Unit-1 Project: Bowling Management System - Reverse Engineering and Refactoring

Course: Software Engineering Course Instructor: Dr. Raghu Reddy

Date of Submission: 20th February 2021

Github link:
https://github.com/bhupendra-sharma/Bowling-Alley-Unit1-SE

Name	**Roll Number**	**Contribution**
Vivek Pareek	2021701001	Analysis, refactoring and implementation of refactored design, code smells identified and removed, original and refactored metrics analysis and document preparation.
Pawan Patidar	2020201031	Created original class and sequence diagrams, document design and preparation.
Bhupendra Sharma	2021201020	Evaluation of original codebase, modularizing the code, improving complexity and removing redundancy, original and refactored metrics analysis, document preparation.
Ziyad Naseem	2021201021	Analysis of code smells and removal of dead and unused code, Designing of refactored class and sequence diagrams, original and refactored metrics analysis, and document preparation.

Total Amount of Hours: 60 Hours

1. Introduction

1.1. Overview

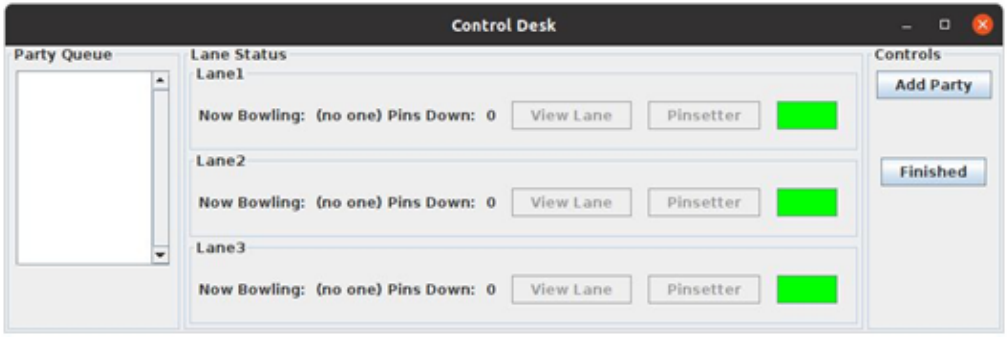
This document compares and discusses the original and the refactored design of the Bowling Management Alley Program.

The program is implemented in Java. The code is a simulator of a Bowling Alley, wherein the system simulates a Control Panel which has the functionalities of Adding a New Patron, Creating Bowling Parties, Assigning Lanes, Keeping Scores, Displaying the Pinsetter Visuals, Displaying the Lane Status and Score, and many other related functionalities.

The Original Design of the Bowling Alley was given with a high-level Software Requirement Specification which detailed high-level functioning of the system, No documentation except for comments in the codebase was available.

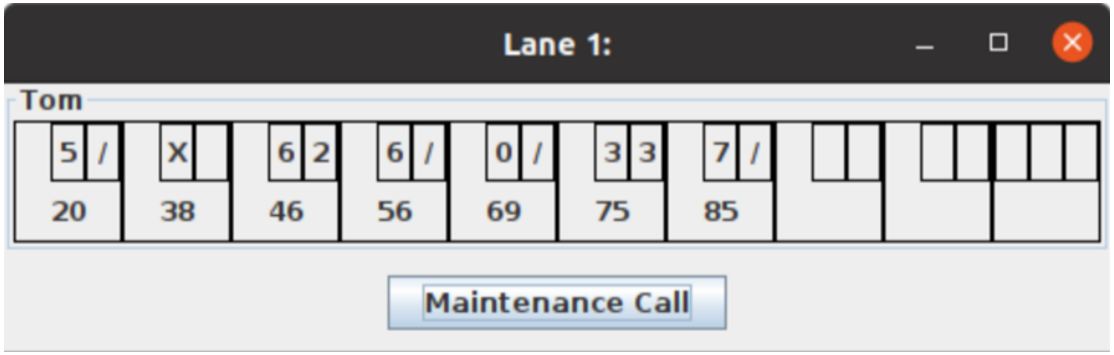
In this project, we aim to reverse engineering the given codebase into its original design, refactor the original design, and effectuate the refactoring into the codebase.

1.2. Functionality and Working of the System



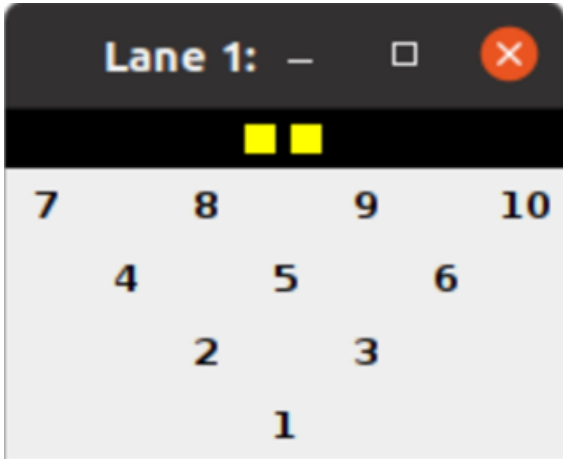
The Control Desk Window shows the central control of the Bowling Alley Management System. From this window, a user can choose to “Add Party” which registers a team of bowlers who would like to be assigned a lane and begin playing. The User can also choose to terminate the Control Desk by clicking the finish button. The Party Queue panel on the right displays the parties that are not assigned to any Lane. As soon as a party registers and a lane is available the party is assigned to that lane and begins playing.

When a User chooses to “Add Party” he is led to another window which allows him to select a set of players (Max: 5) from the existing database to add them to the party. The User can also choose to add a new player by clicking “New Patron”, which registers a new player into the Bowler Database. The User may remove a player from the party using the “Remove Member” option. Once the user is done making the party, he clicks “Finished” to begin the game, which either assigns a lane to the party if the lane is available or puts the party in a queue.



The user can choose to view the status and score of a game by clicking the “View Lane” option from the Control Desk. From here, the User can respond to any maintenance calls made by the parties which are then resolved from the Control Desk.

The user can also choose to look at the Pinsetter view that displays the current state of the Pins on the bowling lane and also displays the current throw number.



2. Original Design

2.1. Overview

The Bowling Management System has a total of 29 classes which are derived from classes like Thread, ActionEvent, Serializable, etc.

Every class is contained within a separate file. Each class has attributes and methods that describe the functionality of that class. The classes in the Bowling Management System interact with one another to simulate the entire working of a Bowling Alley, the Control Desk, and the Games.

