Design Document

Unit 1 project

Bowling Alley Simulation Refactoring

TEAM 30

Date Of Submission : 12-Feb-2021

Table of Contents :

Team Information

Name (Roll)	Work hours	Roles (includes but not limited to)
Dipankar Saha (2020201084)	16	Design Patterns/UML
Satyam Kumar Singh (2020201035)	15	Code Smells/UML
Sai Vishwak Gangam (2020202006)	16	Metric Analysis/Sequence Diagram
Swaraj Renghe (20171119)	15	Code Smells/Sequence Diagram

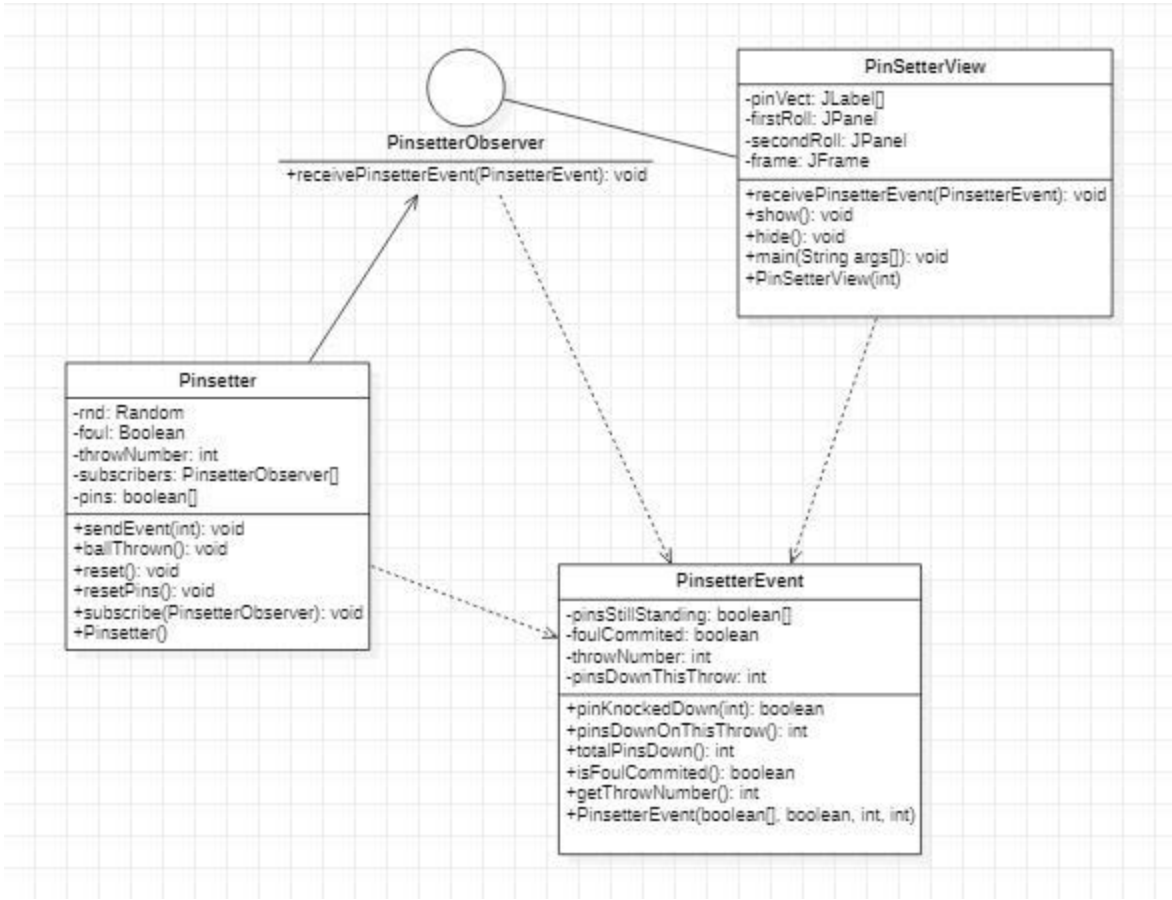
Overview

In this Document we present the Refactored Version of the original product (Bowling Management System) to automate the chain of bowling establishments located across the country. This original product had several designs and code reflecting poor software engineering design principles and some anti-patterns. We have made a fair attempt to recognize and solve those problems. Our refactored product caters to the requirement of installing new pin setting equipment which can detect the numbers of pins knocked down after a bowler has rolled his ball. This information can then be communicated to a scoring station that would be able to automatically score the bowler's game. It also establishes a service for customers that would maintain a history of a bowler's scores, average and other related information.

