39. Bundeswettbewerb Informatik

Runde 1

Junioraufgabe 1: Passwörter

Chuyang Wang, Team-ID 00318

07. November 2020

Inhaltsverzeichnis

Lösungsidee

Die deutschen Worte bestehen aus Vokalen und Konsonanten. Als Vokalen bezeichnen wir die Buchstaben wie „a“, „o“, „e“, „i“, „u“, „ae“ (Ersatz für „ä“), „oe“, „y“, etc. Außerdem werden alle Zweilauten wie „au“, „eu“, „ei“, usw. den Vokalen zugeordnet. Mit Konsonanten sind alle andere Buchstaben, die nicht zu den Vokalen gehören, gemeint. Darüber hinaus gibt es noch Konsonantencluster[1], die aus mehrere Konsonantenbuchstaben bestehen. Zu den Clustern zählen bspw. „pf“, „tsch“, „ch“, etc.

Dadurch, dass man die Vokalen und Konsonanten (Cluster) nebeneinander verbindet, werden neue Wörter entstehen, die der deutschen Aussprachenregel entsprechen (bzw. gut auszusprechen sind). Besonders zu beachten ist aber, dass manche Konsonanten (Cluster) nicht am Anfang oder Ende eines Wortes stehen darf, denn es lässt sich nicht mehr aussprechen. Ein Beispiel dafür ist der Cluster „lb“, der nicht am Anfang stehen kann, weil der Buchstabe „l“ nicht mehr ausgesprochen werden kann.

Indem man die Vokalen und Konsonanten zufällig auswählt und mit dem obigen Verfahren verbindet, bekommt man ein Passwort, das einerseits relativ sicher, andererseits aber auch gut auszusprechen und zu merken ist.

Zusammenfassend werden drei Regeln festgelegt:

1. Das Passwort besteht aus Vokalen und Konsonanten. Die werden zufällig gewählt und zusammen verbindet. Nach einem Vokal folgt ein Konsonant und nach einem Konsonanten folgt ein Vokal. Dazwischen dürfen auch Zahlen stehen. Somit lässt sich das Passwort gut aussprechen, denn so kann man die Konsonanten, die allein nicht auszusprechen sind, mithilfe der Vokalen ausgesprochen werden.

2. Die Konsonanten (Cluster), die am Anfang des Passwortes stehen dürfen, sind beschränkt. In diesem Fall werden nicht alle Cluster zur Verfügung gestellt, sondern nur diejenige, die am Anfang gut auszusprechen sind. Die sind alle Konsonantenbuchstaben und teils der Cluster. Die anderen Cluster, wie z. B. „lb“, werden dann nicht am Anfang des Passwortes gesetzt.

   Falls ein Konsonant zufällig ausgewählt wird, wird dann auch getestet, ob dieser am Ende des Passwortes steht und ob er auch dastehen darf. Dazu zählt bspw. „zw“, die nicht am Ende stehen kann. Ist es der Fall, wird ein neuer Vokal bzw. Konsonant zufällig gewählt, der diesen ersetzen soll, damit das ganze Passwort gesprochen werden kann.

3. Das Passwort darf nur die Nummer enthalten, die nicht größer als 99 und nicht mit 0 anfängt. Bspw. dürfen 7 oder 69 in dem Passwort stehen, während 09 oder 101 nicht. Der Grund dafür ist, dass die Nummer ab 100 viel zu schwer zu merken sind.

   Darüber hinaus darf innerhalb des Passworts nicht zu viele Zahlen stehen, denn man kann nicht so viele Zahlen merken. Die Anzahl der Zahlen wird mit einer Formel berechnet:

[1] Die vollständige Liste für deutsche Konsonantencluster findet man unter https://de.wikipedia.org/wiki/Konsonantencluster#Die_Mehrgraphen_im_Einzelnen [zuletzt besucht: 06.10.2020]

Lass A = Anzahl der erlaubten Zahlen, x = Länge des Passworts, dann:

$$A = \lfloor x/12 + 0.5 \rfloor = 0$$

Der Buchstabe nach einer Zahl wird großgeschrieben. Dadurch wird das Passwort sicherer, weil auch wenn der Angreifer diese Regel weiß, weiß er nicht, wo die Zahl steht. Somit werden 26 weitere Möglichkeiten, nämlich die großen Buchstaben, hinzugefügt.

