Swinburne University Of Technology

Faculty of Information and Communication Technologies

ASSIGNMENT COVER SHEET

Subject Code:	HIT3303
Subject Title:	Data Structures & Patterns
Assignment number and title:	4 – Lists, Iterators, and Design Patterns
Due date:	**April 13, 2011, 10:30 a.m., on paper**
Lecturer:	Dr. Markus Lumpe

Your name:____________________

Marker's comments:

Problem	Marks	Obtained
1	25	
2	86	
Total	111	

Extension certification:

This assignment has been given an extension and is now due on ___________

Signature of Convener:________________________

Problem Set 4: Lists, Iterators, and Design Patterns

Preliminaries

Study or review the following concepts:

1. C++ templates
2. What is difference between value semantics and reference semantics?
3. What is a constant reference?
4. What is a constant object?
5. What is an enumeration type?
6. What is a `typedef` declaration?
7. What is `delete` and how does it work?
8. When do we need destructors?
9. What is an iterator?
10. What is a state machine?

Problem 1:

Define a double-linked list that satisfies the following template class specification:

```
template<class DataType>
class DoubleLinkedNode
{
public:
  typedef DoubleLinkedNode<DataType> Node;

private:
  const DataType* fValue;
  Node* fNext;
  Node* fPrevious;

  DoubleLinkedNode(): fValue((const DataType*)0)
  {
    fNext = (Node*)0;
    fPrevious = (Node*)0;
  }

public:
  static Node NIL;

  DoubleLinkedNode( const DataType& aValue );

  void insertNode( Node& aNode );
  void dropNode();

  const DataType& getValue() const;
  Node& getNext() const;
  Node& getPrevious() const;
};

template<class DataType>
DoubleLinkedNode<DataType> DoubleLinkedNode<DataType>:: NIL;
```

The template class `DoubleLinkedNode` defines the structure of a double-linked list. It uses two pointers: `fNext` and `fPrevious` to connect two adjacent list elements. The constructor takes a constant reference `aValue` as argument and returns a properly initialized list node in which both links are set to the address of *NIL* – the empty list. The methods `getValue`, `getNext`, and `getPrevious` define simple read-only getter functions for the corresponding fields of a `DoubleLinkedNode` object.

The method `insertNode` and `dropNode` build the heart of the class `DoubleLinkedNode`. The method `insertNode` injects the argument `aNode` into the list by making `aNode` the `fNext` node of `this`. The method `dropNode`, on the other hand, removes `this` from the list. That is, `dropNode` has to properly link the remaining list nodes adjacent to `this`.

There is, however, one complication. Template classes are "class blueprints" or, better, abstractions over classes. Before we can use template classes, we have to instantiate them. But to work correctly, the instantiation process requires the complete implementation of the class (see lecture notes page 181). For this reason, when defining template classes, the implementation has to be included in the header file. There are two ways to accomplish this:

- Implement the member functions directly in the class specification (like it is done in Java or C#).
- Implement the member functions outside the class specification but within the same header file.

If you follow this scheme, working with templates is pretty straightforward.

Implement class `DoubleLinkedNode`.

Test harness 1:

```
void testDoubleLinkedNodes()
{
  string s1( "One" );
  string s2( "Two" );
  string s3( "Three" );

  typedef DoubleLinkedNode<string>::Node StringNode;

  StringNode n1( s1 );
  StringNode n2( s2 );
  StringNode n3( s3 );

  n1.insertNode( n3 );
  n1.insertNode( n2 );

  cout << "Three elements:" << endl;

  for ( StringNode* pn = &n1; pn != &StringNode::NIL; pn = &pn->getNext() )
  {
    cout << "(";
    if ( &pn->getPrevious() != &StringNode::NIL )
      cout << pn->getPrevious().getValue();
    else
      cout << "<NULL>";

    cout << "," << pn->getValue()<< ",";

    if ( &pn->getNext() != &StringNode::NIL )
      cout << pn->getNext().getValue();
    else
      cout << "<NULL>";

    cout << ")" << endl;
  }

  n1.getNext().dropNode();

  cout << "Two elements:" << endl;

  for ( StringNode* pn = &n1; pn != &StringNode::NIL; pn = &pn->getNext() )
  {
    cout << "(";
    if ( &pn->getPrevious() != &StringNode::NIL )
      cout << pn->getPrevious().getValue();
    else
      cout << "<NULL>";

    cout << "," << pn->getValue() << ",";

    if ( &pn->getNext() != &StringNode::NIL )
      cout << pn->getNext().getValue();
    else
      cout << "<NULL>";

    cout << ")" << endl;
  }
}
```

Result:

```
Three elements:
(<NULL>,One,Two)
(One,Two,Three)
(Two,Three,<NULL>)
Two elements:
(<NULL>,One,Three)
(One,Three,<NULL>)
```

Problem 2:

Define a bi-directional list iterator for double-linked lists that satisfies the following template class specification:

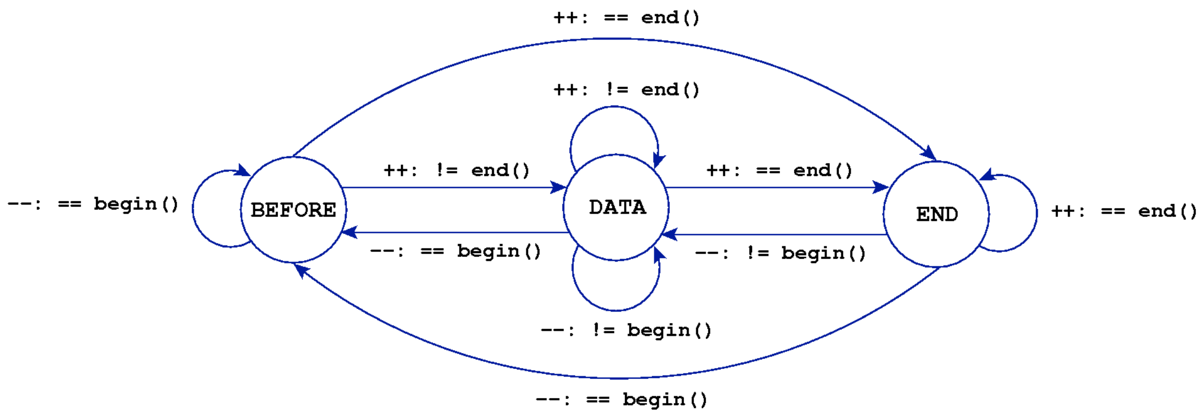
```
#include "DoubleLinkedNode.h"

template<class DataType>
class NodeIterator
{
private:
  enum IteratorStates { BEFORE, DATA , END };

  IteratorStates fState;

  typedef DoubleLinkedNode<DataType> Node;

  const Node* fLeftmost;
  const Node* fRightmost;
  const Node* fCurrent;

public:
  typedef NodeIterator<DataType> Iterator;

  NodeIterator( const Node& aList );

  const DataType& operator*() const;        // dereference

  Iterator& operator++();                   // prefix increment
  Iterator operator++(int);                 // postfix increment
  Iterator& operator--();                   // prefix decrement
  Iterator operator--(int);                 // postfix decrement

  bool operator==( const Iterator& aOtherIter ) const;
  bool operator!=( const Iterator& aOtherIter ) const;

  Iterator begin() const;
  Iterator end() const;
};
```

The bi-directional list iterator implements the standard operators for iterators: dereference to access the current iterator element, both versions of increment to advance the iterator to the next element, and both versions of decrement to go backwards. The list iterator also defines the equivalence predicates and the two factory methods `begin()` and `end()`. The method `begin()` returns a new list iterator positioned before the first element of the double-linked list, whereas `end()` returns a new list iterator that is positioned after the last element of the double-linked list.

Implement the list iterator. Please note that the constructor of the list iterator has to properly set `fLeftmost`, `fRightmost`, and `fCurrent`. In particular, the constructor has to position the iterator on the first element of the list.

An iterator must not change the underlying collection. However, in the case of `NodeIterator` we need a special marker to denote, whether the iterator is "before" the first list element or "after" the last list element. Since we cannot change the underlying list, we need to add "state" to the iterator. Using the iterator state (i.e., `fState`) we can now clearly mark when the iterator is before the first element, within the first and the last element, or after the last element.

To guarantee to correct behavior of the `NodeIterator`, it must implement a "state machine" with three states: `BEFORE`, `DATA`, `END`. The following state transition diagram illustrates, how `NodeIterator` works:



All increment and decrement operators have to test, whether the iterator is still positioned within the collection. In this case the current iterator is different from both `begin()` and `end()`. If the iterator is positioned before the first element, then it is equivalent to `begin()`. If the iterator is positioned past the last element, then it is equivalent to `end()`. Please note that the iterator can in one step become equivalent to `begin()` or `end()`.

Implement class `NodeIterator`.

Test harness 2:

```
void testListIterator()
{
  typedef DoubleLinkedNode<int>::Node IntNode;

  IntNode n1( 1 );
  IntNode n2( 2 );
  IntNode n3( 3 );
  IntNode n4( 4 );
  IntNode n5( 5 );
  IntNode n6( 6 );

  n1.insertNode( n6 );
  n1.insertNode( n5 );
  n1.insertNode( n4 );
  n1.insertNode( n3 );
  n1.insertNode( n2 );

  NodeIterator<int> iter( n1 );

  cout << "Forward iteration I:" << endl;
  for ( ; iter != iter.end(); iter++ )
    cout << *iter << endl;

  cout << "Backward iteration I:" << endl;
  for ( iter--; iter != iter.begin(); iter-- )
    cout << *iter << endl;

  cout << "Forward iteration II:" << endl;
  for ( ++iter; iter != iter.end(); ++iter )
    cout << *iter << endl;

  cout << "Backward iteration II:" << endl;
  for ( --iter; iter != iter.begin(); --iter )
    cout << *iter << endl;
}
```

Result:

```
Forward iteration I:
1
2
3
4
5
6
Backward iteration I:
6
5
4
3
2
1
Forward iteration II:
1
2
3
4
5
6
Backward iteration II:
6
5
4
3
2
1
```

Submission deadline: Wednesday, April 13, 2011, 10:30 a.m.,

Submission procedure: on paper.