Iterators in a Nested Class

Aidan Rutherford

The purpose of this exercise was to implement an iterator based on the patterns devised by the Gang of Four: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides inside a nested class. This was done using C++ and Java. Some of the additional concepts exhibited in this exercise aside from iterators include static and non-static attributes and nested classes.

The iterator implemented in C++ was the first implementation. The C++ program used four files. The first was `GoFitertator.h` posted below.

```
#pragma once

class GoFiterator {
public:
    virtual void first() = 0;
    virtual void next() = 0;
    virtual bool isDone() = 0;
    virtual int currentItem() = 0;
};
```

Figure 1. GoFiterator class Implementation

In the class `GoFiterator` contains four virtual functions that make up the GoF iterator. The functions are `first()`, `next()`, `isDone()`, and `currentItem()`. The `= 0` indicates that these functions are purely virtual meaning that there is no actual implementation as all these methods are overridden in subclasses. The next file was the for the `IntegerBuffer` class.

```
#pragma once
#include "GoFiterator.h"

class IntegerBuffer {
private:
    static const int CAPACITY = 32;
    int data[CAPACITY];
    int dataLength;
    class IntegerBufferIterator : public GoFiterator {
    private:
        IntegerBuffer* collection;
        int cursor;
    public:
        inline IntegerBufferIterator(IntegerBuffer* collection)
            : collection(collection), cursor(0) {}
        void first() override;
        void next() override;
        bool isDone() override;
        int currentItem() override;
    };
public:
    inline IntegerBuffer() : data({0}), dataLength(0) {}
    bool add(int value);
    int add(int* values, int numValues);
    GoFiterator* createIterator();
};
```

Figure 2. IntegerBuffer class with nested iterator class implementation

The `IntegerBuffer` class came with three attributes: one for the capacity of the array, one for the array, and one for the number of things in said array. The class contained four methods: one of which added a single value to the array and the other which copied an array of numbers into the member array. There was also an initializer list constructor that just set everything except the capacity to 0. The last method was used to create the iterator. The iterator class was nested inside the `IntegerBuffer` class. Its members included an integer called `cursor` which was used for pointing at a position in the array and the other was `collection` which points to an instance of the `IntegerBuffer` class. Even though the `IntegerBufferIterator` class is nested in the `IntegerBuffer` class. The nested class cannot access members of the base class, so it needed the address of the instance of the base class for which it was created. The last thing to note about the iterator class is that its parent class is the `GoFiterator` class previously discussed above and inside the iterator class, all the methods are overridden apart from the constructor. Next is the implementation of the member functions shown below.

```
#include "IntegerBuffer.h"

bool IntegerBuffer::add(int value) {
    if(dataLength < CAPACITY) {
        data[dataLength] = value;
        ++dataLength;
        return true;
    } else
        return false;
}

int IntegerBuffer::add(int values[], int numValues) {
    int count = 0;
    for(int i = 0; i < numValues; ++i)
        if(add(values[i]))
            ++count;
    return count;
}

GoFiterator* IntegerBuffer::createIterator() {
    return new IntegerBufferIterator(this);
}

void IntegerBuffer::IntegerBufferIterator::first() {
    cursor = 0;
}

void IntegerBuffer::IntegerBufferIterator::next() {
    if(!isDone())
        ++cursor;
}

bool IntegerBuffer::IntegerBufferIterator::isDone() {
    return cursor == collection->dataLength;
}

int IntegerBuffer::IntegerBufferIterator::currentItem() {
    if(isDone())
        return 0;
    else
        return collection->data[cursor];
}
```

Figure 3. `IntegerBuffer` and `IntegerBufferIterator` method implementations

The biggest thing to note is the use of the `collection` pointer as it allows the use of the IntegerBuffer attributes in the iterator class's methods. Below is the driver code for the iterator.

```
#include <iostream>
#include <iomanip>
#include "IntegerBuffer.h"
using namespace std;

int main() {
    const int numberOfValues = 16;
    int values[numberOfValues] = {23, 12, -6, 14, 0, 37, -26, 5, 11,
        -4, 16, 12, 8, -3, 6, -2};
    IntegerBuffer ibuf;
    ibuf.add(values, numberOfValues);
    ibuf.add(values, numberOfValues);
    cout << "C++ Example\n";
    int column = 0;
    GoFiterator* iter = ibuf.createIterator();
    for(iter->first(); !iter->isDone(); iter->next()) {
        if(column >= 10) {
            cout << endl;;
            column = 1;
        } else
            ++column;
        cout << setw(5) << iter->currentItem();
    }
    cout << endl;
    return 0;
}
```

Figure 4. GoF-style iterator test driver code

The driver code shown above added the array twice to fill the array attribute of the `IntegerBuffer` class. The instance of the iterator class was created but in the `for` loop, it points to each of the methods to call the iterator methods like in the line shown below.

```
for(iter->first(); !iter->isDone(); iter->next())
```

