|  |
| --- |
| SEW |
| Playfair Chiffre |
| Verschlüsselung |

|  |
| --- |
| Felix  22.11.2021  3h |

# Inhaltsverzeichnis

Inhalt

[Inhaltsverzeichnis 1](#_Toc88468033)

[Angabe 1 <<…>> 2](#_Toc88468034)

[Theorie 1 <<…>> 2](#_Toc88468035)

[Klassendiagramm 1 <<…>> 2](#_Toc88468036)

[Quellcode 1 <<…>> 2](#_Toc88468037)

[Testfälle 1 <<…>> 2](#_Toc88468038)

# Angabe 1 <<…>>

1. Read the description („Playfair Chiffre Description.pdf“) of the algorithm carefully and try to reproduce the example.
2. Execute the encryption algorithm **manually** by using:
   1. Key: “Snowball”
   2. Sentence: “Euston saw I was not Sue” (=palindromic sentence!)
   3. Encrypted sentence: “AZBMWOAFDRSDNOBQASCZ”
3. Execute the decryption algorithm **manually** by using:
   1. Key: “Snowball”
   2. Encrypted sentence: “CAWLTAXWYMMENEKZ”
   3. What’s the resulting string? (the result is: Lend me your tablet)
4. Implement the algorithm by means of a console application in C#: Try to find answers to questions a) – e) and then start implementing – not before!
   1. Design a class for the algorithm
   2. Which data structure can be used to store the translation table?
   3. Define the fields you will use!
   4. Do you need any properties?
   5. What could be the task of the constructor of this class?
   6. Implement a method “Decrypt” and a method “Encrypt”
   7. Every rule for the treatment of a “digram” must be kept in a single method!
5. Test your program with the test cases from 2 and 3. Create Unit-Tests for different test cases.
6. Write a Resolutinfile – Create a zip-File and upload both in MS Teams.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

# Theorie 1 <<…>>

1. Encryption, Decryption, SetTable, SetPasswd
2. char[,]
3. string password, char[,] table
4. no
5. setPasswd
6. ok
7. ok

# Klassendiagramm 1 <<…>>

# Ein Bild, das Text enthält. Automatisch generierte Beschreibung

# Quellcode 1 <<…>>

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace PlayfairChiffre

{

public class PlayfairChiffre

{

private string? password;

private char[,]? table;

public PlayfairChiffre(string passwd)

{

SetPassword(passwd);

}

public PlayfairChiffre() :this ("Password") { }

public void SetPassword(string passwd)

{

if(passwd != null)

{

passwd = passwd.ToUpper();

passwd = RemoveNonStrings(passwd);

this.password = passwd;

}

this.table = new char[5, 5];

SetTable();

}

private void SetTable()

{

char active = 'A';

int active\_passwd\_letter = 0;

// 2 For-Schleifen für Tabelle (horizontal und vertikal)

for (int i = 0; i < table.GetLength(0); i++)

{

for (int j = 0; j < table.GetLength(1); j++)

{

// In Schlüsselphase soll Schlüssel in Tabelle geschrieben werden

if(active\_passwd\_letter < password.Length)

{

active = char.Parse(password.Substring(active\_passwd\_letter, 1));

active\_passwd\_letter++;

}

// In "Nicht-Schlüsselphase" soll Alphabet aufgefüllt werden

else if(active\_passwd\_letter == password.Length)

{

active = 'A';

active\_passwd\_letter++;

}

// Überprüfung ob Buchstabe bereits vorkommt oder j,

// wenn ja, active++, aber j bleibt gleich, damit in gleicher Zelle

if (Letter\_already\_exists(active))

{

j--;

active++;

continue;

}

// aktiv errechneten Buchstaben in Zelle schreiben

table[i, j] = active;

// active erhöhen, damit das Alphabet weitergeht

active++;

}

}

}

private bool Letter\_already\_exists(char active)

{

for (int k = 0; k < table.GetLength(0); k++)

{

for (int l = 0; l < table.GetLength(1); l++)

{

if (active == table[k, l] || active == 'J')

{

return true;

}

}

}

return false;

}

public string Encryption(string msg)

{

string[] division = Encryption\_Division(msg.ToUpper());

return Encryption\_Calculation(division);

}

private string[] Encryption\_Division(string msg)

{

msg = RemoveNonStrings(msg);

for (int i = 0; i <= msg.Length; i+=2)

{

try

{

if (msg[i] == msg[i + 1])

msg = msg.Insert(i + 1, "X");

}

catch { }

}

if (msg.Length % 2 == 1)

msg = msg.Insert(msg.Length, "X");

for (int i = 2; i < msg.Length; i+=3)

{

msg = msg.Insert(i, "-");

}

string[] division = msg.Split('-');

return division;

}

private string Encryption\_Calculation(string[] division)

{

string[] code = division;

for (int i = 0; i < division.Length; i++)

{

string item = division[i];

// Letter One X-Position, ...

