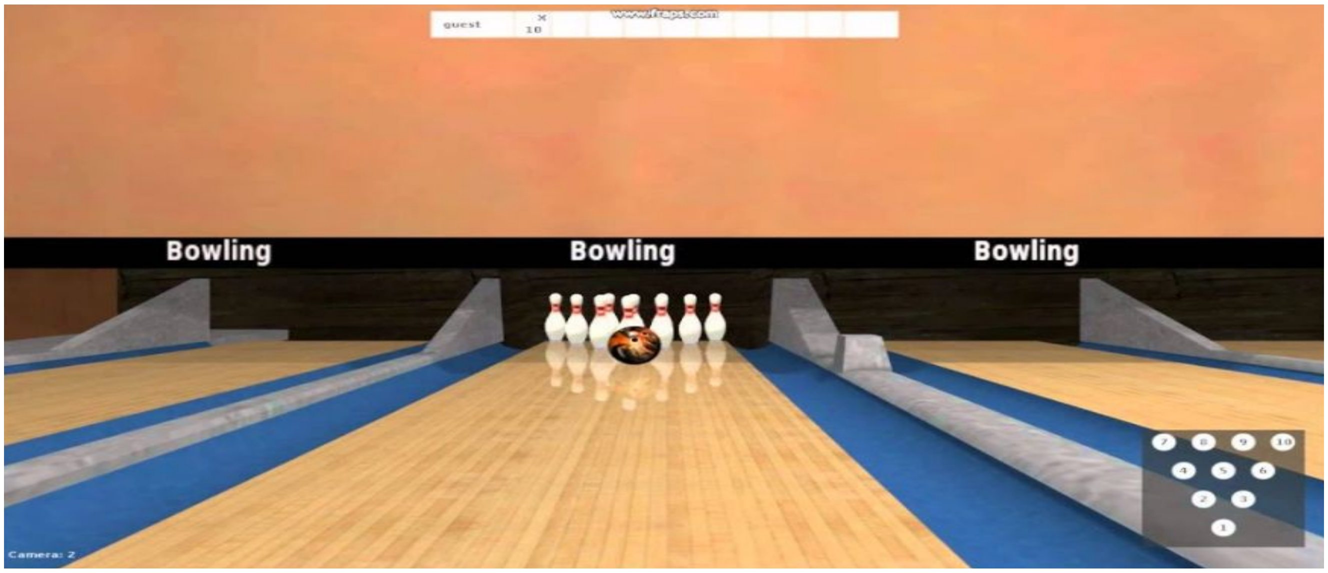
Software Engineering
Unit-1 | Bowling Alley Simulation
Refactored Design Document