The magic of the iterator is that it uses abstraction to hide how some data structure is being traversed. For example, the line above could be used for an array or for a link list, but the developer would not need to worry about choosing the proper functions to call because the program knows which version of the overridden methods to use. An additional, but not necessary feature is to use the `auto` keyword inside the loop. The auto keyword tells the program to infer what type of data is being processed, which could allow the program to work about any datatype. To compile the program was `g++ -std=c++11 IntegerBuffer.cpp test_hw2.cpp -o gof_iterator.exe`. The output for the C++ example is shown below.

```
[ruthera1@prclab1:~/cs332/exercise2/C++]$ gof_iterator.exe
C++ Example
   23   12   -6   14    0   37  -26    5   11   -4
   16   12    8   -3    6   -2   23   12   -6   14
    0   37  -26    5   11   -4   16   12    8   -3
    6   -2
```

Figure 5. C++ GoF-iterator output

The Java iterator example was easier because it did not require a pointer from the nested class. The Java example had two files. The first one shown below is `GoFiterator.java`.

```
abstract class GoFiterator {
    abstract void first();
    abstract void next();
    abstract boolean isDone();
    abstract int currentItem();
}
```

Figure 6. Abstract `GoFiterator` class implementation

This file contained the abstract class for the GoF-style iterator. Like the C++ example it comprised of the same four methods. The `abstract` keyword in Java is the equivalent of `virtual` in C++. The only differences are that the class itself is defined as abstract/virtual which implies that all the methods are abstract/virtual. The other file was the source code which contained the driver, the `IntegerBuffer` class, and the nested iterator class shown below in three separate segments.

```
public class IntegerBuffer {
    int data[];
    int dataLength = 0;

    public IntegerBuffer(int capacity) {
        data = new int[capacity];
    }

    public boolean add(int value) {
        if(dataLength < data.length) {
            data[dataLength] = value;
            ++dataLength;
            return true;
        } else
            return false;
    }

    public int add(int values[]) {
        int count = 0;
        for(int i = 0; i < values.length; ++i)
            if(add(values[i]))
                ++count;
        return count;
    }

    public GoFiterator createIterator() {
        return new GoFiterator();
    }
```

Figure 7. `IntegerBuffer` class implementation with methods

```
public class GoFiterator {
    int cursor = 0;

    public void first() {
        cursor = 0;
    }

    public void next() {
        if(!isDone())
            ++cursor;
    }

    public boolean isDone() {
        return cursor == dataLength;
    }

    public int currentItem() {
        if(isDone())
            return 0;
        else
            return data[cursor];
    }
}
```

Figure 8. `GoFiterator` class implementation with methods

```
public static void main(String args[]) {
    int values[] = {23, 12, -6, 14, 0, 37, -26, 5, 11, -4, 16, 12,
        8, -3, 6, -2};
    IntegerBuffer ibuf = new IntegerBuffer(32);
    ibuf.add(values);
    ibuf.add(values);
    System.out.println("Java");
    int column = 0;
    GoFiterator iter = ibuf.createIterator();
    for(iter.first(); !iter.isDone(); iter.next()) {
        if(column >= 10) {
            System.out.println();
            column = 1;
        } else
            ++column;
        System.out.format("%5d", iter.currentItem());
    }
    System.out.println();
}
```

Figure 9. Driver code implemented inside the `IntegerBuffer` class

The Java code is identical in terms of tasks to the C++ code except for all the extra Java notation. The Java code was compiled with `javac GoFiterator.java IntegerBuffer.java` and was run with `java IntegerBuffer`. After compilation, a `.class` file is created for each class, but in the case of a nested class it created the file `IntegerBuffer$GoFiterator.class`. The output is shown below.

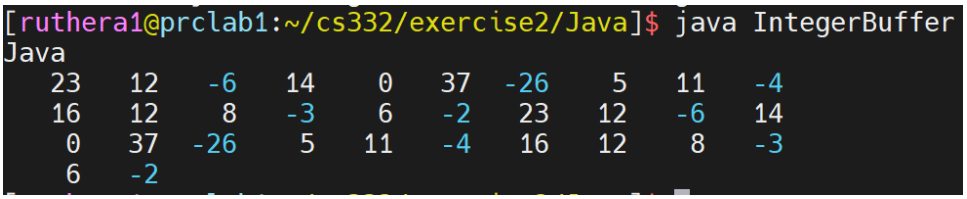

Figure 10. Java GoF-iterator output

The purpose of this exercise was to implement GoF-style iterators in C++ and Java. Iterators use information hiding and encapsulation to hide from the developer what is really going on in the background making robust code very simple to write. At the end it turned out that the Java implementation was a bit simpler than the C++ version, which required and extra pointer to work. Other subjects discussed in this project were nested classes and static and non-static attributes.