Bowling Alley Simulation
SE Unit 1 - Refactoring project

Date of Submission: 20-02-2021

Team Members:

1. Ayush Mittal
   Roll No. : 2021201030
   - Roles: Code Analysis/Refactoring/CodeMR
   - Effort: 23 hours
2. Ira Tyagi
   Roll No. : 2021201031
   - Roles: UML Design/Refactoring/Comparison
   - Effort: 23 hours
3. Arpit Maheshwari
   Roll No. : 2020201078
   - Roles: Code Analysis/Patterns/Sequence Diagram
   - Effort: 22 hours
4. Param Dubey
   Roll No. : 2021204013
   - Roles: UML/Documentation
   - Effort: 22 hours

Introduction

Our project aims to automate the bowling game by developing a software to help the Lucky Strikes Bowling Center (LSBC). It contains all the facilities of modern bowling games like Control Desk, Lane, Pinsetter, Scoring Station and Bowler Management. It has been written entirely in Java. We used the current code as a baseline, understood and reverse engineered it to improve the code based on standard software metrics like coupling, cohesion, elimination of bad code smells etc.

The bowling game allows gamers to register as individuals or as a group.
Players are admitted to the Control Desk, where they are assigned to the appropriate lane. If a group of players wants to join as a party, they will all be assigned to the same lane. The bowling order is determined by the order in which the players are admitted.
Each Lane has its own Scoring Station, which records each player's score.
After each Throw, the updated Pin Setting Equipment automatically detects missing pins and sends their condition to the Scoring Station. Each player's score is calculated at the Scoring Station, and once a game is over, the results are submitted to the Control Desk.
At the Lane's Scoring Station, the player may check the status of Pin Setter.
The Scoring Station also keeps track of individual scores for each Frame.
Players can leave or restart a game after a game has finished.

