ASSIGNMENT 3
Bowling Management System

This project was submitted on 9th April 2020. The team code is Team "BC". The team members are:

- Anirudh Palutla (2018113007)
- Swastik Murawat (2018101022)
- Tanish Lad (2018114005)

The hours of effort put in by each member are:

- **Anirudh Palutla** (Refactoring: 18 hours + New Features: 8 hours + Documentation and Diagrams: 4 hours. Total **30 hours**)
- **Swastik Murawat** (Refactoring: 18 hours + New Features: 4 hours + Documentation and Diagrams: 8 hours. Total **30 hours**)
- **Tanish Lad** (Refactoring: 18 hours + New Features: 8 hours + Documentation and Diagrams: 4 hours. Total **30 hours**)

Overview

The product is a bowling management system for a bowling alley. It allows you to add and save player names and details along with their records for every game. The system has the functionality to simulate a full game of bowling along with a printable scorecard. It simulates a bowling alley, which has multiple lanes. A party can be added to the alley upon which two things may happen - it gets added to a non-occupied lane or it gets added to the party queue and will be assigned a lane as soon as one frees up in order of the queue. During a game, you have the option to view the scores of the bowlers live, and also view the status of the pins and see exactly which pins have been downed.
The system also lets you query for statistics on players such as their highest and lowest scores, their total, and their scores for the last 5 games. There are more query options available to get the all time highest and lowest scorers, best and worst bowlers, and least and most experienced players.
The management system also provides functionality to pause a game in between after which the game can be resumed or the system can be closed - upon which

the game is saved. You have the option to load an existing game from the list of saved games.

UML Class Diagrams:

The UML class diagrams have been divided into 4 for the original and refactored code:-

1. ControlDesk
2. Lane
3. Pinsetter And ScoreReport
4. OtherClass like AddPartyView, NewPatronView
5. NewFeatures (only refactored code)

The UML class diagrams are in the folders:

- ../diagrams/OriginalCode/UMLClassDiagrams
- ../diagrams/RefactoredCode/UMLClassDiagrams

Sequential Class Diagram:

The sequential diagram shows the general flow of the game.
The sequential diagrams are in the folders:

- ../diagrams/OriginalCode/SequentialDiagram
- ../diagrams/RefactoredCode/SequentialDiagram

TABLE: Major classes and their responsibilities

CLASS	**RESPONSIBILITIES**
drive	• Main class to start the game • Creates Alley
Alley	• Creation of ControlDesk
ControlDesk & ControlDeskView	• Gives option to query, add party, or finish the game • Displays active Lanes as LaneViews and displays the Party queue • Handles creation of Lanes • Handles assignment of Parties to Lanes

DatabaseHandler	• Handles reading and writing the information about each game to and from the database
QueriesView & QueriesHandler	• Provides a view and functionality for ad-hoc queries from the database
Lane	• Creates game on single Lane • Manages the game running on this lane with the help of other classes • Gets and manages information about downed pins from PinSetter • Information about current party with Party class • Players managed with Bowler class
LaneView	• Gives option to display the scoreboard with score of each player as the game goes on (in a new window) • Display lane information such as pins downed, current bowler on ControlDeskView window • Give option to display pins (in a new window) • Option to pause, resume and end the game on Lane
Pinsetter	• Creates pinsetter holding info about pins standing • Provides the option to reset pins • Simulates a single throw in a game
PinsetterView	• Creates Pinsetter window which displays current situation of pins • Changes the display of the pins after pins have been knocked down after every throw of the bowler
AddPartyView	• Gives option to add selected members to a party and register new members • Adds newly added bowlers to the list of members • Adds the selected party to the Party queue on finishing
NewPatronView	• Creates the AddNewPatron window • Takes data of new player • Passes the new patron to addParty class
EndGameReport & EndGamePrompt	• Gives functionality to start another new game or finish playing • Creates a window which gets displayed whenever player chooses not to play a new game. • Player can choose to print report or finish the game.
LoadExistingGame View	• Provides functionality to load an existing game from the list of saved games found

Bowler	• Stores information about a single bowler
ScoreCalculator	• Calculates the scores on the scorecard for the bowler after every throw

Original Design Narrative:

The original design followed OOPs concepts which is very helpful in keeping the code modular. In the design we have seperate classes for each component in the game like Lane, ControlDesk. For each component it has been further broken down to class for handling views, handling events and a class for maintaining states. This made it easier to allocate responsibilities to classes individually.