2.2. Class Responsibility Table

#	Class Name	Attributes	Methods	Major Functionalities
1	AddPartyView	• maxSize • win • addPatron • newPatron • remPatron • finished • partyList • allBowlers • party • bowlerdb • lock • controlDesk • selectedNick • selectedMember	• actionPerformed() • valueChanged() • getNames() • updateNewPatron() • getParty()	• Handles the UI • Creating a new patron • Adding a new patron to a party • Removing a new patron from a party • Finish a party selection
2	Alley	• controldesk	• getControlDesk()	• Creates the Control Desk with a given number of lanes. • Creates an object of the ControlDesk • Return the current state of ControlDesk
3	BowlerFile	• BOWLER_DAT	• getBowlerInfo() • putBowlerInfo() • getBowlers()	• Creates a new bowler and adds it to the database • Fetching the details of a single bowler • Fetching the details of all bowlers • Retrieves bowler information from the database and returns Bowler objects with populated fields

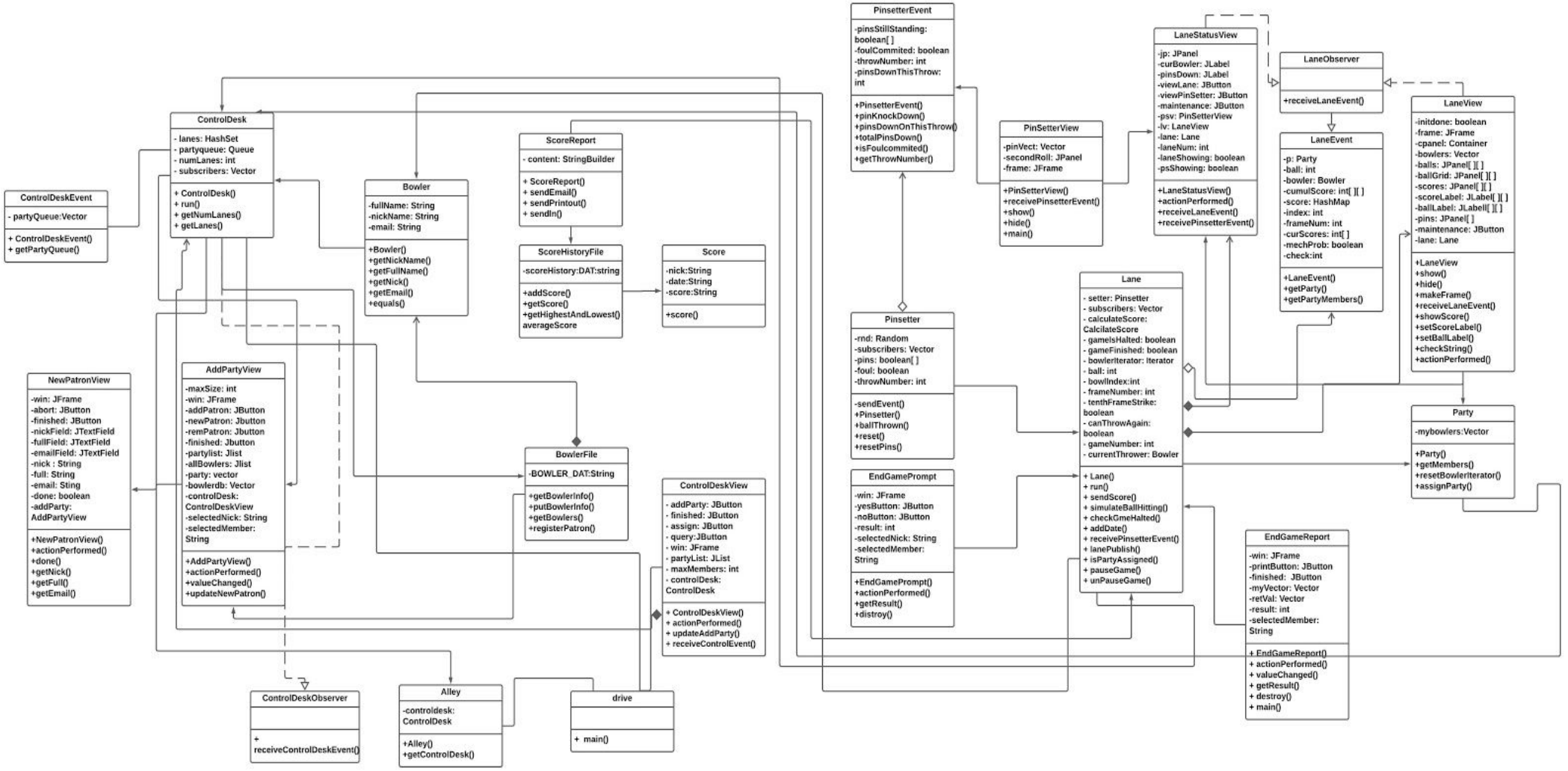
4	Bowler	• fullName • nickName • email	• getNickName() • getFullName() • getNick() • getEmail() • equals()	• Contains all the bowles' information, including the nickname, full name, and email • Handles retrieval of Bowler information
5	ControlDesk	• lanes • partyQueue • numLanes • subscribers	• run() • registerPatron() • assignLane() • addPartyQueue() • getPartyQueue() • getNumLanes() • subscribe() • publish() • getLanes()	• Acts as the control panel for the bowling system • Assigns lanes to a party as soon as a party is registered • Handles the waiting queue of the parties and assigns lanes to them • Creates a new patron • Handles party formation and selection • Returns the party and patron details to the GUI from the ControlDeskView class • Performs the main control desk thread loop
6	ControlDeskEvent	• partyQueue	• getPartyQueue()	• Holds a vector of strings containing the names of the parties in the waiting queue to be displayed on the side panel of the Control Desk View
7	ControlDeskObserver		• receiveControlDeskEvent()	• Interface for classes that observe ControlDesk events
8	ControlDeskView	• addParty • finished • assign • win • partyList • maxMembers • controldesk	• actionPerformed() • updateAddParty() • receiveControlDeskEvent()	• GUI for the control desk • Displays all the controls and input fields for running the Bowling Management program • Contains various event handlers for actionEvent • Receives the broadcast from the corresponding ControlDesk • Registers parties from addPartyView
9	drive		• main()	• This class is the driver of the Bowling Management System program • Creates the Alley given the number of lanes and the maximum number of people allowed in a party • Instantiates the ControlDesk object • Sends the ControlDesk object to the ControlDeskView to render the GUI
10	EndGamePrompt	• win • yesButton • noButton • result • selectedNick • selectedMember	• actionPerformed() • getResult() • destroy()	• Handles the prompt when the game finishes for a party giving them an option to play another game or finish playing. • Handles different inputs including the Yes/No buttons and the actions related to them
11	EndGameReport	• win • printButton	• actionPerformed() • valueChanged()	• Displays the End Game report according to the user's request. • Allows the user the option whether to print the

		• finished • memberList • myVector • retVal	• getResult() • destroy() • main()	• Report or finish without printing.
12	Lane	• party • setter • scores • subscribers • gameIsHalted • partyAssigned • 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()	• Drives from the Thread interface • Simulates the bowling lanes in the alley • Performs cyclic rounds for each of the bowlers turns • Assigns parties to an available lane • Keeps track of the current score of the players in a party • Calculates the scores for a given game • Calls EndGameReport and ScoreReport to generate the reports after the end of the game • Publishes any changes in the states to notify and update all the subscribers • Registers and unregisters observers tracking the lane status and game status
13	LaneEvent	• p • frame • ball • bowler • cumulScore • score • index • frameNum • curScores • mechProb	• isMechanicalProblem() • getFrameNum() • getScore() • getCurScores() • getIndex() • getFrame() • getBall() • getCumulScore() • getParty() • getBowler()	• Handles the game over part, pause and unpause aspects of a game • Contains the getters and setters for all the game functionality
14	LaneEventInterface		• getFrameNum() • getScore() • getCurScores() • getIndex() • getFrame() • getBall() • getCumulScore() • getParty() • getBowler()	• Interface for the LaneEvent Class
15	LaneObserver		• receiveLaneEvent()	• Interface. Not used anywhere

