1 Project Details Overview

The project is to develop upon the application provided, and refactor it to bring to industry standards. The aim of the refactoring is to improve the overall performance in terms of code metrics. Along with refactoring, the project goal also requires us to develop and incorporate three new features into the application. Another basic requirement of the project is the document the code and identify regions of strengths and weaknesses and also record and document the changes made.

The team members are;

- Anirudha Ramesh (20171088)
- Aditya Kumar (2018101098)
- Chirag Shilwant (2020201061)
- John Daniel Mathew (2020201005)

Chirag	• UML Diagram, Code smells identification • Refactoring Lane Class, LoadSavedView.Java
John	• Refactoring and classes Lane Class, Drive.java, Queue.java, ScoreCalculator • Code smells identification, Sequence Diagram
Anirudha	• Refactoring and classes in Control Desk, ControlDeskView, NewPatronView, SaveFile • Code smells identification, Metrics overview,
Aditya	• Refactoring and classes in ScoreHandling, PinSetter • Code smells identification, Metrics overview,Documentation of initial code

during the project, and not time based. But here is an approximate work put in by each member:

Chirag	12 hours
John	12 hours
Anirudha	12 hours
Aditya	12 hours

Table 2: Weekly effort of members

2 Introduction

The Application is a Bowling Alley Management System designed for the Lucky Strikes Bowling Center (LSBC) chain of establishments. The main aim of the application is to automate certain stages of operating the Bowling Center.

The application has quite a few features that have been implemented. Firstly, a feature for managing the assigning of parties to lanes and monitoring the status of the lane has been implemented. Secondly, a feature to automate the scoring of the bowler has also been implemented. Another feature that has been implemented is the ability to automate the pinsetter operation from initial setup to counting pins dropped to resetting the pins again. Besides just the operation of lanes, the application also aims to aid the center in managing their customers and storing a record of the bowlers, and their games. It goes on further to even email and print out the game scores for the the bowlers.

The application has been developed on Java

3 UML Diagrams

The complete UML Class and Sequence Diagram were too big to fit in here. However, they can be found in accompanying to the report. All the UML diagrams were made using *PlantUML* and the code is also provided in *src/UML*

3.1 UML Class Diagram

The UML Class Diagram attempts to incorporate all the relations of the different classes while also classifying them into their respective packages.

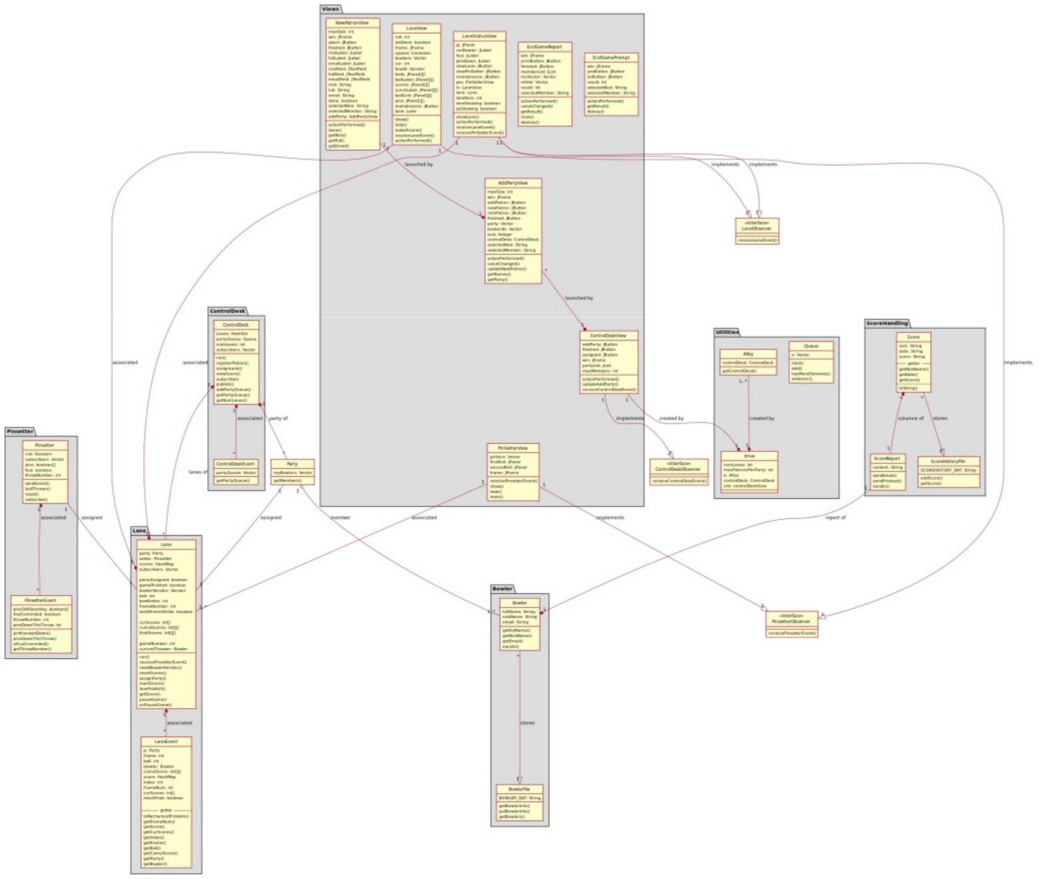


Figure 1: UML Class Diagram (Complete)

[htp] The complete picture can be found in both PNG and SVG format accompanied to this pdf with the name *UMLClass.png* or *UMLClass.svg*.

