Design Document

Assignment Three

Design And Analysis Of Software Systems

Project Information

1. Team Members

Name	*Roll Number*
Adarsh Dharamavedan	2018111012
Jyoti Sunkara	2018101044
Pushkar Talwalkar	2018101010

2. Date Of Submission

Submitted on *9th April 2020, Thursday*.

Overview

The original project contains a java application that allows for the management of a bowling alley. Within the application, a collection of lanes can be monitored through a control desk, parties of bowlers can be assigned to the lane, the configuration of the pinsetter can be viewed, and the bowler's scores can be printed at the completion of the game. This original source materials contains designs, code, and documentation that exhibits poor software engineering design principles as well as anti-patterns.

Our purpose was to improve the designs and source code of this application. Metrics were gathered for the project, and along with visual code and documentation inspection potential areas for refactoring were determined.

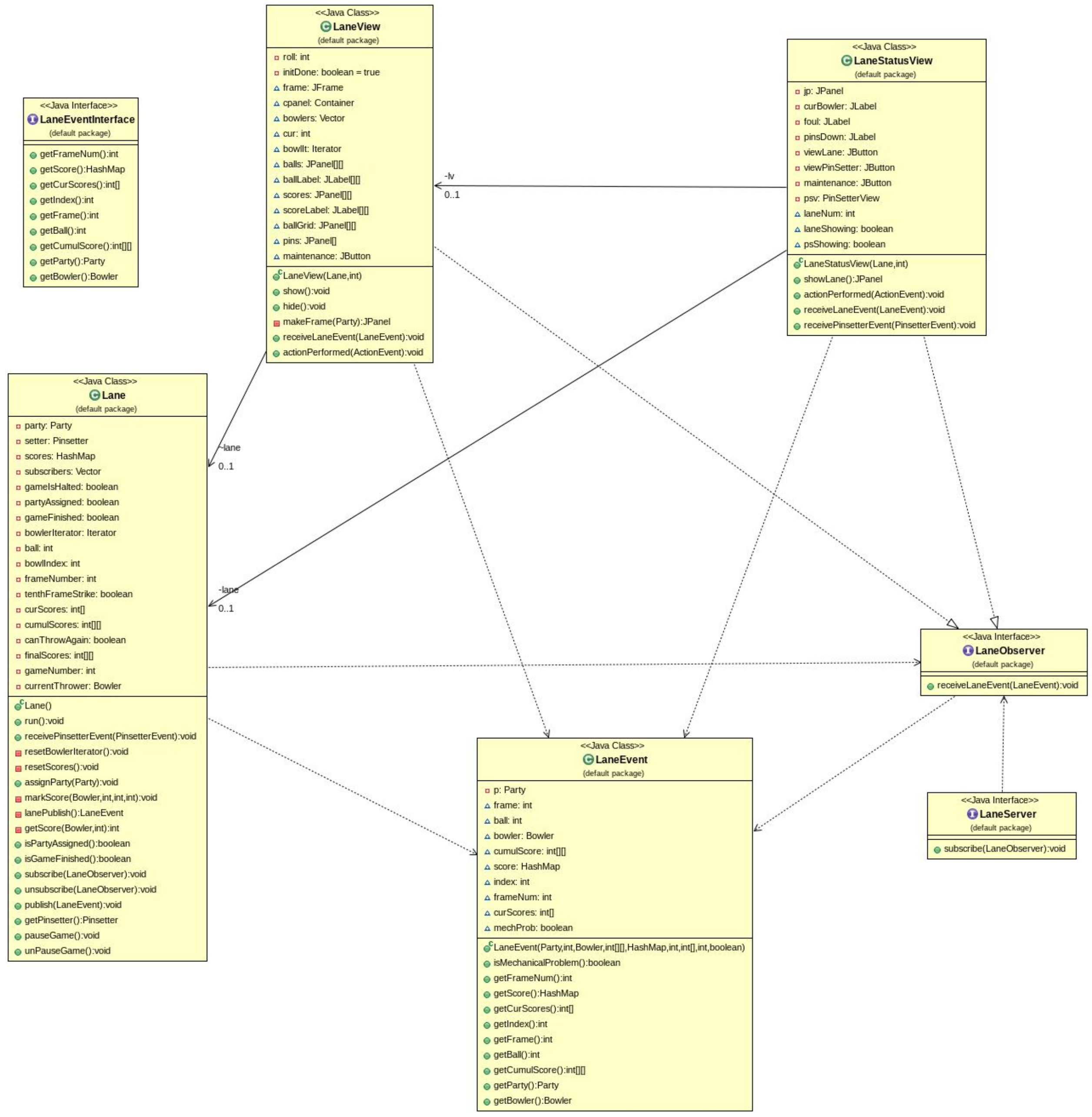
The features and models can be seen as per the UML Class and Sequence diagrams given below.

UML Class Diagrams

Interfaces, inheritance, generalization, association, aggregation, composition, cardinality and role indicators have been shown via appropriate arrows and markings.

