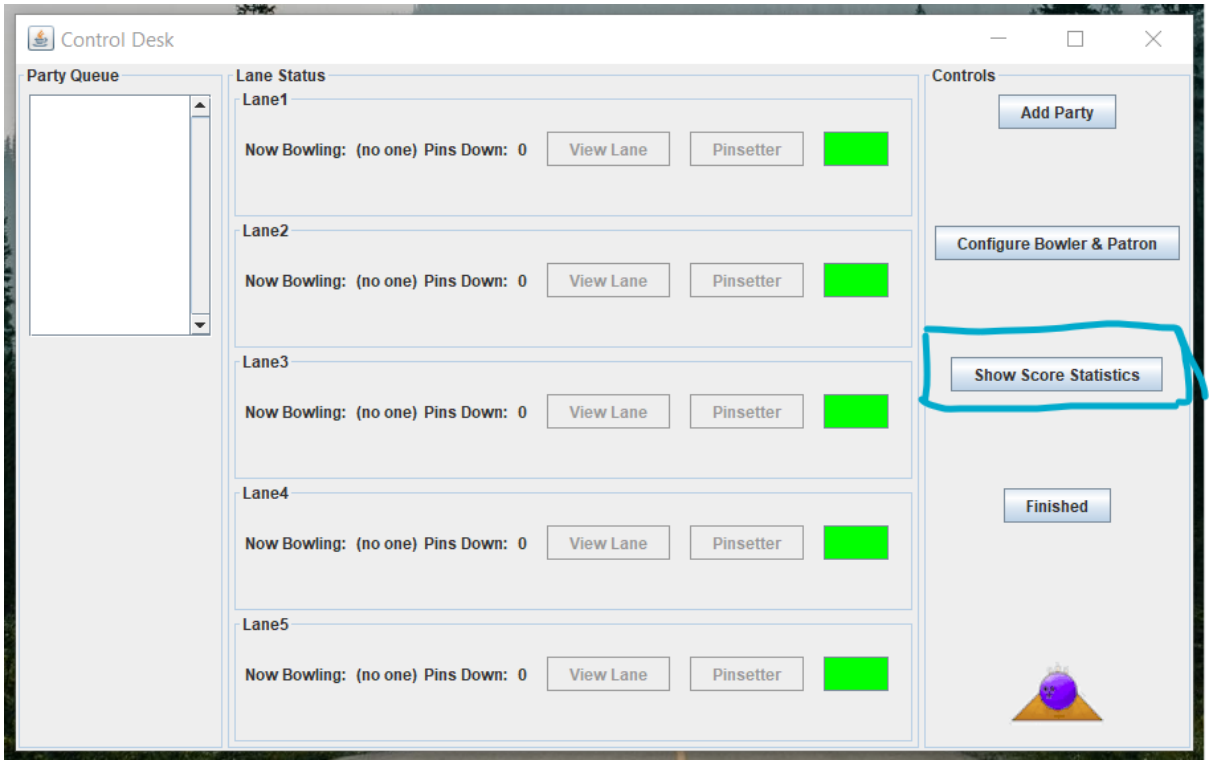
We have given a Button named `Show Score Statistics` in the window screen.

Code Snippets:

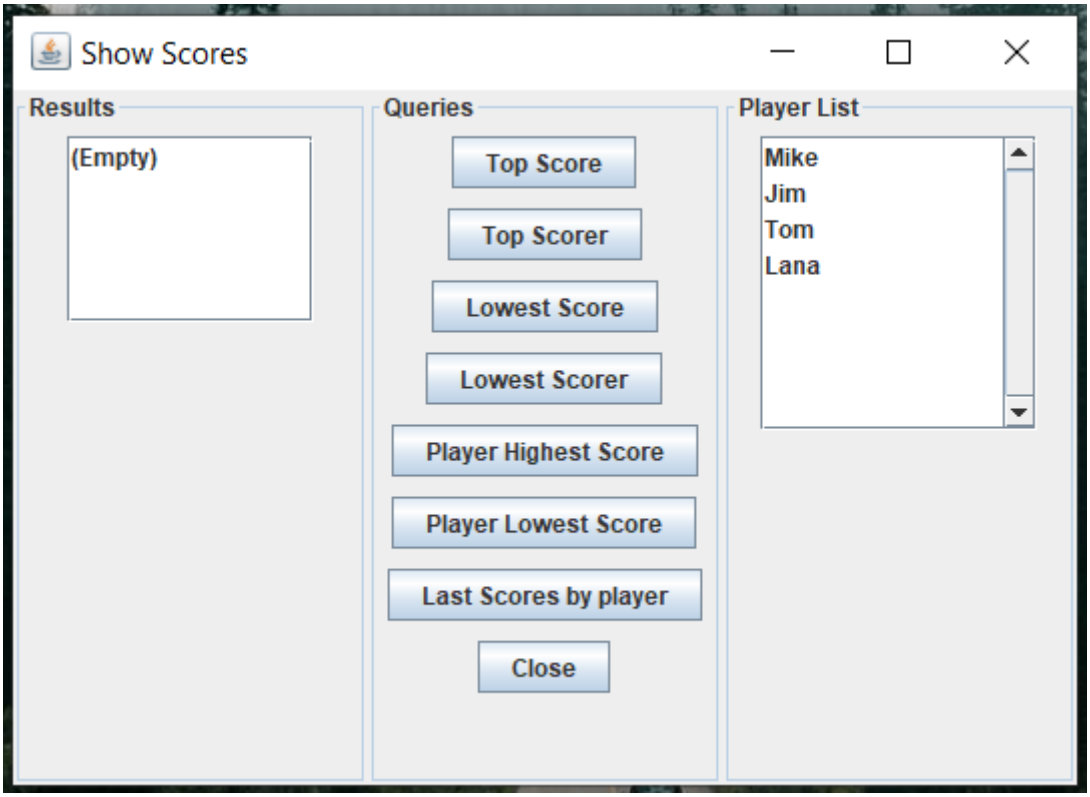
```
public class ShowScoreStats implements ActionListener, ListSelectionListener {

    private JFrame win;

    private JButton finished,lastScores,maxPlayerScore,minPlayerScore;
    private JButton minScore,maxScore,TopScorer,LowestScorer;

    private JList<Vector> allBowlers;
    private JList<Vector> outputList;

    private Vector party;
    private Vector bowlerdb;

    private String selectedNick;

    public ShowScoreStats() {

        win = new JFrame("Show Scores");
        win.getContentPane().setLayout(new BorderLayout());
        JPanel newTestPanel = new JPanel();
        ((JPanel) win.getContentPane()).setOpaque(false);

        JPanel colPanel = new JPanel();
        JPanel newTestPanel2 = new JPanel();
        colPanel.setLayout(new GridLayout(1, 3));

        JPanel newTestPaneli = new JPanel();

        // Controls Panel
        JPanel controlsPanel = new JPanel();
        controlsPanel.setLayout(new GridLayout(9, 2));
        JPanel newTestPanel3 = new JPanel();
        controlsPanel.setBorder(new TitledBorder("Queries"));
```

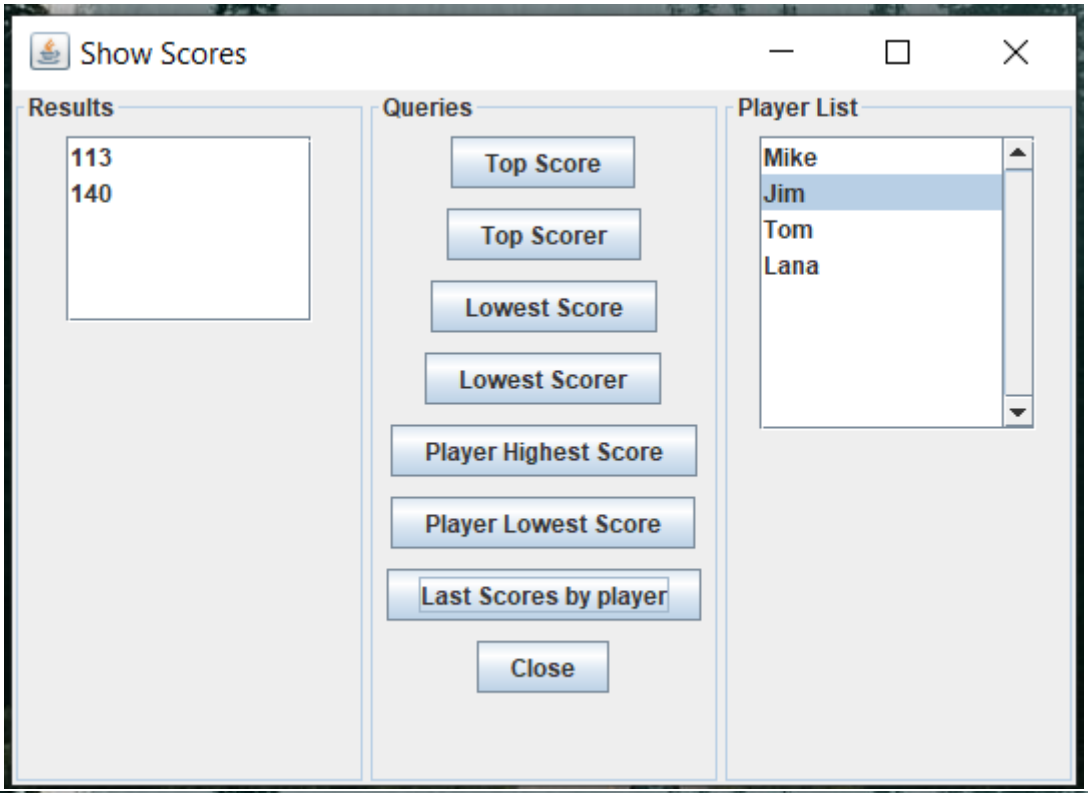
```
public void actionPerformed(ActionEvent e) {
    if(e.getSource().equals(maxScore)) {
        System.out.println("in maxScore");
        party.clear();
        party.add(CreateDB.obtainUniversalMaxScore());
        outputList.setListData(party);
    }

    if(e.getSource().equals(minScore)) {
        System.out.println("in minScore");
        party.clear();
        party.add(CreateDB.obtainUniversalMinScore());
        outputList.setListData(party);
    }

    if (e.getSource().equals(minPlayerScore)) {
        System.out.println("in minPlayerScore");
        if (selectedNick != null) {
            party.clear();
            party.add(CreateDB.obtainMinScoreByPlayer(selectedNick));
            outputList.setListData(party);
        }
    }

    if (e.getSource().equals(maxPlayerScore)) {
        System.out.println("in maxPlayerScore");
        if (selectedNick != null) {
            party.clear();
            party.add(CreateDB.obtainMaxScoreByPlayer(selectedNick));
            outputList.setListData(party);
        }
    }

    if (e.getSource().equals(TopScorer)) {
        System.out.println("in TopScorer");
        party.clear();
        String topScorer = CreateDB.obtainTopScorer();
        party.add(topScorer);
```

