Understanding complex method behaviors: A case study of weave

Introduction to understanding complex method behaviors

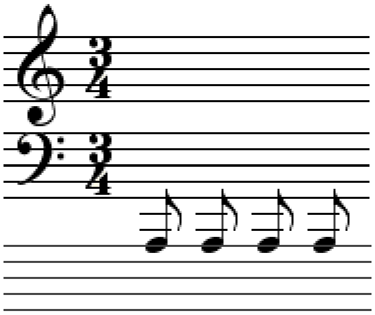
Often in course of your education or even in industry, you will need to figure out what a particular method or function does depending on the current situation. However rather than tediously tracing code line-by-line, sometimes it is easier to determine the method's behavior based on the results of a few well-selected inputs. However, what can actually be considered exemplary inputs and how do we go about choosing them?

A case study of weave

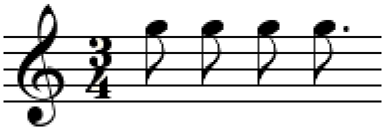
To better understand how to choose your inputs, we will walk through an analysis of the weave method. Below is the actual code for the `weave` method:

```
public void weave(AdvancedSongNode afterThisNode,
                  AdvancedSongNode newNode, int count,
                  int skipAmount) {
   AdvancedSongNode current = afterThisNode;
   for (int i = 0; i < count; i++){
      for (int j = 0; j < skipAmount; j++){
         if (current != null)
            current = current.getNext();
      }
      if(current != null){
         AdvancedSongNode copy = newNode.copy();
         insertAfter(current, copy);
         current = copy;
      }
      else
        break;
   }
}
```

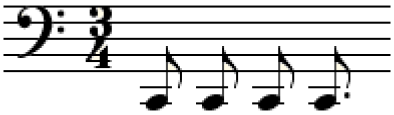
Because `weave` is a linked list method, we know that it somehow operates on a linked list of nodes. Thus it would be a good idea to have distinct nodes within the list so that any change will be clearly reflected.



