12th Feb' 2021

Bowling Alley

Refactoring Code

Team Members

Serial Number	Name	Roll Number	Batch	Git handle
1	Aditya Yadavalli	20171201	UG 2k17	AdityaYadavalli1
2	Anurag Sahu	2018121004	UG 2k17	AnuragSahu
3	Arnav kapoor	20171040	UG 2k17	arnavkapoor
4	Vivek Chandela	20171195	UG 2k17	vikilleaks

Time Spent and Role Played

1. Aditya Yadavalli - 32 to 34 Hrs. Took initiative and found out about all the possible plug-ins and libraries the team can use to make it easier for the team. Removed various code-violations in NewPatron, AddParty, Pinsetter, EndGameReport, EndGamePrompt, Score and Bowler.

2. Anurag Sahu - 10 Hrs Helped in the final report creation and ideas about different aspects of code.

3. Arnav Kapoor - 30 to 35 Hrs. Built a strong understanding of the underlying code to initiate initial discussion about the design choices and flaws. Focused on the largest Lane classes and handled multiple aspects of it primarily from cyclomatic complexity and coupling. Also performed minor cleanups in changes in lane file using PMD and removing legacy functions in dependent files.

4. Vivek Chandela - 10 Hrs Helped in the final report creation and providing resources.

Overview Section

The original codebase is a java application that is intended to automate all the Lucky Strikes Bowling Center's (LBSC) bowling establishments across the country. This application would also help the establishments in keeping track of bowlers that visited them and their previous scores (history). The components of the application are as follows:

1. A collection of lanes which can be monitored through a control desk.
2. Parties of bowlers can be assigned to the lane.
3. Adding new custom players.

4. The status of the current pins can be viewed, and the running scores in each lane is provided.

This original source code requires major refactoring as it contains design and code depicting poor practices.

Our objective was to improve the designs and code of this application. To get a quantitative aspect to this improvement , multiple ‘Metrics’ were gathered for the project to capture the changes.

Tools[1] and Libraries Used

1. PMD: This tool helped us show the possible code violations in the given project. Although it showed these, we didn’t blindly go about fixing everything. We fixed those that we thought we ought to fix to keep the code clean. It showed 1354 code violations at the beginning. After refactoring, we brought it down to 1118. Most of the code violations were design and best code practices violations.
2. SequenceDiagram: This tool/plug-in helped us generate the Sequence Diagram of the project for the report (can see below).
3. Log4J: This library helped us use logging instead of directing printing using System.out/System.err. In general, it is bad practice to use System.out in production level systems because one cannot turn off or change log levels i.e info, warning, error etc. This is mentioned in detail here
4. CodeMR: This tool/plug-in was used to generate the pie charts of various aspects of the code quality that are presented in this report.