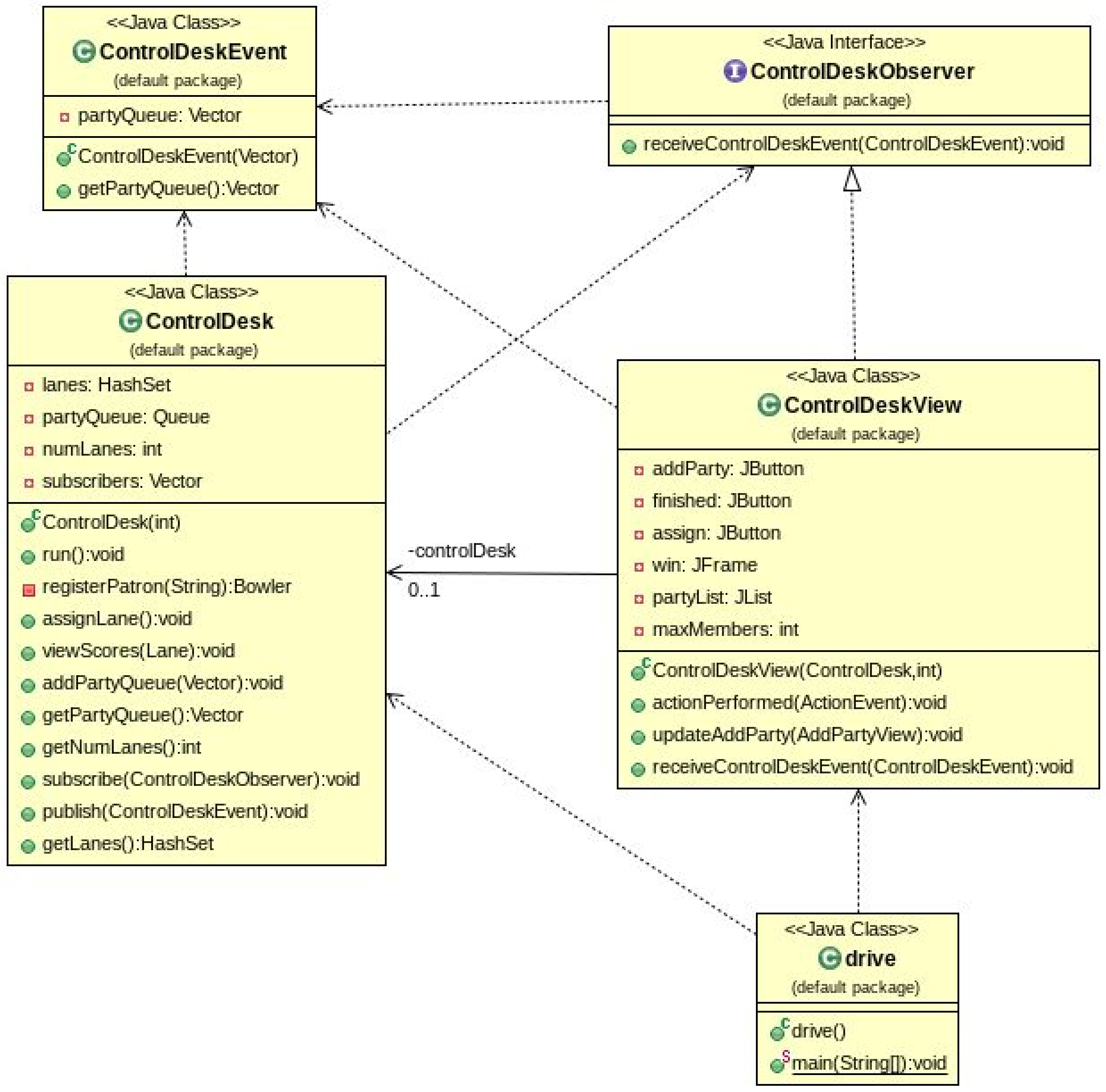
Lane Classes

Following is the UML Class Diagram for all the classes and interfaces dealing with the lanes, lane display and lane events.



Control Desk Classes

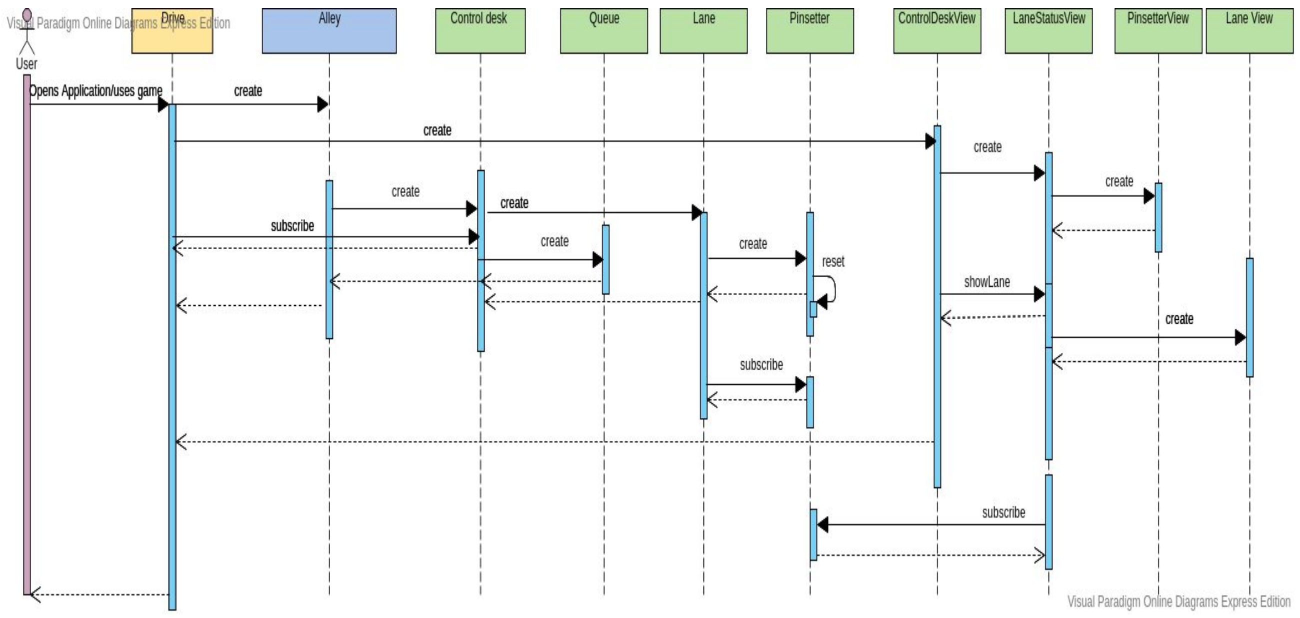
Following is the UML Class Diagram for all the classes and interfaces dealing with the control desk events, control desk view and control desk.



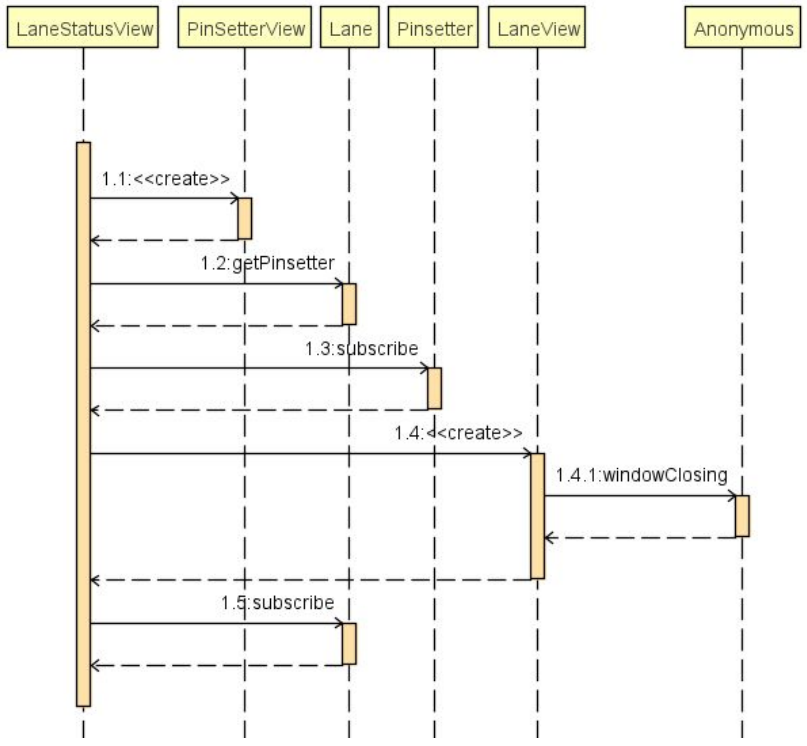
UML Sequence Diagram

Static and dynamic characteristics of the program have been represented in the following sequence diagram for the bowling game.

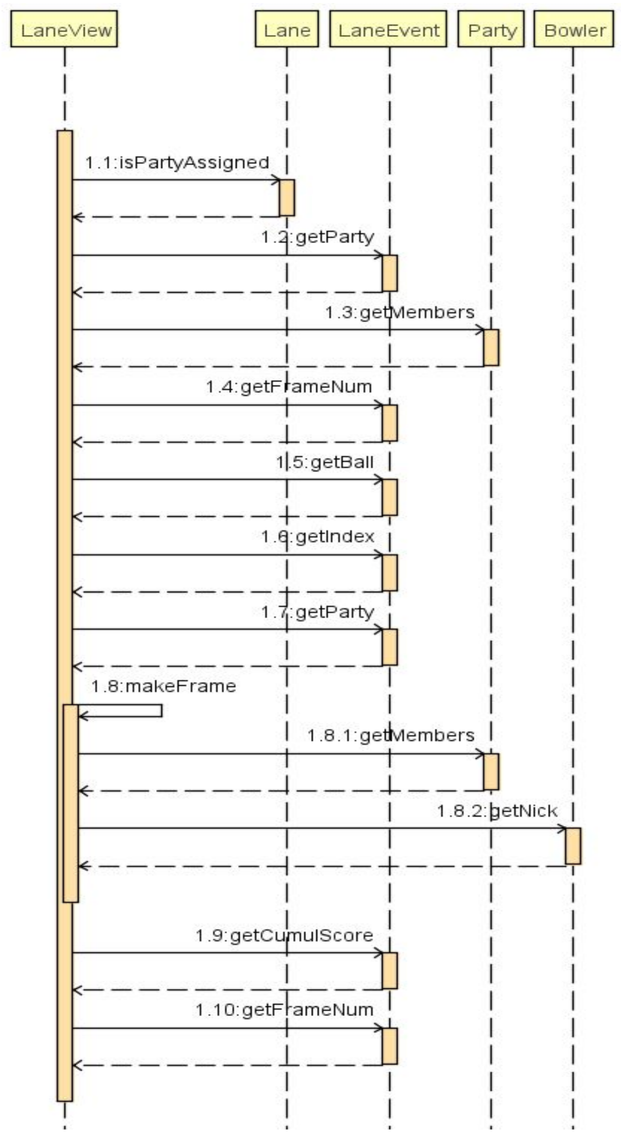
Overall:



Lane Status View:



Receive Lane Event:



Class Responsibility Table

The major classes and their responsibilities have been listed along with the methods they call upon.