Umsetzung

Das Programm wird in C# 8.0 implementiert und mit .Net Core 3.1 CLI kompiliert. Um .Net Core 3.1 SDK herunterzuladen, klicken Sie hier.

Alle Konsonanten und Vokalen (incl. Zweilauten) werden in zwei Arrays gespeichert. Die Konsonantencluster werden in ein anderes Array gespeichert.

```
private static readonly string[] _konsonan-
ten = new string[] { "b", "c", "d", "f", "g", "h", "j", "k", "l", "m", "n", "p",
"q", "r", "s", "t", "v", "w", "x", "z", "ss" };

// Regel 1: Alle Konsonanten-Clusters
private static readonly string[] _clus-
ters = new string[] { "bl", "br", "ch", "chl", "dr", "dw", "fl", "fr", "gn", "gr"
, "kn", "kr", "pf", "pfl", "pfr", "pl", "pn", "pr", "ph", "phl", "phr", "sk", "sk
l", "skr", "sl", "sp", "sph", "spl", "spr", "st", "str", "sz", "sch", "schl", "sc
hm", "schn", "schr", "schw", "tr", "wr", "zw", "bs", "bsch", "bst", "bt", "ch", "
chs", "cht", "dm", "dt", "ft", "gd", "gl", "gs", "gt", "kt", "kl", "ks", "lb", "l
ch", "ld", "lf", "lg", "lk", "lm", "ln", "lp", "ls", "lsch", "lt", "lv", "lz", "m
b", "md", "mn", "mp", "mpf", "mph", "mphl", "ms", "msch", "mt", "nd", "nf", "nft"
, "ng", "ngl", "ngs", "ngst", "nk", "nkt", "ns", "nsch", "nst", "nt", "nx", "nz",
 "ps", "psch", "pst", "pt", "sk", "sp", "sph", "st", "sch", "scht", "schr", "tm",
 "ts", "tsch", "tw", "tz", "tzt", "xt" };

// Regel 1: Alle deutsche Vocale + Zweilauten
private static readonly string[] _vo-
cale = new string[] { "a", "o", "i", "e", "u", "y", "au", "ai", "eu", "ei", "ui",
 "ae", "oe", "ue" };
```

Die Cluster, die am Anfang und Ende stehen dürfen, werden trennend in zwei weiteren Arrays gespeichert. Die dienen zu prüfen, ob diese Cluster an dieser Stelle gesetzt werden darf (vgl. Regel 2).

```
    // Regel 2: Clusters, die am Anfang stehen koennen:
    private static readonly string[] _startClus-
ters = new string[] { "bl", "br", "ch", "dr", "fl", "fr", "gn", "gr", "kn", "kr",
 "pf", "pfl", "pfr", "pl", "pn", "pr", "ph", "phl", "phr", "sk", "skl", "skr", "s
l", "sp", "sph", "spl", "spr", "st", "str", "sz", "sch", "schl", "schm", "schn",
"schr", "schw", "tr", "zw", "ch", "gl", "kl", "sp", "sph", "sch", "scht", "schr",
 "tsch" };

    // Regel 2: Clusters, die am Ende eines Wortes stehen koennen
    private static readonly string[] _endClus-
ters = new string[] { "ch", "pf", "ph", "sk", "skl", "sl", "sp", "sph", "st", "sc
h", "bst", "bt", "ch", "chs", "cht", "dm", "dt", "ft", "gd", "gs", "gt", "kt", "k
s", "lb", "lch", "ld", "lf", "lg", "lk", "lm", "ln", "lp", "ls", "lsch", "lt", "l
z", "mb", "md", "mp", "mpf", "mph", "ms", "msch", "mt", "nd", "nf", "nft", "ng",
"ngs", "ngst", "nk", "nkt", "ns", "nsch", "nst", "nt", "nx", "nz", "ps", "psch",
"pst", "pt", "sk", "sp", "sph", "st", "sch", "ts", "tsch", "tz", "tzt", "xt" };
```

