Unit-1 Project

Bowling Alley Management System

I. Contribution :

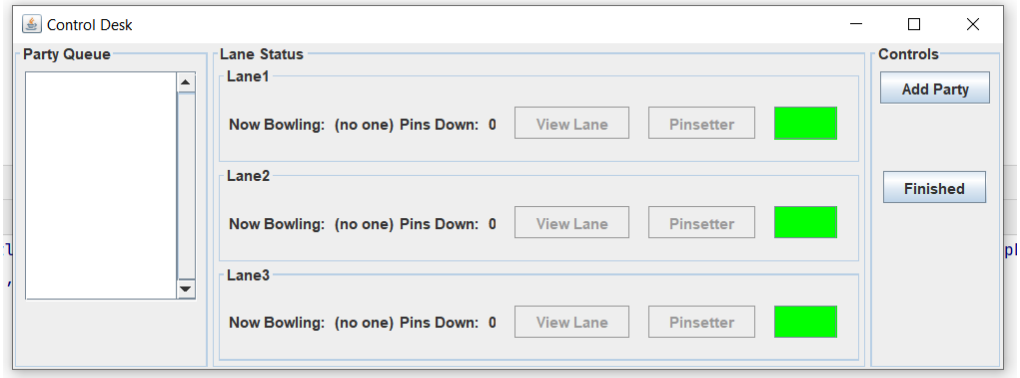
Team Members	Roll No.	Hours of Work	Contributions
Ashutosh Gupta	2021201085	32Hrs	• Initial code analysis • Designing UML Class Diagrams • Reducing Cyclomatic complexity of classes. • Observer - Mediator Pattern Enhancements and Report
Soumodipta Bose	2021201086	33Hrs	• Initial code analysis • Reducing Cyclomatic complexity of classes and design improvements. • View classes overhaul using UI Factory class, deadcode removal and moving methods to appropriate classes. • Metric Analysis and Report
Rahul Katyal	2021201083	12Hrs	• Initial code analysis • Report and documentation
Adwait Raste			No Contribution

II. Overview :

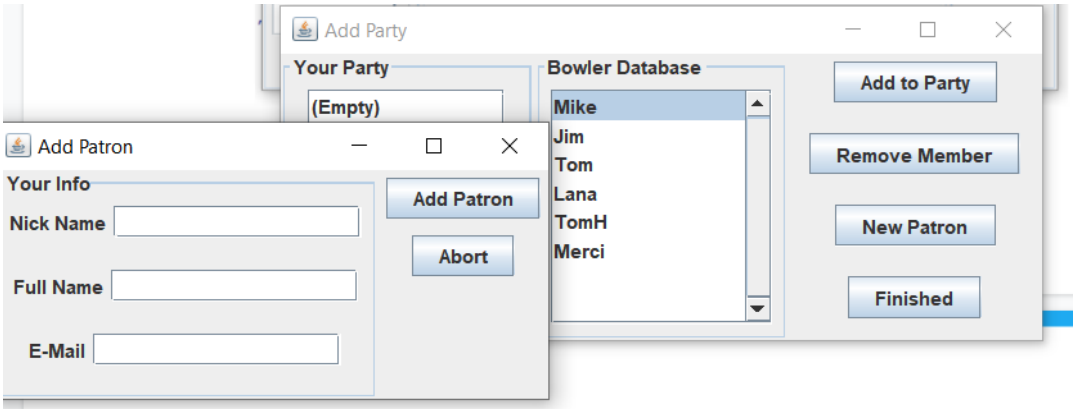
Bowling Management System is a Bowling Game Program written in Java.It is a virtual game that allows participants to enjoy the thrills of bowling from the comfort of their own homes(via laptop).The game simulates a number of aspects that add to the game's overall attractiveness.

Some features of the game are as follows :

- Control Desk : Any active lane's scores can be monitored by the control desk operator. The operator can view the score of a single scoring station or many scoring stations using a configurable display option.

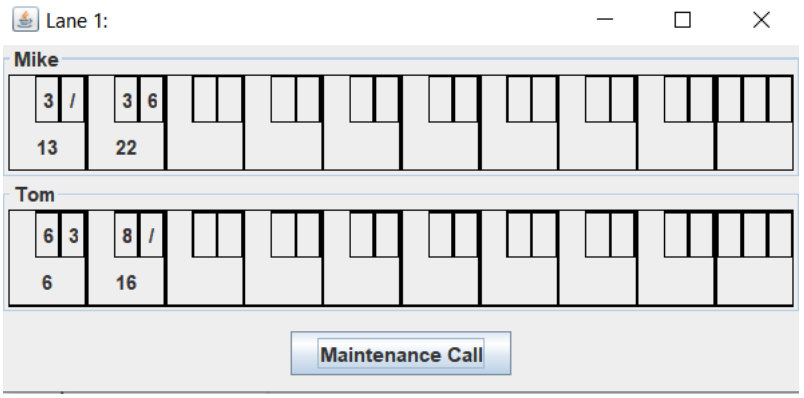


- Creating a new player : To play the game, a NewPatron must be created. The Bowlers database file is then updated to include this player.

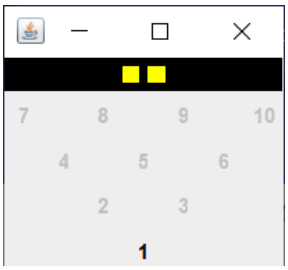


- Adding a new party : A group of bowlers can be recruited to a party, which is then assigned to one of the available lanes to begin a game. If all of the lanes are taken, the party is added to the Queue, which keeps track of the parties that have signed up but have yet to play.

- Viewing the Scoreboard : The scoreboard keeps track of how much each person in the group has earned after their turns. It employs the standard scoring method used in a typical game of bowling, i.e. for a strike : score = 10 + pins dropped on next 2 balls and for a spare: score = 10 + pins dropped on next ball.



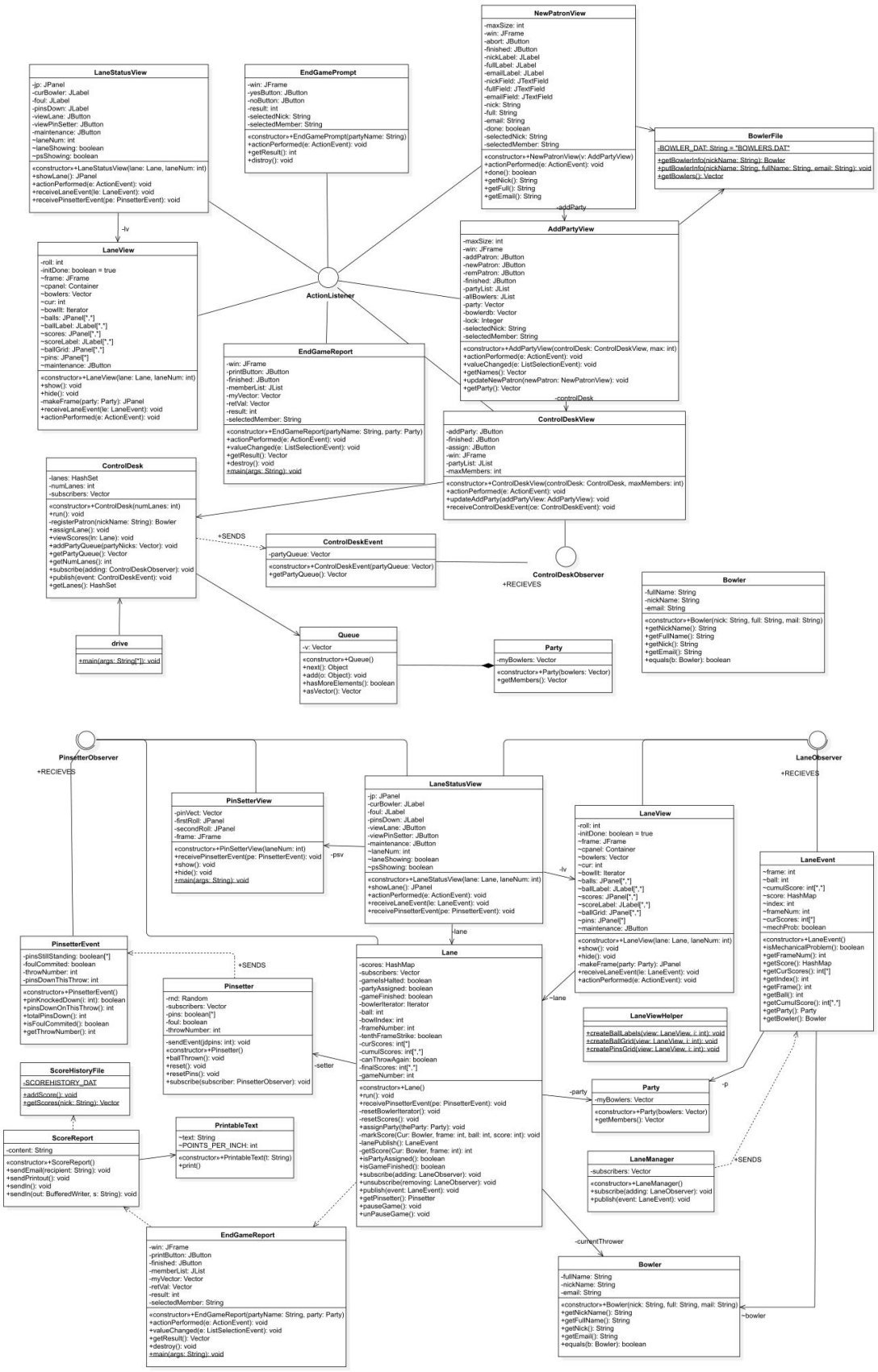
- Viewing the Pinsetter : A specific lane's pinsetter window, which mimics the pins dropped on each ball-throw, can also be watched. The pinsetter will re-rack the pins once two consecutive tosses are detected (put all ten down).



