<u>Software Engineering Course Project 2</u>
<u>Design Document</u>
Bowling Alley Management System

<u>Date of Submission</u>: 25 March 2019

<u>Team Members</u>
Swati Tyagi - 20172096
Ayush Jain - 201501209
Gokul B. Nair - 201502034

Link to the video and UML diagrams : <u>Here</u>

<u>Overview</u>

Context - Automation in Bowling Alley Management

The Lucky Strikes Bowling Centre (LSBC) aims automate their chain of bowling centres. The requirements include upgradation of the Pin Setting Equipment and its interactions with the Scoring Station and Local Databases to automatically detect the number of pins knocked by a player and calculation and record of subsequent scores in each play session.

Technical Details about Bowling

Each such bowling centre is a Bowling Alley, each of which consists of a number of Bowling Lanes. One game can be carried out in each Lane at any instant. Each Lane can handle from one to six players, but not more. Each game consists of a number of Frames, for each of which each player is scored according to the respective performance.

Product Features

The product enables the admission of players as an individual or as a party. The admission of players happens at the Control Desk and an appropriate lane assignment takes place. In case a set of players want to join as a party, all of them will be assigned into a single lane. The order in which the players get admitted determines the order in which each player bowls.

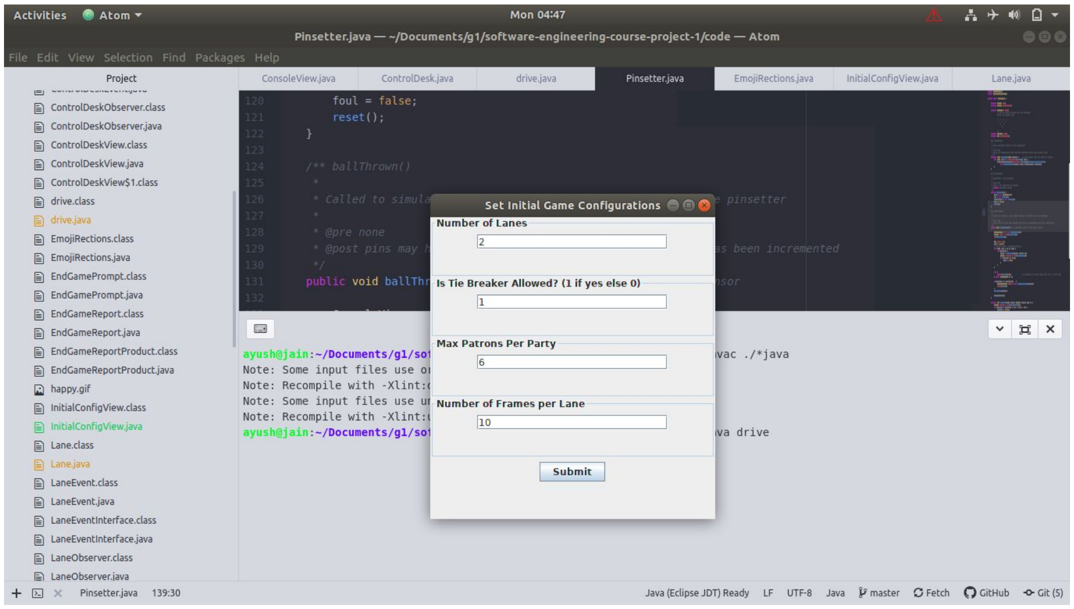
Each Lane houses a unique Scoring Station which keeps track of each players score. The upgraded Pin Setting Equipment detects the missing pins after each Throw automatically and reports the Pins' status to the Scoring Station. The players are rewarded with extra points in case of a Strike or a Spare while in case of a Foul, a zero score is considered. If a bowler throws two consecutive gutters, the player is penalized half of the highest score obtained. The scores are computed at the Scoring Station for each player and the same are reported to the Control Desk once any player completes a game. At the end of 10 frames, second highest player is provided an additional turn to bowl and if the player becomes highest, three additional frames are played between 1st and 2nd highest till the winner is finalized. In case of a tie-break, the one with most strikes is declared the winner.

The player can view the respective Pin Setter status at the Lane's Scoring Station. Also, the individual scores for each Frame is also maintained at the Scoring Station. Once a game ends, the players can choose to leave or restart another game. Player also have a provision to query the station for the highest and lowest scores as well as players. In addition the player can query the station to get his own rank among all the players.

The game has implementation of emoticons to appreciate and shame the player based on their scores.When a player Strikes, then a "Happy" emoticon expresses the achievement while on a gutter "Sad" emoticon is used to convey the low score.

The incorporation of new requirements in the existing system looks as follows:

1. *Game setting configuration*:

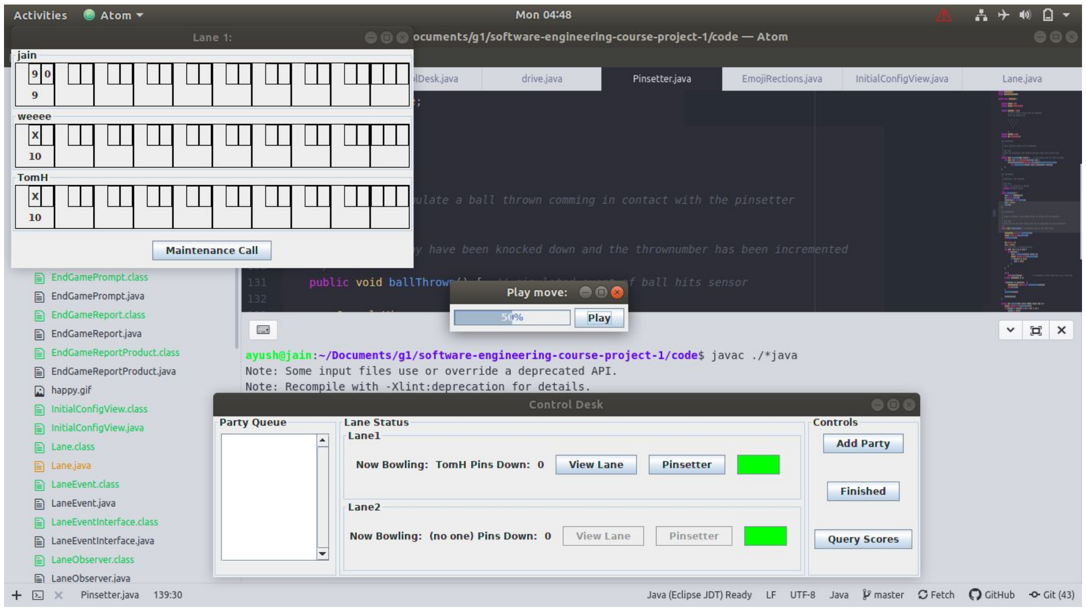


```
        foul = false;
        reset();
    }

    /** ballThrown()
     *
     * Called to simula                                    e pinsetter
     *
     * @pre none
     * @post pins may h                                    as been incremented
     */
    public void ballThr                                    nsor
```

```
ayush@jain:~/Documents/g1/so                               vac ./*java
Note: Some input files use o
Note: Recompile with -Xlint:
Note: Some input files use u
Note: Recompile with -Xlint:
ayush@jain:~/Documents/g1/so                               va drive
```

Number of Lanes	2
Is Tie Breaker Allowed? (1 if yes else 0)	1
Max Patrons Per Party	6
Number of Frames per Lane	10

2. *Play to Throw:* The player have to select/click the play button on a value that indicates the accuracy of throwing the ball. The more closer player is to the middle (50%) of the more, there are chances of knocking off 10 balls at a go.