Die Methode *GeneratePassword* generiert ein Passwort, das einigermaßen gut aussprechbar ist.

```
    /// Methode, die ein Passwort mit gegebener Laenge generiert
    /// @param length: Eine ganze Zahl, die besagt, wie lange das zu generie-
rende Passwort sein sollte.
    /// @return string: Das generierte Passwort
    public static string GeneratePassword(int length)
```

Innerhalb dieser Methode wird eine Liste jedes Mal neu instanziiert, wenn diese Methode aufgerufen wird. Die Vokalen und Konsonanten, die zufällig gewählt sind, werden in diese Liste zugefügt.

```
      List<string> password = new List<string>();
```

Dann wird festgelegt, wie viele Zahlen innerhalb dieses Passworts vorhanden sein dürfen (vgl. Regel 3). Bspw. dürfen die Passwörter, die eine Länge von 1-5 hat, keine Zahlen besitzen ($\forall\, x \in N < 6,\ \lfloor x/12 + 0.5 \rfloor = 0$). Passwörter, die zwischen 6 bis 17 Zeichen lang sind, dürfen maximal eine Zahl haben ($\forall\, x \in 5 < N < 18,\ \lfloor x/12 + 0.5 \rfloor = 1$).

```
      // Rechnen, wie viele Nummer (maximal) in diesem Passwort sein darf
      int maximalNumberInPassword = (int)(length / 12d + 0.5);
      // Speichert, wie viele Nummer schon in dem Passwort zugefuegt sind
      int numberInPasswordCount = 0;
```

Danach fängt die While-Schleife an, zufällige Vokalen, Konsonanten und Zahlen für das Passwort zu wählen.

```
      while (password.GetStringLength() < length)
```

In der Schleife wird eine Zufallszahl erzeugt, die entscheidet, ob an dieser Stelle eine Zahl in das Passwort hinzugefügt werden soll. Die Wahrscheinlichkeit, an dieser Stelle eine Zahl hinzuzufügen, beträgt $\frac{1}{5}$, *solange die maximale Anzahl von Zahlen nicht erreicht wird.*

Falls hier eine Zahl stehen kann, wird eine **weitere** Zufallszahl generiert. Weil es nur Zahlen kleiner als 100 geben dürfen, wird hier das Modulo von dieser weiteren Zufallszahl und *NUMBER_LIMIT*, in diesem Fall 100, in das Passwort hinzugefügt.

```
    private const int NUMBER_LIMIT = 100;
```

```
        // Test ob hier eine Nummer sein koennte
        if ((SafeRandGen.GetByte() % 5 == 1) && maximalNumberInPassword > number-
InPasswordCount && password.IsNumberFollowable())
        {
          // Regel 3: Nummer duerfen maximal 99 sein und 0 darf nicht vor irgend-
einer beliebigen Nummer stehen
          password.Add((SafeRandGen.GetByte() % NUMBER_LIMIT).ToString());
          numberInPasswordCount++;
        }
```

Die statische Klasse *SafeRandGen* enthält eine Methode *GetByte*, die eine zufällige Zahl zwischen 0 und 255 zurückgibt. Dadurch generierte Zufallszahl ist im kryptographischen Sinne „sicher“.

```
// Generiert eine aus der kryptographischen Sicht gesehen sichere Zufallsnummer
    // Also eine Zufallsnummer, die hier als reale Zufallsnummer gesehen wer-
den kann
    private static class SafeRandGen
    {
      private static readonly RNGCryptoServiceProvider rngCsp = new RNGCryptoS-
erviceProvider();

      public static byte GetByte()
      {
        byte[] b = new byte[1];
        rngCsp.GetBytes(b);
        return b[0];
      }
    }
  }
```