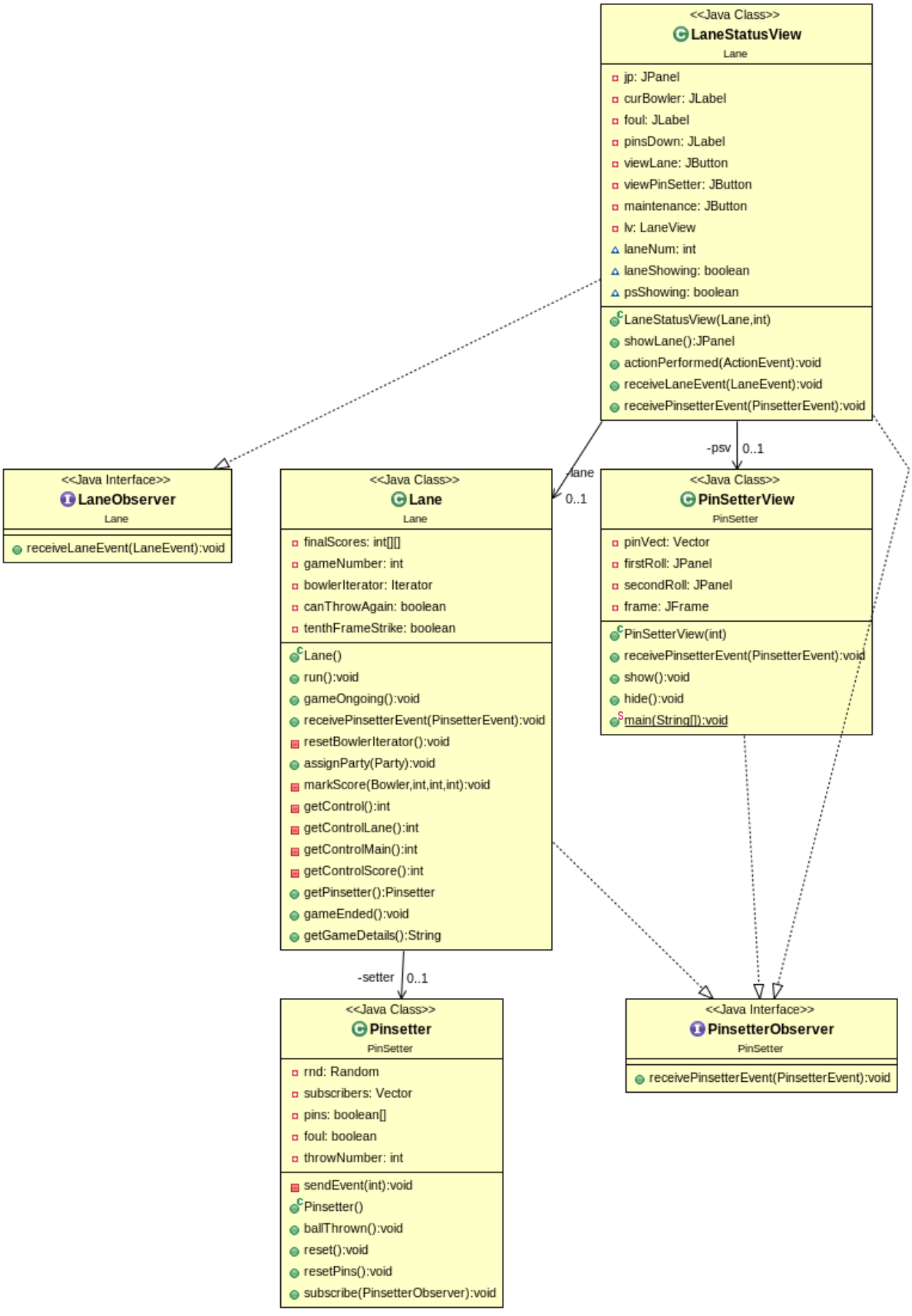
Original Code

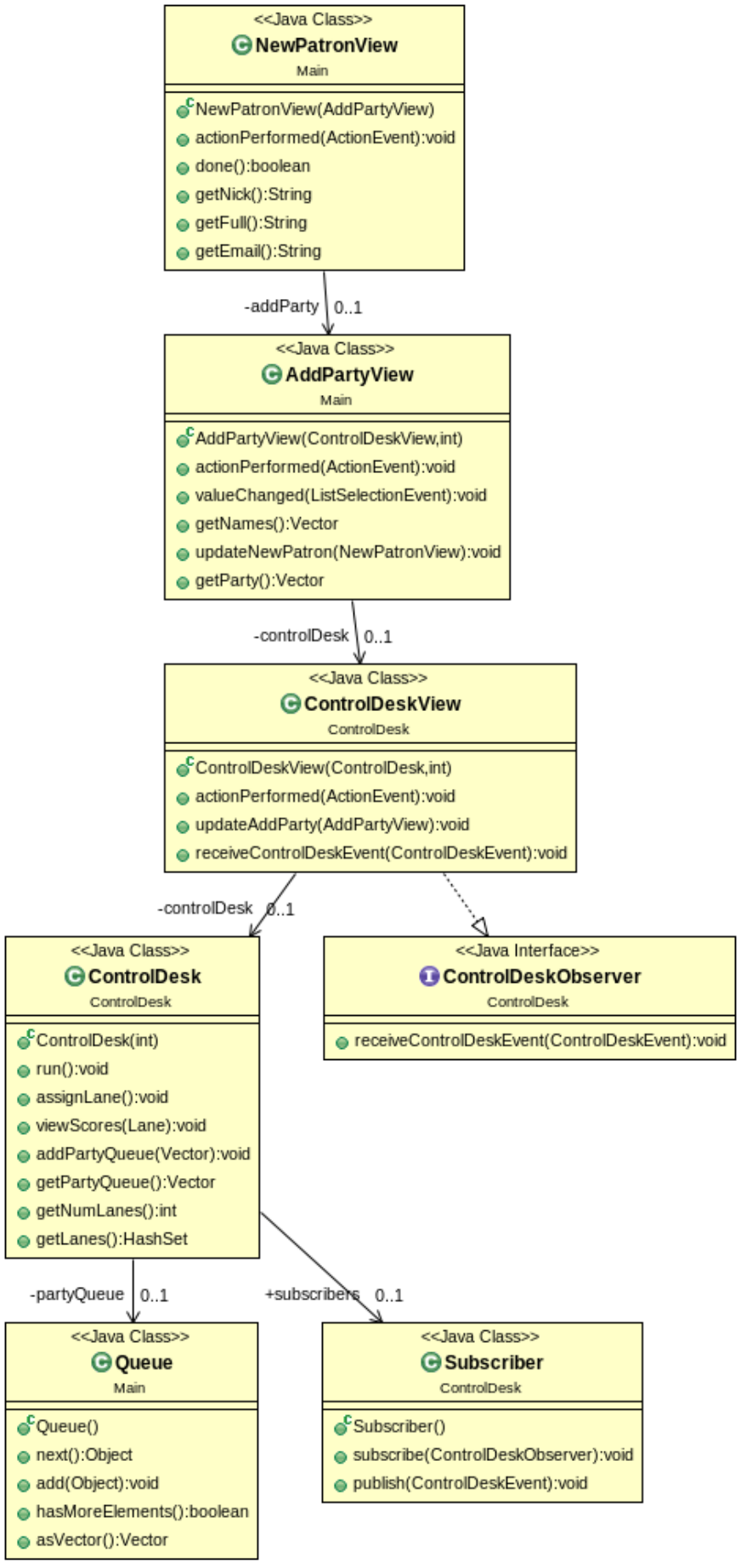
The original code appeared to be very complicated at first, and it took us about a week to understand it completely. However, upon closer inspection of the files, we discovered that all of the functionality required are implemented and, for the most part, well written. All we had to do was clean it up, refactor the code and if needed divide the classes according to functionalities.

Class Diagrams

The original code was contained in a single package. We have divided the code into multiple packages in order to increase the readiability of the code. The packages are created on the basis of their functionality. This will make it easier to add new code in future that performs some related function.
Given below are the UML class diagrams of the refactored code that show the relationships among the relevant classes. The appropriate cardinalities are also included to make the diagram clear.

