Softwareprojekt 2014

Gruppe 6

Kognitive Suche

Technologierecherche

Coding Guidelines

Vadzim Naumchyk

Code Guidelines und Standards zur Erstellung konsistenter Quellcodes.

Inhalt

1. Konsistenter Code durch Gestalten

1.1. Einleitung

In unserem Projekt orientieren wir uns auf Programmieren mit Java in der Umgebung Eclipse. Zum Glück gehört Java nicht zu den so genannten Write-Only Languages, die schwer zu lesen sind (siehe Bild 1)

Java-Motto: *Write once – run everywhere!*

Perl-Anspielung: *Write once – never understand again!*

```
@P=split//,".URRUU\c8R";@d=split//,"\nrekcah xinU / lreP rehtona tsuJ";sub p{
@p{"r$p","u$p"}=(P,P);pipe"r$p","u$p";++$p;($q*=2)+=$f=!fork;map{$P=$P[$f^ord
($p{$_})&6];$p{$_}=/ ^$P/ix?$P:close$_}keys%p}p;p;p;p;p;map{$p{$_}=~/^[P.]/&&
close$_}%p;wait until$?;map{/^r/&&<$_>}%p;$_=$d[$q];sleep rand(2)if/\S/;print
```

Abbildung 1. verschleierter Quellcode in Perl

Ein Beispiel von Mark Jason Dominus, das 2000 beim 5. *Annual Obfuscated Perl Contest* den zweiten Preis gewann (dieses Programm gibt den Text „Just another Perl / Unix hacker“ aus – funktioniert nicht unter Windows).

Das ist natürlich ein übertriebener Vergleich. Aber wenn ein Team von Entwicklern eine Software mit Java programmiert und dabei keine einheitlichen Regeln zum Gestalten und Strukturieren des Quellcodes hat, dann ist das Lesen des Kodes fast so schwer.

Programmiersprachen, genauso wie die natürlichen Sprachen, sind besser zu verstehen, wenn es versucht wird, Mehrdeutigkeit und Unklarheiten durch kollektive Verabredungen (Konventionen) zu vermeiden.

Bei Programmiersprachen gibt es 3 Wege, wie solche Konventionen entstehen:

1. eine verbreitete schriftliche Arbeit zu einer Sprache. (z.B. K&R-Stil für die Sprache C)

2. eine verbreitete Bibliothek oder API. (z.B. Ungarische Notation verbreitete sich dank MS-DOS und Windows API. Oder die Mehrheit von Delphi-Standards nutzt die Kodierungsstil der Bibliothek VCL)

3. Standard vom Entwickler der Sprache (z.B. Microsoft-Standard für C# oder Sun-Standard für Java)

Ein gut geschriebener Code ist

- gut lesbar
- übersichtlich
- verständlich
- erweiterbar
- nicht redundant
- testbar
- wartbar
- foobar

Wenn es nicht der Fall ist, dann ist *Code Refactoring* (Umgestalten von Code) notwendig. Refactoring ist für die *Agile Softwareentwicklung* von zentraler Bedeutung.

Ist ein Code nicht gut, sondern schlecht geschrieben und wurde Refactoring nicht rechtzeitig durchgeführt, so entsteht die so genannte *Technische Schuld* (Technical Debt, Code Debt) der Software. Dabei wird der Aufwand gemeint, der für Verbesserung des Codes notwendig ist. Dieser Aufwand wird größer beim Verschieben wie die Verzugszinsen im Fall mit einer finanziellen Schuld. (TODO-, FIXME-, XXX-Kommentare)

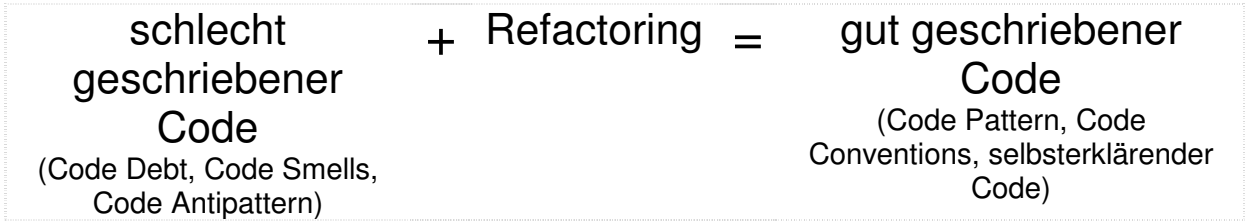

Abbildung 2. Korrigieren vom schlechten Code durch Refactoring.

