DASS Assignment 3

Animesh Sinha; Avani Gupta; Gaurang Tandon;

April 9, 2020

1 Introduction

1.1 Short overview

Hey there, welcome to our Bowling Game project! We were hired specifically *not* to build a new system, but to **fix** an original implementation by some other team in terms of its **maintainability**, and implement only a few new features into it. We have toiled hard to deliver a codebase which satisfies many important software design principles.

We have also included all the three features, namely:

- Maximum multiplayer up to six players
- Pause and play the game, and resume a paused game from memory
- View statistical information about the best and worst players

It was interesting to note that our refactoring was effective enough that implementing all these features required small changes to the code at obvious locations.

1.2 Contributions

Each of us worked for almost 40-45 hours on the codebase. We had these roles:

- Gaurang: Complete backend refactor (splitting out Lane, reducing cyclomatic complexity, observer pattern)
- Animesh: Complete frontend refactor (the View Classes, and entirety of Widget)
- Avani: Building Documentation, miscellaneous fixes

2 Product overview

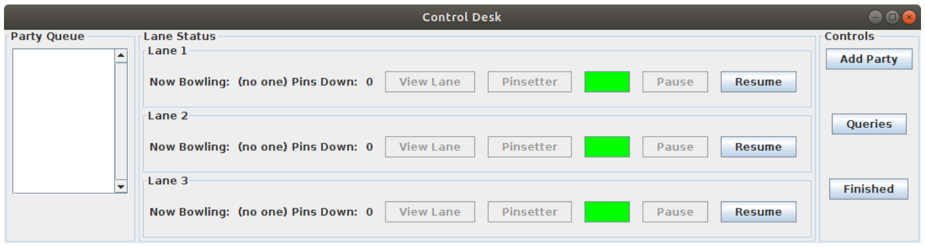
The aim of this product is to serve as a simulator for a Bowling Alley. We want to be able to simulate players coming in groups (”party”) to the alley and being assigned a lane to bowl on. If there’s no lane yet, then they wait in a queue. While bowling, they proceed frame by frame, being able to see pins falling on a pinsettter as well as see their detailed scorecard on a scoresheet.

Once the party is done bowling, they can play another game if they wish to. Otherwise, they can take a printout of their performance and leave the alley. There are three lanes in the alley, each allowing a party to have six players.

In the refactored product, we were also asked to implement additional features, expanding the scope of this simulator. Namely, we now allow a party to pause their game in between and leave the alley. Later, they can resume their unfinished game.

We now also support ad-hoc queries on the user data. Each player can view their best and worst score statistics, as well as their career top five scores.

2.1 Product screenshots

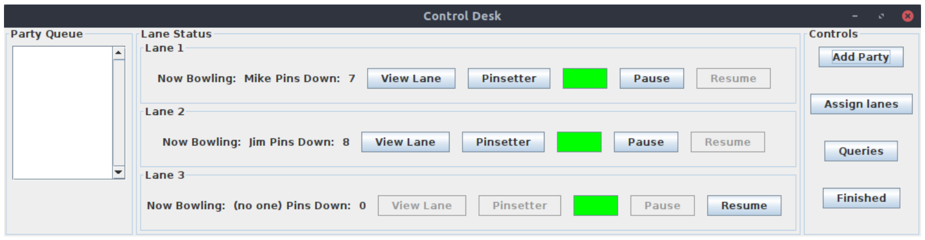


Now Bowling: (no one) Pins Down: 0

Now Bowling: (no one) Pins Down: 0

Now Bowling: (no one) Pins Down: 0

(a) The main control desk, where we can create parties, assign them to lanes, or ask for ad-hoc queries.