Nach der Entscheidung, ob eine Zahl ins Passwort gesetzt wird, wird nun einen Vokal oder Konsonanten (Cluster) zufällig gewählt. Die Erzeugung der Zufallsnummer erfolgt auch durch die Methode *GetByte* von *SafeRandGen*. Die Erweiterungsmethode *GetFollowableChars* gibt ein Array von *string* zurück. Die generierte Zufallszahl wird als *Index des Arrays* genutzt. Durch die Modulo-Rechnung kann man die Größe der Zahl beschränken, sodass der Index nicht die Grenze bzw. die Länge des Arrays überschreiten.

```
// Zufallsnummer mod Laenge der Moeglichkeit, so beschraenkt man den Be-
reich der generierten Zufallsnummer
string stringToAdd = password.GetFollowableChars()[SafeRandGen.GetByte() % pass-
word.GetFollowableChars().Length];
```

In der Erweiterungsmethode *GetFollowableChars(this List<string> sL)* wird festgestellt, ob es ein Vokal oder Konsonant in das Passwort hinzugefügt werden darf. Grundsätzlich wird nach einem Vokal ein Konsonanten (Cluster) folgen, nach einem Konsonanten ein Vokal.

```
/// Erweiterungsmethode, die einen Array zurueck-
gibt, in der alle moegliche Konbination von Buchstaben steht
/// @param string s: Das vorher bereits generierte Teil-Pass-
wort in Form von List<string>
/// @return string[]: Ein Array, das alle moegliche Konbinatio-
nen von Buchstaben beinhaltet
private static string[] GetFollowableChars(this List<string> sL)
```

```
// Nach einem Konsonant folgt ein Vokal
if (_konsonanten.Union(_clusters).Contains(sL.Last().ToLower())) return _vocale;
// Nach einer Zweilaut folgt ein Konsonant
if (sL.Last().Length > 1 && _vocale.Contains(sL.Last().ToLower())) return _konso-
nanten.Union(_clusters).ToArray();
```

Wenn der vorige Vokal nur aus einem Buchstaben besteht, wird noch ggf. weitere Vokale-Buchstaben zurückgegeben, sodass es möglicherweise noch eine Zweilaute bilden kann.

```
// Falls davor nur ein Vokal steht, koennen
      // sowohl Konsonanten, aber auch teilweise andere Vokalen folgen
      // Insofern wird eine Zweilaute gebildet.
      switch (sL.Last().ToLower().ToCharArray()[sL.Last().Length - 1])
      {
        // Bspw. a + u = au = eine gueltige Zweilaut
        case 'a':
          return ((new string[] { "u", "i" }).Union(_konsonanten).Union(_clus-
ters)).ToArray();
        case 'o':
          return ((new string[] { "u" }).Union(_konsonanten).Union(_clus-
ters)).ToArray();
        case 'e':
          return ((new string[] { "u", "i" }).Union(_konsonanten).Union(_clus-
ters)).ToArray();
        case 'i':
          return _konsonanten.Union(_clusters).ToArray();
        case 'u':
          return ((new string[] { "i" }).Union(_konsonanten).Union(_clus-
ters)).ToArray();
        case 'y':
          return _konsonanten.Union(_clusters).ToArray();
        default:
          System.Diagnostics.Debug.WriteLine(sL.ListToString());
          throw new ArgumentException("The given character is not imple-
mented!", nameof(sL));
      }
```

Als Ausnahme dürfen am Anfang und Ende des Passwortes nicht alle Konsonantencluster zur Verfügung gestellt (vgl. Regel 2). Außerdem wird die Stelle nach einer Zahl auch als Anfang betrachtet.

```
      // Regel 2: nur Konsonantenbuchstaben und teils der
      // Cluster koennen am Anfang des Passwortes stehen
      if (sL.Count == 0) return _startClusters.Union(_konsonanten).ToArray();
      // Nach einer Zahl koennen sowohl Vokalen als auch Konsonanten folgen
      if (Char.IsNumber(sL.Last().ToLower().ToCharArray()[0])) return _vo-
cale.Union(_startClusters).Union(_konsonanten).ToArray();
```