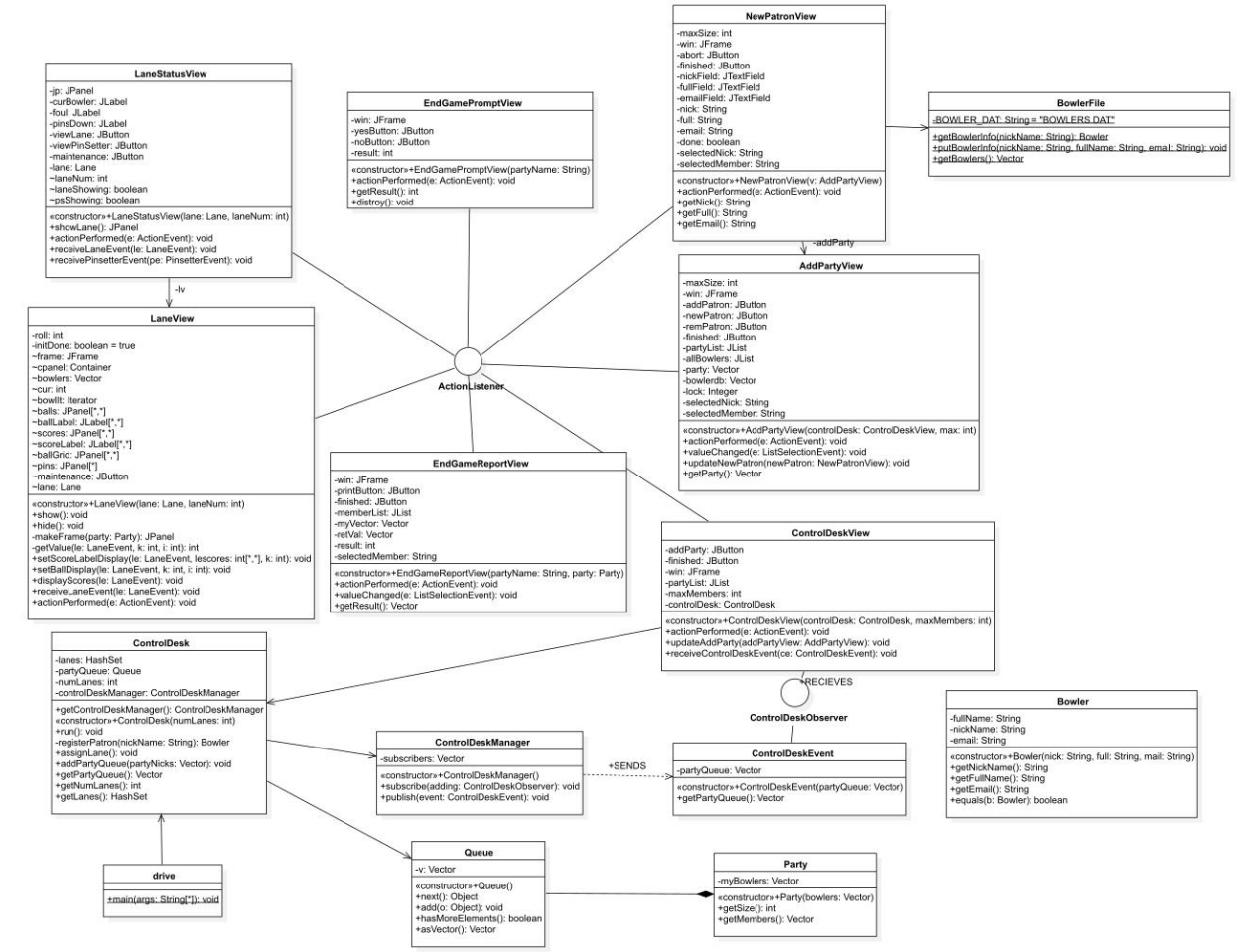
- Maintenance Call : This is simply a simulation of some maintenance work that needs to be done for a specific lane (ball not returned, pinsetter did not re-rack, etc.). Game play has been suspended while the lane is being rebuilt.

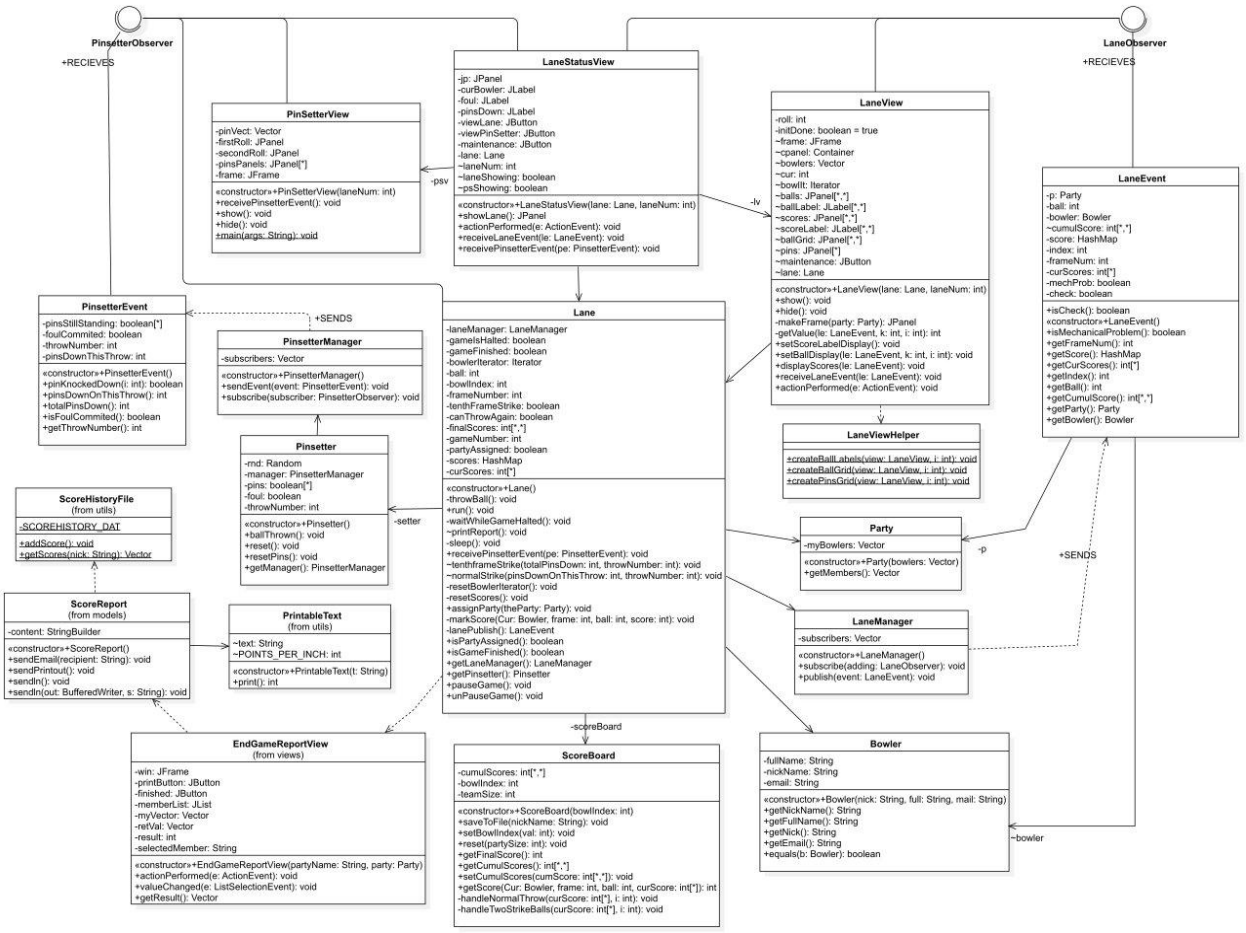
III. Class Diagrams:

We divided our class diagrams based on the abstraction we wanted to display such as one diagram displays the interaction of views and the other show



After Refactoring:

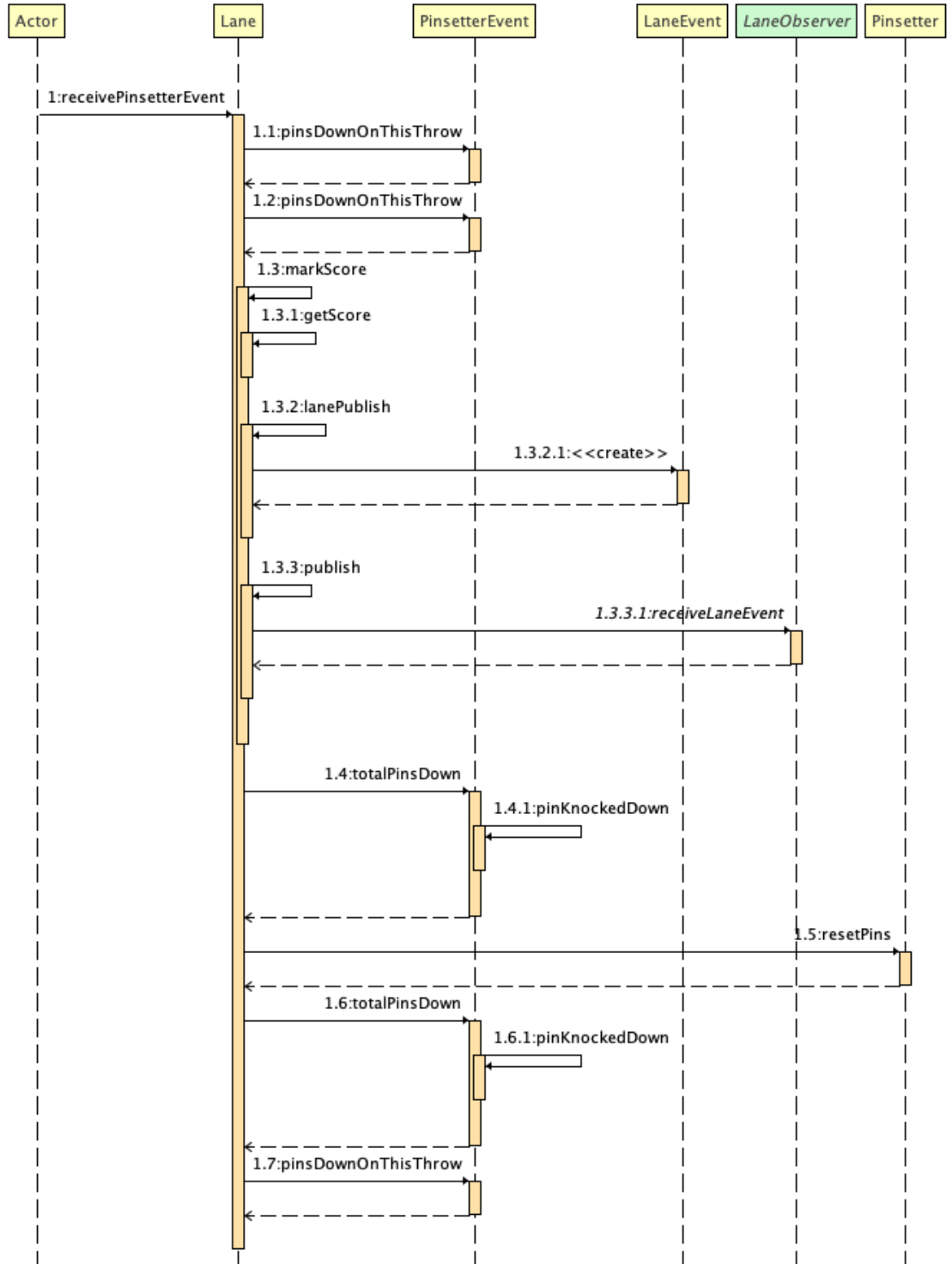




IV. Sequence Diagrams:

Diagram 1:

Before Refactoring:



After Refactoring:

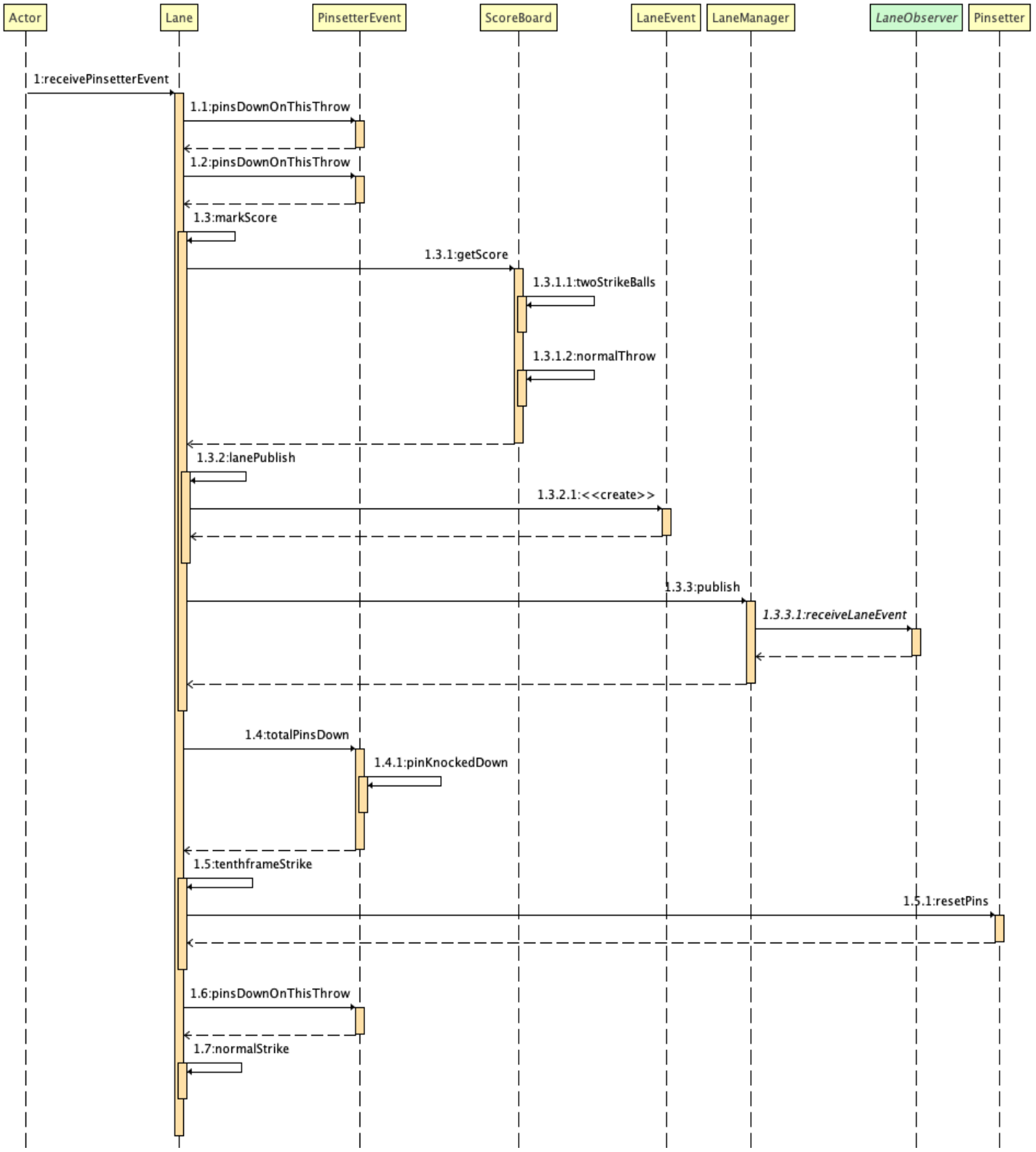
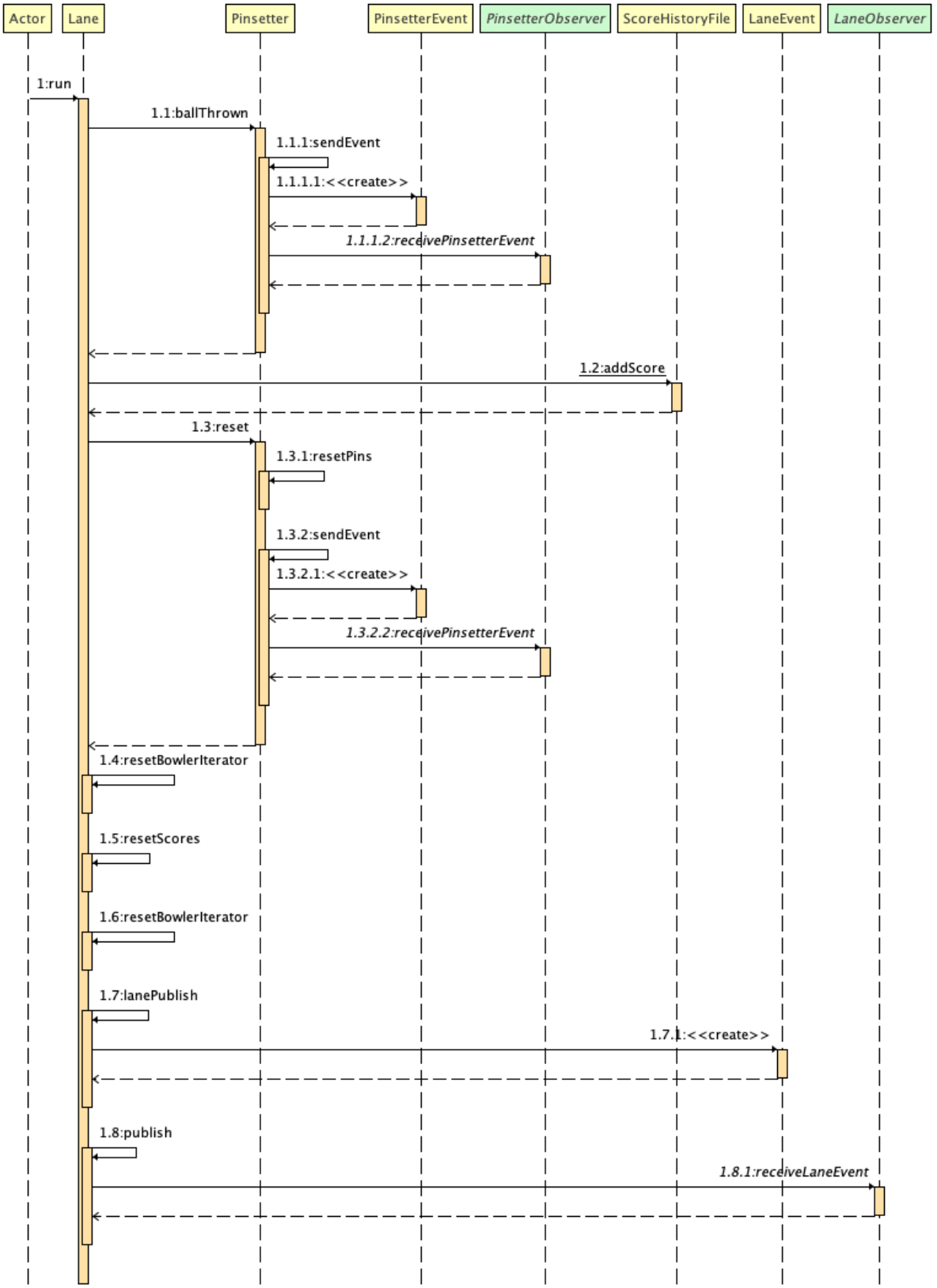
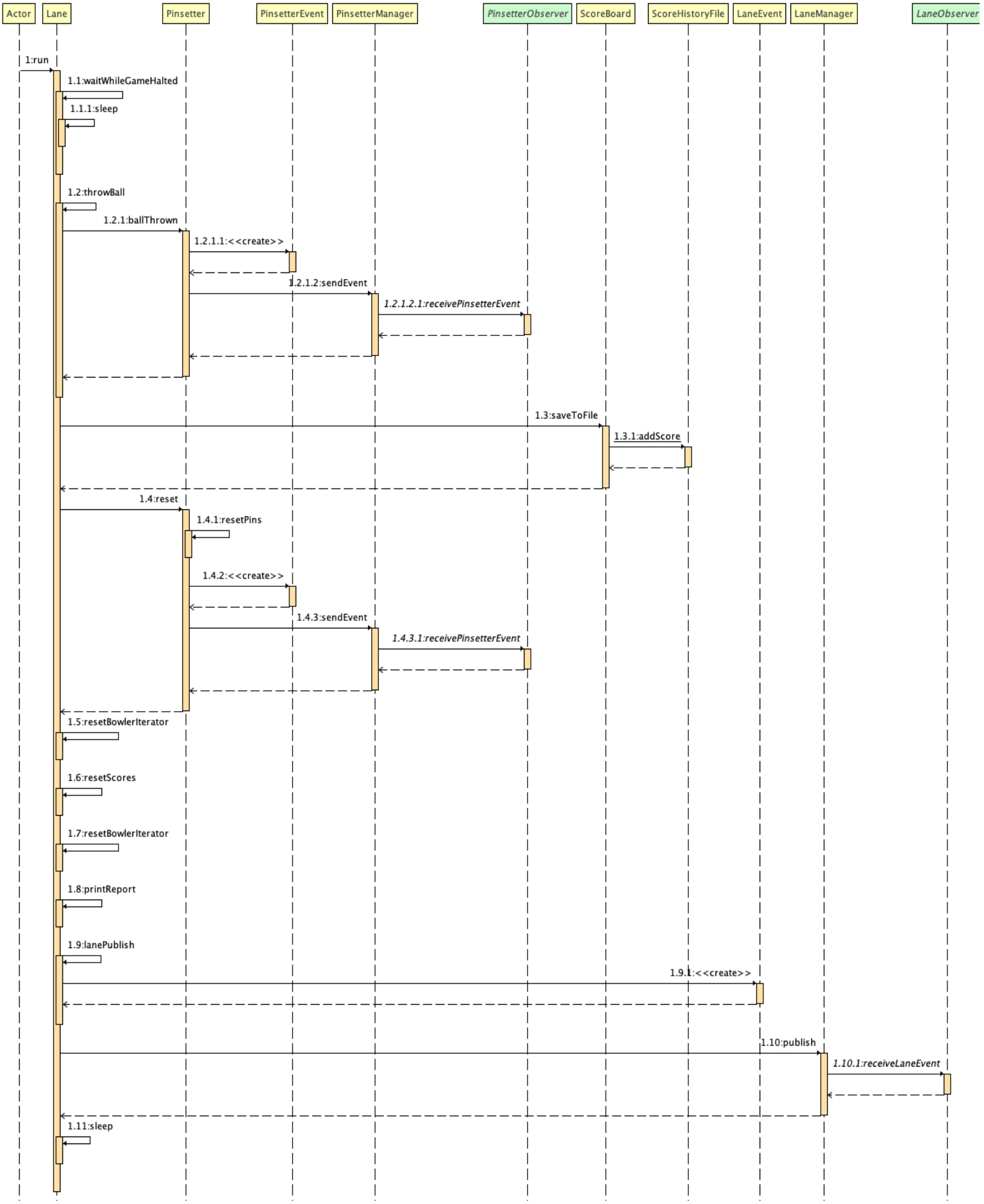


Diagram 2:

Before Refactoring:



After Refactoring:



V. Responsibilities of Each class :

There are a total of 29 files in the Bowling Management System codebase. Each file contains a set of classes and functions that aid in the simulation of the full game.

Following is a list of all the files and their corresponding characteristics:

Sr.N	Class Name	Attributes	Methods	Description
1.	AddPartyView	maxSize win addPatron remPatron finished partyList allBowlers Party Bowlerdb lock	actionPerformed() valueChanged() getNames() updateNewPatron() getParty()	This class is used for adding a new patron to party,removing a patron from a party,creating a new patron and returning the latest state of the party.It displays parties on the Control Desk.
2.	Alley	controlDesk	getControlDesk()	Creates an object of the ControlDesk
3.	Bowler	fullName nickName email	getNickName() getFullName() getNick() getEmail() equals()	It is a Model class which is used to create a new Bowler and stores the information of the Bowler.
4.	Bowlerfile		getBowlerInfo() putBowlerInfo() getBowlers()	Using this class we can add a new bowler and can get details of one or more bowlers.
5.	ControlDesk	Lanes partyQueue numLanes subscribers	run() registerPatron() assignLane() addPartyQueue() getPartyQueue() getNumLanes() subscribe() publish() getLanes()	It assigns a lane to a party as soon as it is formed. It put names of the parties in the waiting PartyQueue which then get displayed on the side pane of the controlDesk.
6.	ControlDeskEvent	partyQueue	getPartyQueue()	It represents the wait queue, containing party names to be displayed in the side panel of the Control Desk.
7..	ControlDeskView	addParty Finished Assign Win partyList maxMembers controldesk	actionPerformed() updateAddParty() receiveControlDeskEvent()	It displays the controldesk.