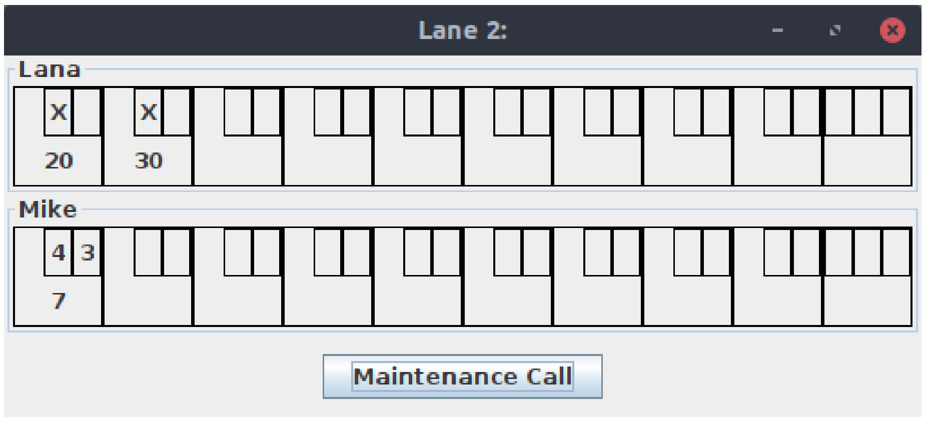
Now Bowling: Mike Pins Down: 7

Now Bowling: Jim Pins Down: 8

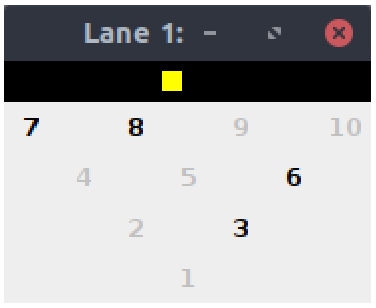
Now Bowling: (no one) Pins Down: 0

(b) When a party is playing on the first two lanes.

Figure 1: Control Desk

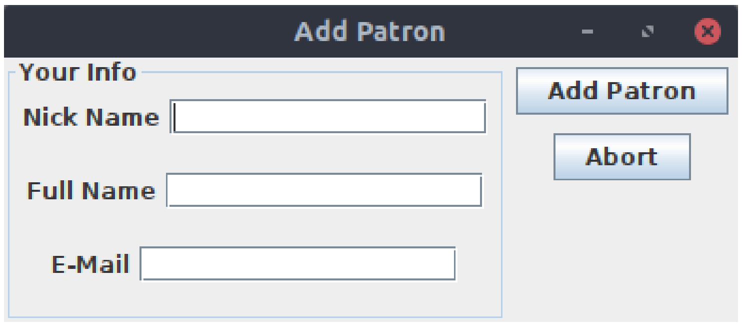


(a) Dynamically updated scoresheet of players in a party as they bowl.

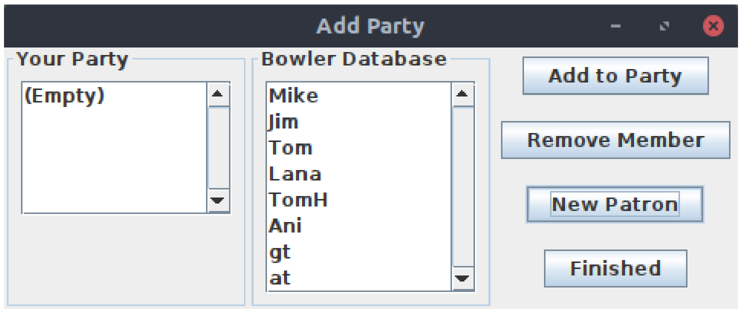


(b) Dynamically updated pinsetter, shows pins standing as dark black, and fallen as light grey

Figure 2: Dynamic lane view elements

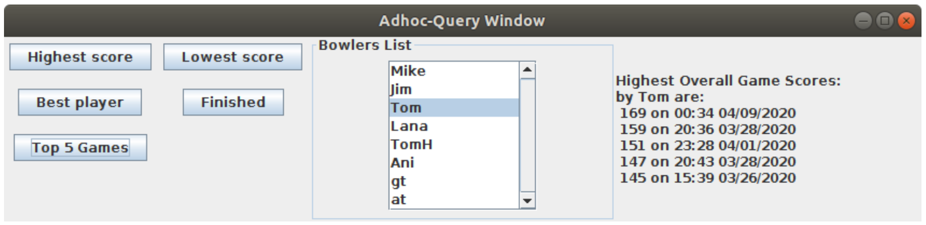


(a) Add patron modal



(b) Add party modal

Figure 3: The data update modals



```
Highest Overall Game Scores:
by Tom are:
 169 on 00:34 04/09/2020
 159 on 20:36 03/28/2020
 151 on 23:28 04/01/2020
 147 on 20:43 03/28/2020
 145 on 15:39 03/26/2020
```

Figure 4: Ad-hoc queries modal, dynamically fetches statistics from the storage

2.2 Our dedication to software engineering principles

Throughout our refactoring, we have consistently upholded major principles of software design. The first is obviously **DRY** - Don't Repeat Yourself. This is a common principle, wherein we make sure that the same functionality is not duplicated in more than one place. In our project, we have ensured that even the simplest of logics, like checking whether a strike occurred or not (`pinsDown == 10`) has been carefully placed in a dedicated method so as to not duplicate it across mutliple places.

