

Design Document

Unit 1 Project

Bowling Alley Simulation Refactoring Project

TEAM 26

Date of Submission:-20th February,2022

TEAM INFORMATION:-

Name (Roll)	Work Hours	Responsibility
Husen Kagdi (2021201077)	25	Refactoring and Code Smells
Subhra Chakravorty (2021201078)	25	Design Documentation and diagrams
Akilesh Panicker (2021201081)	25	Metric Analysis and Code Cleaning
Siddharth Gupta (2021201082)	25	Refactoring and Design Patterns

OVERVIEW:-

The original code is a simulation of a bowling game where the user is the person who is going to bowl. There is an admin of the bowling game called ControlDesk who can monitor and make changes to the functionalities of any other major classes. Here each ball throw of the user is simulated and the score for each party and lane is calculated and sent back to the user after he decides to stop playing.

The original code was fully functioning but it had some problems like redundant code, code repetitions and other code smells. These problems, if not solved, may become a problem for any future developers who want to add any new functionalities or make some changes in the code design. Hence we have refactored the code and analysed the metrics of the original code and refactored code. Our refactored code attempts to reduce the load of major classes or functions by dividing them into multiple classes or modules. For example:- The functions performed by Pinsetter and Lane class were divided into new classes called SimulateThrow and ScoreCalculate which simulated a ball throw and calculated the score of a throw respectively.

