Design and Analysis Of Software Systems

Bowling Game

[Assignment -3]

Team Members:

1 - Kanish Anand - 2018101025 (Refactoring)

2 - Mehul Goyal - 2018101018 (Report and docs)

3 - Shanmukh Karra - 2018101111 (New Features)

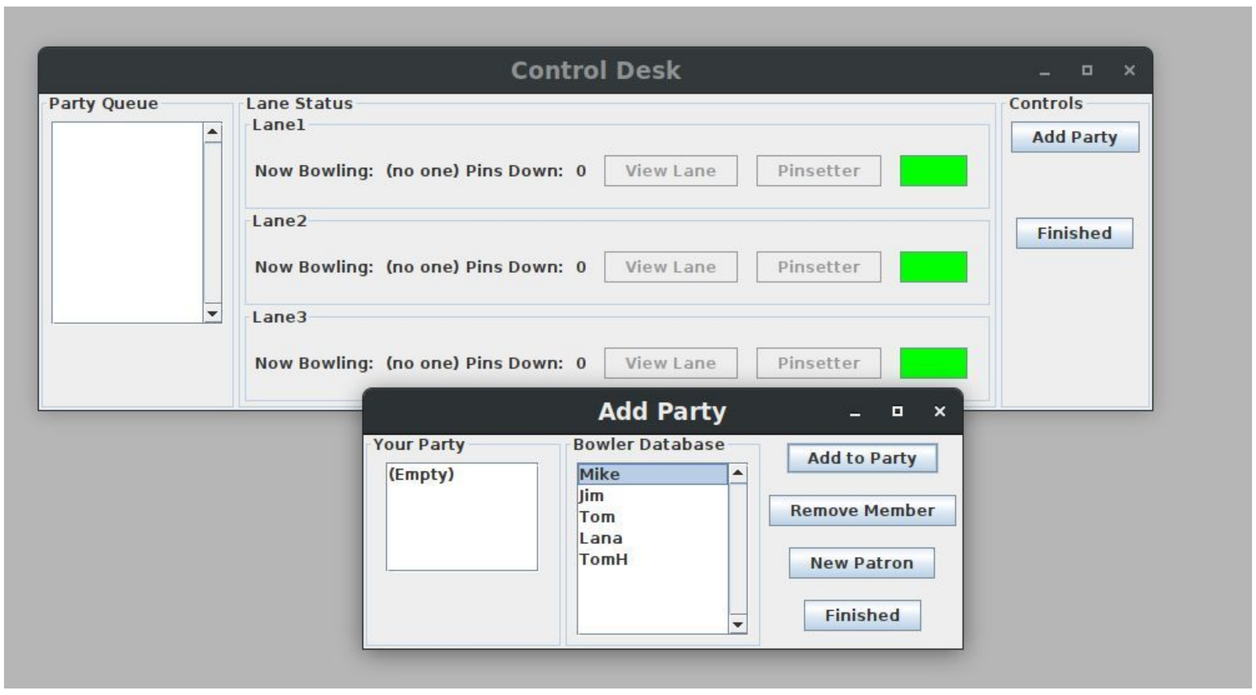
Equal effort was put in by each member, and the total project took around 80 hours.

Introduction

This is a Bowling Game program written in Java.

Overview

The product is a simulator of a bowling game wherein the user of the product is not who is actually bowling. He can select a party of his own with a number of players who can play the game, taking turns to knock over Bowling Pins.



Lane1

Now Bowling: (no one) Pins Down: 0

Lane2

Now Bowling: (no one) Pins Down: 0

Lane3

Now Bowling: (no one) Pins Down: 0

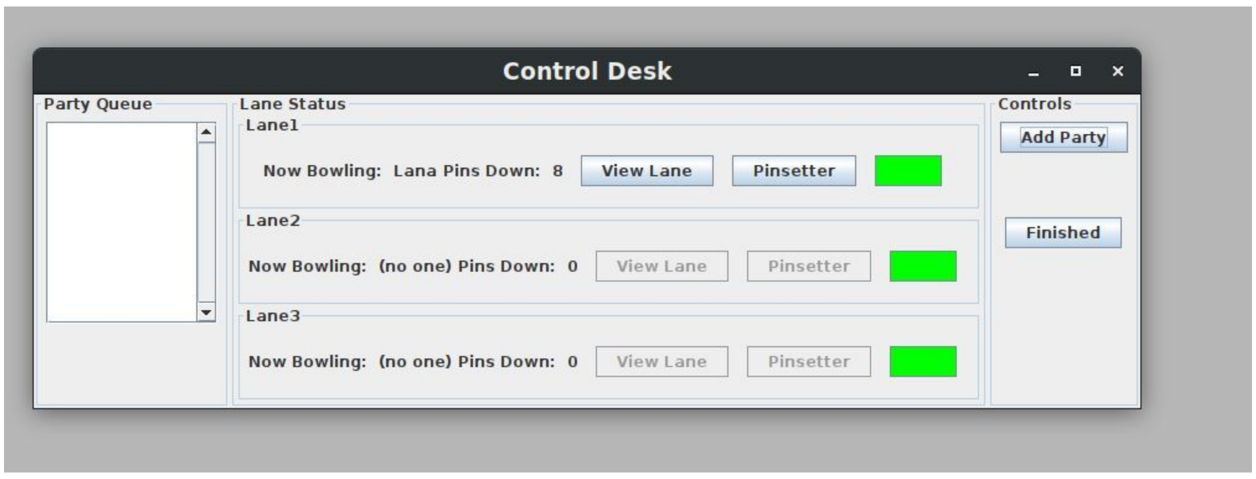
Your Party

(Empty)

Bowler Database

Mike
Jim
Tom
Lana
TomH

The Control Desk, which monitors the number of frames completed by each bowler

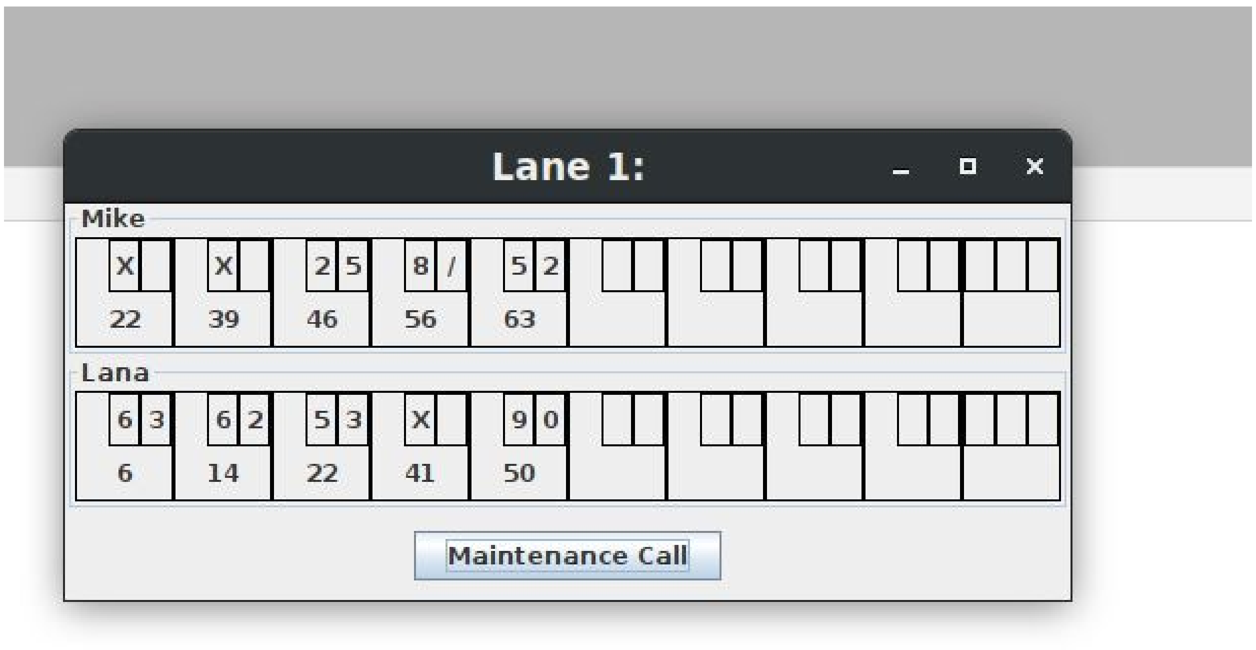


Lane1
Now Bowling: Lana Pins Down: 8 View Lane Pinsetter

Lane2
Now Bowling: (no one) Pins Down: 0 View Lane Pinsetter

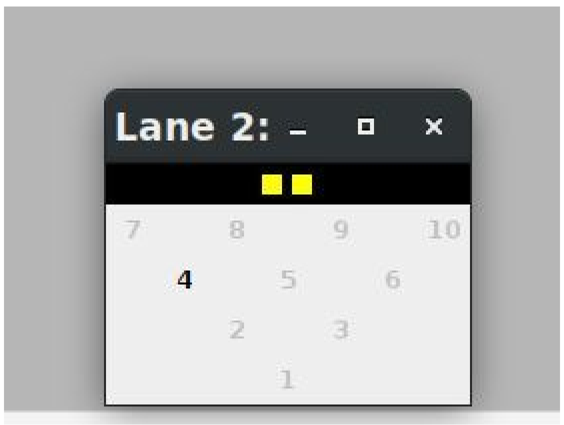
Lane3
Now Bowling: (no one) Pins Down: 0 View Lane Pinsetter

The control desk showing a game in progress:



	1	2	3	4	5	6	7	8	9	10
Mike	X	X	2 5	8 /	5 2					
	22	39	46	56	63					
Lana	6 3	6 2	5 3	X	9 0					
	6	14	22	41	50					

