SWEN 262

Dr. Abdul

Refactoring Project

Team 5: Glenn Todd, Ethan Ligi, JT Novitsky, Michael Ackerman

Refactoring Plan:

We primarily focused on the Lane.java class in terms of refactoring. This class held an extremely large amount of logic and handling that greatly increased the code complexity metric value. This is the metric we used to create our refactoring hotspots throughout the codebase. The run and getScore functions in Lane were used frequently and also contained the most complex logic. We found that for refactoring the getScore function, we could use a state pattern subsystem to reduce overall complexity of the code. Initially, getScore worked was a large if statement tree: checking if the roll was a spare, strike, tenth frame strike, or a normal throw. The state pattern worked well for this since the roll would be the event to trigger a change in state. Each state had its own methods to properly calculate the score(s) for the throw(s) according to the rules of bowling. Similarly, for the run class, we noticed that the logic stemmed from if statements regarding the status of the bowlers and the game. Again, using the state pattern helps to reduce complexity by using one function, whos behavior changes depending on the state of the game, to manage overall behavior of the Lane. Finally, the initial codebase implemented somewhat of an observer pattern, but it was not completely efficient and optimal. We updated this subsystem to reduce overall complexity of the observer and event functionality. Since it was already there, we simply refactored what was there instead of having to completely rework the subsystem.

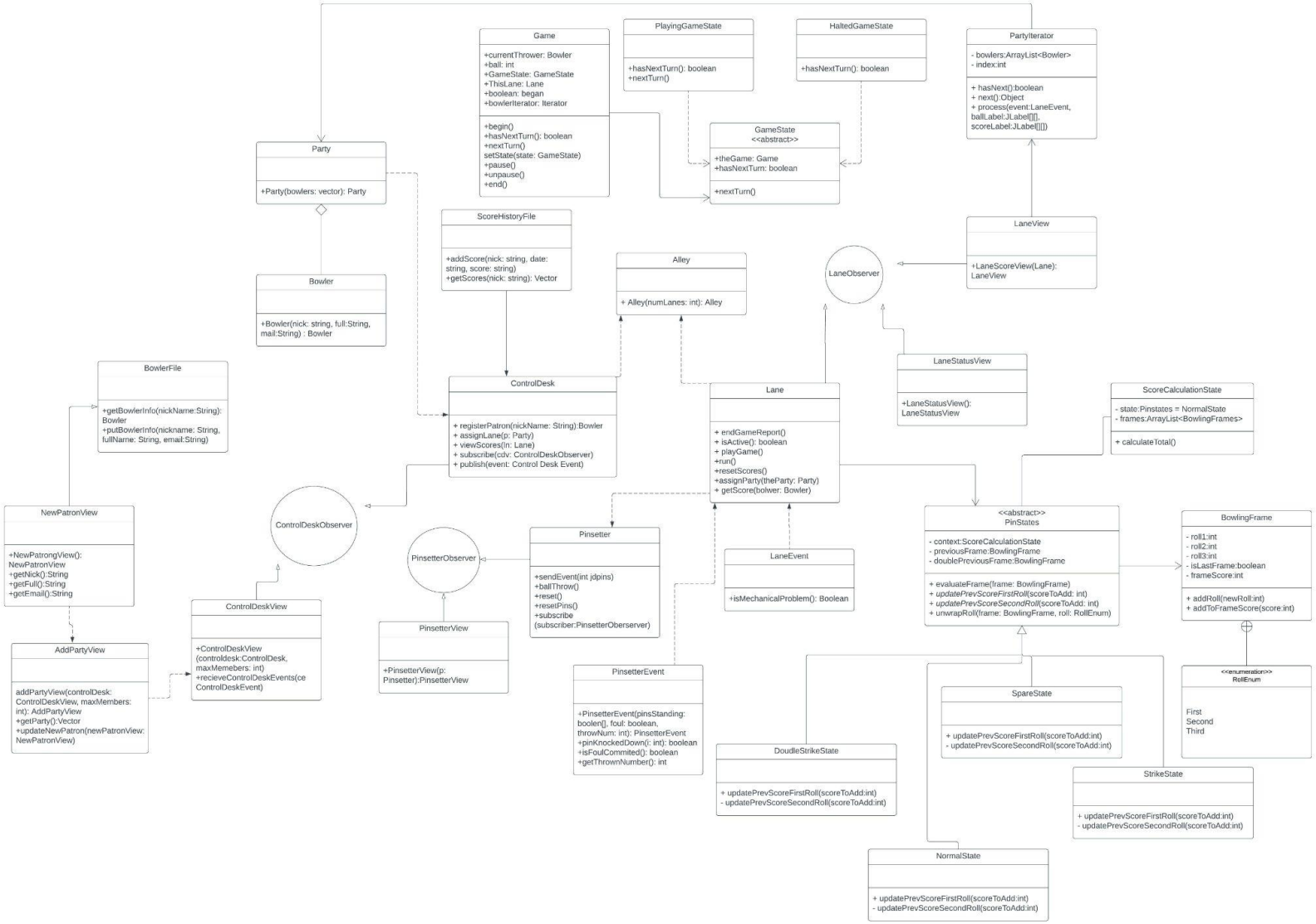
Initial Metrics Gathered:

