Software Engineering Course Project 1
Design Document
Bowling Alley Management System

Date of Submission: 11 February 2019

Team Members
Swati Tyagi - 20172096 - 25 hours
Ayush Jain - 201501209 - 25 hours
Gokul B. Nair - 201502034 - 25 hours

Link to the video and UML diagrams (Class diagrams + Sequence diagrams) : Here

Overview

Context - Automation in Bowling Alley Management

The Lucky Strikes Bowling Centre (LSBC) aims automate their chain of bowling centres. The requirements include upgradation of the Pin Setting Equipment and its interactions with the Scoring Station and Local Databases to automatically detect the number of pins knocked by a player and calculation and record of subsequent scores in each play session.

Technical Details about Bowling

Each such bowling centre is a Bowling Alley, each of which consists of a number of Bowling Lanes. One game can be carried out in each Lane at any instant. Each Lane can handle from one to five players, but not more. Each game consists of a number of Frames, for each of which each player is scored according to the respective performance.

Product Features

The product enables the admission of players as an individual or as a party. The admission of players happens at the Control Desk and an appropriate lane assignment takes place. In case a set of players want to join as a party, all of them will be assigned into a single lane. The order in which the players get admitted determines the order in which each player bowls.

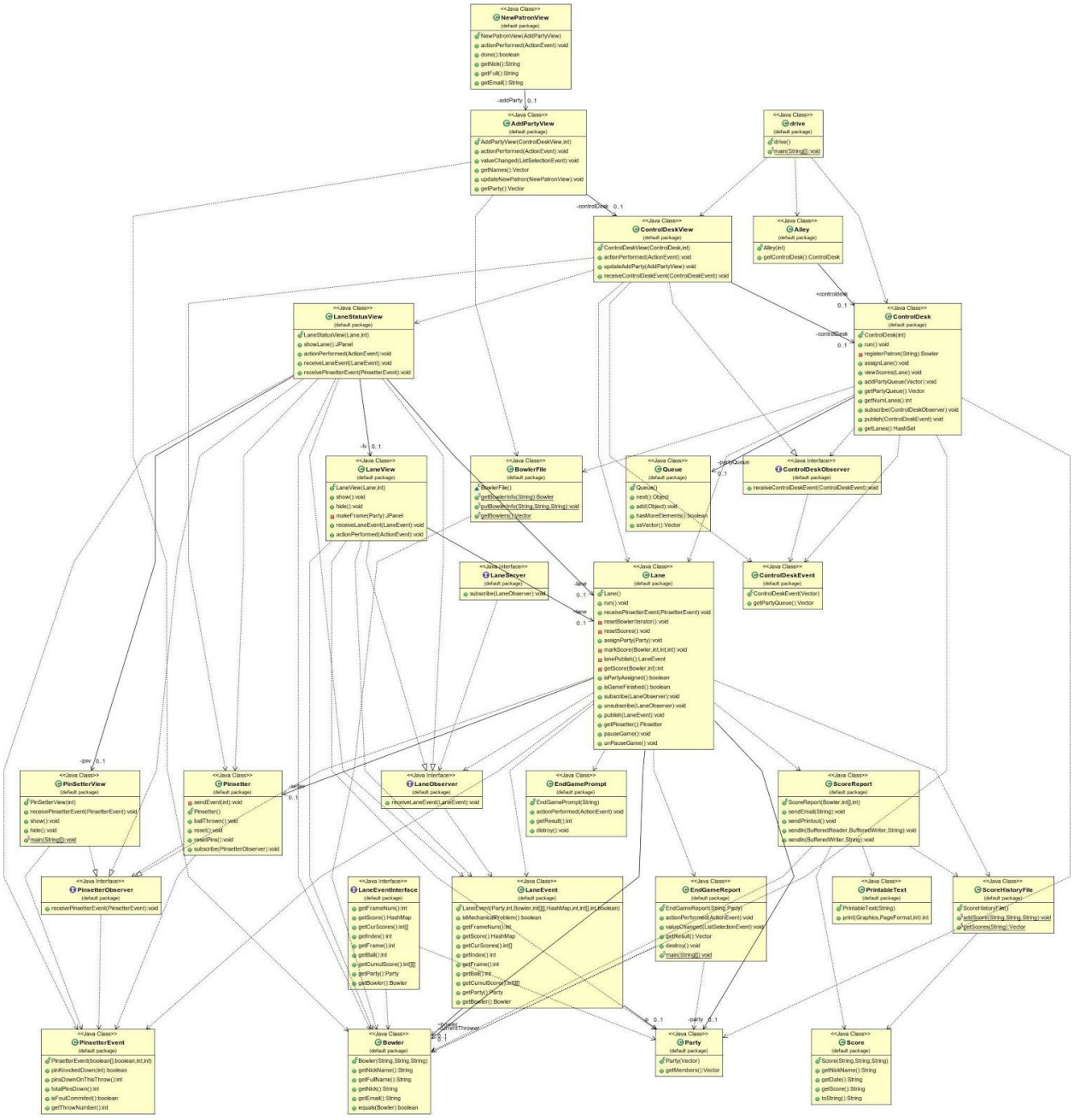
Each Lane houses a unique Scoring Station which keeps track of each players score. The upgraded Pin Setting Equipment detects the missing pins after each Throw automatically and reports the Pins' status to the Scoring Station. The scores are computed at the Scoring Station for each player and the same are reported to the Control Desk once any player completes a game.

The player can view the respective Pin Setter status at the Lane's Scoring Station. Also, the individual scores for each Frame is also maintained at the Scoring Station. Once a game ends, the players can choose to leave or restart another game.