3.2 UML Sequence Diagram

The UML Sequence Diagram attempts to incorporate the sequence of activities that the user and the system performs and their interactions. The

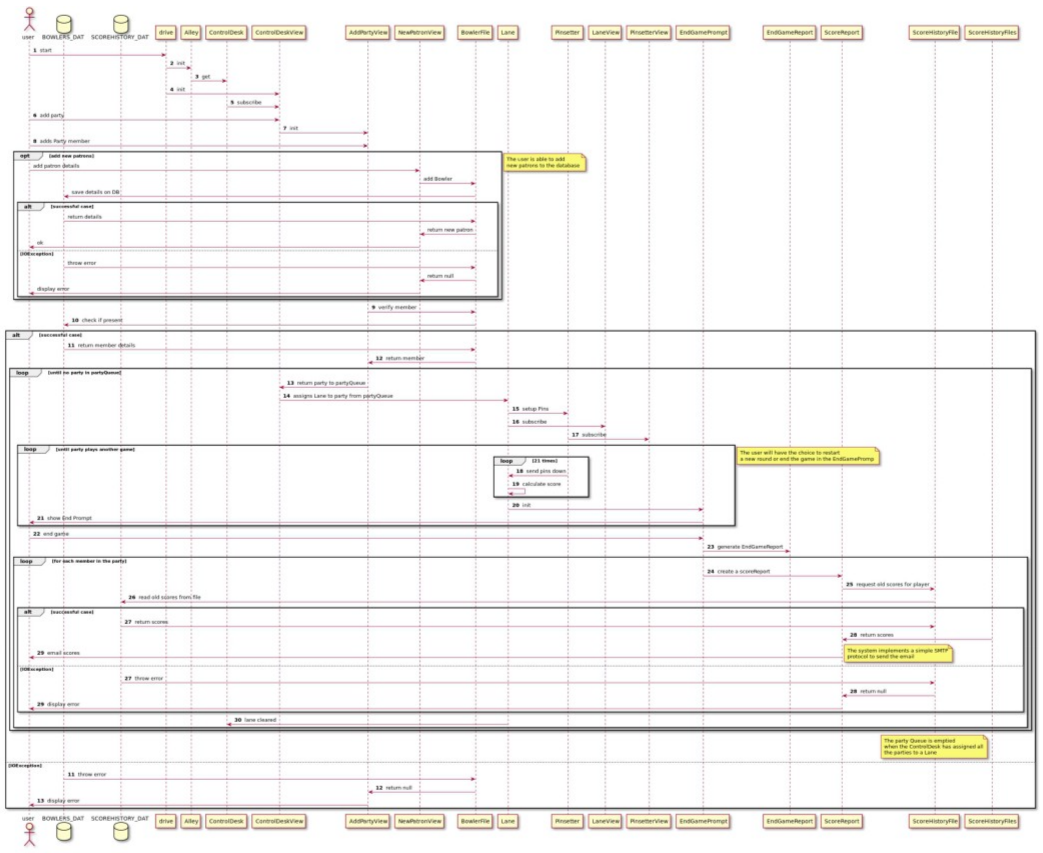


Figure 2: UML Sequence Diagram (Complete)

complete UML Sequence diagram also includes the optional steps that the user may take and the responses to those. It also accounts for alternative cases and error messages.

The complete picture can be found in both PNG and SVG format accompanied to this pdf with the name *UMLSequence.png* or *UMLSequence.svg*.

4 Classes Overview

The following is the overview of all the classes and their responsibilities.

Classes	Responsibilities
AddPartyView	This is the constructor class for the GUI to add parties to the waiting party queue. Interacts with ControlDeskView and NewPatronView
Alley	Class that is the outer container for the bowling simulater
Bowler	Class with all the bowler info
BowlerFile	Class that deals with storing and retrieving the data from the BOWLERS_DAT file
ControlDesk	Class that represents the Control Desk. Interacts with the ControlDeskView. It initializes Lanes and assigns parties to it.
ControlDeskEvent	Represents the state of the ControlDesk. Sent as state update to the subscribers of ControlDesk.
ControlDeskObserver	Interface for classes that observe ControlDeskEvents.
ControlDeskView	Displays a GUI of the ControlDesk. Interacts with the ControlDeskObserver.
drive	Class to begin the application. Calls the ControlDesk and subscribes a ControlDeskView to it.
EndGamePrompt	Constructor for prompt GUI that asks the party whether they would like to play another round.
EndGameReport	A GUI that shows the user the members of the party and allows him to print scores for players.
Lane	Class for the lane object that is assigned to a party. Implements the Pinsetter Observer. It runs the simulation and assigns the score to the bowlers.
LaneEvent	Represents the state of the Lane. Sent as an update to the subscribers of Lane.
LaneEventInterface	Interface to implement Java RMI for LaneEvent
LaneObserver	Interface that is implemented by the other classes like LaneView and LaneStatusView to get the LaneEvent through the receive function
LaneServer	The class that keeps record of the status of the Lane

Table 3: Classes Overview (Part 1)

