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 **extensibility and modularity**, and implement only a few new features into it. We have toiled hard to deliver a codebase which is as extremely maintainable and flexible, and no top of that, satisfies popular laws about software engineering code metrics.

We have also included all three features, namely:

- Maximum multiplayer up to six players
- Being able to 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

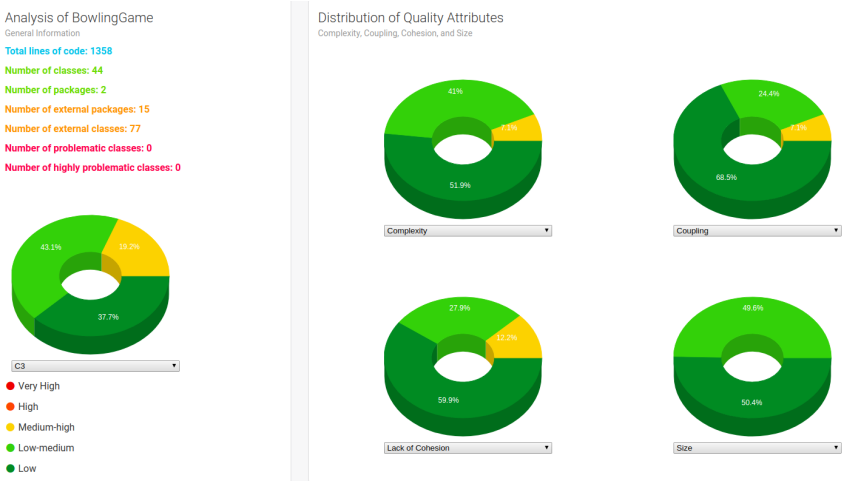
Effort put in: 24 hours of work-time by each member. Role of each member:

- Gaurang: Complete Refactor (Specially reducing cyclomatic complexity, implementing observer pattern)
- Animesh: UI based refactors (from the View Classes), miscellaneous fixes
- Avani: Building Documentation and fixing several miscellaneous code quality errors

2 Code Metrics

2.1 Plots of CodeMR

(a) Before the Refactor



(b) After the Refactor

Figure 1: Code Metrics

2.2 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.

2.2.1 Metric based enlightenment

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: lanePublish
  - Maintaining a observer model: subscribe, unsubscribe, publish
  - Actually running the game for a party: run, assignParty, isPartyAssigned

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

2.2.2 Lack of cohesion

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

2.2.3 Coupling

Before: After:

2.2.4 Nested block depth

Before: 1.511 mean with 1.177 stdev **After:** 1.356 mean with 0.624 stdev

Moreover, the largest offender Lane.java (2.176 mean with 2.065 stdev) was reduced to a mere 1.5 with 0.866.

2.2.5 Instability

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

2.2.6 McCabe cyclomatic complexity

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

Moreover, 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 Refactoring Narrative

3.1

This describes the balancing done in refactored files. Our first aim was to solely reduce the cyclomatic complexity present in different methods across files. This 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

Splitting up each method usually meant that large top level methods were now very easy to understand (as they only called a set of related helper methods sequentially), and the helper methods themselves had narrow work, which meant they could be reused across the codebase for different top level methods.

However, this usually meant that the cohesion in a class decreased considerably, since now there existed several methods, with each method not using all of the properties. This was now the time for us to split the class into many more classes. For example, this is the precise technique we employed with Lane.java, the behemoth monster in this codebase.

We first split it out into a LaneScorer, that was supposed to handle the entire scoring logic in the Lane. Too soon, we realized that the scoring logic per bowler is being duplicated a lot. So, we split it into a BowlerScorer, which handles scoring logic for just one bowler. BowlerScorer is now used in LaneScorer, which is used in Lane. The segregation of logic is now very clear

- Scoring for one bowler: BowlerScorer
- Scoring for an entire lane (multiple bowlers): LaneScorer
- Administering facilities for an entire lane: Lane
- GUI for one bowler's score sheet: BowlerScoreView
- GUI for scores of all bowlers: LaneView

(provide more detail on more splitups)

Now, we looked at duplicated code across the codebase. One feature 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.

Since our Widget.X classes were built upon the Builder pattern, we ran into the issue of Law of Demeters...

However, now we had too many classes and had to deal with too much coupling....

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 The primary classes

TODO: what to write?

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

Stub class for a Bowler object, which contains a name, full name and its email.

4.3.5 BowlerScorer

Companion to LaneScorer; manages the scoring for individual bowlers of one party on one lane.

4.3.6 BowlerScorerTest

A test file to make sure scores are correctly updated in BowlerScorer.

4.3.7 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.

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.

5.3 Repetition of Code

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.

5.4 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.

5.5 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.

5.6 Minor issues

There were other minor issues plaguing the codebase, the most common ones being:

5.6.1 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).

5.6.2 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.

5.6.3 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.

5.6.4 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.

5.6.5 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.

5.6.6 Bad identifier names

5.6.7 IntelliJ ne jo bhi bola

TODO: once minor issues are done, move some of them to a table as required in assignment

Bad identifier names		
c	c	c

6 New features

6.1 Maximum player

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
- This was made possible by using the Score history DAT file, which already logs all the previous scores.

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.