

I. BOWLING MANAGEMENT SYSTEM - DESIGN DOCUMENT :

Title : Bowling Management System **Date of Submission :** April 9, 2020

Team Members :

SNo.	Team Member	Hours of Work/day	Contributions
1.	Mallika Subramanian	15	<ul style="list-style-type: none"> Refactoring to reduce cyclomatic complexity Increasing cohesion amongst classes and methods Analysing and identifying responsibilities of major classes as well as the interlinked classes. Creating UML class and sequence diagrams for the original as well as refactored code
2.	Aryamaan Jain	15	<ul style="list-style-type: none"> Refactoring to reduce cyclomatic complexity Reducing number of methods per class Understanding the metrics to be measured and documenting potential changes that can be made to improve them Implementing the Database layer for ad-hoc queries and the save-quit and restart feature.
3.	E Nikhil	15	<ul style="list-style-type: none"> Identifying the critical code smells in the code Refactoring to get rid of redundant code Implementing the pause/resume Creating Sequence diagrams

II. BRIEF OVERVIEW :

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

- Control Desk :** The control desk operator has the ability to monitor the scores of any active lane. A configurable display option will allow the operator to view the score of an individual scoring station or multiple scoring stations.
- Creating a new player :** A *NewPatron* can be created to play the game. This player is then added to the Bolwers database file. The new player will then be eligible to join a party and play a round of the game.
- Adding a new party :** Selected number of bowlers can be added to a party which is then assigned to one of the free lanes to begin playing a game. In case all the lanes are occupied - the party is then added to the *Queue* that keeps track of the parties that are registered but yet to play.
- Viewing the Pinsetter :** For a particular lane, the user can also view the pinsetter window which simulates the pins dropped on each ball-throw. The pinsetter will re-rack the pins (places all ten down) after two consecutive throws have been detected.
- Viewing the Scoreboard :** The scoreboard keeps track of the score gained by each player in the party after their respective turns. It uses the normal score calculation technique as used in a regular game of bowling - ie for a spare : score = 10 + pins dropped on next ball and for a strike : score = 10 + pins dropped on next 2 balls.
- Maintenance Call :** This is essentially a simulation of some repair work – ball not returned, pinsetter did not re-rack, etc.– that is to take place for a particular lane. The game play is halted for the time the lane is being repaired.

III. UML CLASS DIAGRAMS (Before Refactoring) :

Below are UML diagrams describing some of the major functionalities of the game.

There are several relationships exhibited by the different member classes that make up the components of the game. Some of these are :

- Association :** Shows a relationship between the two classes. One of the classes may have objects of another class being used within it. This is shown by a bold arrow line.
- Dependency :** In some cases it can also show a dependency between two or more classes. Any changes made to a class may cause changes in the class that is dependent on it.
- Composition:** This depicts the relationship between two classes where one class "is entirely made of another class" ie: One of the classes cannot exist if the parent/main class object doesn't exist. This is represented by an arrow with a darkened diamond at the end of the parent class.
- Aggregation :** This resembles the "part of" relationship between 2 classes. ie : One class is "a part" of another class.
- Generalization :** This is used one one class generalises the functionalities of all its subclasses. That is it is an umbrella class for other classes which inherit all properties from the parent as well have some other specifi properties unique to them.

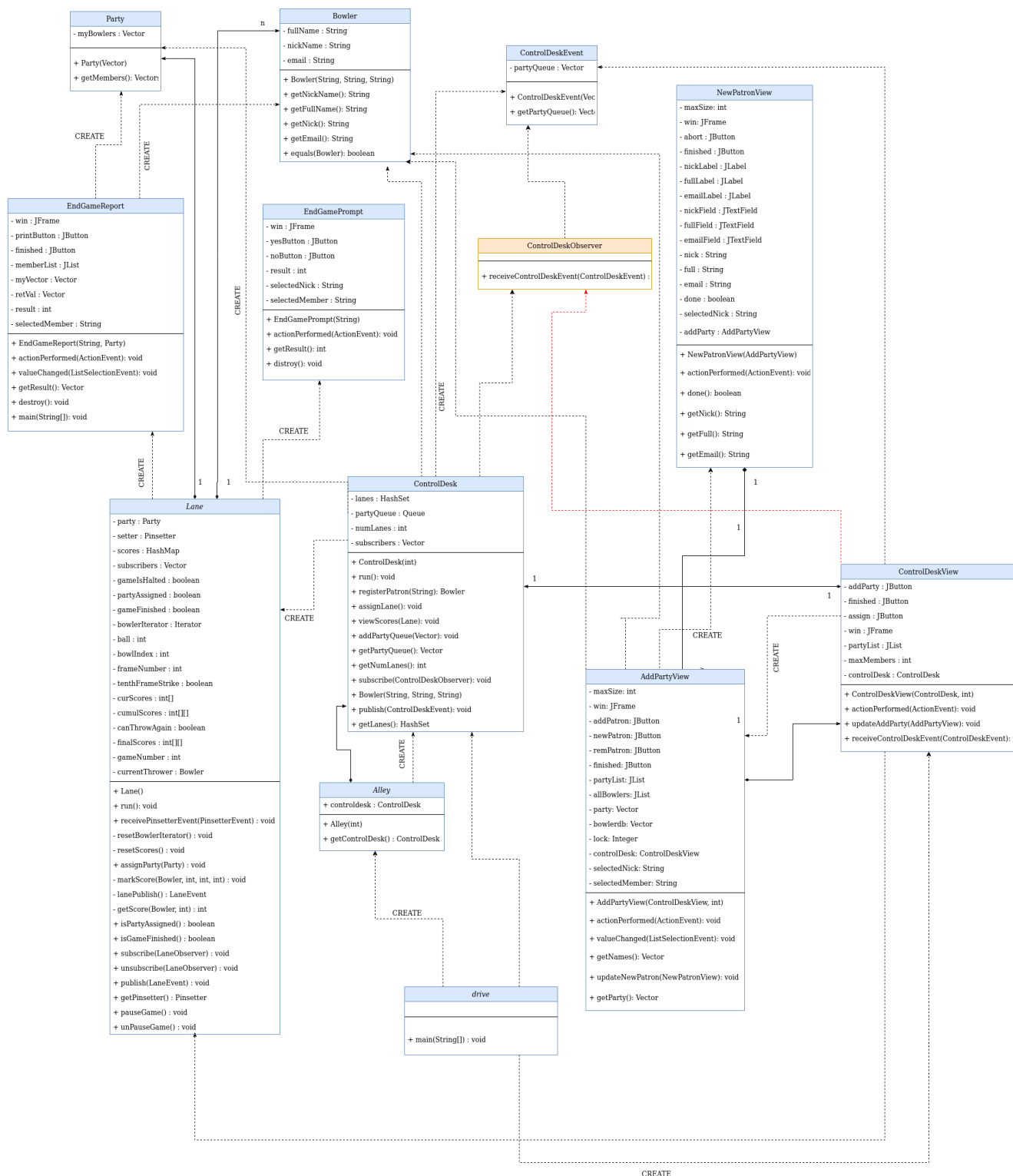
6. Specialisation : This is the exact opposite of generalisation. A specialised class is a subclass which inherits all properties from its parent class and also adds certain specific properties that are unique to it. All specific classes will be under a particular main/parent class.

A. Functionality : This UML diagram represents the functionality of creating a new party for a game and assigning a particular lane for the same. The bowlers can be chosen from an existing list or a NewPatron can be added

- The classes involved in this are :
 - ControlDesk
 - ControlDeskObserver (Interface)
 - ControlDeskEvent
 - ControlDeskView
 - Bowler
 - Party
 - NewPatronView
 - AddPartyView
 - EndGameReport
 - EndGamePrompt
 - Lane
 - Alley
 - Drive
- The cardinalities - *number of participating objects in any association between two classes* - are also mentioned on the relationship arrows of the classes.
- To further indicate the creation of classes, it has also been specified as to which class is create from which parent class. The **create** written above the arrows indicates the parent-child relationship.
- The **ControlDeskObserver** class is an interface class. Essentially serves as an interface between two or more classes that may not be able to interact otherwise. For eg: the **ControlDeskView** and **ControlDeskEvent** classes are interfaced to be able to share information and perform functions. This is also a demonstration of the *Adapter Design Pattern*
- The **drive** class is the driver module for the game to begin. It is linked to the **ControlDesk** class that carries out all the functions to control the game.
- All the methods and attributes associated with each class are shown in the UML class diagram as well. The private attributes and methods are represented by - and the public ones are represented by +.

The several arrows in the diagram represent different relationships between the multiple classes.

- **Associations and Dependencies :**
 - **AddPartyView** and **Bowler** class are also related via association. The **AddPartyView** class has a list of all the Bowlers and hence depends on the Bowler class in order to obtain the Names and Nicknames of the bowlers.
 - The **ControlDesk** and **ControlDeskView** classes are associated as the **ControlDeskView** class uses an object of the **ControlDesk** class.
 - **AddPartyView** and **NewPatronView** are also associated in a similar manner
 - **ControlDeskView** and **AddPartyView** also have an association relation
- **Compositions:**
 - The **NewPatronView** class composes the **AddPartyView** class since a major functionality supported by the **AddPartyView** class is the add a patron to the list of already existing bowlers.
 - The **AddPartyView** class composes the **ControlDeskView** since the main purpose or functionality of the **ControlDesk** is to enable the players to add/create a party with a set of members/patrons and play the game. This is done via the **AddPartyView** class's members and methods.
 - The **ControlDeskView** class inturn composes the control desk class since without the View provided by the GUI there would be no interface to support the **ControlDesk** class.
 - The **Bowler** object is a *composition* of the lane class indicating that if there exists an objec of the **Bowler** class then it must definitely belong to some **Lane** object and cannot exist alone. There exists a *1:n* cardinality indicating that for a particular **Lane** there can be multiple **Bowlers** associated to it and a **Bowler** can be associated to only 1 lane.
 - For **Alley** and **ControlDesk** classes, if there exists an object of the **ControlDesk** then it must have at least one object of **Alley** linked to it. It cannot exists standalone, hence the composition relationship.