S. No	Name	Methods	Responsibility
1	AddPartyView	AddPartyView actionPerformed valueChanged getNames getParty	Handles the display to add a new party.

		updateNewPatron	
2	Alley	Alley getControlDesk	A outer container class, for the bowling simulation
3	Bowler	Bowler getNickName getFullName getNick getEmail equals	Stores the person’s nickname, full name and email for basic retrieval.
4	BowlerFile	getBowlerInfo putBowlerInfo getBowlers	Keep Bowler information in a file for getting a vector of all Bowlers, inserting information and searching for a bowler by nickname.
5	ControlDesk	ControlDesk Run registerPatron assignLane addPartyQueue getPartyQueue getNumLanes Subscribe Publish getLanes	Initializes input number of lanes, assigns lanes, creates a party of vectors of nicknames and maintains wait queue.
6	ControlDeskEvent	ControlDeskEvent getPartyQueue	Maintain a vector of strings containing the names of the parties in the wait queue.
7	ControlDeskView	ControlDiskView actionPerformed updateAddParty receiveControlDeskEvent	Displays the Control Desk View and controls all the possible inputs.
8	drive	main	This is the class which initializes the game, by setting parameters(such as number of lanes and maximum patrons per party) and then initializing all the necessary objects.
9	EndGamePrompt	EndGamePrompt actionPerformed getResult destroy	Display the prompt that shows up at the end of the game and handle all the different inputs to the prompt.
10	EndGameReport	EndGameReport actionPerformed	Display the report at the end of the game as per the user’s request.

Original Design

Weaknesses

The original source material contains designs, code, and documentation that exhibits poor software engineering design principles as well as anti-patterns.

Antipatterns

1. The Blob

No.	*Description Of Weakness*	*Fixed*
1	Single controller class, multiple simple data classes seen throughout implementation in files such as Lane.java and ControllerView.java	Split into appropriate classes wherever necessary

2. Lava Flow

No.	*Description Of Weakness*	*Fixed*
1	Large commented-out code with no explanations	Removed dead code and gained a full understanding of any bugs introduced.
2	Lot’s of “To Do” code	Wrote the code wherever necessary.

The Lava Flow code was refactored as mentioned under the Dead Code Code Smell in detail.

3. Functional Decomposition

No.	*Description Of Weakness*	*Fixed*
1	Classes with a single method in use, no real	Fixed by moving into

	use of the class data structure	existing class and/or combining classes

4. Golden Hammer

No.	*Description Of Weakness*	*Fixed*
1	Several deprecated functions such as show(), hide(), new Integer() used throughout the code.	Development via linting tool such as Intellij helped fix this issue.

5. Cut-And-Paste Programming

No.	*Description Of Weakness*	*Fixed*
1	Duplication/ Dead Code in many .java files	Refactored and eliminated using the DRY principle

Strengths

- The coupling amongst all the classes has been kept low
- A strong cohesion amongst related classes exists
- The size of the code files has been kept low and optimum
- One of the biggest strengths of the original design is the adherence it has to MVC. The model classes are highly independent. The control classes communicate between view and model classes as expected.

Fidelity To Design Documentation

- The original code has for the most part abided to the design documented listing the features it needs to be implemented
- . A bowling alley is composed of a number of bowling lanes and parties bowl on these lanes
- Each lane is equipped with a stand-alone scoring station that lists the bowlers' names and a graphic representation of their scores.
- The pinsetter interface communicates to the scoring station the pins that are left standing after a bowler has completed a throw.

- The control desk operator has the ability to monitor the scores of any active lane. A configurable display option will allow the operator to view the score of an individual scoring station or multiple scoring stations.
  This display feature is not seen on the window panel and hence brings down the fidelity to design.

Code Smells Table

Can be found at the following link:
https://docs.google.com/document/d/1u6xlFiouUHdK0KhxCxtss_RoyURha4Rz4ItPwKqVa-g/edit

Refactored Design

1. Balance Among Competing Criteria

a. Low Coupling

Tightly coupled systems tend to exhibit the following characteristics:

- A change in a class usually forces a ripple effect of changes in other classes.
- Require more effort and/or time due to the increased dependency.
- Might be harder to reuse a class because dependent classes must be included.

Hence, the coupling was reduced as much as possible.
Code Example:

- Transferred the end-of-game code in Lane to LaneStatusView, decoupling the Model from the GUI.

b. High Cohesion

High cohesion tends to be preferable, because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability. In contrast, low cohesion is associated with undesirable traits such as being difficult to maintain, test, reuse, or even understand.
Code Example:

- Replaced the extended Thread call in Lane to an implemented Runnable. Then in the constructor I created a new thread object, passing itself (as a Runnable object) as a parameter), and then started it. This effectively allows for us to extend the Observable class in Lane.

c. Separation Of Concerns
Code Example:

- Moved the drive.java file outside of the ViewControl and Model packages and added it to the main package. Hence the code is split into three main packages which address 3 different issues.

d. Information Hiding
   Code Example: Effective use of classes.

e. Law Of Demeter
   States that,
   - Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
   - Each unit should only talk to its friends; don't talk to strangers.
   - Only talk to your immediate friends.

   Code Example:
   Seperation of the Lane and Observer functionality classes.
   Removed the implements LaneObserver class in the GUI elements, replaced with implements Observer. Changed the logic for adding an Observer to Lane's list of observers to be in line with the java implementation in the appropriate GUI classes.

