Design Document- Unit 1 project

Bowling Alley Simulation Refactoring

Team: 31

Submission date : 20/02/2022

Team Members

SNo.	Team Member	Roll no.	Effort (hours)	Role
1.	Aditya Vishnu Tiwari	2021202029	12	Class diagrams , Documentation
2.	Aniket Vinod Chandekar	2021204001	12	UML, Documentation
3.	Krishnapriya Panicker	2020202020	12	Sequence diagrams, Documentation
4.	Sandeep Deva Misra	2021202026	12	UML,Refactoring to remove code smells, Analyze metrics to improve
5.	Shambhavi Ojha	2021204011	12	Refactoring to remove code smells, Analyze metrics to Improve, Documentation

I. Introduction:

This document discusses the Bowling Management Alley Program's original and refactored designs.
The programme is written in Java. The code is a Bowling Alley simulator, in which the system simulates a control panel with the functionality of adding a new patron, creating bowling parties, assigning lanes, keeping scores, displaying the pinsetter visuals, displaying the lane status and score, and a slew of other related features.

The Bowling Alley's original design was provided with a high-level Software Requirement Specification that detailed the system's high-level functionality. No documentation was available other than comments in the codebase.

We intend to reverse engineer the given codebase into its original design, refactor the original design, and implement the refactoring into codebase in this project.

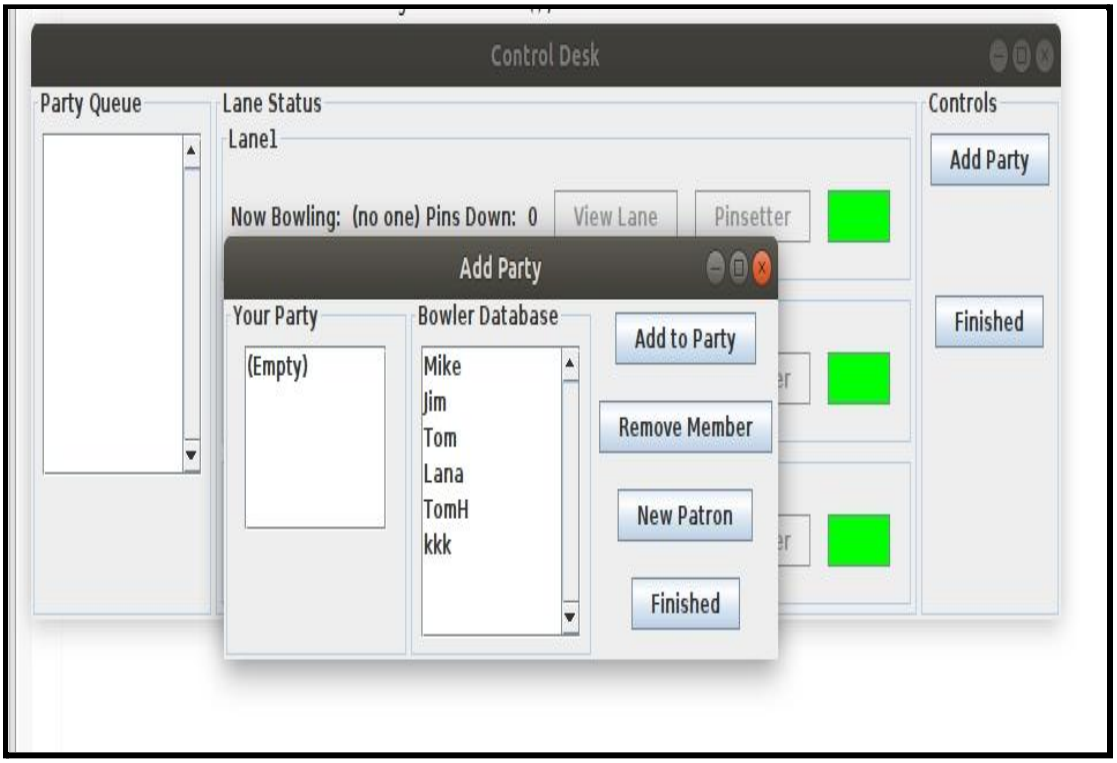
II. Brief Overview:

The Bowling Alley Simulation is a virtual bowling game. It is developed entirely in Java and has implemented the MVC framework.

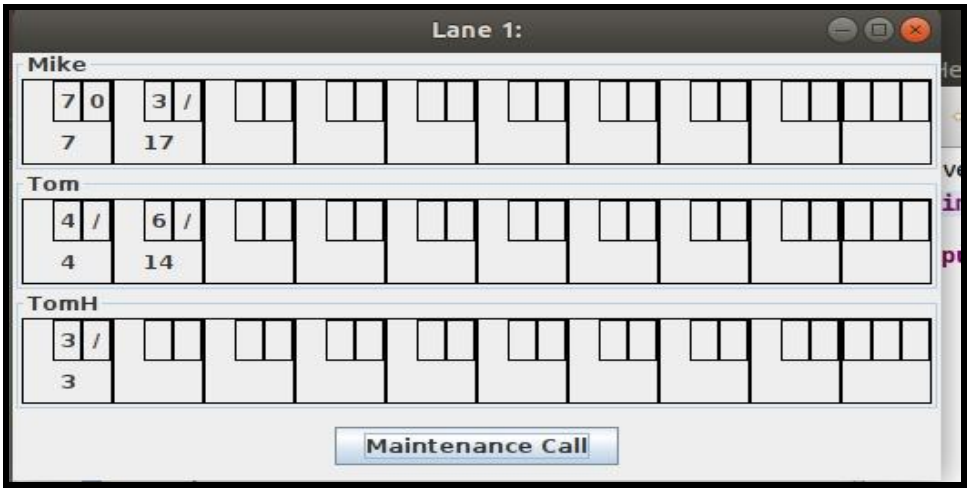
The game enables players to virtually play the bowling game. The basic functionalities of the game are:

- There are multiple lanes that enable multiple parties/teams to enjoy the game simultaneously.

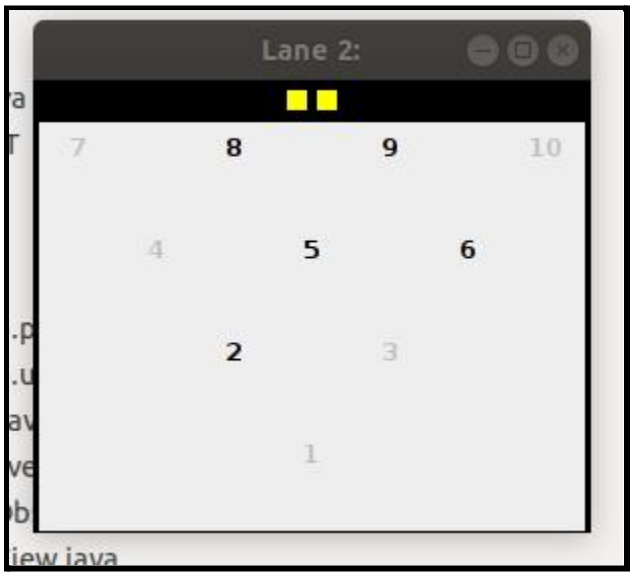
- Parties can be added to the lane. In case all lanes are full, the newly added party goes to a queue. The Queue maintains the track of parties registered but yet to play.
- Each party can consist of one to multiple players.
- While adding a party to lane, new players can be added.



- 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



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



- While viewing the lane of any of the parties, maintenance calls can also be done. The game is virtually halted for that duration. Maintenance calls can be related to any kind of repair work or a problematic situation like pin setter not re-racked, ball not returned or score not updated.(Red colored rectangle indicates that maintenance call was made for that lane.)

- 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. UML Diagrams:

1. Class Diagram (Before Refactoring):

UML CLASS DIAGRAMS (Before Refactoring):

The UML diagrams below describe some of the game's major functionalities.

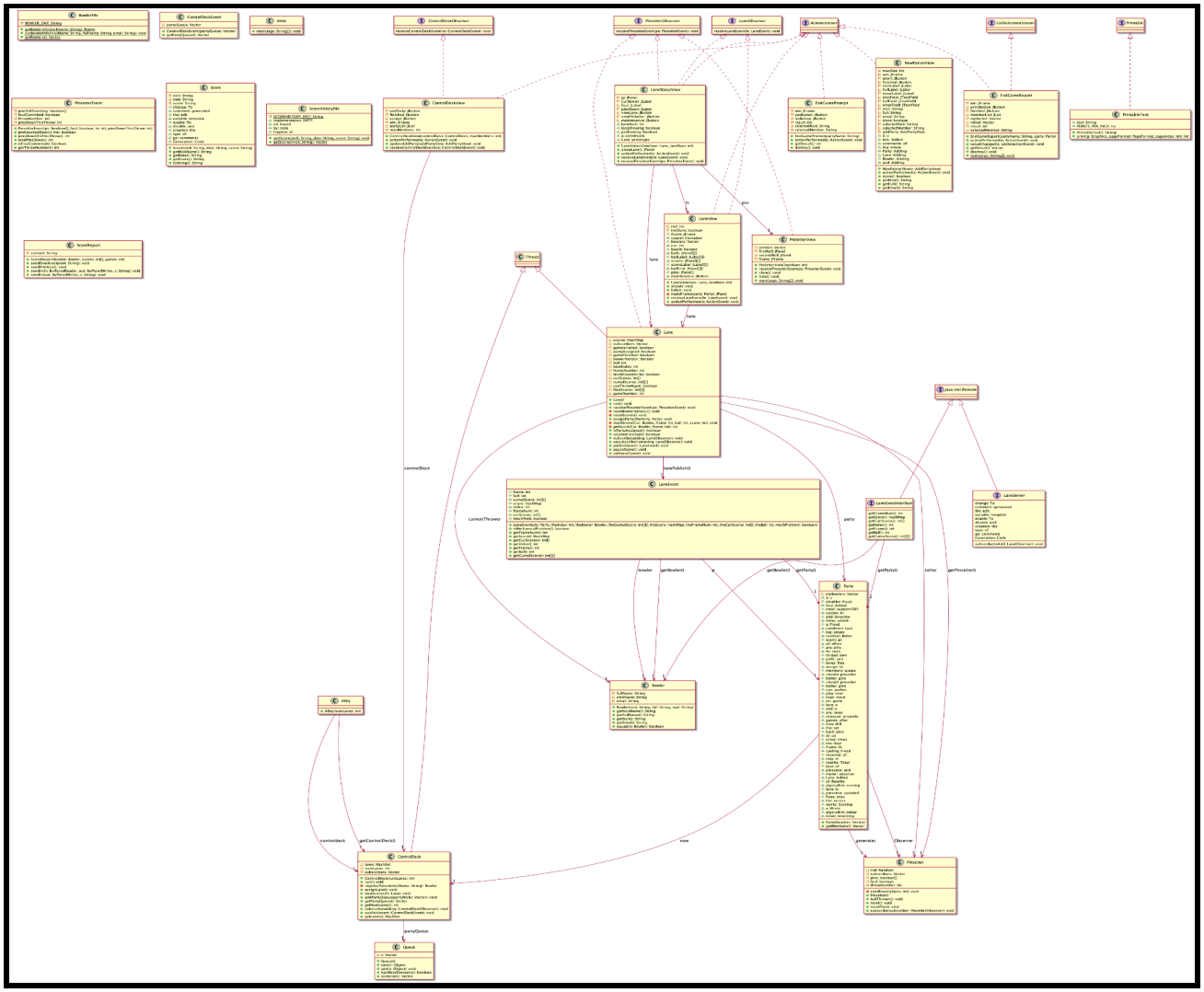
The various member classes that comprise the game's components exhibit a variety of relationships. Among these are:

1. **Association:** Demonstrates a connection between the two classes. Objects from another class may be used within one of the classes. This is indicated by a bold arrow line.
2. **Dependency:** It can also show a dependency between two or more classes in some cases. Any changes made to a class may result in changes to the classes that rely on it.
3. **Composition:** This represents the relationship between two classes in which one class "is entirely made of another class," i.e. one of the classes cannot exist unless the parent/main class object exists. At the end of the parent class, an arrow with a darkened diamond represents this.
4. **Aggregation** is similar to the "part of" relationship between two classes. For example, one class is "a subset" of another.

