Unit 2 (Enhancement) Bowling Alley Simulation- Design Document

Team 13

Title: Bowling Management Simulation.

Date of Submission: March 22nd, 2022.

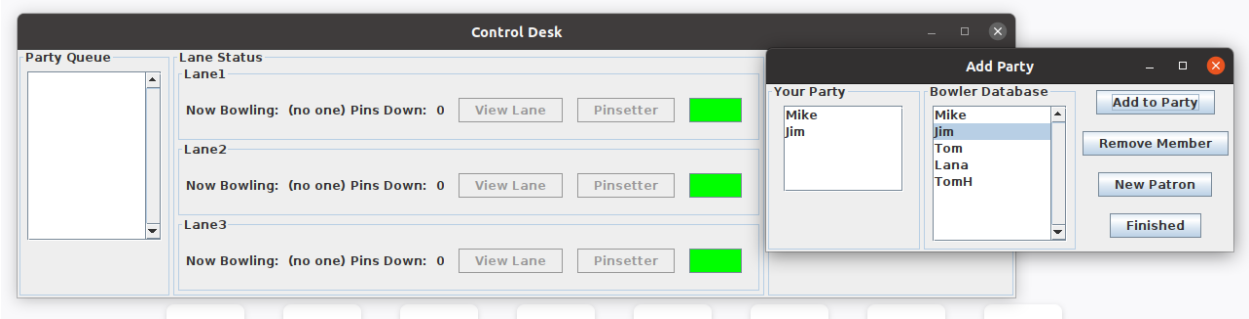
Github Repository:
https://gitlab.com/debashish.roy1998/bowling-alley-enhancement
Earlier we were maintaining things on Github but as per the instructions we have moved the project to GitLab.

Team Member	Work of hour	Contribution
Debashish Roy (2021201034)	35	1) Performed the metric analysis on the updated and previous code. 2) Added extra functionality of tie-breaker, between highest and second-highest user. 3) Added Penalty feature, for the gutters. 4) Generated complete UML for all the classes (before and after adding features). 5) Overview of the classes. 6) Designed patterns for some features and customized some input parameters at run time.
Nikita Rawat (2021201035)	35	1) Added database layer to the code 2) Added functionality for performing ad-hoc queries 3) Made the code extensible and working for multiplayer.
Janmejay Pratap Singh Baghel (2020201089)	35	1)Made a new Smiley Class 2)Added functionality for Smileys 3)UML and Sequence diagrams for some classes in the refactored code
Vinaya Sai Revanth Bachu (2020201090)	35	1) Added the functionality for providing interaction 2) Overview of some classes 3) Sequence diagrams for some classes

OVERVIEW

The bowling alley management system is a game developed in Java that simulates a bowling alley. From the first improvement, we have added some more features. But for the sake of completeness, we are listing all features.

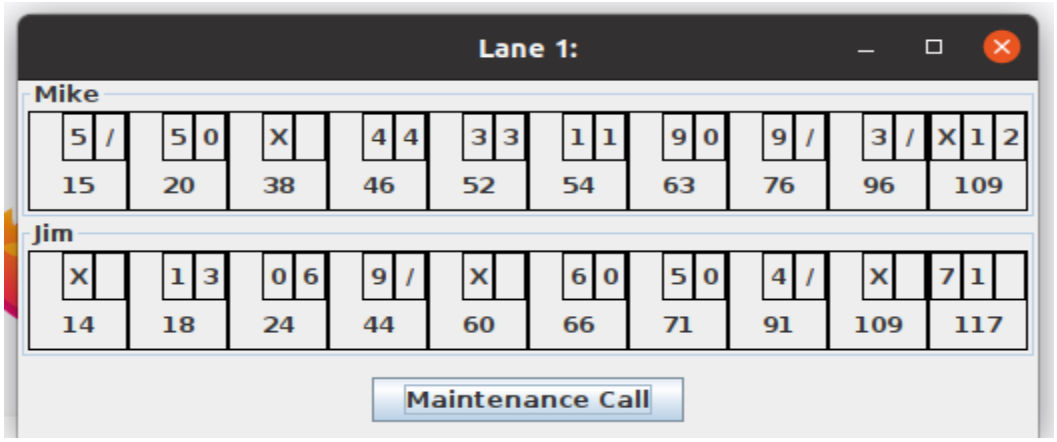
1. **Control Desk** - The control desk is used to control the entire game. It has functionalities like add a party, view pinsetter, view scores, add new patron, receive maintenance call, finish game, etc. After a player has finished a game the control desk is notified of the same and the controller has an option to allow another game for the player or end it.



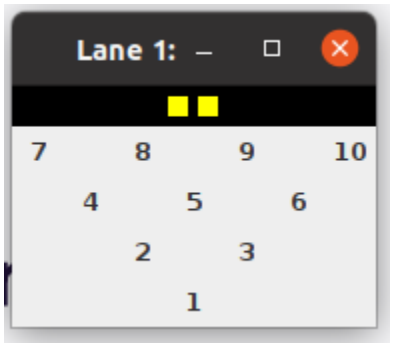
2. **New Lane -** As per the new functionality second-highest player will be given a chance to throw an extra ball, and if the new score will be greater than or equal to the first, then there will be a tie and they will compete. For this, a New Lane has to be created. This class is responsible for creating the new lane.

3. **Add new player -** This interface is used to do a one-time registration for the new bowler. It requires details like the full name, nickname, and email id of the player.

4. **Lane** - Each lane can accommodate one to six bowlers. The bowlers who check-in as a group are assigned the same lane. Implemented the logic of giving an extra throw for the second-highest user.



5. **Pinsetter -** The pinsetter can detect which pins were knocked down after each throw and send the score to the scoring station.



6. **Scoring Station** - The scoring station interface shows a graphical representation of the scores of the bowlers in a lane (in the order in which they checked in). Also, we have implemented the penalty systems.

7. **Maintenance Call** - In case of a mechanical problem in any lane (eg: pinsetter did not re-rack, etc), the bowler can contact the control desk through an interface on the scoring station. The control station sends an acknowledgment of the request back to the station.

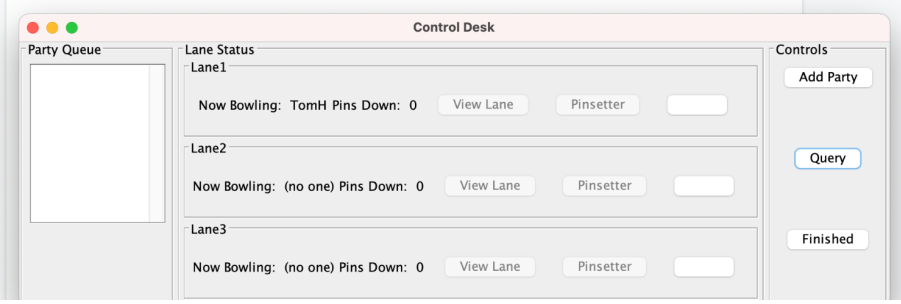
8. **Get bowler history -** At the end of a game, the bowling history of the player is automatically emailed to him. He may also choose to get a printout of the same. *We have used SQLite here instead of the normal file system, such that it can be extended further and also from the security point of view.*

9. **ThrowWindow -** This class object is created and shown whenever the user is about to bowl. The object takes a response from the user and then the bowl is bowled in the alley.

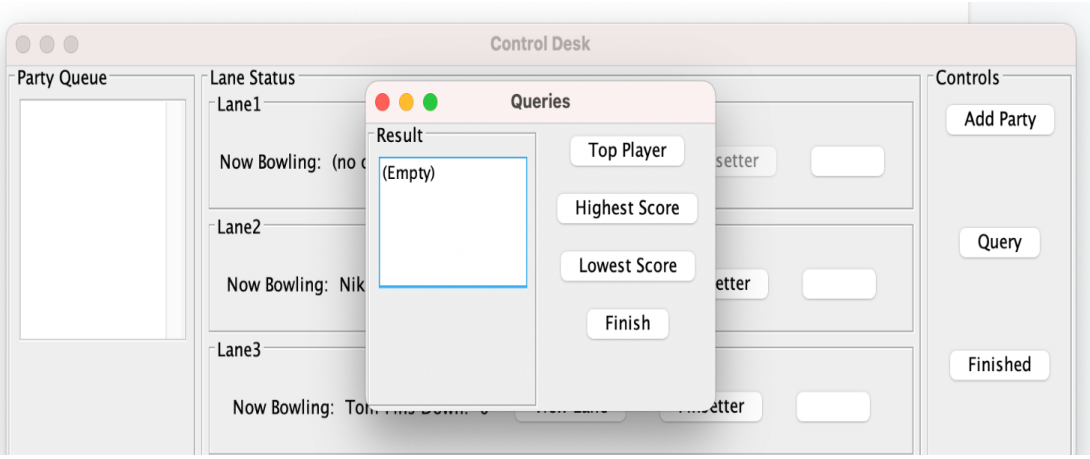
10. **Smiley**- This class object is created by the Pinsetter class. The object takes id as input and shows the corresponding smiley depending upon the score of the player.

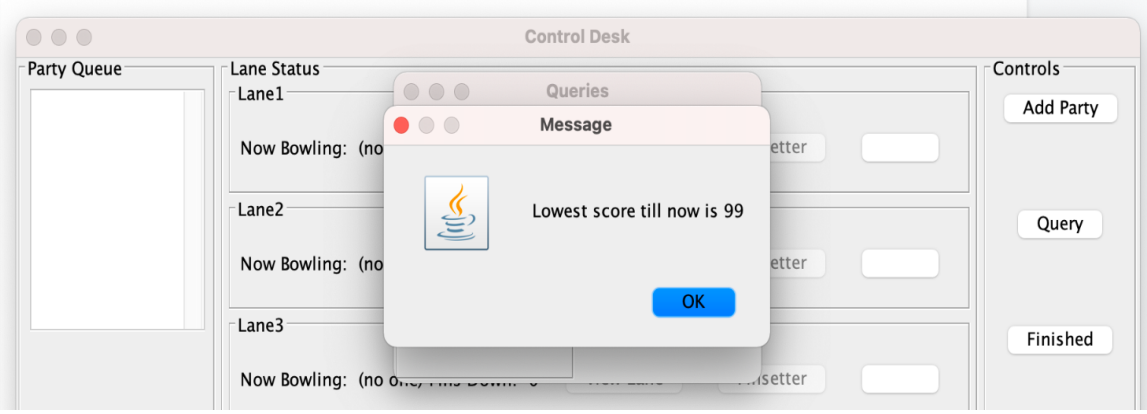
New Features:

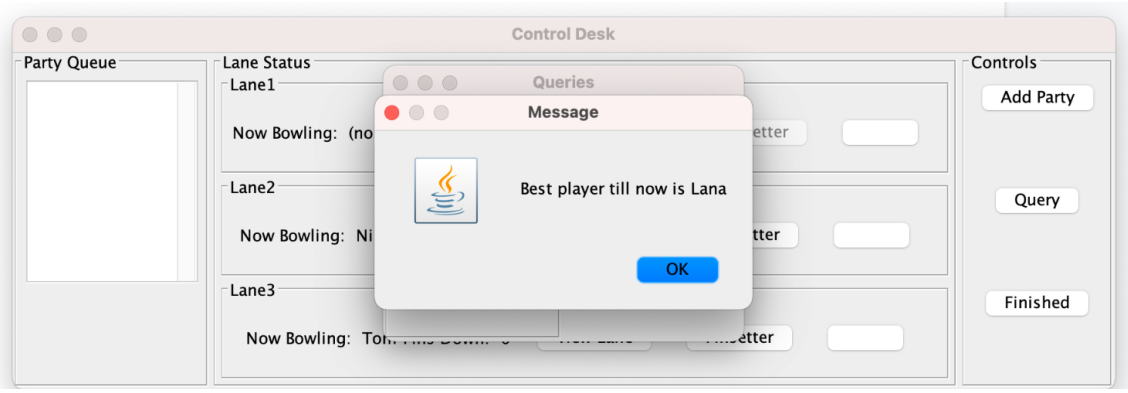
- The information about player names and scores is now stored in a database so that it is persistent. For this, we have used the **SQLite database**. The database name is score DB and it has a table named playerScore that contains three columns namely, playerName, score, and date. All values are stored as strings.