Classes	Responsibilities
LaneStatusView	Constructor for GUI that the shows the status of the Lane. Implements the LaneObserver, and PinsetterObserver
LaneView	Constructs a GUI for Lane. Shows the score for the bowler. Implements the LaneObserver
NewPatronView	Constructor for GUI to add a new Patron. Called from the AddPatryView
Party Lane.	Container Class that contains all the bowlers. Sends this information to the ControlDesk and the
Pinsetter	Class to represent the pinsetter. Simulates the actual knocking down using random variables. Sends the result of knocked down pins to the subscribers through the PinsetterEvent.
PinsetterEvent	Represents the state of the pinsetter. Is sent to the subscribers as a state update.
PinsetterObserver	Interface that is implemented by classes to receive the PinsetterEvent and interpret it.
PinsetterView	Constructor GUI to represent the functioning of the pinsetter. Shows the pins standing up and knocked down at any moment. Implements the PinsetterObserver
PrintableText	Implements Printable. Used for setting up the printing on paper.
Queue Score	Utility class that implements queues using Vectors. Class that represents the score of each bowler.
ScoreHistoryFile	Class that deals with the reading and storing of scores in the SCOREHISTORY_DAT file.
ScoreReport	Class that generates the report based on the scores of the bowler. It also deals with sending email and printing out the report to the user.

Table 4: Classes Overview (Part 2)

5 Narrative (Old Code)

We began by understanding the original codebase and as a result of the same, we got to analyse the codebase quite well. The initial code of the LSBC had a working model of a bowling alley. It had features such as: a basic UI for the ControlDesk containing 3 lanes, a party queue, and controls such as Add Party and Finished. Then a bowler database from which the user can add members to the party queue, remove members from the party queue and add new patrons to the database. The ControlDesk also can see who is currently Bowling, the number of pins down in each lane and also had buttons to view the Lane scoreboard and the Pinsetter. The Lane option also has a maintenance call button to alert the ControlDesk who will trigger the lane functional once the issue is fixed. It also has an option to print the report once the game is finished or play a new game.
The Files implemented include AddPartyView (for adding party queue), Bowler (containing bowler info), ControlDesk files to manage the ControlDesk and all its functionalities, drive (the main starting file), EndGame files having the UI for prompt to play New game or print report, Lane files which manage the functinalities in a lane, PinSetter files to take care of the game pins and their UI, Score files which store the scores and also help generate the report.
This is the UML class diagram representing all the links between classes: [1] UML Class Diagram.

5.1 Strengths

The original codebase does have implementation of a couple of design patterns like Observer Pattern and Adapter Pattern. The Observer Pattern can be observed in the form of a subscription based setup where various event based statuses are broadcast to all objects subscribing for the same. Moreover, Adapter Pattern can be observed during interactions with the file database through the PrintableText Class module. These comprise a few of the strengths of the original codebase.

5.2 Weakness

While most of the code was working and in good condition there were few code smells and some code metrics values were higher than the expected

ranges.
1 class with *medium-high* **Complexity**
3 class with *medium-high* **Lack of Cohesion**
1 class with *medium-high* **Weighted method Count**

First of all we noticed that, the original codebase had a couple of unnecessarily long and complicated classes including the LaneView Class and the GameReport Class. It was clear in these modules that these could be modularised further into smaller classes. We observed Low Cohesion in these classes.
We also found a couple of dead modules or dead classes in the codebase. This includes the LaneServer module and the LaneViewInterface module. We realized that the LaneServer is not quite required in the codebase. However, the LaneViewInterface could very well be used in the codebase.
One more factor which made the code seem unnecessarily complex was its Cyclomatic Complexity, which arises due to there being too many conditions on the same parameter. We found out that many of these conditional statements were actually unnecessary. Moreover, in a few cases, cases of unnecessary nesting of conditional statements were observed.Since the original codebase's Design Document gave very little implementation level information, it may not be fair to compare the satisfied functionalities of the original codebase with the deliverables as defined by the Design Document.

The different types of code smells in the files include Dead code, Conditional Complexity, Duplicate code, Combinatorial Explosion, Indecent Exposure, and Inconsistent Names.

6 Code Smells (Original Code)

[Note : Deprecation, in its programming sense, is the process of taking older code and marking it as no longer being useful within the codebase, usually because it has been superseded by newer code. The deprecated code is not immediately removed from the codebase because doing so may cause regression errors]

