I) BOWLING MANAGEMENT SYSTEM - DESIGN DOCUMENT :

Title information, including the name of the project, the date of submission, a list of all the team members, effort (number of hours) put in by each team member, role played by each team member.

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

Team Members :

SNo.	Team Member	Hours of Work	Contributions
1.	Mallika Subramanian	1234	• Refactoring to reduce cyclometric complexity • Increasing cohesion amongst classes and methods • Analysing and identifying responsibilites of major classes as well as the interlinked classes. • Creating UML class and sequence diagrams for the original as well as refactored code
2.	Aryaman Jain	1234	• Refactoring to reduce cyclometric 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
3.	E Nikhil	1234	• Identifying the critical code smells in the code • Refactoring to get rid of redundant code • Implementing the pause/resume and save/quit feature

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 similates 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 eligable 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 :

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 :

1. 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.
2. 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. This is shown by a dotted arrow line.
3. 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.
4. Aggregation : This resembles the "part of" relationship between 2 classes. ie : One class is "a part" of another class. Here both the classes

5. 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.

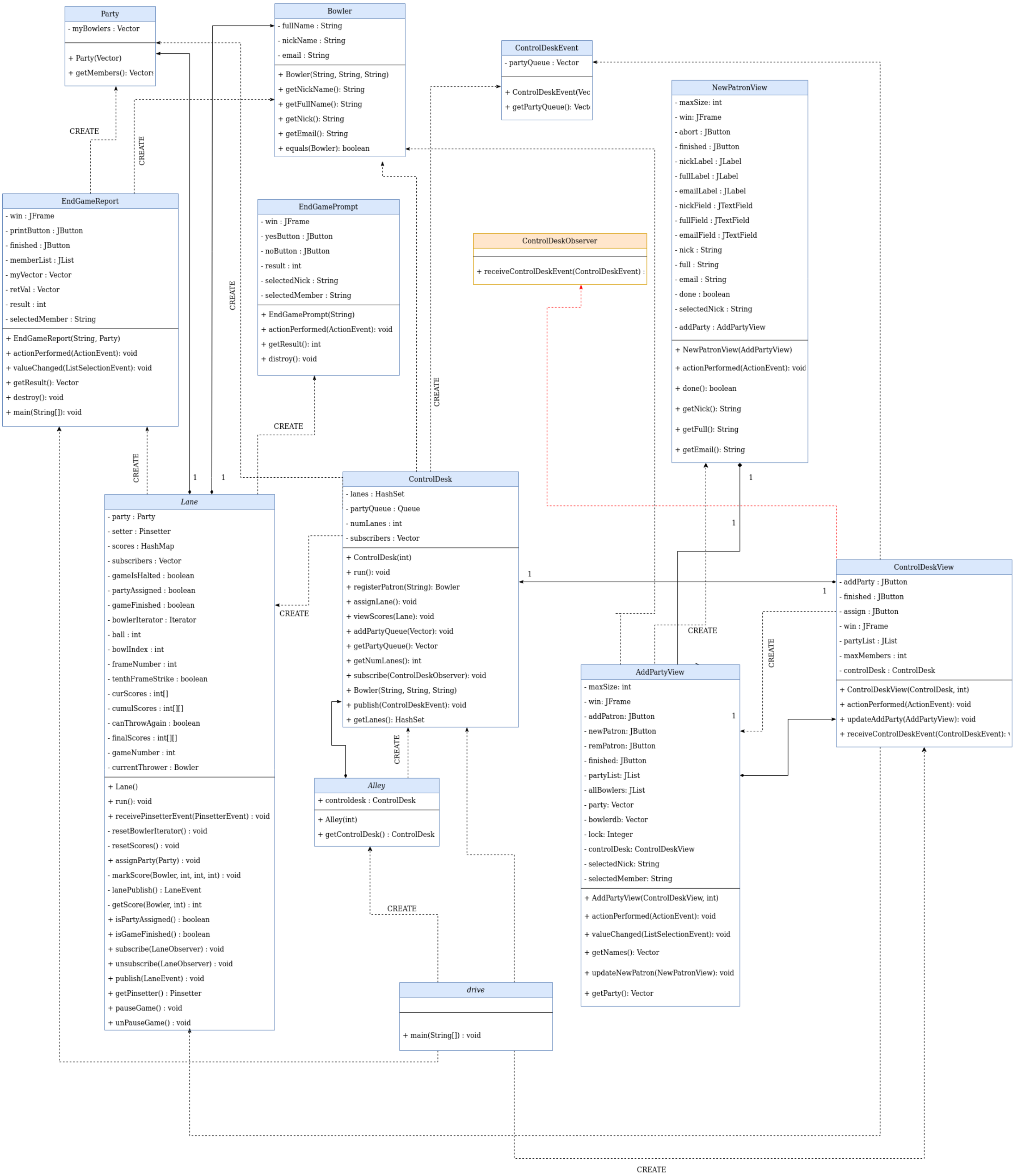
6. Specialisation : This is the exact opposite of generalisation. A specialised class is a subclass which inherits all properties from its parent class and also adds certain specific properties that are unique to it. All specific classes will be under a particular main/parent class.