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



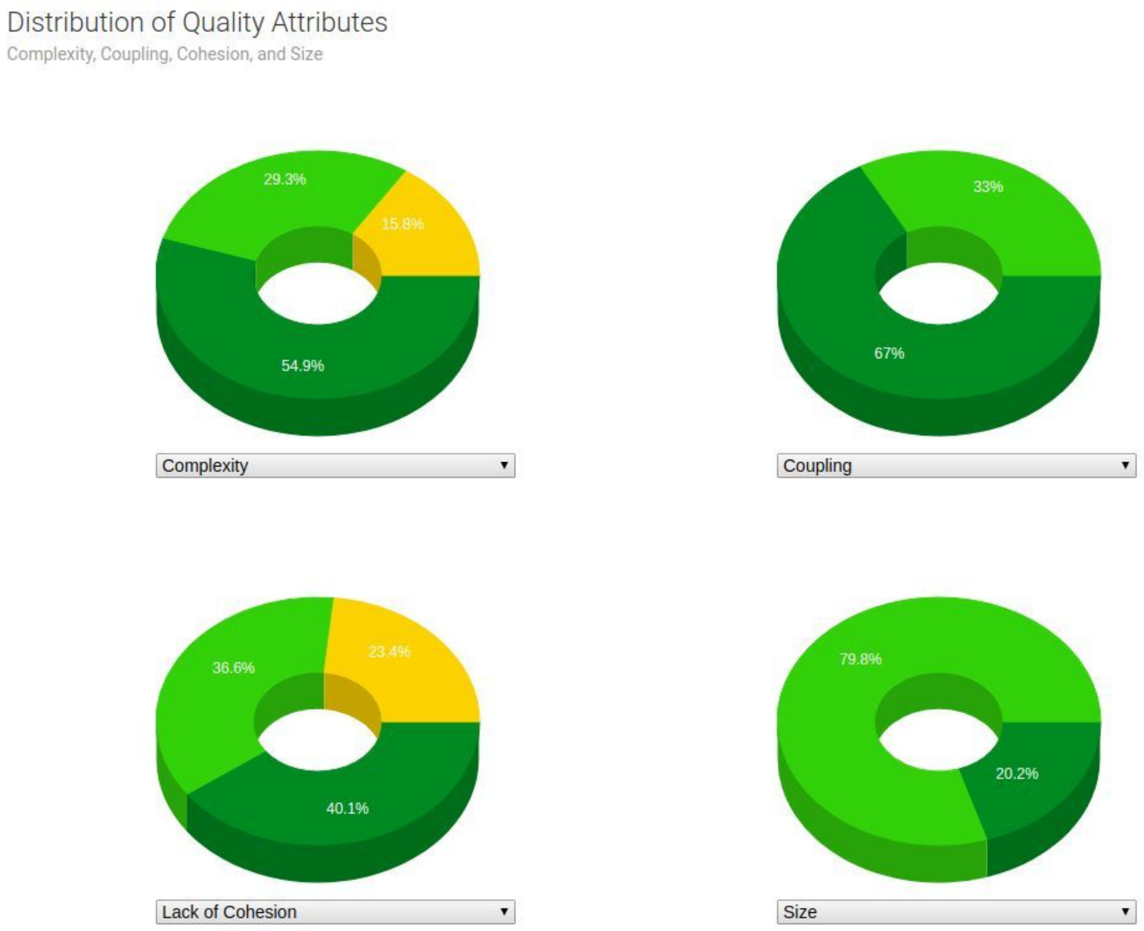
This is the pinsetter interface, which communicates to the scoring station the pins that are left standing after a bowler has completed a throw.

These interfaces are the main features of this game, making it very interactive. There are different windows and buttons too.

For our refactoring, we used the plugin, **CodeMR** and **metrics2**, to analyze metrics of the codebase.

Accordingly, different classes and methods were modified in order to ensure a balance between all metrics. Overall, there are about 28 metrics in CodeMR (Complexity, coupling, Lack of cohesion being some of them).

Original design



The graphs above depict the distribution of some of the metrics.

The original codebase had several weaknesses which were later improved in our design.

For example, there were several instances of repetition of code blocks in methods, which could have been easily replaced with a new method.

The initial design had many metrics wherein the percentage of the graph displaying yellow was high. More the percentage of green in the graph, the better is the metric.The graphs above display that the design had many classes with a Lack of Cohesion (23.4%).

The same can be observed for other metrics too. This indicates that there was not a very clear balance between different criteria, i.e, while a particular criteria was good, others were compromised on.

Strengths of the existing code:

All tasks and functionalities have been well-defined and clearly distributed into corresponding classes, making it easy for the developer to understand and refactor the code.

The codebase has several classes wherein one can observe the Observer pattern, which is basically seen in most processes where I/O handling is involved.

For example, in the file NewPatron.java, we are creating a “new patron” to be added to the game, which involves the user entering the new patron’s details in the fields created in the JPanel. This involves a subject-observer implementation, wherein the subject automatically notifies the observers of state changes, if any.

Codebases was using Observer Pattern Design which is a very good design pattern for such codes.

Responsibilities of Major classes:

Sr.No	Class Name	Class Variables	Functions	Responsibility

1.	AddPartyView	maxSize
		win
		addPatron
		remPatron
		finished
		partyList
		allBowlers
		Party
		Bowlerdb
		lock
			actionPerformed()
			valueChanged()
			getNames()
			updateNewPatron()
			getParty()	

2.	Alley	controlDesk	getControlDesk()	Creates an object of the ControlDesk

3.	Bowler	fullName
		nickName
		email
			getNickName()
			getFullName()
			getNick()
			getEmail()
			equals()
				It holds all the bowler's info.

4.	Bowlerfile		getBowlerInfo()
			putBowlerInfo()
			getBowlers()	

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

				user to click and answer.

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 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, 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

				EndGameReport. Score report is also generated by calling the ScoreReport 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()	

15.	NewPatronView	maxSize
		win
		bbort
		finished
		nickLabel
		fullLabel
		emailLabel
		nickField
		fullField
		emailField
		nick
		full
		Email
		done
		selectedNick
		selectedMember
		addParty
			actionPerformed()
			done()
			getNick()
			getFull()
			getEmail()	

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.

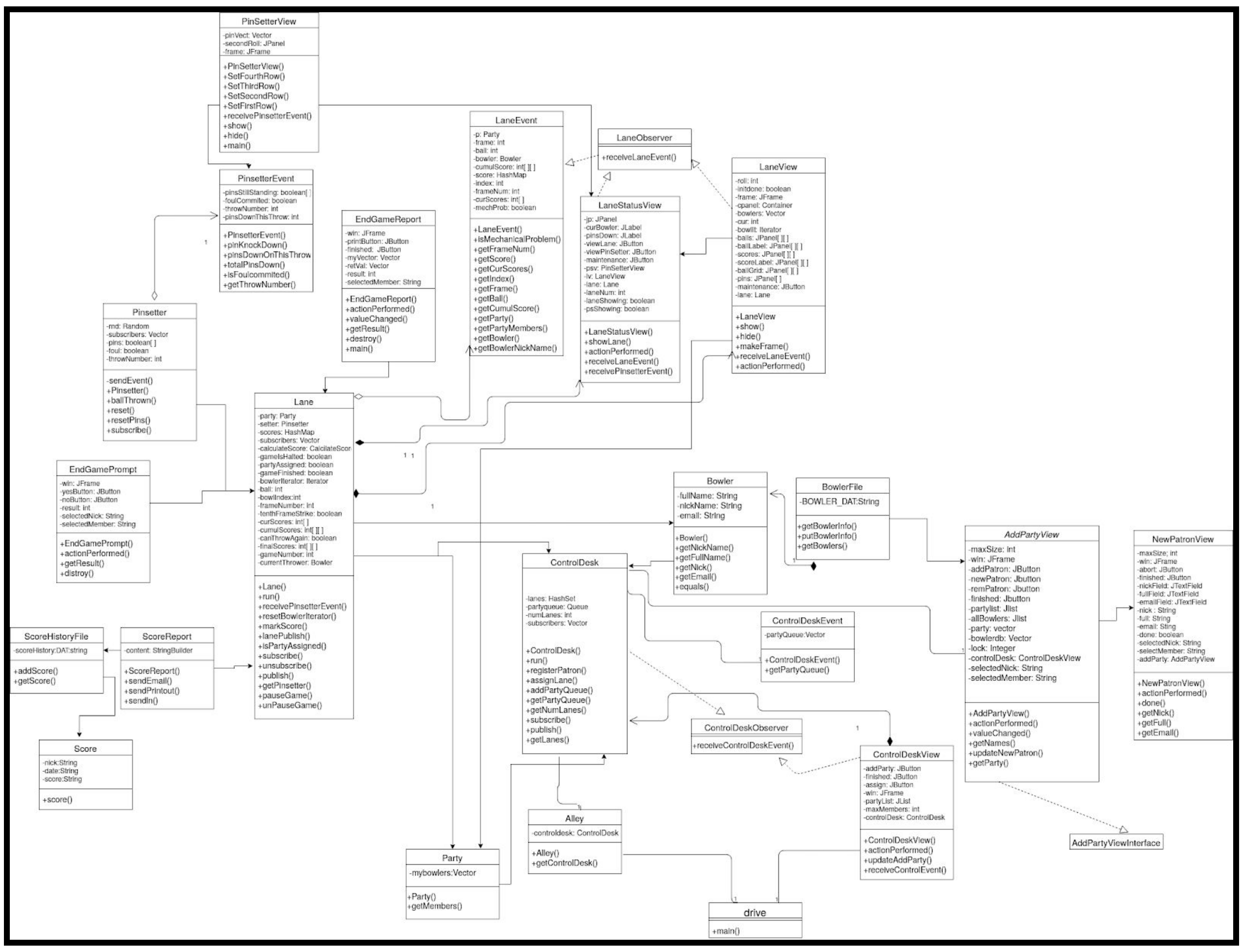
20.	Queue		next()
			add()
			hasMoreElements()
			asVector()
				This class is called to create the PartyQueue for a lane.