16	LaneServer		• subscribe()	• Interface. No used anywhere
17	LaneStatusView	• ip • curBowler • foul • pinsDown • viewLane • viewPinsetter • maintenance • psv • lv • lane • laneNum • laneShowing • psShowing	• showLane() • actionPerformed() • receiveLaneEvent() • receivePinsetterEvent()	• Defines the visuals of a lane and the information displayed for the same • Buttons ViewLane, displays the particular lane simulator and shows the scoreboard for all the patrons in the party for all the ten frames • The Pinsetter view displays all the pins, current throw, and the status of the pins, i.e., knocked out or standing
18	LaneView	• roll • initDone • frame • cpanel • bowlers • cur • bowlIt • balls • ballLabel • Scores • scoreLabel • ballGrid • pins • maintenance • lane	• makeFrame() • receiveLaneEvent() • actionPerformed()	• Renders the GUI view for the Lanes in the alley
19	NewPatronView	• maxSize • win • abort • finished • nickLabel • fullLabel • emailLabel • nickField • fullField • emailField • nick • full • Email • done • selectedNick	• actionPerformed() • done() • getNick() • getFull() • getEmail()	• Getters and setters for the patron informations

		• selectedMember • addParty		
20	Party	• myBowlers	• getMembers()	• Holds a vector of all the players in a party
21	Pinsetter	• rnd • subscribers • pins • foul • throwNumber • sendEvent()	• ballThrown() • reset() • resetPins() • subscribe() • sendEvent()	• Updates the states of the pins across to all the subscribers • Simulates the ball throw action and randomizes the outcome result of the simulated ball throw which results either in a foul, or a random number of pins
22	PinsetterEvent	• pinsStillStanding • foulCommitted • throwNumber • pinsDownThisThrow	• pinKnockedDown() • totalPinsDown() • isFoulCommitted() • getThrowNumber() • receivePinsetterEvent() • show() • hide()	• Functionalities that mimic the dropping of the pins • Checks if foul is committed • Gets information pertaining to a pinsetter
23	PinsetterObserver	•	• receivePinsetterEvent()	•
24	PinSetterView	• pinVect • firstRoll • secondRoll • frame	• receivePinsetterEvent() • show() • hide() • main()	• Pinsetter GUI displaying the current throw • Receives the current state of the pinsetter and updates the GUI of the Pinsetter based on the state of the pins
25	PrintableText	• text • POINTS_PER_INCH	• print()	• Renders the graphical text on the user interface
26	Queue	• v	• next() • add() • hasMoreElements() • asVector()	• Creates a Vectorized Queue • Responsible for creating the PartyQueue in a particular lane
27	Score	• nick • date • score	• getNickName() • getDate() • getScore() • toString()	• Facilitates storing functionality of the scores of a particular player • Contains the getters to retrieve details about the bowler • Sets the scores for the players in the game
28	ScoreHistoryFIle	• SCOREHISTORY_DAT	• getScores() • addScore()	• I/O class for writing the scores of the players into the database after the completion of a game
29	ScoreReport	• content	• sendEmail() • sendPrintout() • sendIn()	• Generates a score report and prints it on the console and emails the same to the respective player