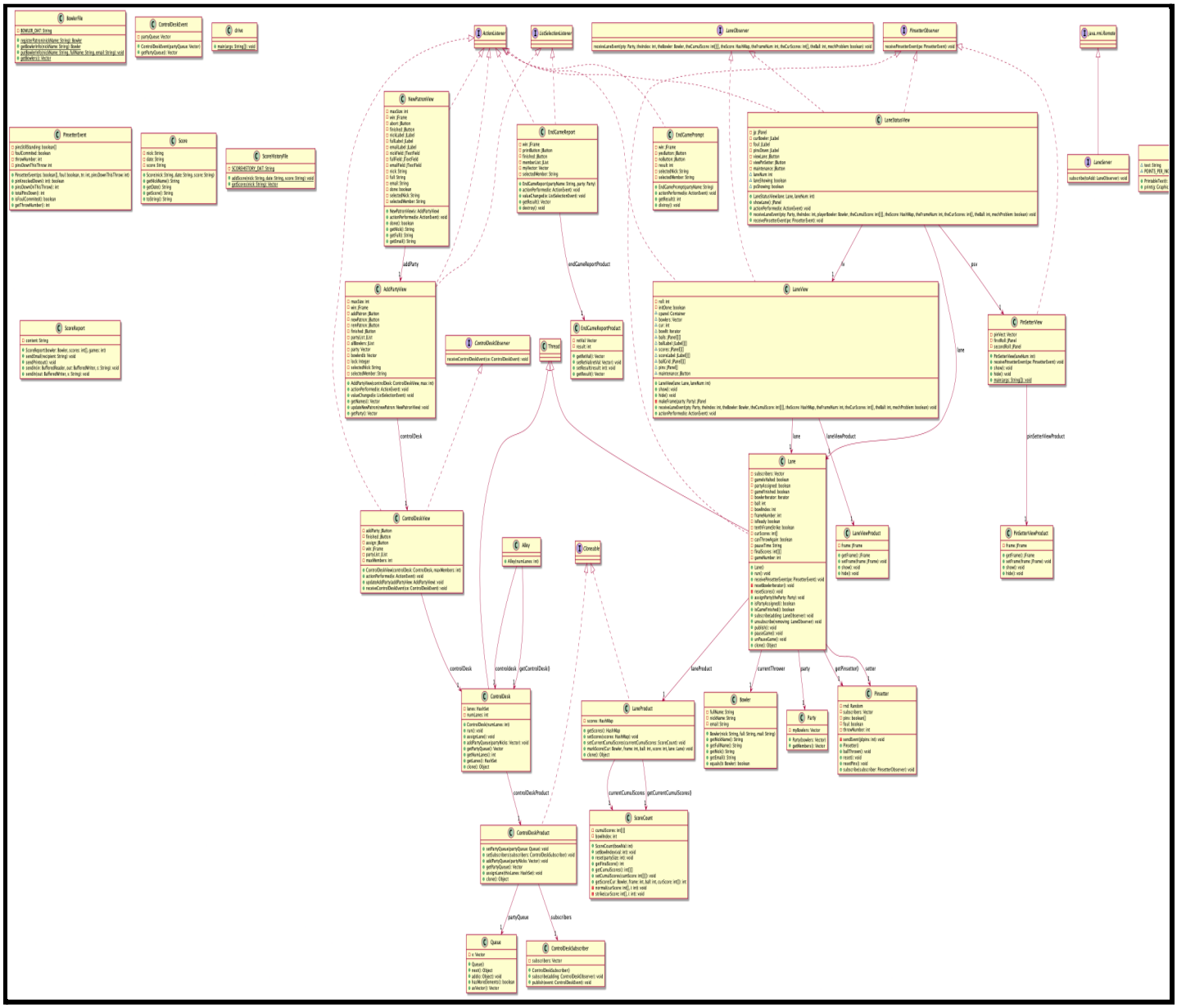
5. **Generalization:** When one class generalizes the functionalities of all its subclasses, this term is used. That is, it acts as a container for other classes, which inherit all of the parent's properties as well as some of their own.
6. **Specialization:** The polar opposite of generalization is specialization. A specialized class is a subclass that inherits all of its parent class's properties and adds some of its own. All specific classes will fall under the umbrella of a single main/parent class.

The Bowling Management System consists of 29 classes that are derived from classes such as Thread, ActionEvent, Serializable, and so on. Each class is contained in its own file. Each class has attributes and methods that describe its functionality. The Bowling Management System classes interact with one another to simulate the operation of a Bowling Alley, the Control Desk, and the Games.

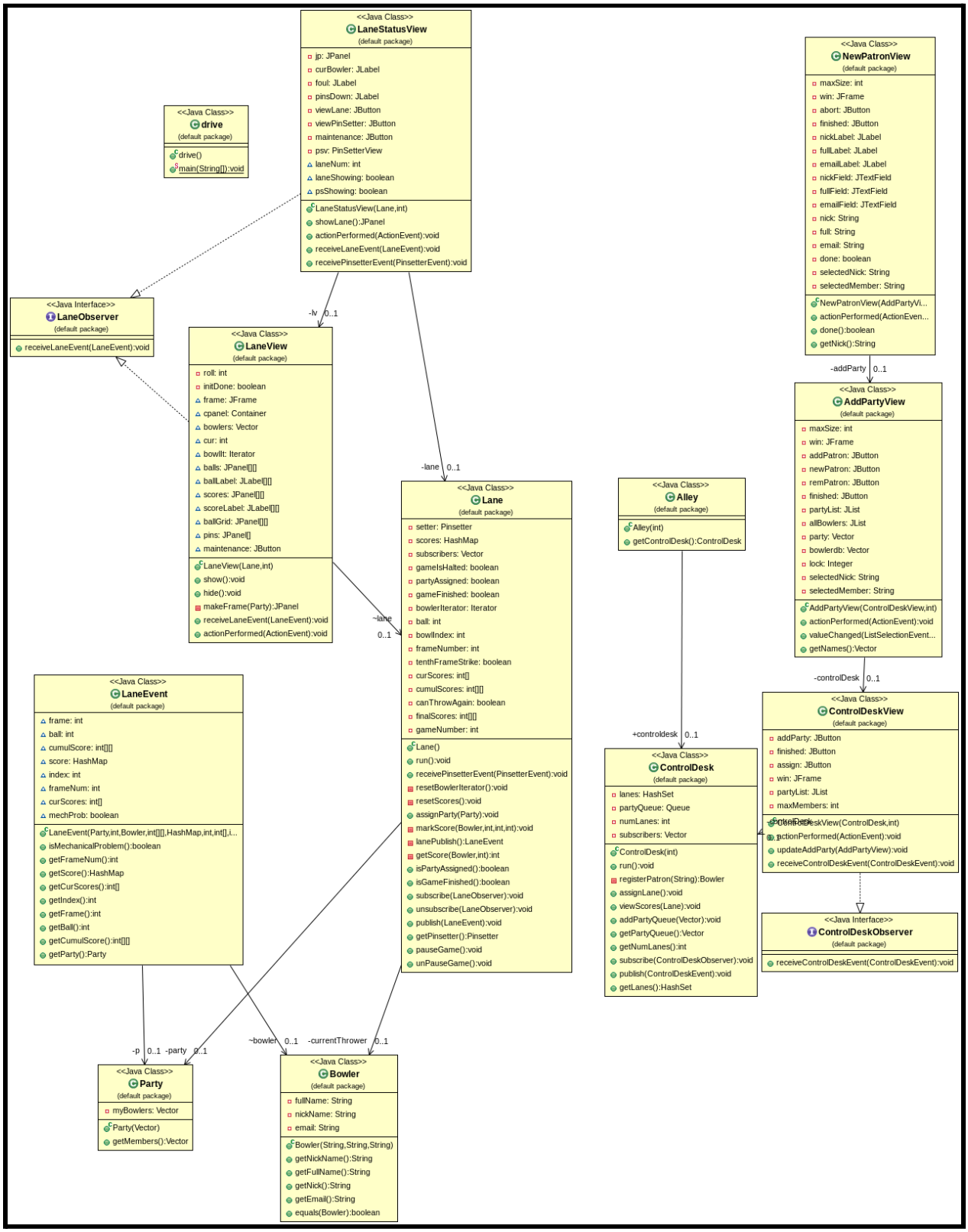
UML Diagram(Before Refactoring):



UML Diagram(After Refactoring):



UI Components:



Lane related Components:

These are the classes involved:

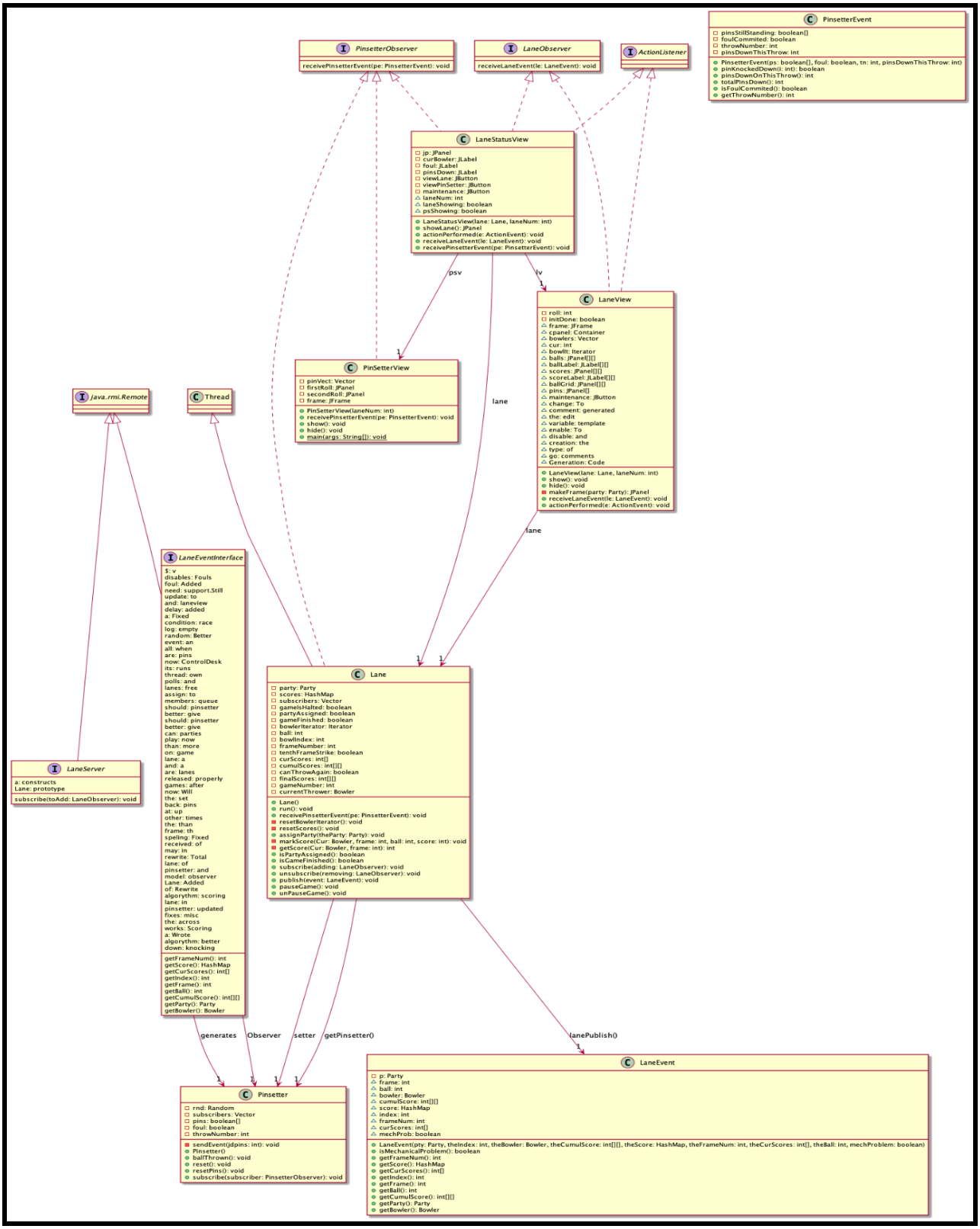
- Lane
- LaneEvent
- LaneObserver
- LaneServer
- LaneView
- LaneStatusView
- LaneEventInterface
- Pinsetter
- PinsetterObserver
- PinsetterEvent
- PinSetterView

The diagram can be summarized as follows:

1. The PinsetterObserver, LaneObserver, and LaneEventInterface classes are interface classes. Essentially, it acts as an interface between two or more classes that would otherwise be unable to interact. For example, the PinSetterView and Pinsetter classes are interfaced in order to share information and perform functions. LaneStatusView and LaneView are also linked to LaneEvent via LaneObserver. This is also a demonstration of the Adapter Design Pattern.
2. The cardinalities - the number of participating objects in any association between two classes - are indicated on the class relationship arrows. It has also been specified which class is created from which parent class to further indicate the creation of classes. The parent-child relationship is indicated by the create written above the arrows.
3. The UML class diagram also depicts all of the methods and attributes associated with each class. Private attributes and methods are denoted by -, while public ones are denoted by +.