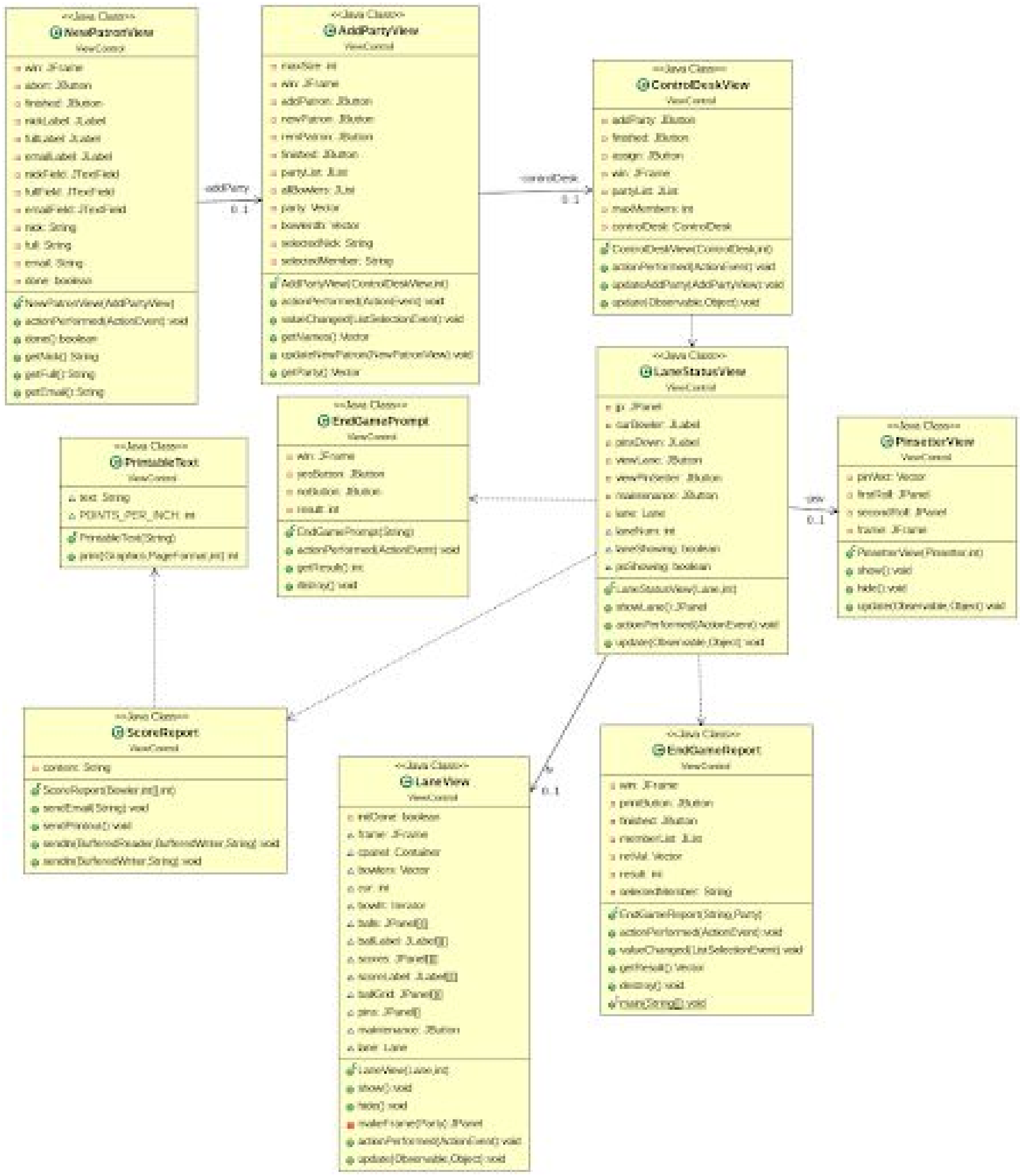
Overview Of Refactored Design

While refactoring code, the key question is the benefit gained versus the effort. If refactoring a complex set of code is deemed too time consuming, it might be a better option to rewrite the code.

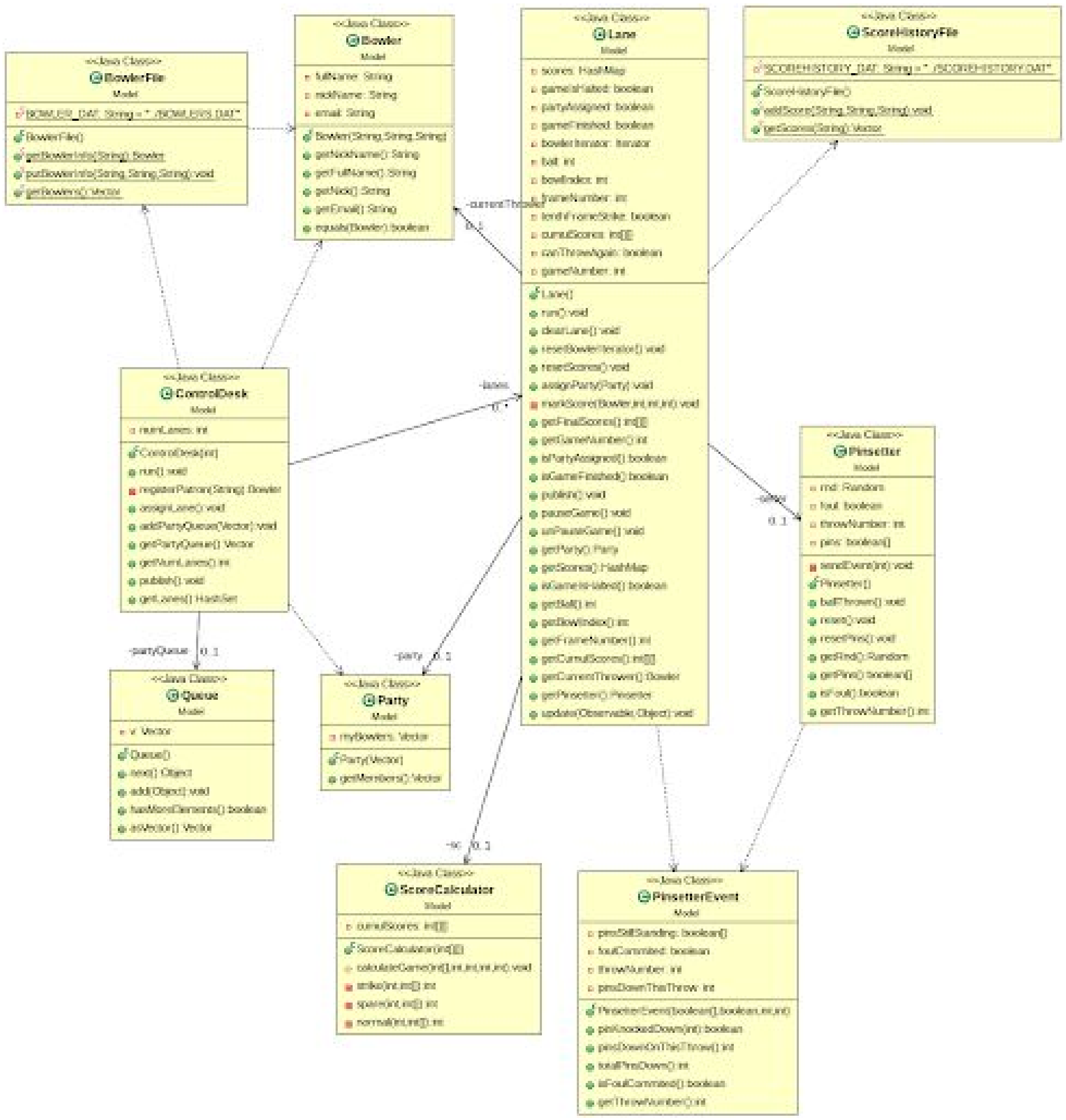
UML Class Diagrams For Refactored Design

Interfaces, inheritance, generalization, association, aggregation, composition, cardinality and role indicators are indicated using appropriate arrows.