2.3 UML Class Diagram (Original Design)



2.4 UML Sequence Diagram (Original Design)

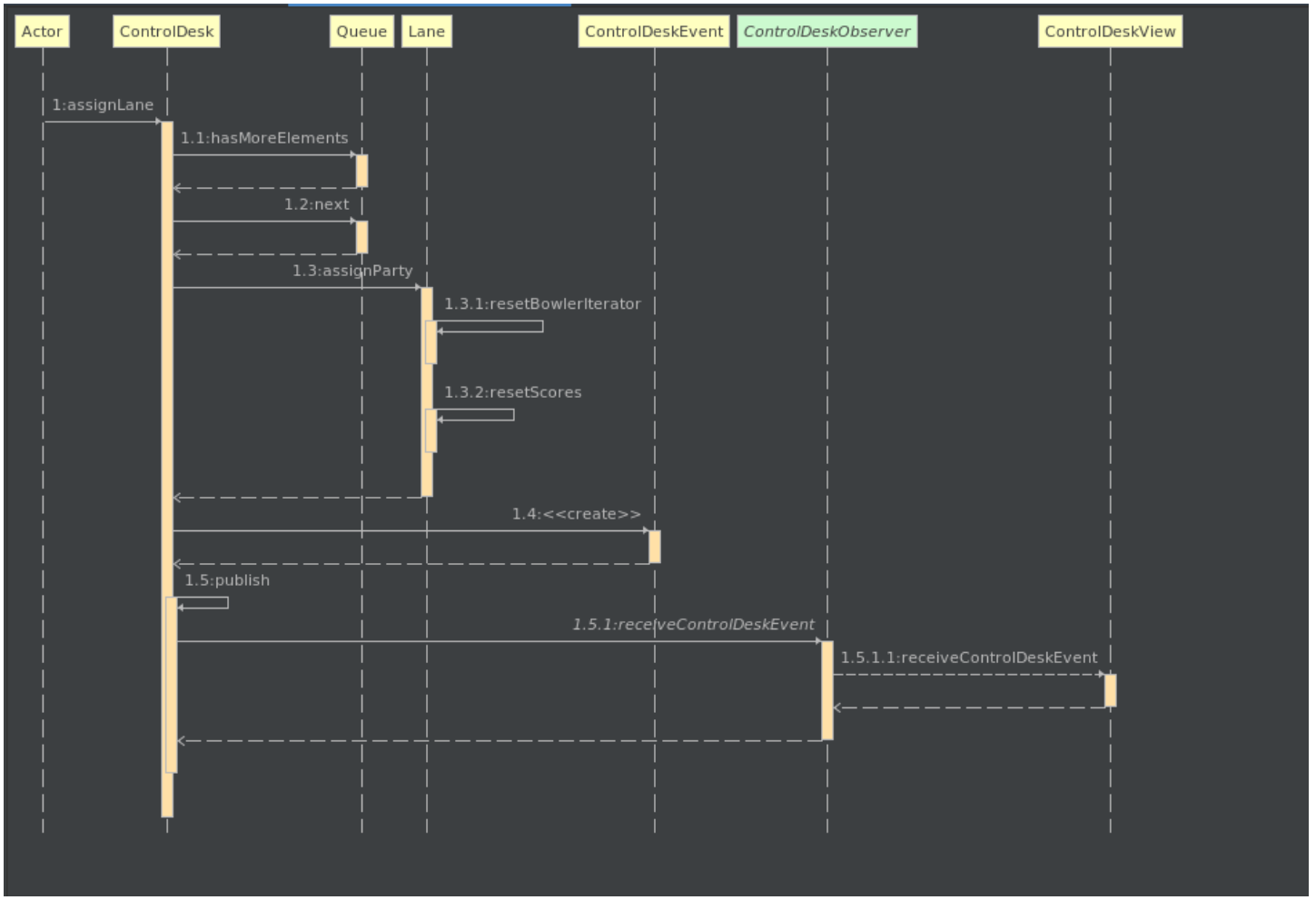


Fig: Original Sequence Diagram for assigning Lane to a party

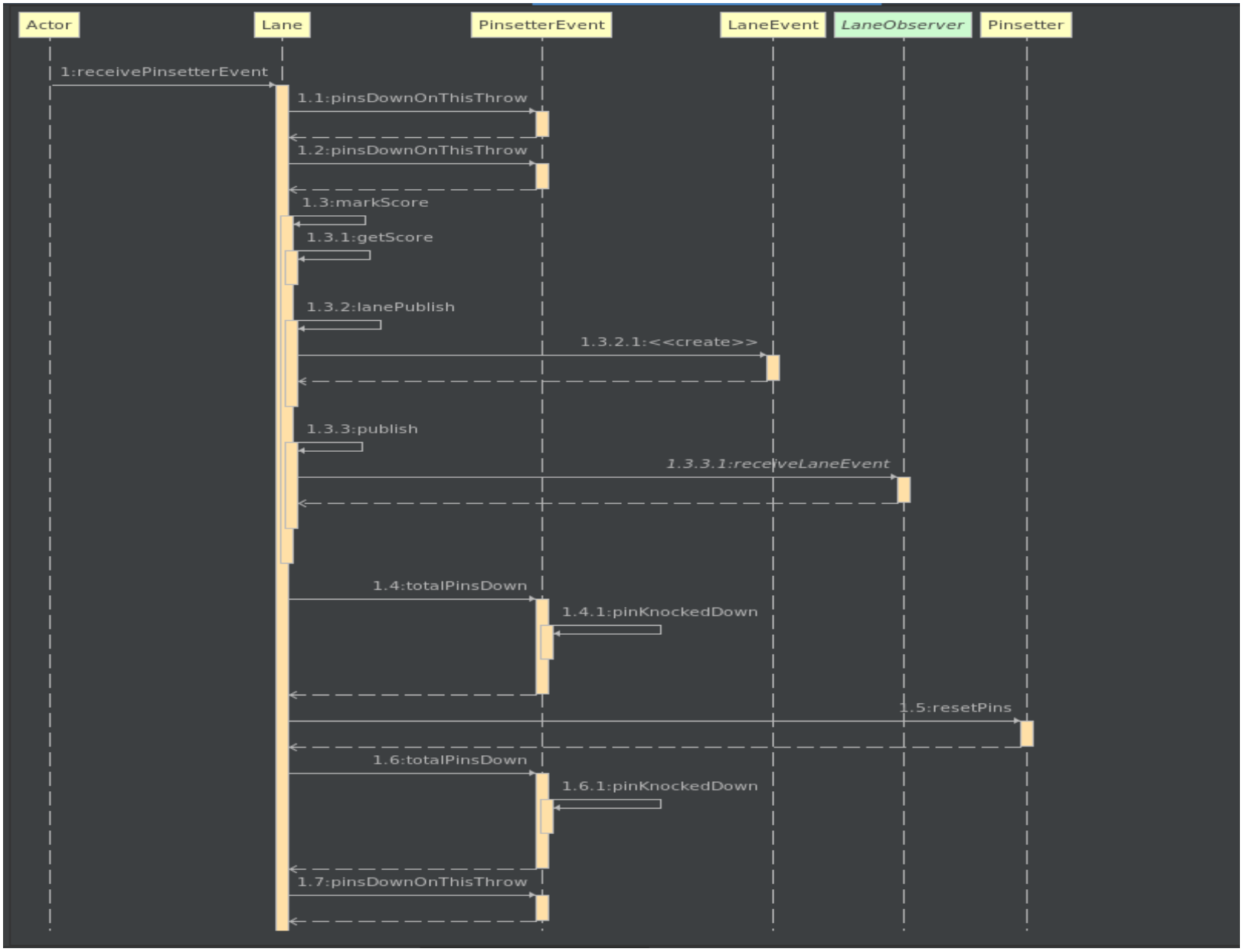


Fig: Original Sequence Diagram for pinsetter event (event of pins knocked down after throw).

2.5 Strengths and Weaknesses

The original codebase has its own strengths and weaknesses in its design and documentation.

2.5.1. Strengths

- The code is appropriately commented with comments describing the functionality of each method and class which eases the process of reverse engineering for a developer.
- All tasks and functionalities of the classes are clearly defined.
- The original design has a decent amount of cohesion amongst related classes making it easier to decipher the original design.
- The entire codebase size is not very huge.
- The codebase implements the structural pattern of Model View Controller, in that it separates the View logic which is the GUI and interaction elements, from the Model which is the program logic making the code modular to an extent.
- The codebase implements the Observer pattern at several instances which triggers certain other chained events.

2.5.2 Weaknesses

- There were a lot of duplicate codes and dead codes present in the codebase.
- The codebase was weak in terms of complexity.
- Multiple classes were present for just handling the file operations.
- Some unnecessary classes were present that were only acting as an intermediary between two other classes.
- Each observer class was implementing its own observer interface which should not be class specific.
- There were several javadoc errors in terms of syntax and formatting.

3. Refactored Design

3.1. Refactoring Overview

We started with going through the code and understanding the current design. We used **IntelliJ IDE** for this project. During the code walkthrough, we came across a lot of dead and unused code and removed it. This helped us to declutter the code. We then started with running the application and understanding the flow of information. We then started with drawing the class diagram and sequence diagram for current design. This was the first week's work.

At the start of the second week, we decided to understand the patterns being implemented in current design. We could see that the current design implements MVC pattern where the views and controllers belong to the same class. We also realized that the observer pattern was being implemented i.e the views were observers of the models. Then we started looking at the metrics using **CODEMR** metric plugin and understood that the size of the project was large, there were quite a few classes with more than optimal coupling between objects and there were many classes that had low cohesion. Then we looked at the design again and realized that the observer pattern could be optimized in the design i.e each observer in the current design had its own observer interface and own data class for communicating with the observable. We started to work on optimizing this design. This was the first half of our second week's work.

In the remaining days of the week, we started by implementing a generic observer interface. All the observers implemented this generic interface. This way we could simplify the design as the observable knew only about the generic observer interface and the actual observers were hidden from the observable. We also removed the data classes which were used to pass data between observer and observable. The observer now got the data using an instance of the observable. Now we started looking into decreasing the complexity of the project. The main issue we could identify in complexity was with Lane class. To reduce complexity, we first started looking at the methods in Lane class. We realized that instead of the lane class having score calculation logic within it, a new class for score calculation was needed. We created a Class for score calculation and then moved the scoring part from Lane into the new class. In this class, we realized that there we modularized the methods so that the code becomes more readable. We also found some unreachable logic in the code and removed that logic.

We also found some classes which were just wrappers of collection. The classes were Queue.java and Party.java. We removed these classes and started using the collection itself. We also found that the button event actions were being handled in a complex way which was increasing the complexity of some methods due to if else logic. We used anonymous classes to solve this problem. We then looked at the metrics of refactored design and could see that we could reduce the coupling, complexity and size of the project. We faced difficulty in decreasing the lack of cohesion in Lane class but we had a simpler and intuitive design, so we were fine with it. At the end of the week, we started working on this document. During coming up with the document, we realized a few more codesmells and tried to solve them. A detailed list is provided in the code smells table(section 3.4) below. This was our approach in refactoring the project.

3.2 UML Class Diagram (Refactored Design)

Here, we present the refactored design class diagrams. We have logically divided the classes into models and viewcontrollers. Some classes such as FileUtils.java will be one step higher in the hierarchy and can be used in both models and viewcontrollers. **Note:** This division is logical and only for the sake of making class diagrams understandable. We have not done this division in code, they are at the same hierarchy and not divided into modules in code.

Figure 1 below represents a class diagram of model classes. We have model classes and relationships among them. Also, the model classes are associated with the EventObserver interface and are hidden to the views. Hence, we have logically divided the entire class diagram into models and views.