The various arrows in the diagram represent various relationships between the various classes, which are as follows.

- Association and Dependencies:
    - The Pinsetter & Lane and PinsetterView & LaneStatusView classes are linked together by a solid arrow, indicating that they share objects from their respective classes.
    - **Dependencies are represented by dotted arrows, which indicate that a change in the class to which the arrow head points will and can result in a change in the dependent class.**


PinsetterObserver
receivePinsetterEvent(pe: PinsetterEvent): void
LaneObserver
receiveLaneEvent(le: LaneEvent): void
ActionListener
PinsetterEvent
pinsStillStanding: boolean[]
foulCommited: boolean
throwNumber: int
pinsDownThisThrow: int
PinsetterEvent(ps: boolean[], foul: boolean, tn: int, pinsDownThisThrow: int)
pinKnockedDown(i: int): boolean
pinsDownOnThisThrow(): int
totalPinsDown(): int
isFoulCommited(): boolean
getThrowNumber(): int
LaneStatusView
jp: JPanel
curBowler: JLabel
foul: JLabel
pinsDown: JLabel
viewLane: JButton
viewPinSetter: JButton
maintenance: JButton
laneNum: int
laneShowing: boolean
psShowing: boolean
LaneStatusView(lane: Lane, laneNum: int)
showLane(): JPanel
actionPerformed(e: ActionEvent): void
receiveLaneEvent(le: LaneEvent): void
receivePinsetterEvent(pe: PinsetterEvent): void
psv
lv
LaneView
roll: int
initDone: boolean
frame: JFrame
cpanel: Container
bowlers: Vector
cur: int
bowlIt: Iterator
balls: JPanel[][]
ballLabel: JLabel[][]
scores: JPanel[][]
scoreLabel: JLabel[][]
ballGrid: JPanel[][]
pins: JPanel[]
maintenance: JButton
change: To
comment: generated
the: edit
variable: template
enable: To
disable: and
creation: the
type: of
go: comments
Generation: Code
LaneView(lane: Lane, laneNum: int)
show(): void
hide(): void
makeFrame(party: Party): JPanel
receiveLaneEvent(le: LaneEvent): void
actionPerformed(e: ActionEvent): void
PinSetterView
pinVect: Vector
firstRoll: JPanel
secondRoll: JPanel
frame: JFrame
PinSetterView(laneNum: int)
receivePinsetterEvent(pe: PinsetterEvent): void
show(): void
hide(): void
main(args: String[]): void
lane
Java.rmi.Remote
Thread
LaneEventInterface
$: v
disables: Fouls
foul: Added
need: support.Still
update: to
and: laneview
delay: added
a: Fixed
condition: race
log: empty
random: Better
event: an
all: when
are: pins
now: ControlDesk
its: runs
thread: own
polls: and
lanes: free
assign: to
members: queue
should: pinsetter
better: give
should: pinsetter
better: give
can: parties
play: now
than: more
on: game
lane: a
and: a
are: lanes
released: properly
games: after
now: Will
the: set
back: pins
at: up
other: times
the: than
frame: th
spelling: Fixed
received: of
may: in
rewrite: Total
lane: of
pinsetter: and
model: observer
Lane: Added
of: Rewrite
algorythm: scoring
lane: in
pinsetter: updated
fixes: misc
the: across
works: Scoring
a: Wrote
algorythm: better
down: knocking
getFrameNum(): int
getScore(): HashMap
getCurScores(): int[]
getIndex(): int
getFrame(): int
getBall(): int
getCumulScore(): int[][]
getParty(): Party
getBowler(): Bowler
Lane
party: Party
scores: HashMap
subscribers: Vector
gameIsHalted: boolean
partyAssigned: boolean
gameFinished: boolean
bowlerIterator: Iterator
ball: int
bowlIndex: int
frameNumber: int
tenthFrameStrike: boolean
curScores: int[]
cumulScores: int[][]
canThrowAgain: boolean
finalScores: int[][]
gameNumber: int
currentThrower: Bowler
Lane()
run(): void
receivePinsetterEvent(pe: PinsetterEvent): void
resetBowlerIterator(): void
resetScores(): void
assignParty(theParty: Party): void
markScore(Cur: Bowler, frame: int, ball: int, score: int): void
getScore(Cur: Bowler, frame: int): int
isPartyAssigned(): boolean
isGameFinished(): boolean
subscribe(adding: LaneObserver): void
unsubscribe(removing: LaneObserver): void
publish(event: LaneEvent): void
pauseGame(): void
unPauseGame(): void
LaneServer
a: constructs
Lane: prototype
subscribe(toAdd: LaneObserver): void
generates
Observer
setter
getPinsetter()
lanePublish()
Pinsetter
rnd: Random
subscribers: Vector
pins: boolean[]
foul: boolean
throwNumber: int
sendEvent(jdpins: int): void
Pinsetter()
ballThrown(): void
reset(): void
resetPins(): void
subscribe(subscriber: PinsetterObserver): void
LaneEvent
p: Party
frame: int
ball: int
bowler: Bowler
cumulScore: int[][]
score: HashMap
index: int
frameNum: int
curScores: int[]
mechProb: boolean
LaneEvent(pty: Party, theIndex: int, theBowler: Bowler, theCumulScore: int[][], theScore: HashMap, theFrameNum: int, theCurScores: int[], theBall: int, mechProblem: boolean)
isMechanicalProblem(): boolean
getFrameNum(): int
getScore(): HashMap
getCurScores(): int[]
getIndex(): int
getFrame(): int
getBall(): int
getCumulScore(): int[][]
getParty(): Party
getBowler(): Bowler

Control Desk related Components:

This UML diagram depicts the functionality of creating a new party for a game and assigning it a specific lane. The bowlers can be chosen from an existing list or added as a NewPatron.
The following classes are involved:

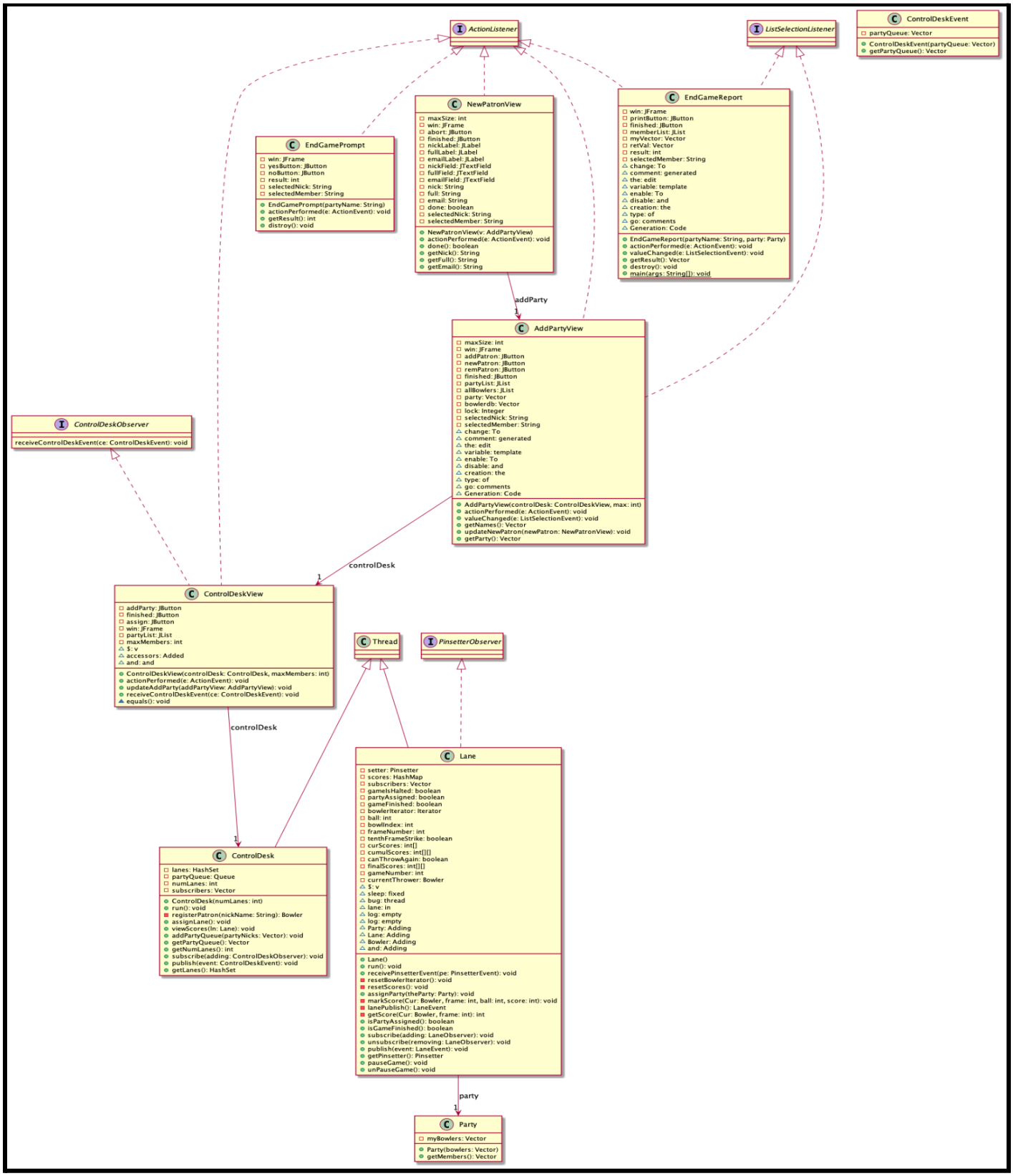
1. ControlDesk
2. ControlDeskObserver (Interface)
3. ControlDeskEvent
4. ControlDeskView
5. Bowler
6. Party
7. NewPatronView
8. AddPartyView
9. EndGameReport
10. EndGamePrompt
11. Lane
12. Alley
13. Drive

The diagram can be summarized as follows:

- The ControlDeskObserver class is a type of interface. Essentially, it acts as a bridge between two or more classes that would otherwise be unable to interact. The ControlDeskView and ControlDeskEvent classes, for example, are interfaced in order to share information and perform functions. This is also an example of the Adapter Design Pattern.
- For the game to begin, the drive class is the driver module. It is linked to the ControlDesk class, which handles all game-control functions.
- On the relationship arrows of the classes, the cardinalities (the number of participating objects in any association between two classes) are also mentioned. It has also been specified which class is created from which parent class to further indicate the creation of classes. The parent-child relationship is indicated by the create written above the arrows.
- The UML class diagram also depicts all of the methods and attributes associated with each class. Private attributes and methods are denoted by -, while public ones are denoted by +.