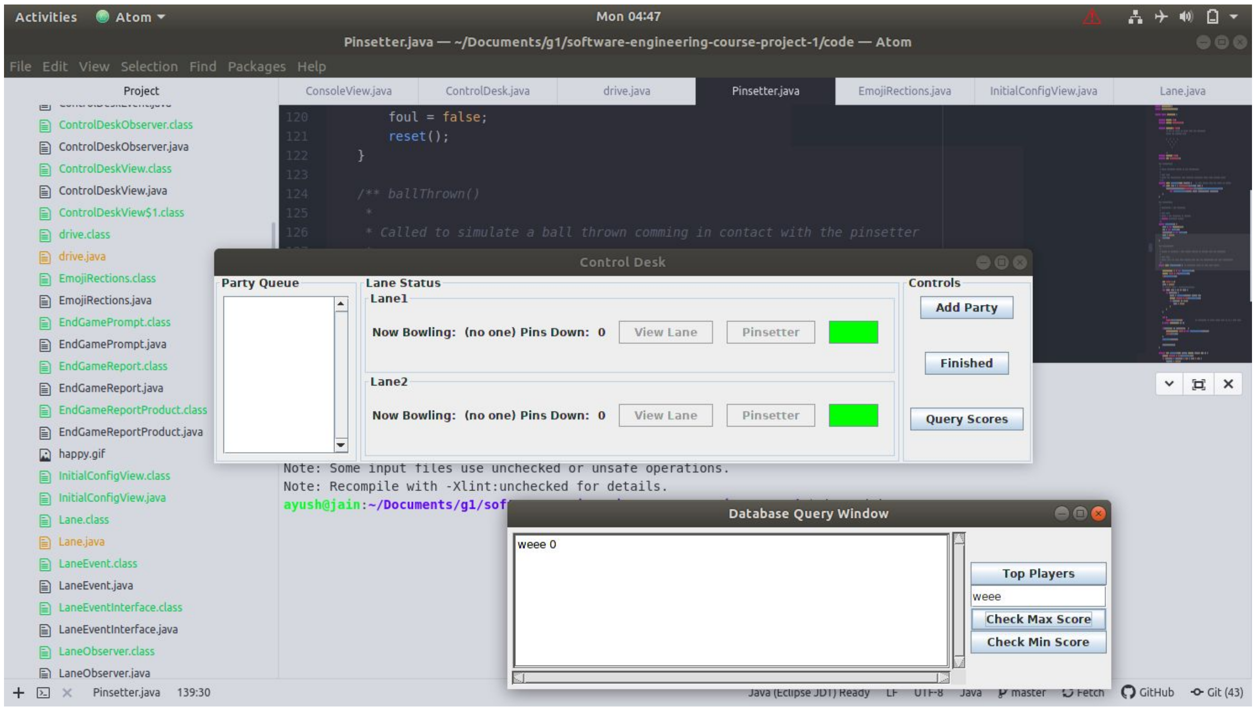
```
ulate a ball thrown comming in contact with the pinsetter

y have been knocked down and the thrownumber has been incremented

public void ballThrown() {         of ball hits sensor
```

```
ayush@jain:~/Documents/g1/software-engineering-course-project-1/code$ javac ./*java
Note: Some input files use or override a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

Lane1 — Now Bowling: TomH Pins Down: 0

Lane2 — Now Bowling: (no one) Pins Down: 0

3. *Querying the system* : A player can query the station to get the highest scored player. Querying gives provision to get the lowest and highest score of a particular player as well.



```
foul = false;
    reset();
}

/** ballThrown()
 *
 * Called to simulate a ball thrown comming in contact with the pinsetter
```

Lane1 — Now Bowling: (no one) Pins Down: 0

Lane2 — Now Bowling: (no one) Pins Down: 0

```
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
ayush@jain:~/Documents/g1/sof
```

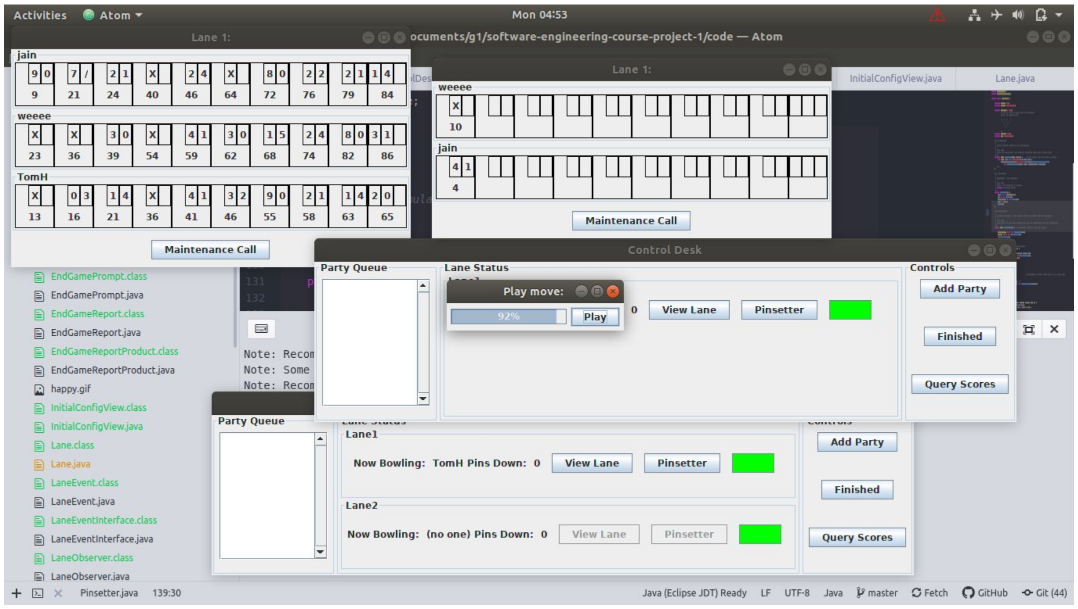
Database Query Window

```
weee 0
```

weee

4. *Extending the Game between the highest and second highest players*: At the end of 10 frames, if the 2nd highest player bowls and scores equal or higher than the highest

player then the game continues with 3 additional frames between 1st and 2nd highest till the winner is finalized. If there is a tie-break, declare the winner that had most strikes.



Lane 1:

jain									
9 0	7 /	2 1	X	2 4	X	8 0	2 2	2 1	1 4
9	21	24	40	46	64	72	76	79	84

weeee									
X	X	3 0	X	4 1	3 0	1 5	2 4	8 0	3 1
23	36	39	54	59	62	68	74	82	86

TomH									
X	0 3	1 4	X	4 1	3 2	9 0	2 1	1 4	2 0
13	16	21	36	41	46	55	58	63	65

Lane 1:

weeee									
X									
10									

jain									
4 1									
4									

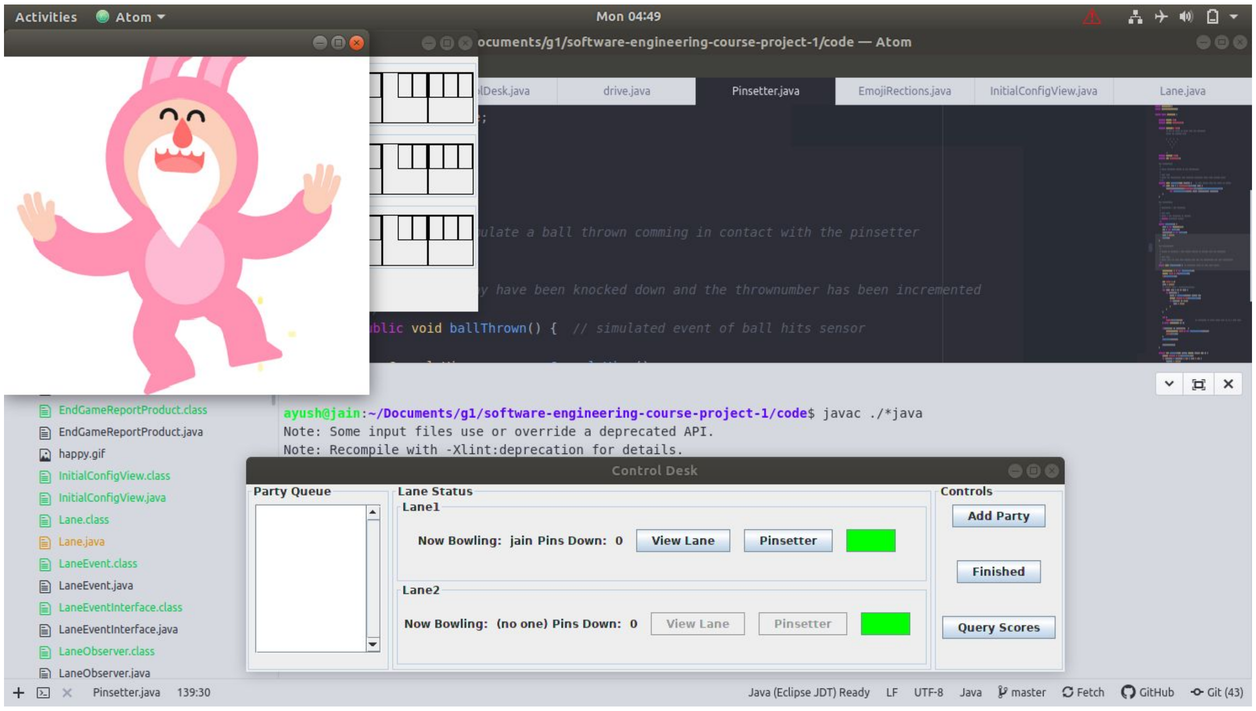
Control Desk

Play move: 92% Play

Lane1 Now Bowling: TomH Pins Down: 0

Lane2 Now Bowling: (no one) Pins Down: 0

5. *Emoticon to react on the score* :



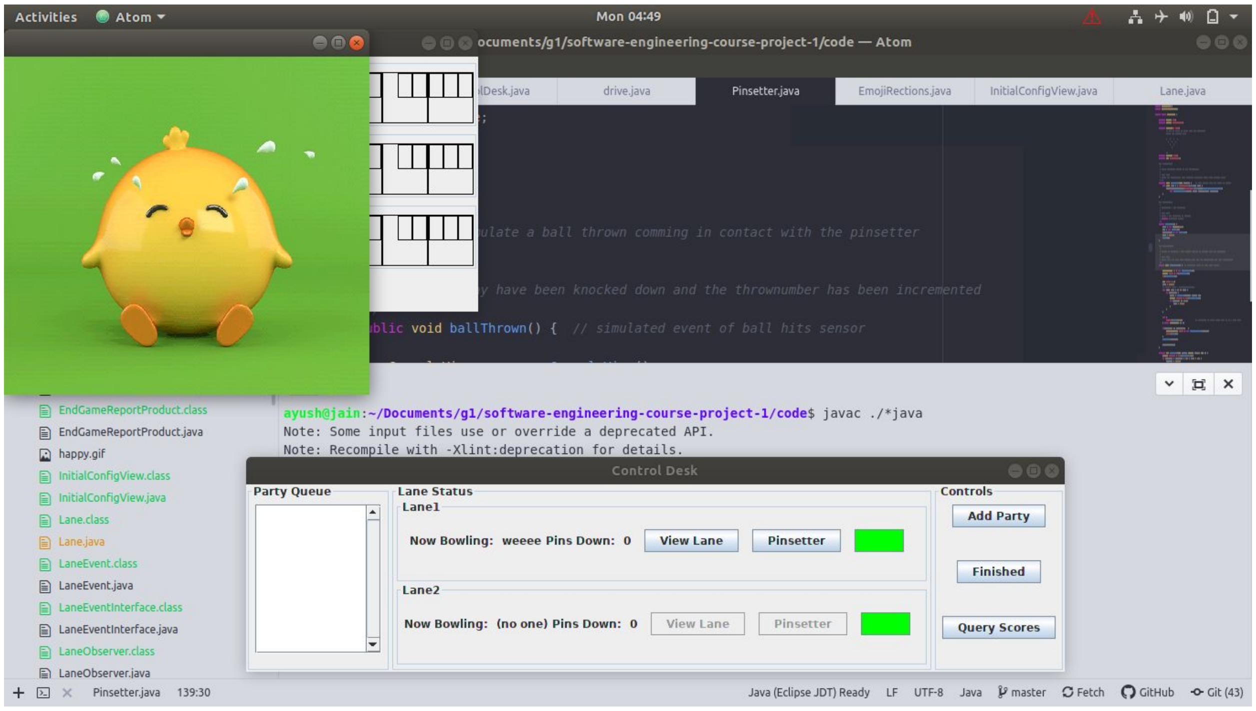
```
ulate a ball thrown comming in contact with the pinsetter

y have been knocked down and the thrownumber has been incremented

blic void ballThrown() {  // simulated event of ball hits sensor
```

```
ayush@jain:~/Documents/g1/software-engineering-course-project-1/code$ javac ./*java
Note: Some input files use or override a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