However, there are few classes in the original design like Lane which are not coded well. This class does too many things and takes on too much responsibility. It is better to break down such classes into smaller classes and allocate each of those smaller classes responsibilities.

The publisher-subscriber design pattern was used heavily within this project, and rightly so due to the asynchronous nature of the events occuring. This can be seen very effectively in classes such as Pinsetter, where PinsetterEvent events occur and are published to the various subscribers waiting for the changes. Interfaces were used very well for this purpose. Information was handled well within the bowling management system due to Java programming's predominant use of private variables along with getters and setters. Due to the usage of the publisher-subscriber pattern, Event classes are very common throughout the code so the need to reset these variables does not arise a lot as these events are created and destroyed every time changes occur.

The code is generally well written and designed, with the exception of a few classes such as Lane and ControlDesk which contain very large methods and needed refactoring. However, data storage is not done in a very efficient manner in this design as all the details related to the bowlers, their scores, etc. are stored in plain-text DAT files. This makes them hard to manipulate and store complex information as getting and writing information from plain-text formats gets increasingly complex as the data to store increases.

The design stays true to its design doc for the most part. It implements the features that are mentioned in the documentation and the code is well-documented. However, the storage of data in this case is redundant as there is no way to access the data through the

application. It also lacks functionality to remove patrons (bowlers) once added. The code also contains a lot of legacy code which has been deprecated and needs to be replaced.

Code Smells:-

TYPE	WHERE?	PROBLEM
Large Class	Lane	This class is doing too much work It calculates Score It marks score on board Method to receive the throw event from pinsetter.
Large Method and many if condition	Lane Method :- recieverPinsetterevent,run, getScore	These many have too many if conditions and loops making complexity of each method high and thus increasing complexity of class.
Redundant method	Lane Method:- isGameFinished()	This function was never used.
Complex method	AddPartyView Method :- actionPerformed()	This function has too many if conditions thus increasing complexity.
Repetition of code	ControlDeskview,LaneStatusView,NewPatronView	All these classes have to display windows in the centre, so they have the same code which is repeated in all these classes.
LongMethod/ Code repetition	ControlDeskview,LaneStatusView,NewPatronView,PinsetterView	All these classes have to create a window with some buttons . This leads to code repetition. This can be easily solved by creating a class and passing the parameter.
Interface not used	Interface :- LaneServer LaneEventInterface	These interfaces were never implemented.

Narrative of Refactored Design

The main purpose of refactoring is to fight technical debt. It transforms a mess into clean code and simple design.
The following points narrate our path from a dirty messy code to a clean simple code.

