CST8132 Lab 3: Guitar Simulation

Due Date
See Blackboard for the Due Date

Purpose
Think about Java Objects and how they communicate. Work with multiple classes and their constructors. Work with examples of the *has-a* relationship, *Association, Aggregation, Composition* between objects.

Description
For this exercise, we will be considering the implementation of a simple guitar simulation. Simulation is the basis of many games, but it also has important industrial applications.

Our goal is to become familiar and comfortable with the notion that Object Oriented Programming is about Objects communicating and interacting. Recall that Java Objects in general have fields (also called attributes, instance variables, or properties) which constitute their state, and methods which constitute their behaviors. These Objects are instances of Java Classes, which have fields (and constructors) to specify the Object's state, and methods to specify the Object's behaviors. When a property of one object is a reference to yet another object, we say the first object is in a *has-a* relationship with the second, and there is an *Association* between them. More specifically, there is an *Aggregation* relationship.

Background
Consider a piece of wire or cord or wire with one end tied or fastened to a wooden board. The other end of the wire is wrapped around a nail or peg embedded in the same board, so the wire is tight. If the peg is somehow turned with fingers or a wrench, the wire wraps further around the peg and the wire becomes tighter. We can simulate this situation with Java code.

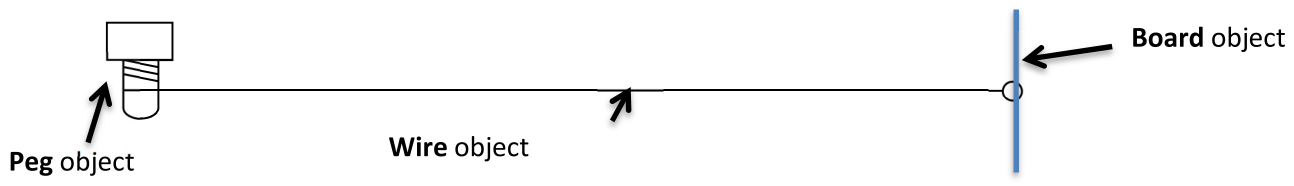
The peg will be represented by a simple Java object that can be turned. Therefore, we have a **Peg** class that has a **turn** method. A **Peg** has a wire wrapped around it. Note the phrase, "has a", in that sentence. A **Peg** *has-a* **Wire**. In other words, a **Peg** has an attribute which is an instance variable that holds a reference to a **Wire** object.

A **Wire** object can tighten. Hence the **Wire** class has a **changeTension** method that increases the **Wire**'s tension attribute by a certain amount (or decreases it if the amount is negative). So we can change the tension of a wire, but what else can we do with it? We can pluck the wire, or in other words, we can tweak it with our finger to make it make sound. So the **Wire** class has a **pluck** method. What else do we know about a **Wire**? The **Wire** is attached to the board. This attachment is an attribute rather than a behavior. To understand why, think of a person who is married to their spouse. We say the person has a spouse, and we speak of their marital status. The behavior or act of having a wedding and getting married is a different concept. Marital status is a property. So we will say the Wire has an attachment to the board, or even better, a **Wire** *has-a* **Board**, the same way we might say a person has a spouse. This means one of the instance variables of the **Wire** class will be a reference to a **Board** object.

Now, it turns out the part of the **Board** where the **Wire** is attached is thin like the membrane of a drum. So when the **Wire** is plucked and it vibrates, the **Board** makes a sound that corresponds to the tension of the **Wire**. The **Board** class has a **soundNote** method that takes a parameter corresponding to the **tension** property of the wire.

What should a **Peg**'s **turn** method do? We need to specify the amount by which the Peg is turned, so the turn method needs a parameter to correspond to that amount, which can be positive or negative. The **Peg**'s **turn** method would contain one line of Java code that simply calls its **Wire**'s **changeTension** method to change the Wire's tension by that amount. We could say that when the Peg's turn method is called, the Peg sends a message to the Wire that it should change its tension (the Peg calls one of the methods of the Wire).