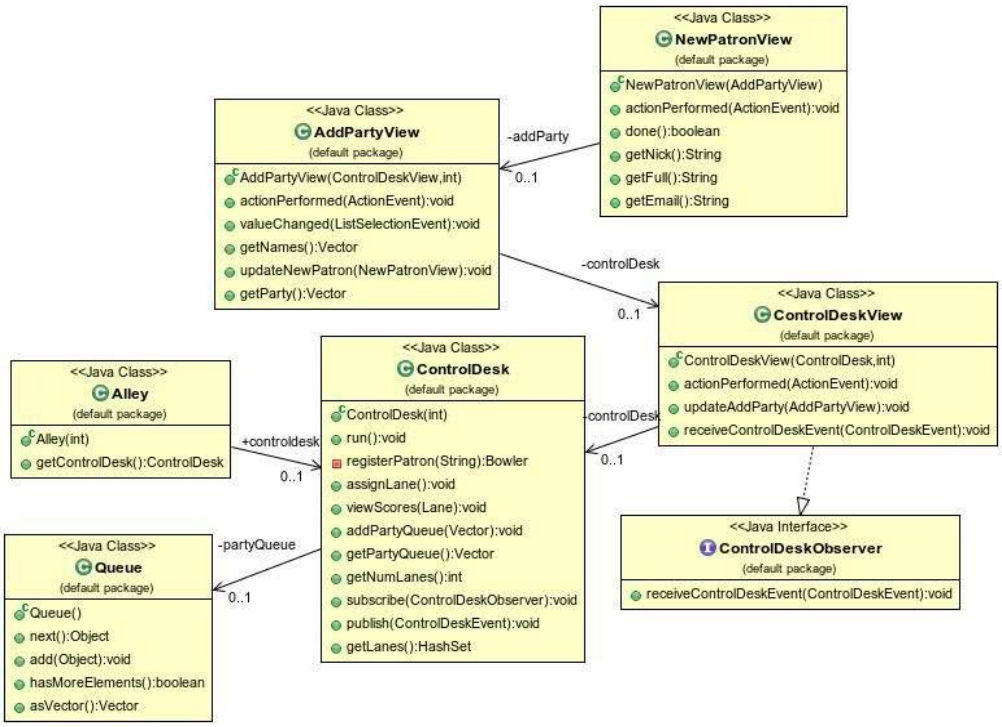
UML Class Diagrams

Original Implementation

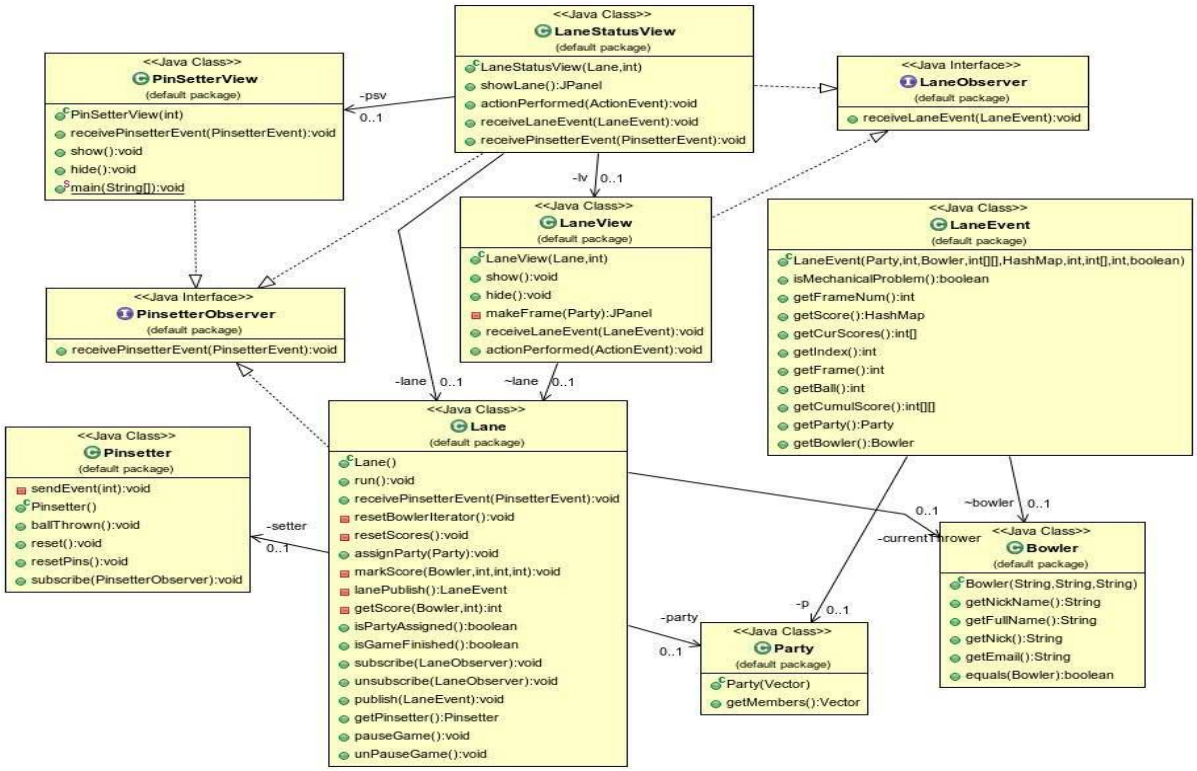
With Depencies



ControlDesk Module

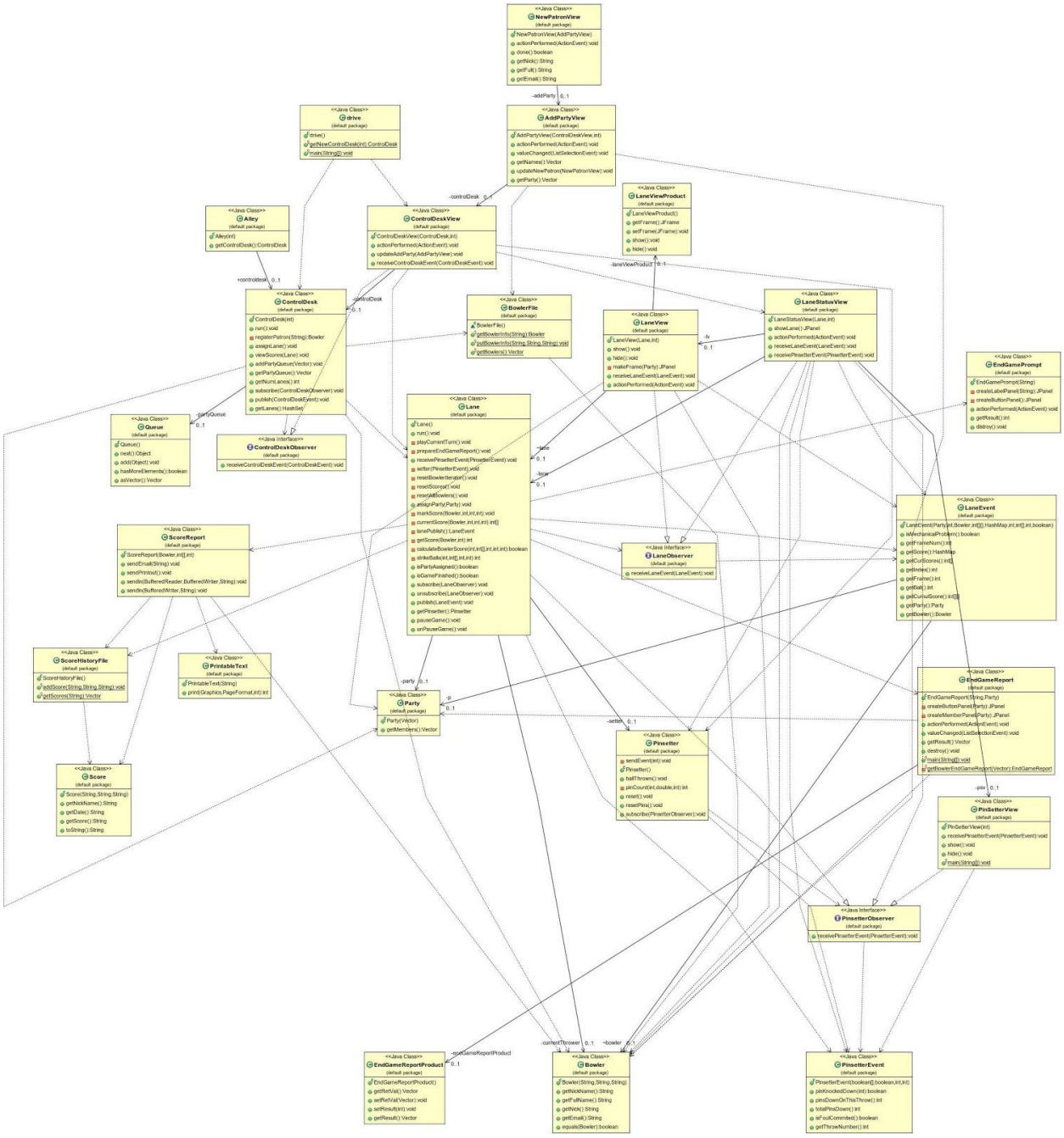


Lane Module

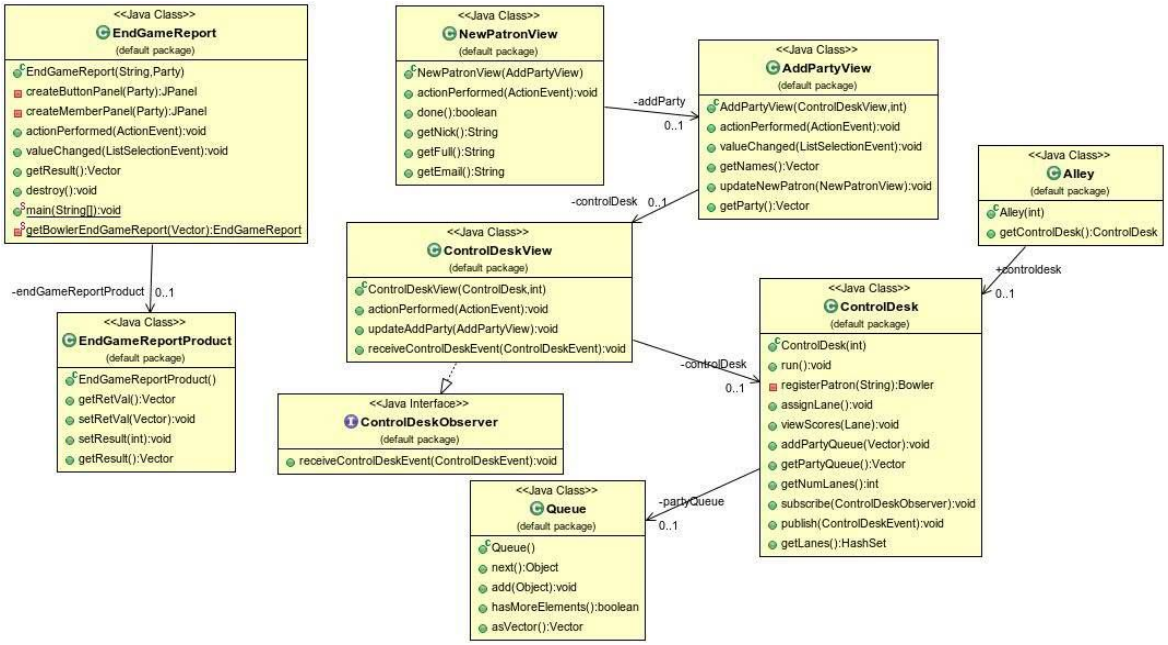


Refactored Implementation

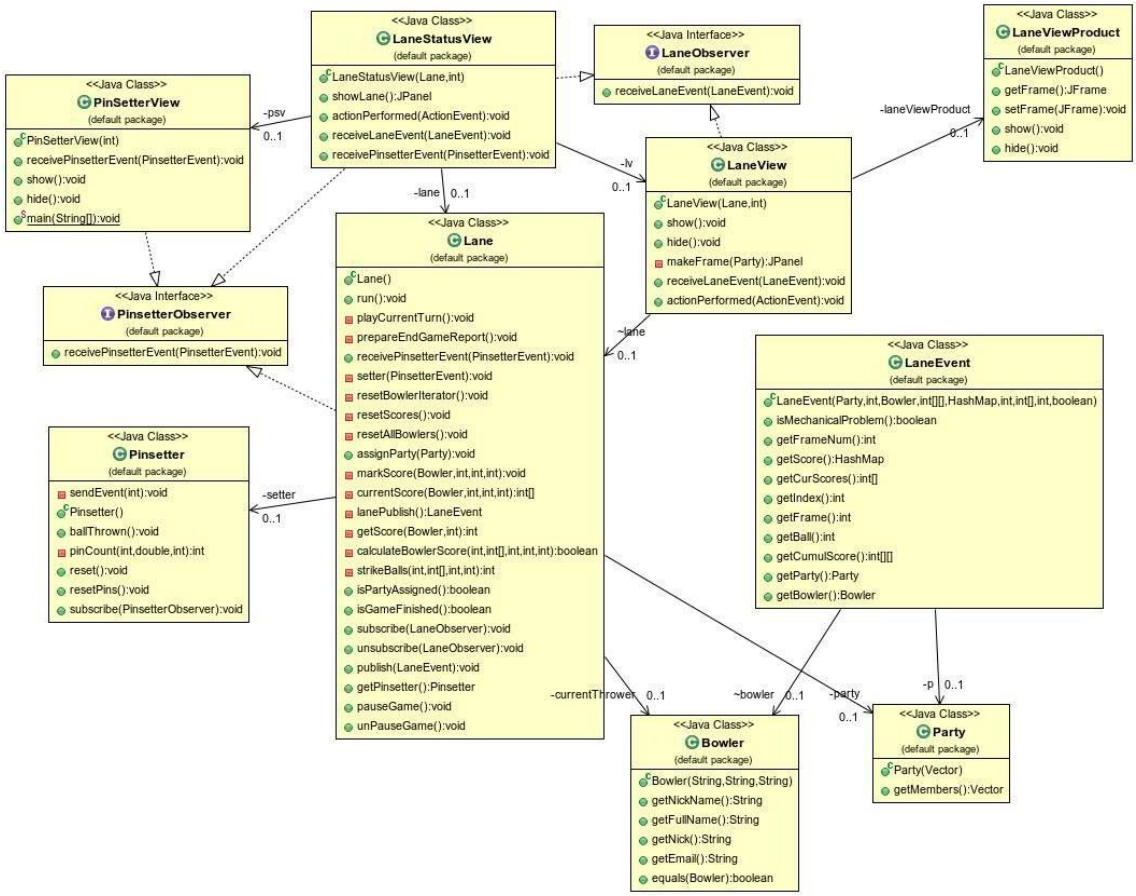
With Dependencies



ControlDesk Module

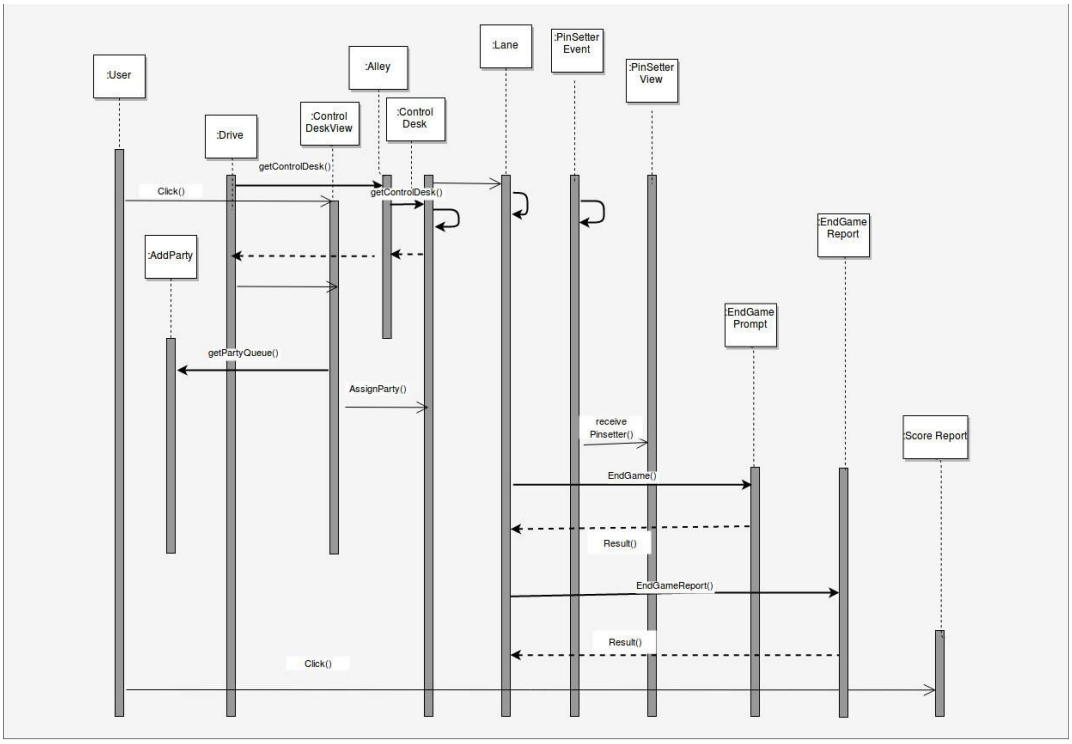


Lane Module

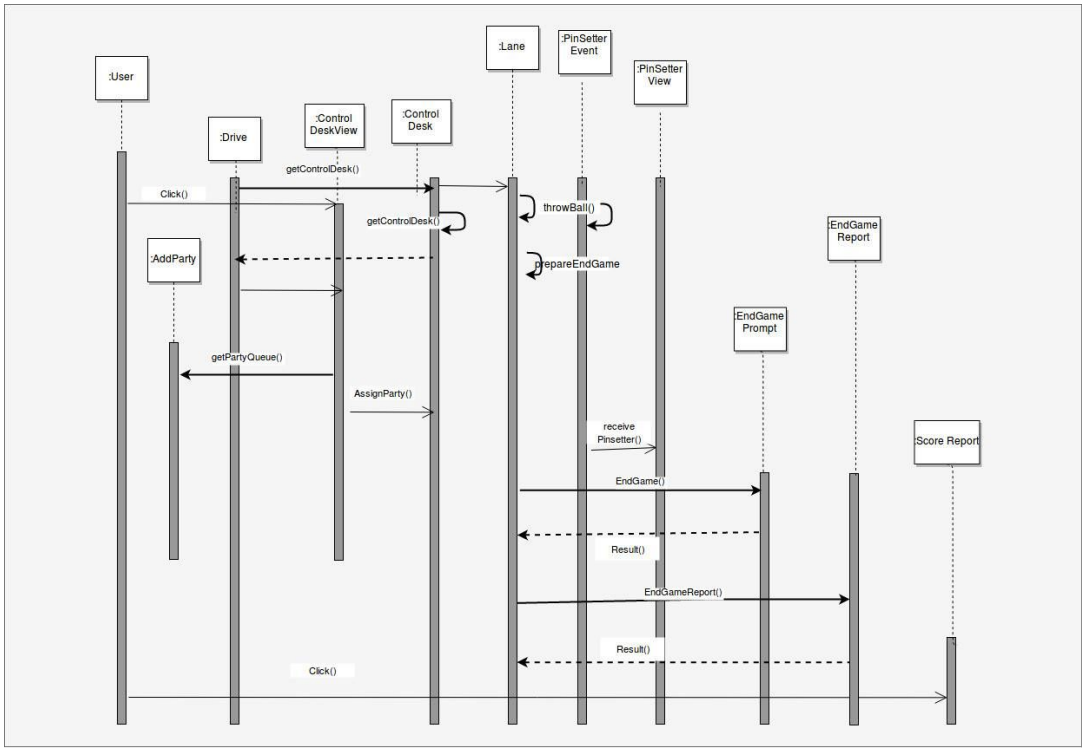


UML Sequence Diagrams

Original Implementation



Refactored Implementation



Class Descriptions

Sl. no.	Class Name	Class Description
1	drive	- Creates the whole bowling alley by creating the respective Control Desk instance.
2	ControlDesk	- Admits a player or a party into the alley. - Adds a party to a waiting queue if no Lane is free. - Assigns Lane to a party appropriately. - Allows objects to be added as subscribers. - Broadcasts event updates to everyone listed as subscriber.
3	Queue	- Creates a new party waiting queue.