File	Smell type	Description	Code
AddPartyView	Dead Code	Unused import statement	import java.text.*
AddPartyView	Dead Code	Private field never used	private Integer Lock
AddPartyView	Dead Code	Variable not used	Insets buttonMargin
AddPartyView	Inconsistent Names	Deprecated Function	win.show()
AddPartyView	Conditional Complexity	Excessive Cyclomatic Complexity	public void actionPerformed(ActionEvent e) public
AddPartyView	Indecent Exposure	Returning direct reference to mutable field	Vector getNames() { return party; } public Vector
AddPartyView	Indecent Exposure	Returning direct reference to mutable field	getParty() { return party; }
BowlerFile	Combinatorial Explosion	More general exception is already used in the throws list	throws IOException, FileNotFoundException
ControlDesk	Dead Code	Empty catch block	catch (Exception e) {}
ControlDesk	Dead Code	Method never used	public void viewScores(Lane ln)
ControlDesk	Duplicate Code	Two different catch branches with the same body	catch (FileNotFoundException e) {} and catch (IOException e) { }
ControlDesk	Duplicate Code	Casting redundant	(Vector)partyQueue.asVector()
ControlDesk	Indecent Exposure	Returning direct reference to private mutable field	public HashSet getLanes() { return lanes; }
ControlDeskView	Dead Code	Unused import statement	import javax.swing.event.*;
ControlDeskView	Dead Code	Variable not used	AddPartyView addPartyWin = new AddPartyView(this, maxMembers)
ControlDeskView	Duplicate Code	Casting redundant	(Pinsetter)curLane.getPinsetter()
ControlDeskView	Duplicate Code	Casting redundant	(Vector) ce.getPartyQueue()
ControlDeskView	Inconsistent Names	Deprecated Function	win.hide()
ControlDeskView	Inconsistent Names	Deprecated Function	win.show()
drive	Dead Code	Unused import statement	import jva.util.Vector
Endgame Prompt	Dead Code	Unused import statement	import javax.swing.border.*;
Endgame Prompt	Dead Code	Unused import statement	import javax.swing.event.*;
Endgame Prompt	Dead Code	Unused import statement	import java.util.*;
Endgame Prompt	Dead Code	Unused import statement	import java.text.*;
Endgame Prompt	Dead Code	Private field never used	private String selectedNick
Endgame Prompt	Dead Code	Private field never used	private String selectedMember
Endgame Prompt	Dead Code	Variable not used	Insets buttonMargin = new Insets(4, 4, 4, 4);
Endgame Prompt	Inconsistent Names	Deprecated Function	win.show()
Endgame Prompt	Inconsistent Names	Deprecated Function	win.hide()
EndGame Report	Dead Code	Unused import statement	import java.text.*
EndGame Report	Dead Code	Private field never used	private Vector myVector;
EndGame Report	Dead Code	Variable not used	Insets buttonMargin = new Insets(4, 4, 4, 4);
EndGame Report	Dead Code	Variable not used	EndGameReport e = new EndGameReport(partyName,
EndGame Report	Inconsistent Names	C- Style array declaration	party); String args[]
Lane	Conditionl Complexity	Excessive Cyclomatic Complexity	public void run()
Lane	Conditionl Complexity	Excessive Cyclomatic Complexity	public void receivePinsetterEvent(PinsetterEvent
Lane	Conditionl Complexity	Excessive Cyclomatic Complexity	pe) { private int getScore(Bowler Cur, int frame)
Lane	Dead Code	Empty catch block	{ catch (Exception e) {}
Lane	Dead Code	Empty else block	else {}
Lane	Dead Code	Empty if block	if (i>1) {}
Lane	Dead Code	Method not used	Public boolean isGameFinished()
Lane	Wrong Syntax	String values compared using == and not equals()	if (thisBowler.getNick() == (String)printIt.next())
Lane	Indecent Exposure	Returning direct reference to private mutable field	public Pinsetter getPinsetter() { return setter }
LaneEventInterface	Duplicate Code	Modifier public is redundant for interface modifiers	
LaneEventInterface	Dead Code	Method not used	public int getFrameNum( ) throws java.rmi.RemoteException; public int[]
LaneEventInterface	Dead Code	Method not used	getCurScores( ) throws java.rmi.RemoteException; public int[][] getCumulScore()
LaneEventInterface	Dead Code	Method not used	throws java.rmi.RemoteException;
Lane StatusView	Dead Code	private field assigned but never used	private JLabel foul;
Lane StatusView	Dead Code	Variable not used	Insets buttonMargin = new Insets(4, 4, 4, 4);
Lane StatusView	Conditionl Complexity	Similiar code with slight variations	First two blocks in public void actionPerformed (ActionEvent e) {
Lane StatusView	Combinatorial Explosion	Excessive Cyclomatic Complexity	public void actionPerformed(ActionEvent e) {
LaneView	Dead Code	Variable not used	private int roll;
LaneView	Dead Code	Variable not used	int cur;
LaneView	Dead Code	Variable not used	Iterator bowlIt
LaneView	Dead Code	Variable not used	Insets buttonMargin = new Insets(4,4,4,4);
LaneView	Dead Code	Empty catch block	catch(Exception e){}
LaneView	Conditionl Complexity	Excessive Cyclomatic Complexity	public void receiveLaneEvent(LaneEvent le)
NewPatronView	Dead Code	Unused import statement	import javax.swing.event.*;
NewPatronView	Dead Code	Unused import statement	import java.util.*
NewPatronView	Dead Code	Unused import statement	import java.text.*
NewPatronView	Dead Code	Private field never used	private String selectedNick,
NewPatronView	Dead Code	Private field never used	selectedMember; private int maxSize;
NewPatronView	Dead Code	Variable not used	Insets buttonMargin = new Insets(4,4,4,4);
NewPatronView	Inconsistent Names	Deprecated Functions	win.show()
NewPatronView	Inconsistent Names	Deprecated Functions	win.hide()
PinsetterObserver	Duplicate Code	Modifier public is redundant for interface modifiers	public void receivePinsetterEvent(PinsetterEvent pe);
PinSetterView	Dead Code	Unused import statement	import java.awt.event.*
PinSetterView	Inconsistent Names	Deprecated Functions	frame.show(0
PinSetterView	Inconsistent Names	Deprecated Functions	frame.hide()
PrintableText	Dead Code	Unused import statement	import java.text.*
PrintableText	Inconsistent Names	C- style array declaration	String lines[] = text.split("\n");
Score	Dead Code	Method not used	public String getNickName()
ScoreHistoryFile	Combinatorial Explosion	More general exception is already used in the throws list	throws IOException, FileNotFoundException

7 Refactoring (Narrative)

7.1 Introduction

When refactoring we didn't only rely on the metric to evaluate the new refactored classes. We applied our personal intuition along with the metrics to evaluate whether a change was worth it or not. The idea was to generate a code that not only had good metrics, but one in which a slight loss in the metric score for logical flow would be appreciated.