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()	

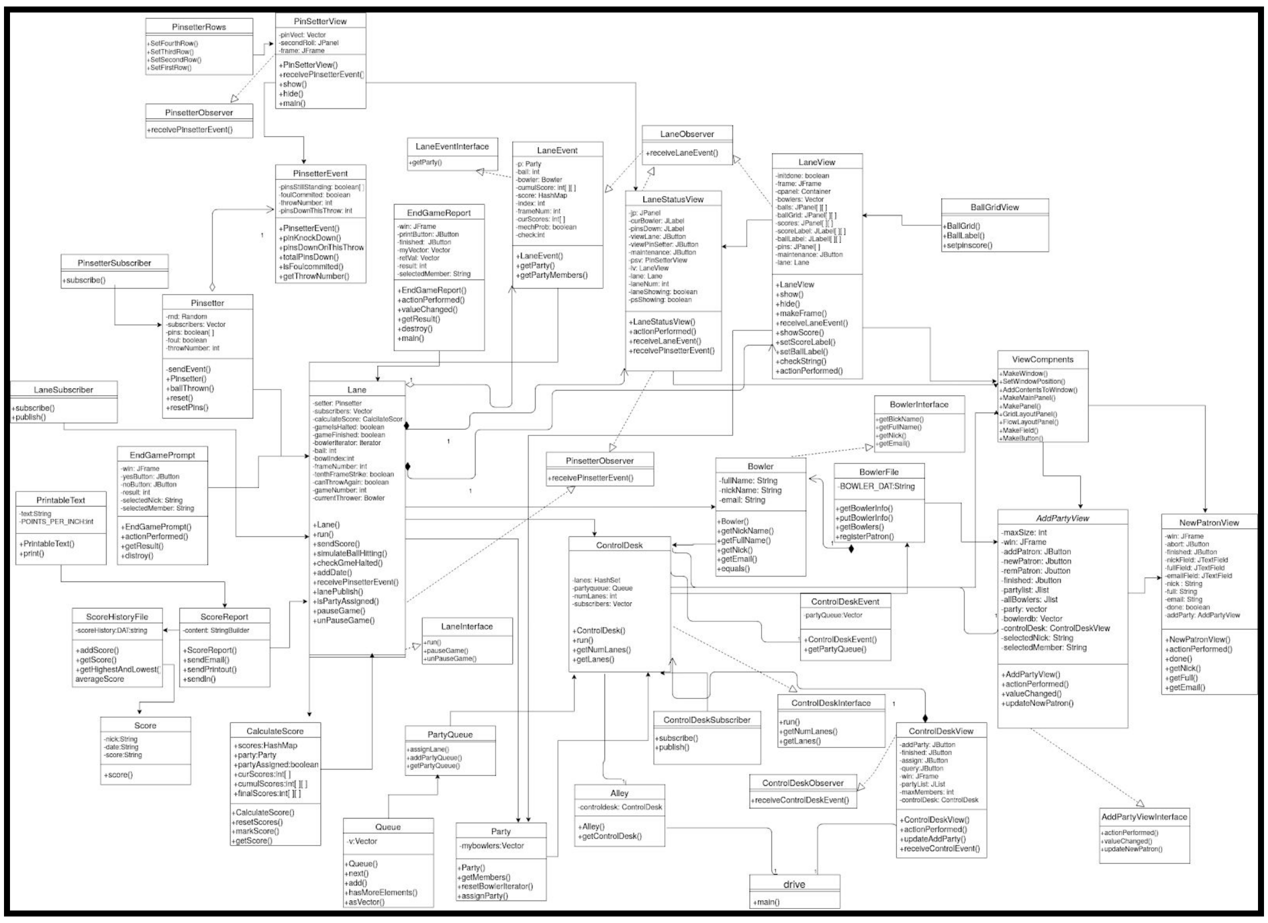
23.	ScoreReport	content	sendEmail()
			sendPrintout()
			sendln()
				This class displays the final scores, previous scores by date.
				This class sends the scores via email to the bowlers.

Original Class Diagram:



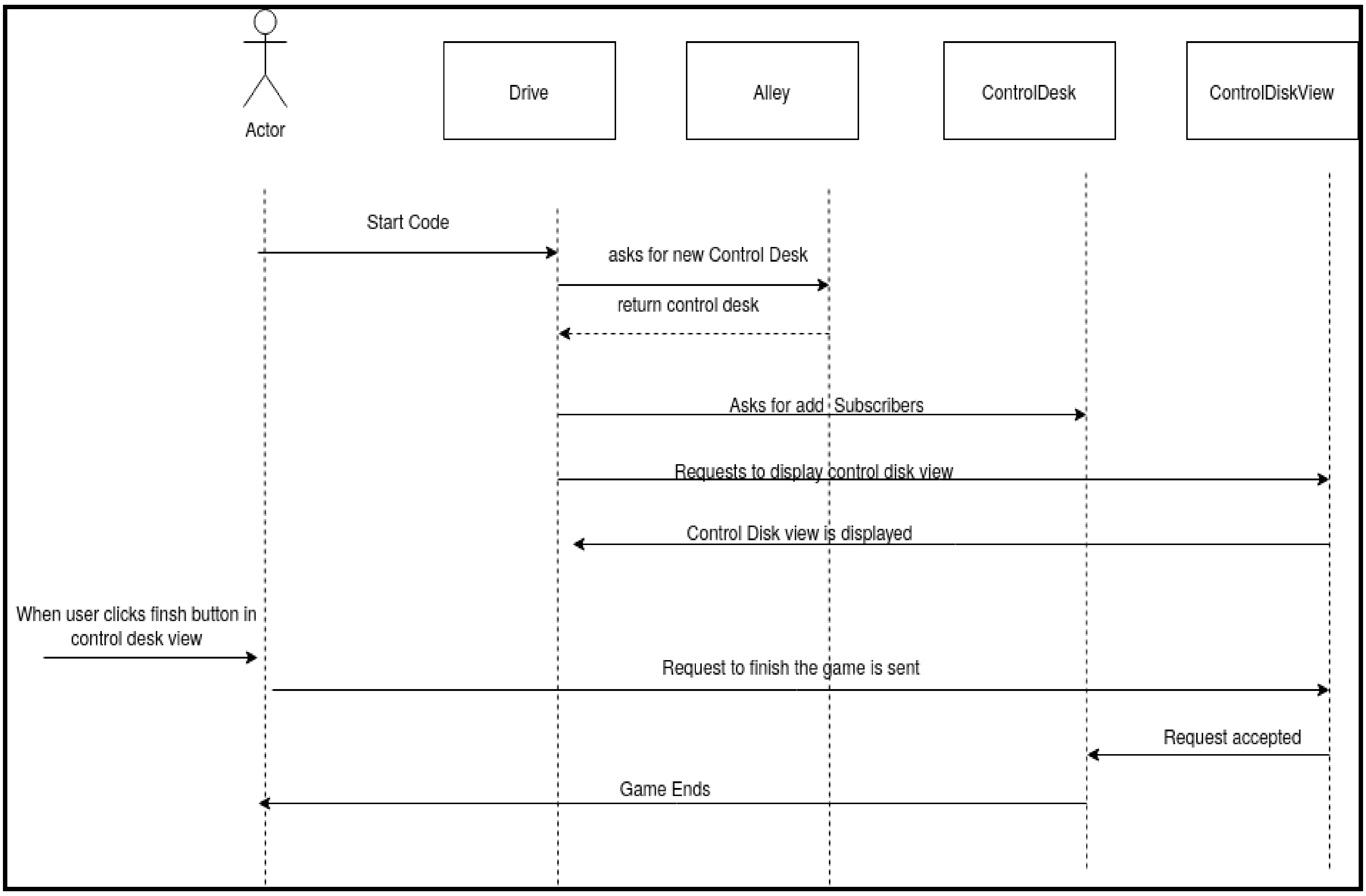
Please check link if not clear

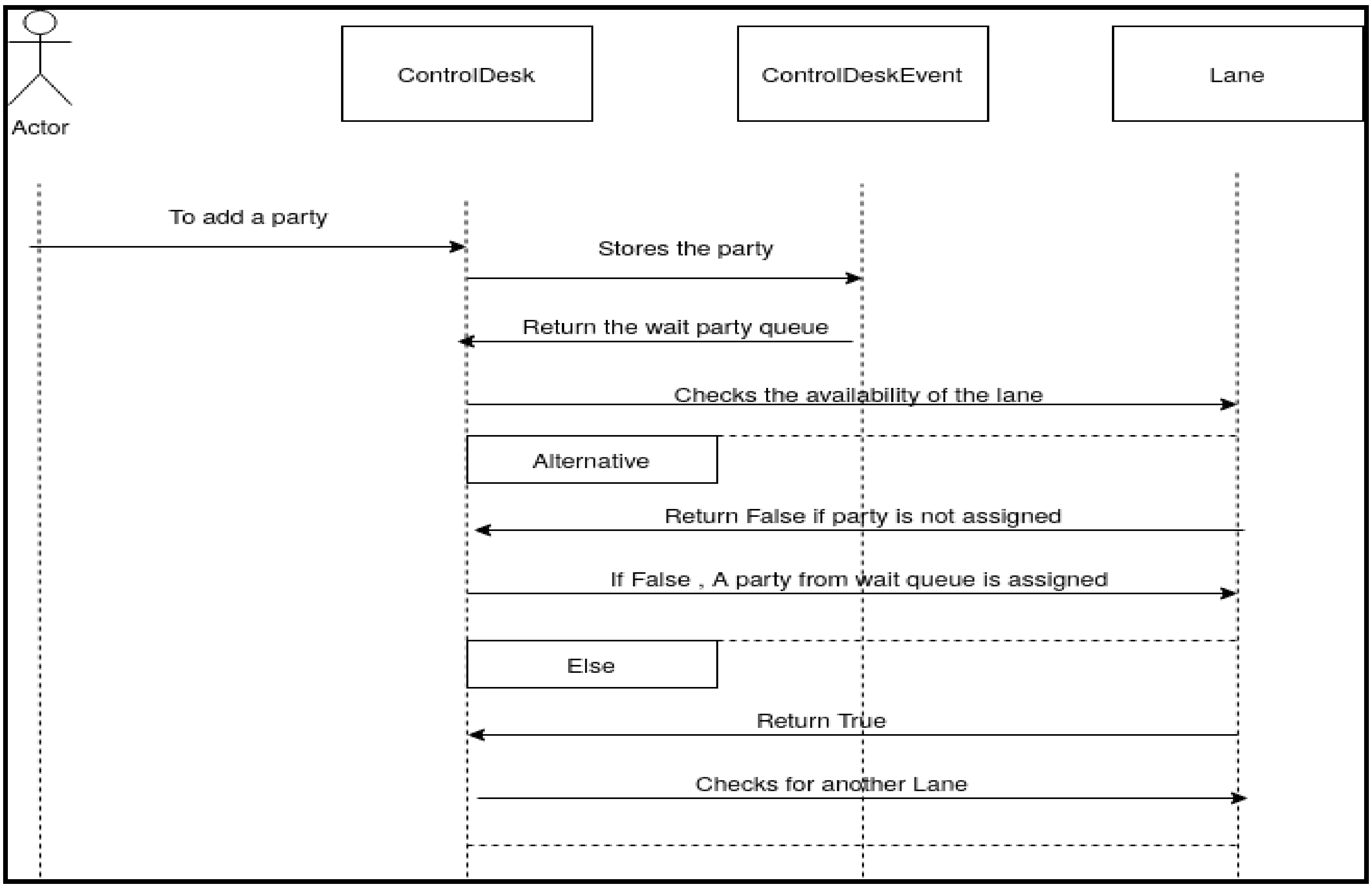
Refactored Class Diagram:

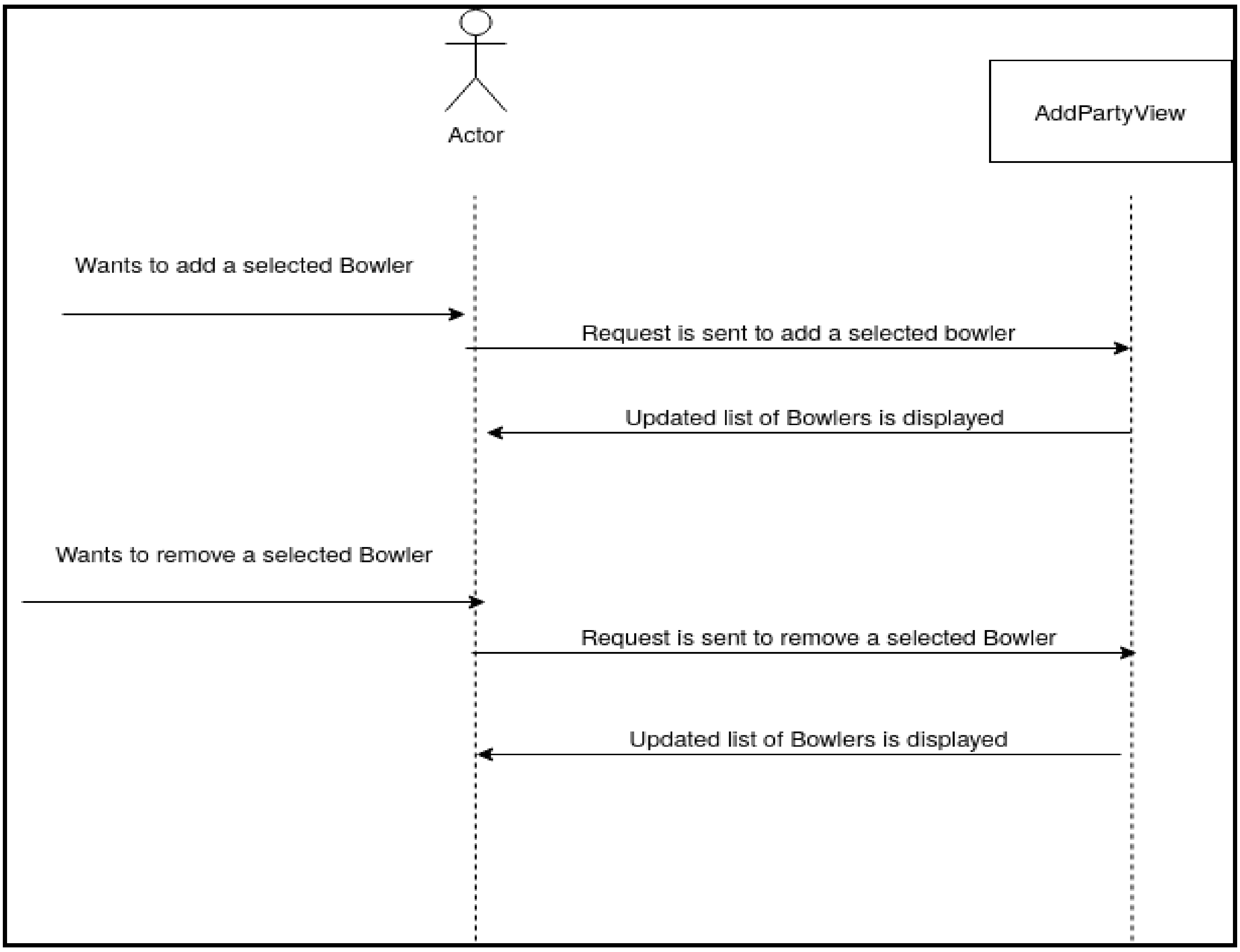


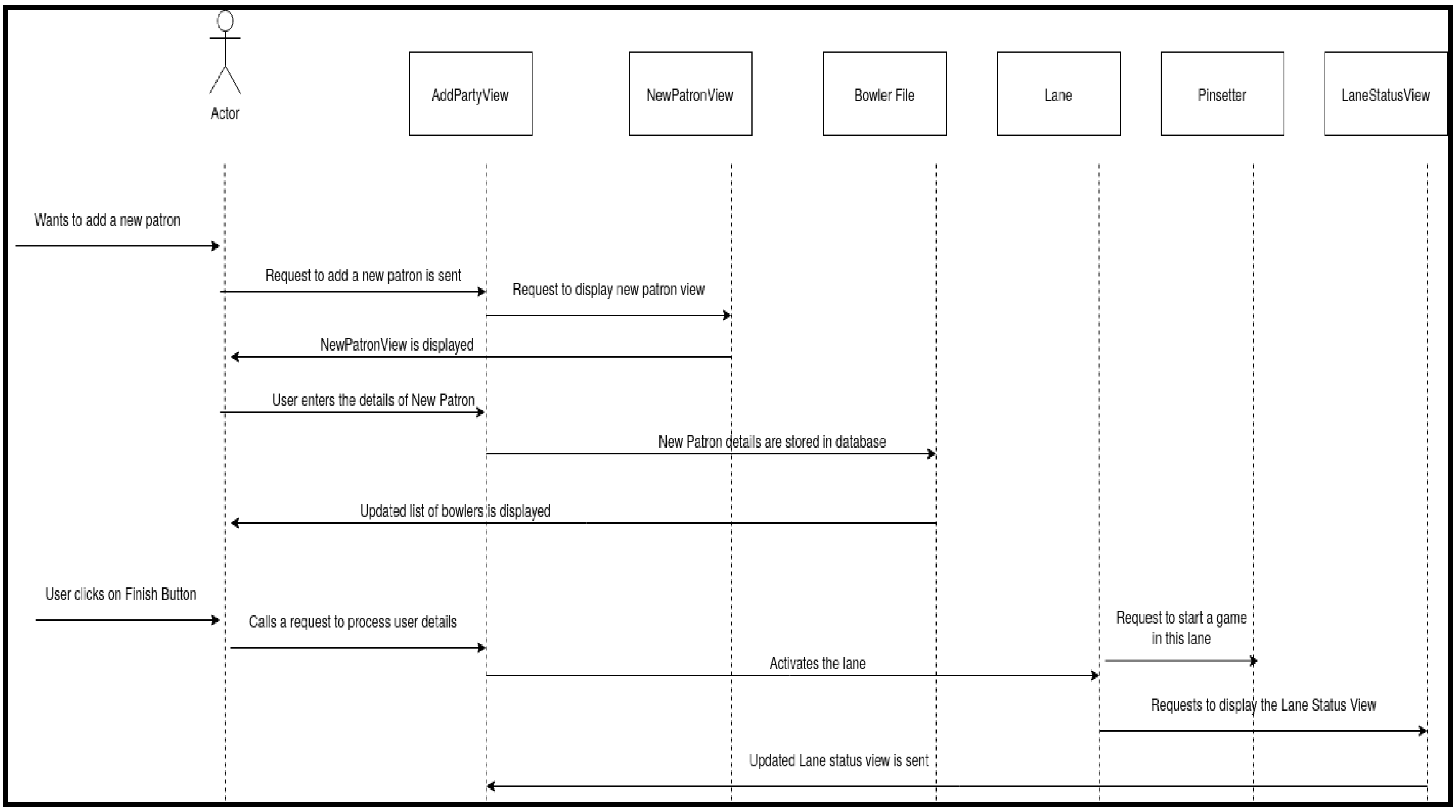
Please check link if not clear

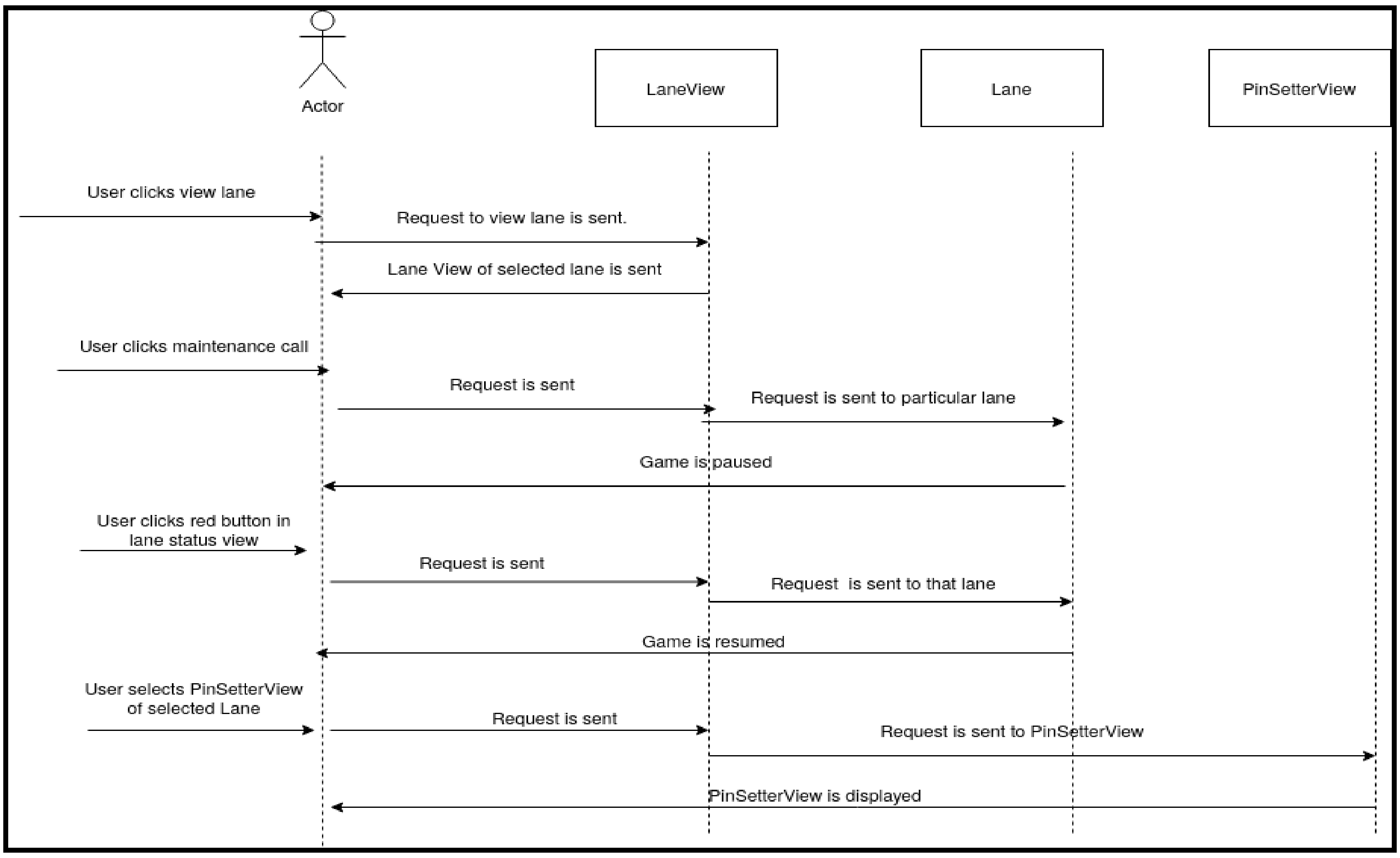
Sequence Diagrams:

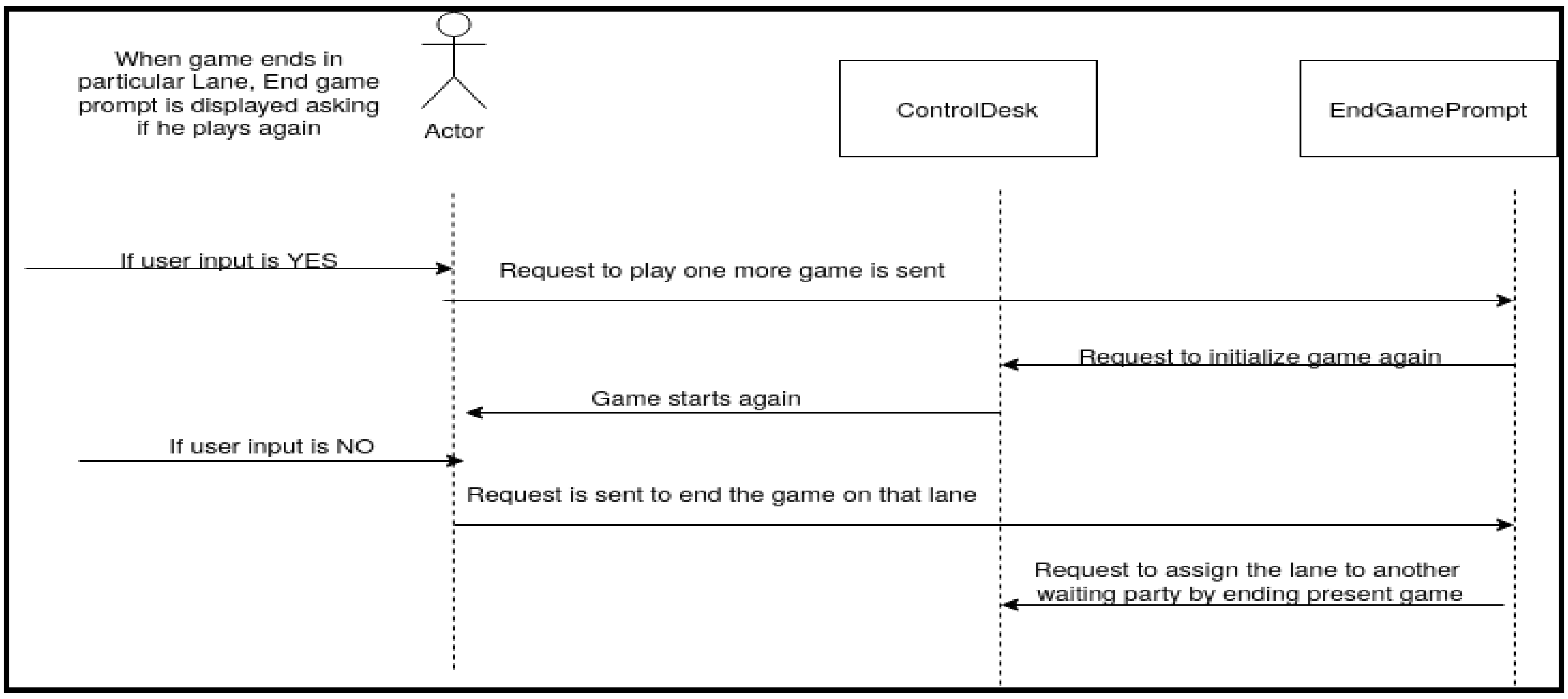

Actor
Drive
Alley
ControlDesk
ControlDiskView
Start Code
asks for new Control Desk
return control desk
Asks for add Subscribers
Requests to display control disk view
Control Disk view is displayed
When user clicks finsh button in control desk view
Request to finish the game is sent
Request accepted
Game Ends


Actor
ControlDesk
ControlDeskEvent
Lane
To add a party
Stores the party
Return the wait party queue
Checks the availability of the lane
Alternative
Return False if party is not assigned
If False , A party from wait queue is assigned
Else
Return True
Checks for another Lane


Actor
AddPartyView
Wants to add a selected Bowler
Request is sent to add a selected bowler
Updated list of Bowlers is displayed
Wants to remove a selected Bowler
Request is sent to remove a selected Bowler
Updated list of Bowlers is displayed


Actor
AddPartyView
NewPatronView
Bowler File
Lane
Pinsetter
LaneStatusView
Wants to add a new patron
Request to add a new patron is sent
Request to display new patron view
NewPatronView is displayed
User enters the details of New Patron
New Patron details are stored in database
Updated list of bowlers is displayed
User clicks on Finish Button
Calls a request to process user details
Request to start a game in this lane
Activates the lane
Requests to display the Lane Status View
Updated Lane status view is sent

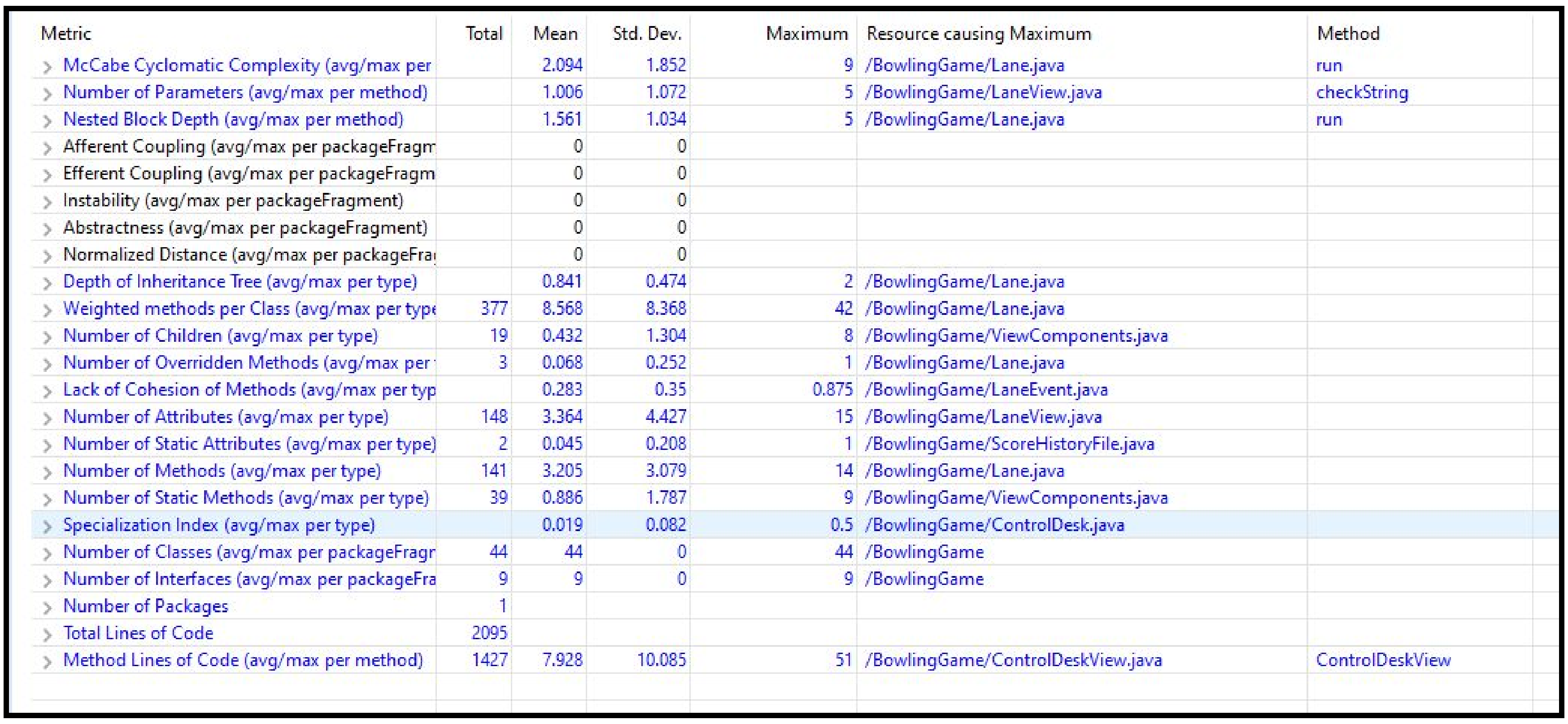

Actor
LaneView
Lane
PinSetterView
User clicks view lane
Request to view lane is sent.
Lane View of selected lane is sent
User clicks maintenance call
Request is sent
Request is sent to particular lane
Game is paused
User clicks red button in lane status view
Request is sent
Request is sent to that lane
Game is resumed
User selects PinSetterView of selected Lane
Request is sent
Request is sent to PinSetterView
PinSetterView is displayed


When game ends in particular Lane, End game prompt is displayed asking if he plays again
Actor
ControlDesk
EndGamePrompt
If user input is YES
Request to play one more game is sent
Request to initialize game again
Game starts again
If user input is NO
Request is sent to end the game on that lane
Request to assign the lane to another waiting party by ending present game

Metric Analysis:

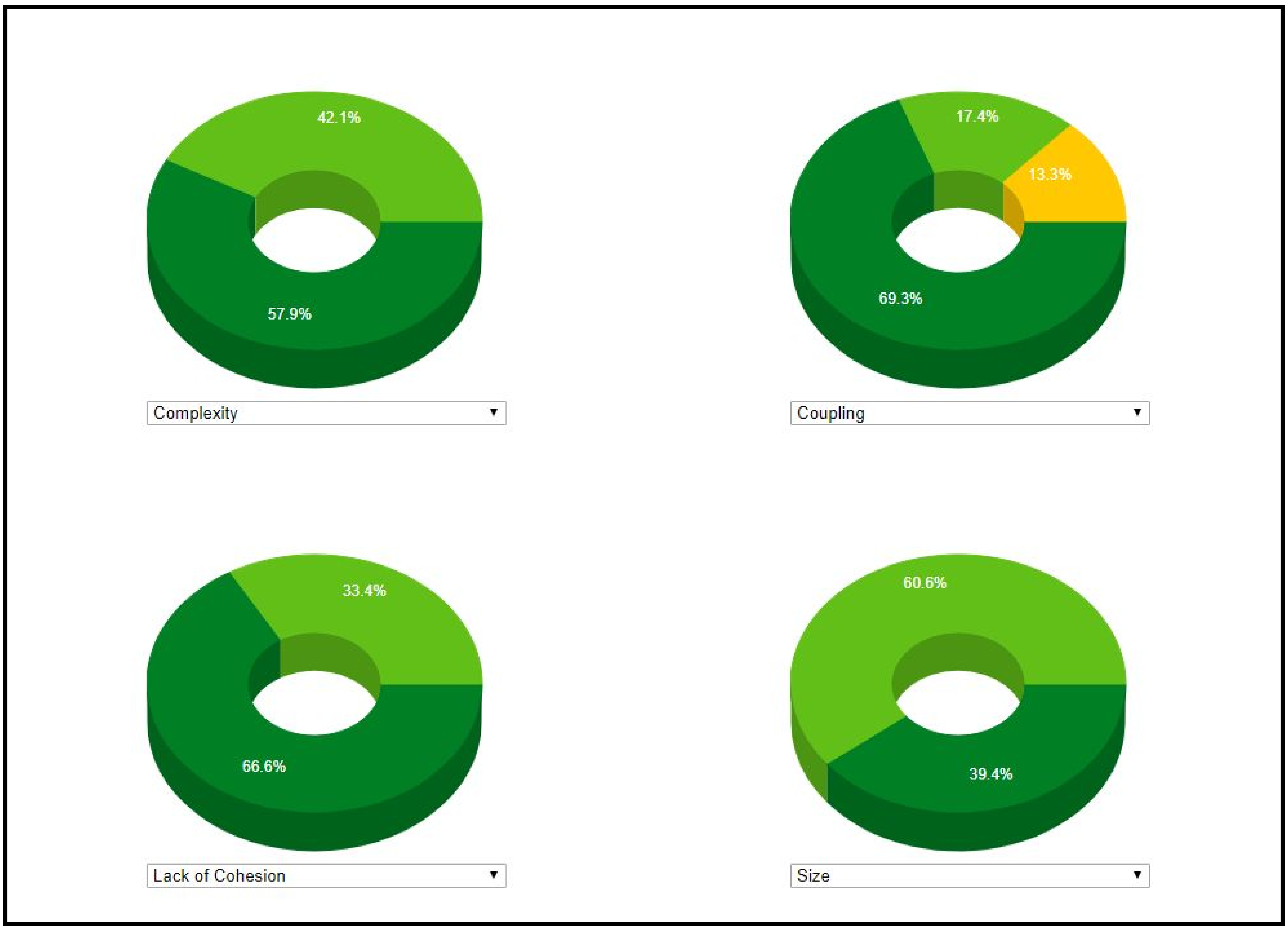
Discussion of Particular Code Smells:

- Cyclomatic Complexity
    - This is a quantitative measure of the number of linearly independent paths through a
    - program's source code.
    - The main problematic files were Lane, LaneView, LaneStatusView and
    - AddPartyView.
    - We were able to successfully bring down metrics of all these files by refactoring.
- Number of Parameters
    - The number of parameters passed amongst methods.
    - We viewed the problematic files and brought down the metric for methods in Lane,
    - Pinsetter event etc. Thus the mean metric has decreased.
    - However we were not able to limit the arguments in the most problematic file which
    - was the constructor of LaneEvent having 9 parameters.
- Nested Block Depth
    - The depth of code statements, lesser means better.
    - Lane, LaneView etc were the problematic files having depth upto 7.
    - We were able to refactor the conditional statements in these files to bring down their
    - depths of complexity. Depth of LaneStatusView was brought down to 2.
- Weighted Methods Per Class
    - The number of methods per class calculated in a weighted fashion for the overall
    - code decreased.
    - This was problematic in Lane, by smartly decomposing the functions we were able to
    - bring down this metric in this file.
- Number of Methods
    - This metric increased slightly since in order to decrease cyclomatic complexity we
    - increased the number of functions.
    - This was also affected by cases of increasing levels of abstraction and improving
    - readability of the code.
- Number of Attributes
    - This metric was relatively unchanged since it was already in the optimum range
    - according to the metrics standards.
- Number of Classes
    - This metric was initially 29 classes in total.
    - We decreased this metric by eliminating two data classes in the project.
- Method Lines of Code
    - This metric was decreased drastically.
    - This was achieved by splitting methods into meaningful subcomponents to increase



Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
McCabe Cyclomatic Complexity (avg/max per		2.094	1.852	9	/BowlingGame/Lane.java	run
Number of Parameters (avg/max per method)		1.006	1.072	5	/BowlingGame/LaneView.java	checkString
Nested Block Depth (avg/max per method)		1.561	1.034	5	/BowlingGame/Lane.java	run
Afferent Coupling (avg/max per packageFragm		0	0			
Efferent Coupling (avg/max per packageFragm		0	0			
Instability (avg/max per packageFragment)		0	0			
Abstractness (avg/max per packageFragment)		0	0			
Normalized Distance (avg/max per packageFra		0	0			
Depth of Inheritance Tree (avg/max per type)		0.841	0.474	2	/BowlingGame/Lane.java	
Weighted methods per Class (avg/max per type	377	8.568	8.368	42	/BowlingGame/Lane.java	
Number of Children (avg/max per type)	19	0.432	1.304	8	/BowlingGame/ViewComponents.java	
Number of Overridden Methods (avg/max per	3	0.068	0.252	1	/BowlingGame/Lane.java	
Lack of Cohesion of Methods (avg/max per typ		0.283	0.35	0.875	/BowlingGame/LaneEvent.java	
Number of Attributes (avg/max per type)	148	3.364	4.427	15	/BowlingGame/LaneView.java	
Number of Static Attributes (avg/max per type)	2	0.045	0.208	1	/BowlingGame/ScoreHistoryFile.java	
Number of Methods (avg/max per type)	141	3.205	3.079	14	/BowlingGame/Lane.java	
Number of Static Methods (avg/max per type)	39	0.886	1.787	9	/BowlingGame/ViewComponents.java	
Specialization Index (avg/max per type)		0.019	0.082	0.5	/BowlingGame/ControlDesk.java	
Number of Classes (avg/max per packageFragr	44	44	0	44	/BowlingGame	
Number of Interfaces (avg/max per packageFra	9	9	0	9	/BowlingGame	
Number of Packages	1					
Total Lines of Code	2095					
Method Lines of Code (avg/max per method)	1427	7.928	10.085	51	/BowlingGame/ControlDeskView.java	ControlDeskView

(View of metrics in Eclipse using Metrics2)



(View of metrics in Eclipse using CodeMR - coupling is outdated)

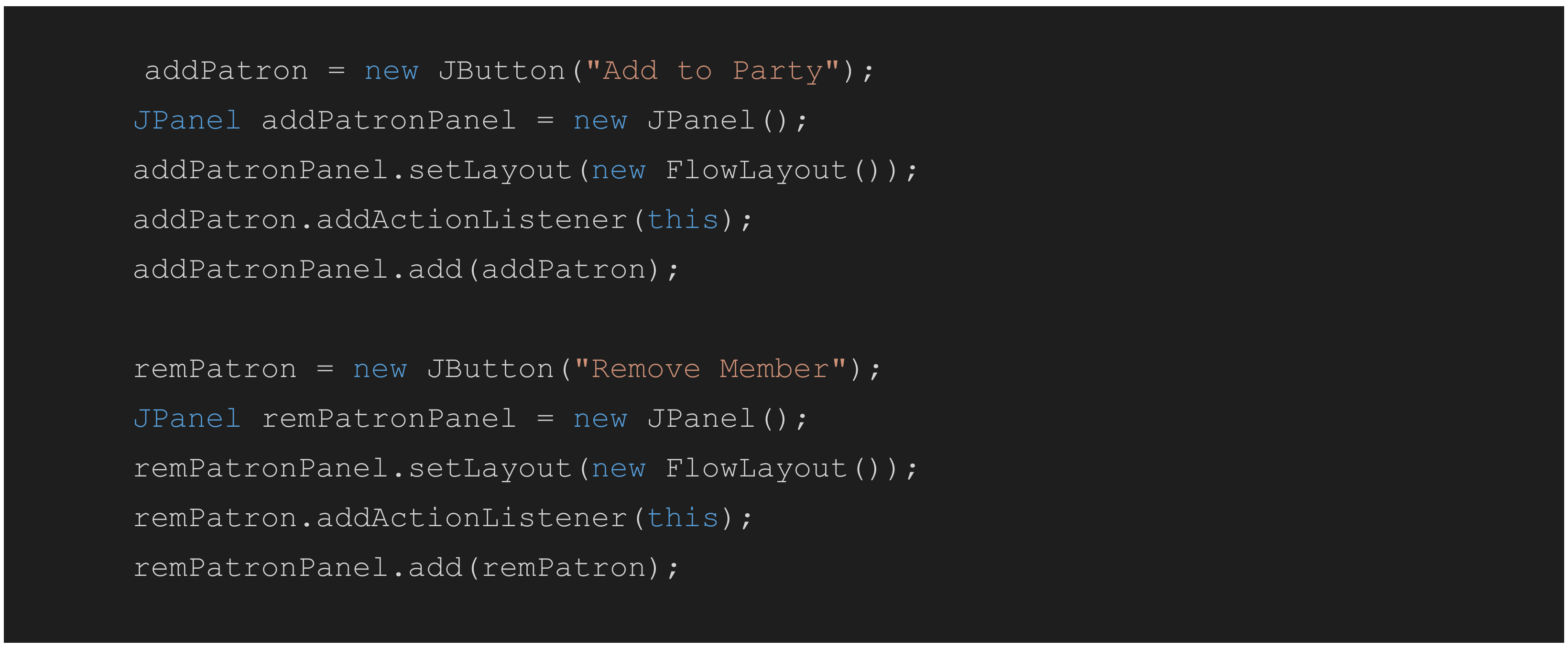
Refactoring :

1. **Code Repetition** : Code Repetition as the name suggests is a repetition of a line or a block of code in the same file or sometimes in the same local environment. People might consider code duplication acceptable but in reality, it poses greater problems to software than what we may have thought. Even code that has similar functionalities are said to be duplications. The main reason for creation for the duplicate code is Copy and Paste Programming.