4	Lane	- Represents the Lane and its Score Station. - If a game is going on for a party, handle the sequence of throws for the bowlers in the party. - If a game is over, it proceeds to preparation of game report. - Provides choice of starting another game to to the party which finished. - Receives Pin Setter outputs and calculates score after each throw accordingly at the Score Station.
5	PinSetter	- Handles resetting, simulation and detection of pins appropriately during each throw.
6	Bowler	- Maintains each Bowler's information.
7	BowlerFile	- Handles interactions with bowlers' information database.
8	Party	- Maintains the Bowlers' list who belong to the party.
9	ScoreHistoryFile	- Handles interactions with stored database.
10	All Event Classes	- Represents respective events related to respective classes.
11	All View Classes	- Provides GUI representations of respective classes.
12	All Observer Classes	- Creates an instance for each subscriber of respective observable components.

Analysis on Original Design

We began by understanding the original codebase and as a result of the same, we got to analyse the codebase quite well.

First of all we noticed that, the original codebase had a couple of unnecessarily long and complicated classes including the LaneView Class and the GameReport Class. It was clear in these modules that these could be modularised further into smaller classes. We observed *Low Cohesion* in these classes.

We also found a couple of dead modules or dead classes in the codebase. This includes the LaneServer module and the LaneViewInterface module. We realized that the LaneServer is not quite required in the codebase. However, the LaneViewInterface could very well be used in the codebase.

One more factor which made the code seem unnecessarily complex was its *Cyclomatic Complexity*, which arises due to there being too many conditions on the same parameter. We found out that many of these conditional statements were actually unnecessary. Moreover, in a few cases, cases of unnecessary nesting of conditional statements were observed.

Since the original codebase's Design Document gave very little implementation level information, it may not be fair to compare the satisfied functionalities of the original codebase with the deliverables as defined by the Design Document. However, the original codebase does have implementation of a couple of design patterns like *Observer Pattern* and *Adapter Pattern*.

The *Observer Pattern* can be observed in the form of a subscription based setup where various event based statuses are broadcast to all objects subscribing for the same. Moreover, *Adapter Pattern* can be observed during interactions with the file database through the PrintableText Class module. These comprise a few of the strengths of the original codebase.

Analysis on Refactored Design

Our approach to refactoring the codebase was basically a second walkthrough through the whole codebase where, unlike the first time, we began taking the refactoring steps in the codebase.

