Unit 2 Submission

Bowling Alley

This document contains details on updates and enhancements made to the original codebase of Bowling Alley.

Contents:

Team Members and Responsibilities

Name	Roll Numbers	Contribution in Unit2
Gajanan Modi	**2019201049**	● implemented all refactoring changes after unit1 mention in refactoring part of Doc ● Made code working for multiplayer, and maximum number of players as 6. Provide an option to add and store players names ● Removed game simulation engine to make game more player Dependent ● Integrated DynamicBallScore to generate score after each click on throw button ● To Reduce Complexity in UserView Integrated someview classed Together And created single view class ● Resume/Pause Button implemented ● Some bug fixing of initial design ● Sequence Diagrams and UML in Doc ● Documentation of Antipatterns, overview, Info about Features implemented by me
Utkarsh Upendra	**2020900047**	● Added Database layer to enable queries ● Added a cache layer on top of the database layer to enable get/updates on configurable parameters in the code ● Added a window to run queries on database ● Added emoticons for each throw Refactored and implemented Game State handling logic using State pattern ● Added a login view and logic to control access to the application ● Explored and ran metrics analysis on the codebases ● Created UML diagrams for enhancements
Antony Martin	**20171030**	● Added score penalty for two consecutive gutters ● Modified end game report to show Winner of the game, and calculate winner of the game based on number of strikes in case of tie breaker ● Changed rules of the game to accommodate for giving a chance to the second highest player to play off against the highest scoring player if he is able to

		get a higher score after the game is over.
Aditya Khandelwal	**20171117**	● Added images of pins for each pin from 1-10 ● Changed the front-end layout of end game report ● Added UML sequence diagrams in the report ● Majorly worked for testing and finding bugs ● Added some style overhaul to the login page and endgame report ● Documented class responsibilities

Overview

The Bowling Management System is a game that is entirely developed in Java. It is a virtual game that enables players to enjoy the fun of bowling from their laptops. The update in game simulates several features that add to the overall appeal of the game. Some of the significant features included in the game are:

- Control Desk: The control desk operator has the ability to monitor the scores of any active lane. To reduce confusion of player we stack two view in single button so that UI will be more user friendly. The score of an individual scoring station or multiple scoring stations.
- Creating a new player: A New Patron can be created to play the game. This player is then added to SQLite Database Connection through establishing connection to database
- Adding a new party: Selected number of bowlers can be added to a party which is then assigned to one of the free lanes to begin playing a game. Here we updated maximum player can play this game to 6
- Viewing the Pinsetter: For a particular lane, the user can also view the pinsetter window which simulates the pins dropped on each ball-throw. The pinsetter will re-rack the pins (places all ten down) after two consecutive throws have been detected.
- Viewing the Scoreboard: We have made the score card more interactive by displaying emoticons according to players scores. We also changed the score generation logic from randomized to input based.

Refactoring post Unit 1

- Set the resetScores and resetBowlerIterator methods to public to allow GUI control.
- Added a few getters within Lane so that LaneStatusView is able to generate the end-of-game routine originally found in Lane.
- Transferred the end-of-game code in Lane to LaneStatusView, decoupling the Model from the GUI.
- We decided to keep the PinsetterEvent since it encapsulates data not inherited to Pinsetter.
- Fixed minor problems with the maintenance button.
- Added the ScoreCalculator class and migrated all of the getScore functionality originally belonging to main into that class. This is needed because as we can see getscore() a single function is complex with different functionality, extending getscores() methods with ScorePenalty will make it more complex

Classes Responsibilities

Class	**Responsibility**
AddPartyView	Creates a popup window with the option to add or remove a patron for a party. It also provides us an option to create a patron which will be in the party. Game is started by pressing the Finished button
Bowlerdb	Provides us a way to interact with the list of Bowlers in following ways - fetching / updating information of bowlers, adding / removing bowlers
ControlDesk	Represents control desk and carries out functionalities of the program specified by AddPartyView and ControlDeskView

ControlDeskView	Creates view, provides us with an UI for us to add a party or to finish the game. And also provides us with the view of waiting parties and the current lanes assigned.
drive	Starts game by creating Alley
EndGamePrompt	Creates a popup window when the game is over, and it provides us the option to restart the game or finish through its GUI interface
EndGameReport	Provides option to print the report or not using a popup window.
EntryPointView	This class represents a login screen
Lane	Keeps track of the current lanes and simulates the bowling game. It assigns lanes to parties, computes the scores, designs the functioning of the game (ensures everyone is getting their turn at the right time)
LaneEvent	Holds the values which define a lane like the frame number, current bowler, throw number and all.
LaneStatusView	Creates the view shown in the center of the ControlDesk where things like current bowler in each lane, lanes, pins down and etc are shown. It also provides us with an option to see pins status
LaneView	Creates view for the number of pins down in each throw for each player
NewPatronView	Creates view which lets us input the details about the new patron we are registering
Pinsetter	Simulates the dropping of pins in the lane by updating the states of each pin. Results are randomly generated for each throw
PinsetterView	Creates view which shows the status of pins so we can see what's going on the Lane
QueryResultView	This class represents a tabular view to display query results from database
QueryView	This class contains behavior to allow users to interact with the database by selecting what queries they want to run
Score	Sets the scores for the players in a game
ScoreCalculator	It is simple implementation of getscore method with some reduce in complexity also CalculatePenalty function is added in this class only

ScoreHistoryFile	Updates the scores in the .DAT file which contains history of scores for each player
ScoreReport	Generates the report for each player at the end of a game and prints/emails it. It includes the current score and previous scores for the given player
ViewUtils	A utility class that contains common methods to create UI Elements like Buttons, Labels and Input Boxes.

Initial Design

Weaknesses:

The original source material contains designs, code, and documentation that exhibits poor software engineering design principles as well as anti-patterns.

- Antipatterns

The Blob

No.	*Description Of Weakness*	*Fix*
1	Single controller class, multiple simple data classes seen throughout implementation in files such as Lane.java and ControllerView.java	Split into appropriate classes wherever necessary

Lava Flow

No.	*Description Of Weakness*	*Fix*
1	Large commented-out code with no explanations	Removed dead code and gained a full understanding of any bugs introduced.
2	Lot’s of “To Do” code	Wrote the code wherever necessary.

The Lava Flow code was refactored as mentioned under the Dead Code Code Smell in detail.

Functional Decomposition

No.	*Description Of Weakness*	*Fix*
1	Classes with a single method in use, no real use of the class data structure	Fixed by moving into existing class and/or combining classes

Golden Hammer

No.	*Description Of Weakness*	*Fix*
1	Several deprecated functions such as show(), hide(), new Integer() used throughout the code.	Development via linting tool such as Intellij helped fix this issue.

Cut-And-Paste Programming

No.	*Description Of Weakness*	*Fix*
1	Duplication/ Dead Code in many .java files	Refactored and eliminated using the DRY principle

Strengths

- The coupling amongst all the classes has been kept low
- A strong cohesion amongst related classes exists
- The size of the code files has been kept low and optimum
- One of the biggest strengths of the original design is the adherence it has to MVC. The model classes are highly independent. The control classes communicate between view and model classes as expected.

Fidelity To Design Documentation

- The original code has for the most part abided to the design documented listing the features it needs to be implemented
- A bowling alley is composed of a number of bowling lanes and parties bowl on these lanes

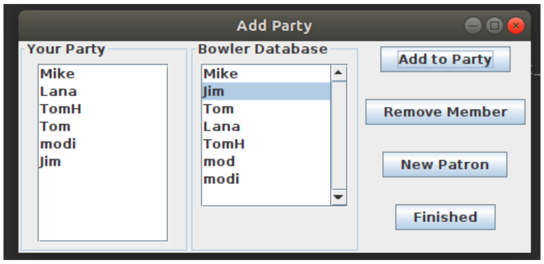
- Each lane is equipped with a stand-alone scoring station that lists the bowlers' names and a graphic representation of their scores.
- The pinsetter interface communicates to the scoring station the pins that are left standing after a bowler has completed a throw.
- The control desk operator has the ability to monitor the scores of any active lane. A configurable display option will allow the operator to view the score of an individual scoring station or multiple scoring stations.
  This display feature is not seen on the window panel and hence brings down the fidelity to design

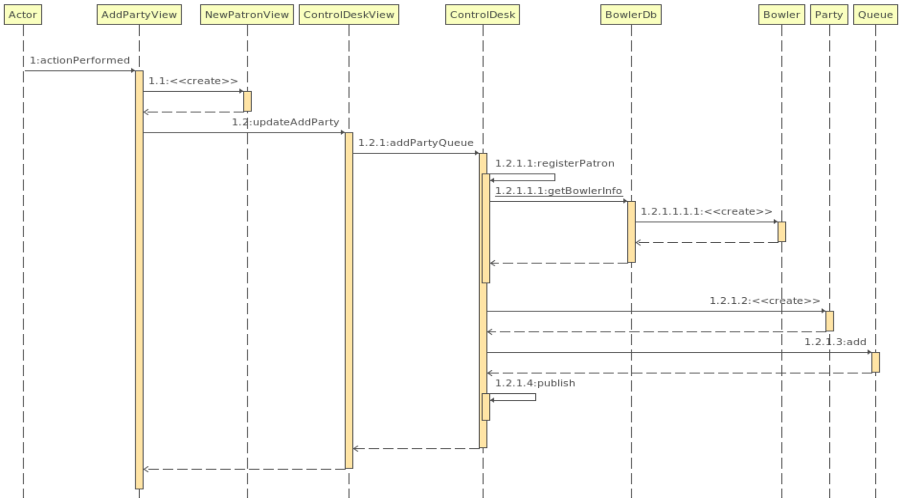
New Design

We did various changes as specified below to improve our design same time added New Features to make it user interactive.