View Control Module



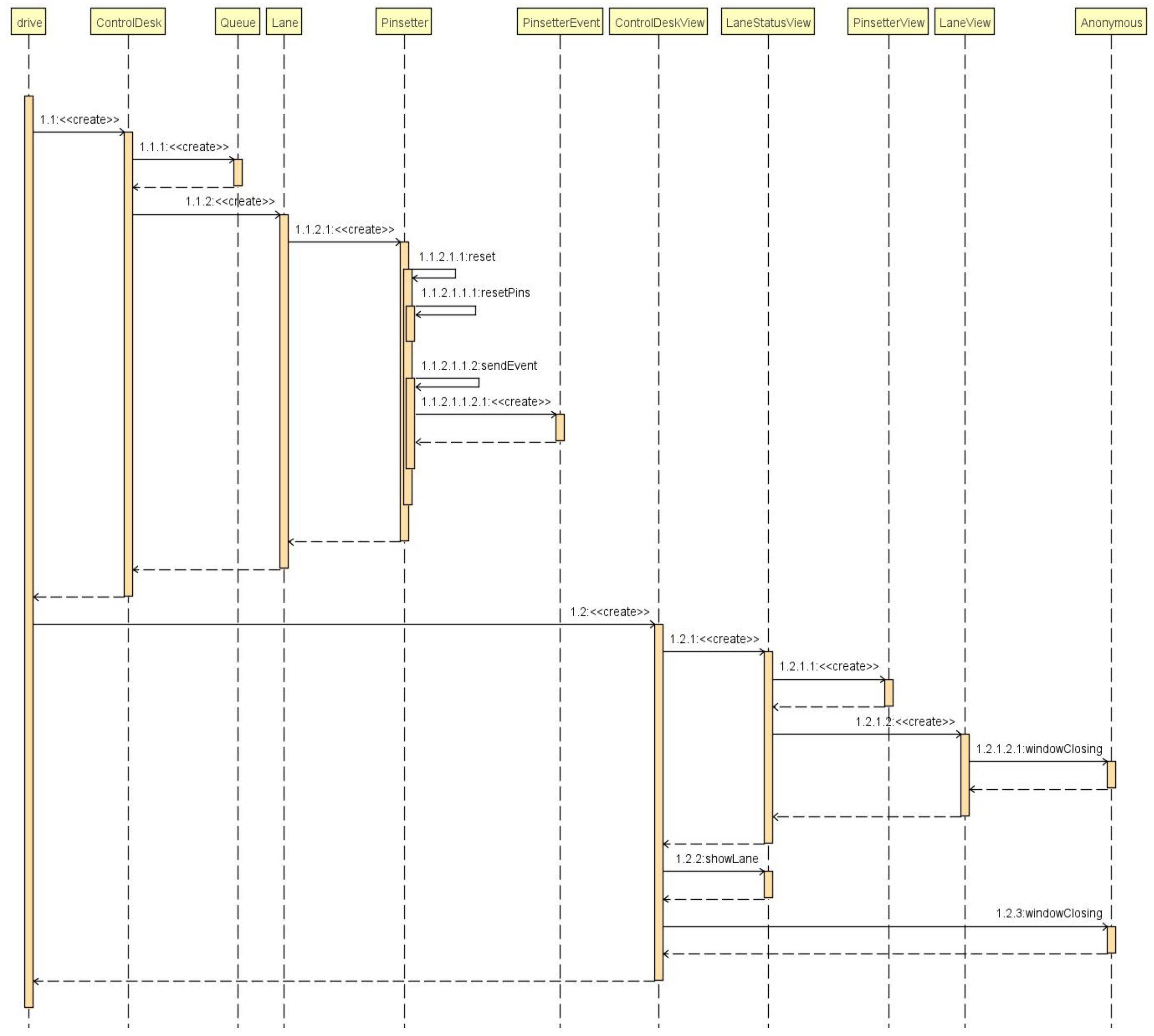
Models Module



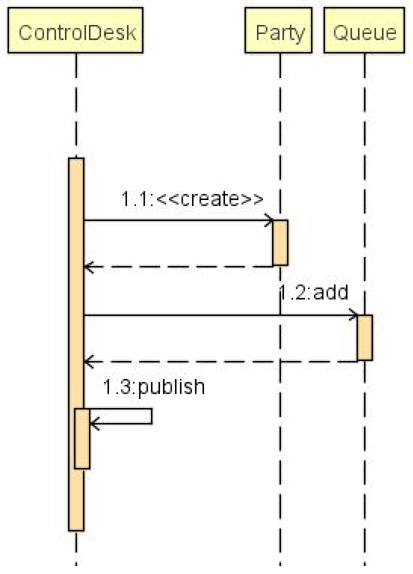
UML Sequence Diagrams For Refactored Design

Static and dynamic characteristics of the program

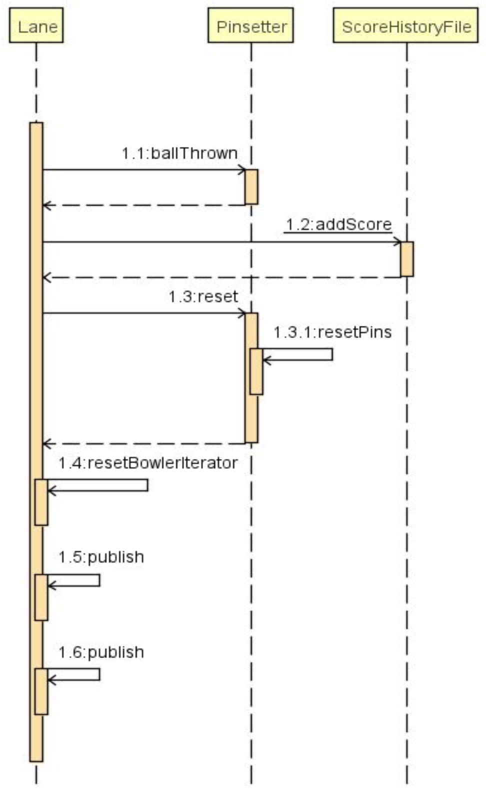
Drive: (i.e.from the class drive in file drive.java)



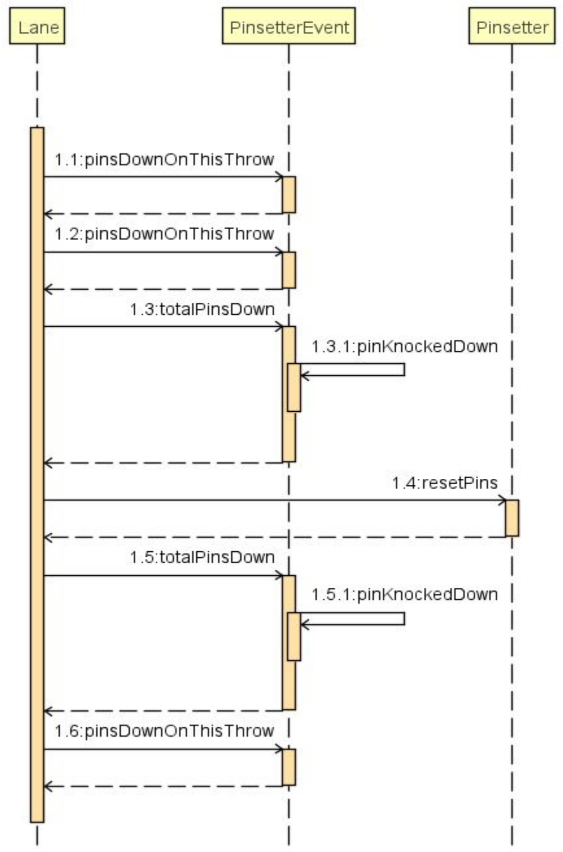
Add Party to Queue:



Run a Lane:



Update a Lane:



Class Responsibility Table (Refactored Design)

New Classes	*Responsibilities*
ScoreCalculated	To universally store the calculated score
NewPatronView	To get the view of different windows in the GUI
Observer	A general observer class that can be called upon to observe Lane and other entities

Design Patterns Used

1. Adapter Pattern

It is often used to make existing classes work with others without modifying their source code by allowing the interface of an existing class to be used as another interface.
Code example:
This was done in the ScoreCalculator class where all of the getScore functionality originally belonging to main was migrated into that class.

2. Proxy Pattern

Proxy is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.
Code example:
This was done in within Lane so that LaneStatusView is able to generate the end-of-game routine originally found in Lane and that has a

3. Singleton Pattern

The singleton pattern is a software design pattern that restricts the instantiation of a class to one "single" instance.

Code example:
The drive class that acts as the main function of this game is instantiated only once in its entire lifetime.