We found that the LaneView Class and the GameReport Class were unnecessarily complicated. To tackle the same issue, we picked up a few interdependent parameters and methods from the above classes and created new classes LaneViewProduct Class and GameReportProduct Class. The same changes helped us deal with *Long Class* code-smell we also ended up obtaining *Higher Cohesion* in the codebase through the above steps.

The LaneServer Class, which we found was a dead module which is not quite required, was removed. Another dead module we found was the LaneViewInterface Java Interface which we coupled with the LaneView Class. Thus, we revived the same dead module.

As a consequence to tackling *Cyclomatic Complexity*, we observed and removed all the unnecessary code snippets which brought about unnecessary conditional statements. One such modules which contained a few instances of such snippets was Lane Class Module. Moreover, the unnecessary instances if nested if conditions were simplified into single if statements with appropriate fusion of conditions using the right logical operations. This also helped reduce the *Long Method* code-smell to quite some extent.

Since the system is a full stack application. Therefore for every event in pinsetter or for every ball thrown there are two things which needs to be updated first one is the the view or frontend, which needs to show the current state of the pindrop and next is the backend, where the Lane should update it's score changed by the current pindrop through the Score Station.

So since the requirement of system here is to notify some classes when some event is triggered in the pinsetter class. That's why *Observer Pattern* is used here. Because of the *Observer Pattern* the pinsetter is not tightly coupled with any of the class which is supposed to receive the event update and this allows any class to become a listener or observer. It just needs to implement the Observable Class' interface. This pattern also helps in *Extension* of codebase, whenever more listeners are need to be added they can easily be plugged in to the system simply by implementing the interface. Moreover, just as in the original codebase, *Adapter Pattern* is also present in the refactored codebase during file database I/O.

One more observation we made on manual analysis of the original codebase was that the Alley class module was just a *Data Class* and was not really a necessary class. The Alley class thus, was removed and replaced with a vector which stored the same information. Hence, we got rid of a class which was acting as an *Unnecessary Middleman* in the code setup.

We also noticed that in many cases, there was *Indecent Exposure* of many data members of a few classes. The same was tackled through *Data Hiding*, wherever required. While doing so, we also reduced *Coupling* as a few otherwise public members which were being taken for granted by other class modules were either refrained from being accessed or accessed through a public member function if necessary.

In this way, we also refined the relationship between various class modules and their respective view, event and in some cases product class modules. This was done in accordance with the *Law of Demeter* and thus further led to *Loose Coupling* in the refactored implementation.

Analysis based on Metrics

Metric values before refactoring for the entire codebase is as follows:

Metric	Total	Mean	Std. Dev.	Maximum
McCabe Cyclomatic Complexity (avg/max per method)		2.319	4.062	38
Number of Parameters (avg/max per method)		0.723	1.131	9
Nested Block Depth (avg/max per method)		1.511	1.177	7
Afferent Coupling (avg/max per packageFragment)		0	0	0
Efferent Coupling (avg/max per packageFragment)		0	0	0
Instability (avg/max per packageFragment)		1	0	1
Abstractness (avg/max per packageFragment)		0.172	0	0.172
Normalized Distance (avg/max per packageFragment)		0.172	0	0.172
Depth of Inheritance Tree (avg/max per type)		0.897	0.48	2
Weighted methods per Class (avg/max per type)	327	11.276	15.991	87
Number of Children (avg/max per type)	6	0.207	0.663	3
Number of Overridden Methods (avg/max per type)	3	0.103	0.305	1
Lack of Cohesion of Methods (avg/max per type)		0.375	0.374	0.91
Number of Attributes (avg/max per type)	138	4.759	5.556	18
Number of Static Attributes (avg/max per type)	2	0.069	0.253	1
Number of Methods (avg/max per type)	133	4.586	3.765	17
Number of Static Methods (avg/max per type)	8	0.276	0.69	3
Specialization Index (avg/max per type)		0.017	0.052	0.2
Number of Classes (avg/max per packageFragment)	29	29	0	29
Number of Interfaces (avg/max per packageFragment)	5	5	0	5
Number of Packages	1			
Total Lines of Code	1814			
Method Lines of Code (avg/max per method)	1284	9.106	17.201	88

Following metrics with the given recommendation has been used to analyze the code base quality:

1. Cyclomatic Complexity:
   Defined for types and methods. Cyclomatic complexity is a popular procedural software metric equal to the number of decisions that can be taken in a procedure.
   Methods where CC is higher than 15 are hard to understand and maintain. Methods where CC is higher than 38 are extremely complex and should be split in smaller methods.

2. Number of parameters:
   Methods where NbVariables is higher than 8 are hard to understand and maintain. Whereas the methods where NbVariables is higher than 15 are extremely complex and should be split in smaller methods.

3. Depth of Inheritance Tree:
   Types where Depth Of Inheritance is higher or equal than 6 might be hard to maintain. However it is not a rule since some time the classes might inherit from third-party classes which have a high value for depth of inheritance. For example, the average depth of inheritance for third-party classes which derive from System.Windows.Forms.Control is 5.3.

4. Number of Children:
   Recommended value is between 1 and 4 :The upper and lower limits of 1 and 3 correspond to a desirable average. This will not stop certain particular classes being the kind of utility classes which provide services to significantly more classes than 3.

5. Number of Attributes:
   Recommended value is from 2 to 5 : A high number of attributes (> 10) probably indicates poor design, notably insufficient decomposition, especially if this is associated with an equally high number of methods. Classes without attributes are particular cases, which are not necessarily anomalies. These can be interface classes, for example, which must be checked.

6. Number of Methods:
   This value needs to remain between 3 and 7 for a class. This would indicate that a class has operations, but not too many. A value greater than 7 may indicate the need for further object-oriented decomposition, or that the class does not have a coherent purpose. A value of 2 or less indicates that this is not truly a class, but merely a data construction.