1. **Making code extensible for a Maximum of 6 players and Multiplayer**

   - This is a minute change in the code. Since the code is already functional for a maxPatronsPerParty size of 5 players by default, all that was to be done in order to implement this feature was to modify this value from 5 to 6.
   - The file to be modified was drive.java. Then to make it multiplayer threads are used as for each new party separate lane is assigned as Maximum number of lanes





can be 3 only 3 parties can play parallely. All player are added to Alley.db Database using Sqlite Connection you can see in this sequence diagram

2. Database Functionality:

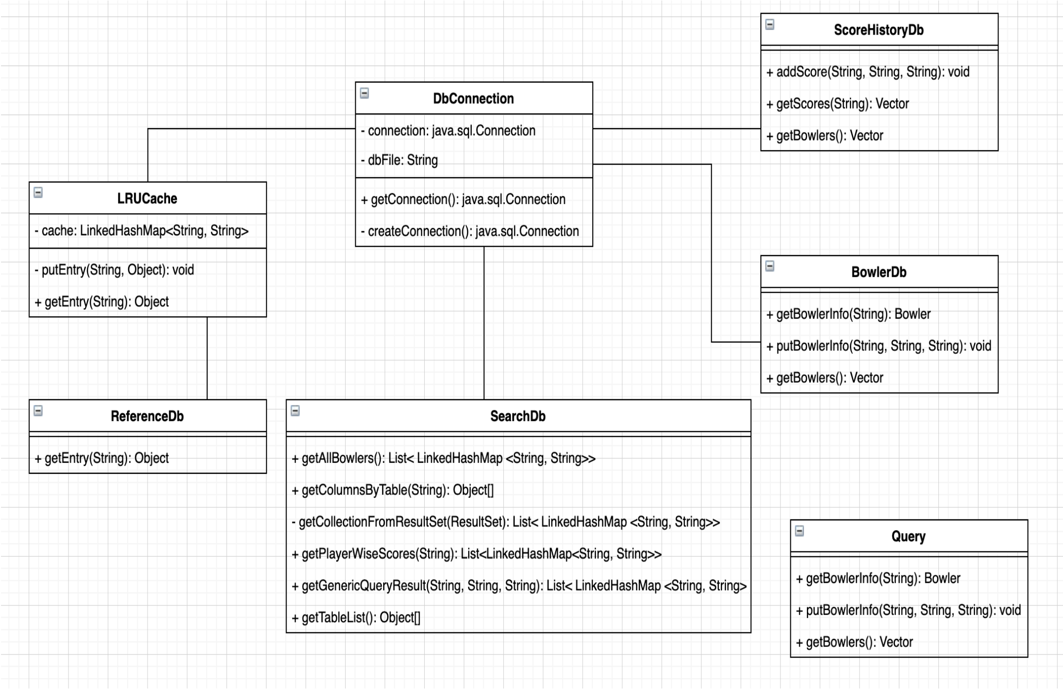
We have integrated a SQLite database to the system which is a file based database engine. This enables seamless execution of queries using standard SQL commands. Currently, the following entities are persisted in the database

- Bowler:
    a. Nick
    b. Full
    c. Email
- Scores
    a. Nick
    b. Date
    c. Score
- User:
    a. userID
    b. Name
    c. password
- Reference (To store the configurable parameters in code)
    a. key
    b. entry

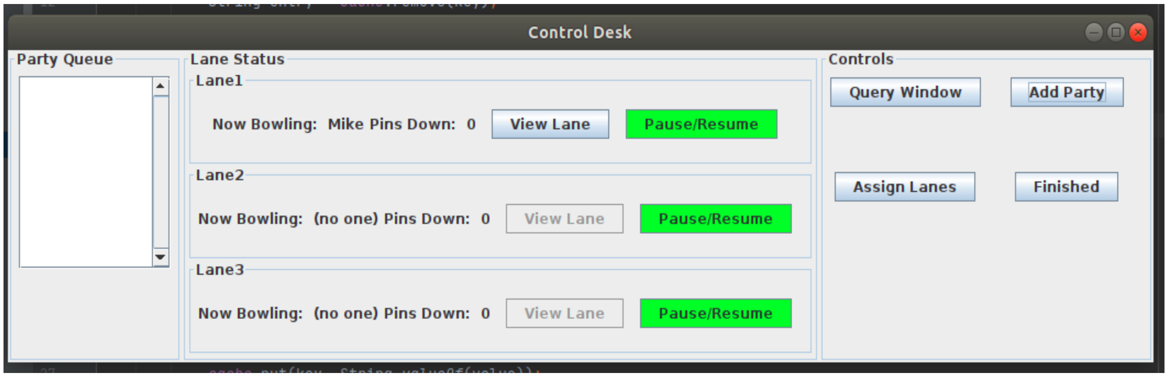
We have also added a LRU cache on top of the database layer for Reference Data. In addition to this, we have a dedicated window for running queries on the entities that are persisted in the database, which also has options to run some predefined common queries:

UML Class diagram:

- **Database connection handling:**


ScoreHistoryDb
+ addScore(String, String, String): void
+ getScores(String): Vector
+ getBowlers(): Vector
DbConnection
- connection: java.sql.Connection
- dbFile: String
+ getConnection(): java.sql.Connection
- createConnection(): java.sql.Connection
LRUCache
- cache: LinkedHashMap<String, String>
- putEntry(String, Object): void
+ getEntry(String): Object
BowlerDb
+ getBowlerInfo(String): Bowler
+ putBowlerInfo(String, String, String): void
+ getBowlers(): Vector
ReferenceDb
+ getEntry(String): Object
SearchDb
+ getAllBowlers(): List< LinkedHashMap <String, String>>
+ getColumnsByTable(String): Object[]
- getCollectionFromResultSet(ResultSet): List< LinkedHashMap <String, String>>
+ getPlayerWiseScores(String): List<LinkedHashMap<String, String>>
+ getGenericQueryResult(String, String, String): List< LinkedHashMap <String, String>
+ getTableList(): Object[]
Query
+ getBowlerInfo(String): Bowler
+ putBowlerInfo(String, String, String): void
+ getBowlers(): Vector

3. Pause and Resume :



Lane1

Now Bowling: Mike Pins Down: 0

Lane2

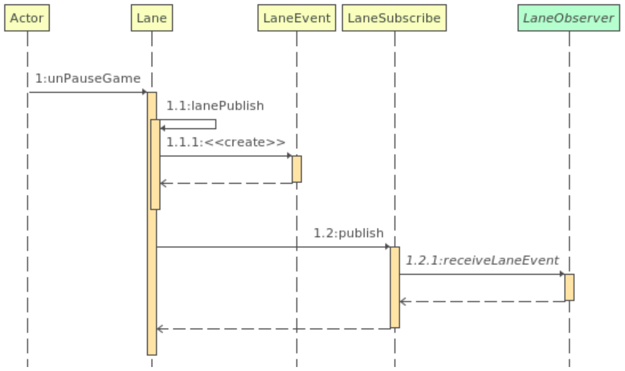
Now Bowling: (no one) Pins Down: 0

Lane3

Now Bowling: (no one) Pins Down: 0

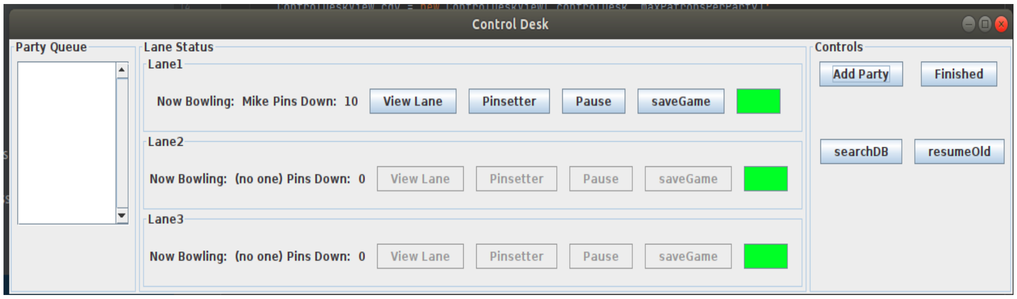
- This feature allows a party to pause their game for later.