Team Number: 6

Date of Submission: 10th Feb'21

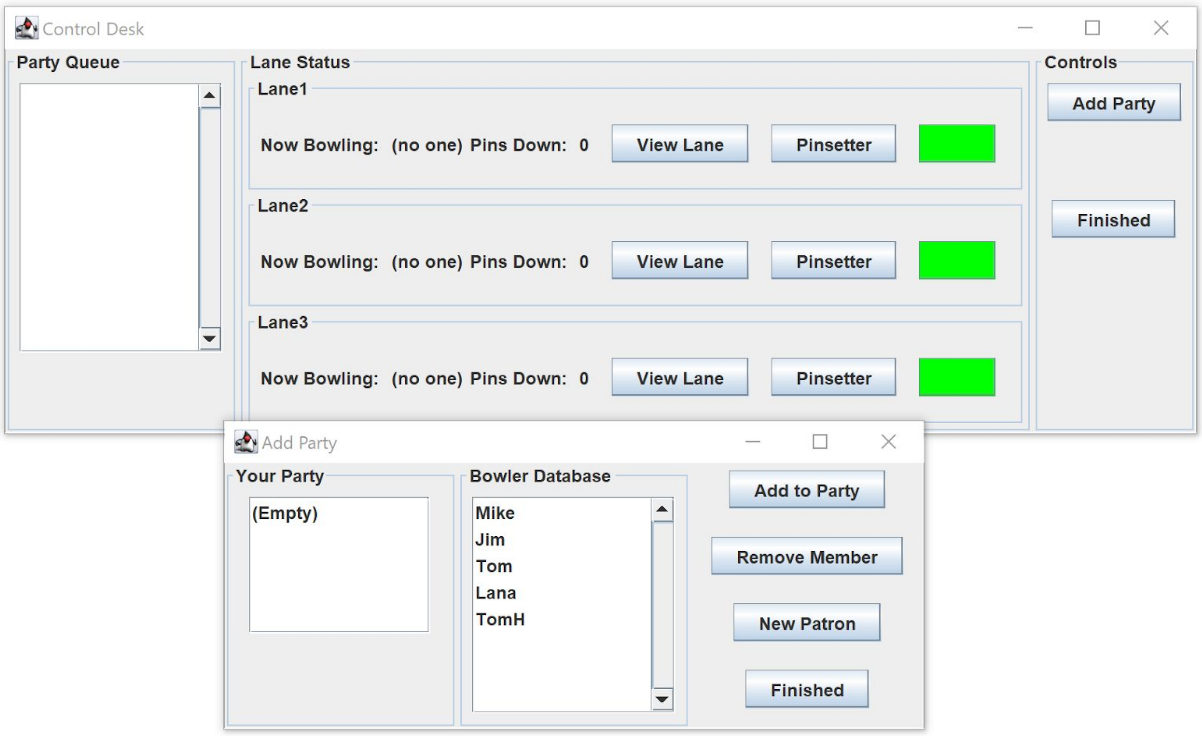
Team Members
Vaibhav Garg - 20171005
Akshay Goindani - 20171108
Hitesh Kumar - 2019201039
Surekha Medapati - 2018900085



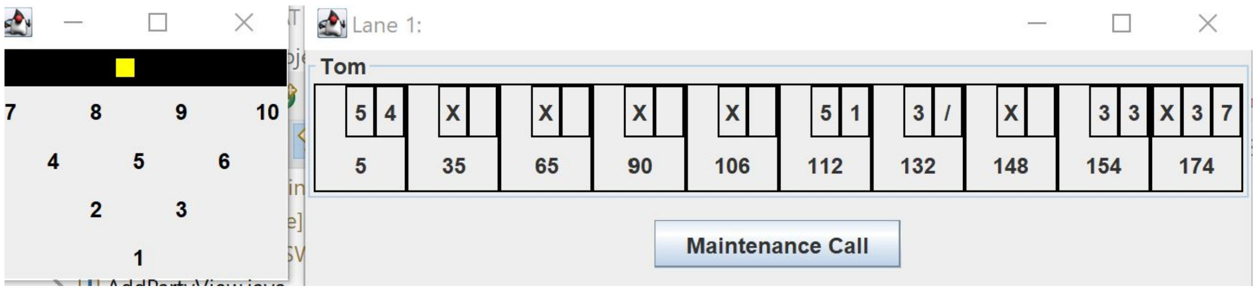
Product Overview:

The Bowling Alley simulation application is a software which aims to automate the workflow of a typical Bowling Alley. It is a fairly complete application in itself as it provides an interface for a new bowler registration up till the final score-report after a party has ended their game. It has been developed entirely using Java and some important features of the product are as follows:

ControlDesk: Handles the main party queue, and looks over the lane allocation process. It also monitors the scores of any active lane.



It has a dynamic display of the game where it shows the number of pins knocked and also the progress of the game(score).



5 4	X	X	X	X	5 1	3 /	X	3 3	X 3 7
5	35	65	90	106	112	132	148	154	174

Pinsetter:The pinsetter interface, which communicates with the scoring station and gives a score based on the pins that are left after every throw. It also re-racks the pins, meaning it places all ten down after two consecutive throws have been detected.

