Bowling Alley Simulation Refactoring

Team 13 Date: 12/2/2020

SR. NO	NAME	ROLL NUMBER	HOURS WORKED	RESPONSIBILITIES
1	Tushar Patil	2019201041	12	Highlighted in misc/Code Metrics.xlsx
2	Vivek Puar	2019201044	12	
3	Sayan Dey	2020201007	12	
4	Abhigyan Ghosh	20171089	12	

Project Overview

It is a virtual game designed in JAVA. The game lets users enjoy the game in a remote setup by simulating the Bowling environment. The key features of this game are:

1. **Create Player**: A new player can be added to the game on demand. Once the player registers, his entry is updated in the system and now he is eligible to play the game after joining any party.
2. **Adding party:** There will be multiple lanes. The players can be grouped and added to the party and assigned to any of the free lanes. In case none of the lanes are available, the parties are assigned the lanes on FCFS basis once any of the lanes are available.
3. **Scoreboard:** It keeps track of the score of each player in the party. The score is calculated as:
4. Score = number of pins dropped
    a. Strike : Pins dropped in next 2 rounds + 10
    b. Spare : Pins dropped in next round + 10
5. **Control Desk:** It has access to the score of any of the lanes.
6. **Pinsetter:** It simulates the pins dropped after each throw. After two consecutive throws it will re-rack the pins.

UML and Other Associated Diagrams

Manually constructed UML diagram is available in *misc/UML.odt*

Automatically generated UML diagram is available in *misc/Initial_UML_class.png*. Automatically generated sequence diagram is available in *misc/Initial_Sequence.png.*

The final UML of the refactored code is available at *misc/Final_UML_class.png*

The responsibility of each major class:

Bowling Alley Simulation is a collection of 29 files, which contains different classes and functions at together makes simulate this game. Below is the list of major classes and their responsibility.

Class	Attributes	Functions	Major Functionality	Connected class
AddPartyView	• Int maxSize • vector party • vector bowlerdb • Integer lock • ControlDeskView controlDesk • String selectedNick • String selectedMember	• Void actionPerformed() • Void valueChanged() • Vector getNames() • Void updateNewPatron() • Vector getParty()	• Add new patron to party • Remove patron from party • Create new party • Finished party selection • Return latest state of party	• NewPatron View
Alley	• ControlDesk controldesk	• ControlDesk getControlDesk()	• Return present state of the controldesk	• ControlDesk
Bowler	• String fullName • String nickname • String email	• String getNickName() • String getFullName() • String getNick() • String getEmail()	• Validation of bowler details • Getter functions	---
BowlerFile	• String BOWLER_DAT	• Vector getBowlers() • Void putBowlerInfo() • Bowler getBowlerInfo()	• Get details of all bowlers • Add bowler info • Get details of one bowler	---
ControlDesk	• Hashset lanes • Queue partyQueue • Int numLanes • Vector subscribers	• Bowler registerPatron() • Void assignLane() • Void addPartyQueue() • Vector getPartyQueue() • Int getNumLanes() • Void subscribe() • Void publish() • HashSet getLanes()	• Registering a patron. • Assigning a lane. • Broadcast an event to subscribing objects. • Creating a new patron. • Finished party selection. • Return party names for displaying in GUI. • Setter and getter functions.	• Lane
ControlDeskEvent	• Vector partyQueue	• Vector getPartyQueue()	• Returns a vector of the names of the parties in the waiting queue	---
ControlDeskObserver	---	• void receiveControlDeskEvent()	• Interface for classes that observe control desk events.	---
ControlDeskView	• Int members • ControlDesk controlDesk	• Void actionPerformed() • Void updateAddParty()	• Handler for action events. • Displaying GUI for the control desk.	• AddPartyView • ControlDesk

		• Void receiveControlDeskEvent()	• Receive new party from addPartyView. • Receive broadcast from controlDesk.	
Drive	• Int numLanes • Int maxPatronPerParty	• Void main()	• Main and driver class for the entire game. • Creates new alley with given number of lanes. • Activate control desk event. • Render the GUI for the control desk via ControlDeskView.	• Alley • ControlDesk • ControlDeskView
EndGamePrompt	• Int result • String selectedNick • String selectedMember	• Void actionPerformed() • Void getResult() • Void destroy()	• Display End Prompt • Destroying the currently active game object.	---
EndGameReport	• Vector myVector • Vector retVal • Int result • String selectedMember	• Void actionPerformed() • Void ValueChanged() • Vector getResult() • Void destroy() • Void main()	• Displaying Endgame Report. • Destroy current active game object.	---
Lane	• Party party • Pinsetter setter • Hashmap scores • Vector subscribers • Boolean gameIsHalted • Boolean partyAssigned • Voolean gameFinished • Iterator bowlerIterator • Int ball • Int bowlIndex • Int frameNumber • Boolean tenthFrameStrike • Int[] curScores • Int[][] cumlScores • Boolean canThrowAgain • Int[][] finalScores • Int gameNumber • Bowler currentThrower	• Void run() • Void receivePinSetterEvent() • Void resetBowlerIterator() • Void resetScores() • Void assignParty() • Void markScore() • LaneEvent lanePublish() • Int getScore() • Boolean isPartyAssigned() • Boolean isGameFinished() • Void subscribe() • Void unsubscribe() • Void publish() • Pinsetter getPinsetter() • Void pauseGame() • Void unpauseGame()	• Keep track and calculates bowler scores. • Simulates bowling alley lanes in game. • Ensures cyclic rounds of each bowlers turn. • Assigns party to lane.	• Bowler • Party • Pinsetter
LaneEvent	• Party p • int frame • int ball	• Boolean isMechanicalProblem()	• Setter and getter functions for all	• Bowler • Party

	• Bowler bowler • int[][] cumulScore • HashMap score • int index • int frameNum • int[] curScores • boolean mechProb	• Int getFrameNum() • Hashmap getScore() • Int[] getCurScores() • Int getIndex() • Int getFrame() • Int getBall() • Int [][] getCumulScore() • Party getParty() • Bowler getBowler()	lane functionalities.	
LaneEventInterface	• Interface class	---	• An interface for multiple class	• Bowler • Party
LaneObserver	• Interface class	---	• An interface for multiple class	---
LaneServer	• Interface class	---	• An interface for multiple class	---
LaneStatusView	• PinSetterView psv • LaneView lv • Lane lane • int laneNum • boolean laneShowing • boolean psShowing	• Jpanel showLane() • Void actionPerformed() • Void receiveLaneEvent() • Void receivePinsetterEvent()	• Rendering GUI for status of lanes.	• Lane • LaneView • PinsetterView
LaneView	• Int roll • Boolean initDone • Vector bowlers • int cur • Iterator bowlIt • Lane lane	• Void show() • Void hide() • JPanel makeFrame() • void receiveLaneEvent() • void actionPerformed()	• Render view GUI for alley lanes.	• Lane
NewPatronView	• Int maxSize • String nick • String full • String email • Boolean done • String selectedNick • String selectedMember • AddPartyView addParty	• Void actionPerformed() • Boolean done() • String getNick() • String getFull() • String getEmail()	• Setter and Getter functions	• AddPartyView
Party	• Vector myBowlers	• Vector getMembers()	• Access bowler belonging to a party	---
Pinsetter	• Random rnd • Vectors subscribers • Boolean[] pins • Boolean foul • Int throwNumber	• Void sendEvent() • Void ballThrow() • Void reset() • Void resetPins() • Void subscribe()	• Updates status of pins across all subscribers. • Simulates a ball being thrown and probabilistically creates a result for the ballThrown function(i.e. either as foul or	• PinsetterObserver

			some number of pins)	
PinsetterEvent	• boolean[] pinsStillStanding • boolean foulCommited • int throwNumber • int pinsDownThisThrow	• Boolean pinKnockedDown() • Int pinsDownOnThisThrow() • Int totalPinsDown() • Boolean isFoulCommited() • Int getThrowNumber()	• Count the number of pins dropped.	---
PinsetterObserver	---	• An interface class	• Interface classes.	---
PinsetterView	• Vector pinVect	• void receivePinsetterEvent()	• Pinsetter GUI to display which pin it is. • Receive current state of Pinsetter and GUI changes accordingly.	---
PrintableText	• String text • Int POINT_PER_INCH	• Int print()	• Display graphical text on GUI with colors.	---
Queue	• Vector v	• Object next() • Void add() • Boolean hasMoreElements() • Vector asVector()	• Creates queue	---
Score	• String nick • String date • String score	• String getNIckName() • String getDate() • String getScore() • String toString()	• Set score for player in game.	---
ScoreHistoryFile	• String SCOREHISTORY_DAT	• Void addScores() • Void getScores()	• Writing scores of player in .DAT file when game finishes.	---
ScoreReport	• String content	• Void sendEmail() • Void sendPrintOut() • Void sendln()	• To generate score report and send via email or pintout to user.	---

Review of Original Design

The main aim of the code provided is to automate the process of detecting the number of pins knocked down after any particular bowler has rolled his ball. This information can then be communicated to a scoring station that would be able to automatically put a score for the bowler's game.

Here, **drive** is the class containing the main() method and after executing it, all other required java files get automatically executed and provides the desired functionality. The UML diagrams can be viewed above.

There are some classes in this design which need medium to high level of refactoring due to presence of code smells which hampered the metrics.

For example,

The class '*Lane*' is having most code smells with very low cohesion and high complexity. Also, some other classes like '*ControlDesk*', '*ControlDeskView*' , '*EndGameReport*', '*EndGamePrompt*', '*LaneEvent*', '*LaneStatusView*', '*NewPatronView*', '*PinSetterView* ', '*ScoreReport*' are having code smells which are causing high complexity/high lines of code/high coupling/lack of cohesion.

The design pattern that can be found in the given design is: **Observer** pattern. Observer is a behavioural design pattern that lets us define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing. Here, the interface '*LaneObserver*' is acting as the observer and it is getting information of the events from the '*LaneEvent*' class.

Review of Refactored Design

While refactoring, we have tried to balance all the metrices, but with special attention on:

decreasing the *Coupling*, *Complexity*, *Size*, *Number of Fields*, *Number of Methods* and increasing the *Cohesion* (decreasing *Lack of Cohesion*).

To achieve this, first of all, we have eliminated all the dead codes and redundant attributes/methods. Also, we have replaced public variables with private variables wherever possible. Also, we have spent considerable amount of time for increasing the reusability of the code by increasing modularity. We have decreased the interdependence of classes as much as possible by assigning every class for one particular subtask. But, as this entire design is to achieve 1 entire task, so some coupling remained for obvious reasons.

But as we have only done refactoring and not changed any external behaviour of the design, so the original design pattern followed in the code remained preserved.

Optimization Metrics

1. RFC (Response for a Class): The number of methods that can be potentially invoked in response to a public message received by an object of a particular class. If the number of methods that can be invoked in a class is high, then the class is considered more complex and can be highly coupled with other classes. Therefore, more tests and maintenance efforts are required.
2. CBO (Coupling Between Object Classes): The number of classes that a class is coupled to. It is calculated by counting other classes whose attributes or methods are used by a class, plus those that use the attributes or methods of the given class. Inheritance relations are excluded. As a measure of coupling CBO metrics is related to the reusability and testability of the class. More coupling means that the code becomes more difficult to maintain because changes in other classes can also cause changes in that class. Therefore, these classes are less reusable and need more testing effort.

3. WMC (Weighted Method Count): The weighted sum of all class' methods and` represents the McCabe complexity of a class. It is equal to the number of methods, if the complexity is taken as 1 for each method. The number of methods and complexity can be used to predict development, maintaining and testing effort estimation. In inheritance if base class has a high number of methods, it affects its child classes, and all methods are represented in subclasses. If the number of methods is high, that class is possibly domain specific. Therefore, they are less reusable. Also, these classes tend to be more change and defect prone.
4. LOC (Lines of Code): Total number of lines in a class/package.
5. CMLOC (Class-Methods Lines of Code): Total number of all nonempty, non-commented lines of methods inside a class.
6. LCOM (Lack of Cohesion of Methods): Measure how methods of a class are related to each other. Low cohesion means that the class implements more than one responsibility. A change request from either a bug or a new feature of one of these responsibilities will result in a change to that class. Lack of cohesion also influences understandability and implies classes should probably be split into two or more subclasses. LCOM3 defined as follows LCOM3 = (m – sum(mA)/a) / (m-1) where:
    - m number of procedures (methods) in class
    - several variables (attributes) in class. a contains all variables whether shared (static) or not.
    - mA number of methods that access a variable (attribute)
    - sum(mA) sum of mA over attributes of a class
7. LCAM (Lack of Cohesion Among Methods [1-CAM]): CAM metric is the measure of cohesion based on parameter types of methods. LCAM = 1-CAM
8. LTCC (Lack of Tight Class Cohesion [1-TCC]): The Lack of Tight Class Cohesion metric measures the lack of cohesion between the public methods of a class. That is the relative number of directly connected public methods in the class. Classes having a high lack of cohesion indicate errors in the design.

Link to metrics document: https://iiitaphyd-my.sharepoint.com/:x:/g/personal/abhigyan_ghosh_research_iiit_ac_in/EXUEeO2DYPpMsKmPRsQRjKgBpgksfZCytuTwBtgFuh_eBw?e=i1lesU

It is also attached along with the submission in *misc/Code Metrics.xlsx*. After opening the above link, you can see 2 sheets. The worksheet "Initial" contains the initial metric values for the code and the worksheet "Final" contains the final metric values for the code.