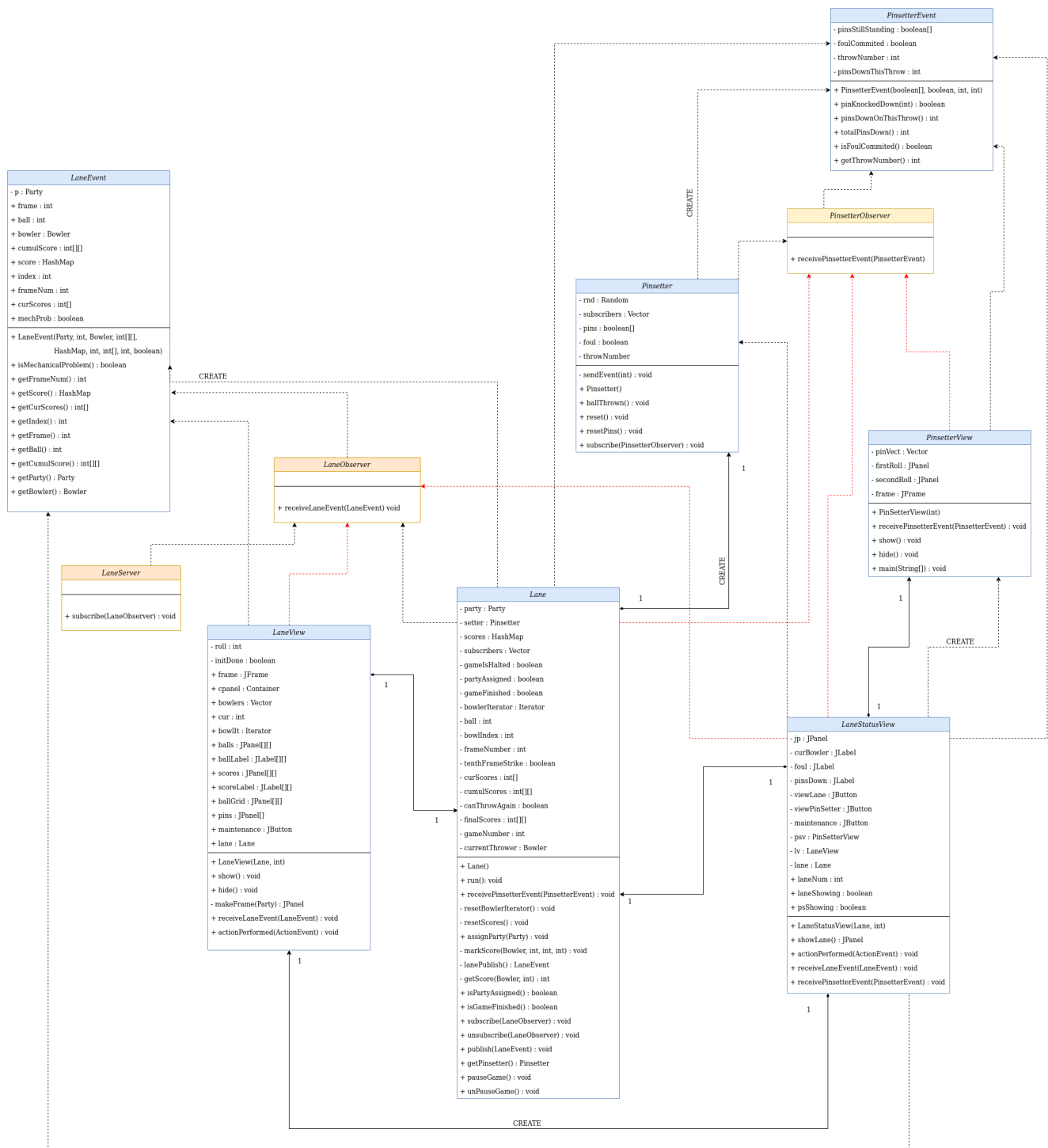
B. Fucntionality : Simulating a Ballthrow and observing corresponding changes in the Score and PinsetterView

- The classes involved in this are :
 - Pinsetter
 - PinsetterObserver (interface)
 - PinsetterEvent
 - PinSetterView
 - Lane
 - LaneEvent
 - LaneObserver
 - LaneServer
 - LaneView
 - LaneStatusView
 - LaneEventInterface

- The cardinalities - *number of participating objects in any association between two classes* - are mentioned on the relationship arrows of the classes.
- To further indicate the creation of classes, it has also been specified as to which class is created from which parent class. The **create** written above the arrows indicates the parent-child relationship.
- The **PinsetterObserver**, **LaneObserver**, **LaneEventInterface** classes are an interface class. Essentially serves as an interface between two or more classes that may not be able to interact otherwise. For eg: the **PinSetterView** and **Pinsetter** classes are interfaced to be able to share information and perform functions. Also, the **LaneStatusView** and **LaneView** are interfaced with **LaneObserver** to **LaneEvent**. This is also a demonstration of the *Adapter Design Pattern*
- All the methods and attributes associated with each class are shown in the UML class diagram as well. The private attributes and methods are represented by - and the public ones are represented by +.

The several arrows in the diagram represent different relationships between the multiple classes.

- **Associations and Dependencies :**
 - **Pinsetter** & **Lane** and **PinsetterView** & **LaneStatusView** classes are related via an association indicated by a solid arrow that shows that they have objects of these respective classes shared between them.
 - Dependencies are indicated by dotted arrows and show that a change in the class to which the arrow head points will and can cause a change to the dependent class.
- **Compositions:**
 - The **LaneView** and **Lane** classes exist only if the **LaneStatusView** class object exists. That is the **LaneStatusView** class composes the remaining classes. This is because, for every Lane there must be a UI view and a status view linked to it.
 - The **LaneStatusView** class also composes the **PinsetterView** class indicating that if there exists a **PinsetterView** object then it must be the view of the pins belonging to some **LaneStatusView** object.
 - Likewise the **Lane** class also composes the **Pinsetter** class.



C. Fucntionality : Scoring the game and maintainga a queue of the various parties

- The classes involved in this are :
 - Score
 - ScoreHistoryFile
 - PrintableText
 - ScoreReport
 - BowlerFile
 - Queue
 - ControlDesk
- The cardinalities - *number of participating objects in any association between two classes* - are mentioned on the relationship arrows of the classes.
- To further indicate the creation of classes, it has also been specified as to which class is create from which parent class. The **create** written above the arrows indicates the parent-child relationship.
- This UML class diagram unlike the previous two shows two disjoint sets of classes. However in order to successfully implement this functionality, all these classes are required.

- All the methods and attributes associated with each class are shown in the UML class diagram as well. The private attributes and methods are represented by - and the public ones are represented by +.

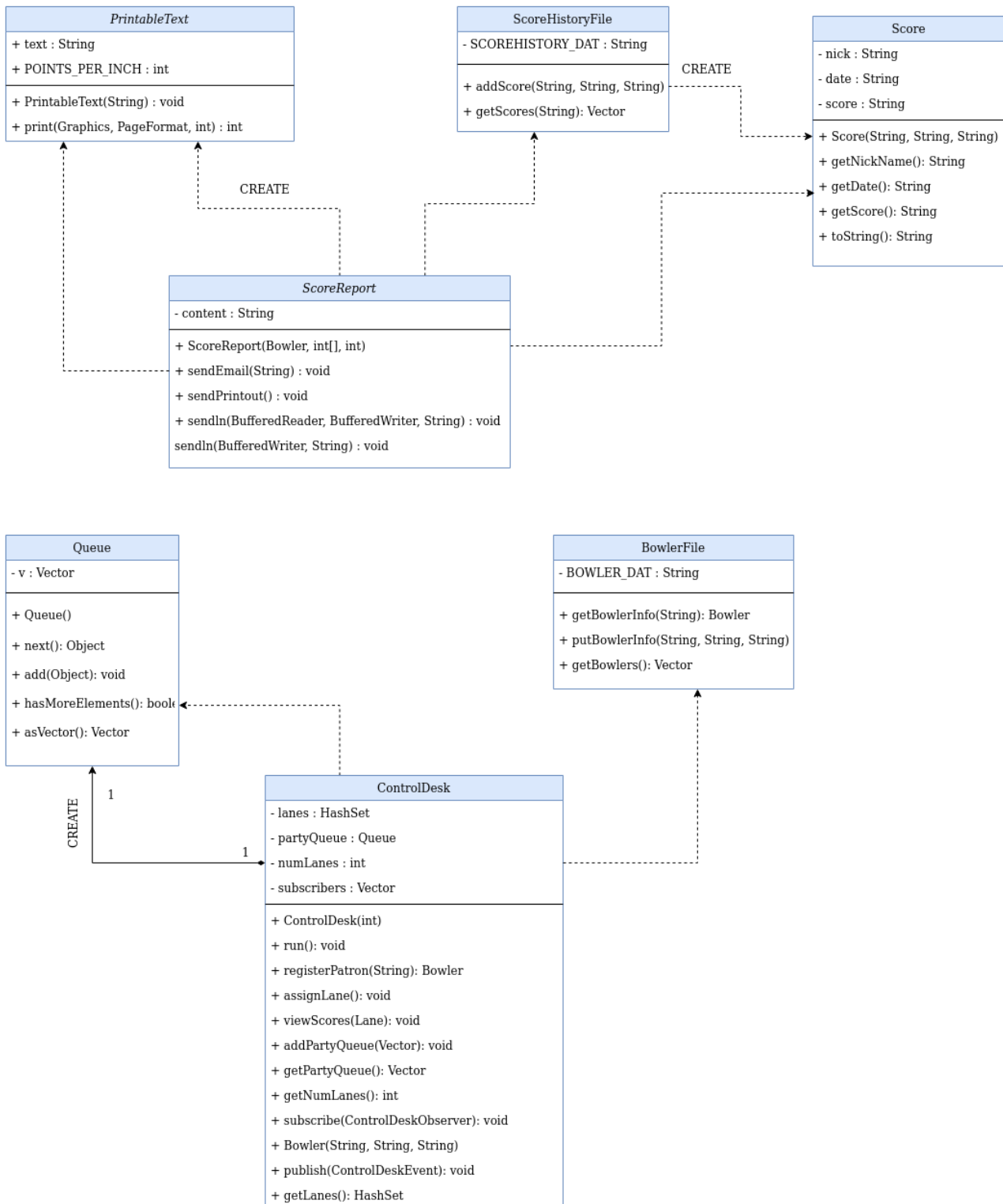
The several arrows in the diagram represent different relationships between the multiple classes.

- **Associations and Dependencies :**

- **Score & ScoreHistoryFile**, **Score & ScoreReport**, **ScoreReport & ScoreHistoryFile** are all related via a dependency indicated by a dotted arrow that shows that they have objects of these respective classes shared between them.

- **Compositions:**

- The **ControlDesk** class composes the **Queue** class indicating that if there exists a **Queue** object comprising the list of all the parties in the queue, then it must be managed by some **ControlDesk** object.



IV. UML CLASS DIAGRAMS (After refactoring)

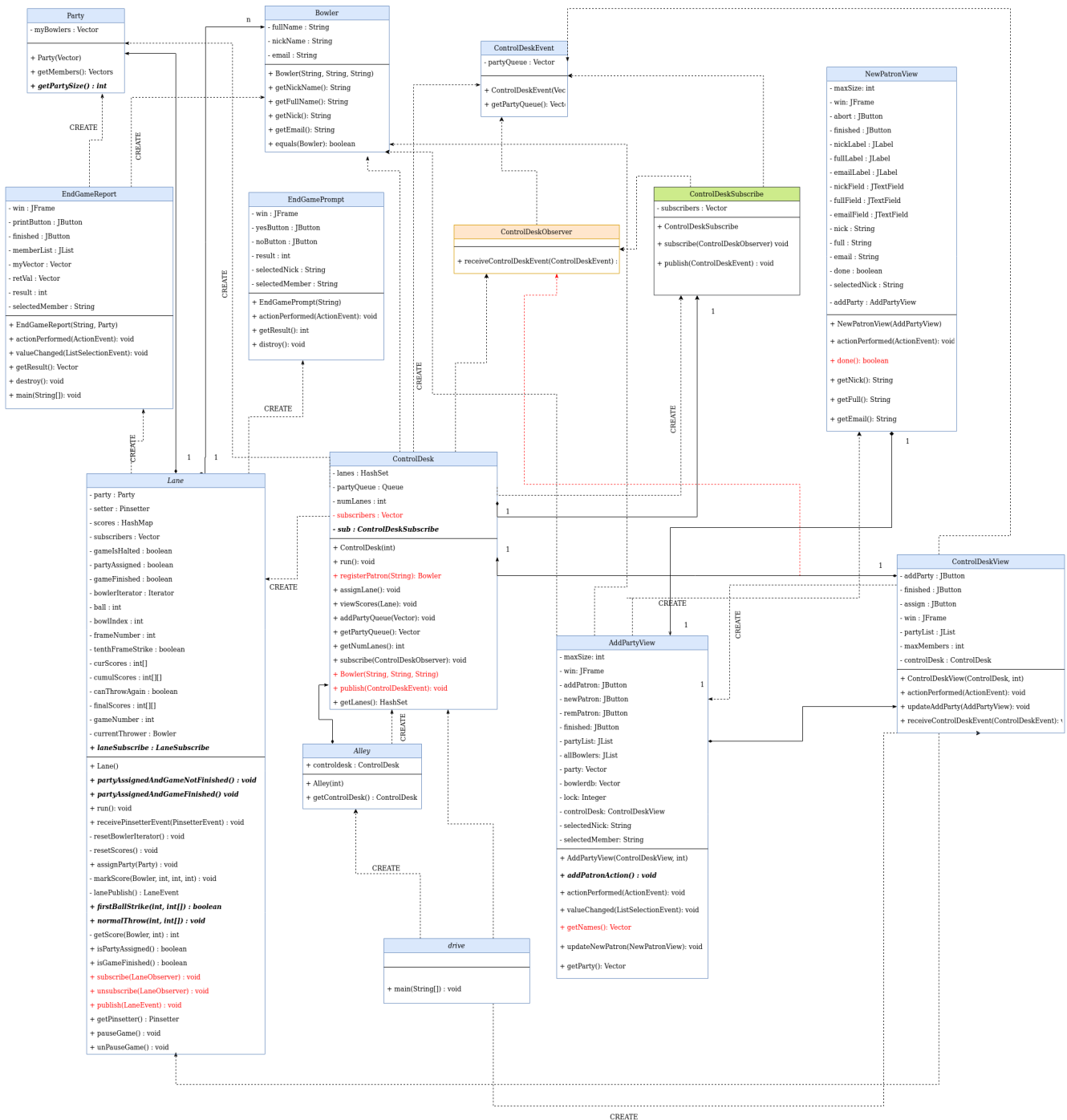
Below are UML diagrams describing the same major functionalities of the game mentioned in the previous section.

The diagram follows a **KEY** that represents the changes and modifications that were done while refactoring and thus resulted in the new UML Diagrams.

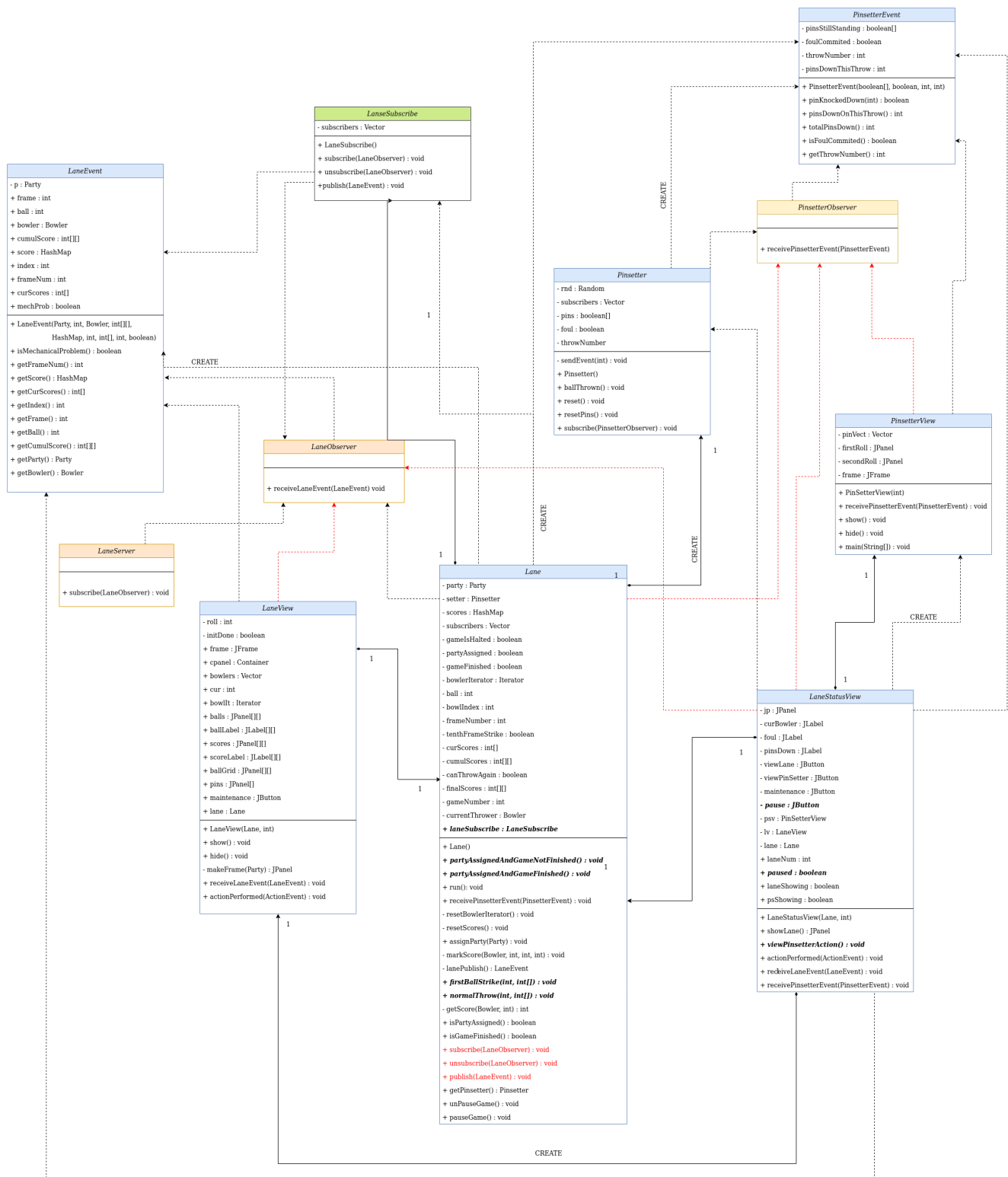
KEY:

- **New classes introduced:** These are represented by a **green** coloured class
- **New attributes/methods introduced:** These are represented by **bold and italics black text**
- **Methods or attributes eliminated:** These are represented by **red text**
- **New Associations or relations among classes:** These are represented by **blue arrows**

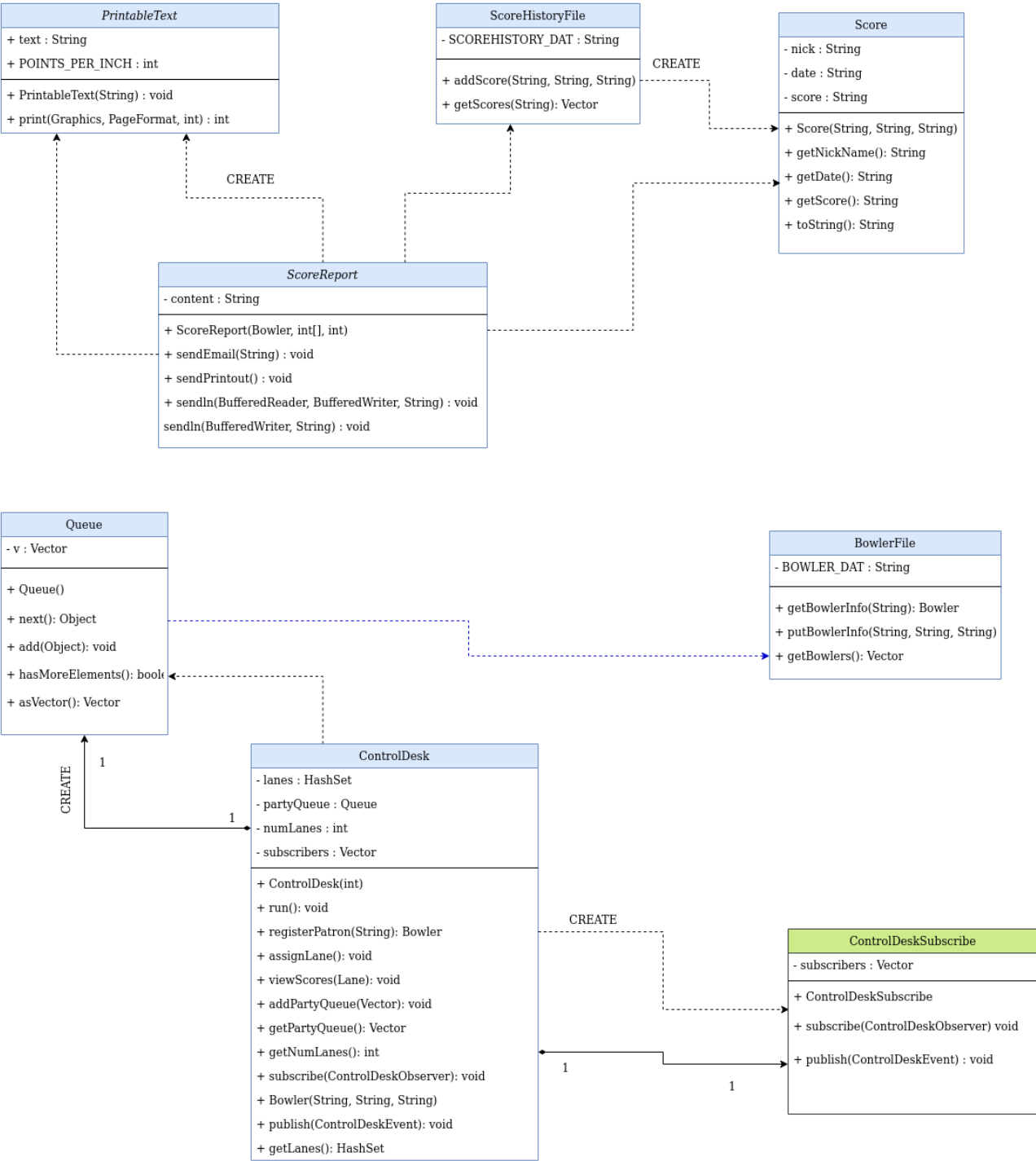
A. Functionality : This UML diagram represents the functionality of creating a new party for a game and assigning a particular lane for the same. The bowlers can be chosen from an existing list or a NewPatron can be added



B. Functionality : Simulating a Ballthrow and observing corresponding changes in the Score and PinsetterView



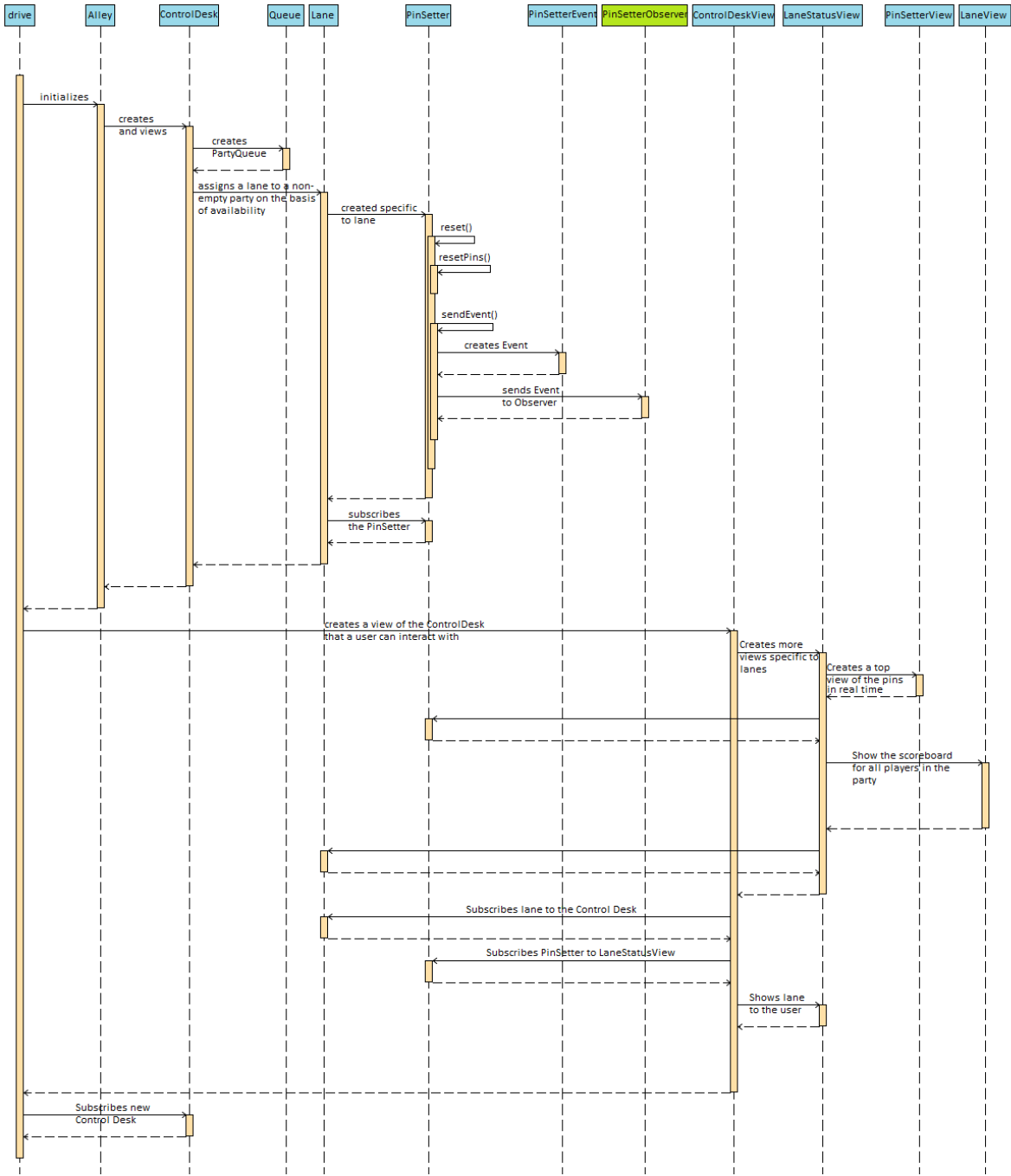
C. Fucntionality : Scoring the game and maintaing a queue of the various parties



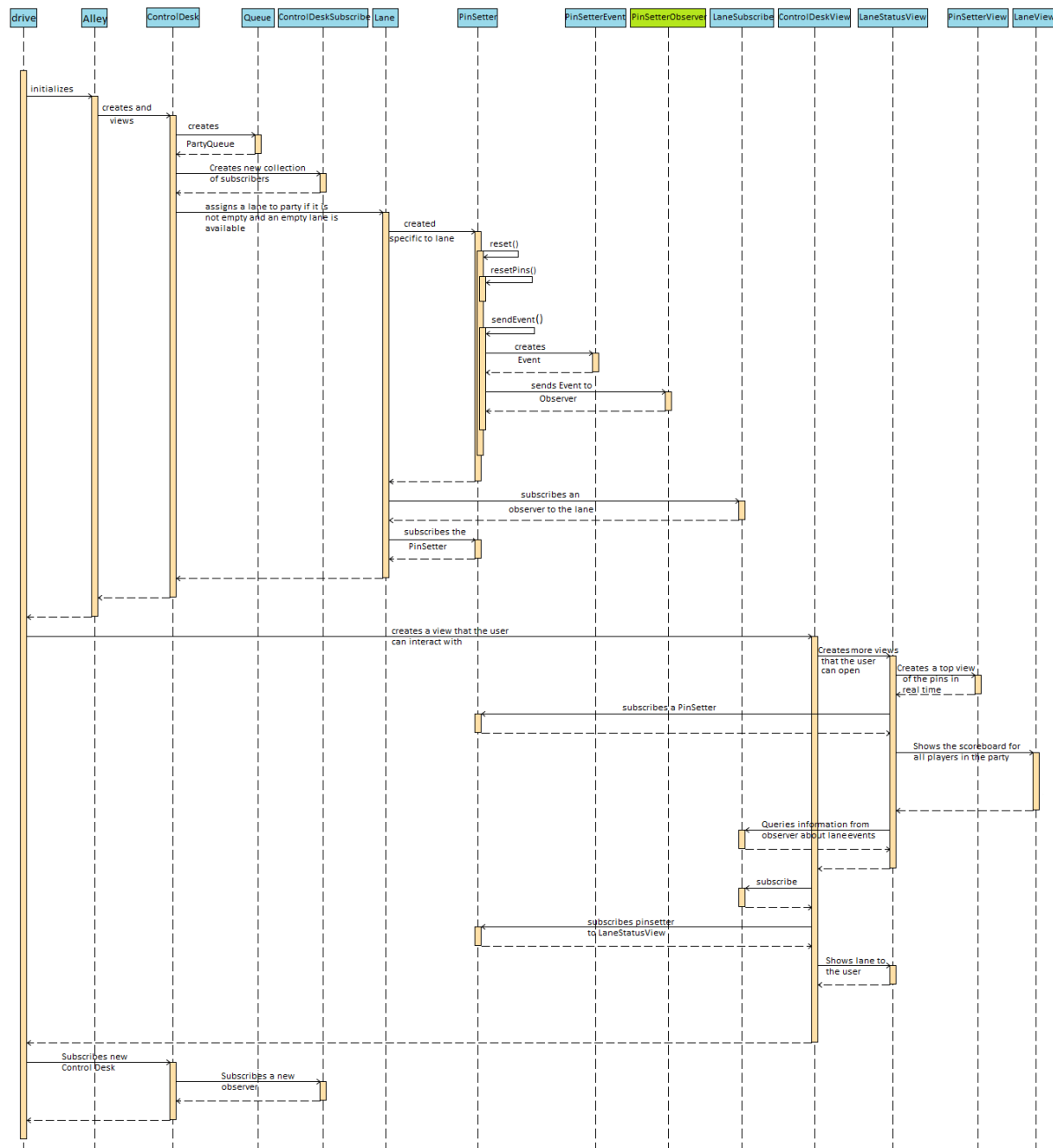
V. SEQUENCE DIAGRAMS :

Below are the sequence diagrams that depict the flow of the various functionalities incorporated into the game. The sequence diagrams have been drawn for both the *original* and the *refactored* code and hence have subtle differences in their major control flow.

BEFORE REFACTORING



AFTER REFACTORING



VI. SUMMARY OF RESPONSIBILITIES OF EACH MAJOR CLASS :

The Bowling Management System codebase has a collection of a total of 29 files. Each file has a collection of classes and functions that help simulate the entire game. Here is a list of all the files and their corresponding characteristics:

SNo.	File Name	Methods	Attribute	Major Functionalities	Interlinked Classes
1.	AddPartyView	<ul style="list-style-type: none">void actionPerformed()void valueChanged()Vector getParty()Vector getNames()void updateNewPatron()	<ul style="list-style-type: none">Vector partyVector bowlerdbControlDeskView controlDeskString selectedNickString selectedMember	<ul style="list-style-type: none">Adding a new patron to partyRemoving a patron from a partyCreating a new patronFinished party selectionReturning the latest state of the party	<ul style="list-style-type: none">NewPatronView