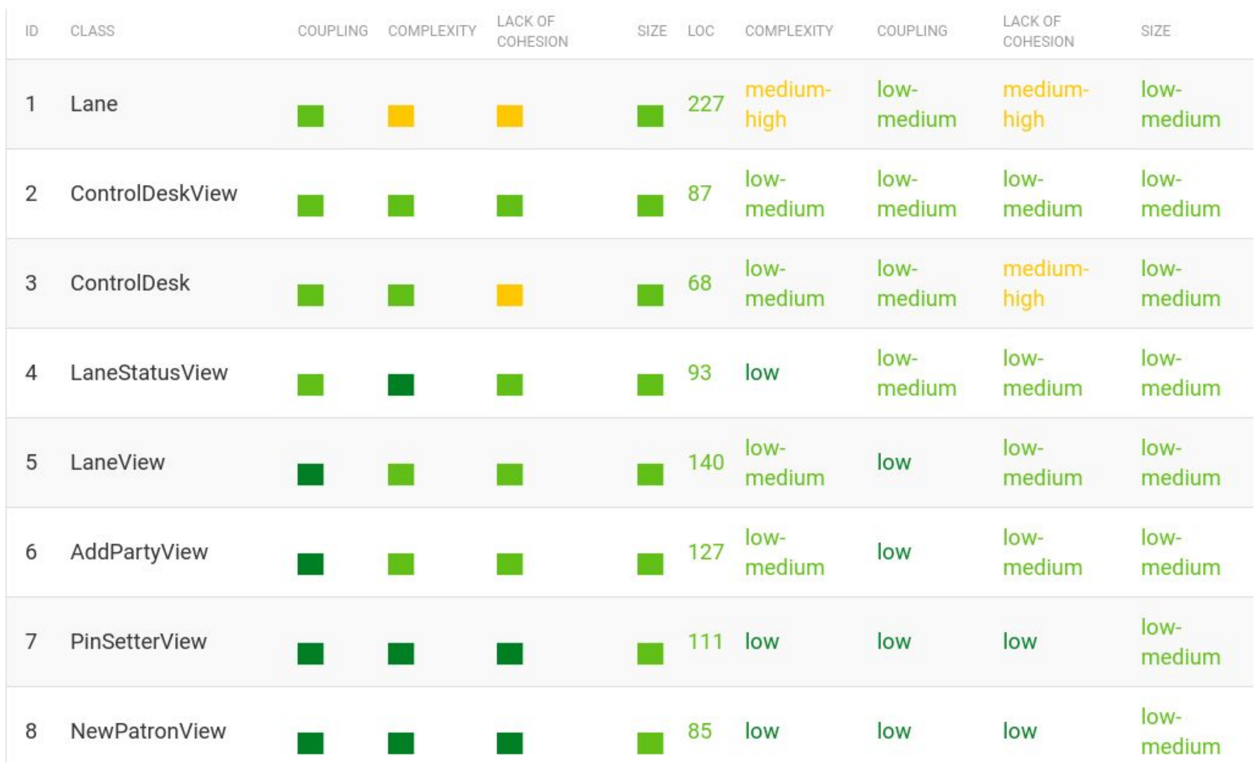
4. Controller Facade Pattern

The controller routes the request to your "model" and interacts with the View. The what is the same (route a simple interface to a more complex set behind the scenes). A controller in an application acts just like a Facade does in that it acts as a "gateway" to a more complex application for a specific use-case.
Code example:
This was done in the code base where 3 packages have been made and the code has been split into Model and ViewControl packages.

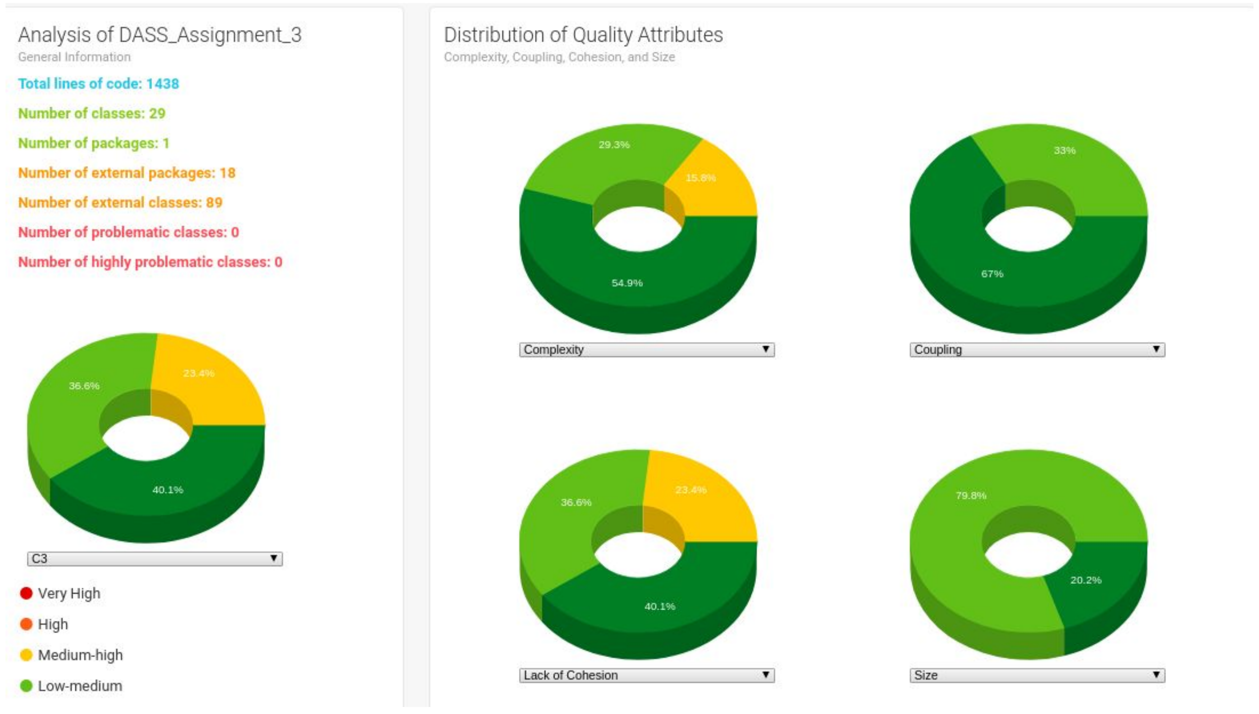
Metrics Analysis

1. Initial Measurements

The initial metrics as see on CodeMR on Intellij are as follows:



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


Analysis of DASS_Assignment_3

General Information

Total lines of code: 1438

Number of classes: 29

Number of packages: 1

Number of external packages: 18

Number of external classes: 89

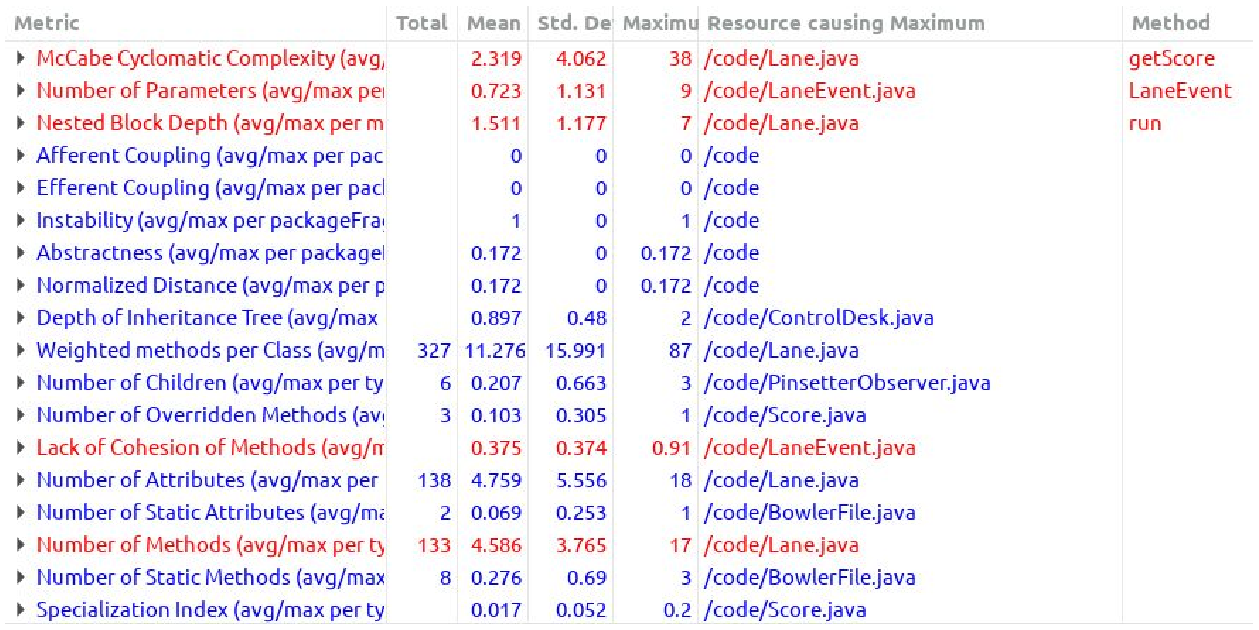
Number of problematic classes: 0

Number of highly problematic classes: 0

Distribution of Quality Attributes

Complexity, Coupling, Cohesion, and Size

The initial metrics on eclipse using the metrics plugin is as follows:



Metric	Total	Mean	Std. De	Maximu	Resource causing Maximum	Method
McCabe Cyclomatic Complexity (avg,		2.319	4.062	38	/code/Lane.java	getScore
Number of Parameters (avg/max pei		0.723	1.131	9	/code/LaneEvent.java	LaneEvent
Nested Block Depth (avg/max per m		1.511	1.177	7	/code/Lane.java	run
Afferent Coupling (avg/max per pac		0	0	0	/code	
Efferent Coupling (avg/max per pacl		0	0	0	/code	
Instability (avg/max per packageFra		1	0	1	/code	
Abstractness (avg/max per packagel		0.172	0	0.172	/code	
Normalized Distance (avg/max per p		0.172	0	0.172	/code	
Depth of Inheritance Tree (avg/max		0.897	0.48	2	/code/ControlDesk.java	
Weighted methods per Class (avg/m	327	11.276	15.991	87	/code/Lane.java	
Number of Children (avg/max per ty	6	0.207	0.663	3	/code/PinsetterObserver.java	
Number of Overridden Methods (av	3	0.103	0.305	1	/code/Score.java	
Lack of Cohesion of Methods (avg/n		0.375	0.374	0.91	/code/LaneEvent.java	
Number of Attributes (avg/max per	138	4.759	5.556	18	/code/Lane.java	
Number of Static Attributes (avg/ma	2	0.069	0.253	1	/code/BowlerFile.java	
Number of Methods (avg/max per ty	133	4.586	3.765	17	/code/Lane.java	
Number of Static Methods (avg/max	8	0.276	0.69	3	/code/BowlerFile.java	
Specialization Index (avg/max per ty		0.017	0.052	0.2	/code/Score.java	

2. Desired Measurements

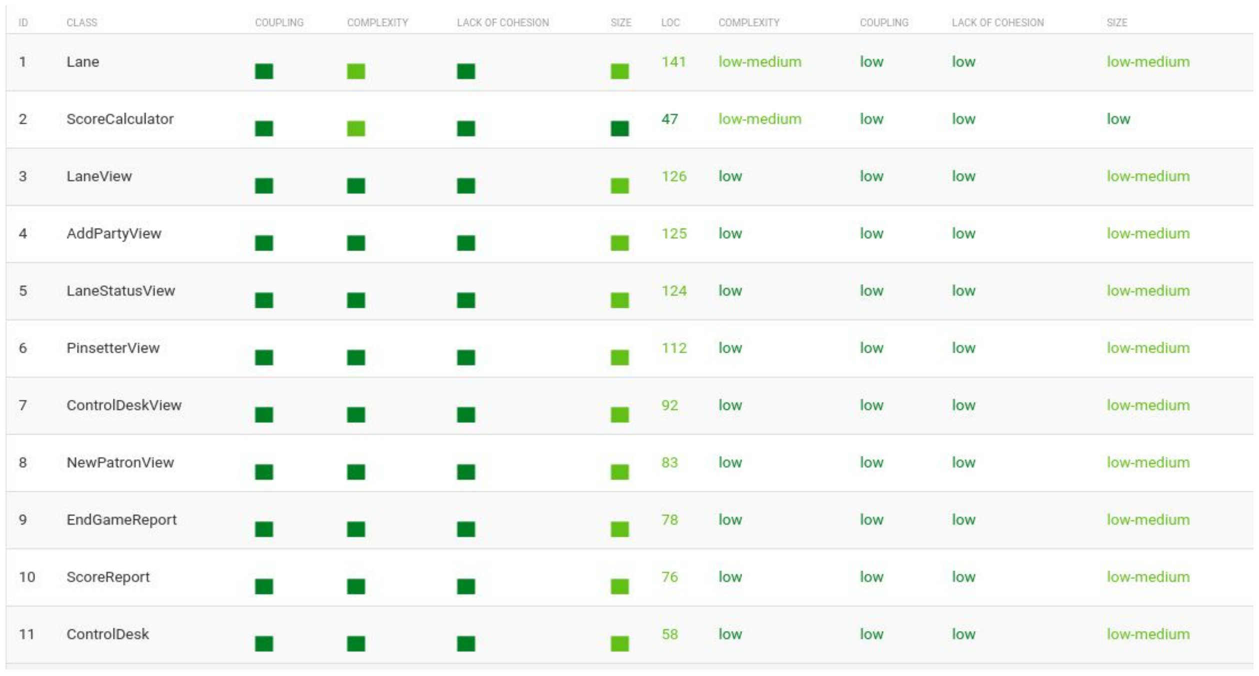
From our analysis of the metrics it was evident that we need to reduce the complexity, coupling and lack of cohesion in the files with large number of lines of code and in general large classes.

We applied several design patterns and removed code smells as documented above in order to achieve the following final results.

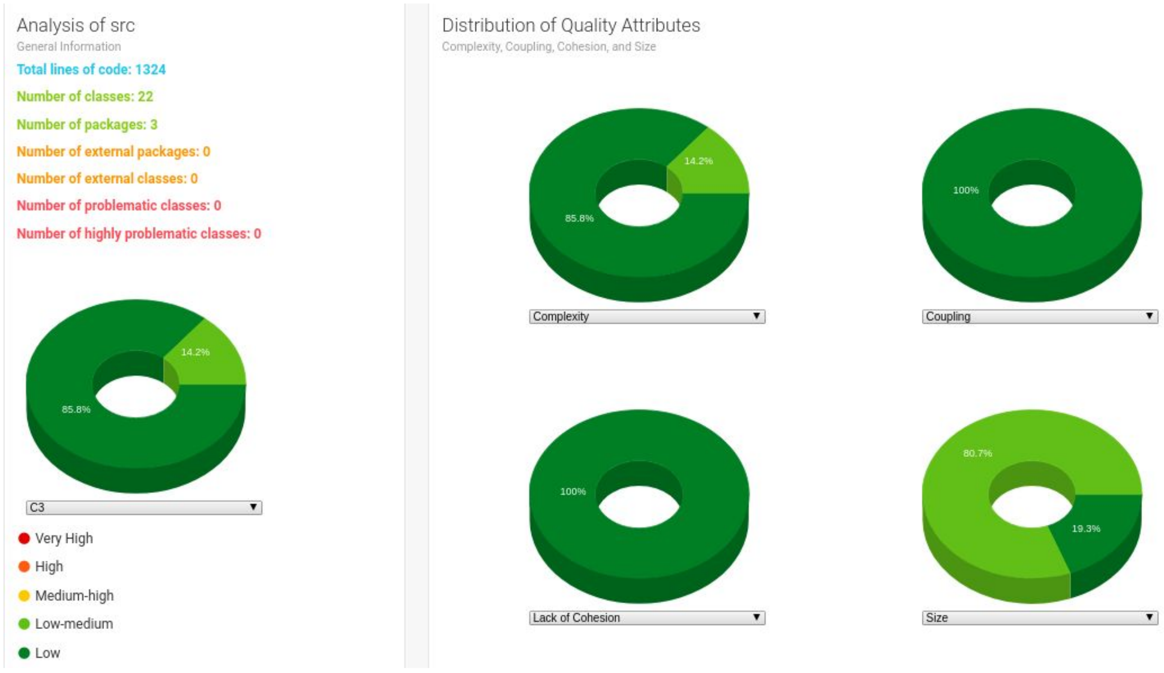
From the above analysis it was clear that many metrics were out of range. I refactored the classes with large numbers of lines of code and brought the numbers of classes down to 22.

