Software Engineering

Bowling Alley Simulation Refactoring Report

Team Members:

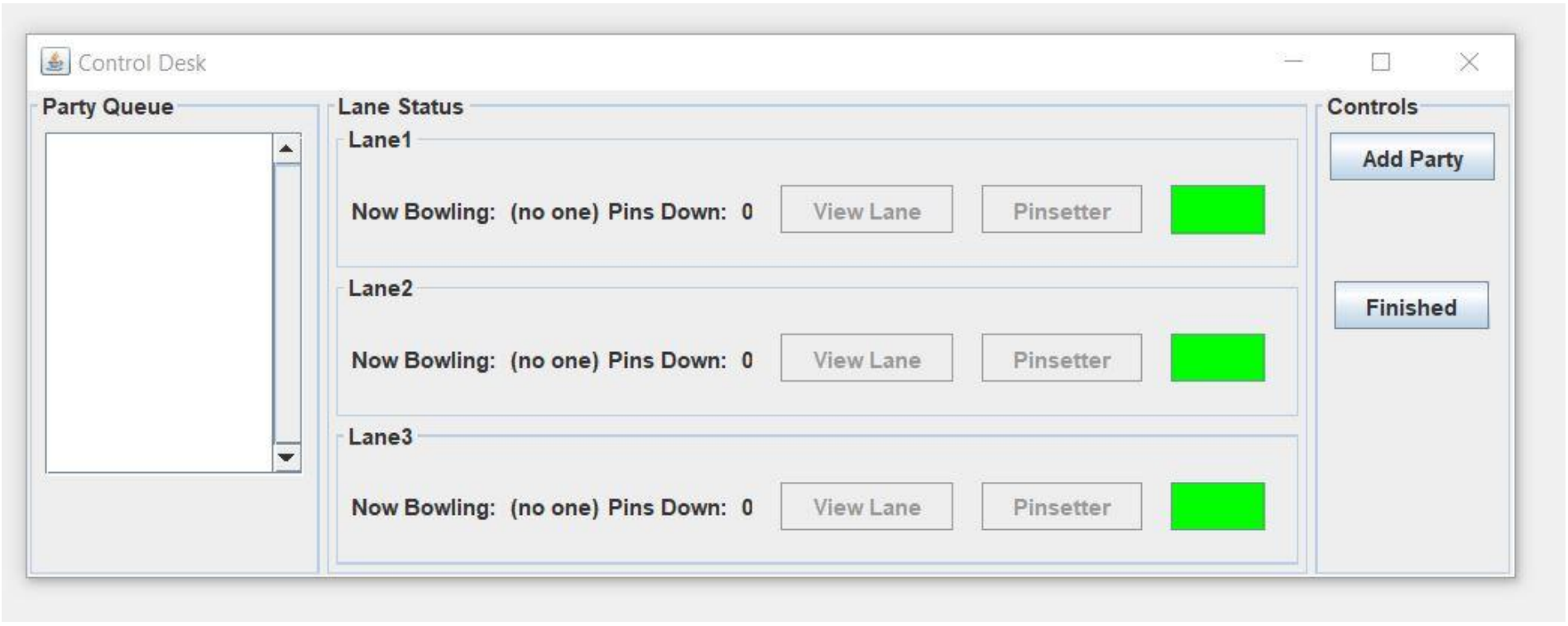
Name	Roll Number	Role	Effort (number of hours)
Vishal Pandey	2021201070	Refactoring and class,sequence diagrams	32 hrs
Padam Prakash	2021201071	Refactoring and Report writing	30 hrs
Sourav Kumar Singh	2021201072	Refactoring and Report Writing	30 hrs
Ankit Parashar	2020201043	Refactoring	26 hrs

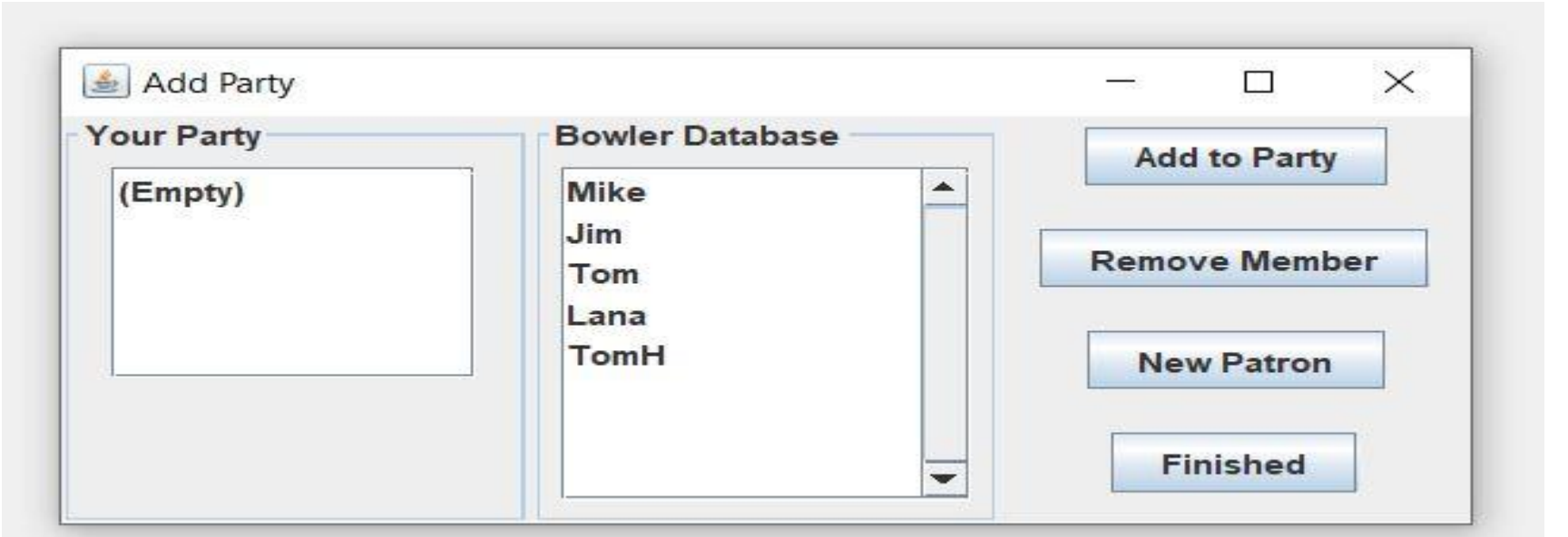
Introduction

In this project we have analyzed and refactor an existing software system- Bowling Alley Simulation. Our team has performed reverse engineering the design from existing code and documentation and proposed the refactoring to improve the program's structure for future maintenance and evolution.

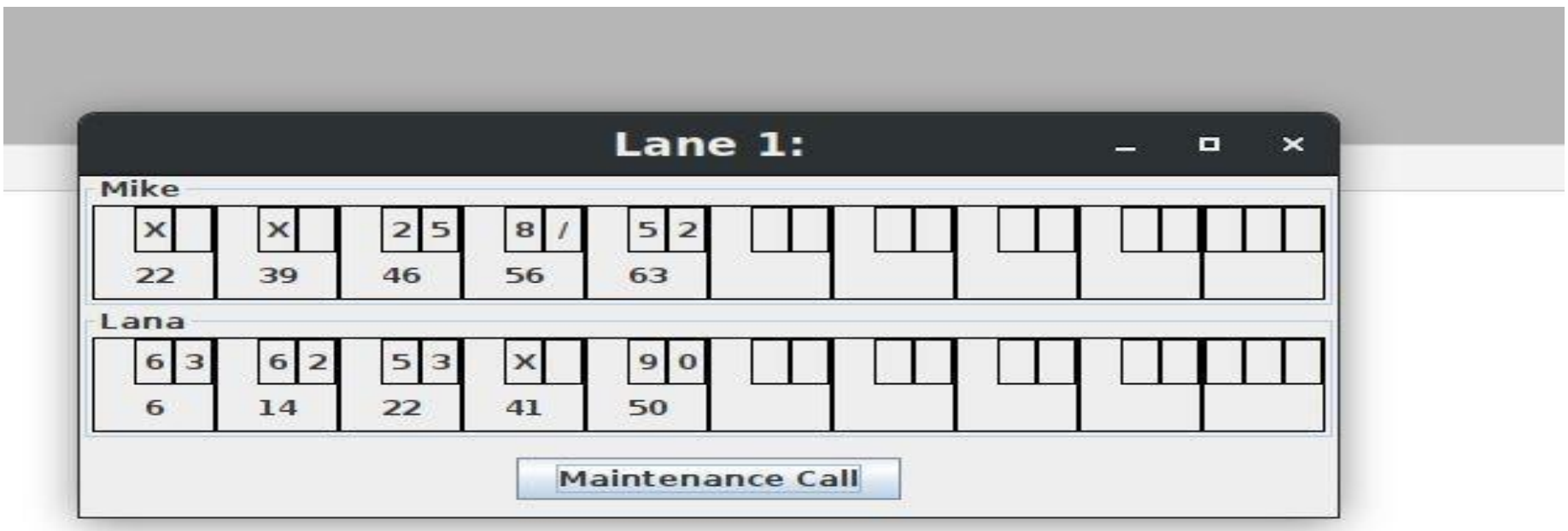
Overview

A bowling establishment is commonly referred to as a bowling alley. A bowling alley is composed of a number of bowling lanes.The product is a simulator of a bowling game that allows users to manage a virtual Bowling Alley. It has been written entirely in Java. It contains a java application that allows for the management of a bowling alley. Within the application, a collection of lanes can be monitored through a control desk, parties of bowlers can be assigned to the lane, the configuration of the pinsetter can be viewed, and the bowler's scores can be printed at the completion of the game.The Control Desk, which monitors the number of frames completed by each bowler. These interfaces make it more interactive. There are different windows and buttons too.





The control desk showing a game in progress:



The Scoring station, which is a dynamic-display of the game and pins knocked over by players in a party:

This is the pinsetter interface, which communicates to the scoring station the pins that are left standing after a bowler has completed a throw

Metrics were gathered for this project, and along with visual code and documentation inspection potential areas for refactoring were determined. For our refactoring, we used the plugin, **CodeMR** and **metrics2**, to analyze metrics of the code.

For maintaining a good metric score we modified the classes and functions. Overall, there are about 25 metrics in CodeMR (COUPLING,COMPLEXITY, LACK OF COHESION,SIZE,LOC,COMPLEXITY,COUPLING, LACK OF COHESION,SIZE being some of them).

Responsibilities of Major classes:

S.No	*Class Name*	*Responsibility*
1.	AddPartyView	It creates a popup window with the option to add or remove a Bowler 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.
2.	Alley	It Creates an object of the ControlDesk. It Initializes the ControlDesk with a given number of lanes.
3.	Bowler	It contains all the bowler's info.
4.	Bowlerfile	It provides following functionalities - fetching / updating information of bowlers, adding / removing bowlers.
5.	ControlDesk	It assigns a lane to a party as soon as it is formed. It puts names of the parties in the waiting PartyQueue which then get displayed on the side pane of the controlDesk.
6.	ControlDeskEvent	It represents the wait queue, containing party names to be displayed in the side panel of the Control Desk. It carries out functionalities of the program specified by AddPartyView and ControlDeskView.
7..	ControlDeskView	It displays the controldesk. Creates a view, provides us with an UI to add a party or to finish the game. It also provides us with the view of waiting parties and the current lanes assigned.
8.	Drive	It Starts game by creating an Alley. The game will start running from this file. We define the 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.Then the object is sent to the ControlDeskView class.
9.	EndGamePrompt	It 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. 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 user to click and answer.
10.	EndGameReport	It Provides option to print the report or not using a popup window.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	It 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 It works on threading.It implements Thread interface. When this.start() is encountered, the thread gets created and the run() function of the class starts running. As soon as this.start() is encountered, a thread is created and the run() function comes into existence. 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. A report is generated in the end describing the scores of the party by calling the class EndGameReport. Score report is also generated by calling the ScoreReport class.

12	LaneEvent	It Holds the values which define a lane like the frame number, current bowler, throw number and all. This class is called when the game gets over, or when the game is paused or when the game needs to be unpaused. (in refactored code we removed this class)
13	LaneStatusView	This class describes how a particular lane looks like and what all information it holds. It Creates the view shown in the center of the ControlDesk where things like the current bowler in each lane, lanes, pins down etc are shown. It also provides us with an options to see pins status
14	LaneView.java	It Creates view for the number of pins down in each throw for each player. It has long methods with high conditional complexity.
15	NewPatronView	It Creates view which lets us input the details about the new patron we are registering.
16	Party	This class declares and initializes a vector. It contains a getter which returns the vector.
17	Pinsetter	Simulates the dropping of pins in the lane by updating the states of each pin. It calculates what pins to knock down and when a foul occurs.
18	PinsetterEvent	It is used to store the details of a pinsetter. check if a pin has been knocked down, the total number of pins down for the pinsetter that generated the event,if a foul was committed on the lane. (in refactored code we removed this class)
19	PinsetterView	It Creates view which shows the status of pins so we can see what's going on the Lane. It describes how the pinsetter of a lane looks like.
20	Score	This class is used to store the scores of a bowler. It Sets the scores for the players in a game. It contains a function returning the name of the bowler, date and score.
21	ScoreReport	It generates the report for each player at the end of a game and prints/emails it.This class displays the final scores, previous scores by date.

Original Code Narrative

Weakness of the existing code:

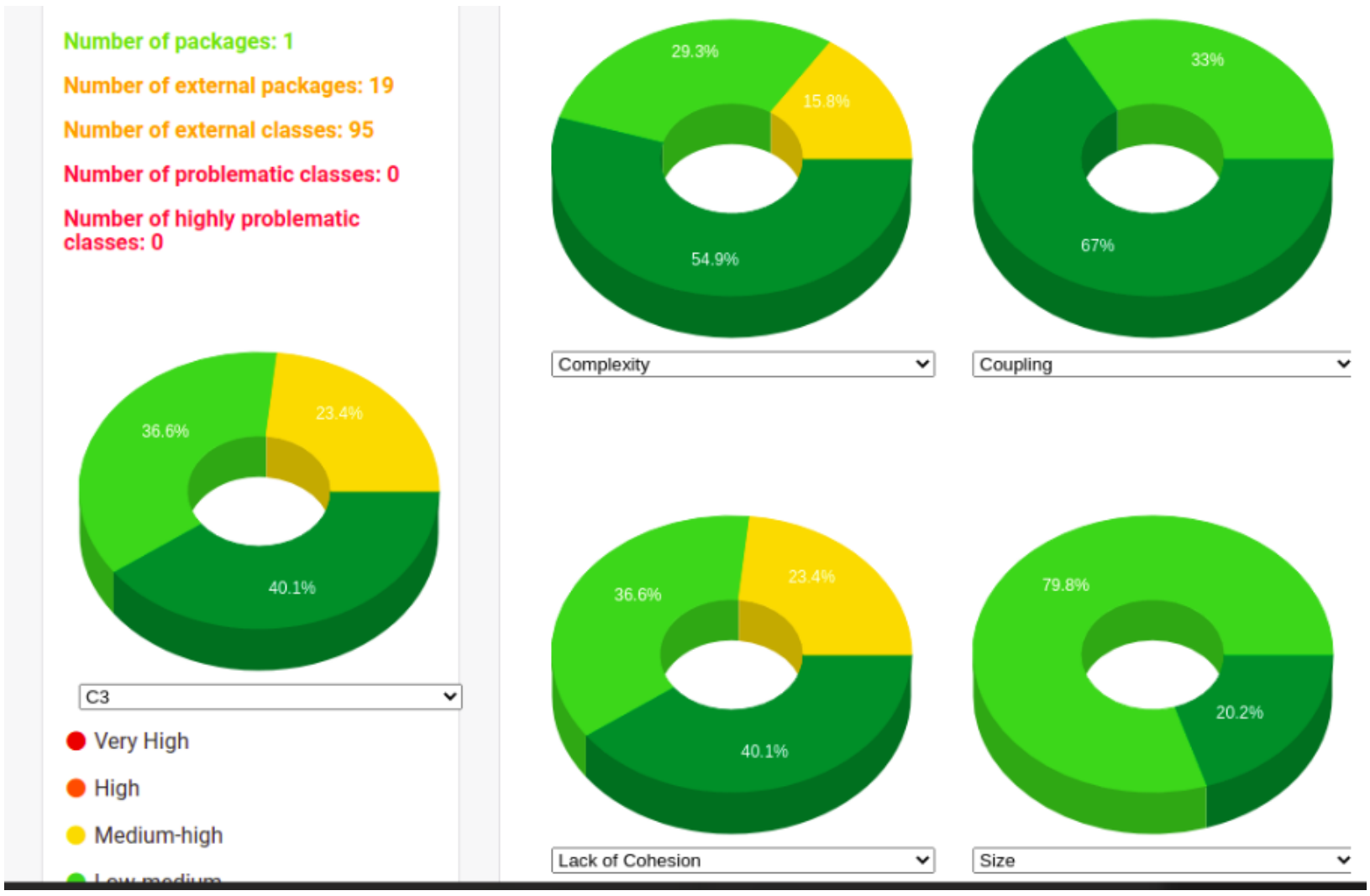
Original design suffers from some poor code design.There were several instances of redundant code, dead code which could have been easily replaced with a new code.Code suffered from poor score of Cohesion and coupling.Similarly other metrics were in yellow color signifying poor score. Number of packages is low.

Strengths of the existing code

The code works properly for all the functionality and corner cases. For some classes coupling is low. Among some related classes, Strong cohesion exists. All tasks are well defined. Identifier names are relevant. The code has been written consistently and the method names communicate their functionality well. The size of the code files has been kept low and optimum. The model classes are highly independent. All functions are distributed into relevant classes. Comments were very consistent throughout the code.

Original design

Some Metrics distribution of original code.



Design analysis among classes

- *Coupling and Cohesion*

For some classes coupling is low. Among some related classes, Strong cohesion exists.

- *Parameters passing*

Some classes do pass too many parameters. For example, LaneEvent constructor takes 8 parameters.

- *Scope Specifier*

Almost all variables are private.

Design analysis within classes

- *Dead Code*

Dead code is poor practice in coding. As these codes only contribute to increase in line of code. These codes were not being used anywhere.

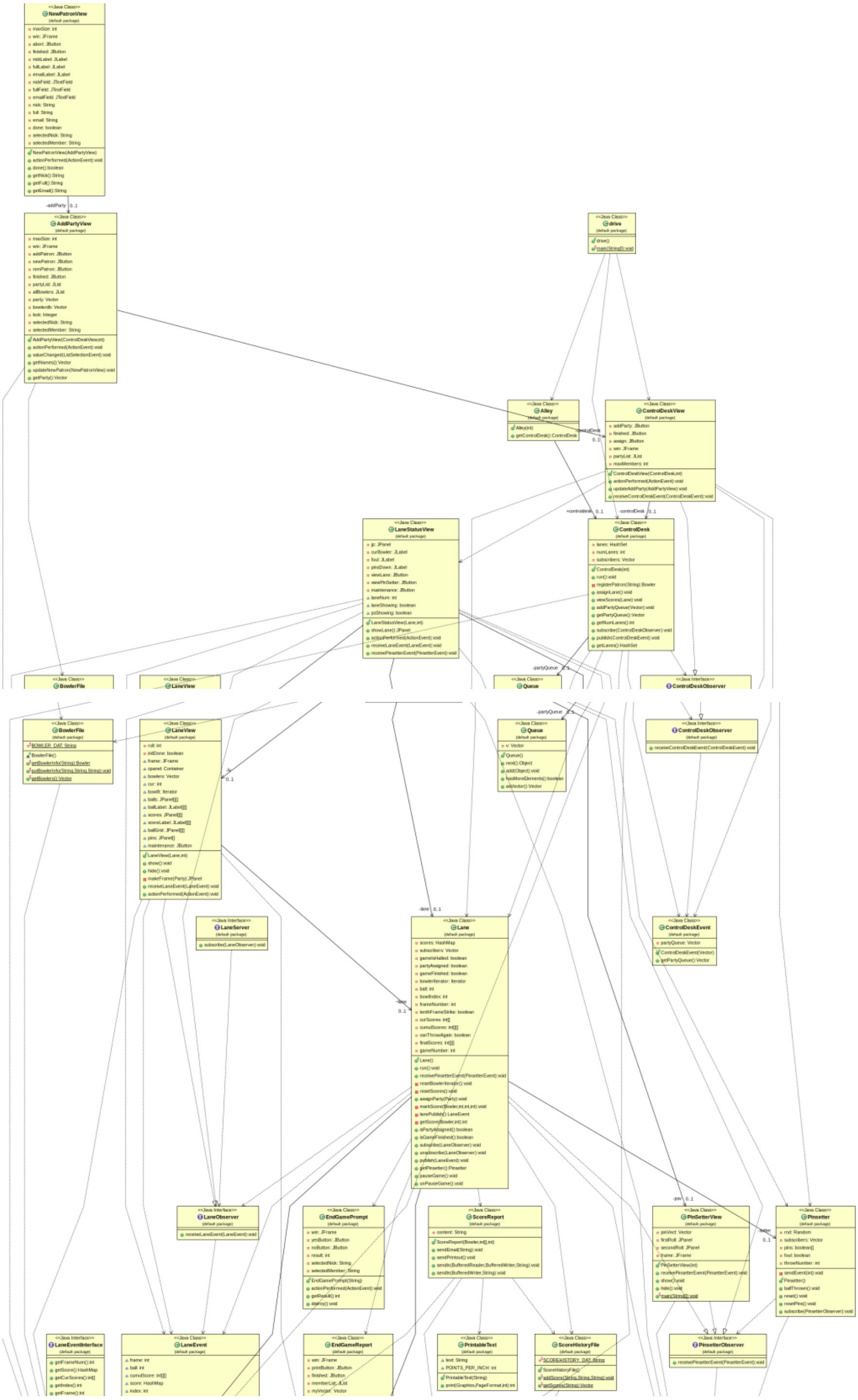
- *Repetitive Code*

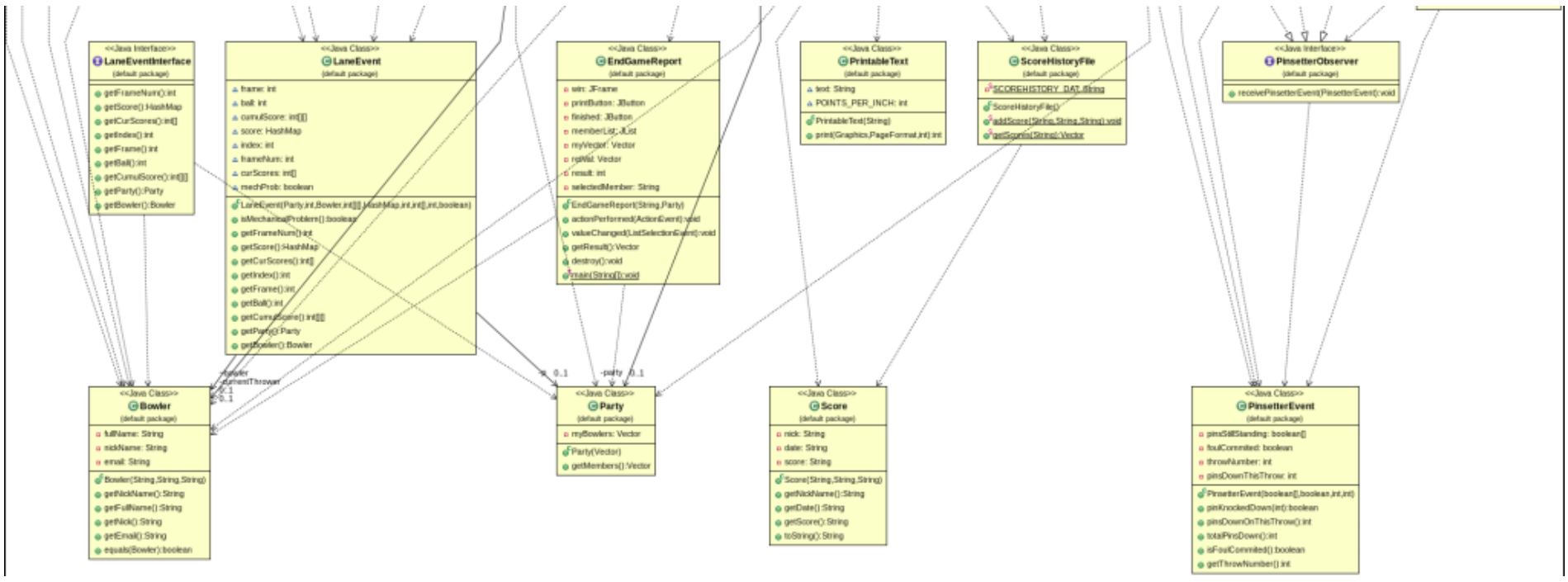
In many places the same code was written to provide some specific functionality.For example Code for creation of buttons, panels, windows is repeated everywhere. These codes need to be put in a function.

- *Cyclomatic Complexity*

Some functions include many if-else conditions which makes code unnecessarily lengthy.Conditional statements should be made precise. Lane.java has the highest cyclomatic complexity.

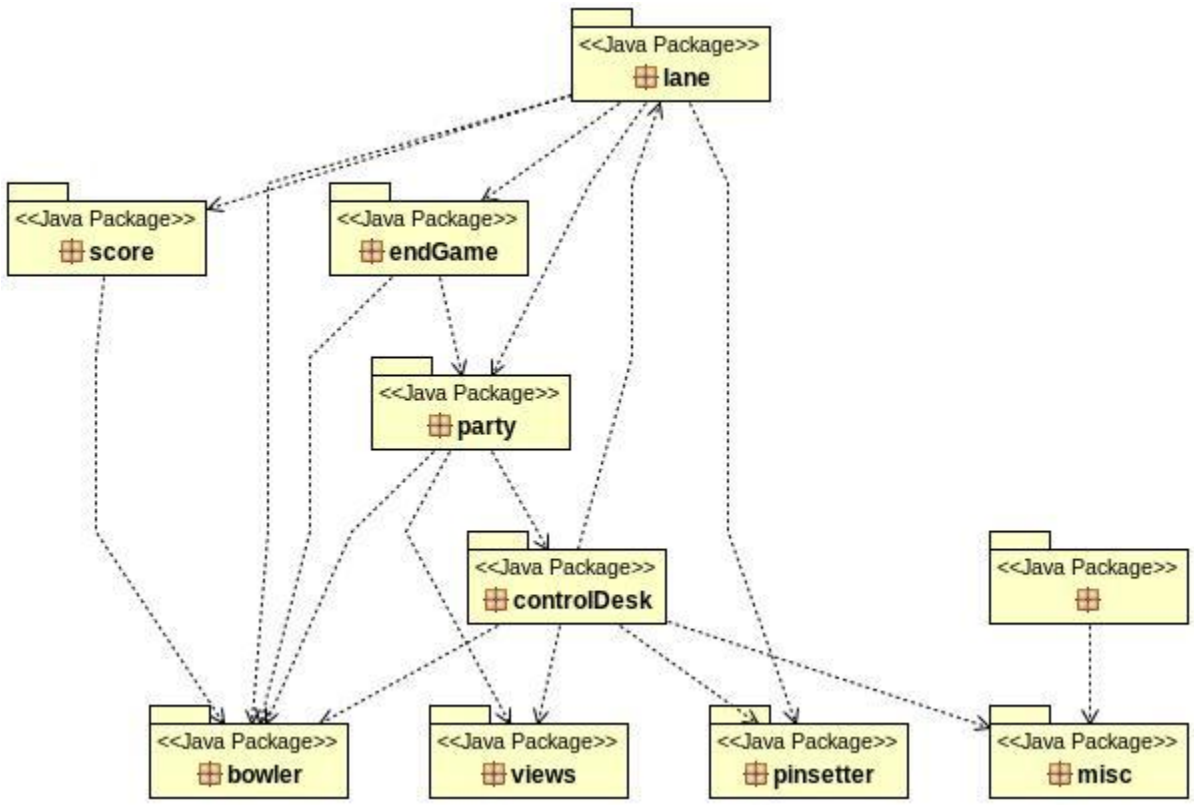
Original Class Diagram:

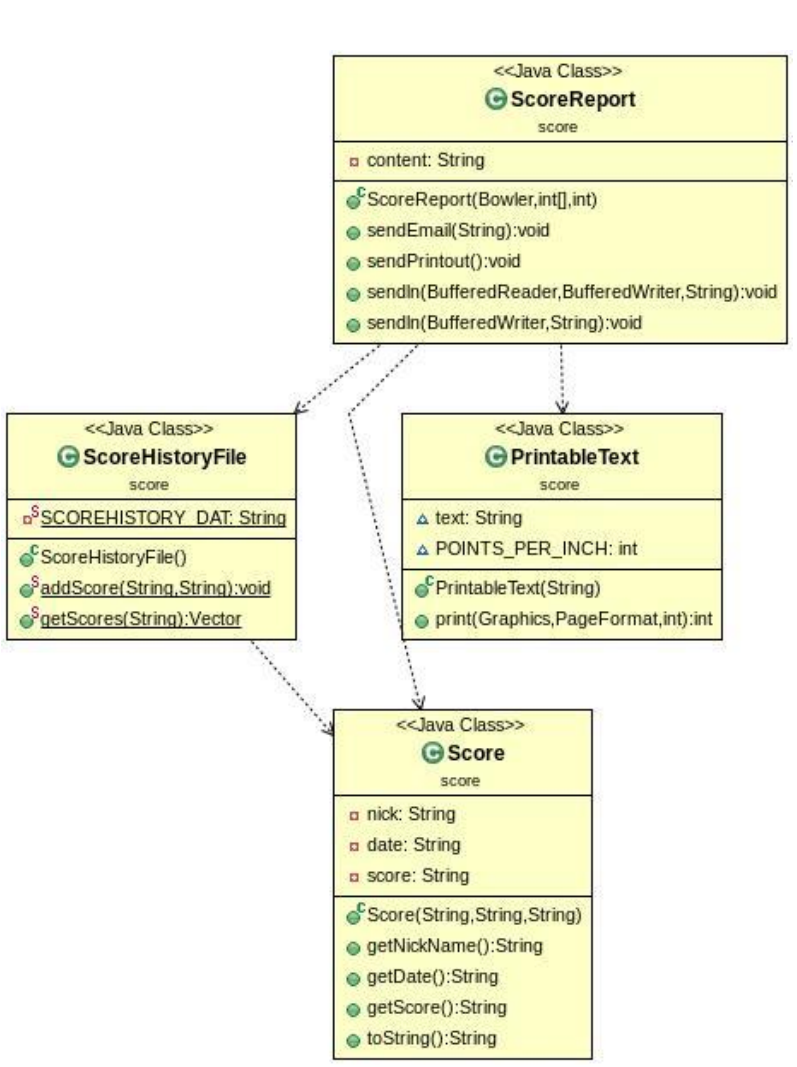


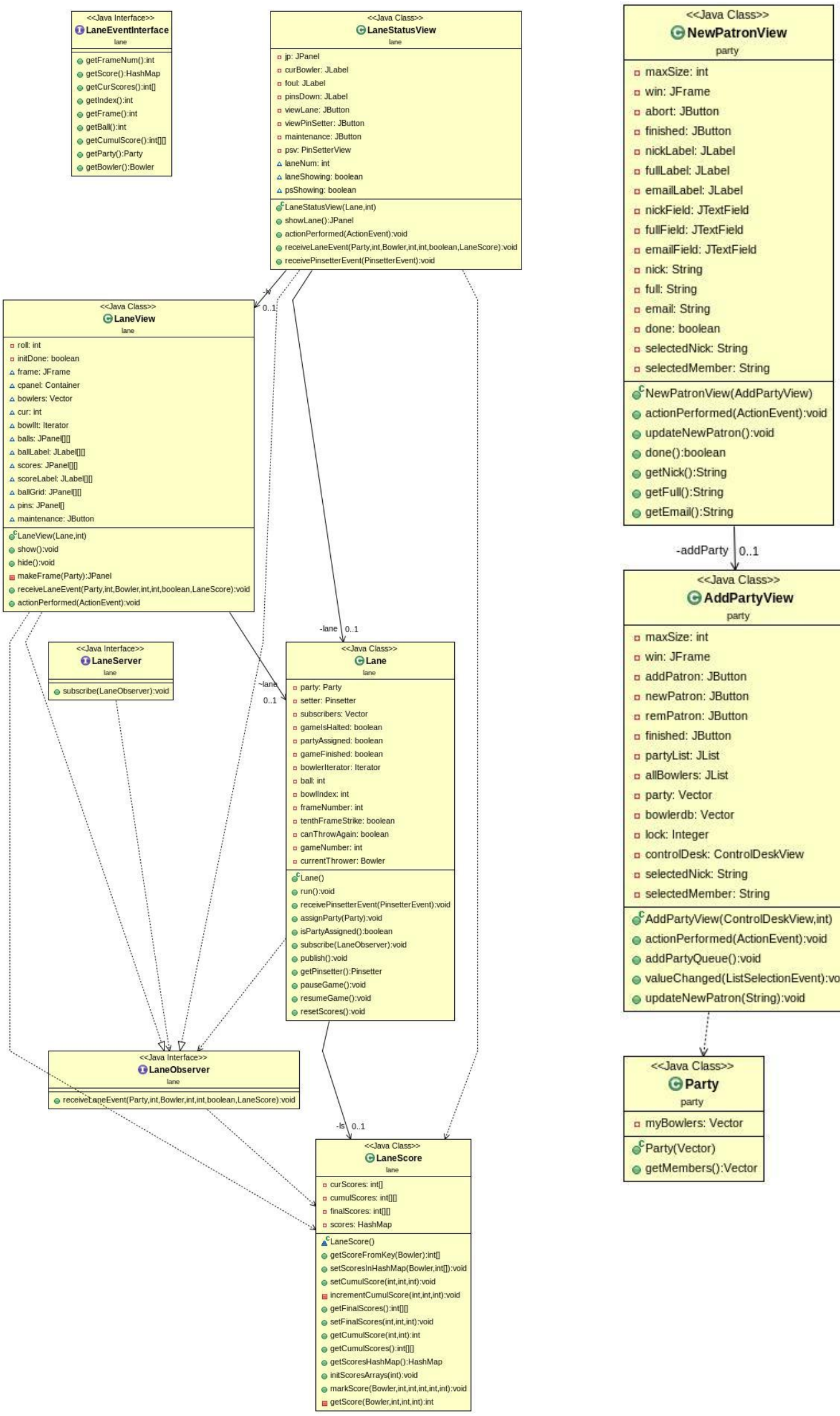


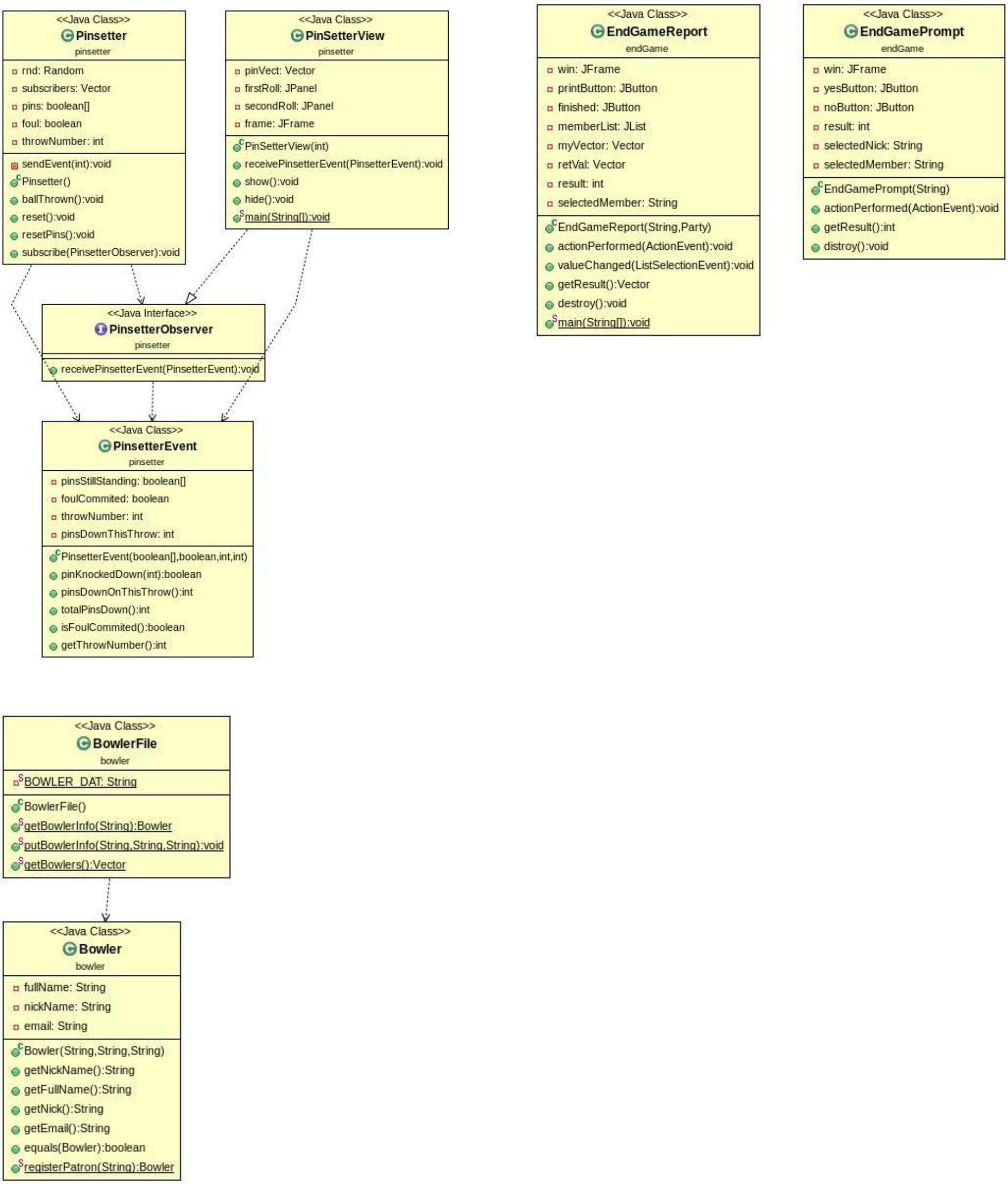
bowling_alley_old_uml_diagram.png

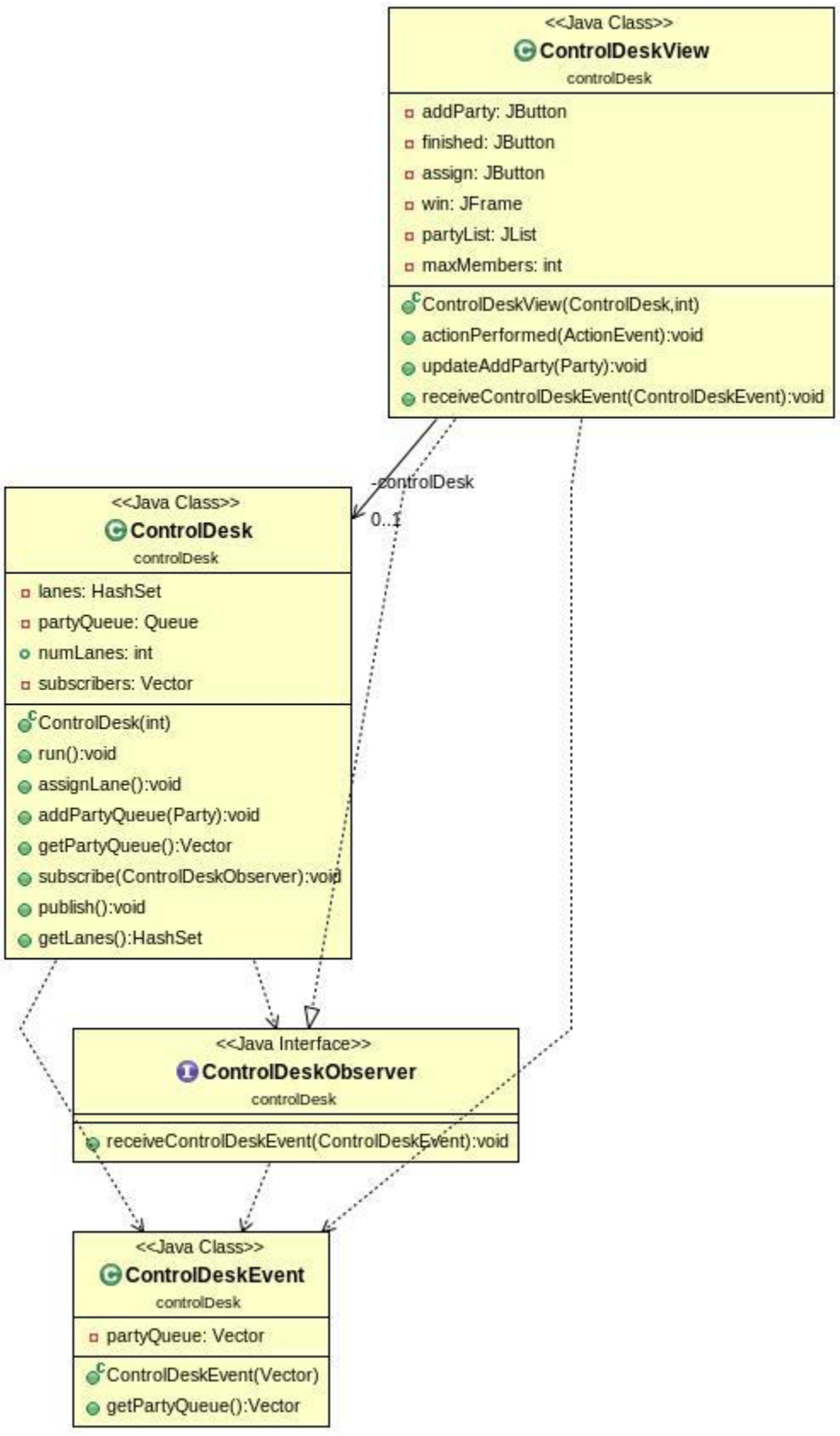
Refactored Class Diagram:



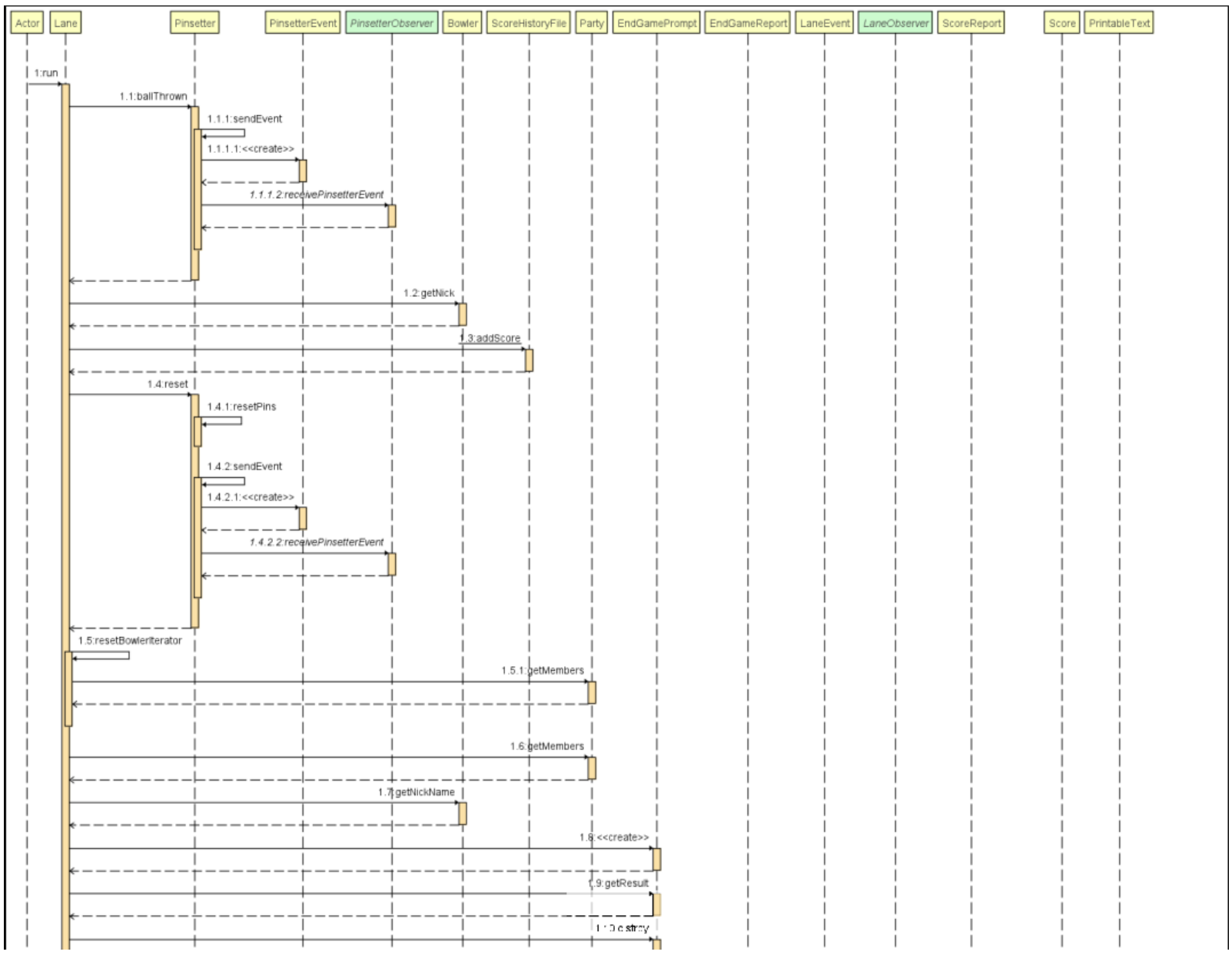



<<Java Interface>>
LaneEventInterface
lane
getFrameNum():int
getScore():HashMap
getCurScores():int[]
getIndex():int
getFrame():int
getBall():int
getCumulScore():int[][]
getParty():Party
getBowler():Bowler
<<Java Class>>
LaneStatusView
lane
jp: JPanel
curBowler: JLabel
foul: JLabel
pinsDown: JLabel
viewLane: JButton
viewPinSetter: JButton
maintenance: JButton
psv: PinSetterView
laneNum: int
laneShowing: boolean
psShowing: boolean
LaneStatusView(Lane,int)
showLane():JPanel
actionPerformed(ActionEvent):void
receiveLaneEvent(Party,int,Bowler,int,int,boolean,LaneScore):void
receivePinsetterEvent(PinsetterEvent):void
<<Java Class>>
NewPatronView
party
maxSize: int
win: JFrame
abort: JButton
finished: JButton
nickLabel: JLabel
fullLabel: JLabel
emailLabel: JLabel
nickField: JTextField
fullField: JTextField
emailField: JTextField
nick: String
full: String
email: String
done: boolean
selectedNick: String
selectedMember: String
NewPatronView(AddPartyView)
actionPerformed(ActionEvent):void
updateNewPatron():void
done():boolean
getNick():String
getFull():String
getEmail():String
-addParty
0..1
<<Java Class>>
AddPartyView
party
maxSize: int
win: JFrame
addPatron: JButton
newPatron: JButton
remPatron: JButton
finished: JButton
partyList: JList
allBowlers: JList
party: Vector
bowlerdb: Vector
lock: Integer
controlDesk: ControlDeskView
selectedNick: String
selectedMember: String
AddPartyView(ControlDeskView,int)
actionPerformed(ActionEvent):void
addPartyQueue():void
valueChanged(ListSelectionEvent):void
updateNewPatron(String):void
<<Java Class>>
Party
party
myBowlers: Vector
Party(Vector)
getMembers():Vector
<<Java Class>>
LaneView
lane
roll: int
initDone: boolean
frame: JFrame
cpanel: Container
bowlers: Vector
cur: int
bowlIt: Iterator
balls: JPanel[][]
ballLabel: JLabel[][]
scores: JPanel[][]
scoreLabel: JLabel[][]
ballGrid: JPanel[][]
pins: JPanel[]
maintenance: JButton
LaneView(Lane,int)
show():void
hide():void
makeFrame(Party):JPanel
receiveLaneEvent(Party,int,Bowler,int,int,boolean,LaneScore):void
actionPerformed(ActionEvent):void
-lane
0..1
<<Java Interface>>
LaneServer
lane
subscribe(LaneObserver):void
~lane
0..1
<<Java Class>>
Lane
lane
party: Party
setter: Pinsetter
subscribers: Vector
gameIsHalted: boolean
partyAssigned: boolean
gameFinished: boolean
bowlerIterator: Iterator
ball: int
bowlIndex: int
frameNumber: int
tenthFrameStrike: boolean
canThrowAgain: boolean
gameNumber: int
currentThrower: Bowler
Lane()
run():void
receivePinsetterEvent(PinsetterEvent):void
assignParty(Party):void
isPartyAssigned():boolean
subscribe(LaneObserver):void
publish():void
getPinsetter():Pinsetter
pauseGame():void
resumeGame():void
resetScores():void
<<Java Interface>>
LaneObserver
lane
receiveLaneEvent(Party,int,Bowler,int,int,boolean,LaneScore):void
-ls
0..1
<<Java Class>>
LaneScore
lane
curScores: int[]
cumulScores: int[][]
finalScores: int[][]
scores: HashMap
LaneScore()
getScoreFromKey(Bowler):int[]
setScoresInHashMap(Bowler,int[]):void
setCumulScore(int,int,int):void
incrementCumulScore(int,int,int):void
getFinalScores():int[][]
setFinalScores(int,int,int):void
getCumulScore(int,int):int
getCumulScores():int[][]
getScoresHashMap():HashMap
initScoresArrays(int):void
markScore(Bowler,int,int,int,int,int):void
getScore(Bowler,int,int,int):int

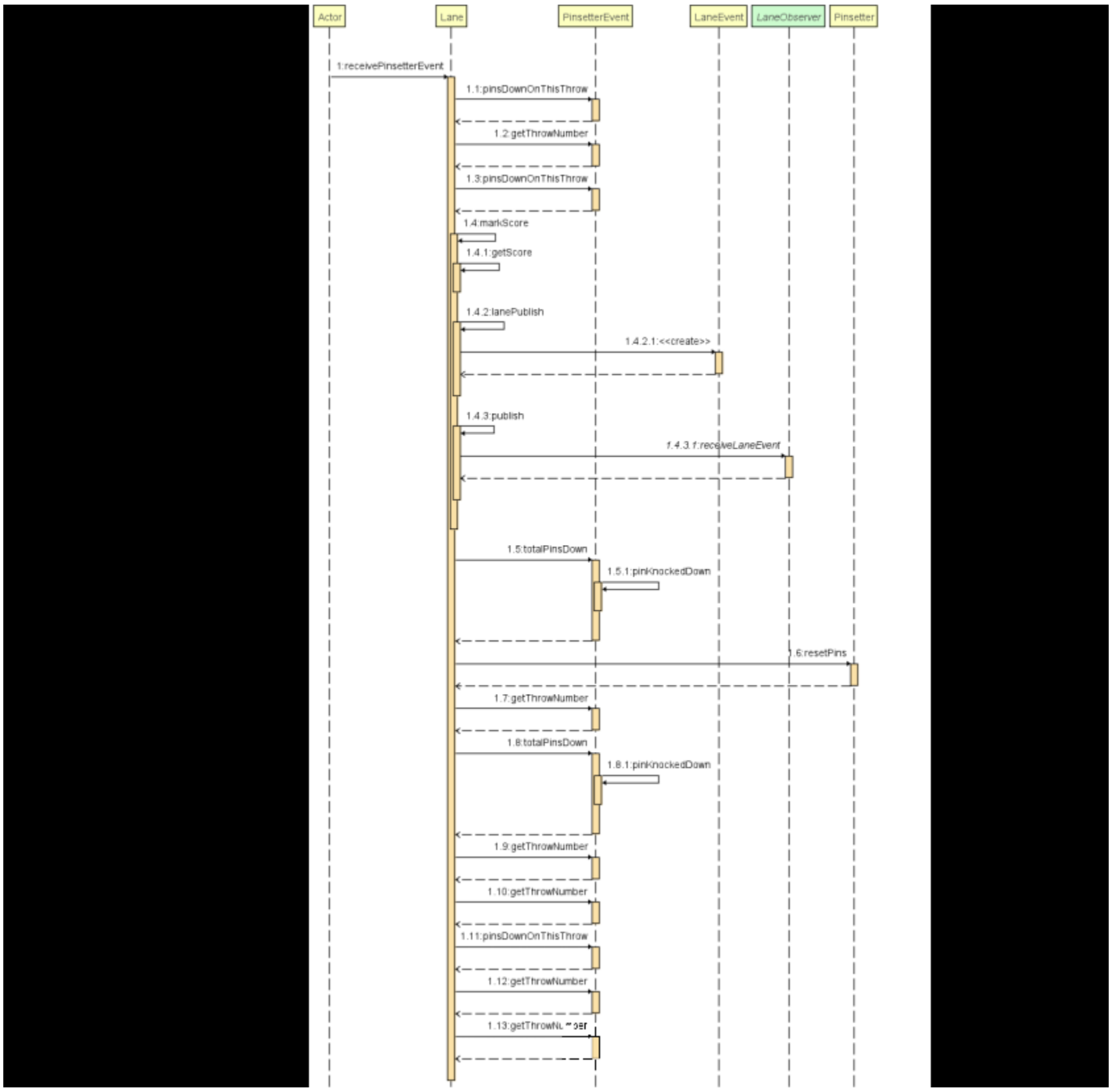

<<Java Class>>
Pinsetter
pinsetter
rnd: Random
subscribers: Vector
pins: boolean[]
foul: boolean
throwNumber: int
sendEvent(int):void
Pinsetter()
ballThrown():void
reset():void
resetPins():void
subscribe(PinsetterObserver):void
<<Java Class>>
PinSetterView
pinsetter
pinVect: Vector
firstRoll: JPanel
secondRoll: JPanel
frame: JFrame
PinSetterView(int)
receivePinsetterEvent(PinsetterEvent):void
show():void
hide():void
main(String[]):void
<<Java Interface>>
PinsetterObserver
pinsetter
receivePinsetterEvent(PinsetterEvent):void
<<Java Class>>
PinsetterEvent
pinsetter
pinsStillStanding: boolean[]
foulCommited: boolean
throwNumber: int
pinsDownThisThrow: int
PinsetterEvent(boolean[],boolean,int,int)
pinKnockedDown(int):boolean
pinsDownOnThisThrow():int
totalPinsDown():int
isFoulCommited():boolean
getThrowNumber():int
<<Java Class>>
EndGameReport
endGame
win: JFrame
printButton: JButton
finished: JButton
memberList: JList
myVector: Vector
retVal: Vector
result: int
selectedMember: String
EndGameReport(String,Party)
actionPerformed(ActionEvent):void
valueChanged(ListSelectionEvent):void
getResult():Vector
destroy():void
main(String[]):void
<<Java Class>>
EndGamePrompt
endGame
win: JFrame
yesButton: JButton
noButton: JButton
result: int
selectedNick: String
selectedMember: String
EndGamePrompt(String)
actionPerformed(ActionEvent):void
getResult():int
distroy():void
<<Java Class>>
BowlerFile
bowler
BOWLER_DAT: String
BowlerFile()
getBowlerInfo(String):Bowler
putBowlerInfo(String,String,String):void
getBowlers():Vector
<<Java Class>>
Bowler
bowler
fullName: String
nickName: String
email: String
Bowler(String,String,String)
getNickName():String
getFullName():String
getNick():String
getEmail():String
equals(Bowler):boolean
registerPatron(String):Bowler


<<Java Class>>
ControlDeskView
controlDesk
addParty: JButton
finished: JButton
assign: JButton
win: JFrame
partyList: JList
maxMembers: int
ControlDeskView(ControlDesk,int)
actionPerformed(ActionEvent):void
updateAddParty(Party):void
receiveControlDeskEvent(ControlDeskEvent):void
-controlDesk
0..1
<<Java Class>>
ControlDesk
controlDesk
lanes: HashSet
partyQueue: Queue
numLanes: int
subscribers: Vector
ControlDesk(int)
run():void
assignLane():void
addPartyQueue(Party):void
getPartyQueue():Vector
subscribe(ControlDeskObserver):void
publish():void
getLanes():HashSet
<<Java Interface>>
ControlDeskObserver
controlDesk
receiveControlDeskEvent(ControlDeskEvent):void
<<Java Class>>
ControlDeskEvent
controlDesk
partyQueue: Vector
ControlDeskEvent(Vector)
getPartyQueue():Vector

Original Sequence Diagrams:

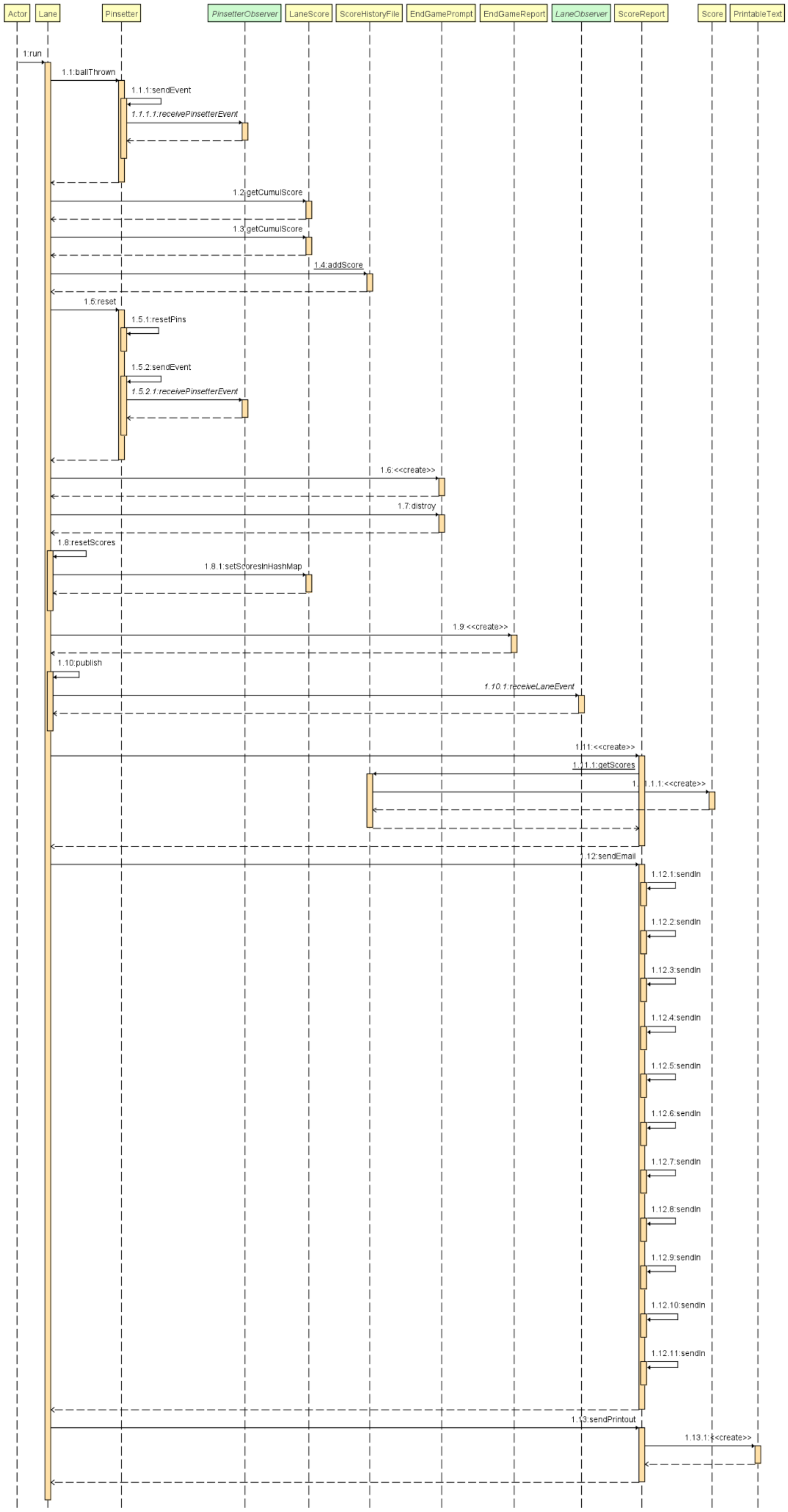


Lane_run_orig.png

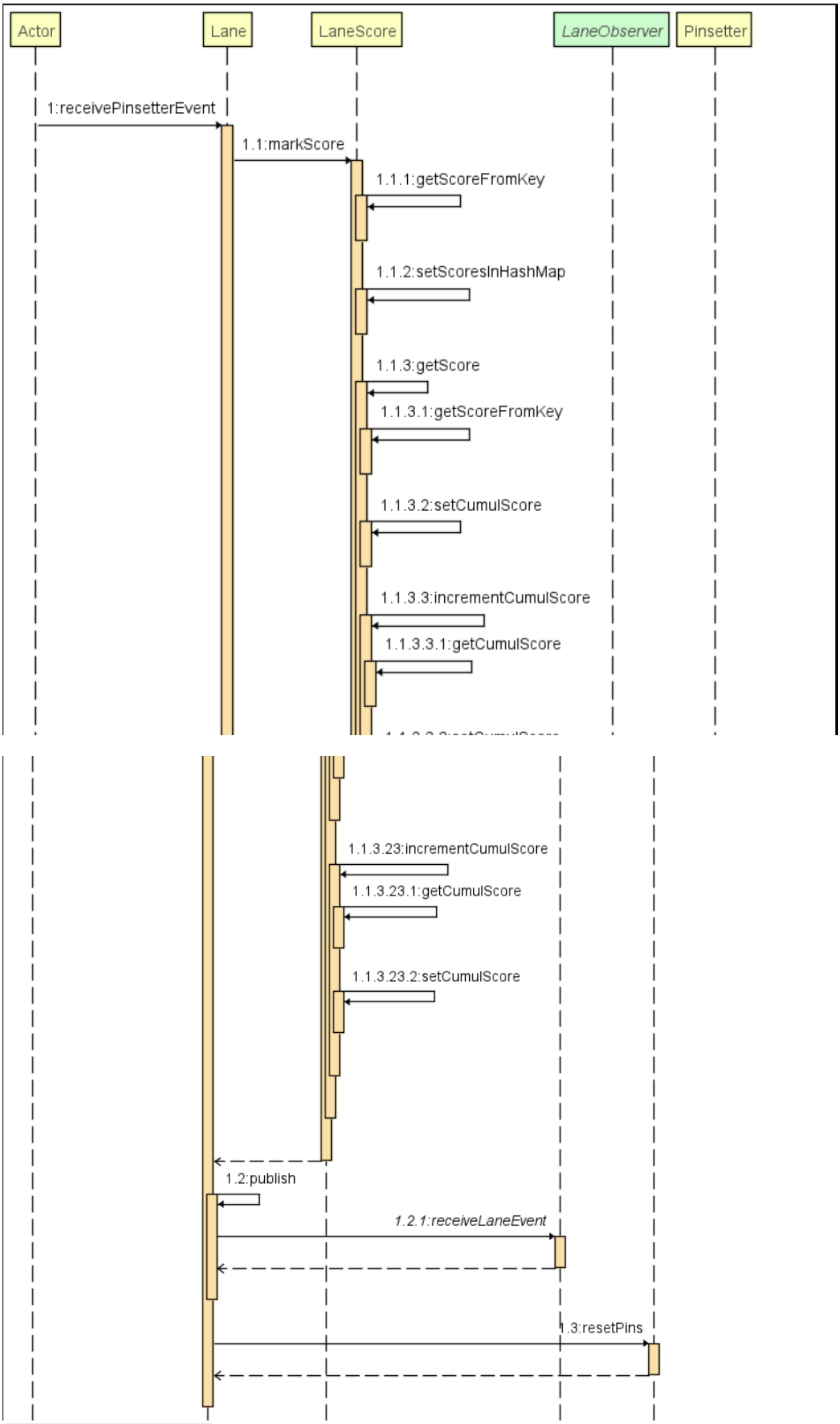


Lane_receivePinsetterEvent_orig.png

Refactored Sequence Diagrams:



Lane_run.png



Lane_receivePinsetterEvent.png

(View of post-refactor metrics in Eclipse using CodeMR)

All metrics

(View of metrics in Eclipse using Metrics2)

Before:

Metric	Total	Mean	Std. De	Maximu	Resource causing Maximum	Method
McCabe Cyclomatic Complexity (avg/		2.319	4.062	38	/BowlingAlley (1)/BowlingAlley/code/Lane.ja	getScore
Number of Parameters (avg/max per		0.723	1.131	9	/BowlingAlley (1)/BowlingAlley/code/LaneEv	LaneEvent
Nested Block Depth (avg/max per me		1.511	1.177	7	/BowlingAlley (1)/BowlingAlley/code/Lane.ja	run
Afferent Coupling (avg/max per packa		0	0	0	/BowlingAlley (1)/BowlingAlley/code	
Efferent Coupling (avg/max per packa		0	0	0	/BowlingAlley (1)/BowlingAlley/code	
Instability (avg/max per packageFrag		1	0	1	/BowlingAlley (1)/BowlingAlley/code	
Abstractness (avg/max per packageF		0.172	0	0.172	/BowlingAlley (1)/BowlingAlley/code	
Normalized Distance (avg/max per pa		0.172	0	0.172	/BowlingAlley (1)/BowlingAlley/code	
Depth of Inheritance Tree (avg/max p		0.897	0.48	2	/BowlingAlley (1)/BowlingAlley/code/Contro	
Weighted methods per Class (avg/ma	327	11.276	15.991	87	/BowlingAlley (1)/BowlingAlley/code/Lane.ja	
Number of Children (avg/max per typ	6	0.207	0.663	3	/BowlingAlley (1)/BowlingAlley/code/PinsetI	
Number of Overridden Methods (avg	3	0.103	0.305	1	/BowlingAlley (1)/BowlingAlley/code/Contro	
Lack of Cohesion of Methods (avg/ma		0.375	0.374	0.91	/BowlingAlley (1)/BowlingAlley/code/LaneEv	
Number of Attributes (avg/max per t	138	4.759	5.556	18	/BowlingAlley (1)/BowlingAlley/code/Lane.ja	
Number of Static Attributes (avg/ma	2	0.069	0.253	1	/BowlingAlley (1)/BowlingAlley/code/ScoreH	
Number of Methods (avg/max per typ	133	4.586	3.765	17	/BowlingAlley (1)/BowlingAlley/code/Lane.ja	
Number of Static Methods (avg/max	8	0.276	0.69	3	/BowlingAlley (1)/BowlingAlley/code/Bowler	
Specialization Index (avg/max per typ		0.017	0.052	0.2	/BowlingAlley (1)/BowlingAlley/code/Score.j	
Number of Classes (avg/max per pacl	29	29	0	29	/BowlingAlley (1)/BowlingAlley/code	
Number of Interfaces (avg/max per p	5	5	0	5	/BowlingAlley (1)/BowlingAlley/code	

