# <REFACTORING PROJECT>

*Design Refactoring Documentation*
*Prepared by <TEAM 5>:*


- Chase Chura <clc3020@g.rit.edu>
- Suky Kuye <ok5879@g.rit.edu>
- Mason Zhong <mz9865@g.rit.edu>
- Sanchit Monga <sm3468@g.rit.edu>
- Nirav Barman <nxb7819@g.rit.edu>

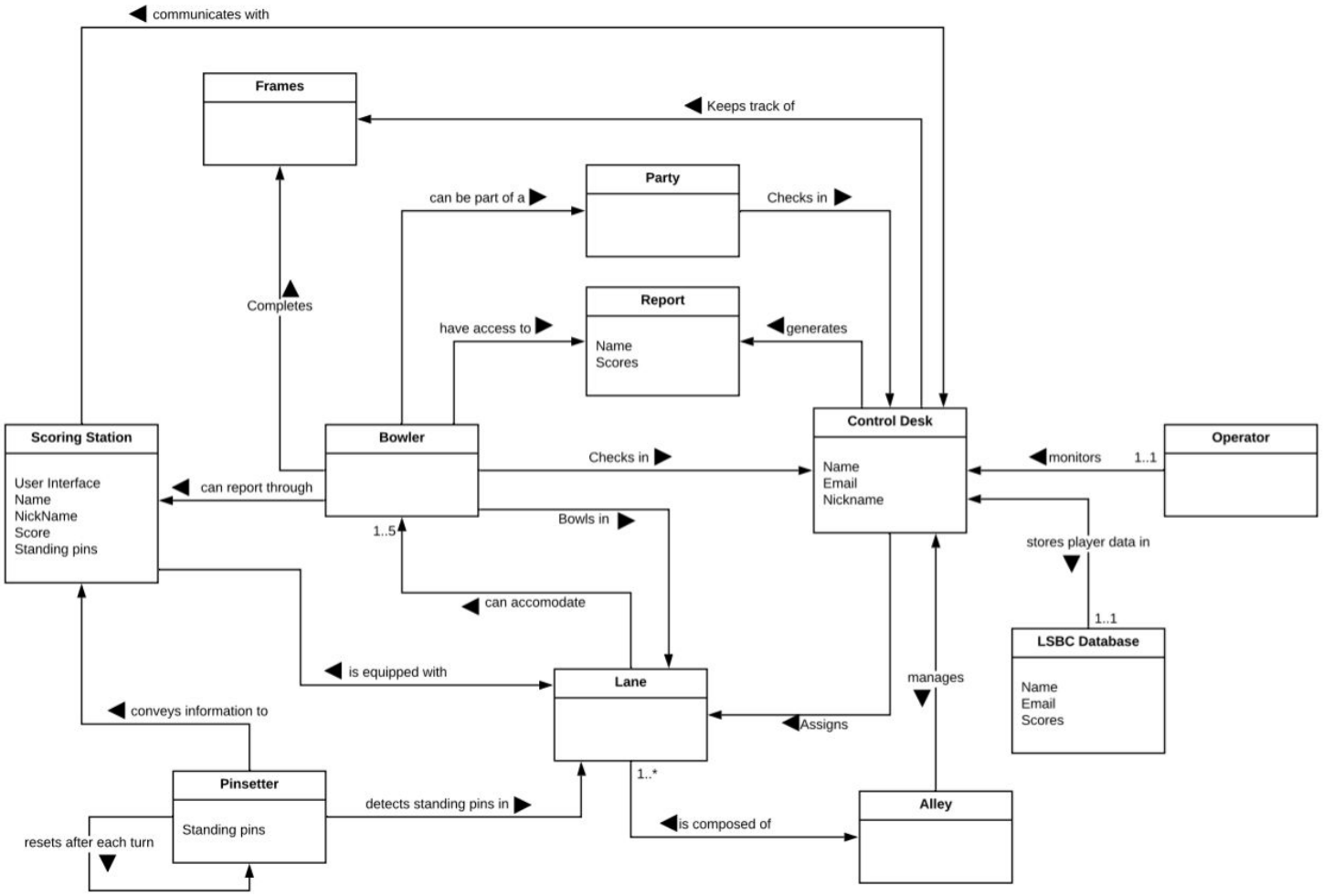
## Product Overview

This is a  Bowling Alley Management System that controls the Lucky Strikes Bowling Center (LSBC). Through a GUI, this system will monitor all parties and lanes within the building, through a central hub called the Control Desk. When bowlers check in as a group or party, they are assigned to an available lane. The order the party checked-in as, is the order they will bowl. Once the party starts bowling, the Control Desk entity will monitor the frames for each bowler.

Each lane is equipped with an individual scoring system. Currently, the system will simulate the entire game. The scoring system relies on an automatic pinsetter to keep track of the pins still standing after a throw, and that information is passed along to the lane to help keep score. There are components that monitor the lane and will get notified every time there is a new lane event. Most of these components are used to update the GUI.

Once the bowler has finished their game, the Control desk is notified by the lane. The lane also notifies the Control Desk when the last bowler has finished their last frame. At that point, the party may return to the Control Desk to decide whether they want to play another game or check ou

### *Domain Model*



# Analysis of Original Design

## *Design Weaknesses and Strengths*

This subsection discusses the original design's weaknesses and strengths, including incorporation of domain model entities and fidelity to the design documentation.

The original design of this software was not up to par. It did not follow many SOLID or GRASP design principles. The most obvious violations were to the Single Responsibility Principle and low coupling. Many classes within this system often held too much responsibility and became too complex, by calling upon multiple methods from other classes, which increased the coupling severely. There are some security issues as well. Some classes have unnecessary access to the file that keeps the bowler information. On top of that, Observers used in that subsystem also receive unnecessary information when notified about an event.

In relation to the design documentation and the final implementation, they did not exactly match. There were interfaces displayed in the UML Class Diagram and Domain Model, but they weren't used in the implementation. Some examples of these interfaces are the LaneEventInterface and LaneServer. In the UML Class Diagram, some methods were missing from some class diagrams. For example: one of the most complex classes in the implementation was missing methods such as run(), addPartyQueue() and other getters/setters. Missing this type of information allows for an inaccurate representation of the provided system and misinforms whoever is reading it.

There were some strengths present, nevertheless. There was an attempt to use the Observer design principle in numerous parts of the system. The multi-threaded implementation combined with the event-driven design, lead to a smoothly running system. The GUI was responsive and ran as expected, there were no hiccups or unexpected bugs. Looking from the outside, this system did exactly what it needed to do, based on the original project requirements.