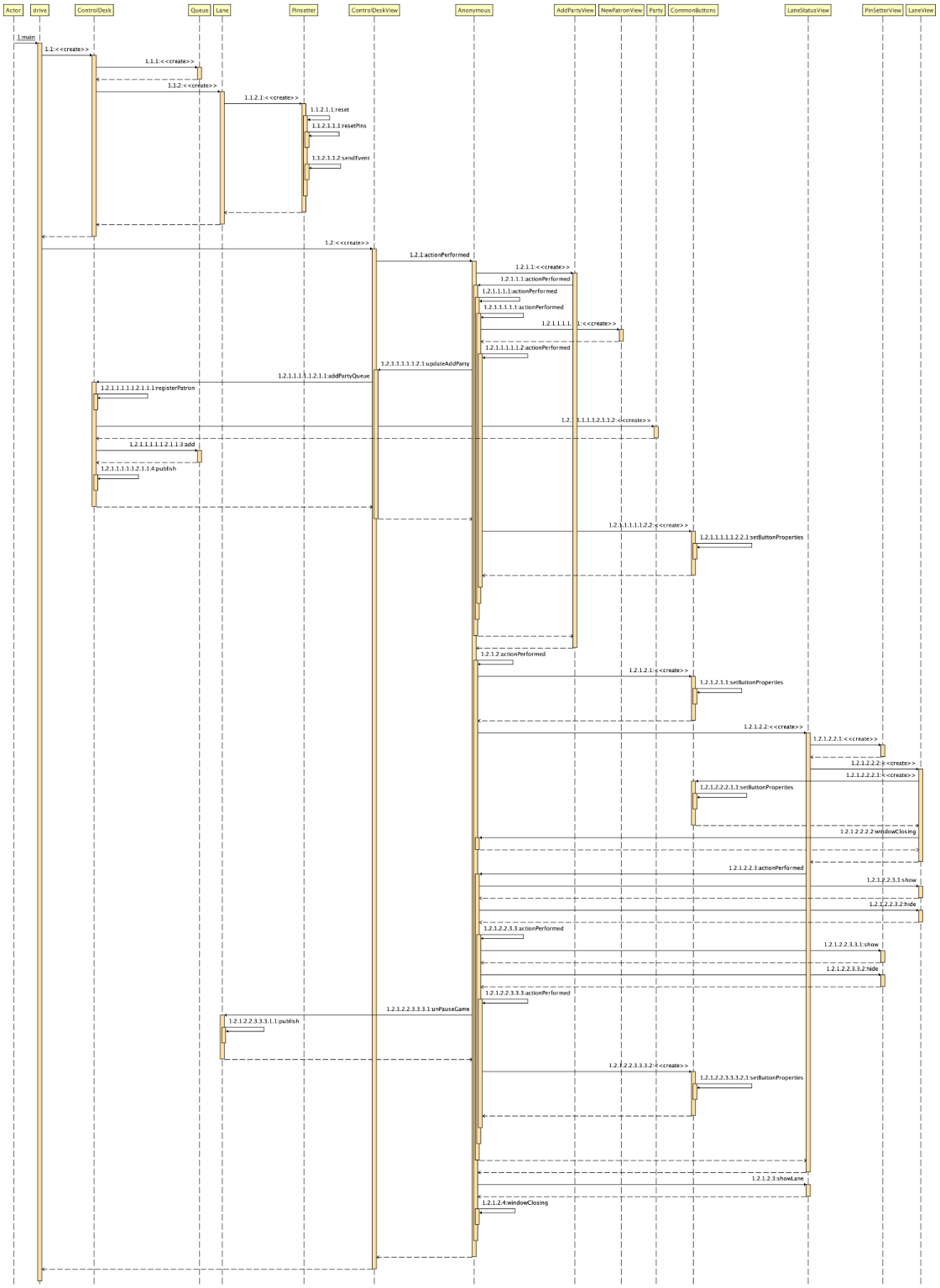
1. The lane class is broken down into multiple subclasses. The LaneStatus View implements the LaneObserver interface and the PinsetterObserver is implemented by Lane, LaneStatusView and the PinSetterView.
2. The second UML class diagram shows the relationships among the Control desk classes and various other relevant classes. The ControlDeskObserver interface is implemented by the ControlDeskView class. The ControlDesk class is associated with the Queue and Subscriber class.


<<Java Class>>
LaneStatusView
Lane
jp: JPanel
curBowler: JLabel
foul: JLabel
pinsDown: JLabel
viewLane: JButton
viewPinSetter: JButton
maintenance: JButton
lv: LaneView
laneNum: int
laneShowing: boolean
psShowing: boolean
LaneStatusView(Lane,int)
showLane():JPanel
actionPerformed(ActionEvent):void
receiveLaneEvent(LaneEvent):void
receivePinsetterEvent(PinsetterEvent):void
-psv
0..1
<<Java Interface>>
LaneObserver
Lane
receiveLaneEvent(LaneEvent):void
<<Java Class>>
Lane
Lane
finalScores: int[][]
gameNumber: int
bowlerIterator: Iterator
canThrowAgain: boolean
tenthFrameStrike: boolean
Lane()
run():void
gameOngoing():void
receivePinsetterEvent(PinsetterEvent):void
resetBowlerIterator():void
assignParty(Party):void
markScore(Bowler,int,int,int):void
getControl():int
getControlLane():int
getControlMain():int
getControlScore():int
getPinsetter():Pinsetter
gameEnded():void
getGameDetails():String
-lane
0..1
<<Java Class>>
PinSetterView
PinSetter
pinVect: Vector
firstRoll: JPanel
secondRoll: JPanel
frame: JFrame
PinSetterView(int)
receivePinsetterEvent(PinsetterEvent):void
show():void
hide():void
main(String[]):void
-setter
0..1
<<Java Class>>
Pinsetter
PinSetter
rnd: Random
subscribers: Vector
pins: boolean[]
foul: boolean
throwNumber: int
sendEvent(int):void
Pinsetter()
ballThrown():void
reset():void
resetPins():void
subscribe(PinsetterObserver):void
<<Java Interface>>
PinsetterObserver
PinSetter
receivePinsetterEvent(PinsetterEvent):void