Issue :

Initial code has many cases of duplication. 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 the same code is repeated for making buttons. So this could be easily made into a function and function can be called multiple times with some parameters.

Problems with previous style : Code which includes duplicate functionality is more difficult to support,

- simply because it is longer, and
- because if it needs updating, there is a danger that one copy of the code will be updated without further checking for the presence of other instances of the same code.

Solution :

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

```
public static JButton MakeButtons(String st, JPanel buttonPanel){
      JButton btn = new JButton(st);
      JPanel btnPanel = new JPanel();
      btnPanel.setLayout(new FlowLayout());
      btnPanel.add(btn);
      buttonPanel.add(btnPanel);
      return btn;
 }
addPatron = MakeButtons("Add to Party",buttonPanel);
```

But after watching all files we noticed that this function was used in not only one but many files and also many functions like this ex. ***Making of Panel, Making of TextField*** etc were repeated multiple times in many files in order to set components on the screen. So we made a centralized class for this name ***ViewComponents,*** this class contains all such functions which are used to show components on the screen. So now we called functions from this file instead of writing functions multiple times.

```
addPatron = ViewComponents.MakeButtons("Add to Party",buttonPanel);
```

In this way we reduce ***code size*** a lot which is one of major metrics for code evaluation.

Advantages :

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

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

   Issue:

3. **Missing Comments :** A comment is a programmer-readable explanation or annotation in the source code of a computer program.Comments are used for integration with source code management systems and other kinds of external programming tools.

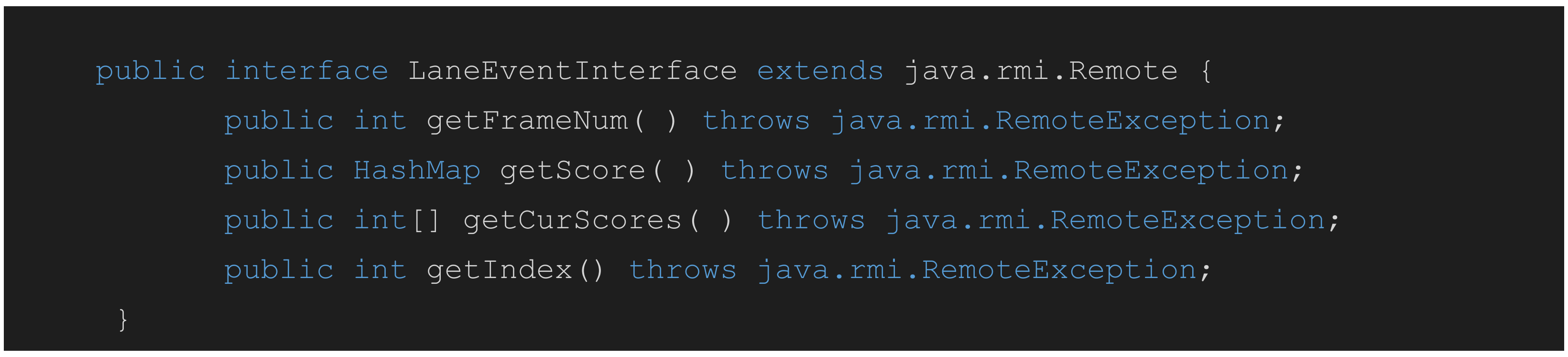
   Issue:

4. **Unnecessary Public Modifier in Interfaces :** In Interfaces public modifier was used with many functions. But all methods in an interface are already

public and abstracts so there was no need to make them public explicitly. Therefore we removed the public modifier from these interfaces.

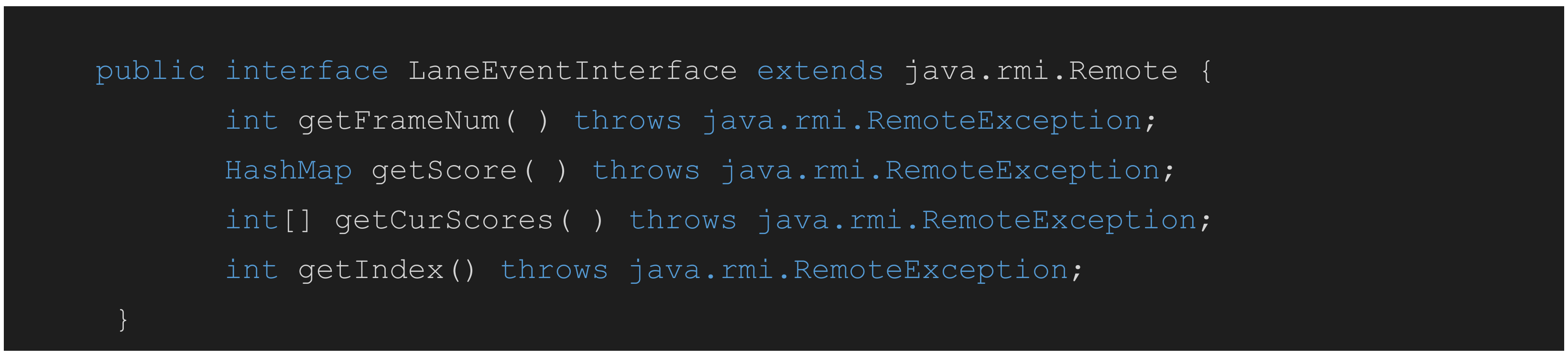
LaneEventInterface.java

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;
 }
```

Advantages of Interfaces being public :

- Fields and methods are implicitly public because the point of an interface is to declare an ... interface that other classes can see. (If you want / need to restrict access, this is done via an access modifier on the interface itself.)
- Fields are static because if they were not you would be declaring visible instance fields on an object ... and that's bad for encapsulation.
- Fields are final because non-final fields would be another way of declaring public static fields ... which are terrible from an OO perspective.
- Methods are abstract because allowing method bodies would effectively turn interfaces into abstract classes.

5. **Removed Unnecessary Imports :**The imports part of a file should be handled by the Integrated Development Environment (IDE), not manually by the developer.Unused and useless imports should not occur if that is the case.Leaving them in reduces the code's readability, since their presence can be confusing.

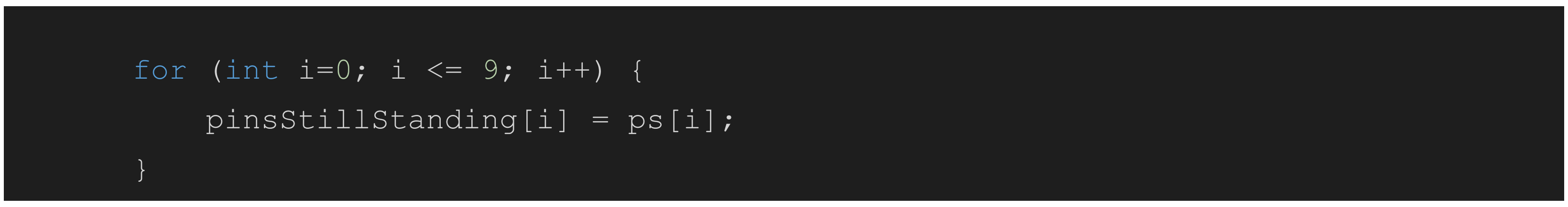
Issue:

6. **Removed Redundant Casting to various fields :** In many files unnecessarily many fields were casted to other fields. Unnecessary casting expressions make the code harder to read and understand.

7. **Redundant manual array Copy :** In file ***PinsetterEvent.java*** to copy a array manual for loop was written instead a copy function should be used because it helps to prevent bugs.

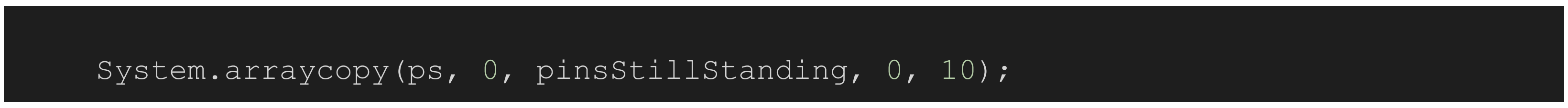
   PinsetterEvent.java

   Before Refactor :



```
for (int i=0; i <= 9; i++) {
    pinsStillStanding[i] = ps[i];
}
```

   ***After* Refactor :**

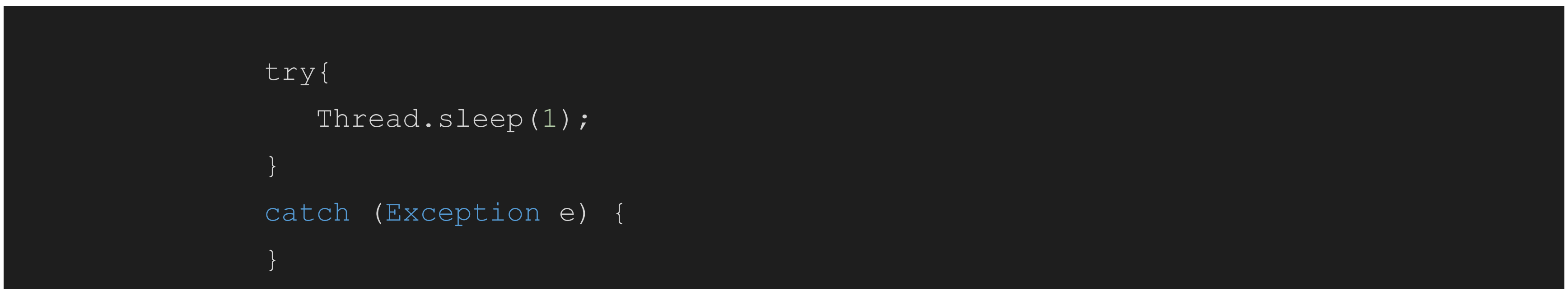