SNo.	File Name	Methods	Attribute	Major Functionalities	Interlinked Classes
2.	Alley	<ul style="list-style-type: none"> ControlDesk getControlDesk() 	<ul style="list-style-type: none"> ControlDesk controldesk 	<ul style="list-style-type: none"> Return Current state of ControlDesk 	<ul style="list-style-type: none"> ControlDesk
3.	Bowler	<ul style="list-style-type: none"> String getNickName() String getFullName () String getNick () String getEmail () 	<ul style="list-style-type: none"> String fullName String nickName String email 	<ul style="list-style-type: none"> Getter functions Validation of the bowler 	NIL
4.	BowlerFile	<ul style="list-style-type: none"> static Bowler getBowlerInfo(String nickName) static void putBowlerInfo(String nickName,String fullName,String email) static Vector getBowlers() 	<ul style="list-style-type: none"> static String BOWLER_DAT 	<ul style="list-style-type: none"> Adding a new bowler Getting details of one bowler Getting details of all bowlers 	NIL
5.	ControlDesk	<ul style="list-style-type: none"> void run() Bowler registerPatron(String nickName) void assignLane() void addPartyQueue(Vector partyNicks) Vector getPartyQueue() int getNumLanes() void publish(ControlDeskEvent event) HashSet getLanes() 	<ul style="list-style-type: none"> HashSet lanes Queue partyQueue int numLanes Vector subscribers 	<ul style="list-style-type: none"> Setter and Getter functions Broadcast an event to subscribing objects. Creating a new patron Finished party selection Returning party names to be displayed in the GUI representation of the wait queue. Main loop for ControlDesk's thread Registering a Patron Assigning a lane 	<ul style="list-style-type: none"> Lane
6.	ControlDeskEvent	<ul style="list-style-type: none"> Vector getPartyQueue() 	<ul style="list-style-type: none"> Vector partyQueue 	<ul style="list-style-type: none"> Returns a vector of the names of the parties in the waiting queue 	
7.	ControlDeskObserver	<ul style="list-style-type: none"> void receiveControlDeskEvent 	NIL	<ul style="list-style-type: none"> Interface for classes that observe control desk events. 	<ul style="list-style-type: none"> -----
8.	ControlDeskView	<ul style="list-style-type: none"> void actionPerformed(ActionEvent e) void updateAddParty(AddPartyView addPartyView) void receiveControlDeskEvent(ControlDeskEvent ce) 	<ul style="list-style-type: none"> int maxMembers ControlDesk controlDesk 	<ul style="list-style-type: none"> Display the GUI for the control desk Handler for actionEvents Receive a new party from andPartyView Receive a broadcast from a ControlDesk 	<ul style="list-style-type: none"> ControlDesk AddPartyView

SNo.	File Name	Methods	Attribute	Major Functionalities	Interlinked Classes
9.	drive	<ul style="list-style-type: none"> static void main() 	<ul style="list-style-type: none"> int numLanes int maxPatronsPerParty 	<ul style="list-style-type: none"> Driver class for the entire game Creates and alley with numLanes number of lanes Activates the control desk object Render the GUI for the control desk via ControlDeskView 	<ul style="list-style-type: none"> ControlDesk Alley ControlDeskView
10.	EndGamePrompt	<ul style="list-style-type: none"> EndGamePrompt(String partyName) void actionPerformed(ActionEvent e) int getResult() void distroy() 	<ul style="list-style-type: none"> int result String selectedNick String selectedMember 	<ul style="list-style-type: none"> Displaying the end prompt Destroying the currently active game object. 	<ul style="list-style-type: none"> -----
11.	EndGameReport	<ul style="list-style-type: none"> EndGameReport(String partyName, Party party) void actionPerformed(ActionEvent e) Vecotr getResult() void distroy() static void main(String args[]) void valueChanged(ListSelectionEvent e) 	<ul style="list-style-type: none"> int result String selectedMember 	<ul style="list-style-type: none"> Displaying the end game repor Destroying the currently active game object. 	<ul style="list-style-type: none"> -----
12.	Lane	<ul style="list-style-type: none"> void run() void receivePinsetterEvent(PinsetterEvent pe) void receivePinsetterEvent(PinsetterEvent pe) void resetScores() void assignParty(Party theParty) void markScore(Bowler Cur, int frame, int ball, int score) LaneEvent lanePublish() void publish(LaneEvent event) Setter and Getter functions 	<ul style="list-style-type: none"> Party party Pinsetter setter HashMap scores Vector subscribers boolean gamelsHalted boolean partyAssigned private boolean gameFinished; Iterator bowlerIterator int ball int bowlIndex int frameNumber boolean tenthFreameStrike intp[curScores int[][] cumulScores boolean canThrowAgain int [][] finalScroes int gameNumber Bowler currentThrowe 	<ul style="list-style-type: none"> Simulates the bowling alley lanes in the game Ensures cylic rounds of each bowlers turn assigns a party to the lane Keeps track and calculates bowlers score 	<ul style="list-style-type: none"> Bowler Party Pinsetter
13.	LaneEvent	<ul style="list-style-type: none"> Setters and getter funcitons only 	<ul style="list-style-type: none"> Party p int frame int ball Bowler bowler boolean mechProb 	<ul style="list-style-type: none"> Setter and getter functions for all lane functionalities 	<ul style="list-style-type: none"> Party Bowler
14.	LaneEventInterface	<ul style="list-style-type: none"> An interface class 	<ul style="list-style-type: none"> ----- 	<ul style="list-style-type: none"> Interfaces the multiple classes 	<ul style="list-style-type: none"> Party Bowler

SNo.	File Name	Methods	Attribute	Major Functionalities	Interlinked Classes
15.	LaneObserver	<ul style="list-style-type: none"> An interface class 	<ul style="list-style-type: none"> ----- 	<ul style="list-style-type: none"> Interfaces the multiple classes 	<ul style="list-style-type: none"> -----
16.	LaneServer	<ul style="list-style-type: none"> An interface class 	<ul style="list-style-type: none"> ----- 	<ul style="list-style-type: none"> Interfaces the multiple classes 	<ul style="list-style-type: none"> -----
17.	LaneStatusView	<ul style="list-style-type: none"> LaneStatusView(Lane lane, int laneNum) JPanel showLane() void receiveLaneEvent(LaneEvent le) void receivePinsetterEvent(PinsetterEvent pe) 	<ul style="list-style-type: none"> PinSetterView psv LaneView lv Lane lane int laneNum boolean laneShowing boolean psShowing 	<ul style="list-style-type: none"> Rendering the GUI for the status of the lanes 	<ul style="list-style-type: none"> PinSetterView LaneView Lane
18.	LaneView	<ul style="list-style-type: none"> void show() void high() JFrame makeFrame void receiveLaneEvent(LaneEvent le) 	<ul style="list-style-type: none"> int cur int roll boolean initDone Iterator bowlIt Lane lane 	<ul style="list-style-type: none"> Render the view GUI for the alley lanes 	<ul style="list-style-type: none"> Lane
19.	NewPatronView	<ul style="list-style-type: none"> void actionPerformed() void valueChanged() Vector getParty() Vector getNames() void updateNewPatron() 	<ul style="list-style-type: none"> int maxSize boolean done Stringf selectedNick AddPartyView addParty String selectedMember 	<ul style="list-style-type: none"> Setter and Getter functions 	<ul style="list-style-type: none"> AddPartyView
20.	Party	<ul style="list-style-type: none"> Vector getMembers() 	<ul style="list-style-type: none"> Vecotr myBowlers 	<ul style="list-style-type: none"> Accessor for members belonging to a party 	<ul style="list-style-type: none"> -----
21.	Pinsetter	<ul style="list-style-type: none"> void ballThrown() void reset() void resetPins() void subscribe(PinsetterObserver subscriber) 	<ul style="list-style-type: none"> Vector subscribers Random rnd boolean[] pins boolean foul int throwNumber 	<ul style="list-style-type: none"> Updates the state of the pins across all subscribers Simulates a ball being thrown and probabilistically creates a result for the ballThrown() function- either as a foul or some number of pins 	<ul style="list-style-type: none"> PinsetterObser
22.	PinsetterEvent	<ul style="list-style-type: none"> boolean pinsKnockedDown() int pinsDownOnThisThrow() int totalPinsDown() boolean isFoulCommitted() int gerThrowNumber 	<ul style="list-style-type: none"> boolean[] pinsStillStanding boolean foulCommitted int throwNumber int pinsDownThisThrow 	<ul style="list-style-type: none"> Includes functionalities that mimic the dropping of pins and probabilistaically (or randomly) determines this 	<ul style="list-style-type: none"> -----
23.	PinsetterObserver	<ul style="list-style-type: none"> An interface class 	<ul style="list-style-type: none"> ----- 	<ul style="list-style-type: none"> Interfaces the multiple classes 	<ul style="list-style-type: none"> -----

SNo.	File Name	Methods	Attribute	Major Functionalities	Interlinked Classes
24.	PinSetterView	<ul style="list-style-type: none"> void receivePinsetterEvent() 	<ul style="list-style-type: none"> Vector pinVect 	<ul style="list-style-type: none"> Constructs a Pin Setter GUI displaying which roll it is Receives the current state of the PinSetter and the method changes how the GUI looks accordingly 	<ul style="list-style-type: none"> -----
25.	PrintableText	<ul style="list-style-type: none"> int print(Graphics g, PageFormat pageFormat, int pageIndex) 	<ul style="list-style-type: none"> String text int POINTS_PER_INCH 	<ul style="list-style-type: none"> Displays the graphical text on the UI including colour 	<ul style="list-style-type: none"> -----
26.	Queue	<ul style="list-style-type: none"> void add(Object o) boolean hasMoreElements() Vector asVector() Object next() 	<ul style="list-style-type: none"> Vector v 	<ul style="list-style-type: none"> Creates a new Queue 	<ul style="list-style-type: none"> -----
27.	Score	<ul style="list-style-type: none"> constructor, getter and setter functions 	<ul style="list-style-type: none"> String nick String date String score 	<ul style="list-style-type: none"> Sets the scores for the players in the game 	<ul style="list-style-type: none"> -----
28.	ScoreHistoryFile	<ul style="list-style-type: none"> Vector getScores(string nick) void addScore() 	<ul style="list-style-type: none"> String SCOREHISTORY_DAT 	<ul style="list-style-type: none"> Writes the scores of the plays into a .DAT file after a game finishes. Makes use of I/O options, reading/writing to a buffer etc 	<ul style="list-style-type: none"> -----
29.	ScoreReport	<ul style="list-style-type: none"> void sendEmail() void sendPrintout() void sendIn() 	<ul style="list-style-type: none"> String content 	<ul style="list-style-type: none"> Generates the ScoreReport and sends it via email/printout to the user. 	<ul style="list-style-type: none"> Bowler

VII. ANALYZING THE ORIGINAL DESIGN :

As mentioned in the design document provided to us with the codebase, most features and functionalities are in line with the document. The only feature that has not been implemented is the system to *Email the score report to the player*.

Below is an analysis of the design of the original code :

- **Design analysis *Within* Classes**
 - **Comments :** The given code is well commented and enables a new coder to easily understand the codebase.
 - **Long Method:** Some files such as `Lane.java` have functions that are too long and hence have high cyclomatic complexity. This is a drawback in the design.
 - **Long Parameter List:** The `LaneEvent.java` file has a long parameter list, this is undesirable since it makes the code more complex.
 - **Duplicate Code:** A few functions are redundant, they perform the same tasks and hence one of the redundant methods can be removed.
 - **Inconsistent/Uncommunicative names:** The code makes use of intuitive function names and instance declarations which makes it easier for a person reading the codebase for the first time to navigate through the different files.
 - **Conditional Complexity:** Several files such as `Lane.java`, `LaneEvent.java`, `LaneStatusView.java` etc have high conditional complexity. These can be improved upon.