## *Use of design patterns*

| **Name: PinsetterObserver** | | **GoF pattern: Observer** |
|---|---|---|
| **Participants** | | |
| **Class** | **Role in pattern** | **Participant's contribution in the context of the application** |
| PinsetterObserver | Observer | This class defines the interface for all components that want to observe events within the Pinsetter class. This interface will be composed of three classes. |
| PinsetterView | ConcreteObserver | This class will listen for the events within the Pinsetter class. Will need to be up to date and notified when an event occurs so as to accurately represent the pinsetter's state to the user. |
| Lane | ConcreteObserver | This class embodies all functionality of a Lane in the Bowling Alley system. It will need to be notified when an event occurs in Pinsetter so that state is represented effectively. |
| LaneStatusView | ConcreteObserver | This class will listen for events within the Pinsetter class. Will need to be up to date in order to have an accurate representation of Pinsetter's state. |
| Pinsetter | ConcreteSubject | This class embodies all of the functionality of pinsetting in the Bowling Alley system. It is important that whenever an event occurs, registered observers are notified. |
| **Deviations from the standard design pattern: There is no explicit Subject in this pattern. It only has the ConcreteSubject Pinsetter.** | | |

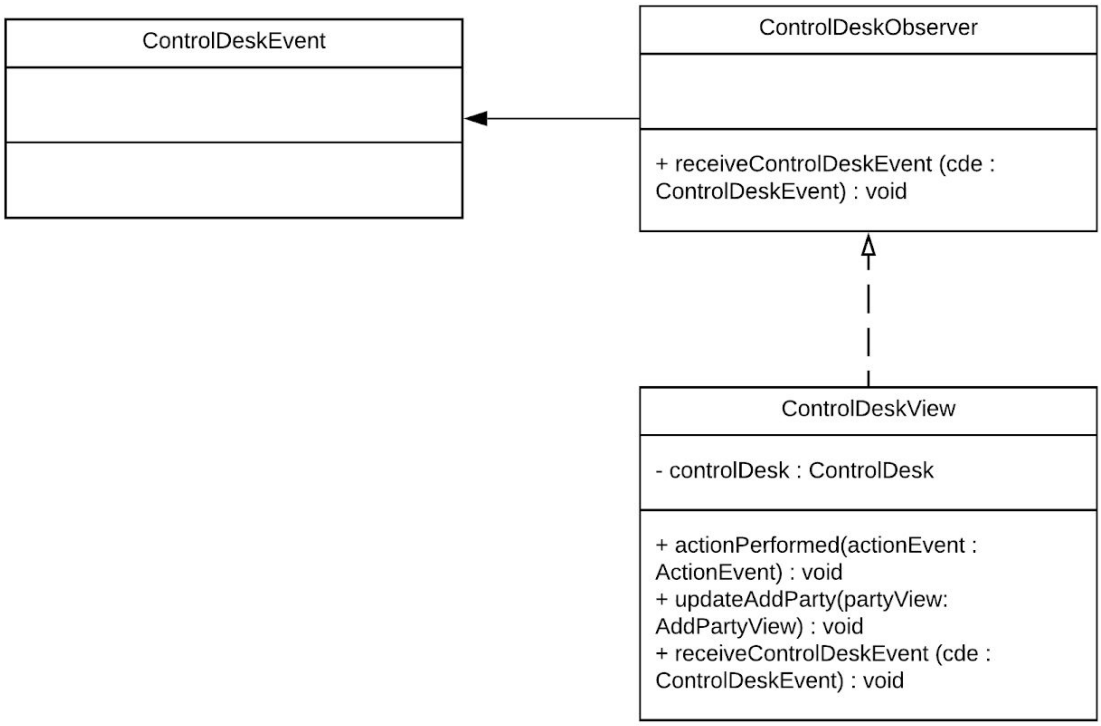
| **Name: LaneObserver** | | **GoF pattern: Observer** |
|---|---|---|
| **Participants** | | |
| **Class** | **Role in pattern** | **Participant's contribution in the context of the application** |
| LaneSubject | Subject | This class defines the interface for a class that wants to be observed as a lane. This will most likely be only the Lane class. |
| LaneObserver | Observer | This class defines the interface for all components that want to observe events within the Lane class. This interface will only be one or two classes. |
| LaneStatusView | ConcreteObserver | This class listens for events within the Lane class. It needs to be up to date in order to accurately represent the status of a lane in real-time. |

| | | |
|---|---|---|
| LaneView | ConcreteObserver | This class listens for events within the Lane class. It needs to be up to date in order to accurately represent the lane's state to the user. |
| Lane | ConcreteSubject | This class embodies all functionality of a Lane in the Bowling Alley system. It is important then when an event occurs in each respective lane, registered observers are notified. |
| **Deviations from the standard design pattern: There is no event reference being passed to observers. Observers receive all information from their reference to the lane class itself.** | | |

| **Name:ControlDeskObserver** | | **GoF pattern: Observer** |
|---|---|---|
| **Participants** | | |
| **Class** | **Role in pattern** | **Participant's contribution in the context of the application** |
| ControlDeskObserver | Observer | This class defines the interface for all components that want to observe events within the ControlDesk class. This interface will be composed of only one classe, ControlDeskView. |
| ControlDeskView | ConcreteObserver | This class will listen for the events within the ControlDesk class. Will need to be up to date and notified when an event occurs so as to accurately represent the ControlDesk state to the user. |
| ControlDesk | ConcreteSubject | This class embodies all of the functionality of the ControlDesk in the Bowling Alley system. It is important that whenever an event occurs, registered observers are notified of changes made in ControlDesk. ControlDeskView will receive these events. |
| **Deviations from the standard design pattern: There is no explicit Subject in this pattern. It only has the ConcreteSubject ControlDesk.** | | |

