Parsen von CSV-Werten

Herr Postel und die komma-separierten Werte

Phillip Ghadir

Wie sieht wartbarer und erweiterbarer Quelltext eigentlich aus? Wir nähern uns der Frage mit einem einfachen Experiment und kommen zu einem interessanten Ergebnis. Sie werden für einen überschaubaren Kontext – wie das Parsen einer CSV-Datei – sehen, was zu einer vermeintlich einfachen Lösung gehört. Mehr dazu gibt es auch auf GitHub.

25469, 311, Autor, 645,
95678, Code, 9875416,
254, Zeile, 5489, 4223,
Wahl, 6589, 24578, 66,
582, 987, Software, 127,
6985, 5414, 32321, Zug,

In diesem Beitrag experimentieren wir damit, wie Code erweiterbar und nachvollziehbar realisiert werden kann. Als Versuchsgegenstand nehmen wir das Parsen von CSV-Dateien [RFC4180]. Dieser Code ist einerseits einfach genug und andererseits bietet er eine ausreichende Komplexität, insbesondere wenn man ganz im Sinne von Postels Gesetz [Postel] eine gewisse Fehlertoleranz ansetzen möchte.

Unseren Versuch beginnen wir mit einer These:

- Expliziter Code ist gut.
- Auf den Punkt gebracht ist gut.
- Eine dem intuitiven Problemverständnis entsprechende Abstraktion ist gut.
- Klare Anhaltspunkte – welcher Teil welche Verantwortung trägt – sind gut.
- Ballast ist schlecht.

Das klingt ein wenig nach „je kürzer der Code desto besser". Aber das ist nicht immer der Fall, wie wir sehen werden.

Die Domäne „CSV"

Eigentlich ist das Parsen von komma-separierten Werten nicht der Rede wert. Das Zerlegen einer Zeichenkette in ihre Bestandteile erfordert typischerweise ein Aufruf von

```
String kommaSeparierteWerte[] = meinString.split(',');
```

Quellcode bei GitHub

Sie finden die kommentierten Quelltexte zu diesem Artikel auf GitHub. Unter `http://github.com/pgh` gibt es das Git-Repository *practitioner-at-javaspektrum*, das die Quelltexte zu diesem Artikel enthält. Wer kein Git verwenden möchte, kann dort den Quelltext über die Weboberfläche ansehen. Sollten sich in den Code Fehler oder aber klare Don'ts eingeschlichen haben, bitte ich um eine kurze E-Mail oder besser noch um entsprechende Pull-Requests oder um das Erstellen eines Tickets. (Ein entsprechendes Issue-Tracking ist bei GitHub direkt integriert und lässt sich ebenfalls über die Weboberfläche verwenden.)

Das Projekt erfordert keine spezielle Entwicklungsumgebung. Die Software sollte leicht abgerufen und lokal in einer Java-IDE Ihrer Wahl kompiliert werden können.

Diese eine Zeile verrichtet ihren Dienst perfekt, wenn die folgenden Voraussetzungen erfüllt sind:

- die Zeichenkette (`meinString`) enthält genau eine Zeile komma-separierter Werte und
- das Trennzeichen (der Separator) kommt nicht im Nutzinhalt vor.

Das Aufsplitten von komma-separierten ganzen Zahlen bzw. Aufzählungswerten ist also prädestiniert für den Einsatz von `String#split( char )`.

Sehr häufig schreibe ich für Dinge, in denen ich einfach nur eine Matrix zerlegen muss:

```
String rows[] = matrix.split( Character.LINE_SEPARATOR );
for ( String value : rows.split(',') ) {
  // do something important with that value
}
```

Und die Welt hört auf, so einfach zu sein, sobald der CSV-Exportierende beliebige Zeichenketten als Werte zulässt. In diesem Fall suchen wir nach einer Alternative.

