Software Development IV – Advanced .NET 420-411-DW
Piano Player (that sounds like a guitar?)

Due: End of day February 17th

This assignment is based on a Nifty Assignment introduced by Kevin Wayne (Princeton University) and refined by Stuart Reges (University of Washington), David Reed (Creighton University), Benedict Brown (University of Pennsylvania), and John DeNiro (U of California, Berkeley). Large segments were copied directly.

Introduction

This assignment is designed to:

- Review interfaces / classes, particularly how they relate in C#
- Use proper encapsulation while creating separate classes
- Use a unit test *framework* to write automated tests of your code

This assignment should be done individually or in teams of two. Please fill in the Google Form https://forms.gle/4YQxJjBBuq4J3N4b8 . If you want a teammate but are having trouble finding one, please indicate this in the form.

See below for GitLab set up help:

Background:

The sounds that we hear in our day-to-day lives travel through the air as waves. When there is a disturbance in the air (like the pluck of a string or a clap of one's hands), the resulting pressure that is produced propagates through the air as a waveform which humans can hear.

A sound wave is an analog signal, which means that over time it is composed of infinitely many values. Obviously, computers have limited space, and therefore cannot store an infinite number of values in memory. In order to represent a sound wave using a computer, we sample it at specific intervals. If you think of a wave as a continuous function f(x), then sampling simply means choosing certain x-values at which to compute the function, and storing the resulting y-values in a list. Therefore, to a computer, a sound wave is just a long list of numbers, where each number is sample of the wave at a certain time.

As you might guess, the more samples we have, the more accurate our sound will be. The **sampling rate** is how many times per second we choose to sample the wave. For CD audio, this value is 44100, which means that 44100 data values are encoded each second, which can be written as 44.1kHz. So we know how general sound is represented, but what about notes? Each note in the modern chromatic scale is associated with a certain **frequency** (measured in Hz). The frequency of a note corresponds to the number of periods of that note's sound wave that occur each second. For example, the 49th key on a modern piano (A) has a frequency of 440 Hz. Similarly, the middle C has a frequency of about 262 Hz.

Hammering a piano wire

Let's take a close look at what happens when we strike a piano wire, since that is what we will be simulating. At first, the wire is highly energized and it vibrates rapidly, creating a fairly complex (meaning rich in harmonics) sound. Gradually, due to friction between the air and the wire, the wire's energy is depleted and the

wave becomes less complex, resulting in a "purer" tone with fewer harmonics. After some amount of time all of the energy from the strike is gone, and the wire stops vibrating.

The Karplus-Strong Algorithm

Now that we have a physical idea of what is happening in a struck wire, how can we model it with a computer? The Karplus-Strong algorithm played a seminal role in the emergence of physically modeled sound synthesis.

A piano wire is simulated as follows:

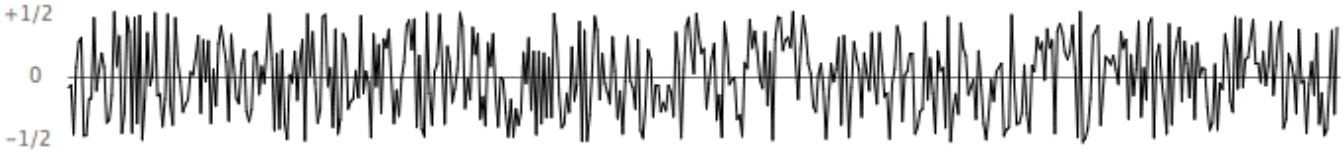
- The frequency of the nth key on a piano is calculated as

$$f_n = 2^{\frac{n-24}{12.0}} \times 440\ Hz$$

  When the wire is initially struck, it contains a sequence of samples full of random noise. The number of samples in the sequence depends on the sampling frequency and the frequency of the note:

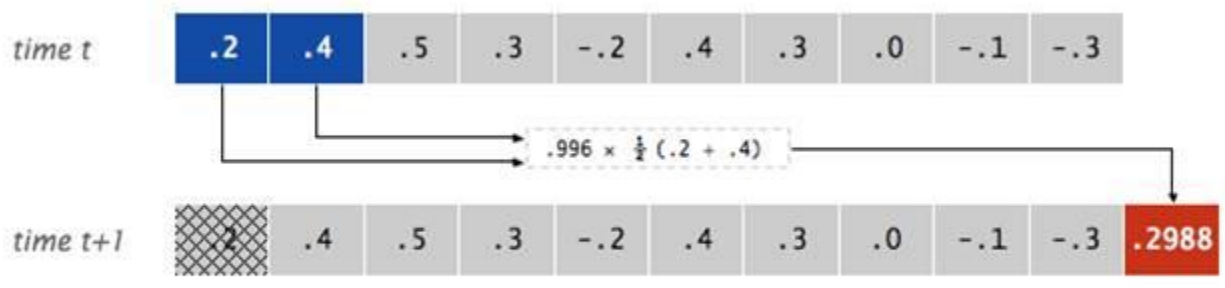
$$N = \frac{sampling\ rate\ (e.g., 44100)}{frequency\ of\ the\ note\ (e.g., 440\ for\ A)}$$

  If the sampling frequency is 44100 Hz/sec, and the frequency of the note is is 440 Hz (for A), then there are 44100/440 or 100 samples (rounded to nearest integer). These 100 samples are initially filled with noise when the wire is struck. Noise is simulated by assigning random real numbers between -0.5 and +0.5.



  The strike then causes a displacement which spreads wave-like over time. The Karplus-Strong algorithm simulates the vibration spreading like a wave: the algorithm repeatedly adds a new displacement sample at the end of the buffer that is the average of the first two samples, scaled by a decay factor, and deletes the first displacement sample from the buffer.

  The diagram below gives a simplified example. For example, at time t=0, the wire is struck. So the N displacement samples (assumes N = 10) are filled with random numbers between -0.5 and +0.5.



the Karplus-Strong update

At time t+1, a sample is added to the end, which is the average of the first two samples multiplied by a decay factor of 0.996. The first sample is removed. This is known as the “feedback mechanism”.

Notice that there are always N displacement measures at every time (the buffer does not change in length).

Why does it work? The two primary components that make the Karplus-Strong algorithm work are the ring buffer feedback mechanism and the averaging operation.

- *The ring buffer feedback mechanism.* The ring buffer, or circular array, models a wire tied down at both ends, in which the energy travels back and forth. The length of the array determines the fundamental frequency of the resulting sound. The feedback mechanism reinforces the fundamental frequency and its harmonics. The energy decay factor (.996 in this case) models the slight dissipation in energy as the wave makes a roundtrip through the string.
- *The averaging operation.* The averaging operation serves as a gentle *low-pass filter* (which removes higher frequencies while allowing lower frequencies to pass, hence the name). Because it is in the path of the feedback, this has the effect of gradually attenuating the higher harmonics while keeping the lower ones, which corresponds closely with how a piano wire sounds.
- *Some intuition.* A more intuitive, but somewhat less precise explanation of the algorithm is the following. When you hammer a piano wire, the middle of the wire bounces up and down wildly. Over time, the tension on the string causes the string to move more regularly and more gently until it finally comes to rest. High frequency wires have greater tension, which causes them to vibrate faster, but also to come to rest more quickly. Low frequency wires are looser, and vibrate longer. If you think about the values in the array as positions over time of a point in the middle of a string, filling the buffer with random values is equivalent to the string bouncing wildly (the pluck). Averaging neighboring samples brings them closer together, which means the changes between neighboring samples becoming smaller and more regular. The decay factor reduces the overall amount the point moves, so that it eventually comes to rest. The final kicker is the array length. Low notes have lower frequencies, and hence longer ring buffers (44,100 / *N* is bigger if *N* is smaller). That means it you will have to go through more random samples before getting to the first round of averaged samples, and so on. The result is it will take more steps for the values in the buffer to become regular and to die out, modeling the longer reverberation time of a low wire.

Project

In this project, you will be simulating a piano. You will read the notes from the keyboard or a file and play the resulting music. Unit testing is a large part of this project.