```
System.arraycopy(ps, 0, pinsStillStanding, 0, 10);
```

8. **Empty Catch statements :** In many files **Catch statements** were empty. It can cause problems if catched code would not print anything so we would not be able to debug the code. So appropriate errors were printed. Ex:

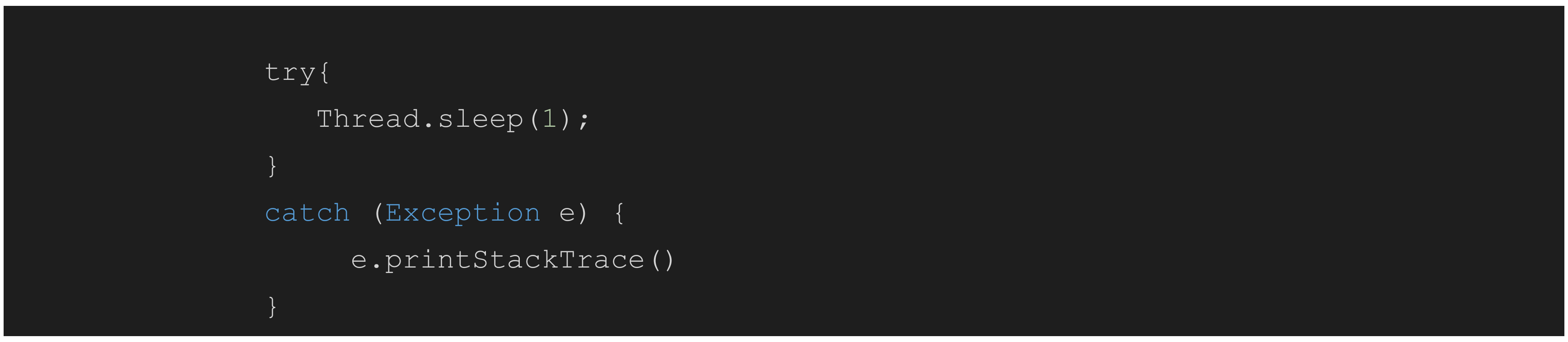
   LaneView.java

   Before Refactor :



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

   ***After* Refactor :**



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

9. **Law of Demeter :** Law of Demeter also known as principle of least knowledge is a coding principle, which says that a module should not know

about the inner details of the objects it manipulates. If a code depends upon internal details of a particular object, there is a good chance that it will break as soon as the internal of that object changes. Since Encapsulation is all about hiding internal details of objects and exposing only operations, it also asserts the Law of Demeter. The Law of Demeter for functions (or methods, in Java) attempts to minimize **coupling** between classes in any program. In short, the intent of this “law” is to prevent you from reaching into an object to gain access to a third object’s methods. The Law of Demeter is often described this way:

“Only talk to your immediate friends.”

or, put another way:

“Don't talk to strangers.”

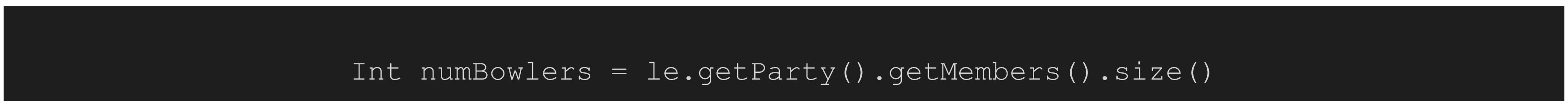
According to Law of Demeter a method M of object O should only call following types of methods :

- Methods of Object O itself
- Methods of Object passed as an argument
- Method of object, which is held in instance variable
- Any Object which is created locally in method M

More importantly, methods should not invoke methods on objects that are returned by any subsequent method calls specified above and as Clean Code says "talk to friends, not to strangers".

Let’s explain this using a example from file ***LaneView.java :***

Before Refactor :



```
Int numBowlers = le.getParty().getMembers().size()
```

In this what is happening is that ***le.getParty()*** returns a Party associated with LaneEvent and ***getMembers()*** function returns a vector of bowlers. So here from file ***LaneView*** we have first called method on its own objects (***getParty()***) but after that we have called a method on an object of our object which is what causes a problem. The problem is that from a given file we are accessing properties of this class’s object. Therefore here we are accessing Strangers and not friends as explained above .So in final code after refactoring we used a function ***getPartyMembers()*** which is called from this file and is a method of this class object and this directly returns Vector of Bowlers of objects Party. So this way we are not accessing objects’ object methods.

After Refactor :

```
            Int numBowlers = le.getPartyMembers().size()
```

Advantages : The advantage of following the Law of Demeter is that the resulting software tends to be more maintainable and adaptable. Since objects are less dependent on the internal structure of other objects, object containers can be changed without reworking their callers.

10. **Assigning appropriate functions to appropriate files :** In many files a lot of varying variety of functions were written whereas according to good coding habits a file should contain all related functions. This causes problem of
    Lack of Cohesion : Cohesion metrics measure how well the methods of a class are related to each other. A cohesive class performs one function. A non-cohesive class performs two or more unrelated functions. A non-cohesive class may need to be restructured into two or more smaller classes. 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 the most messed up file in the entire code it was doing a lot of different types of functions which were not even related. Ex. No 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 ***CalculateScore.java*** and moved all such related functions to calculated score in that file with appropriate parameters. Following functions from **Lane** file were moved to **CalculateScore** file :

    - resetScore()
    - markScore()
    - CalculateScore()

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.

Same thing happened in many other files ex : In **Lane.java** there was a function to assign Party named **assignParty()** , we moved that function to **Party.java** as it should be in Party file to assign a party. There are examples of this thing in many other files too.

Advantages :

- Reduced module complexity (they are simpler, having fewer operations).
- Increased system maintainability, because logical changes in the domain affect fewer modules, and because changes in one module require fewer changes in other modules.
- Increased module reusability, because application developers will find the component they need more easily among the cohesive set of operations provided by the module.

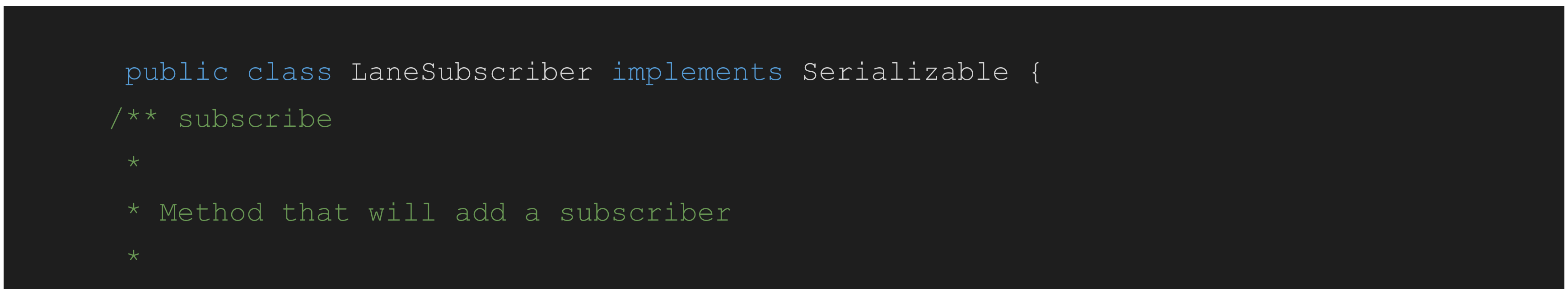
11. Properly Implementing ObserverPattern Design :

The Observer Pattern defines a one to many dependency between objects so that one object changes state, all of its dependents are notified and updated automatically.

- One to many dependency is between Subject(One) and Observer(Many).
- There is dependency as Observers themselves don’t have access to data. They are dependent on Subject to provide them data.

In this project ObserverPattern Design pattern was followed for subscribing and publishing various events which is obviously a very good design pattern to use. But it was not implemented properly. Issue was that all the task of Subscribing and Publishing was done in the same files were everything else was done whereas according to good coding principles all this part of publishing and subscribing should be present in a centralized file which should be used by every other file. So for that we made new files Ex **LaneSubscriber** which as its name suggests does only subscribing and publishing part.

LaneSubscriber.java



```
 public class LaneSubscriber implements Serializable {
/** subscribe
 *
 * Method that will add a subscriber
 *
```

```
     */
    public static void subscribe(Lane lane,LaneObserver adding ) {
        lane.subscribers.add( adding );
    }

    /** publish
     *
     * Method that publishes an event to subscribers
     *
     * @param event Event that is to be published
     */
    public static  void publish( Lane lane,LaneEvent event ) {
        if( lane.subscribers.size() > 0 ) {
            for (Object subscriber : lane.subscribers) {
                ((LaneObserver) subscriber).receiveLaneEvent(event);
            }
        }
    }
}
```

Advantages :

Provides a loosely coupled design between objects that interact. Loosely coupled objects are flexible with changing requirements. Here loose coupling means that the interacting objects should have less information about each other.

Observer pattern provides this loose coupling as:

- Subject only knows that the observer implements the Observer interface.Nothing more.
- There is no need to modify Subject to add or remove observers.
- We can reuse subject and observer classes independently of each other.

12. **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.So we reduced the number of parameters by encapsulating them in a new object.

    Advantages:

    It is generally not a good idea to make "junk drawer" objects that *couple* unrelated parameters whose only relationship to one another is that they need to be passed to the same method or constructor. However, when related parameters are being passed to a constructor or method as part of a highly *cohesive* object, the refactoring known as Introduce Parameter

Object is a nice solution. This refactoring's usage is described as "group[ing] of parameters that naturally go together."

13. **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).

14. **Using enhanced for loop :** Enhanced for loop increases the abstraction level - instead of having to express the low-level details of how to loop around a list or array (with an index or iterator), the developer simply states that they want to loop and the language takes care of the rest. However, all the benefit is lost as soon as the developer needs to access the index or to remove an item. A common coding idiom is expressed at a higher abstraction than at present. This aids readability and clarity.
    Ex.

    ```
     for(Integer i : list){
       ....
    }
    ```

    is clearly better than something like

    ```
    for(int i=0; i < list.size(); ++i){
      ....
    }
    ```

    So we replace normal for loop with enhanced one in many files.

15. **Improving Performance :** In order to improve performance we replaced string concatenation using '+' in loops by function append. For that we replaced string by String builder. This has better performance because string concat always copies full string whereas string append works other way.
16. Made Some classes abstract for which we don’t want to make any objects.
17. **Low Coupling :** Modules should be as independent as possible from other modules, so that changes to a module don’t heavily impact other modules. High coupling would mean that your module knows the way too much about the inner workings of other modules. Modules that know too much about other modules make changes hard to coordinate and make modules brittle. 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, you can easily make changes to the internals of modules without worrying about their impact on other modules in the system. Low coupling also makes it 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:

1) through interfaces we can implement multiple inheritance in java.

2) Interfaces function to break up the complex designs and clear the dependencies between objects.

3) Interfaces make applications loosely coupled.