- A UI button against each lane has been created that allows a user to pause an existing game in a lane
- The lane will remain frozen for that party until the party resumes their game and the game finishes
- Basically, Pause and Resume Have functionality of maintenance call but with that we can pause the game and store the state of existing game but that’s not part of this part



4. **Saving an Existing Party's Game:**

- Here we have created another UI button that allows the user to save the current status of an ongoing game in a particular lane.



Lane1

Now Bowling: Mike Pins Down: 10

Lane2

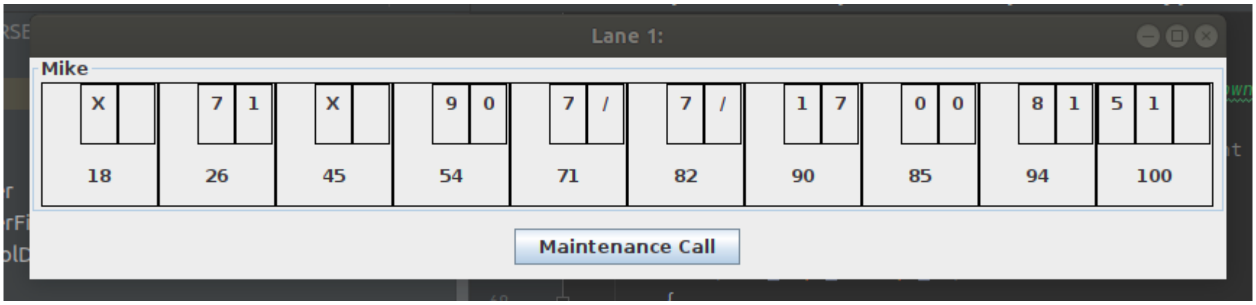
Now Bowling: (no one) Pins Down: 0

Lane3

Now Bowling: (no one) Pins Down: 0

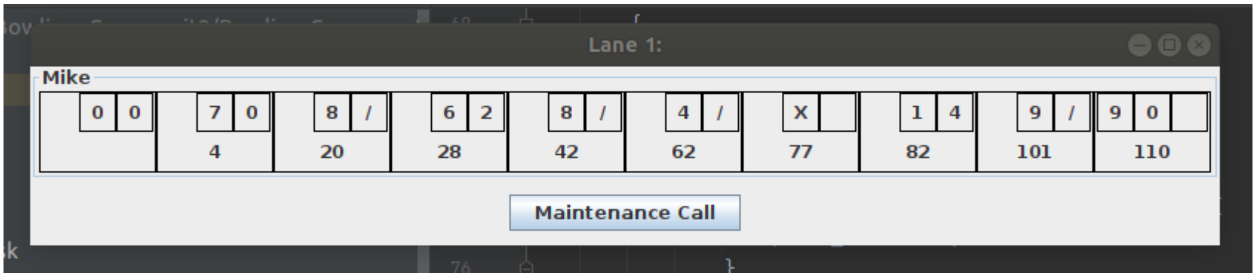
- It is essentially storing a snapshot of the events of that lane into a file.
- Basically, here we generate the file for each game if we click on save game

5. Score calculation With Gutters Penalty



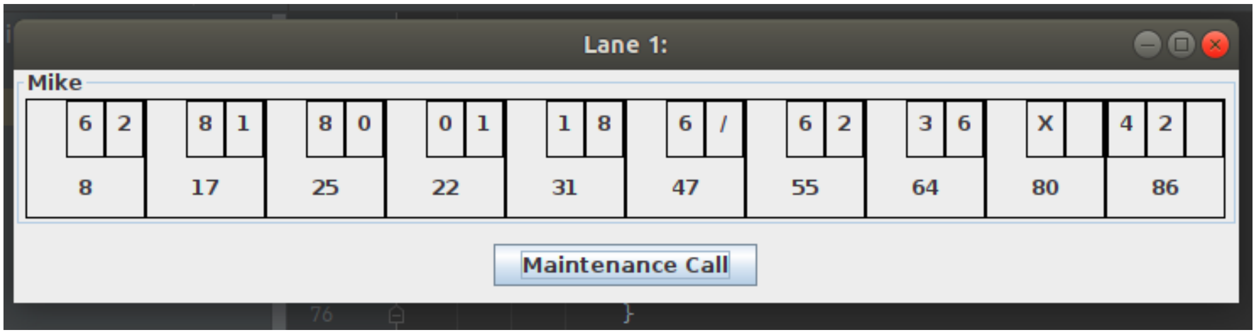
Mike

X	7 1	X	9 0	7 /	7 /	1 7	0 0	8 1	5 1
18	26	45	54	71	82	90	85	94	100



Mike

0 0	7 0	8 /	6 2	8 /	4 /	X	1 4	9 /	9 0
	4	20	28	42	62	77	82	101	110

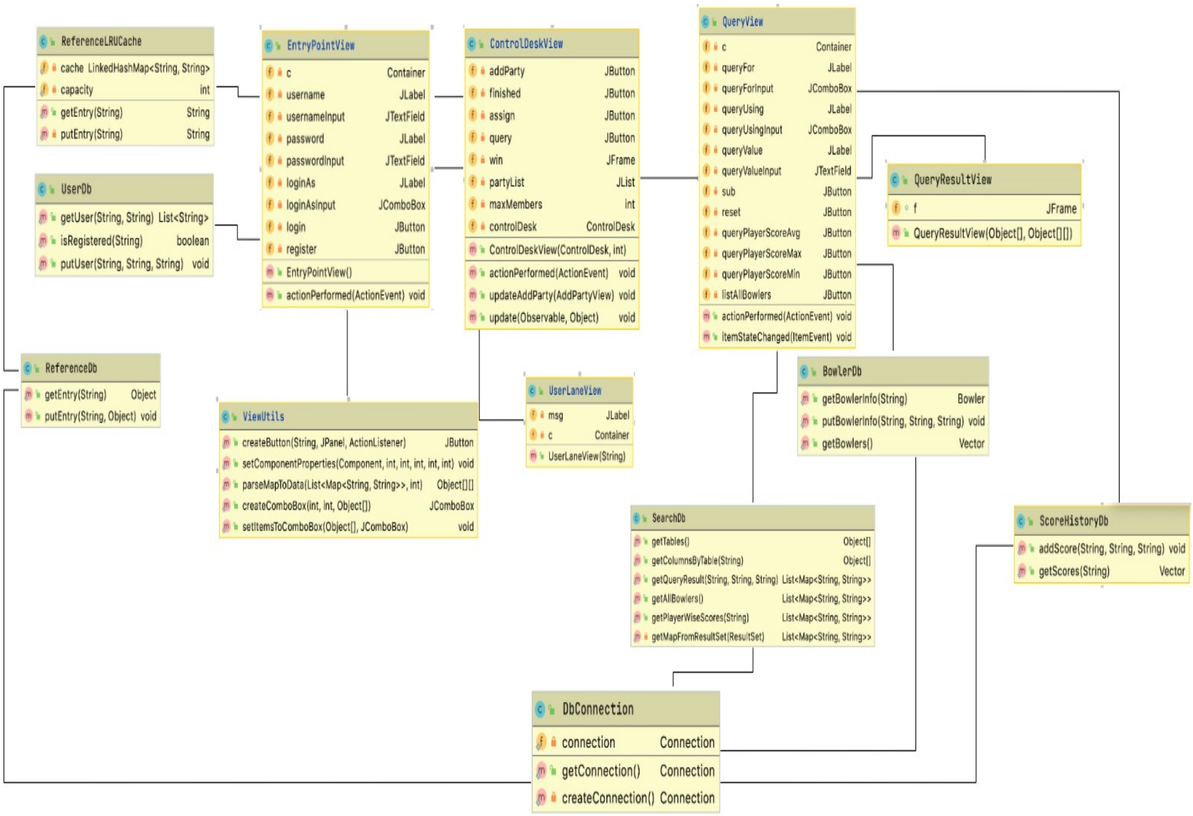


Mike

6 2	8 1	8 0	0 1	1 8	6 /	6 2	3 6	X	4 2
8	17	25	22	31	47	55	64	80	86

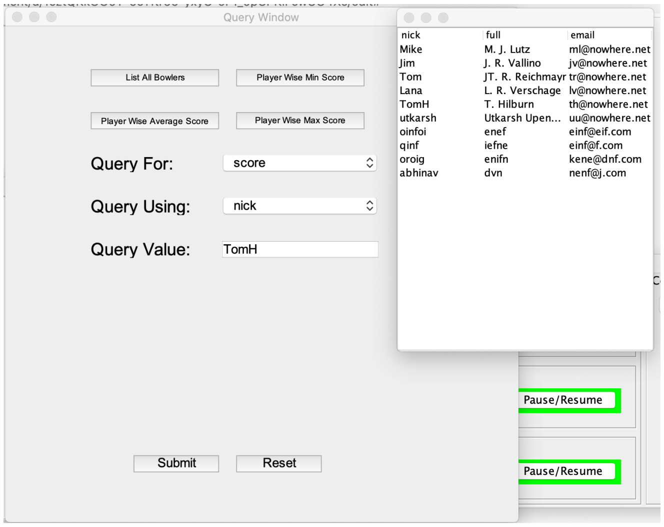
We added a function used to implement a penalty for Gutters – On bowling two consecutive gutters, the player should be penalized 1/2 points of the highest score obtained. If the first 2 consecutive times are at the start of the game, player would be penalized 1/2 of the points that is scored in the next frame