UML Class Diagrams:-

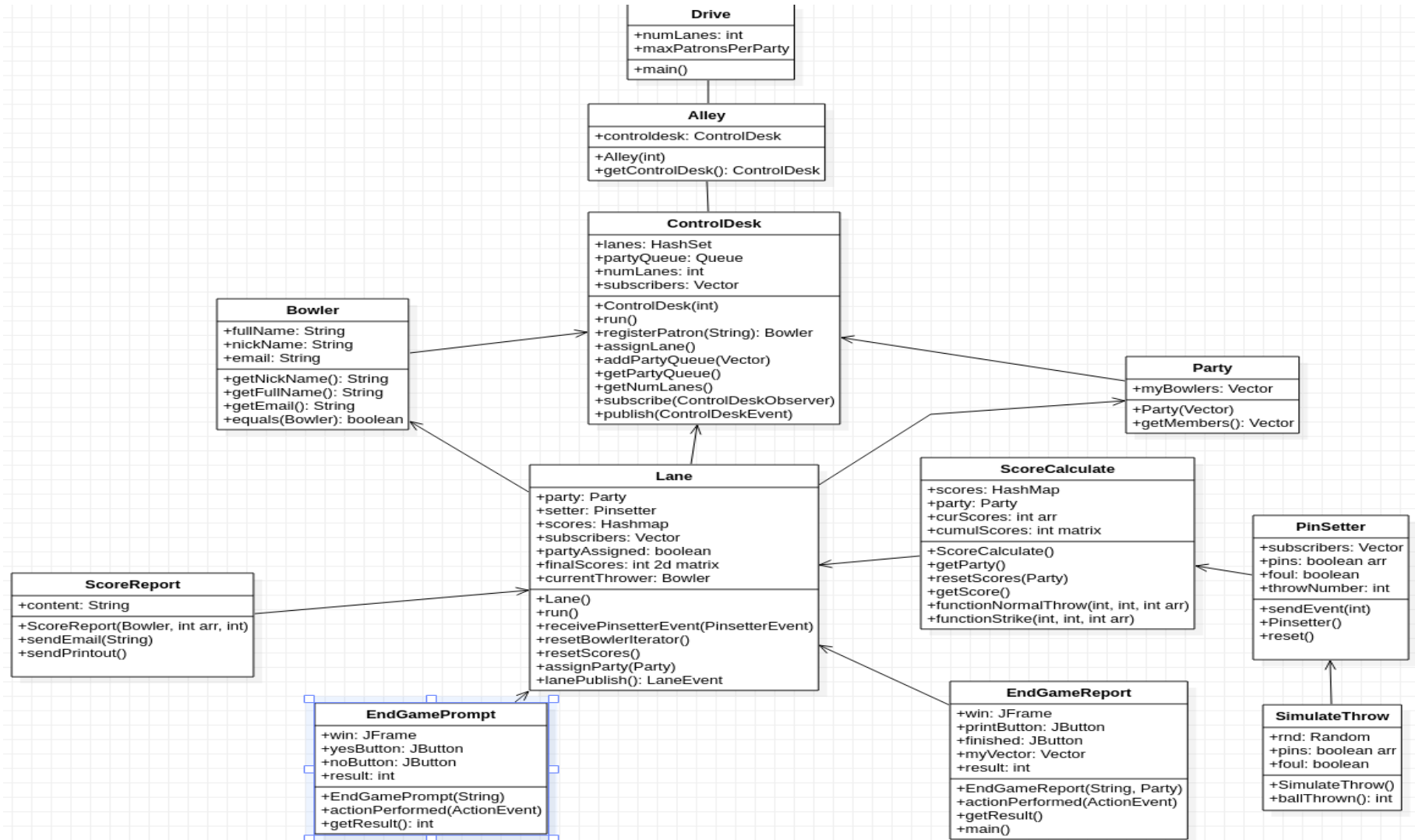


Diagram 1:-UML class diagram of major classes

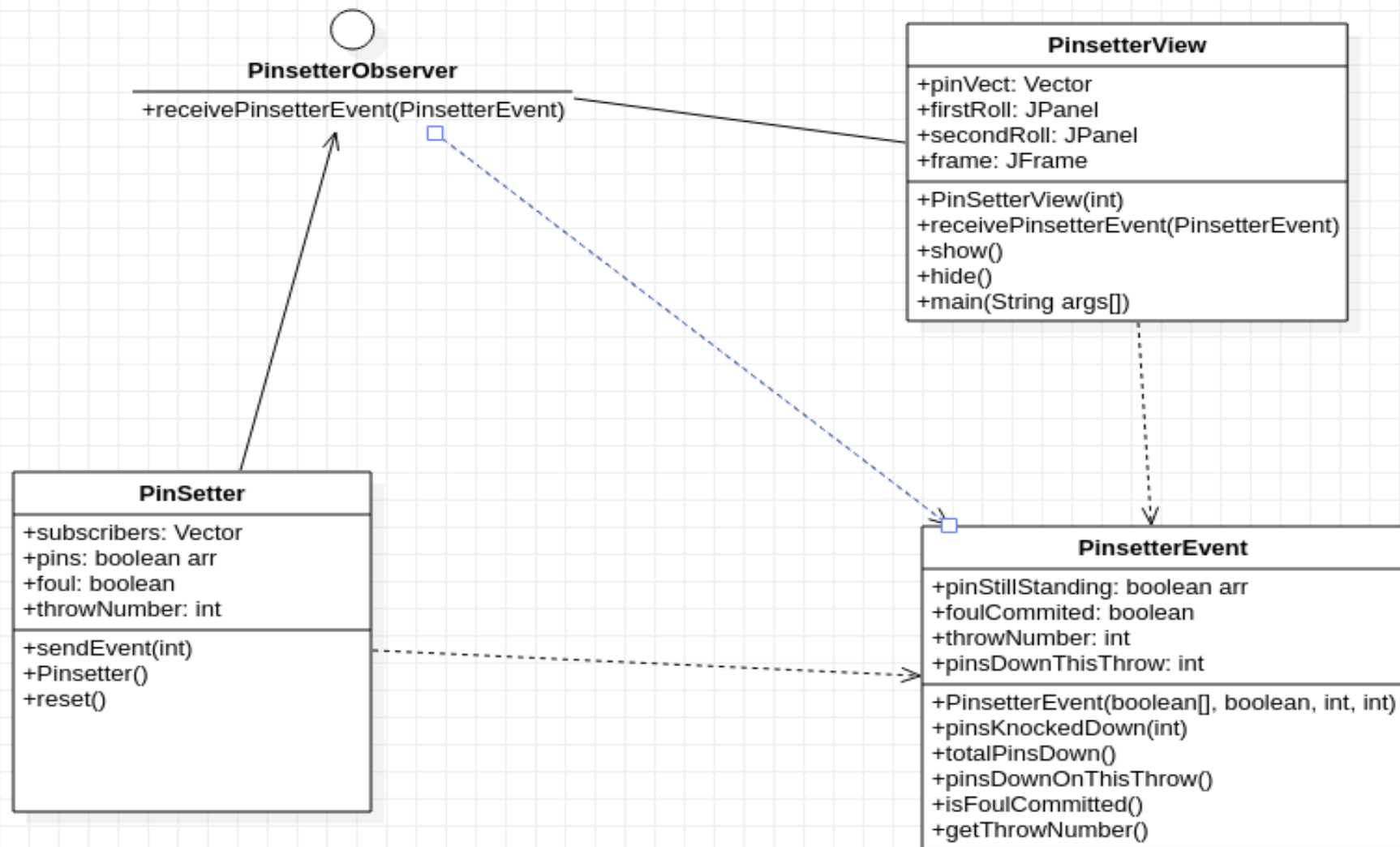


Diagram 2: Pinsetter Class and its design

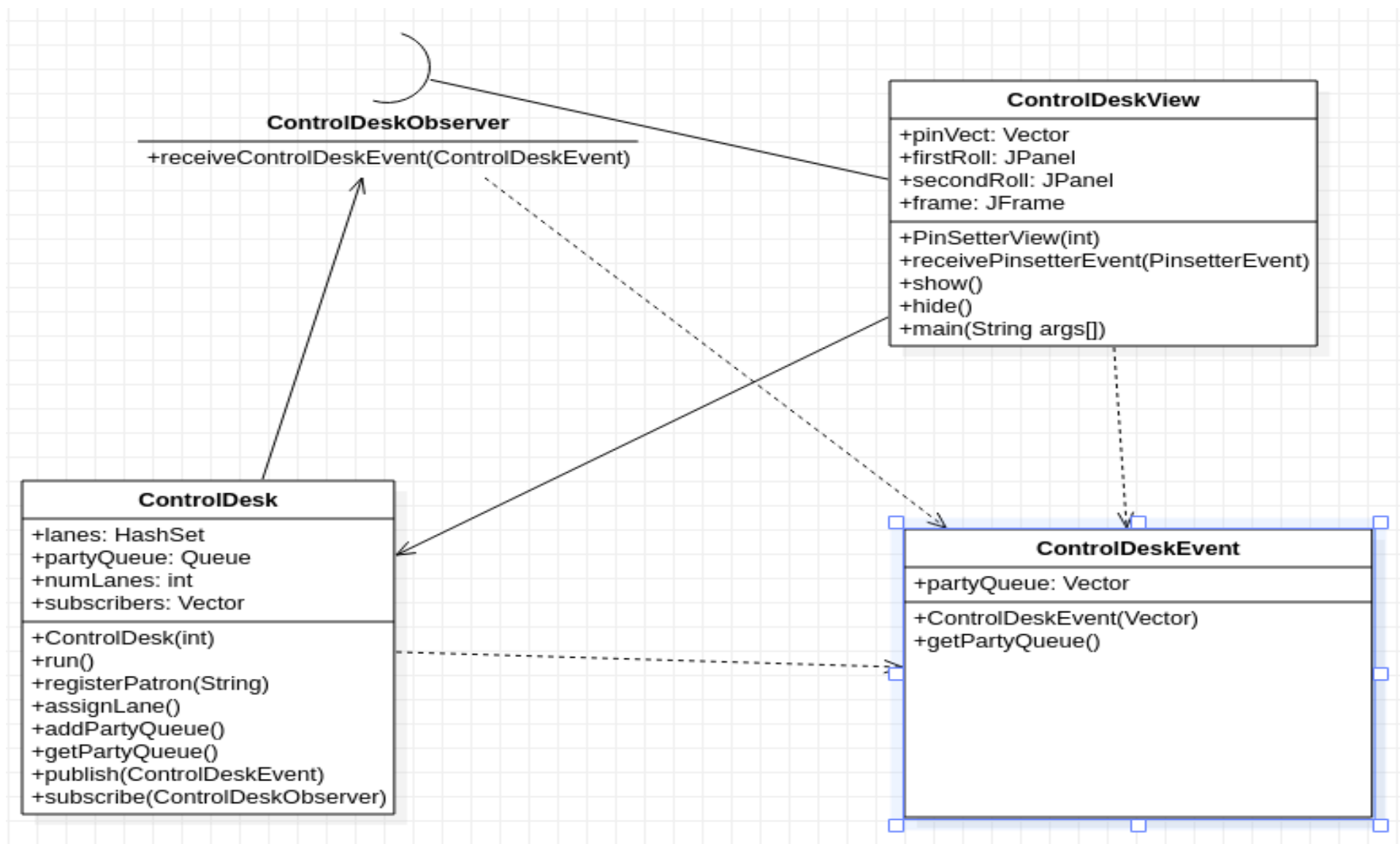


Diagram 3: Control Desk and its design

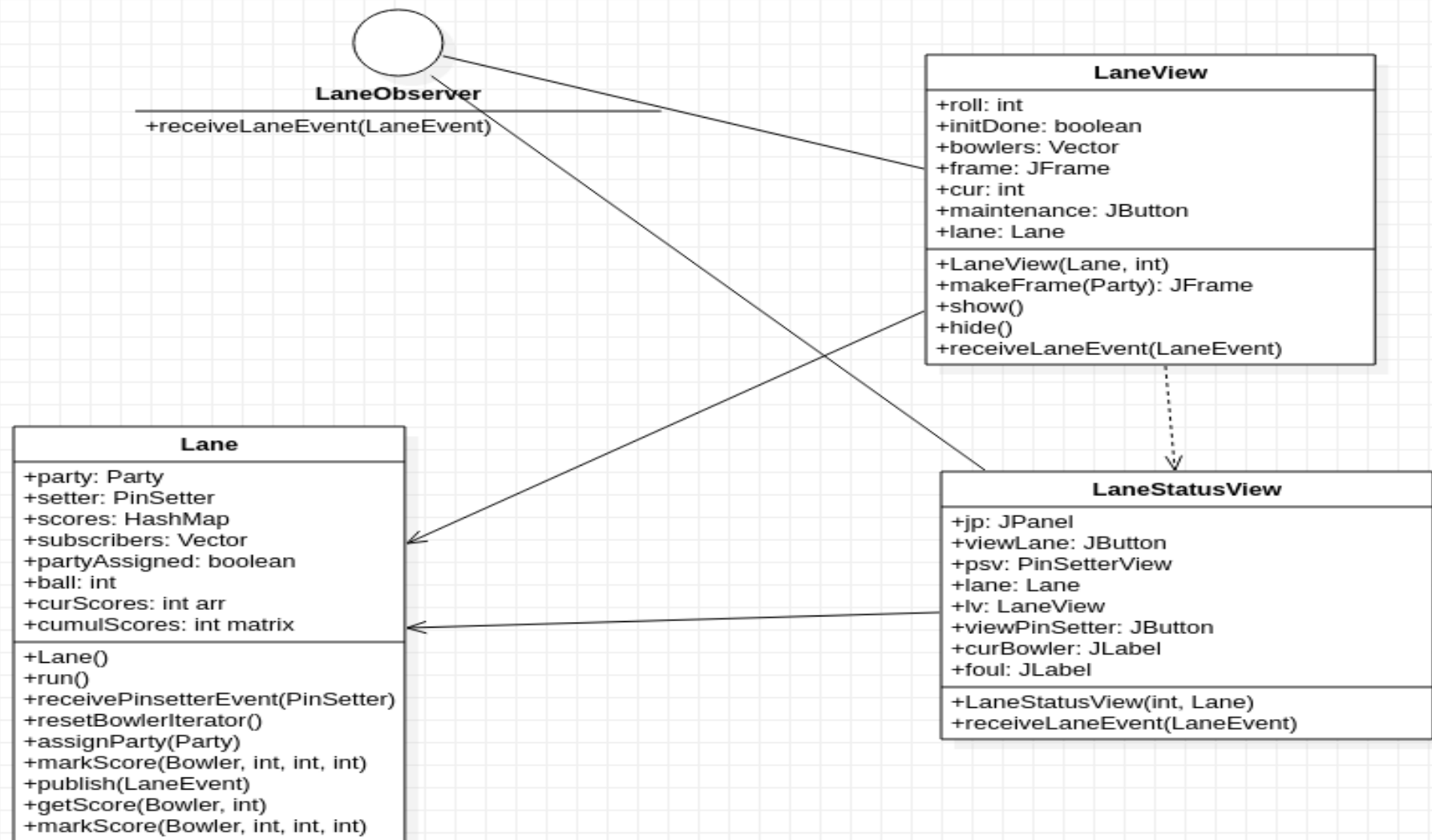


Diagram 4: Lane class and its design

Sequence Diagrams before Refactoring:-

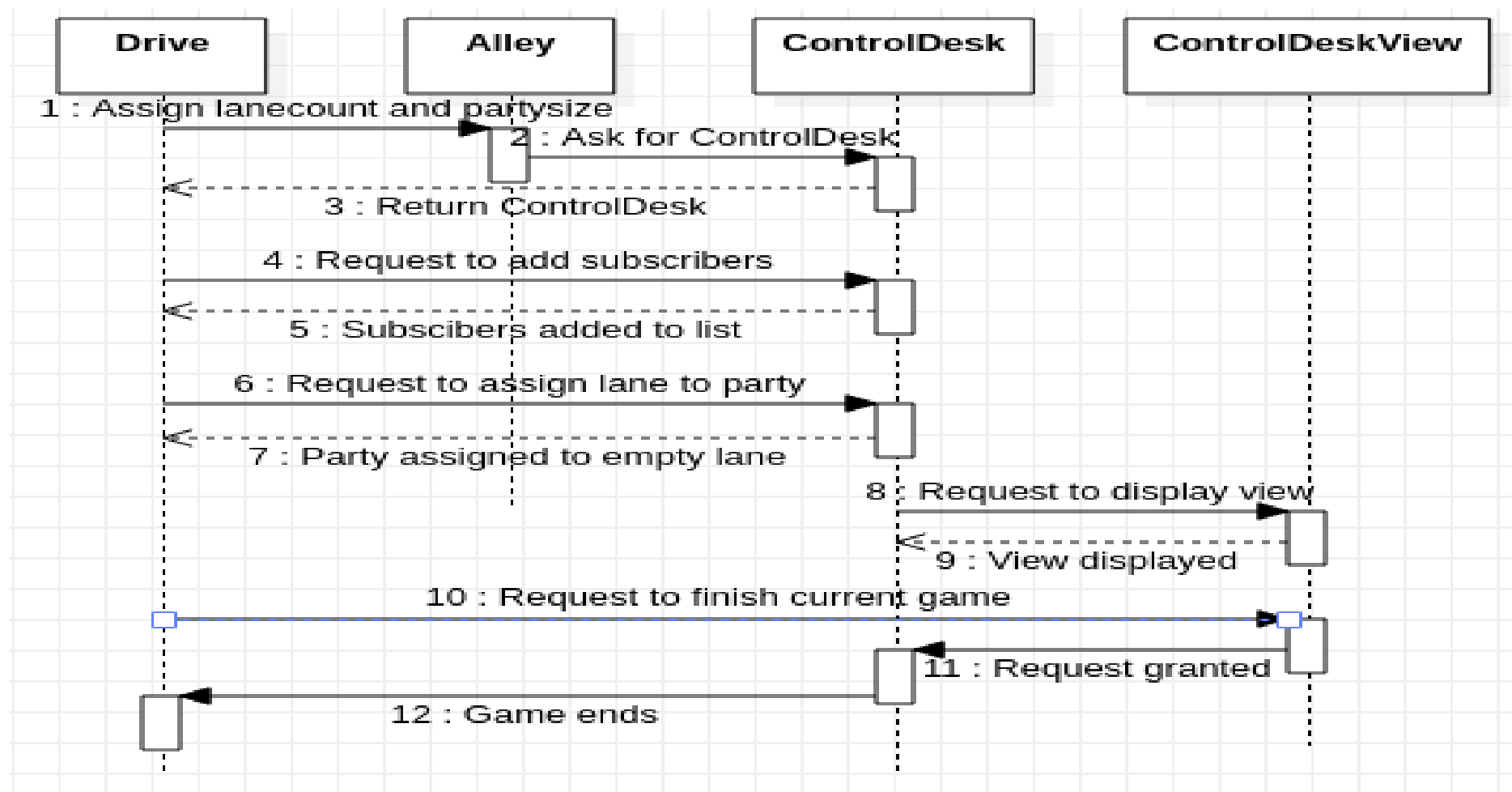


Diagram 5: Assigning lane count, party size and empty lanes to party

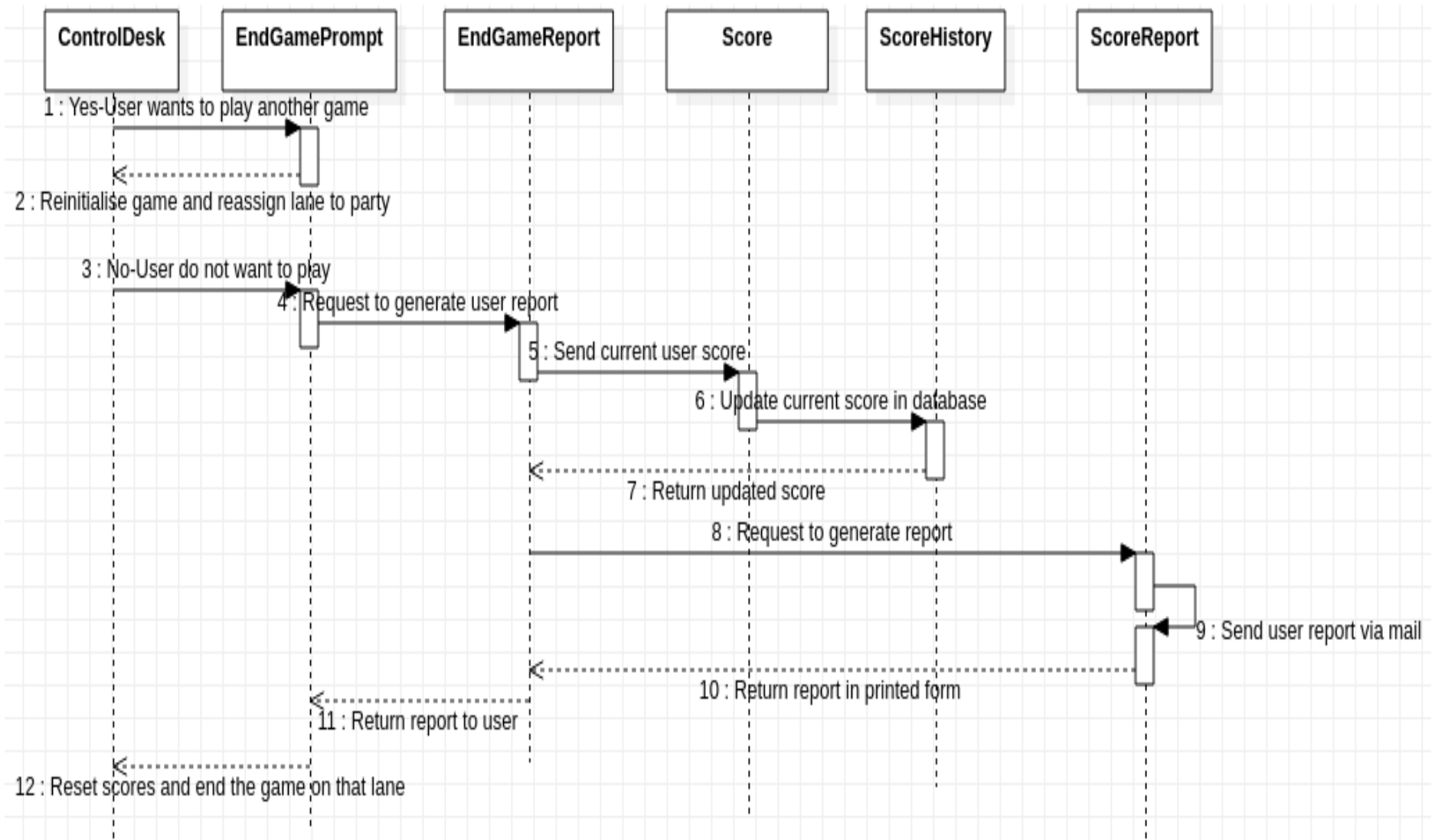


Diagram 6: Endgame prompt,score updation and report generation

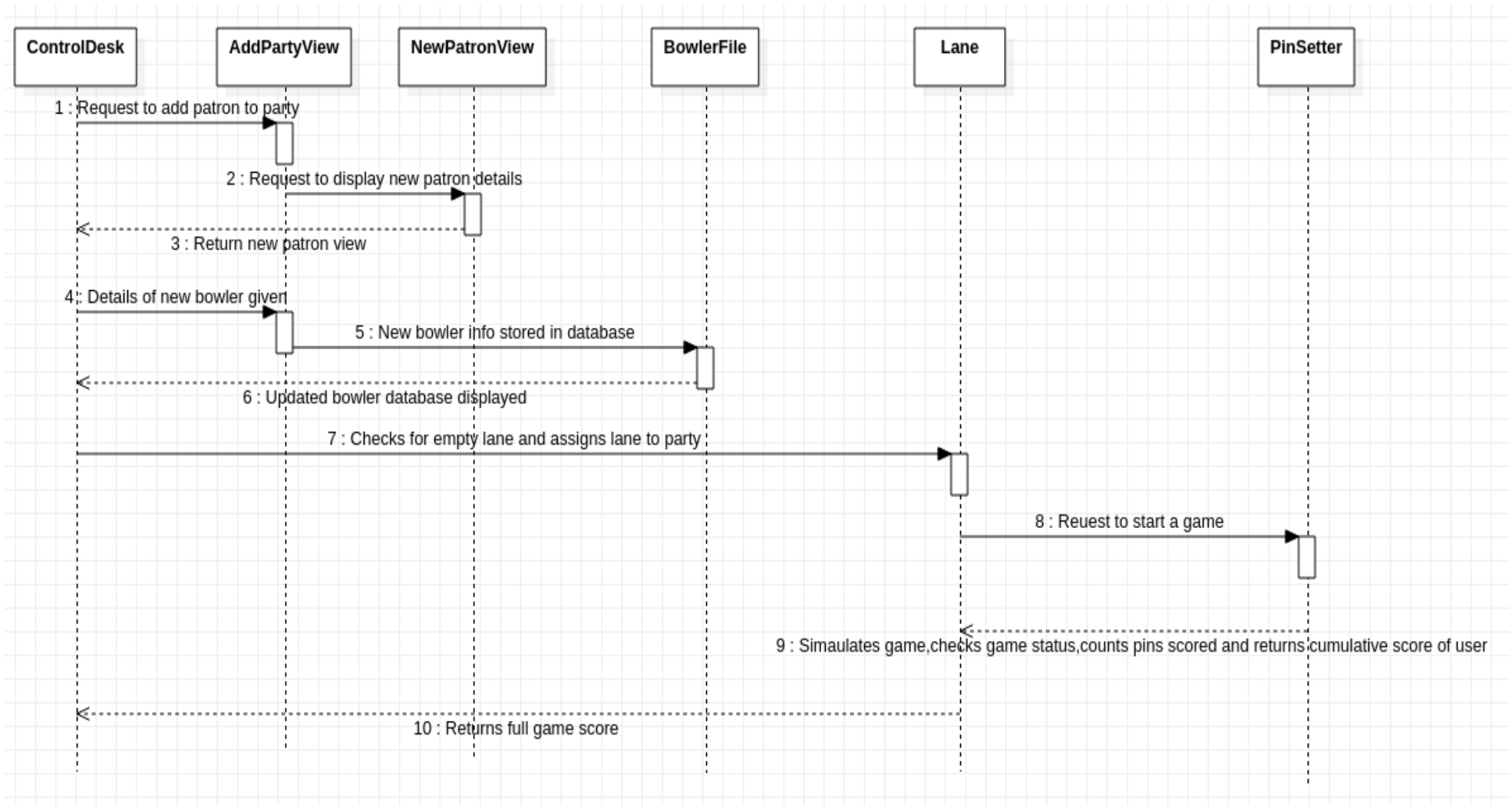


Diagram 7: Add new user,start game and calculate score

Sequence Diagrams after Refactoring:-

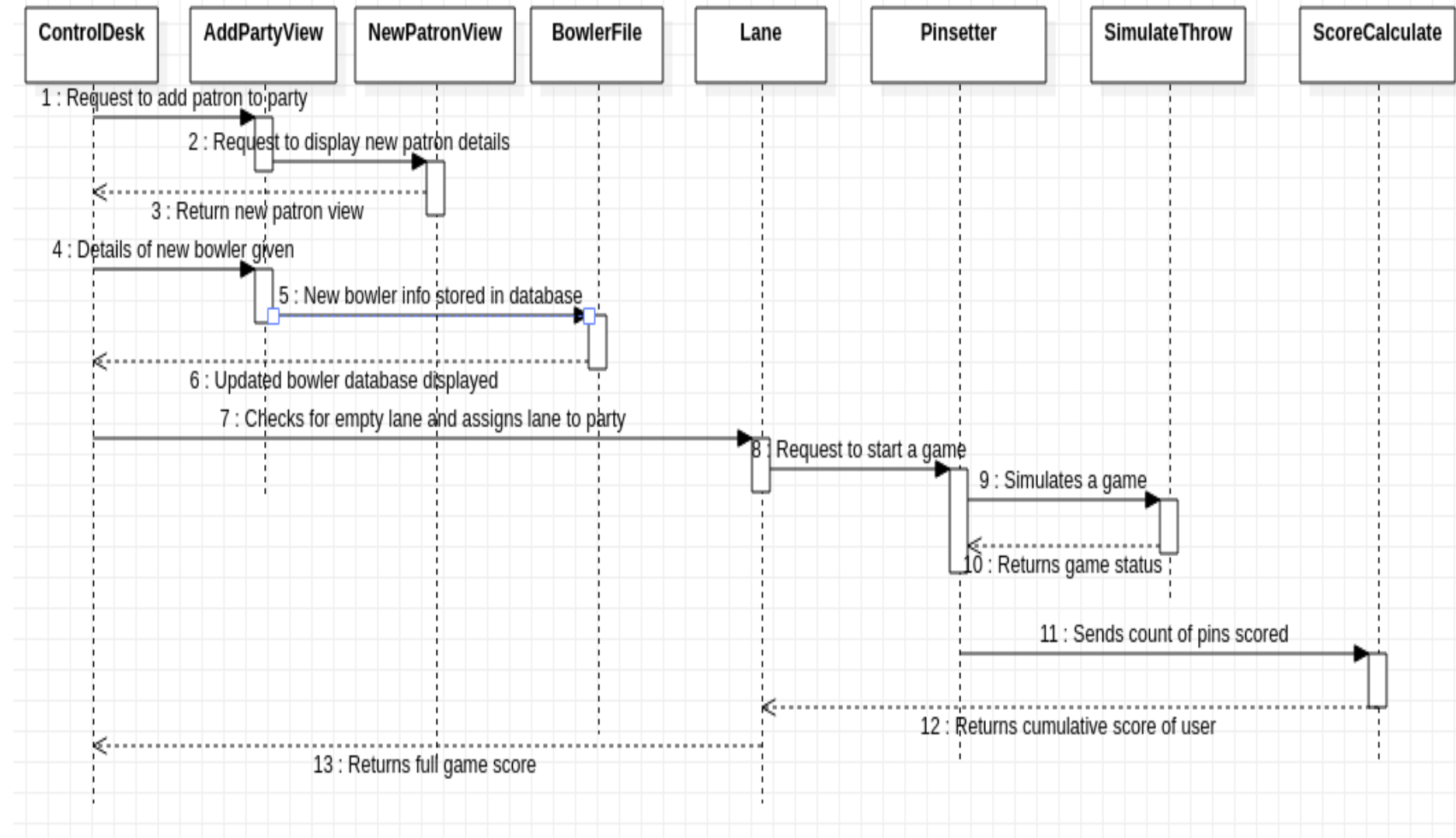


Diagram 8: Add new user,start game and calculate score

Responsibilities of classes:-

Class Name	Responsibility
AddPartyView	GUI to add a party
Bowler	Saves all information of a bowler
BowlerFile	Class to store and receive data from bowler database
ControlDesk	Acts like an admin of all the bowling alleys
ControlDeskView	GUI of ControlDesk
Drive	Start simulation of entire program after assigning lanecount and maxpartysize
Lane	Assigns any empty lane to a party and start the game simulation in a lane
LaneStatusView	GUI to display the status of a lane
Party	Saves information of a party of bowlers