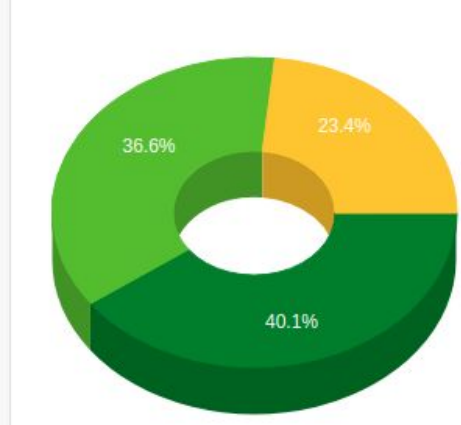
1. We started off with reducing the cyclomatic complexity present in various methods. For that, we had to understand the purpose of the method, and then breaking it down into different classes/methods if it serves more than 1 purpose. Some classes in which reduction of cyclomatic complexity may be visible are: AddPartyView, Lane, ControlDesk, LaneStatusView, etc.
2. Due to increase in the number of methods from the reduction of cyclomatic complexity, the cohesion among methods decreased (the lack of cohesion increased). So our next task was to increase cohesion among methods of the same class. We searched through many classes, and found that there are some methods which are present in many classes and don't have any particular relation to the core functioning of that class. We extracted those methods and put them into new classes, making sure that similar methods go into the same classes. Thus, we successfully managed to increase cohesion. From the original code which had around 30 classes, our refactored code had 50+ classes!
3. Reusability and Elimination of Duplicate Code: We noticed that several parts of the codebase had almost similar design but varied in only 1 or 2 parameters. Presence of duplicate code can be found in all the view classes where Scroll Panels, Button Panels, Window Frames, Text Panels are being created. What we did was we made separate classes for each of these Panels which inherited a common class which did all the work of setting the layout, setting the title, etc. The specific panels did their job of creating those types of Panels. These panels were then instantiated across multiple files reducing the duplicated code and also making the code cleaner. The reduction in duplicate code can be found in AddPartyView, LaneStatusView, ControlDeskView, EndGamePrompt, NewPatronView, etc. (essentially all the view classes along with some other files)
4. The Single Responsibility Principle (SRP): This principle states that every class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class, module or method. This principle was used multiple times to break long, complex classes into small, simpler classes. ScoreCalculator was extracted from Lane, PartyHandler was extracted from ControlDeskView, the LaneStatusView was separated into multiple classes.
5. Coupling: Another big challenge for us was to reduce coupling between classes. This was a big challenge because there is a trade-off between coupling and cohesion.

Increasing cohesion leads to increase in coupling and vice-versa. The balance that we settled on is to reduce coupling and increase cohesion, but not letting any values go to the extremes but trying to minimise and maximise coupling and cohesion respectively.

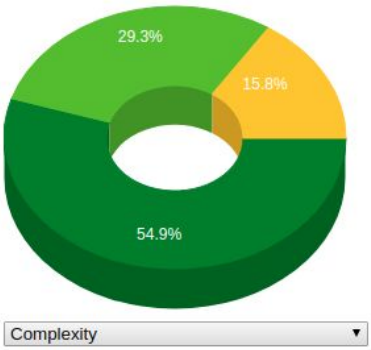
6. Law of Demeter: The Law of Demeter (LoD) is a simple style rule for designing object-oriented systems. "Only talk to your friends" is the motto. So reducing coupling helped in this.
7. Deprecated Methods: There were many instances of use of deprecated code such as .show() and .hide() in various window frames and view classes. We replaced them with modern style .setVisible(true) and .setVisible(false).
8. Extensibility: An extensible program is one which can be extended easily without modifying much of the original code base. We did follow this, first by refactoring the original code base and then only implementing the new features desired. New features were implemented without much hassle only because our refactored code was extensible.
9. IntelliJ Auto Analyser: The IntelliJ IDE helped us by warning us about various code smells and various discouraged code styles in Java, and also fixed many of them automatically. Some of the things it did included: addition of final modifier, detecting excessive cyclomatic complexity, removing unused variables and methods, use of enhanced for-each loop whenever possible.
10. Design Patterns: Various Design Patterns were accounted for when Refactoring. Some of the Design Patterns used are: Builder Pattern (for building various Panels), Observer Pattern, Publisher Pattern, Iterator Pattern (for iterating through various types of objects and being able to change the data type without much change in the original code), etc.
11. Information Hiding: Information hiding is a concept which protects the data from direct modification by other parts of the program. Use of private variables was done to implement this.