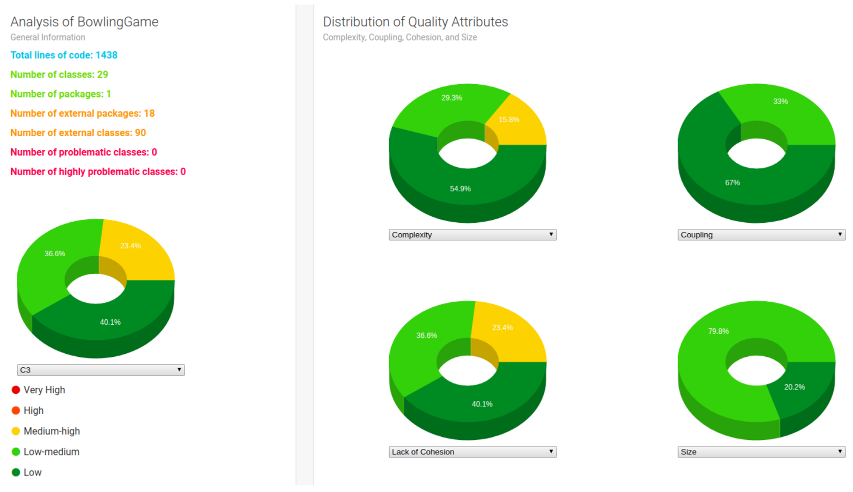
Another principle we stuck to was **Single Responsibility principle**: which states that every class should be responsible for exactly one purpose. Note that this was the hardest to sustain, since the original codebase had a God class (Lane.java) by itself.

Law of Demeter was another arena we conquered. We ensured that our classes are kept independent of each other, that connections between different classes are minimized (**coupling**), and that related classes are kept together (**cohesion**).

We have also extensively used **inheritance** wherever required to convey the semantic logic of our classes. For example, `BowlerInfo` acts as a base class for `Bowler`. The sole responsibility of BowlerInfo is to keep track of metadata of a Bowler, whereas the sole role of a Bowler is to be able to roll balls down a lane. We then implemented `ScorableBowler` which inherits from Bowler, and also adds the ability to keep track of scores across games. This chain of inheritance ensures that our classes have cohesive usage of attributes as well as reusability throughout.

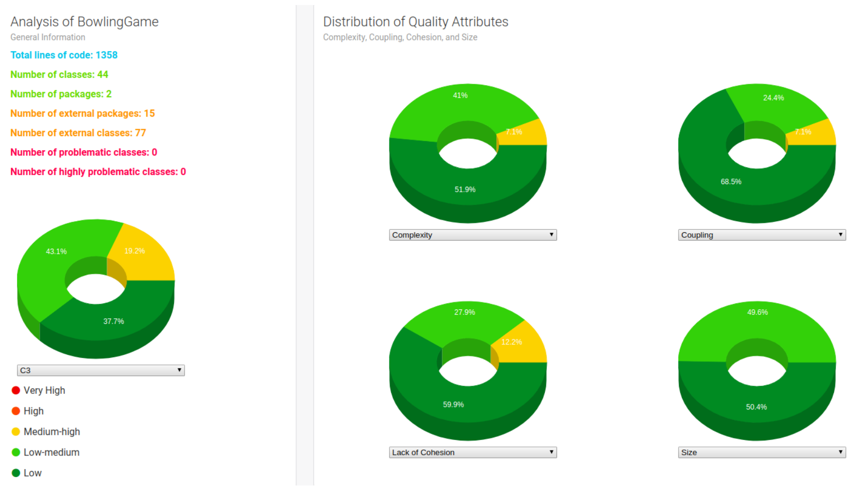
3 Code Metrics

3.1 Plots of CodeMR


Analysis of BowlingGame
General Information
Total lines of code: 1438
Number of classes: 29
Number of packages: 1
Number of external packages: 18
Number of external classes: 90
Number of problematic classes: 0
Number of highly problematic classes: 0

Distribution of Quality Attributes
Complexity, Coupling, Cohesion, and Size

(a) Before the Refactor


Analysis of BowlingGame
General Information
Total lines of code: 1358
Number of classes: 44
Number of packages: 2
Number of external packages: 15
Number of external classes: 77
Number of problematic classes: 0
Number of highly problematic classes: 0

Distribution of Quality Attributes
Complexity, Coupling, Cohesion, and Size

(b) After the Refactor

Figure 5: Code Metrics

3.2 Metric analysis

We investigated metrics in multiple ways: one through the CodeMR plugin for IntelliJ and the other through Metrics2 1.38 plugin for Eclipse.

