Structuring Music

CS1316: Representing Structure and Behavior

Version 3: SongNode and SongPhrase
- SongNode instances will hold pieces (phrases) from SongPhrase.
- SongNode instances will be the nodes in the linked list
  - Each one will know its next.
- Ordering will encode the order in the Part
  - Each one will get appended after the last.

Using SongNode and SongPhrase
Welcome to DrJava.
> import jm.JMC;
> SongNode node1 = new SongNode();
> node1.setPhrase(SongPhrase.riff1());
> SongNode node2 = new SongNode();
> node2.setPhrase(SongPhrase.riff2());
> SongNode node3 = new SongNode();
> node3.setPhrase(SongPhrase.riff1());
> node1.setNext(node2);
> node2.setNext(node3);
> node1.showFromMeOn(JMC.SAX);

All three SongNodes in one Part

How to think about it
Declarations for SongNode

```java
import jm.music.data.*;
import jm.JMC;
import jm.util.*;
import jm.music.tools.*;

public class SongNode {
    /**
     * the next SongNode in the list
     */
    private SongNode next;
    /**
     * the Phrase containing the notes and durations associated with this node
     */
    private Phrase myPhrase;
}
```

Constructor for SongNode

```java
/**
 * When we make a new element, the next part is empty, and ours is a blank new part
 */
public SongNode()
    this.next = null;
    this.myPhrase = new Phrase();
}
```

Setting the phrase

```java
/**
 * setPhrase takes a Phrase and makes it the one for this node
 * @param thisPhrase the phrase for this node
 */
public void setPhrase(Phrase thisPhrase){
    this.myPhrase = thisPhrase;
}
```

Linked list methods

```java
/**
 * Creates a link between the current node and the input node
 * @param nextOne the node to link to
 */
public void setNext(SongNode nextOne){
    this.next = nextOne;
}
```

Using and tracing insertAfter()

```java
> SongNode nodeA = new SongNode();
> SongNode nodeB = new SongNode();
> nodeA.setNext(nodeB);
> SongNode nodeC = new SongNode()
> nodeA.insertAfter(nodeC);
> SongNode oldNext = this.next(); // Save its next
> this.setNext(nextOne); // Insert the copy
> nextOne.setNext(oldNext); // Make the copy point on to the rest
```
Traversing the list

```java
public void showFromMeOn(int instrument) {
    // Make the Score that we'll assemble the elements into
    // We'll set it up with a default time signature and tempo we like
    // (Should probably make it possible to change these -- maybe with inputs?)
    Score myScore = new Score("My Song");
    myScore.setTimeSignature(3,4);
    myScore.setTempo(120.0);
    // Make the Part that we'll assemble things into
    Part myPart = new Part(instrument);
    // Make a new Phrase that will contain the notes from all the phrases
    Phrase collector = new Phrase();
    // Start from this element (this)
    SongNode current = this;
    // While we're not through...
    while (current != null) {
        collector.addNoteList(current.getNotes());
        // Now, move on to the next element
        current = current.next();
    }
    // Now, construct the part and the score.
    myPart.addPhrase(collector);
    myScore.addPart(myPart);
    // At the end, let's see it!
    View.notate(myScore);
}
```

The Core of the Traversal

```java
// Make a new Phrase that will contain the notes from all the phrases
Phrase collector = new Phrase();
// Start from this element (this)
SongNode current = this;
// While we're not through...
while (current != null) {
    collector.addNoteList(current.getNotes());
    // Now, move on to the next element
    current = current.next();
}
```

Then return what you collected

```java
// Now, construct the part and the score.
myPart.addPhrase(collector);
myScore.addPart(myPart);
// At the end, let's see it!
View.notate(myScore);
}
```

getNotes() just pulls the notes back out

```java
/**
 * Accessor for the notes inside the node's phrase
 * @return array of notes and durations inside the phrase
 */
private Note[] getNotes() {
    return this.myPhrase.getNoteArray();
}
```

SongPhrase

- SongPhrase is a collection of static methods.
- We don’t ever need an instance of SongPhrase.
- Instead, we use it to store methods that return phrases.
  - It’s not very object-oriented, but it’s useful here.

SongPhrase.riff11()

```java
import jm.music.data.*;
import jm.JMC;
import jm.util.*;
import jm.music.tools.*;
public class SongPhrase {
    static public Phrase riff11() {
        // Little riff11
        Phrase myPhrase = new Phrase();
        return myPhrase;
    }
}
```
As long as it's a phrase...

- The way that we use SongNote and SongPhrase, any method that returns a phrase is perfectly valid SongPhrase method.