8.	Drive		main()	The game will start running from this file. We define maximum number of patrons that can play the Bowling game and maximum number of lanes that can be present in the ControlDesk. In drive. java , the object of the ControlDesk class is getting called. We send that object to the ControlDeskView class
9.	EndGamePrompt	win yesButton noButton result selectedNick selectedMember	actionPerformed() getResult() destroy()	It print the prompt when the game finishes. It displays a dialog box asking whether you want a party to continue playing in a lane or not. So it offers two buttons, Yes and No for the
10.	EndGameReport	win printButton finished memberList myVector retVal Result selectedMember	actionPerformed() valueChanged() getResult() destroy()	This class is called after the EndGamePrompt. This class asks the user to finish the game without printing the report of the game or to print the report of the party who finished the game with their scores and other details.
11.	Lane	Party setter scores subscribers gameIsHalted gameFinished bowlerIterator ball bowlIndex frameNumber tenthFrameStrike curScores cumulScores canThrowAgain finalScores gameNumber currentThrower	run() receivePinsetterEvent() resetBowlerIterator() resetScores() assignParty() markScore() lanePublish() getScore() isPartyAssigned() isGameFinished() subscribe() unsubscribe() publish() getPinsetter() pauseGame() unPauseGame()	This class is also based on threading.It implements the Thread interface. When this.start() is encountered, the thread gets created and the run() the function of the class starts running. As soon as this.start() is encountered, thread is created and run() function comes into play. Now if the party is assigned to a lane and the game is not finished, till the game is halted, the game waits(sleep is used).Else we check whether we have bowlers to hit a ball in a frame. If yes, then the next bowler in the queue can hit the ball. Now the ballThrown() of the Pinsetter class is called where it generates a random number which on some calculations decides which pin to be knocked down and if it resulted in a foul or not. Further, if the frame is not the ninth one, then the queue shifts to the next frame and the game begins again. As soon as the queue reaches the tenth frame, the game is finished, and we receive the prompt indicating this. This is done by calling the class EndGamePrompt. In the End Game Prompt window, we receive an option that whether we want our party to play again in the same lane or not. If yes, then the game begins again using the above explained run() function by resetting the scores. If no, then the game is stopped and a report is generated describing the scores of the party by calling the class

12.	LaneEvent	P Frame ball bowler cumulScore score index frameNum curScores mechProb	isMechanicalProblem() getFrameNum() getScore() getCurScores() getIndex() getFrame() getBall() getCumulScore() getParty() getBowler()	This class is called when the game gets over, or when the game is paused or when the game needs to be unpaused.
13.	LaneStatusView	jp curBowler foul pinsDown viewLane viewPinsetter maintenance psv lv lane laneNum laneShowing psShowing	showLane() actionPerformed() receiveLaneEvent() receivePinsetterEvent()	This class describes how a particular lane looks like and what all information it holds. The View Lane button displays the particular lane simulator displaying how the bowlers are bowling in all the 10 frames. The PinSetter button displays all the 10 pins and how they're being knocked down.
14.	LaneView	roll initDone frame cpanel bowlers cur bowlIt balls ballLabel Scores scoreLabel ballGrid pins maintenance lane	makeFrame() receiveLaneEvent() actionPerformed()	This class is used to render view GUI for the alley lanes
15.	NewPatronView	maxSize win bbort finished nickLabel fullLabel emailLabel nickField fullField emailField nick full Email done selectedNick selectedMember addParty	actionPerformed() done() getNick() getFull() getEmail()	This class is concerned with addition of new Patrons to the Database (via GUI).

16.	Party	myBowlers	getMembers()	This class declares and initialises a vector. It contains a getter which returns the vector.
17.	Pinsetter	rnd subscribers pins foul throwNumber sendEvent()	ballThrown() reset() resetPins() subscribe()	This class contains ballThrown() function which calculates what pins to knock down and when a foul occurs.
18.	PinsetterEvent	pinsStillStanding foulCommited throwNumber pinsDownThisThrow	pinKnockedDown() pinsDownOnThisThrow () totalPinsDown() isFoulComited() getThrowNumber()	This class is basically used to store the details of a pinsetter.
19.	PinsetterView	pinVect firstRoll secondRoll frame	receivePinsetterEvent() show() hide()	This class describes how the pinsetter of a lane looks like. The pins are represented by the number 1 to 10.
20.	Queue	Vector v	next() add() hasMoreElements() asVector()	This class is called to create the PartyQueue for a lane.
21.	Score	nick date score	getNickName() getDate() getScore() toString()	This class is used to store the scores of a bowler. It basically contains getters returning the name of the bowler, date and score
22.	ScoreHistoryFile		getScores()	This class is used to manage Score database

23.	ScoreReport	content	sendEmail() sendPrintout() sendln()	This class displays the final scores, previous scores by date. This class sends the scores via email to the bowlers.
24	LaneEventInterface			It is used to interface multiple classes
25	LaneObserver			It is used to interface multiple classes
26	LaneServer			It is used to interface multiple classes
27	PinSetterObserver			It is used to interface multiple classes
28	PrintableText	String text int POINTS_PER_INCH	int print(Graphics g, PageFormat pageFormat, int pageIndex)	This class displays graphical text on UI
29	ControlDeskObserver		void receiveControlDeskEvent	This class observe control desk events

VI. Original Design Analysis:

The original code appeared to be very complicated at first. However, upon closer inspection of the files, we discovered that all of the functionality was useful and, for the most part, nicely written given the people who developed them were students who had limited industrial experience. All we had to do was clean-up operations and move things around a bit to make everything fall into place.

Weaknesses of the Original Implementation(Code smells)

- **Long Methods**

  Some of the classes, such as Lane.java, are unreasonably long. It has the most McCabe cyclomatic complexity as well. As we did in our refactored code, these could easily be divided up and redistributed. The file sizes also vary greatly, as certain classes, such as Alley.java and drive.java, contain very little code while others have a lot.

- **Dead Code**

  A lot of the classes had dead code in the form of methods,variables or routes that were not being used or variables and functions that were not called anywhere. These add to the LOC.

- **Duplicate Code**

  This was also frequent in the project initially. Code for creation of buttons, panels, windows is repeated in all the View classes. Even code that performs the exact same functionality or performs the same computation like the conditions in LaneView.java

- **Long Parameter List**

  The LaneEvent.java file has a long parameter list, this is undesirable since it makes the code more complex.

```
public LaneEvent( Party pty, int theIndex, Bowler theBowler, int[][] theCumulScore, HashMap theScore, int theFrameNum, int[]
```

- **Feature Envy**

  This is a common occurrence in the code. Many class functions, such as ControlDesk was using a method to register bowlers called registerPatron which would have been better suited in BowlerFile class.

- **Indecent Exposure**

  All variables are private and have getter and setter functions which is good but there are some classes, such as LaneView, which expose its attributes.

- **Primitive Obsession**

  Objects and data types like the JButton are repeatedly written and rewritten. As a result, it becomes highly complicated. This can be resolved by using a Factory Creator pattern.

- **Lazy Classes**

  There were classes like Alley.java which were doing nothing and just delegating the calls to some other class.

- **Speculative Generality**

  There were classes in the codebase which were written for future use, just in case they are ever used like LaneServer.java interface and LaneEventInterface.java even though there was only one single event possible for Lane event.

- **Bloater/Large Class**

  Lane.java is perhaps the most problematic class that we ever dealt with as it seemed like a nightmare with all the different computations merged into one single class which raised the cyclomatic complexity a lot and reduced readability.

Following lies the class wise bugs/anti-patterns in the class files:

Classes/Interfaces	Description
LaneView.java	• Has long methods with high cyclomatic conditional complexity.
LaneStatusView.java	• Has a high conditional complexity function due to the method actionPerformed(ActionEvent e) given the high number of event listeners in the class. • Has repetitive code for making buttons and various panels.
Lane.java	• This is a God class and needs breaking down for easier readability and debugging. • The class's method is too long and has to be broken down based on the functions run(), and GetScore() function.
ControlDeskView.java	• Variables were declared as hash sets/int instead of var, unnecessary function assignLanes and event listener that was never used. • The class is unnecessarily big and needs to be broken down.
AddPartyView.java	• Has high conditional complexity in the function actionPerformed(ActionEvent e) due to the number of possibilities of the events in the class. • The feature updateNewPatron is highly based on the usage of the NewPatron class and should be moved there instead.
ControlDesk.java	• The functions addpartyQueue and getpartyQueue are highly based on the usage of the class Queue and should be moved there instead. • Because it transmits unneeded information to subscribers and reduces cohesion, the subscribe method should be relocated to a new class.
NewPatronView.java	• The nickPanel, fullPanel, and emailPanel functions all use the same panel startup code, which reduces performance and increases LOC unnecessarily.
I. NewPatronView.java II. EndgamePrompt.java III. AddPartyView.java IV. ControlDeskView.java V. LaneStatusView.java VI. ViewPinSetter	• Several procedures in all of these classes that use the same code to implement buttons and windows should be replaced with a single class whose object is called.