3.3 Refactoring narrative

We present to you here the timeline of events that become the cornerstone of starting our refactoring:

- Immediately, after launching metrics analyzer on the codebase, we noticed the following key components being broken: McCabe cyclomatic complexity and Nested Block Depth, with both being in Lane.java.
- We open Lane.java and notice it's 700 lines mammoth shape with overly complex methods. We take a guess that it's a "God class" but we're not sure.
- To confirm our suspicion, we look at such metrics:
  - **Lack of cohesion**: it has the third highest lack of cohesion among all files.
  - **Number of attributes**: it has the most number of attributes (18!) in a class. To be honest, 18 is a very large number and an indication that Lane.java is trying to do much.
  - **Number of methods**: this was the killing blow. Lane.java has 17 methods, the most among all classes (the second place has only 11)
- We thus concluded that Lane.java is trying to do much, and is indeed a God class.
- In such a system, it is ideal to start by:
  - understanding all the functionality implemented in the God class
  - marking out several sets of cohesive functionality
  - extracting those sets out in several other classes
- We then chalked out the following cohesive sets:
  - Scoring functionality: `markScore, getScore, resetScores, isGameFinished, resetBowlerIterator`
  - Interacting with a pinsetter: `getPinsetter, receivePSEvent, run`
  - Controlling the external UI and Maintaining a observer model: `lanePublish, subscribe, unsubscribe, publish`
  - Actually running the game for a party: `run, assignParty, isPartyAssigned`
- These parts were then systematically split out into separate classes, namely, `ScorableParty, LaneWithPinsetter, Publisher, Lane`
- Once we had separate parts getting compiled, we were already ready to start ironing out complexity and redundancy issues. For quite a lot of time, we worked on these.

- Reducing a method for its cyclomatic complexity usually meant:
  - Understanding what the method is actually doing
  - Break it down into its core parts,
  - Extract those core parts into new methods
  - Reuse those new methods across the codebase
- For code duplication, one thing that really stood out was the extreme disregard for redundancy in the 'XView' classes, as each of them shared a series of crucial logic duplicated. For example, the logic for centering a window, or adding a button, and more. We quickly realized this and extracted out a general set of 'Widget.X' classes, whose main purposes is to provide a consistent UI throughtout the codebase. This is reused throughout the codebase, and helped us creating the UI for the newer features with minimal effort.
- Now that we had so many classes, we ran into issues of coupling. We had to be careful when fixing those since there wasn't a general fix-all medicine for this.
- Once all was done, in the final stages, we made sure that cohesion scores were as high as possible. For classes that had lower cohesion, we grouped their properties and further increased cohesion.
- That was pretty much all :)

3.4 Metric values analysis

We have collected and analyzed the following key metrics, showing the before and after change in them as well.

3.4.1 Lack of cohesion

Before: 0.375 mean with 0.374 std. **After:** 0.321 mean with 0.298 std.

This is the LCOM metric, calculated as a formula. Vales near 0 are good while near 1 are dangerous. This metric wasn't reduced significantly reduced, however, it was decently low before, so even the current scores are good.

3.4.2 Coupling

Before: After:

3.4.3 Nested block depth

Before: 1.511 mean with 1.177 stdev **After:** 1.423 mean with 0.659 stdev

Moreover, the largest offender Lane.java was brought down significantly. Top 3 before:

File	Mean	Std. Dev	Maximum
Lane.java	2.176	2.065	7
Lane.java	2.333	1.374	5
LaneView.java	1.6	1.2	4

and after:

File	Mean	Std. Dev	Maximum
Lane.java	1.5	0.866	4
ScorableParty.java	1.4	0.611	3
BowlerFile.java	2	0.894	3

3.4.4 Instability

Before: 2.319 mean with 4.062 stdev **After:** 1.662 mean with 1.085 stdev

3.4.5 McCabe cyclomatic complexity

The maximum cyclomatic complexity is down significantly. These were the top four cyclomatic complexities previously:

File	Mean	Std. Dev	Maximum
Lane.java	5.118	9.498	38
LaneView.java	5.167	6.543	19
LaneStatusView.java	3.4	3.878	11
AddPartyView.java	3.5	3.452	11

and these are the top four now:

File	Mean	Std. Dev	Maximum
LaneStatusView.java	2.75	2.487	7
Lane.java	1.917	1.656	7
LaneView.java	2	1.414	5
AddPartyView.java	2.625	1.317	5

3.4.6 Attribute count

More the attributes in a class, more likely for it to be violating the Single Responsbility Principle. As is clear from the following metrics, we have ensured that such fundamental principles are not violated.