After

Metric	Total	Mean	Std. De	Maximu	Resource causing Maximum	Method
McCabe Cyclomatic Complexity (avg/		2.438	4.142	37	/code/lane/LaneScore.java	getScore
Number of Parameters (avg/max per		1.078	1.254	5	/code/lane/LaneView.java	receiveLaneEvent
Nested Block Depth (avg/max per me		1.547	1.178	7	/code/lane/LaneScore.java	getScore
Afferent Coupling (avg/max per pack		3.9	3.419	11	/code/bowler	
Efferent Coupling (avg/max per packa		1.5	1.628	6	/code/lane	
Instability (avg/max per packageFrag		0.375	0.298	1	/code	
Abstractness (avg/max per packageF		0.101	0.16	0.429	/code/lane	
Normalized Distance (avg/max per pa		0.56	0.317	1	/code/bowler	
Depth of Inheritance Tree (avg/max p		0.897	0.48	2	/code/controlDesk/ControlDesk.java	
Weighted methods per Class (avg/ma	312	10.759	11.587	47	/code/lane/LaneScore.java	
Number of Children (avg/max per typ	6	0.207	0.663	3	/code/pinsetter/PinsetterObserver.java	
Number of Overridden Methods (avg	3	0.103	0.305	1	/code/controlDesk/ControlDesk.java	
Lack of Cohesion of Methods (avg/ma		0.3	0.315	0.757	/code/lane/Lane.java	
Number of Attributes (avg/max per t	107	3.69	4.58	16	/code/lane/Lane.java	
Number of Static Attributes (avg/ma	2	0.069	0.253	1	/code/bowler/BowlerFile.java	
Number of Methods (avg/max per typ	119	4.103	2.975	11	/code/lane/LaneScore.java	
Number of Static Methods (avg/max	9	0.31	0.7	3	/code/bowler/BowlerFile.java	
Specialization Index (avg/max per typ		0.022	0.066	0.25	/code/controlDesk/ControlDesk.java	
Number of Classes (avg/max per pacl	29	2.9	1.7	7	/code/lane	
Number of Interfaces (avg/max per p	5	0.5	0.922	3	/code/lane	

Class based metrics distribution

Before:

1	Lane					227	medium-high	low-medium	medium-high	low-medium
2	ControlDeskView					87	low-medium	low-medium	low-medium	low-medium
3	ControlDesk					68	low-medium	low-medium	medium-high	low-medium
4	LaneStatusView					93	low	low-medium	low-medium	low-medium
5	LaneView					140	low-medium	low	low-medium	low-medium
6	AddPartyView					127	low-medium	low	low-medium	low-medium
7	PinSetterView					111	low	low	low	low-medium
8	NewPatronView					85	low	low	low	low-medium
9	EndGameReport					79	low	low	low-medium	low-medium
10	ScoreReport					76	low	low	low	low-medium
11	EndGamePrompt					55	low	low	low	low-medium
12	Pinsetter					47	low	low	low	low
13	LaneEvent					41	low	low	medium-high	low
14	BowlerFile					38	low	low	low	low

15	PinsetterEvent					26	low	low	low	low
16	Bowler					25	low	low	low	low
17	PrintableText					21	low	low	low	low
18	ScoreHistoryFile					20	low	low	low	low
19	Score					16	low	low	low	low
20	Queue					12	low	low	low	low
21	LaneEventInterface					10	low	low	low	low
22	drive					8	low	low	low	low
23	Alley					6	low	low	low	low
24	ControlDeskEvent					6	low	low	low	low
25	Party					6	low	low	low	low
26	ControlDeskObserver					2	low	low	low	low
27	LaneObserver					2	low	low	low	low
28	LaneServer					2	low	low	low	low
29	PinsetterObserver					2	low	low	low	low

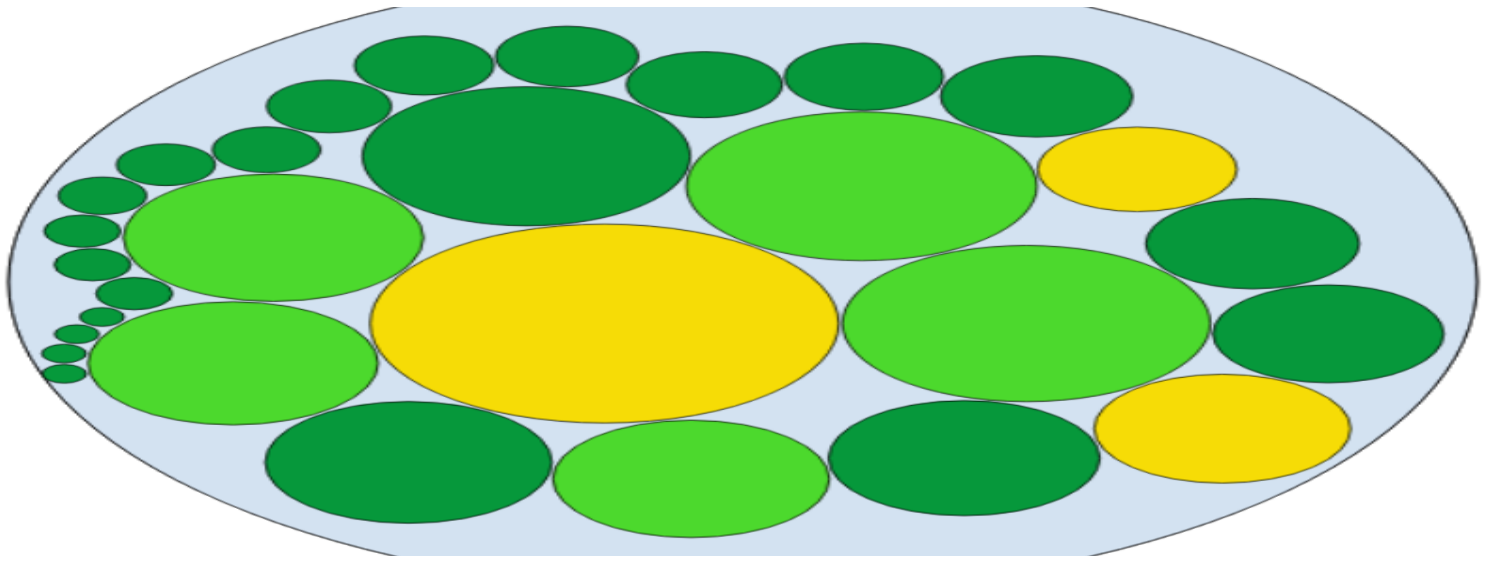
After:

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	Lane					141	low-medium	low-medium	low-medium	low-medium
2	AddPartyView					96	low-medium	low-medium	low-medium	low-medium
3	ControlDeskView					69	low-medium	low-medium	low-medium	low-medium
4	ControlDesk					51	low-medium	low-medium	low-medium	low-medium
5	LaneStatusView					85	low	low-medium	low-medium	low-medium
6	LaneView					120	low-medium	low	low-medium	low-medium
7	LaneScore					90	low-medium	low	low	low-medium
8	PinSetterView					111	low	low	low	low-medium
9	ScoreReport					76	low	low	low	low-medium
10	EndGameReport					75	low	low	low	low-medium
11	NewPatronView					66	low	low	low	low-medium
12	EndGamePrompt					54	low	low	low	low-medium

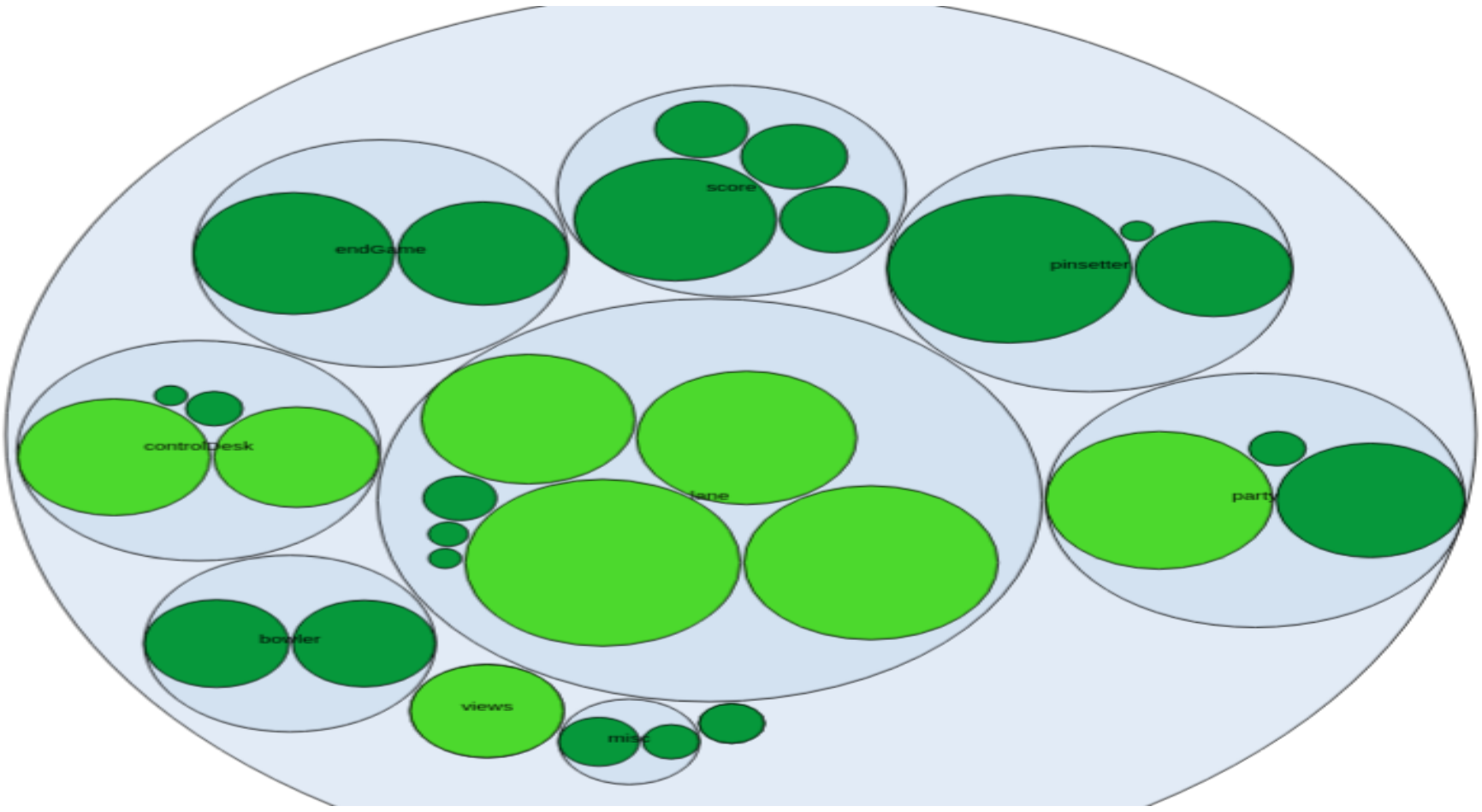
13	Pinsetter					46	low	low	low	low
14	Factory					44	low	low	low-medium	low
15	BowlerFile					39	low	low	low	low
16	Bowler					38	low	low	low	low
17	ScoreHistoryFile					22	low	low	low	low
18	PrintableText					21	low	low	low	low
19	Score					16	low	low	low	low
20	Queue					12	low	low	low	low
21	LaneEventInterface					10	low	low	low	low
22	drive					8	low	low	low	low
23	ControlDeskEvent					6	low	low	low	low
24	Party					6	low	low	low	low
25	Alley					6	low	low	low	low

Packages Structure

Before



After

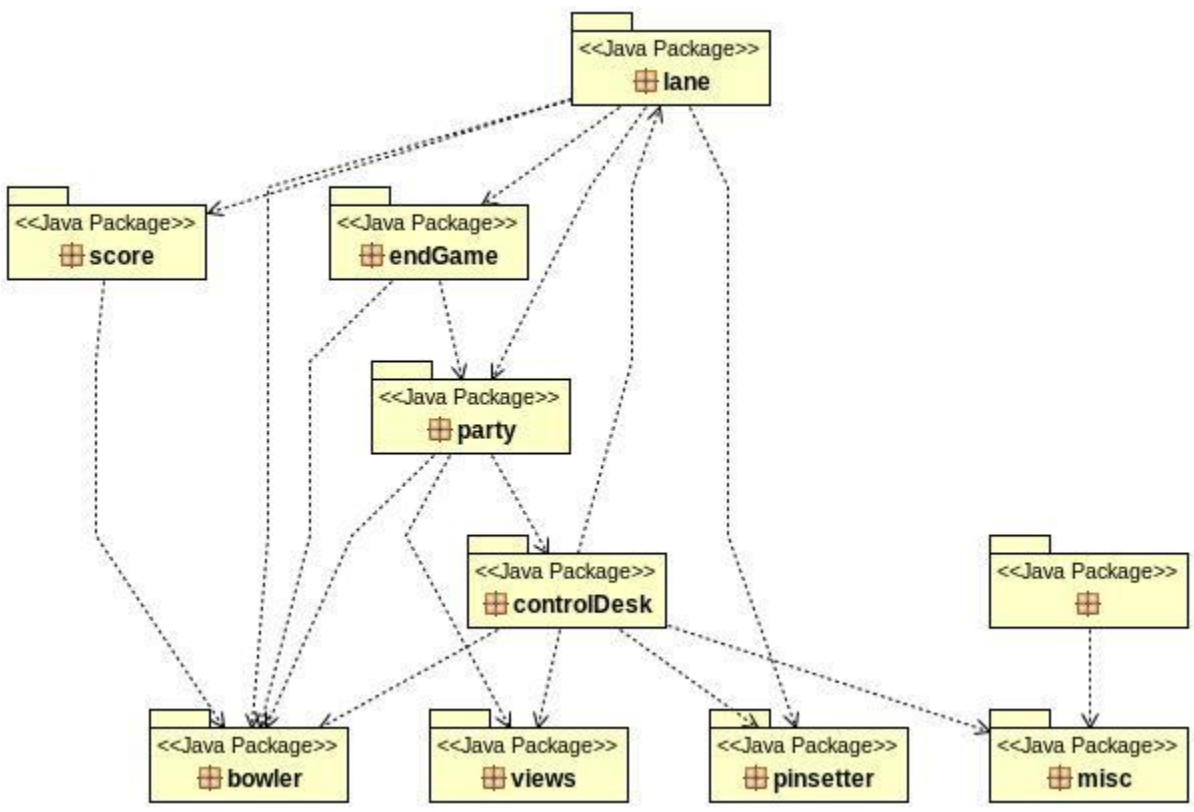


Refactoring :

