DASS Assignment 3 Design Doc

Team - Group AE (31 Row)

1. Kishan Sairam Adapa - 2018101026
2. Akshat Goyal - 20181010
3. Siddireddy Akash Reddy* (Didn't respond and hasn't contributed anything)

Contributions

Members 1 & 2 (Kishan & Akshat) have contributed equally towards the assignment.
Member 3 (Akash) didn't respond to us even till the day of deadline.

Overview

A bowling establishment is commonly referred to as a bowling alley. A bowling alley is composed of a number of bowling lanes. Bowlers may check in as a group or party so that they will be assigned to the same lane. Each lane can accommodate one to five bowlers. The order in which a party checks in determines the order in which they will bowl.

Once a party has started bowling, the control desk can monitor the number of frames completed by each bowler. Each lane is equipped with a stand-alone scoring station that lists the bowlers' names (in the order they checked in) and a graphic representation of their scores. Bowlers alternate rolling the ball or taking throws according to the scoring rules of bowling referenced in the following section

Initial

Class Responsibility Table

Class	**Responsibility**
AddPartyView	Creates a popup window with the option to add or remove a patron for a party. It also provides us an option to create a patron which will be in the party. Game is started by pressing the Finished button
Alley	Initialises the ControlDesk with a given number of lanes
BowlerFile	Provides us a way to interact with the list of Bowlers in following ways - fetching / updating information of bowlers, adding / removing bowlers
ControlDesk	Represents control desk and carries out functionalities of the program specified by AddPartyView and ControlDeskView

ControlDeskView	Creates view, provides us with an UI for us to add a party or to finish the game. And also provides us with the view of waiting parties and the current lanes assigned.
drive	Starts game by creating Alley
EndGamePrompt	Creates a popup window when the game is over and it provides us the option to restart the game or finish through its GUI interface
EndGameReport	Provides option to print the report or not using a popup window.
Lane	Keeps track of the current lanes and simulates the bowling game. It assigns lanes to parties, computes the scores, designs the functioning of the game(ensures everyone is getting their turn at the right time)
LaneEvent	Holds the values which define a lane like the frame number, current bowler, throw number and all.
LaneStatusView	Creates the view shown in the center of the ControlDesk where things like current bowler in each lane, lanes, pins down and etc are shown. It also provides us with an options to see pins status
LaneView	Creates view for the number of pins down in each throw for each player
NewPatronView	Creates view which lets us input the details about the new patron we are registering
Pinsetter	Simulates the dropping of pins in the lane by updating the states of each pin. Results are randomly generated for each throw
PinsetterView	Creates view which shows the status of pins so we can see what's going on the Lane
Score	Sets the scores for the players in a game
ScoreHistoryFile	Updates the scores in the .DAT file which contains history of scores for each player
ScoreReport	Generates the report for each player at the end of a game and prints/emails it. It includes the current score and previous scores for the given player

Narrative

Weaknesses

Classes weren’t independent, coupling was medium overall. Relatively less reuse of code was observed. Also many methods in classes do multiple tasks and can be split into multiple methods which are much simpler and easy to understand. A few typos were found here and there in variable and method names. Work done by the classes wasn't balanced, for example class Lane did a lot of heavy work whereas some others were just data classes like LaneEvent.

Strengths
Codebase was well structured, by dividing into different files for classes, naming of classes and methods was also done relevant to the action they performed. For dealing with views, classes are created and are separated from classes which perform functionality. Thus it was easier to comprehend and understand what is done by the code because of good naming conventions followed. Also we found use of Observer Design Pattern which was described in a later subsection.

Fidelity to the Design Document
Codebase mostly adhered to the design document attached with the code. Some deviations were present like print-out functionality documented in the design document wasn’t working.