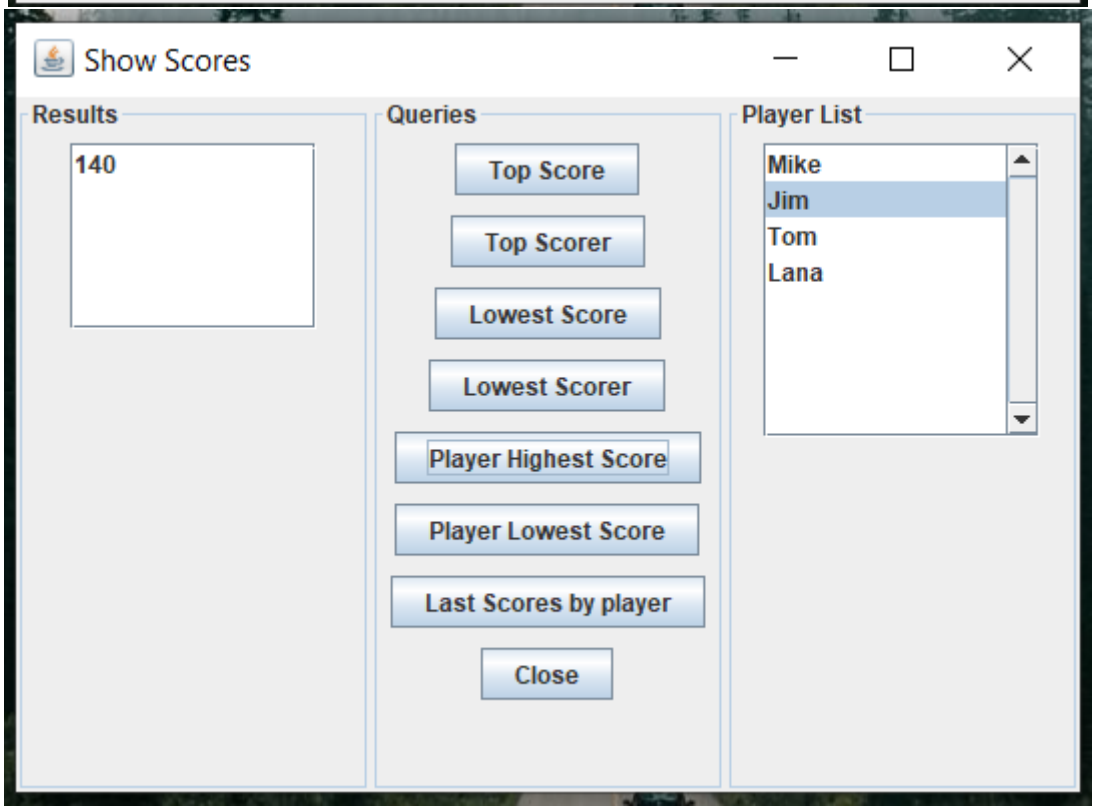
Working samples:

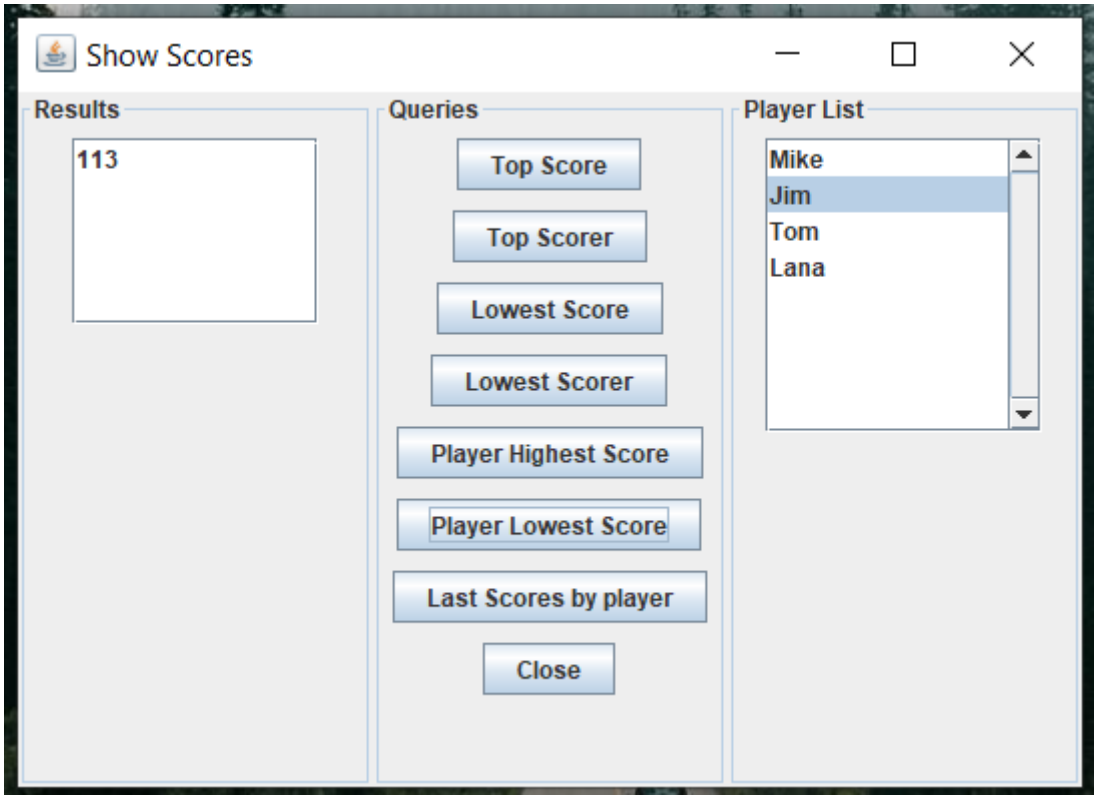


By clicking it a new window will pop up to show a list of adhoc queries such as top score, top scorer, lowest score, lowest scorer, player wise highest score, player wise lowest score, last scores by a player etc. A new class called ` ShowScoreStats ` has been created to implement this functionality of adhoc queries.




Show Scores
Results
113
140
Queries
Top Score
Top Scorer
Lowest Score
Lowest Scorer
Player Highest Score
Player Lowest Score
Last Scores by player
Close
Player List
Mike
Jim
Tom
Lana


Show Scores
Results
140
Queries
Top Score
Top Scorer
Lowest Score
Lowest Scorer
Player Highest Score
Player Lowest Score
Last Scores by player
Close
Player List
Mike
Jim
Tom
Lana



Design Patterns used: While implementing this requirement, the **Decorator Pattern** was utilized. The decorator pattern dynamically adds new responsibilities to an object and is a more flexible alternative to subclassing for adding functionality. Score is the object that we decorate every time we want to acquire the desired output as a query result in this scenario. When a game is completed, the results of the relevant query are dynamically updated.