7. Method Lines of Code:
   Methods where NbLinesOfCode is higher than 20 are hard to understand and maintain. While if it's higher than 40, they are extremely complex and should be split into smaller methods.

8. Lack of Cohesion of Methods:
   Types where LCOM > 0.8 and NbFields > 10 and NbMethods >10 might be problematic. However, it is very hard to avoid such non-cohesive types. Types where LCOMHS > 1.0 and NbFields > 10 and NbMethods >10 should be avoided. This constraint is stronger and thus easier to satisfy than the constraint types where LCOM > 0.8 and NbFields > 10 and NbMethods >10.

The above mentioned metrics are used to identify defects, assess productivity and identify areas of improvement within applications. It is a known fact that less complex code is

easier to maintain. So the measurements are used at the code level to identify defects or overly complex code and make the necessary fixes.

The information gained from these metrics helped to identify if the piece of the code needs to be modified or should be replaced. There are few methods and classes that has some complicated logic which were identified and checked for refactoring as such classes are more error prone.

This also helped prioritize different possible code fixing actions required during refactoring.

Refactoring on the basis of the metrics require a trade-off to be maintained between various values. Fixing one metric may result in shooting the value for a different parameter. Following result is an evidence for the same.

Metrics comparison before and after the refactoring:

1) *drive.java:*

Before:

Metrics - drive.java - Number of Methods (avg/max per type)

Metric	Total	Mean	Std. Dev.	Maximum
McCabe Cyclomatic Complexity (avg/max per method)		1	0	1
Number of Parameters (avg/max per method)		1	0	1
Nested Block Depth (avg/max per method)		1	0	1
Depth of Inheritance Tree (avg/max per type)		1	0	1
Weighted methods per Class (avg/max per type)	2	2	0	2
Number of Children (avg/max per type)	0	0	0	0
Number of Overridden Methods (avg/max per type)	0	0	0	0
Lack of Cohesion of Methods (avg/max per type)		0	0	0
Number of Attributes (avg/max per type)	0	0	0	0
Number of Static Attributes (avg/max per type)	0	0	0	0
Number of Methods (avg/max per type)	0	0	0	0
Number of Static Methods (avg/max per type)	2	2	0	2
Specialization Index (avg/max per type)		0	0	0
Number of Classes	1			
Number of Interfaces	0			
Total Lines of Code	13			
Method Lines of Code (avg/max per method)	7	3.5	1.5	5

Number of children is applicable for an interface which is the number of types that implement it. Hence, this metric is not affecting any of the interface classes in the original code.

2) *Lane.java:*

Before:

Metric	Total	Mean	Std. Dev.	Maximum
McCabe Cyclomatic Complexity (avg/max per method)		5.118	9.498	38
Number of Parameters (avg/max per method)		0.647	1.026	4
Nested Block Depth (avg/max per method)		2.176	2.065	7
Depth of Inheritance Tree (avg/max per type)		2	0	2
Weighted methods per Class (avg/max per type)	87	87	0	87
Number of Children (avg/max per type)	0	0	0	0
Number of Overridden Methods (avg/max per type)	1	1	0	1
Lack of Cohesion of Methods (avg/max per type)		0.851	0	0.851
Number of Attributes (avg/max per type)	18	18	0	18
Number of Static Attributes (avg/max per type)	0	0	0	0
Number of Methods (avg/max per type)	17	17	0	17
Number of Static Methods (avg/max per type)	0	0	0	0
Specialization Index (avg/max per type)		0.118	0	0.118
Number of Classes	1			
Number of Interfaces	0			
Total Lines of Code	294			
Method Lines of Code (avg/max per method)	236	13.882	24.98	88

After:

Metric	Total	Mean	Std. Dev.	Maximum
McCabe Cyclomatic Complexity (avg/max per method)		4.292	6.592	32
Number of Parameters (avg/max per method)		1.042	1.541	5
Nested Block Depth (avg/max per method)		2.25	1.614	6
Depth of Inheritance Tree (avg/max per type)		2	0	2
Weighted methods per Class (avg/max per type)	103	103	0	103
Number of Children (avg/max per type)	0	0	0	0
Number of Overridden Methods (avg/max per type)	1	1	0	1
Lack of Cohesion of Methods (avg/max per type)		0.866	0	0.866
Number of Attributes (avg/max per type)	18	18	0	18
Number of Static Attributes (avg/max per type)	0	0	0	0
Number of Methods (avg/max per type)	24	24	0	24
Number of Static Methods (avg/max per type)	0	0	0	0
Specialization Index (avg/max per type)		0.083	0	0.083
Number of Classes	1			
Number of Interfaces	0			
Total Lines of Code	318			
Method Lines of Code (avg/max per method)	246	10.25	14.449	68

For the lane class, the number of methods parameter was increases as it needed a trade-off with the other metric values like number of decision making trees to reduce the Cyclomatic complexity, method lines of code which we tried to reduce.