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 infromation 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 bolwers.
  - 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 `ControlDeskiew` 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 compostion 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 create 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 infromation 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 **+**.

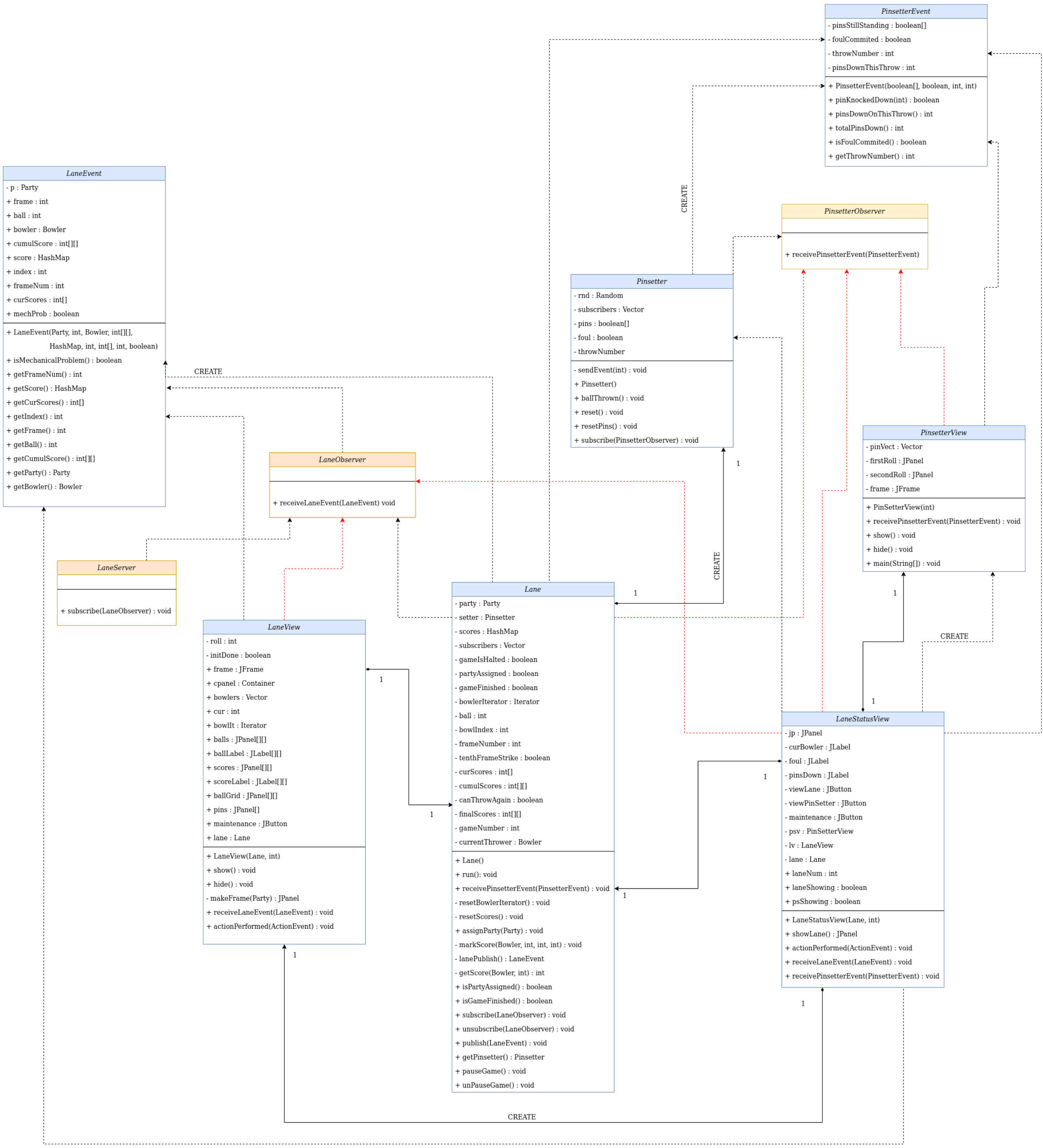
The several arrows in the diagram represent different relationships between the multiple classes.

