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

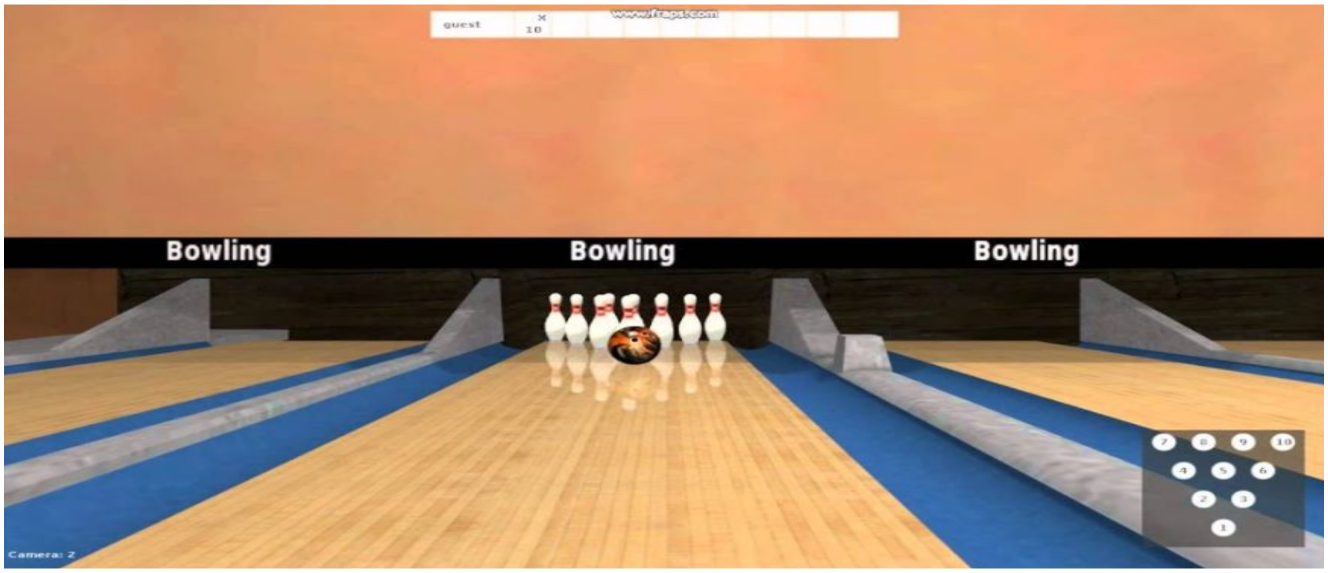
**Team Number:** 6

**Date of Submission:** 12th Feb'21

---

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

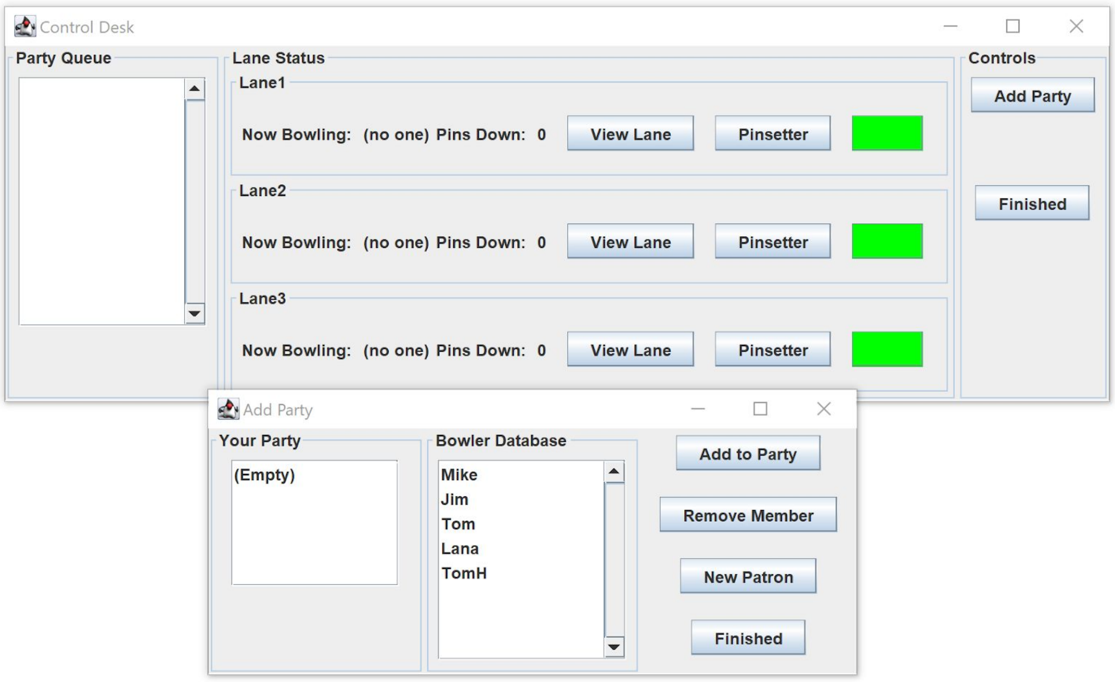
---



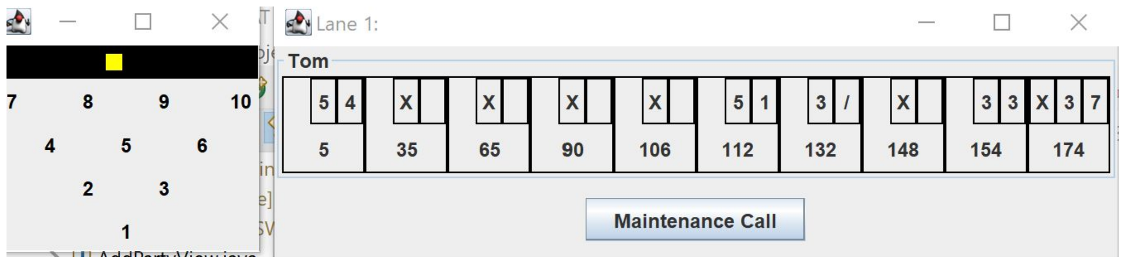
## Product Overview:

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 design features of the product are as follows:

**ControlDesk:** Handles the main party queue, looks over the party formation and lane allocation process. It also provides an interface to monitor the scores of any active lane.



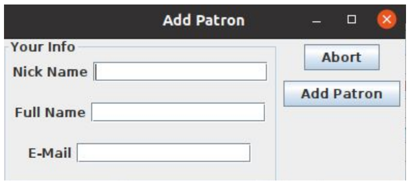
It has a dynamic display of the game where it shows the number of pins knocked and also the progress of the game(score).



**Pinsetter**: The pinsetter, 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 fixed maximum number of bowlers. The order in which members of the party check 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.



