
Design Document

Software Engineering

Unit 2 Project - Bowling Alley (Enhancement)

TEAM - 22

Date of submission: 22-March-2022

Table of Contents

New Requirements	4
Architectural Patterns	23
Design Patterns	24
Sequence Diagrams	31
Class Responsibility Table	37
Original Design	38
Weakness	38
Code Smells	38
Anti-Patterns	39
Strengths	40
Fidelity to the Design Document	41
Narrative of the refactored design	41
Changes made in the code	41
Metrics	42
Before Refactoring	42
After Refactoring	44

Team Members and their Roles

Roll no.	Name	Work Hours	Role
2021201061	Adidala Divishath Reddy	16 hours	Implemented Penalty for Gutters - If a player bowls two consecutive gutters, he or she shall be penalized 1/2 of the highest score. If the first two instances occur at the beginning of the game, the player will be fined 1/2 of the points scored in the subsequent session. Did the related refactoring.
2021201062	Mainak Dhara	18 hours	Made the game interactive, Implemented the Database layer and ad-hoc queries like top player , top score, minimum score, Implemented the ScoreSearch,ScoreSearchView,Made the quoted and existing numbers inside code configurable through drop downs and did the related refactorings.
2021201063	Jasika Shah	17 hours	Implemented the feature that gives the 2 nd top player a chance to bowl at the end of 10 frames. If the player becomes the highest, play 3 more frames between the 1 st and 2 nd highest players until the winner is determined. If there is a tie, the winner will be determined by who had the most strikes. Did the related refactoring.
2020201023	Akshay M	15 hours	Implemented all different emoticons which based on different ranges of score and popup is shown to the user , did the related refactorings.

The project made according to new requirements can be found in the “New_Requirements” folder inside this [git repo](#).

New Requirements

Feature 1

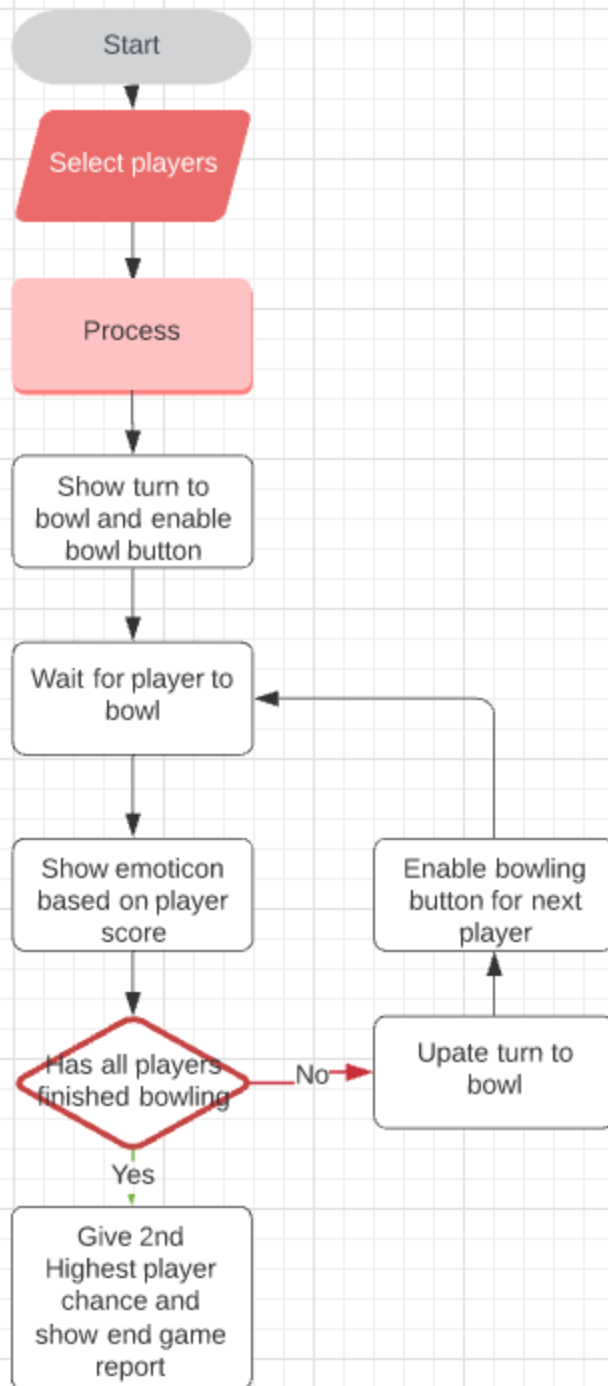
Making the game interactive:

In the previous version of the code the whole of ball throwing simulation was done in an automated manner where there was no user interactivity possible so in order to make the game interactive we implemented a separate bowling button for each player in a particular lane and this button is only enabled for a particular player when it's their turn to bowl so the flow of the bowling game is completely in the hands of the participants. And we also made a different panel for showing the name of the player whose turn is to bowl. That makes the overall design more easy to understand.

Here we used the **Observer** design pattern is used, whenever the “Bowl” button is pressed the LaneEventObserver is notified and the score calculation is carried out then the results are shown through the LaneView

We have also used the **Progressive Disclosure** UI pattern where only the information related to the current bowling game is shown to the user so there is no confusion while playing the game. **Clear Primary Actions** patterns are also being used where a label is maintained which shows the name of the current user's turn to bowl and the button to bowl is only enabled for the player whose turn it is to bowl so no other can intervene in his or her bowling.

Flow is shown in the below diagram



Feature 2

Increasing maxPatronsPerParty to 6:

Here we needed to increase the maxPatronsPerParty to 6 which required change in a single variable so there were no major changes involved.

Feature 3

Adding database layer and implementing ad-hoc queries:

Database layer has been implemented which can be used to store and access scores and players. A new view ScoreSearchView has been implemented which takes care of all the queries. Overall Minimum, Overall Maximum, Player Minimum, Player Maximum, Player Least 5 scores, Player Top 5 scores, Player Last 5 Scores, Last 10 scores, Top 10 Highest scores, Top Player are some of the queries that are implemented. All implementation logic regarding these queries are implemented inside of the ScoreSearch class.

When implementing this requirement, the **Decorator** Pattern was employed. The decorator approach dynamically adds additional responsibilities to an object and is a more flexible alternative to subclassing for adding functionality. All the required queries are added to a different view which can be accessed as per need. After completion of a game.

Leaderboard and **Praise** UI patterns have been used which show players relative positions and perform better.

Query Scores

Bowler List

Mike
Jim
Tom
Lana
TomH
a
aaaa
aasfasf
fsdf

Overall Min Score

Overall Max Score

Player Minimum

Player Maximum

Player Least 5

Player Top 5

Player Last 5

Last 10 Scores

Top 10 Highest

Top Player

Finished

Results			
Lana	13:21	2/2/2022	111
Tom	13:21	2/2/2022	166
Jim	13:21	2/2/2022	95
Mike	13:21	2/2/2022	97
TomH	13:2	2/2/2022	103
Jim	13:2	2/2/2022	132
Jim	12:48	2/2/2022	143
Mike	12:48	2/2/2022	162
Lana	12:42	2/2/2022	144
Mike	12:42	2/2/2022	104

Figure: Query result using last 10 scores

Results			
Mike	13:21	2/2/2022	97
Mike	12:48	2/2/2022	162
Mike	12:42	2/2/2022	104
Mike	12:39	2/2/2022	162
Mike	12:36	2/2/2022	103

Figure: Query results using Player Last 5 scores

Results
193
189
183
181
178
175
175
174
173
170

Figure: Top 10 Highest scores


```

public static Vector<Integer> getPlayerMax(String fetchNick) throws IOException{
    Vector<Integer> scoreList=new Vector<>();
    BufferedReader scoreFileReader=new BufferedReader(new FileReader(SCOREHISTORY_FILE));
    String raw_data;
    while((raw_data=scoreFileReader.readLine())!=null){
        String[] scoreData=raw_data.split( regex: "\\t");
        if(Objects.equals(scoreData[0], fetchNick)){
            scoreList.add(Integer.parseInt(scoreData[2]));
        }
    }
    Collections.sort(scoreList,Collections.reverseOrder());
    scoreFileReader.close();
    return scoreList;
}

public static Vector<Score> getPlayerLastFive(String fetchNick) throws IOException{
    Vector<Score> scoreList=new Vector<Score>();
    BufferedReader scoreFileReader=new BufferedReader(new FileReader(SCOREHISTORY_FILE));
    String raw_data;
    while((raw_data=scoreFileReader.readLine())!=null){
        String[] scoreData=raw_data.split( regex: "\\t");
        if(Objects.equals(scoreData[0], fetchNick)){
            scoreList.add(new Score(scoreData[0],scoreData[1],scoreData[2]));
        }
    }
    Collections.reverse(scoreList);
    scoreFileReader.close();
    return scoreList;
}

```

Figure: Code for implementation of getPlayerMax and getPlayerLastFive

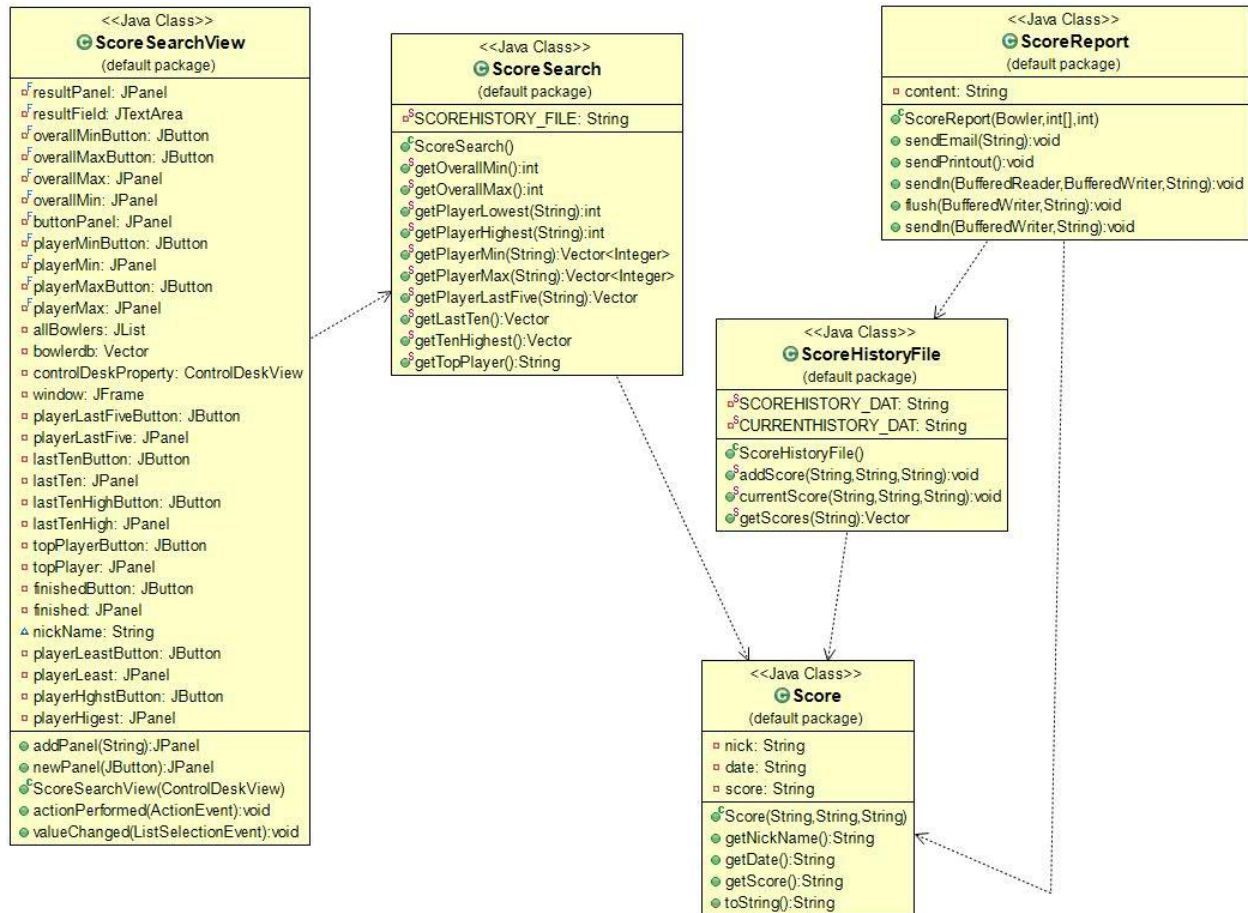
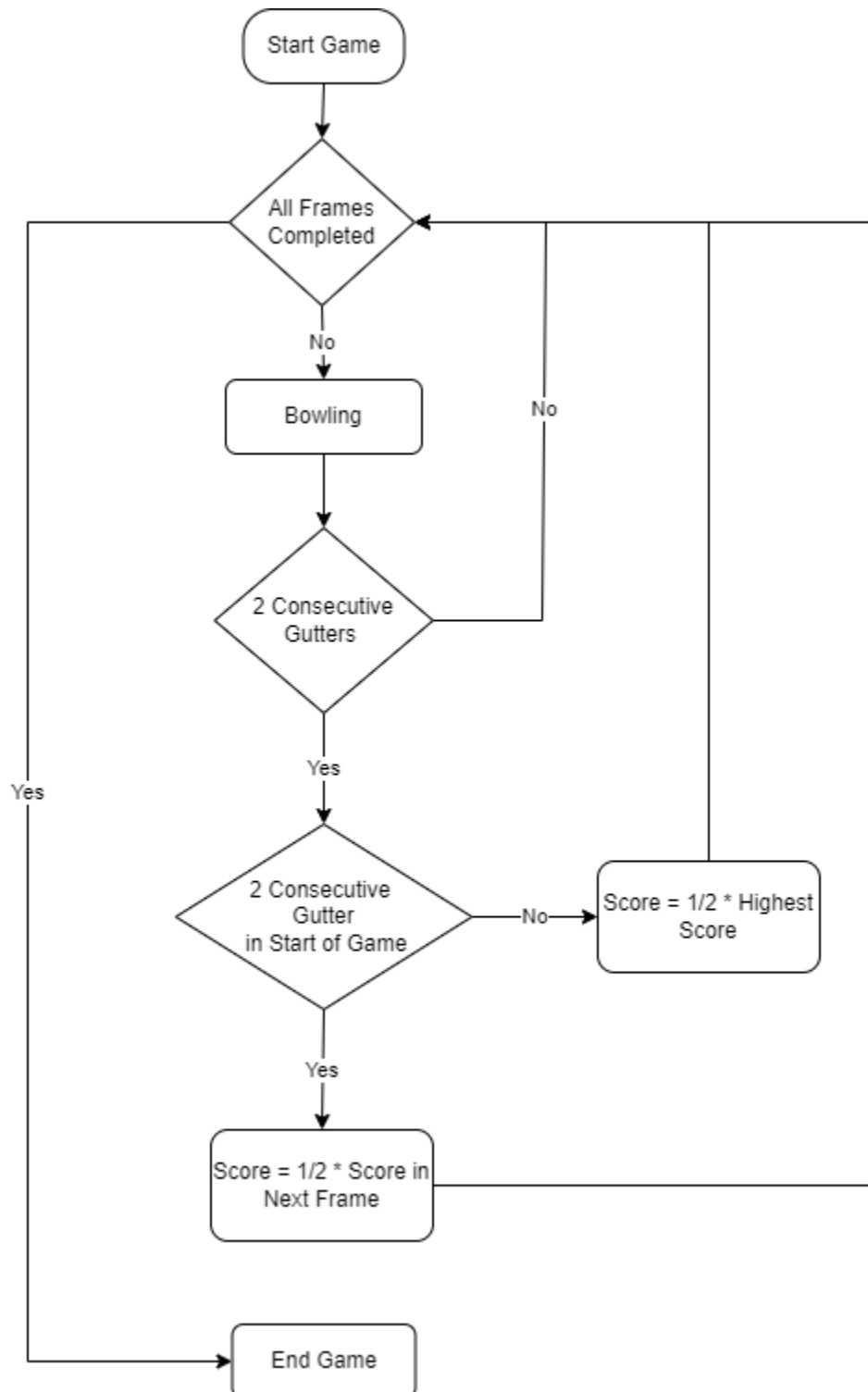