Control Desk

Lane1 Now Bowling: jain Pins Down: 0

Lane2 Now Bowling: (no one) Pins Down: 0



```
ulate a ball thrown comming in contact with the pinsetter

y have been knocked down and the thrownumber has been incremented

ublic void ballThrown() {  // simulated event of ball hits sensor
```

```
ayush@jain:~/Documents/g1/software-engineering-course-project-1/code$ javac ./*java
Note: Some input files use or override a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

Control Desk

Party Queue

Lane Status

Lane1

Now Bowling: weeee Pins Down: 0 View Lane Pinsetter 🟩

Lane2

Now Bowling: (no one) Pins Down: 0 View Lane Pinsetter 🟩

Controls

Add Party

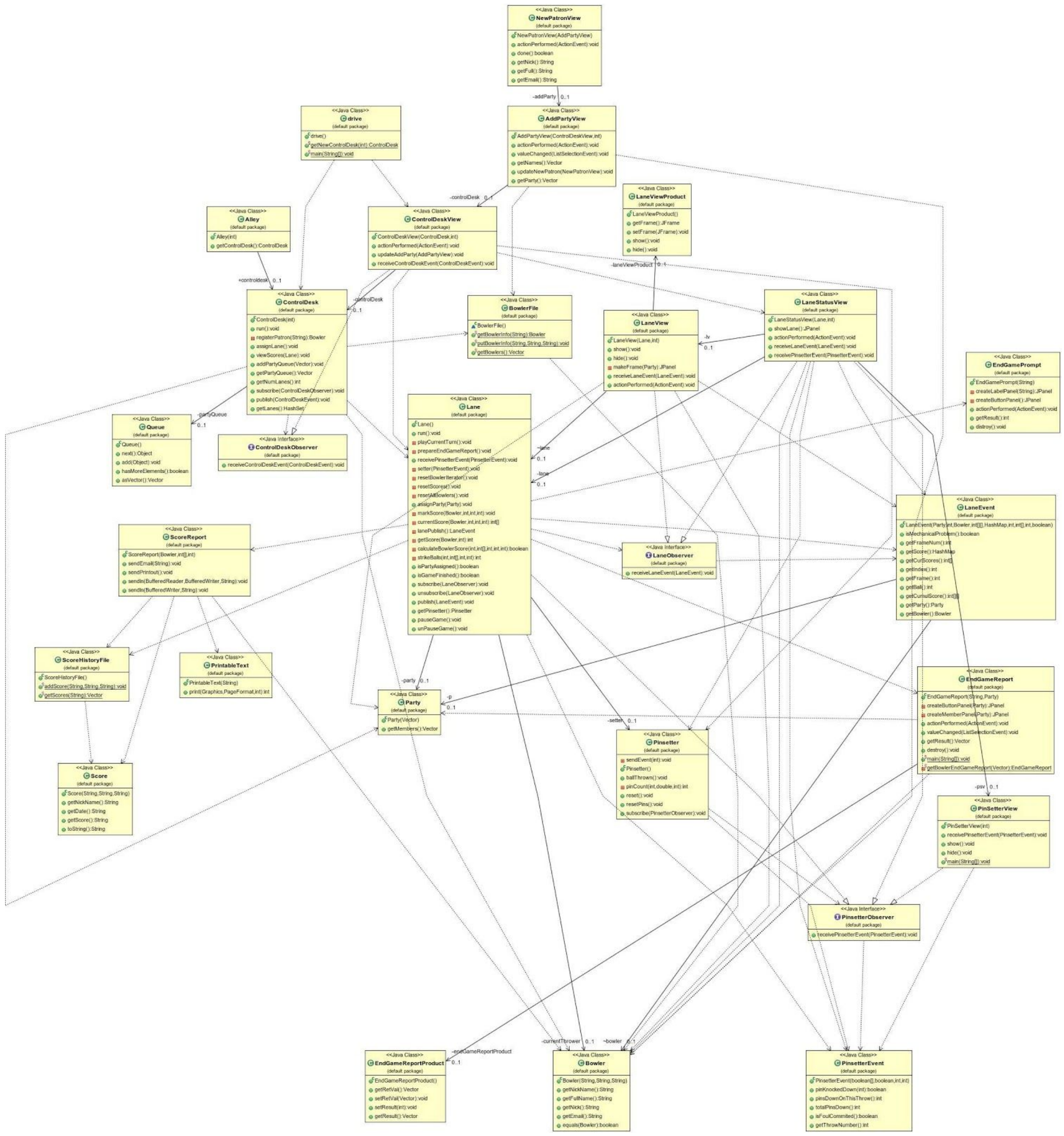
Finished

Query Scores

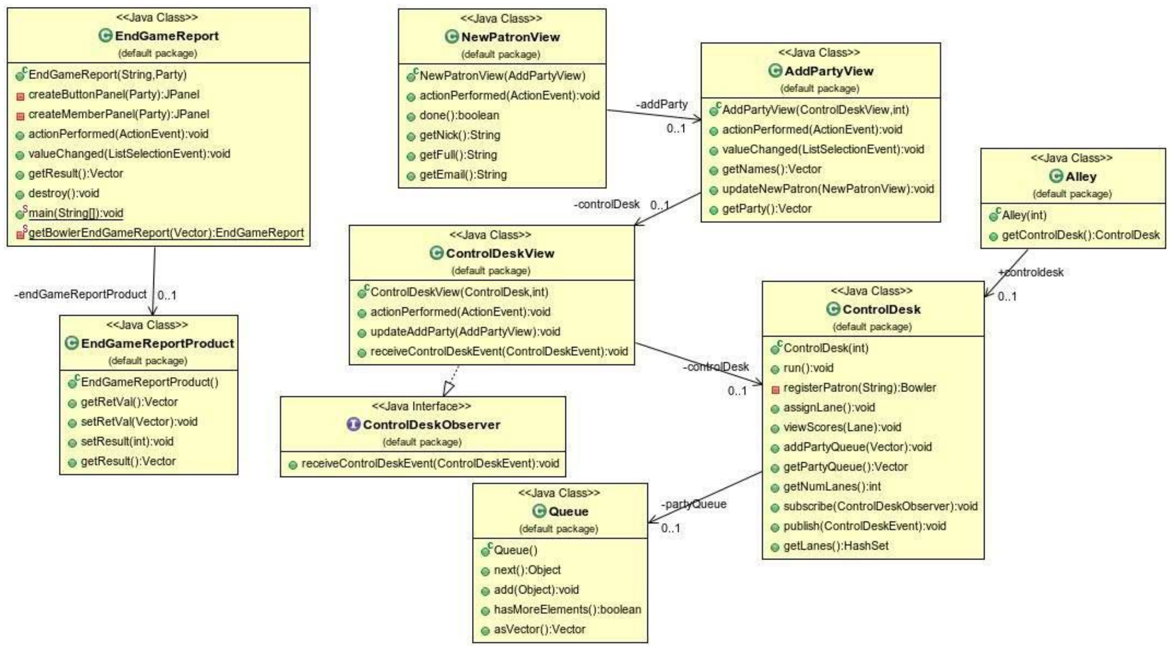
UML Class Diagrams

Original Implementation

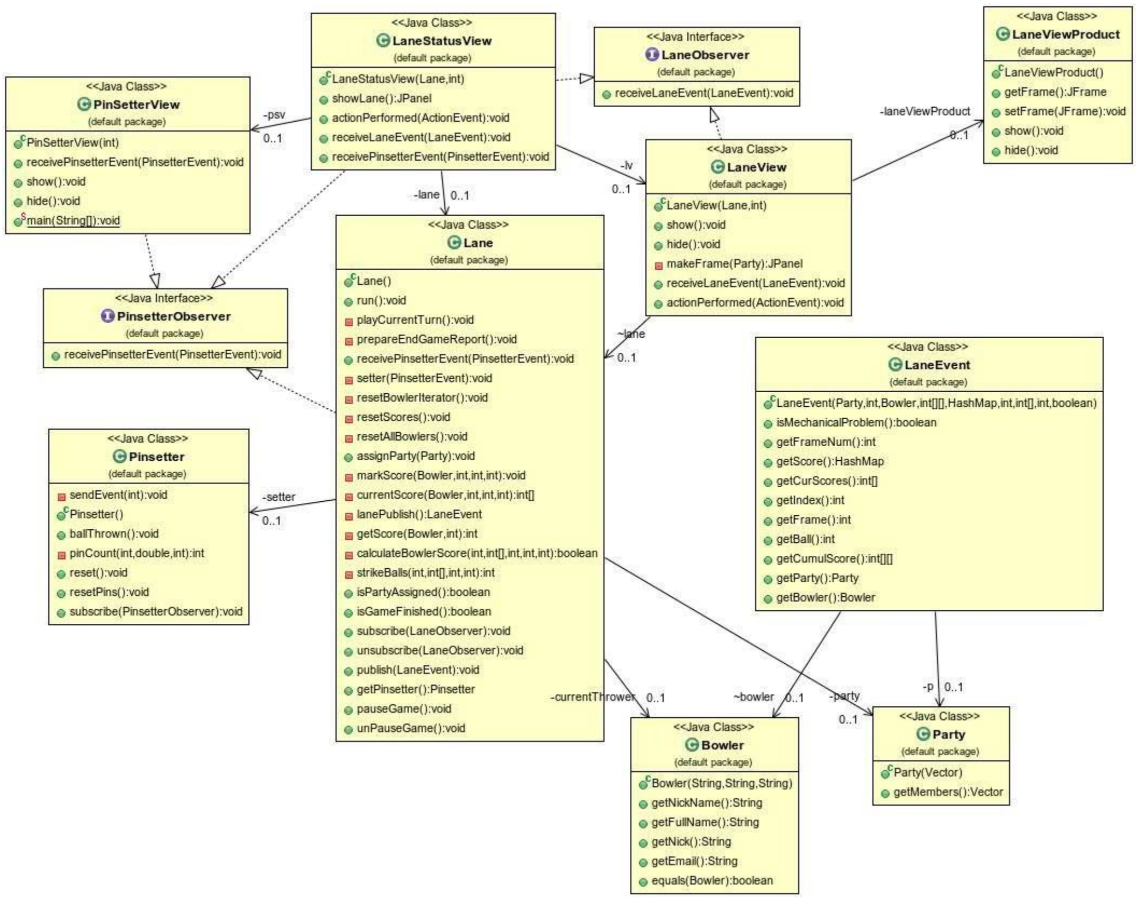
With Dependencies



ControlDesk Module

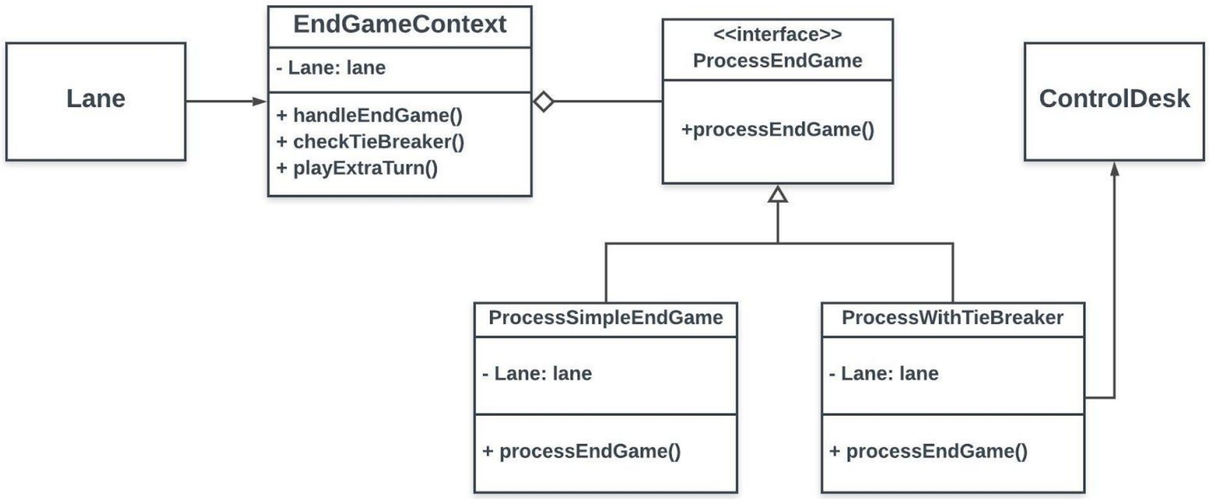


Lane Module

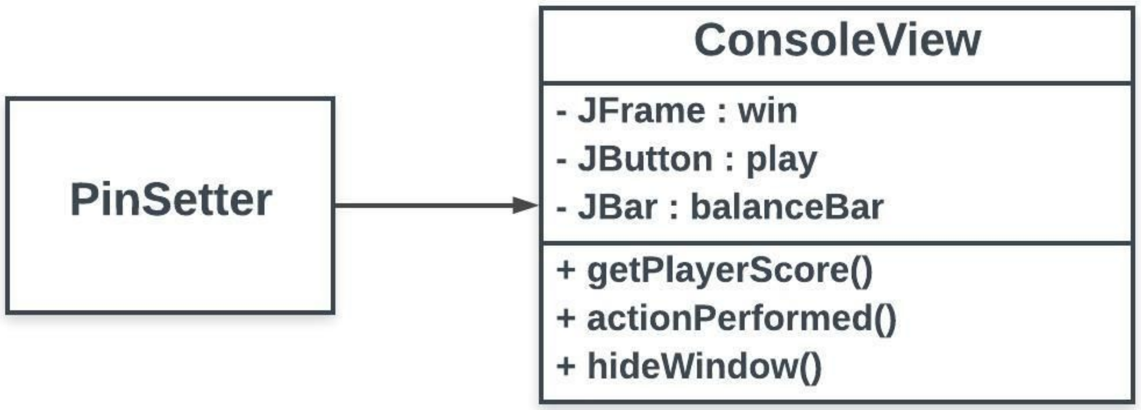


Modified Subsystems Implementation

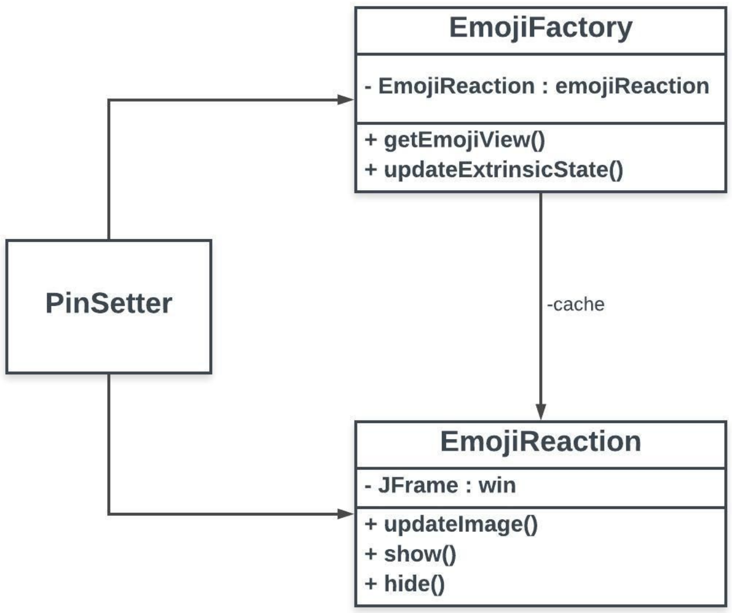
EndGameContext : Strategy Pattern



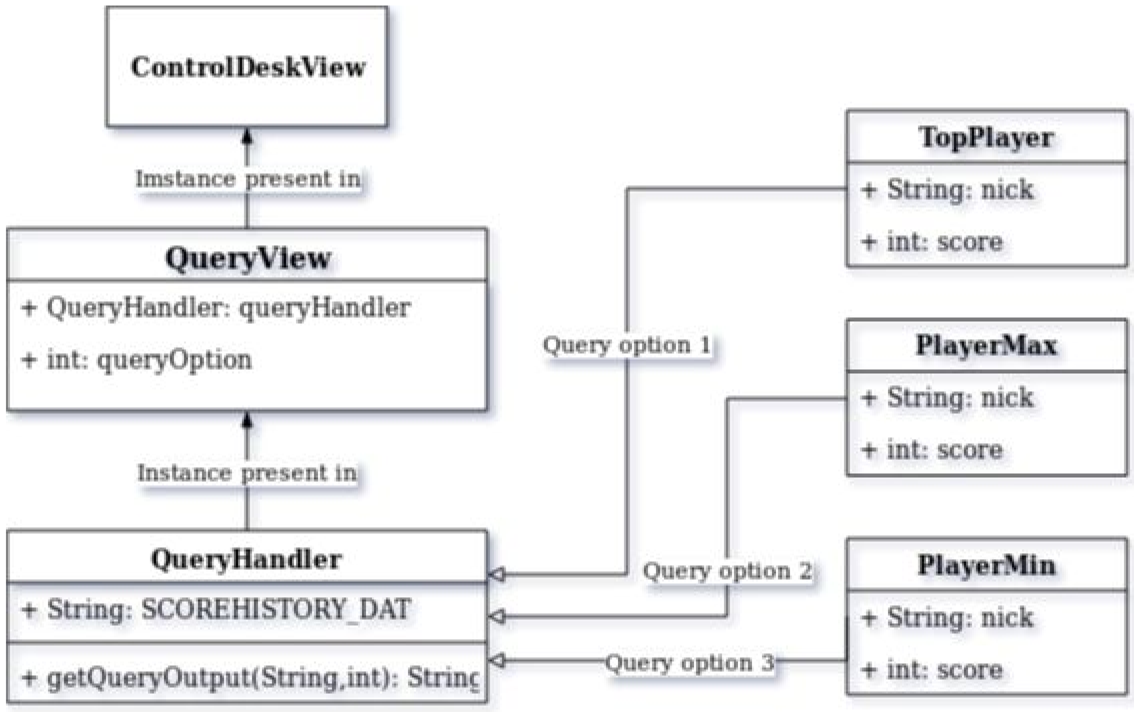
PinSetter



Emoji : Flyweight

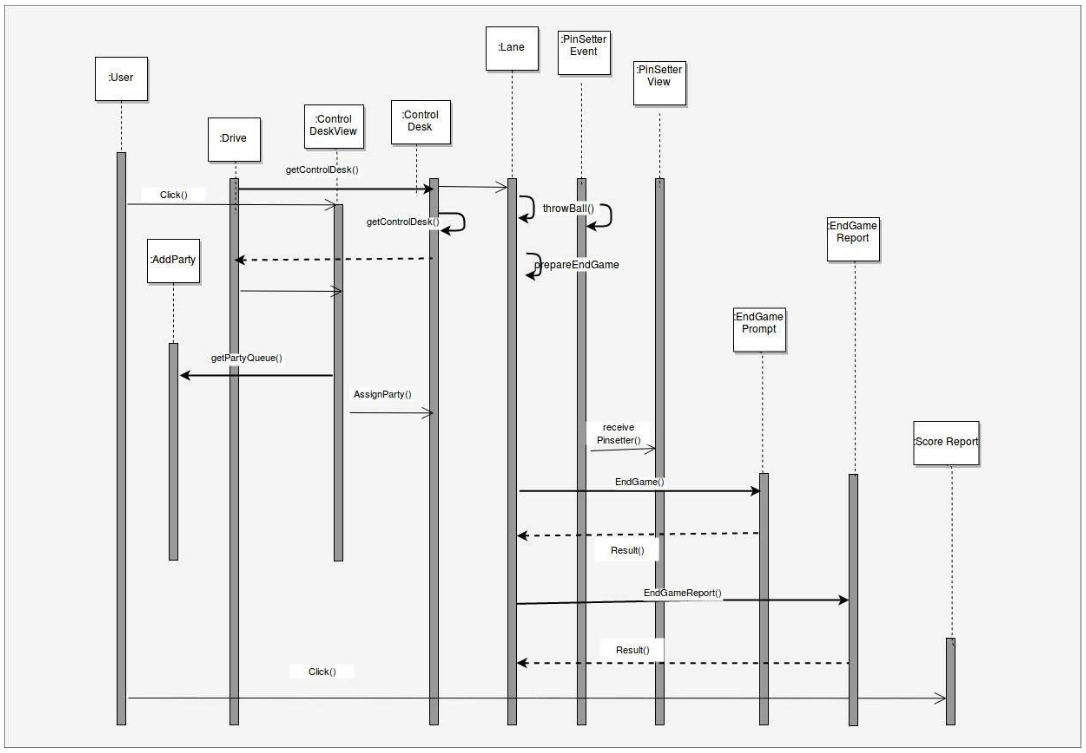


Database Query : Template-Method



UML Sequence Diagrams

Original Implementation



Class Descriptions

Sl. no.	Class Name	Class Description
1	drive	- Creates the whole bowling alley by creating the respective Control Desk instance.
2	ControlDesk	- Admits a player or a party into the alley. - Adds a party to a waiting queue if no Lane is free. - Assigns Lane to a party appropriately. - Allows objects to be added as subscribers. - Broadcasts event updates to everyone listed as subscriber.
3	Queue	- Creates a new party waiting queue.
4	Lane	- Represents the Lane and its Score Station. - If a game is going on for a party, handle the sequence of throws for the bowlers in the party. - If a game is over, it proceeds to preparation of game report. - Provides choice of starting another game to to the party which finished.

		- Receives Pin Setter outputs and calculates score after each throw accordingly at the Score Station.
5	PinSetter	- Handles resetting, simulation and detection of pins appropriately during each throw.
6	Bowler	- Maintains each Bowler’s information.
7	BowlerFile	- Handles interactions with bowlers’ information database.
8	EndGameContext	- maintains a reference to a ProcessEndGame object and decides on runtime which subclass implementation to invoke.
9	ProcessSimpleEndGame	- implements the game ending algorithm using the ProcessEndGame Strategy interface
10	ProcessWithTieBreaker	- implements the tie breaker algorithm with addditional 3 frames using the ProcessEndGame Strategy interface
11	EmojiFactory	- Creates and manages Emoji objects. Creates frame for containing the image
12	EmojiReaction	- Adds storage for the emojis
13	Party	- Maintains the Bowlers’ list who belong to the party.
14	ScoreHistoryFile	- Handles interactions with stored database.
15	QueryHandler	- Takes query as input and displays corresponding output.
16	All Event Classes	- Represents respective events related to respective classes.
17	All View Classes	- Provides GUI representations of respective classes.
18	All Observer Classes	- Creates an instance for each subscriber of respective observable components.

Analysis of Original Design