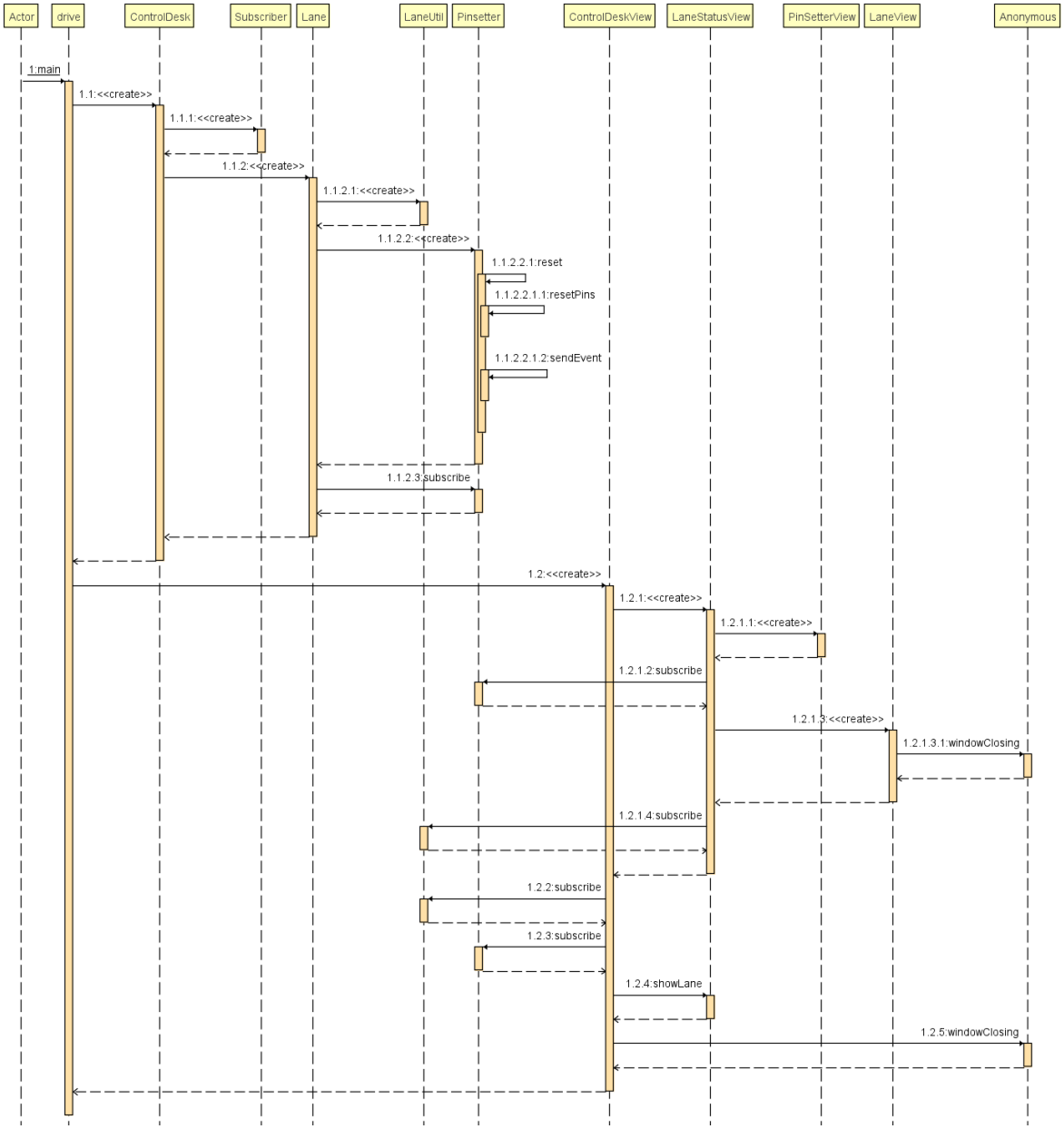

<<Java Class>>
NewPatronView
Main
NewPatronView(AddPartyView)
actionPerformed(ActionEvent):void
done():boolean
getNick():String
getFull():String
getEmail():String
-addParty
0..1
<<Java Class>>
AddPartyView
Main
AddPartyView(ControlDeskView,int)
actionPerformed(ActionEvent):void
valueChanged(ListSelectionEvent):void
getNames():Vector
updateNewPatron(NewPatronView):void
getParty():Vector
-controlDesk
0..1
<<Java Class>>
ControlDeskView
ControlDesk
ControlDeskView(ControlDesk,int)
actionPerformed(ActionEvent):void
updateAddParty(AddPartyView):void
receiveControlDeskEvent(ControlDeskEvent):void
-controlDesk
0..1
<<Java Class>>
ControlDesk
ControlDesk
ControlDesk(int)
run():void
assignLane():void
viewScores(Lane):void
addPartyQueue(Vector):void
getPartyQueue():Vector
getNumLanes():int
getLanes():HashSet
<<Java Interface>>
ControlDeskObserver
ControlDesk
receiveControlDeskEvent(ControlDeskEvent):void
-partyQueue
0..1
+subscribers
0..1
<<Java Class>>
Queue
Main
Queue()
next():Object
add(Object):void
hasMoreElements():boolean
asVector():Vector
<<Java Class>>
Subscriber
ControlDesk
Subscriber()
subscribe(ControlDeskObserver):void
publish(ControlDeskEvent):void