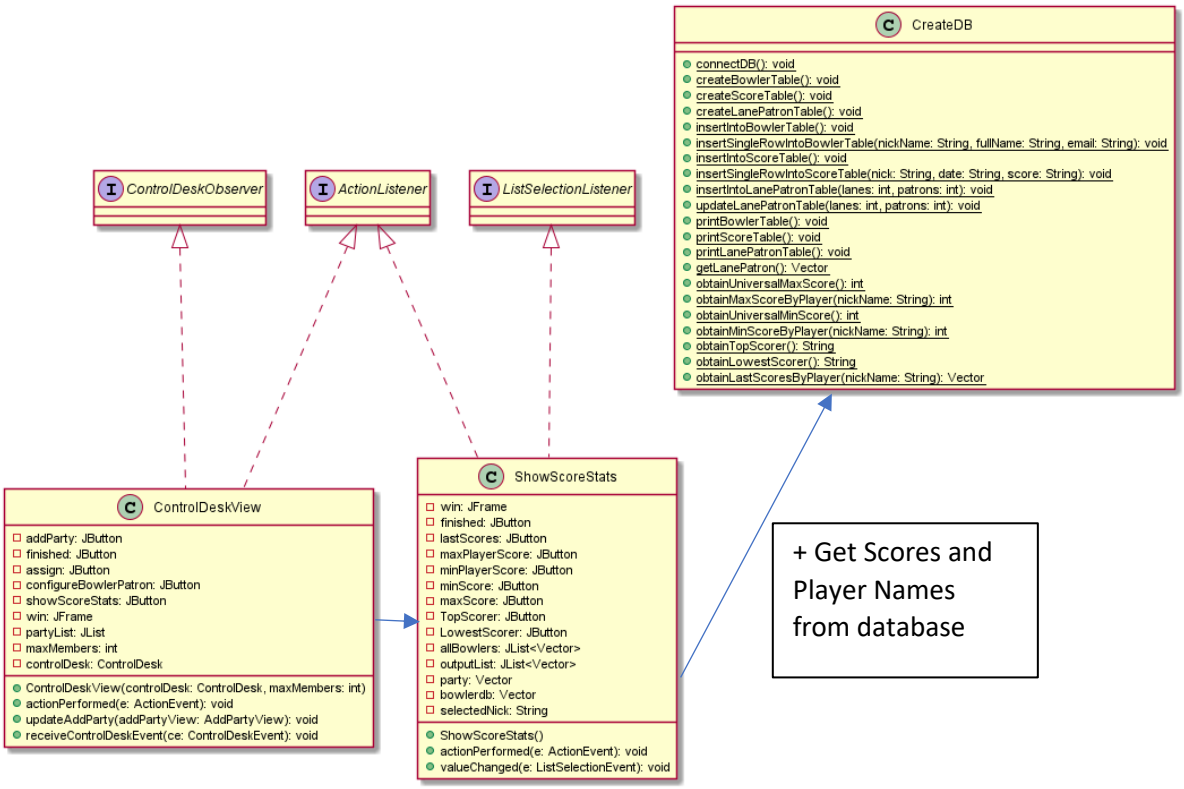
UI Patterns used: The **Breadcrumbs** pattern was utilized to create the user interface for this requirement. As part of this, when the user requests it, a query window will show up for improved visibility, and the user will swiftly perform a query to obtain the needed information. In addition, the Clear Primary Actions pattern has been used to clearly and individually present the query that a user might execute.

New Class description:

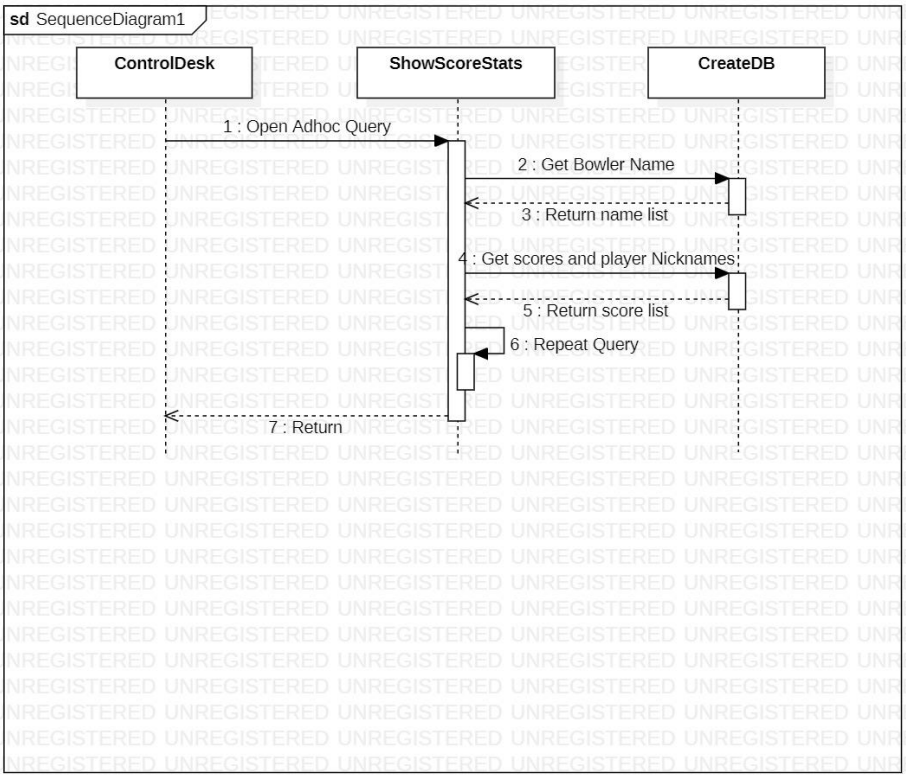
Class Name	Description
ShowScoreStats	Returns results for a variety of queries including top score, top scorer, lowest score, lowest

	scorer, player wise highest score, player wise lowest score, last scores by a player.

Class Diagram:



Sequence Diagram:



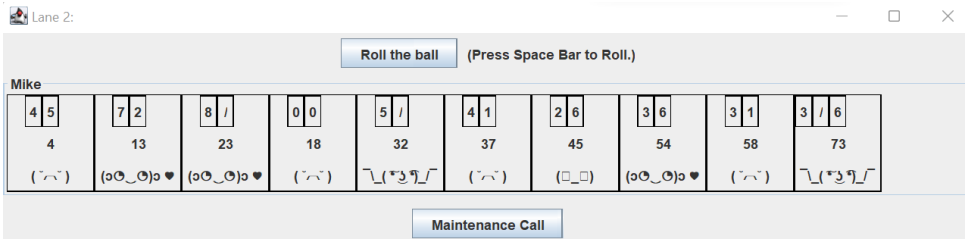
5. Implementing penalty for Gutters:

Gutter is taken as the case when the number of pins down is 0. If there are two consecutive Gutters in one frame, then a penalty must be applied. If this happens in the first frame, then half of the score of the next frame must be deducted. If this happens in an intermediate frame, then half of the score of the max score of all the frames must be deducted.

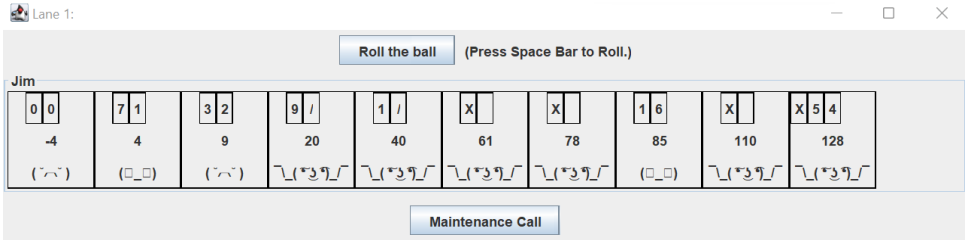
This feature has been implemented in ScoreCalculator.java class.

```
for (int i = 0; i != current + 2; i++) {
    //Two consecutive Gutters handling
    if(i%2==1 && i<19 && curScore[i-1]==0 && curScore[i]==0)
    {
        //If two consecutive gutters occur in the middle
        if(i>1)
        {
            int cur_ind=i-1, max_so_far = 0;
            //Find the max_score of all the frames
            for(int j=0;j<cur_ind;j+=2)
            {
                if((curScore[j]+curScore[j+1])>max_so_far)
                {
                    max_so_far=curScore[j]+curScore[j+1];
                }
            }
            //System.out.println("max_so_far:"+max_so_far);
            double decr_score=(0.5)*(max_so_far);
            cumulScores[bowlIndex][(i/2)]-=(int)decr_score;
        }
        //If two consecutive gutters occur at the start
        else
        {
            //Decrement half of the points of next frame
            if(i<current-1)
            {
                double decr_score=(0.5)*(curScore[i+1]+curScore[i+2]);
                //System.out.println("decr_score:"+decr_score);
                cumulScores[bowlIndex][(i/2)]-=(int)decr_score;
            }
        }
```

Below is the case where two consecutive Gutters have occurred in the 4th frame. The highest score until 4th frame is 10. Since two consecutive Gutters have occurred, a score of 5 is reduced.



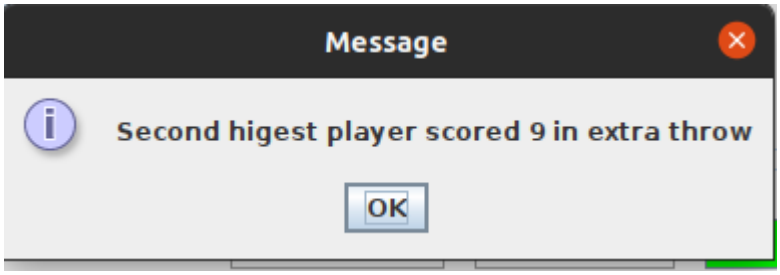
Below is the case where two consecutive Gutters have occurred in the 1st frame. The score in the second frame is 8. Since two consecutive Gutters have occurred, a score of 4 is reduced.