The original refactored codebase currently involves a running implementation of the Bowling Alley in a simple way with little or no control to User for configuring the system or even while playing.

The implementation involves random generation of number with equal probability that gives the number of knocked down pins at each throw. We had to come up with a mechanism where the user can have control over the the pins that will be knocked down so the game can be more interactive for the players playing the game.

The program also have hardcoded game settings which are configurable only inside the code, by changing the variable values as of now. This could be avoided by giving some Station admin, the privileges to configure these parameters at run-time instead of keeping them static for each game.

Current implementation do consider rewards, in the form of extra scores, in case of a strike or spare but have no provision for any penalties in case of gutters. The fouls only accounts for a 0 score.

With respect to the design strengths, the original codebase has the modularization to deal with *Long Class* code-smell and with which we also ended up obtaining *Higher Cohesion* in the codebase. The code is free of any dead modules.

As a consequence to tackling Cyclomatic Complexity, the code has no unnecessary instances of nested if conditions as they were simplified into single if statements with appropriate fusion of conditions using the right logical operations. This also helped reduce the Long Method code-smell to quite some extent.

Since the system is a full stack application. Therefore for every event in pinsetter or for every ball thrown there are two things which needs to be updated first one is the the view or frontend, which needs to show the current state of the pindrop and next is the backend, where the Lane should update it's score changed by the current pindrop through the Score Station. So since the requirement of system here is to notify some classes when some event is triggered in the pinsetter class. That's why Observer Pattern is used here. Because of the Observer Pattern the pinsetter is not tightly coupled with any of the class which is supposed to receive the event update and this allows any class to become a listener or observer. It just needs to implement the Observable Class' interface. This pattern also helps in extension of codebase, whenever more listeners are need to be added they can easily be plugged in to the system simply by implementing the interface. Moreover,Adapter Pattern is also present in the codebase during file database I/O.

Since there are multiple classes which interact with each other very often, but still the code setup doesn't have any class which would be acting as an Unnecessary Middleman.

To avoid any indecent exposure of any data members of any classes, it is properly taken care of through Data Hiding, wherever required. This in turn have also reduced Coupling as a few otherwise public members which were being taken for granted by other class modules are either refrained from being accessed or accessed through a public member function if necessary.

In this way, the relationship between various class modules and their respective view, event and in some cases product class modules is refined in the codebase. This is in accordance with the Law of Demeter and thus further leads to Loose Coupling in the code implementation.