CSV-Parser

Es stehen verschiedene Varianten mit unterschiedlichen Funktionsmerkmalen zur Auswahl. Es folgt eine kleine – garantiert nicht vollständige – Liste an Optionen:

- Verwendung eines CSV-Eh-schon-da-Features einer verwendeten Bibliothek – wie z. B. von der H2-Database RDBMS [H2DB]
- Verwendung einer spezifischen CSV-Bibliothek wie z. B. Apache Commons CSV [CCSV]
- Eigenbau mit Hilfe von ANTLR oder anderer Parser-Generatoren [ANTLR]
- Do it yourself

ANTLR ist in der letzten Ausgabe beschrieben worden [Nack11]. Es gibt eine Grammatik für das Parsen von CSV-Dateien unter [ANTLR-CSV].

Die Verwendung der H2-Datenbank-Funktionalität für das Einlesen von CSV-Dateien funktioniert einfach. H2 ist ein performantes, in Java geschriebenes relationales Datenbank-Management-System. Man kann H2 als eigenständigen Server verwenden oder in eigene Systeme einbetten. Alles, was zur Ausführung benötigt wird, ist in einem circa 1 MB großen Java-Archiv namens `h2*.jar` enthalten.

Wenn man also CSV-Dateien hat, die problemlos von H2 gelesen werden können, ist alles gut. Wenn das CSV-Format allerdings nicht passt, muss es erst passend gemacht werden,

bevor das Einlesen funktionieren kann. Die Strategie für das CSV-Parsen kann man nicht einfach austauschen.

Es bleiben uns in diesem Artikel also die Optionen Apache Commons CSV und Do it yourself.

Verwendung des Apache Commons CSV

Stand 20.5.2011 ist das Projekt noch in der Apache Commons Sandbox. Es ist also noch nicht freigegeben. Um es dennoch einzusetzen, können wir den Nightly Build verwenden oder müssen es selbst per

```
svn co http://svn.apache.org/repos/asf/commons/sandbox/csv/trunk/
```

auschecken und zusammenpacken.

Die Verwendung ist simpel: Man instanziiert einen `org.apache.commons.csv.CSVParser` mit einer `java.io.Reader`-Instanz für den Input als Argument und ruft dann z. B. `CSVParser#getAllValues()` auf. Alternativ kann man auch über die einzelnen Werte iterieren.

Der CSV-Parser ist über eine Strategie gut anpassbar. So kann man zum Beispiel Trenner oder das Maskierungszeichen anpassen, wenn dies erforderlich sein sollte.

Sollte der Input allerdings nicht der Art von CSV entsprechen, welche die Bibliothek erwartet, ist man gezwungen, auch diesen CSV-Parser umzuprogrammieren.

Do it yourself Parser mit FSM in „Wild-West-Manier“

Kommen wir zum Hauptteil dieses Artikels. Mit dem manuell geschriebenen Parser sind wir in der Lage, beliebige Varianten von CSV-Text einzulesen und beliebige Anforderungen umzusetzen. An dieser Stelle schlagen wir die Brücke zu Postels Gesetz, das auch als Allgemeines Robustheitsprinzip bekannt ist:

- „Be conservative in what you do, be liberal in what you accept from others.“ [RFC761]

Um den grundsätzlichen Anforderungen (siehe Kasten „CSV – Die Problemdomäne“) gerecht zu werden, verschaffen wir uns einen Überblick über die verschiedenen Betriebsarten, in denen der Parser ein Zeichen lesen kann. Abbildung 1 zeigt den Zustandsautomaten für die Verarbeitung von CSV-Listen. Wenn man genau hinsieht, stellt man fest, dass der Zustandsraum komplex ist und von zwei Zustandsvariablen abhängt: *Maskiert? (ja/nein)* und *InAnführungszeichen? (ja/nein)*. Dieser Zustandsraum ist in dem deterministischen Endlichen Automaten (Finite-State Machine, FSM) ausmultipliziert, sodass wir auf vier zu unterscheidende Zustände schauen.