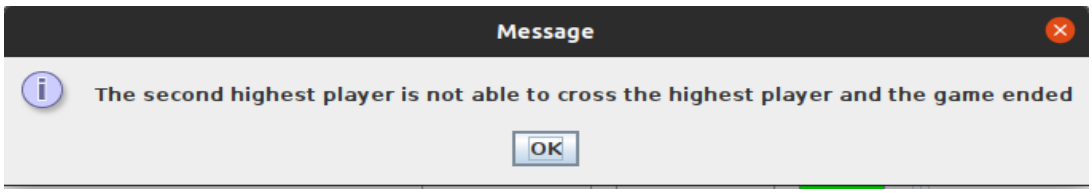
6. Tie Breaker

At the end of 10th Frames, we will provide one more chance to 2nd highest player.

Extra frame window:

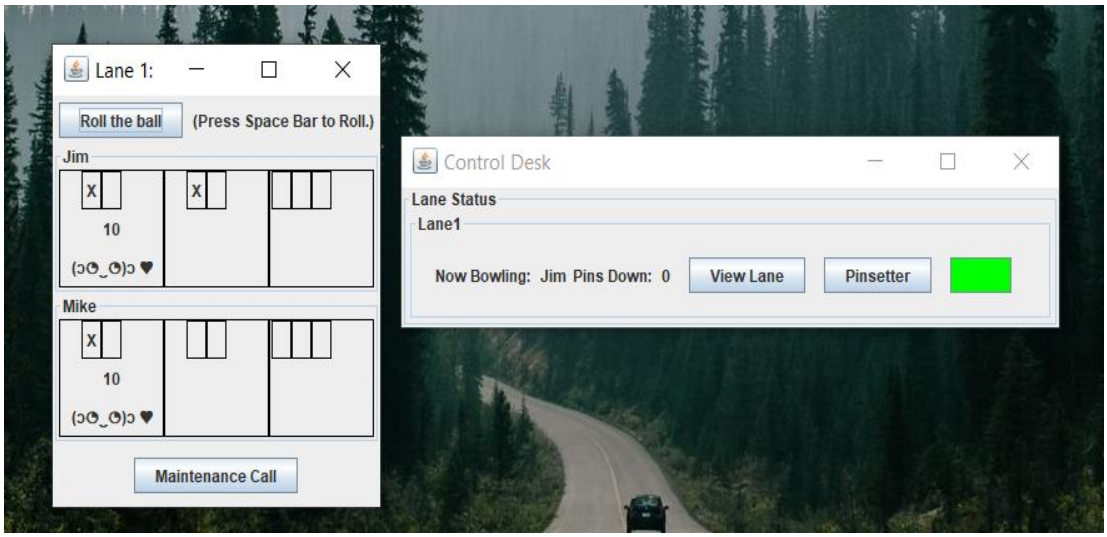


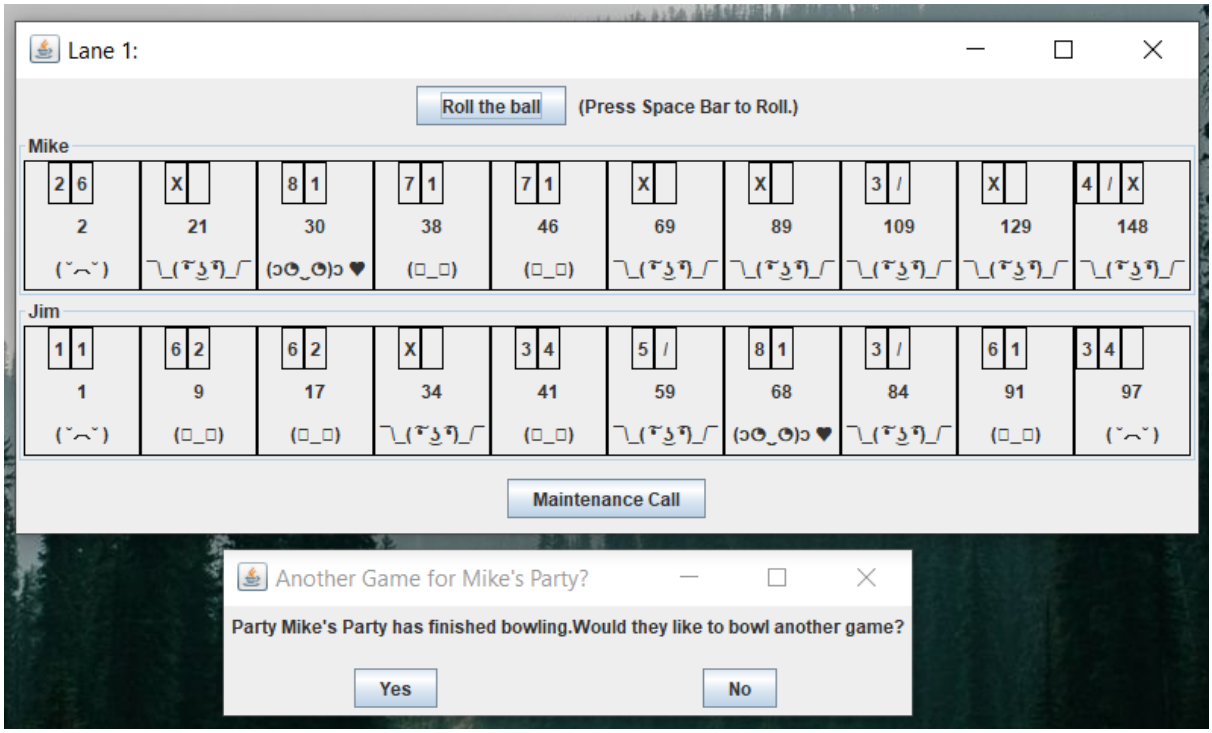
If 2nd highest player is unable to score more than 1st highest player then we will display a message like this.



If he is able to cross then the game will be continued with 3 more frames between those two players.

Playing with an extra 3 frames:





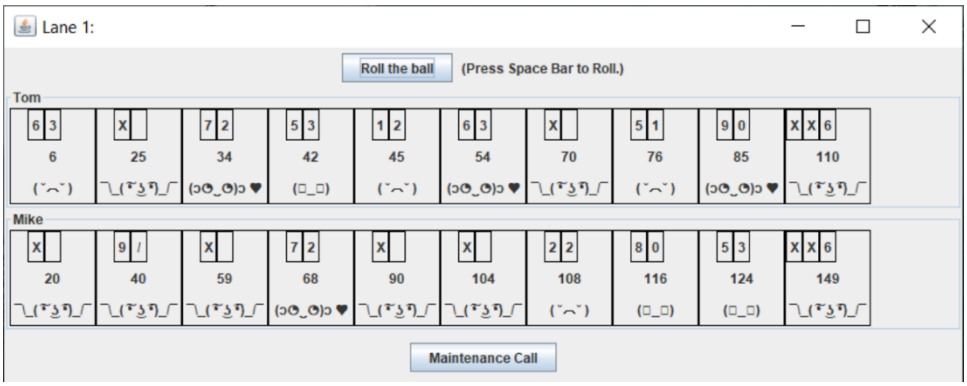
7. Implementing an emoticon based on player score:

We have added emoticons to indicate player reactions when a score is obtained.

- If the score for a particular frame is less than 6, then " (˘ ͜ʖ ˘) " is displayed.
- If the score for a particular frame is between 6 and 8, then " (ಠ_ಠ) " is displayed.
- If the score for a particular frame is between 8 and 10, then " (ↄ◔‿◔)ↄ ♥ " is displayed.
- If the score is not in any of the above ranges, then display " ¯_(ツ)_/¯ ".

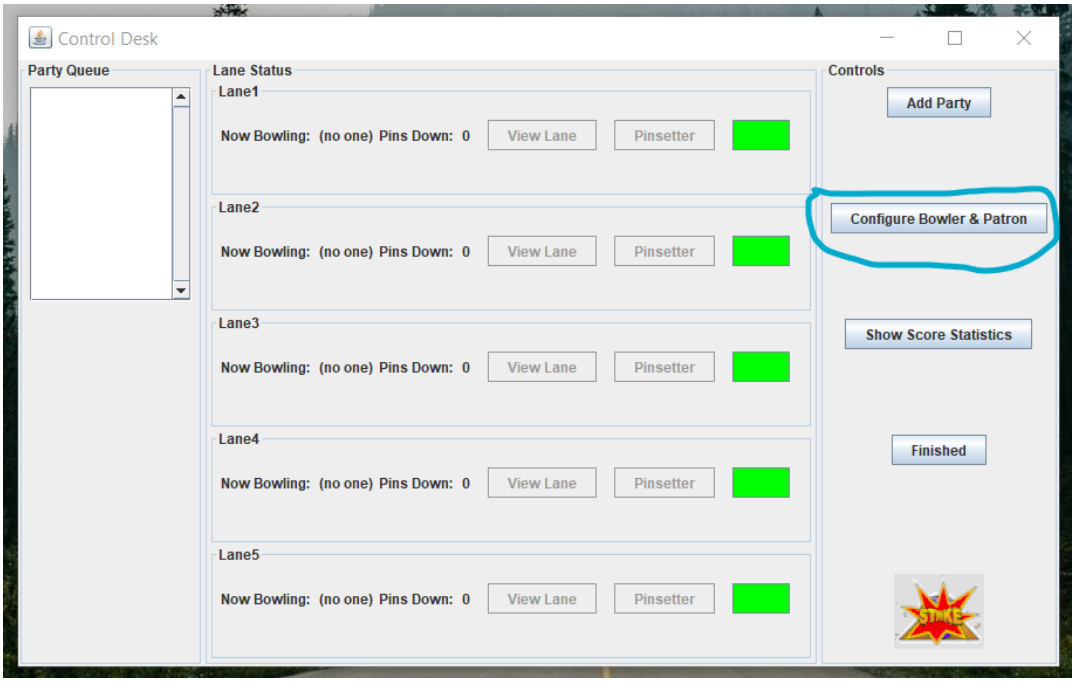
The following **User Interface Design Patterns** come into play in the bowling alley simulator:

1) **Competition:** We show the scores of each player next to each other thus it can be easily compared.
2) **Sunk Cost Effect and Self-monitoring:** The cumulative score after each frame nudges the player to continue playing the game rather than stopping in between. It also tracks the players performance across each frame and gives feedback to change their playing style.
3) **Praise:** When 8 or more pins have fallen we display the "(ɔ◔‿◔)ɔ ♥" emoticon, to encourage the player to try harder.
4) **Negativity bias and loss aversion:** When less than 8 pins have fallen, we display a neutral or sad expression. This takes advantage of our bias towards paying more attention to a negative experience and our desire to avoid loss.