1. Divided Single Package into Multiple Sub Package :

We divide similar classes in one package based on how similar there functionality are
List of subpackages created :

- lane
- score
- endGame
- party
- controlDesk
- bowler
- views
- pinsetter
- misc



2. Low Coupling :

One Module should be independent from other modules, so that when changes are made to one module it doesn't heavily impact other modules. When your module knows too much about the inner workings of other modules is known as High coupling. If Module A knows too much about Module B, changes to the internals of Module B may break functionality in Module A. By aiming for low coupling, we can easily make changes to the internals of modules without worrying about their impact on other modules in the system. With Low coupling it becomes easier to design, write, and test code since our modules are not interdependent on each other. We also get the benefit of easy to reuse and compose-able modules. Problems are also isolated to small, self-contained units of code.For decreasing the coupling we made Interfaces for various classes. Interfaces helps to reduce the coupling which is a very important metric for code analysis.

Advantages of Interfaces:

- Multiple inheritance in java can be achieved through interfaces
- Interfaces function to break up the complex designs and clear the dependencies between objects.
- Interfaces make applications loosely coupled.

3. Improving Performance :

In order to improve performance we used a function append in place of string concatenation using '+' in loops. For that we replaced string by String builder. This has better performance because string append simply modify the string whereas string concat always copies .

4. Code Repetition :

Code Repetition is repetition of a line or a block of code in the same file or sometimes in the same local environment. In general a lot of people consider it as acceptable but in reality, it poses greater problems to software than what we may have imagined. Even code with similar functionalities are said to be duplications.Copy and Paste Programming is one of the main reasons for creation of the duplicate code.

Issue :

In the Initial code the same code was repeated many times in almost every file. Some examples of this are :
In file *AddPartyView.java :*

```
 addPatron = new JButton("Add to Party");
JPanel addPatronPanel = new JPanel();
addPatronPanel.setLayout(new FlowLayout());
addPatron.addActionListener(this);
addPatronPanel.add(addPatron);

remPatron = new JButton("Remove Member");
JPanel remPatronPanel = new JPanel();
remPatronPanel.setLayout(new FlowLayout());
remPatron.addActionListener(this);
remPatronPanel.add(remPatron);
```

In the above example we can see that the same code is repeated for making buttons. Instead of writing the same code multiple times we can convert it into a function and call it multiple times.

Problems with previous style :

Code which includes duplicate functionality is more difficult to support because if at any point we need to update the code, there might be a case where we just update one copy of the code without checking for the presence of other instances of the same code and it also makes the code longer

Solution :

To solve this issue of code repetition initially we simply made a function CreatePanelwithButton and called it multiple times as below :

```
public JPanel CreatePanelWithButton(JButton button, LayoutManager mgr, ActionListener l) {
    JPanel panel = new JPanel();
    panel.setLayout(mgr);
    button.addActionListener(l);
    panel.add(button);

    return panel;
}
```

Advantages :

This organization makes code easier to read and allows the programmer to reuse code throughout a program.

5. Removed Unnecessary Public Modifier in Interfaces :

In Interfaces public modifier was used with many functions. There was no need to make them public explicitly as all methods in an interface are already **public and abstracts** Therefore we removed the public modifier from these interfaces.

LaneEventInterface.java

Before Refactor :

```
public interface LaneEventInterface extends java.rmi.Remote {
        public int getFrameNum( ) throws java.rmi.RemoteException;
        public HashMap getScore( ) throws java.rmi.RemoteException;
        public int[] getCurScores( ) throws java.rmi.RemoteException;
        public int getIndex() throws java.rmi.RemoteException;
    }
```

After Refactor :

```
public interface LaneEventInterface extends java.rmi.Remote {
        int getFrameNum( ) throws java.rmi.RemoteException;
        HashMap getScore( ) throws java.rmi.RemoteException;
        int[] getCurScores( ) throws java.rmi.RemoteException;
        int getIndex() throws java.rmi.RemoteException;
    }
```

6. Removed Unnecessary Imports :

Unused and useless imports should be removed. Leaving them in the code reduces the code's readability, since their presence can be confusing.

7. Removed Deprecated methods :

Some deprecated functions were used in many files ex. Deprecated method of win.hide and win.show was used; we replaced it by setVisible(true/false).

8. Assigning appropriate functions to appropriate files :

Good coding habits suggest that a file should contain all related functions whereas in many files a lot of varying variety of functions were written. This can lead to the problem of **Lack of Cohesion** :

Measure how well the methods of a class are related to each other. High cohesion (low lack of cohesion) tends to be preferable, because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and

understandability. In contrast, low cohesion is associated with undesirable traits such as being difficult to maintain, test, reuse, or even understand. The assumption behind the following cohesion metrics is that methods are related if they work on the same class-level variables.Methods are unrelated if they work on different variables altogether. In a cohesive class, methods work with the same set of variables. In a non-cohesive class, there are some methods that work on different data.

In code in files many functions were not even related to each other so we moved functions to appropriate files or created new files for that purpose.

Ex : **Lane.java** : It was one of the most messed up files in the entire code. It was doing a lot of things which were not even related. likeNo one can say that a file named Lane.java was calculating the score of a bowler which it was doing. There were many more such functions performed by this like it was doing everything related to score in itself. So for this we made a new class ***LaneScore.java*** and moved all such related functions to calculated score in that file with appropriate parameters. Following functions from **Lane** file were moved to **LaneScore** file :

- getScore()
- markScore()

No one can say that these functions are in the Lane file by name of that file. According to good coding practices a file name should be enough to describe
what that file is doing. That's why we kept the name of the new file as CalculateScore

Advantages :

- The complexity of module will be reduced
- Increased system maintainability
- As application developers will find the component they need more easily among the cohesive set of operations provided by the module. Therefore module reusability is increased

9. Unused Variables :

The variables declared initially for some purpose but not used later in the code. They can point to a lack of proper design when not used properly but they can also be powerful if used in the correct way . Example : nickLabel, fullLabel and emailLabel in file **NewPatron.java.** Same thing was done in every file containing unused variables.

10. Empty Catch statements :

In many files **Catch statements** were empty.It would have been really difficult to debug the code if appropriate errors were not printed.

Ex: *LaneView.java*

Before Refactor :

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

***After* Refactor :**

```
try {
    Thread.sleep(1);
} catch (Exception e) {
    System.err.println("Error: " + e);
}
```

11. Removed Redundant Casting to various fields :

Unnecessary casting expressions make the code harder to read and understand.

12. Abstract classes Made a few abstract classes for which we don't want to make any objects.

13. Missing Comments :

Comments are text notes added to the program to provide explanatory information about the source code. They are used in a programming language to document the program and remind programmers of what tricky things they just did with the code and also helps the later generation for understanding and maintenance of code. The compiler considers these as non-executable statements Comments are used for integration with source code management systems and other kinds of external programming tools.

14. Decreasing Number of Parameters :

The file LaneEvent was taking a lot of parameters so we need to reduce them as according to good coding practices the number of parameters in a function should not be so high.

Metric Analysis of Original and Refactored code

1- <u>Complexity</u>

- This Implies being difficult to understand and describes the interactions between a number of entities.
- Before:
- Some classes have high complexity. For example , Lane.java complexity was lying between medium to high. Higher levels of complexity in software increase the risk of unintentionally interfering with interactions and so increases the chance of introducing defects when making changes.
- After:
- We were able to bring down the complexity in the low to medium range. For example, We divided the Lane.Java into two classes Lane.java and LaneScore.java.

2- <u>Number of Parameters</u>

- It is the number of parameters passed amongst methods.
- Before:
- Few classes have a high number of parameters. For example constructor LaneEvent was having 9 parameters.

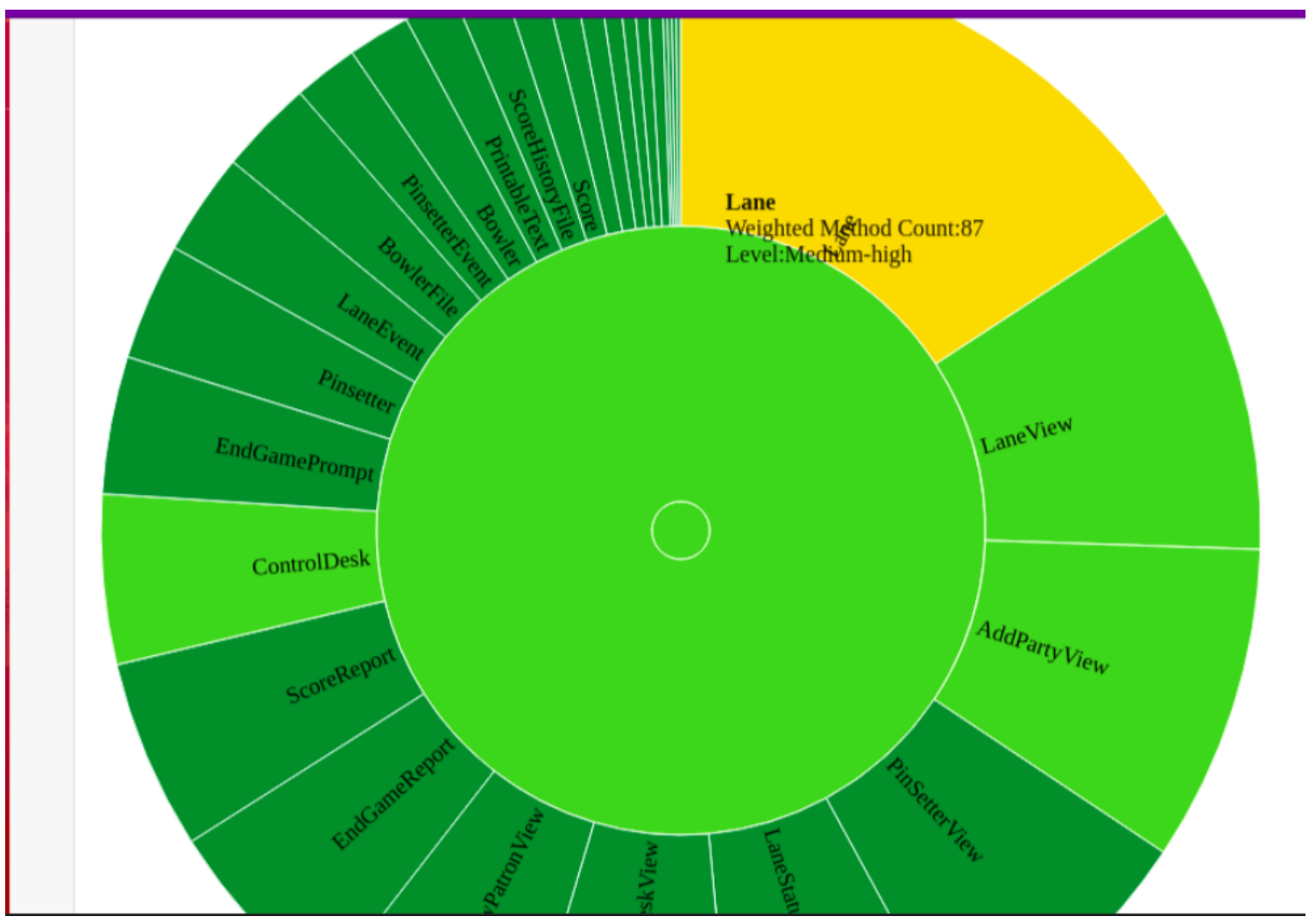
```
public LaneEvent( Party pty, int theIndex, Bowler theBowler, int[][] theCumulScore, HashMap theScore, int theFrameNum, int[] theCurSc
    p = pty;
    index = theIndex;
    bowler = theBowler;
    cumulScore = theCumulScore;
    score = theScore;
    curScores = theCurScores;
    frameNum = theFrameNum;
    ball = theBall;
    mechProb = mechProblem;
}
```

- After:
- We reduced the number of parameters by removing the LaneEvent class.

3 - <u>Weighted Methods count</u>

- The weighted sum of all class' methods represents the McCabe complexity of a class. It is equal to the number of methods, if the complexity is taken as 1 for each method. The number of methods and complexity can be used to predict development, maintaining and testing effort estimation.

- **<u>Before:</u>**



- **<u>After:</u>**

- For example, We divided the Lane.Java into two classes Lane.java and LaneScore.java.