Analysis of Design catering new requirements

As a first step in order to make the game interactive, we tried to include the player's involvement with the system by making him make few selections. Initially, the Station admin can now make selection of number of lanes, number of Frames, number of Patrons and if the he wants to have a tie-breaker or not.
User has full control over the power and based on the accuracy with which he "Play", the number of pins are taken down instead of some random knocking of pins.
The UI functionalities implemented adhere to the UI design patterns. Few of those are highlighted as below:

➢ *Navigation - Progressive Disclosure :* In order to avoid the problem of information overload and to show only the information or features relevant to the user's current activity and delay other information until it is requested, the Querying dashboard is implemented separately. Also, on adding a party when a game starts, the Lanes (game) is not shown right away but User has an option to click on the "View Lane" button to check the game going on the selected lane.

➢ *Notifications :* In order to take care of the fact that the user be informed about important updates and messages, relevant dialog boxes are implemented as and when appropriate actions are taken.

➢ *Continuous Scrolling :* The user needs to view a subset of data that is not easily displayed on a single page. In order to represent multiple players in a box that can accommodate only 8 names, scrolling is implemented and since only a max of 6 names could be added on the selection box.

➢ *Clear Primary Actions :* Primary actions that lead to the completion of a form; for example, clicking "Finished" or "Print Report" or "Play" are implemented to give clear indication of the action to the User.

Naming a few persuasive gaming mechanics design patterns implemented as part of our implementations:

➢ *Fixed rewards :* We have use rewards in the form of appreciation emoticons to encourage continuation or introduction of wanted behavior. The appreciation and shame emoticons also depict the influence of the action.

➢ *Achievements :* Showing the scores after each throw and comparison with other players in the same party are used so that meaningful achievements are recognized.

Apart from the above mentioned UI patterns, few design patterns are also part of the new design that makes the code more reusable and the design more simple.

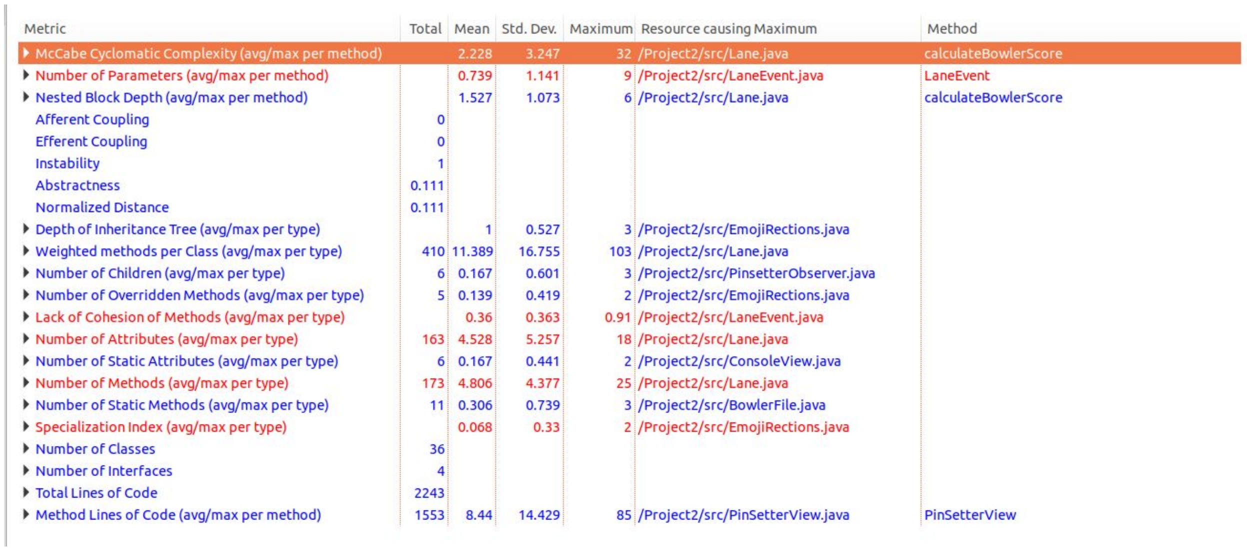
➢ *Strategy Pattern*
As according to the requirement of implementing tie breaker (i.e. to give second highest player a chance to win the game) the lane at the end of game should give second highest player a chance and on the basis of the chance of second highest player game strategy further is decided. That's why we used strategy pattern to solve this problem. In our solution the client is Lane class who needs one of the strategy to be executed on the basis of the context. The context is decided by the class EndGameContext which give second highest player a chance and then decide the strategy to be followed ahead. All the strategy implements a interface ProcessEndGame which contains signature of processEndGame and concrete implementation is inside the concrete strategy i.e. ProcessSimpleEndGame, ProcessWithTieBreaker. So now using the state pattern the strategy is decided during runtime.