method	CogC	ev(G)	iv(G)	v(G)
Lane.getScore(Bowler,int)	111	5	1	38
Lane.run()	55	1	14	19
LaneView.receiveLaneEvent(LaneEvent)	36	1	17	19
Lane.receivePinsetterEvent(PinsetterEvent)	25	1	9	12
LaneStatusView.actionPerformed(ActionEvent)	17	1	11	11

Our focus was on the Lane.getScore(), Lane.run() and LaneView.receiveLaneEvent() functions. We posed the idea of secondary goals, such as increased JavaDocs, a new Model View Controller package system and the introduction of JUnit testing. These would be areas of focus in coming refactoring stages. The alarming metric results above are what we decided to focus on first.

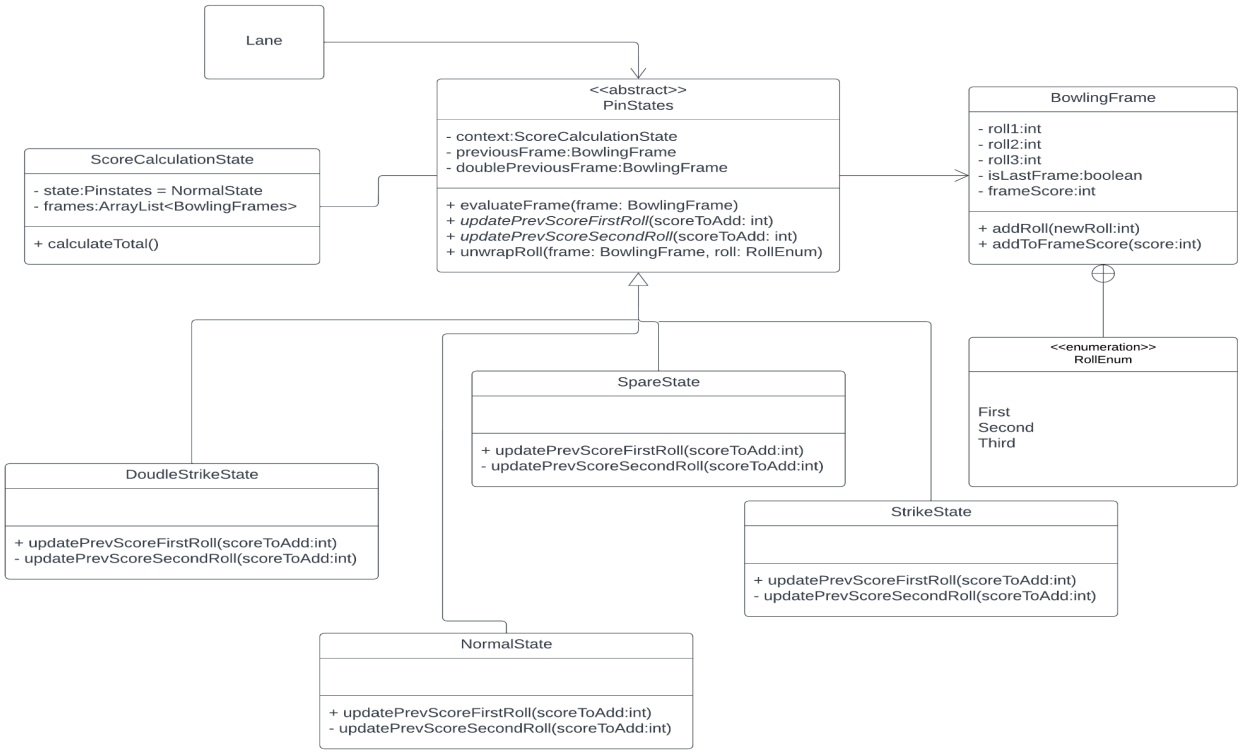
Class Diagram:

Here is our overall class diagram for the bowling game after adding our refactored design patterns.



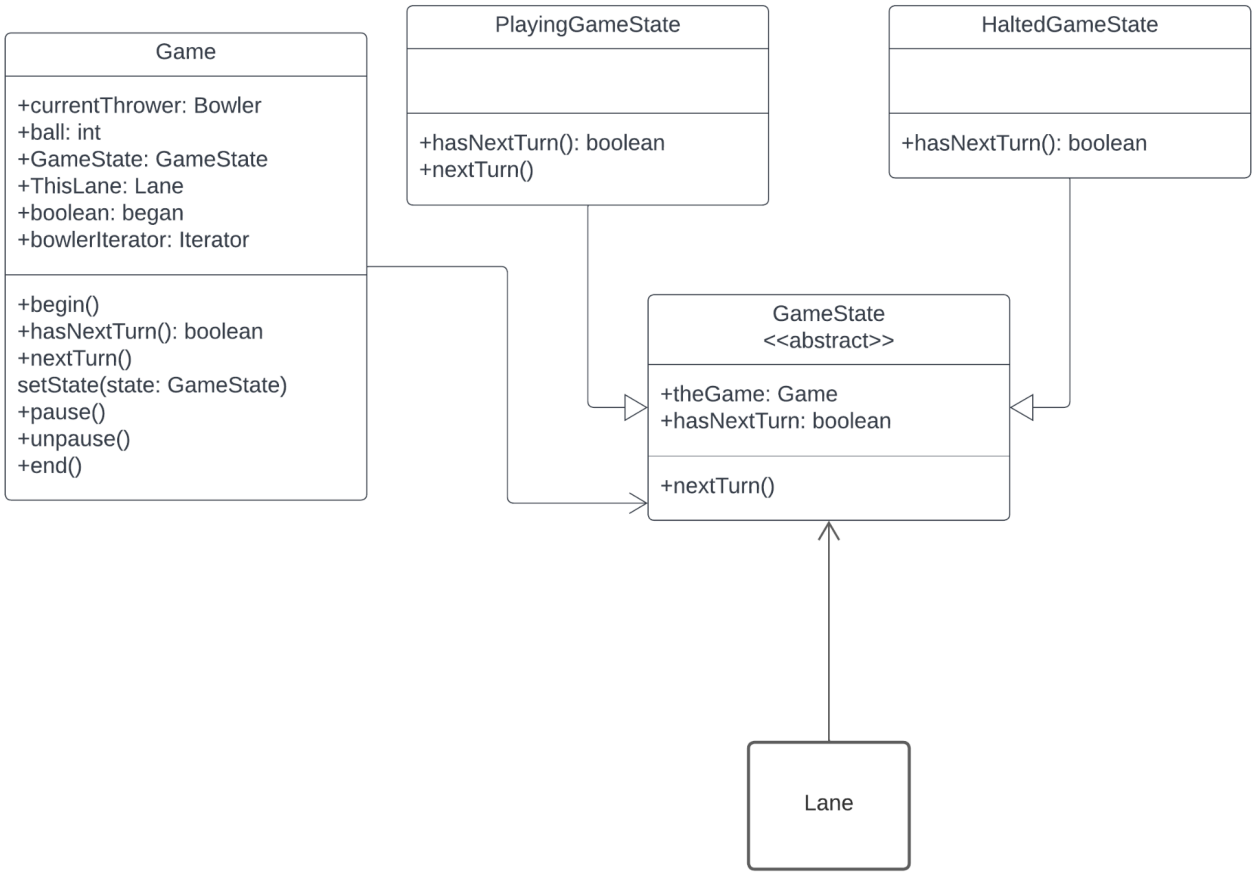
Refactoring Lane.getScore():