Figure: The interconnection between DataBase and Different Score classes

After a game finishes results can be viewed with PrintReport as well last 5 results can be accessed via the RndGamePrompt



Feature 4



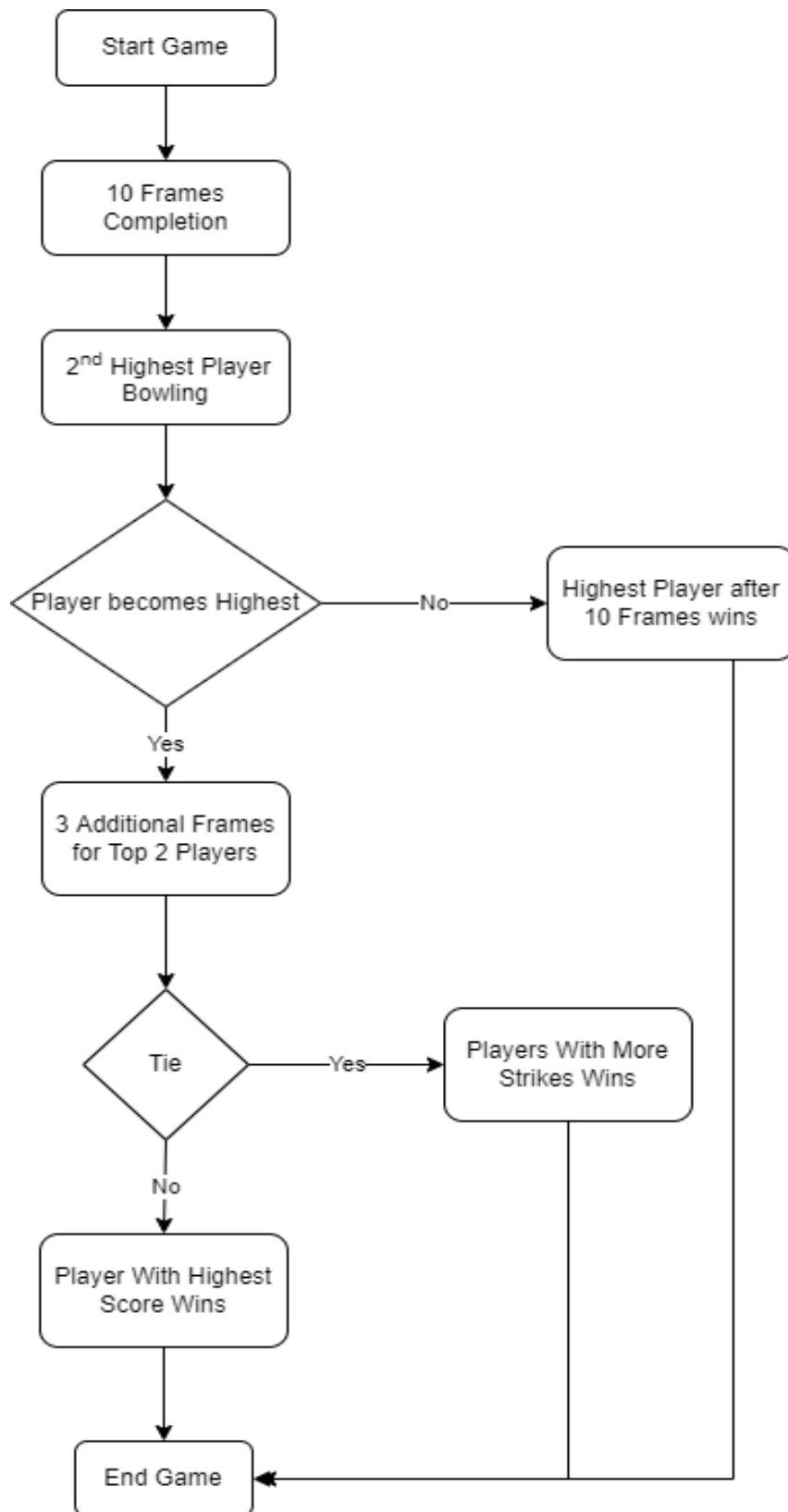


Figure: Block Diagram for Feature-4

As shown in the above diagram, when the game is started bowling will be done. If we don't get two consecutive gutters, then the flow will be just like the previous one. But, If we get two consecutive gutters then we will have two cases. First case, we might get two consecutive gutters in the starting of the game or not in the starting of the game.

Let's look at the first case, If we get two consecutive gutters in the starting of the game then we will have a penalty immediately. Penalty will be half of the highest score till that point i.e half of the highest score will be reduced from current score. Then the game flow will be continued normally with the next frame.

Let's look at the second case, If we get two consecutive gutters not in the starting of the game then we can't give a penalty immediately. We need to give a penalty after we get the results of the next frame. Once we get the results of the next frame, then we will add the results of the two scores of that frame and take half of that value and that value will be the penalty.

```

        this.highestScoresArray[this.bowlIndex] = this.highestScoresArray[this.bowlIndex] > var1.pinsDownOnThisThrow() ?
this.highestScoresArray[this.bowlIndex] : var1.pinsDownOnThisThrow();
        int var2 = var1.pinsDownOnThisThrow();
        byte var3 = 0;
        if (this.PenaltyInNextFrame) {
            if (!this.evenTurn) {
                this.evenTurn = true;
            } else if (this.evenTurn) {
                int var4 = var3 + var2;
                var2 -= var4 / 2;
                this.evenTurn = false;
                this.PenaltyInNextFrame = false;
            }
        } else if (var2 == 0) {
            if (this.evenTurn) {
                if (this.highestScoresArray[this.bowlIndex] != -1) {
                    var2 -= this.highestScoresArray[this.bowlIndex] / 2;
                } else {
                    this.PenaltyInNextFrame = true;
                }
            }
            this.evenTurn = false;
        } else {
            this.evenTurn = true;
        }
    } else {
        this.evenTurn = false;
    }
}

```

Figure: Code for penalty on getting two consecutive gutters

Implementation:

- The implementation of this feature is present in *Lane* class, which is responsible for creation of a new lane, assigning a party to that lane and also displaying scores for players of that party, while the *LaneView* and *LaneStatusView* are responsible for displaying the GUI for lane.
- As shown in the above figure, we have taken an array to store the maximum scores of each player till that time. So, once we get a higher score than the maximum score present in that array of a particular player, we will update it.
- If we get two consecutive gutters in the starting of the game, then the penalty will be taken from the scores of the next frame. *PenaltyInNextFrame* will handle this case with the help of the

evenTurn variable by adding up the two scores in the second turn and not in the first.

- If we get two consecutive gutters not in the starting of the game, then we will directly give a penalty using maxScore present till that time.

Feature 5

- After the end of 10 frames of bowling, the player with the second highest score is given an extra chance.
- If this player becomes the player with the highest score after an extra chance, then the current first and second highest players continue the game with 3 additional frames. Player with the highest score wins the game.
- If there is a tie after the 3 additional frames the player with more strikes wins the game.

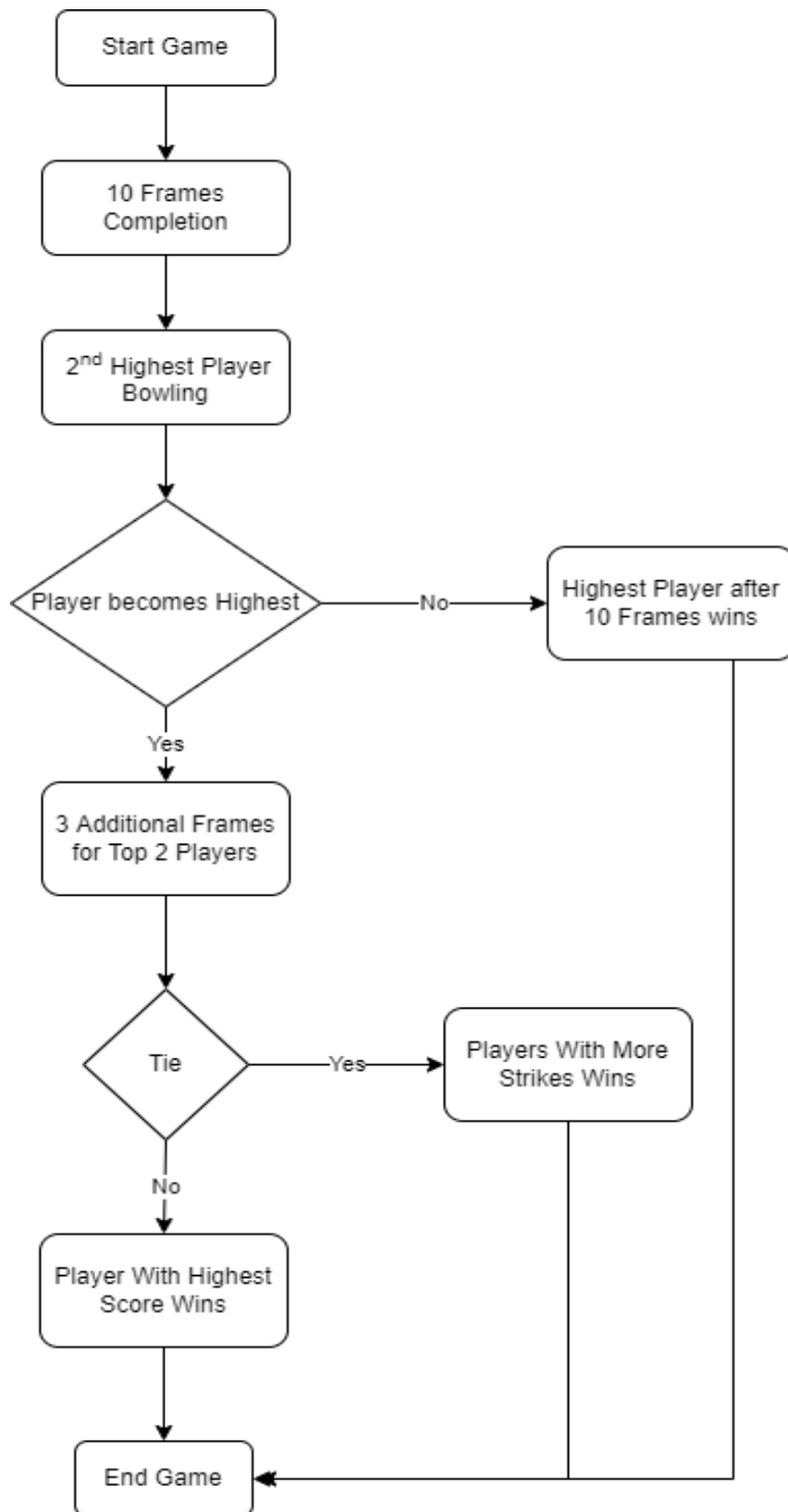


Figure: Block Diagram for Feature-5

Implementation:

- The implementation of this feature is present in *Lane* class, which is responsible for creation of a new lane, assigning a party to that lane and also displaying scores for players of that party, while the *LaneView* and *LaneStatusView* are responsible for displaying the GUI for lane.
- The second highest score is computed at the end of 10 frames.

```
public int getSecondHighestPlayer(int[] finalScores){
    int firstind = -1, fmax = 0;
    int arrSize = finalScores.length;
    int idx = 0;
    while( idx < arrSize ){
        if(finalScores[idx]>fmax)
        {
            fmax = finalScores[idx];
            firstind = idx;
        }
        idx++;
    }
    idx = 0;
    int secondind = -1, smax = 0;
    while( idx < arrSize ){
        if(idx!= firstind && finalScores[idx]>smax)
        {
            smax = finalScores[idx];
            secondind = idx;
        }
        idx++;
    }
    return secondind;
}
```

Figure: Code for computing the 2nd highest score after 10 frames

- For providing an extra chance to the second highest player, initially one extra frame is created through *LaneView* class.
- Inside the *LaneView* class in *receiveLaneEvent* member function, when control reaches the 11th frame, member function *getSecondHighestPlayer* is called by passing the cumulative scores of all players in that game, present in *lescores*.
- The addition of 3 extra frames is done if after the extra chance, this player becomes highest else the game ends with the player with the highest score as winner.

Second Highest Player was unable to cross the Highest player score

Mike

5 /	5 2	8 1	4 2	X	1 5	X	4 4	7 1	1 4	
15	22	31	37	53	59	77	85	93	97	

Bowl!

Jim

7 1	2 4	2 4	X	1 5	5 3	9 /	5 /	4 5	7 2	
7	13	19	35	41	49	64	78	87	95	

Bowl!

Tom

X	6 2	4 3	X	X	3 4	7 /	X	X	9 / 8	
18	26	33	56	73	80	100	129	149	166	

Bowl!

Lana

X	7 /	0 7	6 0	6 3	X	6 0	3 6	7 2	X 8 2	4
20	30	37	43	52	68	74	83	92	111	



Bowl!

Maintenance Call

Figure: In this game the 2nd highest player(here Lana) is given an extra chance. Even after the extra chance Tom continues to be the highest Player hence he wins the game.

Feature 6

After each throw, an emoticon is shown based on the number of pins knocked down. A particular emoji would be shown for a range of values as shown below:

Score Range	Emoticon
0 to 3	
4 to 5	

6 to 8



9 to 10

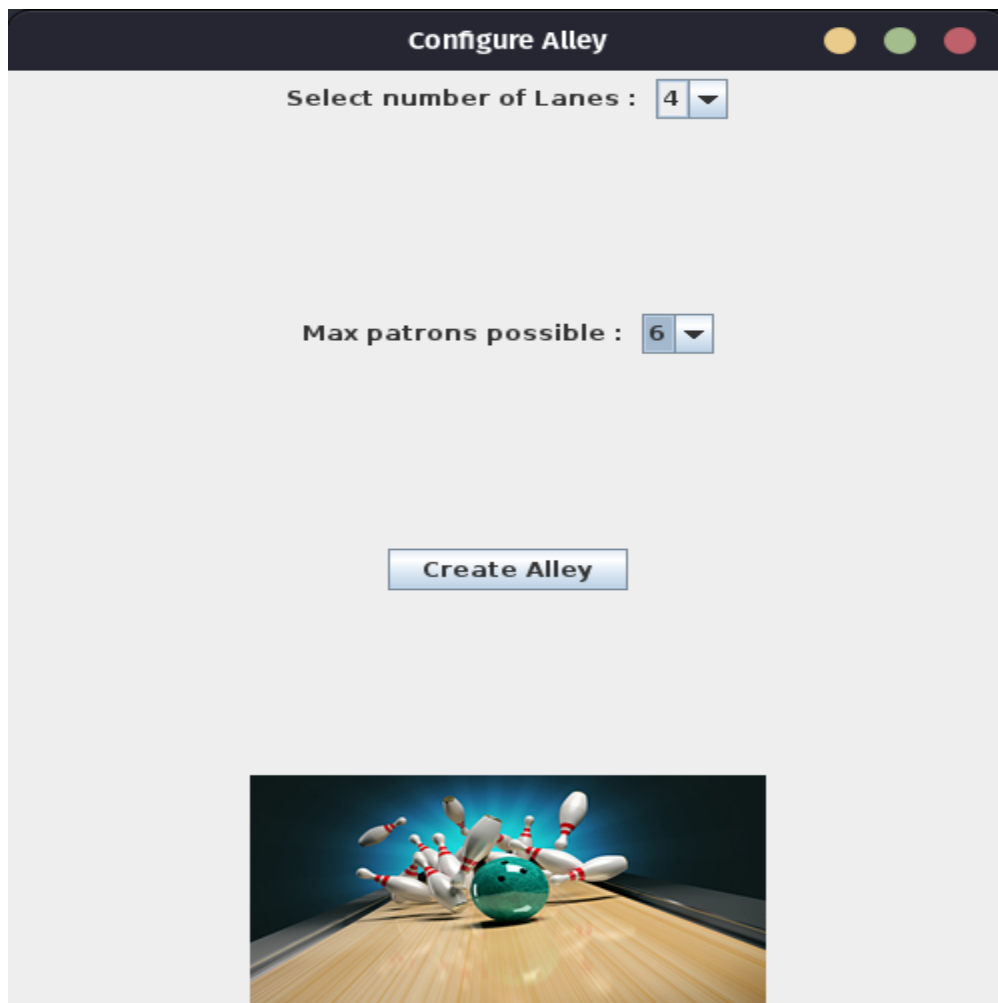


Feature 7

Making all existing numbers configurable

In order to make all the numbers configurable a new view is implemented which is shown at the beginning which has two drop down menus which allows us to configure numLanes and maxPatronsParty. According to the values chosen here Lanes and ControlDesk are configured.

Progressive Disclosure patterns have been used here to show only the options related to configuring alley which helps user to change stuff easily. **Clear Primary Actions** pattern is also used here for dropdown so that various options available to the user stand out.



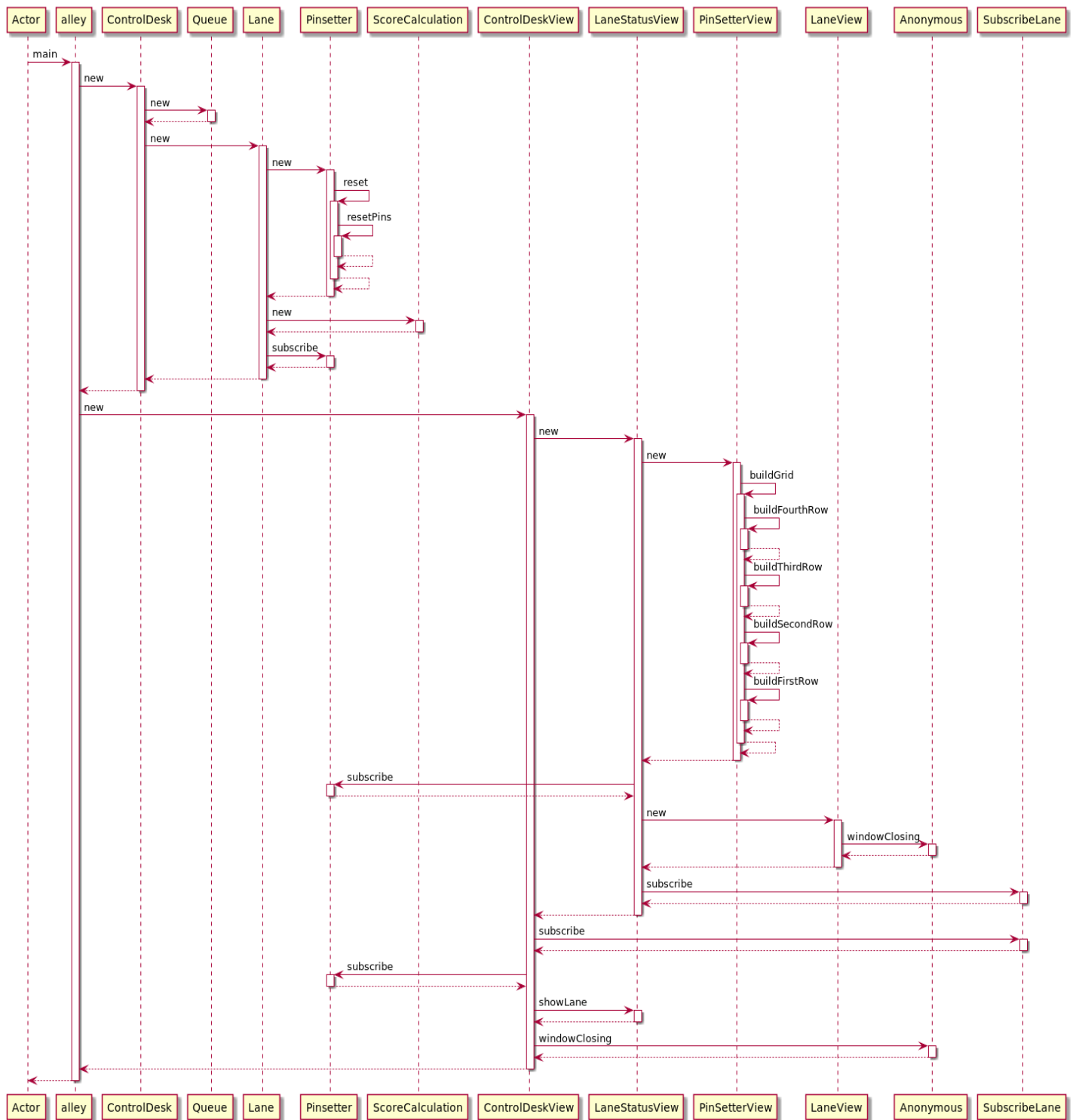


Figure: Sequence diagram showing the different pathways possible from alley

Architectural Patterns

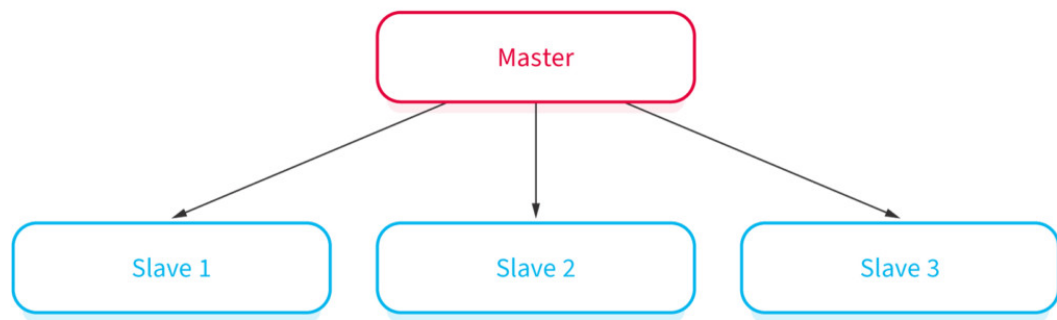
1. **Layered Patterns** :- It's a pattern in which the code is organized in layers and is one of the most widely used. The following are some of the pattern's key characteristics:

The data enters the system through the outermost layer. The data progresses through the layers until it reaches the database layer, which is the innermost.

