Javadoc professionell nutzen

Per Knopfdruck dokumentieren

Javadoc erzeugt mehr als nur Klassendokumentation in HTML. Das Framework verschafft auch Zugriff auf Quellcode-Struktur und Kommentare. Der Coffee-Shop zeigt, welche Freeware-Tools diese Möglichkeiten nutzen und wie man selbst Javadoc-Erweiterungen programmiert. Bernhard Bablok

Wiederverwendung von Code steht und fällt mit der Dokumentation. Quellcode zu lesen, um vorhandene Klassen und Methoden zu finden, ist dafür eher ungeeignet. Dokumentation in separaten Dateien pflegen wäre die Lösung, widerspricht aber den Vorlieben vieler Programmierer: Code und Dokumentation würden zwangsläufig voneinander abweichen. Abhilfe schafft Javadoc [1], das dieser Artikel näher beschreibt. Er erklärt die Architektur des Frameworks, hilft eigene Tools zu schreiben und stellt Javadoc-Erweiterungen der Open-Source-Community vor.

Beim Programmieren erklären

Sun übernahm für Java einige Prinzipien des Literate Programming [2], einer Technik, in der Programmieren und Dokumentieren aufs Engste verzahnt sind. Javadoc kennt spezielle Kommentare und Textmarken, auch Tags genannt, aus denen ein Programm die externe Dokumentation erzeugt. Den Erfolg dieses Konzepts kann man daran ablesen, dass es inzwischen ähnliche Tools gibt, zum Beispiel Doxygen für mehrere Programmiersprachen [3].

Natürlich garantiert auch eine einheitliche Quelle für Code und Dokumentation nicht, dass sie einander immer entsprechen, aber zumindest ist die Wahrscheinlichkeit dafür größer. Und dass Methoden in der Dokumentation vergessen werden, ist gänzlich unmöglich. Was am Anfang ein einfacher Generator für HTML-Seiten war, ist inzwischen ein mächtiges Framework, das statische Codeanalyse durchführt und unterschiedliche Metainformationen erzeugt.

Doclets und Taglets

Javadoc muss nicht extra installiert werden, es ist Teil des JDK. Außer dem Programm selbst und der Manpage bringt das Paket aber nichts mit. Die getrennt erhältliche Dokumentation des JDK enthält neben der API-Doku die komplette Tool-Dokumentation. Sie beschreibt ausführlich, wie man Javadoc anwendet und erweitert – ein weiterer Grund, das umfangreiche Paket (entpackt mehr als 160 MByte) herunterzuladen.

Die Architektur von Javadoc erschließt sich am einfachsten an einem Beispiel. Zwei zentrale Begriffe spielen eine Rolle: Doclets und Taglets. Beide sind Klassen mit einem definierten API. Doclets enthalten den Parser sowie den Output-Generator, Taglets sind für die Ausgabe einzelner Tasks verantwortlich.

Javadoc ist ein Binary, das als einfacher Wrapper das Java-Binary aufruft. Die eigentliche Funktionalität des Dateigenerators steckt in der Java-Klasse »Doclet«. Sun liefert ein Default-Doclet mit, das man aber über die Kommandozeilenoption »-doclet« austauschen kann.

In drei Stufen zur Ausgabe

Der Output von Javadoc lässt sich dreistufig ändern. Am einfachsten ist es – bei der Definition eines eigenen Tag –, dessen Formatierung dem Default-Doclet zu überlassen. Wer auf eine eigene Formatierung besonderen Wert legt, benötigt ein Taglet für das eigene Tag. Zu guter Letzt bleibt die Möglichkeit, das Default-Doclet komplett auszutauschen. Das Ausgabeformat ist dann beliebig, es muss also kein HTML-Code mehr erzeugt werden.

Die folgenden Abschnitte zeigen an einem einfachen Beispiel, wie die beschrieben Möglichkeiten zu nutzen sind. Es soll ein Tag »@bb.todo« implementiert werden, mit dem Entwickler noch offene Punkte im Quellcode markieren.

Javadoc soll die mit »@bb.todo« markierten Texte in der normalen Dokumentation markieren und eine Liste aller betreffenden Klassen und Methoden ausgeben. Der Punkt im Namen hat seinen Grund darin, dass »todo« ein so genanntes Future Standard Tag werden soll, möglicherweise also irgendwann vom Standard-Doclet unterstützt wird. Tags mit einem Punkt im Namen gelten immer als nicht standardisiert.