10 Random Notes
(Could be less random...)

```java
public static Phrase random() {
    Phrase ranPhrase = new Phrase();
    Note n = null;
    for (int i = 0; i < 10; i++) {
        n = new Note((int) (128*Math.random()),0.1);
        ranPhrase.addNote(n);
    }
    return ranPhrase;
}
```

10 Slightly Less Random Notes

```java
public static Phrase randomAboveC() {
    Phrase ranPhrase = new Phrase();
    Note n = null;
    for (int i = 0; i < 10; i++) {
        n = new Note((int) (60+(5*Math.random())),0.25);
        ranPhrase.addNote(n);
    }
    return ranPhrase;
}
```

Going beyond connecting nodes

- So far, we’ve just created nodes and connected them up.
- What else can we do?
- Well, music is about repetition and interleaving of themes.
  - Let’s create those abilities for SongNodes.
Repeating a Phrase

Welcome to DrJava.
> SongNode node = new SongNode();
> node.setPhrase(SongPhrase.randomAboveC());
> SongNode node1 = new SongNode();
> node1.setPhrase(SongPhrase.riff1());
> node.repeatNext(node1,10);
> import jm.JMC;
> node.showFromMeOn(JMC.PIANO);

What it looks like

Here’s making a copy

```java
/**
 * @return another song node with the same notes
 */
public SongNode copyNode()
{
    SongNode returnMe = new SongNode();
    returnMe.setPhrase(this.getPhrase());
    return returnMe;
}
```

Repeating

```
/*
 * Repeat the input phrase for the number of times specified.
 * It always appends to the current node, NOT insert.
 * @param nextOne node to be copied in to list
 * @param count number of times to copy it in.
 */
public void repeatNext(SongNode nextOne, int count) {
    SongNode current = this; // Start from here
    SongNode copy; // Where we keep the current copy
    for (int i=1; i <= count; i++)
    {
        copy = nextOne.copyNode(); // Make a copy
        current.setNext(copy); // Set as next current
        current = copy; // Now append to copy
    }
}
```

Step 1:
```
public void repeatNext(SongNode nextOne, int count) {
    SongNode current = this; // Start from here
    SongNode copy; // Where we keep the current copy
    copy = nextOne.copyNode(); // Make a copy
}
```

Step 2:
```
```
Step 3:
current.setNext(copy); // Set as next

Step 4:
current = copy; // Now append to copy

Step 5 & 6:
copy = nextOne.copyNode; // Make a copy
current.setNext(copy); // Set as next

Step 7 (and so on):
current = copy; // Now append to copy

What happens if the node already points to something?

- Consider repeatNext and how it inserts:
  It simply sets the next value.
- What if the node already had a next?
  repeatNext will erase whatever used to come next.
- How can we fix it?

repeatNextInserting

```java
public void repeatNextInserting(SongNode nextOne, int count){
    SongNode current = this; // Start from here
    SongNode copy; // Where we keep the current copy
    for (int i=1; i <= count; i++)
        { 
            copy = nextOne.copyNode(); // Make a copy
            current.insertAfter(copy); // INSERT after current
            current = copy; // Now append to copy
        }
}
```
Weaving

```java
public void weave(SongNode nextOne, int count, int skipAmount)
{
    SongNode current = this; // Start from here
    SongNode copy; // Where we keep the one to be weaved in
    SongNode oldNext; // Need this to insert properly
    int skipped; // Number skipped currently

    for (int i=1; i <= count; i++)
    {
        copy = nextOne.copyNode(); // Make a copy
        //Skip skipAmount nodes
        skipped = 1;
        while (current.next() != null) && (skipped < skipAmount)
        {
            current = current.next();
            skipped++;
        }
        oldNext = current.next(); // Save its next
        current.insertAfter(copy); // Insert the copy after this one
        current = oldNext; // Continue on with the rest
        if (current.next() == null) // Did we actually get to the end early?
            break; // Leave the loop
    }
}
```

Creating a node to weave

```java
> SongNode node2 = new SongNode();
> node2.setPhrase(SongPhrase.riff2());
> node2.showFromMeOn(JMC.PIANO);
```

Doing a weave

```java
> node.weave(node2, 4, 2);
> node.showFromMeOn(JMC.PIANO);
```

Weave Results

```
Before:
```
```
After:
```

Walking the Weave

```java
public void weave(SongNode nextOne, int count, int skipAmount)
{
    SongNode current = this; // Start from here
    SongNode copy; // Where we keep the one to be weaved in
    SongNode oldNext; // Need this to insert properly
    int skipped; // Number skipped currently
```
Then do an insert

```java
if (current.next() == null) // Did we actually get to the end early?
break; // Leave the loop
oldNext = current.next(); // Save its next
current.insertAfter(copy); // Insert the copy after this one
current = oldNext; // Continue on with the rest
```

Version 4: Creating a tree of song parts, each with its own instrument

- SongNode and SongPhrase offer us enormous flexibility in exploring musical patterns.
- But it’s only one part!
- We’ve lost the ability of having different parts starting at different time!
- Let’s get that back.

The Structure We’re Creating

Starting to look like a tree...

Example Song

```java
import jm.music.data.*;
import jm.JMC;
import jm.util.*;
public class MyFirstSong {
  public static void main(String [] args) {
    Song songroot = new Song();
    SongPart part1 = new SongPart(JMC.PIANO, node1);
    songroot.setFirst(part1);
    SongPart part2 = new SongPart(JMC.STEEL_DRUMS, node2);
    songroot.setSecond(part2);
    songroot.show();
  }
}
```