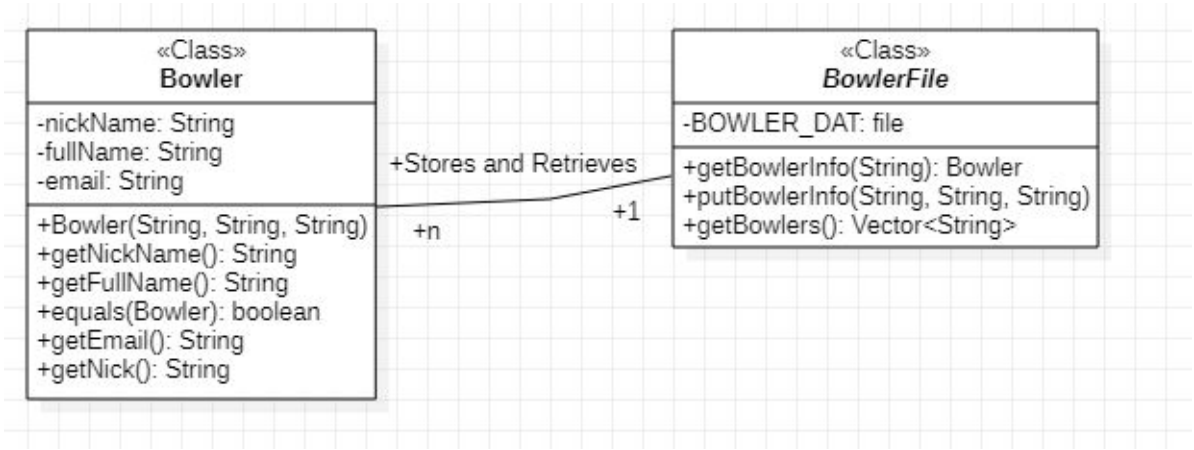
UML Class Diagram

UML Before Refactoring

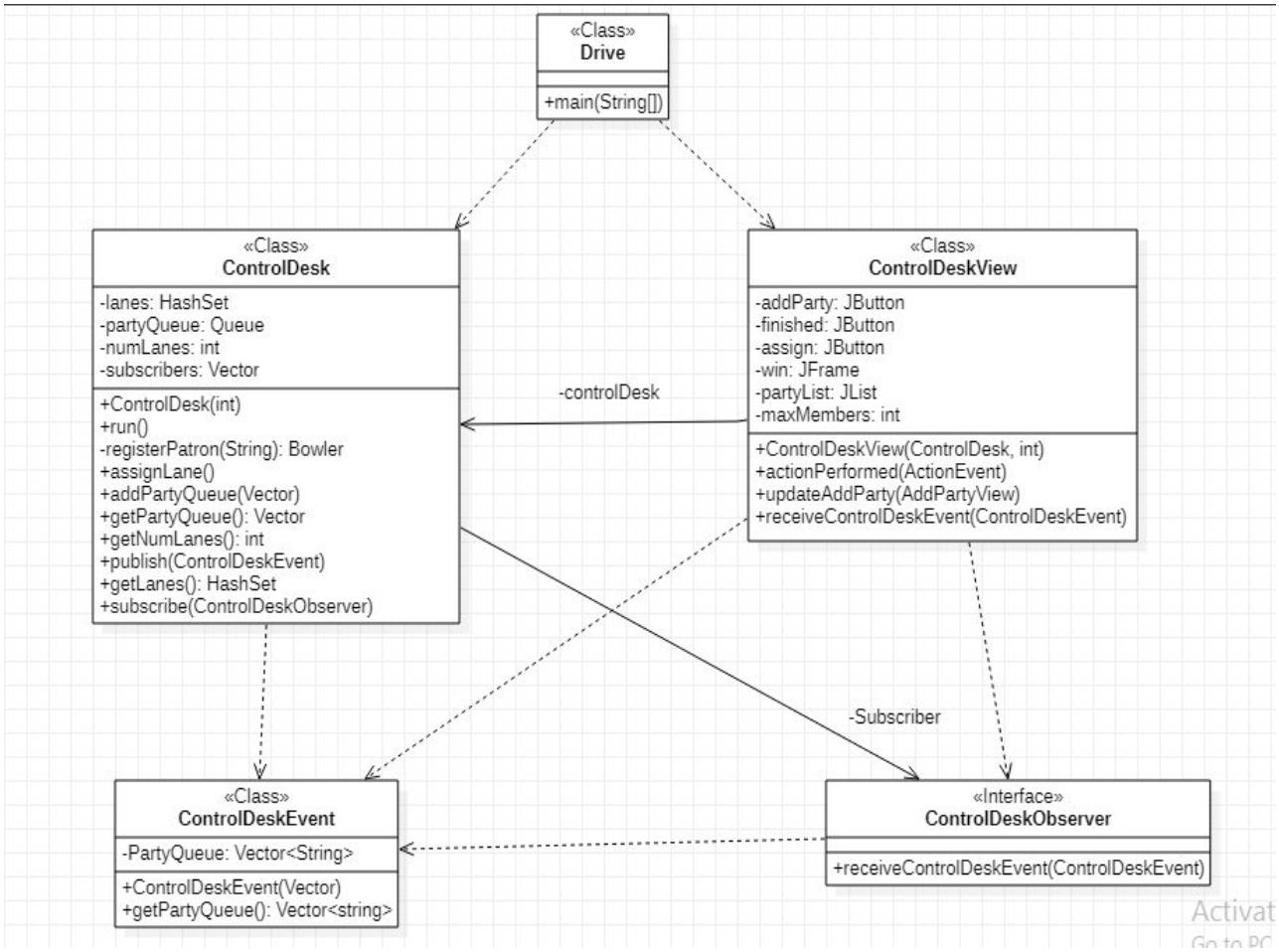
Pinsetter Classes



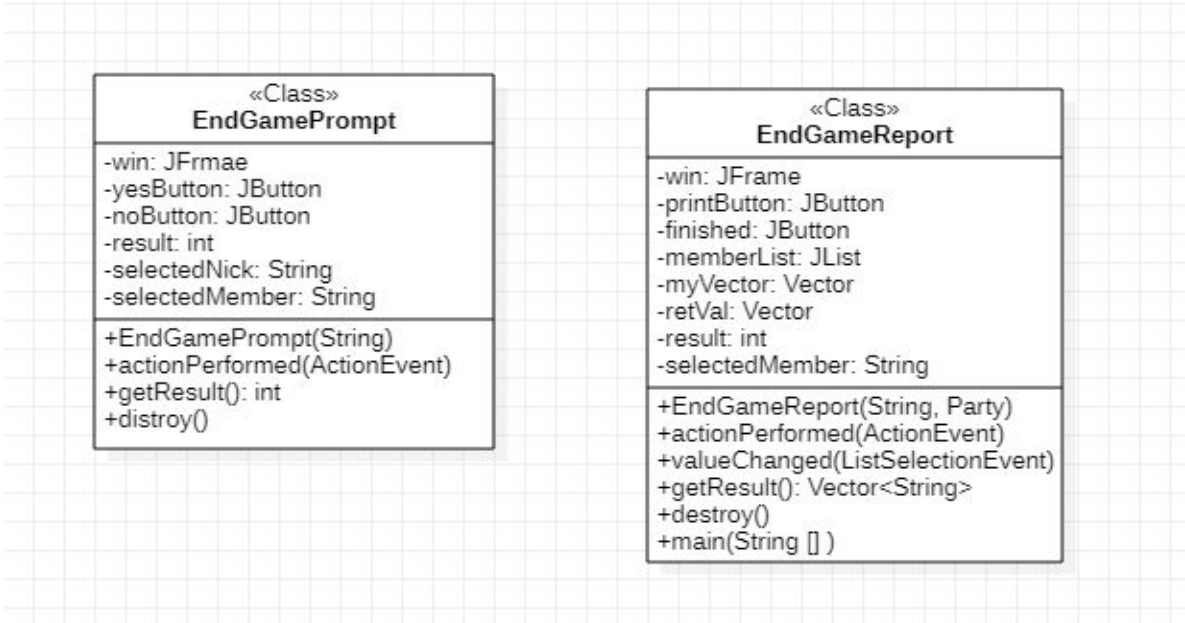
Bowler Classes



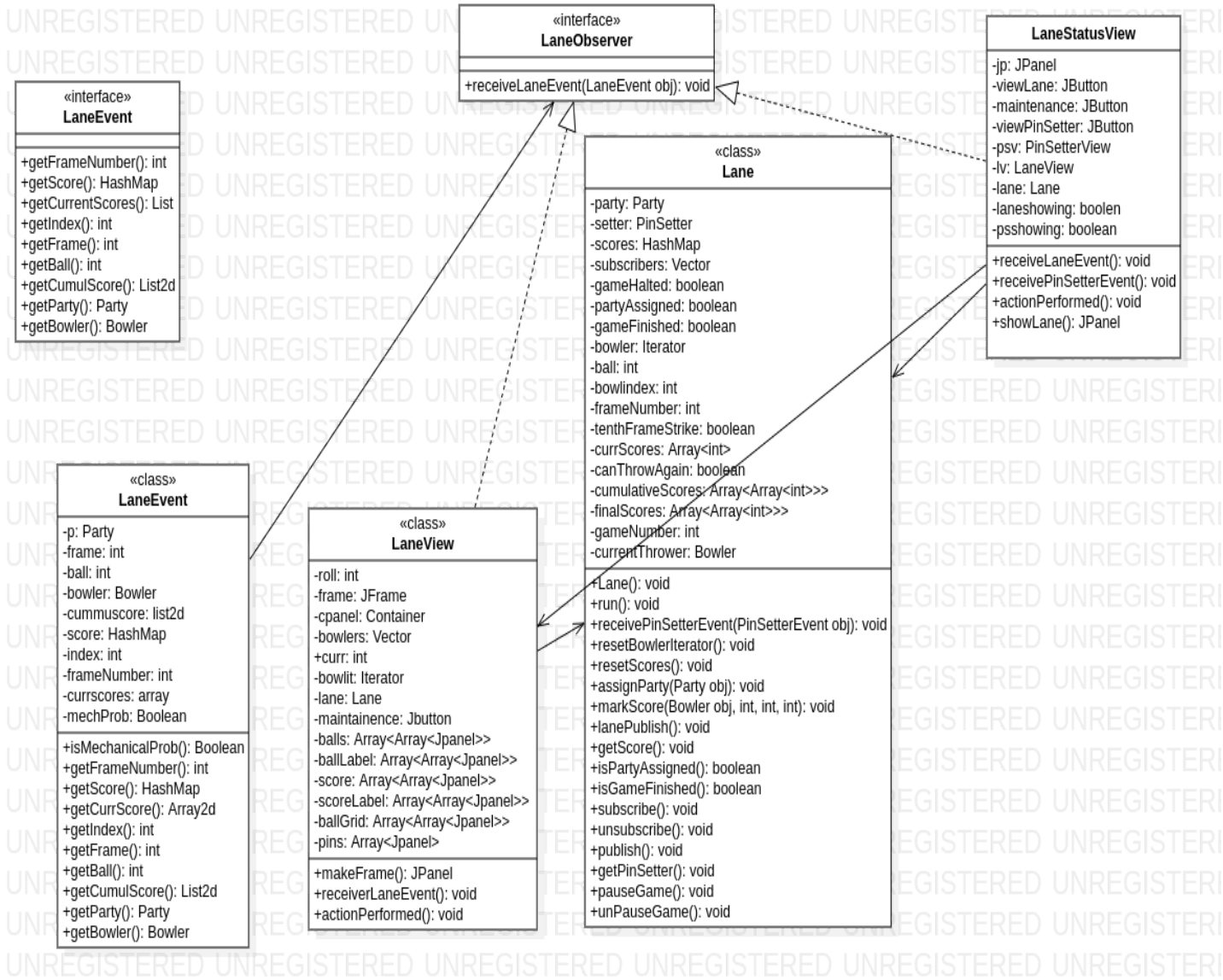
ControlDesk Classes



EndGame Classes

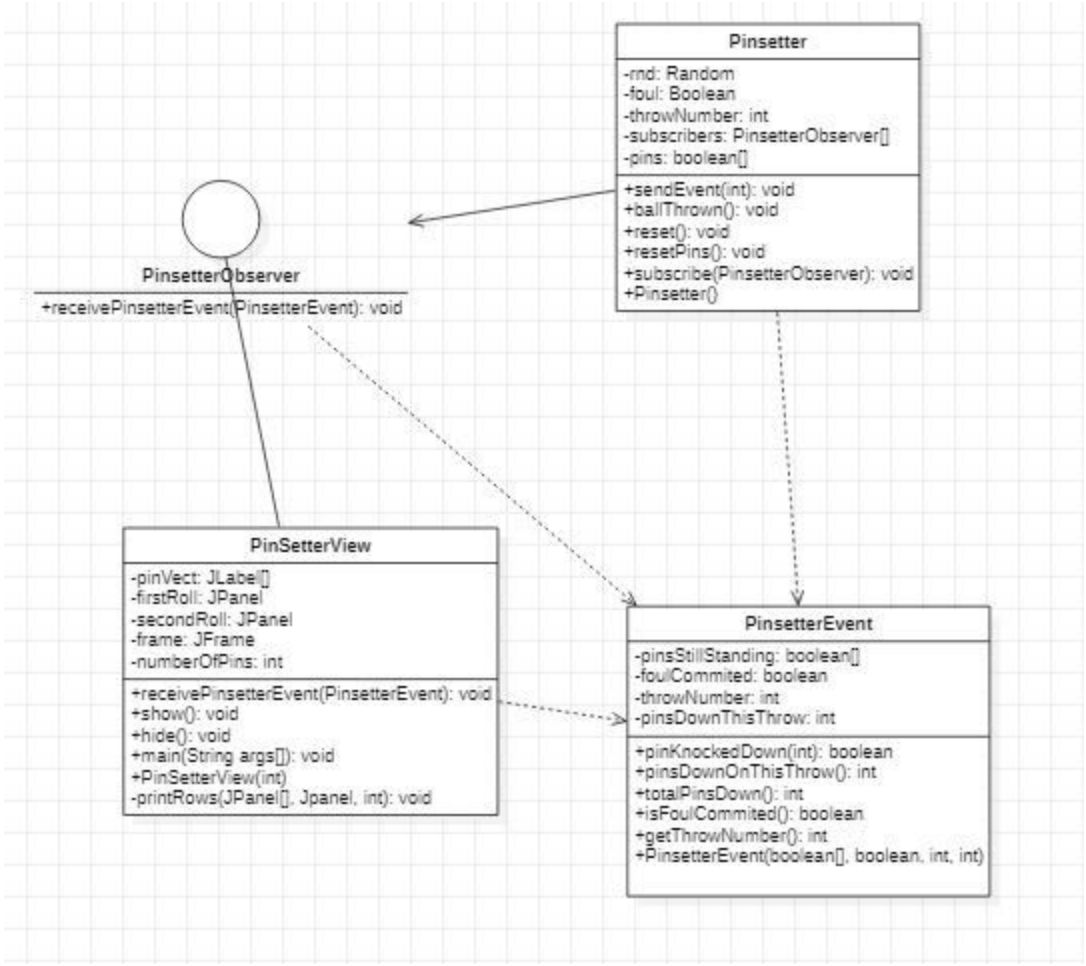


Lane Classes

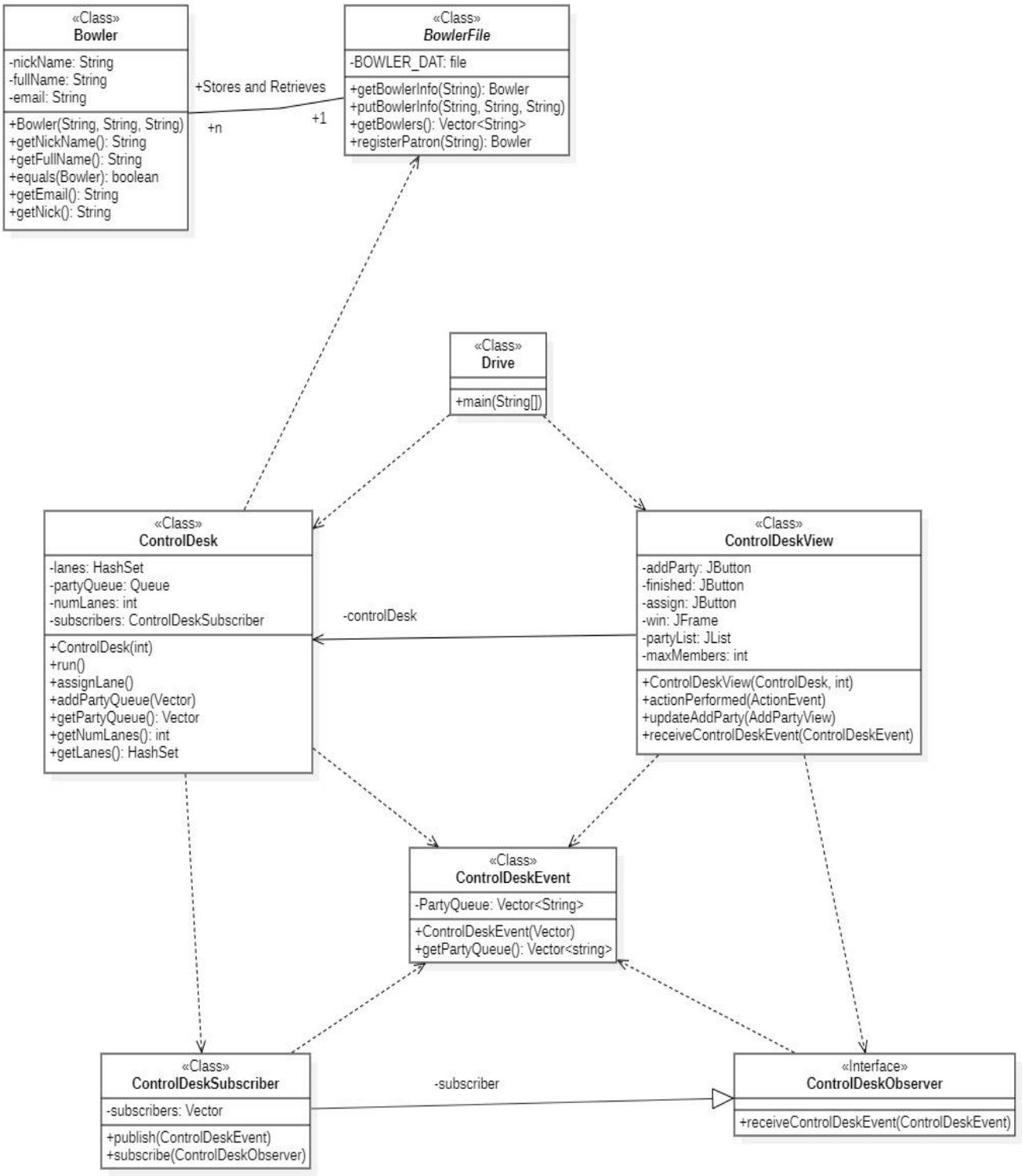


UML After Refactoring

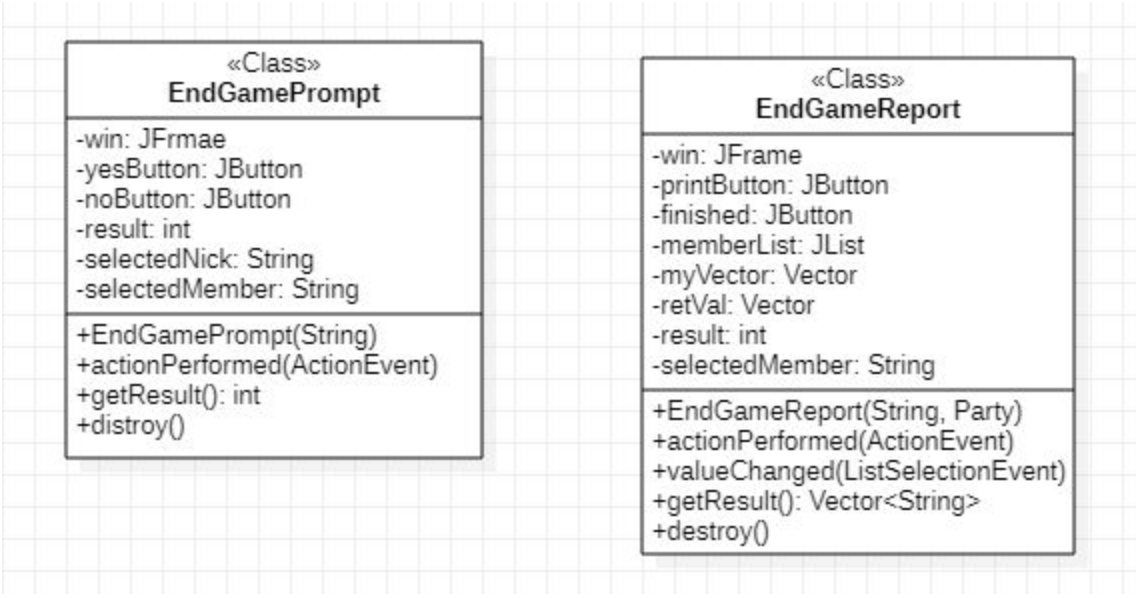
Pinsetter Classes



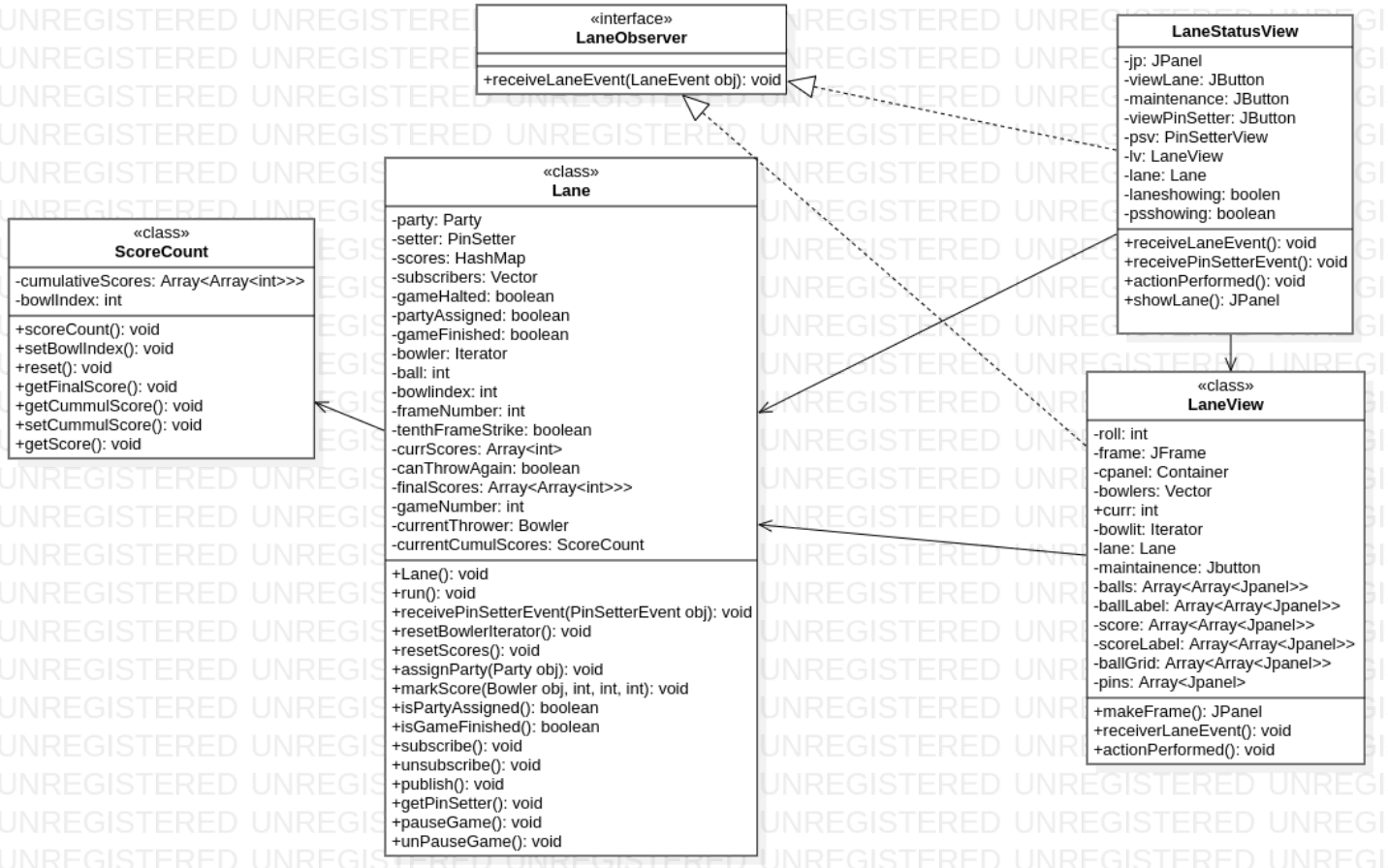
Bowler and ControlDesk classes



EndGame Classes

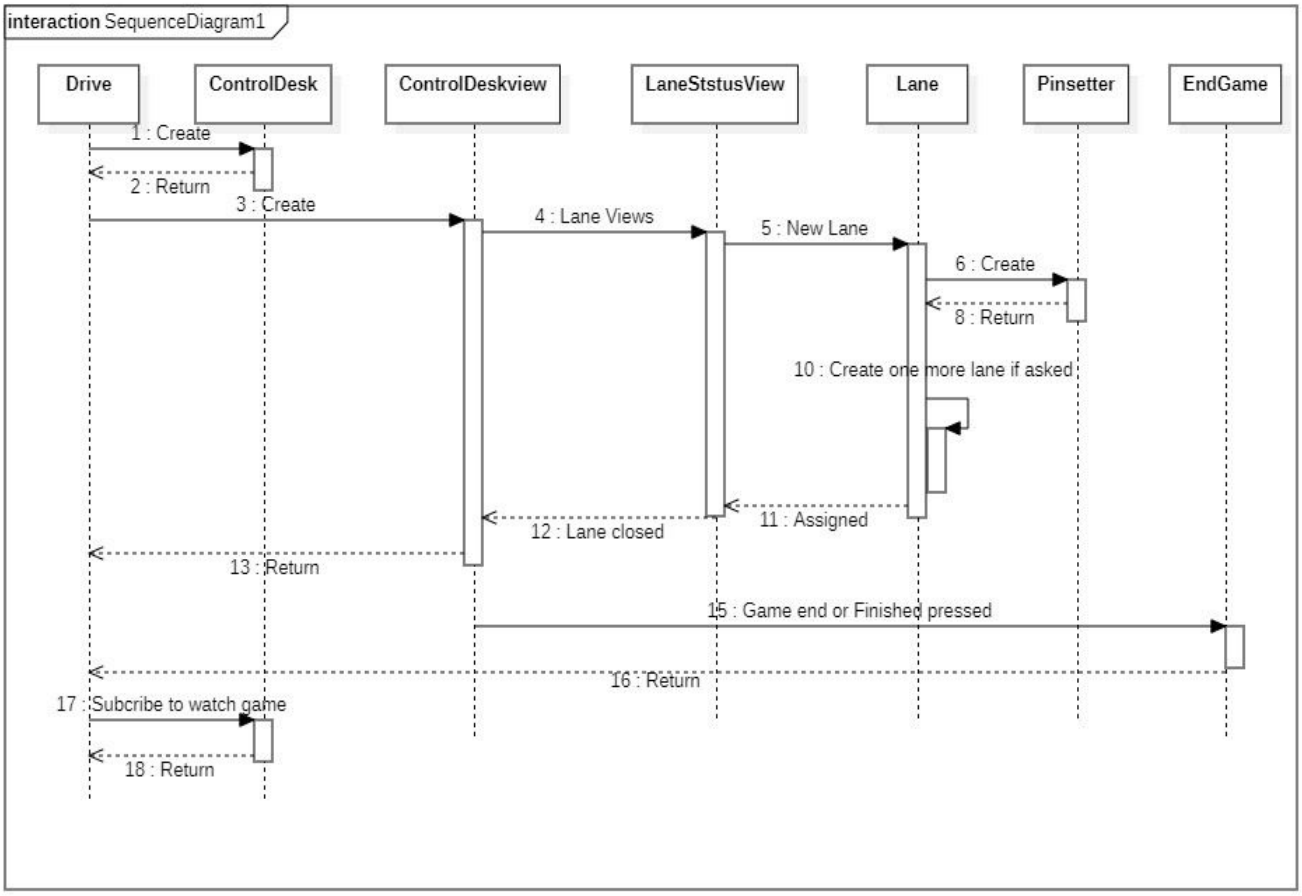


Lane Classes

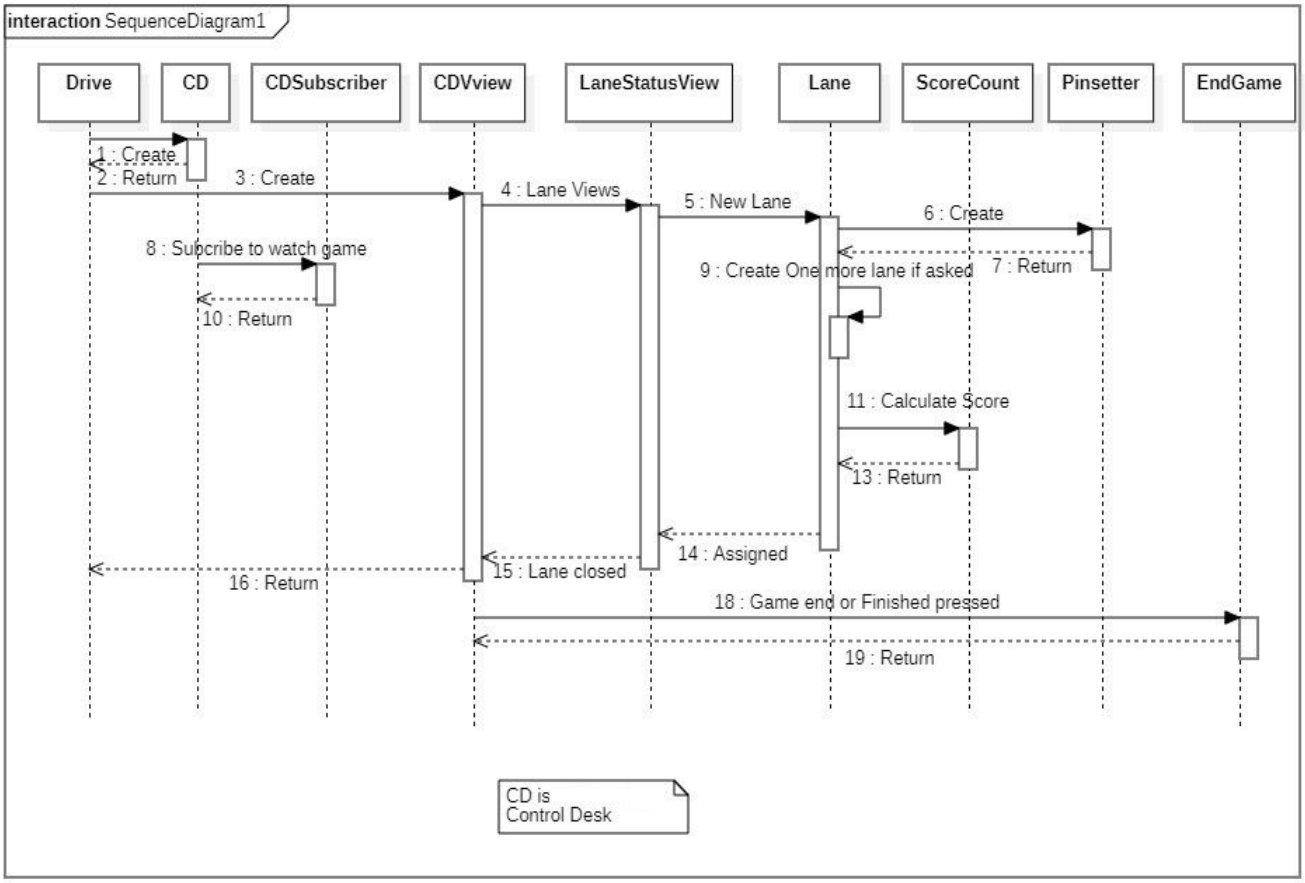


Sequence Diagram

Sequence Diagram Before Refactoring



Sequence Diagram After Refactoring



Responsibilities Of Classes