Die Listings setzen Ant ein, um Javadoc aufzurufen. Wer nicht mit Ant arbeitet, muss die Beispiele in die Kommandozeilenoptionen von Javadoc übersetzen, aber das sollte kein Problem sein.

Fremde Tags verändern

Listing 1 zeigt einen kurzen Ausschnitt aus der Quelldatei mit dem »@bb.todo«-Tag. Das zugehörige Ant-Buildfile hat ein API-Target mit einer Javadoc-Task (Listing 2, Zeilen 273 bis 287). Das geschachtelte »tag«-Element in Zeile 286 wird für Javadoc zur Kommandozeilenoption »-tag«. Die Ausgabe zu den Methoden aus Listing 1 zeigt Abbildung 1. Die »description« erscheint als fetter Text auf derselben Ebene wie die Standard-Tags »@return« oder »@param«.

Um die Todo-Kommentare mit Sicherheit nicht zu übersehen, sollen die entsprechenden Zeilen im nächsten Schritt hervorgehoben werden. Das »To Do:« erscheint dann in roter Schrift, der Text ist rot hinterlegt (siehe Abbildung 2). Listing 3 zeigt das zugehörige Taglet. Es basiert auf einem Beispiel von Sun, das Teil der erwähnten Tool-Dokumentation ist. Die Zeilen 76 bis 83 definieren Konstanten, insbesondere den HTML-Text für die Formatierung.

Dann folgt eine Reihe von Methoden (»getName()«, »inField()«, »inConstructor()«), die durch das Taglet-Interface vorgeben sind. Analog zu den abgedruckten »inField()«- und »inConstructor()«-Methoden gibt es weitere »in*XXX*()«-Methoden (zum Beispiel »inMethod()«), die festlegen, in welchen Bereichen das Tag erscheinen darf. Interessanter ist »isInlineTag()« ab Zeile 188, die hier »false« zurückliefert: Das Tag darf nicht innerhalb eines Kommentartextes erscheinen. Die Methode »register()« ab Zeile 199 registriert das Taglet beim Standard-Doclet.

Am Ende geben die »toString()«-Methoden ab Zeilen 216 die formatierten Tags

Listing 1: Ausschnitt aus »GuiSearcher.java«

```
305   /**
306      Process the exit-command.
307
308      @bb.todo Save window-location and other
    settings in preferences.
309   */
310   ...
317   /**
318      Process the openIndex-command.
319
320      @bb.todo  Implement method!
321   */
```

Listing 2: Eigenes Tag – einfache Formatierung

```
267 <!-- ==== API Target ================== -->
268
269   <target name="api"
270    description="Create Javadoc API
    documentation">
271
272     <mkdir          dir="${api.home}"/>
273     <javadoc sourcepath="${src.home}"
274              classpathref="compile.classpath"
275                   destdir="${api.home}"
276
    overview="${doc.home}/overview.html"
277                   access="private"
278                   version="true"
279                   author="true"
280              windowtitle="${app.windowtitle}"
281                 doctitle="${app.doctitle}"
282                   header="${app.header}"
283                   footer="${app.footer}"
284                   bottom="${app.bottom}"
285             packagenames="*">
286       <tag name="bb.todo" description="To do:"
    />
287     </javadoc>
288   </target>
289 </project>
```

Listing 3: »BBToDoTaglet.java«

```
060 package de.bablokb.tools;
061
062 import com.sun.tools.doclets.Taglet;
063 import com.sun.javadoc.*;
064 import java.util.Map;
065
074 public class BBToDoTaglet implements Taglet {
075
076   private static final String NAME =
    "bb.todo";
077:   private static final String HEADER =
     "To Do:";
078
079   private static final String HTML_START =
080     "<dt><strong><font color=\"red\">" + HEADER
    + "</font></strong></dt>"
081     + "<dd><table cellpadding=2
    cellspacing=0><tr><td bgcolor=\"red\">";
082
083   private static final String HTML_END =
    "</td></tr></table></dd>\n";
084
091   public String getName() {
092     return NAME;
093   }
094
105   public boolean inField() {
106     return true;
107   }
108
119   public boolean inConstructor() {
120     return true;
121   }
122
188   public boolean isInlineTag() {
189     return false;
190   }
191
199   public static void register(Map tagletMap) {
200     BBToDoTaglet tag = new BBToDoTaglet();
201     Taglet t = (Taglet)
        tagletMap.get(tag.getName());
202     if (t != null) {
203       tagletMap.remove(tag.getName());
204     }
205     tagletMap.put(tag.getName(), tag);
206   }
207
216   public String toString(Tag tag) {
217     return HTML_START + tag.text() + HTML_END;
218   }
219
228   public String toString(Tag[] tags) {
229     if (tags.length == 0) {
230       return null;
231     }
232     StringBuffer result = new
        StringBuffer(HTML_START);
233     for (int i = 0; i < tags.length; i++) {
234       if (i > 0) {
235         result.append(", ");
236       }
237       result.append(tags[i].text());
238     }
239     result.append(HTML_END);
240     return result.toString();
241   }
242 }
243
```