Strengths of the Original Implementation

- **High Cohesion:**

  Almost all the related functionalities were already encapsulated under single classes. The elements within the classes were directly related to the functionality that module is meant to provide. Because of which it was very easy to discover what code is related to the functionality. That made it easier to jump between different modules and keep track of all the code in our head.

- **Nomenclature of files and methods**

  The code is well-written, and the method names clearly describe the functionality. All files ending in View, for example, are responsible for the UI of the classes listed preceding 'View.' Variable names are also constant, however the first letter is capitalized in some places and not in others.

- **Comments**

  Comments were well mentioned on the important places in the code, making it very easier to understand what's going around in the code. They provided explanatory information about the source code, helped to understand the flow of the program and to make decisions while refactoring the design.

- **Requirements Satisfied:**

  As per the design document of the team , it is clear that even with limited industrial experience they were able to simulate the bowling alley system and generate the scores of the lane which itself is a great feat in itself.

Use of Design Patterns:

- **Use of Observer Pattern**

  The developers tried to implement the Observer pattern by using different Observer Interfaces and letting the classes implement their receive method after being notified from the Observee, but even this was not done properly and became congested with the other classes , like Lane.

VII. Refactoring

After analyzing the code we started with our refactoring process.

1. **Separating God Class into different classes:**
   We have divided Lane class into three classes namely LaneManager(for managing subscribers,publishing and subscribing logic moved to this class),ScoreBoard(for calculating cumulative Score and creating/ saving records in file) and remaining functionalities with Lane class itself.
2. **Enhancing Observer Pattern**
   Originally the Observer pattern had the code for subscribing the observers, and notifying the subscribers/listeners in their model classes. This violated the Single Responsibility Principle, so we thought to overhaul this by creating a Helper/Manager class that would take care of the notifying functionality for it. This way the model class Lane could carry out the Lane Operations without having to think about notifying and subscribing new observers. We extracted the methods using move and created a LaneManager class. Likewise other manager classes were created that enhanced the overall Observer Pattern. This is visible in the class diagram.
3. **Redundant Code:**
   Redundant Code is the repetition of a line or a chunk of code in the same file or, in some cases, the same local environment. People may think code duplication is fine, but it actually causes more problems for software than we might think. Even duplications of code with similar capabilities are considered duplications. A lot of UI elements were being created using the same lines of code over and over again. So we made a factory creator class called UIComponents that creates labels,panels, buttons on demand with customization.

```
public static JButton createFlowButton(String name, JPanel iterfacePanel, ActionListener listener) {
    JButton button = new JButton(name);
    button.addActionListener(listener);
    JPanel panel = createFlowPanel();
    panel.add(button);
    iterfacePanel.add(panel);
    return button;
}
```

```
abort = UiComponents.createFlowButton( name: "Abort",buttonPanel, listener: this);
```

4. **Repetitive manual array Copy:**

   Instead of writing a manual "for loop" in the file PinsetterEvent.java, a copy method should be used to copy an array as it helps prevent bugs.

5. **Catch Statements Were Empty:**
   There were some files in which the Catch statement was empty which can cause problems if the captured code does not print anything, making it impossible to debug the function. As a result, the proper errors were printed.

```
try {
    Thread.sleep( millis: 1);
} catch (Exception e) {
}
```

```
} catch (Exception e) {
    e.printStackTrace(System.out);}
```

6. **Unused Variables**
   Variables that were declared for a specific purpose but were not used later in the code. When utilized correctly, they can be quite effective, but when used incorrectly, they can reveal design flaws. In the file NewPatron.java, for example, there are three labels: nickLabel, fullLabel, and emailLabel which were not used in the code later and were removed hence.

7. **Dead code removal:**
   Assign button for assign lanes is removed from ControlDeskView as it is not used in the class. Alley Class is removed and is shifted to ControlDesk and drive Class.getScore() function is moved to a new class ScoreBoard.All button code and method getNames is removed from AddPartyView Class.

```
if (e.getSource().equals(assign)) {
    controlDesk.assignLane();
}
```

8. **Irrelevant Public Modifier in Interfaces**
   Many functions in Interfaces used the public modifier. However, because all functions in an interface are inherently public and abstract, there was no need to explicitly make them public. As a result, the public modifier was removed from these interfaces.
9. **Reducing Conditional Statements**
   Many of the if-else conditions contained redundant checks or were not designed in the most efficient manner. LaneView and LaneStatusView are two such examples. The same conditions were being checked again and again, such as being converted to a query and used instead of computing again and again. The conditions that couldn't be pre-computed were fragmented among multiple classes. As a result, the Cyclomatic Complexity of all of our files decreased.

10. **Depreciation enhancement**
    There were a lot of obsolete functions and variables in the code.In buttons and panels the attribute setVisible(value) of boolean type has replaced deprecated functions like .show() and .hide() .Also Instead of adding Strings , StringBuilders were used to improve performance.

11. **Moving Methods to appropriate classes**
    There were methods that didn't belong to a particular class like registerPatron in ControlDesk which operated on data from BowlingFile so we moved the method to the appropriate class.

12. **Packaging classes**
    The codebase was disorganized and it was really unclear what did what and which particular set of classes worked alike, so we packaged the classes based on its utility.

Analysis of Refactored Design

Separation of Concerns (SoC)

We have achieved this by restructuring our code in different packages and classes such that each section is concerned only about related sets of functionalities. This has led to an increase in the modularity of the program.

We have organized the whole program into 6 different packages, each package address a seperate need required for the execution of the program. Furthermore, we have partitioned a few classes such as Lane.java and LaneView.java into two classes each addressing a specific concern within the codebase.

Extensibility

We made it easy to introduce new modules, such as a new implementation for an existing interface, by modularizing the code. Because of high cohesion and modularization it will be easier to extend the functionalities of the program. High cohesion and better organization also enable developers to better understand the code and pre-plan the change that might be required in the future.

Reusability

By separating out the classes and functionalities such that each class performs related action we have increased the reusability of the code. For example, we have created a UiComponents which we used to create views and panels in multiple parts of the program.

Information hiding

One of the key principles of Object oriented programming is Data abstraction and hiding. We have simplified the representation of a number of functions and classes such that unrelated attributes of the methods are abstracted away from the function calling them.

Low Coupling

We began with 29 classes, which is quite a number. Such interdependencies were numerous, and we were able to eliminate them by making the parameters passed local and deleting the superfluous ones.

We continued to add additional classes to our list, as well as break down huge files like Lane into LaneManager and ScoreBoard . We made sure as much as possible that classes were self-contained and didn't require too many other dependencies apart from Lane class, as this would raise coupling to reduce complexity. We also made sure that functions were assigned to the appropriate locations so that they didn't have to interact or travel around as much.

High Cohesion

The term "cohesion" refers to the degree to which a class has a single, well-defined objective. We concentrated on how individual classes are constructed in this section. Classes that are highly coherent are considerably easier to sustain and modify less frequently.

Although the design implemented in the code was already very well balanced in terms of cohesion, that was coming at the cost of increased complexity , poor understandability and God classes. Such a design is difficult to extend further. So we solved the issue by separating out some of the functionalities into a number of classes, at the cost of low Cohesion.

Law of Demeter

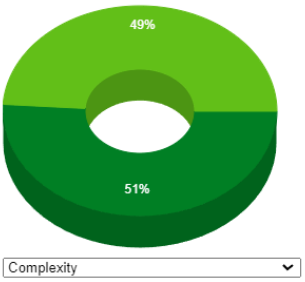
The Law of Demeter is a design strategy that focuses on developing OO software where we maintain the principle of least knowledge. This means that a class should talk to as few classes as possible or in technical terms should not allow access to a third class's methods by accessing the second class. This in-turn reduces the coupling between classes. There were instances in the codebase where the law of demeter was broken such as in LaneView we were directly accessing the Bowlers of a Party after retrieving the Party itself from a class called LaneEvent. This was increasing the coupling hence by breaking the fetching and doing it in two steps instead we were able to avoid this.