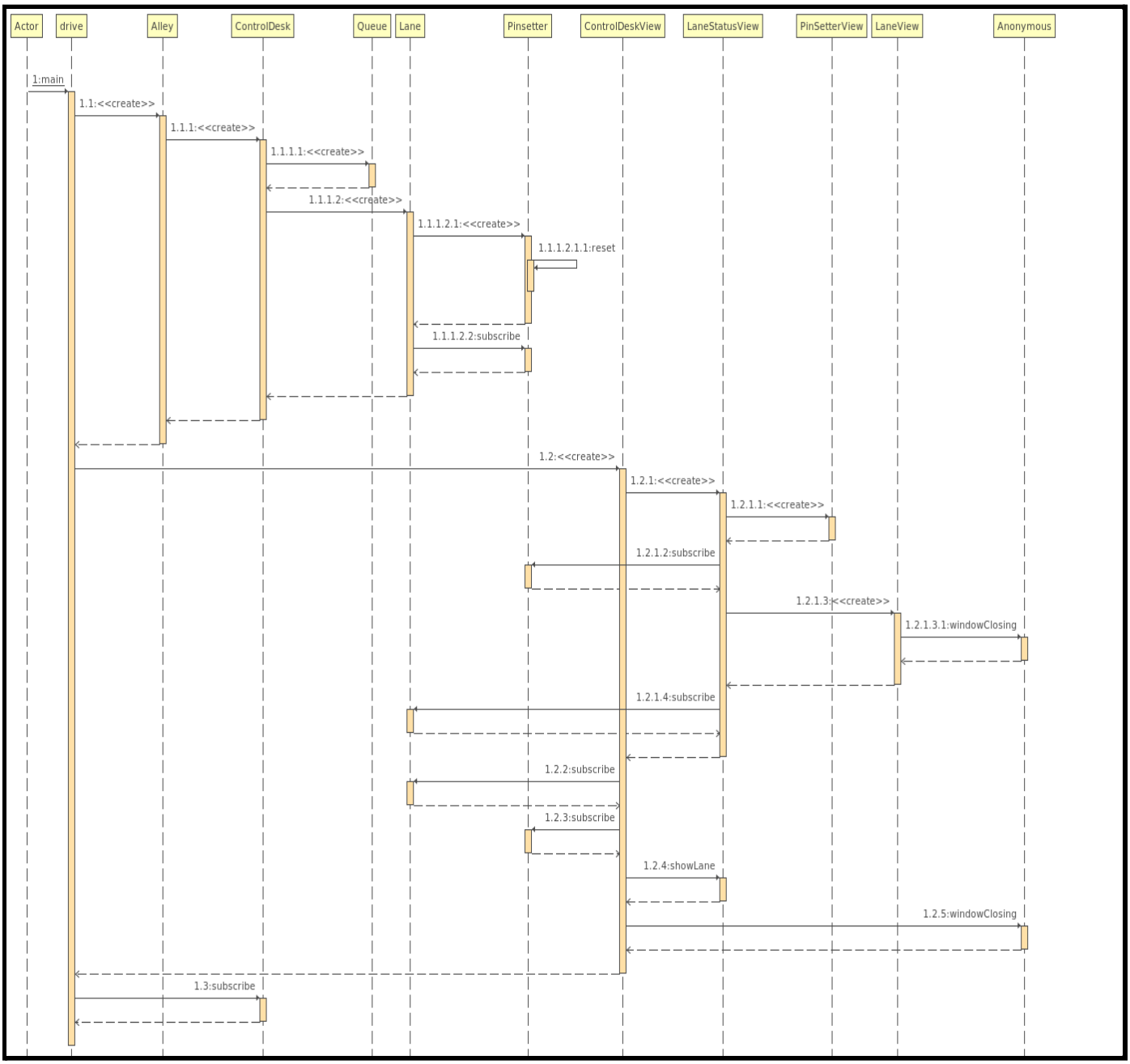
The various arrows in the diagram represent various relationships between the various classes, which are as follows.

- Association and Dependencies:
    - **The ControlDesk and ControlDeskView classes are linked because the ControlDeskView class makes use of a ControlDesk class object.**
    - In a similar way, AddPartyView and NewPatronView are linked.
    - **An association relationship exists between ControlDeskView and AddPartyView as well.**

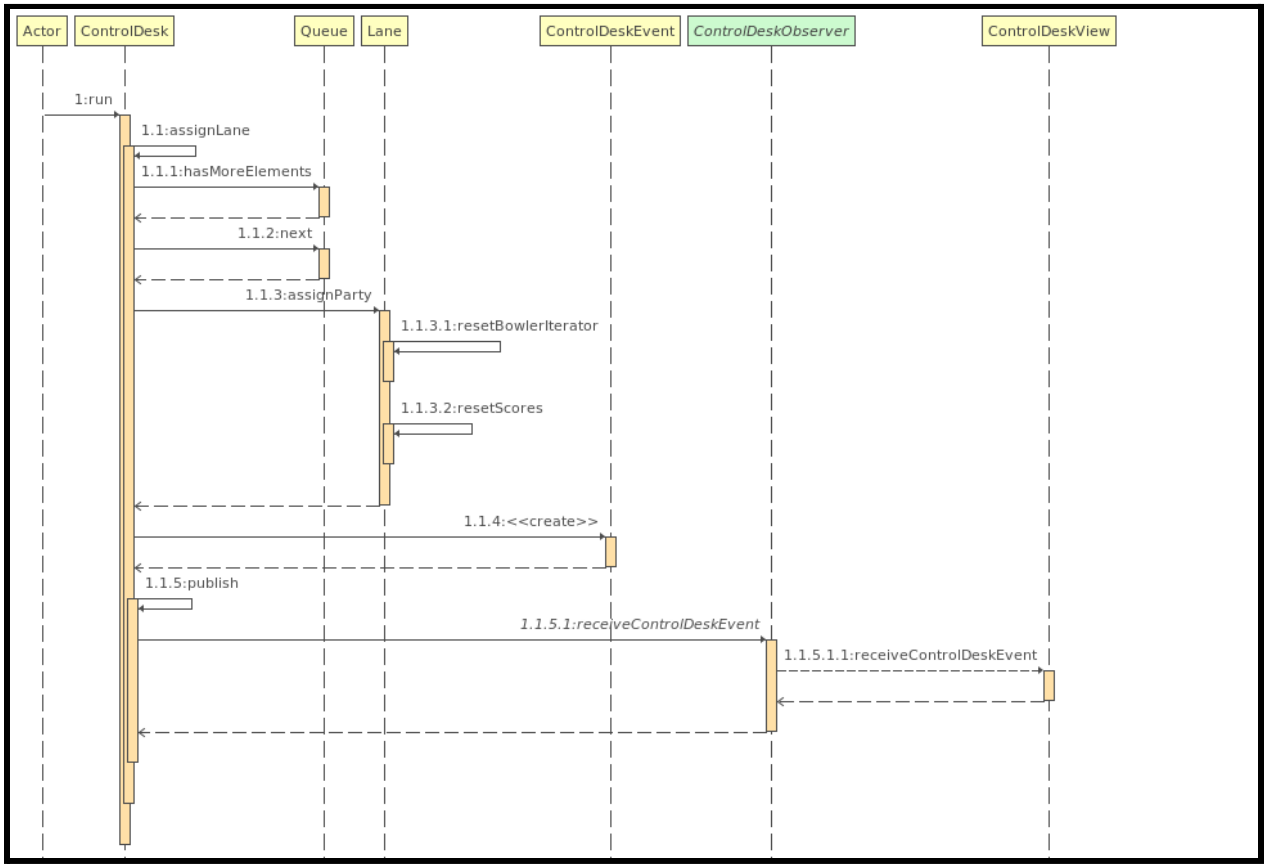

ActionListener
ListSelectionListener
ControlDeskEvent
- partyQueue: Vector
+ ControlDeskEvent(partyQueue: Vector)
+ getPartyQueue(): Vector
NewPatronView
- maxSize: int
- win: JFrame
- abort: JButton
- finished: JButton
- nickLabel: JLabel
- fullLabel: JLabel
- emailLabel: JLabel
- nickField: JTextField
- fullField: JTextField
- emailField: JTextField
- nick: String
- full: String
- email: String
- done: boolean
- selectedNick: String
- selectedMember: String
+ NewPatronView(v: AddPartyView)
+ actionPerformed(e: ActionEvent): void
+ done(): boolean
+ getNick(): String
+ getFull(): String
+ getEmail(): String
EndGameReport
- win: JFrame
- printButton: JButton
- finished: JButton
- memberList: JList
- myVector: Vector
- retVal: Vector
- result: int
- selectedMember: String
change: To
comment: generated
the: edit
variable: template
enable: To
disable: and
creation: the
type: of
go: comments
Generation: Code
+ EndGameReport(partyName: String, party: Party)
+ actionPerformed(e: ActionEvent): void
+ valueChanged(e: ListSelectionEvent): void
+ getResult(): Vector
+ destroy(): void
+ main(args: String[]): void
EndGamePrompt
- win: JFrame
- yesButton: JButton
- noButton: JButton
- result: int
- selectedNick: String
- selectedMember: String
+ EndGamePrompt(partyName: String)
+ actionPerformed(e: ActionEvent): void
+ getResult(): int
+ distroy(): void
addParty
1
AddPartyView
- maxSize: int
- win: JFrame
- addPatron: JButton
- newPatron: JButton
- remPatron: JButton
- finished: JButton
- partyList: JList
- allBowlers: JList
- party: Vector
- bowlerdb: Vector
- lock: Integer
- selectedNick: String
- selectedMember: String
change: To
comment: generated
the: edit
variable: template
enable: To
disable: and
creation: the
type: of
go: comments
Generation: Code
+ AddPartyView(controlDesk: ControlDeskView, max: int)
+ actionPerformed(e: ActionEvent): void
+ valueChanged(e: ListSelectionEvent): void
+ getNames(): Vector
+ updateNewPatron(newPatron: NewPatronView): void
+ getParty(): Vector
ControlDeskObserver
receiveControlDeskEvent(ce: ControlDeskEvent): void
controlDesk
1
ControlDeskView
- addParty: JButton
- finished: JButton
- assign: JButton
- win: JFrame
- partyList: JList
- maxMembers: int
$: v
accessors: Added
and: and
+ ControlDeskView(controlDesk: ControlDesk, maxMembers: int)
+ actionPerformed(e: ActionEvent): void
+ updateAddParty(addPartyView: AddPartyView): void
+ receiveControlDeskEvent(ce: ControlDeskEvent): void
# equals(): void
Thread
PinsetterObserver
controlDesk
1
Lane
- setter: Pinsetter
- scores: HashMap
- subscribers: Vector
- gameIsHalted: boolean
- partyAssigned: boolean
- gameFinished: boolean
- bowlerIterator: Iterator
- ball: int
- bowlIndex: int
- frameNumber: int
- tenthFrameStrike: boolean
- curScores: int[]
- cumulScores: int[][]
- canThrowAgain: boolean
- finalScores: int[][]
- gameNumber: int
- currentThrower: Bowler
$: v
sleep: fixed
bug: thread
lane: in
log: empty
log: empty
Party: Adding
Lane: Adding
Bowler: Adding
and: Adding
+ Lane()
+ run(): void
+ receivePinsetterEvent(pe: PinsetterEvent): void
# resetBowlerIterator(): void
# resetScores(): void
+ assignParty(theParty: Party): void
# markScore(Cur: Bowler, frame: int, ball: int, score: int): void
# lanePublish(): LaneEvent
# getScore(Cur: Bowler, frame: int): int
+ isPartyAssigned(): boolean
+ isGameFinished(): boolean
+ subscribe(adding: LaneObserver): void
+ unsubscribe(removing: LaneObserver): void
+ publish(event: LaneEvent): void
+ getPinsetter(): Pinsetter
+ pauseGame(): void
+ unPauseGame(): void
party
1
ControlDesk
- lanes: HashSet
- partyQueue: Queue
- numLanes: int
- subscribers: Vector
+ ControlDesk(numLanes: int)
+ run(): void
# registerPatron(nickName: String): Bowler
+ assignLane(): void
+ viewScores(ln: Lane): void
+ addPartyQueue(partyNicks: Vector): void
+ getPartyQueue(): Vector
+ getNumLanes(): int
+ subscribe(adding: ControlDeskObserver): void
+ publish(event: ControlDeskEvent): void
+ getLanes(): HashSet
Party
- myBowlers: Vector
+ Party(bowlers: Vector)
+ getMembers(): Vector