6. Implemented UI for retrieving AVG, MAX and MIN Score and Other Details from Database


ReferenceLRUCache
cache LinkedHashMap<String, String>
capacity int
getEntry(String) String
putEntry(String) String
UserDb
getUser(String, String) List<String>
isRegistered(String) boolean
putUser(String, String, String) void
ReferenceDb
getEntry(String) Object
putEntry(String, Object) void
EntryPointView
c Container
username JLabel
usernameInput JTextField
password JLabel
passwordInput JTextField
loginAs JLabel
loginAsInput JComboBox
login JButton
register JButton
EntryPointView()
actionPerformed(ActionEvent) void
ControlDeskView
addParty JButton
finished JButton
assign JButton
query JButton
win JFrame
partyList JList
maxMembers int
controlDesk ControlDesk
ControlDeskView(ControlDesk, int)
actionPerformed(ActionEvent) void
updateAddParty(AddPartyView) void
update(Observable, Object) void
QueryView
c Container
queryFor JLabel
queryForInput JComboBox
queryUsing JLabel
queryUsingInput JComboBox
queryValue JLabel
queryValueInput JTextField
sub JButton
reset JButton
queryPlayerScoreAvg JButton
queryPlayerScoreMax JButton
queryPlayerScoreMin JButton
listAllBowlers JButton
actionPerformed(ActionEvent) void
itemStateChanged(ItemEvent) void
QueryResultView
f JFrame
QueryResultView(Object[], Object[][])
ViewUtils
createButton(String, JPanel, ActionListener) JButton
setComponentProperties(Component, int, int, int, int, int) void
parseMapToData(List<Map<String, String>>, int) Object[][]
createComboBox(int, int, Object[]) JComboBox
setItemsToComboBox(Object[], JComboBox) void
UserLaneView
msg JLabel
c Container
UserLaneView(String)
BowlerDb
getBowlerInfo(String) Bowler
putBowlerInfo(String, String, String) void
getBowlers() Vector
SearchDb
getTables() Object[]
getColumnsByTable(String) Object[]
getQueryResult(String, String, String) List<Map<String, String>>
getAllBowlers() List<Map<String, String>>
getPlayerWiseScores(String) List<Map<String, String>>
getMapFromResultSet(ResultSet) List<Map<String, String>>
ScoreHistoryDb
addScore(String, String, String) void
getScores(String) Vector
DbConnection
connection Connection
getConnection() Connection
createConnection() Connection

We have added a query window to enable users to query data from our persistent storage. At this moment, this window provides the users to run the following queries:

1. **List All Bowlers:**



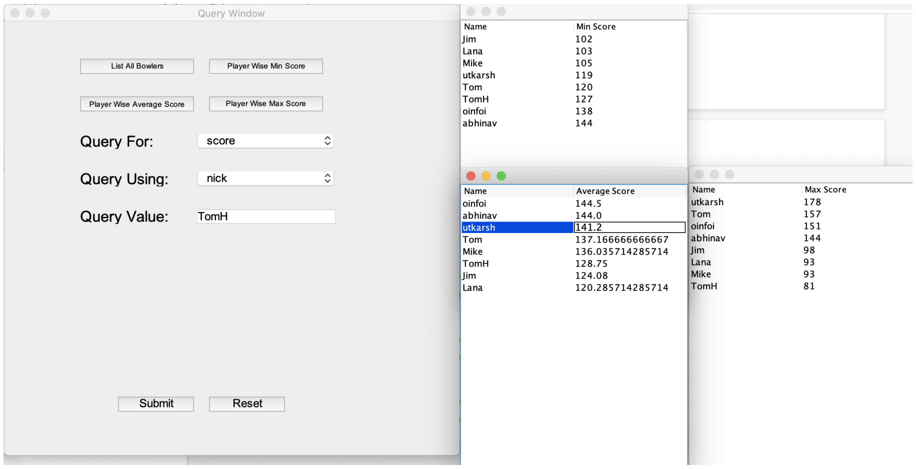
Query For: score

Query Using: nick

Query Value: TomH

nick	full	email
Mike	M. J. Lutz	ml@nowhere.net
Jim	J. R. Vallino	jv@nowhere.net
Tom	JT. R. Reichmayr	tr@nowhere.net
Lana	L. R. Verschage	lv@nowhere.net
TomH	T. Hilburn	th@nowhere.net
utkarsh	Utkarsh Upen...	uu@nowhere.net
oinfoi	enef	einf@eif.com
qinf	iefne	einf@f.com
oroig	enifn	kene@dnf.com
abhinav	dvn	nenf@j.com

2. **Player Wise Max/Min/Avg Scores**



Query For: score

Query Using: nick

Query Value: TomH

Name	Min Score
Jim	102
Lana	103
Mike	105
utkarsh	119
Tom	120
TomH	127
oinfoi	138
abhinav	144

Name	Average Score
oinfoi	144.5
abhinav	144.0
utkarsh	141.2
Tom	137.166666666667
Mike	136.035714285714
TomH	128.75
Jim	124.08
Lana	120.285714285714

Name	Max Score
utkarsh	178
Tom	157
oinfoi	151
abhinav	144
Jim	98
Lana	93
Mike	93
TomH	81

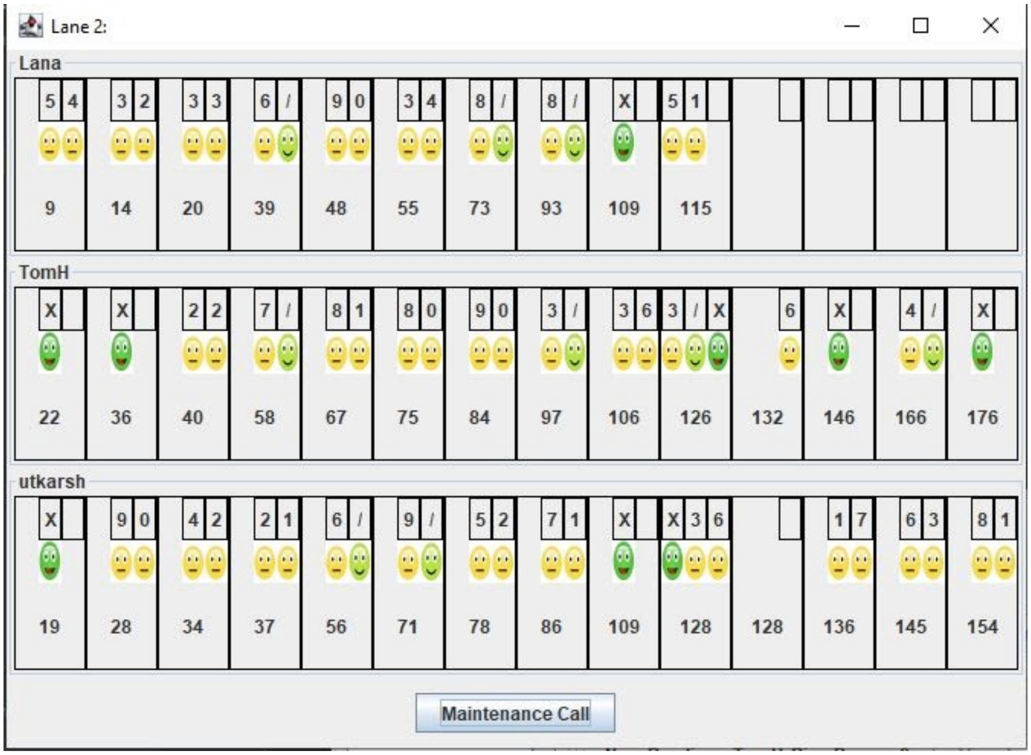
3. **Ad-hoc queries by specifying table name, filter column name and filter value:**