Getting started

Your teamwork must proceed according to the guidelines we used last semester related to working in individual branches and then using a merge request to merge to a team branch. (Further discussion on this can be found at the document (Protected Branch Workflow at https://gitlab.com/dawsoncollege/git-tutorials/blob/master/protected_branch_workflow.md).

Each teammate should work on one or more temporary feature branches, making small, atomic commits. Each teammate's contribution must be submitted to the team for review in a pull request against the staging branch. Each internal pull request must be reviewed by the other teammate before being merged into your team's staging branch.

The quality of your commits, commit messages, merge requests and code review will all count toward your final grade.

To start, one team member should:

- Go to H:/411/Projects, or whichever directory you will be using for this semester's projects. It is recommended you use a USB key as often the H drive becomes full quickly
- Open a Git Bash window
- configure git if needed: check git config --list to see if your user.name and user.email details from last semester are still good. You may need to update your e-mail address to match the one used to create your GitLab account) (See sample below on how to do this)
- clone your team's repo (you will receive a gitlab message once your instructor creates your repository)
- cd to the Piano folder
- create and checkout a staging branch.

```
git config --list
# set user.name ans user.email if needed:
git config --global user.name "Grace Hopper"
git config --global user.email "gracie@ilovetherangers.ca"
git clone https://gitlab.com/dawsoncollege/CSharp411/.....
cd Piano
git checkout -b staging
```

Copy the .gitignore and .gitattributes files from S:/CompSci/411/gitfiles to your folder (place them at the same level as the .git folder).

One team member will next open Visual Studio and create a new Class Library (.NET Framework) project. DO select the Create directory for solution checkbox but Do NOT select the Create new Git Repository checkbox. Name the project PianoPlayer, browse the location to the git repo folder and save.

Next add the provided files to your project in Visual Studio (IRingBuffer.cs, IMusicalInstrument.cs, Audio.cs, KeyboardPiano.cs).

The audio output uses the NAudio library: we will use Nuget, which is the package manager that comes with Visual Studio. A package manager is a tool which simplifies installing and configuring dependancies, such as outside libraries. Go to:
Tools > NuGet Package Manager > Manage NuGet Packages for Solution…
Search for NAudio
Select v1.9.0, Select your PianoPlayer project on the right panel, and Install

This will create a new folder packages with the NAudio library.
In git, add all files, commit and push:

```
git add .
git commit -m "first commit"
git push origin staging
```

All other teammates can now clone the repository, create feature branches off the staging branch and start working :) IMPORTANT: the packages folder will not be tracked by Git. So after you clone the repo, you will not have the packages folder. No worries, Visual Studio will download it again automatically when you build/run.

CircularArray class and unit tests

Write class CircularArray which implements IRingBuffer. Note that the CircularArray uses an array internally. The constructor of the CircularArray has one parameter, the length of the internal array. **Hint**: use a variable to track where the front of the buffer is at any time: don't move all the elements of the array.

Add a second project, a Test project, in Visual Studio called PianoTests. Write test cases to test your CircularArray class. Be sure to test the edge cases.

PianoWire and unit tests

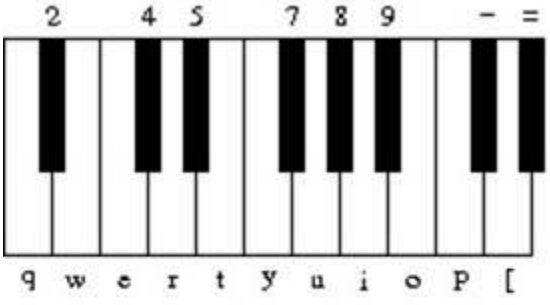
Write class PianoWire which implements IMusicalInstrument. Note that the PianoWire contains a Circular array. The constructor of the PianoWire takes two parameters: the note's frequency and the sampling rate. These parameters are used to create a CircularArray of the appropriate size.