3. Sequence Diagram (Before Refactoring):

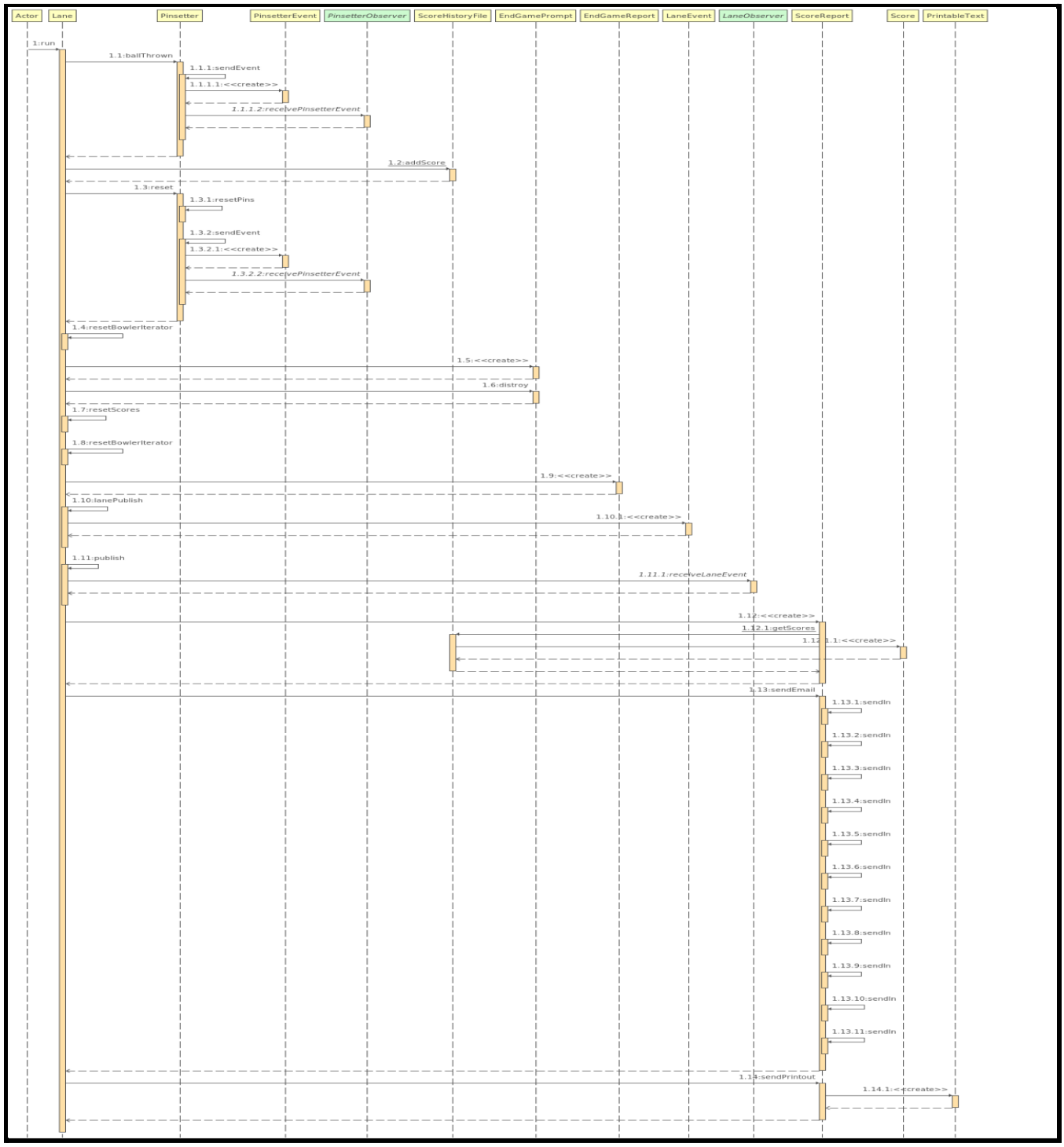
- **drive.main()**



- **ControlDesk.run()**

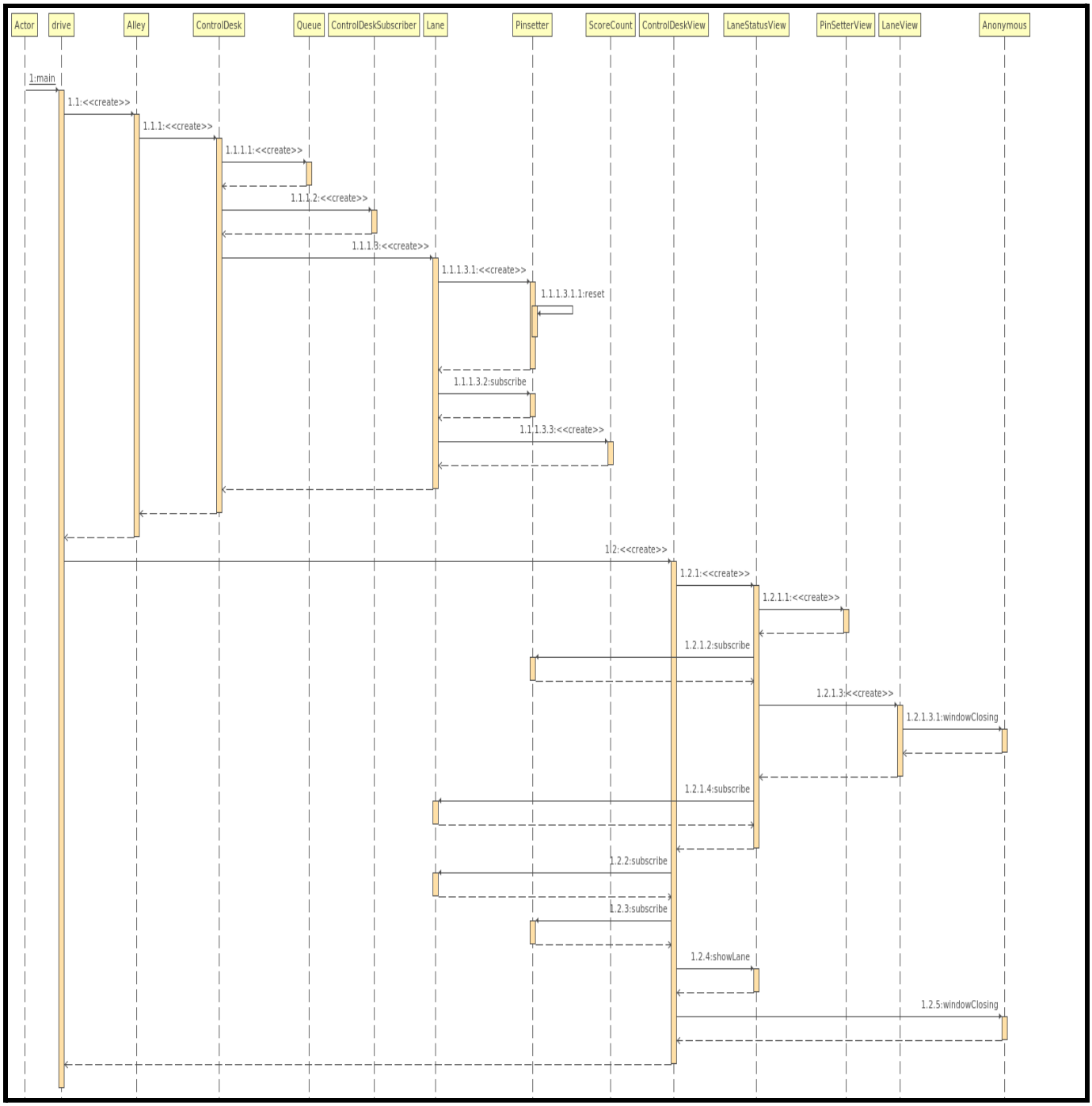


- **Lane.run()**

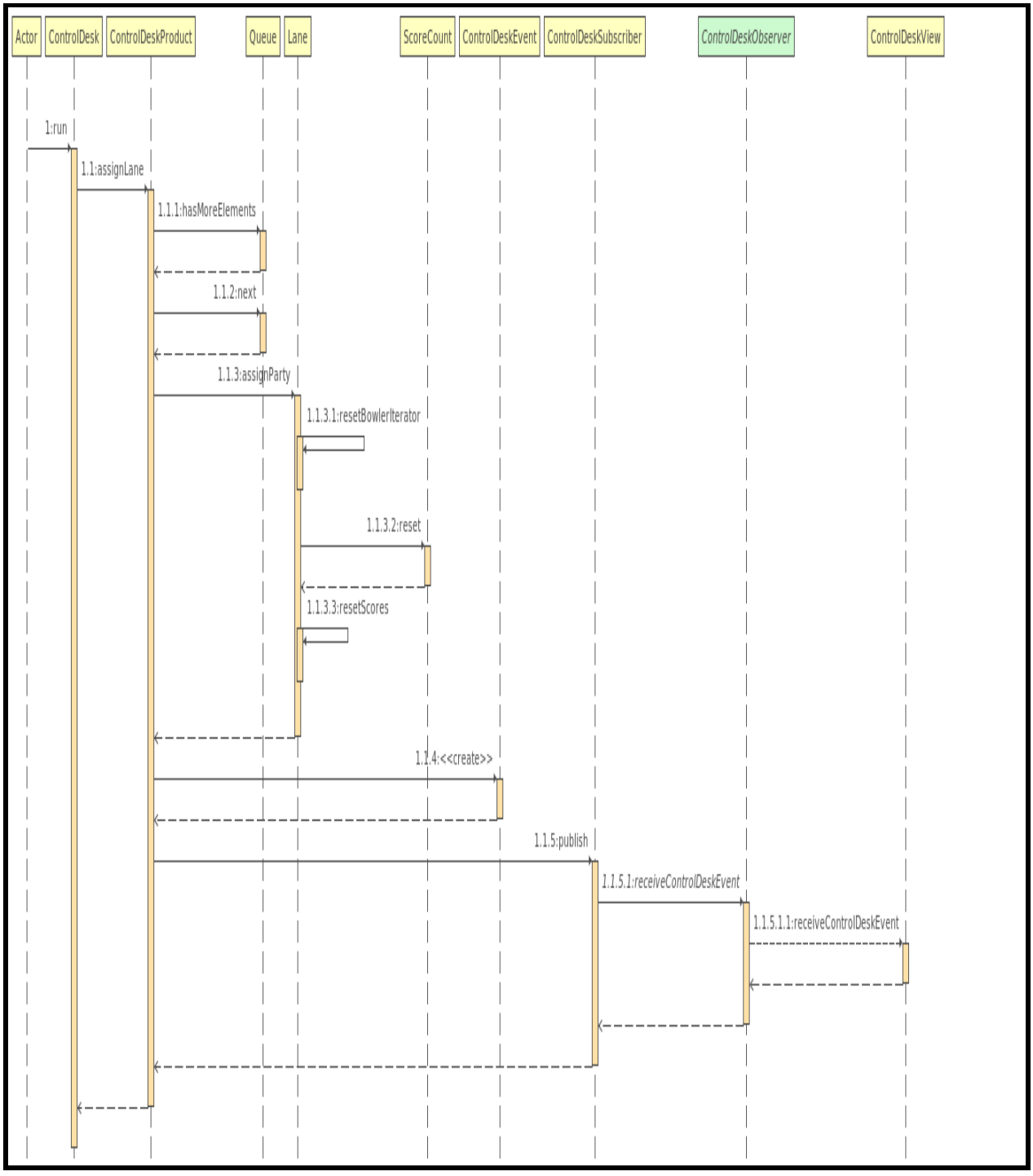

Actor
Lane
Pinsetter
PinsetterEvent
PinsetterObserver
ScoreHistoryFile
EndGamePrompt
EndGameReport
LaneEvent
LaneObserver
ScoreReport
Score
PrintableText
1:run
1.1:ballThrown
1.1.1:sendEvent
1.1.1.1:<<create>>
1.1.1.2:receivePinsetterEvent
1.2:addScore
1.3:reset
1.3.1:resetPins
1.3.2:sendEvent
1.3.2.1:<<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:lanePublish
1.10.1:<<create>>
1.11:publish
1.11.1:receiveLaneEvent
1.12:<<create>>
1.12.1:getScores
1.12.1.1:<<create>>
1.13:sendEmail
1.13.1:sendln
1.13.2:sendln
1.13.3:sendln
1.13.4:sendln
1.13.5:sendln
1.13.6:sendln
1.13.7:sendln
1.13.8:sendln
1.13.9:sendln
1.13.10:sendln
1.13.11:sendln
1.14:sendPrintout
1.14.1:<<create>>