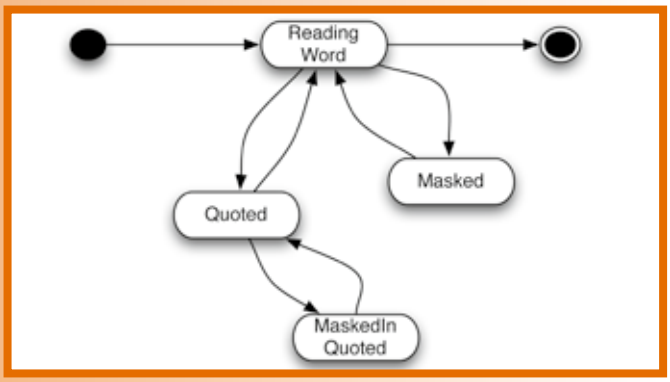

Abb. 1: Zustandsübersicht des FSM-basierten CSV-Parsers

CSV – Die Problemdomäne

Es existiert ein Request for Comments, der das CSV-Format definiert [RCF4180]. Allerdings ist das im RFC beschriebene Format selten anzutreffen. Dafür gibt es aber – je nach Reifegrad der Software, die die CSV-Datei erzeugt – unterschiedlichste und recht kreative Interpretation von CSV.

Nach RFC 4180 kann man wie üblich Werte in Anführungszeichen einschließen und darf in Anführungszeichen auch Zeilenumbrüche aufführen. Der RFC legt fest, dass eine Zeile durch CarriageReturnLineFeed (CRLF) begrenzt wird und dass innerhalb von Anführungszeichen Anführungszeichen durch Anführungszeichen maskiert werden. Klar, oder?

Eine Konvention besagt, dass alle Zeilen die gleiche Anzahl von Werten enthalten. Manchmal trifft man aber CSV-Dateien, in denen in Spalte 1 ein Typ-Diskriminator steht, auf dessen Basis CSV-Produzent und -Abnehmer wissen, wie viele Felder in welcher Reihenfolge darauf folgen. Dann enthalten solche CSVs nicht immer in jeder Zeile die gleiche Anzahl von Werten.

Wird die Konvention eingehalten, CSVs mit jeweils gleich vielen Werten zu liefern, geht man hier von Zeilen eines Typs aus. Manchmal enthalten die CSV-Dateien dann in der ersten Zeile eine Kopfzeile, welche die Spaltennamen definiert. Manchmal aber auch nicht. Der RFC gibt dafür ein optionales Header-Feld an, mit dem der MIME-Typ parametrisiert werden kann.

Viele CSV-Produzenten erzeugen CSVs, die nicht mit Komma, sondern mit einem anderen Trenner (z. B. Semikolon oder Pipe) getrennt werden. Mal mit CRLF und mal mit CR. Viele verwenden als Maskier-Zeichen einen Backslash, wie man es aus Java-Strings auch kennt.

Orientieren wir uns an dem komplexen Zustand bestehend aus zwei Variablen, können wir relativ leicht die Scan-Funktion schreiben:

```
public Collection<String> readline(String line) {
  boolean isQuoted = false;
  boolean isMasked = false;

  StringBuilder sb = new StringBuilder();
  List<String> result = new ArrayList<String>();

  for (char ch : line.toCharArray()) {
    if ( isMasked ) {
      sb.append( ch );
      isMasked = false;
    } else if ( ch == '\\' ) {
      isMasked = true;
    } else if ( isQuoted && ch != '"' ) {
      sb.append( ch );
    } else if ( ch == '"' ) {
      isQuoted = !isQuoted;
    } else if ( ch == ';' ) {
      result.add( sb.toString() );
      sb.setLength(0);
    } else {
      sb.append( ch );
    }
  }
  result.add( sb.toString() );
  return result;
}
```

