Bowling TDD Kata

# Scenario

Write code that can score a bowling game using traditional (or possibly made up) scoring rules.

# Goals of this scenario

* Learn basics of setting up a class under test when the test comes first
* Understand how to use an interface as a contract between unit test and implementation class
* Learn how to read unit test projects
* Learn how to run/organize unit tests
* Practice writing just enough code to pass the test
* Practice refactoring as new requirements (new tests) are uncovered
  + Emergent design
* Practice the 'write individually; share as a team' format of these katas
* Learn to share with team members (show best practices, ask for help)
* Experience unit test naming conventions

# Solution/Projects used

* Bowling TDD Kata.sln
  + $/Bowling/Bowling.csproj - Code files (including per-developer folders)
  + $/Bowling.Test/Bowling.Test.csproj – Unit tests

# Instructions to kata facilitator

1. To personalize the code for each member (such as replace ‘Dev01’ with the developer’s initials), the following changes should be made:
   1. $/Bowling
      1. $/Bowling/GameDev01/Game.Dev01.cs file should be renamed to new naming convention.
      2. $/Bowling/GameDev01/Game.Dev01.cs should have the namespace updated to the new naming convention. Note this must match the namespace referenced in the test project (see below).
      3. Folder names should be renamed for clarity.
   2. $/Bowling.Test
      1. GameTest.cs should have the new namespace updated in the GetAllTestCases() method
      2. The label parameter of SetName() should be updated for clarity.
2. Unit tests should not be added to the project file all at one time. This gives the team an unfair advantage to see what the final requirements are at the beginning of a project. That never happens in the real world. Instead, the unit tests should be added one at a time when the team is ready to tackle the next set of tests. If it helps, you can keep the upcoming tests in the GameTest.NextUp.cs class and not commit this to the repository. Then just move tests one at a time over to the GameTest.cs class and commit it to the repository.
3. The exception classes ($/Bowling/\_Exceptions/\*.\*) are not used until the very end of this scenario. The recommendation is to not make these visible to team members until that exercise is active.
4. The entire list of unit tests to be introduced are included in the appendix

# Initial Instructions to developers

1. You should not modify any code in the unit test project.
2. The code project initially has a folder with a single file both named with your assigned ID. You should only be modifying “your” file(s). Feel free to add additional files or organize your files in separate subfolders as you see fit. You should only work within the code project ($/Bowling/Bowling.csproj).
3. Since the same unit tests are used for everyone, to avoid naming conflicts, there are separate namespaces to segregate out each developer’s code. You’ll want to make sure your code files are contained within your namespace. The namespaces are all the same format:  Bowling.[YourID] (e.g. Bowling.Dev01). The unit test files will need to know this namespace in order to function. Everything should be set up correctly now.
4. Any common files used should be stored in the code project file in folders beginning with an underscore. These folders/files should not be modified except by the kata organizer. For example, there is an interface, IGame, that will be used by the unit tests (and therefore should be what you are implementing during this exercise).

# Exercise 1 – Get your code file to compile

1. Make sure you are implementing the IGame interface in your code class
2. Compile the solution to make sure your code compiles and the unit test compiles (don’t worry about passing the unit tests yet).
3. To avoid other’s code bringing down your code, comment out their unit test calls from the $/Bowling.Test/GameTest.cs class, GetAllTestCases() method. You should not commit these changes to the repository.
4. When you are done, commit your $/Bowling/\* code file (and structural) changes to the repository. Once everyone is done, the projects should compile completely.
5. Notify the entire team when you have completed your portion of Exercise 1.

# Exercise 2 – Pass the ‘All gutter balls’ test

Now comes the fun part, passing tests. From this point on, you should start looking at the Bowling.Test.csproj project to see how we are setting up each new requirement as a test. The goal of TDD is not to write code, then write tests that prove your code. Rather, we write tests that validate what we are wanting our system under test to do and via what interface (the design). Then we write the simplest piece of code possible to make the test pass. The code should be simple and straight-forward, maybe even hackish-looking but definitely not over-engineered.

All test classes will use the IGame interface to exercise your code classes. There are two methods you will need to implement:

1. void ThrowABall(int pPins) – Use this to record the results of felled pins from a single ball roll.
2. int TallyScore() – Use this to calculate the score of the bowling game. Intermediate scores will need to be available as well as final score.

For this first coding go-round, forget that you’ll eventually be adding all the rules of bowling scoring and instead focus just on making this one test pass – A bowler throwing 20 gutter balls and ending up with a score of zero.

After you have passed the test, feel free to refactor your code as needed (rename variables, extract out common code into methods, create separate classes, etc.

When you have this code change complete, don’t commit to the repository just yet. This will keep everyone else honest. Once everyone has responded that they have completed the test, then we’ll commit together and review as a team.

# Exercise 3 - Hit the 10 pin every time (20 single pin throws)

For this test, the bowler is now going to throw 20 single-pin balls. No spares, no strikes, no gutters. Your score should be 20. The new unit test is called **NoBonus\_Roll20Singles\_20Score**. You may have engineered enough into your first code release to likely pass this test already. If so, that’s a sign that you’re thinking too far ahead and should scale back your implementation to match what you’re being asked to deliver.

After you have passed the test, feel free to refactor your code as needed (rename variables, extract out common code into methods, create separate classes, etc.). Remember to keep things legible so everyone can quickly read your code.

# Exercise 4 - Spare me the details and get your mind out of the gutter

Two goals for this exercise:

1. Bowler throws a spare in the first frame (5 pins + 5 pins) and then single pin rolls after that
2. Go through your code and clean it up

Cleaning your code could include (but not limited to)

* Remove unnecessary code
* Refactor any methods/classes
* Clarify variable/method names
* Use different techniques based upon what you saw others do/recommend or what you've read on the internet
* Add comments
* Make your code more easily understandable

Remember, you have the unit tests to prove your code still works so don't be afraid to try something you haven't done before in this refactoring process.