Lane.getScore() was by far the most complex of any of the methods in the entire code base with a CogC of 111. This was the main focus of the refactoring project. We understood a new subsystem would be the best fit to properly address the workload that the single function had. Stated previously, the state pattern helps to reconstruct the logic of calculating the score. To go into more detail, the same kind of data was needed every time the function was called, but depending on the state of the number of pins knocked down on the throw. Using the state diagram, the getScores function was reduced to simply call the calculateTotal method inside the correct state diagram. This drastically reduced the complexity of the getScores function without simply relocating it somewhere else in the code. This subsystem helped to constructive remove the complexity instead of just dividing and relocating it. Below is the class diagram that specifically highlights how we implemented the state pattern. It connects to the rest of the code through the Lane class.



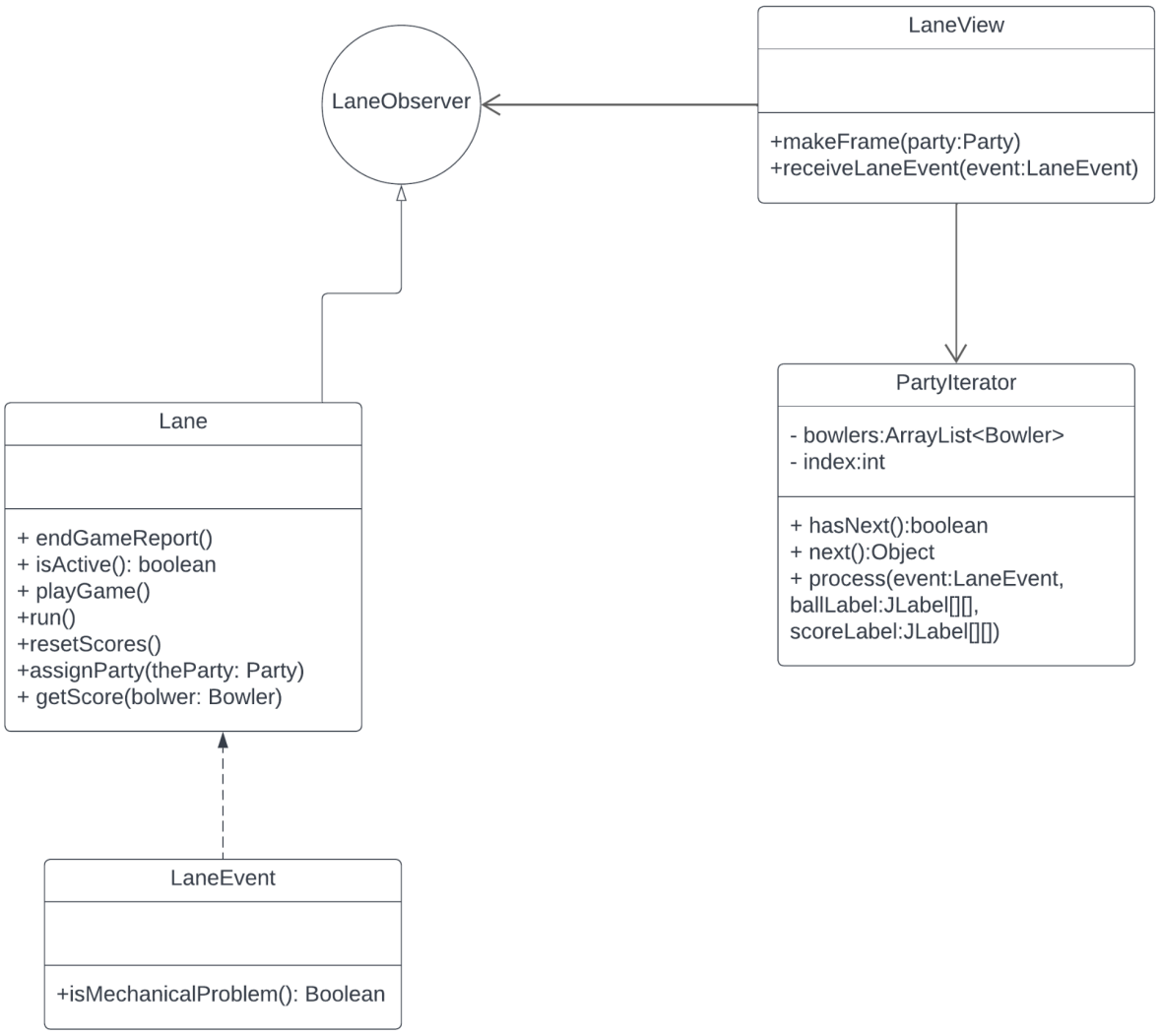
Refactoring Lane.run():