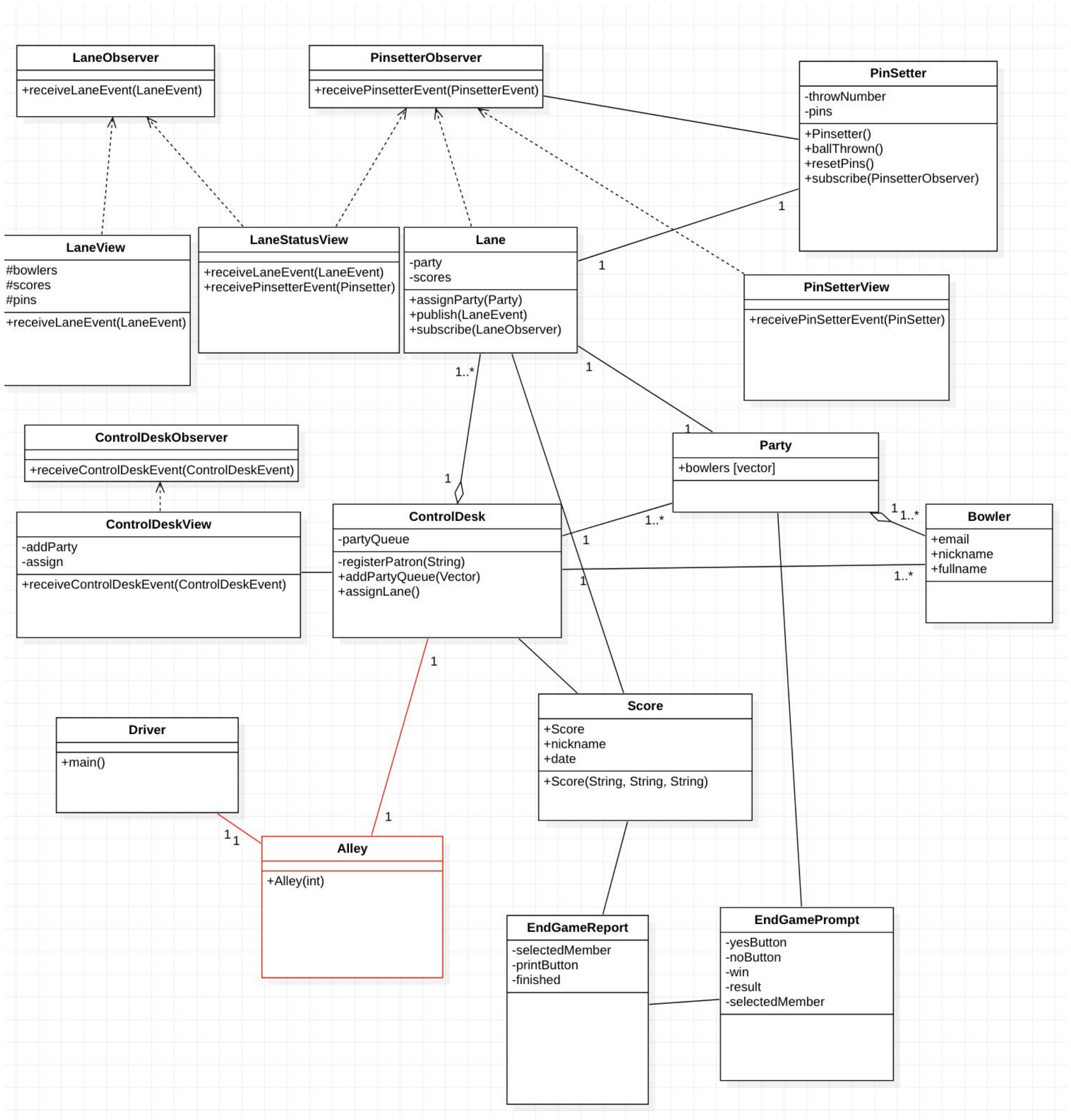
UML Diagrams

An Abstracted Out UML Diagram

[1] All the tools mentioned were plug-ins in IntelliJ (IDE). All the members used IntelliJ for refactoring.

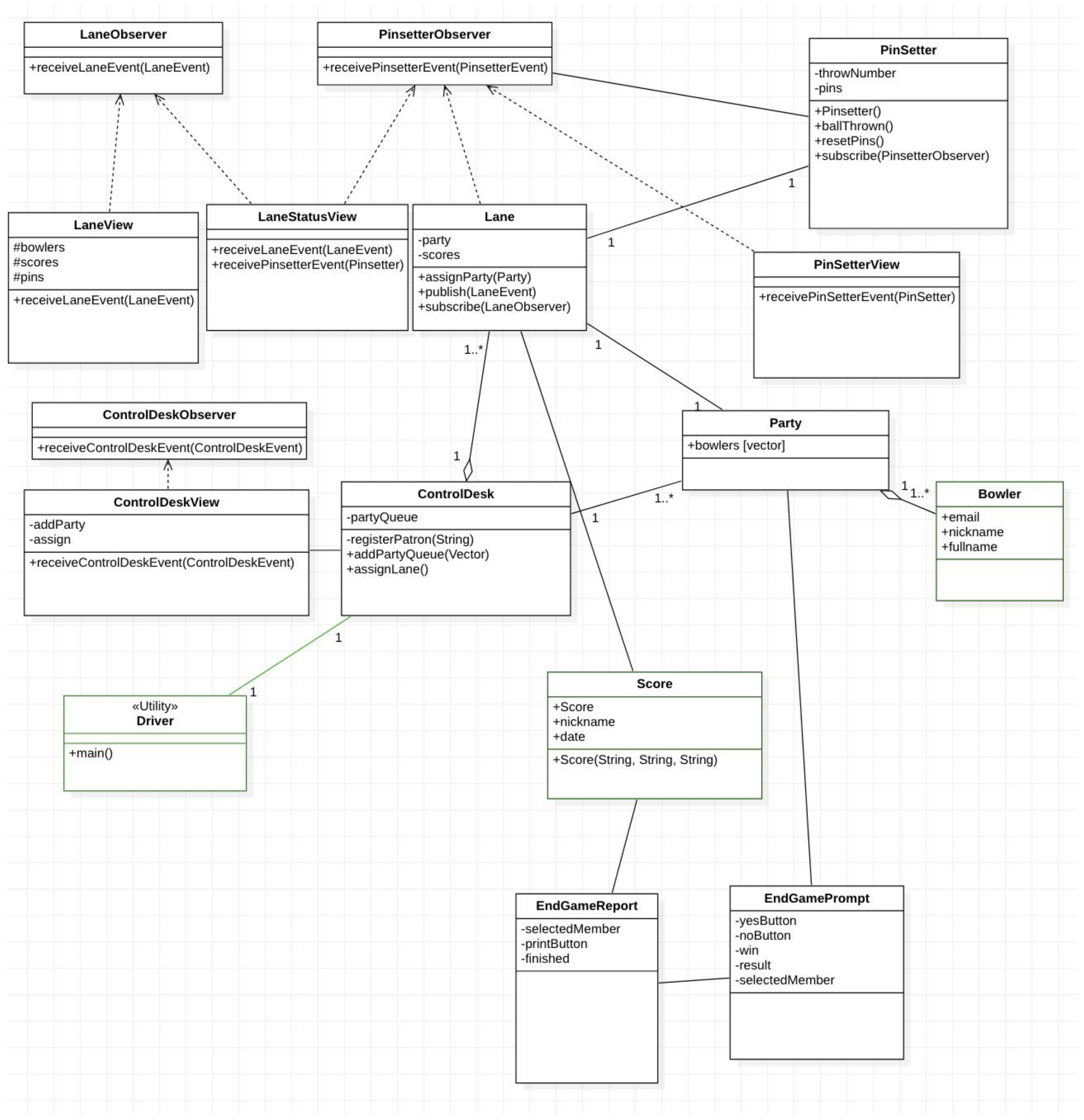
12th Feb’ 2021

Before



After

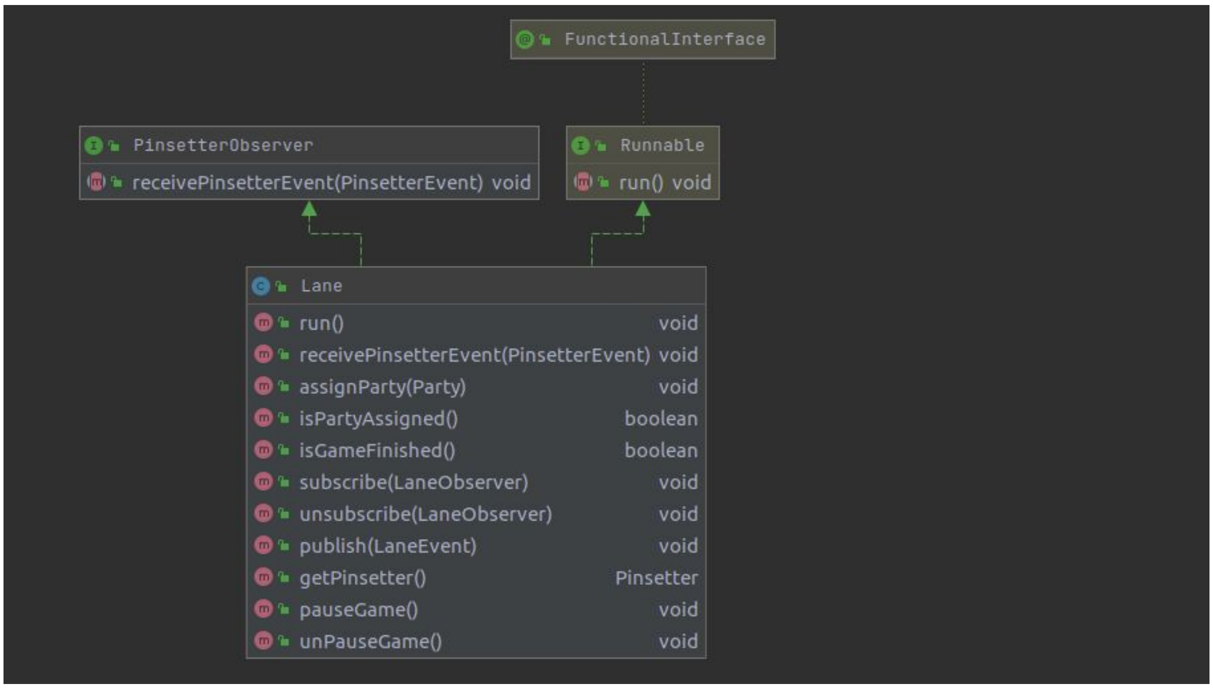
12th Feb' 2021



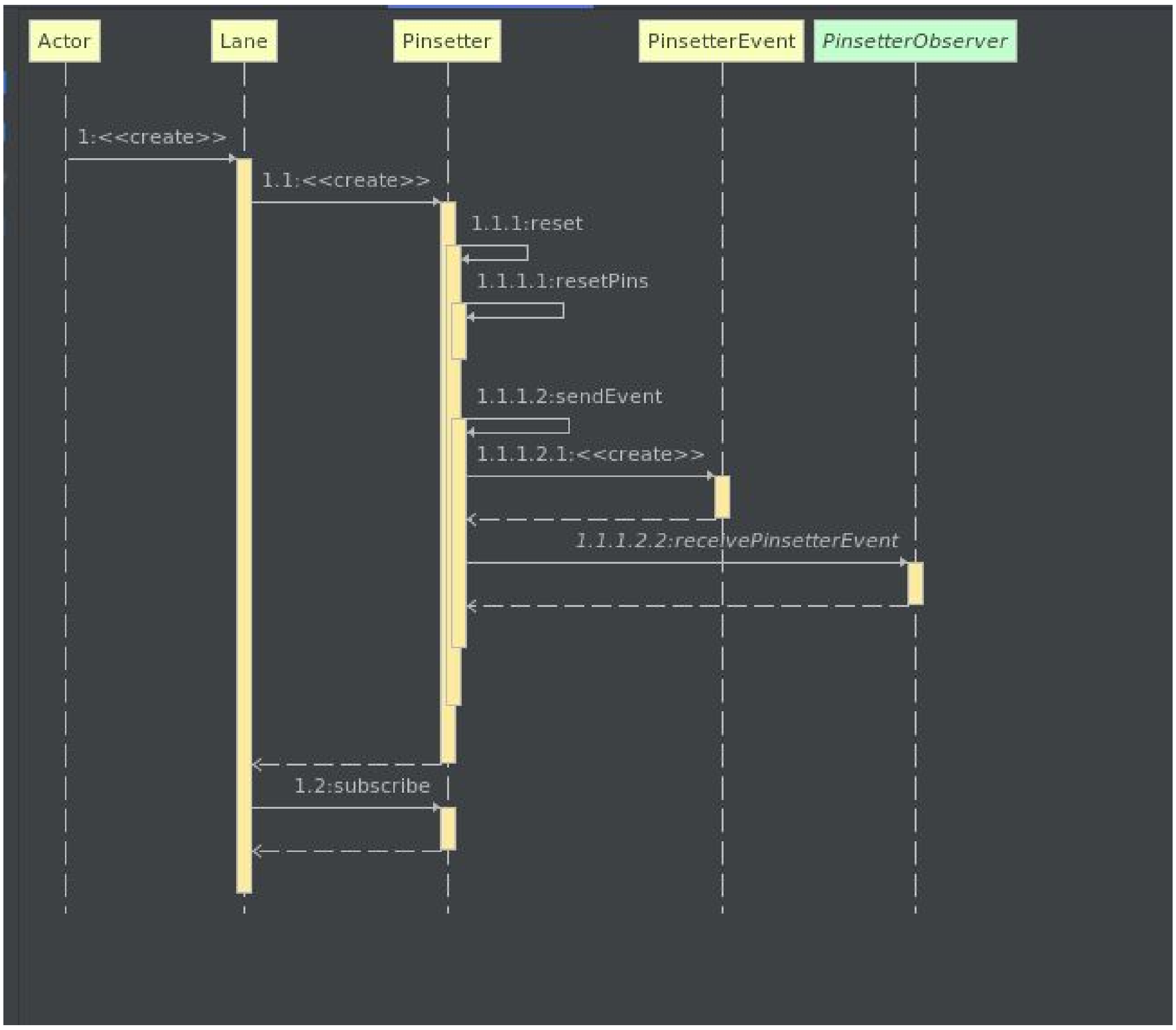
The classes related to the highlighted classes (in green) have been changed to utility classes (BowlerFile, ScoreHistory). Alley class was removed because it had very less utility - just passed around objects.

Focused Lane Class View

UML



Abstract Sequence Diagram (Before & After)



Original Class Details

SNo.	Class Name	Class Responsibility
1.	drive	This class initialized the game by setting the number of lanes, patrons and also instantiating some other important classes.
2.	AddPartyView	This class is responsible for displaying the part where a new party is added.
3.	ControlDesk	This class is responsible for initializing the number of lanes, taking input for number of lanes, assignment of lanes, creating party and name them, also along with that Control Desk maintains the wait queue.
4.	Bowler	This class is kind of a data class where it stores the info about the bowler like full name, email, person’s nickname also provides access to this info by get methods.
5.	ControlDeskView	This class is responsible for displaying the Control desk view and also showing the controls and inputs.
6.	EndGamePrompt	This class displays the GUI at the end of the game and also takes inputs for the prompt.
7.	BowlerFile	This file stores the information about the bowler. The class contains methods like getBowlerInfo, putBolwerInfo.
8.	Alley	This class is responsible for simulating bowling.
9.	ControlDeskEvent	This class maintains the parties name and also the waiting queue.
10.	EndGameReport	At the end of the game this class is responsible for generating the report of the game.
11.	Lane	This class is responsible for instantiating the classes related to Lane like Pinsetter, Map,it also starts important variables like scores, gamenumber, current bowler.
12.	LaneEventInterface	It provides an interface to the Lane class.
13.	LaneStatusView	This class is responsible for displaying the status of the lane line number of pins down, also it takes inputs like maintenance, pinsetter, foul.
14.	LaneView	This class is responsible for displaying the lane, this class gets information from LaneStatusView.
15.	Pinsetter	This class is responsible for placements of the pins in the

		game.
16.	Score	This class is responsible for storing the nickname of the player, date and the score at that time.
17.	ScoreHistory	This class gives the history report of the scores.
18.	ScoreReport	This class is responsible for generating the report of the scores; it can print the report or also email the report.

Describing the original design

We had manually created a filtered down version of the entire design to get a better grasp of the major workflow (The corresponding ***simplified hand curated UML*** under the section abstracted out UML Design) and the way the important classes interact with each other.

The reduced UML paints an overall picture of the design -
The main driver class calls the alley intermediary to create a control desk class. The control desk class has multiple aspects to it , bowlers, lanes, parties and scores. Each of which is controlled in the control dask. The logic of the actual game resides in the lane class which has dependencies with party , pinsetter and scores.

The ancillary features include the end game prompt and report.

Some of the identified anti patterns in the code-base were as follows :-

- Lava flow[2]: Retaining undesirable (redundant or low-quality) code because removing it is too expensive or has unpredictable consequences.

  We saw loads of redundant code spewed out throughout the code base that we fixed.

- God object[3]: Concentrating too many functions in a single part of the design (class)

[2] "Lava Flow AntiPattern - SourceMaking." https://sourcemaking.com/antipatterns/lava-flow.
[3] "God object - Wikipedia." https://en.wikipedia.org/wiki/God_object.

The lane and control desk parts were the prime example of this we split out the functionality as and when needed.