PinSetter	Check status of a throw,calculate score and return it to lane
PinSetterView	GUI to check status after a throw
GeneralView	Contains code to add a button and panel to remove redundancy from all view classes
SimulateThrow	Simulates a single ball throw and returns result to PinSetter
ScoreCalculate	Calculate cumulative score after a throw
EndGamePrompt	Prompt to ask user if he wants to play another game
ScoreReport	Sends score report to an user via email
EndGameReport	Receives score report of user,prints it and hands it to user

Analysis of Original Design:-

Weaknesses:-

- **Redundant Code** : - The original code contained a lot of code which was either not used in the actual design or commented out. These unnecessary lines of code, if not removed, might have confused any future developer about the flow of the code. So the redundant or dead codes found in classes like ControlDesk, Lane, PinSetter, etc were all removed.
- **Code Repetitions** : - Code repetitions were found in many classes. One of the major changes that we did to reduce code repetition was to introduce a new class called GeneralView which contained functions to add a button and panel. Using this single class, all the other view classes can add buttons and panels in their own views instead of each class implementing these functions separately.
- **Unnecessary comments** : - Comments are added to a code to help the reader understand the logic and purpose of the code. Here in the original code design there was a lack of to the point well documented comments which may confuse the reader.
- **Large classes and functions** : - In the original code, few major classes were performing the bulk of the tasks of the project. To reduce this we broke down large classes into smaller classes with a specific functionality. For example:- PinSetter class was broken down into SimulateThrow which only simulated a ball throw and ScoreCalculate class which only calculated the score of a user.

Strengths:-

- Presence of well defined design patterns while designing subtasks of the project.

- Code functionalities were well defined and proper variable and function names were chosen to aid the code reader.
- The overall project in general had a low coupling metric which is an indication of a good design. Coupling refers to the dependencies between the various classes.

Fidelity to the Design Document:-

The original code more or less completed all the requirements mentioned in the initial design document. Design document fidelity refers to how precisely the design document of a model replicates the final product. In the original code, only the classes EndGameReport and ScoreReport responsible for fetching and sending email of game report to user were not working.

Design Patterns:-

- **Observer Pattern** : There are three observer classes present in the code using which event handling is done in the system. Any changes made in any of these three classes will be notified to all their subscribers and then the action will be performed.
- **Adapter Pattern** : Adapter pattern is a structural design pattern that works as a bridge between two incompatible interfaces. In the given system ControlDesk acts as an Adapter by joining Bowlers, Party and Queue subsystems.
- **Singleton Pattern** : A software design pattern that restricts the instantiation of a class to one "single" instance. This is clearly observed in the drive class, which acts as the main function in this program, and is instantiated only once in its lifetime.

Analysis of Refactored Code : -

Three new classes were added to the original code. Their names and responsibilities are given as follows:-

1. **GeneralView** : - Contains functions to add buttons and panels so that all the view classes can use these methods and code repetition can be reduced.
2. **SimulateThrow** : - Simulates a specific ball throw of a user and returns the result of the throw to PinSetter class. This reduces the load of the PinSetter class.
3. **ScoreCalculate** : - Calculates score of an user after taking input from PinSetter class and returns the result back to PinSetter class. This reduces the load on the PinSetter class.

Impact of Refactoring : -

- **Low Coupling** : - Coupling means how related are the classes to each other. Low coupling means that making some big changes in one class will not affect other classes.

In the refactored code, we have tried to make some classes which were heavily dependent on some other class independent by breaking them into subclasses. We have removed ActionListener from all the View classes due to which the buttons are directly calling the lambda method on click. This results in lower complexity as well as lower coupling.

- **High Cohesion** : - Cohesion means what a class or a module of code is doing or is meant to do. High Cohesion means that the class is focussed on its purpose-it only has functions performing what it is meant to do.

In the refactored code, we have tried to increase cohesion by reducing the number of tasks performed by the major classes like Lane,ControlDesk,PinSetter. An example of a change made by us is dividing PinSetter class into two added classes called ScoreCalculate and SimulateThrow.

- **Separation of concerns** : - It is a design principle which separates a program into different parts where each part performs a specific functionality.

In the refactored code, we have divided PinSetter class into SimulateThrow which only simulates a ball throw and ScoreCalculate which is only concerned about calculating the score.

- **Law of Demeter** : - The law of demeter states that each class should have as little information about the other classes as possible. We achieved this by removing code repetitions or unused codes of other classes present in a particular class and by ensuring low coupling and high cohesion. This means that the classes are focusses on their own task and are less dependent on other classes.
- **Information Hiding** : - Information hiding refers to keeping internal class data hidden to restrict its accessibility from outside the class. In the refactored code we have changed the accessibility of attributes and methods exposed to other classes from public to private.
- **Extensibility** : - Extensibility is the ability to extend a system and the effort required to add the extension. These extensions can be added by adding new functionalities or through changing existing ones. In the refactored code we have ensured easy and fast extensibility by reducing coupling and increasing cohesion in the

classes. Since making changes now would not affect other classes too much, adding extensions will be easier.

- **Reusability** : - Reusability is the ability of classes to use an existing code from a source multiple times. Here in the refactored code we have added the GeneralView class whose functions to add buttons and panels are reused multiple times by other view classes.




























Design Patterns Used:-

- **Observer** : - The observer classes present in the refactored code are ControlDeskObserver, LaneObserver and PinSetterObserver. Any changes in event will be notified to all the subscribers of above classes and the action will be performed.
- **Model-View-Controller (MVC)** : - This is a design pattern in the web used to organise code. It specifies that a program or application shall consist of data model, presentation information and control information. In our refactored code, we have used packages to separate the classes into three categories : - model, view and controller.

Metric Analysis:-

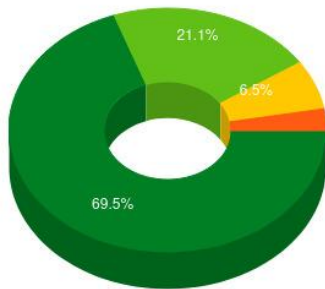
Metric of Original Design

List of all classes (#29)

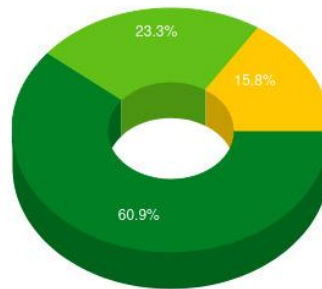
ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	Lane					227	medium-high	low-medium	medium-high	low-medium
2	ControlDeskView					87	low-medium	low-medium	low-medium	low-medium
3	ControlDesk					68	low-medium	low-medium	medium-high	low-medium
4	LaneStatusView					93	low	low-medium	low-medium	low-medium
5	LaneView					140	low-medium	low	low-medium	low-medium
6	AddPartyView					127	low-medium	low	low-medium	low-medium
7	PinSetterView					111	low	low	low	low-medium
8	NewPatronView					85	low	low	low	low-medium
9	EndGameReport					79	low	low	low-medium	low-medium
10	ScoreReport					76	low	low	low	low-medium
11	EndGamePrompt					55	low	low	low	low-medium
12	Pinsetter					47	low	low	low	low
13	LaneEvent					41	low	low	medium-high	low
14	BowlerFile					38	low	low	low	low

Metric of Original Design : -

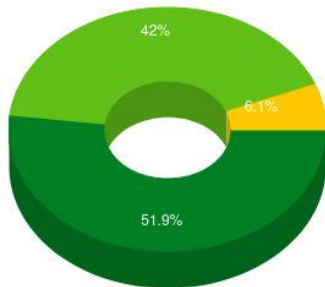
16	Bowler	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	25	low		low	low	low
17	PrintableText	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	21	low		low	low	low
18	ScoreHistoryFile	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	20	low		low	low	low
19	Score	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	16	low		low	low	low
20	Queue	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	12	low		low	low	low
21	LaneEventInterface	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	10	low		low	low	low
22	drive	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	8	low		low	low	low
23	Alley	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	6	low		low	low	low
24	ControlDeskEvent	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	6	low		low	low	low
25	Party	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	6	low		low	low	low
26	ControlDeskObserver	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	2	low		low	low	low
27	LaneObserver	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	2	low		low	low	low
28	LaneServer	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	<div><div></div></div>	2	low		low	low	low



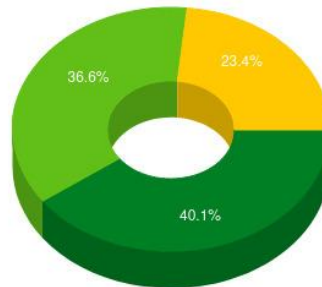
Lack of Tight Class Cohesion ▼



Weighted Method Count ▼



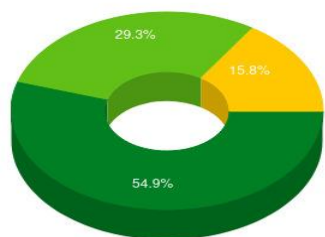
Access to Foreign Data ▼



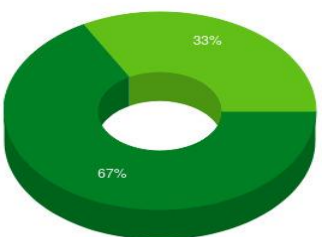
Lack of Cohesion Among Methods(1-CAM) ▼

Distribution of Quality Attributes

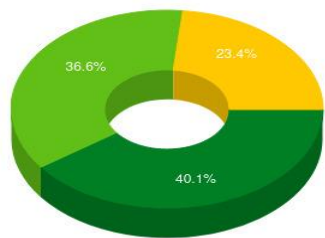
Complexity, Coupling, Cohesion, and Size