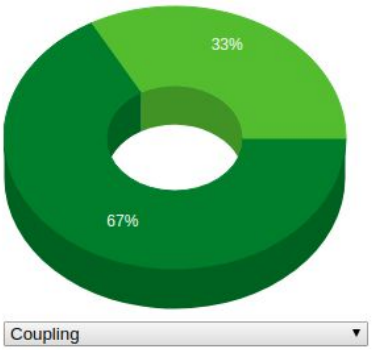
Metrics Analysis

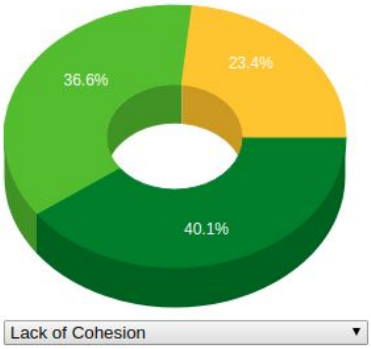
Before the Refactor

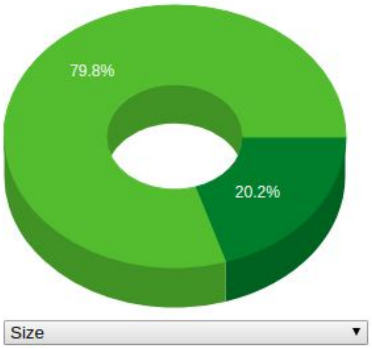


Overall C3 Metrics



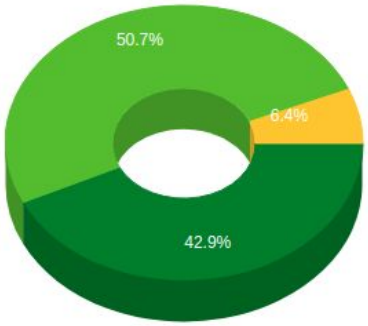




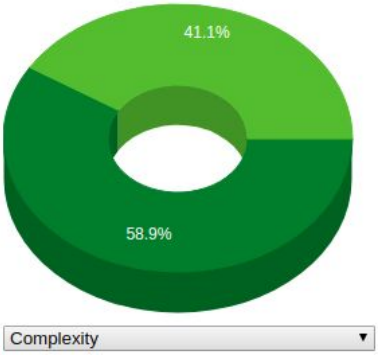


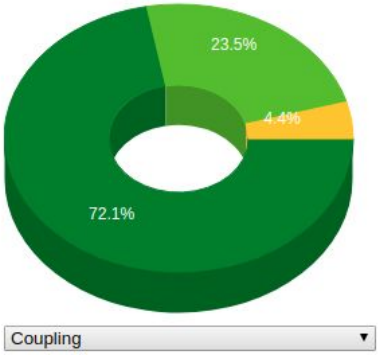
Pie Charts of The Most Important Metrics

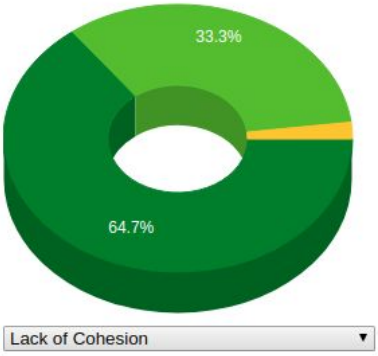
After the Refactor

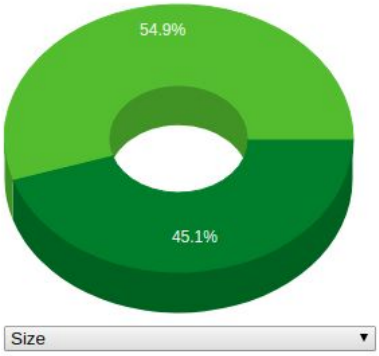


Overall C3 Metrics









Pie Charts of The Most Important Metrics

As can be seen from the Graphs, metrics improved to a good extent.