- Coding by exception[4]: Coding by exception is an accidental complexity in a software system in which the program handles specific errors that arise with unique exceptions

  A lot of conditional statements had been put in to handle specific cases. In the lane class the score computation required separate handling for the last 3 turns however convoluted logic had been put up to handle these cases , with new code base probably being added for each exception that occurred.

- Copy and paste programming[5]: It is the production of highly repetitive computer programming code,

  A large number of code structures were copied across the code base with some of them being redundant

The original design did adhere to the development document to a large extent, was majorly bug free (scoring bug was there which didn't allow to calculate score accurately) and also used patterns (such as observer in control desk to notify that the game is over etc) in the right places.

However, the same codebase also had multiple aspects of poor-design choices which needed to be revamped upon. We cover the identified code smells and their refactoring and overall pattern redesign below.

Code Smells Refactored

Long Methods

Shorter methods are often easy to read, maintain and test. We had a few long methods that we split. Some such methods are AddPartyView() in AddPartyView.java, ControlDeskView() in ControlDeskView.java and PinsetterView() in PinsetterView().

Comments

A lot of files (almost all) had large comments (some amounting to 100 lines!) that we felt were totally unnecessary. They were mainly logging what has been refactored in which version of the refactoring cycle (version).

[4] "Coding by exception - Wikipedia." https://en.wikipedia.org/wiki/Coding_by_exception. Accessed 12 Feb. 2021.
[5] "Copy-and-paste programming - Wikipedia." https://en.wikipedia.org/wiki/Copy-and-paste_programming. Accessed 12 Feb. 2021.

Conditional Complexity

Large conditional blocks have been split. Some branches that we thought were not adding any new functionality to the application were removed. All the actionPerformed() methods in all the View files. As we show later, these kinds of changes reduced the cyclomatic complexity significantly.

Dead Code

We found many import statements and variables that were defined but never used. We mercilessly cut those out. Few examples are: import java.awt.event.* in PinsetterView.java; import javax.swing.event.*, import java.util.*, import java.text.* in NewPatronView.java.

Speculative Generality

The alley class was added as a precaution for incorporating multiple bowling establishments. This comes under speculative generality as the use of it right now is minimalistic.

Middle Man

If a class performs only one action, delegating work to another class, why does it exist at all?

The alley class in essence is a wrapper between the main drive function and the control desk and thus for our utility is something that can be removed safely.

Large Class

The lane and control desk class with its multitude of methods was a huge class that needed to be split to ensure easy readability , this was done both by breaking into smaller methods and reducing the inherent number of lines.

We also saw the need for a pattern such as Builder to incorporate various ‘View’ files. These files essentially were doing the same thing for their main class (AddPartyView for AddParty, PinsetterView for Pinsetter) but underneath, the algorithm is, as one would expect, very different.

Design Tradeoffs

Loose Coupling

One of the major design tradeoffs we felt was essential to have aspects of loose coupling in the code , We modified the code to use generic interfaces as opposed to specific instances of the same. This can be seen in the usage of map as opposed to hashmap and list instead of vector. This design choice was made because we knew that the types of variables used may need to be modified and enhanced in the future to encompass multiple aspects.

Metric Changes

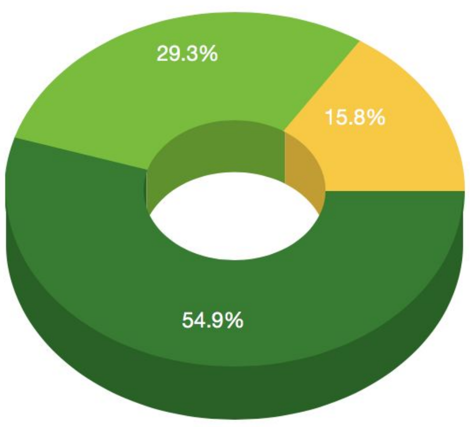
We show how changes in code we made affected it in terms of metrics. However, it should be mentioned that our goal was never to improve the metrics. Infact, the tool we used to show the graphs below, CodeMR, was installed after all the refactoring was done. Our goal was to improve the general readability of the code, follow OOD (Object Oriented Design) and remove any code that was not used to make the application.

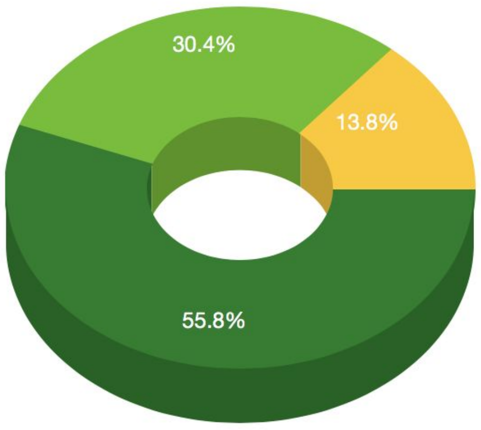
Complexity

Complexity essentially describes how difficult it is to understand the code and the various interactions that the entities in the software. The metrics to capture this are described below:

1. WMC (Weighted Method Complexity): This is equal to the number of all the methods if the weight is taken as 1 for each method. However, depending on the class and the method itself the weight of this class changes. The basic idea is that if the number of methods are high in a class then the class is domain specific. Therefore, it would not be reusable.
2. DIT (Depth of Inheritance Tree): The leaf class gets increasingly hard to test and maintain if the depth of the inheritance tree goes beyond 6.
3. RFC (Response for a class): The number of the methods that can potentially be invoked in response to a public message received by an object of a particular class. If the number of methods that can be invoked at a class is high, then the class is considered more complex. This means it can be highly coupled to other classes. Thus, it is a lot tougher to test and maintain such classes.
4. SI (Specialisation Index): The specialisation index measures the extent to which the subclasses override their ancestors. Such classes get increasingly domain specific and therefore tough to test.

Before/After



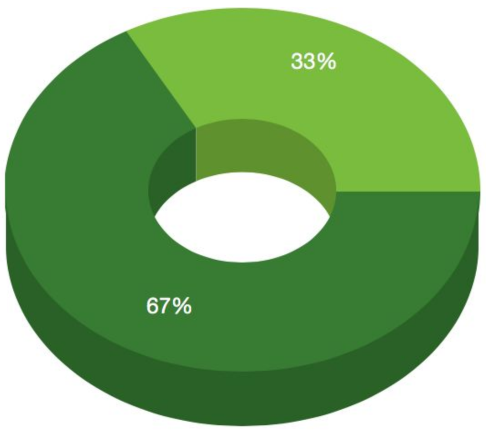


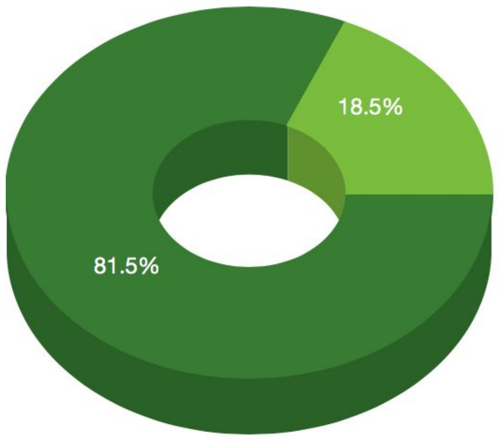
Coupling

The general rule is “higher the coupling, the more difficult it is to maintain, test or extend the code to add new features”. This is true because a change in one of the classes means a change in another. Additionally, if one were to use one of the classes, s/he is forced to use the other too because high coupling usually means high dependency between classes. This is captured as follows:

1. NOC (Number of Children): Number of direct subclasses of a class.
2. Access to Foreign Data: This is the number of classes whose attributes can be accessed by a class.
3. Degree: Number of associations a class has with others. This can be directly seen in the UML diagram.