- **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 and 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.



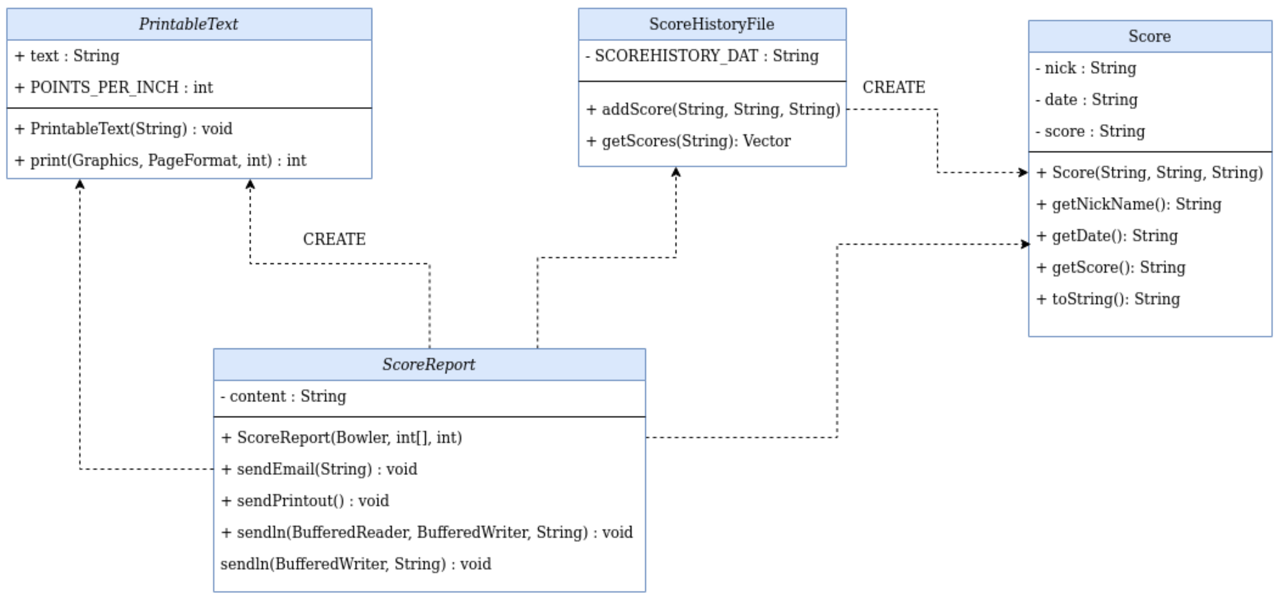
C. Fucntionality : Scoring the game and maintaing a queue of the various parties