4. Sequence Diagram (After Refactoring):

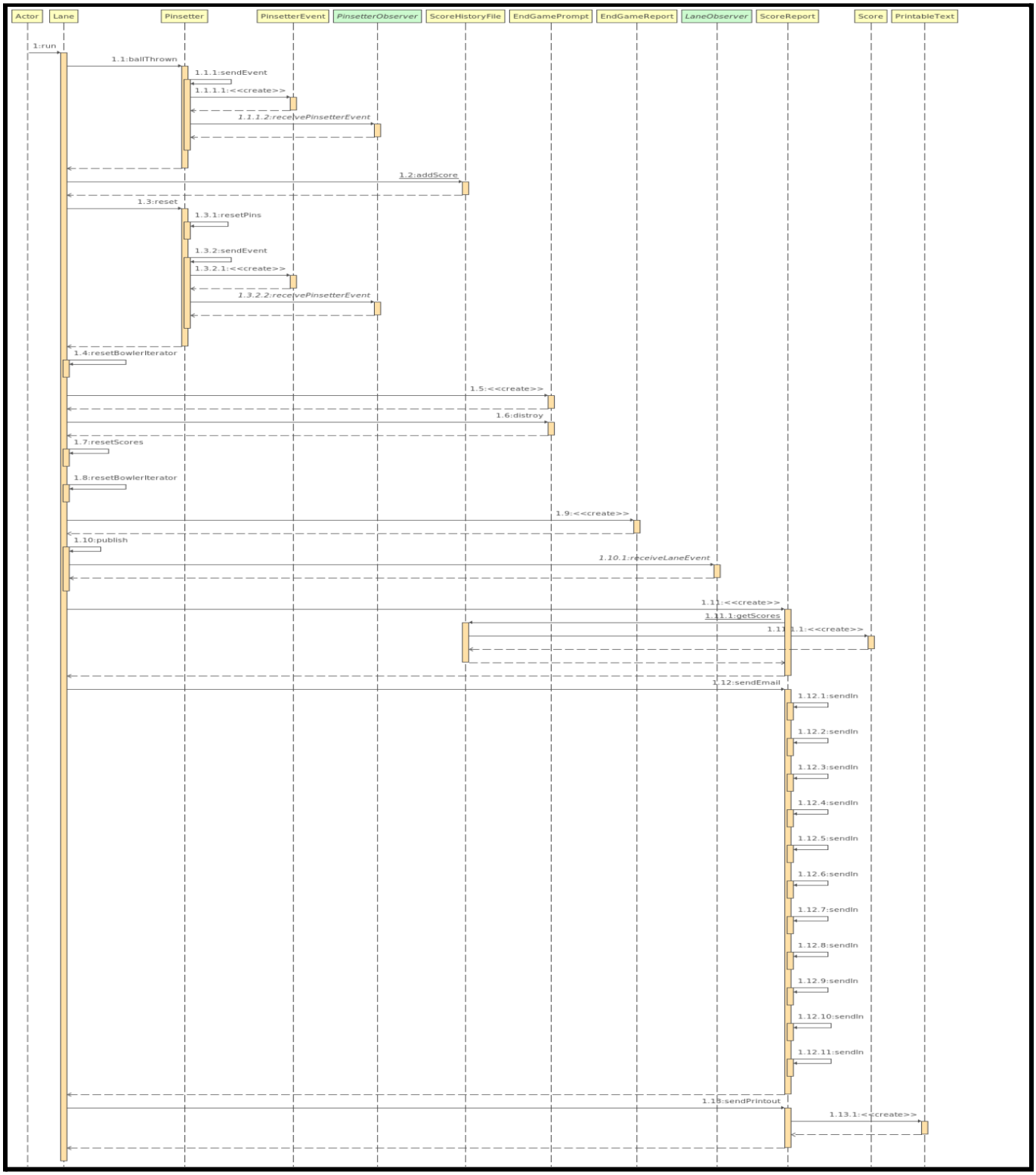
- **drive.main()**



- **ControlDesk.run()**



- **Lane.run()**


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:sendln
1.12.2:sendln
1.12.3:sendln
1.12.4:sendln
1.12.5:sendln
1.12.6:sendln
1.12.7:sendln
1.12.8:sendln
1.12.9:sendln
1.12.10:sendln
1.12.11:sendln
1.13:sendPrintout
1.13.1:<<create>>

IV. Summary of Responsibility of Each Class:

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.	Drive	• static void main()	• Driver class for the program and consists of the main function	• ControlDesk • Alley • ControlDeskView
10.	EndGamePrompt	• EndGamePrompt(String partyName) • void actionPerformed(ActionEvent e) • int getResult()	• Showcases yes and no option in the UI • Clears up the main panel • Returns result	-

		• void destroy()	or error	
11.	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	-
12.	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
13.	LaneEvent	• LaneEvent() • Multiple boolean Getter and setters	• Sets and gets the functionalities used in lane.java	• Party • Bowler
14.	LaneEventInterface	• An interface class	• Multiple interface classes of lane event	• Party • Bowler

15.	LaneObserver	• An interface class	• Interface for receiving lane event	
16.	LaneServer	• An interface class	• Interface class	
17.	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
18.	LaneView	• void show() • void high() • Jframe makeFrame • void receiveLaneEvent(LaneEvent le)	• Render GUI for alley lanes	• Lane
19.	NewPatronView	• void actionPerformed() • void valueChanged() • Vector getParty() • Vector getNames() • void updateNewPatron()	• Display new Patron information • Getter and setter functions for patron information	• AddPartyView
20.	Party	• Vector getMembers()	• Sets new vector of bowlers	
21.	Pinsetter	• void ballThrown() • void reset() void • resetPins() void • subscribe(Pinsetter Observer subscriber)	• Send pin setter events to all subscribers • Simulates a ball thrown coming in contact with the pinsetter • Resetting the pins	• PinsetterObserver
22.	PinsetterEvent	• boolean pinsKnockedDown(	• Creates a pinsetter event	-

		) • int pinsDownOnThisThrow() • int totalPinsDown() • boolean isFoulCommited() • int gerThrowNumber	• Returns a number of parameters related to pins	
23.	PinsetterObserver	• An interface class	• Interface class	-
24.	PinSetterView	• void receivePinsetterEvent()	• Implements the GUI related to pins • Receives a pinsetter event that it makes changes in the GUI	-
25.	PrintableText	• int print(Graphics g, PageFormat pageFormat, int pageIndex)	• Implements and renders graphical text in the GUI	
26.	Queue	• void add(Object o) • boolean hasMoreElements() • Vector asVector() • Object next()	• Creates a new queue	-
27.	Score	• constructor, getter and setter functions	• Returns score along with the particular bowler information	-
28.	ScoreHistoryFile	• Vector getScores(string nick) • void addScore()	• Stores and purchase data from a particular file	-
29.	ScoreReport	• void sendEmail() • void sendPrintout() • void sendln()	• Generates and prints out the score report for the game	• Bowler

V. Analysis of Original Design:

- **Design positives:**
  - **Low Coupling:** The modules have a low coupling metric. Out of 29 classes, 25 have low coupling and 4 have low-medium coupling.
  - **Comments:** The modules are commented properly and provide an overview of basic functionality of the module. The comments enable a new reader to understand the code easily.

- **Design problems:**

  - **Long methods:** Some modules like lane.java, have methods that are too long as compared to an ideal length of the method which in turn increases the complexity. Hence, this issue needs to be reviewed and modularization and optimization needs to be done.
  - **Dead Code:** Some lines of codes are not being used anywhere in the application. Also, there is some code present to be worked upon in the future. Hence, they should be removed.
  - **Multiple Main Methods:** The code has 2 main methods in drive.java and EndGameReport.java modules.
  - **Large number of parameters:** A few methods like LaneEvent.java expects a huge number of parameters, here 9. A good code practice restricts maximum permissible number of parameters to be 3. Methods with more than 3 parameters are not considered optimized and require appropriate justification.
  - **Duplicate code:** There are few instances where methods have duplicate code. This should be removed as it unnecessarily adds to the overall complexity and LOC.
  - **High complexity:** Some modules have high conditional and cyclomatic complexities which can be improved.
  - Methods performing tasks for some other class are present elsewhere. This repositioning should be done.
  - **Use of old tools/ libraries:** AWT components have been used instead of Swing.

- **Design pattern implemented:**

  1. **Adapter Pattern:**
     Adapter Pattern converts the interface of one class to another interface. It bridges the gap between two incompatible interfaces. Here, the class ControlDesk is an adapter that joins the subsystems of bowlers, party and queue. Hence, the implementation of all these 3 modules can be seen while adding a party and

players into it. When the number of parties exceeds, the newly added party is sent to the queue. Therefore, bowlers, party and queue are the adaptee.

2. **Observer Pattern:**
   Observer Pattern defines one to many relationship between objects such that when the state of one object changes, it is notified and updated to all other related objects/ subscribers. Here the implementation of button clicks in various components implement the observer pattern. When a button is clicked, corresponding tasks are carried out by the event handlers.

- **Functionalities working:**
  - Add party
  - Add new player
  - View Lane
  - View pinsetting
  - Maintain a queue for parties
  - Maintain database of scores for all players in all parties

- **Functionalities not working:**
  The **Print Report** functionality is not working.

VI. Code Smells:

There are various code smells existing in the code namely:

1. God/long Class
2. Redundant code
3. Dead Code
4. Multiple main methods present
5. Long methods
6. Long Parameter list
7. Feature envy

VII. Analysis of Refactored Design:

- The aim of refactoring was targeted towards classes which had problematic values of the considered metrics. More work was done to improve these metric values. Mainly, the refactoring included:
  1. Removing redundant and dead code blocks. Also, comment out 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.

