Design Document- Unit 2 project

Bowling Alley Simulation Enhancement

Team: 31

Submission date : 22/03/2022

Team Members

SNo.	Team Member	Roll no.	Effort (hours)	Role
1.	Aditya Vishnu Tiwari	2021202029	12	Working on the code extensiblity and metric analysis, Documentation
2.	Aniket Vinod Chandekar	2021204001	12	Implementing user input for game configuration, Documentation
3.	Krishnapriya Panicker	2020202020	12	Worked on implementing a tie breaking system in the game between the winner and the runner up, Documentation
4.	Sandeep Deva Misra	2021202026	12	Adding database layer, Implement penalty for gutters, Documentation
5.	Shambhavi Ojha	2021204011	12	Making UI interactive, implementing emoticon as per score, Documentation

I. Introduction:

This document goes over the enhancement specifications for the Bowling Management Alley Program. The software uses the MVC approach and is developed in Java. It is a Bowling Alley simulator in which the system replicates a control panel with numerous features such as adding new players, arranging bowling parties, allocating lanes, keeping scores, displaying pinsetter images, displaying lane status and score, and so on.

We assessed the initially given code and its various metrics, such as C3, lines of code, and afterwards refactored the code to improve these metrics in the Unit-1 project. In the Unit-2 project, we attempt to enhance the working and functionalities of our unit-1 refactored codebase and make it as per the unit-2 project needs.

Each of the new requirements that have been implemented has been thoroughly discussed in the report's subsequent sections.

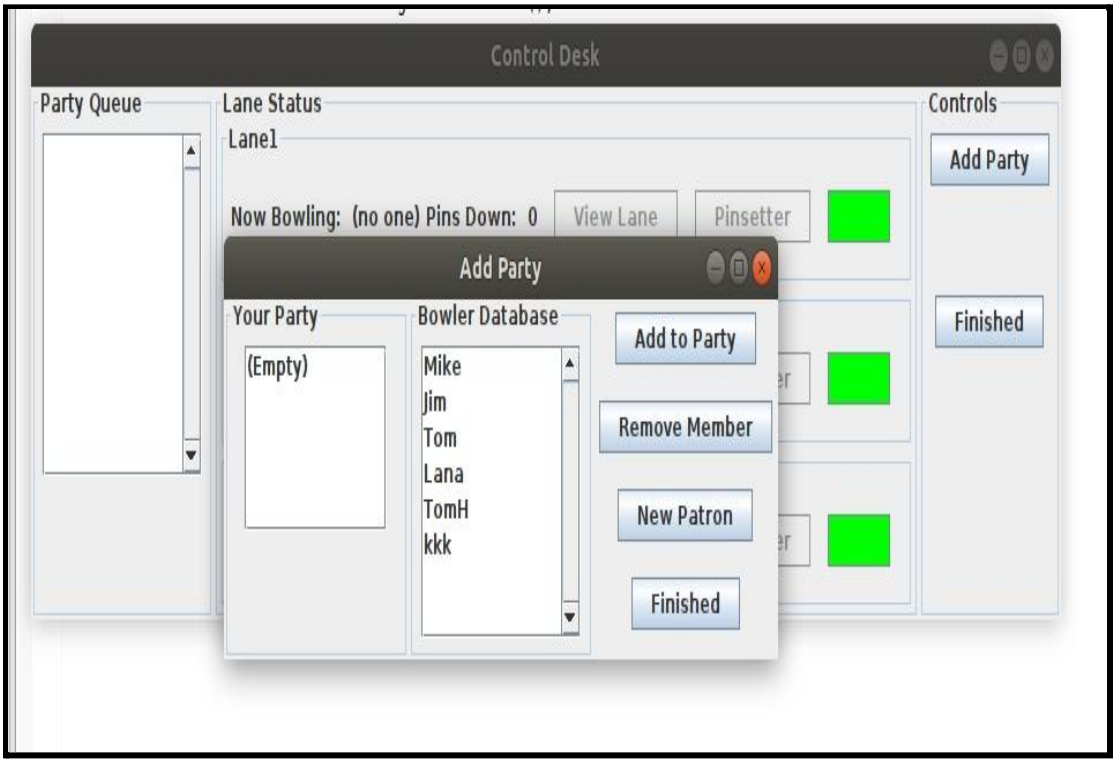
II. Brief Overview of the original system and codebase:

The Bowling Alley Simulation is a bowling simulation game. It is written entirely in Java and employs the MVC framework.

The game allows players to play bowling in a virtual environment. The game's basic features are as follows:

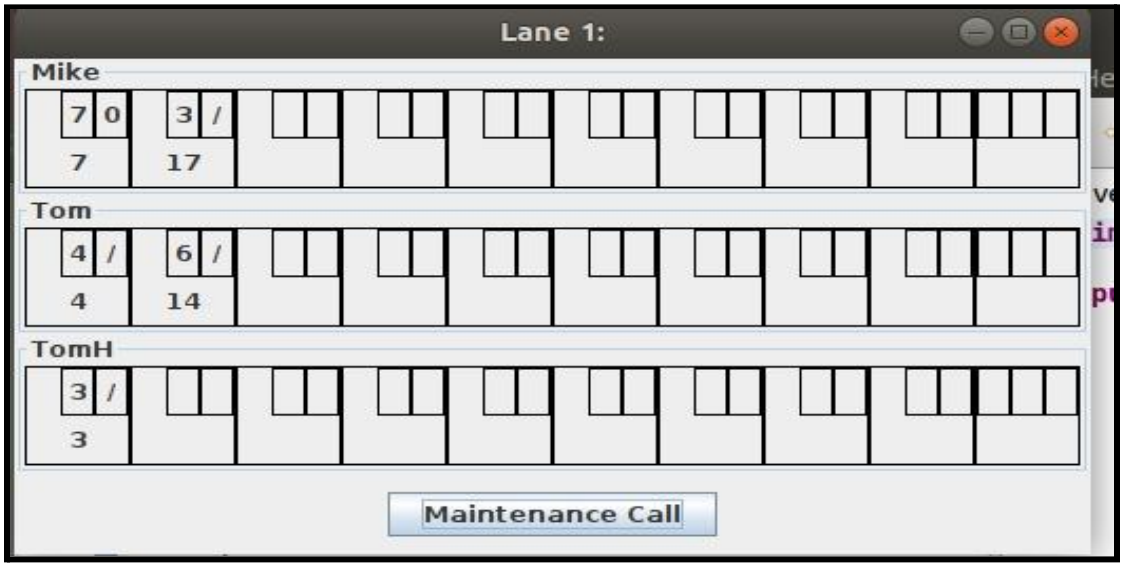
- Multiple lanes allow multiple parties/teams to play the game at the same time.

- The lane can be made more festive with the addition of parties. In the event that all lanes are full, the newly recruited party will be placed in a line. The Queue keeps track of the parties who have signed up but have yet to play.
- Each party can consist of one to multiple players.
- New players can be added to a lane while adding a party to it.



- The pinsetter view shows the pins dropped on each turn. The pins are re-racked after 2 consecutive balls.

- The ongoing game can be viewed along with the live scores. The scoreboard tracks scores gained by each player in a party after they bowled in their respective turns. The scoring scale is as follows:
    - Spare: 10 + pins dropped on next turn
    - Strike: 10 + pins dropped on next 2 turns



- Maintenance calls can be made while viewing the lane of any of the parties. For that time, the game is effectively suspended. Maintenance calls can be related to any type of repair work or a problem scenario, such as a pin setter that hasn't been re-racked, a ball that hasn't been returned, or a score that hasn't been updated. (A red coloured rectangle denotes that that lane has received a maintenance call.)