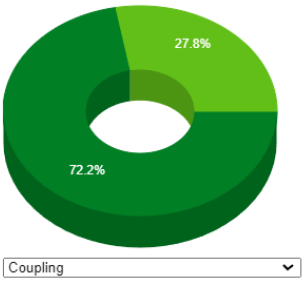
```
int numBowlers = le.getParty().getMembers().size();
```

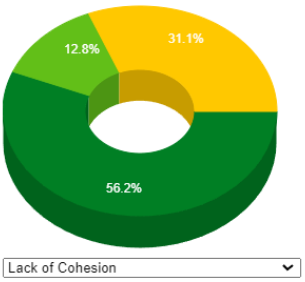
```
Party party=le.getParty();
int numBowlers=party.getSize();
```

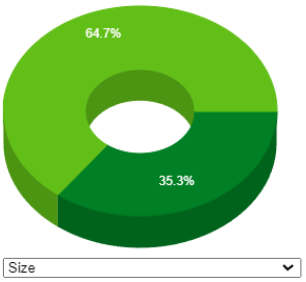
VIII. Metric Analysis

Metrics from CODE-MR









Name	CBO		RFC		WMC		LOC		CMLOC		NOF		NOM		LCOM		LCAM	
	old	new	old	new	old	new	old	new	old	new	old	new	old	new	old	new	old	new
AddPartyView	4	5	63	64	21	20	127	105	118	96	14	14	6	5	0.743	0.696	0.694	0.667
ControlDesk	7	7	40	41	22	19	68	59	63	54	4	4	11	9	0.714	0.625	0.779	0.667
ControlDeskView	6	9	67	62	8	7	87	73	81	67	7	6	4	4	0.619	0.667	0.625	0.625
EndGamePrompt	0	1	22	24	8	8	55	44	50	40	6	4	4	4	0.833	0.583	0.5	0.5
EndGameReport	2	3	36	36	12	9	79	60	71	52	8	8	5	4	0.75	0.708	0.633	0.6
Lane	10	11	76	89	87	48	227	161	208	142	18	18	17	20	0.854	0.846	0.782	0.74
LaneEvent	2	2	11	11	11	11	41	42	30	31	10	10	11	11	0.91	0.9	0.758	0.758
LaneStatusView	7	10	38	43	17	17	93	87	82	76	13	13	5	5	0.712	0.712	0.667	0.667
LaneView	4	5	47	61	31	28	140	98	124	82	15	15	6	10	0.88	0.925	0.694	0.633
NewPatronView	1	2	39	43	8	7	85	57	75	48	17	14	6	5	0.894	0.857	0.556	0.533
PinSetterView	1	2	22	11	11	13	111	43	106	37	4	5	4	5	0.667	0.5	0.6	0
Pinsetter	2	2	12	24	15	12	47	77	41	71	5	5	6	4	0.56	0.733	0.556	0.6
ScoreReport	4	4	29	31	13	13	76	68	74	66	1	1	5	5	0	0	0.567	0.567

METRICS FROM Metrics2

I. Cyclomatic Complexity

Indicates the complexity of a program. It is a numerical measure of the number of linearly independent pathways through the source code of a programme.

Before Refactoring:

Metric	Total	Mean	Std. ...	Maximum	Resource causing Maximum	Method
McCabe Cyclomatic Complexity (avg		2.319	4.062	38	/bowl-old/code/Lane.java	getScore
Lane.java		5.118	9.498	38	/bowl-old/code/Lane.java	getScore
Lane		5.118	9.498	38	/bowl-old/code/Lane.java	getScore
getScore	38					
run	19					
receivePinsetterEvent	12					
resetScores	3					
publish	3					
Lane	1					
resetBowlerIterator	1					
assignParty	1					
markScore	1					
lanePublish	1					
isPartyAssigned	1					
isGameFinished	1					
subscribe	1					
unsubscribe	1					
getPinsetter	1					
pauseGame	1					
unPauseGame	1					
LaneView.java		5.167	6.543	19	/bowl-old/code/LaneView.java	receiveLaneEvent
LaneView		5.167	6.543	19	/bowl-old/code/LaneView.java	receiveLaneEvent
receiveLaneEvent	19					
makeFrame	7					
actionPerformed	2					
LaneView	1					
show	1					
hide	1					

Methods of concern:

The functions **getScore**, **run** and **receivePinsetterEvent** in the **Lane.java** class. So we did a number of things to manage the complexity:

- We separated out a few functions like getScore from this class and created a new class, ScoreBoard, for calculating the scores.
- We reduced the complexity of run and receivePinsetter methods by dividing their responsibility into multiple methods.

The functions **recieveLaneEvent** in LaneView.java class. We managed the complexity of recieveLaneEvent by distributing its responsibilities into multiple functions.

Metric	Total	Mean	Std. ...	Maximum	Resource causing Maximum	Method
McCabe Cyclomatic Complexity (avg,		1.931	1.675	9	/bowl-refactor/refactor/src/models/Score...	getScore
models		2.258	2.102	9	/bowl-refactor/refactor/src/models/Score...	getScore
ScoreBoard.java		3.364	3.199	9	/bowl-refactor/refactor/src/models/Score...	getScore
Lane.java		2.19	2.061	9	/bowl-refactor/refactor/src/models/Lane.j...	run
Lane		2.19	2.061	9	/bowl-refactor/refactor/src/models/Lane.j...	run
run	9					
tenthframeStrike	7					
finishGame	3					
receivePinsetterEvent	3					
normalStrike	3					
resetScores	3					
throwBall	2					
waitWhileGameHalted	2					
sleep	2					
Lane	1					
printReport	1					
resetBowlerIterator	1					
assignParty	1					
markScore	1					
lanePublish	1					
isPartyAssigned	1					
isGameFinished	1					
getLaneManager	1					
getPinsetter	1					
pauseGame	1					
unPauseGame	1					

II. Number of Parameters:

This metric counts measures the number of procedure parameters (arguments). The more parameters a procedure has, the harder it is to write a new call to it.

Before Refactoring:

Metric	Total	Mean	Std. ...	Maximum	Resource causing Maximum	Method
Number of Parameters (avg/max per method)		0.723	1.131	9	/bowl-old/code/LaneEvent.java	LaneEvent
LaneEvent.java		0.818	2.587	9	/bowl-old/code/LaneEvent.java	LaneEvent
LaneEvent		0.818	2.587	9	/bowl-old/code/LaneEvent.java	LaneEvent
LaneEvent	9					
isMechanicalProblem	0					
getFrameNum	0					
getScore	0					
getCurScores	0					
getIndex	0					
getFrame	0					
getBall	0					
getCumulScore	0					
getParty	0					
getBowler	0					

To increase code readability and understandability, we removed the constructor's responsibility for object initialization and introduced a number of setters methods.

After Refactoring:

Metric	Total	Mean	Std. ...	Maximum	Resource causing Maximum	Method
McCabe Cyclomatic Complexity (avg/max per r		1.931	1.675	9	/bowl-refactor/refactor/src/models/Score...	getScore
Number of Parameters (avg/max per method)		0.832	0.992	4	/bowl-refactor/refactor/src/events/Pinsett...	PinsetterEvent
events		0.536	0.823	4	/bowl-refactor/refactor/src/events/Pinsett...	PinsetterEvent
PinsetterEvent.java		0.833	1.462	4	/bowl-refactor/refactor/src/events/Pinsett...	PinsetterEvent
ControlDeskEvent.java		0.5	0.5	1	/bowl-refactor/refactor/src/events/Control...	ControlDeskEvent
LaneEvent.java		0.45	0.497	1	/bowl-refactor/refactor/src/events/LaneEv...	setParty
LaneEvent		0.45	0.497	1	/bowl-refactor/refactor/src/events/LaneEv...	setParty
setParty	1					
setBall	1					
setBowler	1					
setCumulScore	1					
setScore	1					
setIndex	1					
setFrameNum	1					
setCurScores	1					
setMechProb	1					
isCheck	0					
LaneEvent	0					
isMechanicalProblem	0					
getFrameNum	0					
getScore	0					
getCurScores	0					
getIndex	0					
getBall	0					
getCumulScore	0					
getParty	0					

III. Nested block depth

The nested block depth metric measures the depth of conditional nesting in a method or module. Deeply nested conditional statements increase the conceptual complexity of the code and are more likely to be error-prone.

Before Refactoring:

Metric	Total	Mean	Std. ...	Maximum	Resource causing Maximum	Method
McCabe Cyclomatic Complexity (avg/max per r		2.319	4.062	38	/bowl-old/code/Lane.java	getScore
Number of Parameters (avg/max per method)		0.723	1.131	9	/bowl-old/code/LaneEvent.java	LaneEvent
Nested Block Depth (avg/max per method)		1.511	1.177	7	/bowl-old/code/Lane.java	run
Lane.java		2.176	2.065	7	/bowl-old/code/Lane.java	run
Lane		2.176	2.065	7	/bowl-old/code/Lane.java	run
run	7					
getScore	7					
receivePinsetterEvent	5					
resetScores	3					
publish	3					
Lane	1					
resetBowlerIterator	1					
assignParty	1					
markScore	1					
lanePublish	1					
isPartyAssigned	1					
isGameFinished	1					
subscribe	1					
unsubscribe	1					
getPinsetter	1					
pauseGame	1					
unPauseGame	1					

To improve these metrics we analyzed what part of the code could be logically fragmented that was being repeated and broken down into those methods, this however increases our Number of Method count Metric.