aus. Der Aufruf für dieses Taglet unterscheidet sich nur minimal vom ersten Fall, wie Listing 4 zeigt. Zusätzlich zu dem schon vorhandenen »tag«-Tag sind in »build.xml« die Taglet-Klasse sowie deren »CLASSPATH« aufzunehmen. Das erledigt das »taglet«-Tag in den Zeilen 314 bis 316. Das Beispiel zeigt, wie sich das Taglet innerhalb kürzester Zeit an eigene Tags und Formatierungswünsche anpassen lässt. Man bleibt dabei aber auf die normale HTML-Dokumentation als Ausgabeformat beschränkt. Weitergehende Änderungen erfordern selbst geschriebene Doclets.

Doclets selbst gemacht

Das Beispiel-Doclet soll wie oben beschrieben eine Liste aller Klassen und Methoden ausgeben, die mit dem »bb.todo«-Tag markiert sind. Dazu dient ein Java-Programm, das über die Methoden aller Klassen iteriert. Da das Doclet-API alle nötigen Hilfsmittel bereithält, ist das in wenigen Zeilen geschehen.

Das Doclet-API schreibt als Einstiegspunkt die Methode »start(RootDoc)« vor (Listing 5, ab Zeile 55). Das »RootDoc«-Objekt ist die Wurzel eines Doc-Baumes, jeder Knoten enthält die spezifische Information zu einer Klasse, Methode und so weiter. Die Implementation wird so fast trivial: Sie hangelt sich über alle Kindknoten durch den Baum und fragt auf jeder Ebene ab, ob das »bb.todo«-Tag vorhanden ist.

Die »dumpTag()«-Methode des Doclet (Listing 5, Zeilen 109 bis 123) kann beliebig überschrieben werden. Wer seine Todos etwa in einer Datenbank pflegen will, könnte hier SQL-Statements ausgeben oder gleich über JDBC in eine Datenbank schreiben.

Das Doclet ist über Kommandozeilenparameter parametrisierbar (Tag-Name und Ausgabedatei), wie der Beispielaufruf in Listing 6, Zeile 336, zeigt. Die Verarbeitung der Parameter ist hier aus Platzgründen nicht abgedruckt, alle Listings stehen aber auf dem Server zum Download bereit [4].

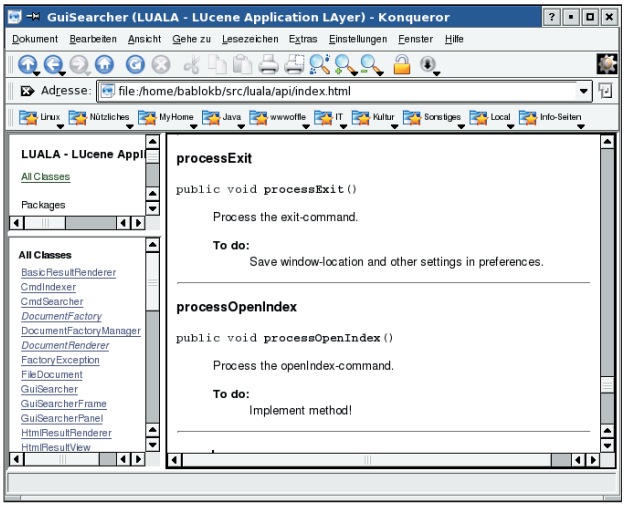

Abbildung 1: Das Zusatz-Tag im normalen Javadoc-Outfit ohne explizite Formatierung. »To Do:« und Text stehen unter der Methodenbeschreibung.

Einfach, aber mächtig

Javadoc erlaubt durch den einfachen Zugriff auf den Dokumentenbaum viele Anwendungen zur Qualitätssicherung. So lässt sich mit eigenen Doclets etwa überprüfen, ob Kommentare vorhanden sind oder ob die Namenskonventionen für Methoden, Parameter und Felder eingehalten wurden.