3. Final Measurements And Cause

As can be seen in the final measurements, all greens have been obtained for all the files. Here, the coupling and lack of cohesion in the files with large number of lines of code and in general large classes has been reduced to low in every file.



ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	Lane	🟩	🟩	🟩	🟩	141	low-medium	low	low	low-medium
2	ScoreCalculator	🟩	🟩	🟩	🟩	47	low-medium	low	low	low
3	LaneView	🟩	🟩	🟩	🟩	126	low	low	low	low-medium
4	AddPartyView	🟩	🟩	🟩	🟩	125	low	low	low	low-medium
5	LaneStatusView	🟩	🟩	🟩	🟩	124	low	low	low	low-medium
6	PinsetterView	🟩	🟩	🟩	🟩	112	low	low	low	low-medium
7	ControlDeskView	🟩	🟩	🟩	🟩	92	low	low	low	low-medium
8	NewPatronView	🟩	🟩	🟩	🟩	83	low	low	low	low-medium
9	EndGameReport	🟩	🟩	🟩	🟩	78	low	low	low	low-medium
10	ScoreReport	🟩	🟩	🟩	🟩	76	low	low	low	low-medium
11	ControlDesk	🟩	🟩	🟩	🟩	58	low	low	low	low-medium


Analysis of src

General Information

Total lines of code: 1324

Number of classes: 22

Number of packages: 3

Number of external packages: 0

Number of external classes: 0

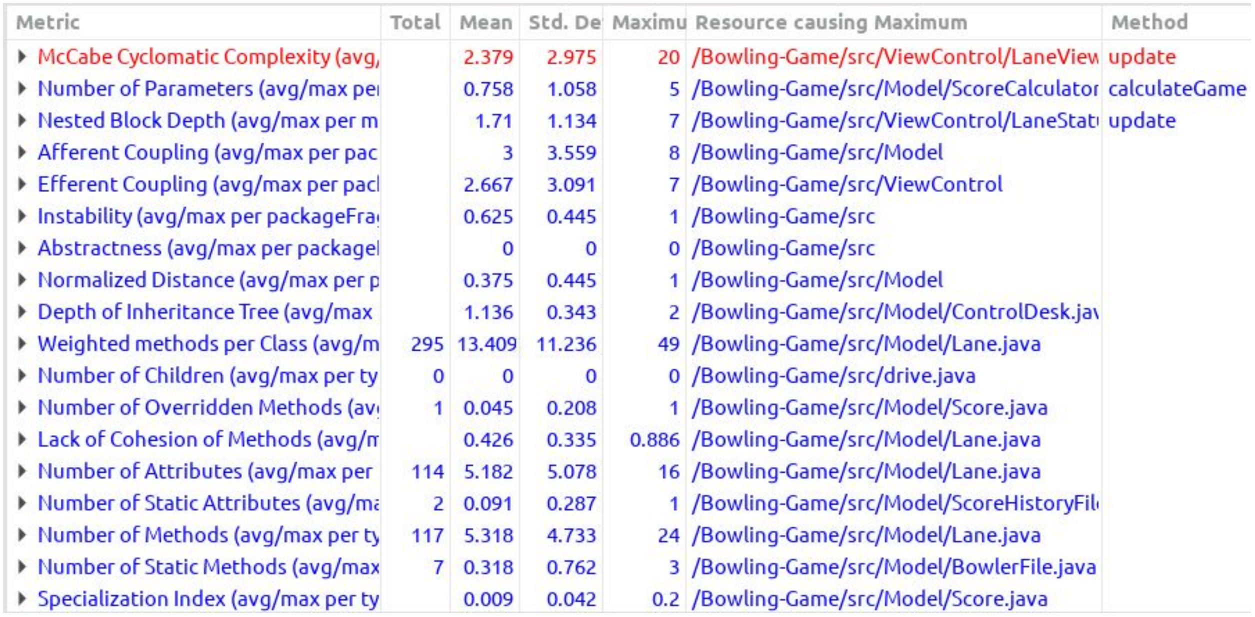
Number of problematic classes: 0

Number of highly problematic classes: 0

Distribution of Quality Attributes

Complexity, Coupling, Cohesion, and Size

On eclipse,the metrics are as follows:



Metric	Total	Mean	Std. De	Maximu	Resource causing Maximum	Method
McCabe Cyclomatic Complexity (avg,		2.379	2.975	20	/Bowling-Game/src/ViewControl/LaneView	update
Number of Parameters (avg/max pe		0.758	1.058	5	/Bowling-Game/src/Model/ScoreCalculator	calculateGame
Nested Block Depth (avg/max per m		1.71	1.134	7	/Bowling-Game/src/ViewControl/LaneStat	update
Afferent Coupling (avg/max per pac		3	3.559	8	/Bowling-Game/src/Model	
Efferent Coupling (avg/max per pac		2.667	3.091	7	/Bowling-Game/src/ViewControl	
Instability (avg/max per packageFra		0.625	0.445	1	/Bowling-Game/src	
Abstractness (avg/max per package		0	0	0	/Bowling-Game/src	
Normalized Distance (avg/max per p		0.375	0.445	1	/Bowling-Game/src/Model	
Depth of Inheritance Tree (avg/max		1.136	0.343	2	/Bowling-Game/src/Model/ControlDesk.ja	
Weighted methods per Class (avg/m	295	13.409	11.236	49	/Bowling-Game/src/Model/Lane.java	
Number of Children (avg/max per ty	0	0	0	0	/Bowling-Game/src/drive.java	
Number of Overridden Methods (av	1	0.045	0.208	1	/Bowling-Game/src/Model/Score.java	
Lack of Cohesion of Methods (avg/n		0.426	0.335	0.886	/Bowling-Game/src/Model/Lane.java	
Number of Attributes (avg/max per	114	5.182	5.078	16	/Bowling-Game/src/Model/Lane.java	
Number of Static Attributes (avg/m	2	0.091	0.287	1	/Bowling-Game/src/Model/ScoreHistoryFil	
Number of Methods (avg/max per ty	117	5.318	4.733	24	/Bowling-Game/src/Model/Lane.java	
Number of Static Methods (avg/max	7	0.318	0.762	3	/Bowling-Game/src/Model/BowlerFile.java	
Specialization Index (avg/max per ty		0.009	0.042	0.2	/Bowling-Game/src/Model/Score.java	

It can be seen that the code base has significantly improved in terms of code quality due to the refactoring. This has been achieved by creating a balance among competing criteria such as low coupling and high cohesion so as to develop efficient classes with no dead or duplicate code.