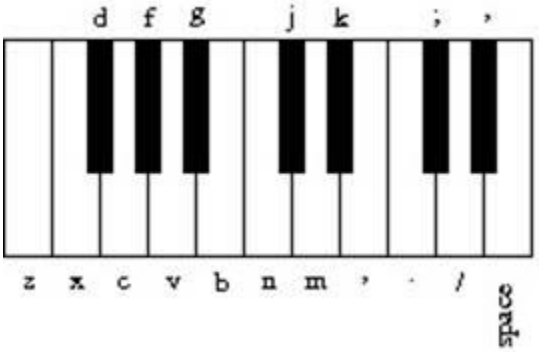
Write test cases to test your Piano Wire class.

Piano class and unit tests

The Piano class implements a simple piano, where different characters correspond to keys (or wires) in a piano. It encapsulates a List<IMusicalInstrument> representing the wires in the piano. Each wire is associated with a key.

The constructor:
`public Piano(string keys = "q2we4r5ty7u8i9op-[=zxdcfvgbnjmk,.;/' ", int samplingRate = 44100)`
takes two parameters. The keys parameter represents the character mapping to the piano keys. The string above maps to these piano keys:





- You will go through every character in the keys string – each one is associated with a note and its frequency is calculated as $2^{\frac{n-24}{12.0}} \times 440\ Hz$ where n is index of the character
- Based on the frequency of the note and the sampling rate, instantiate the PianoWire and add to the List.
- The Piano object has two methods:

  `public void StrikeKey(char key)` Strikes the piano key (wire) corresponding to the specified character. Use the IndexOf method to find the index of the character within the string, which should be the same as the index in the List

  `public double Play()` Goes through all the piano wires and returns the sum of their Sample

Note that the unit tests of the Piano class depend on the unit tests of the previous classes passing.

KeyboardPiano

When your test cases pass, try running the KeyboardPiano class that you have been provided. This class has a main method and will keep waiting for keyboard input. Note that the reaction time may get a bit laggy with more performant computers. There is no code for you to change here.

PlayerPiano

The PlayerPiano class should contain a Main method that reads characters from a file and converts those characters to sounds (using a Piano object). You have been provided the file is called “chopsticks.txt”. Refer to Lab 1 to add the file correctly to the project.

The first line of the file will contain the keys string that defines the character mapping assumed for that file. Your code will need to construct a Piano object using that mapping. The rest of the file will contain strings, one per line, that are to be played in sequence. For example, if a line contained the single character "e", then the wire corresponding to 'e' for that particular piano should be struck. Then call Play on the piano object for 3 times the sampling rate, in order to generate enough samples. Then pause for 400 milliseconds.

If a line contained the sequence "etu", then three wires (corresponding to the characters 'e', 't', and 'u') should be struck one after the other followed by Play invoked 3 * sampling rate, followed by a pause, before going to the next line.

You can simulate a pause with System.Threading.Thread.Sleep(400)

Remember to change your startup object – right click on the project > Properties > Startup object and choose the class with the Main method that you want.

Note on Unit Tests

Your grade for this part will be based on the thoroughness of your tests with some guidelines written below (these will be discussed in class as well)

- Each method you write must have at least one test in it. More generally, each branch in your code should have at least one test in it.
- You should have tests to ensure that your methods throw exceptions when they are supposed to.

Submission

To submit your team's code, open a merge request of your repo's staging branch against master. When creating the merge request:

Set the title of the merge request to Complete

Set your instructor as the assignee

Add a label called grade (you should be able to create a new label if it doesn't exist)

For the over-achievers

Satisfy the over-achiever in you! Here are suggestions on other effects and instruments to synthesize.

- Harp strings: Flipping the sign of the new value before adding it to the end in Sample() will change the sound from guitar-like to harp-like. You may want to play with the decay factors to improve the realism, and adjust the buffer sizes by a factor of two since the natural resonance frequency is cut in half by the change.
- Drums: Flipping the sign of a new value with probability 0.5 before adding it in Sample() will produce a drum sound. A decay factor of 1.0 (no decay) will yield a better sound, and you will need to adjust the set of frequencies used.
- Guitars play each note on one of 6 physical strings. To simulate this you can divide your GuitarString instances into 6 groups, and when a string is plucked, zero out all other strings in that group.
- Pianos come with a damper pedal which can be used to make the strings stationary. You can implement this by, on iterations where a certain key (such as Shift) is held down, changing the decay factor.