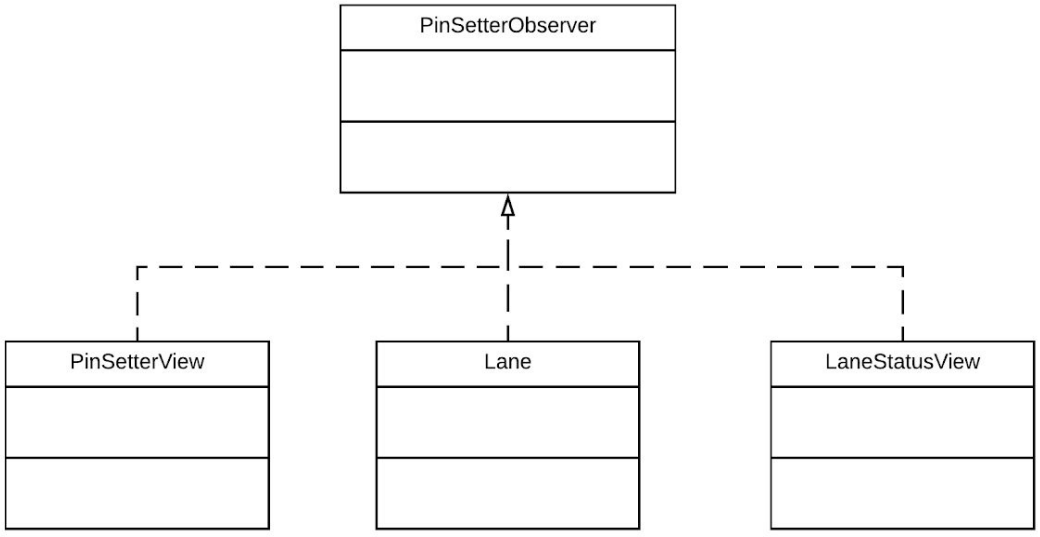
## *Subsystem and Class Structure*

### ControlDesk subsystem



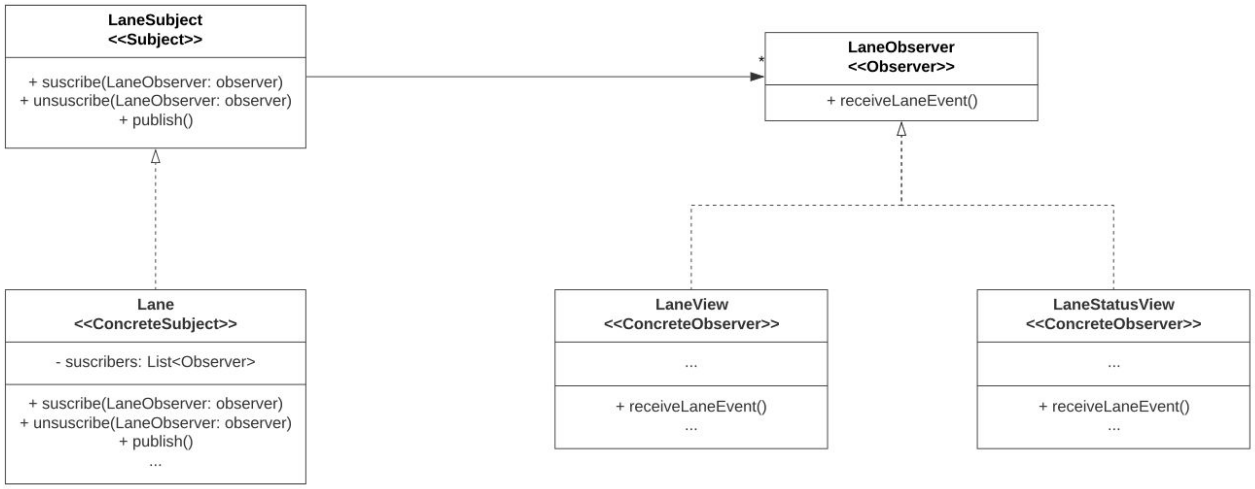

Subsystem that handles operations for the control desk. ControlDeskView receives party queues passed in as ControlDeskEvent which after received, is handled by the receiveControlDeskEvent method implemented from the ControlDeskObserver.

### PinSetterObserver subsystem



This is the subsystem that handles the individual pins in each lane. It keeps track of the number of pins and which pins went down after a roll and updates the Lane class to help calculate the score for the bowlers.

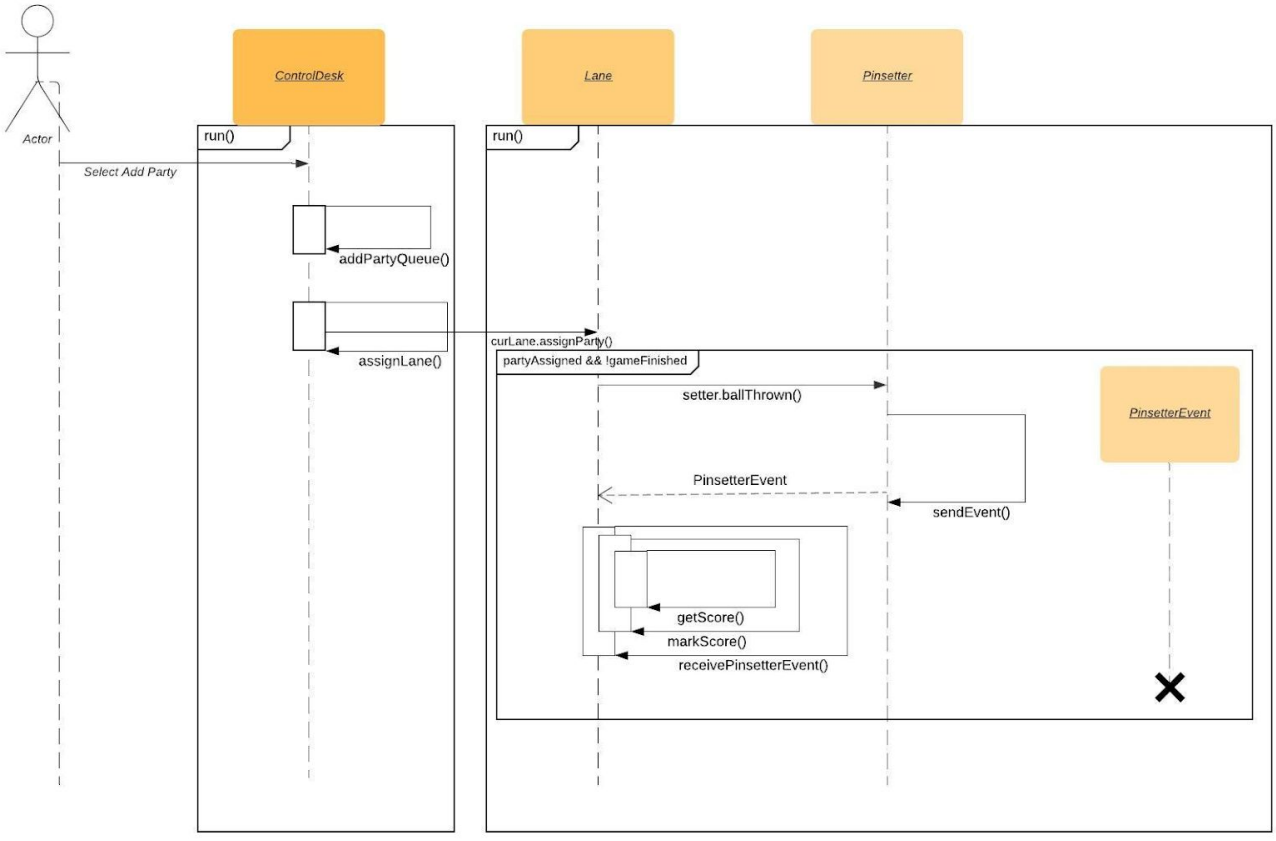
## LaneObserver Subsystem




This is the subsystem that handles operations for the lane. LaneStatusView and LaneView are notified once an event happens in the Lane class. These events range from a party finishing their game, the game score changing and the game being paused/unpaused. There is no event passed to the Concrete Observers.

## *Sequence Diagrams*

This diagram depicts the events that take place when a user wants to add a party to the alley. When the user selects to add a party, the ControlDesk calls addPartyQueue() on itself to add the party to a waitlist to be assigned to a lane. ControlDesk's run method is always running from the time it is initialized, so when a party gets added to the queue, the run method recognizes this and calls assignLane(). Within assignLane() the current lane gets assigned the party at the top of the queue with curLane.assignParty(). This brings us to the Lane class which also has a method run() which is constantly running. Within run() the first conditional checks whether a Party is assigned to the Lane and if the game is finished or not. Since the Party just got assigned to the Lane gameFinished is initialized as false, and so the loop enters the conditional. Within this conditional the game simulation occurs. It starts by calling the Pinsetter method ballThrown() which simulates a ball being thrown. The creates a new PinsetterEvent which gets sent to all of the Pinsetter's subscribers. The Lane calls receivePinsetterEvent() on itself to receive this event, which then causes it to call markScore() such that it can denote the score after the ball was thrown. markScore() calls getScore() to actually calculate what the score was. This loop continues until the game is finished, which happens when the last bowler in the party completes their last throw in the 10th frame of the game.



### *Metric Analysis*

We analyzed the metrics for the original code base which were calculated using the metrics analyzer. We analyzed mostly two types of metrics, Chidamber-Kemerer and Complexity metrics. After analyzing the results produced by the metrics analyzer, we sorted the results and analyzed specifically the parts of the code with the highest coupling, method complexity and cyclomatic complexity.

We first used the **Chidamber-Kerner metrics** method for the calculation and following are the top four that we found with the highest CBO, RFC and WMC.

| Class | CBO (Coupling between objects) | RFC (Response for Class) | WMC (Weighted method Complexity) |
|---|---|---|---|
| Lane | 15 | 62 | 72 |
| LaneView | 6 | 49 | 25 |
| ControlDesk | 10 | 39 | 18 |
| LaneStatusView | 10 | 35 | 17 |

We also looked at **Complexity Metrics** and we then looked at the individual method calls. Following are some of the results that we analyzed:

| Method | iv(G) (Design complexity) | v(G) (Cyclomatic Complexity) |
|---|---|---|
| Lane.getScore() | 1 | 38 |
| Lane.run() | 14 | 19 |
| LaneView.recieveLaneEvent() | 17 | 19 |

Initially these measurements gave us an idea of the system that the classes were highly coupled, and when we actually looked at the code we found long and complex methods. We found a lot of code smells, specifically, parts with long methods, long parameter lists, bad variable names, uncommented code and duplicated code.

# The Refactored Design

## *Refactoring #1: Reduce responsibility of LaneEvent*

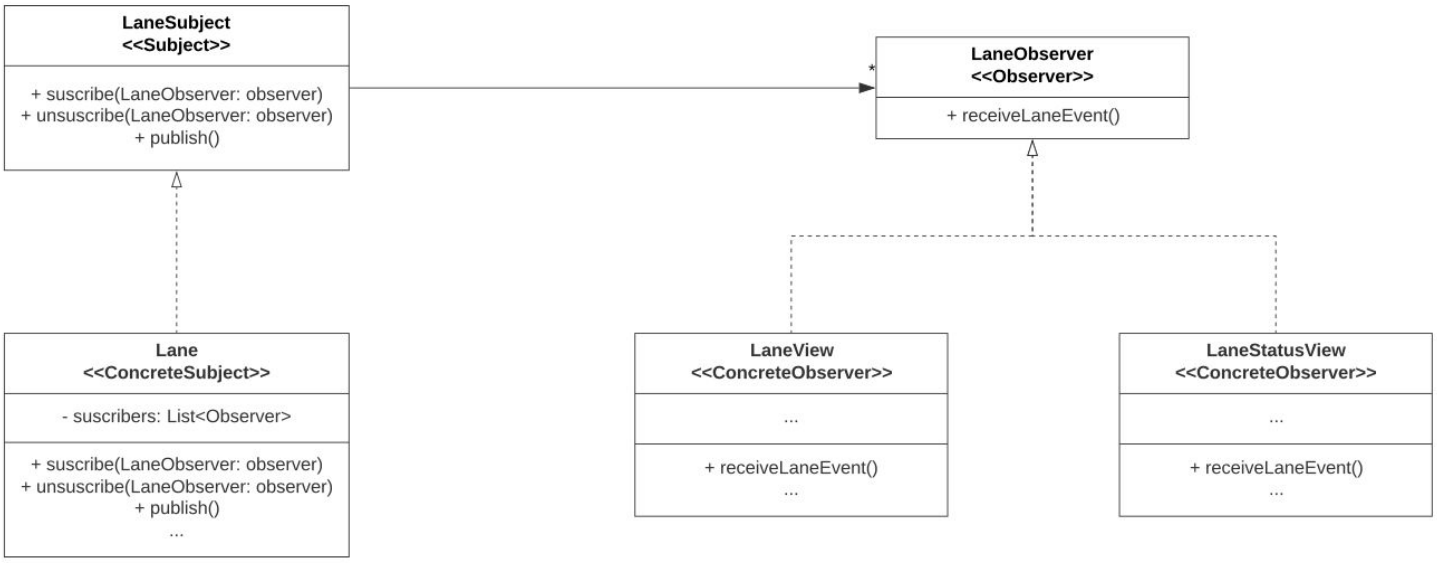
| | |
|---|---|
| **Refactoring identification** | Reduce the responsibility of LaneEvent |
| **Metric evidence** | Chidamber-Kemerer Metrics: Coupling between Objects |
| **Other evidence** | Unnecessary amount of information it carries<br>Too many parameters for the constructor |
| **Standard refactoring pattern (if any)** | Observer |
| **Description of the refactoring** | This class was removed from the system. It added no real value to the system, it's intended purpose was to carry information about the lane event for the Concrete Observers. But, it was discovered that all the Observers had a direct reference to the Lane class already, so they already had access to that information. |
| **Classes involved** | LaneSubject, Lane, LaneEvent, LaneView, LaneStatusView, LaneObserver |

This refactoring centered around removing the class LaneEvent from the system. LaneEvent carried an instantaneous snapshot of the lane at that time. LaneEvent was essentially just holding duplicate data that could have been passed directly from the Lane class. The removal of this class limited the unnecessary sharing of Lane's attributes. It now gives Lane the sole responsibility of sharing it's information and gets to see who calls upon the information. Unfortunately, it increases the coupling between itself and the Concrete Observers, because now they rely upon Lane updating them when an event has occurred, as well as being able to provide more information about it. Information hiding stayed the same. Almost all crucial attributes of the Lane class can be retrieved through getters, but they can't be modified. Not having a dedicated LaneEvent class also reduces the extensibility because of the direct dependency on Lane.

A Subject component was added to complete the Observer design pattern: LaneSubject. This is an interface that provides a skeleton of adding/removing observers, as well a method for notifying them when an event in Lane occurs. This class doesn't add extensibility or reusability because even if another class implements this interface, the observers rely on the Lane class specifically for information on the event.

**Refactored Class Structure**

Provide class diagrams for the areas that you refactored in the design. Follow the guidance provided above for class diagrams.

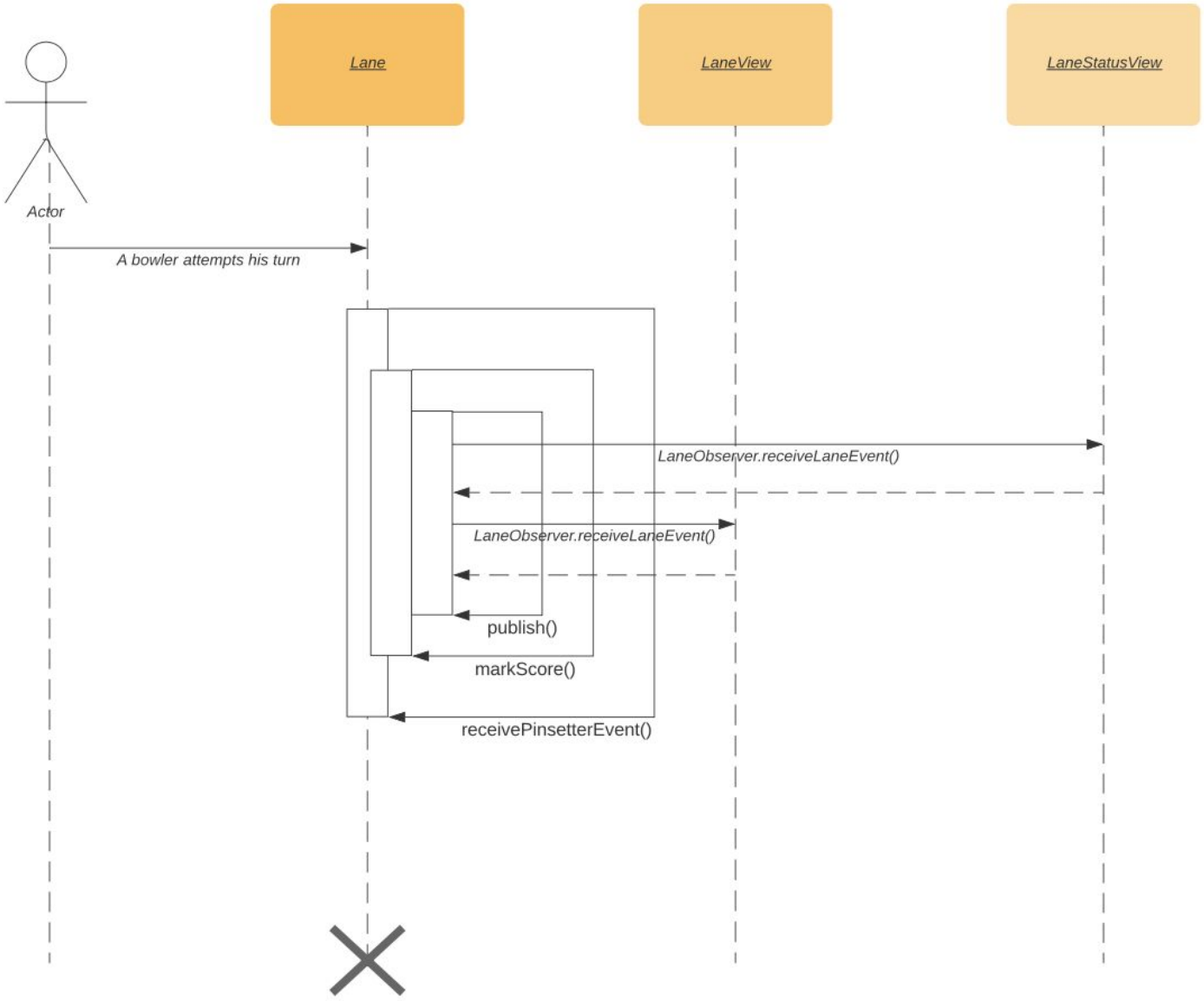


**Design Pattern**

| **Name: LaneObserver** | | **GoF pattern: Observer** |
|---|---|---|
| **Participants** | | |
| **Class** | **Role in pattern** | **Participant's contribution in the context of the application** |
| LaneSubject | Subject | This class defines the interface for a class that wants to be observed as a lane. This will most likely be only the Lane class. |
| LaneObserver | Observer | This class defines the interface for all components that want to observe events within the Lane class. This interface will only be one or two classes. |
| LaneStatusView | ConcreteObserver | This class listens for events within the Lane class. It needs to be up to date in order to accurately represent the status of a lane in real-time. |
| LaneView | ConcreteObserver | This class listens for events within the Lane class. It needs to be up to date in order to accurately represent the lane's state to the user. |
| Lane | ConcreteSubject | This class embodies all functionality of a Lane in the Bowling Alley system. It is important then when an event occurs in each respective lane, registered observers are notified. |
| **Deviations from the standard design pattern: There is no event reference being passed to observers. Observers receive all information from their reference to the lane class itself.** | | |

No new design patterns were introduced with this refactor design. The Observer design pattern was already in place, but removing this class modified the implementation of it, so modifications had to be made. The main modifications were removing all references of the LaneEvent class. This resulted in the publish() method of the Subject carrying no event. All information needed about the event can be retrieved from Lane's getter methods.

## Sequence Diagram



## ***Refactoring #2: Code Cleanup***

| | |
|---|---|
| **Refactoring identification** | Code Clean up |
| **Metric evidence** | |
| **Other evidence** | Found duplicate code, bad variable names and unused code |
| **Standard refactoring pattern (if any)** | N/A |
| **Description of the refactoring** | Remove duplicate code and replace them with method calls, change variable names, remove unused code and comments, change iterator to use for each, replace vectors with Array List |
| **Classes involved** | All |

**MVC**

The first change we made was dividing up all the classes into packages such that they would be grouped in a MVC architecture. This allowed us to get a better understanding of the original code, thus allowing us to easily create a plan to refactor the code and divide up work.