This pattern's simplest implementations have at least three layers: a presentation layer, an application layer, and a data layer. Users interact with the presentation layer via a graphical user interface (GUI), while the application layer handles business logic. A database is used by the data layer to store and retrieve data.

LaneView is acting like a presentation layer, similarly Lane and Laneevent are handling application and data layer.

2. **Master-slave pattern**:-



Master-slave pattern

When clients make multiple requests, the "master-slave architecture pattern" comes in handy. The requests must be handled in a timely manner. The following are some of its most important characteristics:

When the master receives multiple requests at the same time, it launches slaves.

The slaves work in parallel, and the operation is only complete when all slaves have finished processing their requests.

Here each time we start a new game, parallelly lanes will be created after sending requests.

3. **Event-driven pattern:-** Messages can be ingested into an event-driven ecosystem and then broadcast out to whichever services are interested in receiving them using an event-driven system. Here when we click on AddtoParty, that event will drive us to get the available bowler names and gives us options to select the bowlers we need. In this way, for all the other events an event-driven pattern is followed.

Design Patterns

Observer Pattern: Observer is a behavioral design pattern that allows us to specify a subscription mechanism for numerous objects to be notified of any events that occur on the object they're watching. It establishes a one-to-many relationship between objects, ensuring that when one object changes state, all of its dependents are automatically notified and updated. The Observer design pattern is utilized here; whenever the "Bowl" button is clicked, the LaneEventObserver is alerted, and the score computation is performed; the results are then displayed using the LaneView.

Singleton Pattern: The singleton pattern is one of Java's most basic design patterns. This design pattern is classified as a creational pattern since it gives one of the most effective ways to construct an object. This pattern uses a single class that is in charge of creating an object while ensuring that only one object is created. Therefore this pattern has been

employed in the alley class which contains the main function as it is used to construct an object while ensuring that only one is created in the lifetime of this game.

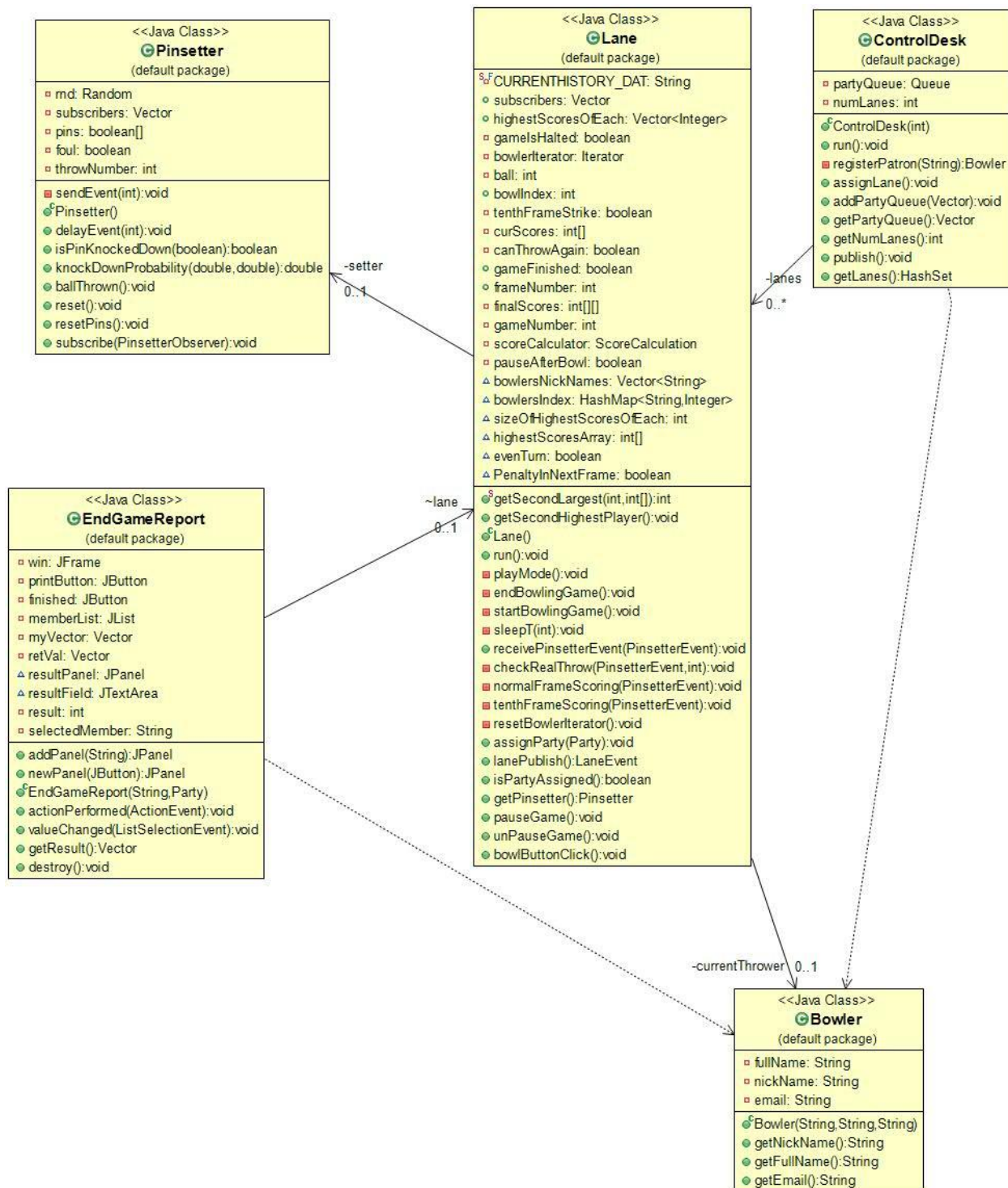
Adapter Pattern: The adapter design pattern is a structural design pattern that allows items with incompatible interfaces to work together. An adapter is a special object that changes one object's interface so that it may be understood by another. Here in this project the ControlDesk object acts as an adapter and allows Party, Bowler and Queue to work together in this system.

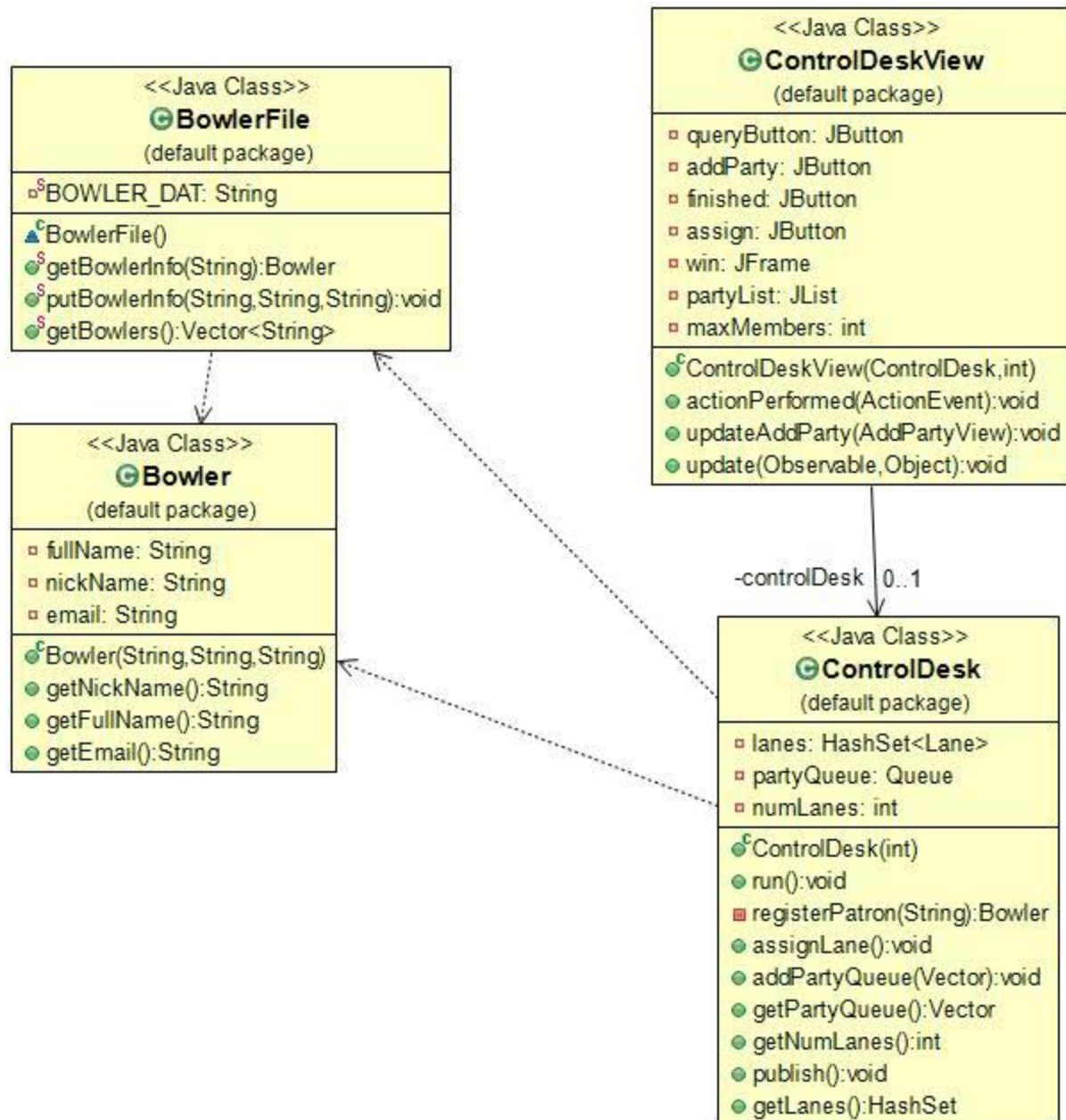
Strategy pattern: A class's behavior or algorithm can be altered at runtime using the Strategy pattern. This design pattern is classified as a behavior pattern. We generate objects that represent numerous strategies and a context object whose behavior varies depending on its strategy object in the Strategy pattern. The context object's running algorithm is changed by the strategy object. In this system strategy pattern has been employed for penalties in the game, as its behavior varies depending on its strategy object.

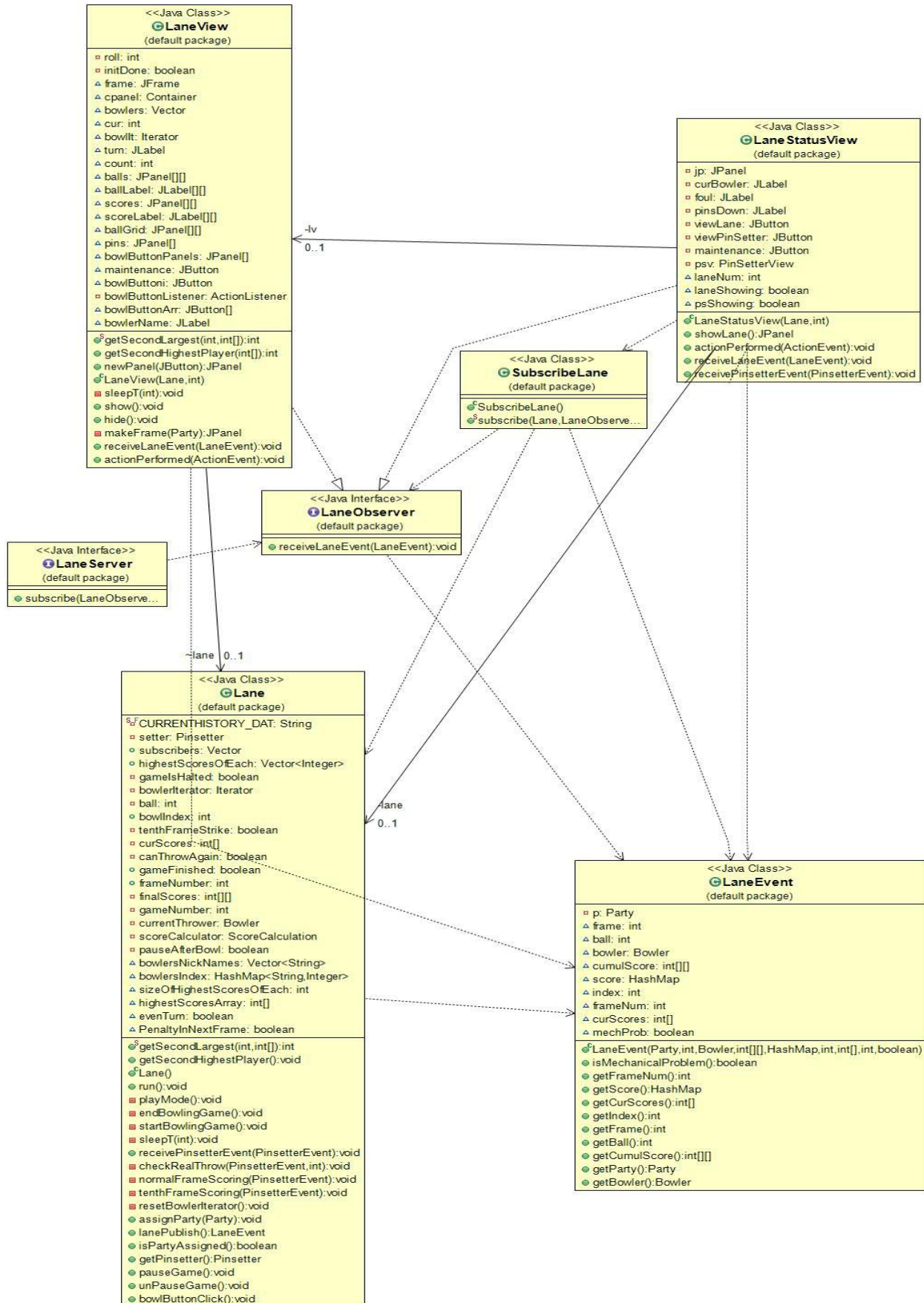
Mediator Pattern: Mediator is a behavioral design technique that allows you to decrease the chaos of object relationships. The pattern prevents the items from communicating directly with one other, forcing them to collaborate only through a mediator object. This pattern has been used inside of ScoreSearch which allows the ScoreSearchView and ScoreSearchHistory to communicate with the database layer.