Sequence Diagrams :

1. Original code



2. Refactored Code



Class Functionalities

Class	Overview
AddPartyView	● Class for GUI components needed to add a party to ControlDesk.
Alley	● Create an alley which contains an instance of ControlDesk class. ● The ControlDesk object is used to maintain multiple Lanes.
Bowler	● Creates new Bowler and stores details like full name, nick name and email. ● Can also be used to compare two Bowlers.
BowlerFile	● Class for interfacing with Bowler database. ● Helps in storing and retrieving bowler information from the database.
ControlDesk	● Class represents the control desk. ● Assigns Lanes to Parties and also maintains a Party Queue for parties currently waiting. ● Add subscribers and broadcast event updates to them.
ControlDeskEvent	● Represents an Event class for ControlDesk. ● Can extract parties in the wait queue.
ControlDeskObserver	● Interface for classes that observe control desk events.
ControlDeskView	● GUI class for ControlDesk. ● Users can add parties to lanes and access other control desk functionalities.
Drive	● Main class. ● Creates an instance of Alley and Control Desk.
EndGamePrompt	● GUI class for asking if the user wants to continue another game once a game is finished on a Lane.
EndGameReport	● GUI class for asking if the user wants to display the final report of the game.
Lane	● Implements Thread class to implement multithreading for creation of multiple instances of lanes. ● Contains Lane elements Pins, Score and Game status. ● Handle scoring for the current bowler bowling on the Lane. ● Can also help in pausing/unpausing the game.
LaneEvent	● Implements LaneEventInterface class.

	• Represent current status of the Lane containing Party, Bowler, scores etc. • Acts as an event class for Lane and LaneObserver class.
LaneEventInterface	• Interface for LaneEvent class.
LaneObserver	• Interface for classes that observe Lane events.
LaneServer	• Interface for subscribe function for LaneObserver class.
LaneStatusView	• GUI class for displaying status and maintenance calls for a lane. • Helps users to access pinsetter.
LaneView	• GUI class for showing scoring of each bowler. • Users can also call for maintenance.
NewPatronView	• GUI class for addition of new party and bowlers.
Party	• Represents the party and stores information about bowlers in the party.
PinSetter	• Class for game simulation by randomly dropping pins. • Also resets pins after each round.
PinSetterEvent	• Class represents the current status of PinSetter like number of pins down, foul committed and throw number etc. • Acts as event class for PinSetter
PinSetterObserver	• Interface for classes that observe PinSetter events.
PinSetterView	• GUI class for displaying the status of Pins after each throw.
PrintableText	• Print class for generating Score Report. • Using Printable interface, helps in setting font, font size, layout and page format etc for report. • A kind of Adapter Design Pattern.
Queue	• Helper class for maintaining the waiting queue of parties at the control desk.
Score	• The score class contains information for name, date and score of a game for the bowler.
ScoreHistoryFile	• Helps in maintaining the Score database.
ScoreReport	• GUI class for showing the scores of the games.

Analysis of Original Code

In the original code using manual analysis and different metrics we are able to find out below mentioned strengths and weaknesses.

Strength in the Original Code

1. Proper documentation comments which helped us to understand the functionalities of the classes and overall project.
2. Standard Nomenclature is followed for most classes and the method names easily communicate their functionalities. Example for each Observer interface we have an “Observer” word at last like LaneEventObserver.
3. Most of the needed functionalities are implemented correctly.
4. Design patterns like Observer, Adapter and abstract factory etc are already implemented in the code, just need some modifications to work properly.

Weakness of Original Code

1. Dead/Lazy Code - There is a lot of dead code available in the current code like LaneServer class if not used in the project. Also, there are some functions and variables that are not used anywhere. Similarly, some of the classes like Alley are not needed as it is simply using the ControlDesk class.
2. Repetitive and Copy/Paste code - Mostly in the view classes there is a need for creation of JButton, JPannel and JFrame again and again. This makes the same code repeated multiple times. This can be improved by using loops and functions.
   Another example of repetitive code is multiple functions for the same functionality like in Bowler class, there were two functions getNick() and getNickName() for fetching the NickName of the Bowler.
3. Large and Spaghetti Code - There are classes which are too large with complex looking functions like Lane.java. It contains 621 lines of code (including comments). Also, it contains functionalities for calculation of score which could be moved in another class. It also contains large functions which use repetitive codes which can be divided into multiple functions. Similarly there are other large classes for View i.e. GUI classes like AddPartyView, LaneView etc. This can be solved by refactoring the classes and functions.
4. Feature Envy - More than its own data, methods access the data of another object. Many class functions, such as in ControlDesk and Lane, are heavily used

in other files. Also they use a lot of data and methods from other classes. This makes a lot of dependency of code in a few classes.