   Example - In the screenshot above, the fields chosen are “score”, “nick” and “TomH”. On clicking submit, it would run the following query on the database and return the results:

   “select * from score where nick=`TomH`”

7. Tie Breaker Round with Extra throw:

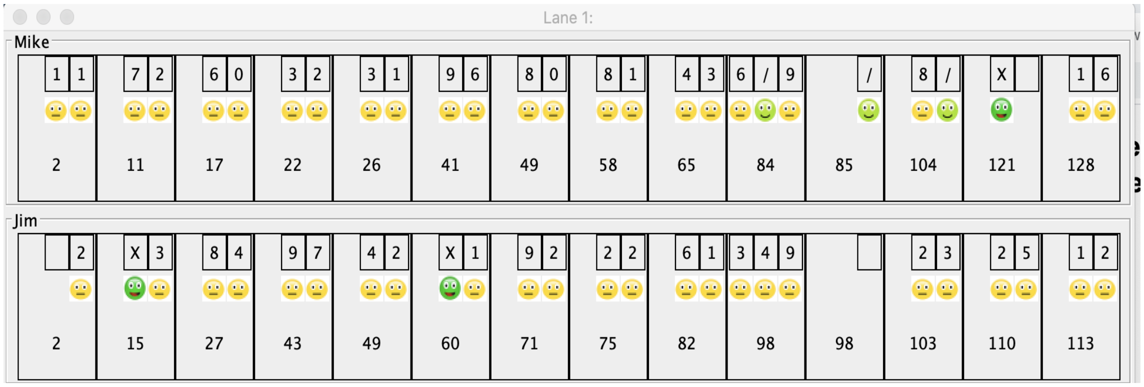
After throws in 10 frame we calculate two players who scored maximum score and a extra chance is provided two second highest player



Lana	5 4	3 2	3 3	6 /	9 0	3 4	8 /	8 /	X	5 1				
	9	14	20	39	48	55	73	93	109	115				
TomH	X	X	2 2	7 /	8 1	8 0	9 0	3 /	3 6	3 / X	6	X	4 /	X
	22	36	40	58	67	75	84	97	106	126	132	146	166	176
utkarsh	X	9 0	4 2	2 1	6 /	9 /	5 2	7 1	X	X 3 6		1 7	6 3	8 1
	19	28	34	37	56	71	78	86	109	128	128	136	145	154

8. Implemented emoticon for each action:

For every throw, based on the score, we have set up a mechanism to display emoticons. The following screenshot represents the same:

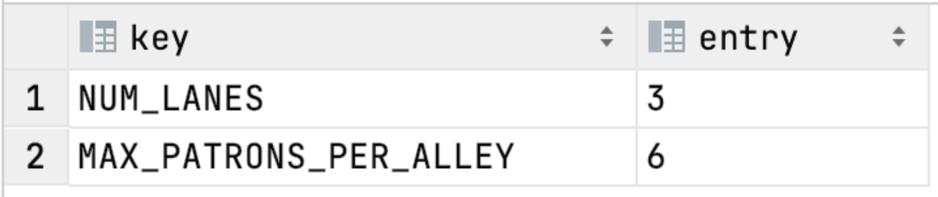


Mike	1 1	7 2	6 0	3 2	3 1	9 6	8 0	8 1	4 3	6 / 9	/	8 /	X	1 6
	2	11	17	22	26	41	49	58	65	84	85	104	121	128
Jim	2	X 3	8 4	9 7	4 2	X 1	9 2	2 2	6 1	3 4 9		2 3	2 5	1 2
	2	15	27	43	49	60	71	75	82	98	98	103	110	113

9. **All the numbers quoted and existing in the code:**

   We have added the mechanism to store configuration in the database, and add a layer of LRU cache on top of it. For this, we created a table called Reference with 2 columns namely “key” and “entry”.

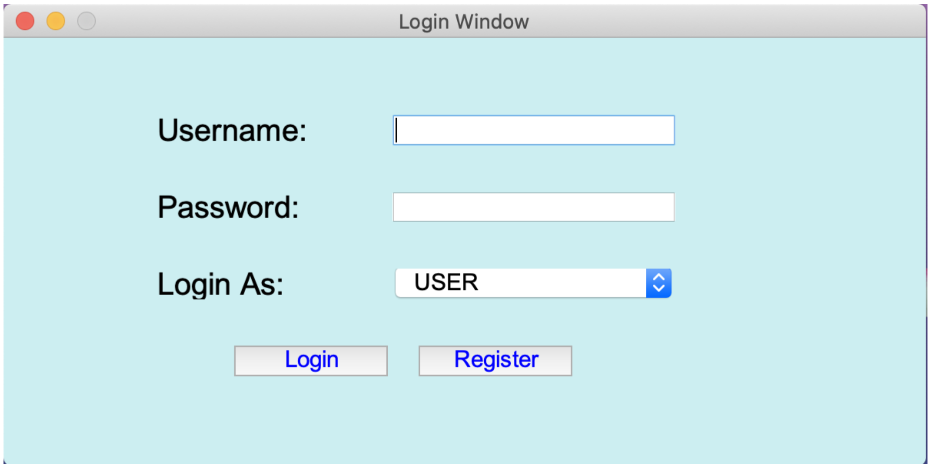
The following is a snapshot of the reference table.



	key	entry
1	NUM_LANES	3
2	MAX_PATRONS_PER_ALLEY	6

10. **Added Authentication for Admin and User:**

    We have leveraged the integration of the database layer to implement User Authentication. The user will have to login to the system before he/she can see the Control Desk. The following is the login screen that the user sees on application startup:



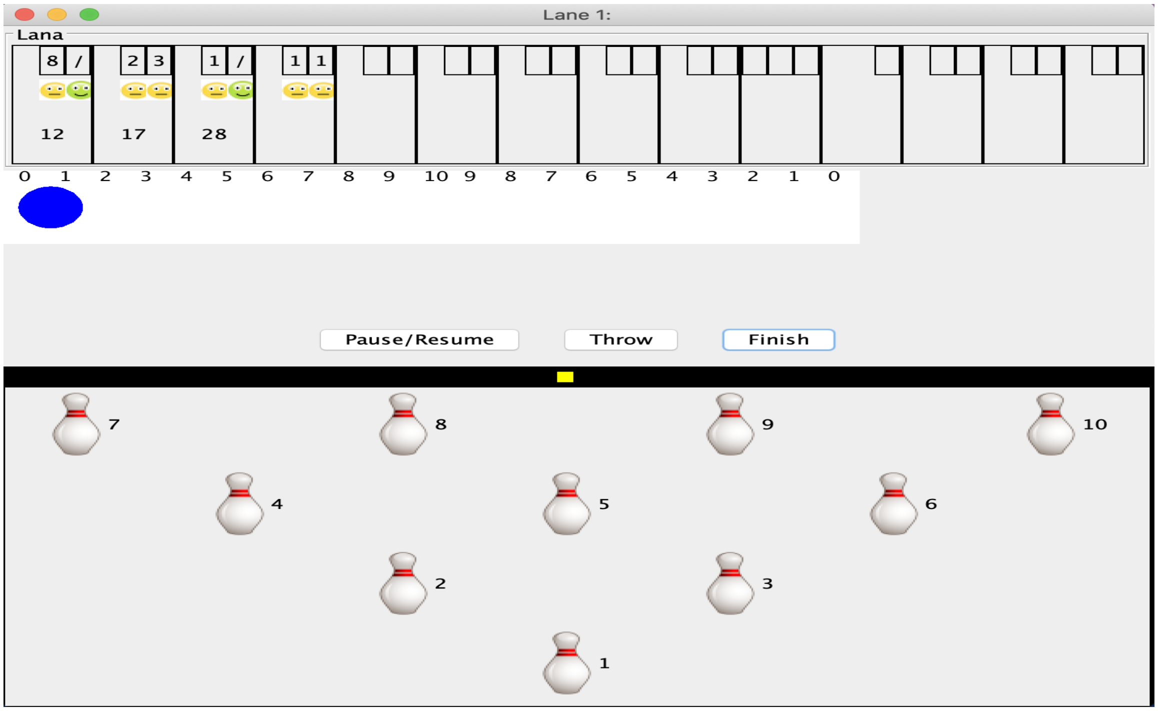
Username:

Password:

Login As: USER

Here, the user can either login or register as a new user. Login would take the user to control desk, whereas Register would create a new user entity and ask the user to re-login:
The following is a view of the user details table that's being used for login.