This was the second most complex method in the code base, coming in at 55 CogC. Again, we noticed that the original structure of the function followed an if statement tree. This pointed us to the conclusion that the state pattern would again be suited to reduce the complexity of the method. Creating the Game class helped to keep the code organized and adhere to the Object Oriented principle of low coupling. During our first attempt at implementing this pattern, we did not use the Game class and it meant that the concrete states were directly effecting the Lane states, which was increasing coupling and making the code very confusing. Through using the game class as the context of the pattern we eliminated our issues and reduced the value of our metric. The two concrete states are if the game is in a playable state compared to if it is not in a playable state. From there, the correct action would be handled by the concrete state depending on the current state of our created Game class.



Refactoring the Observer Pattern:

Our final section of refactoring was adapting the preexisting observer pattern in the codebase. The observer pattern implemented previously was not perfect and still left certain methods within with a high CogC of 36. By resituating how parties were structured and creating a new PartyIterator class, we were able to create a slightly less complex method for handling LaneEvents. You may notice from the final metrics that our PartyIterator.process() function is one of the new top five complex methods, however this is 9 CogC lower than the original LaneView.recieveLaneEvent() function. In addition, the new PartyIterator creates a better structure in addition to reducing complexity.

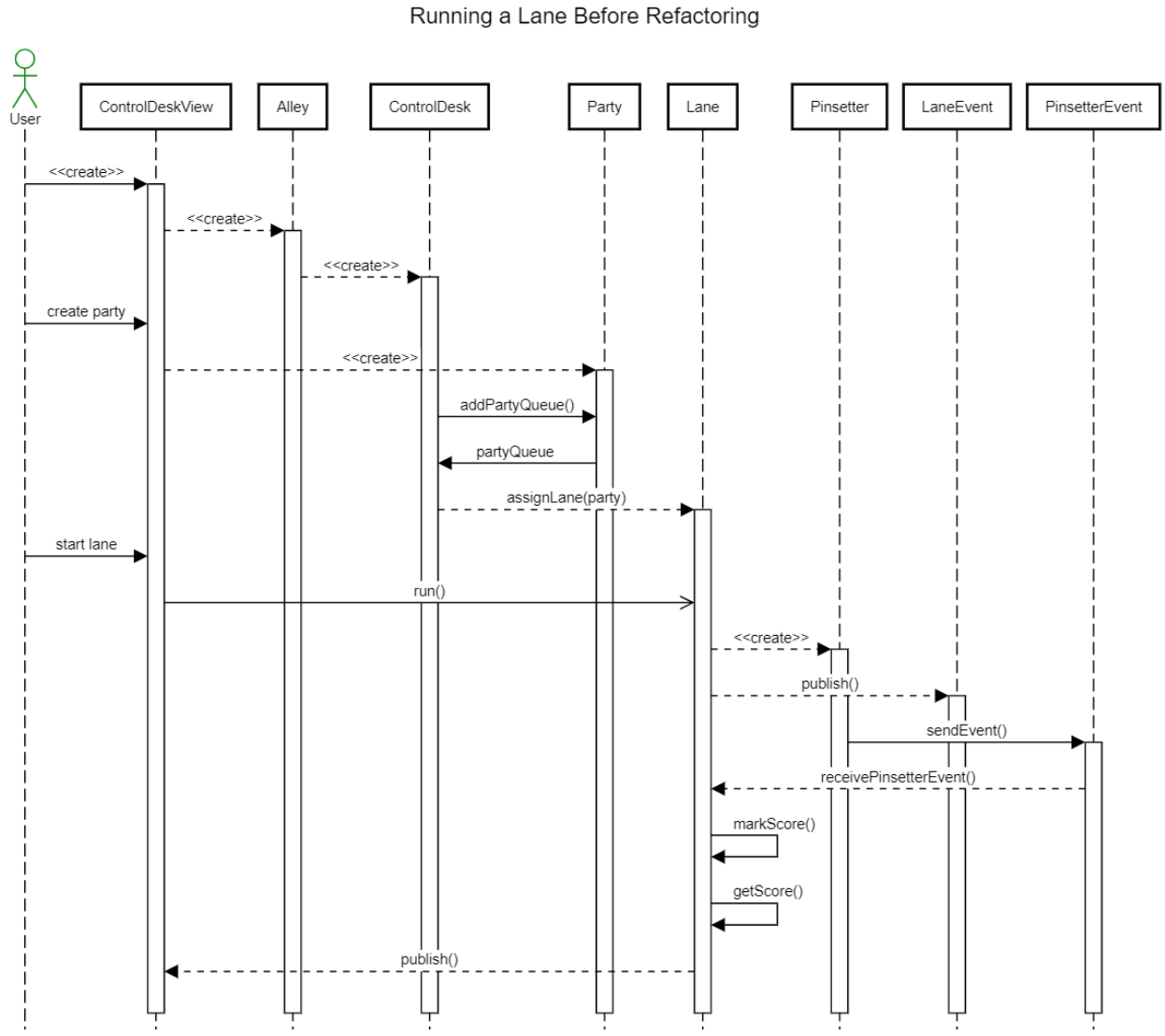


Sequence Diagrams:

The sequence diagrams in this section cover both of the major aspects we refactored, Lane.run() and Lane.getScore() from both before and after our refactoring.

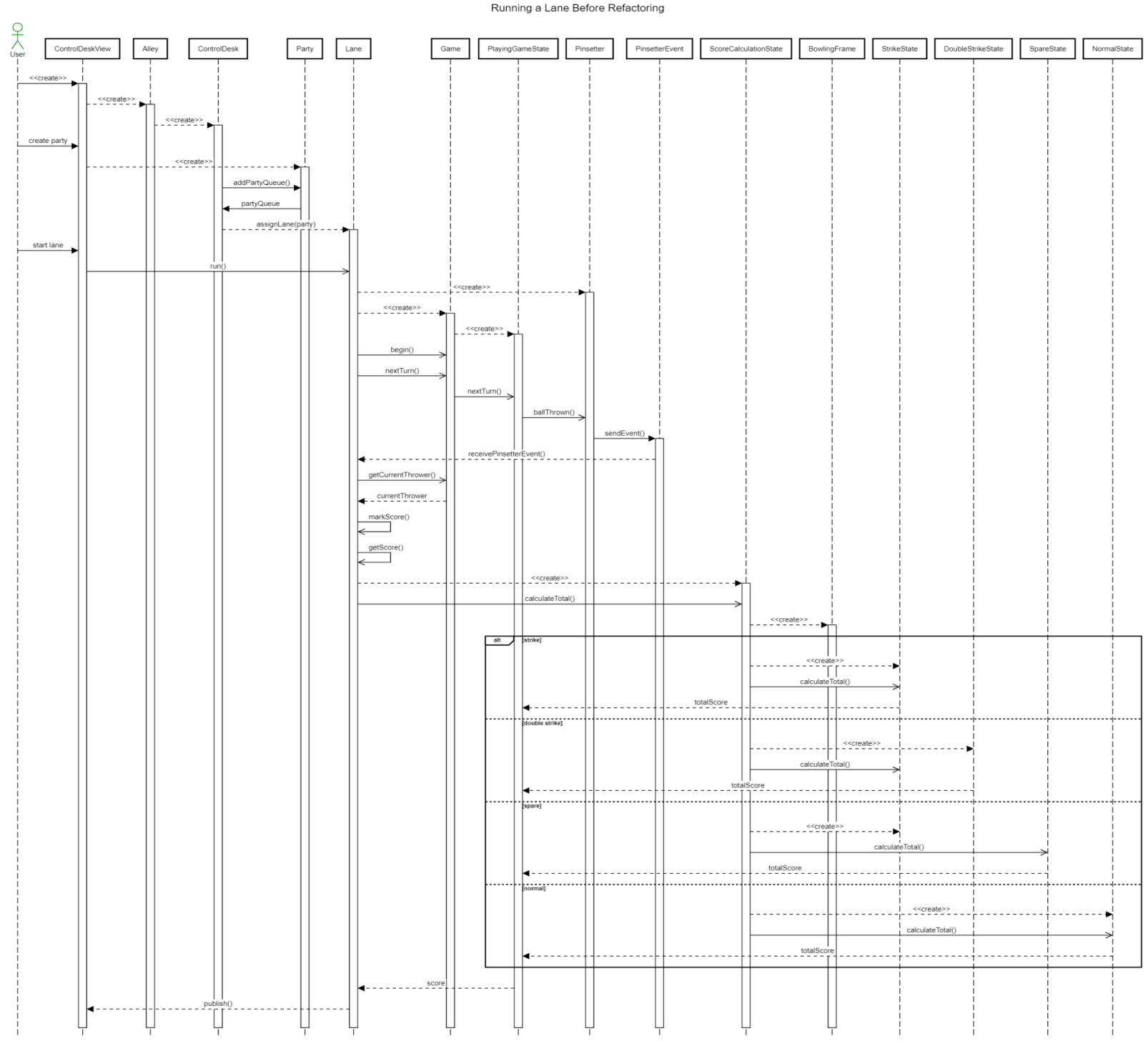
Before Refactoring:

As you can see towards the end of the sequence diagram, before refactoring the getScore was entirely internal, dumping excess responsibility on the Lane.java. You can also see how the run functionality is all controlled by Lane as well. Note all the internal logic and handling from Lane.java.



Sequence Diagram for Lane After Refactoring:

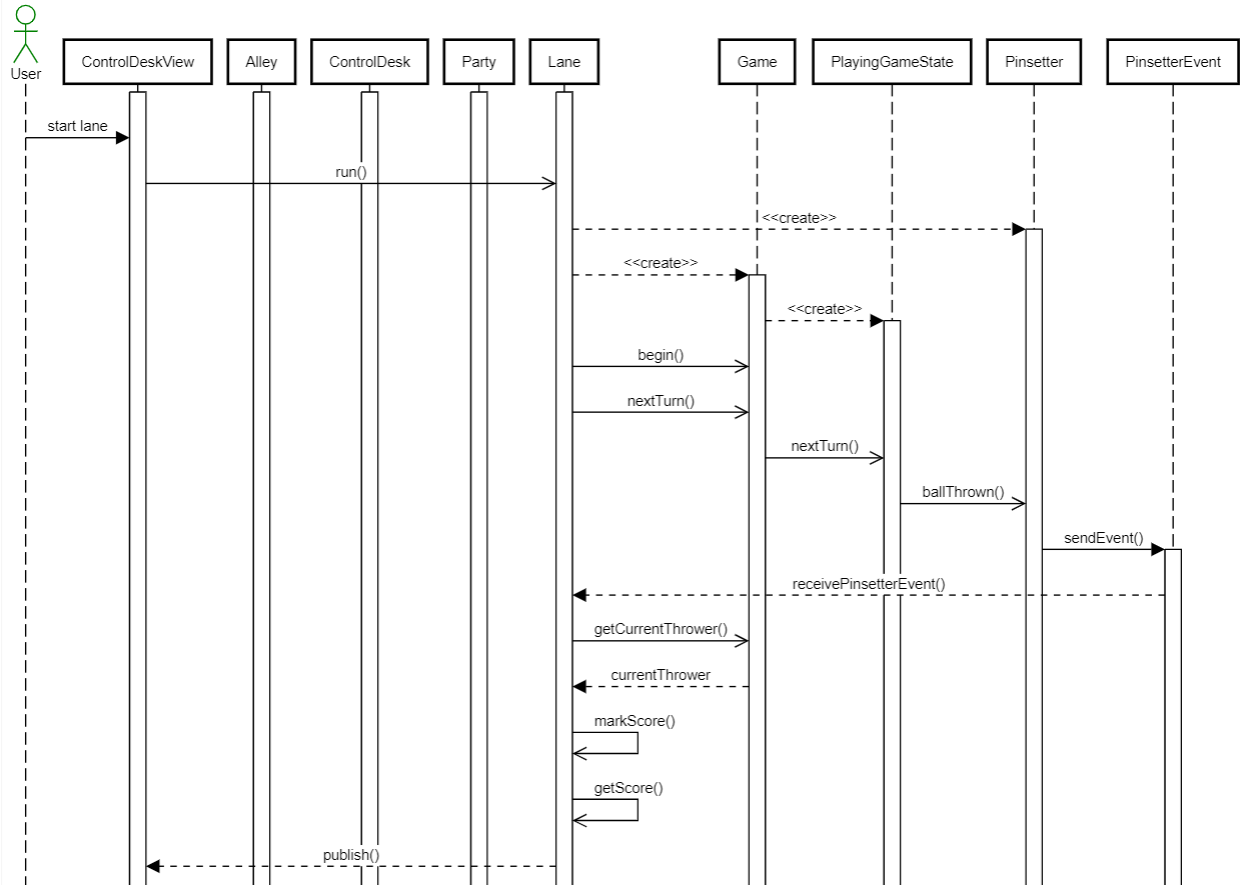
This diagram walks through what happens when a lane is started, with the emphasis on Lane.run and Lane.getScore: our refactoring focuses



Sequence Diagram for Lane After Refactoring (Lane.run() focus):

To increase readability and highlight the actions performed when Lane.run() is called, the Lane.getScore() has been listed simplistically, without the new refactoring progress.

The purpose of this sequence diagram is to compare to the original sequence diagram where Lane would directly call the Pinsetter and interact with many more classes. This sequence diagram shows how the Lane now creates a Pinsetter and a Game and then the two of them communicate and perform the necessary actions to bowl a frame/lane. It effectively shows how influential the state pattern was to reducing the overall workload of the Lane.java class. The nextTurn() function in the Game and PlayingGameState is where the logic of determining what happens at the Lane resides, therefore reducing the complexity of the class.



Sequence Diagram for Lane After Refactoring (Lane.getScore() focus):

To increase readability and focus on the getScore function, it can be assumed that the necessary actions have been taken to call Lane.run() which leads to the Lane.getScore(). Diagraming for this area has been covered in the document already.

This sequence diagram demonstrates exactly why our CogC for Lane.getScore() dropped from 111 to just 1. As you can see, instead of calling itself and returning the score (all internal logic) the logic is spread to many more classes. It is evident that the state pattern allowed for a much more efficient process in determining the score as shown in the diagram below. The work done to calculate has been offloaded and determined by four different concrete classes. Lane.getScore() calls ScoreCalulationState which then determines the state of the Game and performs the correct calculations based on its own arsenal of concrete states which represents the different kind of results a throw can have.