Figure 2 below represents a class diagram of view classes. All view classes implement a single EventObserver interface unlike before where each observer had its own interface.

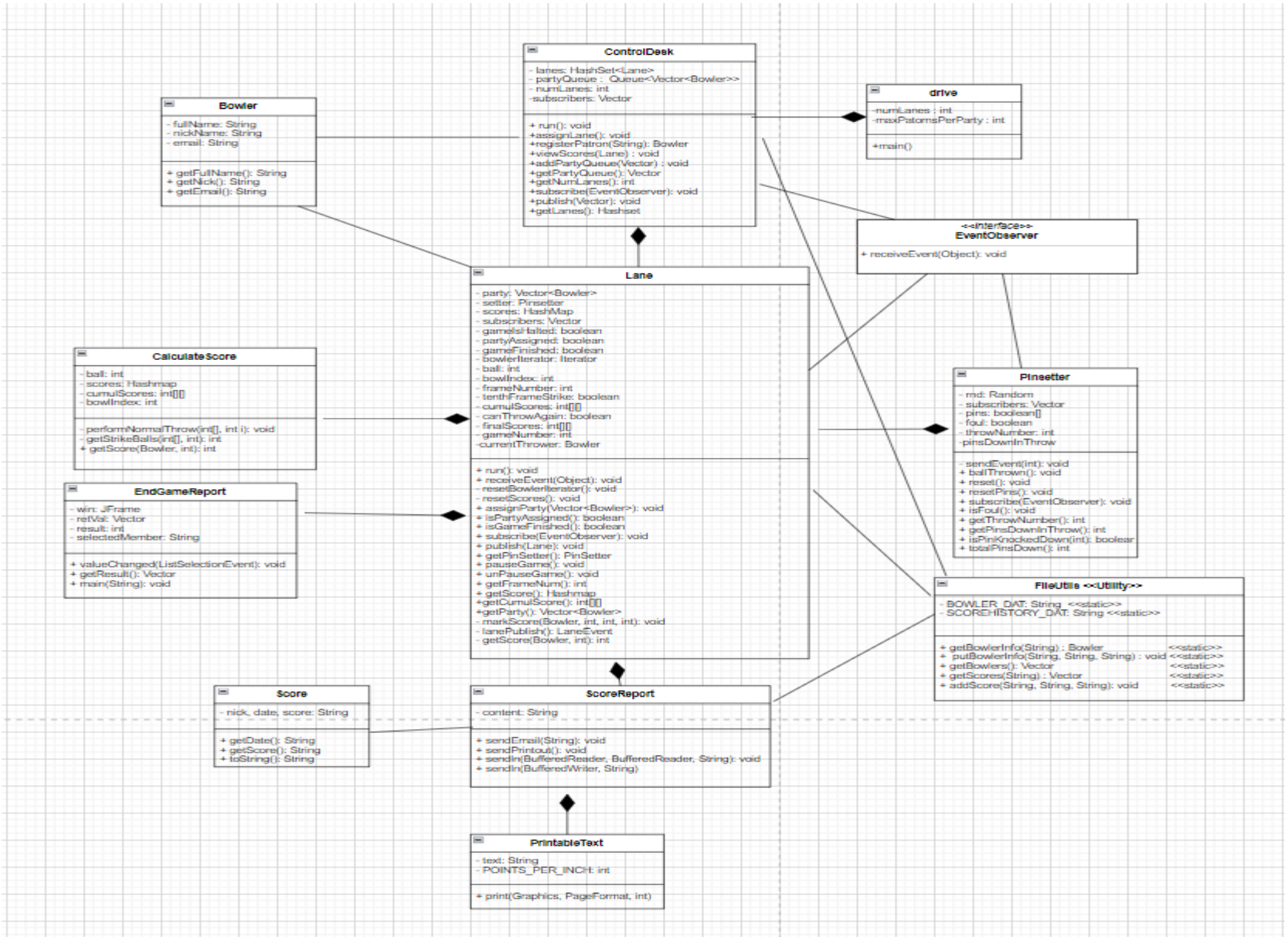


Fig. 1: Refactored class diagram for logical model classes.

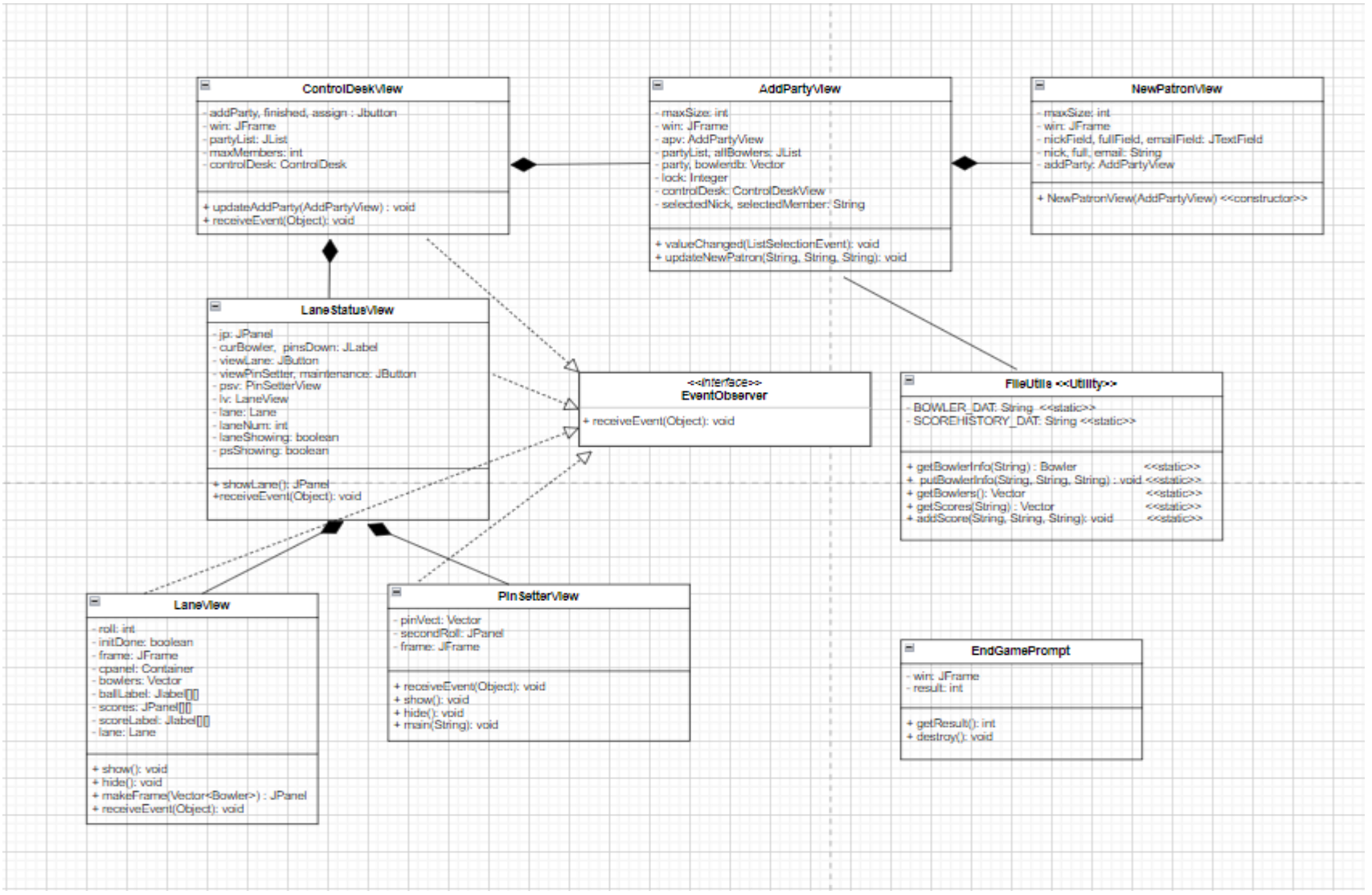


Fig. 2: Refactored Class diagram for logical view classes.