Im Internetzeitalter gilt aber das Prinzip „Downloaden statt selber machen“ und tatsächlich gibt es schon eine Fülle von fertigen Doclets für verschiedene Zwecke. Der beste Ausgangspunkt für die eigene Suche ist die Doclet-Homepage [5], die viele Links zu fertigen Doclets und weiterführenden Texten bietet. Die folgenden Abschnitte präsentieren eine kleine Auswahl.

Das »DocCheck«-Doclet von Sun überprüft, ob die Javadoc-Kommentare vollständig sind. Das ist zum Beispiel bei extern vergebenen Programmieraufträgen nützlich, um zu überprüfen, ob diese Anforderung erfüllt ist. Aber es bewährt sich auch im Entwicklungsprozess, da

Listing 4: »build.xml« für Taglets

```
295   <target name="api" depends="compile-tools"
296    description="Create Javadoc API documentation">
297    ...
313       <tag name="bb.todo" description="To do:"/>
314       <taglet name="de.bablokb.tools.BBToDoTaglet">
315         <path refid="classpath.tools"/>
316       </taglet>
317     </javadoc>
318   </target>
```

Listing 5: »BBToDoDoclet.java«

```
039 public class BBToDoDoclet {
040
041   private static String iTagName =
          "@bb.todo";
042   private static String iOutFile = null;
043   private static PrintWriter iWriter = null;
044 ...
055   public static boolean start(RootDoc root){
056     parseOptions(root.options());
057     ClassDoc[] classes = root.classes();
058     for (int i=0; i < classes.length; i++) {
059       dumpTag(classes[i].tags(iTagName),
              classes[i].qualifiedName());
060       parseClass(classes[i]);
061     }
062     return true;
063   }
064 ...
074   private static void parseClass(ClassDoc
          clazz) {
075     // inner classes
082     // fields
088     // constructors
094     // methods
098     }
099   }
100 ...
109   public static void dumpTag(Tag[] tags,
          String qualifiedName) {
110     try {
111       if (iWriter == null) {
112         if (iOutFile == null)
113           iWriter = new
                PrintWriter(System.out,true);
114         else
              // append, flush
115           iWriter = new PrintWriter(new
              FileWriter(iOutFile,true),  true);
116       }
117     } catch (IOException e) {
118       throw new Error(e);
119     }
120     for (int i=0; i < tags.length; i++) {
121       iWriter.println(qualifiedName + ": " +
            tags[i].text());
122     }
123   }
157 }
```

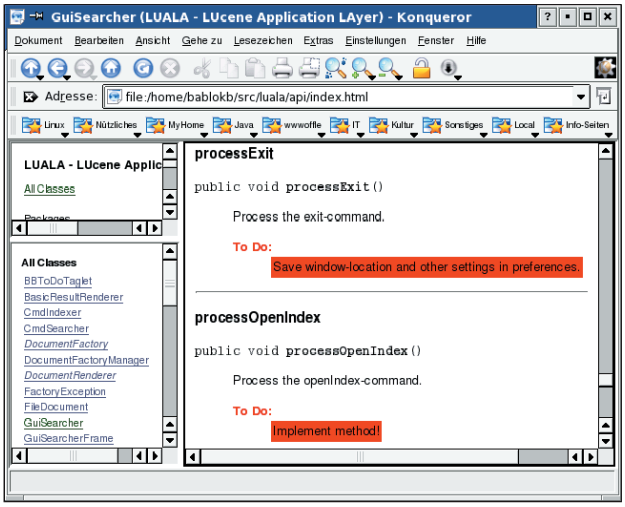

Abbildung 2: Das Zusatz-Tag mit einem Taglet formatiert: »To Do:« erscheint in roter Schrift, der beschreibende Text ist mit rotem Hintergrund unterlegt.

der Programmierer den generierten Output (Vorschläge für fehlende Kommentare) per Cut&Paste in den eigenen Code übernehmen kann.

Am Ende des Entwicklungszyklus eines Produkts hilft das JDiff-Doclet [6]. Es generiert Reports über Änderungen am API. Die JDiff-Homepage zeigt als Beispiel die Unterschiede zwischen verschiedenen Versionen des Java-SDK.

Alternativen zu HTML

Das Standard-Doclet gibt HTML aus. Auf Unix-Systemen gibt es jedoch mit dem Texinfo-System ein altes, aber durchaus lebendiges Hypertextformat. Wer seine API-Dokumentation in den existierenden Info-Baum einbinden will, sollte mit dem Texi-Doclet arbeiten [7].