Before/After



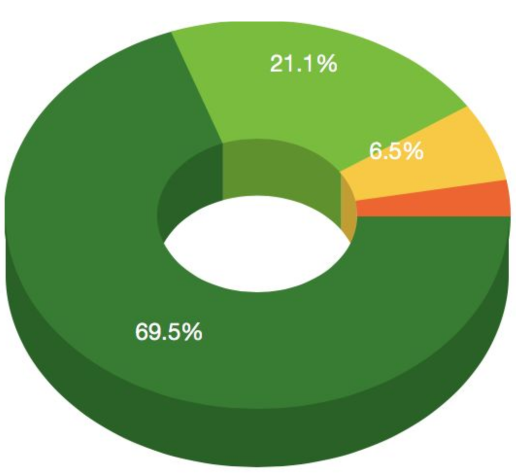


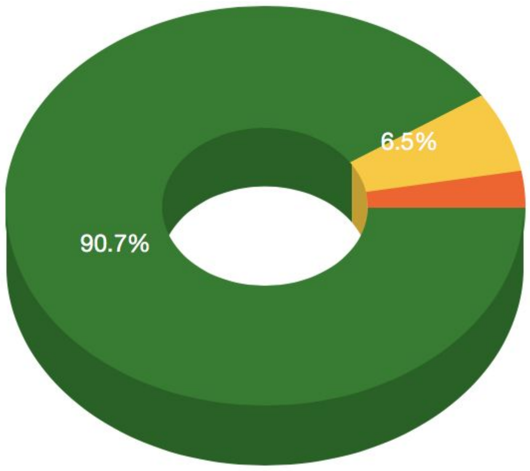
12th Feb’ 2021

Lack of Cohesion

A class should have a responsibility. It should do that and only that. If it does a ‘bunch’ of unrelated things then one should consider splitting them up. That way it's easier to maintain and test those classes and further even extend the current code to add new features as and when there is a requirement.

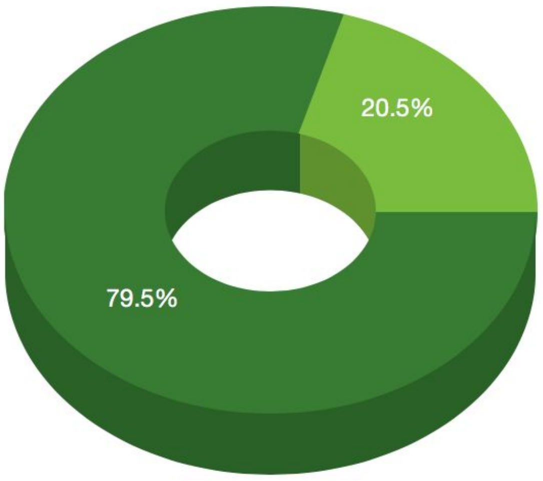
Before/After

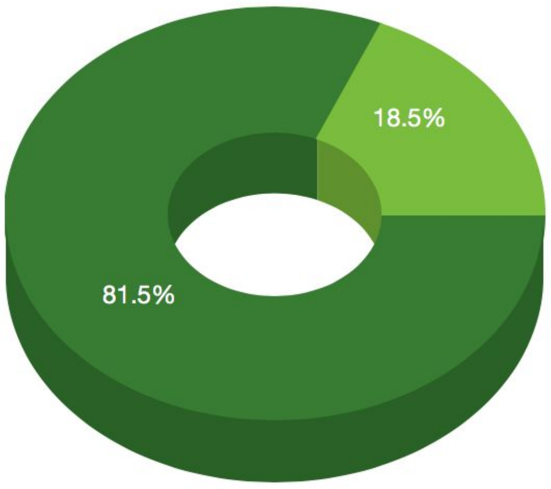




Specialisation Index

The definition is mentioned above. We specifically show the graphs to show the improvement we made on this front.





12th Feb’ 2021

Special Emphasis on Cyclomatic Complexity

Quite often it’s not the smallest or the fastest code that is needed in a system. A code that reads well and flows naturally without too many branches is sometimes the best. It’s a measure of the number of linearly-independent paths through a program module. A lot of the core logic and computation of the application resided in a single method getScore in the Lane class.

Keeping the earlier statement ***code is meant to be read*** , we focused on reducing the cognitive cyclomatic complexity (The metric is similar to Cyclomatic Complexity, but is intended to explicitly measure understandability, which can be quite different from testability)

We were able to reduce the CogC significantly from a value of 111 to 48
(csvs in the github repo (line 50 in metrics file)). This is a reduction of around **150%** , and makes the code base much more approachable and accessible.