Before: 4.759 mean with 5.556 stdev **After:** 2.351 mean with 2.159 stdev

The top four before:

File	Count
Lane.java	18
NewPatronView.java	17
LaneView.java	15
AddPartyView.java	14

The top four now:

File	Count
LaneStatusView.java	8
LaneEvent.java	7
NewPatronView.java	6
AddPartyView.java	6

4 Structure of the New Code

This document outlines the major groups of classes in our code.

4.1 Overall Structure

- **Controllers** implement the core logic of the application. They are responsible for maintaining the state of the view and redrawing the view on any updates. They utilize necessary helper classes to ensure their aim.
  - ControlDesk
  - Pinsetter
  - Lane
- **Views** are concerned with providing a GUI to the end user, and do not implement any logic. They receive instructions from Controller clases via Observer pattern.
  - AddPartyView (No controller)
  - AdhocView (No controller)
  - BowlerScorerView
  - ControlDeskView
  - LaneStatusView
  - LaneView
  - NewPatronView
  - PinsetterView
  - EndGamePrompt
  - EndGameReport
- **Controller helpers** classes provide necessary abstracted help to the main controller classes, as the following, and both are described later in detail.
  - ScorableParty
  - ScorableBowler
- The Events broadcasted as a part of the observer pattern are done through are done through the 'XEvent.java' classes. It wraps all data needed by the observers from the publisher.

  - LaneEvent
  - PinsetterEvent
  - ControlDeskEvent

- **View helper** classes that help the main View classes in displaying certain specific window components, like a panel of buttons or a grid panel, etc.
  - ButtonPanel
  - ConainerPanel
  - FormPanel
  - GridPanel
  - SrollablePanel
  - TextFieldPanel
  - Generic Panel (Abstract superclass)
  - WindowFrame

- **Abstract classes** that layout the necessary functions for classes to implement and decouple our implementations from the requirements due to it's users.
  - Observer
  - Publisher
  - Event
  - LaneWithPinsetter

- **Utility classes** that help with Input/Output and maintaining data, often for specific classes.
  - Util
  - PrintableText
  - BowlerFile
  - ScoreHistoryFile
  - ScoreReport

- Finally, we have the **driver class**, which initializes an Alley with the corrrect parameters and lets it run:
  - BowlingAlleyDriver

- Lastly, there is a single Test file, to automate correctness checks of the lane scoring mechanism. We have not added more as it's not an everchanging piece of code, most of the testing is done through asserts, and throwing errors under unexpected events.
  - BowlerScorerTest

4.2 Responsibilities of each major class

1. `Lane`: Manages an entire party and its scoring on a lane. It makes sure bowlers get to bowl one after the other, are able to play multiple games one after another, and in the end can exit the lane while knowing their scores.

2. `ControlDesk`: Allows parties to be added to the alley, wait in the queue to be assigned a lane, and then assign them to a free lane.

3. `ScorableParty`: a specialization of a party, maintains a list of bowlers, and also manages final cumulative scores and per game scores. Acts as a **mediator** between Lane and a collection of ScorableBowler.

4. `ScorableBowler`: a specialization of a Bowler, in addition to being able to bowl balls down an alley, it can also keep track of its per game score.

5. `Widget` package: maintains several (abstract) classes that **significantly simplify** the setting up of Views in our project.

6. `Pinsetter`: maintains the state of a pinsetter per lane, and ensures a valid number of pins fall down every time a bowler rolls.

4.3 Describing Specific Classes

4.3.1 AddPartyView

GUI for adding a new party to the control desk. This supports being able to add or remove patrons from a party, or enlist a new patron and then add him.

4.3.2 Alley

Represents an Alley, it's a stub class given it basically integrates ControlDesk and its View together, but given our requirements, this is fine.

4.3.3 BowlerFile

Manages the backend of persistently storing patron nickk names, emails, and full names. Has public methods for adding a new patron, searching for existing patron, etc.

4.3.4 Bowler

Bowler which can roll balls and keep track of its own frame.

4.3.5 BowlerFile

Bowler that keeps track of all the patrons in a file.

4.3.6 BowlerFile

Stub class that tracks a bowler's metadata - name, email, and fullname.

4.3.7 BowlerScorerTest

Testing file for the scoring mechanism in bowling. Does different types of rolls and matches the expected cumulativbe score.

4.3.8 BowlerScoreView

GUI that renders a horizontal table of around 20 cells that update everytime that bowler makes a throw. It is unique for each bowler.

4.3.9 ButtonNames

Utility class for all the different buttons names on all different modals.

4.3.10 ControlDesk

Manages the setting up of all the lanes and the buttons and the party queue.

4.3.11 ControlDeskEvent

Event that is supplied to control desk's observers

4.3.12 ControlDeskView

The view for the control desk (figure 1 in section 2)

4.3.13 EndGamePrompt

The prompt that asks a party whether to replay a game or whether to exit the alley.

4.3.14 EndGameReport

A modal asking user to print report or not, and then exit the alley.

4.3.15 Event

A marker interface for the different events.

4.3.16 Frame

A single frame in the bowler's scoring. Each frame handles its own rolls and score contribution.

4.3.17 Lane

Major class managing the party in a lane, cycling through its bowlers, sending events to views, asking for prompts on game end.

4.3.18 LaneEvent

Helpful information for observers of lane contained here.

4.3.19 LaneStatusView

Displayed in control desk in figure 1 for each lane.

4.3.20 LaneView

The dynamic tabular view of a lane in a new window.

4.3.21 LaneWithPinsetter

A simple lane class that manages all the interactions of a lane with the pinsetter. This is a non-scorable lane, it does not handle scoring, which is handled by lane.

4.3.22 LastFrame

A specialization of the Frame class to handle certain special scoring conditions in the last frame since it has three possible rolls.

4.3.23 NewPatronView

Figure 3b in the pdf, adding new patron to a party.

4.3.24 Observer

General interface that helps us implement an observer pattern in our code.

4.3.25 Party

A stub class that keeps information of its bowlers.

4.3.26 PinsetterEvent

Event published by a pinsetter everytime pins fall in it.

4.3.27 PinsetterView

The view for the pinsetter where it displays fallen pins.

4.3.28 PrintableText

Utility class for printing text to screen.

4.3.29 Publisher

Abstract class for implementing the observer pattern, keeps track of subscribers of a class.

4.3.30 ScorableBowler

A bowler specialization that can also keep track of its scoring, specifically its final scores after every game end

4.3.31 ScorableParty

A party specialization with added scope for scoring wherein all the bowlers keep track of their final scores.

4.3.32 Score

Stub class to keep information about the score on a particular date by a particular bowler.

4.3.33 ScoreHistoryFile

Utility class for writing scores to a dedicated file, retrieving scores from that file, and answering ad-hoc queries.

4.3.34 ScoreReport

Utility class to write scores and send them in an email

4.3.35 Util

Other common utility methods used consistently by other classes.

5 Issues with the Old Code

5.1 LANE - the "God class"

The single biggest problem with the original code is the existence of a "God class" - Lane.java. God classes are notorious for violating the foundational principle of software desisn - Single Responsibility Class: whereby each class is supposed to perform only one task in the context of the application, and delegate sub-tasks to other helper classes.

Expectedly enough, when we ran cloc (a CLI utility to count lines of code) on the source code, we found that out of 1814 lines of code (excluding comments), 294 were alone in Lane.java (nearly 17% of entire project), the maximum among all other classes.

5.2 Code repeated throughout codebase

Quite a lot of code was duplicated line by line through many classes in the codebase, especially in the UI, for example:

- Window centering logic in View class constructors.
- List view scrollable pane, attaching listener, etc.
- Setting up Button Panels inside of Flow and Grid Layouts.
- Getting Date Strings, Waiting for an event, and many more.

Here's a real example form AddPartyView.java:

```
addPatron = new JButton("Add to Party");
JPanel addPatronPanel = new JPanel();
addPatronPanel.setLayout(new FlowLayout());
addPatron.addActionListener(this);
addPatronPanel.add(addPatron);

remPatron = new JButton("Remove Member");
JPanel remPatronPanel = new JPanel();
remPatronPanel.setLayout(new FlowLayout());
remPatron.addActionListener(this);
remPatronPanel.add(remPatron);

newPatron = new JButton("New Patron");
JPanel newPatronPanel = new JPanel();
newPatronPanel.setLayout(new FlowLayout());
newPatron.addActionListener(this);
newPatronPanel.add(newPatron);

finished = new JButton("Finished");
JPanel finishedPanel = new JPanel();
finishedPanel.setLayout(new FlowLayout());
finished.addActionListener(this);
finishedPanel.add(finished);
```

As you can see it is clearly duplicated a lot of times.