- **Speculative generalization and Dead Code:** There exist some chunks of dead code that have been left in the codebase with the intention to complete them later. These add to the LOC, and make the code less readable. Hence it is best to remove them.

- **Design Analysis *Between* Classes**

- **Indecent Exposure:** Classes like the `ControlDesk` are subject to this codesmell wherein they have functions that expose themselves to the class where they aren't even required.
- **Feature Envy:** Certain methods from `ControlDesk` can be moved into other classes wherein they are more extensively used. For Eg: the Queue related functions can be moved to the `Queue.java` file.
- **Primitive Obsession:** Objects and datatypes such as the `JButton` are written and rewritten again and again. This causes it to become extremely complex. This can be fixed by writing a class for the same.

VIII. CODE SMELLS

The Bowling Management System codebase is comprised of a total of 29 files. *Here is a list of those files and the code smells found in them:*

SNo.	File/Class Name	Code Smells and their instances
1.	AddPartyView	<ol style="list-style-type: none"> 1. Conditional Complexity <ul style="list-style-type: none"> ◦ The <code>void actionPerformed(ActionEvent e)</code> function has high conditional complexity due to the high number of action events possible within the class. 2. Feature Envy: One of the functions makes extensive use of the <code>NewPatron</code> class and hence it should be moved there: <ul style="list-style-type: none"> ◦ <code>updateNewPatron</code> 3. Primitive Obsession & Combinatorial Explosion: The <code>JButton</code> objects are initialized multiple times in this class for various uses. This causes the same code to be written multiple times. Instead a class should be created for this like how it's done for queues using the <code>Queue</code> class: <ul style="list-style-type: none"> ◦ <code>addPatron</code> ◦ <code>remPatron</code> ◦ <code>newPatron</code> ◦ <code>finished</code>
2.	ControlDesk	<ol style="list-style-type: none"> 1. Indecent Exposure <ul style="list-style-type: none"> ◦ The <code>subscribe</code> function should be moved to a different class because all the components this class expose themselves to this method despite it not needing them. This is also responsible for increased lack of cohesion in this class. 2. Feature Envy: Two of the functions use the <code>Queue</code> class a lot so they should be moved there. <ul style="list-style-type: none"> ◦ <code>addpartyQueue</code> ◦ <code>getpartyQueue</code>
3.	ControlDeskView	<ol style="list-style-type: none"> 1. Large Class <ul style="list-style-type: none"> ◦ This class is very long (taking about 180 lines out of the total 1500 lines of code amidst a total 29 classes). 2. Primitive Obsession & Combinatorial Explosion: The <code>JButton</code> objects are initialized multiple times in this class for various uses. This causes the same code to be written multiple times. Instead a class should be created for this: <ul style="list-style-type: none"> ◦ <code>addParty</code> ◦ <code>assign</code> ◦ <code>finished</code>
4.	Lane	<ol style="list-style-type: none"> 1. Large Class- The <code>Lane</code> class is massive (at about 600 lines) and breaking it down would make it easier to read, understand and troubleshoot. 2. Long Method- The following functions are very big and should be broken down: <ul style="list-style-type: none"> ◦ <code>run()</code> function ◦ <code>receivePinsetterEvent(PinsetterEvent pe)</code> function ◦ <code>getScore()</code> function 3. Conditional Complexity- The following methods have high complexity due to a high number of chained if-else conditions: <ul style="list-style-type: none"> ◦ <code>getScore()</code> function ◦ <code>receivePinsetterEvent()</code> function 4. Dead Code <ul style="list-style-type: none"> ◦ The <code>Strikeballs</code> variable is never used in the <code>getScore(Bowler Cur, int frame)</code> function and should be removed from there. 5. Indecent Exposure- The <code>laneSubscribe</code> function should be removed from the class because it can be handled in a different class which would result in other data and methods in the class to not be exposed while also reducing the number of methods.

SNo.	File/Class Name	Code Smells and their instances
5.	LaneStatusView	<ol style="list-style-type: none"> 1. Conditional Complexity <ul style="list-style-type: none"> ◦ The <code>void actionPerformed(ActionEvent e)</code> function has high conditional complexity due to the high number of action events possible within the class. 2. Primitive Obsession & Combinatorial Explosion: The <code>JButton</code> objects are initialized multiple times in this class for various uses. This causes the same code to be written multiple times. Instead a class should be created for this: <ul style="list-style-type: none"> ◦ <code>viewLane</code> ◦ <code>viewPinSetter</code> ◦ <code>maintenance</code>
6.	LaneView	<ol style="list-style-type: none"> 1. Long Method & Conditional Complexity-The <code>receiveLaneEvent(LaneEvent le)</code> function is very long and has high conditional complexity. It should be broken down if possible for easier understanding and debugging.
7.	NewPatronView	<ol style="list-style-type: none"> 1. Primitive Obsession & Combinatorial Explosion: The <code>JPanel</code> objects are initialized multiple times in this class. This causes the same code to be written multiple times. Instead a class should be created for this: <ul style="list-style-type: none"> ◦ <code>nickPanel</code> ◦ <code>fullPanel</code> ◦ <code>emailPanel</code>
8.	Score	<ol style="list-style-type: none"> 1. Comments-Irrelevant information is provided in the code, not pertaining to the explanation or functioning of the class.

IX. ANALYZING THE REFACTORED DESIGN

Our refactored design reflects a balance among competing criteria. Whilst refactoring our code, we understood that writing good quality code is to some level a bargain between these several parameters. It is essential that some metrics take a lower value however that may reflect in some other metric shooting up. Hence striking the right balance is essential.

There were several paramters that we had taken into consideration while we refactored the code that was provided to us. Here is an indepth analysis of how we chose between the metrics to be finetuned and the steps taken to achieve the same.

- **Low Coupling:**

We started off with a total of 29 classes (quite a huge number in itself). As we went on to increase the functionalities and extend the project, we observed that there were more classes required. Since this was a dire requirement and couldn't be overlooked we had to ensure that the coupling between these class is low.

Low coupling is often a sign of a well-structured computer system. Hence the refactored code includes as few dependencies as possible. We trie to keep the newer classes as standalone classes (such that they arent dependent on other classes). Furthermore, the existing dependencies between classes was also lowered. A major factor that is affected when coupling becomes high is the idea of *Data Abstraction*. If classes freely access the members of other classes, then the whole point of data-hiding is lost. Hence we took care of this by shifting those functions to the classes in which they are most suitable from where they were initially placed (Eg : `updateNewPatron` was shifted into the `NewPatronView` class).

Overall we were successfully able to ensure that on an average amongst all our classes, there exists a low value for the **Coupling metric**

- **High Cohesion :**

Cohesion is used to indicate the degree to which a class has a single, well-focused purpose. Here we focused on how individually classes are designed. Highly cohesive classes are much easier to maintain and less frequently changed. Such classes are more usable than others as they are designed with a well-focused purpose.

Firstly we observed that there tends to be a higher *Lack of Cohesion* for classes with greater number of methods. This is natural since then the functionality of the class isn't contained in a concise manner and the code readability decreases. However here, there was another metric that was to be taken care of. In order to decrease the *Cyclomatic Complexity* we split the rather huge functions into smaller simpler chunks (downside : increased number of methods). Hence we carefully noted the values of the metric and decided which one to choose over the other.

In cases where the cyclomatic complexity was too high (most of the times), we resorted to increasing the methods. In cases where the value for cohesion was dropping below a point, we decided to get rid of some redundant functions that weren't suited in the respective classes. Detailed examples for this can be found in the previous section about CODE SMELLS.

- **Separation of concerns:**

The idea behind this metric is that a software system must be decomposed into parts that overlap in functionality as little as possible. The main reason for creating numerous classes is that each class can encapsulte a particular fuctionality and can be responsible for all the tasks related to that. Following OOPS principles we paid special ephasis to modularize our code using encapsulation and inheritance.

We used the existing classes and created new ones that inherited methods and attributes from the *Parent/God classes*. Furthermore, we also introduced modularity in functions of the various classes by splitting them into smaller modules and invoking them from the main function. This also helped improve our cyclomatic complexity for many functions. Eg: `actionsPerformed` function in the `AddPartyView` class.

- **Information hiding**

Data Abstraction or Information hiding is one of the key techniques in OOPs. Most of the attributes in the existing classes were already private members of the class. Continuing this, we ensured that we incorporate attributes with the private visibility in most cases. We used public variables only when required. Lastly, all the functions we declared were public to ensure that they can be accessed by the objects of the class.

- **The Law of Demeter**

Demeter's Law is an extension of the principle of low coupling and high cohesion. We ensured that this law is obeyed by making fewer classes interact with each other and allowing classes to interact with objects of only immediate neighbour classes (based on Demeter's rule)

- **Extensibility**

Since we ensured low coupling, we made sure that it was easier to introduce new modules, for example a new implementation for an existing interface. The new features that we were asked to implement were easily done without needing to change much code from existing files. As a consequence of high cohesion, it was easier to implement new modules without being concerned with aspects that are not directly related to their functionality. For eg: the module to search the db was easily implemented.

- **Reusability**

On inspecting the code we noticed that some features are similar to the new features that are expected to be implemented. For Eg: the *pause/resume* feature is much like the maintenance call feature that causes the Lane to freeze until a button is clicked. Reusing this code and the methods associated with it, we recreated the new functionality.

X. METRIC ANALYSIS

The following questions have been answered in this section :

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

1. McCabe Cyclomatic Complexity

1.1 Measurements tell about the system

- Indicate the complexity of a program.
- Quantitative measure of the number of linearly independent paths through a program's source code.
- Our maximum cyclomatic complexity was set to 10.
- Functions that violated our constrain are (along with their respective cyclomatic complexity):
 1. Lane.java(Lane)
 1. getScore - 38
 2. run - 19
 3. receivePinsetterEvent - 12
 2. LaneView.java(LaneView)
 1. receiveLaneEvent - 19
 3. LaneStatusView.java(LaneStatusView)
 1. actionPerformed - 11
 4. AddPartyView.java(AddPartyView)
 1. actionPerformed - 11
- Inference: the highest cyclomatic complexity is of `getScore()` method in `Lane.java` and has a big scope of reduced complexity.

1.2 Guide our refactoring

- Split up method into simpler components.
- Complex loops can be made into separate functions.
- Complex conditional branches can be made into separate functions.
- Use smaller methods.