8. All the numbers quoted and existing in the code should be made configurable:

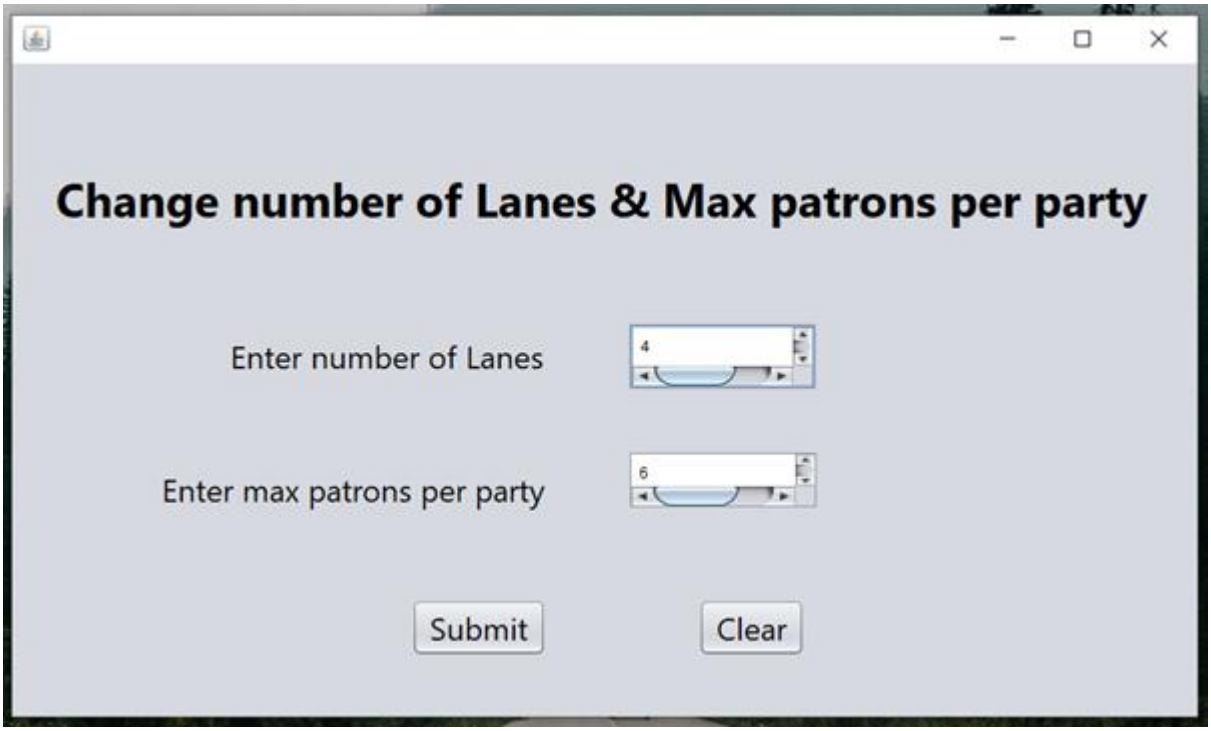
As mentioned in point no. 2, each game's number of lanes and maximum number of players can be customized. This procedure is performed via a JButton titled 'Configure Bowler & Patron.'



When you click that button, a new window will open with the options to enter the number of lanes and the maximum number of patrons per party. The modifications will be saved in the database after you enter the desired settings, and they will be reflected in the following game.

```
private void initComponents() {

    jLabel1 = new javax.swing.JLabel();
    jLabel2 = new javax.swing.JLabel();
    jLabel3 = new javax.swing.JLabel();
    jScrollPane1 = new javax.swing.JScrollPane();
    patrons = new javax.swing.JTextArea();
    jScrollPane2 = new javax.swing.JScrollPane();
    lanes = new javax.swing.JTextArea();
    submit = new javax.swing.JButton();
    clear = new javax.swing.JButton();

    setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);

    jLabel1.setFont(new java.awt.Font("Segoe UI", 1, 36)); // NOI18N
    jLabel1.setText("Change number of Lanes & Max patrons per party");

    jLabel2.setFont(new java.awt.Font("Segoe UI", 0, 24)); // NOI18N
    jLabel2.setText("Enter number of Lanes");
```

```
private void submitActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    int numOfLanes = Integer.parseInt(lanes.getText());
    int numOfPatrons = Integer.parseInt(patrons.getText());

    CreateDB.updateLanePatronTable(numOfLanes, numOfPatrons);
}

private void clearActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    lanes.setText("");
    patrons.setText("");
}

/**
```



9. UI Patterns used in the new design at a glance:

Clear Primary Actions user interface design pattern has been used, “the roll button” is the primary button on top and there is also added information “Press Space Bar to Roll” to show visually guide the user on the course of action.

Competition: We show the scores of each player next to each other thus it can be easily compared.

Sunk Cost Effect and Self-monitoring: The cumulative score after each frame nudges the player to continue playing the game rather than stopping in between. It also tracks the players' performance across each frame and gives feedback to change their playing style.

Praise: When 8 or more pins have fallen, we display the “(ɔ◔‿◔)ɔ ♥” emoticon, to encourage the player to try harder.

Negativity bias and loss aversion: When less than 8 pins have fallen, we display a neutral or sad expression. This takes advantage of our bias towards paying more attention to a negative experience and our desire to avoid loss.

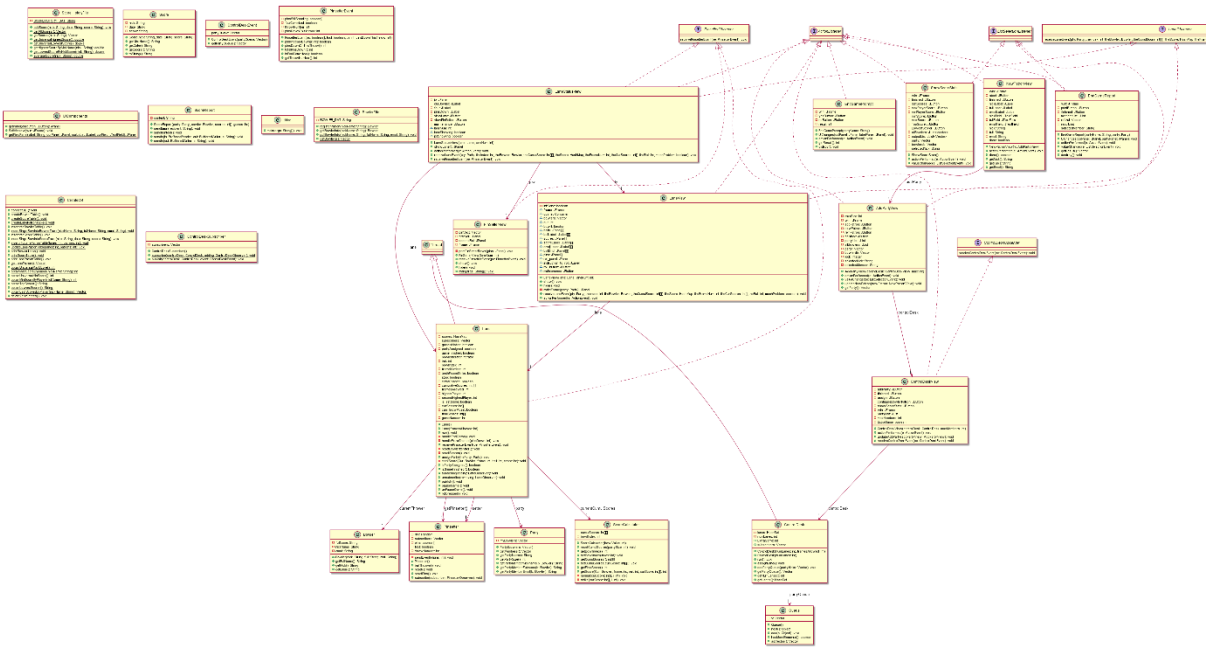
The **Breadcrumbs** pattern was utilized to create the user interface. As part of this, when the user requests it, a query window will show up for improved visibility, and the user will swiftly perform a query to obtain the needed information. In addition, the Clear Primary Actions pattern has been used to clearly and individually present the query that a user might execute.

10. Design patterns used in new design at a glance:

The **observer pattern** has been used to implement the “roll the ball” button functionality. When the “roll the ball” button is clicked laneView Class notifies Lane to roll the ball.