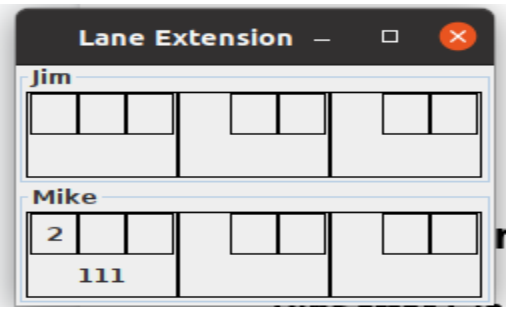
- Another feature that has been added is the ability to make queries through an interface. There are three options available for the user: find the **highest score, lowest score, and find the name of the best player**. This can be accessed by clicking on the Query button on the control desk.





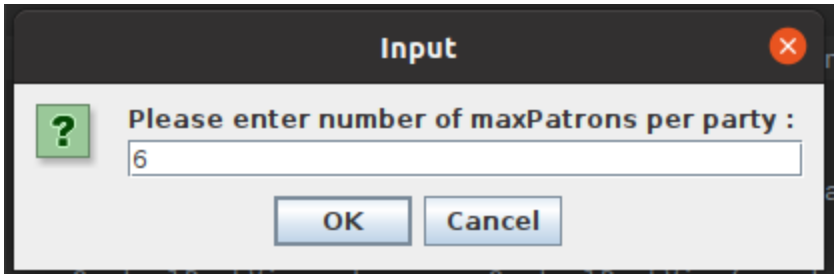


- We have added the tiebreaker. So the second-highest player get a chance to throw one more bowl extra and if the score after the throw becomes equal or greater than the first then it will lead to the tiebreaker.

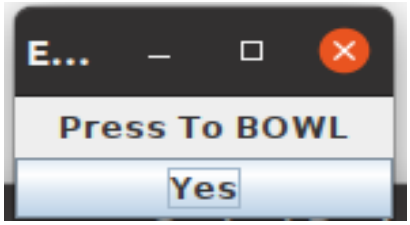


- Added the feature of extra penalty if there are two consecutive gutters. The penalty will be half of the maximum point achieved in one throw.

- We have customized the number of max patrons per party which can be changed at compile time.

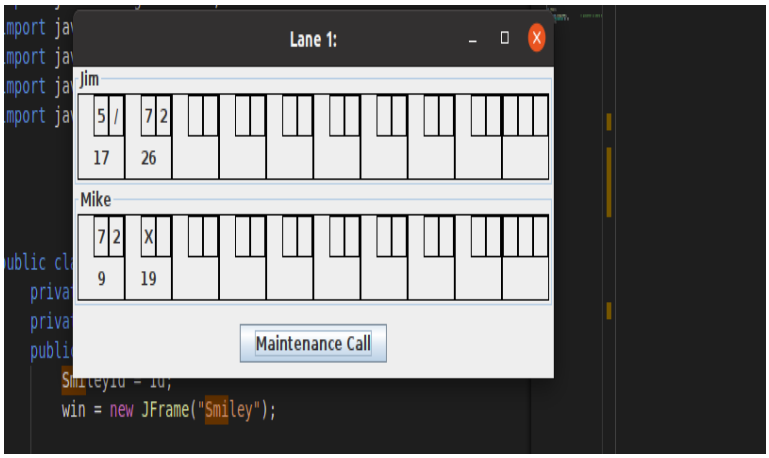


- We have also covered the interaction part. The project now takes the user's consent before bowling a bowl. This is how it looks.

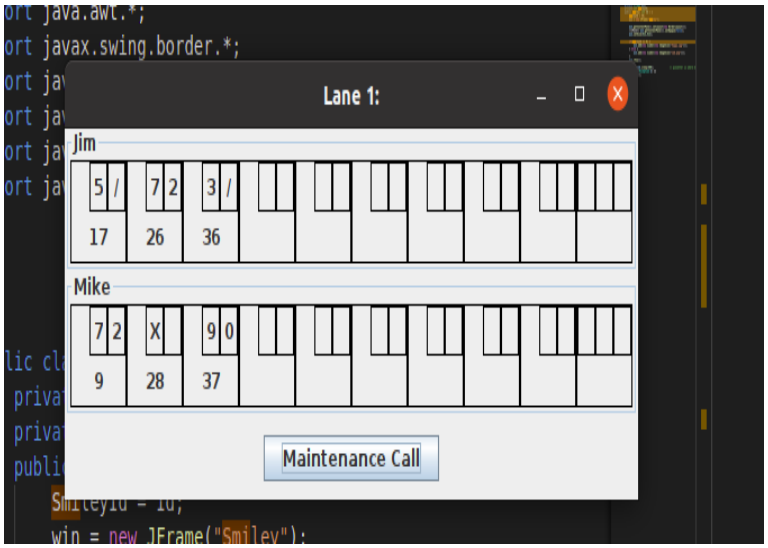


- **Smiley:** We have added support for the smileys depending upon the score of the player. The Smiley class object is created by the Pinsetter class. The object takes id as input and shows the corresponding smiley depending upon the score of the player.

This is how the smiley looks when the player gets a full score:



This is how the smiley looks when the player gets a zero score:



Design Patterns Used:

1) **Iterator Pattern:** For traversal of the lane score iterator pattern is used. It doesn't matter how many throws are given in a frame. We only need to iterate on the frames and can access values.

2) **Observer Pattern :** The event handling done while button click is a an example of observer pattern. We wait on thread, and notify the corresponding event-handler which carries out a task corresponding to the given button click. The three main observer class are **ControlDeskObserver, LaneObserver and PinSetterObserver.**

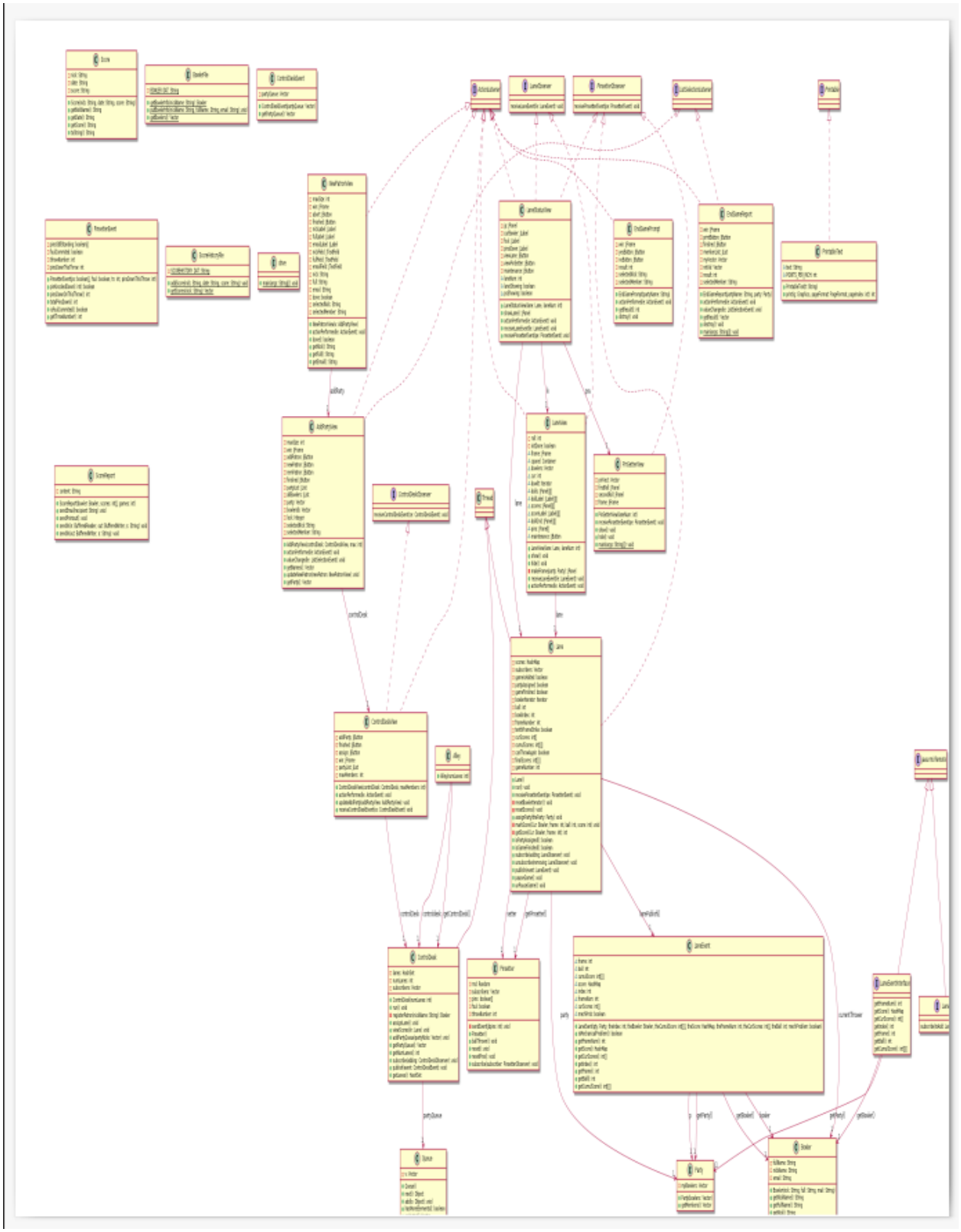
3) **Adapter Pattern :** It is a structural design pattern that works as a bridge between two incompatible interfaces. Here **ControlDesk Class** acts as an Adapter. It joins **Bowlers, Party and Queue** subsystems.

4) **Singleton Pattern :** A software design pattern that restricts the instantiation of a class to one "single" instance. The **drive class**, which acts as the main function in this program, and is instantiated only once in its lifetime.

5) **Layered Architecture:** We have used layered architecture wrt to Database. So as per the requirements we have created a separate layer for the database. And data is handled by the SQL lite Database.

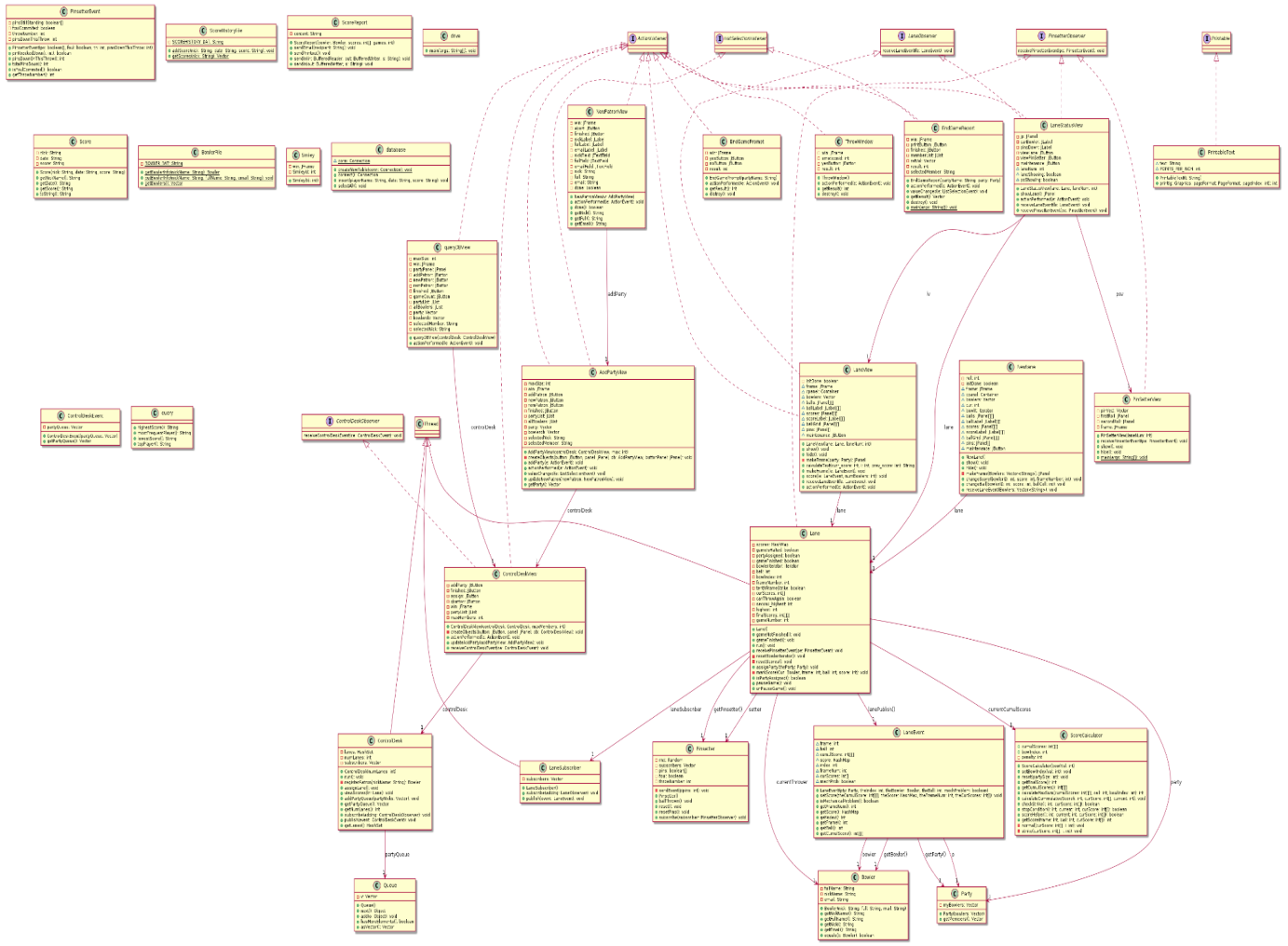
UML Diagrams Before Adding Features

For more clarity we have added this image in the misc section so that it can be zoomed in.

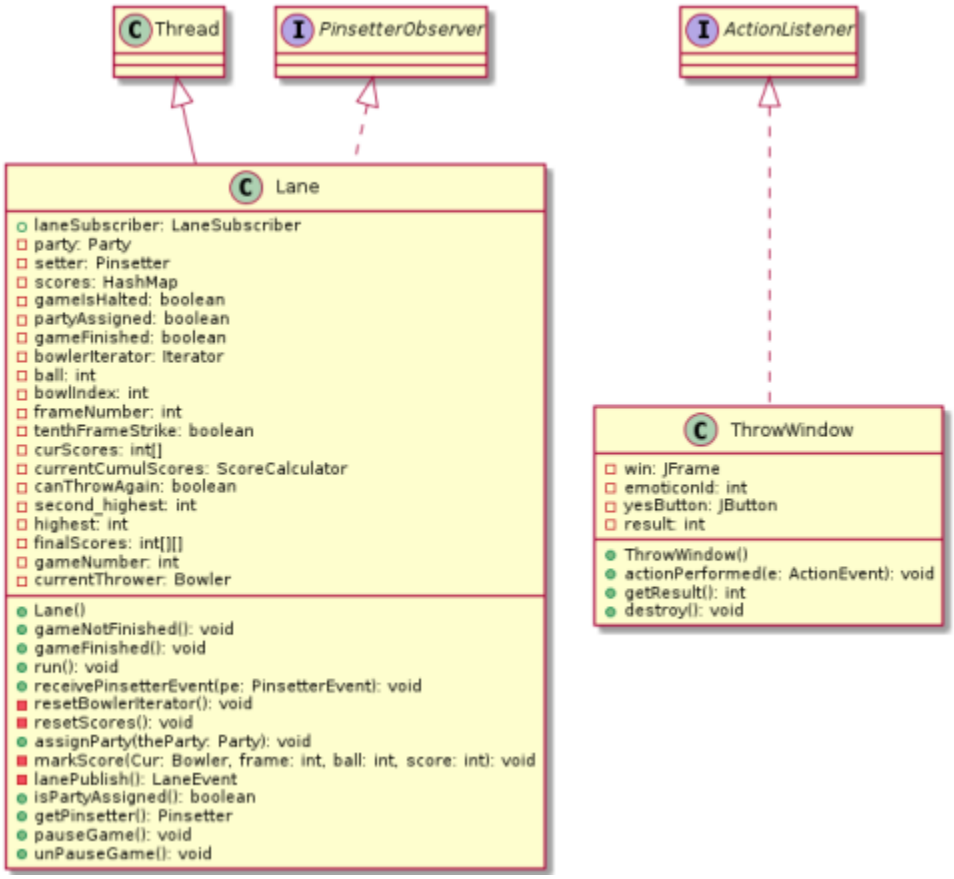


UML Diagrams After Adding Features

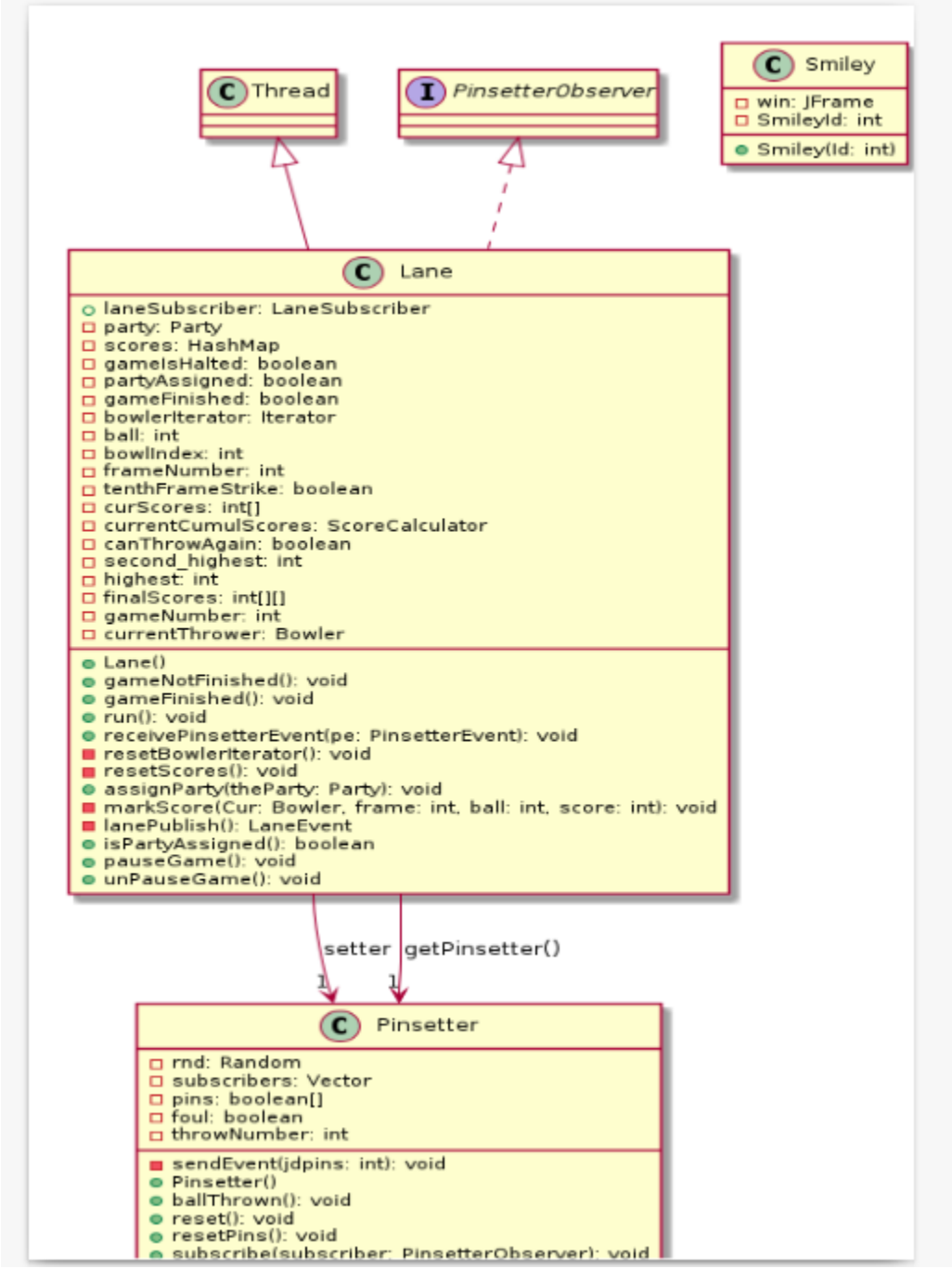
For more clarity we have added this image in the misc section so that it can be zoomed in.



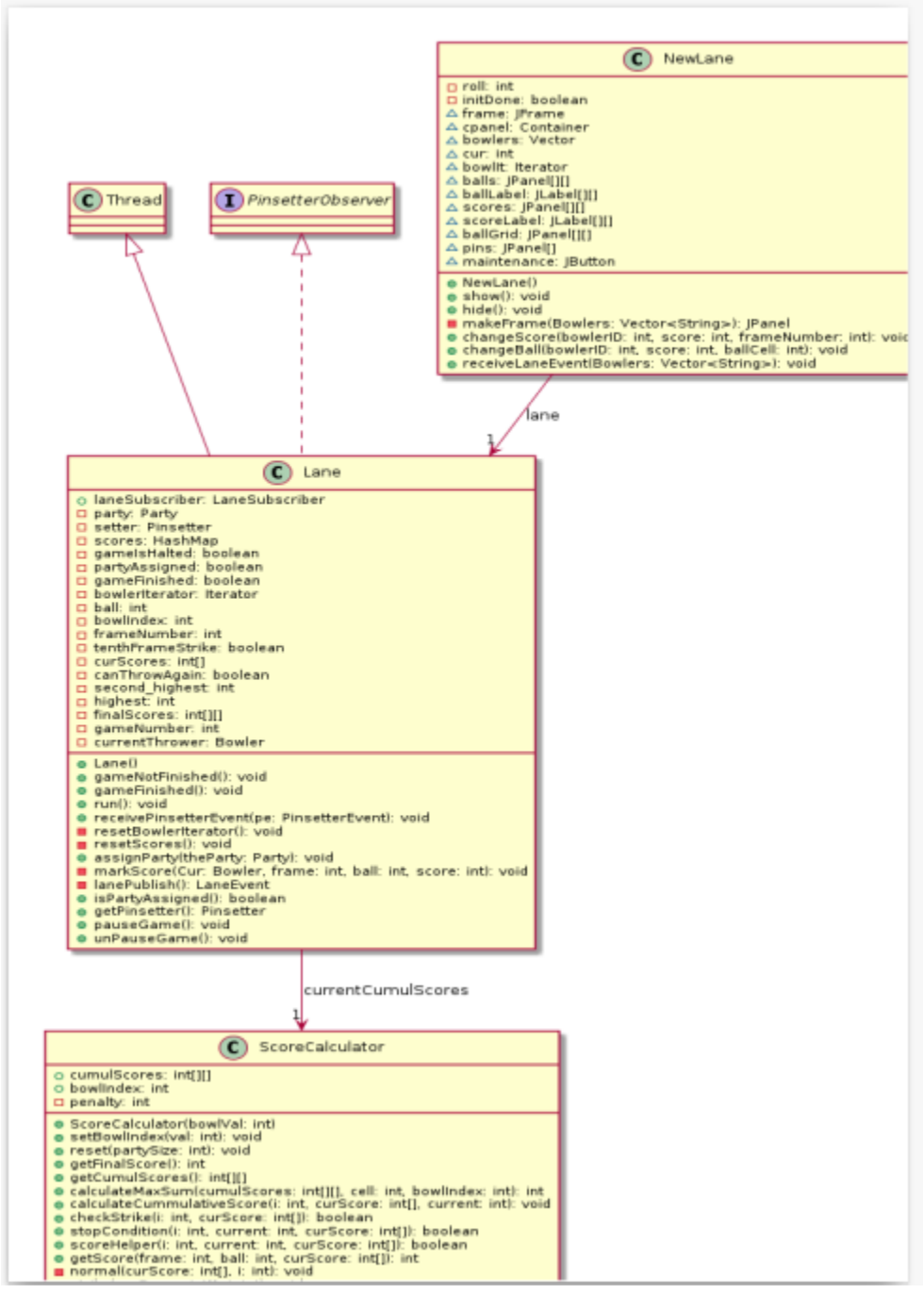
Throw Window Classes (Added for interactive feature)



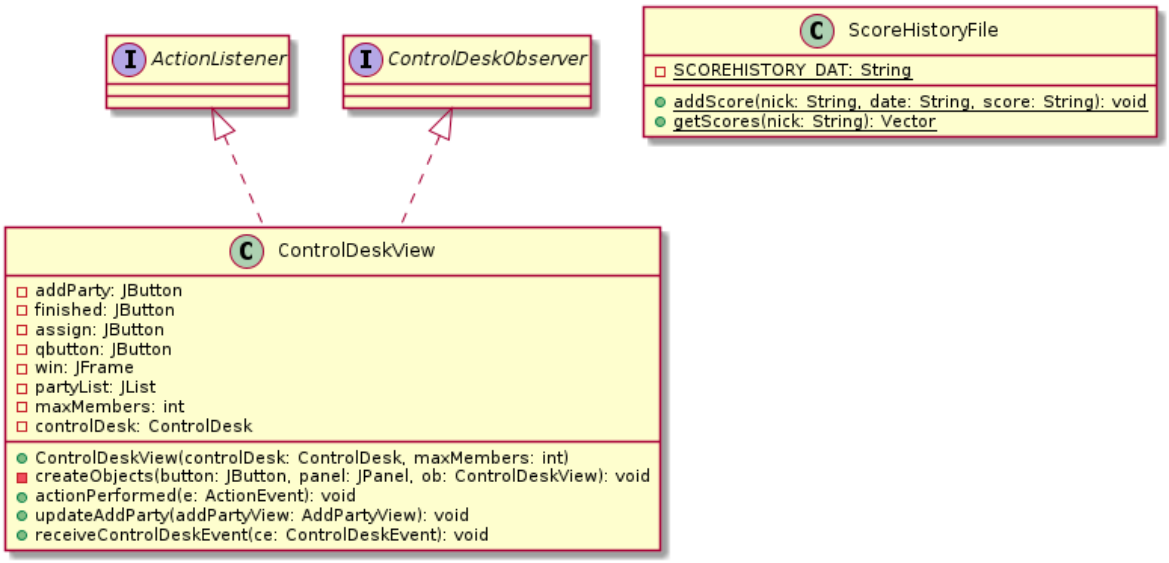
Smiley Classes (Added for displaying emoji based on the score)



New Lane Classes (Added to resolve Tiebreaker)

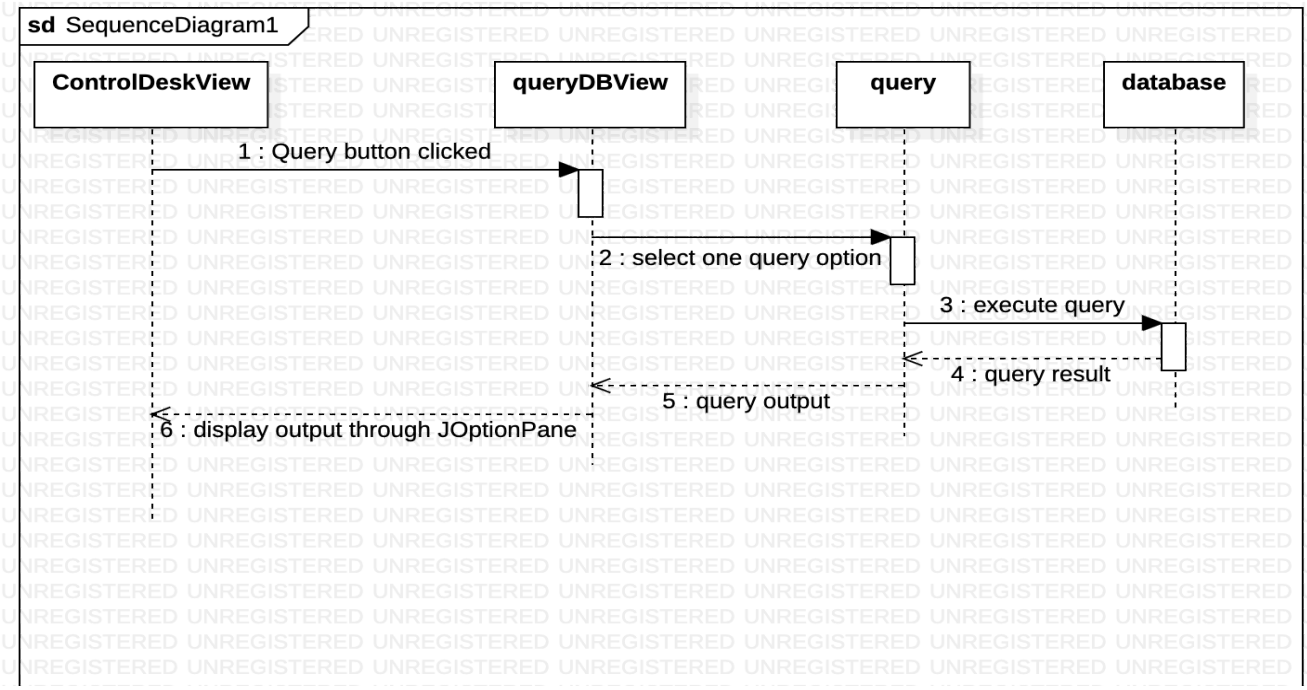


Score History Classes (Added to store the result in SQL Database)

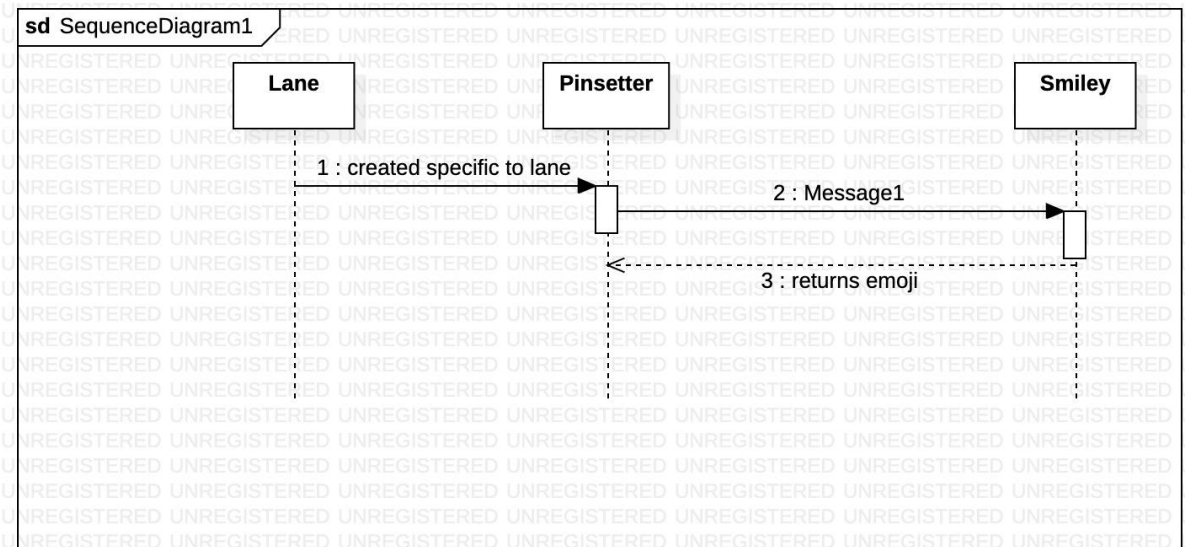


UML Sequence Diagram

- **Sequence diagram of ad-hoc query execution**



Sequence diagram from Smiley Class



Classes Overview

S.No.	Class Name	Responsibility
1.	**Driver**	It is the main class of the project. It initializes the game, sets the parameters like the number of lanes, maximum patrons per party. It also initializes all the necessary objects.
2.	**Control Desk**	It handles the control desk window. It assigns the lane, creates parties, and maintains the wait queue. It is responsible for broadcasting the event to subscribing objects. Creating a new patron.
3.	**Lane**	Lanes are handled by the lane class. This is responsible to start a new thread for each lane and assigning a party to that lane. It also updates the score and acts as an observer to the subscriber. Later we separated it into two more classes.
4.	**Pin Setter**	It performs pin setter simulation and notifies the events.
5.	**Bowler and Bowler File**	Information about the bowler will be stored here. It is used to add a new bowler, get details of all bowlers, validation of the bowler.
6.	**Endgame prompt**	Prompts after the end of the game to prompt if the player wants to play another game.

7.	**EndgameRep.**	It allows the user to make the game report at the end.
8.	**Score Calculator**	It helps the lane class to calculate the score using strike and normal functions. This is the result of refactoring the prior code. Also added the feature for the penalty in case of the gutter.
9.	**Score report**	Creates the final score of each person in the party and sends their respective reports by email.
10.	**AddPartyView**	It will add a new patron to the party. Removing a patron from the party. Creating a new Patron. Finished party selection and returned the latest state of the party.
11.	**ControlDeskEvent**	It returns the vector consisting of the names of the party in the wait queue.
12.	**ControlDeskView**	It handles GUI for the control desk. It facilitates handler for action events. Receive a new party from addPartyView.
13.	**Drive**	It is driver class for the game. Creates alles with the number of lanes. It also helps in activating the control desk object.
14.	**Laneevent**	consists of a setter and getter function for all lane functions.
15.	**PinSetterView**	displays the pinsetter GUI. Receive the current state of the pins and change the GUI looks.
16.	**ThrowWindow**	This class is responsible for pooping a window before the user bowls.
17.	**New Lane**	This class is responsible for creating a new lane in case of a tie-breaker between first and second highest.
18.	**Query**	Contains code for sql queries like get highest score,lowest score,etc
19.	**queryDBView**	Provides user interface to make ad-hoc queries
20.	**database**	This class is used to connect to sqlite database and define the database schema .
21.	**Smiley**	This class is responsible for providing Smiley functionality in the bowling alley submission.

ANALYZING THE ORIGINAL CODE DESIGN

We shall start with the Pros and then will discuss the Cons of the given project.

Pros :-

- **Proper Comments** :- The code was well commented and the purpose and functionality of the most part of the system was provided.
- **Low Coupling** :- The overall system as well as the subsystems were having low coupling metric.

Cons :-

- **Dead Code :-** The original code contains a lot of code which is commented out or has empty blocks. Some variables and methods which are declared, have also not been used anywhere in the system.
- **Large Classes and Big Functions** :- Some classes were burdened with way too many methods and some functions had a lot of code which could have been modularized.
- **Big Nested If Depth** :- Some of the functions which could have been simply written had way too many if's and else.
- **Duplicate Code** :- Similar kind of job was being done in some of the methods where the code was copy pasted.
- **Use of Deprecated Methods** :- Many system functions used in the project were deprecated which could have been replaced by new functions.
- **Too Many Parameters** :- Some functions and constructors had way too many parameters.
- "Print Report" functionality is missing.

CODE SMELLS AND CHANGES

S.No	Class Name	Code Smell Instances and changes performed
1	**LaneView**	Removed variables which are unused. The variables are :- ● private int roll ● int cur ● Iterator bowlIt

		• Insets buttonMargin = new Insets(4,4,4,4) High Cyclomatic Complexity :- • public void receiveLaneEvent(LaneEvent le) High Nested If Depth :- • Public void receiveLaneEvent(LaneEvent le) Improvements we have made :- • Added new functions calculateText,makeFrame to reduce cyclomatic complexity and have reduced Nested If Depth.
2	**LaneStatusView**	Removed variables which are unused. The variables are :- • Insets buttonMargin = new Insets(4,4,4,4) • private JLabel foul • Removed dead code
3	**Lane**	Removed functions which are not used. The functions are :- • public boolean isGameFinished() Also removed empty blocks of code present in this class. Removed variables :- • EndGamePrompt egp • vector subscriber Moved some functions out of Lane to create LaneSubscirbe class and ScoreCalculate class. Added GameFinished and GameNotFinished functions to reduce the cyclomatic complexity of run function, and also have simplified some of the if conditions.Also changed receivePinSetterEvent function to reduce complexity. Changed the if condition to correctly check if 2 strings are not same or not.
4	**LaneEventInterface**	Removed these functions as there are not used anywhere. • public int getFrameNum( ) throws java.rmi.RemoteException • public int[] getCurScores( ) throws java.rmi.RemoteException • public int[][] getCumulScore() throws java.rmi.RemoteException
5	**EndGamePrompt**	Removed Unused Variables :-

		• selectedNick • selectedMember • ButtonMargin Also removed many unused import statements.
6	**EndGameReport**	Removed Unused Variables :- • myVector • EndGame e • ButtonMargin • String partyName Also refactored code in this class.
7	**LaneEvent**	Removed getCurScores() function as it is not used. Also reduced the number of parameters passed to the constructor and added a new set method.
8	**LaneObserver**	Removed extra semicolons
9	**ScoreCalculator (Derived from Lane Class)**	normal function :- • Reduced the nested if depth. getScore :- • Removed unused argument Bowler. • Reduced nested if depth • Created a new function calculateCumulativeScore to reduce the cyclomatic complexity.
10	**AddPartyView**	Removed unused variables :- • lock • Insets buttonMargin • NewPatronView Removed unused getNames() function and unused import statements. In actionPerformed function reduced nested if depth. Removed repeating code by moving it into a new function called createObjects
11	**LaneSubscriber**	Modified the if conditions. Removed unsubscribe() function which is not used.
12	**LaneStatusView**	Reduced the actionperformed function cyclomatic complexity.
13	**NewPatronView**	Removed variables :- • int maxSize • Insets buttonMargin

		• selectedNick • selectedMember Removed unused import statements.
14	**Alley.java**	Removed the class by creating the ControlDesk object directly in the drive.java class.
15	**ControlDeskView.java**	Removed repeating code by moving it into a new function called createObjects
16	**PinSetterView**	Contained Dead code and Deprecated Functions
17	**PinSetterObserver**	Modifier public is redundant for interface
18	**Printable Text**	Unused Import Statement
19	**ScoreHistoryFile**	More General exception is already used in throws list
20	**NewPatronView**	Unused variables and import statements
21	**Pinsetter**	Added functionality to show smiley in a new frame
22	**Smiley**	Made Smiley class

Analyzing the enhanced code

- **Low Coupling** :- The dependencies between the classes were moderate and we have tried to make it low by passing the parameters locally and removing the redundant ones wherever possible. We have extended our class list to break down large files such as Lane into different subclasses. We have made sure that these classes were mostly independent and did not require too many other dependencies to increase the coupling.

- **High Cohesion** :- Cohesion tells about the consistency and organization of different units. The more tasks a single class tries to perform, we have a problem with cohesion there . The long classes had numerous methods which often became unrelated and too broad. We split such classes eg. Lane class has been divided into Lane, LaneSubscribe and ScoreCalculator class.

- **Separation of Concerns** :- Separation of Concerns is a design principle for separating a system into distinct sections such that each section addresses a separate concern. An example of how we achieved this in the refactored design is by creating a separate score calculating class. Previously Lane Class had a method getScore() which calculates the

score but we have created a separate class ScoreCalculator for calculating the score and the updated score is sent to Lane Class to mark.

- **Dead Code Elimination** :- An optimization that removes code which does not affect the program results. We also eliminated extra methods and variables which are mentioned above.

- **Extensibility** :- As we ensured low coupling, we made sure that it was easier to introduce new modules. As we wrote basic code for the UI, we could easily extend it to add specific features in the respective classes.

- **Reusability** :- To ensure the code reusability several methods were written. Wherever in the original code we found that a similar kind of task was done by copy-pasting we modularized the code.

- **Single Responsibility Principle:-** The new classes that have been added ensure that this principle is followed.For instance, the database.java class solely handles the database section, the query.java class handles all the user queries.This also ensures the open/closed principle i.e if code needs to be extended there is no need to modify the existing code.

Metric Analysis

Following questions has been answered in this section.

- What were the metrics for the code base? What did these initial measurements tell you about the system.
- How did you use these measurements to guide your refactoring?
- How did your refactoring affect the metrics? Did your refactoring improve the metrics? In all areas? In some areas? What contributed to these results?

1. McCabe Cyclomatic Complexity

Measurement tell us about the system

- McCabe's cyclomatic complexity is a software quality metric that quantifies the complexity of a software program.
- It indicate how difficult it will be for the user to traverse through the whole code and understand.
- We have used Metric 2.0 and by default the max limit of the McCabe Cyclomatic Complexity is set to 10.

Inference: getScore, run and recieveLaneEvent have very high complexity. If we are able to simply these functions, this parameter will definitely fall down.

Functions that violated this constraints are as follows:

Lane.java(Lane):

- getScore - 38
- run - 19
- receivePinsetterEvent - 12

LaneView.java(LaneView)

- receiveLaneEvent - 19

LaneStatusView.java

- actionPerformed - 11

AddPartyView.java

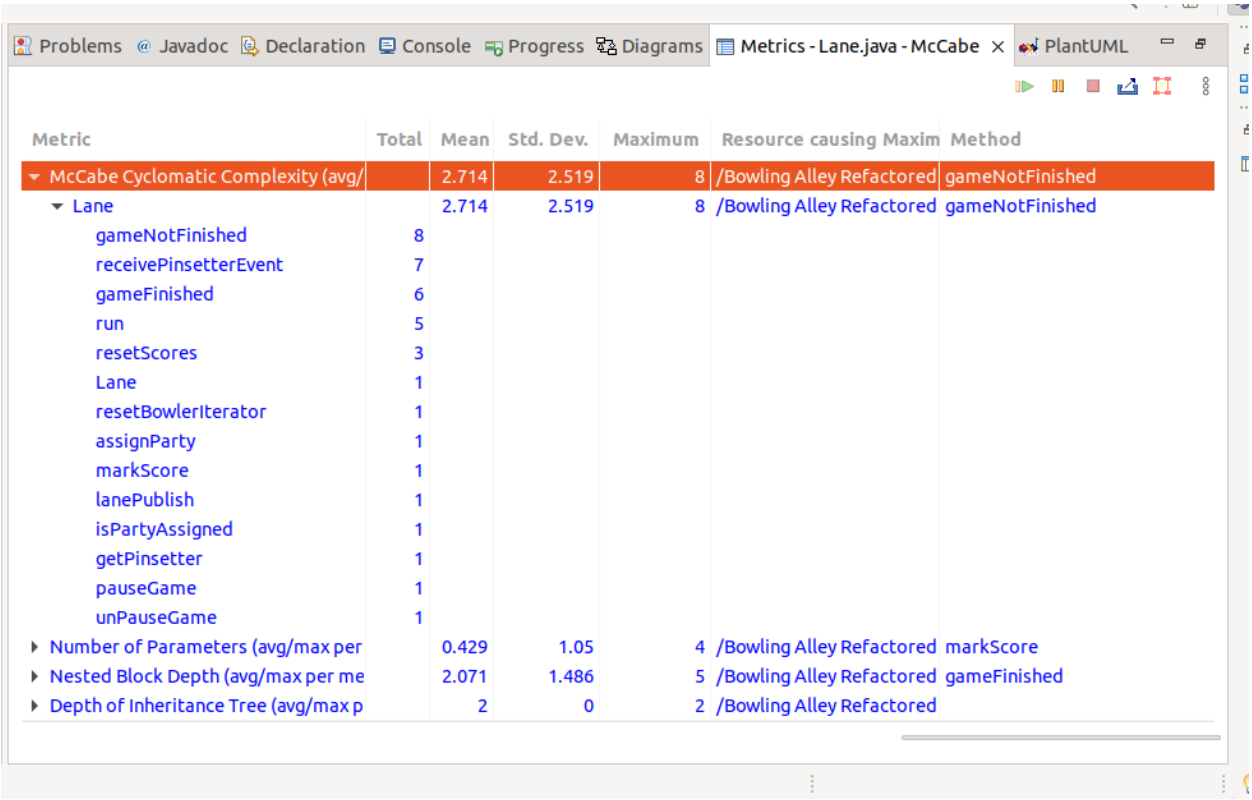
- actionPerformed - 11

Guide for refactoring

We targetted these functions and try to simply it, so that it is much readable. So we try to split up the methods and made different functions for a specific task. Multiple nested loops blocks has been converted to functions. We try to stick to make the function smallers and decompose large functions into multiple smaller ones.

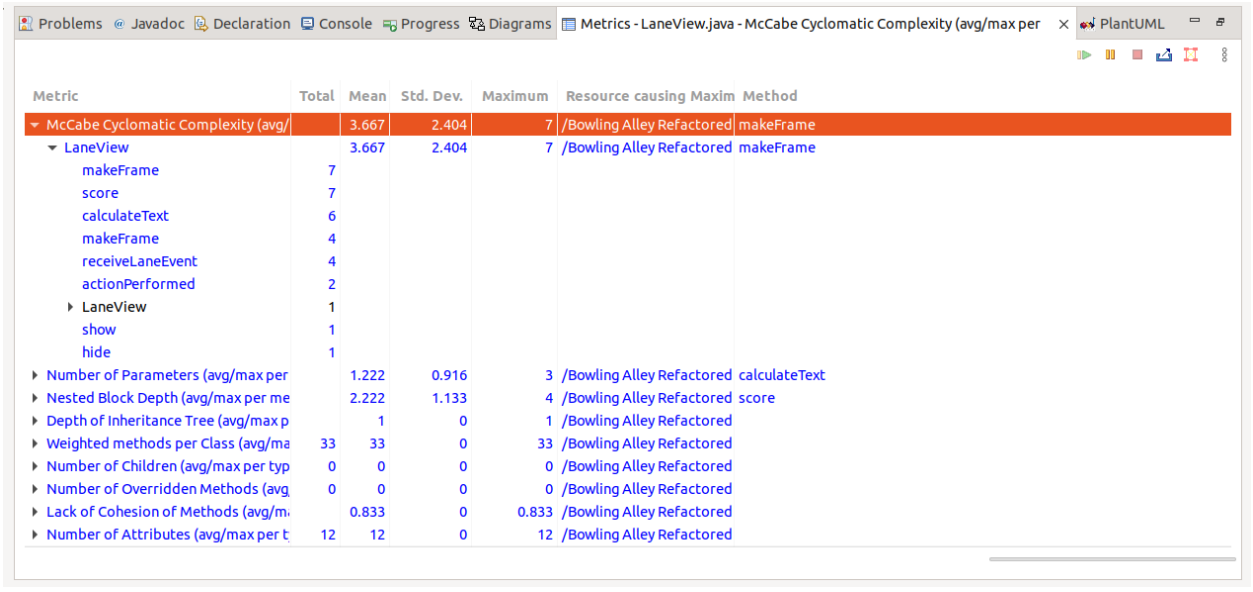
After Refactoring

- **Lane.java**



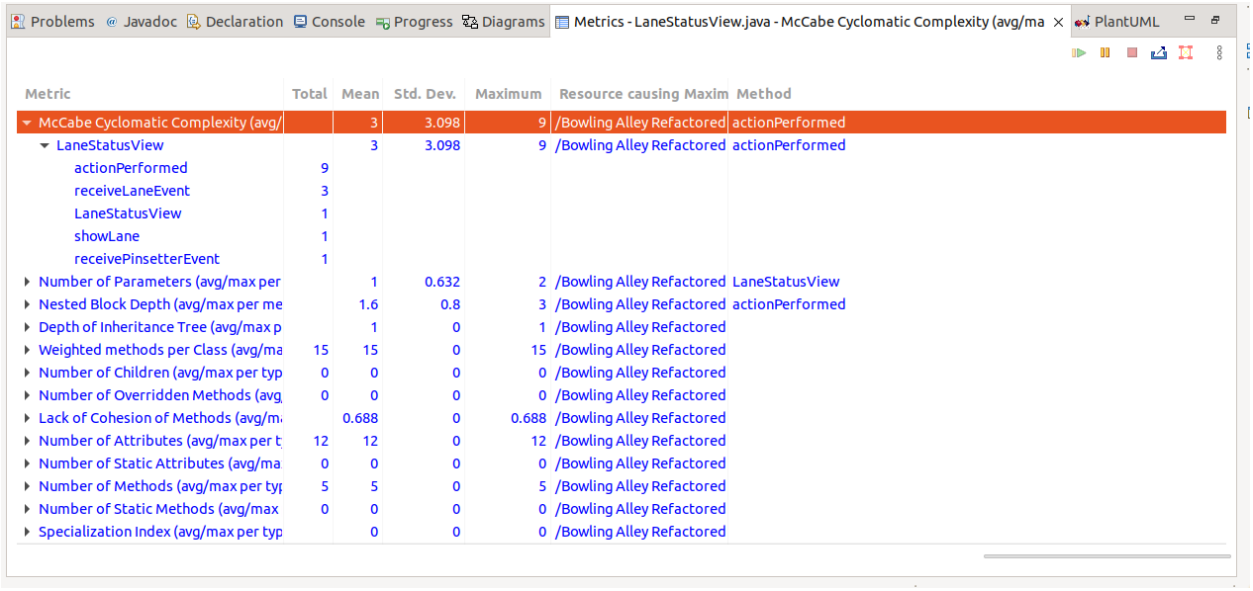
Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maxim	Method
McCabe Cyclomatic Complexity (avg/		2.714	2.519	8	/Bowling Alley Refactored	gameNotFinished
Lane		2.714	2.519	8	/Bowling Alley Refactored	gameNotFinished
gameNotFinished	8					
receivePinsetterEvent	7					
gameFinished	6					
run	5					
resetScores	3					
Lane	1					
resetBowlerIterator	1					
assignParty	1					
markScore	1					
lanePublish	1					
isPartyAssigned	1					
getPinsetter	1					
pauseGame	1					
unPauseGame	1					
Number of Parameters (avg/max per		0.429	1.05	4	/Bowling Alley Refactored	markScore
Nested Block Depth (avg/max per me		2.071	1.486	5	/Bowling Alley Refactored	gameFinished
Depth of Inheritance Tree (avg/max p		2	0	2	/Bowling Alley Refactored	