**Lane Class**

The Lane class contained most of the logic to run the actual simulation of the game. It had a large method, run(), which originally was full of conditionals dictating how to approach each step of a game and what to do once a game was completed. We separated this logic into two new methods to improve the separation of concerns. The first is bowlerSimulation() which now dictates what to do for each bowler for one frame. It sets the thrower to the next in the order, and then calls setter.ballThrown() to have the Pinsetter simulate their throws for the frame. The second method is frameCompleted() which handles what to do at the bottom of each frame, meaning when all of the bowler's in the party completed their throws. The logic for the Lane class was further refined in the Lane.run() refactoring which will be detailed in the next section.

**Vector to ArrayList**

Throughout the code we replaced all Vectors with ArrayLists. We made sure that there was no part of the code that was not thread safe and had arrayLists in there instead of Vectors. Although vectors are thread-safe but if they are used in places where we don't have multiple threads accessing them we can replace them with ArrayLists. As synchronization makes the components of the code run slower and one of the reasons is: ArrayLists and Vectors both resize differently. ArrayLists increments 50 percent of the current array size if the number of elements exceeds its capacity, while vectors increments 100 percent, which doubles the current array size.

These are some of the advantages of ArrayLists over the Vectors, since our system used a large number of Vectors, we replaced them with ArrayLists.

**Iterators to For-each Loop**

After replacing all the Vectors with ArrayLists, we replaced all the iterators with a for-each loop.

Although there are no performance differences between the both, for-each looks syntactically better and is easy to interpret. Since we were not using Vectors, we didn't need the iterators, and instead we added for-each loops in the system.

**Views**

All the view classes in the view package had very long GUI initializations. All the components of the GUI were defined in the constructor and were all mixed up. We took out the initializations of each part of the GUI and created smaller separate methods that define each component. We added descriptions for each part of the GUI that was defined in the methods and organized it so that each component can be interpreted or modified easily.

We also broke up the actionPerformed functions for some of the view classes into smaller methods. They were broken down according to their response to the changes in the system, or when a button was pressed in the GUI.

Although this did increase the overall coupling between the methods in the classes, it did organize and divide up all the GUI's components into parts.

**General Cleanup**

We combed through the original code and made sure all methods were properly commented, and that the comments were in a relatively uniform format. This improved readability and makes it easier for other developers to understand the methods if they were to read the code. Apart from that we changed some of the attribute names to names that made sense in the context they were used. We also broke down some parts of the code into more logical methods so that it looks more organized.

Following is the list of some pieces of unused code that we removed:

- redundant imports
- unused methods
- unused attributes
- redundant exceptions
- unused classes and interfaces
- redundant castings
- redundant initializations
- redundant assignments
- meaningless descriptions and comments

Following are some of the additions to the code:

- Added comments for each and every function along with the description of each parameters
- Divided large functions into smaller methods
- Added descriptions for all the attributes of the system
- Changed names of some attributes to something meaningful and more descriptive
- Indented all the code
- Simplified conditional expression

After going through the code cleanup, we have made the code more readable and organized. The overall coupling of the system has still increased because we broke down large functions into smaller functions but it does support the reusability. The system adheres to the law of demeter, as we have reduced the maximum number of direct calls to other classes. Since we broke down large methods, this makes testing the code and analyzing parts of it much easier and better.

**Description of Changes by each Class**

| Class | Changes |
|---|---|
| BowlerFile | ● Removed unused exception<br>● Replaced **allBowlers** vector with ArrayLists |
| LaneEvent | ● Removed lane event (part of Refactoring #1) |
| LaneSubject | ● Added this class (part of Refactoring #1) |
| AddPartyView | ● Removed unused attributes: **lock**<br>● Broke down the GUI initializations in the constructor into 3 functions based on the 3 components in the GUI (**initializePartyPanel(), initializeBowlerDatabase()** and **initializeButtonsPanel()**).<br>● Broke down the actionPerformed function into 4 smaller functions based on which action triggers the response (**addPatronClicked(), remPatronClicked(), finishClicked() and newPatronClicked()**)<br>● Removed unused method: **getNames**<br>● Replaced **bowlerdb** vector with ArrayList |
| ControlDeskView | ● Broke down the GUI initializations in the constructor into 3 functions based on the 3 components that are part of the GUI (**initializeControlPanel(), initializeLaneStatusPanel()** and **initializePartyQueuePanel()**)<br>● Updated the receiveControlDeskEvent to receive an array instead of a vector |
| EndGamePrompt | ● Broke down the GUI initializations in the constructor into 2 functions based on the 2 components that are part of the GUI (**initializeLabelPanel() and initializeButtons()**)<br>● Removed redundant attributes: **selectedNick** and **selectedMember**<br>● Added description for the variables and changed the name of some confusing attributes.Changed the variable name **result** to **buttonPressed**, as the integer represents which button was pressed. Also added the description for the attribute |
| EndGameReport | ● Broke down the GUI initializations in the constructor into 2 functions based on the 2 components that are part of the GUI (**initializePartyPanel()** and **initializeButtonPanel()**)<br>● Removed unused functions: **main()** and **destroy()** |

| | |
|---|---|
| Lane | • Broke down **run()** into smaller functions to separate concerns (**bowlerSimulation()** and **frameCompleted()**) |
| LaneStatusView | • Broke down the GUI initializations in the constructor into a smaller function based on the components that are part of the GUI (**initializeButtonPanel()**)<br>• Removed the **foul** JLabel as it was never being used |
| LaneView | • Reduced the complexity of **receiveLaneEvent()** and divided the GUI initialization into a smaller method (**initializeButtonPanel()**).<br>• Simplified Conditional expression in **receiveLaneEvent().** |
| NewPatronView | • Broke down the GUI initializations in the constructor into 2 functions based on the 2 components that are part of the GUI (**initializeAddNewPatronPanel()** and **initializeButtonsPanel()**) .<br>• Removed unused method:**done**(). |
| PinsetterEvent | • Updated variable names in constructor to be more relevant |
| PinSetterView | • Broke down the GUI initializations in the constructor into 2 functions based on the 2 components that are part of the GUI (**initializeRolls()** and **initializeGrid()**)<br>• Optimized the **initializeGrid()** part for the GUI in the constructor. Used a loop and stored everything in the lists instead of adding the JPanels in the list one by one |
| Queue | • Updated to use ArrayList<Object> instead of vector<br>• Changed **asVector()** to **getQueue()** |