Lane Management: The software can have a variable number of lanes and a single lane is assigned to one party. Each lane can accommodate a certain fixed number of bowlers. The order in which a party checks in determines the order in which they will bowl.

Creating a new player/patron: Bowlers perform a one-time registration the first time they check in at the control desk by providing their full name and an email address. When a bowler has checked out at the control desk after completing his games, a report is generated containing the bowler's scores from games just completed, previous scores and his current average. Bowlers information and game scores is stored in the database. This report is automatically emailed to the bowler. The bowler may also request a printout.

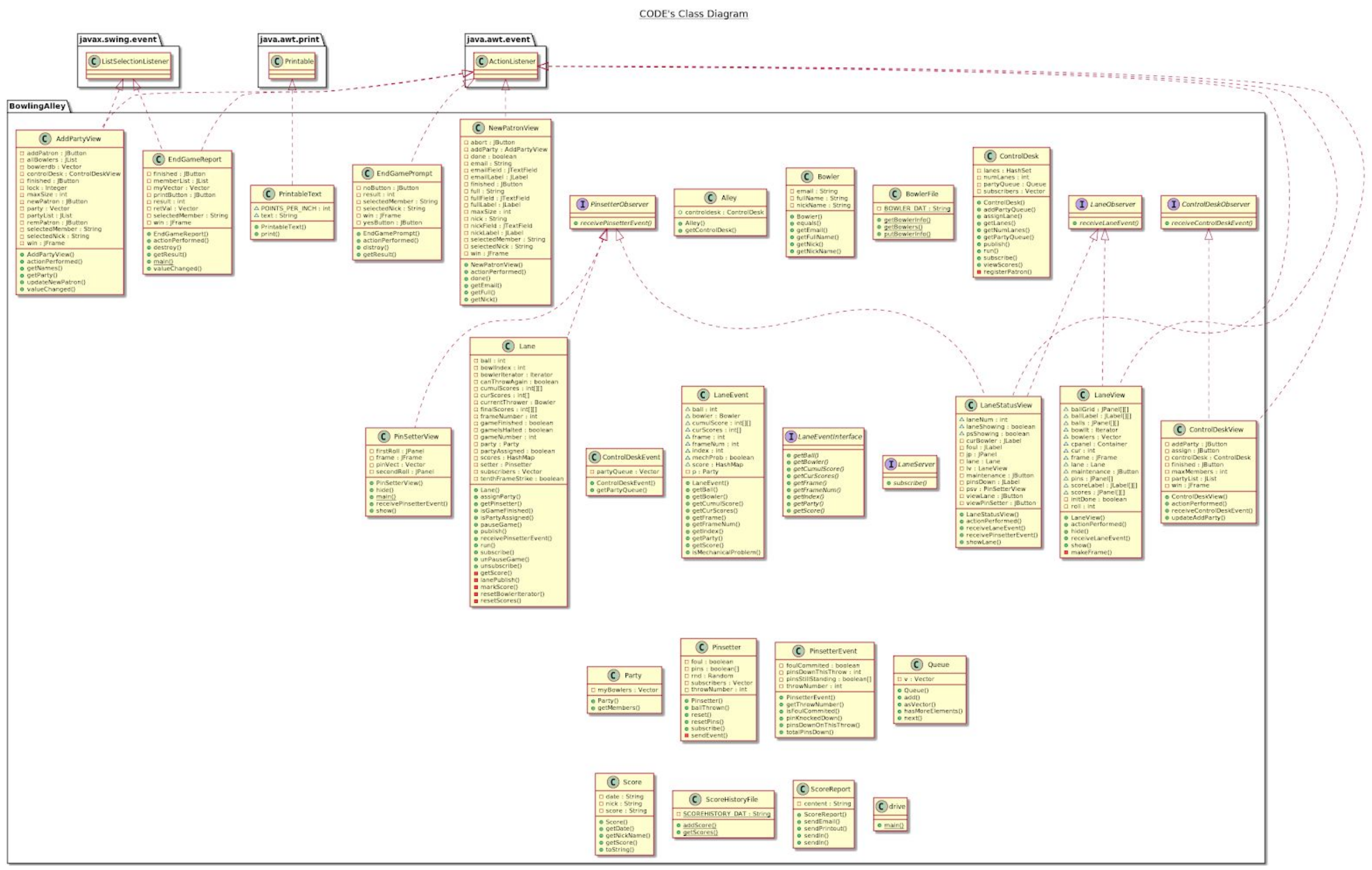
View Scoreboard: The scoreboard keeps track of the points gained by each player. It follows normal bowling score calculation technique as a strike is achieved when all the pins are knocked down on the first roll(score=10+pins dropped on next 2 turns), and a spare is achieved if all the pins are knocked over on a second roll(score=10+pins dropped on a next turn).

Maintenance call: It does the repair work like pins are not re-racked, balls are not returned etc., the game is automatically halted at the time of maintenance call. The control station sends an acknowledgement of the request back to the scoring station.

Original Design Analysis

Class Diagram

**The diagram was generated using the PlantUML plug-in from IntelliJ*



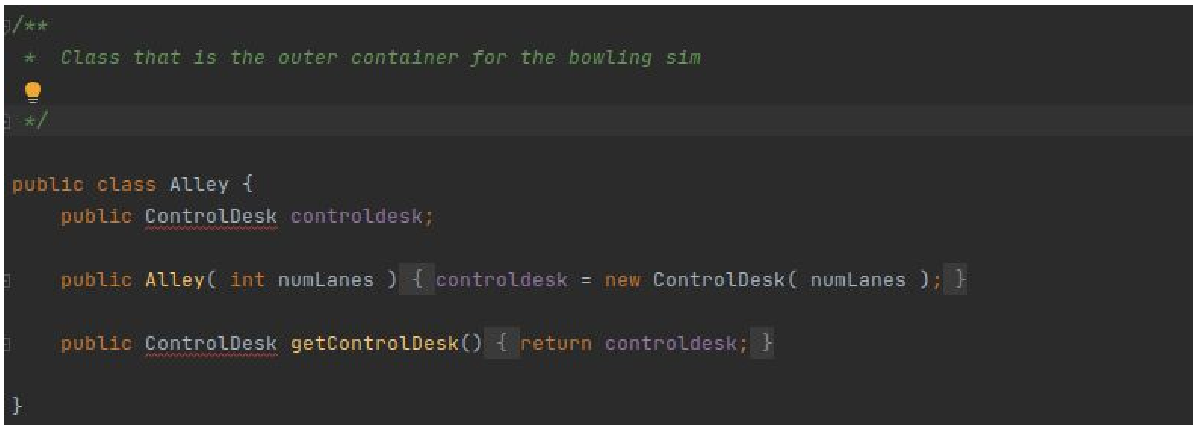
<u>Major Control Class Responsibilities</u>

Class	**Responsibility**
ControlDesk	Firstly, the control desk instantiates all the lanes in the bowling alley, and then it acts as a higher-level manager between the bowlers and the alley by maintaining a party queue and assigning any available lane to the party which has been waiting for the longest period of time. It also notifies all its observers/subscribers in case of certain state changes.
Lane	A lane instantiates the pinsetter associated with itself and its primary responsibility is to simulate the game at a somewhat higher level by making the current bowler throw the ball, process the event received from its pinsetter, and accordingly calculate and manage the scores. It also notifies its observers after every throw and does some post-game processing.
BowlerFile	This class interacts directly with the bowler database and provides functionality for adding a new bowler to the database and retrieving bowler info from the database.
Pinsetter	A pinsetter handles the 10 pins associated with it, simulates a

	random throw, and notifies all its observers of the outcomes and implications of the throw.
ScoreHistoryFile	This class interacts directly with the score history database and provides functionality for writing and reading scores from the database.

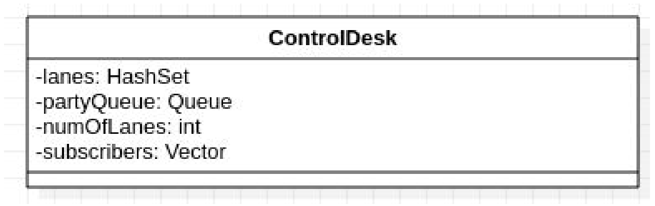
<u>Design Flaws</u>

1. The Alley class is merely behaving as a container for the ControlDesk class and has no real purpose leading to unnecessary coupling and low cohesion.



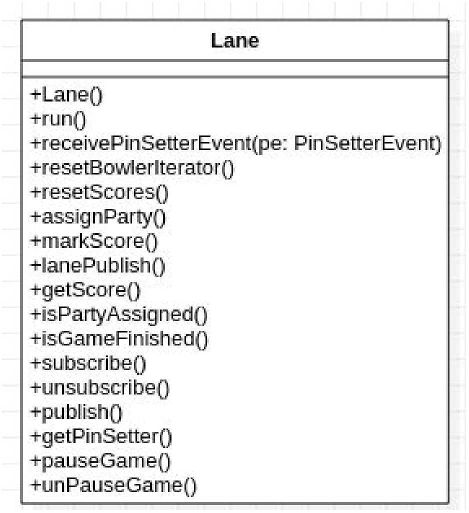
```java
/**
 *  Class that is the outer container for the bowling sim
 *
 */

public class Alley {
    public ControlDesk controldesk;

    public Alley( int numLanes ) { controldesk = new ControlDesk( numLanes ); }

    public ControlDesk getControlDesk() { return controldesk; }

}
```

2. The maxPatronsPerParty should be a ControlDesk attribute and not a ControlDeskView attribute for proper encapsulation.



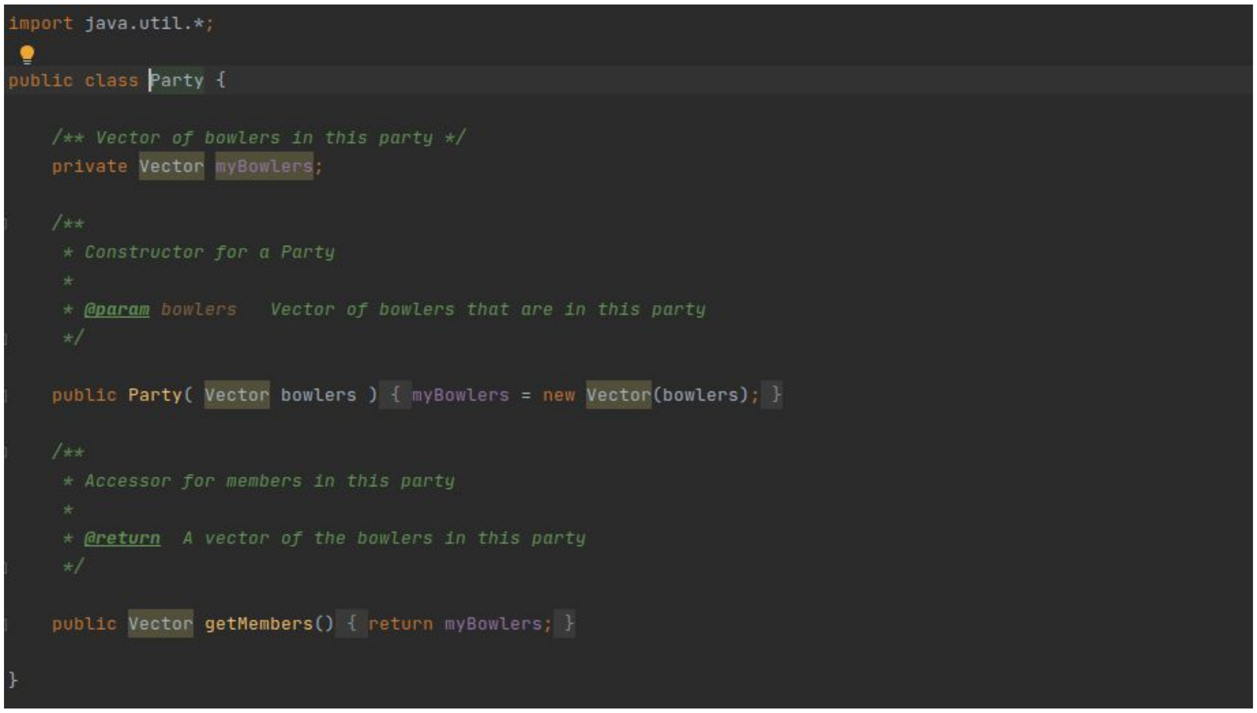
3. Both *run* and *getScore* methods of the Lane class have extremely high cyclomatic complexity and are also very difficult to read and understand.