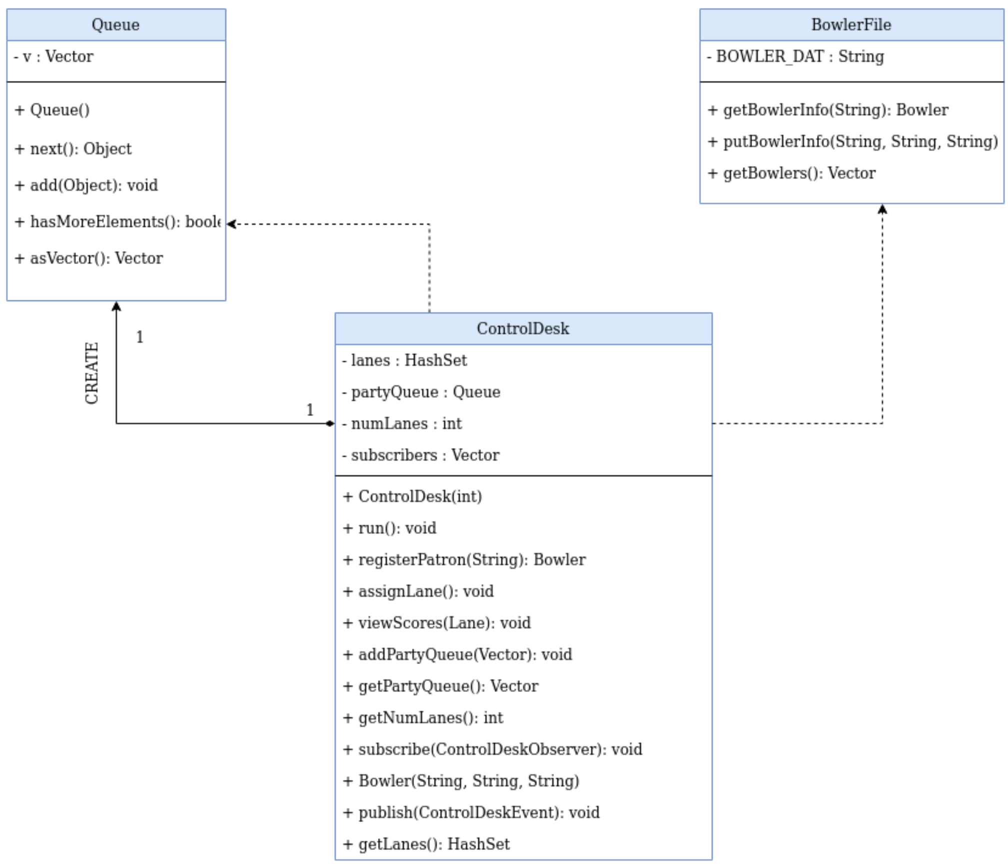
- The classes involved in this are :
    - Score
    - ScoreHistoryFile
    - PrintableText
    - ScoreReport
    - BowlerFile
    - Queue
    - ControlDesk
- The cardinalities - *number of participating objects in any association between two classes* - are mentioned on the relationship arrows of the 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.