Der Ablauf lässt sich einfach zusammenfassen:

- Gehe Zeichen für Zeichen durch die Zeichenkette.
- Wenn gerade der Modus *„Maskiert? = ja“* ist, hänge das aktuelle Zeichen einfach an und deaktiviere den Modus *Maskiert?*.
- Ist man nicht im *Maskiert?*-Modus, wird geprüft, ob das Maskier-Zeichen kommt. Damit wird der *Maskiert?*-Modus aktiviert.
- Sonst wird geprüft, ob man im Modus *InAnführungszeichen?* ist. In dem Fall wird für Anführungszeichen der Modus beendet und sonst einfach das Zeichen eingefügt.
- Ist man nicht im *Maskiert?*-Modus und wird ein Anführungszeichen gelesen, wird der Modus *InAnführungszeichen?* invertiert.
- Wenn weder *Maskiert?* noch *InAnführungszeichen?* aktiv ist, führt ein Semikolon dazu, dass das aktuell gelesene Wort als vollständig erkannt wird.
- Sonst wird jedes Zeichen einfach an das aktuell gelesene Wort angefügt.

Alternative FSM-Implementierung mit Enums

Eine alternative Implementierung wollen wir auf Basis von Aufzählungen realisieren. Abbildung 2 skizziert die Struktur.

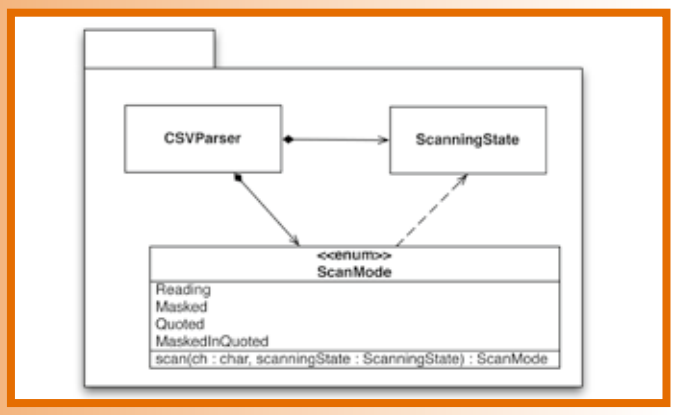

Abb. 2: Struktur des Enum-basierten CSV-Parsers

```
public Collection<String> readline(String line) {
  ScanningState scanningState = new ScanningState();
  ScanMode s = ScanMode.ReadingWord;
  for ( char ch : line.toCharArray() ) {
    s = s.scan( ch, scanningState );
  }
  return scanningState.getResult();
}
```

Jeder Aufzählungswert von **ScanMode** überschreibt die Methode:

```
ScanMode scan( char, ScanningState );
```

Beginnend mit dem initialen Status „ReadingWord“ wird jetzt für jedes Zeichen in der übergebenen Zeichenkette die Methode **scan** auf dem aktuellen Modus-Objekt aufgerufen. Der aktuelle Modus wird auf Basis des **scan**-Rückgabewertes definiert. Wir sehen, die Schleife ist denkbar einfach:

```
enum ScanMode {
  ReadingWord() {
    @Override
    public ScanMode scan(char ch, ScanningState scanningState) {
      switch ( ch ) {
        case '"': return Quoted;
        case '\\' : return Masked;
        case ';' :
          scanningState.result.add( scanningState.sb.toString() );
          scanningState.sb.setLength( 0 );
          return ReadingWord;
        case '\n' :
          scanningState.isLineComplete = true;
          return LineEnd;
        default:
          scanningState.sb.append( ch );
          return ReadingWord;
      }
    }},
  LineEnd() {
    @Override
    public ScanMode scan(char ch, ScanningState scanningState) {
      if ( ch == '\n' || ch == '\r' ) {
        scanningState.isLineComplete = true;
        return LineEnd;
      } else {
        scanningState.isLineComplete = false;
        return ReadingWord.scan(ch, scanningState);
      }
    }
  },
  Masked() {
    @Override
    public ScanMode scan(char ch, ScanningState scanningState) {
      scanningState.sb.append( ch );
      return ReadingWord;
    }},
  Quoted() {
    @Override
    public ScanMode scan(char ch, ScanningState scanningState) {
      switch ( ch ) {
        case '"':
          return ReadingWord;
        case '\\':
          return MaskedInQuoted;
        default:
          scanningState.sb.append( ch );
          return Quoted;
      }
    }},
  MaskedInQuoted() {
    @Override
    public ScanMode scan(char ch, ScanningState scanningState) {
      scanningState.sb.append( ch );
      return Quoted;
    }};

  public abstract ScanMode scan(char ch, ScanningState scanningState);
}
```