- **LaneView.java**



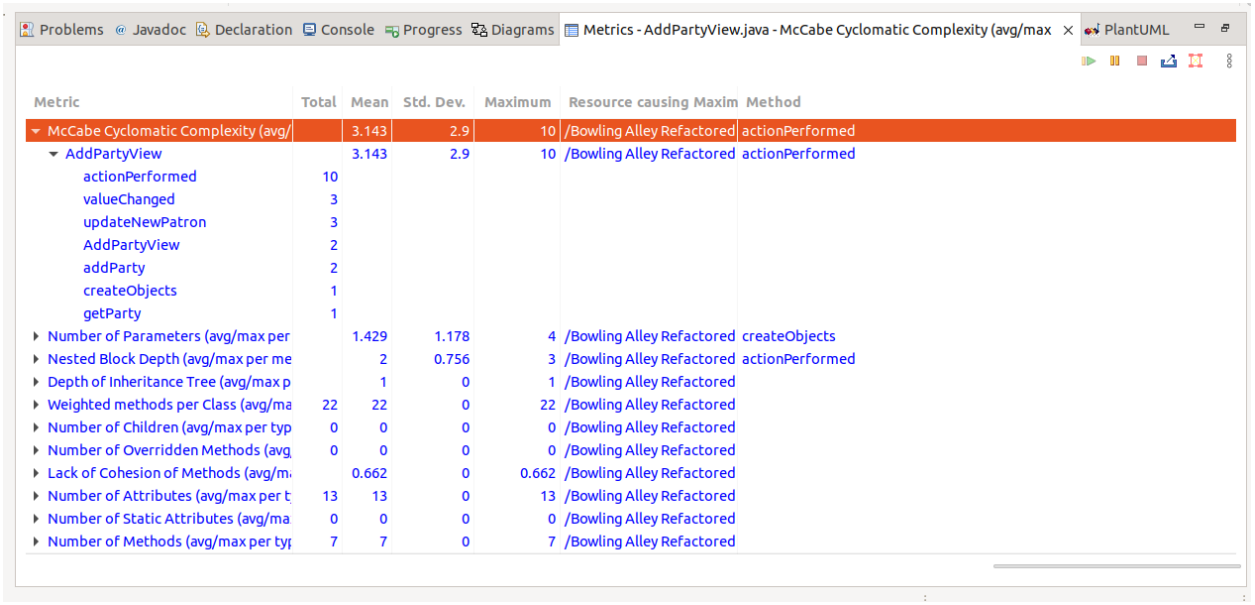
Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maxim	Method
McCabe Cyclomatic Complexity (avg/		3.667	2.404	7	/Bowling Alley Refactored	makeFrame
LaneView		3.667	2.404	7	/Bowling Alley Refactored	makeFrame
makeFrame	7					
score	7					
calculateText	6					
makeFrame	4					
receiveLaneEvent	4					
actionPerformed	2					
LaneView	1					
show	1					
hide	1					
Number of Parameters (avg/max per		1.222	0.916	3	/Bowling Alley Refactored	calculateText
Nested Block Depth (avg/max per me		2.222	1.133	4	/Bowling Alley Refactored	score
Depth of Inheritance Tree (avg/max p		1	0	1	/Bowling Alley Refactored	
Weighted methods per Class (avg/ma	33	33	0	33	/Bowling Alley Refactored	
Number of Children (avg/max per typ	0	0	0	0	/Bowling Alley Refactored	
Number of Overridden Methods (avg	0	0	0	0	/Bowling Alley Refactored	
Lack of Cohesion of Methods (avg/ma		0.833	0	0.833	/Bowling Alley Refactored	
Number of Attributes (avg/max per t	12	12	0	12	/Bowling Alley Refactored	

- **LaneStatusView.java**



Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maxim	Method
McCabe Cyclomatic Complexity (avg/		3	3.098	9	/Bowling Alley Refactored	actionPerformed
LaneStatusView		3	3.098	9	/Bowling Alley Refactored	actionPerformed
actionPerformed	9					
receiveLaneEvent	3					
LaneStatusView	1					
showLane	1					
receivePinsetterEvent	1					
Number of Parameters (avg/max per		1	0.632	2	/Bowling Alley Refactored	LaneStatusView
Nested Block Depth (avg/max per me		1.6	0.8	3	/Bowling Alley Refactored	actionPerformed
Depth of Inheritance Tree (avg/max p		1	0	1	/Bowling Alley Refactored	
Weighted methods per Class (avg/ma	15	15	0	15	/Bowling Alley Refactored	
Number of Children (avg/max per typ	0	0	0	0	/Bowling Alley Refactored	
Number of Overridden Methods (avg	0	0	0	0	/Bowling Alley Refactored	
Lack of Cohesion of Methods (avg/m		0.688	0	0.688	/Bowling Alley Refactored	
Number of Attributes (avg/max per t	12	12	0	12	/Bowling Alley Refactored	
Number of Static Attributes (avg/ma	0	0	0	0	/Bowling Alley Refactored	
Number of Methods (avg/max per typ	5	5	0	5	/Bowling Alley Refactored	
Number of Static Methods (avg/max	0	0	0	0	/Bowling Alley Refactored	
Specialization Index (avg/max per typ		0	0	0	/Bowling Alley Refactored	

- **AddPartyView.java**



Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maxim	Method
McCabe Cyclomatic Complexity (avg/		3.143	2.9	10	/Bowling Alley Refactored	actionPerformed
AddPartyView		3.143	2.9	10	/Bowling Alley Refactored	actionPerformed
actionPerformed	10					
valueChanged	3					
updateNewPatron	3					
AddPartyView	2					
addParty	2					
createObjects	1					
getParty	1					
Number of Parameters (avg/max per		1.429	1.178	4	/Bowling Alley Refactored	createObjects
Nested Block Depth (avg/max per me		2	0.756	3	/Bowling Alley Refactored	actionPerformed
Depth of Inheritance Tree (avg/max p		1	0	1	/Bowling Alley Refactored	
Weighted methods per Class (avg/ma	22	22	0	22	/Bowling Alley Refactored	
Number of Children (avg/max per typ	0	0	0	0	/Bowling Alley Refactored	
Number of Overridden Methods (avg	0	0	0	0	/Bowling Alley Refactored	
Lack of Cohesion of Methods (avg/m		0.662	0	0.662	/Bowling Alley Refactored	
Number of Attributes (avg/max per t	13	13	0	13	/Bowling Alley Refactored	
Number of Static Attributes (avg/ma	0	0	0	0	/Bowling Alley Refactored	
Number of Methods (avg/max per typ	7	7	0	7	/Bowling Alley Refactored	

2. Number of parameters

Measurements tell us about the system

- It is very difficult to understand the code if the we pass to many parameter to the functions.
- By default in Metric 2.0 it is set to maximum of 5.

Inference
Function LaneEvent have higher number of parameter. If we can reduce it to 5 this issue will be resolved.

Functions that violated this constraints are as follows:
LaneEvent.java(LaneView)

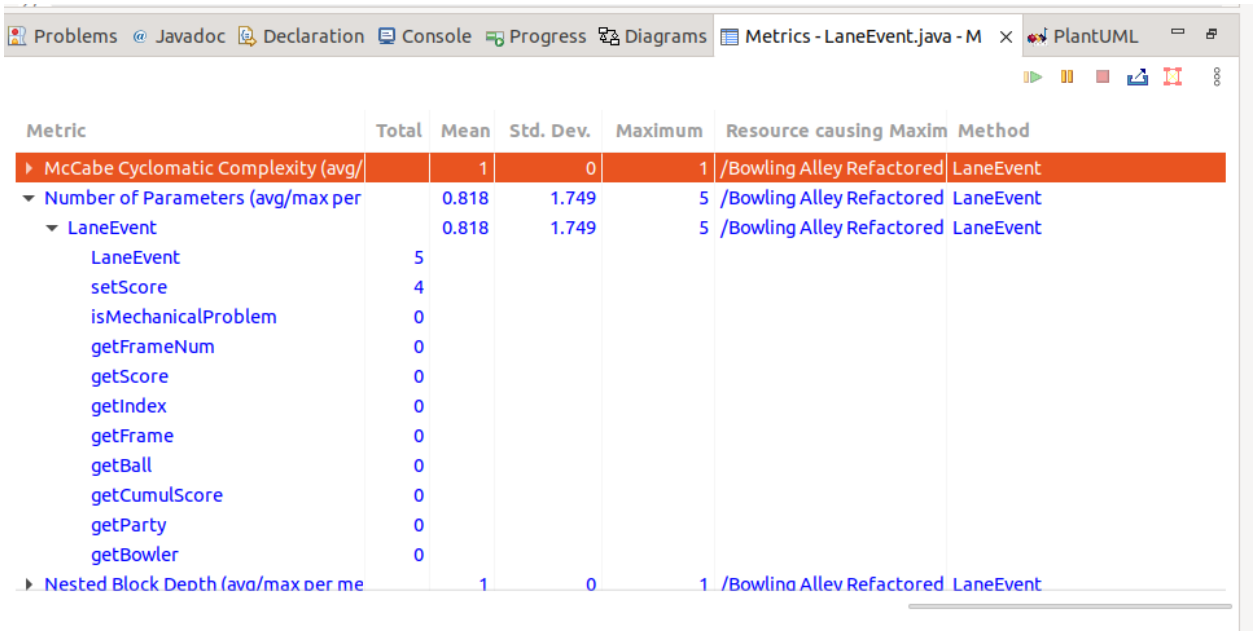
- LaneEvent - 9

Guide for refactoring

- Instead of passing all the parameter at once we have added a extra function which facilitates the exchange of parameter.
- Objects can also be passed but we used the earlier one.
- Splitting up the method also reduces the parameter, this can also help.

After Refactoring

- **LaneEvent.java**



Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maxim	Method
McCabe Cyclomatic Complexity (avg/		1	0	1	/Bowling Alley Refactored	LaneEvent
Number of Parameters (avg/max per		0.818	1.749	5	/Bowling Alley Refactored	LaneEvent
LaneEvent		0.818	1.749	5	/Bowling Alley Refactored	LaneEvent
LaneEvent	5					
setScore	4					
isMechanicalProblem	0					
getFrameNum	0					
getScore	0					
getIndex	0					
getFrame	0					
getBall	0					
getCumulScore	0					
getParty	0					
getBowler	0					
Nested Block Depth (avg/max per me		1	0	1	/Bowling Alley Refactored	LaneEvent

3. Nested Block Depth

Measurements tell us about the system

- It should be kept lower because as the nested block depth increases it will be difficult for us to track the flow as a human point of view.
- Metrics set maximum block depth to 5.

