PROGRAMMIERMETHODEN: 6. Aufgabenblatt

SS 2012
Prof. Dr.-Ing. C. Gips

FH Bielefeld
University of
Applied Sciences

RegExps, Logging, Testen, Versionierung

Abgabe: KW24 (11. – 15. Juni 2012)
Punkte: 30

In dieser Aufgabe soll schrittweise ein Programm zum Syntax-Highlighting für (einfachen) Java-Quelltext entstehen. Sie kennen dies von Entwicklungsumgebungen wie beispielsweise Eclipse oder auch von Editoren. Weiterhin sollen Sie Ihr Programm systematisch testen und für die Entwicklung die Versionsverwaltung Git nutzen.

Lesen Sie dieses Blatt bitte bis zum Ende durch, bevor Sie mit der Bearbeitung beginnen.

Zum Erreichen der vollen Punktzahl muss Ihr Code den wichtigsten **Grundregeln von sauberer Programmierung** genügen, d.h.

- **jedes** mit `public` markierte Element (Klassen, Methoden, Attribute) muss mit einem JavaDoc-Kommentar versehen sein, der Auskunft über den Sinn und das Verhalten der Klasse/Methode/Attribut gibt und der die Inputparameter und den Rückgabewert beschreibt,
- **alle anderen** Klassen/Methoden müssen einen normalen Java-Kommentar haben, der den Zweck der Methode kurz beschreibt,
- für die Namen von Klassen, Methoden, Variablen/Konstanten wurden "sprechende" Namen verwendet,
- alle Parameter der Methoden werden auf Gültigkeit (Wertebereich) geprüft (falls möglich), und
- es wird sinnvoll mit Exceptions umgegangen, d.h. entweder wird eine Exception am Ort des Auftretens behandelt (falls dies möglich und sinnvoll ist) oder an die aufrufende Methode weitergereicht, um dort behandelt oder weitergereicht zu werden. Als "Behandlung" ist in der Regel nicht ausreichend, einfach nur einen Stacktrace oder eine Meldung auf `System.out` auszugeben! Falls Sie vergessen haben, wie dies geht, können Sie das in der Semesterliteratur nachlesen.

1. Aufgabe (8 Punkte): Reguläre Ausdrücke

Im ersten Schritt muss der Java-Quelltext in passende Token aufgeteilt werden. Ein solches Programm wird auch Lexer genannt und stellt die erste Phase eines Parsers dar.[1]

Beispiel: Der Java-Quellcode

```
// total irrer Kommentar
public void myMethod(String s) {}
```

enthält die Token `total irrer Kommentar`, `public`, `void` und `myMethod(String s) {}`.

Ihr Lexer erhält den zu untersuchenden Quellcode über einen Stream, beispielsweise über einen `BufferedReader`. Er muss dann schrittweise mit Hilfe geeigneter **regulärer Ausdrücke** prüfen, ob ein bestimmtes Token am Streamanfang vorliegt. Falls dies so ist, wird die gefundene Zeichenkette aus dem Stream entfernt und als entsprechendes Token-Objekt in einer Liste gespeichert. Dies wird solange wiederholt, bis der Stream leer ist. Dann wird die Liste der erkannten Token zurückgeliefert.
Falls kein Token gefunden wird und der Stream noch nicht vollständig abgearbeitet ist, muss ein passender Fehler geworfen werden.

[1] Der nächste Schritt wäre dann der Aufbau eines Syntaxbaumes und das Prüfen der korrekten Syntax. Dies ist zwar noch viel spannender, geht aber leider deutlich über Ihre aktuellen Kenntnisse in Theoretischer Informatik hinaus.

Ihr Lexer muss folgende Token unterscheiden und erkennen:

- einzeilige Kommentare (beginnend mit `//` bis zum Zeilenende)
- mehrzeilige Kommentare (alles zwischen `/*` und `*/`)
- Java-Doc-Kommentare (alles zwischen `/**` und `*/`)
- Strings (alles zwischen `"` und `"`)
- Zeichen (alles zwischen `'` und `'`), zu behandeln wie Strings
- Java-Schlüsselwörter (siehe unten)
- einfache Annotationen (ohne Parameter o.ä.), d.h. Zeichenketten beginnend mit einem `@`, welches auf eine Wortgrenze folgt, bis zur nächsten Wortgrenze
- sonstigen Text, d.h. allen restlichen Text zwischen zwei Token der eben erklärten Art

Als Java-Schlüsselwörter müssen Sie mindestens implementieren: import, class, public, private, final, static, return, if, else, while, try, catch, finally. Es steht Ihnen frei, Erkennungsroutinen für weitere Schlüsselwörter zu implementieren.