Apart from the glaring errors in code redundancy spanning all classes that were fixed by complete restructuing, some code exists which is used by multiple functions but not native to a class in the inteheritance tree. A few examples are writing to a file, or getting the date string. We have made a **Util class** for all of this, abstracted out the common logic there.

5.3 Misplaced Observer pattern logic

- Each of the three Controller classes implemented their subscriber logic by using their own `subsriber` vector and their own `subscribe` method. This was **duplicated** across all three controllers.
- Moreover, placing Observer pattern logic in the Controller class was a **semantically poor design choice**, as a Controller's sole purpose is to control observers, and implementing barebone's of Observer goes against this. It is like writing `pop` and `push` functions of a stack just so that you can use a stack.
- We fixed this by introduing a general `Publisher` class.

5.4 Reinventing the wheel

- The original code wrote its own implementation of a Queue class in Queue.java
- This was completely useless as Java built-in modules already provide several different implementations of a queue.
- Not only that, the original Queue.java was not even **generified**, instead, it was tied down to a Party class. That meant, the Queue **wasn't even reusable** for other data types if we wanted to reuse it.
- At first, we generified it, but later we ended up deleting and replacing with a built-in `java.util.LinkedList` without any loss of functionality.

5.5 Excessively high cyclomatic complexity

Whoever wrote this code was surely a magician, otherwise as it is simply impossible for humans to write a method which has dozens of if-elses and magic constants spread all over it. You may look at `private int getScore` in Lane.java if you do not believe us. We shall not reproduce its code here, but rest assured that by looking at, your eyes shall bleed and you shall never be unsee the tyranny of harmless looking if-elses.

We ended up having to rewrite that method from scratch since there was simply no way to determine how it was calculating the score.

This isn't the only method though, you may also look at `Lane.run` or `Lane.receivePinSetterEvent` or `LaneView.receiveLaneEvent`. There are more, but a consistent pattern across them is **deep nesting of if-elses inside loops or more if-elses**, which evidently increases the complexity manifold.

5.6 Feature envy

This is an extremely common code smell, where in the midst of development, developers forget that class A is doing a lot of work from class B, when in fact that work should have been done by class B itself. We identified a few feature envies in different places, such that we count at least three accesses as ”repeatedly accessed”

1. Party and Bowler accessed repeatedly in `Lane.run`
2. Lane accessed repeatedly in `LaneStatusView.actionPerformed`: this was actually a simpler case of code duplication unlike others
3. Party accessed repeatedly in `Lane.assignParty`

We fixed all such cases by moving the relevant work to that class itself, instead fo calling its methods repeatedly.

5.7 Minor issues

There were other minor issues plaguing the codebase, the most common ones are the following. Also note that we have only shown one or two occurrences of these minor issues, however, they occurred much more frequently in the codebase.

5.7.1 Overly Broad access specifiers

- Many methods and properties **marked public** without a reason, or if not that, had overly broad access specifiers (package-private instead of private).
- Since most of the code is contained in a package and only the driver should ever need to be called externally, these access specifiers have been changed to **private where possible**, package-private otherwise, and protected in very few cases (where inheritance from Widget is required).

5.7.2 Local values stored as class properties

This pattern is common across files, especially in LaneView.java. Many of the 2D JPanel arrays are not required to be class properties, and could be simply moved to local variables. This corresponds to the principle of **information hiding**, where only as little information is exposed outside a method as is really necessary.

5.7.3 Badly Written Loops and Conditionals

- Many loops used iterators to iterate over vectors. We replaced them with the new for-in loop in Java.
- It has this syntax: `for(int value :  array)`, which is **succint and clear** as opposed to the verbose iterator syntax.

5.7.4 Outdated Vector collection used

- `Vector` had been used throughout the code base to maintain dynamically sized collections of object.
- It is widely known that vectors used synchronous blocking operations and therefore are very slow as compared to their array/`ArrayList` counterparts.
- Therefore, we replaced them with ArrayList throughout.

5.7.5 Comment-based version control

The files were using a unique approach to version control, namely, putting edit-log comments at the top of each file, timestamped, with "useful" messages like *"Added things"* and *"Works beautifully"* (Lane.java) This comment log sometimes grew to a sprawling size of more than 100 lines (Lane.java: 130+ lines)

As we already know, such a version control is completely useless when compared to Git.