5. Other problems include wrong if else structure, lack of loops and cyclomatic complexity etc.

Apart from this, a couple of design patterns, such as the Observer Pattern and the Adapter Pattern, are implemented in the original source. The Observer Pattern can be seen in a subscription-based arrangement where various event-based statuses are broadcast to all objects that have subscribed. Furthermore, the PrintableText Class module can detect Adapter Pattern during interactions with the file database. Template design pattern is also used between LaneEvent and LaneEventInterface. LaneEventInterface contains a structure of functionalities needed in LaneEvent and then LaneEvent class implements it.

Approach to Improve Code

Class	Analysis	Refactored Design
AddPartyView	● Contains repeated code for JButton and JFrame. ● Cyclomatic complexity can be improved for conditional statements.	● A separate class named SuperView for common view related functionalities is created. Then these functionalities are used to improve the code quality by reducing redundant code. ● Also some conditional statements like in actionPerformed are simplified.
Alley	● Lazy class ● Can be removed as directly using the ControlDesk class.	● Removed class and Control Desk object directly used in drive class.
Bowler	● Contains multiple functions for the same functionality - getNickname and getNick.	● Removed getNickName() function.
BowlerFile	● No changes needed.	
ControlDesk	● Big class with a lot of	● Attribute name changed - adding to

	dependencies. ● Some functions like registerPatron can be moved to other classes.	newSubscriber ● Moved registerPatron to BowlerFile class. ● Improved subscriber and publish methods.
ControlDeskEvent	● No changes needed.	
ControlDeskObserver	● No changes needed.	
ControlDeskView	● Complex code ● Cyclomatic complexity can be improved for conditional statements. ● Contains repeated code for JButton and JFrame.	● SuperView class is used to improve the code quality by reducing redundant code. ● Also some conditional statements like in actionPerformed are simplified.
Drive	● Main class for execution	● Remove use of Alley class.
EndGamePrompt	● Conditional Statements can be simplified. ● Contains repeated code for JButton and JFrame.	● SuperView class is used to improve the code quality by reducing redundant code. ● Also some conditional statements like in actionPerformed are simplified.
EndGameReport	● Eliminate dead code. ● Remove duplicate code.	● SuperView class is used to improve the code quality by reducing redundant code. ● Removed dead code.
Lane	● Acting as a God Class with a lot of classes depends on it. ● Complex and Spaghetti code which can be splitted into functions. ● Also, score calculation can be separated out in a different class to improve cohesion and coupling. ● Refactor class	● Separated Code by creating LaneUtil and ScoreUtil classes. ● The ScoreUtil class contains code for score calculation and improves cohesion of overall code. ● The LaneUtil class contains code for basic functionalities like pause game, publish, reset scores and getscores. ● Modified Lane class contains code for basic functionalities of game play at lane. ● Also, large functions like run() are divided into smaller functions.

LaneEvent	● Contains Large constructors with 8 parameters i.e. code smell of Bloaters type. ● Remove dead code and if needed split classes according to functionality. ● Implement LaneEventInterface to apply abstract patterns in class.	● Created another class LaneEventUtil for separating code for bowl and current frame. ● Removed dead code. ● Implement LaneEventInterface to apply abstract patterns in class.
LaneEventInterface	● No changes needed. ● Not implemented in any class.	● Implement it in LaneEvent class.
LaneObserver	● No changes needed.	
LaneServer	● Dead Code ● Remove this class	● Removed the class as it was not needed.
LaneStatusView	● Contains repeated code for JButton and JFrame. ● Conditional statements can be simplified.	● SuperView class is used to improve the code quality by reducing redundant code. ● Simplified code in actionPerformed().
LaneView	● Contains repeated code for JButton and JFrame. ● Contains long methods with high conditional complexity.	● SuperView class is used to improve the code quality by reducing redundant code. ● Removed dead code and splitted receiveLaneEvent function.
NewPatronView	● Contains repeated code for JButton and JFrame.	● SuperView class is used to improve the code quality by reducing redundant code.
Party	● No changes needed.	
PinSetter	● No changes needed.	
PinSetterEvent	● No changes needed.	
PinSetterObserver	● No changes needed.	
PinSetterView	● Contains a lot of repeated code that can be simplified just by	● Removed the main function as it is a dead code. ● Simplified code by using loops and

	using loops.	reduced LOC.
PrintableText	● No changes needed.	
Queue	● No changes needed.	
Score	● No changes needed.	
ScoreHistoryFile	● No changes needed.	
ScoreReport	● No changes needed.	

Due to the above mentioned refactoring and modifications following new classes are introduced.

Class	**Overview**
ScoreUtil	● Class contains code for score calculation, cumulative scores.
SuperView	● Class for common view related functionalities like JFrame and JButton.
LaneUtil	● Class contains code for basic functionalities like pause game, publish, reset scores and getscores.
LaneEventUtil	● Class contains frame and current bowl related information.
Subscriber	● Class to improve functionality of ControlDesk by separating subscriber and publishing methods from it.