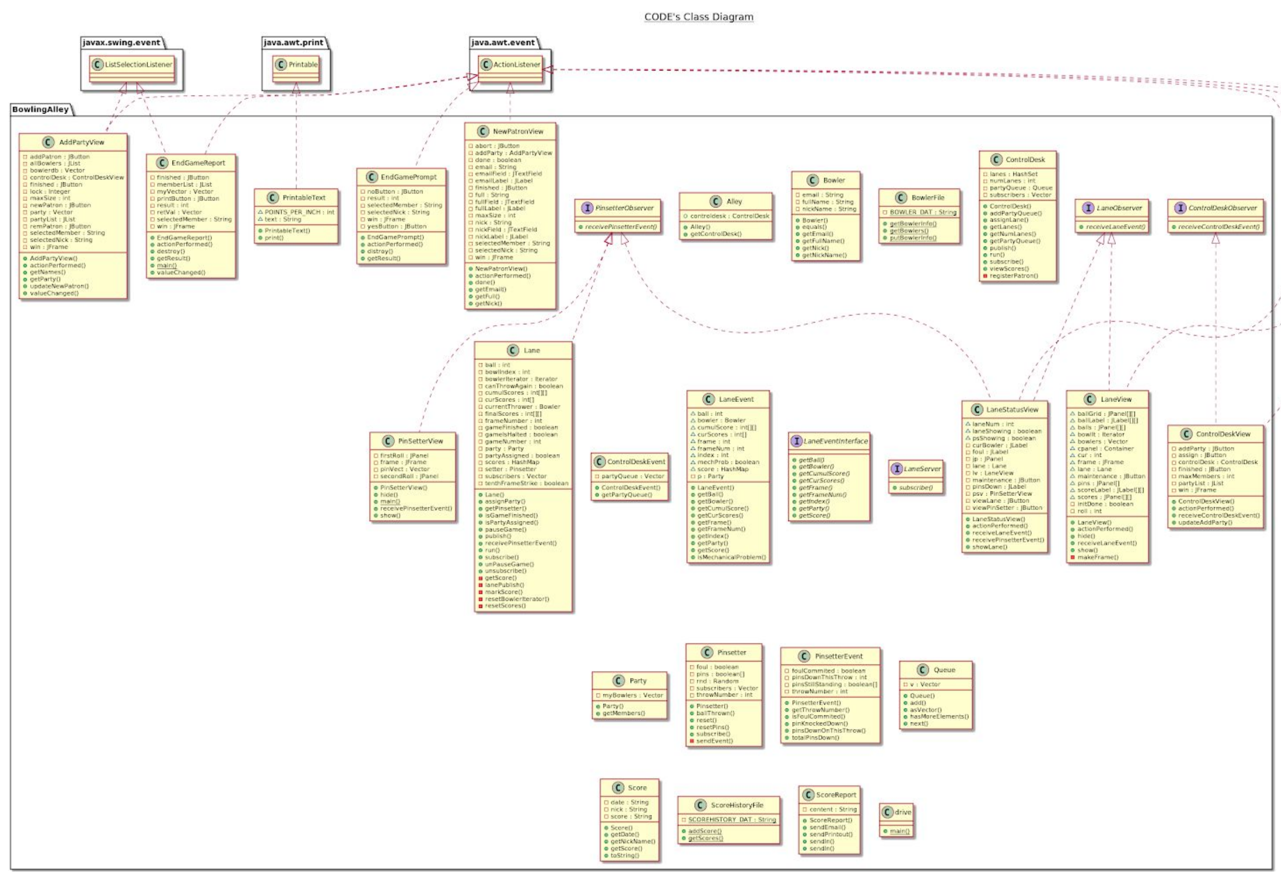
**View Scoreboard:** The scoreboard keeps track of the points gained by each player. It follows normal bowling score calculation technique as a strike is achieved when all the pins are knocked down on the first roll(score=10+pins dropped on next 2 turns), and a spare is achieved if all the pins are knocked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



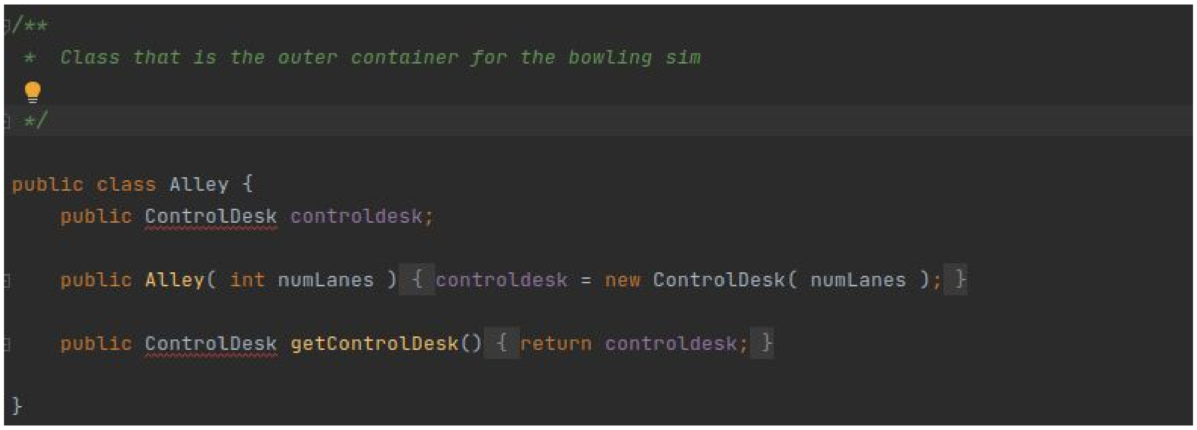
## Major Control Class Responsibilities

| **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. |

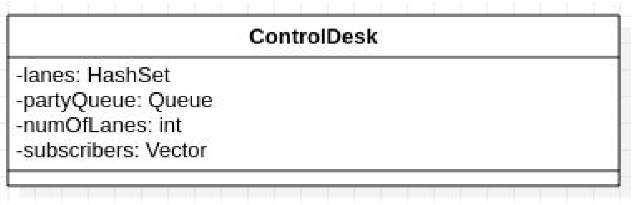
<u>Design Flaws</u>

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.



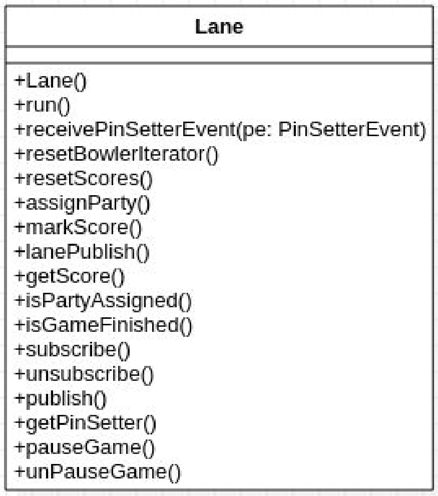
```
/**
 *  Class that is the outer container for the bowling sim
 *
 */

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.



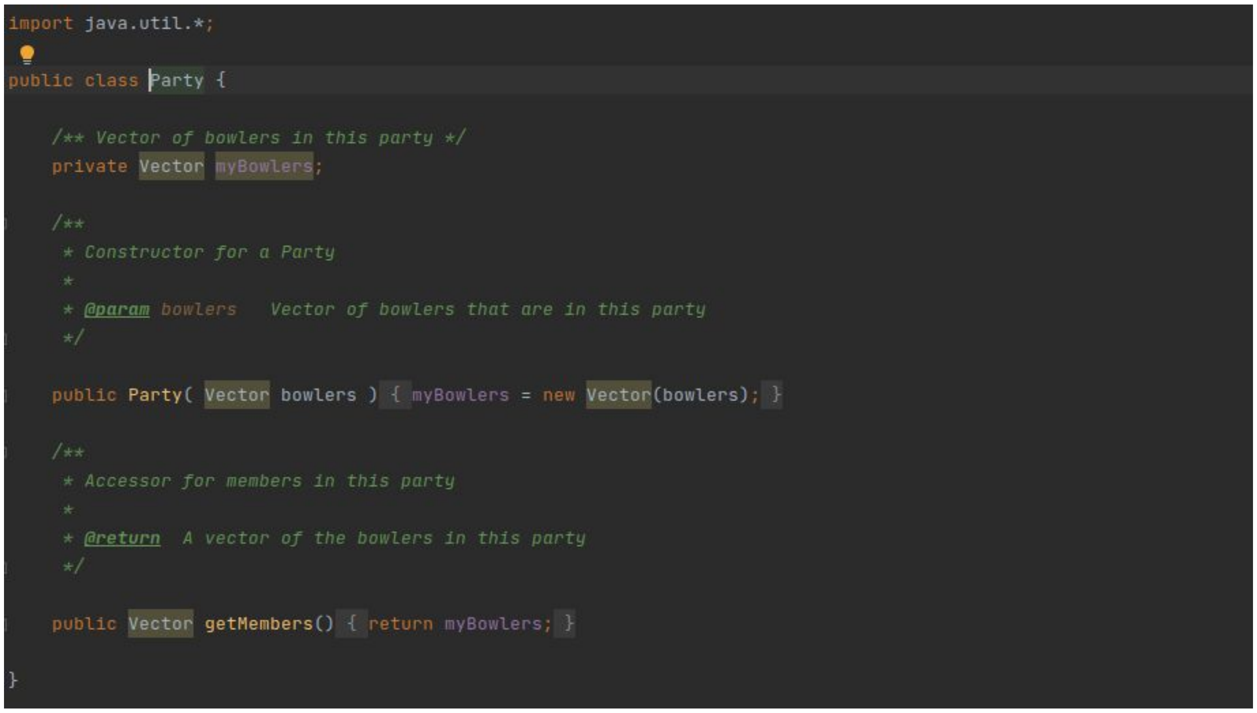
3. Both *run* and *getScore* methods of the Lane class have extremely high cyclomatic complexity and are also very difficult to read and understand.

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.

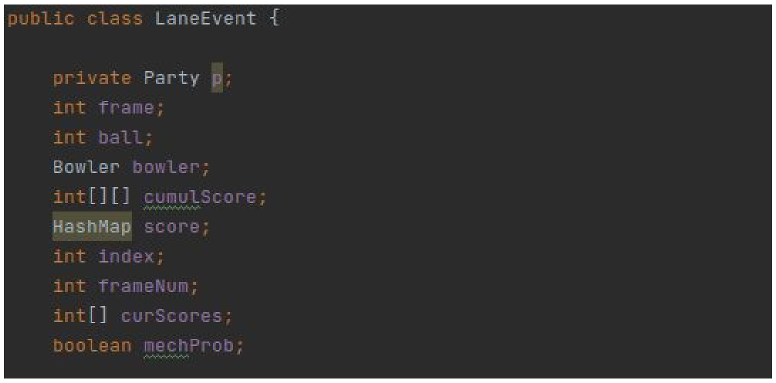


```java
import java.util.*;

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 public.



```
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>Code Smells</u>

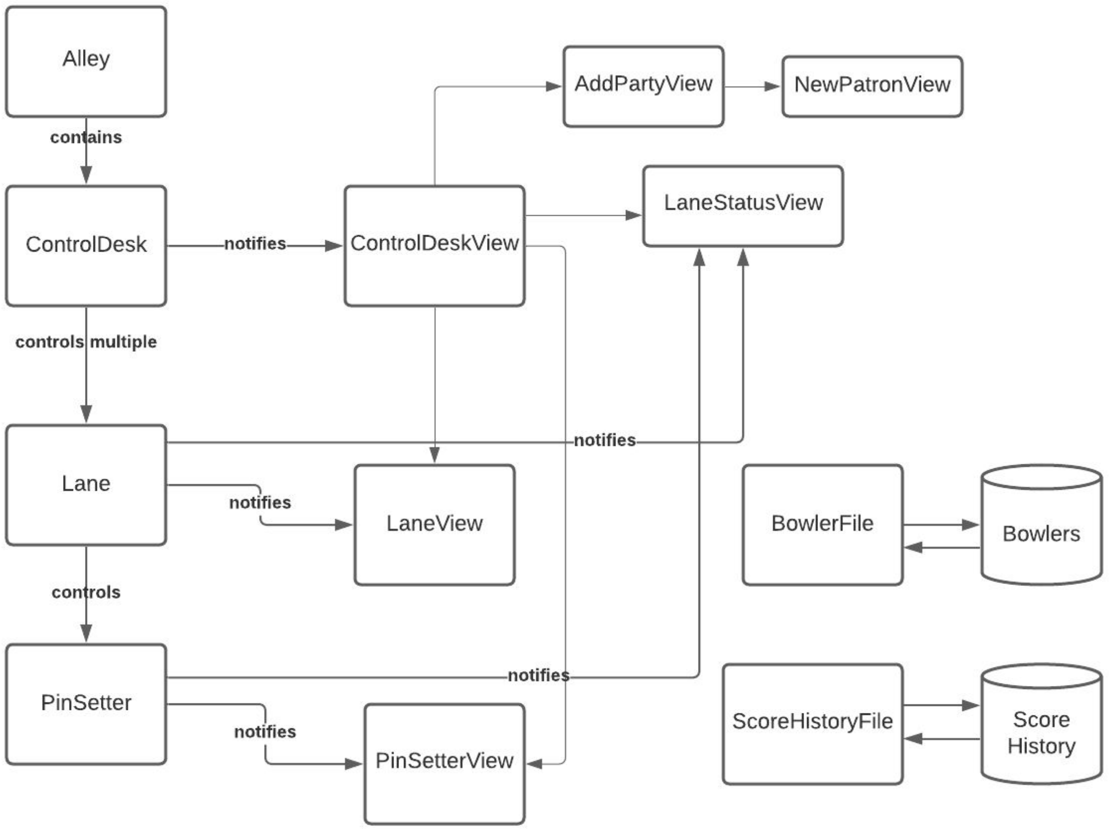
| **Java Class File** | **CodeSmell Description** |
|---|---|
| *AddPartyView* | ● Code Repetition of making Jbuttons<br>● Deprecation of methods show(), hide()<br>● actionPerformed had many nested if-else conditions which leads to High Complexity |
| *ControlDesk* | ● viewScores() is dead code<br>● It has extensively used methods like addpartyQueue() and getpartyQueue |
| *ControlDeskView* | ● Copy paste of Making Jbuttons and assignPanel button is not being used |
| *Lane* | ● Redundancy of methods like lanePublish() and publish were always used together<br>● Very Large Class of many methods and variables<br>● getScore(), receivePinsetterEvent(), run() have very high complexity due to more nested if-else conditions |
| *LaneStatusview* | ● actionPerformed() had repeated if conditions which leads to high conditional complexity |
| *LaneEvent* | ● Unused getFrame() and getCurScores() functions and data class is used for storage and getter methods only. No operations on data. It has a Long Parameter List. |
| *NewPatron* | ● Rewriting of Jbuttons on Jpanel |
| *PinsetterView* | ● Making Buttons Rewritten code |

<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. Naming of classes and methods were also done relevant to the action they performed. For view functionality the files are created separately and perform their actions.Thus it is easier to comprehend and understand what is the flow of code. Comments are nice with reasonably good naming conventions.

3. 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.



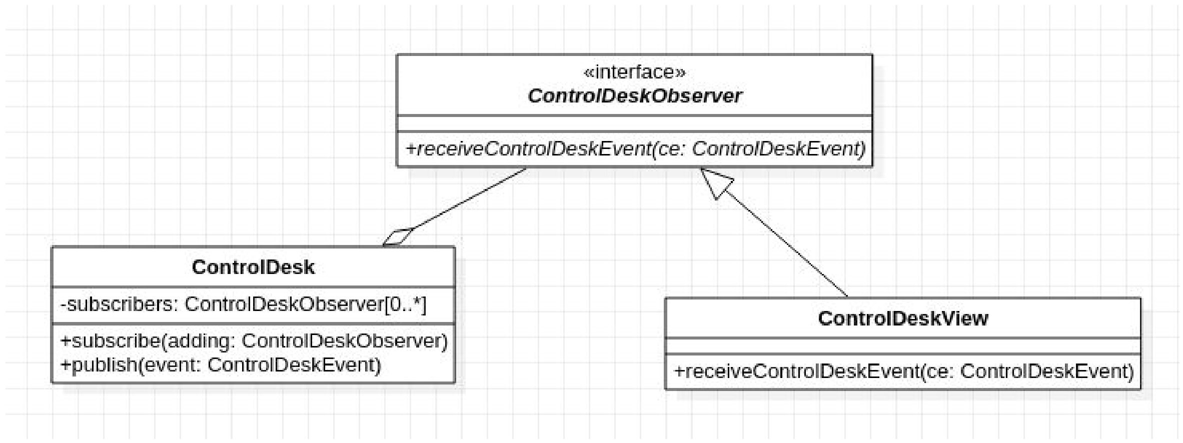
4. There is a decent amount of separation of concerns, the View classes are separate from the Business Logic Classes and both are separate from the Database handling classes.

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

6. 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.

Design Patterns Used

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.

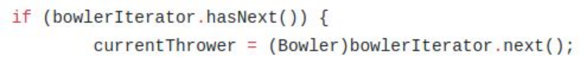
2. MVC Pattern

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

| Model Files | Controller Files | View Files |
|---|---|---|

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

3. Iterator Pattern



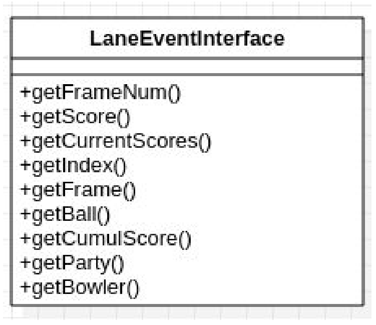
```
if (bowlerIterator.hasNext()) {
        currentThrower = (Bowler)bowlerIterator.next();
```

It has been used to iterate over the bowlers within a party and abstracts out the details of how exactly the Bowler access is happening and other internals.

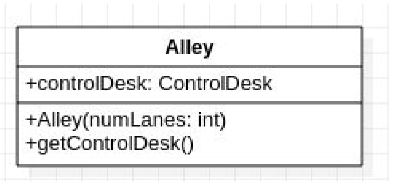
## Refactored Design Analysis

Code Changes

1. **Dead Code Removal:** Deleted the LaneEventInterface class as it was not being implemented/realized by any class.

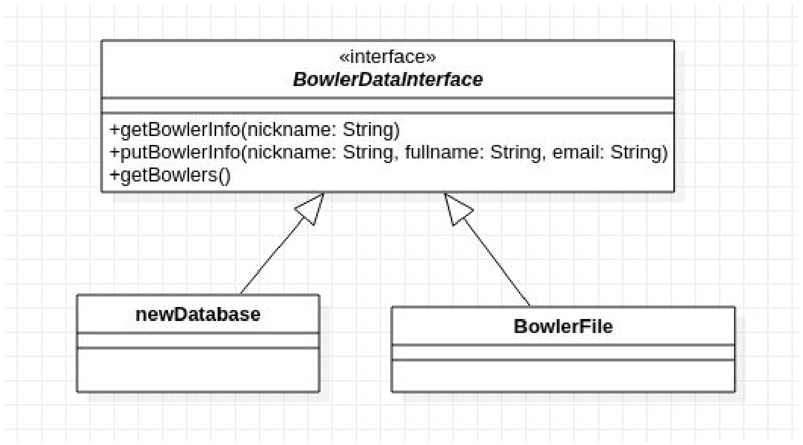


2. **Unnecessary Abstraction:** Alley Class was behaving like a Container class, so removed it and Drive class now interacts directly with the ControlDesk class.



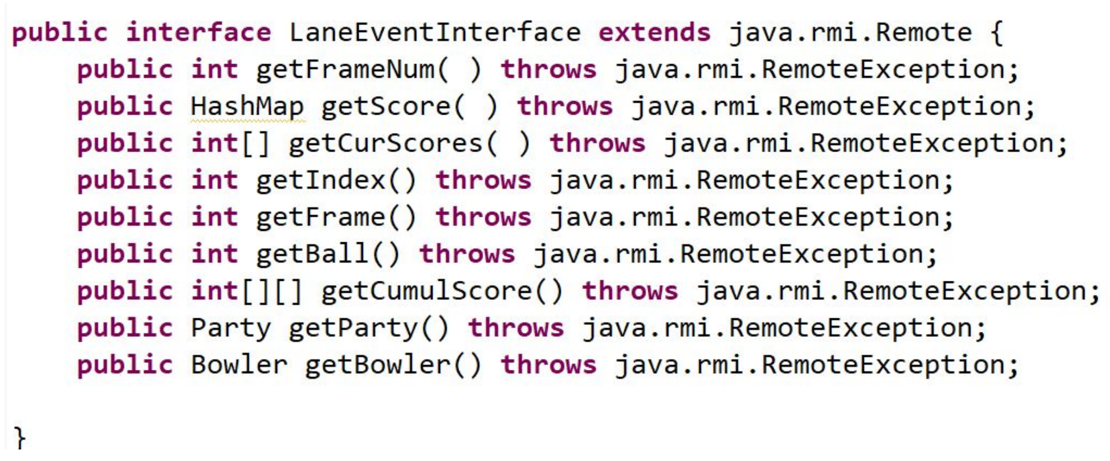
3. **Better Encapsulation:** Maximum members per party should be an attribute of ControlDesk and not ControlDeskView, so changed accordingly.

4. **Database Abstraction:** For loose coupling, introduced two new interfaces which can be implemented in case the application needs to be integrated with a newer database.


«interface»
BowlerDataInterface
+getBowlerInfo(nickname: String)
+putBowlerInfo(nickname: String, fullname: String, email: String)
+getBowlers()
newDatabase
BowlerFile

5. **Unused Variables:** Some variables which are declared initially are not used later in the code. Example selectedNick, selectMember in NewPatron.java. They can also point to a lack of proper design when not used. Unused variables are removed in every java file.

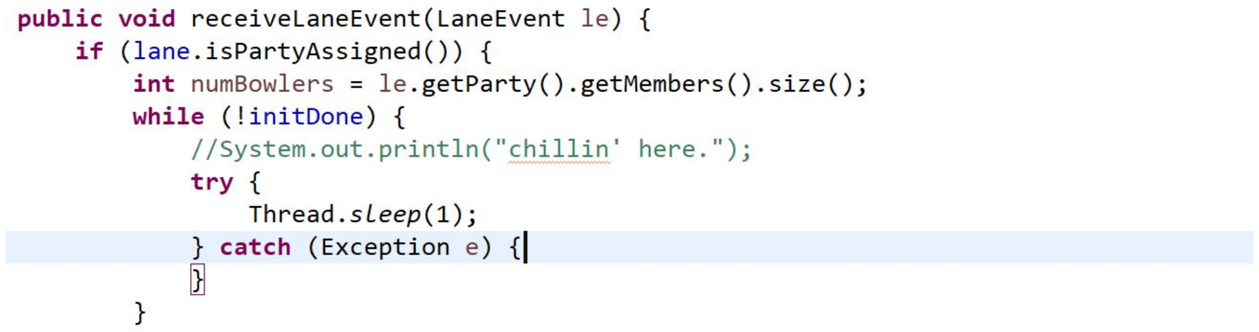
6. **Public Modifier in Interfaces**: All methods in an interface are already public and abstract so there was no need to make them public explicitly. Interface's public modifier was used with many functions.



```
public interface LaneEventInterface extends java.rmi.Remote {
    public int getFrameNum( ) throws java.rmi.RemoteException;
    public HashMap getScore( ) throws java.rmi.RemoteException;
    public int[] getCurScores( ) throws java.rmi.RemoteException;
    public int getIndex() throws java.rmi.RemoteException;
    public int getFrame() throws java.rmi.RemoteException;
    public int getBall() throws java.rmi.RemoteException;
    public int[][] getCumulScore() throws java.rmi.RemoteException;
    public Party getParty() throws java.rmi.RemoteException;
    public Bowler getBowler() throws java.rmi.RemoteException;

}
```

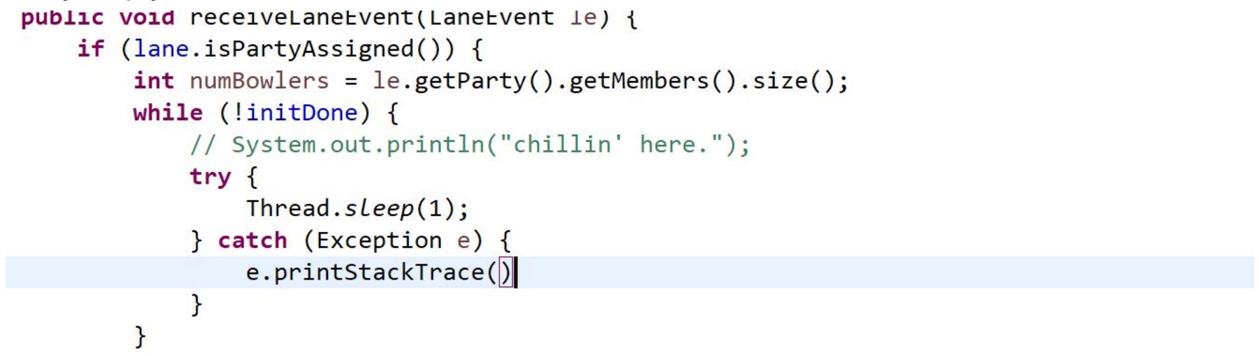
Therefore, we removed the public modifiers from Interface files.

7. **Empty Catch Statements:** In many files Catch statements were empty. If catch code would not print anything it would be difficult to debug. So appropriate errors were printed.



```java
public void receiveLaneEvent(LaneEvent le) {
    if (lane.isPartyAssigned()) {
        int numBowlers = le.getParty().getMembers().size();
        while (!initDone) {
            //System.out.println("chillin' here.");
            try {
                Thread.sleep(1);
            } catch (Exception e) {
            }
        }
```

Every empty catch statement has been refactored.



```java
public void receiveLaneEvent(LaneEvent le) {
    if (lane.isPartyAssigned()) {
        int numBowlers = le.getParty().getMembers().size();
        while (!initDone) {
            // System.out.println("chillin' here.");
            try {
                Thread.sleep(1);
            } catch (Exception e) {
                e.printStackTrace()
            }
        }
```

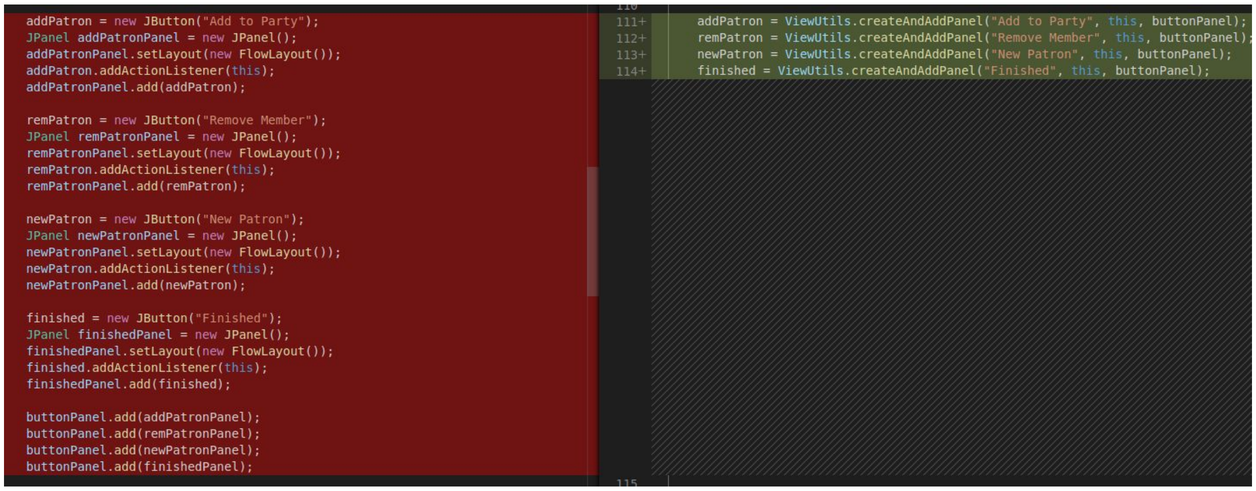
8. **Unnecessary Imports:** Unused and useless imports affects the code's readability. Example java.text.* which is never used in the class. Removed unused imports in every file.

9. **Deprecated warnings:** The deprecated warnings are removed wherever possible. Some examples are show(), hide(), new Integer etc., are modified as setVisible(true), setVisible(false), Integer.valueOf().

10. **Method typo mistakes:** In EndGamePrompt has a method named *"distroy*" which is a typo. Renamed to destroy in every file. Method typo mistakes are fixed.

11. **Redundant casting to various fields:** Unnecessary casting expressions make the code harder  to understand. So removed redundant casting in the codebase.

12. **Code Repetition:** The main reason for creation for the duplicate code is copy and paste programming. It will increase Lines of Code(LoC). In AddPartview the same code is repeated for making buttons, so a static ViewUtils class was created to have static functions for these purposes. One can see below how drastically the Lines of Code have decreased.



```
addPatron = new JButton("Add to Party");
JPanel addPatronPanel = new JPanel();
addPatronPanel.setLayout(new FlowLayout());
addPatron.addActionListener(this);
addPatronPanel.add(addPatron);

remPatron = new JButton("Remove Member");
JPanel remPatronPanel = new JPanel();
remPatronPanel.setLayout(new FlowLayout());
remPatron.addActionListener(this);
remPatronPanel.add(remPatron);

newPatron = new JButton("New Patron");
JPanel newPatronPanel = new JPanel();
newPatronPanel.setLayout(new FlowLayout());
newPatron.addActionListener(this);
newPatronPanel.add(newPatron);

finished = new JButton("Finished");
JPanel finishedPanel = new JPanel();
finishedPanel.setLayout(new FlowLayout());
finished.addActionListener(this);
finishedPanel.add(finished);

buttonPanel.add(addPatronPanel);
buttonPanel.add(remPatronPanel);
buttonPanel.add(newPatronPanel);
buttonPanel.add(finishedPanel);
```

```
addPatron = ViewUtils.createAndAddPanel("Add to Party", this, buttonPanel);
remPatron = ViewUtils.createAndAddPanel("Remove Member", this, buttonPanel);
newPatron = ViewUtils.createAndAddPanel("New Patron", this, buttonPanel);
finished = ViewUtils.createAndAddPanel("Finished", this, buttonPanel);
```

13. **Long Methods:** Large methods in Lane.java, for ex: run() and getscore() each method is longer than 100 lines consisting mostly of if-else conditional blocks. It makes code too difficult to read as well as troubleshooting the code in the long run will be hard. Cyclomatic complexity analysis shows the complexity of each class. We refactored these classes by modularizing the method into two or more methods, which improves the code quality to a great extent.

| **Method Name** | **Cyclomatic Complexity** | **Essential Complexity** | **New Complexity** |
|---|---|---|---|
| *Lane.getScore* | 38 | 5 | 8 |
| *Lane.run* | 19 | 1 | 3 |

14. **Conditional complexity:** Reduced the number of lines of many conditional blocks by simplifying the multiple nested if statements into one. Many if statements were always true, we corrected them. Refactored the unreachable statements.

**Before**

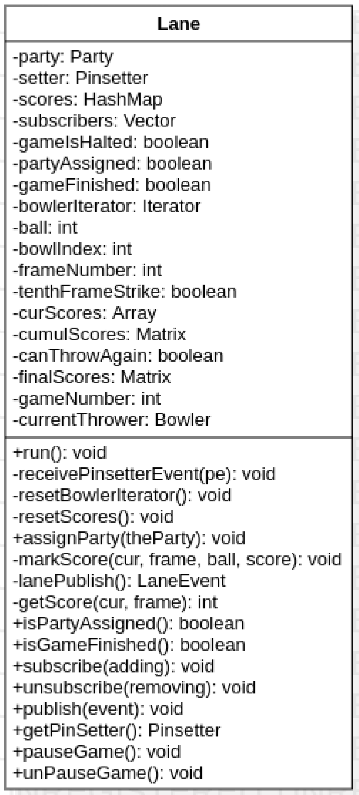
```
if (e.getSource().equals(viewLane)) {
      if ( lane.isPartyAssigned() ) {
            if ( laneShowing == false ) {
                  lv.show();
                  laneShowing=true;
            } else if ( laneShowing == true ) {
                  lv.hide();
                  laneShowing=false;
            }
      }
}
```

**After**

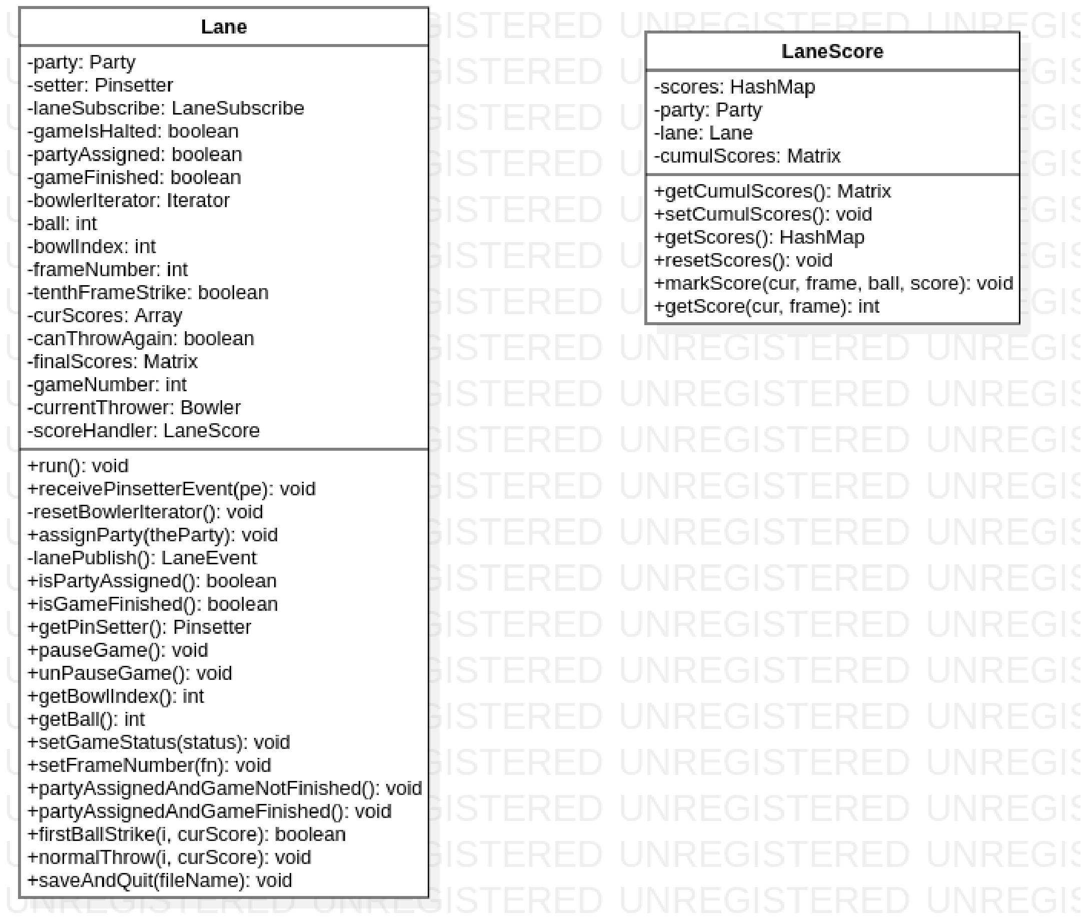
```
if(e.getSource().equals(viewLane) && (lane.isPartyAssigned())){
    if (!laneShowing ) {
        lv.show();
        laneShowing = true;
    } else {
        lv.hide();
        laneShowing = false;
    }
}
```

15. **Indecent Exposure:** Many classes that unnecessarily expose their internals. We refactored classes to minimize their public surface. This is done by making the variables **private.**

16. **Low Cohesion:** Initially the Lane class had the methods to calculate score for a bowler. This leads to low cohesion, in order to increase cohesion, we separated the methods calculating the score from the Lane class and made another class LaneScore which has been given the responsibility of calculating scores. The following class diagrams show the difference between original Lane class and refactored Lane class.



The above class diagram represents the Lane Class in the original code. After refactoring, the class diagram is as follows:

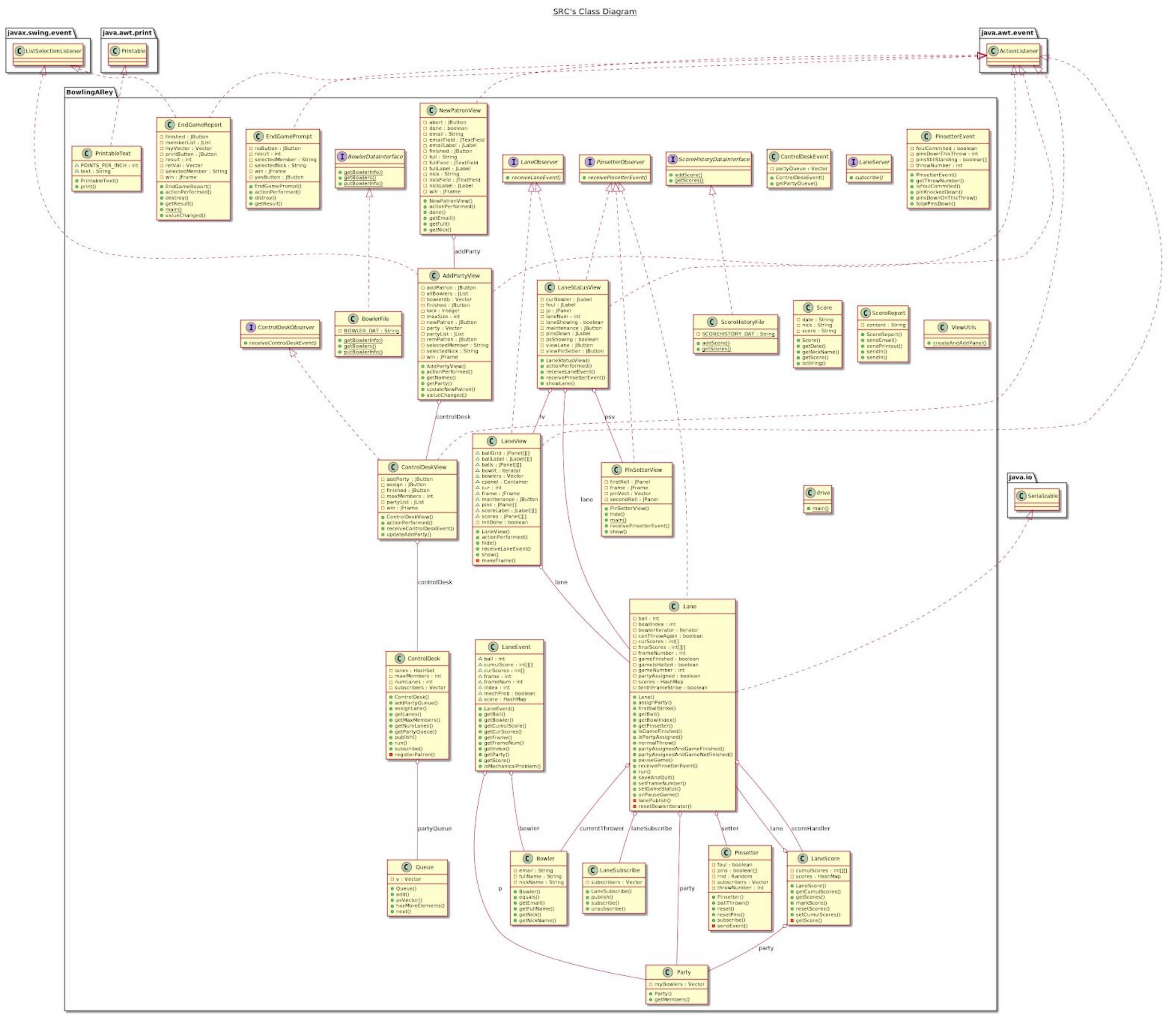


Lane
-party: Party
-setter: Pinsetter
-laneSubscribe: LaneSubscribe
-gameIsHalted: boolean
-partyAssigned: boolean
-gameFinished: boolean
-bowlerIterator: Iterator
-ball: int
-bowlIndex: int
-frameNumber: int
-tenthFrameStrike: boolean
-curScores: Array
-canThrowAgain: boolean
-finalScores: Matrix
-gameNumber: int
-currentThrower: Bowler
-scoreHandler: LaneScore
+run(): void
+receivePinsetterEvent(pe): void
-resetBowlerIterator(): void
+assignParty(theParty): void
-lanePublish(): LaneEvent
+isPartyAssigned(): boolean
+isGameFinished(): boolean
+getPinSetter(): Pinsetter
+pauseGame(): void
+unPauseGame(): void
+getBowlIndex(): int
+getBall(): int
+setGameStatus(status): void
+setFrameNumber(fn): void
+partyAssignedAndGameNotFinished(): void
+partyAssignedAndGameFinished(): void
+firstBallStrike(i, curScore): boolean
+normalThrow(i, curScore): void
+saveAndQuit(fileName): void
LaneScore
-scores: HashMap
-party: Party
-lane: Lane
-cumulScores: Matrix
+getCumulScores(): Matrix
+setCumulScores(): void
+getScores(): HashMap
+resetScores(): void
+markScore(cur, frame, ball, score): void
+getScore(cur, frame): int

In the refactored design, the LaneScore has the private attributes which are required for score computation. Those attributes are also required by the Lane class, therefore we have created getters and setters for those attributes for e.g., attribute *scores*.

## Final Class Diagram

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

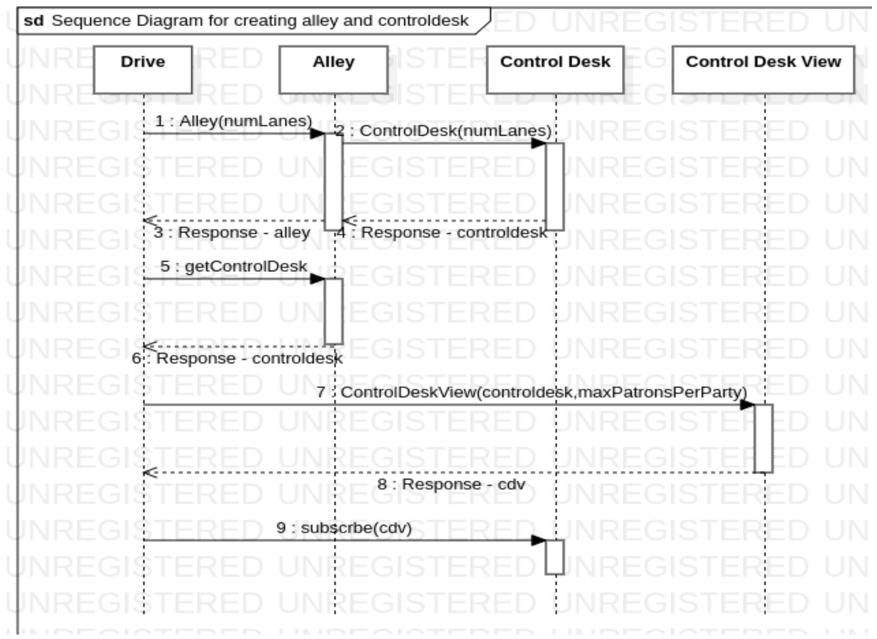


For refactoring code several parameters have to be taken into consideration to strike the right balance. Modules should be as **independent** as possible from other modules, so if we modify a module it doesn't heavily impact other modules. We started off with 29 classes and we observed that there were more classes required eg: BowlerDataInterface, ScoreHistoryDataInterface are added to ensure **low coupling**. **Cohesion** is closely associated with making sure that a class is designed with a single, well-focused purpose. For example DB **abstraction** class is only focused on handling data functionality adhering to high cohesion. Highly cohesive classes are much easier to maintain and less frequently changed. We observed that there tends to be higher lack of cohesion for classes with greater number of methods which decreases code readability.

The main reason for creating numerous classes is that each class can **encapsulate** a particular functionality and can be responsible for all the tasks related to it. This also helped to ensure **separation of concerns**. Most of the attributes in the existing classes were already private members of the class. We used public variables only when required and necessary for better **information hiding** and following the **Law of Demeter**.

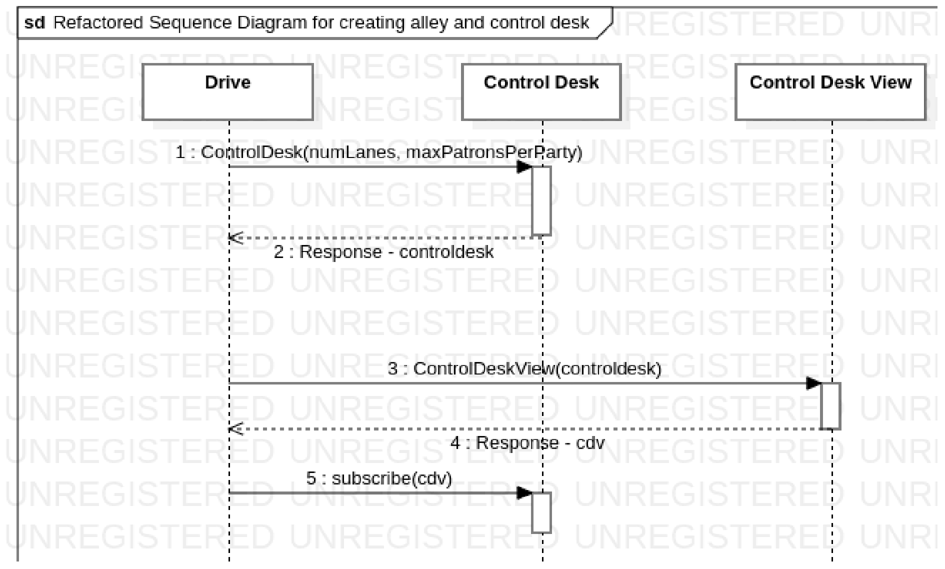
## Code Quality Comparison

Sequence Diagram for Starting the App *in the original code*


sd Sequence Diagram for creating alley and controldesk
Drive
Alley
Control Desk
Control Desk View
1 : Alley(numLanes)
2 : ControlDesk(numLanes)
3 : Response - alley
4 : Response - controldesk
5 : getControlDesk
6 : Response - controldesk
7 : ControlDeskView(controldesk,maxPatronsPerParty)
8 : Response - cdv
9 : subscrbe(cdv)

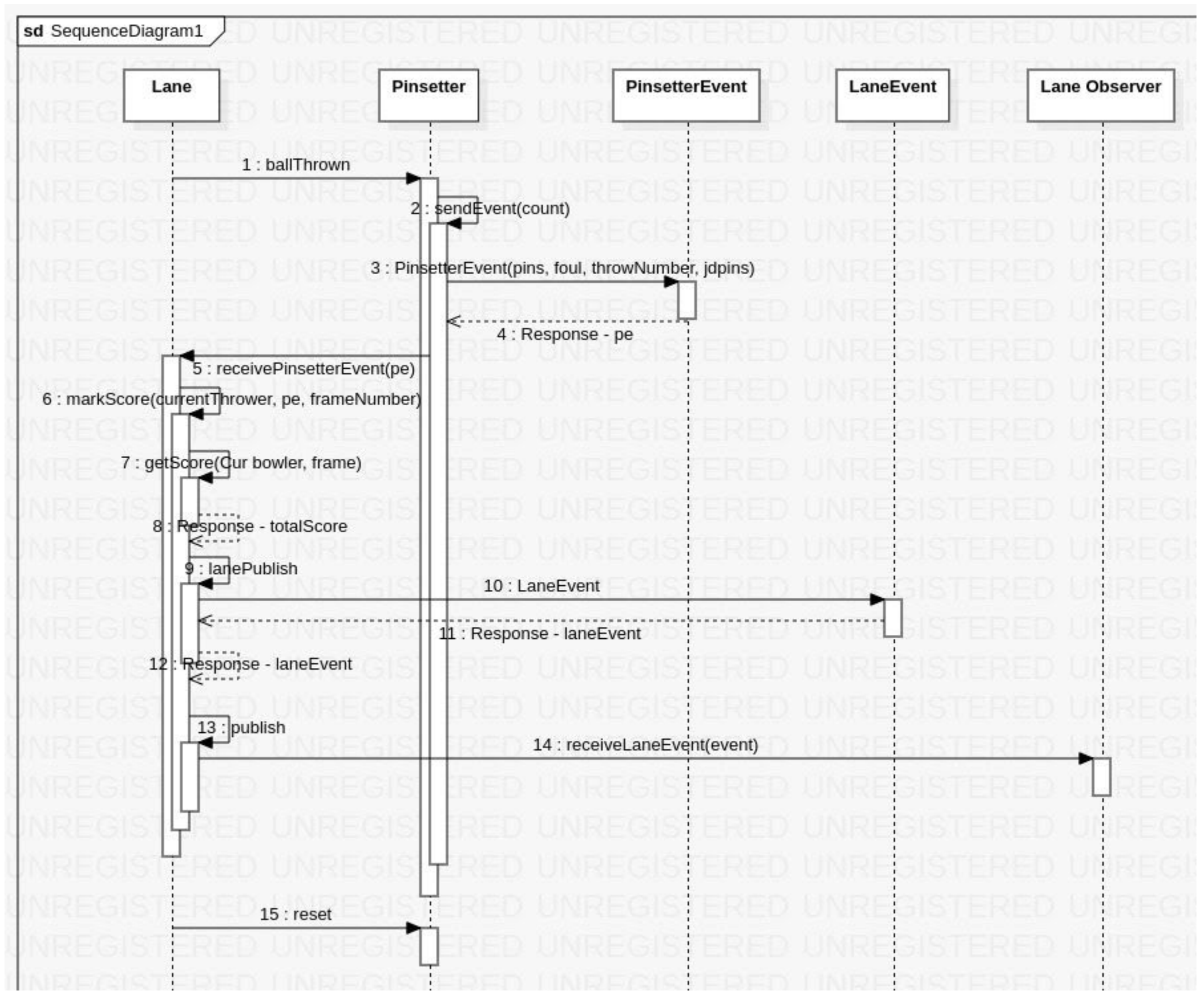
The above sequence diagram shows that, first the drive class constructs an Alley object which in turn constructs a ControlDesk object. The drive class then gets the ControlDesk from Alley. After that a ControlDeskView is created and the drive class requests the ControlDesk to add ControlDeskView as a subscriber.

Sequence Diagram for Starting the App *in the refactored code*


sd Refactored Sequence Diagram for creating alley and control desk
Drive
Control Desk
Control Desk View
1 : ControlDesk(numLanes, maxPatronsPerParty)
2 : Response - controldesk
3 : ControlDeskView(controldesk)
4 : Response - cdv
5 : subscribe(cdv)

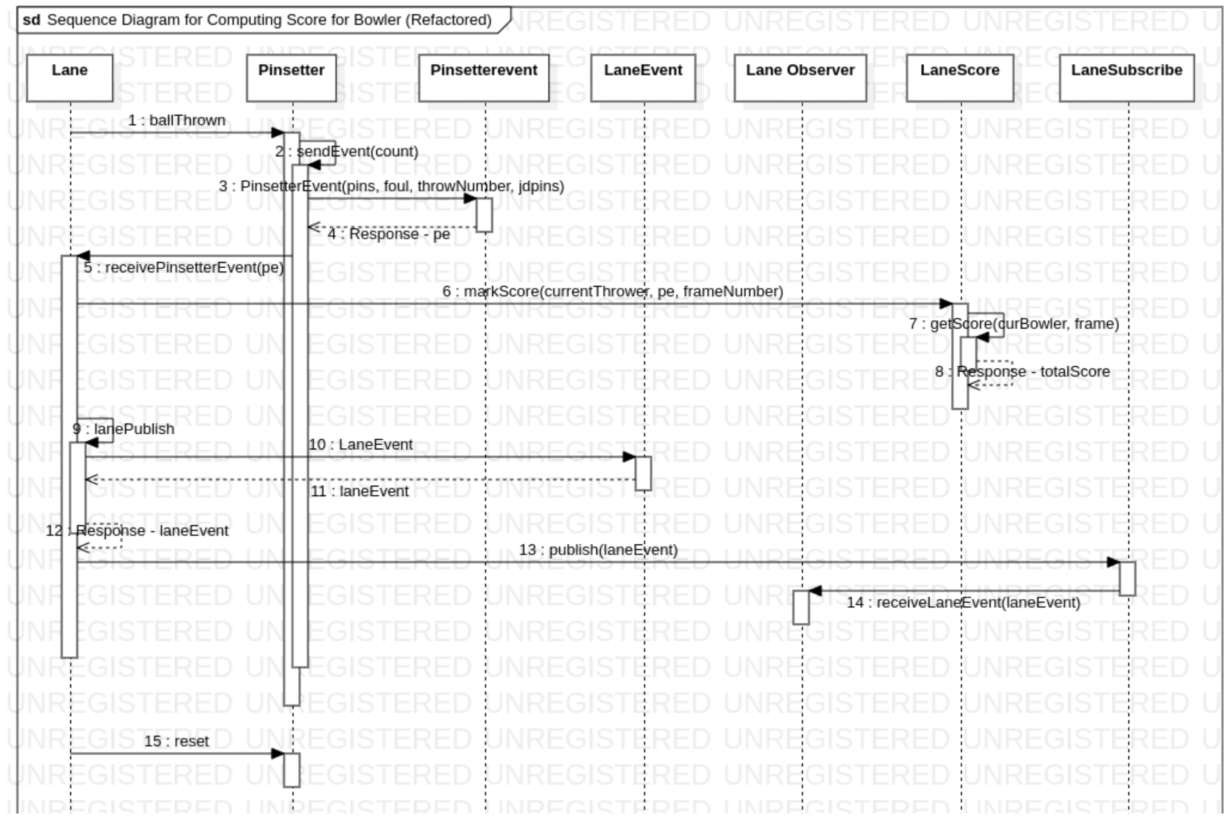
The above sequence diagram represents how refactoring has simplified starting of the application. Here, we can see that the Alley class is removed, which helps to decrease the number of parameters, function calls and also has better encapsulation. The drive class directly sends a request to the ControlDesk without creating an Alley.

<u>Sequence Diagram for Score Computation of a Bowler in the original code</u>


sd SequenceDiagram1
Lane
Pinsetter
PinsetterEvent
LaneEvent
Lane Observer
1 : ballThrown
2 : sendEvent(count)
3 : PinsetterEvent(pins, foul, throwNumber, jdpins)
4 : Response - pe
5 : receivePinsetterEvent(pe)
6 : markScore(currentThrower, pe, frameNumber)
7 : getScore(Cur bowler, frame)
8 : Response - totalScore
9 : lanePublish
10 : LaneEvent
11 : Response - laneEvent
12 : Response - laneEvent
13 : publish
14 : receiveLaneEvent(event)
15 : reset

The above diagram shows the sequence diagram for the score computation of a bowler in the original code. Here, we can see that the Lane class is doing all the score computation, for e.g., the markScore method is present in the Lane class, which calls the method getScore which is also present in the Lane class. Since, the lane class is providing so many functionalities which leads to low cohesion.

Sequence Diagram for Score Computation of a Bowler in the refactored code


sd Sequence Diagram for Computing Score for Bowler (Refactored)
Lane
Pinsetter
Pinsetterevent
LaneEvent
Lane Observer
LaneScore
LaneSubscribe
1 : ballThrown
2 : sendEvent(count)
3 : PinsetterEvent(pins, foul, throwNumber, jdpins)
4 : Response - pe
5 : receivePinsetterEvent(pe)
6 : markScore(currentThrower, pe, frameNumber)
7 : getScore(curBowler, frame)
8 : Response - totalScore
9 : lanePublish
10 : LaneEvent
11 : laneEvent
12 : Response - laneEvent
13 : publish(laneEvent)
14 : receiveLaneEvent(laneEvent)
15 : reset

The above diagram shows the sequence diagram of the score computation of a bowler in the refactored code. In order to tackle the low cohesion problem, we created a new class LaneScore which computes the score for a bowler. In the refactored design, all the score computation is done by LaneScore class. This is also evident from the sequence diagram, where the Lane class requests (Relation 6) the LaneScore class to calculate the score.

Using Metrics

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.

Current Design metrics major results

| **Method Name** | **Cyclomatic Complexity** | **Essential Complexity** | **Lines of Code** |
|---|---|---|---|
| *AddPartyView.actionPerformed* | 11 | 1 | 27 |
| *Lane.getScore* | 38 | 5 | 114 |
| *Lane.receivePinsetterEvent* | 12 | 1 | 42 |
| *Lane.run* | 19 | 1 | 80 |
| *LaneStatusView.actionPerformed* | 11 | 1 | 30 |
| *LaneView.receiveLaneEvent* | 19 | 1 | 66 |
| *AddPartyView.AddPartyView* | 2 | 1 | 85 |

| **Class Name** | **CBO** | **WMC** |
|---|---|---|
| *AddPartyView* | 4 | 17 |
| *ControlDesk* | 10 | 18 |
| *Lane* | 15 | 72 |
| *EndGameReport* | 3 | 11 |
| *LaneEvent* | 6 | 11 |

The values above clearly indicate that there is definitely some amount of refactoring required in the given codebase as in a few cases the values are way above the recommended thresholds.

These metric values also helped a great deal in identifying the key places where refactoring was required. For example, it is clear that the Lane class in the original design is very less cohesive and highly coupled due to the high complexity and CBO values, so our focus was to simplify it aggressively.

Final Refactored Design metrics major results

| Method Name | Cyclomatic Complexity | Essential Complexity | Lines of Code |
|---|---|---|---|
| *AddPartyView.actionPerformed* | 11 | 1 | 27 |
| *LaneScore.getScore* | 12 | 5 | 37 |
| *Lane.receivePinsetterEvent* | 12 | 1 | 42 |
| *Lane.run* | 7 | 1 | 16 |
| *LaneStatusView.actionPerformed* | 11 | 1 | 30 |
| *LaneView.receiveLaneEvent* | 19 | 1 | 62 |
| *AddPartyView.AddPartyView* | 2 | 1 | 61 |

| Class Name | CBO | WMC |
|---|---|---|
| *AddPartyView* | 5 | 17 |
| *ControlDesk* | 9 | 18 |
| *Lane* | 16 | 62 |
| *EndGameReport* | 3 | 11 |
| *LaneEvent* | 7 | 11 |

Refactoring the code vastly improved complexity values for some methods and also improved decently in the LoC aspect but at the same time it led to an increase in CBO values due to the introduction of new classes and interfaces. For example, the CBO for Lane class increased but its associated complexity values decreased. This shows how there is not one perfect design and trade-offs need to be considered while refactoring a sufficiently large codebase. It might even be impossible to have a design which ends up improving all the metrics so one needs to prioritize on the metrics important in the current context.

## Work Distribution

| Member | Role | Effort(in hours) |
|---|---|---|
| **Vaibhav Garg** | ● Worked on original code analysis<br>● Application-level refactoring changes<br>● Final document structuring | 18 |
| **Akshay Goindani** | ● Worked on calculating metrics<br>● Created Sequence Diagrams for code comparison<br>● Refactored Lane Class | 18 |
| **Hitesh Kumar** | ● Refactoring to reduce cyclomatic complexity<br>● Reducing number of methods per class<br>● Refactoring to get rid of redundant code and indecent exposure of variables. | 18 |
| **Surekha Medapati** | ● Identifying Code smells<br>● Refactoring and cleaning code<br>● Design Analysis and Documentation | 18 |