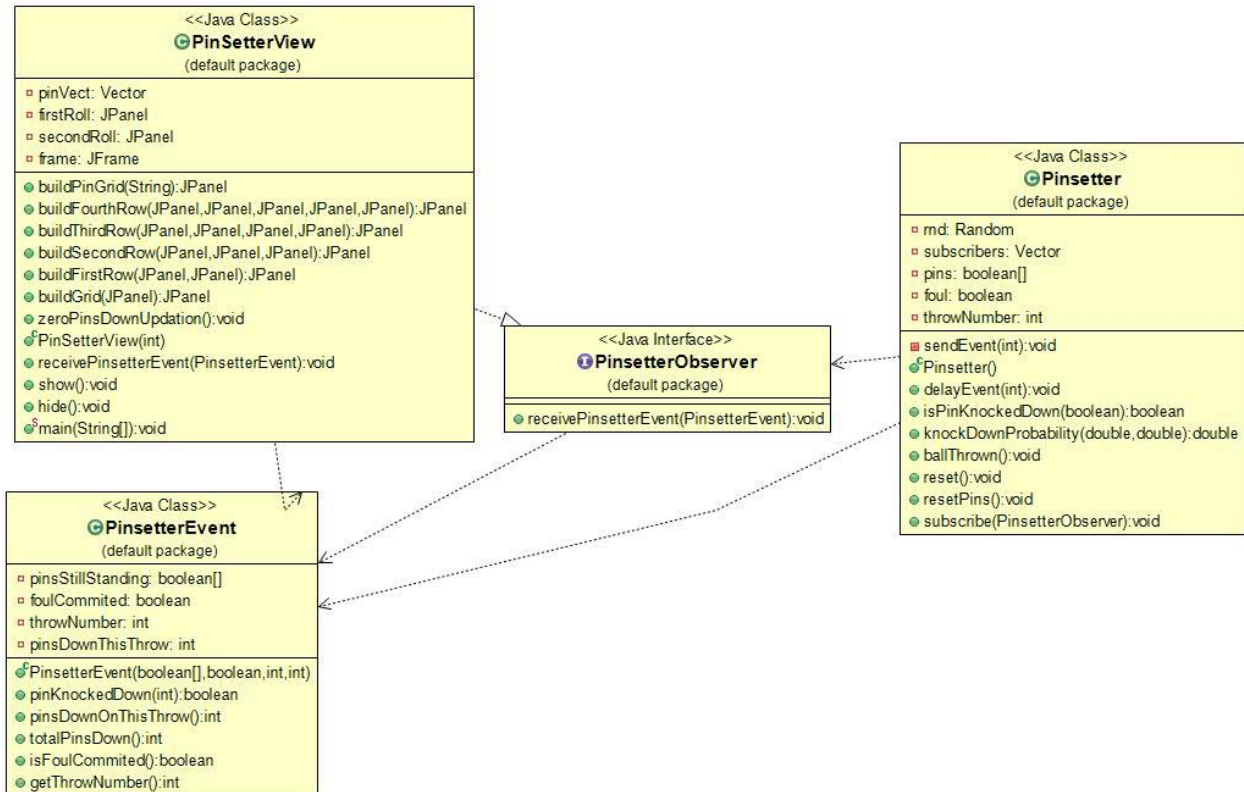
Class Diagrams

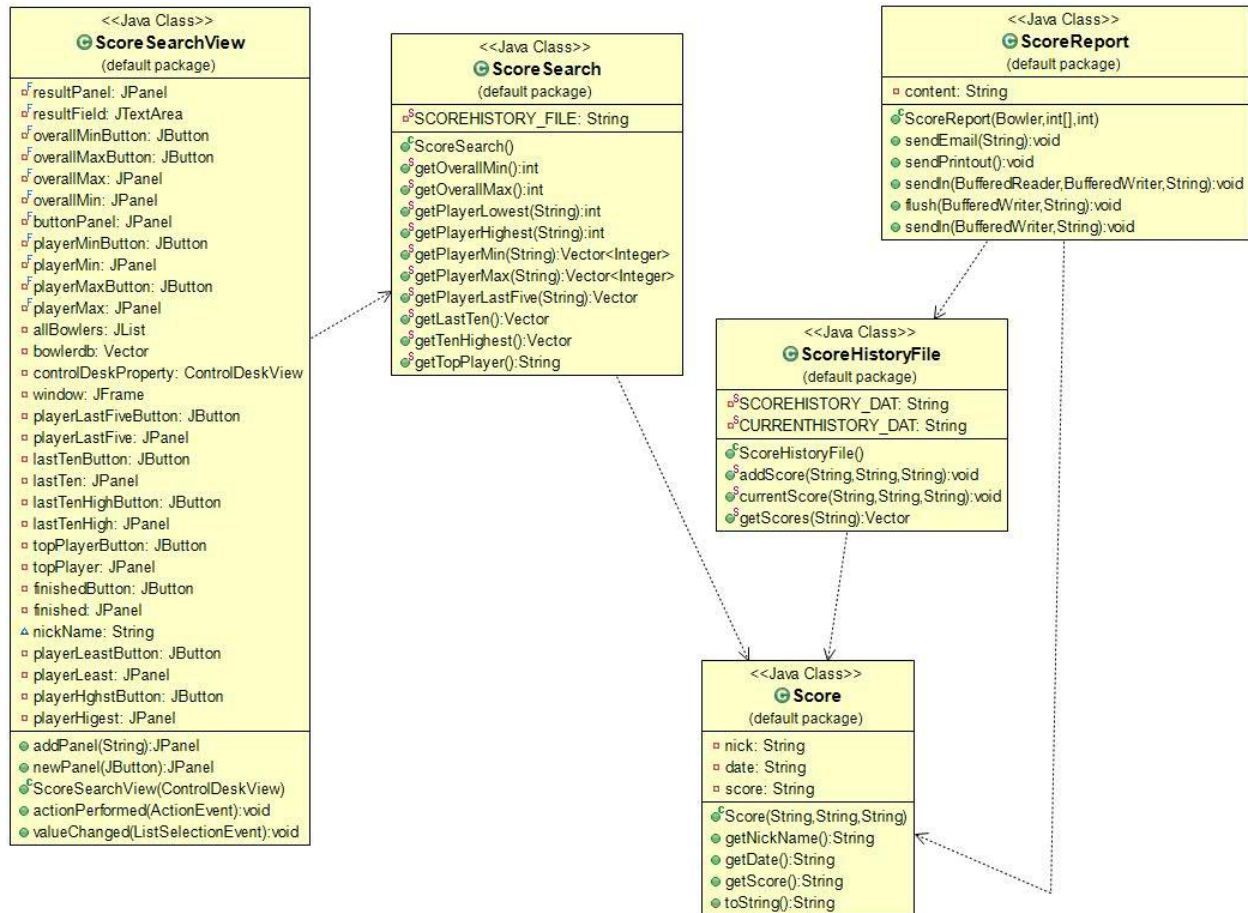
Overall class diagrams for before and after refactoring and all other class diagrams can be found [here](#)