Inference

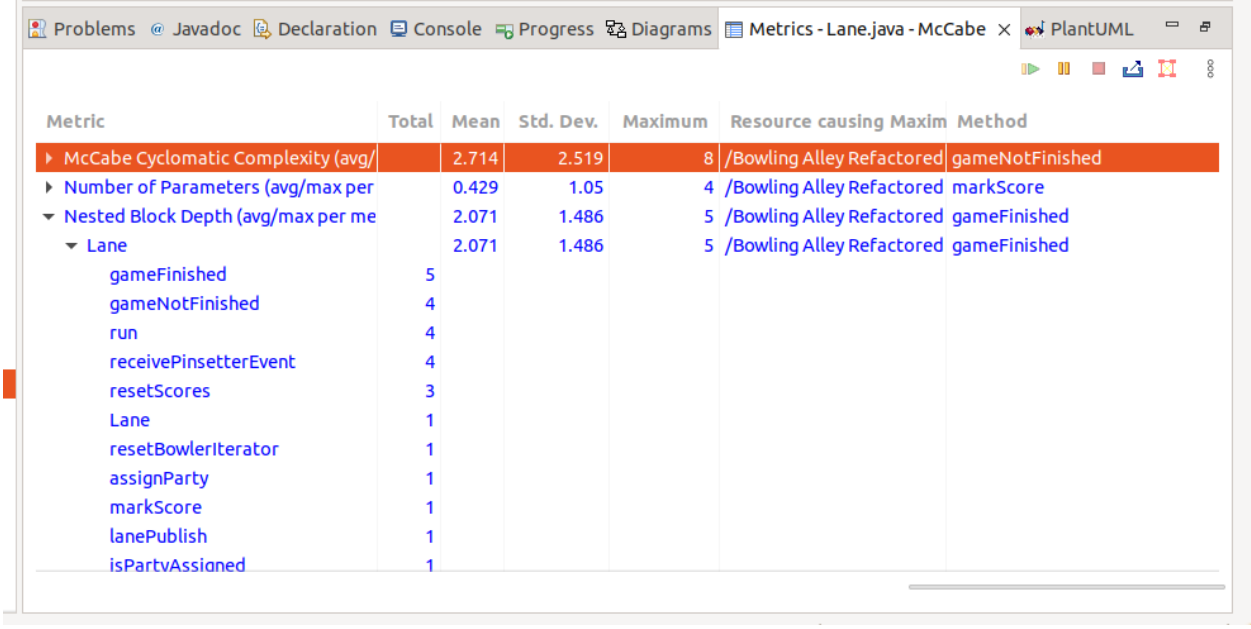
- Lane class have function run() that violates the constraints and can be rectified by rephrasing the if conditions and thus reducing the number of blocks.

Functions that violated this constraints are as follows:
Lane.java(Lane)
- run - 7

Guide for refactoring
- We have added the functions to avoid the nested block depth.
- We replaced redundant blocks with functions.

After Refactoring

- **Lane.java**



4. Lack of Cohesion of Methods

Measurements tell us about the system
- Cohesion is a measure of the degree to which the elements of the module are functionally related. A good software design will have high cohesion.
- Getters and setter used can increase cohesion.

Inference
- Getters didn't need splitting
- LaneView class does needed splitting. But sorted out while solving for first 3 parameters.

Functions that violated this constraints are as follows:

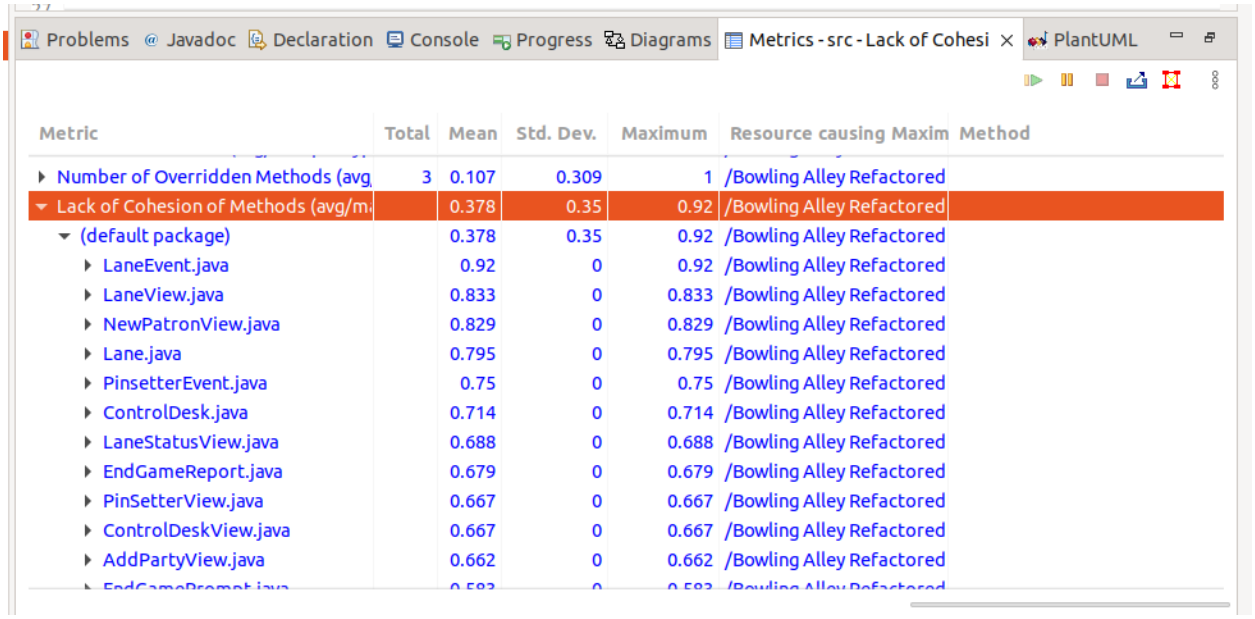
- No function violated any constraints here.

Guide for refactoring

- Split up into simple components by adding functions or making new classes altogether such that all the related functions are in a same module.

After Refactoring

- Very minor change in the metric.



Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maxim	Method
Number of Overridden Methods (avg	3	0.107	0.309	1	/Bowling Alley Refactored	
Lack of Cohesion of Methods (avg/m		0.378	0.35	0.92	/Bowling Alley Refactored	
(default package)		0.378	0.35	0.92	/Bowling Alley Refactored	
LaneEvent.java		0.92	0	0.92	/Bowling Alley Refactored	
LaneView.java		0.833	0	0.833	/Bowling Alley Refactored	
NewPatronView.java		0.829	0	0.829	/Bowling Alley Refactored	
Lane.java		0.795	0	0.795	/Bowling Alley Refactored	
PinsetterEvent.java		0.75	0	0.75	/Bowling Alley Refactored	
ControlDesk.java		0.714	0	0.714	/Bowling Alley Refactored	
LaneStatusView.java		0.688	0	0.688	/Bowling Alley Refactored	
EndGameReport.java		0.679	0	0.679	/Bowling Alley Refactored	
PinSetterView.java		0.667	0	0.667	/Bowling Alley Refactored	
ControlDeskView.java		0.667	0	0.667	/Bowling Alley Refactored	
AddPartyView.java		0.662	0	0.662	/Bowling Alley Refactored	

5. Method lines of code

Measurements tell us about the system

- Metric set the maximum limit to 100.
- It should not be too large. Because larger functions are hard to read.
- Usually large functions tend to have higher chances of bug.

Inference

- Try to keep number of line in any function less.

Functions that violated this constraints are as follows:

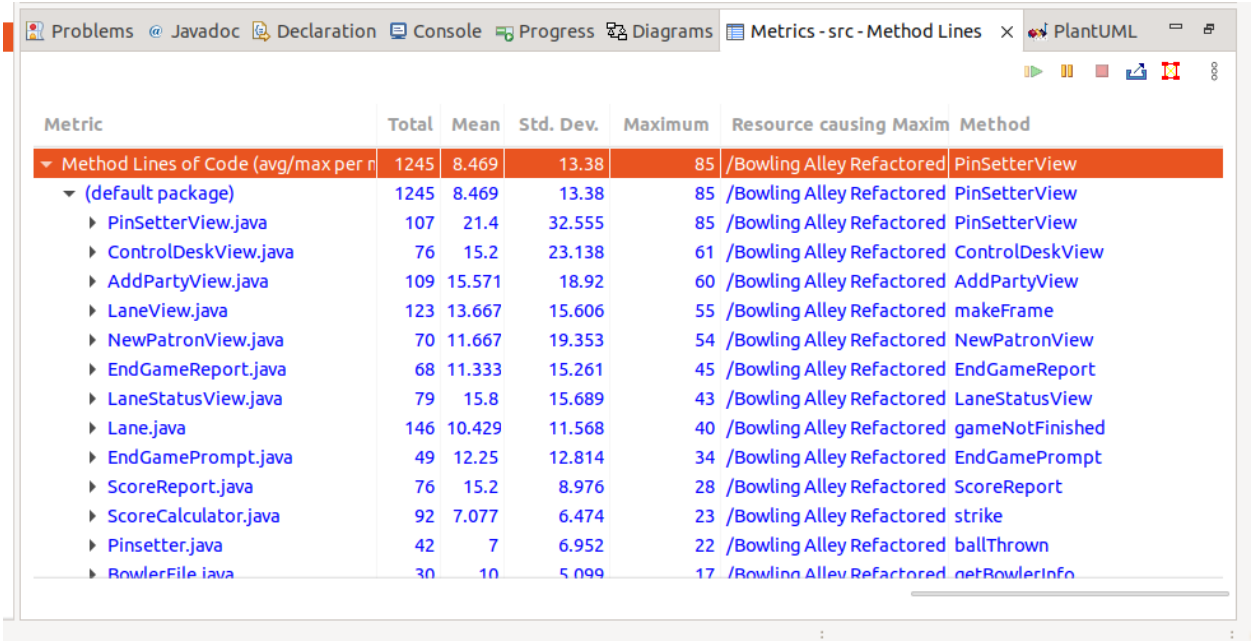
- No function violated any constraints here.

Guide for refactoring

- Decompose large functions into multiple functions.

After Refactoring

- The value decrease a bit. Since we refactor large functions into smaller one. But no significant changes.



Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maxim	Method
Method Lines of Code (avg/max per r	1245	8.469	13.38	85	/Bowling Alley Refactored	PinSetterView
(default package)	1245	8.469	13.38	85	/Bowling Alley Refactored	PinSetterView
PinSetterView.java	107	21.4	32.555	85	/Bowling Alley Refactored	PinSetterView
ControlDeskView.java	76	15.2	23.138	61	/Bowling Alley Refactored	ControlDeskView
AddPartyView.java	109	15.571	18.92	60	/Bowling Alley Refactored	AddPartyView
LaneView.java	123	13.667	15.606	55	/Bowling Alley Refactored	makeFrame
NewPatronView.java	70	11.667	19.353	54	/Bowling Alley Refactored	NewPatronView
EndGameReport.java	68	11.333	15.261	45	/Bowling Alley Refactored	EndGameReport
LaneStatusView.java	79	15.8	15.689	43	/Bowling Alley Refactored	LaneStatusView
Lane.java	146	10.429	11.568	40	/Bowling Alley Refactored	gameNotFinished
EndGamePrompt.java	49	12.25	12.814	34	/Bowling Alley Refactored	EndGamePrompt
ScoreReport.java	76	15.2	8.976	28	/Bowling Alley Refactored	ScoreReport
ScoreCalculator.java	92	7.077	6.474	23	/Bowling Alley Refactored	strike
Pinsetter.java	42	7	6.952	22	/Bowling Alley Refactored	ballThrown
BowlerFile.java	30	10	5.099	17	/Bowling Alley Refactored	getBowlerInfo

6. Depth of Inheritance Tree

Measurements tell us about the system

- Limit on the depth of inheritance was set to 5 by the Metric 2.0.
- Our maximum depth was 2.
- We have removed unnecessary inheritance.
- With such low value of depth of inheritance tree indicate the code base is relatively small and less complex.
- In mid size project if the value is less, it implies poor usage of OOPs concept.

Inference

- The code metric of the original code is strong in this aspects.

Functions that violated this constraints are as follows:

- No function violated any constraints here.