Our simple guitar has (is *Composed* of) six **Peg**, **Wire** pairs (one pair shown below) and a **Board**. The *Composition* relationship is a form of *Aggregation*. *Composition* means *consists-of,* which is a stronger form of *has-a.* So the **Guitar** class has an array of **Wires**, an array of **Pegs**, and a **Board** as attributes.



The Wire vibrates at a certain frequency corresponding to its tension, and when the corresponding Peg is turned, the wire's frequency changes. We will keep things really simple and consider the unit of turning a Peg the same as a unit of tension. For example, one of the behaviors of the Guitar will be that **turn(1,-2)** will mean that the first Peg (that is, Peg **1**, the first argument) is turned just enough in the loosening direction to decrease the wire's tension or vibrating frequency by two units (a decrease of 2, the **-2** which is the second argument). We will call this second argument the change-in-tension, or in other words, delta-tension. For now, our wires can have any tension, but wires with tensions less than zero will be floppy, and when plucked they will sound like they have a tension of zero (no sound).

There are not many lines of code to write. The main work is to figure out what goes where, and which methods call which other methods.

You are given the following items:

1. UML for the objects in this guitar simulation (see below). The constructors (whose parameters are often used to initialize properties of the object being constructed) are not mentioned in the UML so you will need to determine those yourself. Ask for help if you get stuck.
2. Java library **Guitarlib.jar** for the **Board** class, which you will download from blackboard. This will be the mechanism through which our simulation activates the sound card on your computer. You will use this library (your **Wire** object's **vibrate** method will call the **Board** object's **soundNote** method) but you are NOT expected to understand how the **Board** class works. You only need to know the **soundNote** method is there, and you need to use that method at the appropriate place in the code. Follow the instructions from the class to "add" the library into your project.
3. A Java source code file called **SimpleSong.java** containing a main method for testing your guitar simulation. Download this file from Blackboard. When you run this file, you are playing a simple song on your guitar simulation. Feel free to change this file to make different sound patterns, but note that at this point our guitar doesn't yet have frets, so we will be limited in the notes we can produce, although you could **turn** the **Peg** to create different sounds. We will work on removing those limitations in a future assignment.
4. Constant declarations in **Midi** class (also available from **GuitarLib**). You will use these constants into the appropriate place in your **Guitar.java** file. Alternatively, you can use **Midi'**s **GUITAR_TABLE** which is an array of those constants – up to you. These constants are just integers that correspond to the tensions and therefore vibrating frequencies of the six wires.

Your assignment is to implement the simulation by writing Java **Classes, Guitar, Wire and Peg** according to the UML. Format your code with proper indentation and other formatting, using the coding conventions of the first Hybrid Activity for guidance. Test plan and external documentation are not required for this exercise, but in future labs they will be required.

Hints:

1. You will create only *one* **Board** object. One attribute (instance variable) of the **Guitar** class is a reference to the one **Board** object. The constructor of the **Guitar** class will pass the reference of this Board object as a parameter to the **Wire** constructor as each wire is created a total of six times, so that in its constructor, each **Wire** can copy this reference into its **Board** property. Your **Guitar** constructor will create (that is, **new Board()**, **new Wire(...)**, etc, all of the objects the guitar is composed of.
2. You will have only *six* **Wire** objects, and *six* **Peg** objects, all created in the constructor for the Guitar class. The **Wire** objects are created one by one, and each is passed as an argument to the corresponding **Peg** constructor as the pegs are created, so that each **Peg** object in its constructor can set its **Wire** attribute.
3. In **Guitar.java**, the methods take an integer **N** as a parameter, and then those methods need to do something with the **N**th object (that is call a method on the **N**th object). You will probably be using **N** as the index into an array. You will discover that often your methods are implemented with a single method call on another object. The main point of this assignment is for you to think about which objects should invoke which methods on other objects.