1.3 Refactoring affect metrics

Metrics - Lane.java - McCabe Cyclomatic Complexity (avg/max per method)						
Metric	Total	Mean	Std. Dev	Maximum	Resource causing Maximum	Method
▼ McCabe Cyclomatic Complexity (avg/max per method)	4.778	4.973	16	/Bowling-Management-System/code/Lane.java	normalThrow	
▼ Lane	4.778	4.973	16	/Bowling-Management-System/code/Lane.java	normalThrow	
normalThrow	16					
receivePinsetterEvent	12					
firstBallStrike	12					
getScore	12					
partyAssignedAndGameNotFinished	8					
run	7					
partyAssignedAndGameFinished	6					
resetScores	3					
Lane	1					
resetBowlerIterator	1					
assignParty	1					
markScore	1					
lanePublish	1					
isPartyAssigned	1					
isGameFinished	1					
getPinsetter	1					
pauseGame	1					
unPauseGame	1					
► Number of Parameters (avg/max per method)	0.667	1.106	4	/Bowling-Management-System/code/Lane.java	markScore	
► Nested Block Depth (avg/max per method)	2.333	1.599	5	/Bowling-Management-System/code/Lane.java	partyAssignedAndGameNotFinished	
► Depth of Inheritance Tree (avg/max per method)	2	0	2	/Bowling-Management-System/code/Lane.java		
► Weighted methods per Class (avg/max per class)	86	86	0	86	/Bowling-Management-System/code/Lane.java	

Metrics - LaneView.java - McCabe Cyclomatic Complexity (avg/max per method)						
Metric	Total	Mean	Std. Dev	Maximum	Resource causing Maximum	Method
▼ McCabe Cyclomatic Complexity (avg/max per method)	5.167	6.543	19	/Bowling-Management-System/code/LaneView.java	receiveLaneEvent	
▼ LaneView	5.167	6.543	19	/Bowling-Management-System/code/LaneView.java	receiveLaneEvent	
receiveLaneEvent	19					
makeFrame	7					
actionPerformed	2					
► LaneView	1					
show	1					
hide	1					
► Number of Parameters (avg/max per method)	0.833	0.687	2	/Bowling-Management-System/code/LaneView.java	LaneView	
► Nested Block Depth (avg/max per method)	2.333	1.374	5	/Bowling-Management-System/code/LaneView.java	receiveLaneEvent	
► Depth of Inheritance Tree (avg/max per method)	1	0	1	/Bowling-Management-System/code/LaneView.java		
► Weighted methods per Class (avg/max per class)	31	31	0	31	/Bowling-Management-System/code/LaneView.java	
► Number of Children (avg/max per type)	0	0	0	0	/Bowling-Management-System/code/LaneView.java	
► Number of Overridden Methods (avg/max per method)	0	0	0	0	/Bowling-Management-System/code/LaneView.java	
► Lack of Cohesion of Methods (avg/max per method)	0.88	0	0.88	/Bowling-Management-System/code/LaneView.java		
► Number of Attributes (avg/max per type)	15	15	0	15	/Bowling-Management-System/code/LaneView.java	
► Number of Static Attributes (avg/max per type)	0	0	0	0	/Bowling-Management-System/code/LaneView.java	
► Number of Methods (avg/max per type)	6	6	0	6	/Bowling-Management-System/code/LaneView.java	
► Number of Static Methods (avg/max per type)	0	0	0	0	/Bowling-Management-System/code/LaneView.java	
► Specialization Index (avg/max per type)	0	0	0	0	/Bowling-Management-System/code/LaneView.java	
Number of Classes	1					
Number of Interfaces	0					
Total Lines of Code	166					
► Method Lines of Code (avg/max per method)	133	22.167	25.99	62	/Bowling-Management-System/code/LaneView.java	receiveLaneEvent

uml-class-1

Metric	Total	Mean	Std. Dev	Maximum	Resource causing Maximum	Method
▼ McCabe Cyclomatic Complexity (avg/max per method)		3.143	2.231	8	/Bowling-Management-System/code/AddPartyView	actionPerformed
▼ AddPartyView		3.143	2.231	8	/Bowling-Management-System/code/AddPartyView	actionPerformed
actionPerformed	8					
addPatronAction	4					
valueChanged	3					
updateNewPatron	3					
AddPartyView	2					
getNames	1					
getParty	1					
▶ Number of Parameters (avg/max per method)		0.714	0.7	2	/Bowling-Management-System/code/AddPartyView	AddPartyView
▶ Nested Block Depth (avg/max per method)		2.143	0.833	3	/Bowling-Management-System/code/AddPartyView	addPatronAction
▶ Depth of Inheritance Tree (avg/max per class)		1	0	1	/Bowling-Management-System/code/AddPartyView	
▶ Weighted methods per Class (avg/max per class)	22	22	0	22	/Bowling-Management-System/code/AddPartyView	
▶ Number of Children (avg/max per type)	0	0	0	0	/Bowling-Management-System/code/AddPartyView	
▶ Number of Overridden Methods (avg/max per class)	0	0	0	0	/Bowling-Management-System/code/AddPartyView	
▶ Lack of Cohesion of Methods (avg/max per class)		0.75	0	0.75	/Bowling-Management-System/code/AddPartyView	
▶ Number of Attributes (avg/max per type)	14	14	0	14	/Bowling-Management-System/code/AddPartyView	
▶ Number of Static Attributes (avg/max per type)	0	0	0	0	/Bowling-Management-System/code/AddPartyView	
▶ Number of Methods (avg/max per type)	7	7	0	7	/Bowling-Management-System/code/AddPartyView	
▶ Number of Static Methods (avg/max per type)	0	0	0	0	/Bowling-Management-System/code/AddPartyView	
▶ Specialization Index (avg/max per type)		0	0	0	/Bowling-Management-System/code/AddPartyView	
Number of Classes	1					
Number of Interfaces	0					
Total Lines of Code	157					

2. Number of parameters

2.1 Measurements tell about the system

- Our maximum number of parameters was set to 5.
- It should be kept low for simpler understanding of code.
- Functions that violated our constrain are (along with their respective number of parameters):
 1. LaneEvent.java(LaneEvent)
 1. LaneEvent - 9
- Inference: There was just one method that had higher than 5 parameters.

2.2 Guide our refactoring

- We can pass an object instead of high number of parameters.
- We can split the method if the number of parameters can be partitioned well with the resulting methods.

3. Nested block depth

3.1 Measurements tell about the system

- Our maximum nested block depth was set to 5.
- It should be kept low for simpler understanding of code.
- Functions that violated our constrain are (along with their respective nested block depth):
 1. Lane.java(Lane)
 1. run - 7
 2. getScore - 7
- Inference: Lane class has methods which violate our constrain and needs improvement.

3.2 Guide our refactoring

- Split up method into simpler components.
- Put deeper blocks into meaningful functions.

3.3 Refactoring affect metrics

Metrics - Lane.java - Nested Block Depth (avg/max per method)						
Metric	Total	Mean	Std. Dev	Maximum	Resource causing Maximum	Method
▶ McCabe Cyclomatic Complexity (avg/i		4.778	4.973	16	/Bowling-Management-System/code/Lane.java	normalThrow
▶ Number of Parameters (avg/max per		0.667	1.106	4	/Bowling-Management-System/code/Lane.java	markScore
▼ Nested Block Depth (avg/max per me		2.333	1.599	5	/Bowling-Management-System/code/Lane.java	partyAssigne
▼ Lane		2.333	1.599	5	/Bowling-Management-System/code/Lane.java	partyAssigne
partyAssignedAndGameFinished	5					
receivePinsetterEvent	5					
firstBallStrike	5					
partyAssignedAndGameNotFinis	4					
normalThrow	4					
run	3					
resetScores	3					
getScore	3					
Lane	1					
resetBowlerIterator	1					
assignParty	1					
markScore	1					
lanePublish	1					
isPartyAssigned	1					
isGameFinished	1					
getPinsetter	1					
pauseGame	1					
unPauseGame	1					
▶ Depth of Inheritance Tree (avg/max pe		2	0	2	/Bowling-Management-System/code/Lane.java	
▶ Weighted methods per Class (avg/max	86	86	0	86	/Bowling-Management-System/code/Lane.java	

4. Lack of Cohesion of Methods

4.1 Measurements tell about the system

- Our limit for LCOM was set to 0.85
- It should be kept low, although it can go high for other reasons too like using getters and setters in java.
- Classes that violated our constrain are (along with their respective LCOM):
 1. LaneEvent - 0.91
 2. NewPatronView - 0.894
 3. LaneView - 0.88
 4. Lane - 0.851
- Inference
 1. LaneEvent has high LCOM because of getters and dosen't need to be split.
 2. NewPatronView has high LCOM because of getters and dosen't need to be split.
 3. LaneView does needs splitting.
 4. Lane view does have one liner functoins, may need splitting.

4.2 Guide our refactoring

- Split up method into simpler components.

4.3 Refactoring affect metrics

Metrics - LaneView.java - Lack of Cohesion of Methods (avg/max per type)						
Metric	Total	Mean	Std. Dev	Maximum	Resource causing Maximum	Method
▶ McCabe Cyclomatic Complexity (avg/max per type)		5.167	6.543	19	/Bowling-Management-System/code/LaneView	receiveLaneE
▶ Number of Parameters (avg/max per type)		0.833	0.687	2	/Bowling-Management-System/code/LaneView	LaneView
▶ Nested Block Depth (avg/max per method)		2.333	1.374	5	/Bowling-Management-System/code/LaneView	receiveLaneE
▶ Depth of Inheritance Tree (avg/max per class)		1	0	1	/Bowling-Management-System/code/LaneView	
▶ Weighted methods per Class (avg/max per class)	31	31	0	31	/Bowling-Management-System/code/LaneView	
▶ Number of Children (avg/max per type)	0	0	0	0	/Bowling-Management-System/code/LaneView	
▶ Number of Overridden Methods (avg/max per type)	0	0	0	0	/Bowling-Management-System/code/LaneView	
▼ Lack of Cohesion of Methods (avg/max per type)		0.88	0	0.88	/Bowling-Management-System/code/LaneView	
LaneView		0.88				
▶ Number of Attributes (avg/max per type)	15	15	0	15	/Bowling-Management-System/code/LaneView	
▶ Number of Static Attributes (avg/max per type)	0	0	0	0	/Bowling-Management-System/code/LaneView	
▶ Number of Methods (avg/max per type)	6	6	0	6	/Bowling-Management-System/code/LaneView	
▶ Number of Static Methods (avg/max per type)	0	0	0	0	/Bowling-Management-System/code/LaneView	
▶ Specialization Index (avg/max per type)		0	0	0	/Bowling-Management-System/code/LaneView	
Number of Classes	1					
Number of Interfaces	0					
Total Lines of Code	166					
▶ Method Lines of Code (avg/max per method)	133	22.167	25.99	62	/Bowling-Management-System/code/LaneView	receiveLaneE