- The new classes created were:
    1. **scoreCount.java:** To maintain the scores of players. This class contains one method that deals with all scoring criteria and calculations.
    2. **controlDeskSubscriber:** This class is made to handle all subscriber related functionality for class controlDesk. Observer pattern is implemented for the publish-subscribe functionality in the original code and the same is followed here.
    3. **endGameReportProduct:** This module contains various methods related to results and value used in end game reports. This is done for modularizing and improving coupling and cohesion metrics.
    4. **controlDeskProduct:** This module contains methods related to Queue and lane assignment. The methods placed here are picked up from controlDesk.java file so as to improve C3 metrics.
    5. **LaneProduct:** This module extracted some score related functions to improve modularity of Lane Class.
    6. **LaneViewProduct:** This module implements JFrame and related methods to show and hide LaneView module. This improves modularity and other metrics for the LaneView module.
    7. **PinsetterViewProduct:** This module implements JFrame and related methods to show and hide the PinsetterView module. This improves modularity and other metrics for the PinsetterView module.

- We eliminated **LaneEvent** and **LaneEventInterface** modules that were redundant.
- Overall, we were able to improve the C3 metric for the initial code. Apart from that we were able to reduce the cyclomatic Complexity for certain modules. All other metrics improved are mentioned in the Metrics Analysis section of this document.
- **Low Coupling:** We tried to make cohesion low by passing the parameters locally and removing the redundant ones. Also, we have extended long classes to break them down into different subclasses.
- **High Cohesion:** Long classes like Controldesk had numerous methods which contained methods related to other classes. Such methods are moved to the appropriate class file. Also, some methods were too broad so they are split into multiple subclasses to divide the task and achieve cohesion.
- **Separation of Concerns:** It refers to separating a system into distinct sections such that each section addresses a separate concern. For ex. Lane Class had a method to calculate the score but we have created a separate class ScoreCount for calculating the score and the updated score is sent to Lane Class to mark.
- **Dead Code Elimination:** We removed the multiple main methods in the system that weren't being called/used anywhere else. We also eliminated extra classes, redundant code and other blocks that were not being used anywhere.

VIII. Metric Analysis:

- **Metric Analysis of existing code:**

1. Cyclomatic complexity and other metrics:

Metric	Total	Mean	Std. Dev.	Maxi...	Resource causing Maximum	Method
McCabe Cyclomatic Complexity (avg/max per method)		2.12	3.773	38	/BowlingAlley/code/Lane.java	getScore
Number of Parameters (avg/max per method)		0.681	1.064	9	/BowlingAlley/code/LaneEvent.java	LaneEvent
Nested Block Depth (avg/max per method)		1.434	1.1	7	/BowlingAlley/code/Lane.java	run
Afferent Coupling (avg/max per packageFragment)		0	0	0	/BowlingAlley/code	
Efferent Coupling (avg/max per packageFragment)		0	0	0	/BowlingAlley/code	
Instability (avg/max per packageFragment)		1	0	1	/BowlingAlley/code	
Abstractness (avg/max per packageFragment)		0.147	0	0.147	/BowlingAlley/code	
Normalized Distance (avg/max per packageFragment)		0.147	0	0.147	/BowlingAlley/code	
Depth of Inheritance Tree (avg/max per type)		0.912	0.445	2	/BowlingAlley/code/Lane.java	
Weighted methods per Class (avg/max per type)	352	10.353	14.761	86	/BowlingAlley/code/Lane.java	
Number of Children (avg/max per type)	6	0.176	0.617	3	/BowlingAlley/code/PinsetterObserver.java	
Number of Overridden Methods (avg/max per type)	3	0.088	0.284	1	/BowlingAlley/code/Lane.java	
Lack of Cohesion of Methods (avg/max per type)		0.347	0.363	0.91	/BowlingAlley/code/LaneEvent.java	
Number of Attributes (avg/max per type)	143	4.206	5.178	17	/BowlingAlley/code/Lane.java	
Number of Static Attributes (avg/max per type)	2	0.059	0.235	1	/BowlingAlley/code/ScoreHistoryFile.java	
Number of Methods (avg/max per type)	158	4.647	3.613	18	/BowlingAlley/code/Lane.java	
Number of Static Methods (avg/max per type)	8	0.235	0.644	3	/BowlingAlley/code/BowlerFile.java	
Specialization Index (avg/max per type)		0.014	0.048	0.2	/BowlingAlley/code/Score.java	
Number of Classes (avg/max per packageFragment)	34	34	0	34	/BowlingAlley/code	
Number of Interfaces (avg/max per packageFragment)	5	5	0	5	/BowlingAlley/code	
Number of Packages	1					
Total Lines of Code	1912					
Method Lines of Code (avg/max per method)	1312	7.904	16.11	88	/BowlingAlley/code/Lane.java	getScore

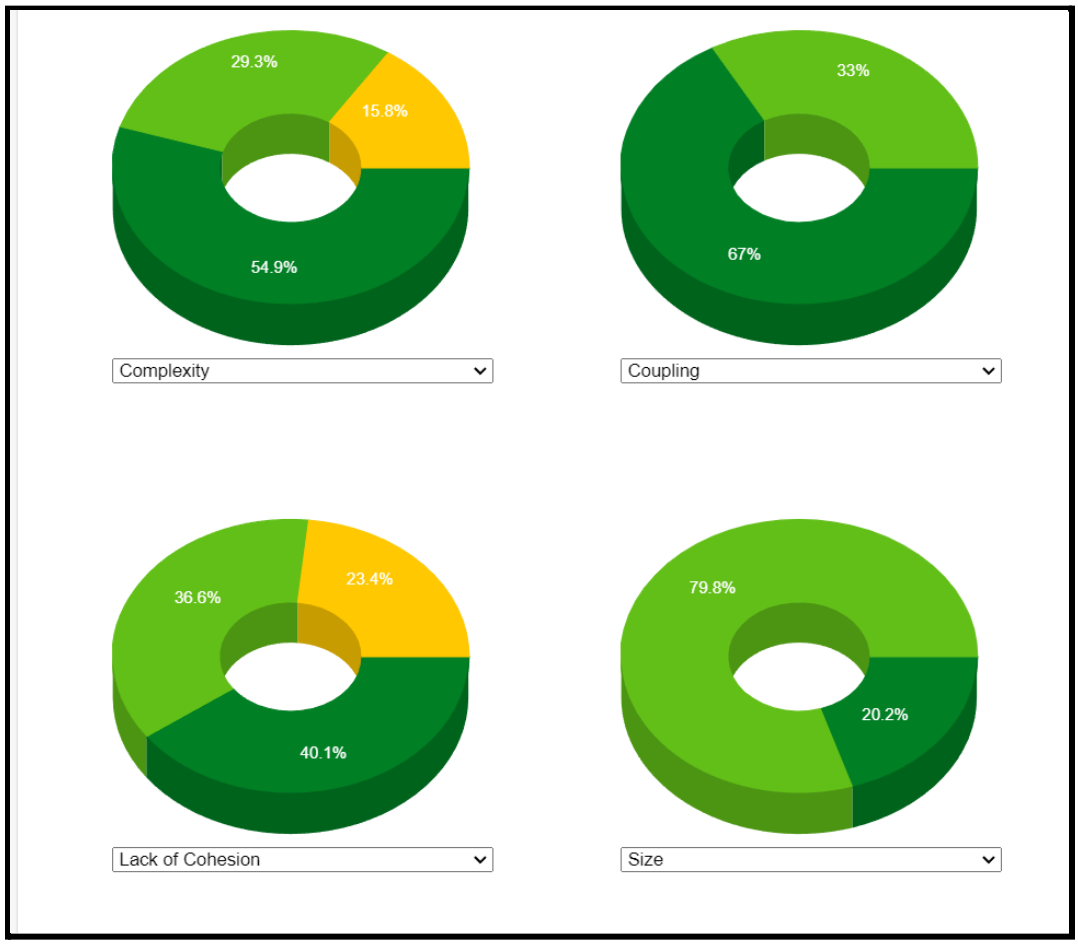
2. **Metric values** for separate modules:

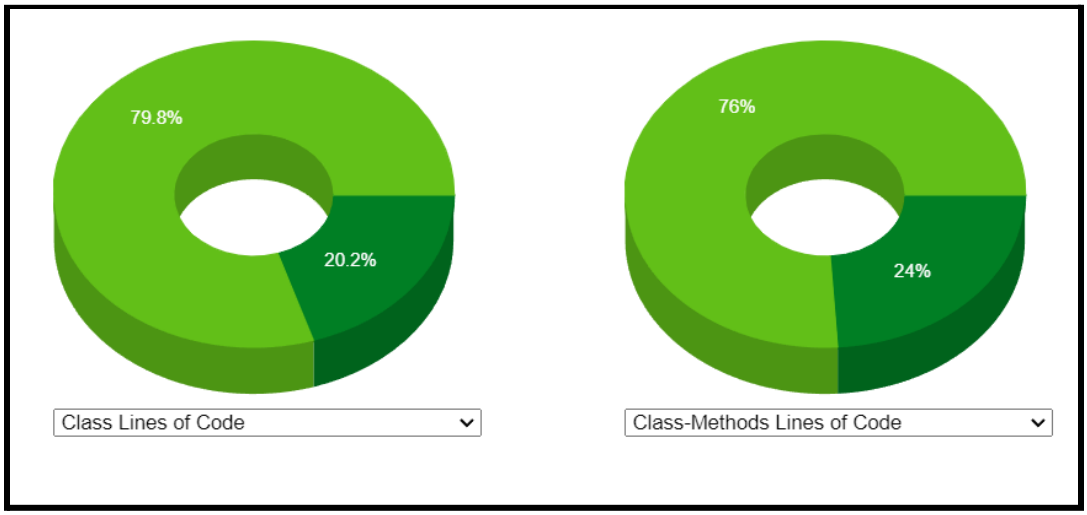
ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	Lane					227	medium-high	low-medium	medium-high	low-medium
2	ControlDeskView					87	low-medium	low-medium	low-medium	low-medium
3	ControlDesk					68	low-medium	low-medium	medium-high	low-medium
4	LaneStatusView					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

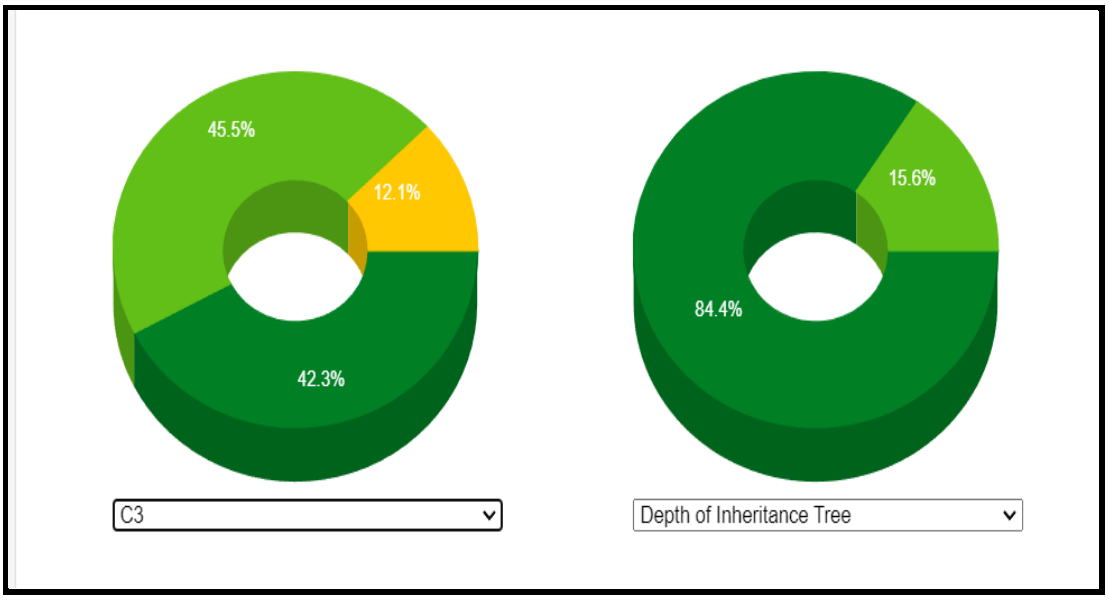
11	EndGamePrompt					55	low	low	low	low-medium
12	Pinsetter					47	low	low	low	low
13	LaneEvent					41	low	low	medium-high	low
14	BowlerFile					38	low	low	low	low
15	PinsetterEvent					26	low	low	low	low
16	Bowler					25	low	low	low	low
17	PrintableText					21	low	low	low	low
18	ScoreHistoryFile					20	low	low	low	low
19	Score					16	low	low	low	low
20	Queue					12	low	low	low	low

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	ControlDeskObserver					2	low	low	low	low
27	LaneObserver					2	low	low	low	low
28	LaneServer					2	low	low	low	low
29	PinsetterObserver					2	low	low	low	low

3. Metrics considered:







- **<u>Metric Analysis of refactored code:</u>**

1. **Cyclomatic complexity in lane class** reduced from 38 to 19 by refactoring changes. Weighted methods per Class also reduced from 86 to 48 in Lane class.