Am Ende der Schleife in der Methode *GeneratePassword* werden die Buchstaben nach einer Zahl großgeschrieben, um die Sicherheit des Passworts zu verbessern.

```
// der Buchstabe nach einer Zahl wird gross geschrieben
if (password.Count > 0 && int.TryParse(password.Last(), out _))
          stringToAdd = char.ToUpper(stringToAdd[0]) + stringToAdd.Substring(1);
```

Endlich wird der zufällige Vokal oder Konsonant in die Liste zugefügt.

```
        // Fuegt das neue Teil des Passworts hinzu
        password.Add(stringToAdd);
```

Nachdem das Passwort genug lang ist, endet die Schleife und die Liste wird in ein *string* umgewandelt und zurückgegeben.

```
      // Am Ende gibt man das generierte Passwort zurueck
      return password.ListToString().Substring(0, length);
```

Beispiele

Erwartete Formen von Daten

Dieses Programm ist **Framework-abhängig** komprimiert (wegen Größenbeschränkung für Hochladen).

Sollten Sie .Net Core 3.1 Runtime nicht auf ihrem Computer haben, können Sie es [hier](https://dotnet.microsoft.com/download/dotnet-core/3.1) runterladen.

Ohne die Runtime-Bibliothek installiert zu werden, kann das Programm **NICHT** ausgeführt werden.

Angabe für das Programm:

```
A L
```

Zwei Integer *A* und *L*, die durch ein Leerzeichen (, ‘) getrennt werden sollen. Der Integer A gibt an, wie viele Passwörter generiert werden sollen, wobei L gibt an, wie lange das jeweilige Passwort sein soll. Hier sollen $A \in N \leq Int32.MaxValue, L \in N \leq Int32.MaxValue$.

Beispiel für die Angabe:

```
10 16
```

Das bedeutet, dass das Programm 10 Passwörter von 16 Zeichen generieren soll. Die generierten Passwörter werden in der Console gezeigt.

Nach der Ausgabe des Ergebnisses wird das Programm den Nutzer auffordern, eine beliebige Taste zu drücken, um das Programm zu schließen.

Ergebnis für A=10, L=16

```
glauklaiknytschu
koensch8Pfoepfro
skruimphluskleu7
schtynst26Aenfok
68Guepfroempfixt
gloeskloend39Ple
11Blaumphlumolva
schreigte90Wofib
75Uitzuenseilsac
pfru45Ceuveiboms
```

Ergebnis für A=4, L=8

```
sleupneu
trysleuq
deuschla
proe78Pf
```

Ergebnis für A=4, L=20

```
fleileispleispraidt3
qynipfraipscheu84Pib
schnuipf63Glumtaildu
```

```
le67Muphuilgeunschem
```

Quellcode

```
1.  using System;
2.  using System.Linq;
3.  using System.Collections.Generic;
4.  using System.Security.Cryptography;
5.
6.  namespace JuniorAufgabe1
7.  {
8.    public static class PasswordGenerator
9.    {
10.
11. Die Vokalen und Konsonanten warden hier nicht gezeigt.
12. Der vollständige Quellcode befindet sich in Junioraufgabe1>Quellcode
13.
14.     // Regel 3: Die Zahl darf maximal 99 sein, also kleiner als 100
15.     private const int NUMBER_LIMIT = 100;
16.
17.
18.     // Programmeingang
19.     // Es soll in der ersten Zeile zwei Integer, A L,  gegeben werden, die d
    urch ein Leerzeichen getrennt werden (vgl. Dokumentation-Beispiel)
20.     // A representiert die Anzahl der Testfaelle bzw. Anzahl der zu generier
    ende Passwoerter. A ist eine positive ganze Zahl, die nicht groesser als 2,1
    47,483,647 sein sollte.
21.     // L representiert die Laenge jedes Passwortes. L ist eine positive ganz
    e Zahl, die nicht groesser als 2,147,483,647 sein sollte.
22.     // Jedoch ist es schon sehr sinnvoll, eine relative kleine Zahl fuer A o
    der L zu geben. Bspw. A = 10, L = 16
23.     public static void Main(string[] args)
24.     {
25.       string[] input = Console.ReadLine().Split(' ');
26.       int testCases = Convert.ToInt32(input[0]);
27.       int length = Convert.ToInt32(input[1]);
28.
29.       for (int i = 0; i < testCases; i++)
30.       {
31.         Console.WriteLine(GeneratePassword(length));
32.       }
33.
34.       Console.WriteLine("Drueck eine beliebige Taste zu schliessen...");
35.       Console.ReadKey();
36.     }
37.
38.     /// Methode, die ein Passwort mit gegebener Laenge generiert
39.     /// @param length: Eine ganze Zahl, die besagt, wie lange das zu generie
    rende Passwort sein sollte.
40.     /// @return string: Das generierte Passwort
41.     public static string GeneratePassword(int length)
42.     {
43.       // Check given argument
44.       if (length < 1) throw new ArgumentException("Password must contain at
    least one character!", nameof(length));
45.
46.       List<string> password = new List<string>();
47.
48.       // Rechnen, wie viele Nummer (maximal) in diesem Passwort sein darf
49.       int maximalNumberInPassword = (int)(length / 12d + 0.5);
50.       // Speichert, wie viele Nummer schon in dem Passwort zugefuegt sind
51.       int numberInPasswordCount = 0;
52.
53.
54.       while (password.GetStringLength() < length)
55.       {
```

```
56.
57.         // Test ob hier eine Nummer sein koennte
58.         if ((SafeRandGen.GetByte() % 5 == 1) && maximalNumberInPassword > nu
    mberInPasswordCount && password.IsNumberFollowable())
59.         {
60.           // Regel 3: Nummer duerfen maximal 99 sein und 0 darf nicht vor ir
    gendeiner beliebigen Nummer stehen
61.           password.Add((SafeRandGen.GetByte() % NUMBER_LIMIT).ToString());
62.           numberInPasswordCount++;
63.         }
64.
65.         // Zufallsnummer mod Laenge der Moeglichkeit, so beschraenkt man den
     Bereich der generierten Zufallsnummer
66.         string stringToAdd = password.GetFollowableChars()[SafeRandGen.GetBy
    te() % password.GetFollowableChars().Length];
67.
68.         // Falls es dann zu lange waere, fang direkt die naechste Schleife a
    n
69.         if (password.GetStringLength() + stringToAdd.Length > length) contin
    ue;
70.         // Testet, wenn diese neue String am Ende ist, ob diese auch als End
    ung stehen darf
71.         else if (password.GetStringLength() + stringToAdd.Length == length)

72.         {
73.           if (_clusters.Except(_endClusters).Contains(stringToAdd)) continue
    ;
74.           if (_unendableChars.Contains((password.ListToString() + stringToAd
    d).Substring(length - 1, 1))) continue;
75.         }
76.
77.         // der Buchstabe nach einer Zahl wird gross geschrieben
78.         if (password.Count > 0 && int.TryParse(password.Last(), out _))
79.           stringToAdd = char.ToUpper(stringToAdd[0]) + stringToAdd.Substring
    (1);
80.
81.
82.         // Fuegt das neue Teil des Passworts hinzu
83.         password.Add(stringToAdd);
84.       }
85.
86.       // Am Ende gibt man das generierte Passwort zurueck
87.       return password.ListToString().Substring(0, length);
88.     }
89.
90.
91.     /// Erweiterungsmethode, die die Laenge aller string dieser List<string>
     zurueckgibt
92.     /// @param List<string> sL
93.     /// @return int
94.     private static int GetStringLength(this List<string> sL)
95.     {
96.       int length = 0;
97.       foreach (string s in sL)
98.       {
99.         length += s.Length;