5.7.6 Useless comments

Many classes and methods carried useless comments with them. A few actual examples:

```
/**
 * Class to represent the pinsetter
 */

class Pinsetter{ ... }
```

or this gold comment:

```
/** Pinsetter()
 *
 * Constructs a new pinsetter
 *
 * @pre none
 * @post a new pinsetter is created
 * @return Pinsetter object
 */
public Pinsetter() {
```

While it can surely be agreed that the comment is correct, it is widely recognized that **comments suffer from aging**. For example, if the Pinsetter class was renamed, or the constructor was simply moved from its current position in the file to somewhere else, someone may forget to change the comment. Later in time, the comment will definitely not be helpful.

Moreover, it is widely accepted that **Writing comments is good, but not having to write comments is better.** (read and read). In general, throughout the codebase, we have removed such useless comments where possible, and retained useful comments where required.

5.7.7 Bad identifier names

There were several identifier names that both violated the Camel Case naming format and often used 1 letter names like i. All that has been changed to semantically useful names, then number of these parameters has been drastically reduced, and subsumed in the inheritance heirarchy.

5.7.8 Magic constants

Many many magic constants exist throughout the codebase. We can look at makeFrame method in LaneView.java:

```
balls = new JPanel[numBowlers][23];
ballLabel = new JLabel[numBowlers][23];
scores = new JPanel[numBowlers][10];
scoreLabel = new JLabel[numBowlers][10];
```

or even at this gold comment in Lane.java:

```
finalScores = new int[party.getMembers().size()][128]; //Hardcoding a max of 128 games, bite
```

As we know well, magic constants are discouraged throughout industry as they hamper **reusability** as well as make the codebase **hard to understand**. We do not know if repeated magic constants refer to the same thing or meant something else. They make things messy, so to speak.

5.7.9 Semantically incorrect subclassing

As you can read on Stackoverflow, since the Lane class is not really specializing any of Thread's behavior by extending it, the original code had two instances of semantically incorrect `extends Thread` - ControlDesk and Lane.

5.7.10 Dependent variables

There were a handful of instances where two class properties were **tightly coupled** to each other, such that if you changed any one and forgot to change the other, the entire logic would break. An example would be `isPartyAssigned` property in Lane.java, which was always true when `party != null` and false otherwise. We had fixed this by replacing the single usage of isPartyAssigned with the above conditonal.

5.7.11 Incorrect task distribution between classes

It was quite often the case that some work, that ideally should have been done by one file, was done by another. For example: there are three occurrences of `.getNickName() + "'s Party"` throughout the codebase. It's supposed to generate a party's name. Not only is it duplicated, moreover, it is not present inside Party.java. **The name of the party is being decided by classes other than the Party itself!**

We fixed this by providing a `Party.getName()` method, as any sane and logical coder would have done.

5.7.12 Auto-Fixes using the IntelliJ linter

There are massive fixed using the intelliJ linter primarilty on bad access specifiers, lack of defensive copy, etc. All of that has been fixed. A list of them is as follows:

	Mean	**Std. Dev**	**Maximum**

6 New features

6.1 Maximum players in an Alley

This was simple to implement. We only changed the number of maximum members in Alley.

6.2 Ad-hoc queries on user data

- We have provided a queries button in the UI of the main panel, using which user can invoke a panel which allows for three types of queries: (1) Best scorer (2) Worst scorer (3) Highest cumulative score so far (4) Top 5 career best scores of a bowler
- This was made possible by using the Score history DAT file, which already logs all the previous scores.
- Classes implementing this functionality:
  - AdhocView.java: for managing the View of the queries
  - ScoreHistoryFile.java: for managing the backend of the actual queries.

6.3 Pause and resume games on a Lane

- This was relatively simple to implement as we have split up Lane into Lane, ScorableParty, and ScorableBowler. Each of these classes now have a `saveState` and `loadState` method.

- Every time the lane is paused, first the ScorableParty calls saveState on each ScorableBowler it has. Each bowler then saves its own state in order. Finally, the party then saves certain specific information. Load state proceeds in a similar fashion.

- As you can see, all the three `saveState` calls are **decoupled** from each other. We can happily change the logic in one class, and it would not affect the other classes in any way.

- As for the frontend, we have provided two extra buttons in LaneStatusView.java that lets the users pause and resume the game.

- By design and logically, paues button is only enabled when the party is playing, and resume button is enabled only when party is paused or is not playing.