11. Added UI to make game user dependent by removing simulation Engine:

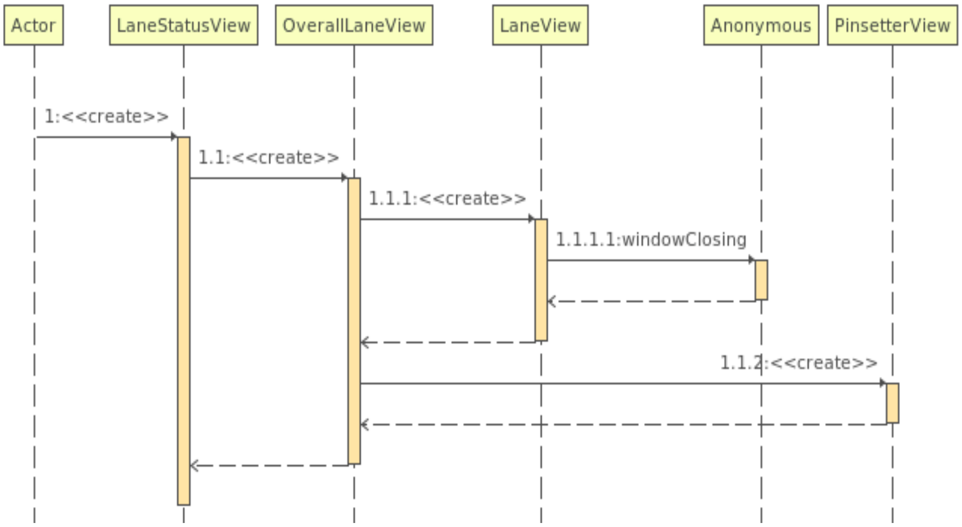


In above snapshot as we can see that there are two views which are combine as single view (Pinsetter and LaneStatus View) so as to make the UI more intuitive to the user.

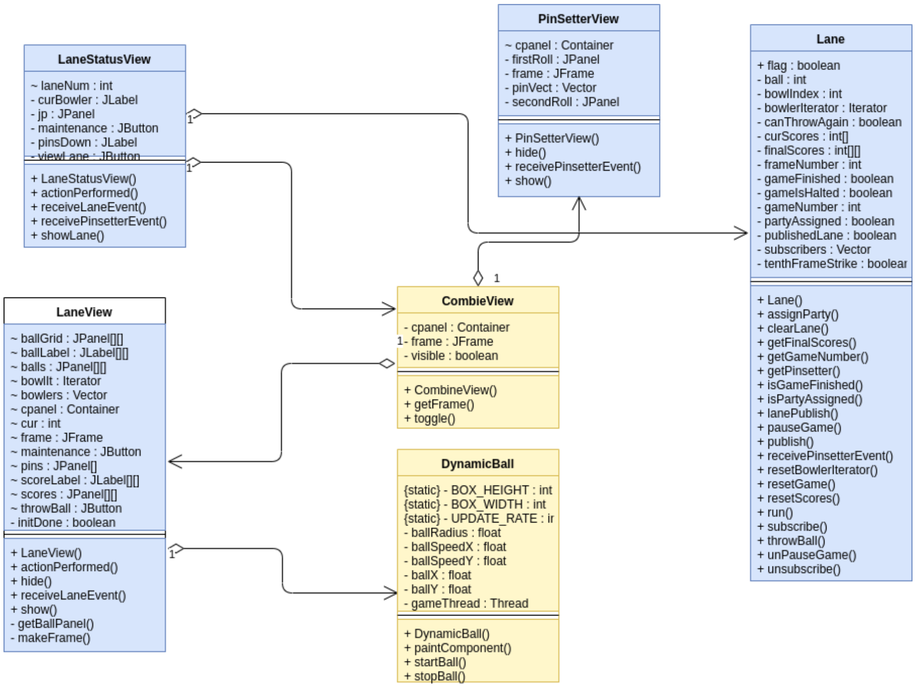
Throw button every time selects point from the scale and depending upon scale at which user throws ball we generate number of pins fallen

In earlier design to check which pins are actually falling we have to click on button pinsetter in lanestatus view but that's unnecessary and makes GUI complex every player will want to visualise things when it is playing

Another problem in earlier design was it is much randomized as every throw is generated randomly. But here ability to make a throw is given to the user.

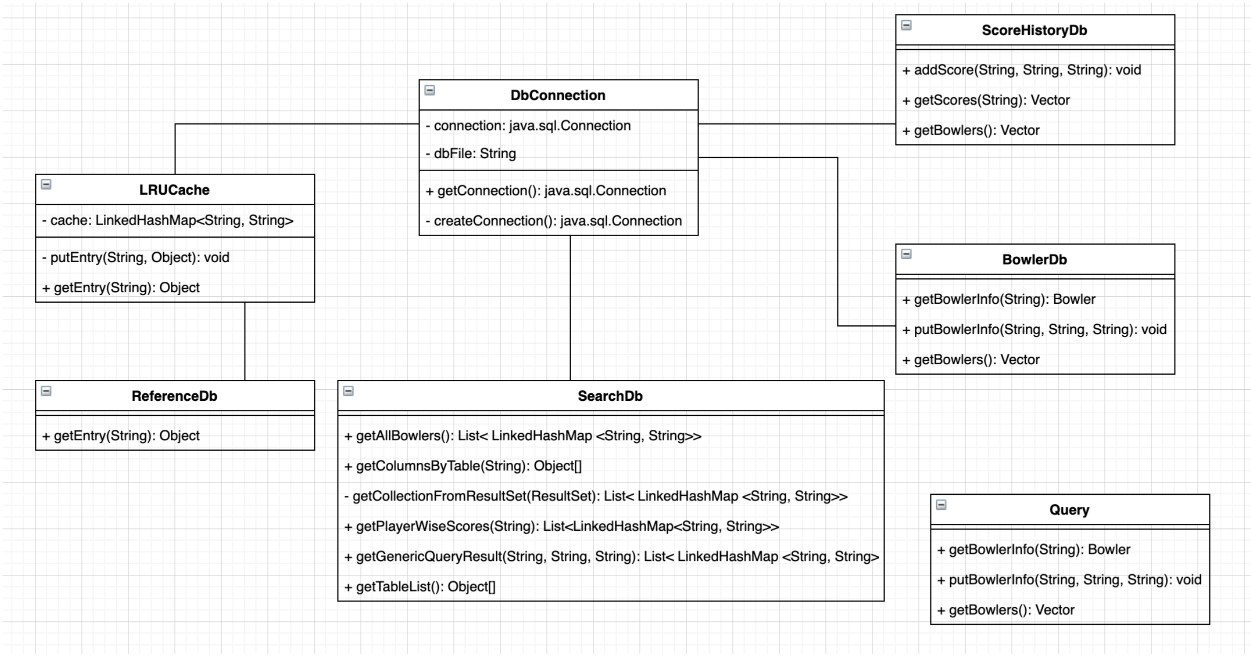


- Above sequence diagram shows flow and connections of lanestatusview
- Also to implement above functionality we have to introduce to new classes called as CombineView and DynamicBall
- Below UML diagram is created can show you relation of new classes with old one



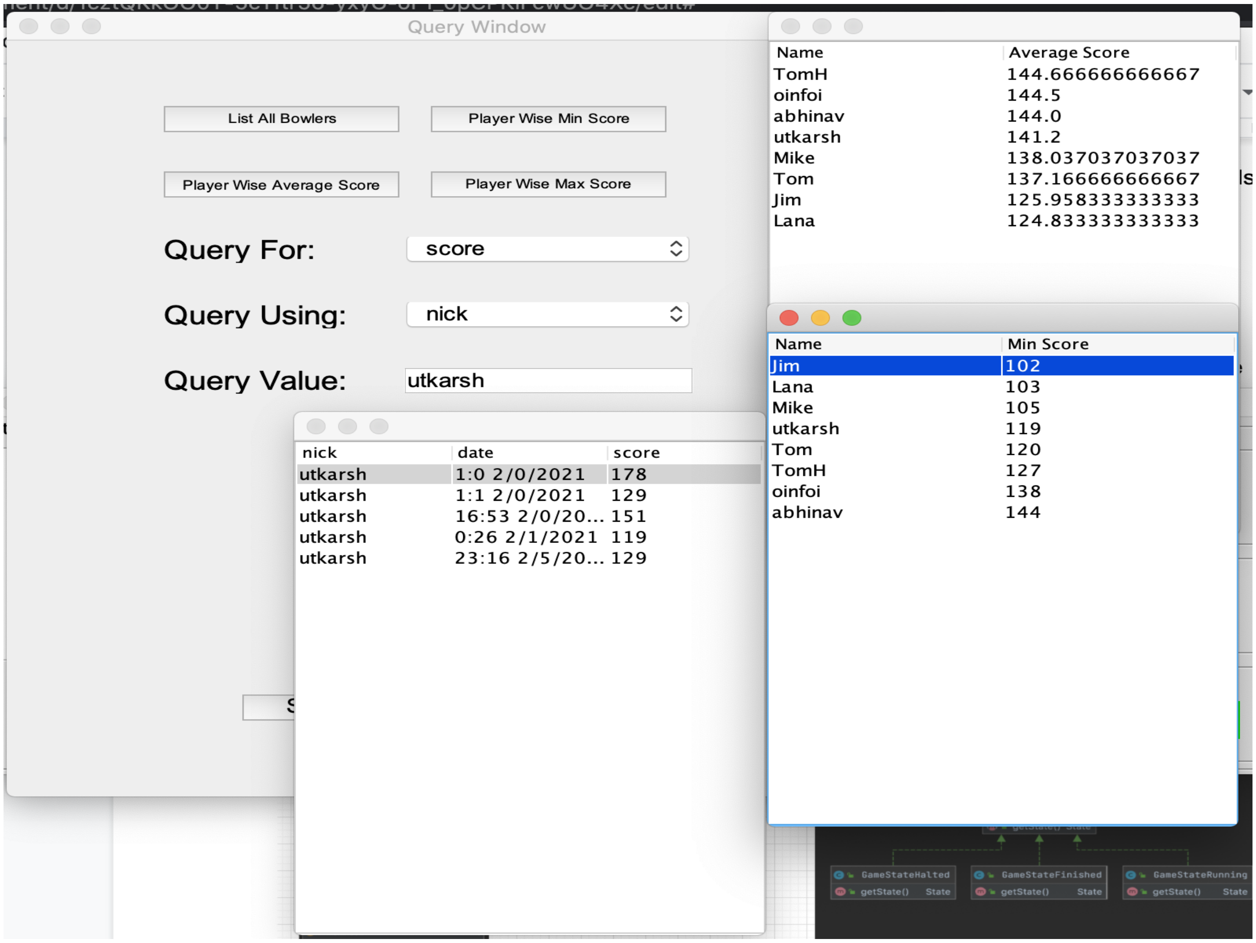
Patterns Used in Design:

1. **Facade Pattern for Database Connection Management** - Database Connection Code/Logic is completely abstracted from the other classes. There is only a single method called 'getConnection()' that returns a Connection object that the classes can use to make database queries.


ScoreHistoryDb
+ addScore(String, String, String): void
+ getScores(String): Vector
+ getBowlers(): Vector
DbConnection
- connection: java.sql.Connection
- dbFile: String
+ getConnection(): java.sql.Connection
- createConnection(): java.sql.Connection
LRUCache
- cache: LinkedHashMap<String, String>
- putEntry(String, Object): void
+ getEntry(String): Object
BowlerDb
+ getBowlerInfo(String): Bowler
+ putBowlerInfo(String, String, String): void
+ getBowlers(): Vector
ReferenceDb
+ getEntry(String): Object
SearchDb
+ getAllBowlers(): List< LinkedHashMap <String, String>>
+ getColumnsByTable(String): Object[]
- getCollectionFromResultSet(ResultSet): List< LinkedHashMap <String, String>>
+ getPlayerWiseScores(String): List<LinkedHashMap<String, String>>
+ getGenericQueryResult(String, String, String): List< LinkedHashMap <String, String>
+ getTableList(): Object[]
Query
+ getBowlerInfo(String): Bowler
+ putBowlerInfo(String, String, String): void
+ getBowlers(): Vector

2. **Singleton Pattern for Database Connection -** The database connection object handling class (DbConnection) has been created to provide a singleton Connection object i.e. there would be at all times, only one connection to the database.

3. **Table Filter UI pattern** has been used to display query results from the database.



Query For: score

Query Using: nick

Query Value: utkarsh

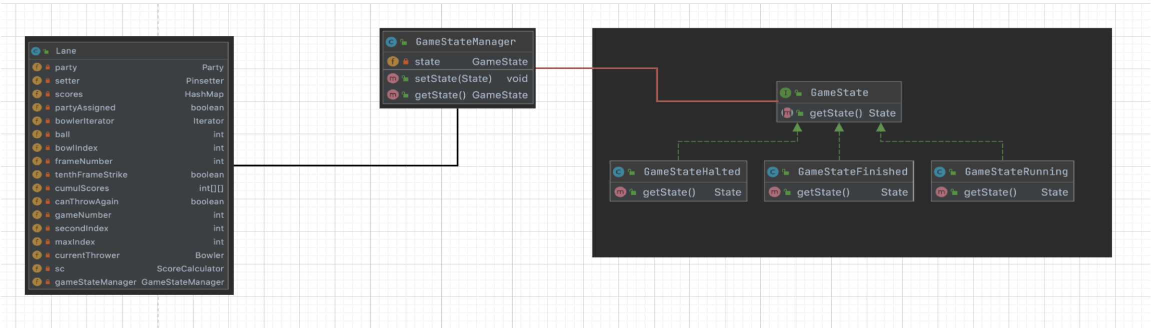
nick	date	score
utkarsh	1:0 2/0/2021	178
utkarsh	1:1 2/0/2021	129
utkarsh	16:53 2/0/20...	151
utkarsh	0:26 2/1/2021	119
utkarsh	23:16 2/5/20...	129

Name	Average Score
TomH	144.666666666667
oinfoi	144.5
abhinav	144.0
utkarsh	141.2
Mike	138.037037037037
Tom	137.166666666667
Jim	125.958333333333
Lana	124.833333333333

Name	Min Score
Jim	102
Lana	103
Mike	105
utkarsh	119
Tom	120
TomH	127
oinfoi	138
abhinav	144

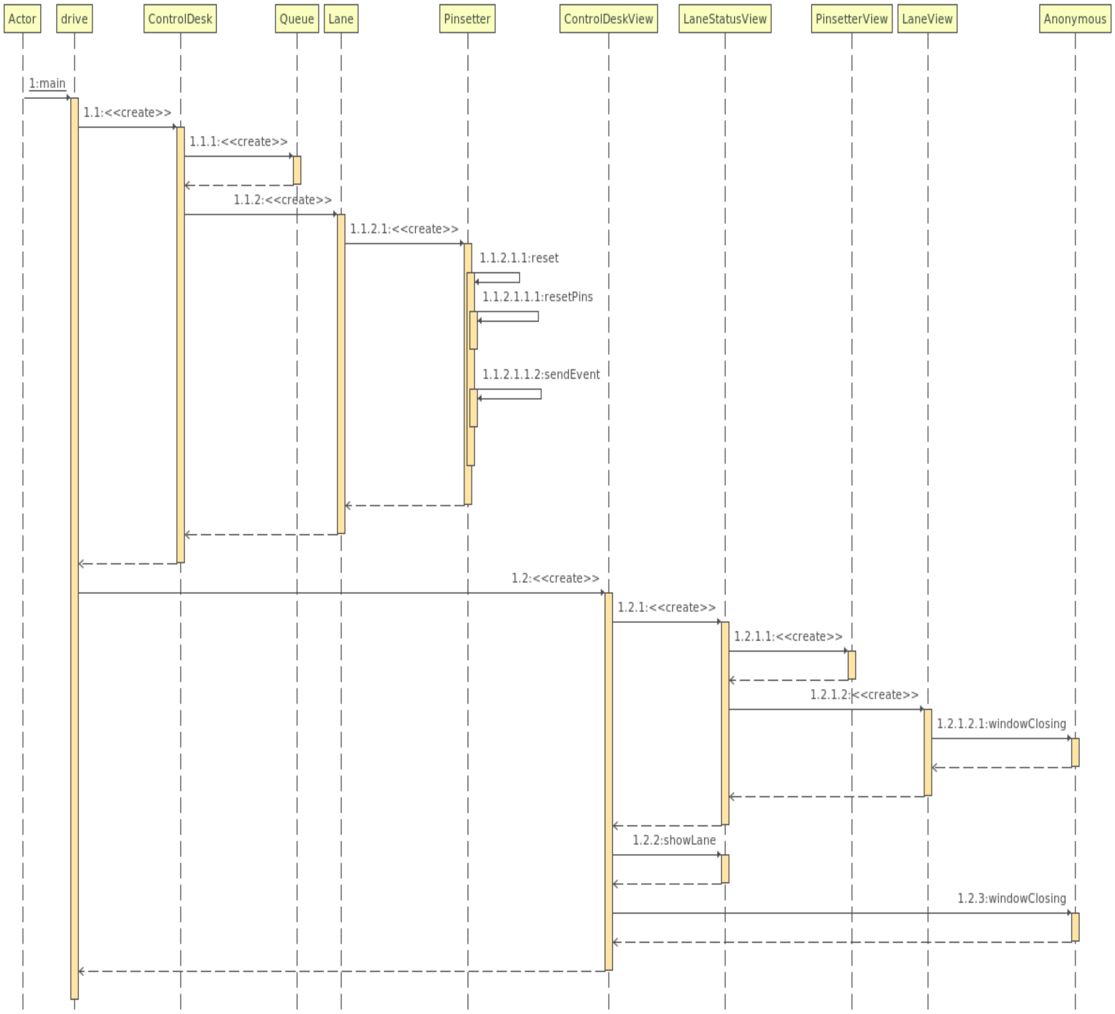
4. **Observer Pattern** has been used in the View Layer to subscribe to the events from the Models.