a1



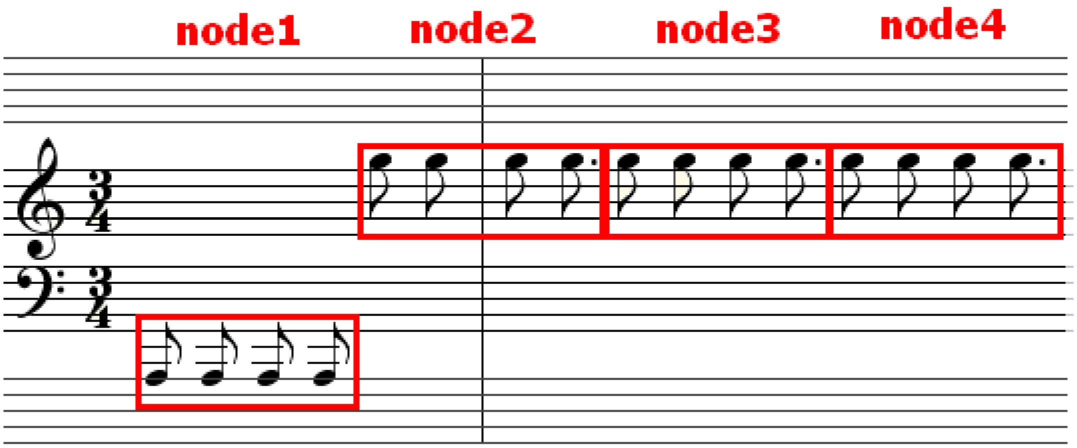
g5



c2

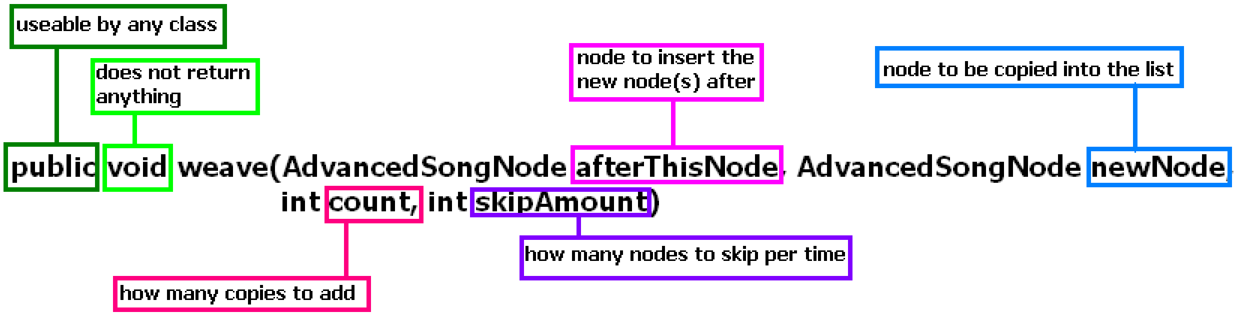
Using these three distinct nodes, we can now create a list to work with. Consider the following list created these following lines of code:

```
AdvancedSongList llist = new AdvancedSongList();
AdvancedSongNode a1 = new AdvancedSongNode(AdvancedSongPhrase.a1());
AdvancedSongNode g5 = new AdvancedSongNode(AdvancedSongPhrase.g5());
AdvancedSongNode c2 = new AdvancedSongNode(AdvancedSongPhrase.c2());
llist.add(a1);
llist.repeatNextInserting(a1, g5, 3);
```



Analyzing the inputs

Now that we have a linked list and some nodes to work with we should analyze the inputs of `weave` to properly test it.



useable by any class

does not return anything

node to insert the new node(s) after

node to be copied into the list

```
public void weave(AdvancedSongNode afterThisNode, AdvancedSongNode newNode,
                  int count, int skipAmount)
```

how many nodes to skip per time

how many copies to add

Cases

We have to choose representative cases so as to establish a pattern of behavior for `weave`. The list is reset back to the original list above after each case.

Case 1: `count = 0` and `skipAmount = 0`

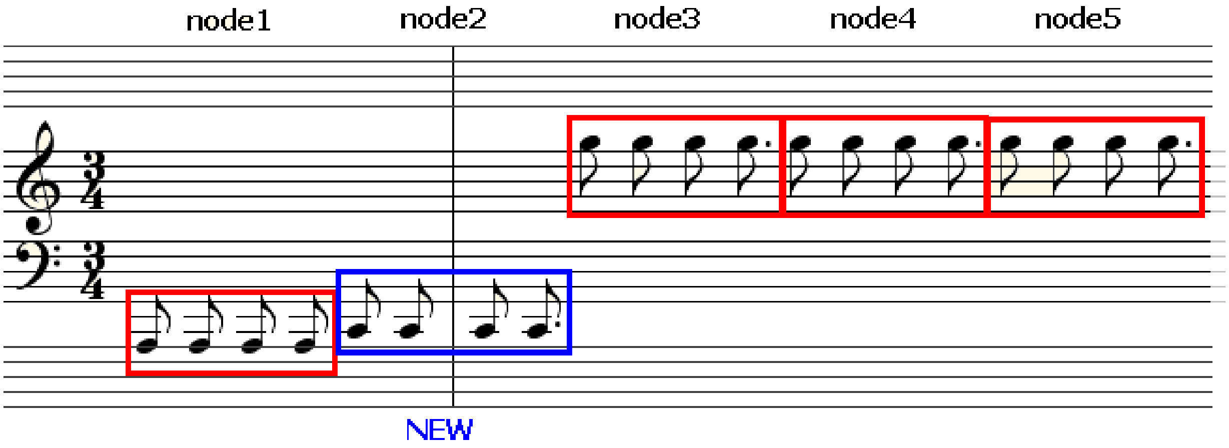
First let us consider what will happen if we did a case when `count` and `skipAmount` equal zero

```
llist.weave(a1, c2, 0, 0);
```

The result is that there is no change in the list. This makes sense because when count equals zero this effectively means that zero new nodes will be mixed into the list.

Case 2: `count = 1` and `skipAmount = 0`
What will happen when `count` equals one and `skipAmount` equals zero?

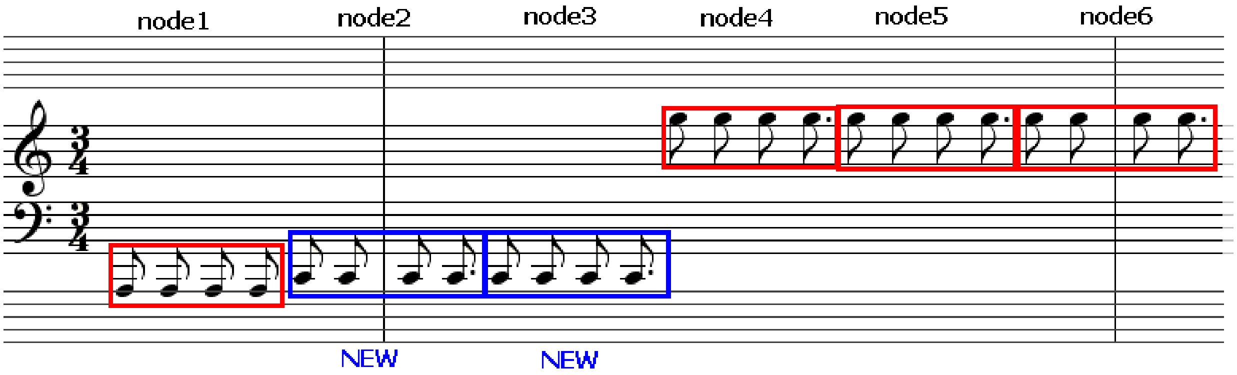
```
llist.weave(a1, c2, 1, 0);
```



We see the result is that one new node is added to the list and no nodes are skipped, but maybe we should see another case before deciding.

Case 3: `count = 2` and `skipAmount = 0`
What will happen when `count` equals two and `skipAmount` equals zero?

```
llist.weave(a1, c2, 2, 0);
```

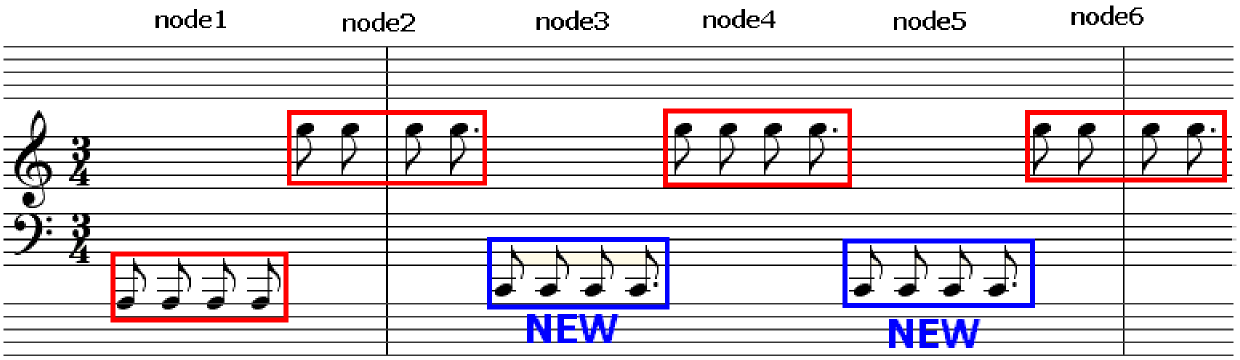


The result is that two new nodes are added to the list and again no nodes are skipped. From the threes case seen above, it is safe to say that `weave` for `skipAmount` equals to zero functions like the `repeatNextInserting` method.

Case 4: `count = 2` and `skipAmount = 1`

Assuming from the previous cases that the `count` parameter only changes the number of new nodes that will be added to the list, what will happen when we start to vary the `skipAmount`?

```
llist.weave(a1, c2, 2, 1);
```

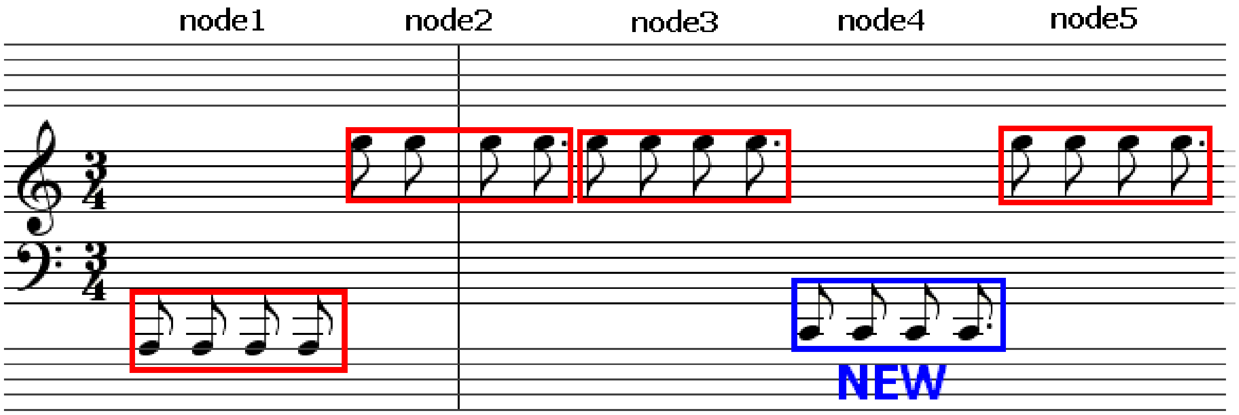


Considering this case of `skipAmount` equal to one, we can conclude that one node is skipped each time meaning that the new nodes are spaced one old node apart.

Case 5: `count = 2`, `skipAmount = 2` and the list is too short

What will happen when `skipAmount` equals to two? From our previous cases, we realize that the list will be too short to accommodate our request for this particular list. What will happen?

```
llist.weave(a1, c2, 2, 2);
```



The method accurately skips two nodes this time, however only added one new node to the list, because the other node's location would lie somewhere beyond the end of the list.

Case 6: `skipAmount` >= size of the list and the list is too short again

This time we will try a `skipAmount` greater than the size of the list and see how the method will react.

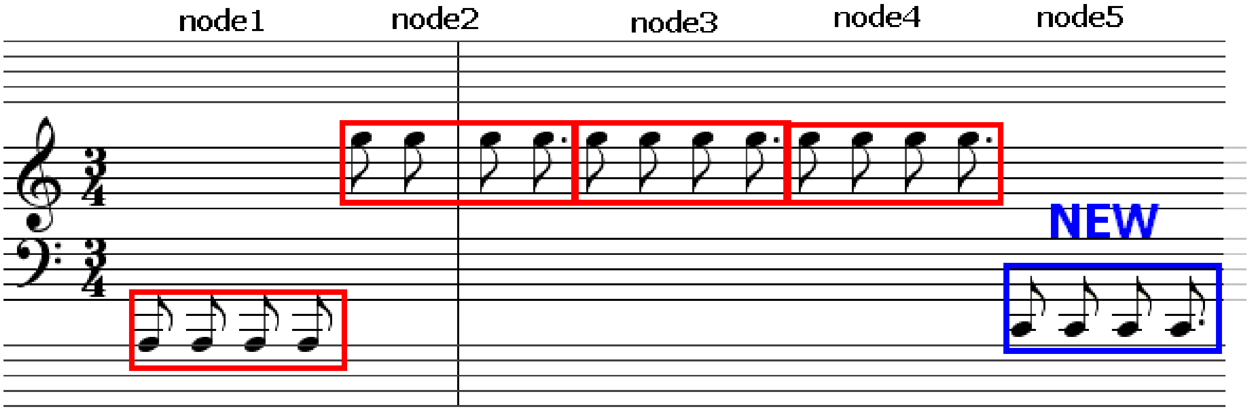
```
        llist.weave(a1, c2, 2, 4);
```

There is no change in the list.

Case 7: the list is just long enough

Based on the previous examples, we know that the `weave` method will not tolerate adding beyond its size, but will it act as expected if the list is just long enough?

```
        llist.weave(a1, c2, 1, 3);
```



The `weave` method seems to work in cases where the list is just long enough.

Final conclusions about the weave method

Based on the various cases tested previously, we can conclude that `weave` will behave like `repeatNextInserting` for cases where `skipAmount` is equal to zero. In the general case, the `weave` will skip the amount of node specified (excluding `afterThisNode`) and mix in the amount of copies specified as long as their next location is not beyond the list.