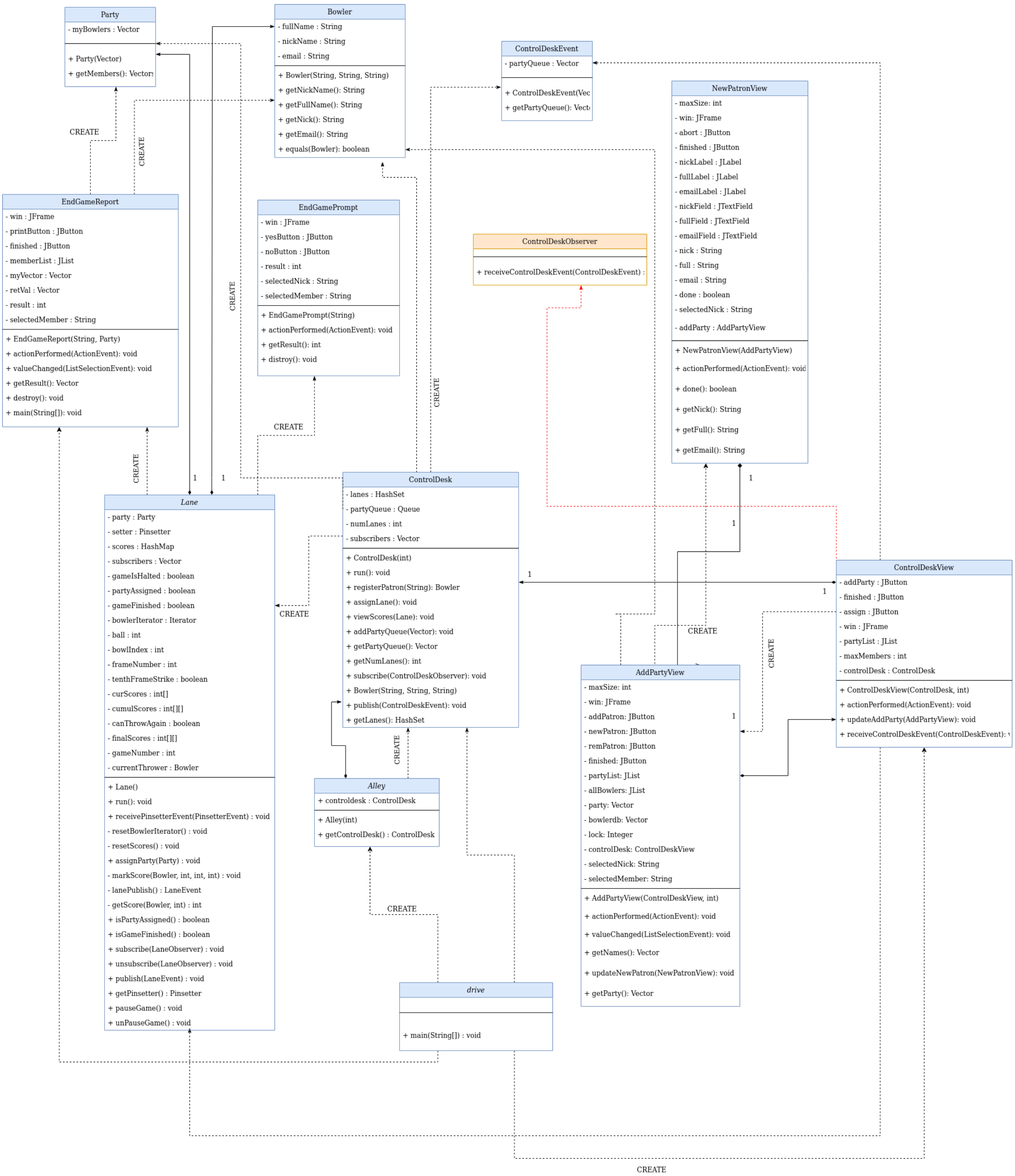




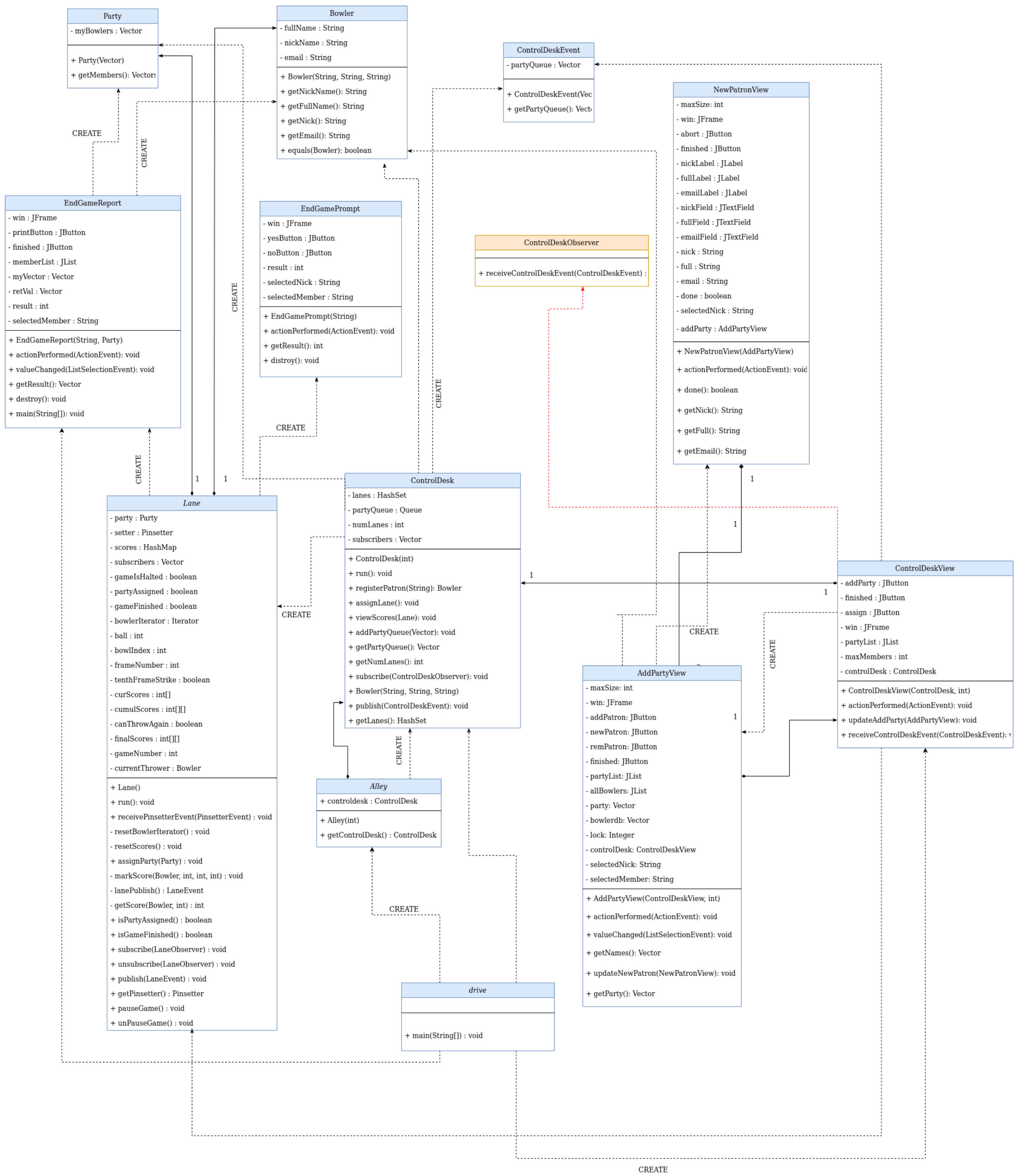
IV) SEQUENCE DIAGRAMS :

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



V) 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 funcitons 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

SNo.	File Name	Methods	Attribute	Major Functionalities	Interlinked Classes
1.	AddPartyView	• void actionPerformed() • void valueChanged() • Vector getParty() • Vector getNames() • void updateNewPatron()	• Vector party • Vector bowlerdb • ControlDeskView controlDesk • String selectedNick • String selectedMember	• Adding a new patron to party • Removing a patron from a party • Creating a new patron • Finished party selection • Returning the latest state of the party	• NewPatronViev
2.	Alley	• ControlDesk getControlDesk()	• ControlDesk controldesk	• Return Current state of ControlDesk	• ControlDesk
3.	Bowler	• String getNickName() • String getFullName ( ) • String getNick ( ) • String getEmail ( )	• String fullName • String nickName • String email	• Getter functions • Validation of the bolwer	NIL
4.	BowlerFile	• static Bowler getBowlerInfo(String nickName) • static void putBowlerInfo(String nickName,String fullName,String email) • static Vector getBowlers()	• static String BOWLER_DAT	• Adding a new bowler • Getting details of one bowler • Getting details of all bowlers	NIL
5.	ControlDesk	• void run() • Bowler registerPatron(String nickName) • void assignLane() • void addPartyQueue(Vector partyNicks) • Vector getPartyQueue() • int getNumLanes() • void publish(ControlDeskEvent event) • HashSet getLanes()	• HashSet lanes • Queue partyQueue • int numLanes • Vector subscribers	• 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	• Lane
6.	ControlDeskEvent	• Vector getPartyQueue()	• Vector partyQueue	• Returns a vector of the names of the parties in the waiting queue	
7.	ControlDeskObserver	• void receiveControlDeskEvent	NIL	• Interface for classes that observe control desk events.	• --------

SNo.	File Name	Methods	Attribute	Major Functionalities	Interlinked Classes
8.	ControlDeskView	• void actionPerformed(ActionEvent e) • void updateAddParty(AddPartyView addPartyView) • void receiveControlDeskEvent(ControlDeskEvent ce)	• int maxMembers • ControlDesk controlDesk	• Display the GUI for the control desk • Handler for actionEvents • Receive a new party from andPartyView • Receive a broadcast from a ControlDesk	• ControlDesk • AddPartyView
9.	drive	• static void main()	• int numLanes • int maxPatronsPerParty	• 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	• ControlDesk • Alley • ControlDeskVi
10.	EndGamePrompt	• EndGamePrompt(String partyName) • void actionPerformed(ActionEvent e) • int getResult() • void distroy()	• int result • String selectedNick • String selectedMember	• Displaying the end promt • Destroying the currently active game object.	• --------
11.	EndGameReport	• EndGameReport(String partyName, Party party) • void actionPerformed(ActionEvent e) • Vecotr getResult() • void distroy() • static void main(String args[]) • void valueChanged(ListSelectionEvent e)	• int result • String selectedMember	• Displaying the end game repor • Destroying the currently active game object.	• --------