Für jedes Token gibt es einen entsprechenden regulären Ausdruck, der die Zeichenkette erkennt. Bilden Sie die Token über eine Klassenhierarchie mit gemeinsamer abstrakter Klasse ab. Welche Schnittstelle muss diese aufweisen?

Beachten Sie, daß die Token untereinander nicht gleichberechtigt sind. Kommentare und Strings können zwar andere Token (beispielsweise andere Kommentare oder Java-Schlüsselwörter) prinzipiell enthalten, diese sollen aber dann nicht als eigene Token erkannt werden. Der Lexer muss beim Abarbeiten des Streams darauf achten.
Beispiel: Der Java-Quellcode

```
// total irre: public void myMethod(String s) {}
```

enthält nur das Token `total irre: public void myMethod(String s)` (Typ: einzeiliger Kommentar).

Realisieren Sie **eigene Exception-Klassen**, die Sie nutzen, um dem Lexer mitzuteilen, daß eine Tokenklasse keinen Match gefunden hat (also gerade nicht zuständig ist) bzw. die Sie im Lexer nutzen, falls kein Token gefunden werden konnte und der Stream noch nicht leer ist. Wählen Sie eine geeignete Oberklasse.

Hinweis: Wer sich unterfordert fühlt, kann zusätzlich versuchen, **Annotationen innerhalb von Kommentaren** als eigenständiges Token zu erkennen.

Beachten Sie: Hier geht es vor allem um den Umgang mit regulären Ausdrücken. Nutzen Sie also nicht irgendwelche Lexer-/Parser-/Scanner-Generatoren o.ä., sondern implementieren diesen Lexer "zu Fuß"!

---

2. Aufgabe (2 Punkte): Highlighting

Erweitern Sie die Token-Klassen um eine Methode, die den Inhalt des Tokens als formatierten String zurückliefert.

Nutzen Sie für die **Formatierung HTML-Tags**. Beispielsweise könnten Java-Schlüsselwörter fett und dunkelrot formatiert werden, Kommentare kursiv und in einem dunklen Grau. Die einzelnen Tokenarten sollen sich in ihrer Formatierung deutlich voneinander unterscheiden; die verschiedenen Kommentare können die gleiche Formatierung nutzen.

Beispiel (Schlüsselwort `public` in rot+fett):

```
<font color="red"><b>public</b></font>
```

Verweis auf Textformatierung mit HTML:

- `selfhtml.org: Schrift ändern`
- `selfhtml.org: physische Auszeichnung`

---

3. Aufgabe (3 Punkte): GUI mit Swing

Finden Sie heraus, welche von `JTextComponent` abgeleiteten Klassen HTML-Dokumente darstellen können und die Bearbeitung der dargestellten Inhalte ermöglichen.

Bauen Sie eine GUI auf, deren zentrales Element eine Instanz dieser Klasse ist. Das Element soll auf Textänderungen reagieren, indem nach einer Änderung der **aktuelle Textinhalt als Stream an den Lexer** aus dem vorigen Aufgabenteil übergeben wird und aus der resultierenden Liste von Token jeweils die **formatierten Stringrepräsentationen** abgerufen werden.
Diese Strings müssen in ein minimales HTML-Gerüst eingebettet werden und können dem Element als neuer Text gesetzt werden. Verweis auf Aufbau einer HTML-Datei: `selfhtml.org: Grundgerüst`. Ihre durch die Token erzeugten HTML-Schnipsel gehören zwischen die Tags `<body>` und `</body>`.

Da das Element HTML-formatierten Text interpretieren und formatiert darstellen kann, haben Sie so ein einfaches Syntax-Highlighting realisiert.

Tipp: Überlegen Sie, wie Sie aus einem String einen `BufferedReader` erzeugen können, den Ihr Lexer als Input benötigt.

---

4. Aufgabe (5 Punkte): Logging

Bauen Sie in Ihr Programm zwei Logger (`java.util.logger.Logger`) ein. Beide Logger sollen beim **Start des Programms/der GUI über Kommandozeilenparameter** an einer zentralen Stelle enabled bzw. disabled werden können. Bei jeder Textänderung in der GUI sollen die Token (wenn sie einen Match hatten) an den einen Logger ihre unformatierte String-Repräsentation übergeben und an den anderen die HTML-formatierte String-Repräsentation.

Fügen Sie dem ersten Logger einen Handler hinzu, damit eine eigenständig formatierte Ausgabe auf der Konsole möglich wird. Der zweite Logger soll in verschiedene Dateien ausgeben können. Wählen Sie hier unter den im `java.util.logger`-Paket zur Verfügung gestellten Handlern die passenden Klassen aus. Implementieren Sie einen eigenen `PlainTextFormatter` und `HTMLFormatter`, die von jeweils `java.util.logger.Formatter` abgeleitet sind. Sie können sich am `SimpleFormatter` orientieren.