3) *Pinsetter.java*

Before:

Metric	Total	Mean	Std. Dev.	Maximum
McCabe Cyclomatic Complexity (avg/max per method)		2.5	2.062	7
Number of Parameters (avg/max per method)		0.333	0.471	1
Nested Block Depth (avg/max per method)		2	1	4
Depth of Inheritance Tree (avg/max per type)		1	0	1
Weighted methods per Class (avg/max per type)	15	15	0	15
Number of Children (avg/max per type)	0	0	0	0
Number of Overridden Methods (avg/max per type)	0	0	0	0
Lack of Cohesion of Methods (avg/max per type)		0.56	0	0.56
Number of Attributes (avg/max per type)	5	5	0	5
Number of Static Attributes (avg/max per type)	0	0	0	0
Number of Methods (avg/max per type)	6	6	0	6
Number of Static Methods (avg/max per type)	0	0	0	0
Specialization Index (avg/max per type)		0	0	0
Number of Classes	1			
Number of Interfaces	0			
Total Lines of Code	63			
Method Lines of Code (avg/max per method)	42	7	6.952	22

After:

Metric	Total	Mean	Std. Dev.	Maximum
McCabe Cyclomatic Complexity (avg/max per method)		2.286	1.278	5
Number of Parameters (avg/max per method)		0.714	1.03	3
Nested Block Depth (avg/max per method)		2	0.926	4
Depth of Inheritance Tree (avg/max per type)		1	0	1
Weighted methods per Class (avg/max per type)	16	16	0	16
Number of Children (avg/max per type)	0	0	0	0
Number of Overridden Methods (avg/max per type)	0	0	0	0
Lack of Cohesion of Methods (avg/max per type)		0.567	0	0.567
Number of Attributes (avg/max per type)	5	5	0	5
Number of Static Attributes (avg/max per type)	0	0	0	0
Number of Methods (avg/max per type)	7	7	0	7
Number of Static Methods (avg/max per type)	0	0	0	0
Specialization Index (avg/max per type)		0	0	0
Number of Classes	1			
Number of Interfaces	0			
Total Lines of Code	67			
Method Lines of Code (avg/max per method)	44	6.286	4.832	17

4) *EndGameReport.java*

Before:

Metric	Total	Mean	Std. Dev.	Maximum
McCabe Cyclomatic Complexity (avg/max per method)		2	0.816	3
Number of Parameters (avg/max per method)		0.833	0.687	2
Nested Block Depth (avg/max per method)		1.833	0.687	3
Depth of Inheritance Tree (avg/max per type)		1	0	1
Weighted methods per Class (avg/max per type)	12	12	0	12
Number of Children (avg/max per type)	0	0	0	0
Number of Overridden Methods (avg/max per type)	0	0	0	0
Lack of Cohesion of Methods (avg/max per type)		0.719	0	0.719
Number of Attributes (avg/max per type)	8	8	0	8
Number of Static Attributes (avg/max per type)	0	0	0	0
Number of Methods (avg/max per type)	5	5	0	5
Number of Static Methods (avg/max per type)	1	1	0	1
Specialization Index (avg/max per type)		0	0	0
Number of Classes	1			
Number of Interfaces	0			
Total Lines of Code	99			
Method Lines of Code (avg/max per method)	71	11.833	15.507	46

After:

Metric	Total	Mean	Std. Dev.	Maximum
McCabe Cyclomatic Complexity (avg/max per method)		1.444	0.685	3
Number of Parameters (avg/max per method)		0.889	0.567	2
Nested Block Depth (avg/max per method)		1.333	0.471	2
Depth of Inheritance Tree (avg/max per type)		1	0	1
Weighted methods per Class (avg/max per type)	13	13	0	13
Number of Children (avg/max per type)	0	0	0	0
Number of Overridden Methods (avg/max per type)	0	0	0	0
Lack of Cohesion of Methods (avg/max per type)		0.833	0	0.833
Number of Attributes (avg/max per type)	7	7	0	7
Number of Static Attributes (avg/max per type)	0	0	0	0
Number of Methods (avg/max per type)	7	7	0	7
Number of Static Methods (avg/max per type)	2	2	0	2
Specialization Index (avg/max per type)		0	0	0
Number of Classes	1			
Number of Interfaces	0			
Total Lines of Code	103			
Method Lines of Code (avg/max per method)	70	7.778	6.562	18

As per the standards, the LCOM ~ 0.8 and No of attributes and No of Methods less than 10 which is close enough to the recommended range.

5) *EndGamePrompt.java*

Before:

Metric	Total	Mean	Std. Dev.	Maximum
McCabe Cyclomatic Complexity (avg/max per method)		2	1	3
Number of Parameters (avg/max per method)		0.5	0.5	1
Nested Block Depth (avg/max per method)		1.75	0.829	3
Depth of Inheritance Tree (avg/max per type)		1	0	1
Weighted methods per Class (avg/max per type)	8	8	0	8
Number of Children (avg/max per type)	0	0	0	0
Number of Overridden Methods (avg/max per type)	0	0	0	0
Lack of Cohesion of Methods (avg/max per type)		0.833	0	0.833
Number of Attributes (avg/max per type)	6	6	0	6
Number of Static Attributes (avg/max per type)	0	0	0	0
Number of Methods (avg/max per type)	4	4	0	4
Number of Static Methods (avg/max per type)	0	0	0	0
Specialization Index (avg/max per type)		0	0	0
Number of Classes	1			
Number of Interfaces	0			
Total Lines of Code	71			
Method Lines of Code (avg/max per method)	50	12.5	13.238	35

After:

Metric	Total	Mean	Std. Dev.	Maximum
McCabe Cyclomatic Complexity (avg/max per method)		1.667	0.943	3
Number of Parameters (avg/max per method)		0.5	0.5	1
Nested Block Depth (avg/max per method)		1.5	0.764	3
Depth of Inheritance Tree (avg/max per type)		1	0	1
Weighted methods per Class (avg/max per type)	10	10	0	10
Number of Children (avg/max per type)	0	0	0	0
Number of Overridden Methods (avg/max per type)	0	0	0	0
Lack of Cohesion of Methods (avg/max per type)		0.875	0	0.875
Number of Attributes (avg/max per type)	6	6	0	6
Number of Static Attributes (avg/max per type)	0	0	0	0
Number of Methods (avg/max per type)	6	6	0	6
Number of Static Methods (avg/max per type)	0	0	0	0
Specialization Index (avg/max per type)		0	0	0
Number of Classes	1			
Number of Interfaces	0			
Total Lines of Code	79			
Method Lines of Code (avg/max per method)	54	9	5.715	17