While implementing the show scores window, the **Decorator Pattern** was utilized. The decorator pattern dynamically adds new responsibilities to an object and is a more flexible alternative to subclassing for adding functionality. Score is the object that we decorate every time we want to acquire the desired output as a query result in this scenario. When a game is completed, the results of the relevant query are dynamically updated.

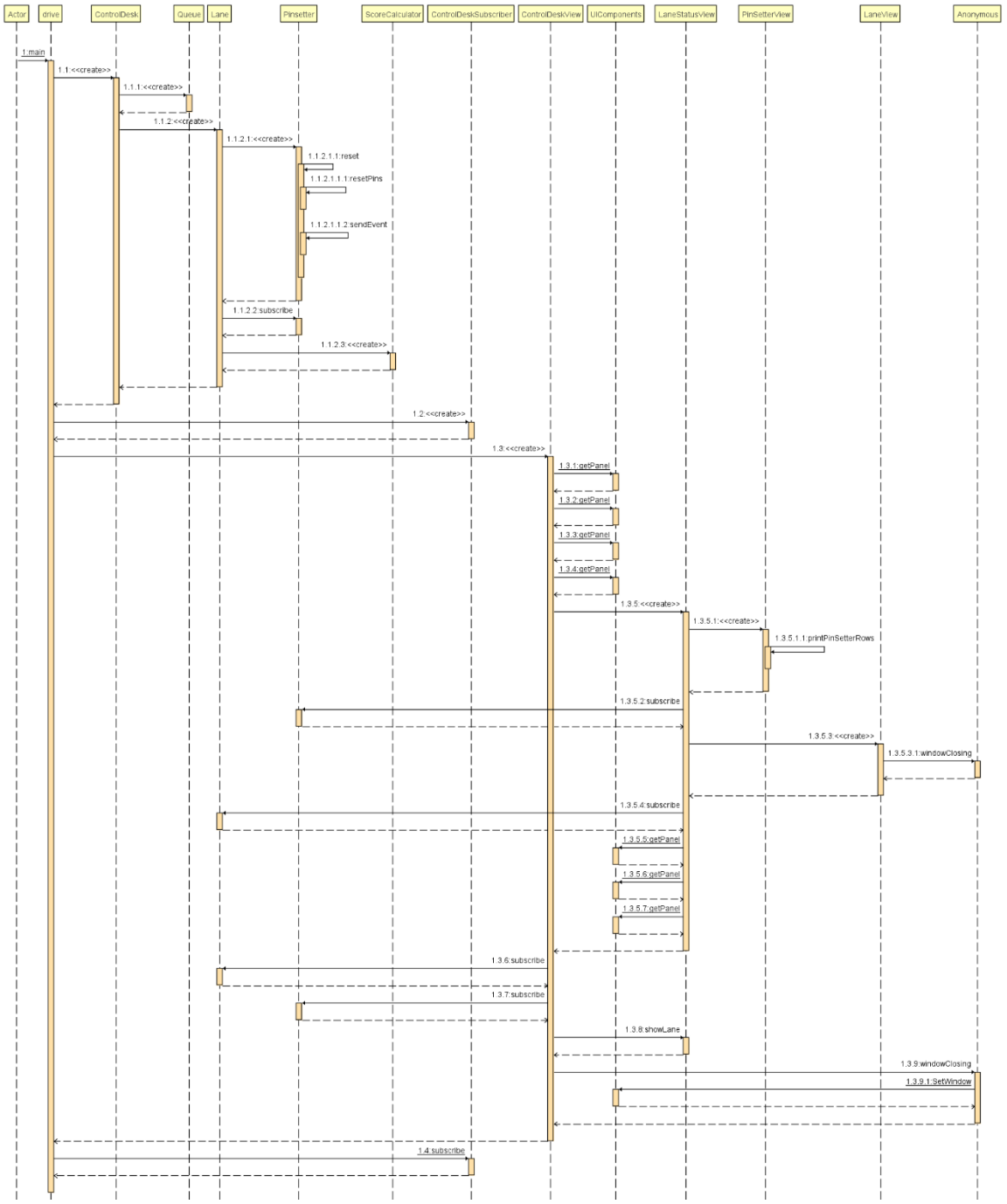
11. UML Class diagram after adding new features:



[Click here to view the image]

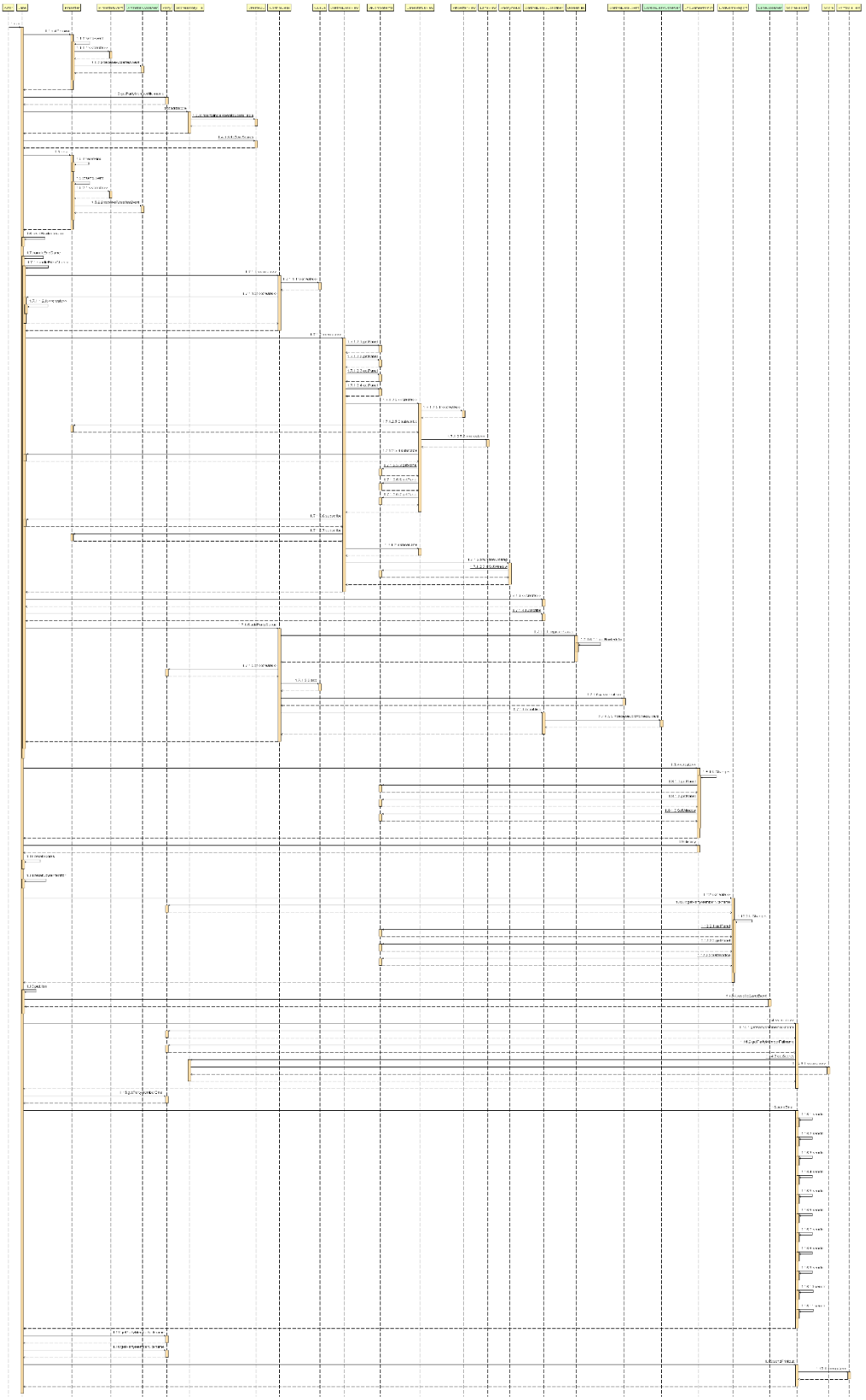
12. UML Sequence diagram after adding new features:

Sequence Diagram of main() method in Drive.java.