Code Smells Table

Class File	**Code Smell Type**	**Code Smell**
AddPartyView	Code Repetition	Making buttons
AddPartyView	Deprecation	methods show(), hide()
AddPartyView	High Complexity	Many nested if-else conditions in actionPerformed
ControlDesk	Dead Code	Unused method viewScores()
ControlDeskView	Code Repetition	Making buttons
ControlDeskView	Redundancy	assignPanel button not being used
Lane	Large Class	Many methods and variables
Lane	High Complexity	getScore(), recievePinsetterEvent(), run() methods have very high complexity due to if-else conditions
Lane	Redundancy	Methods lanePublish() and publish() were always being used together
LaneStatusView	Redundancy and Complexity	A if condition was repeated thrice in actionPerformed()

LaneEvent	Dead Code	Unused getFrame() and getCurScores()
LaneEvent	Data Class	Only used for storage, and methods are getters only. No methods were operating on data.
NewPatron	Code Repetition	Making buttons
PinsetterView	Code Repetition	Making buttons

Design Pattern

Here we can see the use of the Observer Pattern. It is used when we are interested in the state of an object and we want to get notified whenever a change of state occurs.

We get responses for clicks of buttons. In the code, we wait on thread for a button click by the user, and notify the corresponding code segment which is responsible to give response to the button click and it (notification is done here by changing state here, for example - we change gameIsHalted variable in Lane class to halt the game)

Also Adapter Pattern can be observed here. The adapter design pattern is one of the structural design patterns and it's used so that two unrelated interfaces can work together. The object that joins these unrelated interfaces is called an Adapter.

Here in the initial design, it can be observed that ControlDesk acts as an Adapter. It joins Bowlers, Party and Queue.

Initial UML Class Diagram

- Drawn (Only main classes are drawn, interfaces and views are not shown here)
- Generated (Everything including interfaces, views)

Initial Sequence Diagrams

- AddParty
- ControlDesk
- NewPatron
- Drive
- Lane

Refactoring

Narrative

We have Refactored the initial codebase using various techniques, they are listed below. The decision to take steps was influenced by metrics and also general readability of code. The metrics are listed in the next section where we describe each of the metrics we have used and analyse our results of refactoring by comparison with initial values.

- Methods
    - getScores method has two core sub logics - one to handle Normal Throw, other to handle Two Strike Balls. These two logics by themselves have high cyclomatic complexities and halstead difficulties.
        - Split getScores into three methods - getScores, handleNormalThrow, handleTwoStrikeBalls
    - Thread run of Lane Class has a logic to end the game and is incorporated in the while(true) loop. This can be separated into new method to make the code more readable and also reduce complexity
        - Created endGame method
    - recieveLaneEvent method creates frame for the LaneEvent by using makeFrame(party) method. This is done inside an if condition checking whether frame has to be made for the given LaneEvent.
        - We created a new method by overloading makeFrame(party) with signature makeFrame(LaneEvent) to create frame for a specific LaneEvent.
        - And we have used this method inside recieveLaneEvent method. This makes the code more readable and also decreases complexity of recieveLaneEvent
    - recieveLaneEvent method has logic to set scores. This has been refactored into a new method setScores for improved readability and reduced complexity
        - Created setScores method
    - EndGamePrompt has a method named `distroy` which is a typo. Renamed it to `destroy`
    - publish(lanePublish()) is the way method publish is used always, thus two methods are redundant. And also only it is used only in this class.
        - Incorporated functionality of lanePublish into publish itself and removing lanePublish method
    - Lane's main thread logic writes score to file inside the while(true) loop itself, this can be refactored out into another method which deals with this File Operations. It improves readability of code, reduces complexity
        - Created writeScoreToFile method
    - Removed dead code as mentioned in code smells
- Loops
    - Use proper breaks conditions in while(true) loops instead of unhandled exceptions

    - Remove redundant Iterator and a Index storing its location (pointer scoreIt, myIndex) and use directly iterate over vector using for loop with index in endGame (which was initially this loop was in run method of Lane)
    - Refactored clumsy for loop from 0 to 21 in setScores (which was initially in receiveLaneEvent) by creating a variable as `value` and then using it in conditionals rather than long query from HashMap each time which makes both the code very unreadable and also inefficient.
- Conditionals
    - Removing redundant if conditions
    - Reorganizing conditions to reduce nested if's where ever unnecessary
    - Converted simple nested if's which can be made into single if statement with && operator
    - This was done in many methods, some of the notable places are
        - receivePinsetterEvent
        - setScores
        - handleTwoStrikeBalls
        - handleNormalThrow
        - actionPerformed (LaneStatusView)
        - run (Lane)
- Remove deprecated warnings wherever possible. Some places are
    - Date format string
    - Used Integer.valueOf instead of new Integer(int)
- Class *(Design pattern was used, mentioned in next subsection)*
    - Operations on cumulScores in Lane were observed to be complex
    - The most complex functions are getScores, and their sub methods handleNormalThrow, handleTwoStrikeBalls.
    - Thus cumulScores is decoupled from Lane into a new class CumulLaneScores inorder to reduce overall complexity
    - This class is used by Lane class. Signature is as shown in figure

```
class CumulLaneScores {
      - bowlIndex : int
      - cumulScores : int[][]
      + CumulLaneScores()
      + getCumulScores()
      + getFinalScore()
      + getScore()
      + reset()
      + setBowlIndex()
      + writeScoreToFile()
      - handleNormalThrow()
      - handleTwoStrikeBalls()
  }
```

Design Pattern

Refactored design by creating a new class is following the Command Pattern. The Observer Pattern and Adapter Pattern remain unchanged, they were described in the Design Pattern subsection of Initial previously.

Command Pattern is used to implement loose coupling in a request-response model. In command pattern, the request is sent to the invoker and invoker passes it to the encapsulated command object. Command object passes the request to the appropriate method of Receiver to perform the specific action.

Here in our implementation, command object is currentCumulScores, this can be seen in Lane.java where in Lane class, we use `private CumulLaneScores currentCumulScores;`

Class Responsibility Table

Class	**Responsibility**
….	… (Continued from initial design)
CumulLaneScores	Stores and performs operations on cumulative scores as per the logic of the bowling game, used by Lane Class. This class handles all the scoring mechanism logic in Game.

Refactored UML Class Diagram

- Drawn (Only main classes are drawn, interfaces and views are not shown here)
- Generated (Everything including interfaces, views)

Refactored Sequence Diagrams

- AddParty (Hasn't changed from initial one)
- ControlDesk
- NewPatron (Hasn't changed from initial one)
- Drive
- Lane

Comparison of metrics before and after Refactoring

We have used MetricsReloaded Plugin for Intellij IDEA. Values output by plugin are stored in this sheet. And the analysis is done here.

Class Metrics

- **Average Operation Complexity** - The average cyclomatic complexity of the non-abstract methods in each class. Inherited methods are not counted for purposes of this metric. Cyclomatic complexity is a measure of the number of distinct execution paths through each method. This can also be considered as the minimal number of

tests necessary to completely exercise a method's control flow. In practice, this is 1 + the number of if's, while's, for's, do's, switch cases, catches, conditional expressions, &&'s and ||'s in the method.
    - Values above 3.00 are indicated in red by Intellij indicating problematic
    - After refactoring, none of the classes have value above 3.0
    -

	Initial	Refactored
Mean	1.869156375	1.804176768
Standard Deviation	0.8942867475	0.6930318986
Max	4.235294118	3

    - We can observe from values of mean, standard deviation and max, now the distribution is lot better after Refactoring
- **Weighted Method Complexity** - The total cyclomatic complexity of the methods in each class.
    - Values above 35 are indicated in red by Intellij indicating problematic
    - After refactoring, only one was above 35. And max was brought from 72 to 41
    -

	Initial	Refactored
Mean	11.54166667	10.88
Standard Deviation	14.15129174	9.257069371
Max	72	41

- *For metrics following from here, ideal value ranges weren't found properly. But for all of these metrics, the lower the better. It can be observed from how the standard deviation, mean, max came down (and hence distribution moving towards ideal) in the Refactored one, we can say that refactoring has worked very well in improving the code quality of the codebase.*
- **Coupling Between Objects** - Calculates the number of classes or interfaces which each class is "coupled" with. A class is declared to be coupled with another if it depends on that class or is depended on by that class. Dependencies due to inheritance are not counted for purposes of this metric.
    -

	Initial	Refactored
Mean	4.875	4.84
Standard Deviation	3.530149646	3.543538721
Max	15	15

- **Halstead Difficulty** - The Halstead Difficulty is intended to correspond to the level of

difficulty of understanding a class.