Submission

The submission process for this assignment is the same as before:

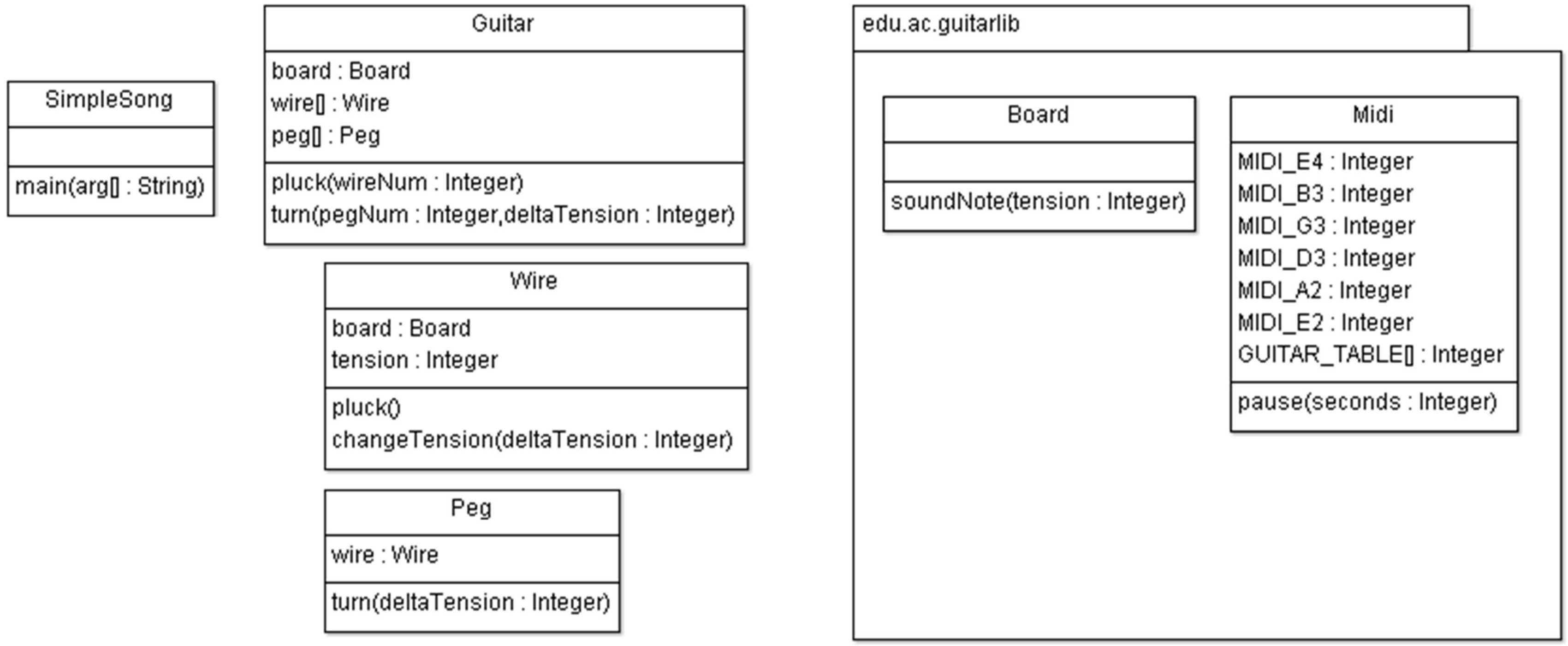
1. demonstrate your simulation to your lab instructor.
2. submit a zip archive of your sources folder (i.e. ‘src’ in Eclipse): Lastname_Firstname_CST8132_Lab3.zip

Grading Scheme

Successful sound demo: 7 marks; Readability: 3 marks

UML (next page)

UML


SimpleSong
main(arg[] : String)
Guitar
board : Board
wire[] : Wire
peg[] : Peg
pluck(wireNum : Integer)
turn(pegNum : Integer,deltaTension : Integer)
Wire
board : Board
tension : Integer
pluck()
changeTension(deltaTension : Integer)
Peg
wire : Wire
turn(deltaTension : Integer)
edu.ac.guitarlib
Board
soundNote(tension : Integer)
Midi
MIDI_E4 : Integer
MIDI_B3 : Integer
MIDI_G3 : Integer
MIDI_D3 : Integer
MIDI_A2 : Integer
MIDI_E2 : Integer
GUITAR_TABLE[] : Integer
pause(seconds : Integer)