100.            }
101.            return length;
102.          }
103.
104.
105.          /// Erweiterungsmethode, die eine List<string> als eine gasamte s
    tring zurueckgibt
106.          /// @param List<string> sL: Eine Liste von string
107.          /// @return string
108.          private static string ListToString(this List<string> sL)
```

```
109.           {
110.             string str = "";
111.             foreach (string s in sL)
112.             {
113.               str += s;
114.             }
115.             return str;
116.           }
117.
118.
119.           /// Erweiterungsmethode, die ein Array zurueckgibt, in dem alle m
    oegliche Konbination von Buchstaben steht
120.           /// @param string s: Das vorher bereits generierte Teil-
    Passwort in Form von List<string>
121.           /// @return string[]: Ein Array, das alle moegliche Konbinationen
     von Buchstaben beinhaltet
122.           private static string[] GetFollowableChars(this List<string> sL)

123.           {
124.             // Regel 2: nur Konsonantenbuchstaben und teils der
125.             // Cluster koennen am Anfang des Passwortes stehen
126.             if (sL.Count == 0) return _startClusters.Union(_konsonanten).To
    Array();
127.             // Nach einer Zahl koennen sowohl Vokalen als auch Konsonanten
    folgen
128.             if (Char.IsNumber(sL.Last().ToLower().ToCharArray()[0])) return
     _vocale.Union(_startClusters).Union(_konsonanten).ToArray();
129.             // Nach einem Konsonant folgt ein Vokal
130.             if (_konsonanten.Union(_clusters).Contains(sL.Last().ToLower())
    ) return _vocale;
131.             // Nach einer Zweilaut folgt ein Konsonant
132.             if (sL.Last().Length > 1 && _vocale.Contains(sL.Last().ToLower(
    ))) return _konsonanten.Union(_clusters).ToArray();
133.             // Falls davor nur ein Vokal steht, koennen
134.             // sowohl Konsonanten, aber auch teilweise andere Vokalen folge
    n
135.             // Insofern wird eine Zweilaute gebildet.
136.             switch (sL.Last().ToLower().ToCharArray()[sL.Last().Length - 1]
    )
137.             {
138.               // Bspw. a + u = au = eine gueltige Zweilaut
139.               case 'a':
140.                 return ((new string[] { "u", "i" }).Union(_konsonanten).Uni
    on(_clusters)).ToArray();
141.               case 'o':
142.                 return ((new string[] { "u" }).Union(_konsonanten).Union(_c
    lusters)).ToArray();
143.               case 'e':
144.                 return ((new string[] { "u", "i" }).Union(_konsonanten).Uni
    on(_clusters)).ToArray();
145.               case 'i':
146.                 return _konsonanten.Union(_clusters).ToArray();
147.               case 'u':
148.                 return ((new string[] { "i" }).Union(_konsonanten).Union(_c
    lusters)).ToArray();
149.               case 'y':
150.                 return _konsonanten.Union(_clusters).ToArray();
151.               default:
152.                 System.Diagnostics.Debug.WriteLine(sL.ListToString());
153.                 throw new ArgumentException("The given character is not imp
    lemented!", nameof(sL));
154.             }
155.           }
156.
157.           /// Erweiterungsmethode, die zeight, ob jetzt eine Nummer kommen
    darf
```

```
158.           /// @param List<string> sL
159.           /// @return bool
160.           private static bool IsNumberFollowable(this List<string> sL)
161.           {
162.             if (sL.Count == 0) return true;
163.             if (sL.Count < 2) return false;
164.             bool isVK = _konsonanten.Union(_endClusters).Contains(sL.Last()
    ) && _vocale.Contains(sL[^2]);
165.             bool isKV = _vocale.Contains(sL.Last()) && _konsonanten.Union(_
    clusters).Contains(sL[^2]);
166.             bool isEndWithIllegalChar = _unendableChars.Contains(sL.Last()[
    ^1].ToString());
167.             return (isVK || isKV) && !isEndWithIllegalChar;
168.           }
169.
170.
171.           // Generiert eine aus der kryptographischen Sicht gesehen sichere
     Zufallsnummer
172.           // Also eine Zufallsnummer, die hier als reale Zufallsnummer gese
    hen werden kann
173.           private static class SafeRandGen
174.           {
175.             private static readonly RNGCryptoServiceProvider rngCsp = new R
    NGCryptoServiceProvider();
176.
177.             public static byte GetByte()
178.             {
179.               byte[] b = new byte[1];
180.               rngCsp.GetBytes(b);
181.               return b[0];
182.             }
183.           }
184.         }
185.       }
```