Problem Set 1: Introduction to Visual Studio .NET & C#

Problem 1
Create a new Visual Studio .NET Windows Application project “HelloWorld”:

Implement the Windows application “HelloWorld.exe” that, when executed, looks like the following:

This application contains three .NET elements: a Form, a Label, and a Button. The Visual Studio .NET development environment initializes these three elements some standard values. In order to get the same look and feel, you need to change the following values using the “Properties window”:

- **Form**
  - FormBorderStyle = FixedToolWindow
  - Size = 430, 166
  - Text = Hello World!

- **Label**
  - AutoSize = True
  - Font = Tacoma, bold, 48pt
  - ForeColor = Blue
  - Dock = Top
  - Text = Hello World!
  - TextAlign = MiddleCenter
• Button
  o Text = &Close
  o Dock = Fill
  o ForeColor = Aqua
  o Font = Tacoma, bold, 20pt
  o Click event = method that calls Close()

Double-clicking at the button can set the button's Click event. The IDE will switch to the code view, which will enable you to specify C# code.

Problem 2
A palindrome is a string that is spelled the same way forward and backward. For example, the string “radar” is a palindrome. However, we can also build complete sentences that form a palindrome: “Able was I, I saw Elba!”

Implement a Windows application “CheckPalindrome.exe” that, when executed, looks like the following:

In order to check, whether a given string is a palindrome or not, you need to compare the given string with its reverse. However, a given string may contain whitespace and punctuation characters that need to be excluded from the test. Furthermore, upper-case and lower-case characters are not distinguished. Therefore, “Radar”, “radar”, or “RaDaR” are all palindromes. To compare string you have to use the “Equals” method. Check the documentation to find a way to build iteratively a reverse string from a given string.

The check should be implemented in the Button's Click event handler.
The object “MessageBox” can be used to display a message box that can contain text, buttons, and symbols that inform and instruct the user. Use the MessageBox’s method “Show” to display the result (note, the operator + can be used for string concatenation).

**Problem 3**

Legend has it that in temple in the Far East, priests were attempting to move a stack of disks from one peg to another. The initial stack has a user-defined disks threaded onto one peg and arranged from bottom to top by decreasing size. The priests are attempting to move the stack from this peg to a second peg under the constraints that exactly one disk is moved at a time and at no time may a larger disk be placed above a smaller disk. A third peg is available for temporarily holding disks.

Solve the problem using recursion. That is, moving $n$ disks can be viewed in terms of moving only $n - 1$ disks (and hence the recursion) as follows:

1. Move $n - 1$ disks from peg 1 to peg 2, using peg 3 as temporary holding area.
2. Move the last disk (the largest) from peg 1 to peg 3.
3. Move $n - 1$ disks from peg 2 to peg 3, using peg 1 as a temporary holding area.

The process ends when the last task involves moving $n = 1$ (i.e., the base case). This task is accomplished by simply moving the disk, without the need for a temporary holding area.

Implement a .NET class library called “TowersOfHanoiLib” (Windows file name “TowersOfHanoiLib.dll”) using a fully object-oriented approach. That is, define the namespace “TowersOfHanoiLib” and within this namespace three classes: “Disk”, “Tower”, and “Priest”. These classed have to provide the following public member functions:

```csharp
namespace TowersOfHanoiLib {
    public class Disk {
        // public constructor
        public Disk( int aSize );

        // read-only property that returns the size of the disk.
        public int Size { get; };

        // returns <0 if this disk is larger
        // returns  0 if both disks have the same size
        // returns >0 if the other disk (aDisk) is the larger
        public int CompareTo( Disk aDisk )
    }
}
```
public class Tower {

    // public constructor
    public Tower( string aName );

    // read-only property that returns the name of the tower
    public string Name { get; };

    // read-only property that returns the number of disks
    // stored here.
    public int Count { get; };

    // place a disk on top of all other disks. If “aDisk” is
    // larger than the top-level disk on the tower, write an
    // error message on the screen (i.e., Console).
    public void Add( Disk aDisk );

    // remove a disk from the top of the tower
    public Disk Remove();
}

public class Priest {

    // The method MoveDisks implements the above algorithm.
    // The parameter aMoves is used to store strings, which
    // represent the precise instructions that it will take
    // to move the disks from the starting peg to the
    // destination peg.
    public void MoveDisks( ArrayList aMoves,
                          Tower aStart,
                          Tower aTarget,
                          Tower aTemp )
    }

You may find the classes Stack and ArrayList useful, which are defined in the namespace
“System.Collections”.
The strings that represent the moves should have the following format:

“<source> → <destination>”

For example, to move a stack of three disks from peg 1 to peg 3, your program should
generate the following series of move strings:

“1 --> 3”
“1 --> 2”
“3 --> 2”
“1 --> 3”
“2 --> 1”
“2 --> 3”
“1 --> 3”
Problem 4

Implement the console application “TowersOfHanoi.exe” that uses the class library “TowersOfHanoiLib.dll”. In the Main function, create three Tower objects and store an initial number of disks on the first tower by creating the appropriate disk objects and repeatedly calling the first Tower’s Add() method. The application should request a user input (using the methods Console.ReadLine() and Int32.Parse()) to set the initial number of disks. Furthermore, create a Priest object and call the method “MoveDisks”. The first argument is an ArrayList object. After the call of “MoveDisks” the ArrayList object will contain a list of strings that represent the performed moves. Print all strings contained in the ArrayList object using the “foreach” statement. The console application “TowersOfHanoi.exe” should produce the following output:

Add the line

    Console.Read();

as last statement to the Main function. You have to press any key to terminate the application. This might be useful, when debugging the program.
### Problem 5

Around 1550 Blaise de Vigenère, a French diplomat from the court of Henry III of France, developed a new scrambling technique that uses 26 alphabets to cipher a text. The **Vigenère Cipher** is a polyalphabetic substitution technique based on the following *tableau*:

| Key\Letter | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| A           | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A |
| B           | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B |
| C           | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C |
| D           | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D |
| F           | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F |
| G           | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G |
| H           | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H |
| J           | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J |
| K           | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K |
| L           | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L |
| M           | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M |
| N           | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N |
| O           | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| P           | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P |
| Q           | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q |
| S           | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S |
| T           | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T |
| U           | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U |
| V           | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V |
| W           | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W |
| Y           | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y |
| Z           | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |

The Vigenère cipher uses this table together with a keyword to encode a message. For example, suppose we wish to scramble the following message:

**TO BE OR NOT TO BE THAT IS THE QUESTION**

using the keyword **RELATIONS**. We begin by writing the keyword, repeated as many times as necessary, above the message. To derive the encoded text using the tableau, for each letter in the message, one finds the intersection of the row given by the corresponding keyword letter and the column given by the message letter itself to pick out the encoded letter.

**Keyword:**

RE LA TI ONS RE LA TION SR ELA TIONSREL

**Message:**

TO BE OR NOT TO BE THAT IS THE QUESTION

**Scrambled Message:**

LT NF IA CCM LT NF NQPH BK YTF KDTGMATZ
Decoding of an encrypted message is equally straightforward. One writes the keyword repeatedly above the message:

Keyword: \(\text{RELATIONS RELATIONS RELATIONS RELATIONS}\)
Scrambled Message: \(\text{LT NF IA CCM LT NF NQPH BK YTF KDTGMATZ}\)
Decoded Message: \(\text{TO BE OR NOT TO BE THAT IS THE QUESTION}\)

This time one uses the keyword letter to pick a row of the table and then traces the row to the column containing the encoded letter. The index of that column is the decoded letter.

Implement a .NET class library called “VigenereLibrary” (Windows file name “VigenereLibrary.dll”). That is, define the namespace \textit{VigenereLibrary} and within this namespace the class \textit{Cipher}:

```csharp
namespace VigenereLibrary {
    public class Cipher {
        // public constructor
        public Cipher();
        // public method to encode a string.
        public string Encode( char[] aKey, string aText );
        // public method to encode a string
        public string Decode( char[] aKey, string aText );
    }
}
```

Note:

The methods \textit{Encode} and \textit{Decode} only cipher characters (or letters). All other characters remain unchanged (i.e, the cipher process ignores them). Furthermore, the Vigenère cipher uses upper case characters only. That is, both the keyword and the message have to be first converted to upper-case characters strings before applying the cipher. For example

Keyword: \(\text{RELATIONS RELATIONS RELATIONS RELATIONS}\)
Message: \(\text{A horse! a horse! my kingdom for a horse.}\)
Encoded Message: \(\text{SMASMN! P VHJXQ! NS TXBZVTY GIA P VHJXQ.}\)
Decoded Message: \(\text{A HORSE! A HORSE! MY KINGDOM FOR A HORSE.}\)
**Problem 6**
Implement the Windows application “TooManySecrets.exe” that uses the class library “VigenereLibrary.dll”.

**MainForm:**
- Title = “TooManySecrets”
- FormBorderStyle: Fixed3D
- textBox1: ScrollBars = Both
- textBox2: -
- label1: Text = “Enter Text to Encode/Decode”
- label2: Text = “Key”
- button1: Text = “&Encode”
- button2: Text = “&Decode”

**Output:**
- Title = “Output”
- FormBorderStyle: Fixed3D
- textBox1: ScrollBars = Both
  - ReadOnly = True
  - TabStop = False
- button1: DialogResult = OK
  - Text = &OK
The MainForm.textBox1 contains the text to encode/decode using the keyword specified in textBox2. Both buttons define a Click event handler in which the corresponding operation is performed. The result is then being used to set the Text property of Output.textBox1. To set the text and show the Output form, you need to create an Output object, set the Output.textBox1.Text property, and call the ShowDialog method. ShowDialog will display Output as a “Modal Dialog Window”. Click OK to close the window.

Submission deadline: Wednesday, September 14, 2005, 4:10 p.m.
Submission procedure: on paper in class (.cs files only) and electronically using the turmin-hw1 script, which is located in ~/cs430x/public/bin. Please use the printout of the submission confirmation email as cover page and check the problems that you have solved.

In order to submit your homework solutions, go (using your CS UNIX account) into the directory that contains your solution (i.e., C#-source files and all related project files). In that directory run the command “~/cs430x/public/bin/turmin-hw1”. After a successful submission, your will receive a confirmation email. Before the due date, you can resubmit your solutions as often as you like.

On the department’s Windows XP systems you can use the command csc to compile C#-programs. However, it is recommended to use Visual Studio .NET, because most assignments require some GUI work.