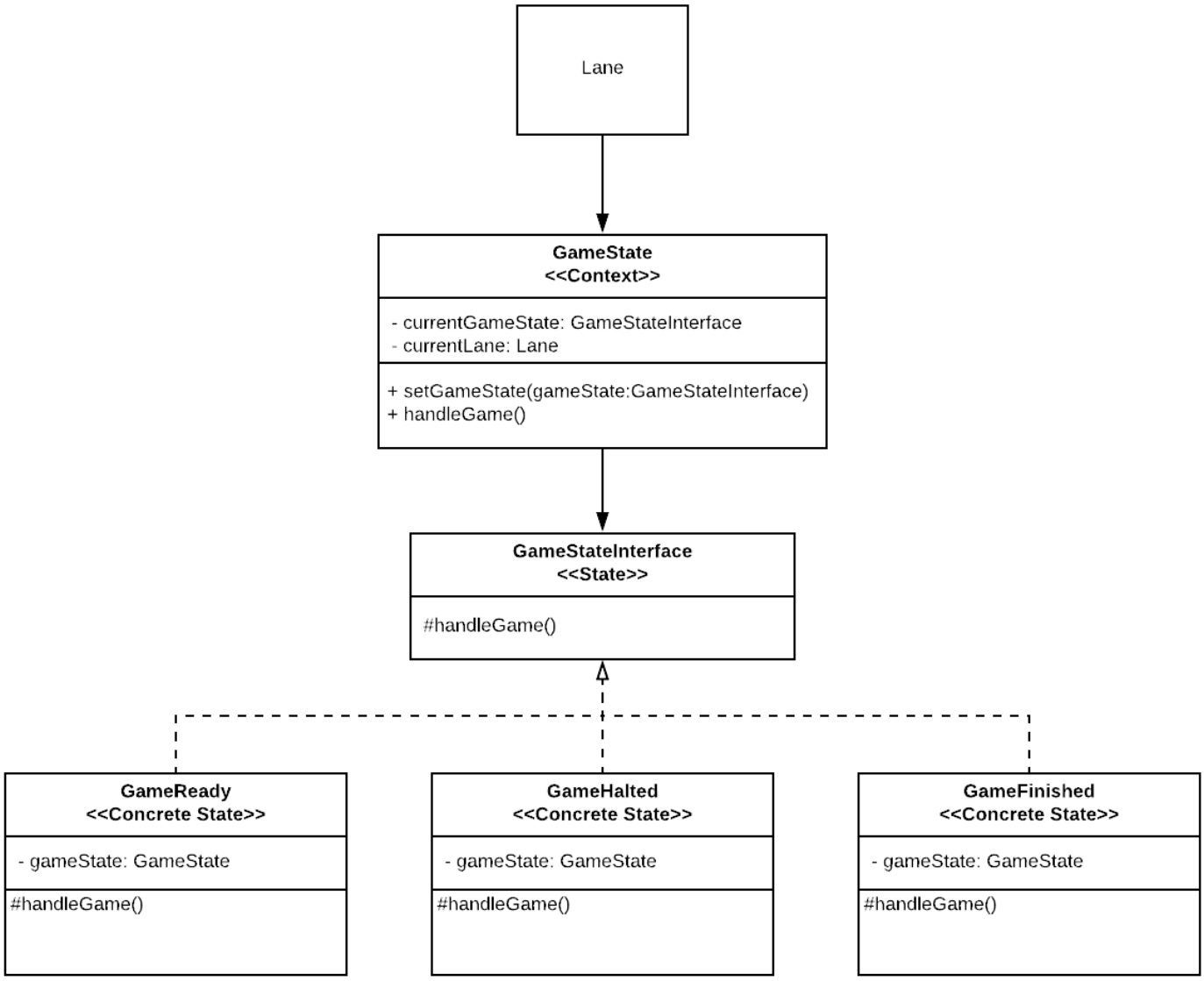
## ***Refactoring #3: Fix Lane.run()***

| | |
|---|---|
| **Refactoring identification** | Reduce complexity of Lane.run() method |
| **Metric evidence** | High cyclomatic and design complexity |
| **Other evidence** | Mess of nested if statements and was very confusing to follow |
| **Standard refactoring pattern (if any)** | State Pattern |
| **Description of the refactoring** | The run logic for this method was broken up into three different states that can be seen. A game can be running, or ready. A game can be halted while there is maintenance running on the lane, and the game can be finished. We created three Concrete State classes with this in mind, and moved the functionality that was in the initial run into these three classes. State is then set in the Lane.run() method to whichever state it is currently in. |
| **Classes involved** | Lane |

To reduce the complexity of Lane's run() method, we decided to refactor it and apply the state design pattern. Initially, the game operations when the game is ready to be played, finished and halted, were all handled in Lane.run(). This method had a significant conditional complexity.

With the state pattern, we were able to split the responsibility of the Lane.run() method into separate concrete state objects. This was easily refactored by putting the operations within the conditionals in their corresponding concrete state. Thanks to this refactor, we have reduced the conditional complexity of the Lanerun() method and also encapsulated the method's logic in dedicated concrete state classes, adhering to the Single-Responsibility principle.