4. There is low cohesion in the Lane class since it is doing both things, simulating the game and calculating the scores.



5. LaneEventInterface is not being used anywhere and is dead code.

6. There are many data classes like Bowler and Party.

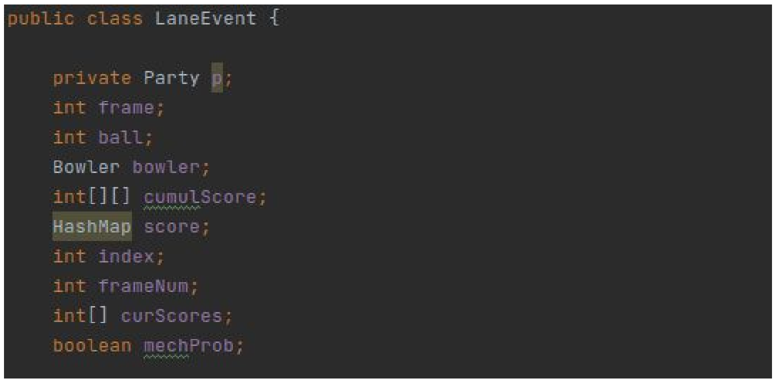


```java
import java.util.*;

public class Party {

    /** Vector of bowlers in this party */
    private Vector myBowlers;

    /**
     * Constructor for a Party
     *
     * @param bowlers   Vector of bowlers that are in this party
     */

    public Party( Vector bowlers ) { myBowlers = new Vector(bowlers); }

    /**
     * Accessor for members in this party
     *
     * @return  A vector of the bowlers in this party
     */

    public Vector getMembers() { return myBowlers; }

}
```

7. The BowlerFile and ScoreHistoryFile are highly coupled with the database which makes it difficult to integrate another kind of database in the future.

8. All the source code is in just one directory without any proper packaging and dependency structure.

9. Some attributes are unnecessarily not private.

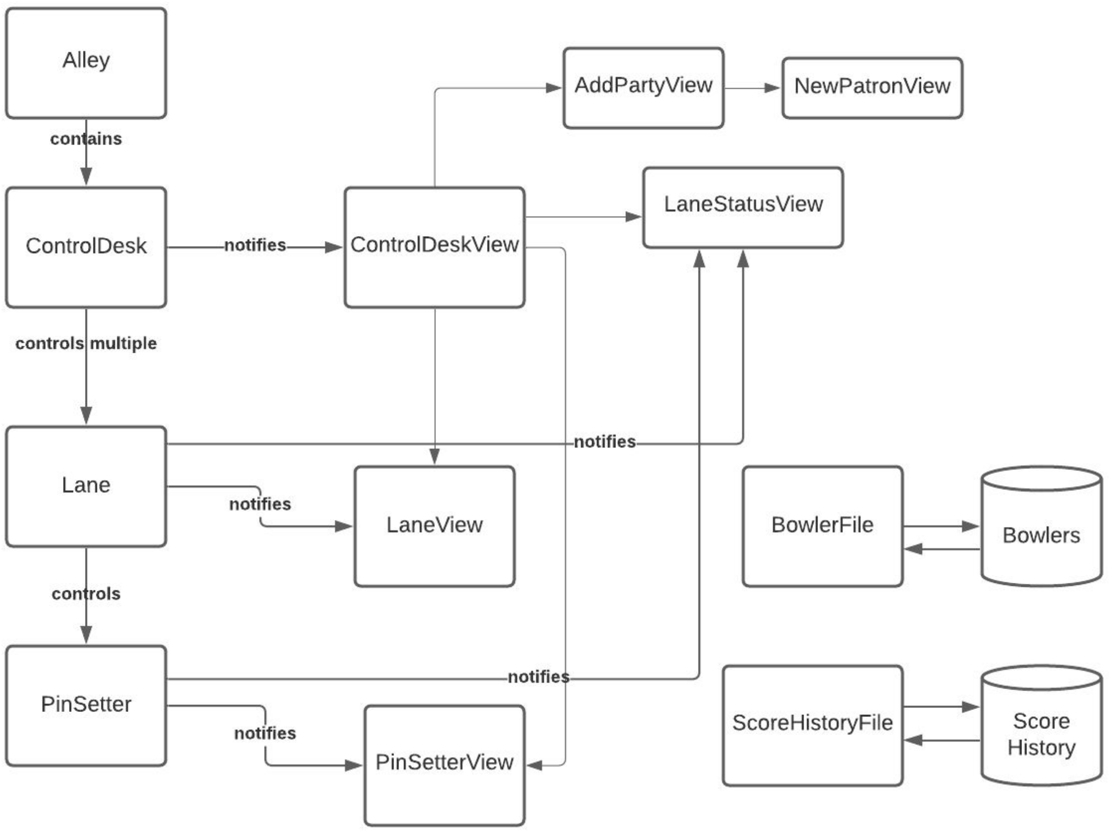


```
public class LaneEvent {

    private Party p;
    int frame;
    int ball;
    Bowler bowler;
    int[][] cumulScore;
    HashMap score;
    int index;
    int frameNum;
    int[] curScores;
    boolean mechProb;
```

<u>Design Strengths and Fidelity to the Design Documentation</u>

1. If we look at this piece of code as a black-box, it successfully delivers what has been asked by the attached design document, the code works as it should and is also fairly easy to run.

2. The code is fairly modular and hence easier to understand, there is a visible hierarchy of control and an overview like shown below can be easily interpreted from the given code.



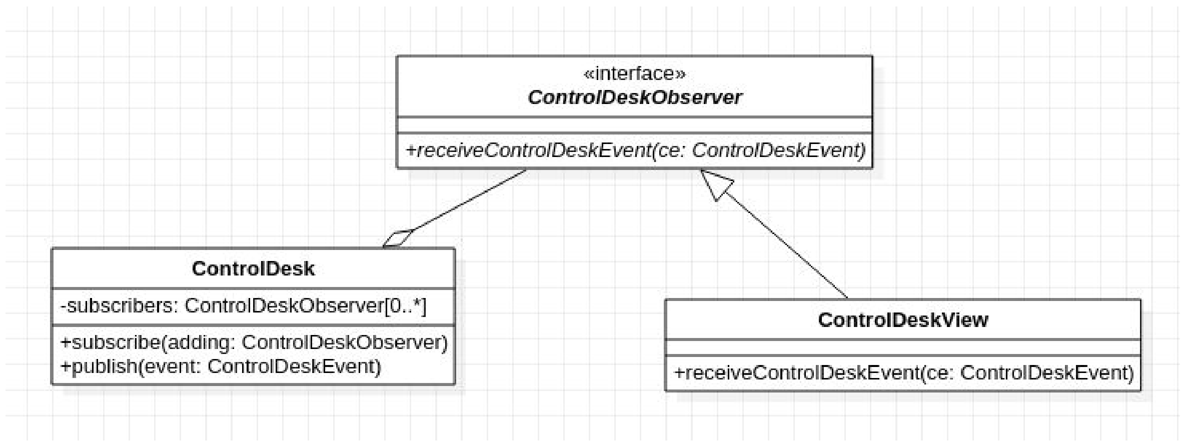
3. There is a decent amount of separation of concerns, the View classes are separate from the Business Logic Classes and both are separate from the Database handling classes.