Über den ersten Logger sollen alle Token als unformatierter Text in der Konsole ausgegeben werden, d.h. ohne den normalerweise üblichen Zusatz von Datum, Klasse, Methode und Level je Logmessage – der Code soll auf der Console genauso wie in der GUI aussehen (nur eben unformatiert).

Über den zweiten Logger (bzw. dessen Handler) sollen parallel zwei HTML-Dateien geschrieben werden. In eine Datei sollen alle Token HTML-formatiert ausgegeben werden, in die andere sollen alle Token bis auf die Kommentare HTML-formatiert ausgegeben werden. Auch hier soll wieder nur der (formatierte) Quellcode ausgegeben werden, d.h. die sonst üblichen Logmeldungszusätze wie Datum, Quelle und Level sollen nicht mit erscheinen!

Hinweis: Sie können ausnahmsweise die Loglevel für die Erkennung der Tokenarten durch die Formatter im Logger zweckentfremden. Durch diesen Kunstgriff können die Handler und die Formatter unterscheiden, welche Tokenart gerade geloggt wird und damit das Ausblenden der Kommentare erreichen. Da es in dieser Aufgabe um das Einüben des Umgangs mit dem Logger geht, ist dieser "Missbrauch" ausnahmsweise erlaubt.

5. Aufgabe (5 Punkte): Testsuiten aufbauen

Testen Sie Ihre Implementierung aus dem vorangegangenen Aufgabenteil systematisch. Versuchen Sie, diese Tests bereits in der Entwicklungsphase anzulegen und zu nutzen.

- Als Werkzeug für die Erstellung und Durchführung der Tests nutzen Sie JUnit 4.x.
- Entwickeln Sie zunächst für alle Klassen Modultests. Achten Sie darauf, daß **alle** Methoden getestet werden.
- Entwickeln Sie ausserdem Integrationstests, die das Zusammenwirken mehrerer Klassen testen.
- Entwickeln Sie zusätzlich noch Systemtests, die das Verhalten des gesamten Programms testen.

Achten Sie darauf, daß Sie jeweils sinnvolle Tests entwickeln. Neben einfachen Guttests, d.h. Tests für das normale/erwartete Verhalten, muss es auch Negativtests geben. Hierbei werden unerwartete oder fehlerhafte Eingaben/Parameter übergeben und somit das Verhalten von Methoden/Programmen unter diesen Umständen geprüft. Vergessen Sie nicht, auch die Grenzen von Parameterwertebereichen zu untersuchen! Falls in einer Methode Exceptions auftreten, muss dies auch abgetestet werden.

Fassen Sie verschiedene Tests zu sinnvollen Testsuiten zusammen.

Hinweis: Bei der Abgabe müssen Sie die Wahl Ihrer Testfälle begründen können!

6. Aufgabe (5 Punkte): Versionsverwaltung nutzen

Nutzen Sie für die Arbeiten an diesem Übungsblatt Git für die Versionsverwaltung des Codes.

1. Legen Sie sich ein Repository an.
2. Pflegen Sie Änderungen sinnvoll und regelmässig ein.
   Beachten Sie: Ein Commit sollte nicht zu groß sein und normalerweise jeweils **eine** inhaltliche Änderungseinheit umfassen, d.h. es können durchaus mehrere Dateien an einem Commit beteiligt sein.
   Beachten Sie: Das Projekt muss nach jedem Commit kompilierbar sein.
3. Kommentieren Sie Ihre Commits sinnvoll und korrekt (siehe Vorlesung):
   (a) erste Zeile: Zusammenfassung, max. 40 Zeichen
   (b) zweite Zeile: Leerzeile
   (c) danach dann die Langform des Kommentar

   Beachten Sie: In der Regel reicht die erste Zeile (Zusammenfassung). Bei wichtigeren Änderungen sollte aber dringend die Langform des Kommentars verwendet werden.
   Beachten Sie: Die Kommentare müssen auch für Aussenstehende sinnvoll und verständlich sein.
4. Nutzen Sie fortgeschrittene Methoden wie Tagging und Branching.
5. Tauschen Sie regelmäßig Patches per EMail mit Ihren Gruppenmitgliedern aus und pflegen Sie diese in Ihr Repository ein. Bei Zugriff auf einen gemeinsamen Server oder ein gemeinsames Verzeichnis (geht auch mit USB-Sticks!) können Sie stattdessen auch mit `push/pull` arbeiten.

7. Aufgabe (2 Punkte): Exception Handling

Arbeiten Sie die Kapitel zum Exception-Handling unter Java in der Semesterliteratur durch.

Erklären Sie die Bedeutung und die Funktionsweise des `finally`-Blockes bei der Exception-Behandlung mit `try` und `catch` ...