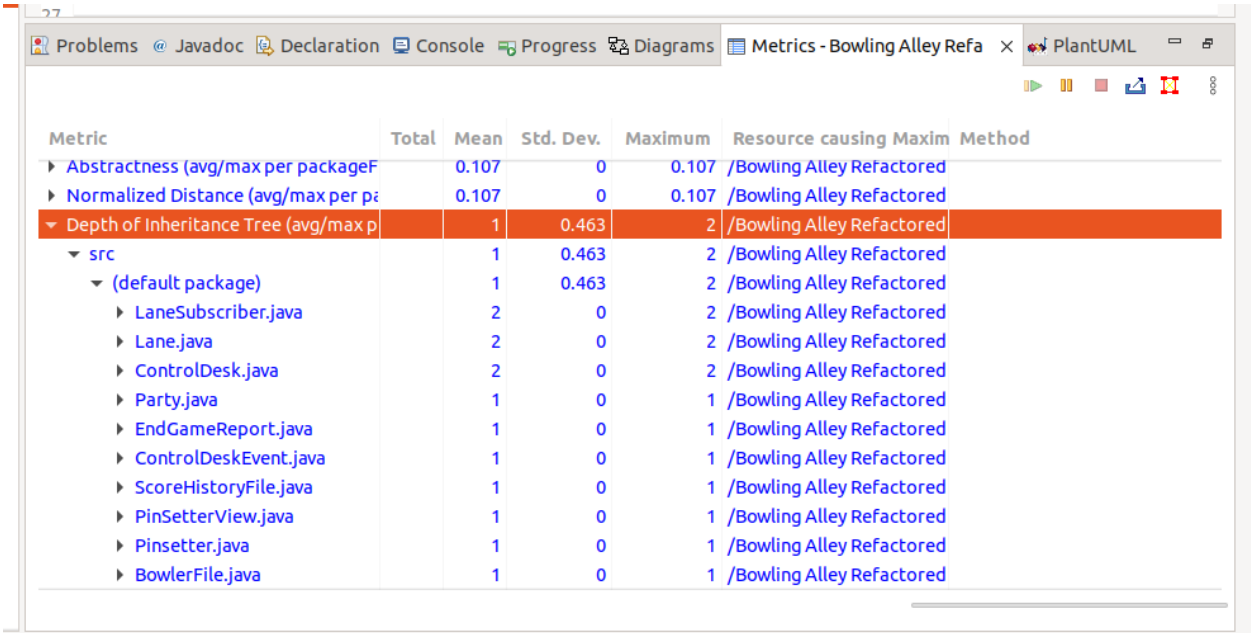
Guide for refactoring

- We made sure that the values remain maintained after recfactoring.

- If the requirements are not ment we can decompose classes.

After Refactoring

- Values retained the same.



Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maxim	Method
Abstractness (avg/max per packageF		0.107	0	0.107	/Bowling Alley Refactored	
Normalized Distance (avg/max per pa		0.107	0	0.107	/Bowling Alley Refactored	
Depth of Inheritance Tree (avg/max p		1	0.463	2	/Bowling Alley Refactored	
src		1	0.463	2	/Bowling Alley Refactored	
(default package)		1	0.463	2	/Bowling Alley Refactored	
LaneSubscriber.java		2	0	2	/Bowling Alley Refactored	
Lane.java		2	0	2	/Bowling Alley Refactored	
ControlDesk.java		2	0	2	/Bowling Alley Refactored	
Party.java		1	0	1	/Bowling Alley Refactored	
EndGameReport.java		1	0	1	/Bowling Alley Refactored	
ControlDeskEvent.java		1	0	1	/Bowling Alley Refactored	
ScoreHistoryFile.java		1	0	1	/Bowling Alley Refactored	
PinSetterView.java		1	0	1	/Bowling Alley Refactored	
Pinsetter.java		1	0	1	/Bowling Alley Refactored	
BowlerFile.java		1	0	1	/Bowling Alley Refactored	

7. Number of methods

Measurements tell us about the system

- Classes with less methods are less complex.
- In the original code there are classes Lane.java which has 17 methods.
- So we need to reduce this as it will difficult for people who will read this in future.

Inference

- Decompose the class into smaller ones. So, lane should be reduced.
- Lane Event have higher getters and setters, so can be ignored.
- Lane Interface can be ignored for the same reasons.

Classes that violated this constraints are as follows:

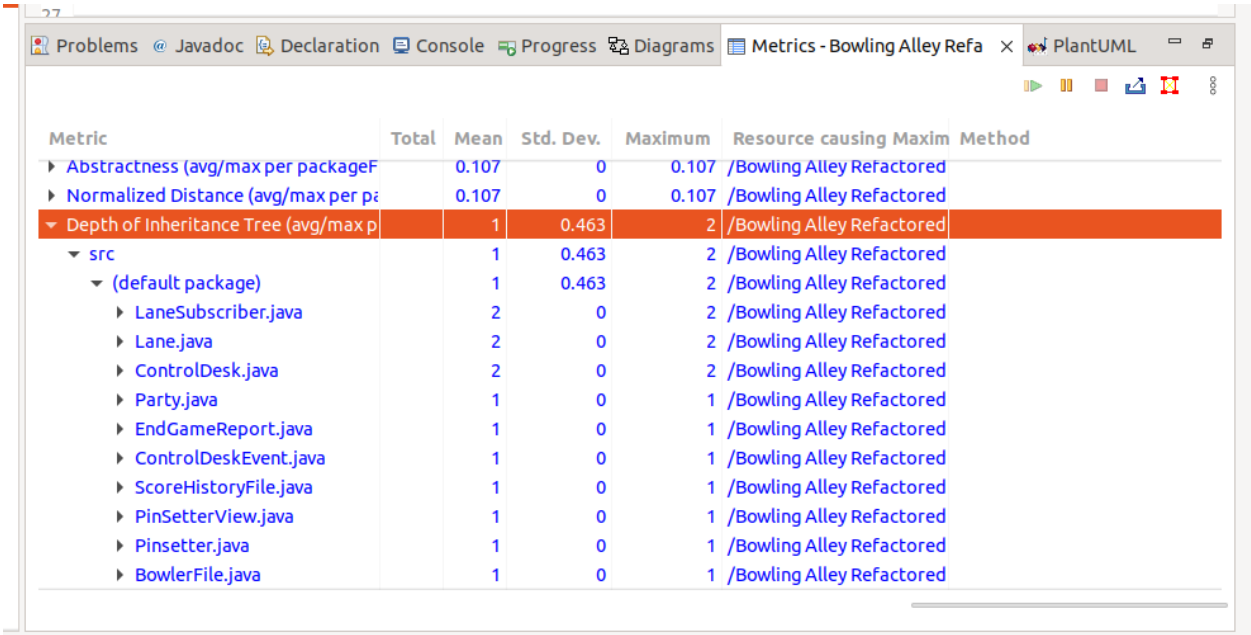
- Lane - 17
- LaneEvent - 11
- ControlDesk - 11
- LaneEventInterface - 9

Guide for refactoring

- We tried to decompose the Lane class into two more classes. ScoreCalculator and Lanesubsribe.

- To reduce the complexity. We were only able to reduce the dominant Lane Class.
- Removed unused methods.

After Refactoring



Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maxim	Method
Abstractness (avg/max per packageF		0.107	0	0.107	/Bowling Alley Refactored	
Normalized Distance (avg/max per pa		0.107	0	0.107	/Bowling Alley Refactored	
Depth of Inheritance Tree (avg/max p		1	0.463	2	/Bowling Alley Refactored	
src		1	0.463	2	/Bowling Alley Refactored	
(default package)		1	0.463	2	/Bowling Alley Refactored	
LaneSubscriber.java		2	0	2	/Bowling Alley Refactored	
Lane.java		2	0	2	/Bowling Alley Refactored	
ControlDesk.java		2	0	2	/Bowling Alley Refactored	
Party.java		1	0	1	/Bowling Alley Refactored	
EndGameReport.java		1	0	1	/Bowling Alley Refactored	
ControlDeskEvent.java		1	0	1	/Bowling Alley Refactored	
ScoreHistoryFile.java		1	0	1	/Bowling Alley Refactored	
PinSetterView.java		1	0	1	/Bowling Alley Refactored	
Pinsetter.java		1	0	1	/Bowling Alley Refactored	
BowlerFile.java		1	0	1	/Bowling Alley Refactored	

8. Number of classes

Measurements tell us about the system

- Higher number of classes shows the more complex systems.
- Earlier we have 29 classes, now we have increased this.
- It tells the project is relatively small sized.

Inference

- The code metric has given strong indication in this sense.
- So we can try to add a few extra classes by decomposing current one.

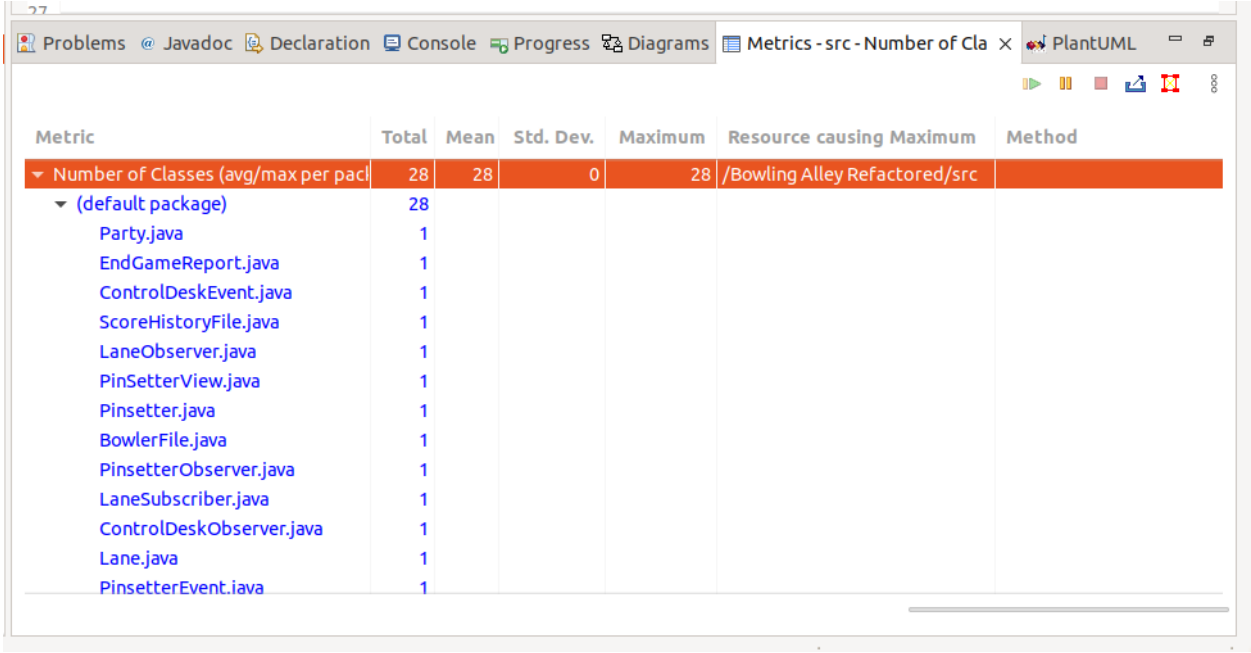
Violated of metric Limit:

- None

Guide for refactoring

- We tried to keep the metric value same.
- We have reduced useless classes like Alley and some interface.
- We have decomposed the Lane class and added two new class.
- We removed 3 classes and added 2 extra class. So keeping the count same to 28.

After Refactoring


Problems
Javadoc
Declaration
Console
Progress
Diagrams
Metrics - src - Number of Cla
PlantUML
Metric
Total
Mean
Std. Dev.
Maximum
Resource causing Maximum
Method
Number of Classes (avg/max per pacl
28
28
0
28
/Bowling Alley Refactored/src
(default package)
28
Party.java
1
EndGameReport.java
1
ControlDeskEvent.java
1
ScoreHistoryFile.java
1
LaneObserver.java
1
PinSetterView.java
1
Pinsetter.java
1
BowlerFile.java
1
PinsetterObserver.java
1
LaneSubscriber.java
1
ControlDeskObserver.java
1
Lane.java
1
PinsetterEvent.java
1