➢ *Flyweight pattern*
The emoji window will be used multiple time in the game play and for different emoji but at a time single emoji will be presented. Therefore it is not efficient to everytime create new window object and render image in the window. Creating new objects will cost us with wastage of the memory and also every time creating complex window object from the scratch will waste a lot of time. That's why we used Flyweight design pattern with emoji display window as flyweight object having intrinsic state as the windows parameters and JFrame, Jpanel etc objects required to make window since they will be same always. This flyweight has image as it's extrinsic state. The client (Pinsetter) will contact to emoji factory for getting new emoji window. The factory will create the instance if doesn't exist otherwise return the reference to existing flyweight object.

➢ *Template Method Pattern :* The Query Handling Mechanism goes through a branched pipeline which branches out based on the specifics of the query itself. The initial part of the pipeline is common irrespective of what the query is. The specification of the query itself comes up in subclasses hence completing the whole pipeline to be taken appropriately. This makes the Query Handler extendable such that, if any new query types have to be added later, it can be done by not introducing huge changes in the QueryHandler Class.

Analysis based on Metrics

Metric values after implementing the new requirements for the entire codebase is as follows:



Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
McCabe Cyclomatic Complexity (avg/max per method)		2.228	3.247	32	/Project2/src/Lane.java	calculateBowlerScore
Number of Parameters (avg/max per method)		0.739	1.141	9	/Project2/src/LaneEvent.java	LaneEvent
Nested Block Depth (avg/max per method)		1.527	1.073	6	/Project2/src/Lane.java	calculateBowlerScore
Afferent Coupling	0					
Efferent Coupling	0					
Instability	1					
Abstractness	0.111					
Normalized Distance	0.111					
Depth of Inheritance Tree (avg/max per type)		1	0.527	3	/Project2/src/EmojiRections.java	
Weighted methods per Class (avg/max per type)	410	11.389	16.755	103	/Project2/src/Lane.java	
Number of Children (avg/max per type)	6	0.167	0.601	3	/Project2/src/PinsetterObserver.java	
Number of Overridden Methods (avg/max per type)	5	0.139	0.419	2	/Project2/src/EmojiRections.java	
Lack of Cohesion of Methods (avg/max per type)		0.36	0.363	0.91	/Project2/src/LaneEvent.java	
Number of Attributes (avg/max per type)	163	4.528	5.257	18	/Project2/src/Lane.java	
Number of Static Attributes (avg/max per type)	6	0.167	0.441	2	/Project2/src/ConsoleView.java	
Number of Methods (avg/max per type)	173	4.806	4.377	25	/Project2/src/Lane.java	
Number of Static Methods (avg/max per type)	11	0.306	0.739	3	/Project2/src/BowlerFile.java	
Specialization Index (avg/max per type)		0.068	0.33	2	/Project2/src/EmojiRections.java	
Number of Classes	36					
Number of Interfaces	4					
Total Lines of Code	2243					
Method Lines of Code (avg/max per method)	1553	8.44	14.429	85	/Project2/src/PinSetterView.java	PinSetterView

Following metrics with the given recommendation has been used to analyze the code base quality:

1. Cyclomatic Complexity:
   Defined for types and methods. Cyclomatic complexity is a popular procedural software metric equal to the number of decisions that can be taken in a procedure.