Armed with those goal, we aimed to strike a balance among competing criteria such as low coupling, high cohesion, separation of concerns, information hiding, the Law of Demeter, extensibility, reusability, etc.

We achieved those by various different processes. Firstly, we reduced the complexity of functions separating them into smaller more specific functions through **Extract Method** process so as to achieve **Separation of concerns**. We also converted many large and confusing binary expressions to smaller function calls through **Pull Up Conditionals** process which allowed us to reuse them and reduce the complexity of conditionals.Thirdly, to reduce the complexity of other functions we changed the algorithm used so as to allow to reduce the complexity drastically. A notable example of this is the *getScore* function in Lane class.

Fourthly, to increase the cohesion within the class we remove unnecessary derived class variables, and replaced them with the method to get the variables itself. We also sifted through the functions and removed the functions with little or no use. This increased the cohesion further. We also extracted delegate classes using the **Extract Class** process, and reduced the complexity of the base class and increased the cohesion among the classes.

However, none of these were independent of each other, most of the positive changes in one metric caused a negative effect on another one. For example, extracting methods did help reduce the complexity but decreased cohesion. Similarly there were instances where the conditionals could be replaced by switch case or TYPE Objects but they weren't done because that would disrupt the logical flow among the functions.

In our code, we ensured data hiding by removing all unnecessary public functions and variables. We also ensured that in our methods most of the variables would be internal to the method and didn't depend on other methods. Only the required classes were public while rest were private. We also strictly followed the Law of Demeter to make sure that a method did not manipulate the variables of another object directly as all our functions used getters and setters.

Re-usability also played a key role in our development. For example in the Search module, the same modules for getting best result for one bowler is used again in the module for getting the best result overall. Similarly, in SearchView, the text field was reused for all the inputs.

We also tried our best to make the code as extensible as possible. This is also supported by our implementation of the first and second feature. We tried to keep in line with the application's current design pattern when implementing the new features to ensure that the codebase is as uniform as possible.

7.2 Lane

The Lane class had initially 3 high complexity functions namely — *run()*, *receivePinsetterEvent()* and *getScore()*.

The function *getScore()* had by far the highest complexity among the three, and was a major cause in making the class complex. We rewrote the algorithm for this function as the one that they were using was bad. After rewriting the function, we were able to decrease the complexity by 3 folds. We then refactored the new function and using the **Extract Method** procedure, we were able to again decrease the complexity of the new function significantly.

We then separated the *getScore* functions from the rest of the variables and functions from *Lane* class, and using the **Extract Delegate** process, we were able to escalate the function as a new class *ScoreCalculator*. This was highly beneficial in reducing the complexity of the *Lane* class, however it did slightly increase the coupling of the Lane class.

To further reduce the complexity of the Lane class, we refactored the *run()* function and using the **Extract Method** process we were able to extract 2 other functions from it, each with significantly lesser complexity than *run*. We also refactored the *receivePinsetterEvent()* function using the **Extract Method** process and were able to extract another method from it, reducing the complexity of both the functions.

To increase the cohesion between the methods, we removed one-time class variables that could be derived from other variables and which were used only in one function. We also found out *public* functions that were not being used anywhere in the project and removed those functions as well. This saw a measurable increase in the cohesion of the class.