Hypertextsysteme sind während der Programmierarbeit nützlich, aber wer sich einen Überblick über ein API verschaffen möchte, nutzt lieber gedruckte Informationen. Das PDF-Doclet von [8] erzeugt eine Druckversion der API-Dokumentation. Das Projekt ist auch ein Beweis für die Leistungsfähigkeit der Open-Source-Szene, denn das PDF-Doclet ist der Sun-eigenen Alternative (PDF über MIF-Dateien, Framemaker und Distiller) deutlich überlegen.

Alle bisher vorgestellten Doclets gehen den Weg vom Programmcode zur Dokumentation. XDoclet [9] arbeitet zum Teil andersrum: Es erzeugt Java-Code aus dem Javadoc-Kommentar. Auslöser für die XDoclet-Entwicklung war der Wildwuchs von Klassen und Interfaces in dem Bereich der EJB (Enterprise Java Beans). Neben der eigentlichen Implementationsklasse muss der Entwickler zusätzlich auch noch die Home- und Remote-Interfaces einrichten. Das ist aber nicht besonders aufregend, da diese Interfaces im Wesentlichen nur triviale Varianten der Methodensignaturen sind. Hinzu kommt die Beschreibung der Bean-Klassen in den Deployment-Deskriptoren.

Hilfe bei EJBs

Die reine Lehre (Program against interfaces, not classes – nach Schnittstellen, nicht Klassen programmieren) empfiehlt auch bei normalen Java-Klassen, erst ein Interface zu definieren und dann entsprechende Klassen zu implementieren. Oft spart man sich allerdings die Mühe, da der Gewinn in vielen Bereichen marginal ist. Bei EJBs kommt man aber nicht darum herum, weshalb es nahe liegt, den umgekehrten Weg zu gehen und die Interfaces aus der Klassendefinition zu generieren.

Genau das macht XDoclet. Es wertet spezielle Tags (zum Beispiel »@ejb.bean« oder »ejb.interface-method«) aus und generiert daraus die passenden Dateien. Über Templates lässt sich dieser Prozess fein steuern, was insbesondere hinsichtlich der Unterschiede der J2EE-Implementationen auch nötig ist. XDoclet unterstützt unter anderem JBoss, Weblogic und Websphere. Natürlich liefert XDoclet auch Tasks für Ant mit, damit lässt sich der Entwicklungszyklus noch stärker automatisieren.

finally{}

Diese Beispiele sollen genügen, obwohl es noch viele interessante Doclets vorzustellen gäbe. Gerade für das im Beispiel behandelte Todo-Doclet gibt es im Internet ausgefeilte Alternativen.

Mit Javadoc hat Sun ein leicht erweiterbares Tool für die Java-Entwicklung geschaffen. Ein Teil der Ideen (besonders der Ansatz von XDoclet) ist in die nächste Version 1.5 des SDK eingeflossen: Mit so genannten Annotations lassen sich beliebige Informationen im Quelltext hinterlegen. Diese und andere neue Features von Java 1.5 sind Thema im nächsten Coffee-Shop. *(ofr)* ■

Listing 6: »build.xml« für Doclets

```
<target name="dump-todos" depends="compile-tools"
  description="Dump documented todo-items">

  <javadoc sourcepath="${src.home}"
           classpathref="classpath.api"
                access="private"
           packagenames="*">
      <doclet name="de.bablokb.tools.BBToDoDoclet">
        <path refid="classpath.tools"/>
        <param name="-o" value="TODO"/>
      </doclet>
    </javadoc>
  </target>
</project>
```

Infos

[1] Javadoc-Homepage: [http://java.sun.com/j2se/javadoc/]
[2] Literate Programming: [http://www.literateprogramming.com/]
[3] Doxygen: [http://www.stack.nl/~dimitri/doxygen/]
[4] Listings zu diesem Coffee-Shop: [http://www.linux-magazin.de/Service/Listings/2004/09/Coffeeshop]
[5] Doclet-Homepage: [http://doclet.com]
[6] JDiff-Doclet: [http://www.jdiff.org]
[7] Texi-Doclet: [http://texidoclet.sf.net]
[8] PDF-Doclet: [http://pdfdoclet.sf.net]
[9] XDoclet: [http://xdoclet.sourceforge.net/]

Der Autor

Bernhard Bablok arbeitet bei der AGIS mbH als Anwendungsentwickler. Wenn er nicht Musik hört, mit dem Radl oder zu Fuß unterwegs ist, beschäftigt er sich mit Themen rund um die Objektorientierung. Er ist unter [coffee-shop@bablokb.de] zu erreichen.