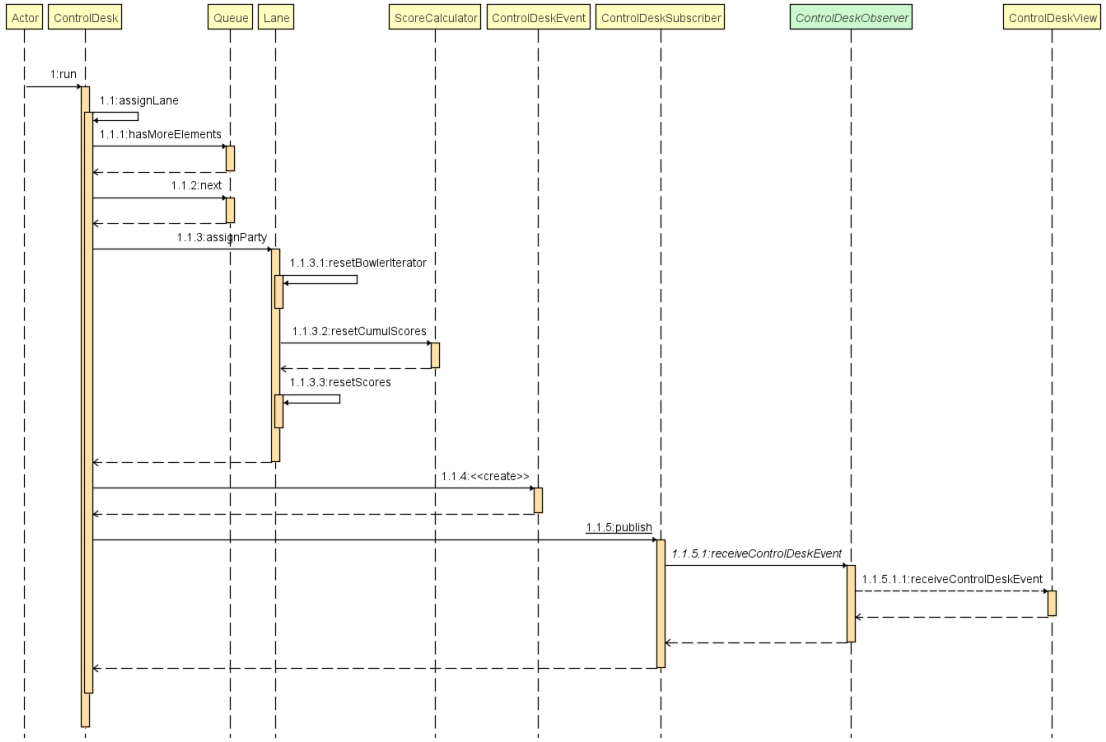
[Click here to view the image]

Sequence Diagram of run() method in Lane.java



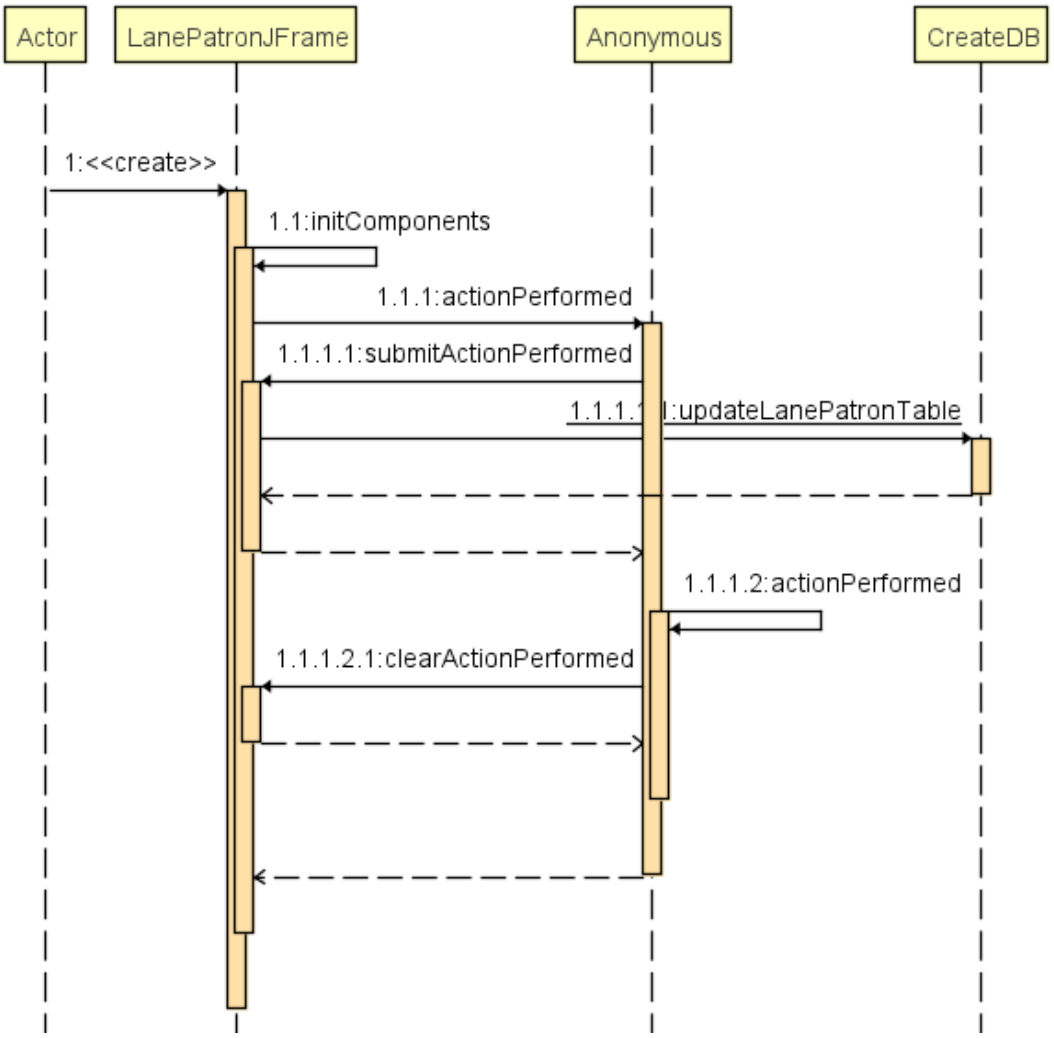
Click here to view the image

Sequence Diagram of run() method in ControlDesk.java.



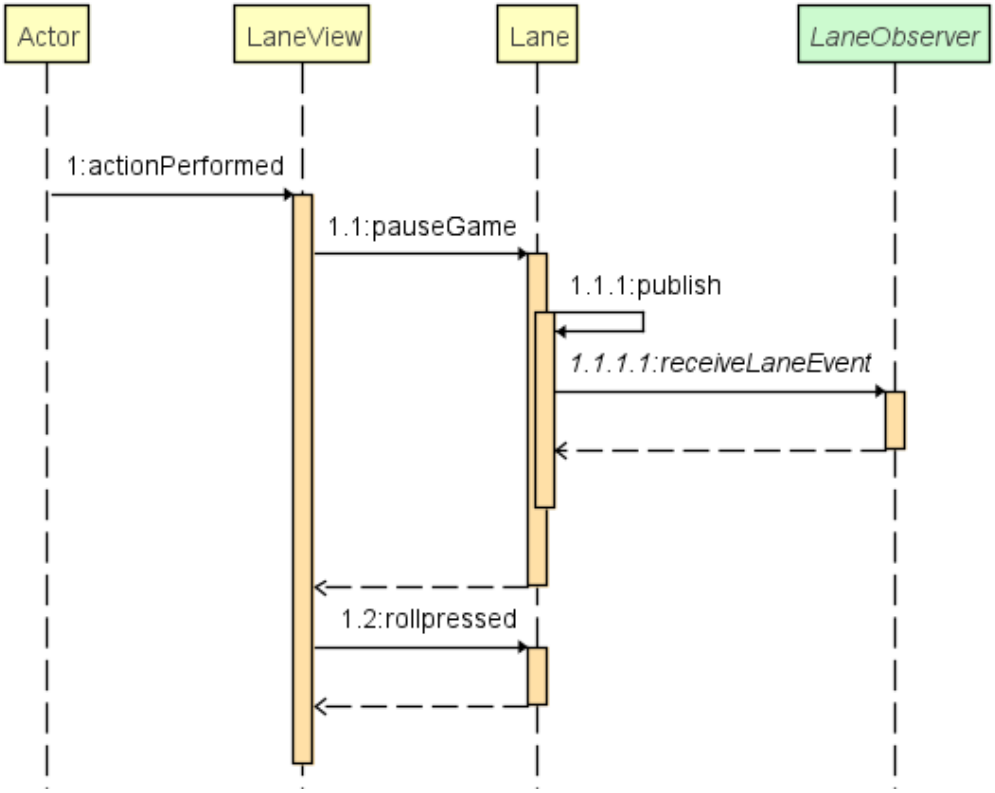
Click here to view the image

Sequence Diagram of initcomponents() method in LanePatronJFrame.java.