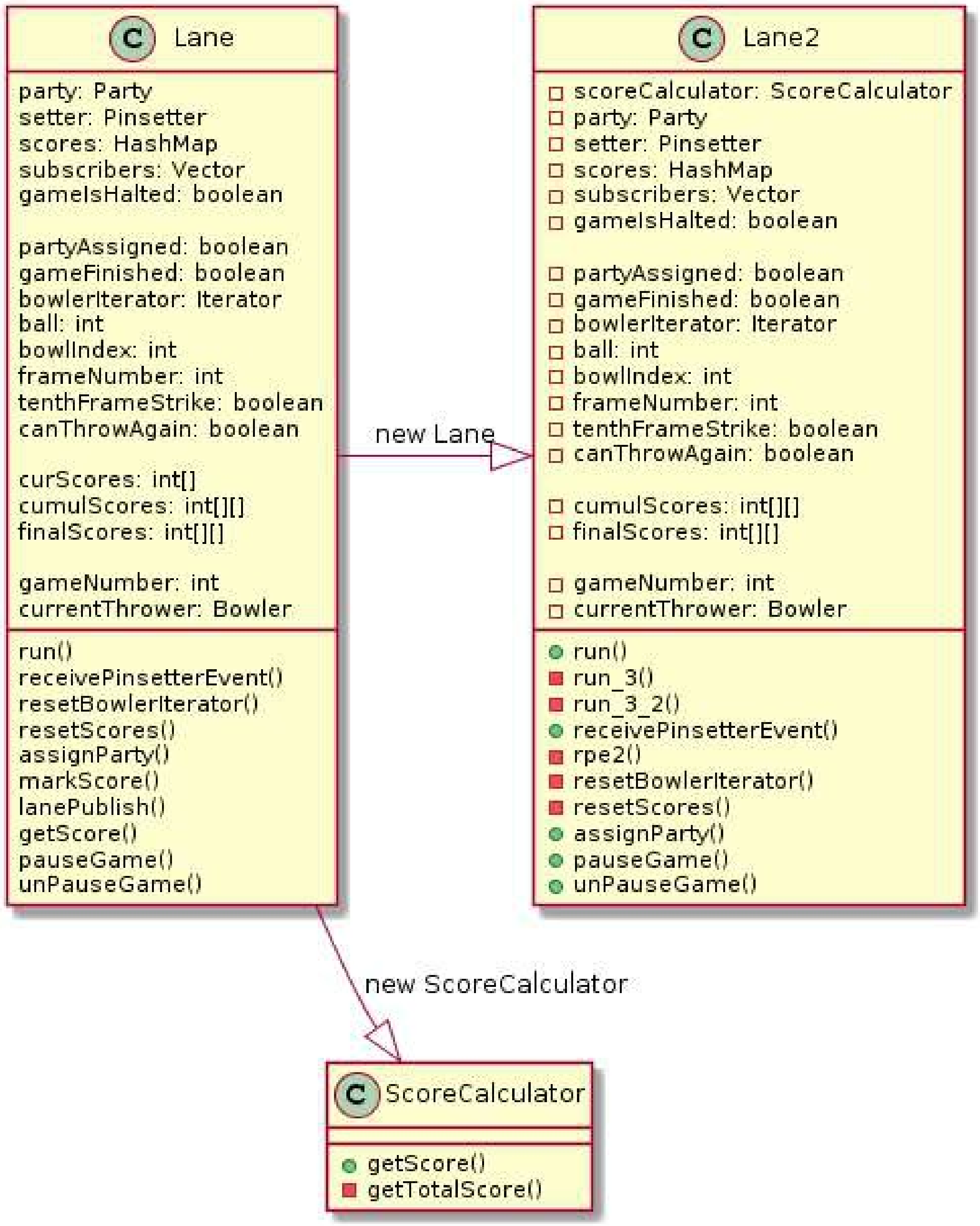


Figure 3: Class Diagram of the new Lane and ScoreCalculator class

8 Code Metrics

We chose the following metrics to assist us in analysing the initial code and finding its weaknesses:

8.1 Complexity Metrics:

8.1.1 Cyclomatic Complexity (v(G)):

This measures the number of possible independent paths through a method or function. If the cyclomatic complexity of a method is greater than 10, it would be very difficult to test. Thus, any such method should be broken down into multiple simpler methods

8.1.2 Essential Cyclomatic Complexity (ev(G)):

Essential Complexity (ev(G)) is a measure of the degree to which a module contains unstructured constructs. This metric measures the degree of structuredness and the quality of the code. It is used to predict the maintenance effort and to help in the modularization process. A method with high v(G) but low ev(G) can be easily refactored by dividing it into smaller methods but if ev(G) itself is high, the code is considered non-structured and demands major changes in logic.

8.1.3 Module Design Complexity Metric (iv(G)):

Module Design Complexity (iv(G)) is the complexity of the design-reduced module and reflects the complexity of the module's calling patterns to its immediate subordinate modules. This metric differentiates between modules which will seriously complicate the design of any program they are part of and modules which simply contain complex computational logic.

8.1.4 Weighted Method Count (WMC):

The weighted sum of all class' methods and' represents the McCabe complexity of a class. It is equal to number of methods, if the complexity is taken as 1 for each method. The number of methods and complexity can be used to predict development, maintaining and testing effort estimation. In inheritance if base class has high number of methods, it affects its' child classes and all methods are represented in subclasses. If number of methods is high, that class possibly domain specific. Therefore, they are less reusable. Also, these classes tend to more change and defect prone.

8.2 Coupling Metrics:

8.2.1 Coupling between objects (CBO):

The number of classes that a class is coupled to. It is calculated by counting other classes whose attributes or methods are used by a class, plus those that use the attributes or methods of the given class. Inheritance relations are excluded. As a measure of coupling CBO metric is related with reusability and testability of the class. More coupling means that the code becomes more difficult to maintain because changes in other classes can also cause changes in that class. Therefore, these classes are less reusable and need more testing effort.

8.2.2 Response for a Class (RFC):

The number of the methods that can be potentially invoked in response to a public message received by an object of a particular class. If the number of methods that can be invoked at a class is high, then the class is considered more complex and can be highly coupled to other classes. Therefore, more test and maintain effort is required.

8.2.3 Cyclic Dependencies (CYC):

It is a relation between two or more modules which either directly or indirectly depend on each other to function properly. Indicates tight coupling of the mutually dependent modules which reduces or makes impossible the separate re-use of a single module. Can be fixed by applying design patterns like the observer pattern.

8.3 Cohesion Metrics

8.3.1 Lack of Cohesion of Methods (LCOM):

Measure how methods of a class are related to each other. Low cohesion means that the class implements more than one responsibility. A change request by either a bug or a new feature, on one of these responsibilities will result in change of that class. Lack of cohesion also influences understandability and implies classes should probably be split into two or more subclasses.