Metrics - NewPatronView.java - Lack of Cohesion of Methods (avg/max per type)						
Metric	Total	Mean	Std. Dev	Maximum	Resource causing Maximum	Method
▶ McCabe Cyclomatic Complexity (avg/max per type)		1.4	0.8	3	/Bowling-Management-System/code/NewPatronView	actionPerform
▶ Number of Parameters (avg/max per type)		0.4	0.49	1	/Bowling-Management-System/code/NewPatronView	NewPatronView
▶ Nested Block Depth (avg/max per method)		1.2	0.4	2	/Bowling-Management-System/code/NewPatronView	actionPerform
▶ Depth of Inheritance Tree (avg/max per class)		1	0	1	/Bowling-Management-System/code/NewPatronView	
▶ Weighted methods per Class (avg/max per class)	7	7	0	7	/Bowling-Management-System/code/NewPatronView	
▶ Number of Children (avg/max per type)	0	0	0	0	/Bowling-Management-System/code/NewPatronView	
▶ Number of Overridden Methods (avg/max per type)	0	0	0	0	/Bowling-Management-System/code/NewPatronView	
▼ Lack of Cohesion of Methods (avg/max per type)		0.804	0	0.804	/Bowling-Management-System/code/NewPatronView	
NewPatronView		0.804				
▶ Number of Attributes (avg/max per type)	14	14	0	14	/Bowling-Management-System/code/NewPatronView	
▶ Number of Static Attributes (avg/max per type)	0	0	0	0	/Bowling-Management-System/code/NewPatronView	
▶ Number of Methods (avg/max per type)	5	5	0	5	/Bowling-Management-System/code/NewPatronView	
▶ Number of Static Methods (avg/max per type)	0	0	0	0	/Bowling-Management-System/code/NewPatronView	
▶ Specialization Index (avg/max per type)		0	0	0	/Bowling-Management-System/code/NewPatronView	
Number of Classes	1					
Number of Interfaces	0					
Total Lines of Code	96					
▶ Method Lines of Code (avg/max per method)	70	14	20.938	55	/Bowling-Management-System/code/NewPatronView	NewPatronView

Metrics - Lane.java - Lack of Cohesion of Methods (avg/max per type) ⓘ						
Metric	Total	Mean	Std. Dev	Maximum	Resource causing Maximum	Method
▶ McCabe Cyclomatic Complexity (avg/max per type)		4.778	4.973	16	/Bowling-Management-System/code/Lane.java	normalThrow
▶ Number of Parameters (avg/max per type)		0.667	1.106	4	/Bowling-Management-System/code/Lane.java	markScore
▶ Nested Block Depth (avg/max per type)		2.333	1.599	5	/Bowling-Management-System/code/Lane.java	partyAssigne
▶ Depth of Inheritance Tree (avg/max per type)		2	0	2	/Bowling-Management-System/code/Lane.java	
▶ Weighted methods per Class (avg/max per type)	86	86	0	86	/Bowling-Management-System/code/Lane.java	
▶ Number of Children (avg/max per type)	0	0	0	0	/Bowling-Management-System/code/Lane.java	
▶ Number of Overridden Methods (avg/max per type)	1	1	0	1	/Bowling-Management-System/code/Lane.java	
▼ Lack of Cohesion of Methods (avg/max per type)		0.83	0	0.83	/Bowling-Management-System/code/Lane.java	
Lane 0.83						
▶ Number of Attributes (avg/max per type)	18	18	0	18	/Bowling-Management-System/code/Lane.java	
▶ Number of Static Attributes (avg/max per type)	0	0	0	0	/Bowling-Management-System/code/Lane.java	
▶ Number of Methods (avg/max per type)	18	18	0	18	/Bowling-Management-System/code/Lane.java	
▶ Number of Static Methods (avg/max per type)	0	0	0	0	/Bowling-Management-System/code/Lane.java	
▶ Specialization Index (avg/max per type)		0.111	0	0.111	/Bowling-Management-System/code/Lane.java	
Number of Classes	1					
Number of Interfaces	0					
Total Lines of Code	285					
▶ Method Lines of Code (avg/max per type)	225	12.5	12.455	40	/Bowling-Management-System/code/Lane.java	firstBallStriki

5. Method lines of code

5.1 Measurements tell about the system

- Our limit for LCOM was set to 100.0
- It should generally not be high.
- Functions with more lines of code are more bug prone.
- Classes that violated our constrain are: None
- Inference: The code given is strong in this metric.

5.2 Guide our refactoring

- The code given is strong in this metric.
- We should make sure that this is maintained after refactoring.
- If metric constrains are not met, then split funciton into smaller functions.

5.3 Refactoring affect metrics

Strength of the Metric is retained.

6. Depth of inheritance tree

6.1 Measurements tell about the system

- Our limit for DIT was set to 5.0
- It should typically be kept between 2 to 5.
- If there is a majority of DIT values below 2, it may represent poor exploitation of the advantages of OO design and inheritance.
- It is recommended a maximum DIT value of 5 since deeper trees constitute greater design complexity as more methods and classes are involved.
- Since the code is relatively small, no lower limit was kept since not much scope of inheritance is there in relatively small code.
- Classes that violated our constrain are: None
- Inference: The code given is strong in this metric.

6.2 Guide our refactoring

- The code given is strong in this metric.
- We should make sure that this is maintained after refactoring.
- If metric constrains are not met, then split funciton into smaller functions.

6.3 Refactoring affect metrics

Strength of the Metric is retained.

7. Number of methods

7.1 Measurements tell about the system

- Our limit for number of methods was set to 7.
- Classes that violated our constrain are (along with their respective number):
 1. Lane - 17
 2. LaneEvent - 11
 3. ControlDesk - 11
 4. LaneEventInterface - 9
- Inference
 1. Lane has the highest number of methods per class and should be reduced
 2. LaneEvent has high number of methods due to getters and setters and hence can be ignored.
 3. Control desk has high number and should be reduced.
 4. LaneEventInterface is interface and has getters and setters and hence can be ignored.

7.2 Guide our refactoring

- Split up class into simpler classes with fewer methods.
- Try to merge smaller methods if possible and not violating other metrics.
- Remove dead code and unused methods.

7.3 Refactoring affect metrics

- Successful File eg :

Decreased number of methods by 4

Metrics - ControlDesk.java - Number of Methods (avg/max per type) ⓘ						
Metric	Total	Mean	Std. Dev	Maximum	Resource causing Maximum	Method
▶ McCabe Cyclomatic Complexity (avg/max per type)		1.75	1.09	4	/Bowling-Management-System/code/ControlDesk	assignLane
▶ Number of Parameters (avg/max per type)		0.375	0.484	1	/Bowling-Management-System/code/ControlDesk	ControlDesk
▶ Nested Block Depth (avg/max per type)		1.625	0.857	3	/Bowling-Management-System/code/ControlDesk	run
▶ Depth of Inheritance Tree (avg/max per type)		2	0	2	/Bowling-Management-System/code/ControlDesk	
▶ Weighted methods per Class (avg/max per type)	14	14	0	14	/Bowling-Management-System/code/ControlDesk	
▶ Number of Children (avg/max per type)	0	0	0	0	/Bowling-Management-System/code/ControlDesk	
▶ Number of Overridden Methods (avg/max per type)	1	1	0	1	/Bowling-Management-System/code/ControlDesk	
▶ Lack of Cohesion of Methods (avg/max per type)		0.625	0	0.625	/Bowling-Management-System/code/ControlDesk	
▶ Number of Attributes (avg/max per type)	4	4	0	4	/Bowling-Management-System/code/ControlDesk	
▶ Number of Static Attributes (avg/max per type)	0	0	0	0	/Bowling-Management-System/code/ControlDesk	
▼ Number of Methods (avg/max per type)	8	8	0	8	/Bowling-Management-System/code/ControlDesk	
ControlDesk						
▶ Number of Static Methods (avg/max per type)	0	0	0	0	/Bowling-Management-System/code/ControlDesk	
▶ Specialization Index (avg/max per type)		0.25	0	0.25	/Bowling-Management-System/code/ControlDesk	
Number of Classes	1					
Number of Interfaces	0					
Total Lines of Code	53					
▶ Method Lines of Code (avg/max per type)	29	3.625	3.238	9	/Bowling-Management-System/code/ControlDesk	assignLane

- Unsuccessfull File eg :

Increase in numeber of methods by 1

Metrics - Lane.java - Number of Methods (avg/max per type) 83						
Metric	Total	Mean	Std. Dev	Maximu	Resource causing Maximum	Method
▶ McCabe Cyclomatic Complexity (avg/		4.778	4.973	16	/Bowling-Management-System/code/Lane.ja	normalThrow
▶ Number of Parameters (avg/max per		0.667	1.106	4	/Bowling-Management-System/code/Lane.ja	markScore
▶ Nested Block Depth (avg/max per me		2.333	1.599	5	/Bowling-Management-System/code/Lane.ja	partyAssigne
▶ Depth of Inheritance Tree (avg/max pe		2	0	2	/Bowling-Management-System/code/Lane.ja	
▶ Weighted methods per Class (avg/ma	86	86	0	86	/Bowling-Management-System/code/Lane.ja	
▶ Number of Children (avg/max per typ	0	0	0	0	/Bowling-Management-System/code/Lane.ja	
▶ Number of Overridden Methods (avg/	1	1	0	1	/Bowling-Management-System/code/Lane.ja	
▶ Lack of Cohesion of Methods (avg/ma		0.83	0	0.83	/Bowling-Management-System/code/Lane.ja	
▶ Number of Attributes (avg/max per ty	18	18	0	18	/Bowling-Management-System/code/Lane.ja	
▶ Number of Static Attributes (avg/max	0	0	0	0	/Bowling-Management-System/code/Lane.ja	
▼ Number of Methods (avg/max per typ	18	18	0	18	/Bowling-Management-System/code/Lane.ja	
Lane						
▶ Number of Static Methods (avg/max p	0	0	0	0	/Bowling-Management-System/code/Lane.ja	
▶ Specialization Index (avg/max per typ		0.111	0	0.111	/Bowling-Management-System/code/Lane.ja	
Number of Classes	1					
Number of Interfaces	0					
Total Lines of Code	285					
▶ Method Lines of Code (avg/max per m	225	12.5	12.455	40	/Bowling-Management-System/code/Lane.ja	firstBallStriki

For the unsuccessful files however, we managed to better a different metric and hence gave in on the number of methods in this case.

8. Number of classes

8.1 Measurements tell about the system

- Our limit for number of methods was set to 30.
- The number of classes in given package is 29.
- Classes that violated our constrain are: None
- Inference: The code given is strong in this metric.

8.2 Guide our refactoring

- The code given is strong in this metric.
- We should make sure that this is maintained after refactoring.
- If metric constrains are not met, then split package into smaller packages.

8.3 Refactoring affect metrics

In order to include the newer implementations and functionalities, there was no option but to include new classes. Hence this metric was compromised.