Betrachten wir die Struktur des Codes, stellen wir fest, dass das Wissen über den Zustandsautomaten einfach codiert ist: Jeder Zustand ist ein Aufzählungswert in der entsprechenden Enum **ScanMode** geworden. Die Enum deklariert für alle zugehörigen Werte die Funktion **scan**, die für die Verarbeitung von Ereignissen (in diesem Falle: „ein Zeichen gelesen“) zuständig ist und in jedem Aufzählungswert definiert wird. Abbildung 3 stellt stereotypisch die Struktur dar.

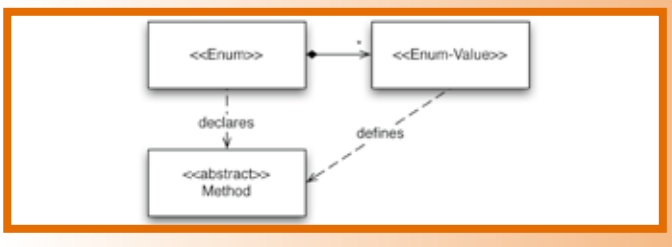

Abb. 3: Verwendung von Enums zur Strukturierung von Funktionalität

Metrik	FSM mit Enums	Wild-West-Manier
Anzahl der Code-Zeilen	313 Zeilen	73 Zeilen
Anzahl der Klassen	3 Klassen	1 Klasse
Anzahl der Ausführungspfade	12	7
Maximale Anzahl der Ausführungspfade in einer Methode	5	7
Delta zwischen Problemanalyse und Code	direkte Entsprechung	Interpretation der Wechselwirkungen erforderlich
Laufzeitkomplexität	O(n)	O(n)

Tabelle 1: Gegenüberstellung der CSV-Parser-Lösungsansätze

Welche Implementierung ist besser?

Bei dem direkten Vergleich zwischen den beiden vorgestellten Ansätzen für das triviale Problem des CSV-Parsens kommen wir auf die in Tabelle 1 zusammengestellten Kennzahlen.

Die Beurteilung, welcher Code der tragfähigere ist, fällt nicht leicht. Einerseits lässt sich das Parsen von CSV-Dateien mit einer einfachen Methode in einer kompakten Klasse realisieren. Dank der Problemdomäne können wir von einer marginalen Fehleranfälligkeit ausgehen.

Hingegen erlaubt die Implementierung, die sich auf die kanonische Abbildung von Zustandsautomaten auf manuell zu erstellende Implementierung stützt, eine Anpassung der Software an neue Anforderungen, ohne nach erfolgreicher Problemanalyse auch noch den Code analysieren zu müssen.

Auch die Arten von Fehlern, die wir bei zukünftigen Anpassungen der FSM-basierten Implementierung erwarten können, lassen sich leichter vorhersagen:

- Transitionsfehler: Wir identifizieren eine Transition nicht korrekt (Fehler in der switch-/case-Kaskade im zugehörigen Anfangszustand).
- Wir führen die erforderlichen Aktionen für eine Transition nicht korrekt aus.
- Nach der Ausführung der Transition wird der falsche Folgezustand zurück geliefert.