5. **State** Pattern is used to manage the state of a Game. The state can have the following values:
    a. RUNNING
    b. HALTED
    c. FINISHED

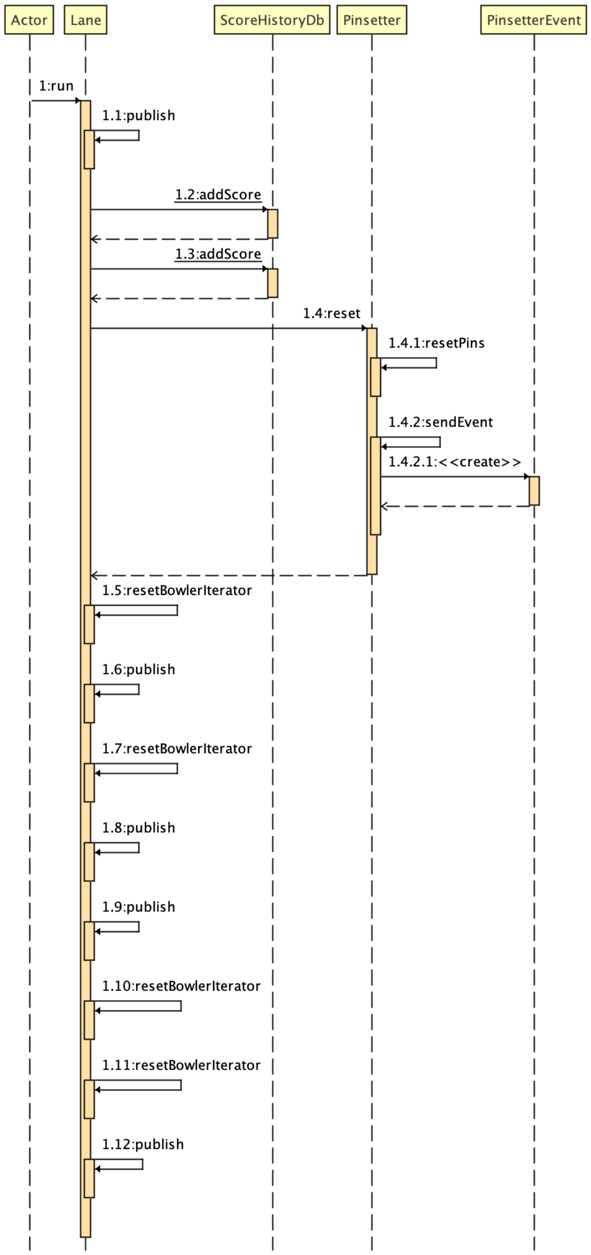


UML Sequence Diagram

Drive Class



Lane Class

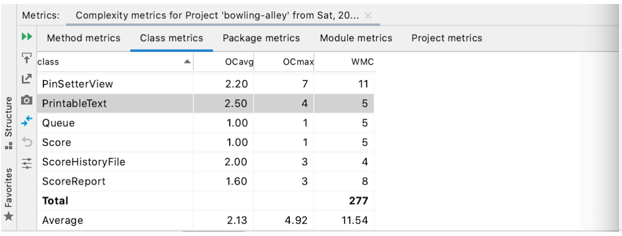

Actor
Lane
ScoreHistoryDb
Pinsetter
PinsetterEvent
1:run
1.1:publish
1.2:addScore
1.3:addScore
1.4:reset
1.4.1:resetPins
1.4.2:sendEvent
1.4.2.1:<<create>>
1.5:resetBowlerIterator
1.6:publish
1.7:resetBowlerIterator
1.8:publish
1.9:publish
1.10:resetBowlerIterator
1.11:resetBowlerIterator
1.12:publish

Strength Of Current Design

- We improved design by making it more attractive
- We implemented various pattern to improve our design
- Mainly initial design was much dependent on software we make it more user dependent

Metrics Analysis

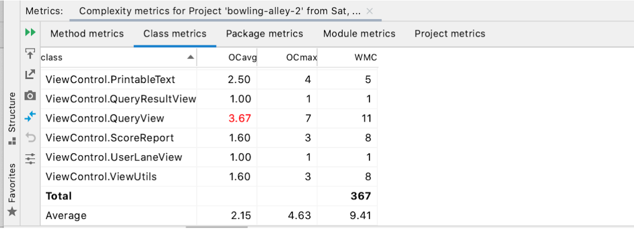
The following screenshot compares complexity metrics between the code in its initial state and the code in its latest state. As is evident, the Average **Weighted Method Complexity in the latest state of the code is 9.41 which is lower in comparison to that of its initial state i.e. 11.54**



Metrics: Complexity metrics for Project 'bowling-alley' from Sat, 20...

Method metrics | Class metrics | Package metrics | Module metrics | Project metrics

class	OCavg	OCmax	WMC
PinSetterView	2.20	7	11
PrintableText	2.50	4	5
Queue	1.00	1	5
Score	1.00	1	5
ScoreHistoryFile	2.00	3	4
ScoreReport	1.60	3	8
Total			**277**
Average	2.13	4.92	11.54

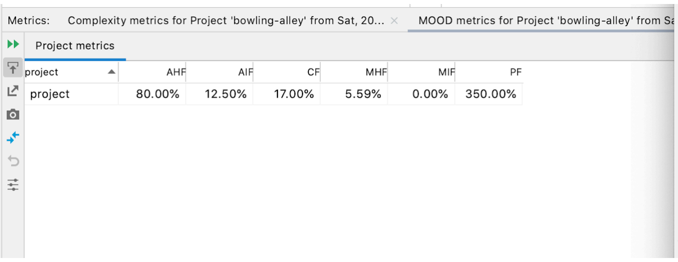


Metrics: Complexity metrics for Project 'bowling-alley-2' from Sat, ...

Method metrics | Class metrics | Package metrics | Module metrics | Project metrics

class	OCavg	OCmax	WMC
ViewControl.PrintableText	2.50	4	5
ViewControl.QueryResultView	1.00	1	1
ViewControl.QueryView	3.67	7	11
ViewControl.ScoreReport	1.60	3	8
ViewControl.UserLaneView	1.00	1	1
ViewControl.ViewUtils	1.60	3	8
Total			**367**
Average	2.15	4.63	9.41

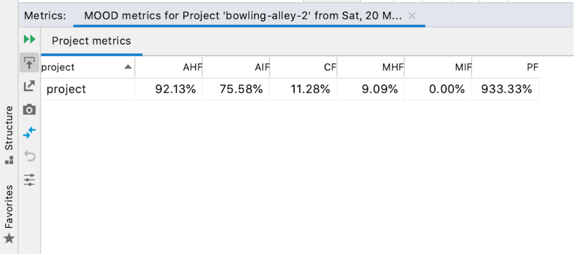
We also ran MOOD metrics on the initial and final codebases, using the [Metrics Reloaded](#) plugin, and got the following metrics:



Metrics: Complexity metrics for Project 'bowling-alley' from Sat, 20... | MOOD metrics for Project 'bowling-alley' from Sa

Project metrics

project	AHF	AIF	CF	MHF	MIF	PF
project	80.00%	12.50%	17.00%	5.59%	0.00%	350.00%



Metrics: MOOD metrics for Project 'bowling-alley-2' from Sat, 20 M...

Project metrics

project	AHF	AIF	CF	MHF	MIF	PF
project	92.13%	75.58%	11.28%	9.09%	0.00%	933.33%

Analysis of MOOD metrics:

The MOOD metrics, defined by Fernando Brito e Abreu, are designed to provide a summary of the overall quality of an object-oriented project. The original MOOD metrics suite consists of 6 metrics. Described below are 3 of the 6 which we found relevant to our system.

1. **MHF -** Method Hiding Factor determines how abstracted the methods are across classes. Because of the refactoring, **the MHF metric went up from 5.59% to 9.09%.**

   A generally acceptable value for MHF lies between 8-25%*

2. **AHF -** Attribute Hiding Factor, like Method Hiding Factor represents the level of abstraction over attributes across classes. The ideal value of AHF is 100%* which represents that all the attributes across a project are private. As we can see, after refactoring and enhancements, **AHF of the new code went up from 80% to 92%.**

3. **CF -** Coupling Factor (CF alias COF) measures the actual couplings among classes in relation to the maximum number of possible coupling. It has been generally accepted that CF should not be greater than 12%*. **As is evident from the metrics above, the CF of the refactored code is lower and in control, compared to that of the initial code.**

***Source -** **https://www.aivosto.com/project/help/pm-oo-mood.html**