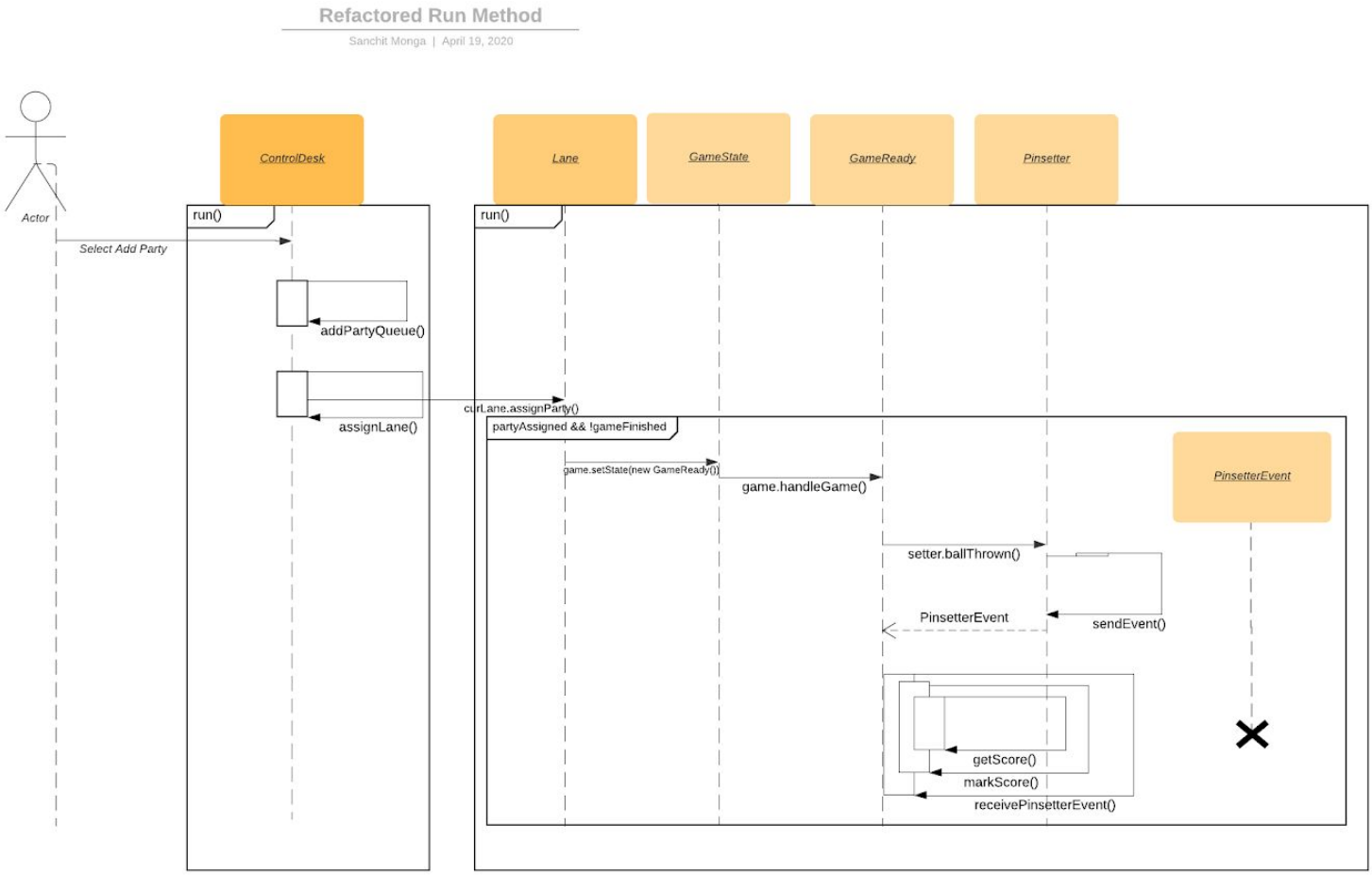
## Refactored Class Structure



**Design Patterns**

| **Name:RunGame** | | **GoF pattern: State** |
|---|---|---|
| **Participants** | | |
| **Class** | **Role in pattern** | **Participant's contribution in the context of the application** |
| GameState | Context | This is the class that will control which state the game is set to. It will take in the current LaneIt will also call the appropriate handleGame() method for the given state that it is currently in. |
| GameStateInterface | State | This is the interface that defines the handleGame() method that all of the Concrete States will implement. |
| GameReady | ConcreteState | This is the Concrete State that represents the game in its running, or ready state. This class has the handleMethod() that will control Lane functionality while it is running. |
| GameFinished | ConcreteState | This is the Concrete State that represents the game in its finished state. The handleGame() will end the game, and restore all values back to their original values. |
| GameHalted | ConcreteState | This is the Concrete State that represents the game in its halted, or paused state. This will pause the current thread until the user restarts it. |
| **Deviations from the standard design pattern: None** | | |

## Sequence Diagram



## Implementation

For the LaneEvent refactoring, a few things were done in the implementation to make improvements. The LaneEvent class was deleted and all references to it were removed from the project. The getter methods from the LaneEvent classes were injected into the Lane class, so all the classes that relied on LaneEvent (LaneView, LaneStatusView) were modified to use the new methods added to the Lane class instead. Since the LaneEvent class was no more, all references with method calls within subsystem classes were removed.

For the code cleanup refactoring we wanted to make the program more readable, and we also worked to improve performance and reduce complexity. We created model, view, controller, and utility packages and assigned each class to the appropriate package. We then combed through for any unused or inappropriate code and comments so we could remove them, and moved redundant code to new methods to increase the polymorphism. With this we also divided down large methods into smaller methods to separate concerns.

To improve performance we replaced all vectors with ArrayLists as they are slightly more efficient in this context, and allowed us to replace iterators with forEach loops to increase readability. Lastly we formatted all the code and ensured that each method had appropriate and descriptive commenting. This refactoring affected each and every class, but the Lane class and the classes in the View package underwent the biggest changes as they were the biggest contributors to overly complex methods and redundant code.

By implementing the state design pattern to the Lane.run() method, we reduced the conditional complexity of the method. The immense responsibility that it held was split into the concrete states, GameReady, GameFinished and GameHalted; where the game’s state was stored in GameState. This way, we can have a better adherence to the Single-Responsibility principle.

### *Metric Analysis*

To determine the improvements of the refactored design we used the same metrics to analyze the code as we did before:

These are the **Chidamber-Kerner metrics** results after refactoring the code:

| Class | CBO (Coupling between objects) | RFC (Response for Class) | WMC (Weighted method Complexity) |
|---|---|---|---|
| Lane | 15 | 60 | 80 |
| LaneView | 5 | 50 | 26 |
| ControlDesk | 10 | 34 | 17 |
| LaneStatusView | 9 | 36 | 18 |

These are the **Complexity Metrics** results after refactoring the code:

| Method | iv(G) (Design complexity) | v(G) (Cyclomatic Complexity) |
|---|---|---|
| Lane.getScore() | 1 | 37 |
| Lane.run() | 6 | 9 |
| LaneView.recieveLaneEvent() | 14 | 19 |

After calculating and analyzing the average results of the metrics, we came to the conclusion that there were definitely some overall improvements. The average cyclomatic complexity of the system decreased from 2.45 to 2.18 and the design complexity decreased from 1.94 to 1.77.

The average coupling between the objects increased from 4.79 to 4.93 since we created a lot of new smaller functions. The average weighted method complexity fell from 11.54 to 11.07 which makes sense because we divided large methods into smaller methods.

After getting rid of the LaneEvent we made changes in the Lane to include some additional methods, because of which the weighted method complexity of the class went up from 72 to 80.

Also the individual metrics for lane.run() method for which Cyclomatic complexity decreased from 19 to 9, since we refactored the method using the state pattern. The overall design complexity for AddPartyView.actionPerformed() decreased from 10 to 5 which was the result of dividing all the functionality into smaller methods which are part of code clean up.

## Reflection

Initially, receiving a system of this magnitude with no prior experience in the topic can be a little bit overwhelming. It is already difficult enough to get up to speed quickly, but when the code is poorly written and documented, it makes it 10x difficult. Our refactoring process began with identifying the hot spots where a refactoring would be beneficial. This included analyzing code metrics to identify classes with high complexity/coupling, as well as each member of the team browsing through the project directory and manually identifying things other weaknesses, such as duplicate code.

The metric analysis was particularly useful because there was a surplus of different categories to choose from, which showed a broad range of data. Our team focused on the Chidamber-Kemerer, Complexity and Dependency metrics. The data provided from those metrics, as well as each member's manual analysis, allowed for us to create an extensive list of potential areas to refactor. The list narrowed down by impact on the system and time constraints. However, there is a surplus of refactorings that could've been done for the system.

This project reiterated the importance of planning and design to our team. Majority, if not all, of the design issues and bugs that persist today could've been eliminated if the original software design team had taken more time to design how their system would work. Our lives would also have been easier if there was helpful documentation.