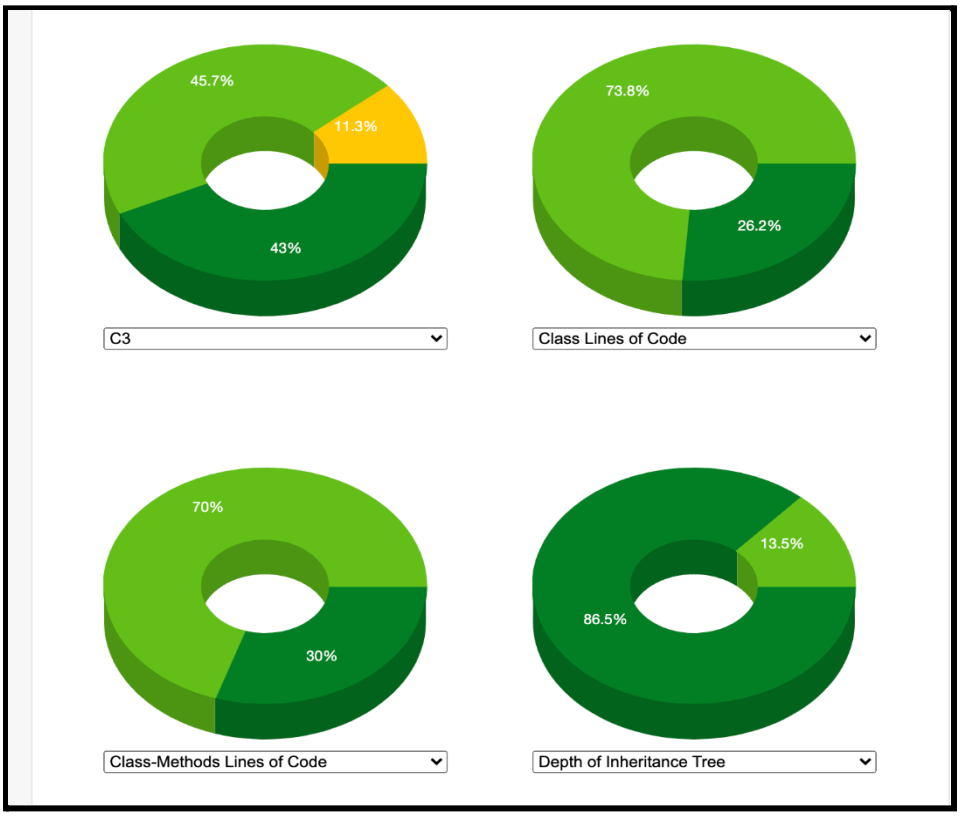
- After all turns of all players for one cycle of game is completed, the game asks whether the same party wants to play another round of game or not. If yes is selected, the same party plays the game again and if no is selected, their game is finished and a report is generated.

III. Brief Overview of the unit-1 refactored system and codebase:

The functionalities of the refactored code is the same as that of the original design. However, we made an attempt to analyze and improve the code metrics. Mainly, the refactoring included:

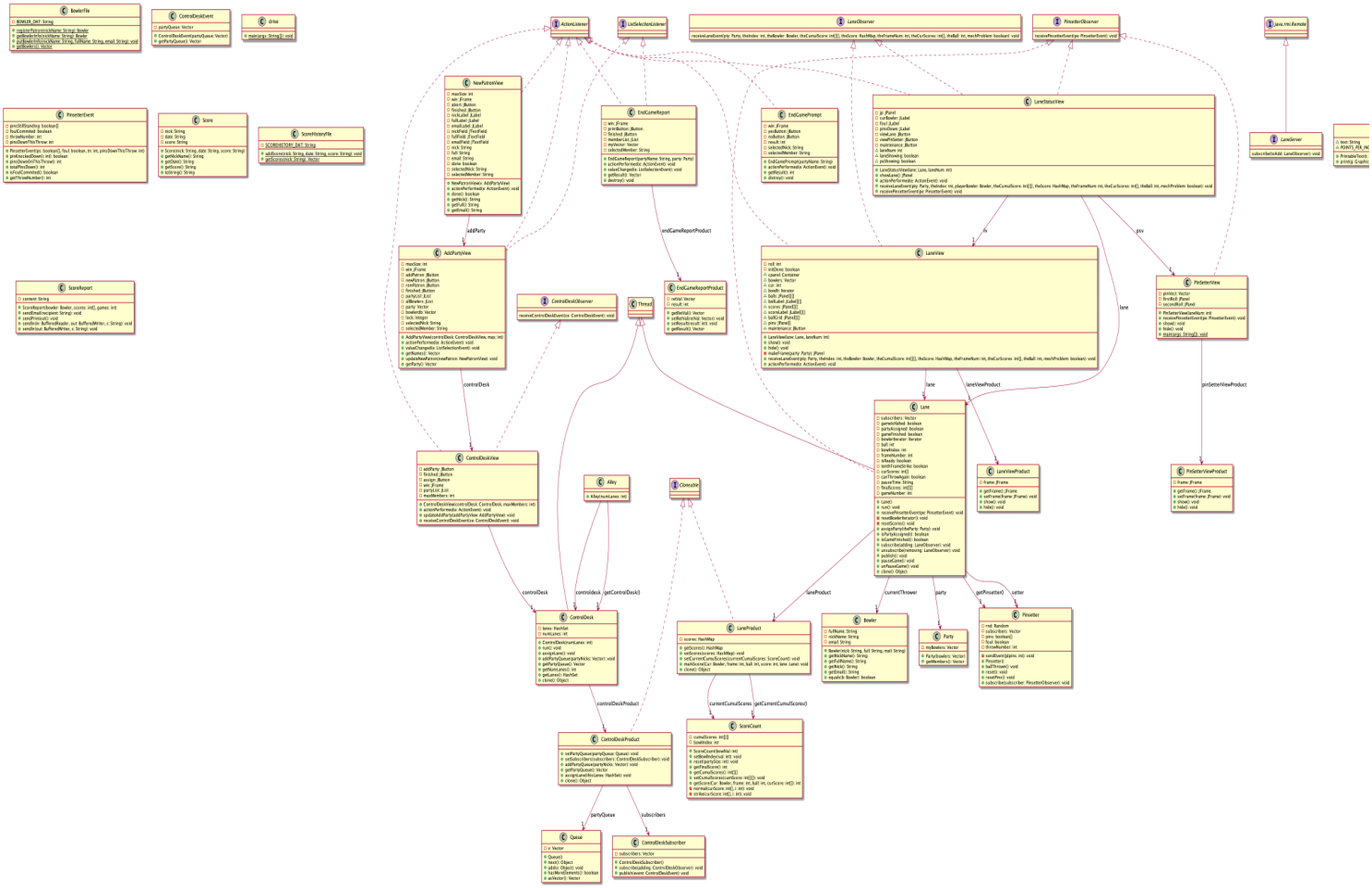
1. Removing redundant and dead code blocks. Also, remove the code that is kept for future development.
2. Move methods performing class specific tasks in the appropriate classes.
3. Remove the extra main method keeping just the one that is being used.
4. Perform modularization to solve the code smell problem of long/ god class.
5. Modularize long modules by putting task specific code blocks in seperate/new classes.
6. Improve the C3 metric for the codebase.
7. We also improved cyclomatic complexity and removed code smells like feature envy, ill positioned modules etc.

Overall, the final metrics obtained were:

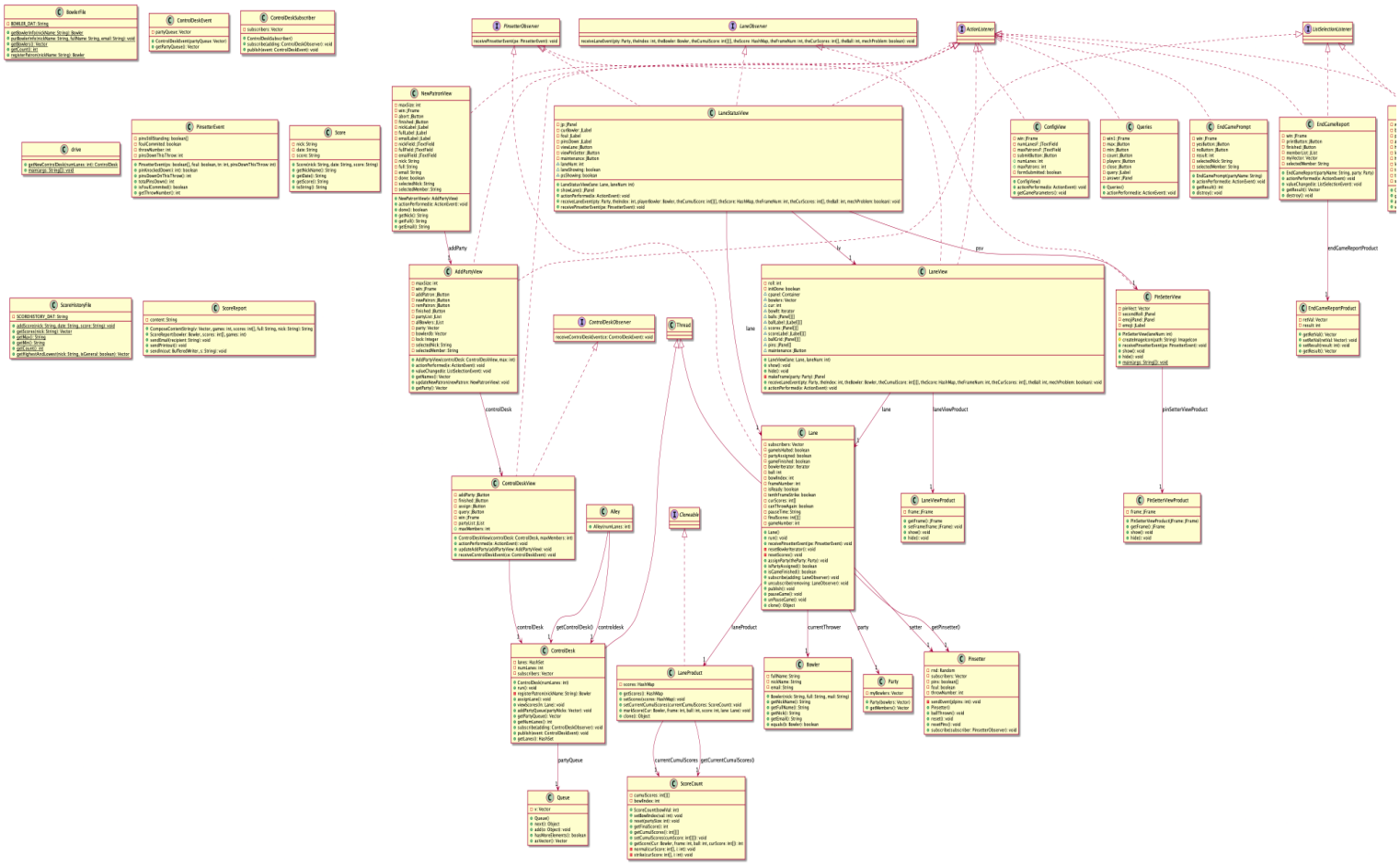


IV. Class Diagrams:

Before:

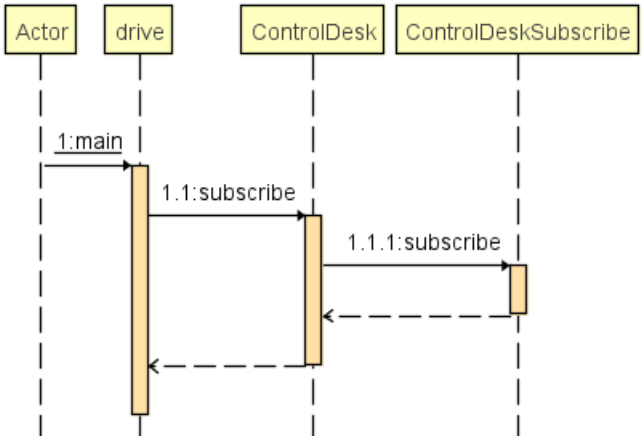


After:

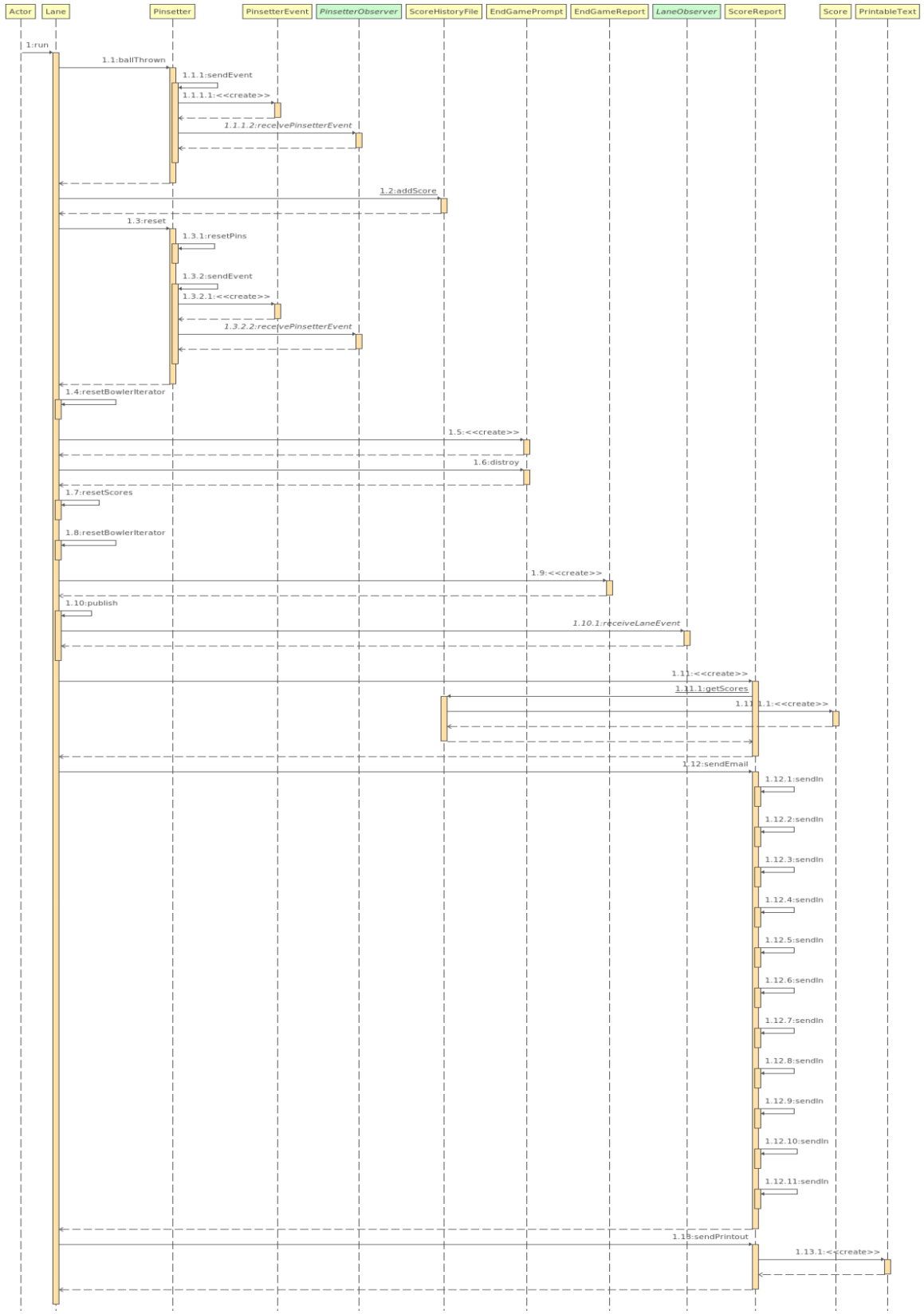


Sequence Diagrams

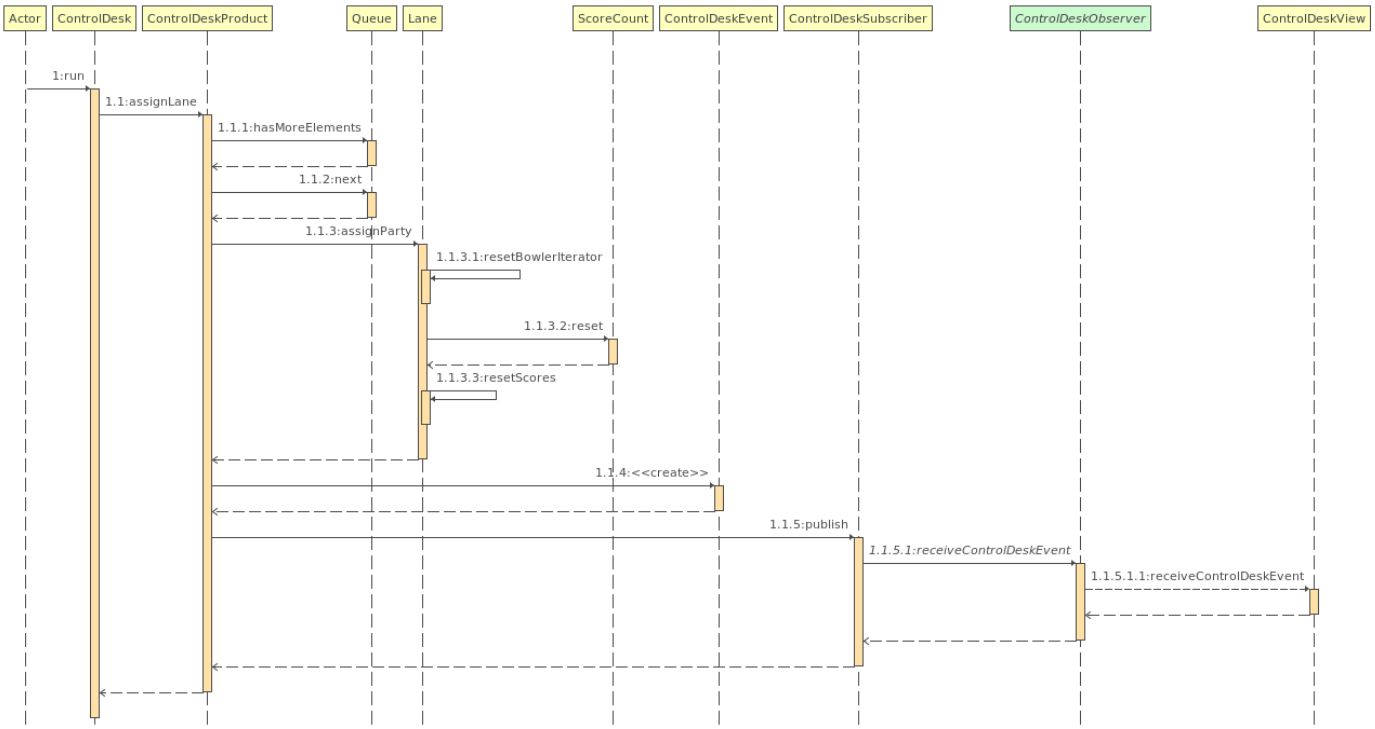
drive



Lane


Actor
Lane
Pinsetter
PinsetterEvent
PinsetterObserver
ScoreHistoryFile
EndGamePrompt
EndGameReport
LaneObserver
ScoreReport
Score
PrintableText
1:run
1.1:ballThrown
1.1.1:sendEvent
1.1.1.1:<<create>>
1.1.1.2:receivePinsetterEvent
1.2:addScore
1.3:reset
1.3.1:resetPins
1.3.2:sendEvent
1.3.2.1:<<create>>
1.3.2.2:receivePinsetterEvent
1.4:resetBowlerIterator
1.5:<<create>>
1.6:distroy
1.7:resetScores
1.8:resetBowlerIterator
1.9:<<create>>
1.10:publish
1.10.1:receiveLaneEvent
1.11:<<create>>
1.11.1:getScores
1.11.1.1:<<create>>
1.12:sendEmail
1.12.1:sendIn
1.12.2:sendIn
1.12.3:sendIn
1.12.4:sendIn
1.12.5:sendIn
1.12.6:sendIn
1.12.7:sendIn
1.12.8:sendIn
1.12.9:sendIn
1.12.10:sendIn
1.12.11:sendIn
1.13:sendPrintout
1.13.1:<<create>>

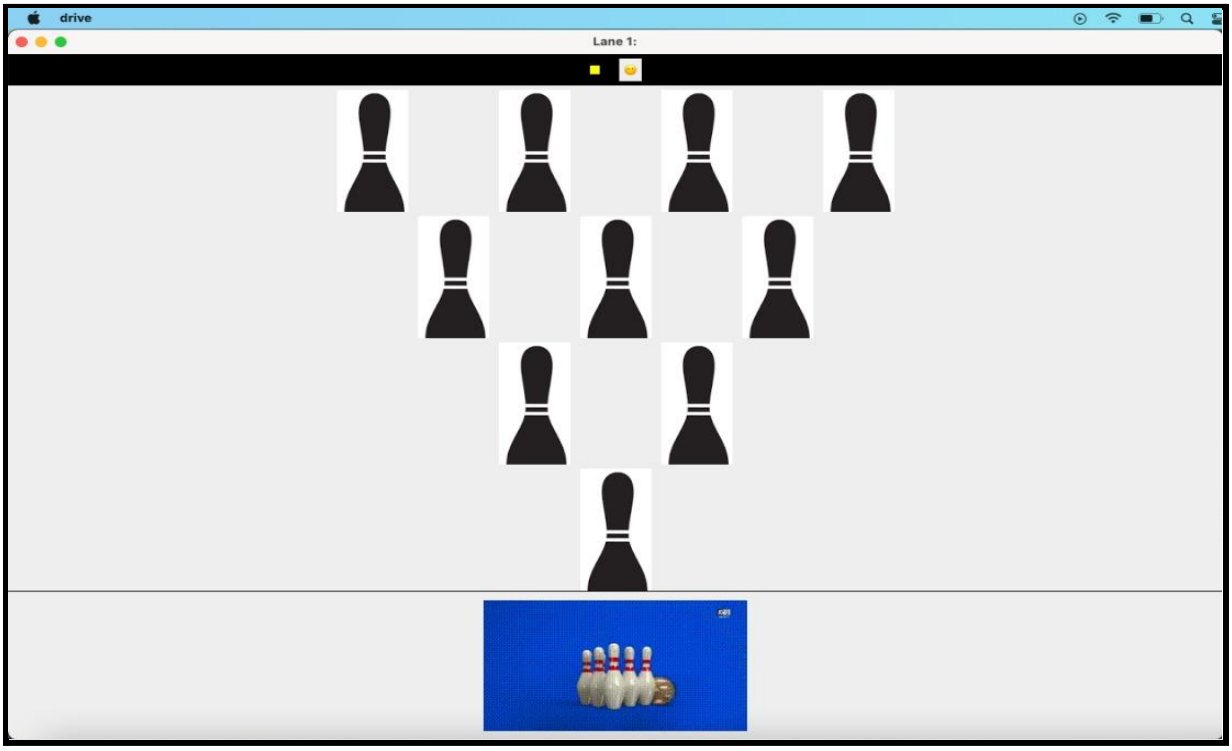
ControlDesk



V. Newly added requirements:

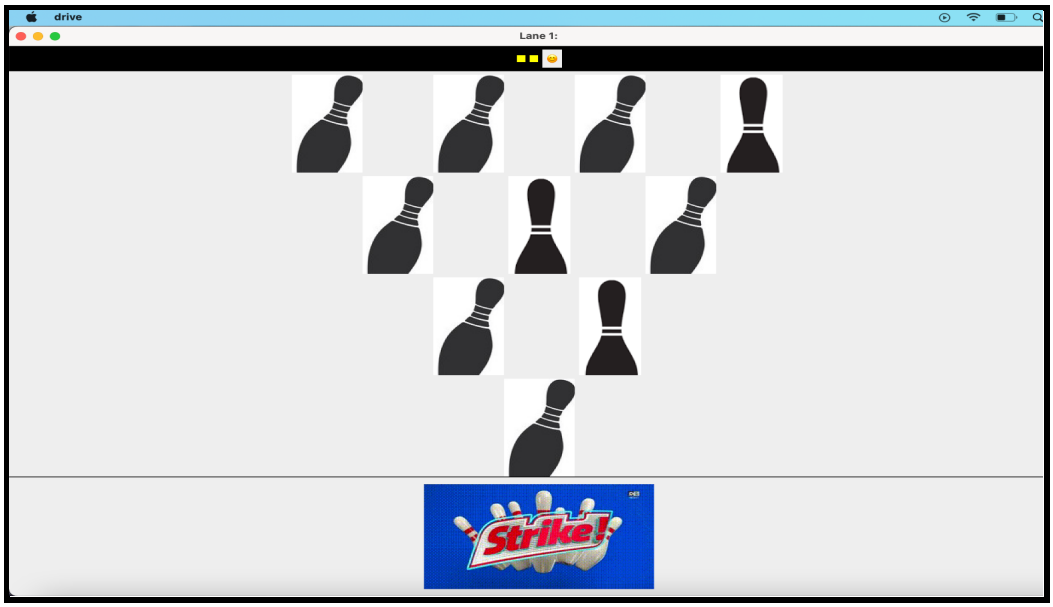
1. Make the game and UI interactive:

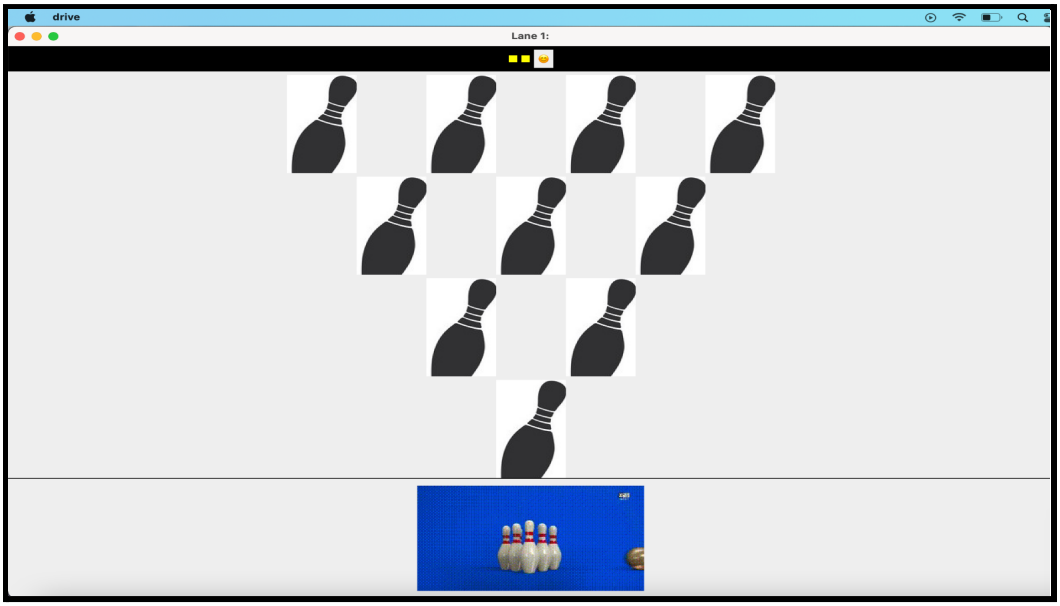
- Original design: The pinsetter view in the original codebase is properly functional but it is not interactive. The view was depicted by ASCII arrangement of numbers representing the pins which changed colors when pins were knocked down after every throw for each player. Initially, all pins were black. When a pin was knocked down, it was depicted by changing the color of the labels to grey from black (standing pins). After each frame, the labels were returned to its original state, i.e., black and the next frame is set up.
- New Pinsetter View: In the enhanced design, we implement a graphical interface for the pinsetter view.



As depicted by the above figure, all the bowling pins are in standing position initially. During the beginning of each frame there is a gif implemented for depicting the bowling game.

When a pin is knocked down, the depiction of the knocked out pin is updated. It gets displayed as a fallen pin as depicted in the below picture.

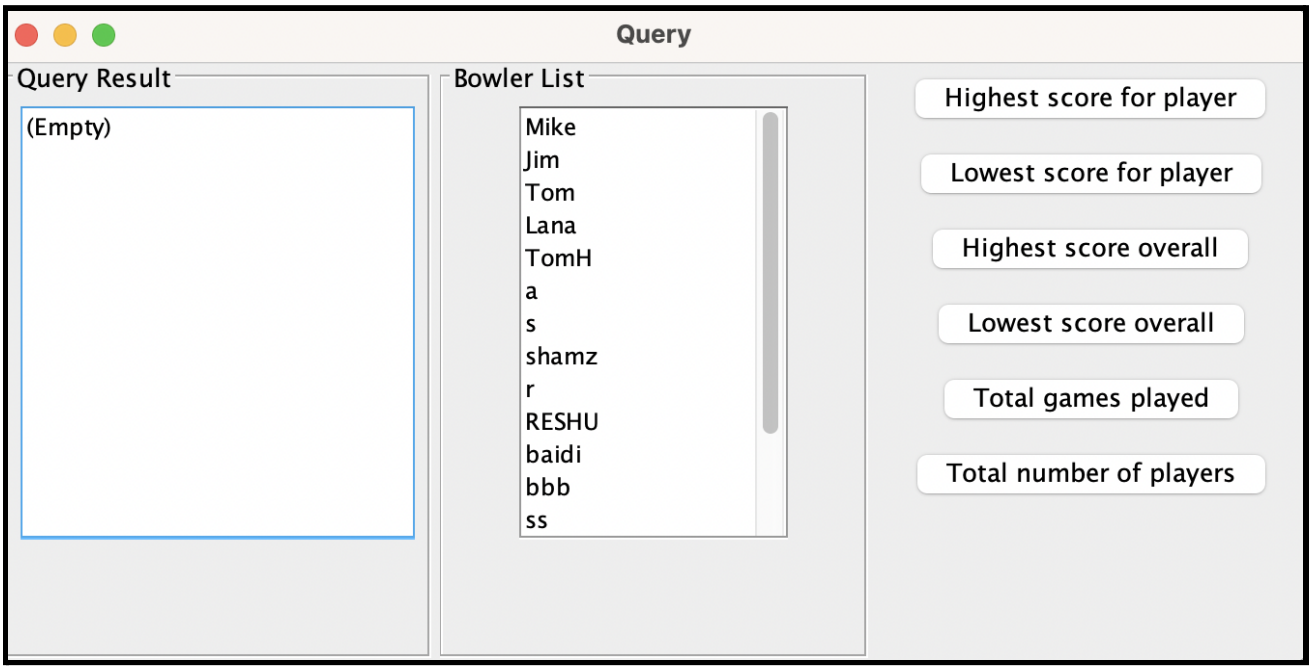




This setting of gif and pins is refreshed after each frame. This change of the is obtained by changing the label of the JLabel from normal string to ImageIcon in the pinSetterView class. Here, the images are initialized beforehand in the method to implement JLabel in the corresponding rows of the bowling game set up.

2. **Make the code extensible and for multiplayer**
   In the original codebase, the number of players was restricted to 5, but here, according to the requirement, it has been changed to 6. We created a class ConfigView to fulfill the 7th requirement, where a field concerned with the number of players is changed to 6 as default value. This field is extracted from the class by drive class to initialize the program.
3. **Add database layer and ad-hoc queries**
   Two classes named QueryView and Queries are created. The ScoreHistoryFile class is modified to derive information such as highest score, lowest score, number of players etc. The database is parsed to obtain information about the games played. From this, the above information are derived. The Queries class processes the input query to extract information from the database. The QueryView generates the UI required for this.

   We have implemented a total of 6 queries:

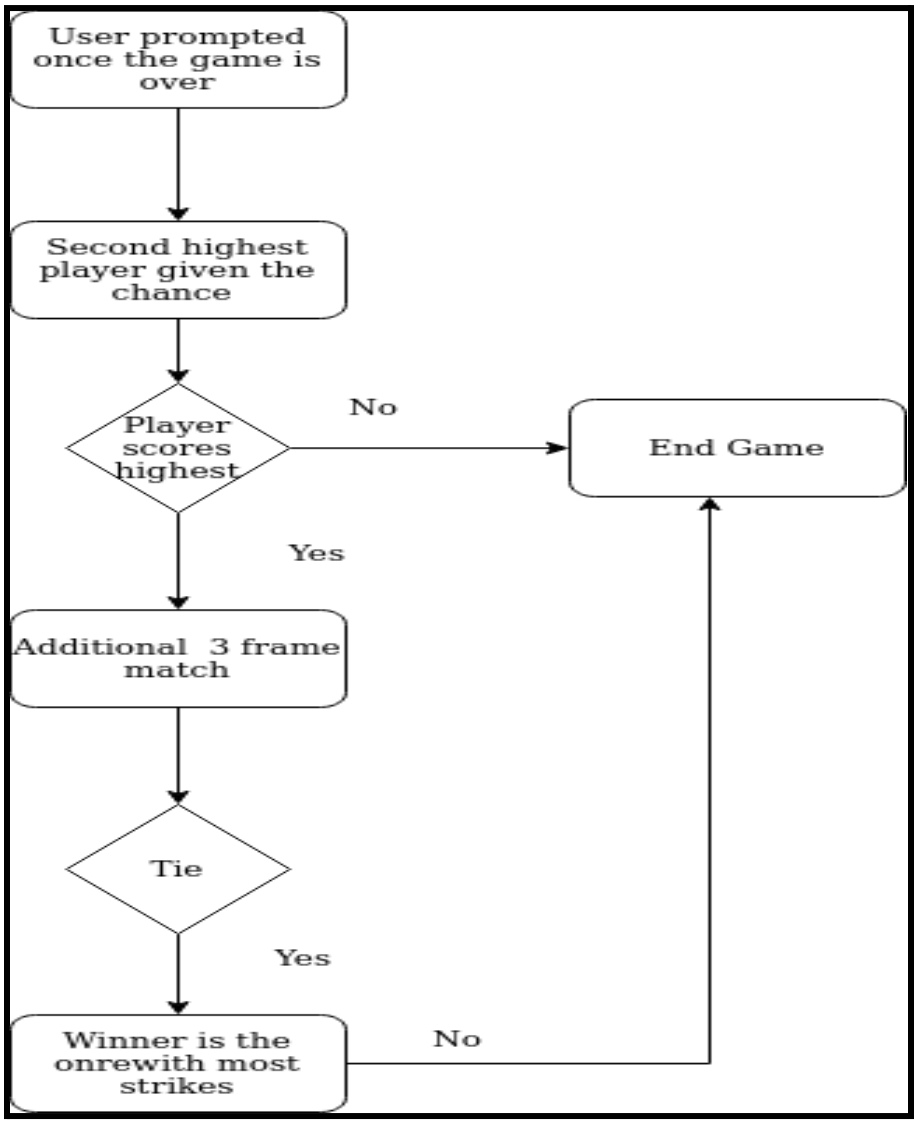


4. **Include score penalty for gutters**
   We have implemented this requirement in the ScoreCount class itself inside its getScore method. In case of two consecutive gutters, the highest score obtained by the player is reduced by ½ points and if this event occurs in the beginning of the game, the highest score is reduced by ½ of the points scored in the next frame.

5. Implement tie-breaker

The project requirement was that at the end of 10 frames, provide a chance to the 2nd highest player to bowl. If the player becomes highest, continue with 3 additional frames between 1st and 2nd highest till the winner is finalized. If there is a tie-break, declare the winner that had the most strikes.

Representation of the above requirement is:



6. Add emoticons as per player's score

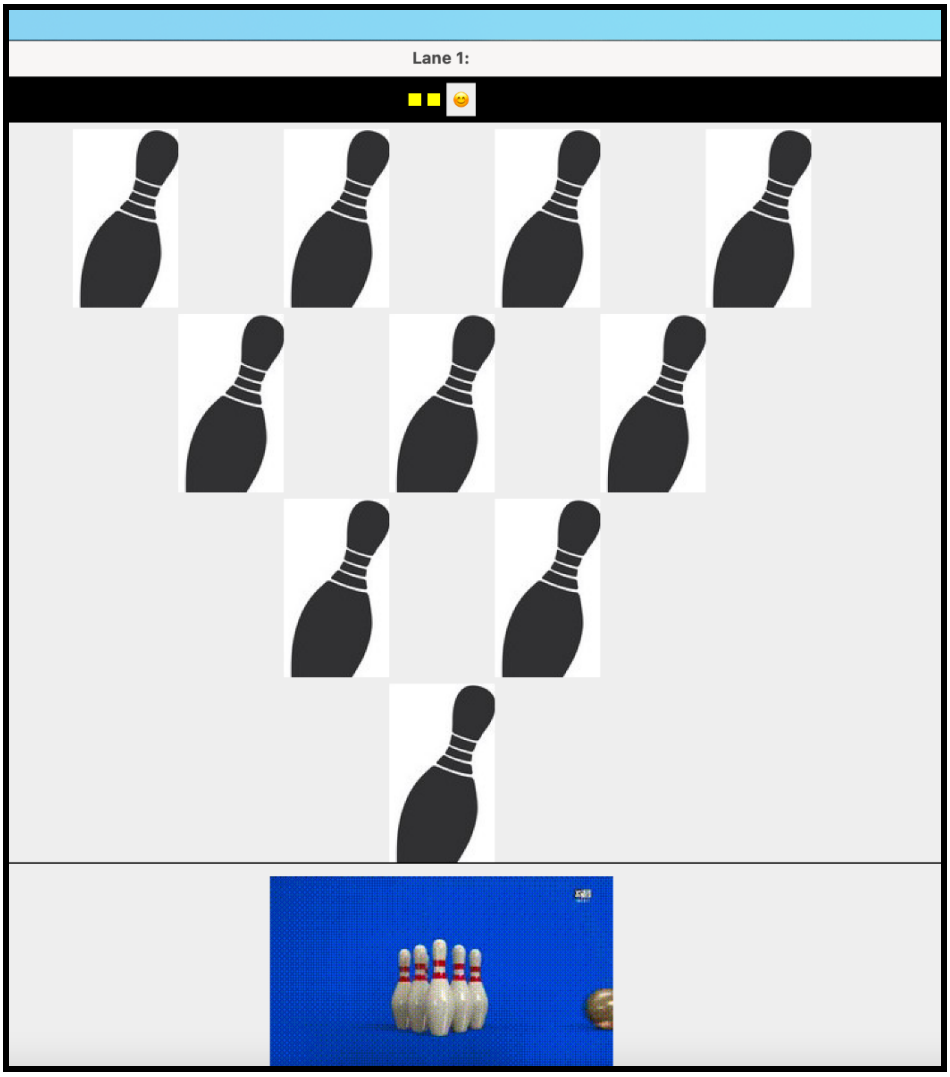
The project requirement required us to implement emoticons on the basis of the player's score. The emoticons need to depict the feeling of embarrassment, envy, happiness etc.

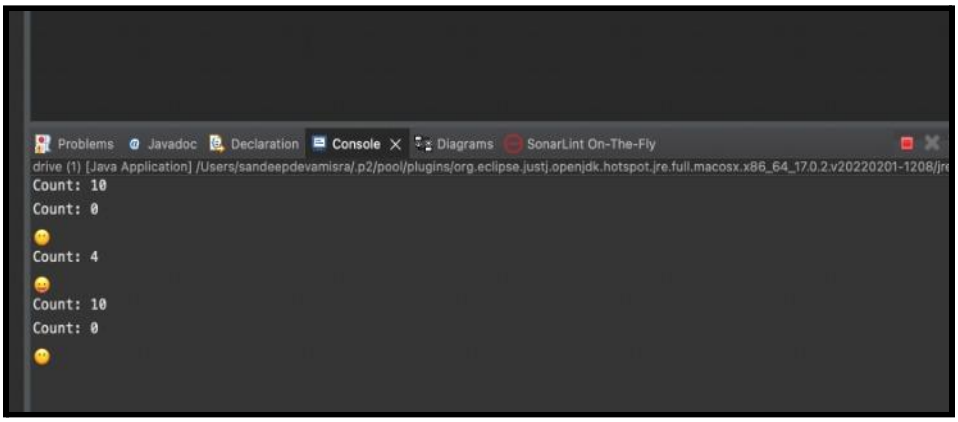
To implement this requirement, while calculating score at every throw for the player, we use the particular score at a throw to decide and display an emoji. The score-emoji mapping is as follows:

Score	Emoji
0	Laughing in embarrassment
1-2	Sad
3-4	Crying
5-9	Happy
10	Very happy and celebrating

The emoticon is instantiated according to the score after every throw for each player. This is done in the pinsetterView class.

These emoticons are updated in the pinsetter view for every player as well as updated on console when the code is running as shown below.


Lane 1:


Problems
Javadoc
Declaration
Console
Diagrams
SonarLint On-The-Fly
drive (1) [Java Application] /Users/sandeepdevamisra/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.x86_64_17.0.2.v20220201-1208/jre
Count: 10
Count: 0
Count: 4
Count: 10
Count: 0

7. **Make the numbers in codebase configurable**

All the numbers quoted and existing in the code should be made configurable. This includes the team members, number of lanes, and number of frames for a participant which can be input by the user. In this project, we wanted to keep things similar to the real-life bowling experience where the number of lanes and the number of frames per team remains constant. However, we can take 'n' people in a team with us.

So, the number of people in a team is considered to be a variable in this case. In the previous implementation of the Bowling Alley, the team member limit was not defined and one could play with any number of participants in a team. In order to address this issue, we have made a new class 'InitialConfigView' which asks the input for the maximum number of patrons in the party from the user. This is asked from the user before the game begins. Once the user has given input, the window disappears after clicking 'Submit'. This value is further taken into account when simulating the bowling alley control desk as it limits the no. of participants we can add. Through this, we are giving better management functionality to the admin/supervisor of the game which was not previously present.

VI. Classes and their responsibilities:

SNo.	Name of File	Methods	Functionality	Interlinked class
1.	AddPartyView	• void actionPerformed() • void valueChanged() • Vector getParty() • Vector getNames() • void updateNewPatron()	• Add new Patron to the list and showcase it in the GUI • Update the information in the database • Functionality of buttons(create, remove, add) and its user interface • Return the party information	• NewPatronView
2.	Alley	• ControlDesk • getControlDesk()	• Specifies a 'controldesk' and returns it	• ControlDesk
3.	Bowler	• String getNickName() • String getFullName ( ) • String getNick ( ) • String getEmail ( )	• Sets the value of nickname, fullname and email of the bowler	-
4.	BowlerLife	• static Bowler getBowlerInfo(String nickName) • static void putBowlerInfo(String nickName,String fullName,String email) • static Vector getBowlers()	• Class for interfacing with the polar database • Stores and retrieves bowler information from database that includes nicknames, full names and emails • Returns	-

			information of all bowlers	
5.	ControlDesk	• void run() • Bowler registerPatron(String nickName) • void assignLane() void addPartyQueue(Vector partyNicks) • Vector getPartyQueue() • int getNumLanes() void publish(ControlDeskEvent event) • HashSet getLanes()	• Consists of collection of lanes, party wait queues, number of links and collection of subscribers • Managers lanes, bowler information and queues for the GUI	• Lane
6.	ControlDeskEvent	• Vector getPartyQueue()	• Returns party queue and represents the control desk event	-
7.	ControlDeskObserver	• void receiveControlDeskEvent	• Interface class	-
8.	ControlDeskView	• void actionPerformed(ActionEvent e) • void updateAddParty(AddPartyView addPartyView) • void receiveControlDeskEvent(ControlDeskEvent ce)	• Displays the UI for control desk • Coordinates with attributes like controls , add party , assigned names etc.	• ControlDesk • AddPartyView
9.	ConfigView	• void actionPerformed(ActionEvent e) • void getGameParameters()	• Makes the parameters configurable	• drive
10.	Drive	• static void main()	• Driver class for the	• ControlDesk • Alley

			program and consists of the main function	• ControlDeskView
11.	EndGamePrompt	• EndGamePrompt(String partyName) • void actionPerformed(ActionEvent e) • int getResult() • void destroy()	• Showcases yes and no option in the UI • Clears up the main panel • Returns result or error	-
12.	EndGameReport	• EndGameReport(String partyName, Party party) • void actionPerformed(ActionEvent e) • Vector getResult() void destroy() • static void main(String args[]) • void valueChanged(ListSelectionEvent e)	• Displays the end game report • Organizes the buttons and components to the panel • Destroys the active object	-
13.	Lane	• void run() • void receivePinsetterEvent(PinsetterEvent pe) • void receivePinsetterEvent(PinsetterEvent pe) • void resetScores() • void assignParty(Party theParty) • void markScore(Bowler Cur, int frame, int ball, int score) • LaneEvent lanePublish( ) • void publish(LaneEvent event) • Setter and Getter functions	• Simulates lane action and hitting by thrown • Send scores to the control desk • Resets the pins, scores and bowlers	• Bowler • Party • Pinsetter
14.	LaneEvent	• LaneEvent()	• Sets and gets	• Party

		• Multiple boolean Getter and setters	the functionalities used in lane.java	• Bowler
15.	LaneEventInterface	• An interface class	• Multiple interface classes of lane event	• Party • Bowler
16.	LaneObserver	• An interface class	• Interface for receiving lane event	
17.	LaneServer	• An interface class	• Interface class	
18.	LaneStatusView	• LaneStatusView(Lane lane, int laneNum) • JPanel showLane() • void receiveLaneEvent(LaneEvent le) • void receivePinsetterEvent(PinsetterEvent pe)	• Shows and manages the GUI for lane status	• PinSetterView • LaneView • Lane
19.	LaneView	• void show() • void high() • Jframe makeFrame • void receiveLaneEvent(LaneEvent le)	• Render GUI for alley lanes	• Lane
20.	NewPatronView	• void actionPerformed() • void valueChanged() • Vector getParty() • Vector getNames() • void updateNewPatron()	• Display new Patron information • Getter and setter functions for patron information	• AddPartyView
21.	Party	• Vector getMembers()	• Sets new vector of bowlers	
22.	Pinsetter	• void ballThrown() • void reset() void • resetPins() void • subscribe(Pinsetter	• Send pin setter events to all subscribers • Simulates a	• PinsetterObserver

		Observer subscriber)	ball thrown coming in contact with the pinsetter • Resetting the pins	
23.	PinsetterEvent	• boolean pinsKnockedDown( ) • int pinsDownOnThisThrow() • int totalPinsDown() • boolean isFoulCommited() • int gerThrowNumber	• Creates a pinsetter event • Returns a number of parameters related to pins	-
24.	PinsetterObserver	• An interface class	• Interface class	-
25.	PinSetterView	• void receivePinsetterEvent()	• Implements the GUI related to pins • Receives a pinsetter event that it makes changes in the GUI	-
26.	PrintableText	• int print(Graphics g, PageFormat pageFormat, int pageIndex)	• Implements and renders graphical text in the GUI	
27.	Queries	• void actionPerformed(ActionEvent e)	• Process input query to retrieve queries from the database	• QueryView • ScoreHistoryFile
28.	QueryView	• vector getHighestAndLowestCaller(String selectedNick, Boolean isGeneral) • void actionPerformed(ActionEvent e) • void	• Generate the UI for the queries	• QueryView • ScoreHistoryFile

		valueChanged(ListSelectionEvent e)		
29.	Queue	● void add(Object o) ● boolean hasMoreElements() ● Vector asVector() ● Object next()	● Creates a new queue	-
30.	Score	● constructor, getter and setter functions	● Returns score along with the particular bowler information	-
31.	ScoreHistoryFile	● Vector getScores(string nick) ● void addScore() ● static String getMax() ● static String getMin() ● static int getCount() ● static Vector getHighestAndLowest ●	● Stores and purchase data from a particular file	-
32.	ScoreReport	● void sendEmail() ● void sendPrintout() ● void sendln()	● Generates and prints out the score report for the game	● Bowler

VII. Original codebase v/s refactored code v/s enhanced codebase: A comparison in terms of design and system

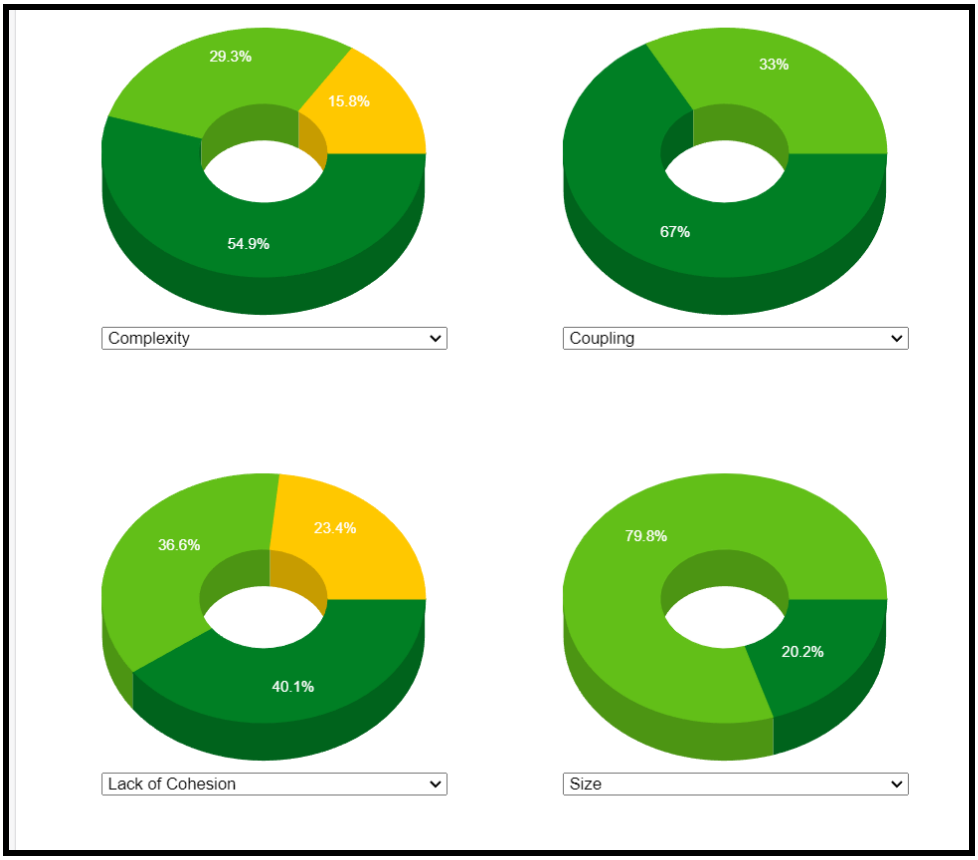
- **Strength of original codebase:**
  - ❖ Proper commenting making the code understandable
  - ❖ Modularity