3.3 UML Sequence diagram (Refactored Design)

Below are the sequence diagrams for assigning Lane to a party and pinsetter event (event of pins knocked down after ball is thrown) in our refactored design. We can see that the observables in this case are hidden to the actual observer and interact through the EventObserver(generic) interface. This is the main change of our design, the observer design pattern has been made generic.

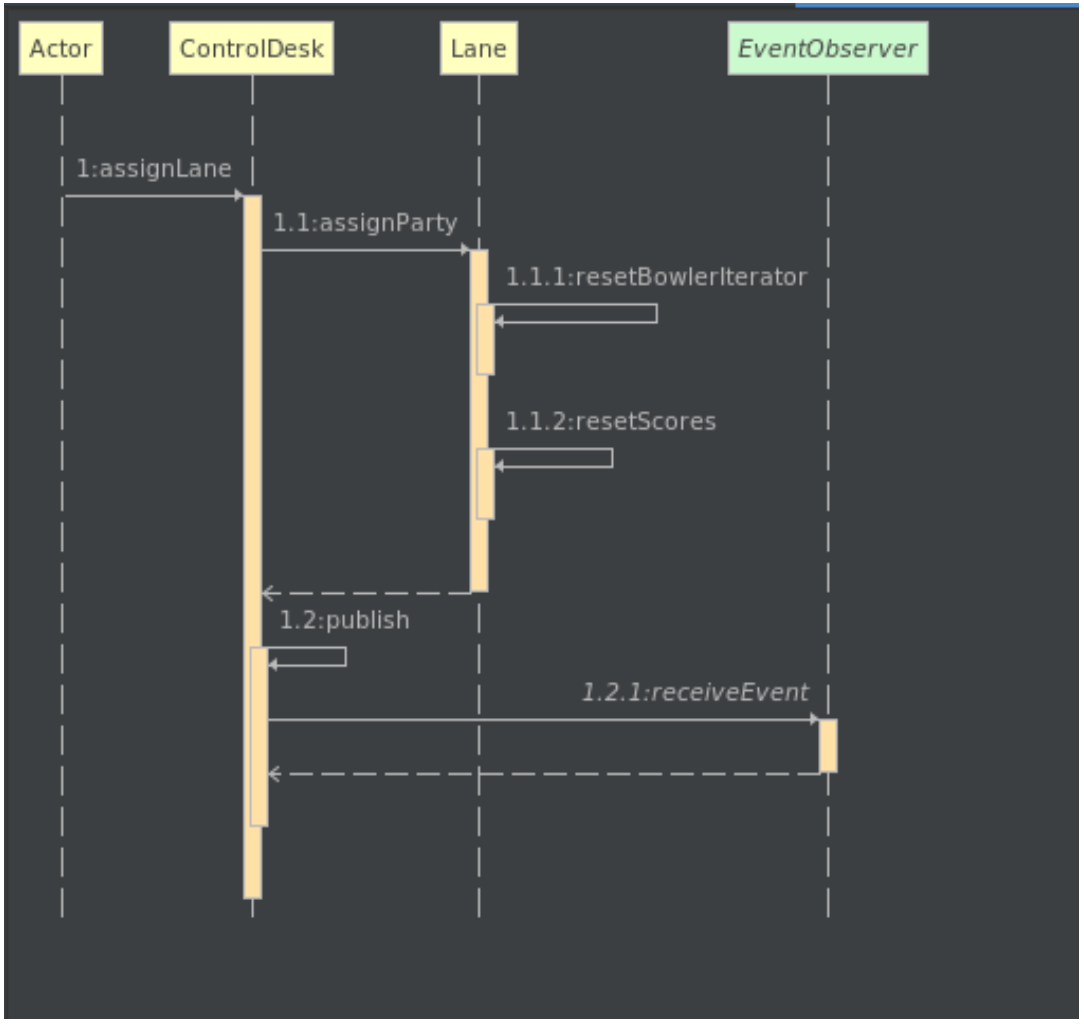


Fig: Sequence diagram for event of assigning lane to party in refactored design.