Analysis of Refactored Code -

1. **High Cohesion :** Cohesion is the measure of how much elements of a class are functionally related.
    - In long classes, there was a low cohesion example in Lane, ControlDesk and LaneEvent, we divided these classes, keeping elements with same/similar functionalities together.
    - Breaked the methods which were long. This enhanced coupling because now methods in class were calling each other**.**

2. **Low Coupling :** As dependency/interaction of a class on other classes increases with other classes, it's coupling increases. Few classes like Lane, were too long, having dependency on a large number of other classes. Dividing such classes, such that new classes do not require many other classes, with a notion of minimum overlapping of functionality. We tried to have as few neighbors for a class as possible, this reduced coupling in final codes.

3. **Separation of concern :** Splitting big classes, having multiple functions did Separation of concern**.** Reducing coupling and increasing cohesion helped in achieving separation of concern.

4. **Information Hiding :** Abstraction helped in information hiding, for example in refactored codes, we separated subscriber's methods from ControlDesk and made a separate class named Subscribers. This hide the implementation details from the ControlDesk class. We made attributes private and implemented their getter and setter functions which helps in data abstraction.

5. **The Law of Demeter :** According to the law of Demeter, classes should know about and interact with as few other classes as possible, i.e The Law of Demeter encourages to minimize coupling and enhancing cohesion. This we have achieved by lowering coupling and enhancing cohesion as specified above.

6. **Extensibility :** Extensibility is the level of effort required to implement the extension and a measure of the ability to extend a system. Keeping dependency low on other classes helps in easy and effective extensibility.

7. **Reusability :** Making the codes modular helped in increasing reusability. We made a SuperView class for buttons and panels. In Spite of implementing buttons and panels again and again, we have used that SuperView class. It helps in easy debugging and reducing lines of code.

As mentioned earlier in analysis of the original code section, design patterns, such as the Observer Pattern and the Adapter Pattern, are implemented in the original source. These design pattern implementations help in balancing the coupling, cohesion and also improves the overall functionality of code. But these patterns are not implemented correctly in some classes like ControlDesk and needed some modifications to improve its working. Need of Observer pattern in the bowling system arises like when an event in the pinsetter class is triggered, the bowling system must inform certain classes. The Observer Pattern permits any class to become a listener or observer because the pinsetter is not firmly associated with any of the classes that are expected to receive the event update. Only the Observable Class' interface needs to be implemented. This style also aids with codebase extension, since more listeners can be readily added to the system by implementing the interface. Moreover, the PrintableText Class module can detect Adapter Pattern during interactions with the file database. Template design pattern is also used between LaneEvent and LaneEventInterface. LaneEventInterface contains a structure of functionalities needed in LaneEvent and then LaneEvent class implements it.

Metrics Analysis -

Metrics Analysis of initial codes on codeMR :

List of all classes (#30)

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
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	Hello					3	low	low	low	low
27	ControlDeskObserver					2	low	low	low	low
28	LaneObserver					2	low	low	low	low
29	LaneServer					2	low	low	low	low
30	PinsetterObserver					2	low	low	low	low

Analysis of coupling, lack of cohesion, complexity, size, lines of code on initial codes is shown in above tables. Yellows and light green in tables show/indicate

where we should improve initial codes. For example -

- Yellow in the Lack of cohesion column in Lane with high LoC, tells that Lane is a very big class with many functions which were not much related with each other. This indicates we should separate/split Lane class into 2 or more classes such that each class has methods with overlapping functionality. Similar for ControlDeck, LaneEvent and LaneView classes.
- Classes like Lane, ControlDesk, Ponsetter and many more have light green color in size column, this indicates they would be having high lines of code for sure and hence in such a file we should check for dead codes, and try to mudalrise the code if possible.
- Light green is relatively less complex and actually tells inefficient coding practice. Most of the files in the initial code were having badly implemented conditional statements. Mutually exclusive conditions were checked again and again, and a lot of comparisons were made in condition checking.

How metrics analysis helped in refactoring-

Every time after making some major changes we did analysis, when we tried to reduces complexity by reducing line of codes in classes by dividing them into small classes(keeping one functionality in one class), it showed that we did too much of division as coupling and lack of cohesion remained same, it was light green and yellow for newly made small classes. And in spite of decreasing, the complexity increased in some cases.

At every stage, Matrix analysis helped to visualize our current progress and various factors like cohesion and coupling in terms of diagrams and numbers. Which kept us working in the right direction. Changing one factor affect others too, this analysis helped to keep a good balance between the above mentioned factors.