4. The GUI is user-friendly and any new user should be able to explore the application easily.

5. Almost all of the class attributes follow the Principle of Least Knowledge and the classes do a good job at encapsulation.Out of the total 120 methods, only a few have cyclomatic complexity beyond the accepted threshold.

<u>Design Patterns Used</u>

1. Observer Pattern


«interface»
ControlDeskObserver
+receiveControlDeskEvent(ce: ControlDeskEvent)
ControlDesk
-subscribers: ControlDeskObserver[0..*]
+subscribe(adding: ControlDeskObserver)
+publish(event: ControlDeskEvent)
ControlDeskView
+receiveControlDeskEvent(ce: ControlDeskEvent)

There are several places where the Observer Pattern has been used in the given code. The ControlDeskView observes the ControlDesk so that it can be notified whenever there is a change in the current PartyQueue so that it can reflect the changes accordingly. Both LaneView and LaneStatusView observe Lane so that the GUI and the backend states are consistent with each other. And similarly PinsetterView observes the PinSetter.

2. MVC Pattern

   The code can be seen in the context of a Model-View-Controller application

Model Files	Controller Files	View Files
BowlerFile.java *ScoreHistoryFile.java*	*ControlDesk.java* *ControlDeskEvent.java* *ControlDeskObserver.java* *Lane.java* *LaneEvent.java* *LaneEventInterface.java* *LaneObserver.java* *LaneServer.java* *Party.java* *Pinsetter.java* *PinsetterEvent.java* *PinsetterObserver.java* *Queue.java* *Score.java* *Alley.java* *Bowler.java* *drive.java*	*AddPartyView.java* *ControlDeskView.java* *EndGamePrompt.java* *EndGameReport.java* *LaneStatusView.java* *LaneView.java* *NewPatronView.java* *PinSetterView.java* *PrintableText.java* *ScoreReport.java*

Refactored Design Analysis

Code Quality Comparison

<u>Using Metrics</u>

We will be using the following metrics to evaluate the current design and the refactored design. We majorly consider two types of metrics - class metrics and method metrics:

1. **Method Metrics**
    a. **Cyclomatic Complexity** - Cyclomatic complexity of a method is the quantitative measure of the number of linearly independent paths in it. It is used to indicate the complexity of a program. It is computed using the Control Flow Graph of the method. Lower value indicates good design.
    b. **Essential Complexity** - The degree to which a method contains unstructured constructs. Ideally, this metric should also have lower values.
    c. **Lines of Code -** Total number of lines of code in a method.

2. **Class Metrics**
    a. **Coupling Between Objects (CBO) -** Coupling between objects (CBO) is a count of the number of classes that are coupled to a particular class i.e. where the methods of one class call the methods or access the variables of the other. An increase of CBO indicates the reusability of a class will decrease. Thus, the CBO values for each class should be kept as low as possible.
    b. **Weighted Methods for Class (WMC) -** WMC measures the complexity of a class. It is usually calculated by adding the individual complexities of its methods. High value of WMC indicates the class is more complex than that of low values.

Work Distribution within the Team

Member	Role	Effort(in hours)
Vaibhav Garg		12
Akshay Goindani		
Hitesh Kumar		
Surekha Medapati		