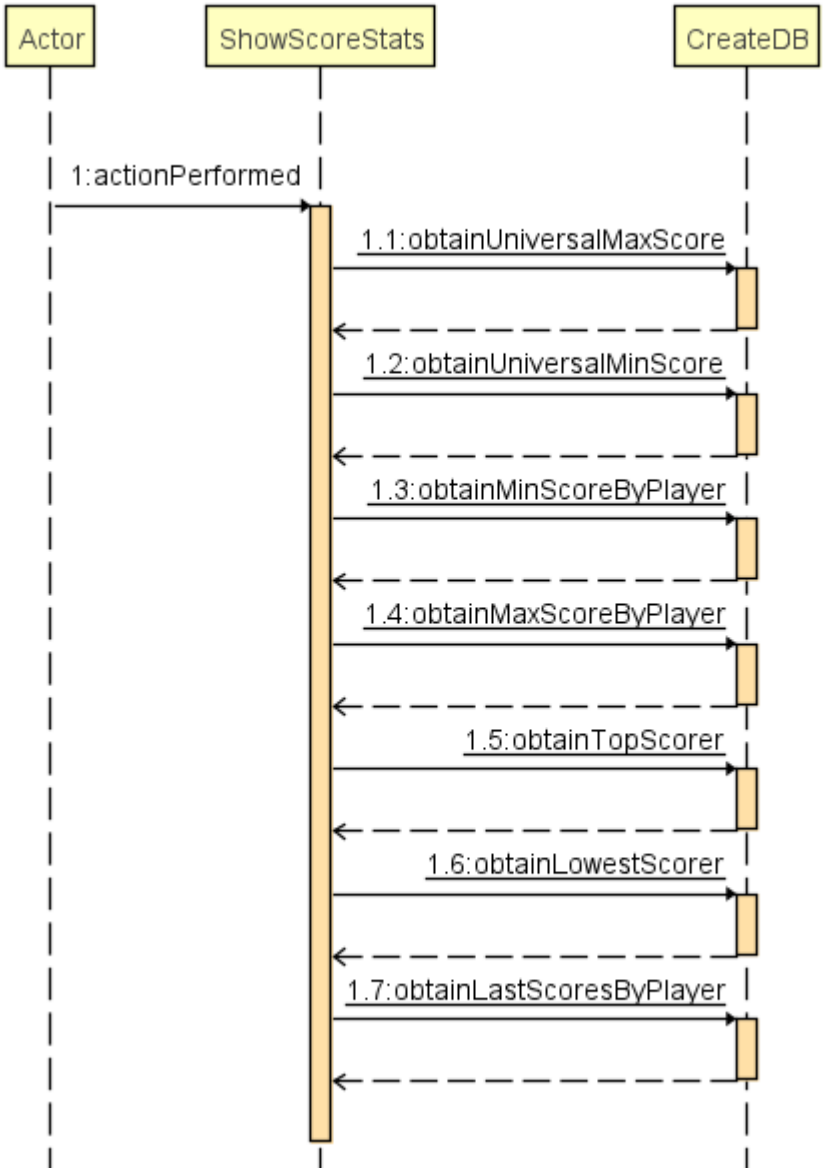
Click here to view the image

Sequence Diagram of actionPerformed () method in LaneView.java.



Click here to view the image

Sequence Diagram of actionPerformed () method in ShowScoreStats.java.



Click here to view the image

13. Quick metric Analysis after adding new features:

Analysis of BowlingAlleySimulation

General Information

Total lines of code: 2341

Number of classes: 31

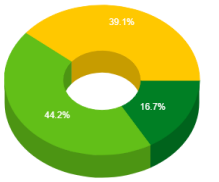
Number of packages: 1

Number of external packages: 31

Number of external classes: 238

Number of problematic classes: 0

Number of highly problematic classes: 0

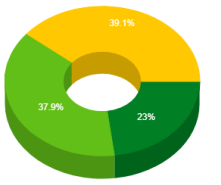


C3

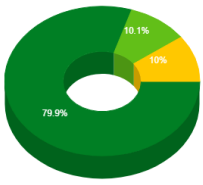
- Very High
- High
- Medium-high
- Low-medium
- Low

Distribution of Quality Attributes

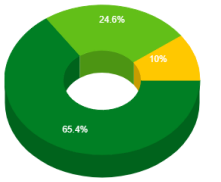
Complexity, Coupling, Cohesion, and Size



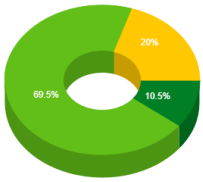
Complexity



Coupling



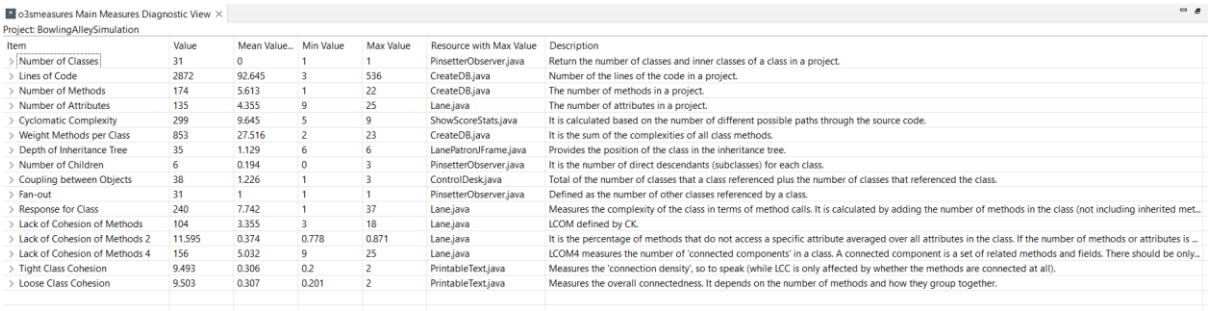
Lack of Cohesion



Size

List of all classes (#31)

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	Lane					235	medium-high	medium-high	medium-high	low-medium
2	ControlDeskView					96	low-medium	low-medium	low-medium	low-medium
3	ControlDesk					56	low-medium	low-medium	low	low-medium
4	LaneStatusView					84	low	low-medium	low-medium	low-medium
5	CreateDB					469	medium-high	low	low	medium-high
6	LanePatronJFrame					129	medium-high	low	low	low-medium
7	ScoreCalculator					82	medium-high	low	low	low-medium
8	ShowScoreStats					201	low-medium	low	low	low-medium
9	LaneView					150	low-medium	low	low-medium	low-medium
10	ScoreHistoryFile					138	low-medium	low	low	low-medium
11	AddPartyView					109	low-medium	low	low-medium	low-medium
12	PinSetterView					74	low-medium	low	low-medium	low-medium
13	NewPatronView					64	low-medium	low	low	low-medium
14	ScoreReport					76	low	low	low	low-medium
15	BowlerFile					69	low	low	low	low-medium
16	EndGameReport					63	low	low	low-medium	low-medium
16	EndGameReport					63	low	low	low-medium	low-medium
17	Pinsetter					49	low	low	low	low
18	EndGamePrompt					46	low	low	low	low
19	PinsetterEvent					25	low	low	low	low
20	PrintableText					21	low	low	low	low
21	UIComponents					17	low	low	low	low
22	Party					16	low	low	low	low
23	Score					16	low	low	low	low
24	Bowler					14	low	low	low	low
25	Queue					12	low	low	low	low
26	drive					9	low	low	low	low
27	ControlDeskSubscr...					9	low	low	low	low
28	ControlDeskEvent					6	low	low	low	low
29	ControlDeskObserver					2	low	low	low	low
30	LaneObserver					2	low	low	low	low
31	PinsetterObserver					2	low	low	low	low



o3smeasures Main Measures Diagnostic View

Project: BowlingAlleySimulation

Item	Value	Mean Value...	Min Value	Max Value	Resource with Max Value	Description
Number of Classes	31	0	1	1	PinsetterObserver.java	Return the number of classes and inner classes of a class in a project.
Lines of Code	2872	92.645	3	536	CreateDB.java	Number of the lines of the code in a project.
Number of Methods	174	5.613	1	22	CreateDB.java	The number of methods in a project.
Number of Attributes	135	4.355	9	25	Lane.java	The number of attributes in a project.
Cyclomatic Complexity	299	9.645	5	9	ShowScoreStats.java	It is calculated based on the number of different possible paths through the source code.
Weight Methods per Class	853	27.516	2	23	CreateDB.java	It is the sum of the complexities of all class methods.
Depth of Inheritance Tree	35	1.129	6	6	LanePatronJFrame.java	Provides the position of the class in the inheritance tree.
Number of Children	6	0.194	0	3	PinsetterObserver.java	It is the number of direct descendants (subclasses) for each class.
Coupling between Objects	38	1.226	1	3	ControlDesk.java	Total of the number of classes that a class referenced plus the number of classes that referenced the class.
Fan-out	31	1	1	1	PinsetterObserver.java	Defined as the number of other classes referenced by a class.
Response for Class	240	7.742	1	37	Lane.java	Measures the complexity of the class in terms of method calls. It is calculated by adding the number of methods in the class (not including inherited met...
Lack of Cohesion of Methods	104	3.355	3	18	Lane.java	LCOM defined by CK.
Lack of Cohesion of Methods 2	11.595	0.374	0.778	0.871	Lane.java	It is the percentage of methods that do not access a specific attribute averaged over all attributes in the class. If the number of methods or attributes is ...
Lack of Cohesion of Methods 4	156	5.032	9	25	Lane.java	LCOM4 measures the number of 'connected components' in a class. A connected component is a set of related methods and fields. There should be only...
Tight Class Cohesion	9.493	0.306	0.2	2	PrintableText.java	Measures the 'connection density', so to speak (while LCC is only affected by whether the methods are connected at all).
Loose Class Cohesion	9.503	0.307	0.201	2	PrintableText.java	Measures the overall connectedness. It depends on the number of methods and how they group together.

Click here to view the image

- Lines of Code (LoC):

Lines of Code is a traditional metric which is nothing but the total number of lines of code written.

The LoC metric value after implementing new features is 2872.

- Cyclomatic Complexity:

This metric is used to measure the complexity of the code. It is calculated by counting the number of independent paths in the Control Flow Graph.

The Cyclomatic complexity of the codebase after implementing the new features is 299.

- Coupling between Objects:

Coupling is used to measure the dependency between classes, modules and methods. The lower the coupling is the better the design. If the coupling is low, then the individual entities can be modified easily.

The coupling for the current codebase is 38.

- Cohesion:

Cohesion metric is used to measure how closely the methods of a class are related to each other. High cohesion is preferred because it reduces code complexity and increases maintainability as all the related methods will be placed with each other.

The Cohesion for the codebase is extremely high/high/intermediate for all the classes.