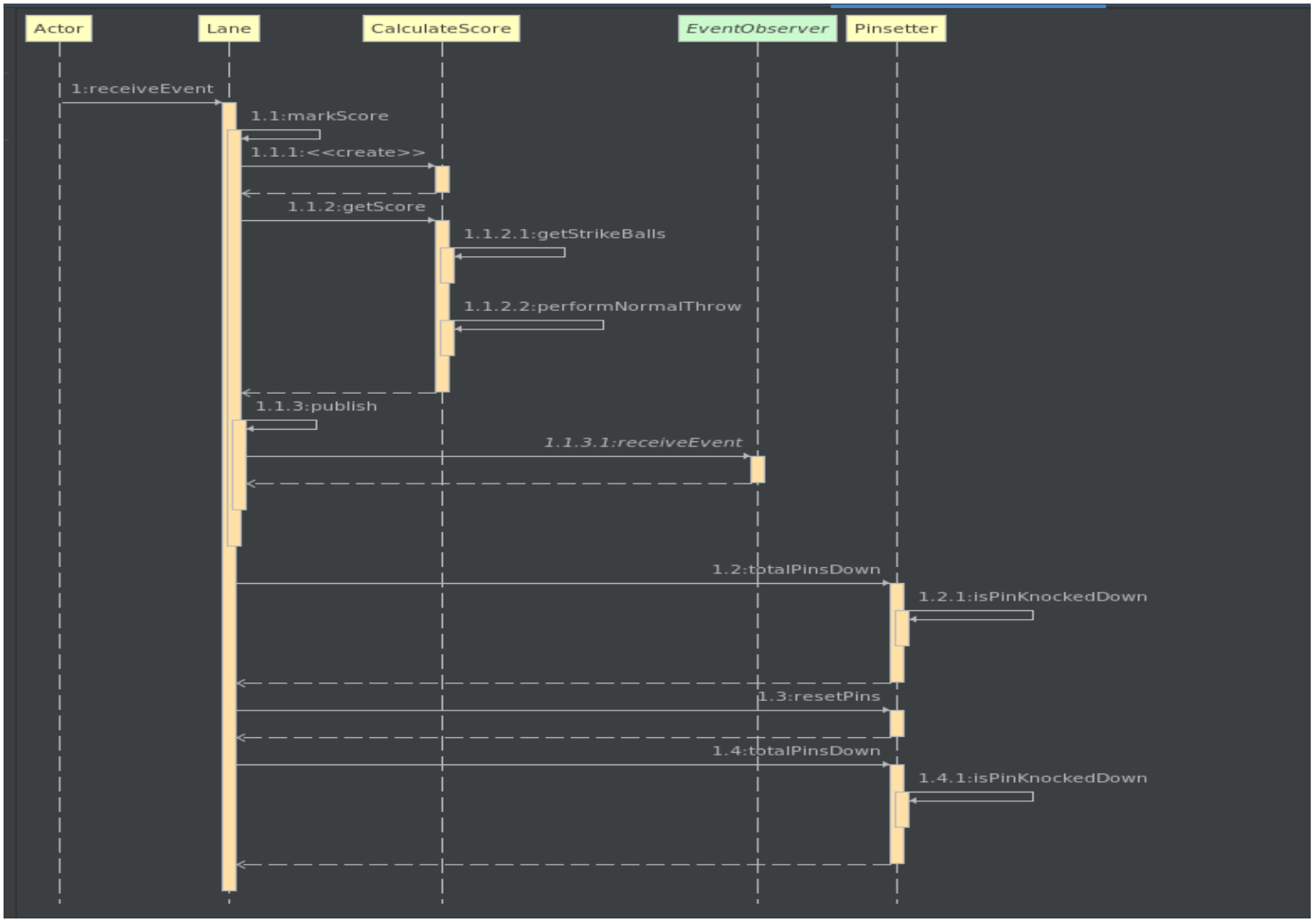


Fig: Sequence diagram for pinsetter event in refactored design.

3.4 Code Smells (Identification and Solution)

#	Code Smell	Files	Description	Solution
1.	Duplicate Code	Bowler.java	getNick() and getNickName() methods have the same definition in Bowler.java file.	Removed getNick() and called getNickName() wherever getNick() was used.
2.	Large Class + Multiple Responsibility + Lack of Modularization	Lane.java	Scoring related methods should not belong to Lane.java	Segregated scoring logic from Lane.java to CalculateScore.java and modularized getScore() method in CalculateScore.java
3.	Unnecessary Classes + Middle Man	Alley.java LaneEvent.java ControlDeskEvent.java PinSetterEvent.java Queue.java	These classes were acting just as either intermediate classes to communicate between 2 classes or they were just Data classes. Queue.java was unnecessary because it can be replaced by java Queue collection.	Used Java Collection to implement Queue.Removed Alley a it was just intermediate class.Event classes were handled by Models themselves.
4.	Multiple Classes for stateless functionality	ScoreHistoryFile.java BowlerFile.java	Multiple classes were present for just handling file operations	Added file handling logic in FileUtils.java for simpler design.
5.	Multiple Observer Interfaces	LaneObserver.java PinSetterObserver.java ControlDeskObserver.java	Each observer class was implementing its own observer interface.Interface should not be class specific.	Implemented a generic observer interface as EventObserver.java for handling all observer classes.Removed all other individual interface and data classes.
6.	Unused Method(Dead Code)	AddPartyView.java Bowler.java EndGameReport.java Lane.java NewPatronView.java Score.java	Presence of unused methods in given classes.	Removed all the unused methods that were present in the mentioned classes that helped in increasing code readability and reducing LOC.
7.	Unused Classes (Dead Code)	LaneEvenInterfacet.java LaneServer.java	Presence of classes and Interfaces that were not used in the project.	Remove the unused Interface and class.
8.	Inappropriate Intimacy	LaneStatusView.java Lane.java	LaneStatusView.java is calling Lane.getBowler().getNick() to obtain a bowler's nickname which causes Inappropriate Intimacy.	Created a getBowlerNick() function in Lane.java that will directly return bowler's nickname and can help in reducing Inappropriate Intimacy
9.	Long Parameter List	LaneEvent.java	Constructor of LaneEvent.java expects more than 6 parameters .	This was just a data class so it was removed and it help in avoiding functionChanged the names with the information they represent using calls will create a long parameter list.

10.	Deprecated Code	LaneStatusView.java LaneView.java Lane.java	Deprecated code such as new Integer(),was present in mentioned classes.	Replaced it with the Integer.toString()
11.	Conditional Complexity	AddPartyView.java ControlDeskView.java LaneStatusView.java LaneView.java NewPatronView.java	Button Events were handled by Java ActionListener Interface which was checking instances of each event resulting in multiple conditional statements.	Removed Java ActionListener interface implementation and handled events using anonymous classes.
12.	Wrong Commenting style for methods	In almost all classes	In almost all classes	Added and edited comments wherever required.
13.	Indecent Exposure	All Model Classes	‘Jframe frame’ and ‘Container cpanel’ were unnecessarily package-private leaving the scope of modification within the class.	Since no change is expected in these data members, so made them both final leaving no scope of accidental usage.

4. Metrics

4.1 Metrics used

We are going to group metrics into four categories: complexity, coupling, size and cohesion. Below are the metrics we have used for refactoring.

4.1.1. Complexity

Complexity takes into account the number of paths that can be taken in a program, interaction between entities and difficulty in understanding.Higher levels of complexity in software increase the risk of unintentionally interfering with interactions and so increases the chance of introducing defects when making changes.

Complexity metrics:

1. RFC (Response for class):
   The number of methods that can be potentially invoked in response to a public message received by an object of a particular class.If number of methods that can be invoked is high, then the class is considered more complex and hard to maintain.
2. WMC (Weighted Method Count)
   The weighted sum of all methods of a class. Higher the value, more complex a class is.

4.1.2. Size

Size is a metric that is measured by the number of lines or methods in the code. A very high count might indicate that a class or method is trying to do too much work and should be split up. It might also indicate that the class might be hard to maintain.Below are some of the Size metric:

1. CLOC OR LOC (Class Lines of Code)
2. NOM (Number of Methods)
3. NoCls (Number of Classes)
4. NOF (Number of Fields)

4.1.3. Coupling

Coupling is the measure of dependency of modules with each other.Let say we have 2 classes A and B,so A and B have coupling if any of the below follows:

- A has an attribute that refers to (is of type) B.
- A calls on services of an object B.
- A has a method that references B (via return type or parameter).
- A has a local variable which type is class B.
- A is a subclass of (or implements) class B.

Some of the coupling metrics are:

1. CBO(coupling between Objects)
2. ATFD(Access to Foreign Data)

4.1.4. Cohesion

Measure of how well the methods of a class are related to each other. High cohesion and low coupling tend to be preferable.

Cohesion metrics:

1. LCOM_3(Lack of cohesion of methods)

   LCOM_3 = (m - sum(ma)/a) / (m-1)

   - m number of procedures (methods) in class
   - a number of variables (attributes) in class. a contains all variables whether shared (static) or not.
   - mA number of methods that access a variable (attribute)
   - sum(mA) sum of mA over attributes of a class

2. LCAM(Lack of cohesion among methods)
   - CAM is the measure of cohesion based on parameter type of methods. LCAM = 1-CAM.