CLASS NAME	MAJOR RESPONSIBILITY
AddPartyView	GUI components need to add a party
Bowler	Holds all bowler information
BowloerFile	Interface to the Bowler database
ControlDesk	Initialize the main control desk and assigns Lanes to Parties
ControlDeskView	GUI components to represent of the control desk

Drive	Initialize the Game and create alley
Lane	Initialize lane, Simulate bowling game
LaneStatusView	GUI to Displays Lane Status, Lane's Pinsetter
Party	Container that holds bowlers
PinSetter	Represent the pinsetter that simulates dropping of pins on each throw
PinSetterView	GUI to show the status of pins after each throw
ScoreCount	Calculate scores got different ball types (normal/strike)
ScoreReport	Generate score report buffer, and send to user
ScoreHistoryFile	Manages Score database

Analysis of Original Design

Cons

1. **Dead Code :** The original design had a lot of code that was either commented out or not being used anywhere in the system. Those unwanted items were removed wherever necessary Including variables, methods in classes like : PinsetterView, LaneEvent. There were also multiple main methods in the system, which weren't being called/used anywhere.

2. **Single Large Method :** Most of the classes had very long methods trying to do too much, So we tried modularizing the code further in Classes like : LaneView broken into ScoreCount, PinsetterView, AddPartyView etc.

3. **Duplicate Code :** Similar kind of job was being done in some methods by simply copy-pasting the previous code. We reuse the code through creation of methods.

4. **Data Hiding :** Classes like the LaneEvent had its attributes not hidden from other classes using it's objects, So we made it private and provided appropriate getters & setters.

5. **Use of Old/Deprecated Tools :** Functions like show(), hide() etc. and use of AWT instead of newer Swing, and avoiding Generics, etc. was observed.

Pros

1. **Proper Comments :** The code was well commented, and purpose and functionality of most of the part of the system was provided.

2. **Low Coupling :** The overall system as well as the subsystems were having low coupling metric.

Fidelity to the Design Document

The original codebase mostly fulfilled the design document requirements attached with the code. Except for the "Print Report" functionality presented at the end of a game, which was not working.