- ❖ Overall low or medium coupling
- ❖ Design pattern: observer pattern and adapter pattern implemented

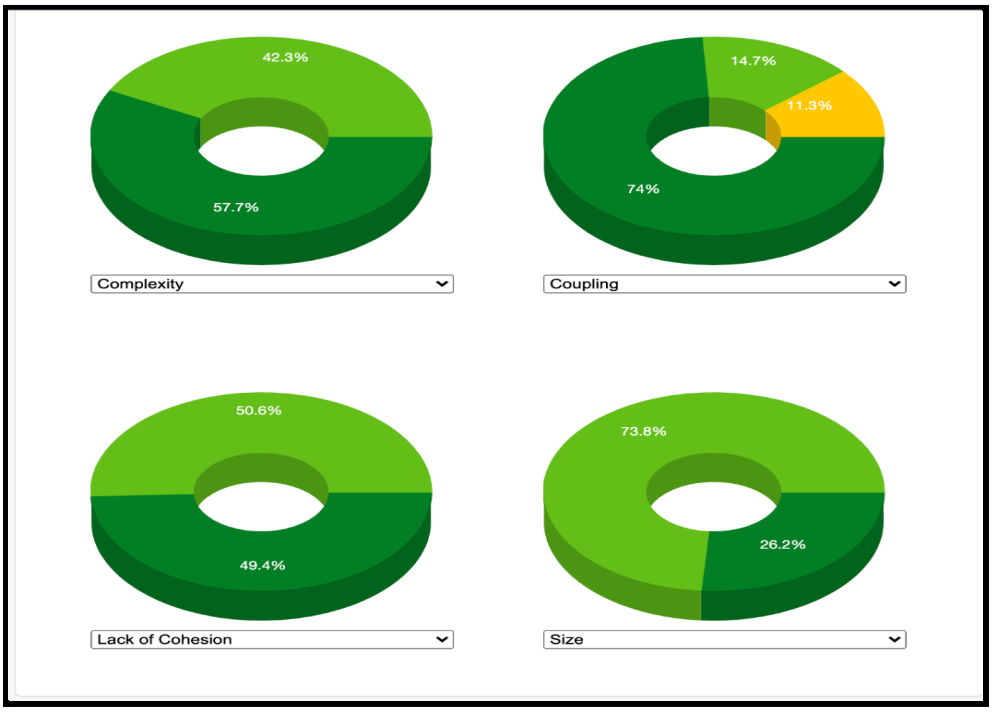
- **Improved metrics by refactoring:**
  - ❖ Improved cyclomatic complexity
  - ❖ Dead code elimination
  - ❖ Broke down long methods to improve coupling, cohesion and improve modularity
  - ❖ Removed multiple main methods
  - ❖ Removed god class
  - ❖ Repositioned appropriate modules
  - ❖ Removed unused code, both variables and methods
  - ❖ Feature envy
- **Improved and new functionalities by enhancement:**
  - ❖ UI made interactive: pinsetter view updated to display actual pins instead of boxes
  - ❖ Smiley implemented as per score bracket after each throw for each player
  - ❖ Database layer added to score player and score details
  - ❖ Ad-hoc queries implemented
  - ❖ Code made extensible for multiple players
  - ❖ Score penalty implemented for gutters
  - ❖ Maximum number of players made configurable

VIII. Code Metrics:

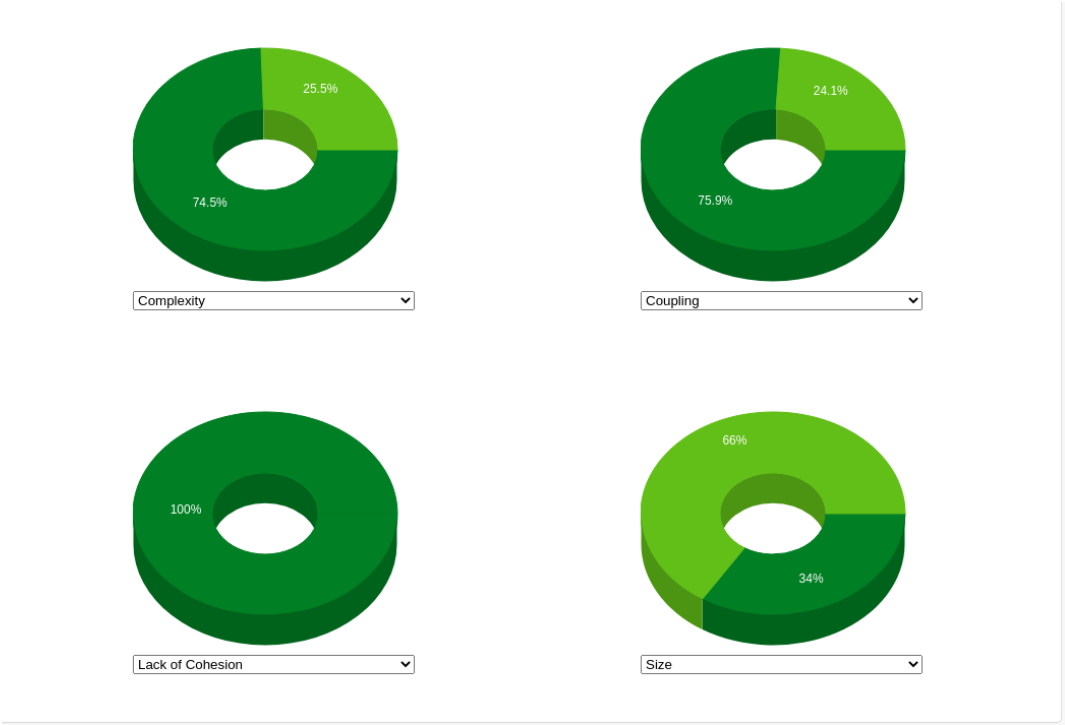
- **Original Codebase:**



- **Refactored Codebase:**



- **Enhanced Codebase:**



- **Metrics analysis- refactored code v/s enhanced code:**
    1. **Complexity:** Maintained overall. No. of classes with low complexity increased.
    2. **Coupling:** This metric was improved. The desired state is low coupling for any software solution and we were able to achieve this. We reduced the medium coupling and brought it down to all classes having low coupling.
    3. **Lack of Cohesion:** This metric too was improved as we were able to achieve 100% classes with high cohesion as desired.
    4. **Size:** Maintained overall. No.of classes with small size increased.

We were able to reduce and maintain the C3 metric, i.e., Complexity, coupling and cohesion overall and maintain it to be as desired.

1. **What were the metrics for the code base? What did these initial measurements tell you about the system?**
   Several metrics for the code base can be used for analysis of the original and the refactored code. The ones chosen and used by us are: Coupling, Cohesion, Lines Of Code, Complexity, Size of classes, Size of methods, C3,depth of inheritance tree and cyclomatic complexity.Values of these metrics for the initially existing code highlighted

the design positives as well as design problems as discussed in the section where we analyzed the original design of the code. Also, the exact metric values for initial and refactored code has been covered above.

2. **How did you use these measurements to guide your refactoring?**
   The summary of the metrics helped us to concentrate on the problematic areas that were increasing the complexity of the code and refactor modules that would improve these metrics. Moreover, we were able to target specific modules as the metrics helped us in identifying code smells and anti-patterns in the existing code. The measurements of the metrics helped us to refactor and then measure the extent to which our refactoring improved the code base.