   Methods where CC is higher than 15 are hard to understand and maintain. Methods where CC is higher than 38 are extremely complex and should be split in smaller methods.

2. Number of parameters:
   Methods where NbVariables is higher than 8 are hard to understand and maintain. Whereas the methods where NbVariables is higher than 15 are extremely complex and should be split in smaller methods.

3. Depth of Inheritance Tree:
   Types where Depth Of Inheritance is higher or equal than 6 might be hard to maintain. However it is not a rule since some time the classes might inherit from third-party classes which have a high value for depth of inheritance. For example, the average depth of inheritance for third-party classes which derive from System.Windows.Forms.Control is 5.3.

4. Number of Children:
   Recommended value is between 1 and 4 :The upper and lower limits of 1 and 3 correspond to a desirable average. This will not stop certain particular classes being the kind of utility classes which provide services to significantly more classes than 3.

5. Number of Attributes:
   Recommended value is from 2 to 5 : A high number of attributes (> 10) probably indicates poor design, notably insufficient decomposition, especially if this is associated with an equally high number of methods. Classes without attributes are particular cases, which are not necessarily anomalies. These can be interface classes, for example, which must be checked.

6. Number of Methods:
   This value needs to remain between 3 and 7 for a class. This would indicate that a class has operations, but not too many. A value greater than 7 may indicate the need for further object-oriented decomposition, or that the class does not have a coherent purpose. A value of 2 or less indicates that this is not truly a class, but merely a data construction.

7. Method Lines of Code:
   Methods where NbLinesOfCode is higher than 20 are hard to understand and maintain. While if it's higher than 40, they are extremely complex and should be split into smaller methods.

8. Lack of Cohesion of Methods:
   Types where LCOM > 0.8 and NbFields > 10 and NbMethods >10 might be problematic. However, it is very hard to avoid such non-cohesive types. Types where LCOMHS > 1.0 and NbFields > 10 and NbMethods >10 should be avoided. This constraint is stronger and thus easier to satisfy than the constraint types where LCOM > 0.8 and NbFields > 10 and NbMethods >10.

The above mentioned metrics are used to identify defects, assess productivity and identify areas of improvement within applications. It is a known fact that less complex code is easier to maintain. So the measurements are used at the code level to identify defects or overly complex code and make the necessary fixes.

The information gained from these metrics helped to identify if the piece of the code needs to be modified or should be replaced. There are few methods and classes that has some complicated logic which were identified and checked for refactoring as such classes are more error prone.

This also helped prioritize different possible code fixing actions required during refactoring.

Refactoring on the basis of the metrics require a trade-off to be maintained between various values. Fixing one metric may result in shooting the value for a different parameter. Following result is an evidence for the same.