	Initial	Refactored
Mean	43.34662769	42.53210891
Standard Deviation	54.50338747	44.3489924
Max	231.7987013	178.5535714

- **Halstead Effort** - The Halstead Effort is intended to correspond to the level of effort necessary to understand a class.

	Initial	Refactored
Mean	125719.0885	87605.37706
Standard Deviation	331320.5228	155366.989
Max	1548282.845	648454.1418

- **Message Passing Coupling** - Message Passing Coupling is simply defined as the number of method calls in a class. Classes with high Message Passing Coupling may be less stable, and require higher amounts of integration testing.

	Initial	Refactored
Mean	29.66666667	28.12
Standard Deviation	31.92881575	30.26868569
Max	91	89

Method Metrics

- **Essential Cyclomatic Complexity** - Essential complexity is a graph-theoretic measure of just how ill-structured a method's control flow is. Essential complexity ranges from 1 to v(G), the cyclomatic complexity of the method.

	Initial	Refactored
Mean	1.0546875	1.094202899
Standard Deviation	0.4029144562	0.5388238226
Max	5	5

  - ***After refactoring, this is the only metric (including classes) which***

moved a little bit away from ideal range. Rest all moved towards the ideal range.

- **Design Complexity** - The design complexity is related to how interlinked a methods control flow is with calls to other methods. Design complexity ranges from 1 to v(G), the cyclomatic complexity of the method. Design complexity also represents the minimal number of tests necessary to exercise the integration of the method with the methods it calls.
    - Values above 9 are indicated in red by Intellij indicating problematic. After refactoring distribution moved towards the ideal range as can be seen by changes below in the table.
    -

	Initial	Refactored
Mean	1.9375	1.862318841
Standard Deviation	2.380200481	1.85689494
Max	17	11

- **Cyclomatic Complexity** - Cyclomatic complexity is a measure of the number of distinct execution paths through each method. This can also be considered as the minimal number of tests necessary to completely exercise a method's control flow. In practice, this is 1 + the number of if's, while's, for's, do's, switch cases, catches, conditional expressions, &&'s and ||'s in the method.
    - Values above 10 are indicated in red by Intellij indicating problematic. After refactoring distribution moved towards the ideal range as can be seen by significant changes in standard deviation, max below in the table.
    -

	Initial	Refactored
Mean	2.453125	2.275362319
Standard Deviation	4.257202096	2.660245628
Max	38	14

- *For metrics following from here, ideal value ranges weren't found properly. But for all of these metrics, the lower the better. It can be observed from how the standard deviation, mean, max came down (and hence distribution moving towards ideal) in the Refactored one, we can say that refactoring has worked very well in improving the code quality of the codebase.*
- **Halstead Difficulty** - The Halstead Difficulty is intended to correspond to the level of difficulty of understanding a method.
    -

	Initial	Refactored

Mean	7.587995967	7.11709201
Standard Deviation	14.26050198	10.20427599
Max	107.8043478	59.29310345

- **Halstead Effort** - The Halstead Effort is intended to correspond to the level of effort necessary to understand a method.
    -

	Initial	Refactored
Mean	6031.324245	3511.191818
Standard Deviation	26273.30081	10834.29036
Max	249384.2478	87092.3867

- **Quality Criteria Profile (Maintainability)** - This is a synthetic metric, designed to estimate the difficulty of maintenance for a given method. Lower scores are better. Quality Criteria Profile (Maintainability) is defined as: QCP_MAINT = (3*N) + EXEC + CONTROL + NEST + (2*V(g)) + BRANCH
    -

	Initial	Refactored
Mean	109.28125	99.71014493
Standard Deviation	207.0217736	159.445078
Max	1415	862