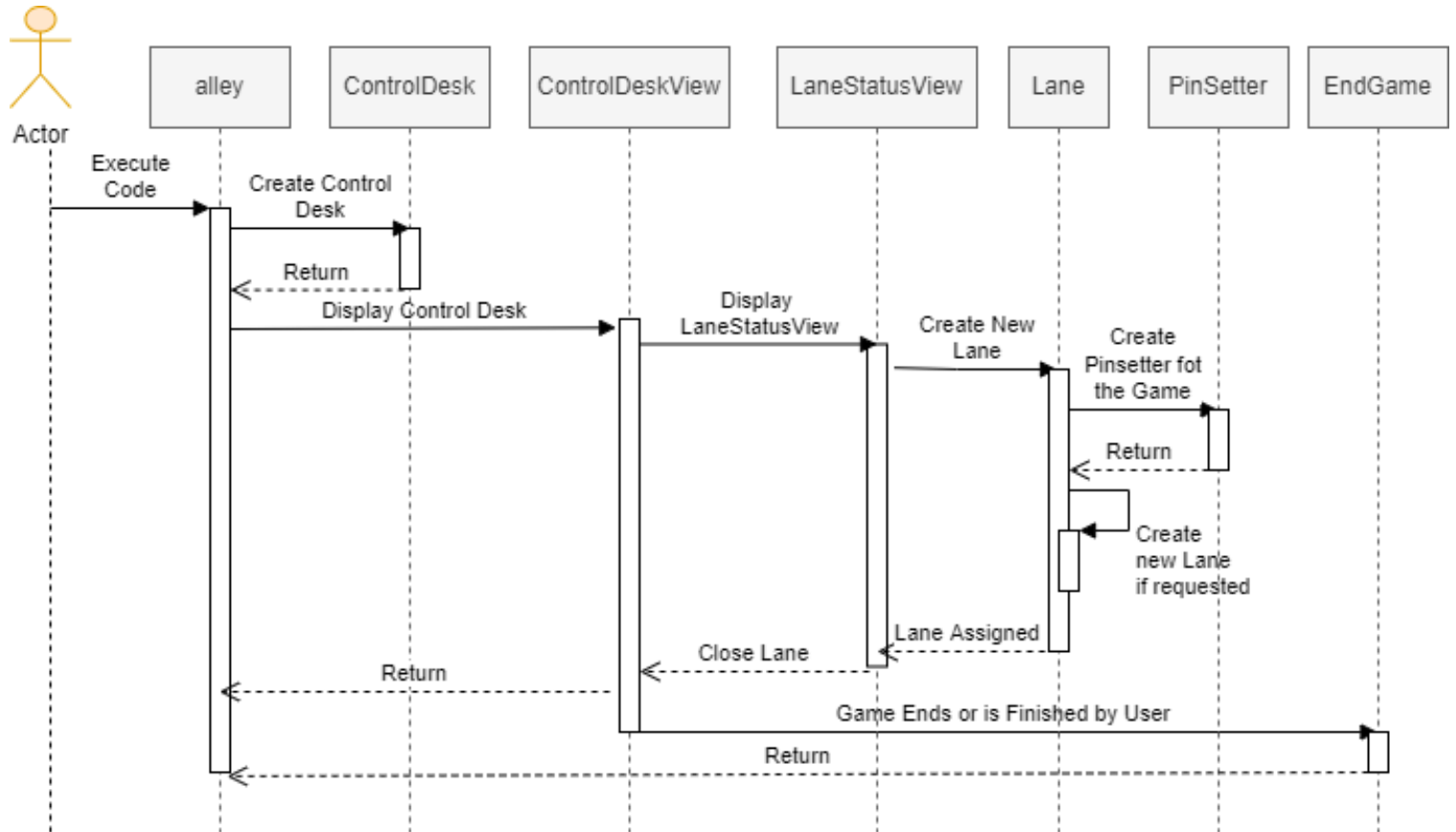


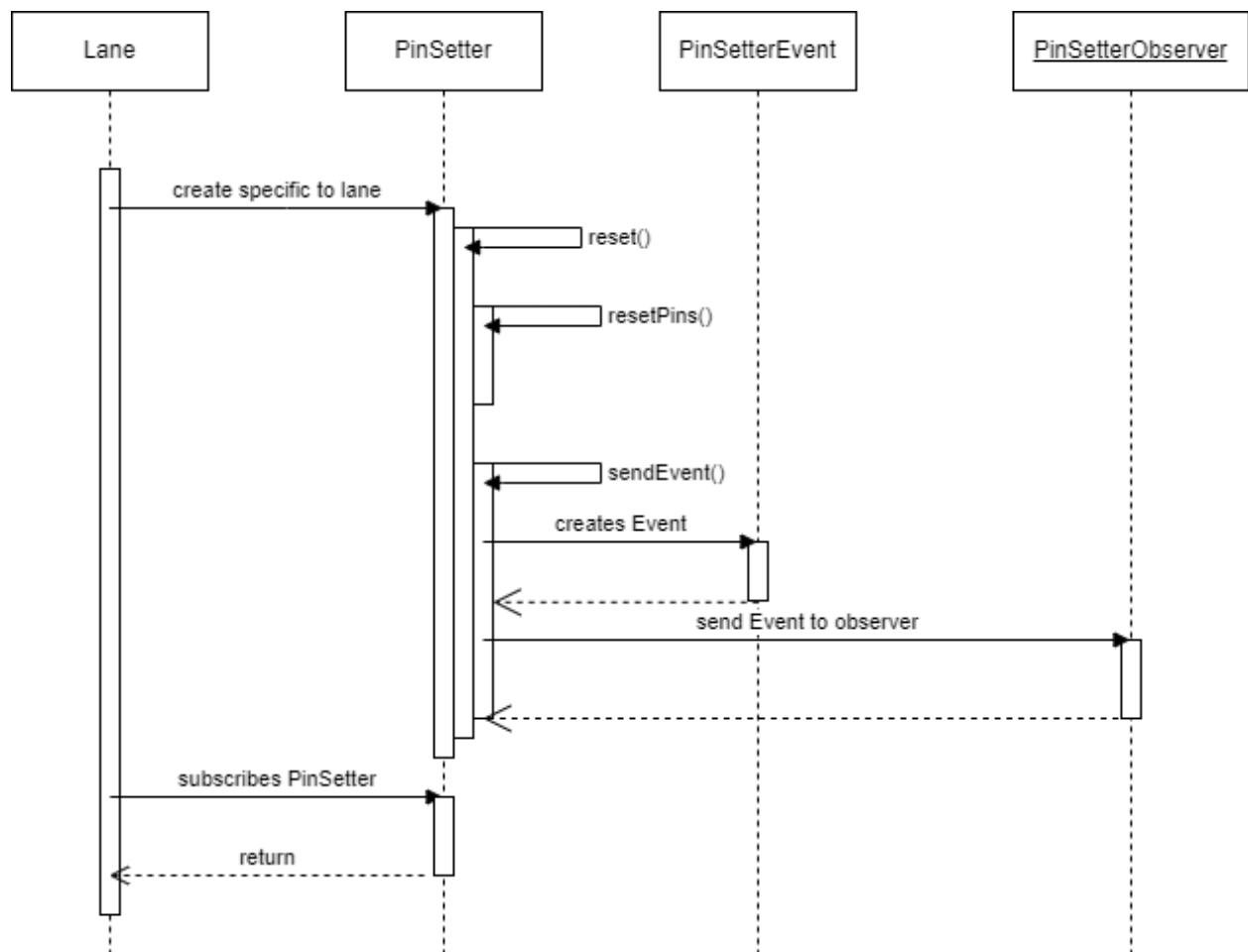


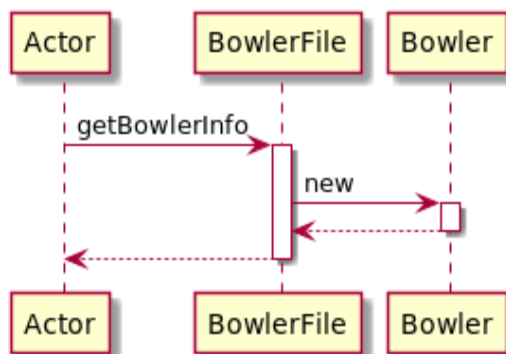
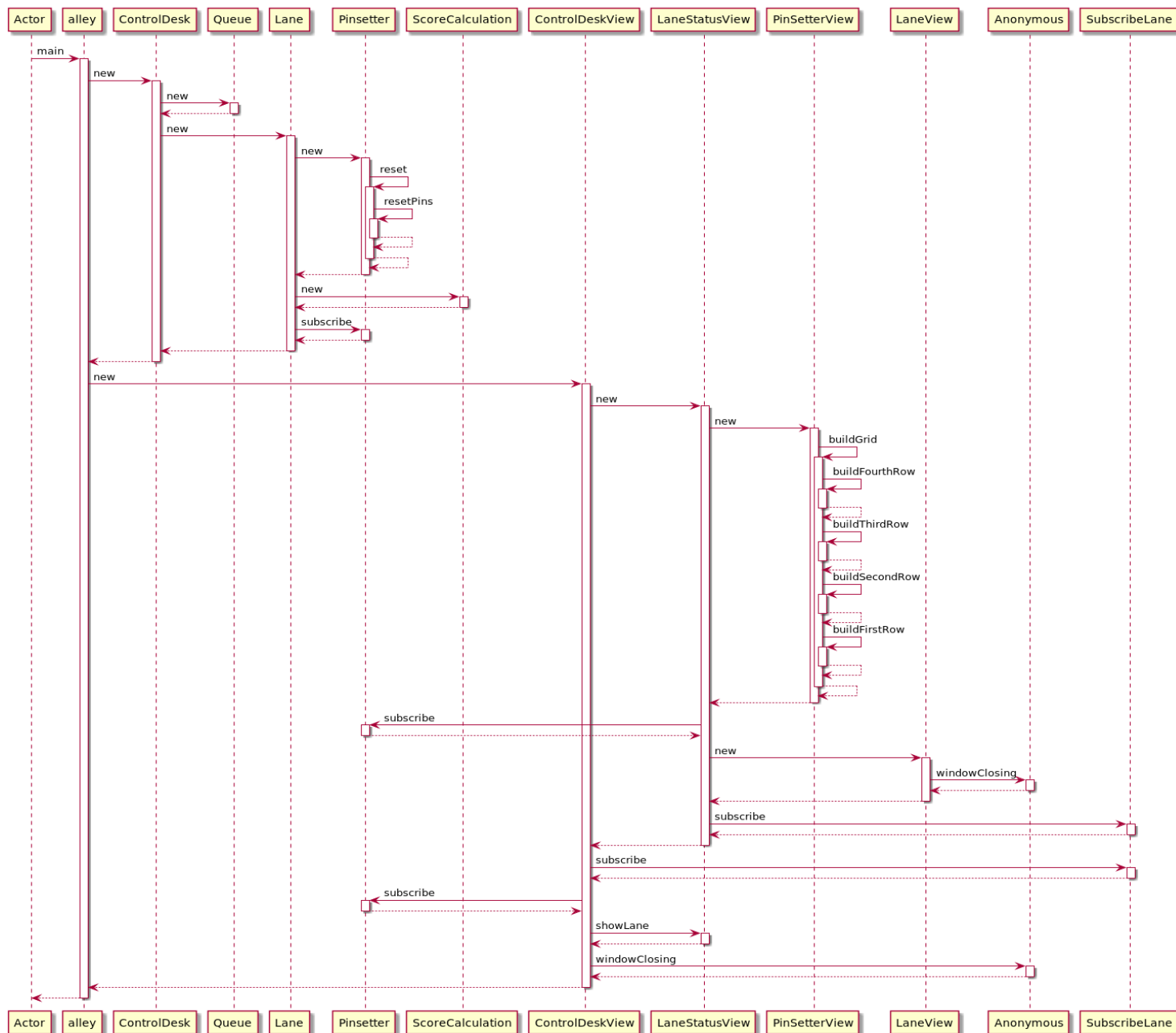