After Refactoring:

Metric	Total	Mean	Std. ...	Maximum	Resource causing Maximum	Method
McCabe Cyclomatic Complexity (avg/max per r		1.931	1.675	9	/bowl-refactor/refactor/src/models/Score...	getScore
Number of Parameters (avg/max per method)		0.832	0.992	4	/bowl-refactor/refactor/src/events/Pinsett...	PinsetterEvent
Nested Block Depth (avg/max per method)		1.578	0.881	5	/bowl-refactor/refactor/src/models/Lane.j...	run
models		1.71	0.957	5	/bowl-refactor/refactor/src/models/Lane.j...	run
Lane.java		1.714	1.03	5	/bowl-refactor/refactor/src/models/Lane.j...	run
Lane		1.714	1.03	5	/bowl-refactor/refactor/src/models/Lane.j...	run
run	5					
receivePinsetterEvent	3					
tenthframeStrike	3					
resetScores	3					
throwBall	2					
finishGame	2					
waitWhileGameHalted	2					
sleep	2					
normalStrike	2					
Lane	1					
printReport	1					
resetBowlerIterator	1					
assignParty	1					
markScore	1					
lanePublish	1					
isPartyAssigned	1					
isGameFinished	1					
getLaneManager	1					
getPinsetter	1					

IV. Weighted Methods per class:

The WMC metric is defined as the sum of complexities of all methods declared in a class. This metric is a good indicator how much effort will be necessary to maintain and develop a particular class.

Before Refactoring:

Metric	Total	Mean	Std. ...	Maximum	Resource causing Maximum
Weighted methods per Class (avg/max per type)	327	11.2...	15.991	87	/bowl-old/code/Lane.java
Lane.java	87	87	0	87	/bowl-old/code/Lane.java
LaneView.java	31	31	0	31	/bowl-old/code/LaneView.java
ControlDesk.java	22	22	0	22	/bowl-old/code/ControlDesk.java
AddPartyView.java	21	21	0	21	/bowl-old/code/AddPartyView.java
LaneStatusView.java	17	17	0	17	/bowl-old/code/LaneStatusView.java

To improve this metric for classes like Lane, LaneView and ControlDesk we sought out to separate and fragment the different execution paths to bring some closure among the methods.

After Refactoring

Weighted methods per Class (avg/max per type)	46	46	0	46	/bowl-refactor/refactor/src/models/Lane.j...	
Lane	46					

Weighted methods per Class (avg/max per type)	28	28	0	28	/bowl-refactor/refactor/src/views/LaneVie...	
LaneView	28					
Weighted methods per Class (avg/max per type)	7	7	0	7	/bowl-refactor/refactor/src/views/Control...	
ControlDeskView	7					

V. Method lines of code

Lines of code (LOC), is a software metric used to measure the size of a computer program by counting the number of lines in the text of the program's source code.

Before Refactoring:

Method Lines of Code (avg/max per method)	236	13.8...	24.98	88	/bowl-old/code/Lane.java	getScore
Lane	236	13.8...	24.98	88	/bowl-old/code/Lane.java	getScore
getScore	88					
run	72					
receivePinsetterEvent	25					
resetScores	10					
Lane	8					
assignParty	8					
markScore	7					
publish	6					
lanePublish	2					
pauseGame	2					
unPauseGame	2					
resetBowlerIterator	1					
isPartyAssigned	1					
isGameFinished	1					
subscribe	1					
unsubscribe	1					
getPinsetter	1					

After Refactoring:

Method Lines of Code (avg/max per method)	136	6.476	6.185	27	/bowl-refactor/refactor/src/models/Lane.j...	run
Lane	136	6.476	6.185	27	/bowl-refactor/refactor/src/models/Lane.j...	run
run	27					
finishGame	14					
tenthframeStrike	12					
lanePublish	11					
resetScores	10					
Lane	9					
throwBall	8					
receivePinsetterEvent	8					
assignParty	8					
markScore	7					
sleep	5					
normalStrike	5					
waitWhileGameHalted	3					
pauseGame	2					
unPauseGame	2					
resetBowlerIterator	1					
isPartyAssigned	1					
isGameFinished	1					
getLaneManager	1					
getPinsetter	1					
printReport	0					

VI. Depth of Inheritance

The Depth of Inheritance Tree (DIT) metric provides for each class a measure of the inheritance levels from the object hierarchy top.
Since In our codebase the number of classes are small there weren't much complex inheritance design we could leverage, so we sought out to preserve the existing metrics.

Depth of Inheritance Tree (avg/max per type)		0.897	0.48	2	/bowl-old/code/Lane.java
Lane.java		2	0	2	/bowl-old/code/Lane.java
ControlDesk.java		2	0	2	/bowl-old/code/ControlDesk.java
PinsetterEvent.java		1	0	1	/bowl-old/code/PinsetterEvent.java
drive.java		1	0	1	/bowl-old/code/drive.java
PrintableText.java		1	0	1	/bowl-old/code/PrintableText.java
EndGamePrompt.java		1	0	1	/bowl-old/code/EndGamePrompt.java
Queue.java		1	0	1	/bowl-old/code/Queue.java
PinSetterView.java		1	0	1	/bowl-old/code/PinSetterView.java
BowlerFile.java		1	0	1	/bowl-old/code/BowlerFile.java
Pinsetter.java		1	0	1	/bowl-old/code/Pinsetter.java
EndGameReport.java		1	0	1	/bowl-old/code/EndGameReport.java
Alley.java		1	0	1	/bowl-old/code/Alley.java
Score.java		1	0	1	/bowl-old/code/Score.java
ControlDeskEvent.java		1	0	1	/bowl-old/code/ControlDeskEvent.java
Party.java		1	0	1	/bowl-old/code/Party.java
ScoreReport.java		1	0	1	/bowl-old/code/ScoreReport.java
LaneView.java		1	0	1	/bowl-old/code/LaneView.java
Bowler.java		1	0	1	/bowl-old/code/Bowler.java
LaneStatusView.java		1	0	1	/bowl-old/code/LaneStatusView.java
AddPartyView.java		1	0	1	/bowl-old/code/AddPartyView.java
ControlDeskView.java		1	0	1	/bowl-old/code/ControlDeskView.java
NewPatronView.java		1	0	1	/bowl-old/code/NewPatronView.java

VII. Lack of Cohesion of Methods:

The Lack of Cohesion in Methods metric is a measure for the number of not connected method pairs in a class representing independent parts having no cohesion.

Before Refactoring:

LaneEvent.java		0.91	0	0.91	/bowl-old/code/LaneEvent.java
NewPatronView.java		0.894	0	0.894	/bowl-old/code/NewPatronView.java
LaneView.java		0.88	0	0.88	/bowl-old/code/LaneView.java
Lane.java		0.851	0	0.851	/bowl-old/code/Lane.java
EndGamePrompt.java		0.833	0	0.833	/bowl-old/code/EndGamePrompt.java
PinsetterEvent.java		0.75	0	0.75	/bowl-old/code/PinsetterEvent.java
AddPartyView.java		0.729	0	0.729	/bowl-old/code/AddPartyView.java
EndGameReport.java		0.719	0	0.719	/bowl-old/code/EndGameReport.java
ControlDesk.java		0.714	0	0.714	/bowl-old/code/ControlDesk.java
LaneStatusView.java		0.712	0	0.712	/bowl-old/code/LaneStatusView.java
PinSetterView.java		0.667	0	0.667	/bowl-old/code/PinSetterView.java
ControlDeskView.java		0.619	0	0.619	/bowl-old/code/ControlDeskView.java
Pinsetter.java		0.56	0	0.56	/bowl-old/code/Pinsetter.java
Bowler.java		0.533	0	0.533	/bowl-old/code/Bowler.java
Score.java		0.5	0	0.5	/bowl-old/code/Score.java

After Refactoring:

Lack of Cohesion of Methods (avg/max per type)		0.939	0	0.939	/bowl-refactor/refactor/src/events/LaneEv...
LaneEvent	0.939				

The increased value is due to the additional number of getters and setters which were used to reduce the parameters in a method.

Lack of Cohesion of Methods (avg/max per type)		0.857	0	0.857	/bowl-refactor/refactor/src/views/NewPatr...
NewPatronView	0.857				
Lack of Cohesion of Methods (avg/max per type)		0.849	0	0.849	/bowl-refactor/refactor/src/models/Lane.j...
Lane	0.849				

VIII. Number of classes:

There were 29 classes in total, all of which were bundled into a single package, we grouped together similar classes inside a single package. This helps to give the program a better structure and organization. By virtue of how the code presents itself, developers are nudged to make smarter choices about where the actual system boundaries lie.

Before Refactoring:

Metric	Total	Mean	Std. ...	Maximum	Resource causing Maximum
Number of Classes	29				
PinsetterEvent.java	1				
drive.java	1				

After Refactoring:

Number of Classes (avg/max per packageFragment)	32	4.571	2.718	9	/bowl-refactor/refactor/src/views
views	9				
models	8				
utils	5				
events	3				
managers	3				
observers	3				
(default package)	1				