8.4 Analysis of Code Metrics

8.4.1 Initial Analysis

Initially the only metrics that stand out are cohesion and complexity. This implies that the inheritance structure of the code is already pretty good. But high cohesion indicates that a lot of classes are performing multiple functions. Coming to complexity, a lot of methods have high cylcomatic complexity but most of them have low essential complexity. This indicates that the logic isn't that complex and the methods should be broken up into smaller ones.

8.4.2 Using metrics to guide refactoring

Lane class has very high WMC (72). On further inspecting the cyclomatic complexity for individual methods, we can see that run() and receivePinsetterEvent() are the methods with very high complexity but low ev(G). This indicates that these methods have a lot of conditions which can be converted to individual methods to achieve low cyclomatic complexity. We also see that many classes within the code lacked cohesion. In some cases this is unavoidable because getters and setters will always cause low cohesion. Thus we decided that classes like *LaneEvent* are fine even if they have low cohesion because it consists of just getters and setters. In some cases such as *ControlDesk*, the classes were performing multiple functions and needed to be split up.

8.4.3 Metrics after refactoring

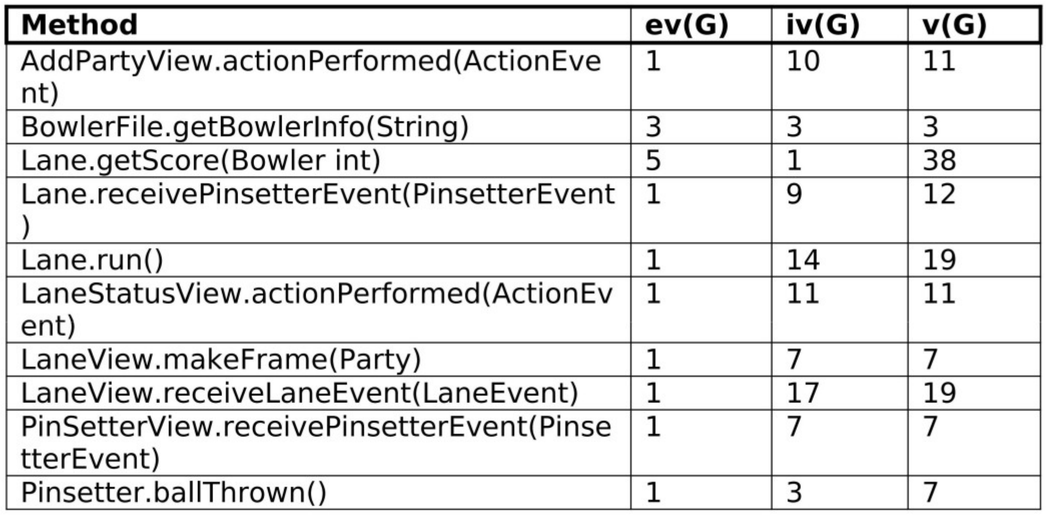
We were able to achieve low complexity in most classes and methods by appropriately splitting them up and rewriting the logic if required. We focused less on cohesion since java requires a lot of getters and setter to properly implement encapsulation. In lot of cases, to improved the readibility of code we had to create

8.5 Code Metrics Table

Here is the table of all the code metrics. It was taken using multiple plugins, including CodeMR and MetricsReloaded.

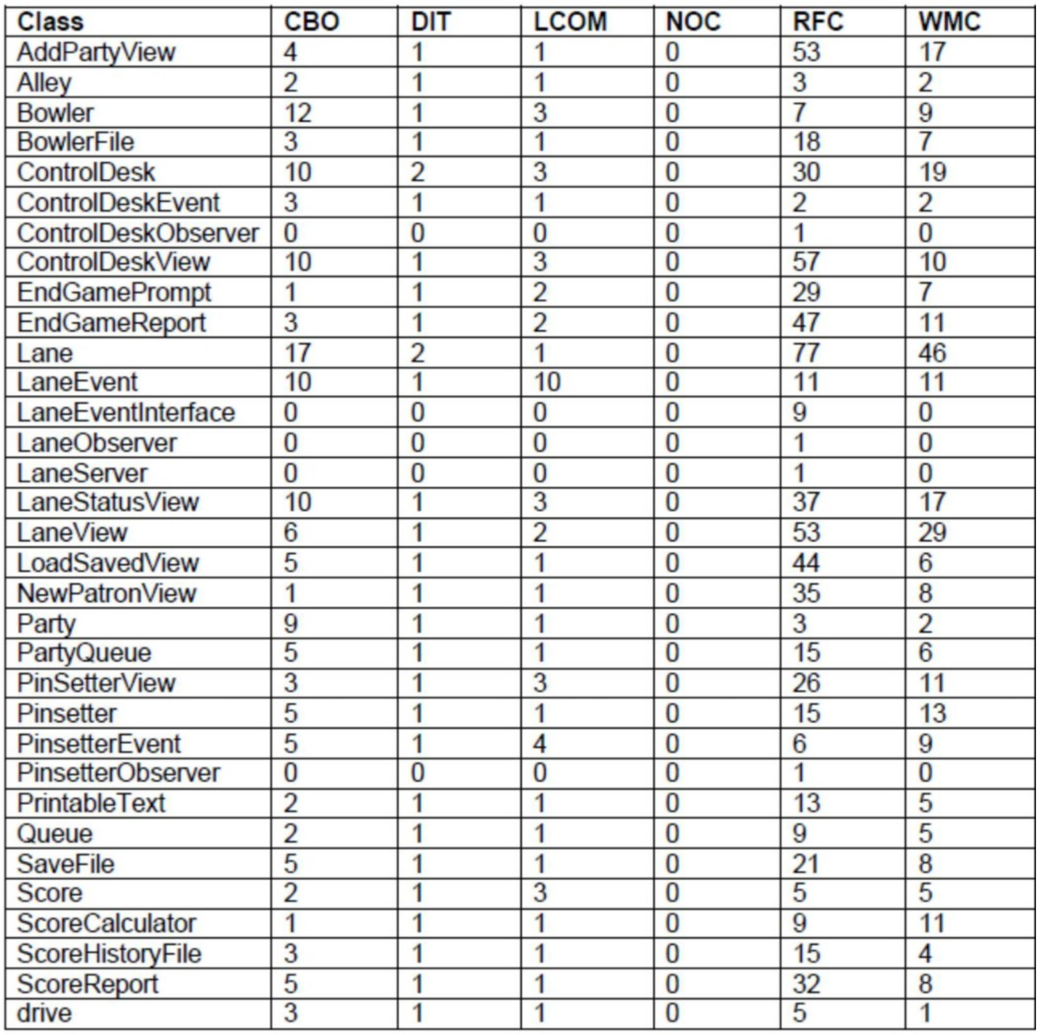
Class	CBO	LCOM	RFC	WMC	CYC
AddPartyView	4	1	53	17	2
Alley	2	1	3	2	0
Bowler	10	3	7	9	0
BowlerFile	3	1	17	6	0
ControlDesk	10	3	39	18	0
ControlDeskEvent	3	1	2	2	0
ControlDeskObserver	0	0	1	0	2
ControlDeskView	8	2	54	9	0
EndGamePrompt	1	2	29	7	0
EndGameReport	3	2	47	11	0
Lane	15	1	62	72	0
LaneEvent	6	10	11	11	0
LaneEventInterface	0	0	9	0	0
LaneObserver	0	0	1	0	2
LaneServer	0	0	1	0	0
LaneStatusView	10	3	35	17	0
LaneView	6	1	49	25	0
NewPatronView	1	1	35	8	0
Party	6	1	3	2	0
PinSetterView	3	3	26	11	0
Pinsetter	5	1	15	13	0
PinsetterEvent	5	4	6	9	0
PinsetterObserver	0	0	1	0	0
PrintableText	2	1	13	5	0
Queue	1	1	9	5	0
Score	2	3	5	5	0
ScoreHistoryFile	3	1	15	4	0
ScoreReport	5	1	32	8	0
drive	3	1	5	1	0

Figure 4: Code Metrics Table 1



Method	ev(G)	iv(G)	v(G)
AddPartyView.actionPerformed(ActionEvent)	1	10	11
BowlerFile.getBowlerInfo(String)	3	3	3
Lane.getScore(Bowler int)	5	1	38
Lane.receivePinsetterEvent(PinsetterEvent)	1	9	12
Lane.run()	1	14	19
LaneStatusView.actionPerformed(ActionEvent)	1	11	11
LaneView.makeFrame(Party)	1	7	7
LaneView.receiveLaneEvent(LaneEvent)	1	17	19
PinSetterView.receivePinsetterEvent(PinsetterEvent)	1	7	7
Pinsetter.ballThrown()	1	3	7

Figure 5: Code Metrics Table 2



Class	CBO	DIT	LCOM	NOC	RFC	WMC
AddPartyView	4	1	1	0	53	17
Alley	2	1	1	0	3	2
Bowler	12	1	3	0	7	9
BowlerFile	3	1	1	0	18	7
ControlDesk	10	2	3	0	30	19
ControlDeskEvent	3	1	1	0	2	2
ControlDeskObserver	0	0	0	0	1	0
ControlDeskView	10	1	3	0	57	10
EndGamePrompt	1	1	2	0	29	7
EndGameReport	3	1	2	0	47	11
Lane	17	2	1	0	77	46
LaneEvent	10	1	10	0	11	11
LaneEventInterface	0	0	0	0	9	0
LaneObserver	0	0	0	0	1	0
LaneServer	0	0	0	0	1	0
LaneStatusView	10	1	3	0	37	17
LaneView	6	1	2	0	53	29
LoadSavedView	5	1	1	0	44	6
NewPatronView	1	1	1	0	35	8
Party	9	1	1	0	3	2
PartyQueue	5	1	1	0	15	6
PinSetterView	3	1	3	0	26	11
Pinsetter	5	1	1	0	15	13
PinsetterEvent	5	1	4	0	6	9
PinsetterObserver	0	0	0	0	1	0
PrintableText	2	1	1	0	13	5
Queue	2	1	1	0	9	5
SaveFile	5	1	1	0	21	8
Score	2	1	3	0	5	5
ScoreCalculator	1	1	1	0	9	11
ScoreHistoryFile	3	1	1	0	15	4
ScoreReport	5	1	1	0	32	8
drive	3	1	1	0	5	1

Figure 6: Final Metrics