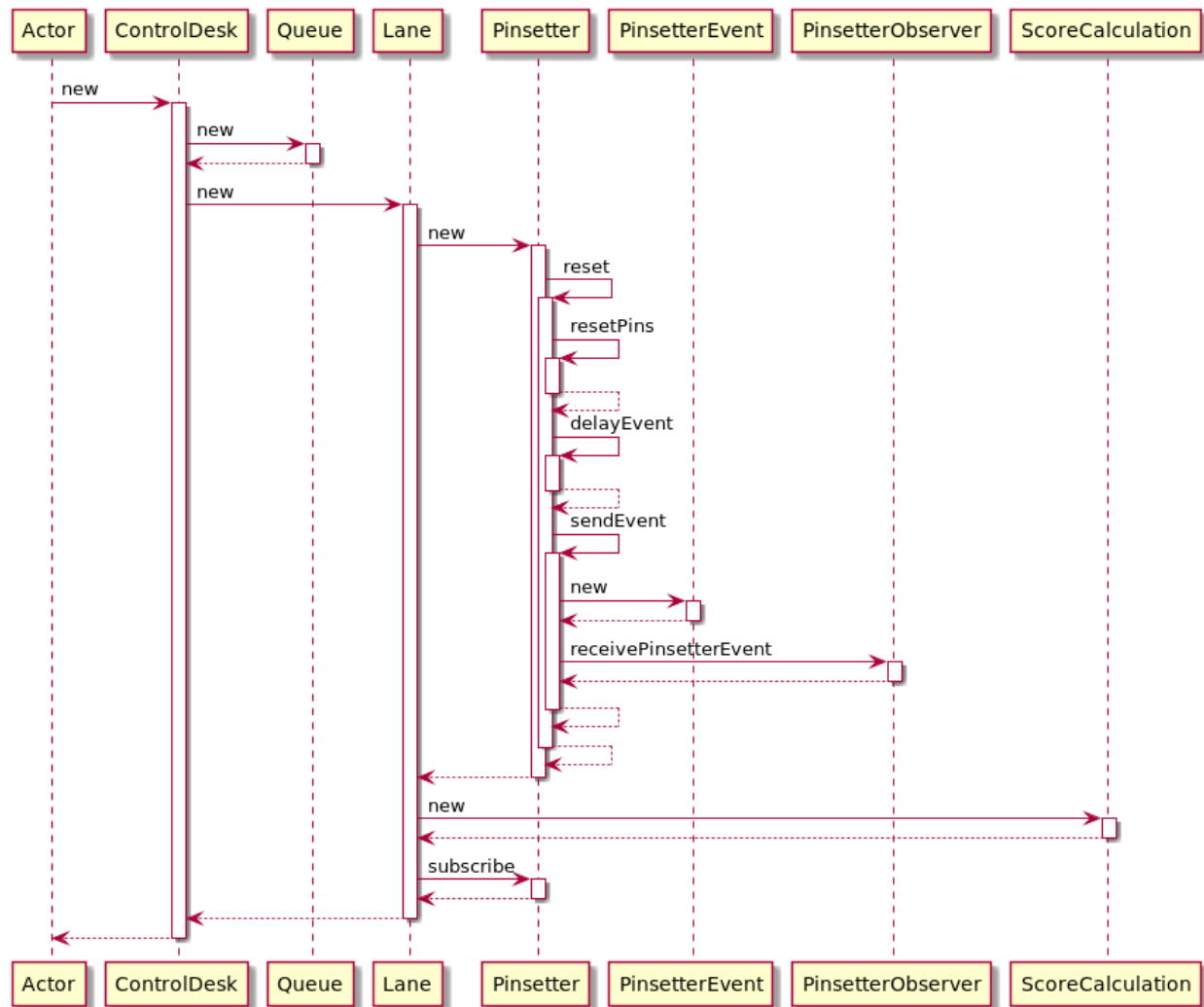
Sequence Diagrams

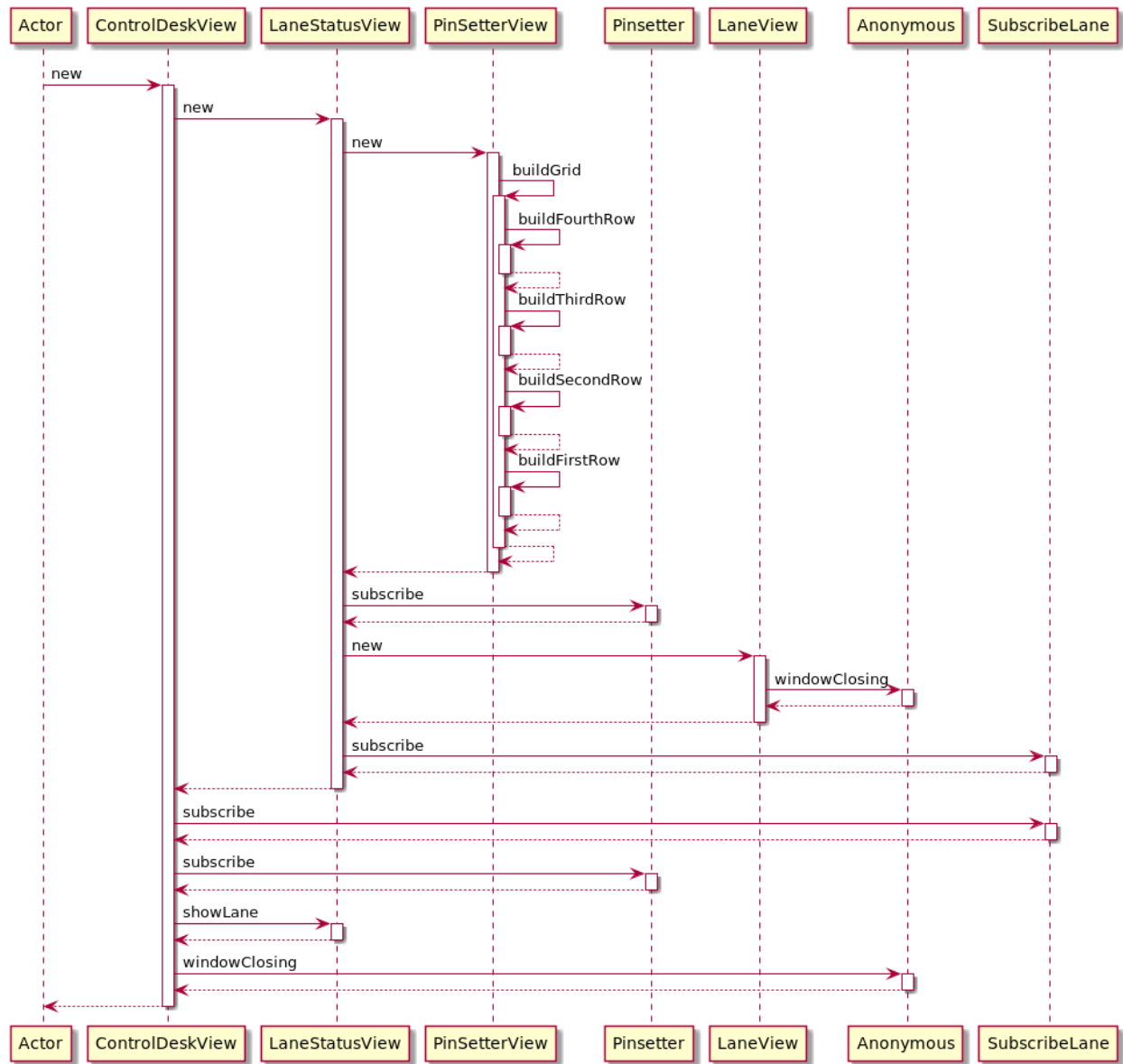
All sequence diagrams can be found [here](#)

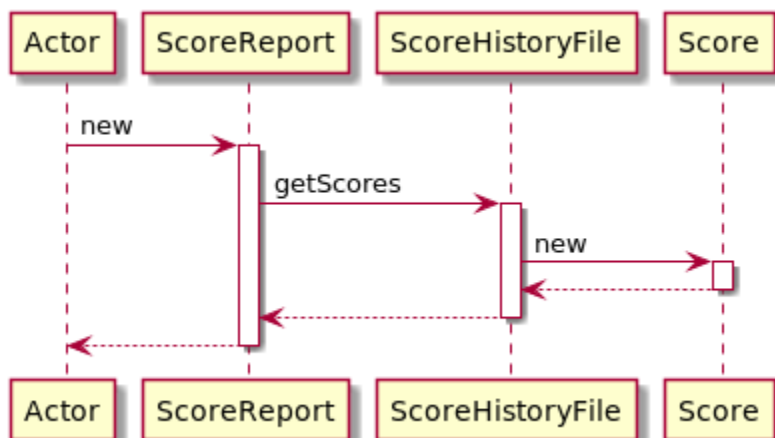
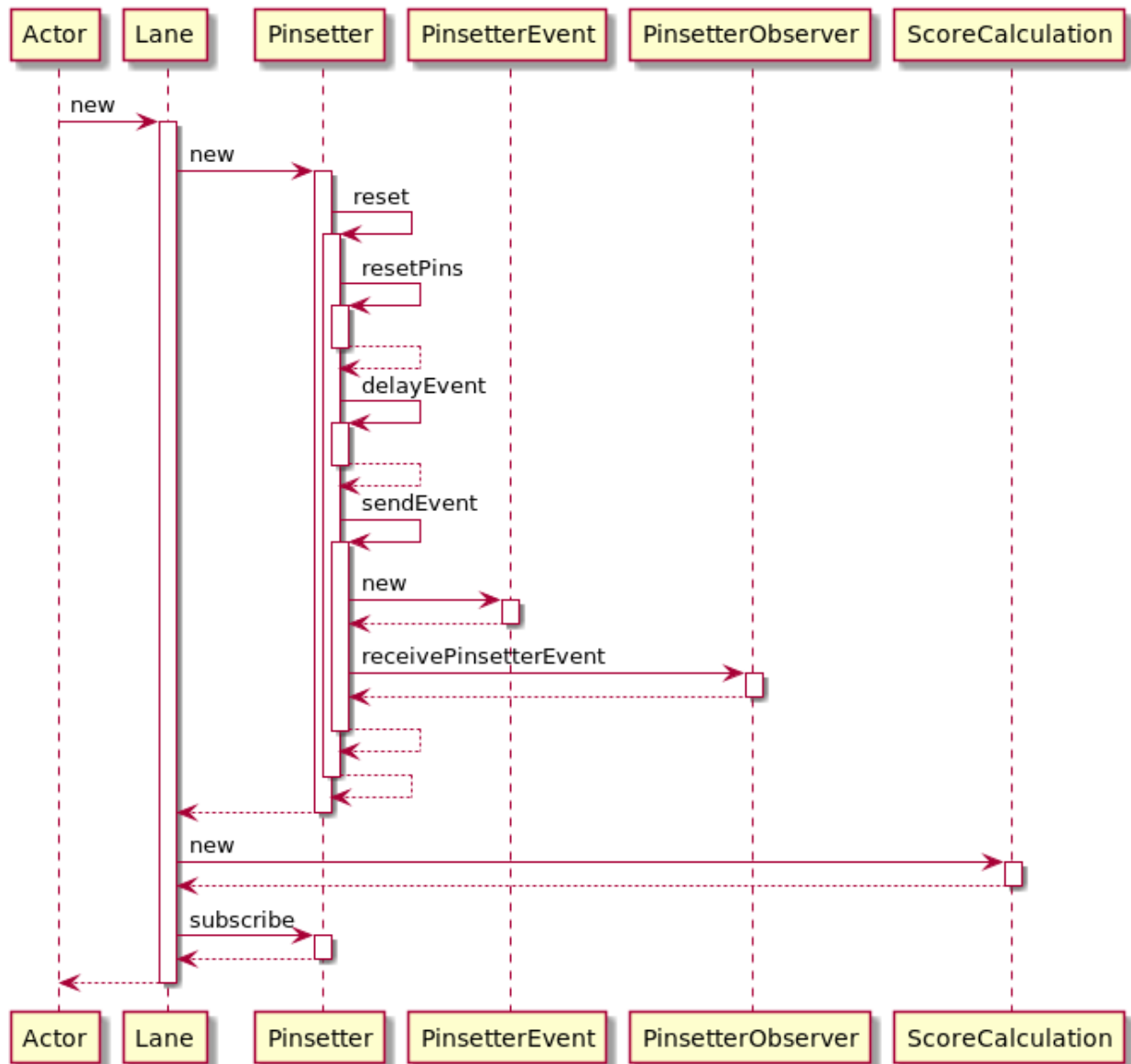












Class Responsibility Table

Class Name	Responsibility
Bowler	Contains information about the bowler (nickname, fullname, email)
BowlerFile	It can be used to interact with the bowler database.
ControlDesk	It assigns Lane to the party, adds a party to the waiting queue if no lane is free.
EndGamePrompt	Displays the prompt that pops up at the end of the game.
EndGameReport	Displays the game report at the end.
Lane	It creates a new lane and starts it's thread, assigns party to a lane and marks score for the bowlers on the board.
Party	It maintains the bowlers list who belong to the party
Pinsetter	It handles resetting, simulation and detection of pins appropriately during each throw.
Queue	It creates a new party waiting queue.
ScoreReport	It creates a final score report for all the people in a party and sends the report as an email to each person.
emoticon	We will have four emoticons, and they will pop up depending on the

	score scored in each frame.
ScoreSearch	It will have logic of accessing database and give results to ScoreSearchView
ScoreSearchView	It will have ten buttons like highest score, lowest score, Top player, etc.

Original Design

Weakness

The original project consists of code and design that conveys poor software design and practices, as well as bugs that affect the application's performance. We used code smells and anti-patterns to describe the system flaws.

Code Smells

Type	Class name	Issue
Large Class	Lane.java	Methods that could potentially be classified as a separate class
Long Methods	AddPartyView.java, Lane.java, PinSetterView.java,	Massive constructors and/or functions that could be broken down into separate functions

	LaneStatusView.java	
Dead/Duplicate code	AddPartyView.java, Lane.java, LaneServer.java, ScoreReport.java	There are a lot of repeating statements that create buttons and other UI elements.
God object	ControlDesk ControlDeskView LaneStatusView ScoreReport	Observers usually have access to a diverse set of classes. For example, LaneStatusView has access to LaneView, Lane, and PinSetterView.
Lazy/Freeloader Class	ControlDeskObserver.java, ControlDeskInterface.java, LaneServer.java, LaneObserver.java, LaneEventInterface.java, PinSetterEvent.java, Queue.java	Custom observer interfaces and classes were created for each type of observer, which could have been avoided by using a single observer interface or one provided by Java.
Feature Envy	Lane.java	Excessively employs methods from various classes

Anti-Patterns

Type	Class name	Issue
Lava Flow	Lane.java	There's a lot of TODO code. Unnecessary stipulation branches. Redundant code.
	LaneServer.java	Didn't use this class

Repeating Yourself	AddPartyView.java	Statements that are repeated to create buttons and panes
	ScoreReport.java	String code duplication
	PinsetterView.java	Repetitive statements that generate buttons and panes
Functional Decomposition	ControlDeskEvent.java	There is only one method and variable.
Data Class	PinSetterEvent.java LaneEvent.java	Only properties and getter-setter methods are present in this class. Doesn't perform any significant operations on the data
Not Invented Here syndrome	Queue.java	Instead of using the one provided by Java, the project employs a custom queue class.
	All classes that use observable pattern	Instead of using the built-in observer and observable interfaces, I created custom observer interfaces and event classes for each type of observer.

Strengths

1. One of this project's major strengths is that it employs the MVC pattern. View classes communicate with their corresponding model classes and vice versa.
2. It employs the Observable pattern to implement the aforementioned pattern.
3. The majority of the classes' coupling has been kept to a minimum.

Fidelity to the Design Document

The provided project met all of the requirements in the design document and did not deviate from it by adding any unnecessary functionalities.

Narrative of the refactored design

As previously stated, our key refactoring goals were to reduce coupling, promote cohesiveness, and eliminate code smells. While refactoring our code, we used a variety of ways to ensure that all of our Key metrics were met. For example, to reduce complexity and the number of blobs in the original code, we added additional classes: The control desk class in the Bowling Alley game is the center of the game. The control desk assigns a new Lane to a new party as soon as it is established, allowing them to add members to their game. This class also has functions for displaying names on the control panel. We implemented Observer pattern , singleton pattern, adapter pattern and following changes were made to de code,

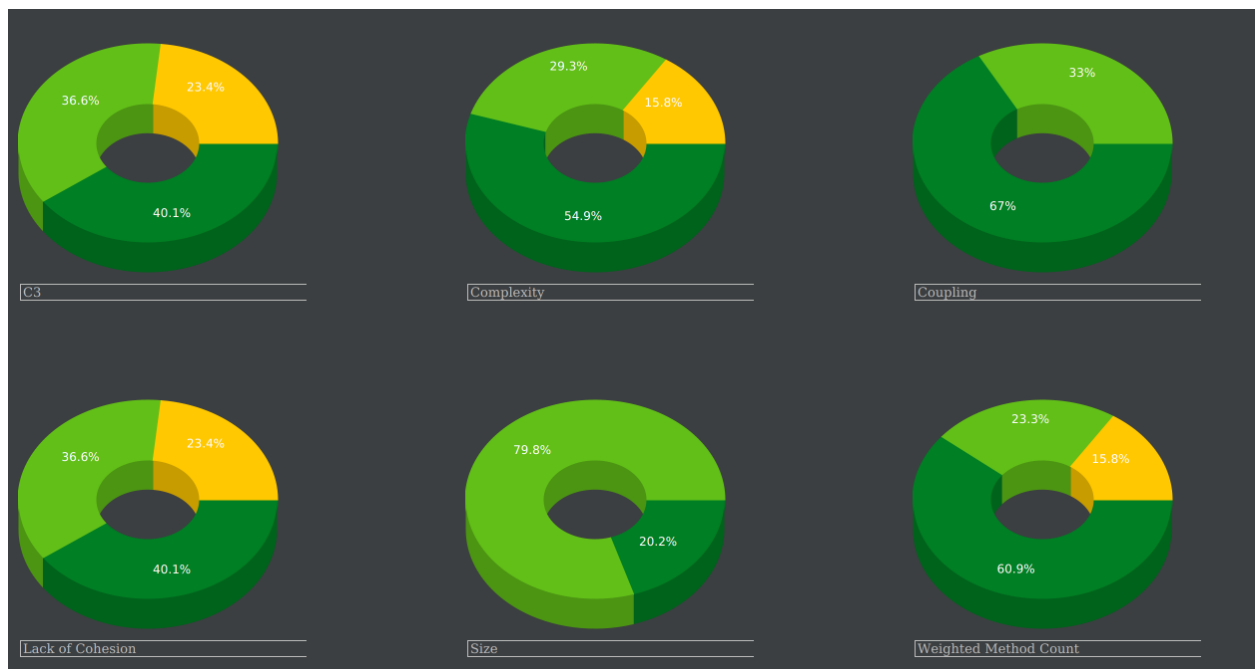
Changes made in the code

- In AddPartyView.java, redundant code is eliminated by creating two new methods named addPanel and newPanel.
- Alley.java is removed and drive class is modified and renamed as alley.java
- In Bowler.java, two methods named getNickName and getNick have the same definitions. So one method is removed and modified calls to that method accordingly.
- In ControlDesk.java observer pattern was implemented.
- In EndGamePrompt.java, LaneView.java and NewPatronView.java, redundant code is eliminated by creating a method named newPanel.
- In EndGameReport.java, redundant code is eliminated by creating two new methods named addPanel and newPanel.
- In ScoreReport.java, a new method named flush is created and removes redundant code in sendIn methods.

- In Lane.java code complexity is reduced by creating two new classes ScoreCalculation.java and SubscribeLane.java
- ScoreCalculation.java handles all the calculation of scores
- PinsetterObserver.java observer pattern was implemented
- Emoticon.java handles the emoticons which pop up.
- ScoreSearch.java accesses database and give results to ScoreSearchView
- ScoreSearchView handles ten buttons like highest score, lowest score, Top player, etc.

Metrics

Before Refactoring

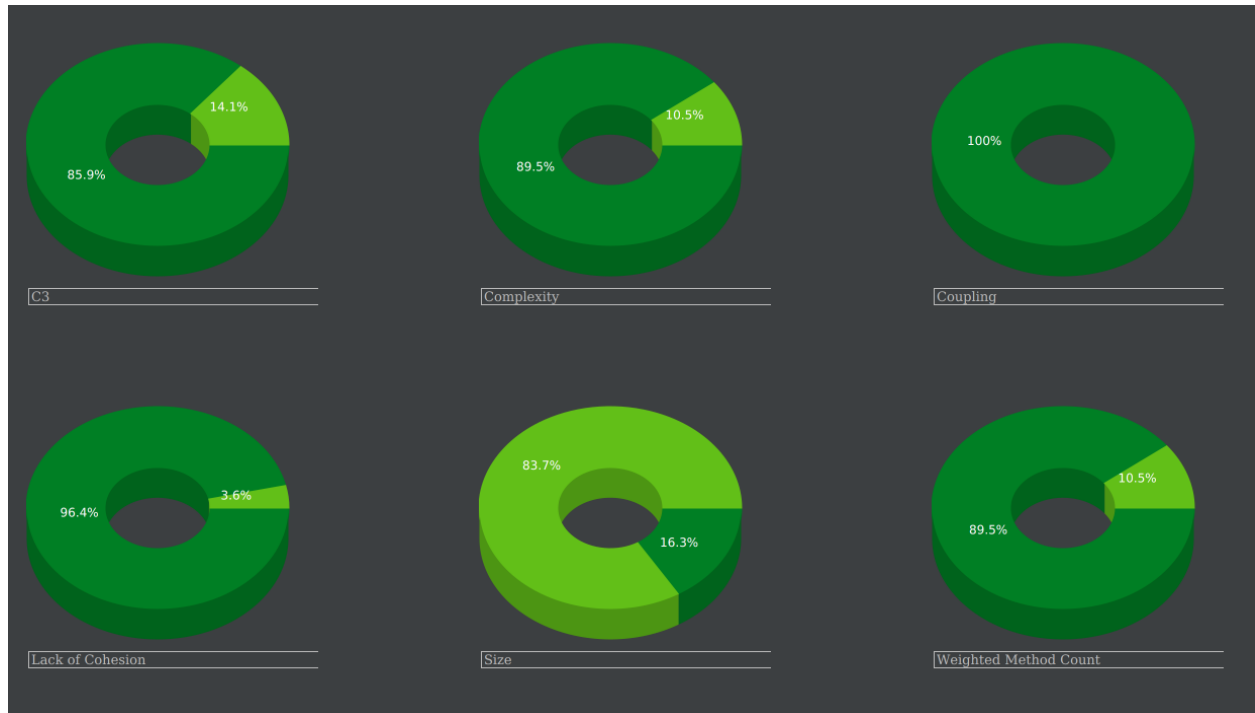


Name	Complexity	Coupling	Size	Lack of Cohe...	RFC	SRFC	DIT	WMC	LOC	CMLOC	NOM	NOF
code												
> AddPartyView	low-medium	low	low-medium	low-medium	63	36	1	21	127	118	6	14
> Alley	low	low	low	low	2	0	1	2	6	4	2	1
> Bowler	low	low	low	low	7	4	1	9	25	21	6	3
> BowlerFile	low	low	low	low	12	9	1	6	38	36	0	0
> ControlDesk	low-medium	low-medium	low-medium	medium-high	40	26	2	22	68	63	11	4
> ControlDeskEvent	low	low	low	low	2	0	1	2	6	4	2	1
> ControlDeskObserver	low	low	low	low	1	0	1	1	2	1	1	0
> ControlDeskView	low-medium	low-medium	low-medium	low-medium	67	38	1	8	87	81	4	7
> EndGamePrompt	low	low	low-medium	low	22	18	1	8	55	50	4	6
> EndGameReport	low	low	low-medium	low-medium	36	30	1	12	79	71	5	8
> Lane	medium-high	low-medium	low-medium	medium-high	76	42	2	87	227	208	17	18
> LaneEvent	low	low	low	medium-high	11	0	1	11	41	30	11	10
> LaneEventInterface	low	low	low	low	9	0	1	9	10	9	9	0
> LaneObserver	low	low	low	low	1	0	1	1	2	1	1	0
> LaneServer	low	low	low	low	1	0	1	1	2	1	1	0
> LaneStatusView	low	low-medium	low-medium	low-medium	38	22	1	17	93	82	5	13
> LaneView	low-medium	low	low-medium	low-medium	47	36	1	31	140	124	6	15
> NewPatronView	low	low	low-medium	low	39	20	1	8	85	75	6	17
> Party	low	low	low	low	2	0	1	2	6	4	2	1
> PinSetterView	low	low	low-medium	low	22	17	1	11	111	106	4	4
> PinSetter	low	low	low	low	12	9	1	15	47	41	6	5
> PinSetterEvent	low	low	low	low	6	1	1	9	26	21	6	4
> PinSetterObserver	low	low	low	low	1	0	1	1	2	1	1	0
> PrintableText	low	low	low	low	10	8	1	5	21	18	2	2
> Queue	low	low	low	low	8	3	1	5	12	10	5	1
> Score	low	low	low	low	5	0	1	5	16	12	5	3
> ScoreHistoryFile	low	low	low	low	10	8	1	4	20	18	0	0
> ScoreReport	low	low	low-medium	low	29	23	1	13	76	74	5	1
> drive	low	low	low	low	4	2	1	1	8	7	0	0






























From the above metrics, it was evident that we needed to reduce complexity, coupling, and Lack of cohesion in many of the classes. We made the changes as mentioned previously to achieve our goal.

As previously documented, we used several design patterns and removed code smells to achieve the following final results. Many metrics were clearly out of range based on the above analysis. I refactored the classes that contained a large number of lines of code.

After Refactoring



Name	Complexity	Coupling	Size	Lack of Coh...	CBO	RFC	SRFC	DIT	NOC	WMC	LOC
Unit2											
AddPartyView	low-medium	low	high	low						447	2003
Bowler	low-medium	low	low-medium	medium-high	4	60	38	1	0	23	121
BowlerFile	low	low	low	low	0	4	0	1	0	4	14
ControlDesk	low	low	low	low	1	13	10	1	0	6	40
ControlDeskView	low-medium	low-medium	low-medium	low-medium	5	38	23	2	0	20	60
Emoticon	low-medium	low-medium	low-medium	low-medium	7	58	37	1	0	9	95
Emoticon	low	low	low	low	0	9	8	1	0	6	20
EndGamePrompt	low	low	low-medium	low	0	23	19	1	0	9	55
EndGameReport	low-medium	low	low-medium	low-medium	4	56	45	1	0	18	118
Lane	medium-high	medium-high	low-medium	low-medium	12	91	49	2	0	58	209
LaneEvent	low	low	low	medium-high	2	11	0	1	0	11	41
LaneEventInterface	low	low	low	low	2	9	0	1	0	9	10
LaneObserver	low	low	low	low	1	1	0	1	2	1	2
LaneServer	low	low	low	low	1	1	0	1	0	1	2
LaneStatusView	low	low-medium	low-medium	low-medium	8	35	22	1	0	17	93
LaneView	medium-high	low	low-medium	medium-high	4	56	47	1	0	51	235
NewPatronView	low	low	low-medium	low	1	40	20	1	0	8	85
Party	low	low	low	low	0	2	0	1	0	2	6
PinSetterView	low	low	low-medium	low-medium	1	29	24	1	0	18	104
Pinsetter	low	low	low-medium	low-medium	3	15	12	1	0	17	53
PinsetterEvent	low	low	low	low	0	6	1	1	0	9	26
PinsetterObserver	low	low	low	low	1	1	0	1	3	1	2
PrintableText	low	low	low	low	0	10	8	1	0	5	21
Queue	low	low	low	low	0	8	3	1	0	5	12
Score	low	low	low	low	0	5	0	1	0	5	16
ScoreCalculation	low-medium	low	low-medium	low	4	12	6	1	0	44	88
ScoreHistoryFile	low-medium	low	low	low	1	12	9	1	0	5	29
ScoreReport	low	low	low-medium	low	4	30	23	1	0	13	73
ScoreSearch	low-medium	low	low-medium	low	1	24	14	1	0	25	110
ScoreSearchView	low-medium	low	low-medium	low-medium	3	57	41	1	0	36	200
SubscribeLane	low	low	low	low	3	5	3	1	0	4	7
alley	low	low	low-medium	low	2	17	14	1	0	7	56

1	Lane					209	medium-high	medium-high	low-medium	low-medium
2	ControlDeskView					95	low-medium	low-medium	low-medium	low-medium
3	LaneStatusView					93	low	low-medium	low-medium	low-medium
4	LaneView					235	medium-high	low	medium-high	low-medium
5	ScoreSearchView					200	low-medium	low	low-medium	low-medium
6	AddPartyView					121	low-medium	low	medium-high	low-medium
7	EndGameReport					118	low-medium	low	low-medium	low-medium
8	ScoreSearch					110	low-medium	low	low	low-medium
9	ScoreCalculation					88	low-medium	low	low	low-medium
10	ControlDesk					60	low-medium	low	low-medium	low-medium
11	PinSetterView					104	low	low	low-medium	low-medium
12	NewPatronView					85	low	low	low	low-medium
13	ScoreReport					73	low	low	low	low-medium
14	alley					56	low	low	low	low-medium

As can be seen above, there has been a significant improvement in the metrics after refactoring was done. This was accomplished by striking a compromise between competing objectives like low coupling and high cohesion, resulting in efficient classes with no dead or duplicate code.

The metrics helped us in finding which classes need more refactoring based on their coupling/cohesion. Eg: Lane class needed the most refactoring as nothing was on the lower side in metrics: complexity, coupling, or lack of cohesion.

RFC:Response For a Class

The number of methods that can potentially be invoked in response to a public message received by an object of a specific class.

DIT: Depth of Inheritance

Except for classes that extended the Observer class, inheritance has remained unchanged.

WMC: Weighted Method Count

The weighted sum of all class methods represents a class's McCabe complexity.

WMC has increased for a few classes because we removed the event classes and moved the event methods to the observable itself.

LOC: Lines of Code. Quality Attribute Related To: Size

Tried lowering LOC removing unnecessary code Modularizing code everywhere possible.

The code quality of the code base has significantly improved as a result of the refactoring. This was accomplished by striking a balance between competing criteria such as low coupling and high cohesion in order to create efficient classes with no dead or duplicate code.