Metric	Total	Mean	Std. Dev.	Maxi...	Resource causing Maximum	Method
McCabe Cyclomatic Complexity (avg/max per method)		2.417	3.21	19	/Downloads_BowlingAlley/BowlingAlley/code/Lane.java	run
Number of Parameters (avg/max per method)		0.976	1.57	9	/Downloads_BowlingAlley/BowlingAlley/code/LaneStatusVi...	receiveLaneEvent
Nested Block Depth (avg/max per method)		1.646	1.105	7	/Downloads_BowlingAlley/BowlingAlley/code/Lane.java	run
Afferent Coupling (avg/max per packageFragment)		0	0	0	/Downloads_BowlingAlley/BowlingAlley/code	
Efferent Coupling (avg/max per packageFragment)		0	0	0	/Downloads_BowlingAlley/BowlingAlley/code	
Instability (avg/max per packageFragment)		1	0	1	/Downloads_BowlingAlley/BowlingAlley/code	
Abstractness (avg/max per packageFragment)		0.138	0	0.138	/Downloads_BowlingAlley/BowlingAlley/code	
Normalized Distance (avg/max per packageFragment)		0.138	0	0.138	/Downloads_BowlingAlley/BowlingAlley/code	
Depth of Inheritance Tree (avg/max per type)		0.931	0.45	2	/Downloads_BowlingAlley/BowlingAlley/code/Lane.java	
Weighted methods per Class (avg/max per type)	307	10.586	11.485	48	/Downloads_BowlingAlley/BowlingAlley/code/Lane.java	
Number of Children (avg/max per type)	6	0.207	0.663	3	/Downloads_BowlingAlley/BowlingAlley/code/PinsetterObs...	
Number of Overridden Methods (avg/max per type)	3	0.103	0.305	1	/Downloads_BowlingAlley/BowlingAlley/code/Lane.java	
Lack of Cohesion of Methods (avg/max per type)		0.35	0.357	0.894	/Downloads_BowlingAlley/BowlingAlley/code/NewPatronVi...	
Number of Attributes (avg/max per type)	133	4.586	5.642	20	/Downloads_BowlingAlley/BowlingAlley/code/Lane.java	
Number of Static Attributes (avg/max per type)	2	0.069	0.253	1	/Downloads_BowlingAlley/BowlingAlley/code/BowlerFile.java	
Number of Methods (avg/max per type)	119	4.103	3.133	15	/Downloads_BowlingAlley/BowlingAlley/code/Lane.java	
Number of Static Methods (avg/max per type)	8	0.276	0.826	4	/Downloads_BowlingAlley/BowlingAlley/code/BowlerFile.java	
Specialization Index (avg/max per type)		0.021	0.066	0.286	/Downloads_BowlingAlley/BowlingAlley/code/ControlDesk.j...	
Number of Classes (avg/max per packageFragment)	29	29	0	29	/Downloads_BowlingAlley/BowlingAlley/code	
Number of Interfaces (avg/max per packageFragment)	4	4	0	4	/Downloads_BowlingAlley/BowlingAlley/code	
Number of Packages	1					
Total Lines of Code	1746					
Method Lines of Code (avg/max per method)	1231	9.693	16.642	85	/Downloads_BowlingAlley/BowlingAlley/code/PinSetterVie...	PinSetterView

2. **Metric values** for separate modules:

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	Lane					162	low-medium	medium-high	low-medium	low-medium
2	ControlDeskView					87	low-medium	low-medium	low-medium	low-medium
3	LaneStatusView					93	low	low-medium	low-medium	low-medium
4	ControlDeskProduct					32	low	low-medium	low-medium	low
5	LaneView					140	low-medium	low	low-medium	low-medium
6	AddPartyView					127	low-medium	low	low-medium	low-medium
7	ScoreCount					60	low-medium	low	low	low-medium
8	ControlDesk					32	low-medium	low	low	low
9	PinSetterView					111	low	low	low	low-medium
10	NewPatronView					85	low	low	low	low-medium
11	ScoreReport					76	low	low	low	low-medium
12	EndGameReport					66	low	low	low-medium	low-medium
13	EndGamePrompt					55	low	low	low	low-medium
14	BowlerFile					47	low	low	low	low
15	Pinsetter					47	low	low	low	low
16	PinsetterEvent					26	low	low	low	low
17	Bowler					25	low	low	low	low
18	PrintableText					21	low	low	low	low
19	LaneProduct					21	low	low	low-medium	low
20	ScoreHistoryFile					20	low	low	low	low

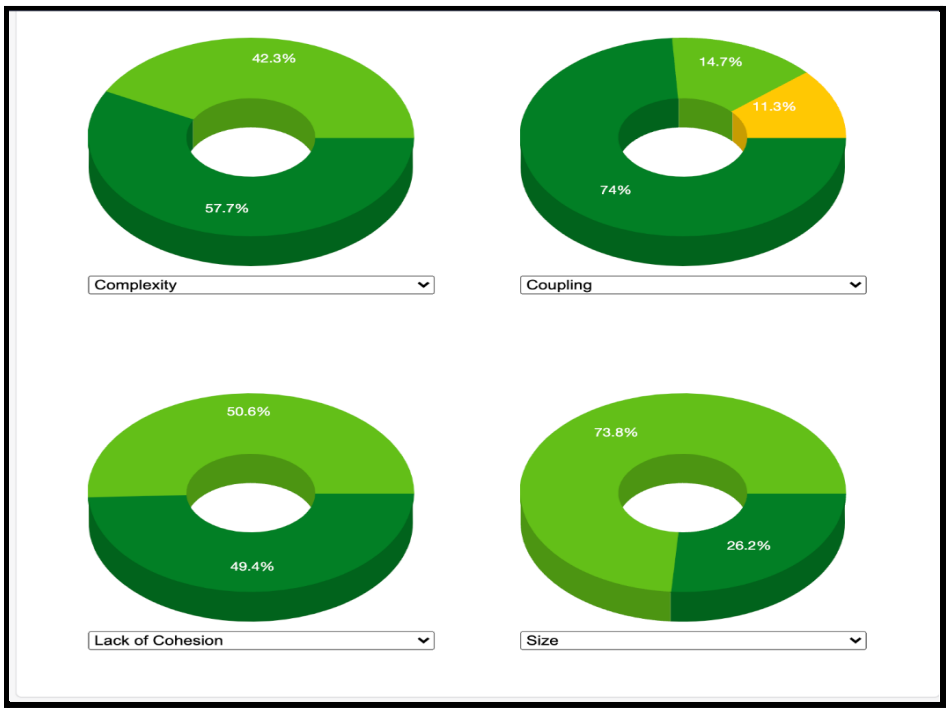
21	Score					16	low	low	low	low
22	EndGameReportProduct					16	low	low	low	low
23	Queue					12	low	low	low	low
24	PinSetterViewProduct					10	low	low	low	low
25	LaneViewProduct					10	low	low	low	low
26	ControlDeskSubscr...					9	low	low	low	low
27	drive					7	low	low	low	low
28	Alley					6	low	low	low	low
29	ControlDeskEvent					6	low	low	low	low
30	Party					6	low	low	low	low

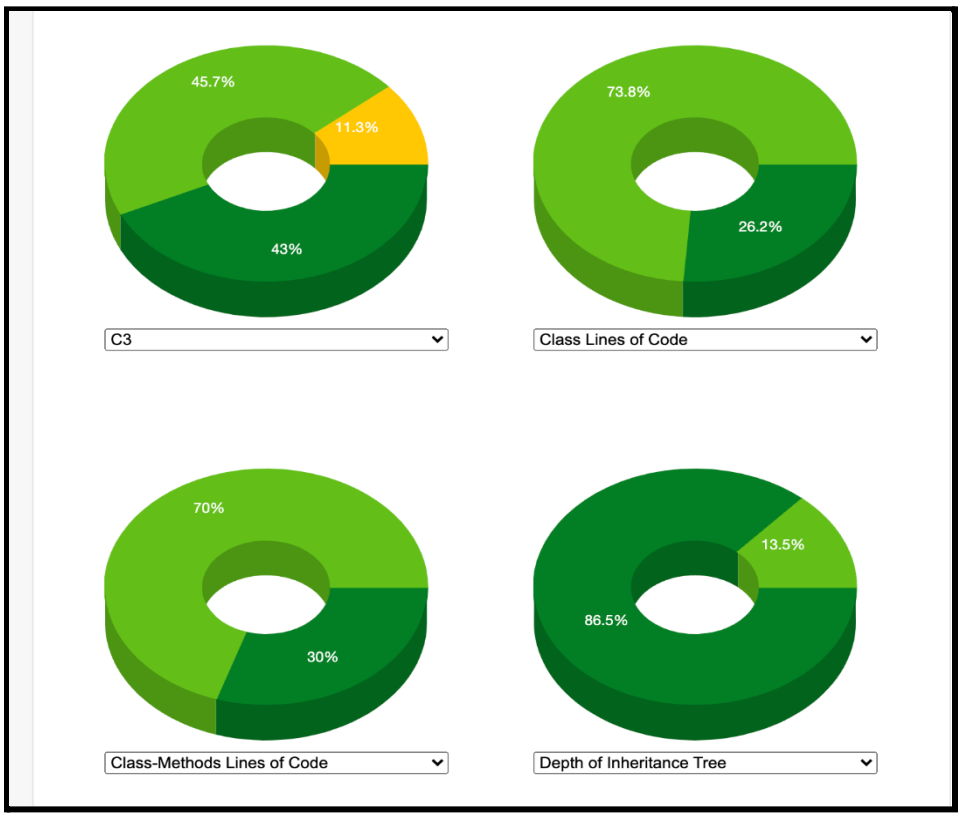
31	ControlDeskObserver					2	low	low	low	low
32	LaneObserver					2	low	low	low	low
33	LaneServer					2	low	low	low	low
34	PinsetterObserver					2	low	low	low	low

The major changes that came are in metric values of the modules:

- **Lane:** Complexity and lack of cohesion reduced to reduced to low-medium from medium-high
- **ControlDesk:** Coupling reduced to low from low-medium and lack of cohesion reduced from high to low
- As the **LaneEvent** methods are modularized and repositioned, this class that had medium-high lack of cohesion is eliminated.

3. Metrics considered:





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.

3. **How did your refactoring affect the metrics? Did your refactoring improve the metrics? In all areas? In some areas? What contributed to these results?**
   The refactoring we did improved the considered metrics significantly. We were able to reduce complexity and lack of cohesion of Lane Class. We modularized the lane related modules and dealt with redundant and dead codes. We moved the score calculation functionality to a separate class. Also, for the ControlDesk module, we again worked on redundant and dead code removal and also moved methods to appropriate classes, i.e., registerPatron() to BowlerFile. Also, we separated subscriber methods into a new class. Data encapsulation has been increased by making the attribute private to LaneEvent class. Apart from these major changes, we worked towards improving overall code and modularizing long classes like pinsetter and lane. Also, we dealt with multiple main methods present in the code by commenting out EndGameReport file's main method as the main method of drive module is being used.