int l1x = -1, l1y = -2, l2x = -3, l2y = -4;

for (int j = 0; j < table.GetLength(0); j++)

{

// Buchstaben suchen

for (int k = 0; k < table.GetLength(1); k++)

{

if(table[j, k] == char.Parse(item.Substring(0, 1)))

{

l1x = k;

l1y = j;

}

if (table[j, k] == char.Parse(item.Substring(1, 1)))

{

l2x = k;

l2y = j;

}

}

}

// neue Buchstaben schreiben, je nach Fall

code[i] = "";

if (l1y == l2y)

{

if (l1x + 1 < 5)

code[i] += table[l1y, l1x + 1];

else code[i] += table[l1y, 0];

if (l2x + 1 < 5)

code[i] += table[l2y, l2x + 1];

else code[i] += table[l2y, 0];

}

else if (l1x == l2x)

{

if (l1y + 1 < 5)

code[i] += table[l1y + 1, l1x];

else code[i] += table[0, l1x];

if (l2y + 1 < 5)

code[i] += table[l2y + 1, l2x];

else code[i] += table[0, l2x];

}

else

{

code[i] += table[l1y, l2x];

code[i] += table[l2y, l1x];

}

}

StringBuilder sb = new StringBuilder();

foreach (var item in code)

{

sb.Append(item);

}

return sb.ToString();

}

public string Decryption(string code)

{

string[] division = Decryption\_Division(code.ToUpper());

return Decryption\_Calculation(division);

}

private string[] Decryption\_Division(string msg)

{

for (int i = 2; i < msg.Length; i += 3)

{

msg = msg.Insert(i, "-");

}

string[] division = msg.Split('-');

return division;

}

private string Decryption\_Calculation(string[] division)

{

string[] code = division;

for (int i = 0; i < division.Length; i++)

{

string item = division[i];

// Letter One X-Position, ...

int l1x = -1, l1y = -2, l2x = -3, l2y = -4;

for (int j = 0; j < table.GetLength(0); j++)

{

// Buchstaben suchen

for (int k = 0; k < table.GetLength(1); k++)

{

if (table[j, k] == char.Parse(item.Substring(0, 1)))

{

l1x = k;

l1y = j;

}

if (table[j, k] == char.Parse(item.Substring(1, 1)))

{

l2x = k;

l2y = j;

}

}

}

// neue Buchstaben schreiben, je nach Fall

code[i] = "";

if (l1y == l2y)

{

if (l1x - 1 >= 0)

code[i] += table[l1y, l1x - 1];

else code[i] += table[l1y, 4];

if (l2x - 1 >= 0)

code[i] += table[l2y, l2x - 1];

else code[i] += table[l2y, 4];

}

else if (l1x == l2x)

{

if (l1y - 1 >= 0)

code[i] += table[l1y - 1, l1x];

else code[i] += table[4, l1x];

if (l2y - 1 >= 0)

code[i] += table[l2y - 1, l2x];

else code[i] += table[4, l2x];

}

else

{

code[i] += table[l1y, l2x];

code[i] += table[l2y, l1x];

}

}

StringBuilder sb = new StringBuilder();

foreach (var item in code)

{

sb.Append(item);

}

return sb.ToString();

}

private string RemoveNonStrings(string value)

{

string nonstrings = " !\"§$%&/()=\\²³{[]}+\*~#´'+-\_.:,;<>|^°0123456789";

foreach (char c in nonstrings)

{

for (int i = 0; i < value.Length; i++)

{

if (value.Substring(i, 1) == c.ToString())

value = value.Remove(i, 1);

}

}

for (int i = 0; i < value.Length; i++)

{

if (value.Substring(i, 1) == "Ü")

{

value = value.Remove(i, 1);

value = value.Insert(i, "UE");

}

if (value.Substring(i, 1) == "Ö")

{

value = value.Remove(i, 1);

value = value.Insert(i, "OE");

}

if (value.Substring(i, 1) == "Ä")

{

value = value.Remove(i, 1);

value = value.Insert(i, "AE");

}

}

return value;

}

}

}

# Testfälle 1 <<…>>

using Microsoft.VisualStudio.TestTools.UnitTesting;

using PlayfairChiffre;

namespace UT\_PlayfairChiffre

{

[TestClass]

public class UnitTest\_PlayfairChiffre

{

[TestMethod]

public void TestEncryption()

{

PlayfairChiffre.PlayfairChiffre pc = new PlayfairChiffre.PlayfairChiffre("Snowball");

string test = pc.Encryption("Euston saw I was not Sue");

Assert.AreEqual(test, "AZBMWOAFDRSDNOBQASCZ");

}

[TestMethod]

public void TestDecryption()

{

PlayfairChiffre.PlayfairChiffre pc = new PlayfairChiffre.PlayfairChiffre("Snowball");

string test = pc.Decryption("CAWLTAXWYMMENEKZ");

Assert.AreEqual(test, "LENDMEYOURTABLET");

}

}

}

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung