Karlsruher Institut für Technologie

Programmieren – Wintersemester 2013/14
Forschungsgruppe Verifikation trifft Algorithmik
`http://baldur.iti.kit.edu/programmieren`
Dr. Carsten Sinz · Markus Iser · Florian Merz

Abschlussaufgabe 2

Ausgabe: 03.03.2013 – 21:00
Abgabe: 01.04.2013 – 13:00

Allgemeine Hinweise

- Achten Sie darauf nicht zu lange Zeilen, Methoden und Dateien zu erstellen[1]
- Programmcode muss in englischer Sprache verfasst sein
- Kommentieren Sie Ihren Code angemessen: So viel wie nötig, so wenig wie möglich
- Wählen Sie geeignete Sichtbarkeiten für Ihre Klassen, Methoden und Attribute
- Achten Sie auf fehlerfrei kompilierenden Programmcode[1]
- Halten Sie alle whitespace Regeln ein[1]
- Halten Sie die Regeln zu Variablen-, Methoden und Paketbenennung ein und wählen Sie aussagekräftige Namen[1]
- Halten Sie die Regeln zu Javadoc Dokumentation ein[1]
- Nutzen Sie nicht das default package[1]
- Achten Sie auf die korrekte Sichtbarkeit Ihrer Klassen[1]
- Halten Sie auch alle anderen checkstyle Regeln ein[1]

Ordnung auf der Festplatte

Fritz der Informatiker ist mit Aufräumen beschäftigt. Und als echter Informatiker gehört das Aufräumen der Dateien auf seinem Rechner natürlich auch dazu. Da ihm die Ordnung auf seiner Festplatte sehr am Herzen liegt, erstellt er nun schon seit geraumer Zeit Ordner und verschiebt Dateien. Er ist frustriert, da er bisher mit keiner Systematik zufrieden war und daher immer wieder von vorne anfangen musste. Dabei muss die Küche doch auch noch aufgeräumt werden...

Er hält inne und überlegt, wie eine optimale Ordnerstruktur aussehen könnte. Am Besten wäre es, wenn er die Ordnerstruktur dynamisch anpassen könnte anhand von vorgegebenen Stichworten (Tags). Außerdem sollten Dateien, die er häufig verwendet, schneller erreichbar sein. Also plant er, in einer Datenbank zu jeder Datei eine Liste von Tags abzuspeichern, anhand derer dann eine automatische Ordnerstruktur aufgebaut werden kann. Außerdem erfasst er in der Datenbank, wie häufig er eine Datei schon benutzt hat. Dann überlegt er sich, wie ein Programm aussehen soll, das ihm automatisch eine optimale Ordnerstruktur berechnet.

Fritz schaut auf die Uhr, und stellt fest, dass es schon halb sieben ist. Um sieben Uhr kommt Besuch, jetzt muss er die Küche aufräumen.

Ihre zweite Abschlussaufgabe besteht darin, ihm zu helfen, dieses großartige Programm zum automatischen Erstellen einer Ordnerstruktur zu schreiben.

[1] Der Praktomat wird die Abgabe zurückweisen, falls diese Regel verletzt ist.

Allgemeine Hinweise

Diese Aufgabe ist weniger algorithmisch als das erste Blatt und hat einen Schwerpunkt auf selbständiger Modellierung von Klassen, weshalb es auch diesbezüglich weniger Vorgaben gibt (keine Interfaces, etc.).

Die zweite Abschlussaufgabe besteht darin, eine Liste von Dateien, die mit einer Menge von Meta-Informationen versehen sind (Tags, Häufigkeit des Zugriffs, ...), so in einer baumartige Struktur zu ordnen, dass sie "gut" gefunden werden können. Den berechneten Baum können Sie sich als einen künstlich erzeugten Verzeichnisbaum vorstellen, dessen Ordnernamen sich jeweils aus den Tags ergeben, und dessen Struktur dynamisch nach den Zugriffshäufigkeiten und den Tags berechnet wird. Die Eingabe besteht aus einer Datei, deren Pfad als Kommandozeilenargument Ihrem Programm übergeben wird, und die alle benötigten Daten enthält. Die Ausgabe besteht dann aus dem berechneten Verzeichnisbaum.

A Eingabe

Sie erhalten als Eingabe eine Datei, die zeilenweise eine Liste von Dateien und zugehörige Metadaten spezifiziert. Jede Zeile enthält kommasepariert Einträge, die die Datei beschreiben. Davon ist der erste Eintrag ein eindeutiger Bezeichner der Datei (normalerweise der Dateipfad), der zweite gibt einen der fünf möglichen Dateitypen an: `image`, `audio`, `video`, `text` oder `program`. Der dritte Eintrag gibt an, wie oft auf die jeweilige Datei zugegriffen wurde. Es folgt eine beliebig lange Liste von Tags. Dabei gibt es zwei Arten von Tags: *Labels* (auch *binäre Tags* genannt), die durch ihr Vorhandensein die Zugehörigkeit zu einer bestimmten Gruppe ausdrücken und *mehrwertige Tags*, die aus einem Schlüssel-Wert-Paar bestehen. Mehrwertige Tags enthalten links von einem Gleichheitszeichen einen Tag-Bezeichner und rechts davon einen Wert.

A.1 Beispiel

```
musik/happy.mp3,audio,30,genre=funk,author=Pharrell Williams,fun
dokumente/Abschlussaufgabe 1,program,5,author=me
dokumente/Abschlussaufgabe 2,program,14,author=me,fun
dokumente/FunnyQuotes.txt,text,7,author=various,fun
video/Familienfeier.mpg,video,4,author=me
bilder/Oma.jpg,image,1,family
```

Der ersten Zeile dieser Beispieldatei ist zu entnehmen, dass die Datei `musik/happy.mp3` vom Typ `audio` ist und bereits 30-mal abgespielt wurde. Außerdem ist für diese Datei das Label **fun** gesetzt und es sind zwei weitere mehrwertige Tags vorhanden: **author** mit dem Wert *Pharrell Williams* und **genre** mit dem Wert *funk*.

B Modellierung

Zentrale Elemente der Modellierung dieser Aufgabe sind das Dokument und das Tag.

B.1 Das Tag

Jedes Tag hat einen Bezeichner und einen Wertebereich. Am Wertebereichs kann man drei Arten von Tags unterscheiden.

- *Binäre Tags* (oder *Labels*), diese sind entweder vorhanden oder nicht, was einem Wertebereich von {`defined`, `undefined`} entspricht (z.B. **fun**).
- *Mehrwertige Tags*, die einen von mehreren explizit angegebenen Werten annehmen können (z.B. **Genre** mit Wertebereich {*Classic*, *Jazz*, *Rock*, *House*, *Dubstep*, `undefined`}). Beachten Sie, dass auch mehrwertige Tags implizit immer den Wert `undefined` im Wertebereich enthalten, für den Fall, dass kein Wert definiert wurde (z.B. **Genre** fehlt bei Fotos).

- Zu guter Letzt gibt es noch *numerische Tags*, die einen Spezialfall der mehrwertigen Tags darstellen und deren Wertebereich ganze Zahlen umfasst (z.B. **Length** bei Musikstücken oder ähnliches). Numerische Tags werden teilweise nach fest vorgegebenen Regeln dynamisch in mehrwertige Tags übersetzt (s.u.). Auch hier benutzen wir den Wert `undefined`, um auszudrücken, dass kein Wert angegeben wurde.

Die Tags können anhand ihrer Werte implizit einer dieser drei Arten zugeordnet werden. Dabei gelten die folgenden zusätzlichen Regeln:

- Werte von numerischen Tags sind alle ganze Zahlen im Integer-Bereich
- Werte von mehrwertigen Tags beginnen immer mit einem Buchstaben und sind ansonsten eine Kombination aus Zahlen und (lateinischen) Buchstaben (**und evtl. Leerzeichen**)
- Werte von binären Tags werden nicht explizit angegeben

Alle anderen Werte können nicht zugeordnet werden (d.h. erzeugen einen Fehler).

Tag-Bezeichner müssen eindeutig sein, d.h. es darf keine binären, mehrwertigen oder numerischen Tags geben, die den gleichen Bezeichner haben. Anders ausgedrückt: ein Tag ist entweder binär oder mehrwertig oder numerisch; gemischte Tags sind verboten. Tag-Bezeichner sind außerdem case-insensitive, d.h. z.B. **length**, **Length** und **lenGth** bezeichnen dasselbe Tag. Gültige Tag-Bezeichner beginnen mit einem lateinischen Buchstaben und sind sonst eine Mischung aus lateinischen Buchstaben und Ziffern.[2] Ein explizit angegebener Wert *undefined* ist nicht dasselbe wie der im Aufgabenblatt implizit genutzte Wert `undefined` für Definitionslücken bei Tags. Mehrwertige und numerische Tags dürfen pro Dokument höchstens einmal gesetzt werden, d.h. wenn mehr als einmal das gleiche Tag definiert wird, erzeugt das einen Fehler.

B.2 Das Dokument

Ein Dokument hat einen eindeutigen Bezeichner, das ist in der Regel ein Verzeichnispfad. Jedem Dokument ist außerdem eine Menge von Tags zugeordnet. Dabei darf jedes Tag höchstens einmal pro Datei definiert sein (z.B. nicht mehr als ein **Genre** pro Titel). Außerdem kennt ein Dokument die Anzahl der Zugriffe auf sich selbst (das heißt, wie oft es vom Anwender in der Vergangenheit benutzt wurde).

Es gibt fünf Arten (Typen) von Dokumenten: `image`, `audio`, `video`, `text` und `program`. Je nach Typ lassen Dokumente Ihren Tags bestimmte Sonderbehandlungen angedeihen. Bei dieser Sonderbehandlung werden Tags automatisch umgeschrieben in andere Tags und nur diese modifizierten Tags zum Dokument gespeichert.

Insbesondere bei automatisch erzeugten Tags gilt die Regel, dass ein Tag auf einem Dokument nicht mehrfach definiert sein darf. Bei Bildern dürfen deshalb nicht **size** und **imagesize** gleichzeitig definiert sein.

B.2.1 Image-Dokumente

Size. Existiert für ein Dokument vom Typ `image` ein numerisches Tag **Size** (das die Größe der Datei in Bytes angeben soll), wird dieses nicht hinzugefügt. Stattdessen wird ein neues mehrwertiges Tag **ImageSize** erstellt mit dem Wertebereich {*Icon*, *Small*, *Medium*, *Large*}. Die genauen Übersetzungsregeln können Sie Tabelle 1 entnehmen.

Size	**ImageSize**
$size < 10000$	*Icon*
$10000 \leq size < 40000$	*Small*
$40000 \leq size < 800000$	*Medium*
$800000 \leq size$	*Large*

Tabelle 1: Übersetzungsregeln für **Size** nach **ImageSize** bei Image-Dokumenten.

[2] Regulärer Ausdruck: [a-zA-Z][a-zA-Z0-9]*

B.2.2 Audio-Dokumente

Length. Existiert für ein Dokument vom Typ `audio` ein numerisches Tag **Length**, so wird dies nicht hinzugefügt. Stattdessen wird ein neues mehrwertiges Tag **AudioLength** erstellt mit dem Wertebereich {*Sample*, *Short*, *Normal*, *Long*}. Die genauen Übersetzungsregeln können Sie Tabelle 2 entnehmen.

Length	**AudioLength**
$length < 10$	*Sample*
$10 \leq length < 60$	*Short*
$60 \leq length < 300$	*Normal*
$300 \leq length$	*Long*

Tabelle 2: Übersetzungsregeln für **Length** nach **AudioLength** bei Audio-Dokumenten.

Genre. Existiert für ein Dokument vom Typ `audio` ein mehrwertiges Tag **Genre**, wird dies nicht hinzugefügt. Stattdessen wird ein Tag **AudioGenre** mit gleichem Wert erzeugt und hinzugefügt.

B.2.3 Video-Dokumente

Length. Existiert für ein Dokument vom Typ `video` ein numerisches Tag **Length**, wird dies nicht hinzugefügt. Stattdessen wird ein neues mehrwertiges Tag **VideoLength** erstellt mit dem Wertebereich {*Clip*, *Short*, *Movie*, *Long*}. Die genauen Übersetzungsregeln können Sie Tabelle 3 entnehmen.

Length	**VideoLength**
$length < 300$	*Clip*
$300 \leq length < 3600$	*Short*
$3600 \leq length < 7200$	*Movie*
$7200 \leq length$	*Long*

Tabelle 3: Übersetzungsregeln für **Length** nach **VideoLength** bei Video-Dokumenten.

Genre. Existiert für ein Dokument vom Typ `video` ein mehrwertiges Tag **Genre**, wird dies nicht hinzugefügt. Stattdessen wird ein Tag **VideoGenre** mit gleichem Wert erzeugt und hinzugefügt.

B.2.4 Text-Dokumente

Words. Existiert für ein Dokument vom Typ `text` ein numerisches Tag **Words**, so wird dies nicht hinzugefügt. Stattdessen wird ein neues mehrwertiges Tag **TextLength** erstellt mit dem Wertebereich {*Short*, *Medium*, *Long*}. Die genauen Übersetzungsregeln können Sie Tabelle 4 entnehmen.

Words	**TextLength**
$words < 100$	*Short*
$100 \leq words < 1000$	*Medium*
$1000 \leq words$	*Long*

Tabelle 4: Übersetzungsregeln für **Words** nach **TextLength** bei Text-Dokumenten

Genre. Existiert für ein Dokument vom Typ `text` ein mehrwertiges Tag **Genre** wird dies nicht hinzugefügt. Stattdessen wird ein Tag **TextGenre** mit gleichem Wert erzeugt und hinzugefügt.

B.2.5 Ausführbare Programme

Executable. Jedes Dokument des Typs `program` erhält automatisch ein binäres Tag **executable**, unabhängig davon, ob dies in der Datenbasis spezifiziert ist oder nicht.

C Automatische Gliederung der Dokumentenliste

Wir wollen nun die automatische Untergliederung der Dokumentenliste bzw. den automatisierten Aufbau der Ordnerstruktur näher betrachten. Dabei sollen die Dateien schrittweise anhand der Werte zu einem ausgewählten Tag in "Ordner" einsortiert werden. Dies wird so lange rekursiv wiederholt, bis die einzelnen Teillisten "übersichtlich", d.h. klein genug sind.

Wir bezeichnen im Folgenden die Menge der vorhandenen Dokumente mit D. Jedes Tag t teilt die Menge D in disjunkte Teilmengen basierend auf dem Tag-Wert, der einem Dokument zugeordnet ist. Wir schreiben $t(d) = v$, wenn das Tag t für das Dokument d den Wert v hat. Zum Beispiel gilt für die Datei happy.mp3 aus obiger Dokumentenliste folgendes:

$$\begin{aligned}\mathsf{audiogenre}(\mathsf{musik/happy.mp3}) &= \mathsf{funk}\\ \mathsf{author}(\mathsf{musik/happy.mp3}) &= \mathsf{Pharrell\ Williams}\\ \mathsf{fun}(\mathsf{musik/happy.mp3}) &= \mathsf{defined}\\ \mathsf{family}(\mathsf{musik/happy.mp3}) &= \mathsf{undefined}\end{aligned}$$

Beachten Sie, dass hier Tags, die in der Dokumentenliste verwendet werden, aber nicht für die Datei angegeben sind, den Wert `undefined` bekommen.

Wir können die Liste der Dokumente nun für einen ersten Strukturiertungsschritt anhand eines gegebenen Tags entsprechend seines Wertebereichs in Teilmengen $D_{t=v} = \{d \in D \mid t(d) = v\}$ aufteilen. So können wir zum Beispiel mit dem Tag fun die obige Dokumentenliste in folgende Teilmengen aufteilen:

$$\begin{aligned}D_{\mathsf{fun=defined}} = \{&\mathsf{musik/happy.mp3}, \mathsf{dokumente/FunnyQuotes.txt},\\ &\mathsf{dokumente/Abschlussaufgabe\ 2}\}\\ D_{\mathsf{fun=undefined}} = \{&\mathsf{dokumente/Abschlussaufgabe\ 1}, \mathsf{video/Familienfeier.mpg},\\ &\mathsf{bilder/Oma.jpg}\}\end{aligned}$$

Um verschiedene mögliche Gliederungen der Dokumentenliste (anhand verschiedener Tags) bewerten zu können, verwenden wir die Anzahl der Dokumentenzugriffe in der Vergangenheit. Ziel ist, häufig benötigte Dokumente schnell wiederzufinden, d.h. sie sollten auf einer möglichst niedrigen Ebene der Strukturierungshierarchie zu finden sein. Um dieses Ziel zu erreichen, gehen wir wie folgt vor.

Jedes Dokument d aus einer Liste von Dokumenten D wird mit einer Wahrscheinlichkeit $p(d, D)$ ausgewählt, die sich aus der Anzahl der früheren Zugriffe $z(d)$ und der Summe $Z(D) = \sum_{d \in D} z(d)$ aller Dokumentenzugriffe in der Liste als $p(d, D) = \frac{z(d)}{Z(D)}$ ergibt. So ist zum Beispiel $p(\mathsf{happy.mp3}, D) = \frac{30}{61}$. Wenn sich die zugrunde liegende Liste ändert, so ändert sich auch die Wahrscheinlichkeit, weil sich die Summe aller Dokumentenzugriffe geändert hat. Sei jetzt zum Beispiel die Liste $D_{\mathsf{fun=defined}}$ gegeben, dann ist $p(\mathsf{happy.mp3}, D_{\mathsf{fun=defined}}) = \frac{30}{51}$.

Die Unsicherheit darüber, welches Dokument aus einer gegebenen Liste von Dokumenten D als nächstes ausgewählt wird, kann man messen. Dazu lässt sich der Begriff der *informationstheoretischen Entropie*[3] verwenden. Sie hängt von den Wahrscheinlichkeiten ab, mit denen jedes Element der Liste ausgewählt wird (bzw. in der Vergangenheit ausgewählt wurde). Die Unsicherheit $H(D)$ lässt sich wie folgt berechnen:

```
uncertainty(D) {
  result = 0;
  foreach d ∈ D {
    result -= p(d, D) * log2(p(d, D));
  }
  return result;
}
```

[3] Siehe z.B. `http://de.wikipedia.org/wiki/Entropie_(Informationstheorie)`.

Wenn wir die Menge D über ein Tag t aufteilen und uns für eine Teilmenge $D_{t=v}$ entscheiden, fällt die verbleibende Unsicherheit, d.h. $H(D_{t=v}) \leq H(D)$. Ohne zu wissen, welchen Wert ein Tag annimmt, möchten wir berechnen, über welches Tag wir sinnvollerweise zuerst entscheiden. Das heißt, wir berechnen die *erwartete* verbleibende Unsicherheit, falls wir uns für einen beliebigen Wert eines bestimmten Tags t entscheiden.

Hierzu benötigen wir noch die Wahrscheinlichkeit, mit der ein Dokument aus D in der Teilmenge $D_{t=v}$ liegt. Diese lässt sich anhand

$$p(D_{t=v}, D) = \sum_{d \in D_{t=v}} p(d, D)$$

berechnen. Die Wahrscheinlichkeit, dass **fun** = *defined* ist, ist in obigem Beispiel $p(D_{\textbf{fun}=defined}, D) = \frac{51}{61}$.

Die erwartete verbleibende Unsicherheit nach einer Entscheidung über den Wert des Tags t (mit Wertebereich W_t) wird dann wie folgt berechnet:

```
remainingUncertainty(D, t) {
  result = 0;
  foreach v ∈ W_t {
    result += p(D_{t=v}, D) * uncertainty(D_{t=v});
  }
  return result;
}
```

Jetzt können wir den erwarteten Informationsgewinn beim Aufteilen von D an Tag t berechnen.

```
expectedInformationGain(D, t) {
  return uncertainty(D) - remainingUncertainty(D, t);
}
```

Wir strukturieren die Dokumente also so nach Tags, dass die Unsicherheit darüber, welches Dokument gemeint ist, möglichst schnell fällt, oder anders ausgedrückt, dass der erwartete Informationsgewinn möglichst hoch ist.

C.1 Beispiel

Bei der oben gegebenen Liste von Dokumenten

$$D = \{\ \texttt{musik/happy.mp3},\ \texttt{dokumente/Abschlussaufgabe 1},\ \texttt{dokumente/Abschlussaufgabe 2},\ \texttt{dokumente/FunnyQuotes.txt},\ \texttt{video/Familienfeier.mpg},\ \texttt{bilder/Oma.jpg}\ \}$$

sind die erwarteten Informationsgewinne die folgenden:

$$\begin{aligned}
\text{expectedInformationGain}(D, \textbf{author}) &= 1.49 \\
\text{expectedInformationGain}(D, \textbf{audiogenre}) &= 1.00 \\
\text{expectedInformationGain}(D, \textbf{fun}) &= 0.64 \\
\text{expectedInformationGain}(D, \textbf{executable}) &= 0.89 \\
\text{expectedInformationGain}(D, \textbf{family}) &= 0.12
\end{aligned}$$

D Algorithmus

Suchen Sie zunächst in der gesamten Liste an Dokumenten D nach demjenigen Tag t, für das der Informationsgewinn (`expectedInformationGain(D, t)`) am größten ist. Dabei wird jeweils `undefined` wie ein eigenständiger Wert behandelt. Teilen Sie dann die Liste an dem Tag mit dem maximalen Informationsgewinn, und zwar so, dass Sie pro Wert eine Liste erhalten. Wiederholen Sie diese Prozedur auf den neu entstanden Teillisten solange, bis der erwartete Informationsgewinn für alle verbleibenden Tags kleiner als $\epsilon = 10^{-3}$ ist. Dabei entsteht ein *Baum*, dessen Verzweigungen Entscheidungen über ein bestimmtes Tag und einen Tag-Wert entsprechen. An den Blättern befinden sich "kleine" Listen von Dokumenten, für die sich eine weitere Untergliederung nicht mehr lohnt. Im Beispiel würden wir uns im ersten Schritt für **author** entscheiden.

E Ausgabe

E.1 Erster Teil

Geben Sie schon während der Algorithmus läuft jeweils in einer Zeile die berechneten Informationsgewinne aus, sofern diese größer als $\epsilon = 10^{-3}$ sind. Dabei beginnt jede Zeile mit einem Slash. Danach folgen durch Slash getrennt die bisherigen Entscheidungen über die Tags (das sind durch "=" getrennte Tag–Wert–Paare). Nach einem weiteren Tag folgt das Tag, für das der Informationsgewinn berechnet wurde. Dann folgt nach einem Gleichheitszeichen der erwartete Informationsgewinn für das betrachtete Tag.

E.1.1 Beispiel

```
/author=1,49
/fun=0,64
/audiogenre=1,00
/executable=0,89
/family=0,12
/author=me/fun=0,97
/author=me/executable=0,67
/author=me/fun=undefined/executable=0,99
```

E.2 Zweiter Teil

Geben Sie eine Zeile mit drei Bindestrichen ("---") als Markierung für den Beginn des zweiten Teils der Ausgabe aus. Geben Sie dann eine Liste der Dateien aus, und zwar so, dass in jeder Zeile der vollständige Pfad (Pfadkomponenten durch Slash "/" getrennt) durch den oben berechneten Baum gegeben ist. Geben Sie am Ende jedes Pfades in Gänsefüßchen ("") den Dokumenten-Bezeichner aus.

E.2.1 Beispiel

```
---
/author=Pharrell Williams/"musik/happy.mp3"
/author=undefined/"bilder/Oma.jpg"
/author=me/fun=undefined/executable=undefined/"video/Familienfeier.mpg"
/author=me/fun=undefined/executable=defined/"dokumente/Abschlussaufgabe 1"
/author=me/fun=defined/"dokumente/Abschlussaufgabe 2"
/author=various/"dokumente/FunnyQuotes.txt"
```

F Aufgabenstellung

Überlegen Sie sich eine sinnvolle Java-Modellierung für die Umsetzung der oben beschriebenen Konzepte, insbesondere für Dokumente, Tags und (Strukturierungs-)Baum.

Schreiben Sie ein Java-Programm, das auf der Kommandozeile einen Parameter entgegennimmt, den Namen der Eingabedatei. Ihr Programm soll dann die Eingabedatei lesen und daraus den Strukturierungs-Baum wie oben beschrieben generieren und ausgeben.

Setzen Sie dabei die erlernten Konzepte der objektorientierten Programmierung ein, um Daten und Funktionalität sinnvoll zu strukturieren. Achten Sie auch darauf, die Eingabe zu validieren, indem Sie sich bei jedem Schritt überlegen, welche Annahmen über die Eingaben Sie machen.