4.2 Metric Analysis and Comparison

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO	LOC	RFC	WMC	LCOM	SRFC	ATFD	LCAM	NOF
drive.java													
	low-medium	low	medium-high	low		1316		247					
Lane	medium-high	low-medium	low-medium	high	7	172	68	55	0.891	36	1	0.816	17
AddPartyView	low	low	low-medium	low	3	130	37	8	0.75	27	0	0.533	10
Bowler	low	low	low	low	0	14	4	4	0.667	0	0	0.375	3
CalculateScore	low-medium	low	low-medium	low	1	70	5	36	0.25	3	0	0.375	4
ControlDesk	low-medium	low	low-medium	low-medium	4	61	36	20	0.714	24	1	0.7	4
ControlDeskView	low	low	low-medium	low	4	93	49	5	0.9	31	2	0.533	5
EndGamePrompt	low	low	low-medium	low	0	56	19	5	0.5	16	0	0.333	2
EndGameReport	low	low	low-medium	low	1	75	30	8	0.75	26	0	0.5	4
EventObserver	low	low	low	low	0	2	1	1	0.0	0	0	0.0	0
FileUtils	low	low	low-medium	low	2	57	14	10	0.625	9	0	0.1	0
LaneStatusView	low	low	low-medium	low	4	99	20	7	0.708	14	1	0.5	12
LaneView	low-medium	low	low-medium	low-medium	2	132	35	30	0.679	31	1	0.64	7
NewPatronView	low	low	low-medium	low	1	74	16	1	0.0	15	0	0.0	8
PinSetterView	low	low	low-medium	low	1	111	22	12	0.556	17	1	0.6	3
Pinsetter	low-medium	low	low-medium	low-medium	1	63	15	22	0.759	8	0	0.606	6
PrintableText	low	low	low	low	0	21	10	5	0.0	8	0	0.4	2
Score	low	low	low	low	0	14	4	4	0.444	0	0	0.375	3
ScoreReport	low	low	low-medium	low	4	65	29	13	0.0	23	2	0.567	1
drive	low	low	low	low	2	7	3	1	0.0	1	0	0.0	0

Fig: Refactored metrics

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO	LOC	RFC	WMC	LCOM	SRFC	ATFD	LCAM	NOF
drive.java													
	low-medium	low	medium-high	low		1438		327					
ControlDesk	low-medium	low-medium	low-medium	medium-high	7	68	40	22	0.714	26	2	0.779	4
ControlDeskView	low-medium	low-medium	low-medium	low-medium	6	87	67	8	0.619	38	4	0.625	7
Lane	medium-high	low-medium	low-medium	medium-high	10	227	76	87	0.854	42	2	0.782	18
LaneStatusView	low	low-medium	low-medium	low-medium	7	93	38	17	0.712	22	3	0.667	13
AddPartyView	low-medium	low	low-medium	low-medium	4	127	63	21	0.743	36	1	0.694	14
Alley	low	low	low	low	1	6	2	2	0.0	0	0	0.25	1
Bowler	low	low	low	low	0	25	7	9	0.533	4	0	0.556	3
BowlerFile	low	low	low	low	1	38	12	6	0.0	9	0	0.167	0
ControlDeskEvent	low	low	low	low	0	6	2	2	0.0	0	0	0.25	1
ControlDeskObserver	low	low	low	low	1	2	1	1	0.0	0	0	0.0	0
EndGamePrompt	low	low	low-medium	low	0	55	22	8	0.833	18	0	0.5	6
EndGameReport	low	low	low-medium	low-medium	2	79	36	12	0.75	30	0	0.633	8
LaneEvent	low	low	low	medium-high	2	41	11	11	0.91	0	0	0.758	10
LaneEventInterface	low	low	low	low	2	10	9	9	0.0	0	0	0.0	0
LaneObserver	low	low	low	low	1	2	1	1	0.0	0	0	0.0	0
LaneServer	low	low	low	low	1	2	1	1	0.0	0	0	0.0	0
LaneView	low-medium	low	low-medium	low-medium	4	140	47	31	0.88	36	2	0.694	15
NewPatronView	low	low	low-medium	low	1	85	39	8	0.894	20	0	0.556	17
Party	low	low	low	low	0	6	2	2	0.0	0	0	0.25	1
PinSetterView	low	low	low-medium	low	1	111	22	11	0.667	17	1	0.6	4
Pinsetter	low	low	low	low	2	47	12	15	0.56	9	0	0.556	5
PinsetterEvent	low	low	low	low	0	26	6	9	0.75	1	0	0.5	4
PinsetterObserver	low	low	low	low	1	2	1	1	0.0	0	0	0.0	0
PrintableText	low	low	low	low	0	21	10	5	0.0	8	0	0.4	2
Queue	low	low	low	low	0	12	8	5	0.0	3	0	0.4	1
Score	low	low	low	low	0	16	5	5	0.5	0	0	0.4	3
ScoreHistoryFile	low	low	low	low	1	20	10	4	0.0	8	0	0.0	0
ScoreReport	low	low	low-medium	low	4	76	29	13	0.0	23	2	0.567	1
drive	low	low	low	low	3	8	4	1	0.0	2	1	0.0	0

Fig: Original metrics

4.2.1. Complexity

→ WMC(Weighted Method Count):
Weighted Method Count has been reduced since we have removed a lot of methods from the code. It was 327 for the original code, after refactoring the count decreased to 247.

→ RFC(Response For a Class):
Response For a Class for the entire code has reduced after refactoring the code. For major classes such as Lane it has decreased from 76 to 68, and for ControlDeskView it has reduced from 67 to 49.

4.2.2. Cohesion

→ LOC(Lack of Cohesion):
For the entire project Lack of Cohesion has decreased.

→ LCOM(Lack of Cohesion of Methods):
LCOM has been decreased for overall classes except Lane class. For Lane it has slightly increased since data class(LaneEvent) was removed and merged with Lane. For NewPatronView it was 0.894 and it has been reduced to 0 as we have made it a proper View class as no methods were necessary. For LaneView it was 0.88 and reduced it to 0.69. In the case of Lane it has increased from 0.85 to 0.89.

→ LCAM(Lack of Cohesion Among Methods):
LCAM has been decreased for all the classes except for Lane class. For ControlDesk it was 0.779 and it has been reduced to 0.7. For AddPartyView it was 0.694 and it has reduced to 0.533.But for Lane it has increased from 0.78 to 0.81.

4.2.3. Coupling

→ CBO(coupling between Objects):
For the entire project coupling has decreased. If we check for major classes such as Lane it was reduced from 10 to 7 and for ControlDesk it has reduced from 7 to 4.

→ ATFD(Access to Foreign Data):
For the entire project ATFD has decreased. If we check for major classes such as LaneStatusView it was reduced from 3 to 1 and for ControlDeskView it has reduced from 4 to 2 and for Lane it has reduced from 2 to 1.

4.2.4 Size

→ CLOC OR LOC (Class Lines of Code):
For each class, Lines of code has reduced as there was presence of dead code which has been removed. For overall original project it was 1432 and has been reduced 1316 after refactoring.

→ NOM (Number of Methods):
Number of methods has increased because we have tried to modularize the code. There were a few functions that were large in size and performed multiple functions. They have been modularized resulting in an increase in the number of methods. For example in Lane we had 17 methods now it has 21 methods.

→ NoCls (Number of Classes):
Number of classes along with the interface have decreased as there were some classes that were not used, data classes that have been removed. For the overall original project total classes and interfaces were 29 and after refactoring ,the number of classes and interfaces are 19. We have removed the following classes and Interfaces:

- ControlDeskEvent.java
- PinSetterEent.java
- LaneEvent.java
- ControlDeskObserver.java
- LaneObserver.java
- PinSetterObserver.java
- LaneEventInteraface.java
- LaneServer.java
- ScoreHistoryFile.java
- Queue.java
- Party.java
- Alley.java

→ NOF (Number of Fields):

Number of fields has been reduced as there were some unused attributes in some classes. So this metric has been reduced for some classes and is the same for remaining classes. Before refactoring we had 17 numbers of fields for NewPatronView and after refactoring we have 8 numbers of fields for NewPatronView. In case of LaneView number of fields have reduced from 15 to 7.