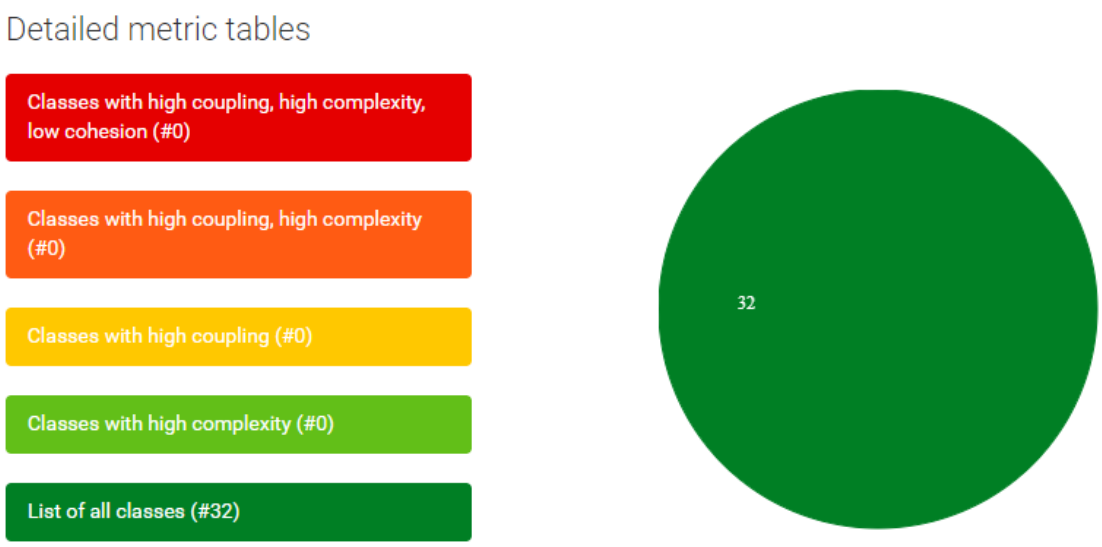
After Refactoring -

Major steps taken to refactor the code and their effects on metrics-

- We identified classes which could be broken; this reduces coupling, enhanced cohesion and reduced line of codes in those classes.
- Identified dead codes, replaced repeated codes with loops and improved methods that improved LoC.
- Breaked long methods into two, this increased functional dependency, this leads to higher cohesion.
- Identifying a few classes and interfaces which could be replaced/removed, we did it.
- Created a class for buttons and panel, this helped in eliminating repeated implementation of them at many places, reduced Loc, and took a notion of code reusability.

After refactoring the codes, most of the Yellows got removed, we decreased coupling, enhanced cohesion, reduced complexity, removed many dead codes and hence decreased LoC.

We removed all the Yellows except one, Lack of Cohesion remained Yellow even after taking the measures mentioned in the report. Except for Lack of Cohesion in Lane class, we improved on all the factors in most of the classes, improved the codes and made them simpler and better.



Metrics Analysis of Refactored Codes -

List of all classes (#32)

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	Lane					151	low-medium	low-medium	medium-high	low-medium
2	ControlDeskView					74	low-medium	low-medium	low-medium	low-medium
3	ControlDesk					49	low-medium	low-medium	low-medium	low
4	LaneStatusView					81	low	low-medium	low-medium	low-medium
5	LaneView					133	low-medium	low	low-medium	low-medium
6	AddPartyView					107	low-medium	low	low-medium	low-medium
7	ScoreUtil					60	low-medium	low	low	low-medium
8	LaneUtil					50	low-medium	low	low	low
9	LaneEvent					23	low-medium	low	low-medium	low
10	PinSetterView					82	low	low	low	low-medium

11	ScoreReport					76	low	low	low	low-medium
12	NewPatronView					63	low	low	low	low-medium
13	EndGameReport					60	low	low	low	low-medium
14	Pinsetter					47	low	low	low	low
15	BowlerFile					47	low	low	low	low
16	EndGamePrompt					45	low	low	low	low
17	PinsetterEvent					26	low	low	low	low
18	Bowler					23	low	low	low	low
19	PrintableText					21	low	low	low	low
20	ScoreHistoryFile					20	low	low	low	low
21	LaneEventUtil					17	low	low	low	low
22	Score					14	low	low	low	low
23	SuperView					14	low	low	low	low
24	Queue					12	low	low	low	low
25	LaneEventInterface					10	low	low	low	low
26	Subscriber					9	low	low	low	low
27	ControlDeskEvent					6	low	low	low	low
28	drive					6	low	low	low	low
29	Party					6	low	low	low	low
30	ControlDeskObserver					2	low	low	low	low
31	PinsetterObserver					2	low	low	low	low
32	LaneObserver					2	low	low	low	low

Analysis of New folder

General Information

Total lines of code: 1338

Number of classes: 32

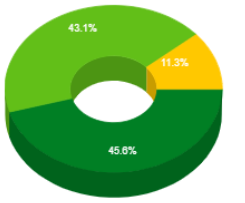
Number of packages: 5

Number of external packages: 19

Number of external classes: 95

Number of problematic classes: 0

Number of highly problematic classes: 0

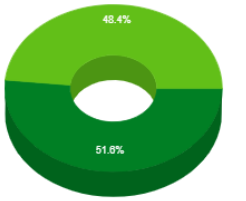


C3

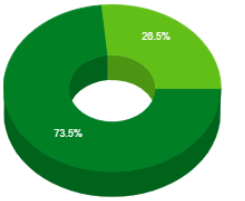
- Very High
- High
- Medium-high
- Low-medium
- Low

Distribution of Quality Attributes

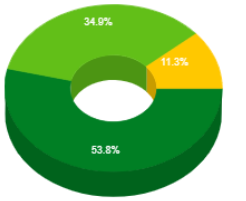
Complexity, Coupling, Cohesion, and Size



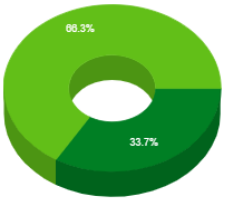
Complexity



Coupling



Lack of Cohesion



Size