Da die Fehlerursache direkt lokalisiert werden kann, braucht man im Fehlerfalle den Quelltext nicht komplett zu betrachten. Selbst wenn der CSV-Parser nur in kompilierter Form vorläge, wäre die Variante mit den Enums von Außen leicht über aspektorientierte Ansätze zu beobachten. Die Implementierung der einfachen Parser-Methode bietet dafür keinerlei Anknüpfungspunkte.

Fazit

Trotz der zahlreichen verfügbaren Frameworks für das Parsen von CSV-Daten haben wir zwei Ansätze gegenüber gestellt, wie wir selbst eine solche Lösung implementieren könnten. Dank der Vielzahl unterschiedlicher Interpretationen des CSV-Formats ist das Szenario CSV-Parser sogar nicht einmal vollständig an den Haaren herbei gezogen.

Wir haben gesehen, dass der verwendete Ansatz, Zustandsautomaten kanonisch auf den Code abzubilden, potenziell zu umfangreicherem Quellcode führt. Allerdings lassen sich Änderungsanforderungen an der Funktionalität sehr einfach implementieren, sofern das Problem verstanden ist, da eine triviale Abbildung von der Struktur des Endlichen Automaten auf die Struktur der Implementierung existiert.

An welchen Stellen der vorgestellte und aus Sicht des Autors recht tragfähige Ansatz die Wartbarkeit eher verschlechtert, haben wir hier nicht untersucht. Sicherlich bestimmt die Größe des Endlichen Automaten (sprich die Anzahl der Transitionen sowie die Anzahl der Zustände) über das Volumen der Implementierung. Ich wage dennoch die These, dass nicht mehr Code, sondern mehr Fallunterscheidungen und Wechselwirkungen die Wartung erschweren und nicht einfach die Anzahl der Zeichen im Quelltext.

Insbesondere für etwas komplexere Fälle als den hier vorgestellten CSV-Parser überwiegt der Nutzen des Aufzählungswert-basierten Ansatz im Vergleich zur erforderlichen Infrastruktur (vgl. Abb. 2 und 3).

Links

[ANTLR] Homepage des Parser-Generators ANother Tool for Language Recognition, `http://antlr.org/`
[ANTLR-CSV] Nathaniel Harward, CSV Grammar, `http://www.harward.us/~nharward/antlr/csv.g`
[CCSV] Apache Commons CSV (Apache Commons Sandbox), `http://commons.apache.org/sandbox/csv/`
[H2DB] H2 Database, `http://www.h2database.com`
[Nack11] K. Nacke, DSL ganz einfach mit ANTLR, in: JavaSPEKTRUM, 03/2011
[Postel] Postels Gesetz (Wikipedia), `http://de.wikipedia.org/wiki/Postels_Gesetz`
[RFC761] J. Postel (Editor), Transmission Control Protocol (TCP), Request for Comment 761, The Internet Engineering Task Force (IETF), 1980, `http://tools.ietf.org/html/rfc761`
[RFC4180] Y. Shafranovich, Common Format and MIME Type for Comma-Separated Values (CSV) Files, Request for Comment 4180, The Internet Engineering Task Force (IETF), 2005, `http://tools.ietf.org/html/rfc4180`
[SCXML] Apache Commons SCXML (State Chart XML), `http://commons.apache.org/scxml/`

Weiterführende Information

https://github.com/pgh/practitioner-at-javaspektrum

Phillip Ghadir baut am liebsten tragfähige, langlebige Softwaresysteme. Er ist Mitglied der Geschäftsleitung bei innoQ und hat sich früh auf Architekturen für verteilte, unternehmenskritische Systeme spezialisiert. Darüber hinaus ist er Mitbegründer und aktives Mitglied des iSAQB, des International Software Architecture Qualification Board.
E-Mail: phillip.ghadir@innoq.com