Design Patterns

- **Observer Pattern :** The event handling done in the system on button click is a good example of observer pattern. Here we wait on thread for an event like a button click by the user, and notify the corresponding event-handler which carries out a task corresponding to the given button click.

- **Adapter Pattern :** As we know that Adapter pattern is a structural design pattern that works as a bridge between two incompatible interfaces. So in the given system the ControlDesk Class acts as an Adapter. It joins 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 Design

Responsibilities Of Newly created Classes

CLASS NAME	MAJOR RESPONSIBILITY
ControlDeskSubscriber	Maintain the Subscribers
ScoreCount	Calculate the score for every throw

How we achieved :

- **Low Coupling :**
  The dependencies between the classes were moderate and we have tried to make it low by passing the parameters locally and removing the redundant ones wherever possible.
  We have extended our class list to break down large files such as Lane into different subclasses. We have made sure that these classes were mostly independent and did not require too many other dependencies to increase the coupling.

- **High Cohesion**
  Cohesion tells about the consistency and organization of different units. The more tasks a single class tries to perform, we have a problem with cohesion there . The long classes had numerous methods which often became unrelated and too broad. We split such classes eg. ControlDesk to two Classes(ControlDesk and ControlDeskSubscriber) to divide the task and achieve cohesion.

- **Separation of Concerns**
  Separation of Concerns is a design principle for separating a system into distinct sections such that each section addresses a separate concern. An example of how we achieved in the refactored design is by creating a separate score calculating class. Previously Lane Class had a method getScore() which calculates the score but we have created a separate class ScoreCount for calculating the score and the updated score is sent to Lane Class to mark.

- **Dead Code Elimination**
  An optimization that removes code which does not affect the program results. We removed the multiple main methods in the system that weren't being called/used anywhere else. We also eliminated extra classes like, LaneEvent and PinsetterEvent. The implementations of these classes were unnecessary, and their functioning could be handled by existing classes without taking a hit on code complexity or cohesiveness.

- **Information Hiding**
  Data Abstraction or Information hiding is a crucial aspect of OOPS. Classes like the LaneEvent had its attributes exposed to the other classes through its objects, So we made it hidden by making it private. Appropriate getters & setters were provided.

- **The Law of Demeter**
  The fundamental notion of the LoD is that a given object should assume as little as possible about the structure or properties of anything else (including its subcomponents). That we achieved by ensuring low coupling and high cohesion. We have tried to make the classes as independent as possible so they have very few neighbours.

- **Extensibility**
  As we ensured low coupling, we made sure that it was easier to introduce new module. As we wrote basic code for the UI, we could easily extend it to add specific features in the respective classes.

- **Reusability**
  To ensure the code reusability several methods were written. Wherever in the original code we found that a similar kind of task was done by copy-pasting we modularized the code.

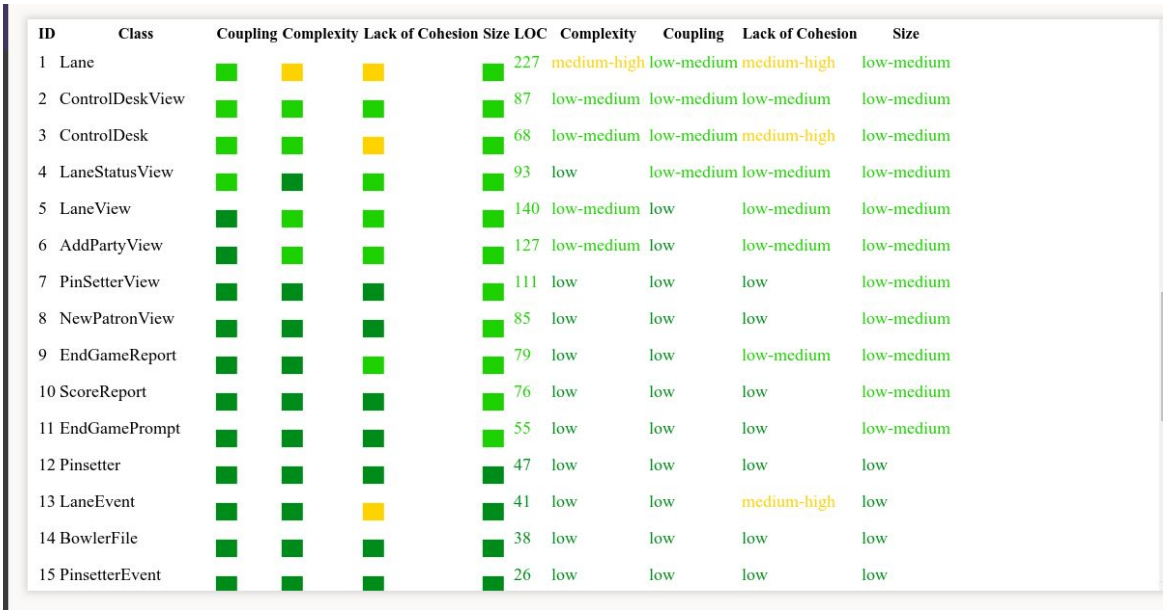
Design Patterns Used

- **Observer Pattern**
  The three observer classes present in the project are ControlDeskObserver, LaneObserver and PinSetterObserver any change in the event will be notified to all the subscribers of above classes and the respective action is done.

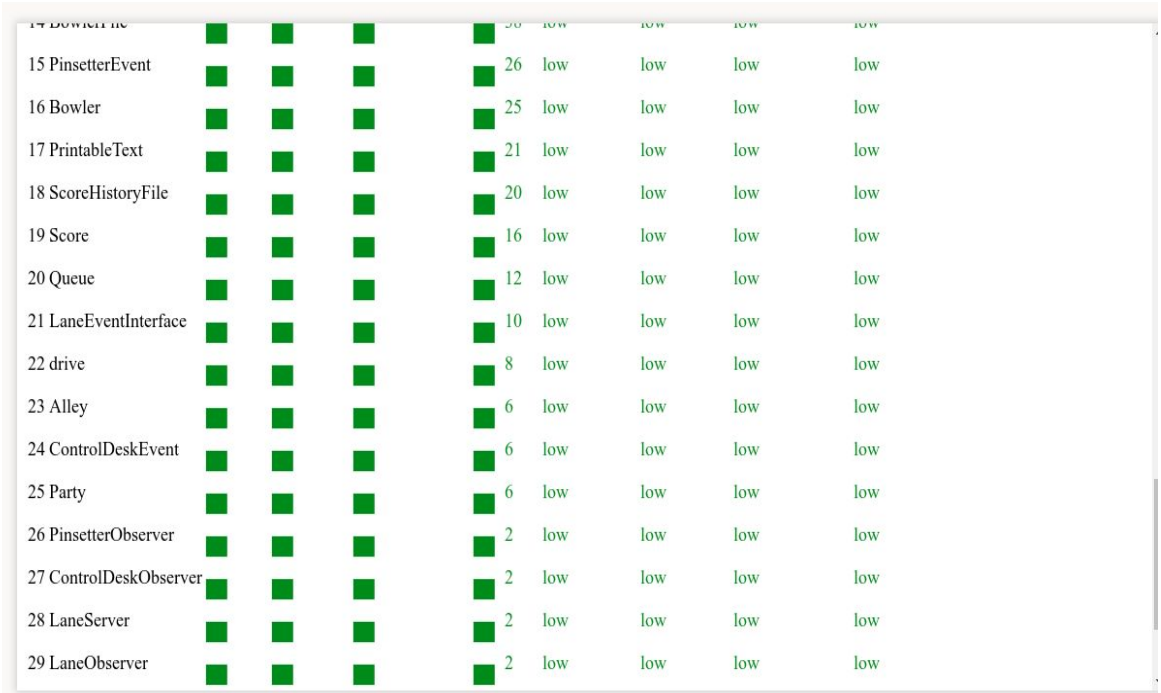
- **Singleton Class**
  This is a software design pattern that restricts the instantiation of a class to one single instance. In our design for example, the drive class is the main function of this game and is instantiated only once in a lifetime.

Metric Analysis

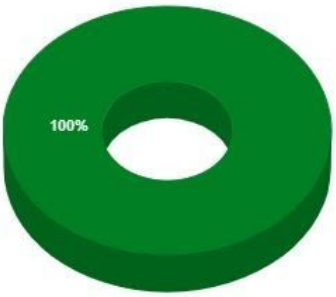
Metric of Original Design



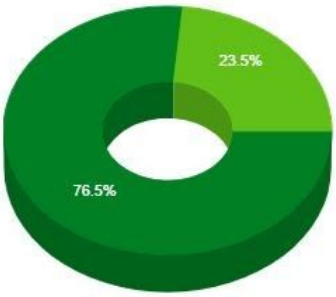
ID	Class	Coupling	Complexity	Lack of Cohesion	Size	LOC	Complexity	Coupling	Lack of Cohesion	Size
1	Lane					227	medium-high	low-medium	medium-high	low-medium
2	ControlDeskView					87	low-medium	low-medium	low-medium	low-medium
3	ControlDesk					68	low-medium	low-medium	medium-high	low-medium
4	LaneStatusView					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
15	PinsetterEvent					26	low	low	low	low



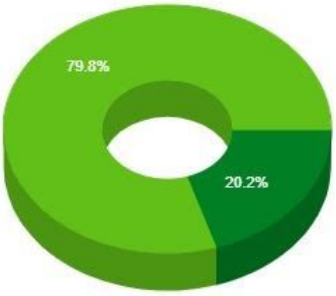
14 BowlerFile					30	low	low	low	low
15 PinsetterEvent					26	low	low	low	low
16 Bowler					25	low	low	low	low
17 PrintableText					21	low	low	low	low
18 ScoreHistoryFile					20	low	low	low	low
19 Score					16	low	low	low	low
20 Queue					12	low	low	low	low
21 LaneEventInterface					10	low	low	low	low
22 drive					8	low	low	low	low
23 Alley					6	low	low	low	low
24 ControlDeskEvent					6	low	low	low	low
25 Party					6	low	low	low	low
26 PinsetterObserver					2	low	low	low	low
27 ControlDeskObserver					2	low	low	low	low
28 LaneServer					2	low	low	low	low
29 LaneObserver					2	low	low	low	low



