View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Server für wissenschaftliche Schriften der Hochschule Hannover

# Eine Java-Bibliothek zur Spezifikation von Variabilität in automatisch bewerteten Programmieraufgaben

HOCHSCHULE HANNOVER
UNIVERSITY OF APPLIED SCIENCES AND ARTS
*Fakultät IV Wirtschaft und Informatik*

Robert Garmann[1]

Bericht

25. Januar 2018

Hochschule Hannover
Fakultät IV – Wirtschaft und Informatik
Ricklinger Stadtweg 120
30459 Hannover

---

[1] E-Mail: robert.garmann@hs-hannover.de

## Zusammenfassung

Wir führen schrittweise in den Einsatz einer Java-Bibliothek ein, um Variationspunkte und deren Wertemengen in automatisiert bewerteten Programmieraufgaben zu spezifizieren und als XML-Datei zu exportieren. Solche Variationspunkte kommen bei individualisierbaren Programmieraufgaben zum Einsatz, bei denen jede Studentin und jeder Student eine eigene Variante einer Programmieraufgabe erhält.

## Schlagwörter

Individuelle Programmieraufgaben, Grader, Autobewerter, E-Assessment, Variabilität, Java, XML

## DDC Klassifikation

004 Datenverarbeitung; Informatik

## GND-Schlagwörter

Programmierung, E-Learning, Computerunterstütztes Lernen, Java <Programmiersprache>, XML, Übung <Hochschule>, Lernaufgabe

## ACM CCS (2012)

• **Social and professional topics~Computer science education** • **Social and professional topics~Student assessment** • *Applied computing~Computer-assisted instruction* • *Applied computing~E-learning*

# Inhalt

1 Kartesische Produkte und Vereinigungsmengen ..... 4
1.1 Eine einfache individualisierbare Aufgabe ..... 4
1.2 Eine Aufgabe mit einem dritten Variationspunkt t ..... 6
2 Definition und Referenzierung ..... 8
2.1 Aufgabe mit nicht-symmetrischen Abhängigkeiten ..... 8
2.2 Definition und Referenzierung ..... 9
3 Derive: Ableitung eines Variationspunkts ..... 11
3.1 Eine Aufgabe mit einem anderen dritten Variationspunkt n ..... 11
3.2 Ableitung des Variationspunktes n ..... 13
3.3 Ableitung eines Variationspunktes von mehreren anderen Variationspunkten ..... 14
3.3.1 Ein neuer Variationspunkt w ..... 14
3.3.2 Ableitung von n aus v und w ..... 15
3.4 Ableitung einer Wertmenge für einen Variationspunkt ..... 17
3.5 Schlüsselumordnungen ..... 19
3.6 Leere Ableitung ..... 20
3.7 Sequenz mehrerer Ableitungen ..... 21
4 Zusammenfassung ..... 22

# 1 Kartesische Produkte und Vereinigungsmengen

## 1.1 Eine einfache individualisierbare Aufgabe

Wir betrachten eine einfache individualisierbare Aufgabe:

> *Write a `main` method in class `%vp{c}` in the default package that reads a word or a sentence from user input and counts the number of occurrences of the vowel %vp{v} in the input. Your program should count both upper and lower case letters.*
>
> *Example (user input is **highlighted**):*
>
> ```
> Give me some text, please: authorize tambourine precaution
> Found 3 %vp{v} in your input!
> ```

Aufgabe 1

Die Variationspunkte (Vp) sind hervorgehoben dargestellt und besitzen folgende Wertemengen:

| Variationspunkt | Datentyp | Sinnvolle Werte |
|---|---|---|
| c | String | Gültiger Java-Bezeichner, der im Aufgabenkontext für Studierende nachvollziehbar ist. Z. B. "Program", "Counter", "CountVowels". |
| v | char | Vokale 'a', 'e', 'i', 'o', 'u'. |

Der Aufgabentext ist mit der merkwürdig anmutenden Benutzereingabe so entworfen, dass unabhängig von der Ausprägung des Variationspunkts v immer 3 Vokale die richtige Antwort sind. Dadurch sparen wir hier zunächst einen weiteren Variationspunkt ein, den wir jedoch gleich einführen werden.

Sinnvolle Werte des CVp (composite variation point) dieser sehr einfachen Aufgabenversion entnimmt man einer Teilmenge des zweidimensionalen Raums String $\times$ char. Da die beiden Vp unabhängig voneinander sind, kann man sich die Teilmenge der sinnvollen CVp-Werte als einen Bereich vorstellen, der durch „Slicing" der Ebene in beiden Dimensionen entsteht. Abbildung 1 hebt die resultierende sinnvolle Variantenmenge grafisch durch dunkle kleine Quadrate hervor. Insgesamt resultieren in diesem Beispiel 15 sinnvolle Varianten. Die sinnvollen Varianten lassen sich als kartesisches Produkt zweier Mengen wie folgt beschreiben

{ "Program", "Counter", "CountVowels" } $\times$ { 'a', 'e', 'i', 'o', 'u' }

Wir führen eine alternative Schreibweise ein, die sich später als geeignet heraus stellen wird, um die Menge der sinnvollen Varianten programmgesteuert zu erzeugen:

```
CVSpec.build(Vp.s("c"), Vp.c("v"))
    .combineGroup()
        .collect("Program", "Counter", "CountVowels")    // valid values for c
        .collect('a','e','i','o','u') // valid values for v
    .endCombineGroup()
.endBuild();
```

Hierbei bezeichnet combineGroup ein kartesisches Produkt mehrerer Mengen. Die Elemente dieses Produkts sind die Zeilen, die mit .collect beginnen. Jede .collect-Zeile definiert eine Menge mit den in Klammern angegebenen Elementen. Die Namen und Datentypen der Variationspunkte haben wir in der ersten Zeile angegeben, hinter dem Kürzel build[2]. Vp.s("c") erzeugt hierbei einen Variationspunkt vom Datentyp **S**tring mit Bezeichnung „**c**".

[2] Alle Operationen dieser Notation sind letztendlich Java-Methoden der hier beschriebenen Klassenbibliothek (Arbeitstitel: libvts.jar).

Eine Darstellung der Menge in einem XML-Format schlagen wir wie folgt vor:

```
<?xml version="1.0" ?>
<v:cvSpec xmlns:v="urn:to-be-specified">
  <v:cvp>
    <v:vp key="c" type="string"></v:vp>
    <v:vp key="v" type="character"></v:vp>
  </v:cvp>
  <v:combineGroup>
    <v:collect>
      <v:string value="Program"></v:string>
      <v:string value="Counter"></v:string>
      <v:string value="CountVowels"></v:string>
    </v:collect>
    <v:collect>
      <v:character value="97"></v:character>
      <v:character value="101"></v:character>
      <v:character value="105"></v:character>
      <v:character value="111"></v:character>
      <v:character value="117"></v:character>
    </v:collect>
  </v:combineGroup>
</v:cvSpec>
```

Zeichen (character) werden in diesem Format als Zeichencode angegeben[3].

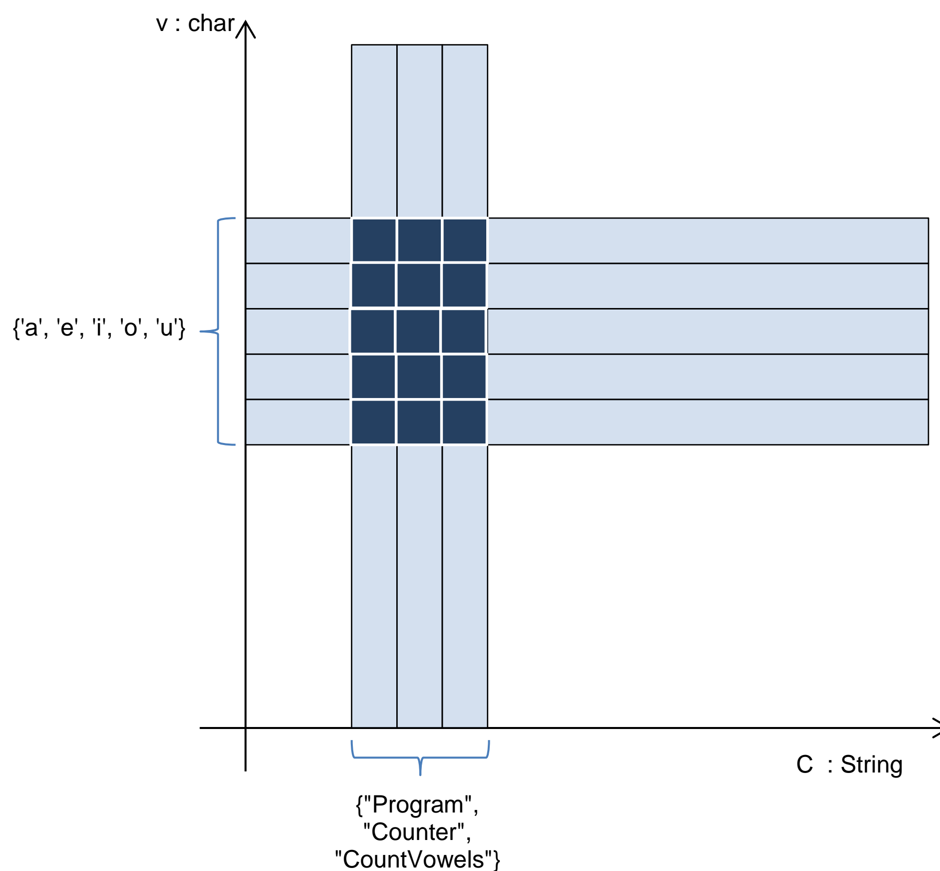


Abbildung 1: Visualisierung der sinnvollen Werte des CVp der Aufgabe 1

[3] Der Grund, warum wir den Zeichencode in den serialisierten XML-Strom schreiben, und nicht das Zeichen selbst, ist an dieser Stelle schwer zu motivieren. Letztendlich hängt die Wahl der Serialisierung als Zeichencode damit zusammen, dass wir in bestimmten Situationen auch das 0-Zeichen serialisieren wollen (als Differenz zweier Zeichen kann das vorkommen). Das 0-Zeichen darf jedoch in keinem XML-Dokument enthalten sein.

## 1.2 Eine Aufgabe mit einem dritten Variationspunkt t

Wir ändern Aufgabe 1 geringfügig ab, um von der merkwürdig anmutenden Eingabe weg zu kommen. Dazu führen wir einen dritten Variationspunkt t ein[4]:

> *Write a `main` method in class `%vp{c}` in the default package that reads a word or a sentence from user input and counts the number of occurrences of the vowel `%vp{v}` in the input. Your program should count both upper and lower case letters.*
>
> *Example (user input is **highlighted**):*

```
Give me some text, please: %vp{t}
Found 3 %vp{v} in your input!
```

Aufgabe 2

Der CVp besteht nun aus den folgenden Variationspunkten:

<table>
<tr><th>Variationspunkt</th><th>Datentyp</th><th colspan="2">Sinnvolle Werte</th></tr>
<tr><td>c</td><td>String</td><td colspan="2">Gültiger Java-Bezeichner, der im Aufgabenkontext für Studierende nachvollziehbar ist. Z. B. "Program", "Counter", "CountVowels".</td></tr>
<tr><td>v</td><td>char</td><td colspan="2">Vokale 'a', 'e', 'i', 'o', 'u'.</td></tr>
<tr><td rowspan="6">t</td><td rowspan="6">String</td><td colspan="2">Abhängig vom Wert v sind mehrere sinnvolle Eingaben denkbar.</td></tr>
<tr><td>v</td><td>Einige sinnvolle Werte für t</td></tr>
<tr><td>'a' oder 'e'</td><td>“Great to see you again!”<br>“Another great place”</td></tr>
<tr><td>'i'</td><td>“Have you imagined this?”</td></tr>
<tr><td>'o'</td><td>“Why should we go around?”</td></tr>
<tr><td>'u'</td><td>“Wounderful tube”</td></tr>
</table>

Wir wollen mit diesen Variationspunkten demonstrieren, dass es möglich ist, dass einige Variationspunkte in komplexer Weise voneinander abhängen. Hier ist zu den Ausprägungen 'a' oder 'e' des Variationspunktes v eine Menge von zwei sinnvollen Elementen für t angegeben, während alle verbleibenden Ausprägungen von v eine (jeweils abweichende) einelementige Menge für t auflistet. Alle Werte für t sind so entworfen, dass jeweils genau 3 Vokale v in dem Text vorkommen. Außerdem sind die Texte t etwas weniger merkwürdig als im ersten Beispiel.

Hier können nun die Variationspunkte v und t nicht unabhängig voneinander gewählt werden. Die folgende Variantenmenge ist daher ungeeignet, da sie viele nicht sinnvolle Kombinationen enthält:

```
CVSpec.build(Vp.s("c"), Vp.c("v"), Vp.s("t"))
    .combineGroup()
        .collect("Program", "Counter", "CountVowels")
       . collect('a','e','i','o','u'),
       . collect("Great to ...", "Another great ...", "Have you ...", "Why should ...", "Wounderful ...")
    .endCombineGroup()
```

[4] Die Variation des eingegebenen Textes ist didaktisch vermutlich nicht unbedingt sinnvoll und notwendig, wie wir später sehen werden. Um das Beispiel hier nicht unnötig kompliziert zu machen, haben wir uns für diese leicht zu beschreibende Änderung des Beispiels entschieden, um bereits in diesem einfachen Beispiel komplexere Variationspunkt-Abhängigkeiten zu induzieren. Solche komplexeren Abhängigkeiten kommen in realen Aufgaben an didaktisch wertvollerer Stelle häufig vor.

Abbildung 2 illustriert die tatsächlich sinnvollen Kombinationen von v und t als „slices“ der Ebene. Die grafische Veranschaulichung zeigt, dass die Slices der Dimensionen v und t nicht voneinander unabhängig sind, sondern dass es für jeden der vier v-Slices { 'a', 'e' }, { 'i' }, { 'o' }, { 'u' } je genau einen sinnvollen zugehörigen t-Slice gibt. Der folgende mathematische Ausdruck gibt die Menge der sinnvollen CVp-Werte an:

$$
\begin{aligned}
&\{ \text{"Program"}, \text{"Counter"}, \text{"CountVowels"} \} \\
&\times \Big( (\{ \text{'a'}, \text{'e'} \} \times \{ \text{“Great to …”}, \text{“Another great …”} \}) \\
&\quad \cup (\{ \text{'i'} \} \times \{ \text{“Have you …”} \}) \\
&\quad \cup (\{ \text{'o'} \} \times \{ \text{“Why should …”} \}) \\
&\quad \cup (\{ \text{'u'} \} \times \{ \text{“Wounderful …”} \}) \Big)
\end{aligned}
$$

Textuell verwenden wir die Bezeichnung “collect” für die Vereinigungsmenge und schreiben:

```
CVSpec.build(Vp.s("c"), Vp.c("v"), Vp.s("t"))
    .combineGroup()
        .collect("Program", "Counter", "CountVowels")
        .collectGroup()
            .combineGroup().collect('a', 'e').collect("Great to see you again!",
                                                   "Another great place")     .endCombineGroup()
            .combine(              'i',            "Have you imagined this?")
            .combine(              'o',            "Why should we go around?")
            .combine(              'u',            "Wounderful tube")
        .endCollectGroup()
    .endCombineGroup()
.endBuild();
```

Diese Schreibweise hat den Vorteil, dass sie wie eine Fallunterscheidung für den Variationspunkt v gelesen werden kann. Neu hierbei eingeführt wurde die Schreibweise combine(…), die eine Kurzform für das kartesische Produkt mehrerer einelementiger Mengen bezeichnen soll:

combine(v1, …, vn) ⇔ combineGroup().collect(v1). ….collect(vn).endCombineGroup()

Außerdem bezeichnet collectGroup eine Vereinigungsmenge mehrerer enthaltener Teilmengen.

Im XML-Format schreiben wir:

```
<?xml version="1.0" ?>
<v:cvSpec xmlns:v="urn:to-be-specified">
  <v:cvp>
    <v:vp key="c" type="string"></v:vp>
    <v:vp key="v" type="character"></v:vp>
    <v:vp key="t" type="string"></v:vp>
  </v:cvp>
  <v:combineGroup>
    <v:collect>
      <v:string value="Program"></v:string>
      <v:string value="Counter"></v:string>
      <v:string value="CountVowels"></v:string>
    </v:collect>
    <v:collectGroup>
      <v:combineGroup>
        <v:collect>
          <v:character value="97"></v:character>
          <v:character value="101"></v:character>
        </v:collect>
        <v:collect>
          <v:string value="Great to see you again!"></v:string>
          <v:string value="Another great place"></v:string>
        </v:collect>
      </v:combineGroup>
      <v:combine>
        <v:character value="105"></v:character>
        <v:string value="Have you imagined this?"></v:string>
      </v:combine>
      <v:combine>
        <v:character value="111"></v:character>
        <v:string value="Why should we go around?"></v:string>
      </v:combine>
```

```
      <v:combine>
        <v:character value="117"></v:character>
        <v:string value="Wounderful tube"></v:string>
      </v:combine>
    </v:collectGroup>
  </v:combineGroup>
</v:cvSpec>
```

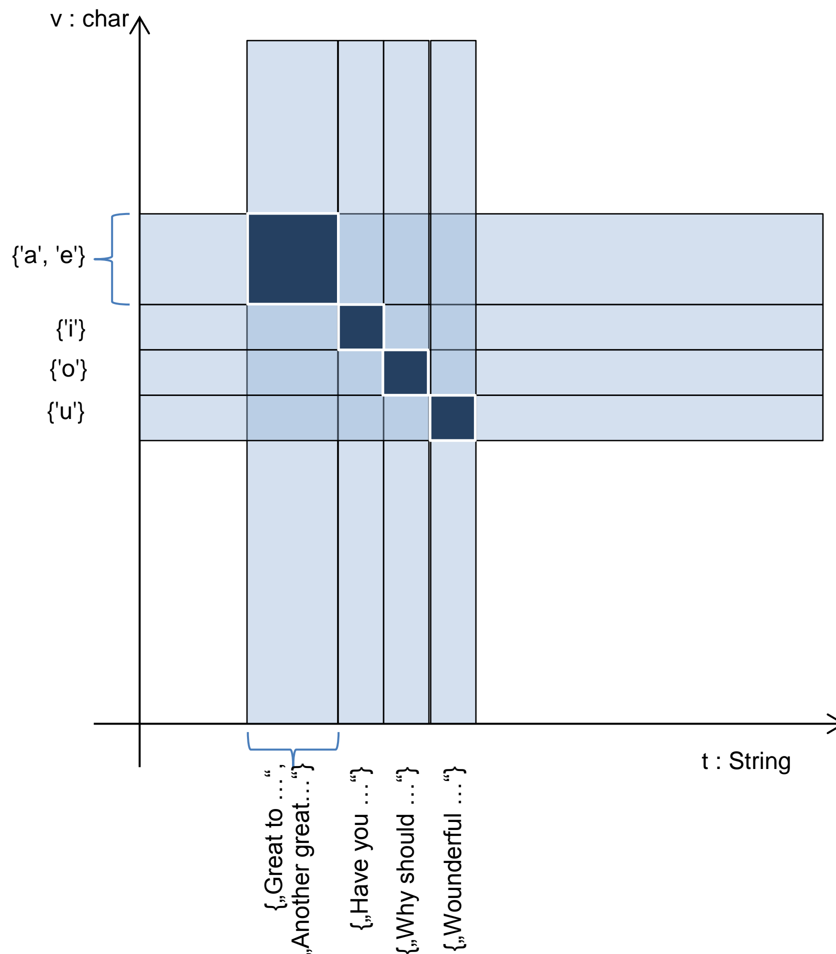


Abbildung 2: Sinnvolle Kombinationen der Variationspunkte v und t in Aufgabe 2

# 2 Definition und Referenzierung

## 2.1 Aufgabe mit nicht-symmetrischen Abhängigkeiten

Wir schauen uns erneut Aufgabe 2 an, gehen nun aber von einer etwas geänderten Anwendungsdomäne aus. Wir definieren die folgenden, leicht abgewandelten sinnvollen Wertkombinationen:

<table>
<tr><th>Variationspunkt</th><th>Datentyp</th><th colspan="2">Sinnvolle Werte</th></tr>
<tr><td>c</td><td>String</td><td colspan="2">Gültiger Java-Bezeichner, der im Aufgabenkontext für Studierende nachvollziehbar ist. Z. B. "Program", "Counter", "CountVowels".</td></tr>
<tr><td>v</td><td>char</td><td colspan="2">Vokale 'a', 'e', 'i', 'o', 'u'.</td></tr>
<tr><td rowspan="6">t</td><td rowspan="6">String</td><td colspan="2">Abhängig vom Wert v sind mehrere sinnvolle Eingaben denkbar.</td></tr>
<tr><td>v</td><td>Einige sinnvolle Werte für t</td></tr>
<tr><td>'a' oder 'e'</td><td>“Great to see you again, boy!”<br>“Another great place”</td></tr>
<tr><td>'i'</td><td>“Have you imagined this?”</td></tr>
<tr><td>'o'</td><td>“Great to see you again, boy!”<br>“Why should we go around?”</td></tr>
<tr><td>'u'</td><td>“Wounderful tube”</td></tr>
</table>

Der t-Wert „Great to see you again, boy!” ist sinnvoll mit den v-Werten 'a', 'e' oder 'o' kombinierbar. Umgekehrt gilt nicht, dass aus einem gegebenen v-Wert aus { 'a', 'e', 'o' } die gleiche Menge sinnvoller t-Werte ableitbar ist.

## 2.2 Definition und Referenzierung

Wollen wir die sinnvollen Wertkombinationen von v und t nun in Gestalt von kartesischen Produkten darstellen, gäbe es zwei Varianten, wobei in beiden Varianten entweder v-Ausprägungen oder t-Ausprägungen redundant erwähnt werden müssen. Die beiden folgenden Definitionen erzeugen die gleiche Gesamtwertemenge. In der ersten Definition wird das Literal „Great to see you again, boy!” wiederholt eingesetzt:

```
CVSpec.build(Vp.s("c"), Vp.c("v"), Vp.s("t"))
    .combineGroup()
        .collect("Program", "Counter", "CountVowels")
        .collectGroup()
            .combineGroup()
                .collect('a', 'e')
                .collect("Great to see you again, boy!", "Another great place")
            .endCombineGroup()
            .combine('i', "Have you imagined this?")
            .combineGroup()
                .val('o')
                .collect("Great to see you again, boy!", "Why should we go around?")
            .endCombineGroup()
            .combine('u', "Wounderful tube")
        .endCollectGroup()
    .endCombineGroup()
.endBuild();
```

Unabhängig davon haben wir eine neue Vokabel „val“ eingeführt, die für eine einelementige Menge genutzt werden kann und deutlicher als die Vokabel „collect“ ausdrückt, dass es sich um einen Einzelwert handelt:

val(v) ⇔ collect(v).

In der folgenden zweiten Definition wird die Wiederholung der Literale für t vermieden. Stattdessen wiederholen sich die Literale 'a', 'e', 'o':

```
CVSpec.build(Vp.s("c"), Vp.c("v"), Vp.s("t"))
   .combineGroup()
      .collect("Program", "Counter", "CountVowels")
      .collectGroup()
         .combineGroup().collect('a','e')    .val("Another great place")         .endCombineGroup()
         .combineGroup().val('i')            .val("Have you imagined this?")     .endCombineGroup()
         .combineGroup().collect('a','e','o').val("Great to see you again, boy!").endCombineGroup()
         .combineGroup().val('o')            .val("Why should we go around?")    .endCombineGroup()
         .combineGroup().val('u')            .val("Wounderful tube")             .endCombineGroup()
      .endCollectGroup()
   .endCombineGroup()
.endBuild();
```

So lange die zu wiederholenden Ausprägungen lediglich einfache Texte oder Zeichen wie in diesem kleinen Beispiel sind, ist die Redundanz überschaubar. Wenn wir uns jedoch unter t und v stattdessen jeweils einen kompletten Teilbaum mit wiederum mehreren verschachtelten collect / combine Operationen vorstellen, entstünde erhebliche Redundanz in der Spezifikation.

Aus diesem Grunde führen wir die Möglichkeit ein, Teilbäume mit einem Bezeichner versehen zu definieren und später wiederzuverwenden. In der folgenden Realisierung wird zunächst eine Variantenmenge unter dem Bezeichner id1 definiert. In diesem Fall ist es eine eindimensionale Menge (Schlüssel „t“) mit einem Element. Es könnte hier jedoch eine beliebig komplexe Menge definiert werden. Die definierte Menge darf als eigene Vp-Schlüssel eine Teilmenge oder alle Vp-Schlüssel des unmittelbar umschließenden Konstrukts angeben. Weiter unten wird die so definierte Menge mit ref(„id1“) an zwei Stellen inkludiert. Eine ref()-Operation sucht nach einem define-Konstrukt entweder unter den vorhergehenden Geschwistern des unmittelbar umschließenden Konstrukts, in dem sich die ref()-Operation befindet, oder unter den jeweils vorhergehenden Geschwistern aller mittelbar umschließenden Konstrukte, wobei die Suche von ref() ausgehend vom innersten zum äußersten Konstrukt verläuft und abbricht, sobald die „id1“ gefunden wurde. Bei mehreren defines muss jedes define einen eindeutigen Schlüssel über das Gesamtkonstrukt besitzen.

```
CVSpec.build(Vp.s("c"), Vp.c("v"), Vp.s("t"))
    .combineGroup()
        .collect("Program", "Counter", "CountVowels")
        .collectGroup()
            .define("id1", "t")
                .val("Great to see you again, boy!")
            .endDefine()
            .combineGroup()
                .collect('a', 'e')
                .collectGroup()
                    .val("Another great place")
                    .ref("id1")
                .endCollectGroup()
            .endCombineGroup()
            .combine('i', "Have you imagined this?")
            .combineGroup()
                .val('o')
                .collectGroup()
                    .ref("id1")
                    .val("Why should we go around?")
                .endCollectGroup()
            .endCombineGroup()
            .combine('u', "Wounderful tube")
        .endCollectGroup()
    .endCombineGroup()
.endBuild();
```

In XML schreiben wir:

```
<?xml version="1.0" ?>
<v:cvSpec xmlns:v="urn:to-be-specified">
  <v:cvp>
    <v:vp key="c" type="string"></v:vp>
    <v:vp key="v" type="character"></v:vp>
```

```
    <v:vp key="t" type="string"></v:vp>
  </v:cvp>
  <v:combineGroup>
    <v:collect>
      <v:string value="Program"></v:string>
      <v:string value="Counter"></v:string>
      <v:string value="CountVowels"></v:string>
    </v:collect>
    <v:collectGroup>
      <v:define id="id1">
        <v:cvp>
          <v:vp key="t" type="string"></v:vp>
        </v:cvp>
        <v:val>
          <v:string value="Great to see you again, boy!"></v:string>
        </v:val>
      </v:define>
      <v:combineGroup>
        <v:collect>
          <v:character value="97"></v:character>
          <v:character value="101"></v:character>
        </v:collect>
        <v:collectGroup>
          <v:val>
            <v:string value="Another great place"></v:string>
          </v:val>
          <v:ref id="id1"></v:ref>
        </v:collectGroup>
      </v:combineGroup>
      <v:combine>
        <v:character value="105"></v:character>
        <v:string value="Have you imagined this?"></v:string>
      </v:combine>
      <v:combineGroup>
        <v:val>
          <v:character value="111"></v:character>
        </v:val>
        <v:collectGroup>
          <v:ref id="id1"></v:ref>
          <v:val>
            <v:string value="Why should we go around?"></v:string>
          </v:val>
        </v:collectGroup>
      </v:combineGroup>
      <v:combine>
        <v:character value="117"></v:character>
        <v:string value="Wounderful tube"></v:string>
      </v:combine>
    </v:collectGroup>
  </v:combineGroup>
</v:cvSpec>
```

# 3 Derive: Ableitung eines Variationspunkts

## 3.1 Eine Aufgabe mit einem anderen dritten Variationspunkt n

Wir ändern die Aufgabe 2 erneut geringfügig ab und lösen den Variationspunkt t durch einen Variationspunkt n ab:

> *Write a `main` method in class `%vp{c}` in the default package that reads a word or a sentence from user input and counts the number of occurrences of the vowel %vp{v} in the input. Your program should count both upper and lower case letters.*
>
> *Example (user input is **highlighted**):*
>
> ```
> Give me some text, please: The quick brown fox jumps over the lazy dog.
> Found %vp{n} %vp{v} in your input!
> ```

Aufgabe 3

Die Variationspunkte sind nun:

| Variationspunkt | Datentyp | Sinnvolle Werte |
|---|---|---|
| c | String | Gültiger Java-Bezeichner, der im Aufgabenkontext für Studierende nachvollziehbar ist. Z. B. "Program", "Counter", "CountVowels". |
| v | char | Vokale 'a', 'e', 'i', 'o', 'u'. |
| n | int | Abhängig vom Wert v ist immer nur ein Wert für n sinnvoll, nämlich genau die Anzahl der Vokale v im Satz " The quick brown fox jumps over the lazy dog." |

Hier können die Variationspunkte v und n erneut nicht unabhängig voneinander gewählt werden. Abbildung 3 zeigt die Menge der sinnvollen Variationspunktwerte für v und n. Für jedes gegebene v gibt es hier nur einen sinnvollen Wert für n. Die Menge aller sinnvollen CVp beschreiben wir textuell z. B. wie folgt:

```
CVSpec.build(Vp.s("c"), Vp.c("v"), Vp.i("n"))
    .combineGroup()
        .collect("Program", "Counter", "CountVowels")
        .collectGroup()
            .combineGroup().collect('a', 'i').val(1).endCombineGroup()
            .combineGroup().val('e')         .val(3).endCombineGroup()
            .combineGroup().val('o')         .val(4).endCombineGroup()
            .combineGroup().val('u')         .val(2).endCombineGroup()
        .endCollectGroup()
    .endCombineGroup()
.endBuild();
```

Oder in XML:

```
<?xml version="1.0" ?>
<v:cvSpec xmlns:v="urn:to-be-specified">
  <v:cvp>
    <v:vp key="c" type="string"></v:vp>
    <v:vp key="v" type="character"></v:vp>
    <v:vp key="n" type="integer"></v:vp>
  </v:cvp>
  <v:combineGroup>
    <v:collect>
      <v:string value="Program"></v:string>
      <v:string value="Counter"></v:string>
      <v:string value="CountVowels"></v:string>
    </v:collect>
    <v:collectGroup>
      <v:combineGroup>
        <v:collect>
          <v:character value="97"></v:character>
          <v:character value="105"></v:character>
        </v:collect>
        <v:val>
          <v:integer value="1"></v:integer>
        </v:val>
      </v:combineGroup>
      <v:combineGroup>
        <v:val>
          <v:character value="101"></v:character>
        </v:val>
        <v:val>
          <v:integer value="3"></v:integer>
        </v:val>
      </v:combineGroup>
      <v:combineGroup>
        <v:val>
          <v:character value="111"></v:character>
        </v:val>
        <v:val>
```

```
          <v:integer value="4"></v:integer>
        </v:val>
      </v:combineGroup>
      <v:combineGroup>
        <v:val>
          <v:character value="117"></v:character>
        </v:val>
        <v:val>
          <v:integer value="2"></v:integer>
        </v:val>
      </v:combineGroup>
    </v:collectGroup>
  </v:combineGroup>
</v:cvSpec>
```

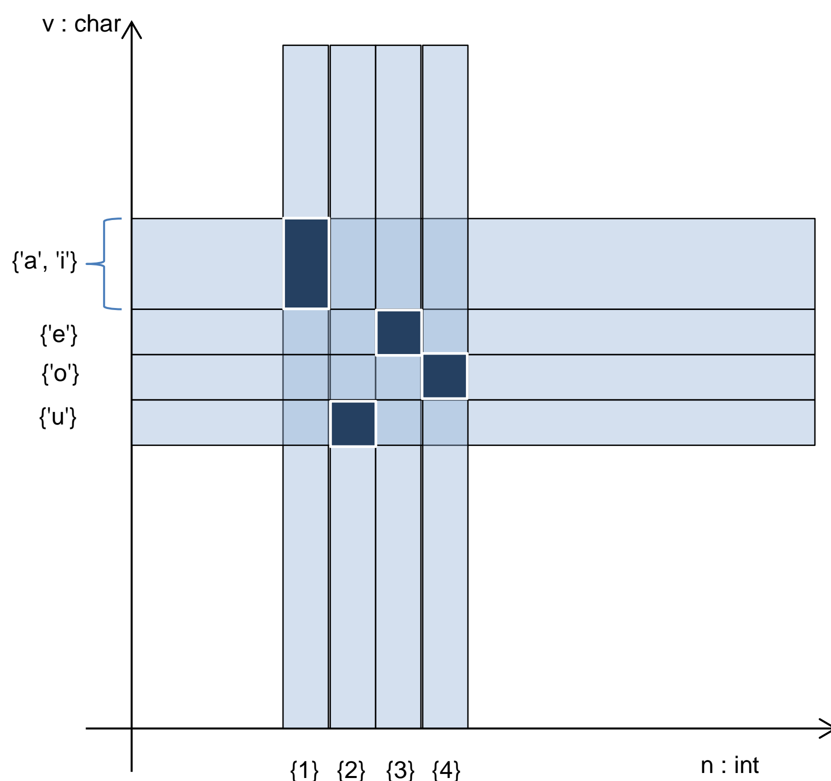


Abbildung 3: Visualisierung der sinnvollen Werte des CVp der Aufgabe 3

## 3.2 Ableitung des Variationspunktes n

Die Aufgabe 3 unterscheidet sich von Aufgabe 1 und Aufgabe 2 dadurch, dass es leicht wäre, die sinnvollen n-Werte automatisch aus den anderen Variationspunktwerten zu berechnen, statt diese umständlich wie hier geschehen, einzeln aufzulisten:

```
CVSpec.build(Vp.s("c"), Vp.c("v"), Vp.i("n"))
    .combineGroup()
        .collect("Program", "Counter", "CountVowels")
        .collectGroup()
            .combineGroup().collect('a', 'i').val(1).endCombineGroup()
            .combineGroup().val('e')         .val(3).endCombineGroup()
            .combineGroup().val('o')         .val(4).endCombineGroup()
            .combineGroup().val('u')         .val(2).endCombineGroup()
        .endCollectGroup()
    .endCombineGroup()
.endBuild();
```

Die Auflistung der sinnvollen v, n-Kombinationen ist mühselig und könnte automatisch berechnet werden.

Wir wünschen uns eine neue Operation, die als ersten Operanden die Menge der sinnvollen v-Werte { 'a', 'e', 'i', 'o', 'u' } entgegen nimmt und als zweiten Parameter eine Berechnungsvorschrift, die aus jedem der gültigen v-Werte die dazugehörigen gültigen n-Werte berechnet. Am Ende soll die neue Operation die Vereinigungsmenge der kartesischen Produkte von jeweils zueinander passenden Teilmengen von v- und n-Werten bilden.

Der entsprechend überarbeitete Code sieht so aus:

```
CVSpec.build(Vp.s("c"), Vp.c("v"), Vp.i("n"))
    .combineGroup()
        .collect("Program", "Counter", "CountVowels")
        .collect('a','e','i','o','u')
        .deriveVal(VpType.INTEGER, jsSrc)
    .endCombineGroup()
.endBuild();
```

wobei `jsSrc` den folgenden Inhalt besitzt:

```
/**
 * Calculates a new variation point value from other variation point values.
 * @param {Object} obj - an object with variation point values
 * @param {String} obj.v - lower case vowel character
 * @returns {Number} number of vowels v in the text "The quick brown fox jumps over the lazy dog."
 */
function apply(obj) {
    "use strict"
    var text= "The quick brown fox jumps over the lazy dog.".toLowerCase();
    var cnt= 0;
    for (var i = 0, len = text.length; i < len; i++) {
        if (text.charAt(i) === obj.v) cnt++;
    }
    return cnt;
};
```

Javascript-Code 1

Die neue Operation namens deriveVal werden wir im folgenden Abschnitt an einem leicht ausgebauten Beispiel einführen. Zu erkennen ist jedoch bereits jetzt, dass die sinnvollen n-Werte durch eine Javascript-Funktion berechnet werden. Der Parameter dieser Funktion ist ein Javascript-Objekt mit den Werten der unmittelbar davor definierten c- und v-Werte. Die Funktion berechnet durch eine Schleife die zugehörige sinnvolle Zahl n. Aus dieser wird dann hinter den Kulissen automatisch eine einelementige Menge mit genau diesem einen sinnvollen n-Wert erzeugt.

## 3.3 Ableitung eines Variationspunktes von mehreren anderen Variationspunkten

### 3.3.1 Ein neuer Variationspunkt w

Wir ändern die Aufgabe 3 erneut geringfügig ab und führen einen weiteren Variationspunkt w ein:

> *Write a `main` method in class `%vp{c}` in the default package that reads a word or a sentence from user input and counts the total number of occurrences of the vowels %vp{v} and %vp{w} in the input. Your program should count both upper and lower case letters.*
>
> *Example (user input is **highlighted**):*
>
> *`Give me some text, please:` **`The quick brown fox jumps over the lazy dog.`***
> *`Found %vp{n} %vp{v} and %vp{w} in your input!`*

Aufgabe 4

Die Variationspunkte sind nun:

| Variationspunkt | Datentyp | Sinnvolle Werte |
|---|---|---|
| c | String | Gültiger Java-Bezeichner, der im Aufgabenkontext für Studierende nachvollziehbar ist. Z. B. "Program", "Counter", "CountVowels". |
| v | char | Vokale 'a', 'e', 'i', 'o', 'u'. |
| w | char | Vokale 'a', 'e', 'i', 'o', 'u'. Eigentlich wäre es sinnvoll, w hier auf Werte außer v zu beschränken und zudem zu fordern, dass w größer als v ist. Wir verzichten zunächst darauf und leben erst einmal mit dem Umstand, dass dadurch in einigen Varianten auch zwei Mal der gleiche Vokal stehen kann. |
| n | int | Abhängig von den Werten v und w ist immer nur ein Wert für n sinnvoll, nämlich genau die Anzahl der Vokale v und w im Satz „The quick brown fox jumps over the lazy dog." |

In diesem Beispiel ist also eine weitere Abhängigkeit eingezogen. Die Menge aller sinnvollen CVp beschreiben wir textuell z. B. wie folgt:

```
CVSpec.build(Vp.s("c"), Vp.c("v"), Vp.c("w"), Vp.i("n"))
    .combineGroup()
        .collect("Program", "Counter", "CountVowels")
        .collectGroup()
            .combine('a', 'a', 1)
            .combine('a', 'e', 4)
            .combine('a', 'i', 2)
            .combine('a', 'o', 5)
            .combine('a', 'u', 3)
            .combine('e', 'a', 4)
            .combine('e', 'e', 3)
            .combine('e', 'i', 4)
            .combine('e', 'o', 7)
            .combine('e', 'u', 5)
            .combine('i', 'a', 2)
            .combine('i', 'e', 4)
            .combine('i', 'i', 1)
            .combine('i', 'o', 5)
            .combine('i', 'u', 3)
            .combine('o', 'a', 5)
            .combine('o', 'e', 7)
            .combine('o', 'i', 5)
            .combine('o', 'o', 4)
            .combine('o', 'u', 6)
            .combine('u', 'a', 3)
            .combine('u', 'e', 5)
            .combine('u', 'i', 3)
            .combine('u', 'o', 6)
            .combine('u', 'u', 2)
        .endCollectGroup()
    .endCombineGroup()
.endBuild();
```

### 3.3.2 Ableitung von n aus v und w

Hier wird nun schnell klar, dass wir den derive-Mechanismus gewinnbringend einsetzen können. Wir erstellen die folgende, leicht zu programmierende Funktion: f(v,w) = n. Hier ist ein möglicher Javascript-Quelltext für diese Funktion:

```
/**
 * Calculates a new variation point value from other variation point values.
 * @param {Object} obj - an object with variation point values
 * @param {String} obj.v - lower case vowel character
 * @param {String} obj.w - another lower case vowel character
 * @returns {Number} number of vowels v or w in the text "The quick brown fox jumps over the lazy dog."
 */
function apply(obj) {
    "use strict"
    var text= "The quick brown fox jumps over the lazy dog.".toLowerCase();
    var cnt= 0;
    for (var i = 0, len = text.length; i < len; i++) {
        var c= text.charAt(i);
        if (c === obj.v || c === obj.w) cnt++;
    }
    return cnt;
};
```

Javascript-Code 2

Man könnte unter Nutzung dieser Funktion schreiben:

```
        CVSpec.build(Vp.s("c"), Vp.c("v"), Vp.c("w"), Vp.i("n"))
            .combineGroup()
                .collect("Program", "Counter", "CountVowels")
                .collect('a','e','i','o','u')
                .collect('a','e','i','o','u')
                .deriveVal(VpType.INTEGER, jsSrc)  // jsSrc ist der Quelltext von oben
             .endCombineGroup()
        .endBuild();
```

Das Element „deriveVal“ bezeichnet einen Automatismus, der aus einem gegebenen Vp-Paar (v,w) einen sinnvollen Wert für n berechnet. Da der durch die Funktion „apply“ berechnete Skalar keine Menge ist und somit eigentlich nichts als Operand in einem kartesischen Produkt zu suchen hat, interpretieren wir das Ergebnis von deriveVal als einelementige Menge, die genau diesen Skalar enthält. Wir werden später sehen, dass es manchmal sinnvoll ist, mit derive… mehrelementige Mengen zu berechnen.

Bisher haben wir mit combineGroup() ein herkömmliches kartesisches Produkt mehrerer Mengen bezeichnet. Hier verwenden wir combineGroup() etwas anders, denn die drei Mengen sind nicht alle vorab bekannt. Die dritte durch deriveVal berechnet Menge hängt jeweils von anderen Werten ab. Tatsächlich berechnet das combineGroup()-Konstrukt hier die Menge aller Tripel (v,w,n), für die gilt: v,w∈{'a','e','i','o','u'}, n= apply(v,w). Dies ist kein klassisches kartesisches Produkt mehr, kann aber weiterhin als Kombinationsoperation dreier Variantenmengen aufgefasst werden.

Die Ableitungsoperation stellen wir in XML wie folgt dar:

```
<?xml version="1.0" ?>
<v:cvSpec xmlns:v="urn:to-be-specified">
  <v:cvp>
    <v:vp key="c" type="string"></v:vp>
    <v:vp key="v" type="character"></v:vp>
    <v:vp key="w" type="character"></v:vp>
    <v:vp key="n" type="integer"></v:vp>
  </v:cvp>
  <v:combineGroup>
    <v:collect>
      <v:string value="Program"></v:string>
      <v:string value="Counter"></v:string>
      <v:string value="CountVowels"></v:string>
    </v:collect>
    <v:collect>
      <v:character value="97"></v:character>
      <v:character value="101"></v:character>
      <v:character value="105"></v:character>
      <v:character value="111"></v:character>
      <v:character value="117"></v:character>
    </v:collect>
    <v:collect>
      <v:character value="97"></v:character>
      <v:character value="101"></v:character>
```

```
      <v:character value="105"></v:character>
      <v:character value="111"></v:character>
      <v:character value="117"></v:character>
    </v:collect>
    <v:derive dataType="integer" aggregateType="value">
      <v:jsSource><![CDATA[/**
 * Calculates a new variation point value from other variation point values.
 * @param {Object} obj - an object with variation point values
 * @param {String} obj.v - lower case vowel character
 * @param {String} obj.w - another lower case vowel character
 * @returns {Number} number of vowels v or w in the text
 *                  The%20quick%20brown%20fox%20jumps%20over%20the%20lazy%20dog.
 */
function apply(obj) {
    "use strict"
    var text= decodeURIComponent("The%20quick%20brown%20fox%20jumps%20over%20the%20lazy%20dog.").toLowerCase();
    var cnt= 0;
    for (var i = 0, len = text.length; i < len; i++) {
        var c= text.charAt(i);
        if (c === obj.v || c === obj.w) cnt++;
    }
    return cnt;
};
]]></v:jsSource>
    </v:derive>
  </v:combineGroup>
</v:cvSpec>
```

## 3.4 Ableitung einer Wertmenge für einen Variationspunkt

Wir können die Aufgabe alternativ umgekehrt modellieren, indem wir die Rollen von v, w und n vertauschen. Nun berechnen wir die sinnvollen n-Werte nicht aus v und w, sondern umgekehrt die sinnvollen w-Werte aus n und v. Für gegebene Werte n und v kann es mehrere passende w-Werte geben. Deshalb berechnet die folgende Javascript-Funktion ein Array als Ergebnis:

```
/**
 * Calculates an array of new variation point values from other variation point values.
 * @param {Object} obj - an object with variation point values
 * @param {Number} obj.n - number of vowels in "The quick brown fox jumps over the lazy dog."
 * @param {String} obj.v - lower case vowel character
 * @returns {String[]} w - The vowels w such that the number of vowels v or w
 *                         in the text "The quick brown fox jumps over the lazy dog." is exactly obj.n
 */
function apply(obj) {
    "use strict"
    var text= "The quick brown fox jumps over the lazy dog.".toLowerCase();
    var vowels= "aeiou";
    var w= [];
    for (var i=0; i<vowels.length; i++) {
        var vowel= vowels.charAt(i);
        var cnt= 0;
        for (var k = 0, len = text.length; k < len; k++) {
            var c= text.charAt(k);
            if (c === obj.v || c === vowel) cnt++;
        }
        if (cnt === obj.n) {
            w.push(vowel);
        }
    }
    return w;
};
```

Javascript-Code 3

Mit dieser Funktion können wir nun definieren:

```
CVSpec.build(Vp.s("c"), Vp.i("n"), Vp.c("v"), Vp.c("w"))
    .combineGroup()
        .collect("Program", "Counter", "CountVowels")
        .range(1,7)
        .collect('a','e','i','o','u')
        .deriveCollect(VpType.CHARACTER, jsSrc) // jsSrc ist der Quelltext von oben
    .endCombineGroup()
.endBuild();
```

Die neue Operation „deriveCollect“ erwartet von der Javascript-Funktion ein Array als Rückgabewert. Das Array wird als Menge der w-Werte interpretiert, die mit den unmittelbar davor angegebenen Werten für c, n, v kombiniert werden können.

Die ebenfalls neue Operation „range“ definiert einen Ganzzahlbereich von 1 bis 7, jeweils inklusive:

rangeI(1,7) ⇔ collect(1,2,3,4,5,6,7).

Als XML-Definition lässt sich die Menge wie folgt beschreiben:

```
<?xml version="1.0" ?>
<v:cvSpec xmlns:v="urn:proforma:variability:v0.1">
  <v:cvp>
    <v:vp key="c" type="string"></v:vp>
    <v:vp key="n" type="integer"></v:vp>
    <v:vp key="v" type="character"></v:vp>
    <v:vp key="w" type="character"></v:vp>
  </v:cvp>
  <v:combineGroup>
    <v:collect>
      <v:string value="Program"></v:string>
      <v:string value="Counter"></v:string>
      <v:string value="CountVowels"></v:string>
    </v:collect>
    <v:range>
      <v:vRange>
        <v:firstInteger value="1"></v:firstInteger>
        <v:lastInteger value="7"></v:lastInteger>
        <v:count>7</v:count>
      </v:vRange>
    </v:range>
    <v:collect>
      <v:character value="97"></v:character>
      <v:character value="101"></v:character>
      <v:character value="105"></v:character>
      <v:character value="111"></v:character>
      <v:character value="117"></v:character>
    </v:collect>
    <v:derive dataType="character" aggregateType="collection">
      <v:jsSource><![CDATA[/**
 * Calculates an array of new variation point values from other variation point values.
 * @param {Object} obj - an object with variation point values
 * @param {Number} obj.n - number of vowels in The%20quick%20brown%20fox%20jumps%20over%20the%20lazy%20dog.
 * @param {String} obj.v - lower case vowel character
 * @returns {String[]} w - The vowels w such that the number of vowels v or w in the text
 *                        The%20quick%20brown%20fox%20jumps%20over%20the%20lazy%20dog. is exactly obj.n
 */
function apply(obj) {
    "use strict"
    var text= decodeURIComponent("The%20quick%20brown%20fox%20jumps%20over%20the%20lazy%20dog.").toLowerCase();
    var vowels= "aeiou";
    var w= [];
    for (var i=0; i<vowels.length; i++) {
        var vowel= vowels.charAt(i);
        var cnt= 0;
        for (var k = 0, len = text.length; k < len; k++) {
            var c= text.charAt(k);
            if (c === obj.v || c === vowel) cnt++;
        }
        if (cnt === obj.n) {
            w.push(vowel);
        }
    }
    return w;
};
]]></v:jsSource>
    </v:derive>
  </v:combineGroup>
</v:cvSpec>
```

## 3.5 Schlüsselumordnungen

Je nach Kontext kann es hilfreich sein, die Namen der Variationspunkte in der ursprünglichen Reihenfolge c,v,w,n beizubehalten, und zwar unabhängig davon, ob n aus v und w berechnet wird oder umgekehrt w aus n und v. Dazu ist es erforderlich, in inneren Teilmengen die Schlüsselreihenfolge explizit anzugeben:

```
CVSpec.build(Vp.s("c"), Vp.c("v"), Vp.c("w"), Vp.i("n")) // original order
    .combineGroup()   // no reordering means: inherit from surrounding set
        .collect("Program", "Counter", "CountVowels")
        .combineGroup("n", "v", "w")     // explicitly reordered keys for the generation of a subset
           .range(1,7)
           .collect('a','e','i','o','u')
           .deriveCollect(VpType.CHARACTER, jsSrc) // jsSrc ist der Quelltext von oben
       .endCombineGroup()
   .endCombineGroup()
.endBuild();
```

Solche Schlüsselumordnungen kann man für combineGroup() und collectGroup() angeben. Fehlt die Schlüsselumordnung, wird die Reihenfolge der Schlüssel des unmittelbar umschließenden Konstrukts geerbt.

Das XML-Konstrukt für die vorstehende Definition lautet:

```
<?xml version="1.0" ?>
<v:cvSpec xmlns:v="urn:to-be-specified">
  <v:cvp>
    <v:vp key="c" type="string"></v:vp>
    <v:vp key="v" type="character"></v:vp>
    <v:vp key="w" type="character"></v:vp>
    <v:vp key="n" type="integer"></v:vp>
  </v:cvp>
  <v:combineGroup>
    <v:collect>
      <v:string value="Program"></v:string>
      <v:string value="Counter"></v:string>
      <v:string value="CountVowels"></v:string>
    </v:collect>
    <v:combineGroup>
      <v:cvp>
        <v:vp key="n" type="integer"></v:vp>
        <v:vp key="v" type="character"></v:vp>
        <v:vp key="w" type="character"></v:vp>
      </v:cvp>
      <v:range>
        <v:vRange>
          <v:firstInteger value="1"></v:firstInteger>
          <v:lastInteger value="7"></v:lastInteger>
          <v:count>7</v:count>
        </v:vRange>
      </v:range>
      <v:collect>
        <v:character value="97"></v:character>
        <v:character value="101"></v:character>
        <v:character value="105"></v:character>
        <v:character value="111"></v:character>
        <v:character value="117"></v:character>
      </v:collect>
      <v:derive dataType="character" aggregateType="collection">
        <v:jsSource><![CDATA[/**
 * Calculates an array of new variation point values from other variation point values.
 * @param {Object} obj - an object with variation point values
 * @param {Number} obj.n - number of vowels in The%20quick%20brown%20fox%20jumps%20over%20the%20lazy%20dog.
 * @param {String} obj.v - lower case vowel character
 * @returns {String[]} w - The vowels w such that the number of vowels v or w in the text
 *                         The%20quick%20brown%20fox%20jumps%20over%20the%20lazy%20dog. is exactly obj.n
 */
function apply(obj) {
    "use strict"
    var text= decodeURIComponent("The%20quick%20brown%20fox%20jumps%20over%20the%20lazy%20dog.").toLowerCase();
    var vowels= "aeiou";
    var w= [];
```

```
    for (var i=0; i<vowels.length; i++) {
        var vowel= vowels.charAt(i);
        var cnt= 0;
        for (var k = 0, len = text.length; k < len; k++) {
            var c= text.charAt(k);
            if (c === obj.v || c === vowel) cnt++;
        }
        if (cnt === obj.n) {
            w.push(vowel);
        }
    }
    return w;
};
]]></v:jsSource>
      </v:derive>
    </v:combineGroup>
  </v:combineGroup>
</v:cvSpec>
```

Die Angabe der Datentypen der Vp in inneren Knoten muss selbstverständlich mit den Datentypen der beim Wurzelknoten angegebenen Datentypen übereinstimmen.

## 3.6 Leere Ableitung

Um das Problem zu beseitigen, dass in Aufgabe 4 v und w gleiche Werte besitzen können, fordern wir nun, dass der Vokal w größer als v sein muss. Dies lässt sich leicht dadurch erreichen, dass wir den Javascript-Quelltext geeignet anpassen:

```
/**
 * Calculates an array of new variation point values from other variation point values.
 * @param {Object} obj - an object with variation point values
 * @param {Number} obj.n - number of vowels in "The quick brown fox jumps over the lazy dog."
 * @param {String} obj.v - lower case vowel character
 * @returns {String[]} w - The vowels w such that the number of vowels v or w
 *                         in the text "The quick brown fox jumps over the lazy dog." is exactly obj.n
 */
function apply(obj) {
    "use strict"
    var text= "The quick brown fox jumps over the lazy dog.".toLowerCase();
    var vowels= "aeiou";
    var w= [];
    for (var i=0; i<vowels.length; i++) {
        var vowel= vowels.charAt(i);
        if (vowel > obj.v) {
            var cnt= 0;
            for (var k = 0, len = text.length; k < len; k++) {
                var c= text.charAt(k);
                if (c === obj.v || c === vowel) cnt++;
            }
            if (cnt === obj.n) {
                w.push(vowel);
            }
        }
    }
    return w;
};
```

Javascript-Code 4

Den restlichen Teil der Definition lassen wir unverändert:

```
        CVSpec.build(Vp.s("c"), Vp.i("n"), Vp.c("v"), Vp.c("w"))
            .combineGroup()
                .collect("Program", "Counter", "CountVowels")
                .range(1,7)
                .collect('a','e','i','o','u')
                .deriveCollect(VpType.CHARACTER, jsSrc) // jsSrc ist der Quelltext von oben
            .endCombineGroup()
        .endBuild();
```

Es kann nun vorkommen, dass das von der Javascript-Funktion berechnete Array leer ist. Dies ist ganz sicher für den Eingabeparameter obj.v === 'u' der Fall. Die deriveCollect-Operation führt in diesem Fall intentionsgemäß zu einer leeren Teilmenge.

## 3.7 Sequenz mehrerer Ableitungen

Wir erstellen eine neue Aufgabenvariante:

> *Write a `main` method in class `%vp{c}` in the default package that reads a word or a sentence from user input and counts the number of occurrences of the vowel `%vp{v}` in the input. Your program should count both upper and lower case letters.*
>
> *Example (user input is **highlighted**):*
>
> ```
> Give me some text, please: The quick brown fox jumps over the lazy dog.
> Found %vp{n} letter%vp{l} %vp{v} in your input!
> ```

Aufgabe 5

Die Variationspunkte sind nun:

| **Variationspunkt** | **Datentyp** | **Sinnvolle Werte** |
|---|---|---|
| c | String | Gültiger Java-Bezeichner, der im Aufgabenkontext für Studierende nachvollziehbar ist. Z. B. "Program", "Counter", "CountVowels". |
| v | char | Vokale 'a', 'e', 'i', 'o', 'u'. |
| n | int | Abhängig vom Wert v ist immer nur ein Wert für n sinnvoll, nämlich genau die Anzahl der Vokale v im Satz „The quick brown fox jumps over the lazy dog." |
| l | String | Abhängig vom Wert n wird hier entweder der leere String „" (für den Singular) oder das Plural-„s" eingesetzt.[5] |

Hier lässt sich der Variationspunkt n vom Variationspunkt v ableiten und der Variationspunkt l ist direkt von n ableitbar. Entsprechend schreiben wir zwei derive-Operationen:

```
CVSpec.build(Vp.s("c"), Vp.c("v"), Vp.i("n"), Vp.s("l"))
    .combineGroup()
        .collect("Program", "Counter", "CountVowels")
        .collect('a','e','i','o','u')
        .deriveVal(VpType.INTEGER, jsSrcN)
        .deriveVal(VpType.STRING, jsSrcL)
    .endCombineGroup()
.endBuild();
```

[5] Im konkreten Beispiel hätte man diesen Variationspunkt auch einsparen können, wenn man die Variationspunkte n, den Text „letter" und den Variationspunkt l einfach zu einem gemeinsamen String-Variationspunkt verschmolzen hätte. Hier soll der zusätzliche Variationspunkt l demonstrieren, dass kaskadierende Ableitungen möglich sind.

Für `jsSrcN` setzen wir den in Javascript-Code 1 abgebildete Quelltext ein. Für den Quelltext `jsSrcL` nutzen wir den folgenden:

```
/**
 * Calculates a new variation point value from other variation point values.
 * @param {Object} obj - an object with variation point values
 * @param {Number} obj.n - number of vowels in %TEXT%
 * @returns {String} l - if n==1, return "" else return the plural "s".
 */
function apply(obj) {
    "use strict"
    if (obj.n === 1) return "";
    return "s";
};
```

Javascript-Code 5

Dieses Beispiel demonstriert, wie mehrere voneinander abhängende Ableitungen umgesetzt werden können.

# 4 Zusammenfassung

In diesem Aufsatz haben wir eine Notation und zugehörige Implementierung gezeigt, wie Variationspunkte und deren Wertebereiche und Randbedingungen für individualisierbare, automatisch bewertete Programmieraufgaben beschrieben werden können. Es wurde demonstriert, wie eine gegebene Java-Klassenbibliothek genutzt werden kann, um die durch die Notation beschriebenen Variantenmengen zu erzeugen und um entsprechende XML-Dateien einzulesen und zu erzeugen.