Swinburne University Of Technology

Faculty of Information and Communication Technologies

ASSIGNMENT COVER SHEET

Subject Code: HIT3303/8303
Subject Title: Data Structures & Patterns
Assignment number and title: 4, Unit Testing – Find the Bug
Due date: May 5, 2009, 14:30 am
Lecturer: Dr. Markus Lumpe

Your name:_____

Marker's comments:

Problem	Marks	Obtained
1	10	
Total	10	

Extension certification:

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

Signature of Convener:_____

Problem Set 3: Unit Testing – Find the Bug

Preliminaries

Study or review the following concepts:

1. What is unit testing?
2. What is a reference?
3. What is the purpose of constant references?
4. How is the C-stack organized?
5. What stack model does C++ use by default?
6. What is the purpose of a constructor?
7. When do we need to define a destructor?
8. What is the guarantee of a destructor?

Problem 1

Consider the following code. Somewhere there is a bug that will cause a runtime exception either `Segmentation fault` or `Bus error`. This is a serious problem and need to be fixed immediately. Apply unit testing to locate the problem and devise a solution. There is **only** one bug in the code. Unfortunately, it is in a very subtle place and caused by local stack copies of values for which we build constant references. This assignment requires code comprehension and debugging code, two elements of professional software development.

Answer the following questions:

1. Where is the problem?
2. What is a proper problem solution (i.e., bug fix)?
3. What causes the problem?

You only need to submit your answers. No code is required.

The faulty code

Node.h:

```
#ifndef NODE_H_
#define NODE_H_

template<class DataType>
class Node
{
public:
  const DataType& fValue;
  Node<DataType>* fNext;
  Node<DataType>* fPrevious;

  Node( const DataType& aValue,
        Node<DataType>* aNext = (Node<DataType>*)0,
        Node<DataType>* aPrevious = (Node<DataType>*)0 ) : fValue( aValue )
  {
    fNext = aNext;
    if ( fNext != (Node<DataType>*)0 )
      fNext->fPrevious = this;

    fPrevious = aPrevious;
    if ( fPrevious != (Node<DataType>*)0 )
      fPrevious->fNext = this;
  }

  ~Node()
  {
    if ( fNext != (Node<DataType>*)0 )
      fNext->fPrevious = fPrevious;

    if ( fPrevious != (Node<DataType>*)0 )
      fPrevious->fNext = fNext;
  }
};


#endif
```

NodeIterator.h:

```
#ifndef NODEITERATOR_H_
#define NODEITERATOR_H_

#include "Node.h"

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

  const Node<DataType>* fTop;
  const Node<DataType>* fLast;
  const Node<DataType>* fCurrent;
  IteratorStates fState;

public:
  NodeIterator( Node<DataType>* aList )
  {
    fTop = aList;

    // set current to top element;
    fCurrent = fTop;

    // set fLAST
    if ( aList )
    {
      fLast = aList;
      while ( fLast->fNext != (Node<DataType> *)0 )
        fLast = fLast->fNext;
    }
    else
      fLast = (Node<DataType> *)0;

    // set state
    fState = fCurrent ? DATA : END;
  }

  DataType operator*() const
  {
    return fCurrent->fValue;
  }

  NodeIterator& operator++()  // prefix increment
  {
    if ( fState == DATA )
    {
      fCurrent = fCurrent->fNext;

      // end of list?
      if ( fCurrent == (Node<DataType>*)0 )
          fState = END;
    }

    return *this;
  }
```

```
  NodeIterator operator++(int)      // postfix increment
  {
    NodeIterator<DataType> Result = *this;
    ++(*this);
    return Result;
  }

  bool operator==( const NodeIterator& aOtherIter ) const
  {
    return (fCurrent == aOtherIter.fCurrent) &&
           (fTop == aOtherIter.fTop) &&
           (fState == aOtherIter.fState);
  }

  bool operator!=( const NodeIterator& aOtherIter ) const
  {
    return !(*this == aOtherIter);
  }

  NodeIterator end()
  {
    NodeIterator<DataType> Result = *this;
    Result.fCurrent = (Node<DataType>*)0;
    Result.fState = END;

    return Result;
  }

};

#endif
```

List.h:

```
#ifndef LIST_H_
#define LIST_H_

template<class ElementType>
class List
{
private:

#include "NodeIterator.h"

  Node<ElementType>* fTop;
  Node<ElementType>* fLast;

public:

  typedef NodeIterator<ElementType> ListIterator;

  List()
  {
    fTop = (Node<ElementType>*)0;
    fLast = (Node<ElementType>*)0;
  }

  List( const List& aOtherList )
  {
    fTop = (Node<ElementType>*)0;
    fLast = (Node<ElementType>*)0;

    for ( ListIterator iter = aOtherList.GetIterator();
                                    iter != iter.end(); iter++ )
      Add( *iter );
  }

  ~List()
  {
    while ( fTop )
      DeleteLast();
  }

  void Add( const ElementType& aElement )
  {
    Node<ElementType>* lNewNode =
            new Node<ElementType>( aElement,
                                   (Node<ElementType>*)0, fLast );

    if ( fTop == (Node<ElementType>*)0 )
      fTop = lNewNode;

   fLast = lNewNode;
  }
```

```
  void DeleteLast()
  {
    if ( fLast != (Node<ElementType>*)0 )
    {
      Node<ElementType>* lElem = fLast;
      fLast = fLast->fPrevious;

      if ( fLast == (Node<ElementType>*)0 )
        fTop = (Node<ElementType>*)0;

      delete lElem;
    }
  }

  ListIterator GetIterator() const
  {
    return ListIterator( fTop );
  }
};

#endif
```

main.cpp:

```
#include "List.h"

#include <iostream>
#include <string>

using namespace std;

int main()
{
  List<string> l;
  string s1( "One" );
  string s2( "Two" );
  string s3( "Three" );
  string s4( "Four" );

  l.Add( s1 );
  l.Add( s2 );
  l.Add( s3 );
  l.Add( s4 );

  cout << "The list:" << endl;

  for ( List<string>::ListIterator iter = l.GetIterator();
                                   iter != iter.end(); iter++ )
    cout << *iter << endl;

  List<string> copy = l;

  cout << "The copy:" << endl;

  for ( List<string>::ListIterator iter = copy.GetIterator();
                                   iter != iter.end(); iter++ )
    cout << *iter << endl;

  return 0;
}
```

Submission deadline: Tuesday, May 5, 2009, 14:30 a.m.

Submission procedure: on paper in class.