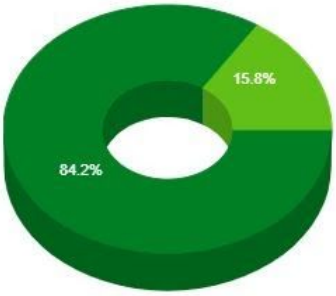
Number of Methods



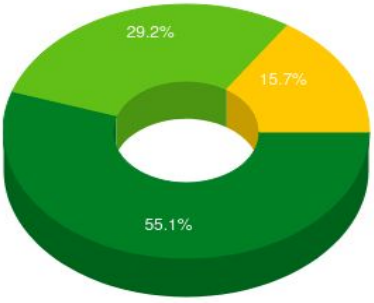
Class-Methods Lines of Code



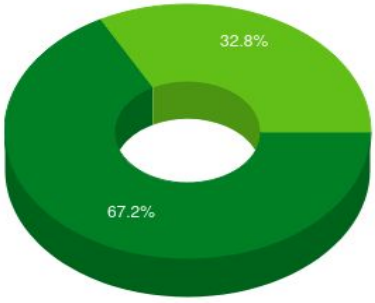
Class Lines of Code



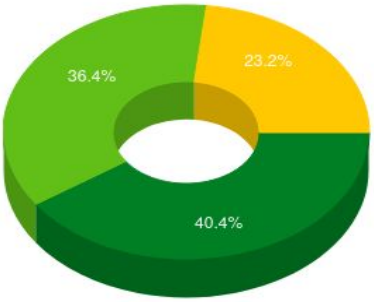
C3



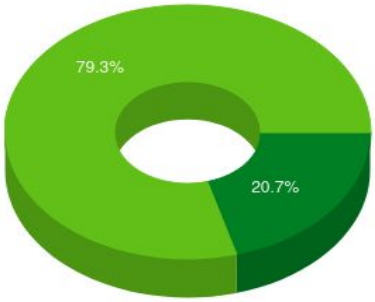
Complexity



Coupling

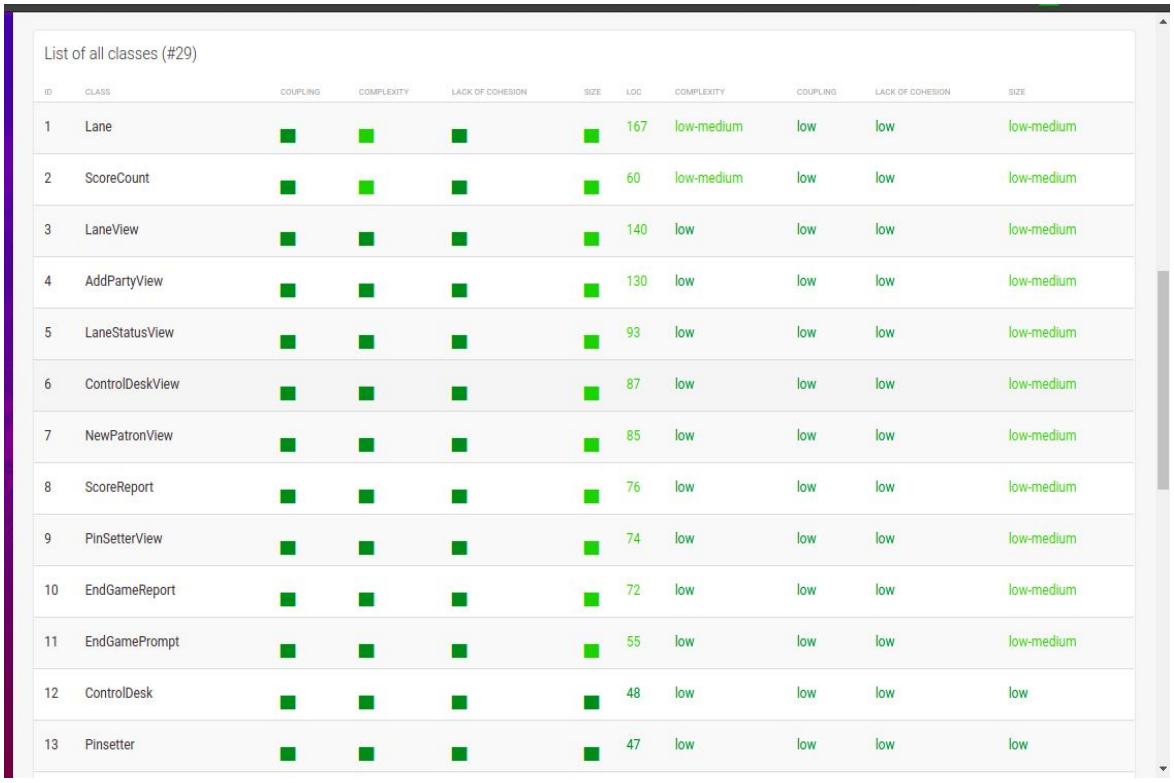


Lack of Cohesion



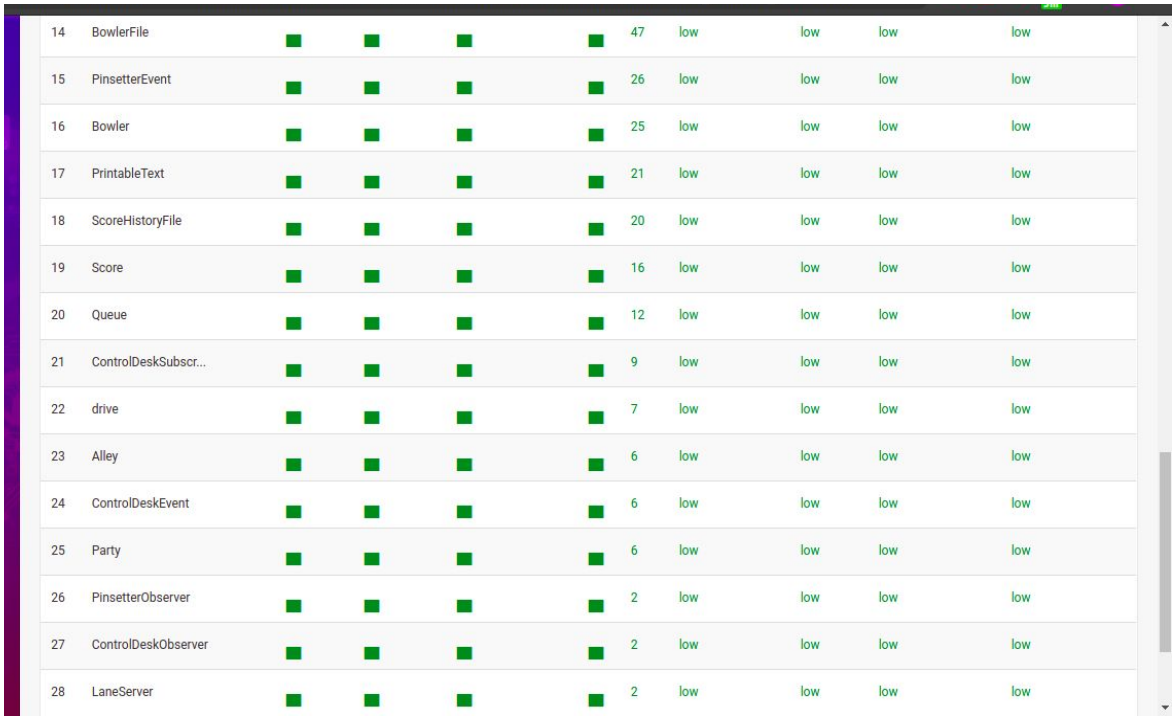
Size

Metrics of Refactored Design

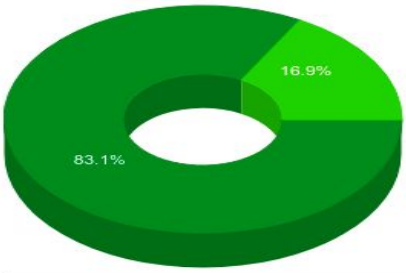


List of all classes (#29)

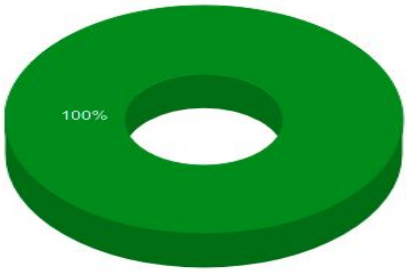
ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	Lane					167	low-medium	low	low	low-medium
2	ScoreCount					60	low-medium	low	low	low-medium
3	LaneView					140	low	low	low	low-medium
4	AddPartyView					130	low	low	low	low-medium
5	LaneStatusView					93	low	low	low	low-medium
6	ControlDeskView					87	low	low	low	low-medium
7	NewPatronView					85	low	low	low	low-medium
8	ScoreReport					76	low	low	low	low-medium
9	PinSetterView					74	low	low	low	low-medium
10	EndGameReport					72	low	low	low	low-medium
11	EndGamePrompt					55	low	low	low	low-medium
12	ControlDesk					48	low	low	low	low
13	Pinsetter					47	low	low	low	low



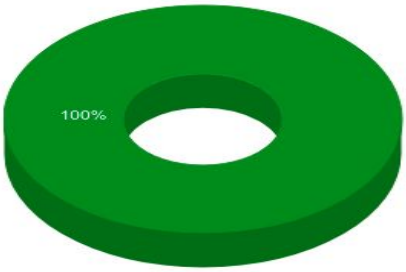
14	BowlerFile					47	low	low	low	low
15	PinsetterEvent					26	low	low	low	low
16	Bowler					25	low	low	low	low
17	PrintableText					21	low	low	low	low
18	ScoreHistoryFile					20	low	low	low	low
19	Score					16	low	low	low	low
20	Queue					12	low	low	low	low
21	ControlDeskSubscr...					9	low	low	low	low
22	drive					7	low	low	low	low
23	Alley					6	low	low	low	low
24	ControlDeskEvent					6	low	low	low	low
25	Party					6	low	low	low	low
26	PinsetterObserver					2	low	low	low	low
27	ControlDeskObserver					2	low	low	low	low
28	LaneServer					2	low	low	low	low



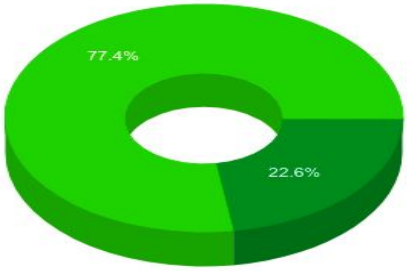
Complexity



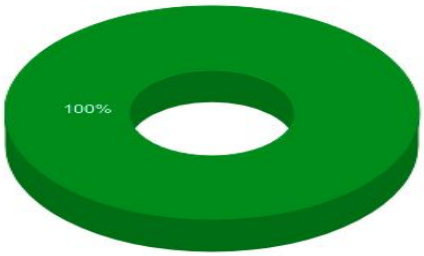
Coupling



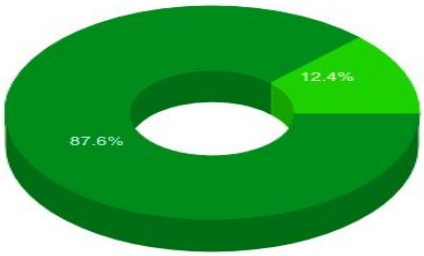
Lack of Cohesion



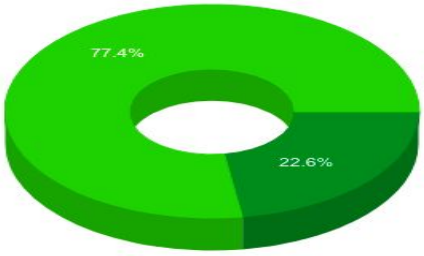
Size



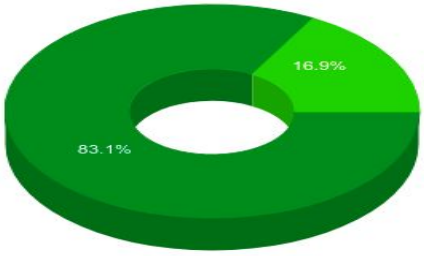
Number of Methods



Class-Methods Lines of Code



Class Lines of Code



C3

Discussion of metric

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

   There were several metrics for the code base used for analysis of the original and the refactored code like : Coupling, Cohesion, Lines Of Code, Cyclomatic Complexity, Modularity, Data Hiding, Size of classes & methods, extensibility, reusability etc. as shown in the graphs and tables here.

   These measurements highlighted several aspects of the original system that we have discussed in detail in the analysis section of the original design here.

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

   The insights that we gathered from these measurement helped to to refactor the original design in an organized and well targeted manner. The detailed discussion of how these measurement helped us to identify the anti-patterns etc. and our approach to solve these problems to an acceptable extent is here.

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?*

   As observed by looking at the metric measurements shown here. We can conclude that we have decreased the complexity and cohesion of Lane Class by deleting the Redundant classes like LaneEventInterface and LaneEvent. We have also created Separate class for score calculation which helped us to increase the cohesiveness of Lane class and the newly created class. The same way we have created a separate class for subscribers and improved the cohesiveness of ControlDesk class. Data encapsulation has been increased by making the attribute private to LaneEvent class. Apart from this we have tried to improve other areas of overall code by various means which can be found here.