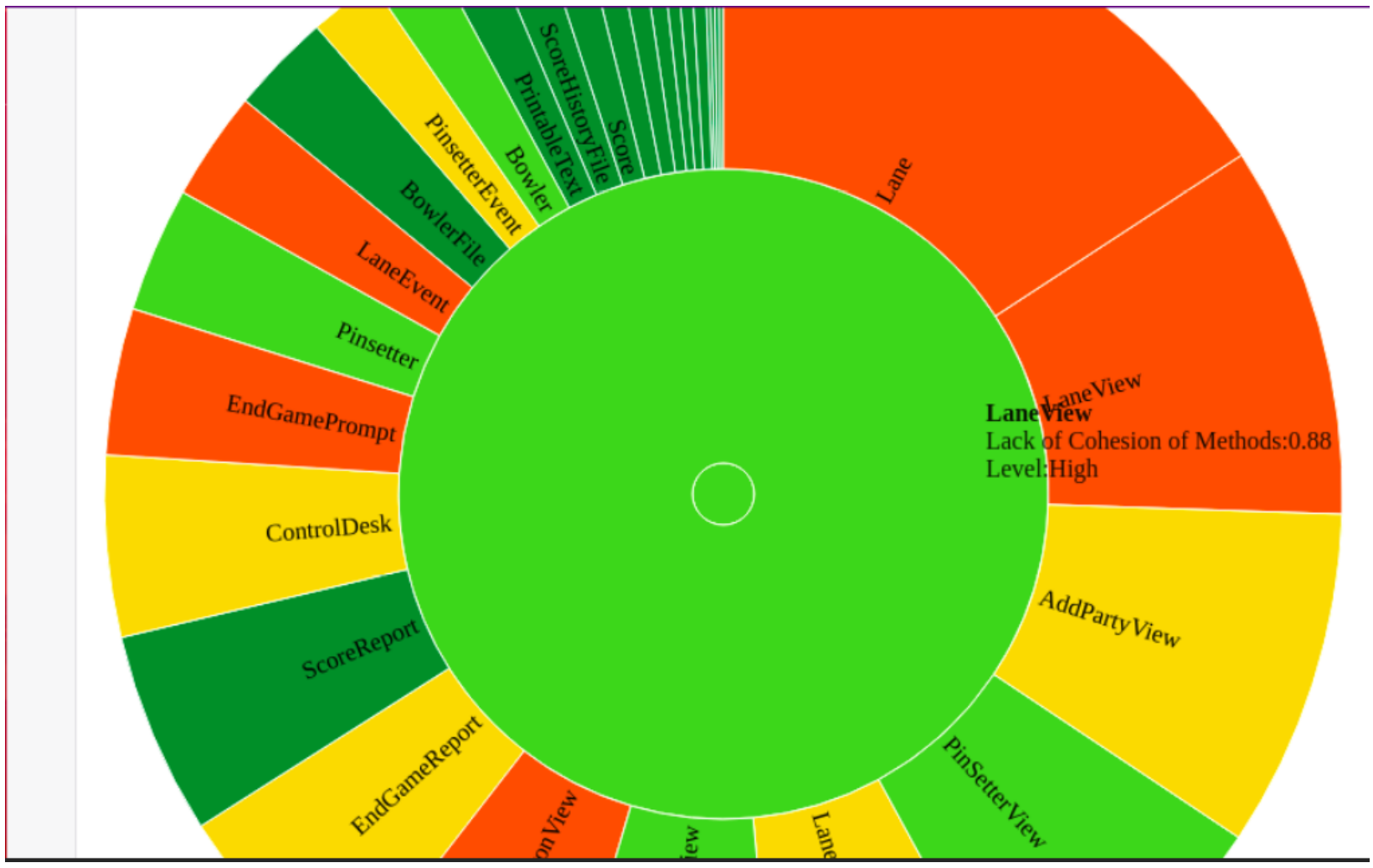
4 - <u>Number of Methods</u>

- It is the number of methods in a class.
- Before:
- Earlier 133 methods were there.By looking at the metrics , we infer that there were some functions that were not being called.
- After:
- After merging few methods of same functionality like PauseGame and ResumeGame and removing dead functions code , we have reduced the number of methods to 119
- It increases readability and reduces the size of code.

5 - <u>Lack of cohesion of methods:</u>

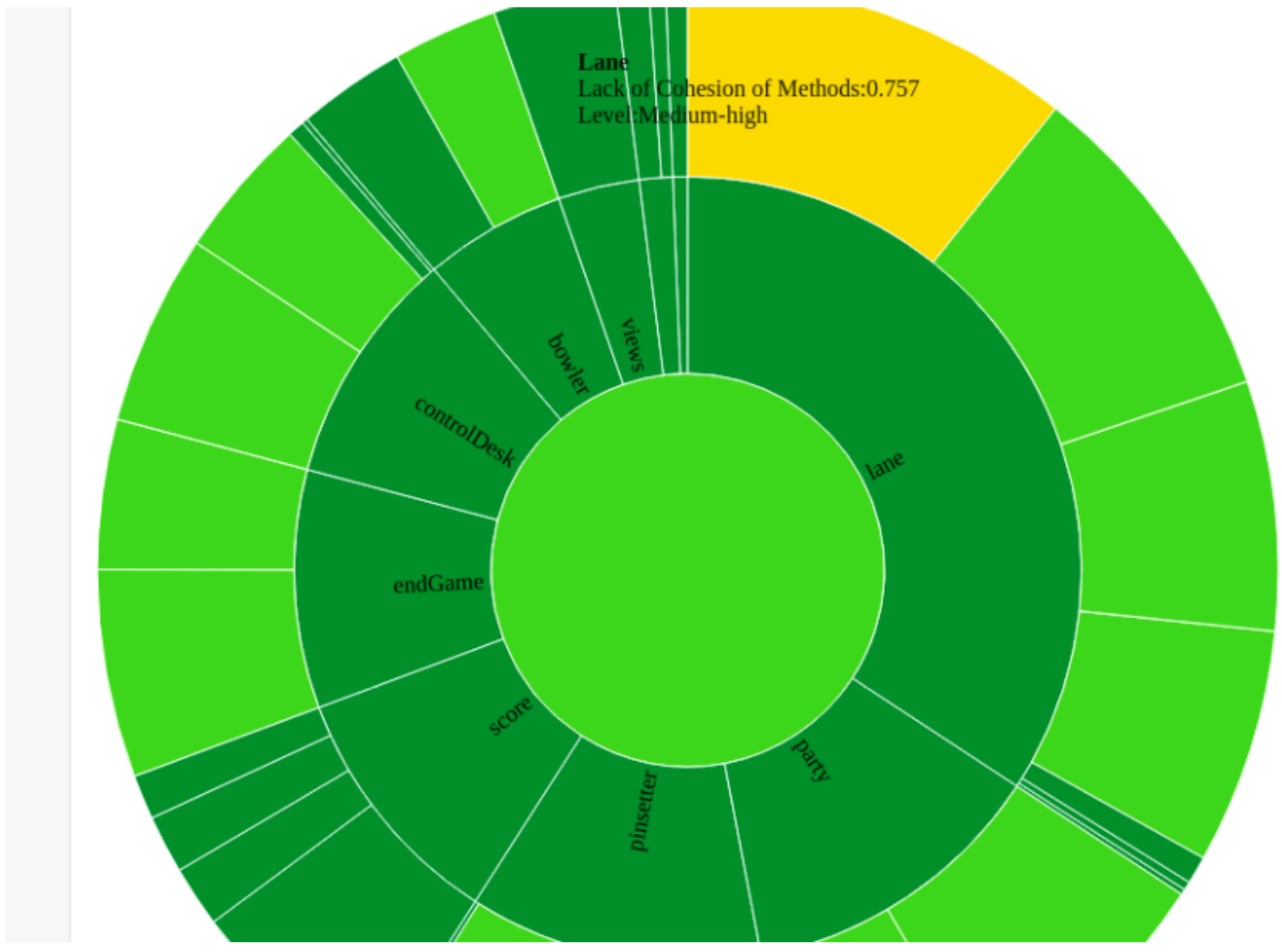
- Measure how methods of a class are related to each other. Low cohesion means that the class implements more than one responsibility. Lack of cohesion also influences understandability and implies classes should probably be split into two or more subclasses.

- **<u>Before :</u>**



- A lot of classes were having high LCOM metric which implies a lot of those classes had methods which were either never being called or were accessing variables outside the class. Also, many methods had unused variables.

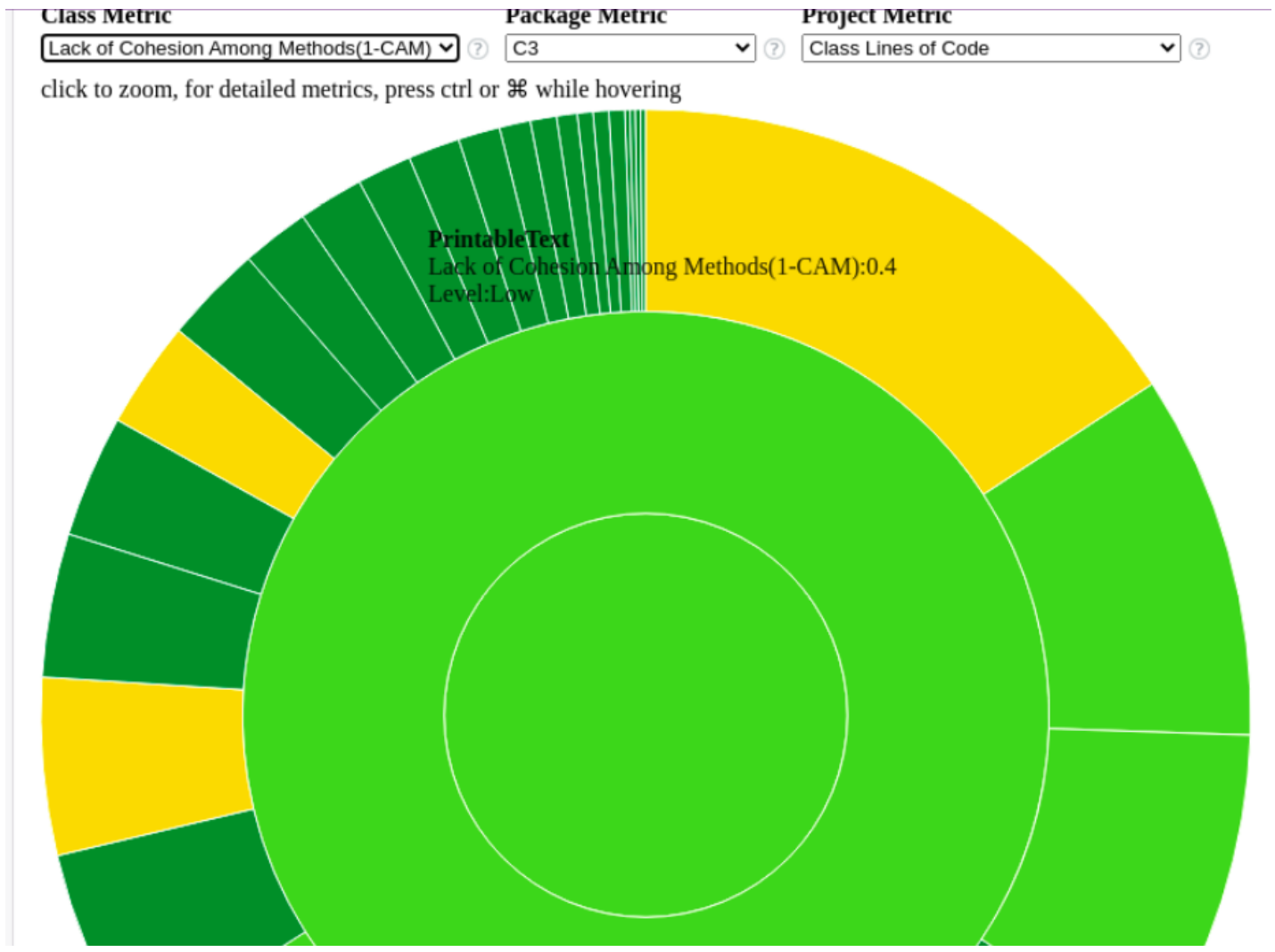
- **After:**



- We were able to bring down the LCOM metric by properly restructuring the code and making sure methods were in the right classes.
- Also, a lot of unused variables were removed.

6- Lack Of Cohesion Among Methods -

- CAM metric is the measure of cohesion based on parameter types of methods. LCAM = 1-CAM
- **Before -**
    - A lot of classes like Lane.java,
      LaneEvent and ControlDesk has LCAM in medium-high range.



After -

- After refactoring we were able to bring the
  LCAM metric for above 3 classes in low-medium and low range.

7 - Number of Attributes

- Before :
- Earlier it contained 138 attributes.
- After:
- Some of them were not being used.So we removed them and reduced the number to 107.

8 - Number of Classes

- Before:
- There were 29 classes.
- After:
- We added LaneScore.java and Factory.java and removed LaneEvent.java and PinSetterEvent.java. So the number of classes remained the same.

9 - <u>Method Lines of Code</u>

- Total number of all nonempty, non-commented lines of methods inside a class.
- Before:
- Total 1284 and mean 9.106
- After:
- Total 1194 mean 9.328
- We observed that a lot of methods were either not being called ever or they were a part of a class which were just container classes so those classes and its methods could be removed.
- The total no of methods line of code has decreased but the mean seems to increase due to the fact that no of methods decreased after refactoring.