# Exercises 5-9 – Tenth frame, strikes and wrap-around scoring. Oh My!

We’re down to the last set of exercises. This round we will introduce some advanced bonus scoring.

Fair warning: The rules are made up and the points don’t matter. Actually, the scoring is a non-standard (i.e. totally made up) rule, not Olympic bowling rules.

The goals for this exercise:

1. Since the bowling alley wants to turn lanes faster, they have decided that any bonus pin scoring in the 10th frame should not incur additional throws. Instead, the additional throws will have already been made. If you score a strike in the 10th frame, the first two throws of the game will be used to calculate your bonus score. Likewise, a spare in the 10th frame will add the very first throw of the game as its bonus.
2. In exercise 5, there is a spare thrown in the 10th frame. Use the scoring rule above to calculate the bonus of that frame (and total for the game).
3. Exercise 6 is all spares, all the time.
4. In exercise 7, we finally get to a strike. If you knock down all 10 pins in one throw, you get a strike (traditional recognition of such). And like traditional bowling, the next two throws are added to that frame as a bonus. Also like traditional bowling, a single throw knocking down all 10 pins closes that frame (no more throws in that frame).
5. In exercise 8, we are scoring a bonus in the 10th frame using the “wrap-around” rule above.
6. In exercise 9, we get to test the perfect game (10 throws of 10 pins each). Of course the wrap-around scoring still results in a 300 score in this scenario.

Please work on one test (in order) at a time. Once you have the test for exercise 5 (and all previous tests) passing, then go on to solve the test for exercise 6.

As last time, be sure to refactor/clean your code:

* Remove unnecessary code
* Refactor any methods/classes
* Clarify variable/method names
* Use different techniques based upon what you saw others do/recommend or what you've read on the internet
* Add comments
* Make your code more easily understandable

Remember, you have the unit tests to prove your code still works so don't be afraid to try something you haven't done before in this refactoring process.

# Final Exercises (12-17) - Hardening and maintaining other's code

For this last exercise, there is no new functionality to add. We’ve implemented all the scoring rules we need and have completed the “happy path” for our codebase. Now it is time to harden the code against edge cases as well as implement contextual feedback to the user of our class.

As might happen in the real world, after having the developed code in the hands of QA testers for a period of time, we have found that the external code that calls into your IGame implementation is not written to the same high standards that you used to write your code. There are business rules that are violated by calling code that we would like to identify. To that end, you are asked to modify the IGame implementation code to better catch these exceptions and notify the calling code about the ‘errors of their ways’ by providing specific details about the erroneous calls made.

Rather than use a generic exception and hope that the string message provided is suitable, we have chosen to implement an agreed-upon exception tree that will allow us to catch specific exception types regardless of which implementation class(es) that are called. You will find those exceptions in the **$/Bowling/\_Exceptions** folder.

There is one base exception class (**BowlingException**) and two inherited exceptions (**IncorrectBallsException** and **IncorrectGameException**). The base exception class should not be used in the exercises, only the inherited classes. There are new unit tests with names beginning with **Error\_** in exercises 12-17 that will ensure the proper exceptions are being thrown.

To make things more challenging, we'll assume that the original author of the IGame code that needs hardening has left on a remote wilderness vacation and is unreachable leaving other team members to maintain his/her code. Names should be assigned or drawn to determine who gets to maintain whose code.

You are free (and encouraged) to refactor code as you like. Remember with shared code ownership this is as much yours as it is theirs. The unit tests are in place to make sure that your changes are acceptable to outside calling code so feel free to use your safety net.

When we are done, we’ll do a code review. Think about the following:

* 1. How much time did it take to get familiar with the code and find where to make modifications?
  2. Were you comfortable with your modifications?
  3. Did you find you needed to do any refactoring to support the exception handling? If so, how easy was the code to modify?
  4. How much time did you spend on these exercises (ranging from very little to a whole lot)?

# Appendix A – Unit tests and exercise introduced

Names of unit tests used throughout the scenario are provided below.

|  |  |
| --- | --- |
| Unit test name | Exercise introduced |
| NoBonus\_Roll20Gutterballs\_ZeroScore | Exercise 2 |
| NoBonus\_Roll20Singles\_20Score | Exercise 3 |
| Spare\_RollASpareFirstFrameThenSingles\_29Score | Exercise 4 |
| Spare\_RollSinglesThenASpareTenthFrame\_29Score | Exercise 5 |
| Spare\_Roll5PinsEachBall\_150Score | Exercise 6 |
| Strike\_RollAStrikeFirstFrameThenSingles\_30Score | Exercise 7 |
| Strike\_RollSinglesThenAStrikeTenthFrame\_30Score | Exercise 8 |
| Strike\_Roll10Strikes\_300Score | Exercise 9 |
| Misc\_RollStepUpDownPattern\_140Score | Exercise 10 |
| Misc\_RollStepDownUpPattern\_Score | Exercise 11 |
| Error\_TryToThrow11PinsIn1Ball\_ExpectBallsException | Exercise 12 |
| Error\_TryToThrow11PinsIn2Balls\_ExpectBallsException | Exercise 13 |
| Error\_TryToThrowNegativePins\_ExpectBallsException | Exercise 14 |
| Error\_TryToThrow21Singles\_ExpectGameException | Exercise 15 |
| Error\_TryToThrowTooFewBallsAndTallyScore\_ExpectGameException | Exercise 16 |
| Error\_TryToThrow11Strikes\_ExpectGameException | Exercise 17 |