SNo.	File Name	Methods	Attribute	Major Functionalities	Interlinked Classes
12.	Lane	• 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	• Party party • Pinsetter setter • HashMap scores • Vector subscribers • boolean gameIsHalted • 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	• 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	• Bowler • Party • Pinsetter
13.	LaneEvent	• Setters and getter funcitons only	• Party p • int frame • int ball • Bowler bowler • boolean mechProb	• Setter and getter functions for all lane functionalities	• Party • Bowler
14.	LaneEventInterface	• An interface class	• ------	• Interfaces the multiple classes	• Party • Bowler
15.	LaneObserver	• An interface class	• ------	• Interfaces the multiple classes	• ------
16.	LaneServer	• An interface class	• ------	• Interfaces the multiple classes	• -------
17.	LaneStatusView	• LaneStatusView(Lane lane, int laneNum) • JPanel showLane() • void receiveLaneEvent(LaneEvent le) • void receivePinsetterEvent(PinsetterEvent pe)	• PinSetterView psv • LaneView lv • Lane lane • int laneNum • boolean laneShowing • booleean psShowing	• Rendering the GUI for the status of the lanes	• PinSetterView • • LaneView • Lane
18.	LaneView	• void show() • void high() • Jframe makeFrame • void receiveLaneEvent(LaneEvent le)	• int cur • int roll • boolean initDone • Iterator bowlIt • Lane lane	• Render the view GUI for the alley lanes	• Lane

SNo.	File Name	Methods	Attribute	Major Functionalities	Interlinked Classes
19.	NewPatronView	• void actionPerformed() • void valueChanged() • Vector getParty() • Vector getNames() • void updateNewPatron()	• int maxSize • boolean done • Strinf selectedNick • AddPartyView addParty • String selectedMember	• Setter and Getter functions	• AddPartyView
20.	Party	• Vector getMembers()	• Vecotr myBowlers	• Accessor for members belonging to a party	• ------
21.	Pinsetter	• void ballThrown() • void reset() • void resetPins() • void subscribe(PinsetterObserver subscriber)	• Vector subscribers • Random rnd • boolean[] pins • boolean foul • int throwNumber	• 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	• PinsetterObser
22.	PinsetterEvent	• boolean pinsKnockedDown() • int pinsDownOnThisThrow() • int totalPinsDown() • boolean isFoulCommited() • int gerThrowNumber	• boolean[] pinsStillStanding • boolean foulCommited • int throwNumber • int pinsDownThisThrow	• Includes functionalities that mimic the dropping of pins and probabilistaically (or randomly) determines this	• ------
23.	PinsetterObserver	• An interface class	• ------	• Interfaces the multiple classes	• -------
24.	PinSetterView	• void receivePinsetterEvent()	• Vector pinVect	• 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	• ------
25.	PrintableText	• int print(Graphics g, PageFormat pageFormat, int pageIndex)	• String text • int POINTS_PER_INCH	• Displays the graphical text on the UI including colour	• ------
26.	Queue	• void add(Object o) • boolean hasMoreElements() • Vector asVector() • Object next()	• Vector v	• Creates a new Queue	• -----

SNo.	File Name	Methods	Attribute	Major Functionalities	Interlinked Classes
27.	Score	• constructor, getter and setter functions	• String nick • String date • String score	• Sets the scores for the players in the game	• ------
28.	ScoreHistoryFile	• Vector getScores(string nick) • void addScore()	• String SCOREHISTORY_DAT	• Writes the scores of the playes into a .DAT file after a game finishes. Makes use of I/O options, reading/writing to a buffer etc	• ------
29.	ScoreReport	• void sendEmail() • void sendPrintout() • void sendln()	• String content	• Generates the ScoreReport and sends it via email/printout to the user.	• Bowler

VI) ANALYZING THE ORIGINAL DESIGN :

VII) CODE SMELLS

VIII) ANALYZING THE REFACTORED DESIGN

IX) 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 cyclometic 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

TODO

---

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 partioned well with the resulting methods.

2.3 Refactoring affect metrics

TODO

---

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

TODO

---

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 nees 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

TODO

---

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

TODO

---

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

TODO

---

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

TODO

---

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

TODO