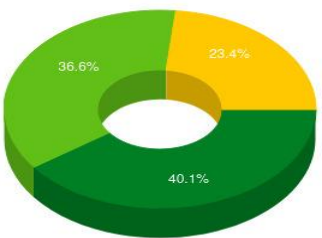
Complexity ▼



Coupling ▼






























































C3 ▼

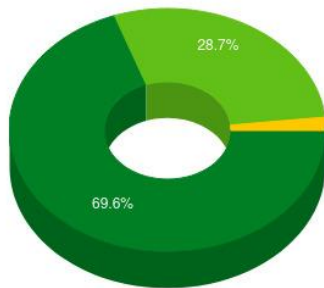


Lack of Cohesion ▼

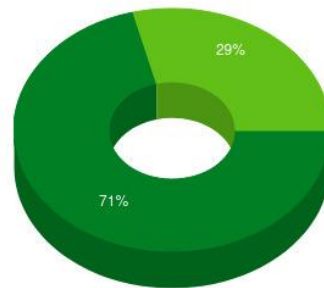
Metric of Refactored Design : -

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	Lane					167	low-medium	medium-high	medium-high	low-medium
2	ControlDeskView					80	low-medium	low-medium	low	low-medium
3	ControlDesk					57	low-medium	low-medium	low-medium	low-medium
4	LaneStatusView					97	low	low-medium	low	low-medium
5	LaneView					137	low-medium	low	low-medium	low-medium
6	scoreCalculate					129	low-medium	low	low-medium	low-medium
7	AddPartyView					124	low	low	low-medium	low-medium
8	PinSetterView					111	low	low	low	low-medium
9	NewPatronView					82	low	low	low	low-medium
10	EndGameReport					79	low	low	low-medium	low-medium
11	ScoreReport					76	low	low	low	low-medium
12	EndGamePrompt					55	low	low	low	low-medium
13	BowlerFile					36	low	low	low	low
14	Pinsetter					34	low	low	low	low
15	SimulateThrow					33	low	low	low	low
16	PinsetterEvent					26	low	low	low	low
17	Bowler					22	low	low	low	low

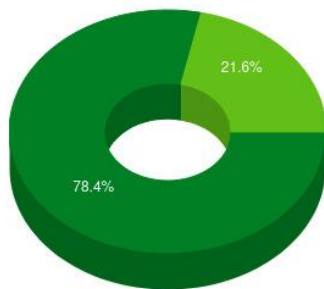
17	Bowler	<div></div>	<div></div>	<div></div>	<div></div>	22	low	low	low	low
18	LaneEvent	<div></div>	<div></div>	<div></div>	<div></div>	21	low	low	low	low
19	PrintableText	<div></div>	<div></div>	<div></div>	<div></div>	21	low	low	low	low
20	ScoreHistoryFile	<div></div>	<div></div>	<div></div>	<div></div>	20	low	low	low	low
21	Score	<div></div>	<div></div>	<div></div>	<div></div>	16	low	low	low	low
22	GeneralView	<div></div>	<div></div>	<div></div>	<div></div>	16	low	low	low	low
23	Queue	<div></div>	<div></div>	<div></div>	<div></div>	12	low	low	low	low
24	LaneEventInterface	<div></div>	<div></div>	<div></div>	<div></div>	10	low	low	low	low
25	drive	<div></div>	<div></div>	<div></div>	<div></div>	8	low	low	low	low
26	Party	<div></div>	<div></div>	<div></div>	<div></div>	6	low	low	low	low
27	Alley	<div></div>	<div></div>	<div></div>	<div></div>	6	low	low	low	low
28	ControlDeskEvent	<div></div>	<div></div>	<div></div>	<div></div>	6	low	low	low	low
29	PinsetterObserver	<div></div>	<div></div>	<div></div>	<div></div>	2	low	low	low	low
30	LaneServer	<div></div>	<div></div>	<div></div>	<div></div>	2	low	low	low	low
31	ControlDeskObserver	<div></div>	<div></div>	<div></div>	<div></div>	2	low	low	low	low
32	LaneObserver	<div></div>	<div></div>	<div></div>	<div></div>	2	low	low	low	low



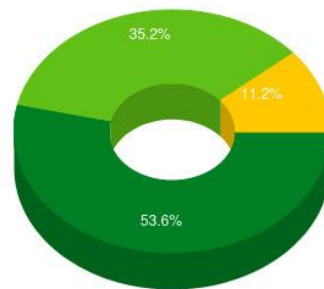
Lack of Tight Class Cohesion



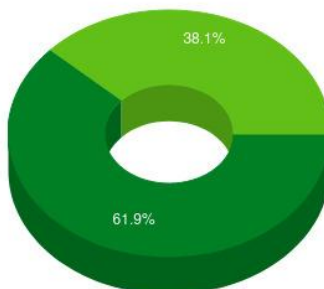
Weighted Method Count



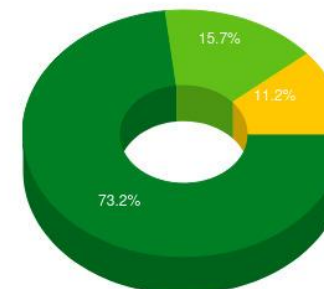
Access to Foreign Data



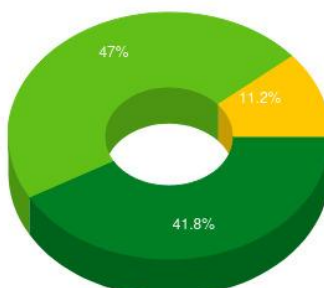
Lack of Cohesion Among Methods(1-CAM)



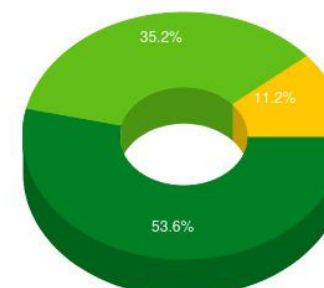
Complexity



Coupling



C3



Lack of Cohesion

Discussion of metric : -

1. What were the metrics for the code base? What did these initial measurements tell you about the system?

Ans 1) The code metrics used by us to compare the initial code design and the refactored code design were:- Coupling, Cohesion, Lines of Code in class, C3, Size of class, complexity, number of methods and lines of code in methods.

These metrics taken in the initial original code helped us in understanding the strengths and weaknesses in the original code. These strengths and weaknesses have already been discussed here:- [Analysis of Original Design:-](#). These also gave an indication of the type of refactoring we needed to perform since the original code performed well in some metrics and averagely in other metrics. So we focussed more on improving those metrics.

2. How did you use these measurements to guide your refactoring?

Ans 2) These measurements of code metrics helped us focus on the weaknesses present in the classes of the original design. This helped us to plan our refactoring stage in a systematic and organised manner. We also found out the code smells and the flaws in design using these metrics.

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?

Ans 3) We have given the results of the metrics of the original code and the refactored code. We can observe by comparing the metrics that more or less all issues in the code which were reducing the metrics have been resolved. The issues in the classes causing higher coupling, lower cohesion and complexity issues like the PinSetter, Lane, ControlDesk, etc were resolved by various ways like dividing into subclasses, removing redundant and repetitive codes and reducing interdependencies between the classes. We have created new classes for score calculation, throw simulation and general view to reduce coupling and increase cohesion. The impact of the refactoring on the original code can be found here:- [Impact of Refactoring](#)

[:-](#)