	Code Debt	**Antipattern**	**Code Smells**
Unterschied	ist eine Entscheidung, wie groß die Schuld sein darf.	mangelnde Erfahrung, fehlende Qualifikation	
Ursachen	fachlicher Druck ungenügende qualitätssichernde Prozesse ungenügendes technisches Wissen ungenügende Kommunikation parallele Entwicklung hintangestelltes Refactoring		
Auswirkung	Wartbarkeit, Erweiterbarkeit		
Beispiele	• Hintanstellen technischer und fachlicher Softwaredokumentation • Fehlende technische Infrastruktur	Programmierungs-Anti-Pattern:	Code-Duplizierung Lange Methode

	wie Versionsverwaltung, Datensicherung, Build-Tools, Kontinuierliche Integration • Hintanstellen, Verzicht oder ungenügende Umsetzung automatisierter Modultests und Regressionstests • Fehlende Coding Standards und Code Ownership • Missachtung von TODO oder FIXME oder XXX Hinweisen im Code • Missachtung von Codewiederholungen und anderen Code Smells • Verwendung von Programmierungs-Anti-Pattern • Missachtung von Compilerwarnings und Ergebnissen statischer Code-Analyse • Hintanstellen der Korrektur von zu großem oder zu komplexen Code und Design • Fehlerhafte Definition oder Umsetzung der Architektur durch enge Kopplung, zyklische Abhängigkeiten der Komponenten oder das Fehlen geeigneter Schnittstellen und Fassaden	Doppelt überprüfte Sperrung Zwiebel Copy and Paste Lavafluss Magische Werte Reservierte Wörter Unbeabsichtigte Komplexität Architektur- bzw. Entwurfs-Anti-Pattern: Big Ball of Mud Gasfabrik Gottobjekt Innere-Plattform-Effekt Spaghetticode Sumo-Hochzeit (auch Code Smells:) Nichtssagender Name Redundanter Code Indecent Exposure Contrived Complexity Zu lange Namen Zu kurze Namen Über-Callback Komplexe Verzweigungen Tiefe Verschachtelungen	Große Klasse Lange Parameterliste Divergierende Änderungen Schrotkugeln Chirurgie Neid Datenklumpen Neigung zu elementaren Typen Case-Anweisungen in OOCode Parallele Vererbungshierarchien Faule Klasse Spekulative Allgemeinheit Temporäre Felder Nachrichtenketten Middle Man Unangebrachte Intimität Alternative Klassen mit verschiedenen Schnittstellen Inkomplette Bibliotheksklasse Datenklasse Ausgeschlagenes Erbe Kommentare

Abbildung 3. Code Debt, Code Smells, Antipattern

Strenges Anhalten an Pattern ist ein Antipattern!

1.2. Namenskonvention

Namen von Klassen, Methoden, Feldern/Variablen, Typen

- eindeutig
- nicht zu lang
- nicht zu kurz

Camel_SNAKE-kebab

Camel Case	Snake Case	Kebab Case
camelCase / lowerCamelCase	TRAIN_CASE	kebab-case
PascalCase / UpperCamelCase	spinal_case	

Abbildung 4. Naming-Cases

Hungarian Notation

Namenskonvention bei Bezeichnung von Variablen, Konstanten, Funktionen, Methoden usw.

Prinzip besteht darin, dass ein Name folgende Struktur hat:

$\boxed{< Präfix >< Datentyp >< Bezeichner >}$ als lowerCamelCase, oder

$\boxed{< Präfix > \_ < Datentyp >< Bezeichner >}$ als Kombination von Snake- und Camel-Case.

Ziel der ungarischen Notation besteht darin, durch den Bezeichner auf den Typ und die Verwendung einer Variablen zu verweisen.

Bei Präfixen und Datentypen handelt es um einen oder zwei Buchstaben.

Ein Beispiel:

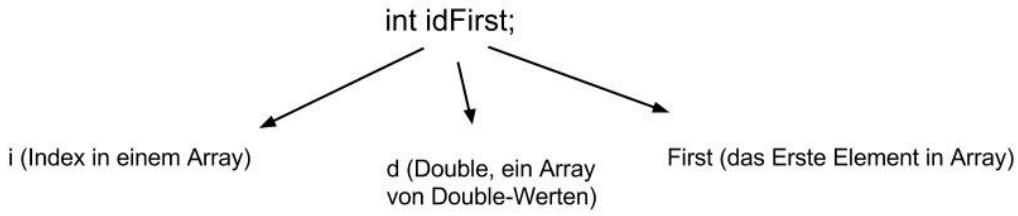

Abbildung 5. Ungarische Notation in Variablenbezeichnung.

Mehr zur ungarischen Notation:

http://msdn.microsoft.com/en-us/library/aa260976(v=vs.60).aspx

https://de.wikipedia.org/wiki/Ungarische_Notation

1.3. Formatierung

Benutzen / Nicht-Benutzen von Klammern (z.B. wenn if-Block nur eine Anweisung besitzt)

Einrückungsstil – meist umstrittene Quellcodeformatierung

Das Klammersetzen nach dem Muster aus der C-Bibel wird auch Egyptian Brackets genannt, weil sie an Hände-Haltung eines Ägypters erinnern.

Abbildung 6. Egyptian Brackets - eine Slang-Bezeichnung für K&R-Stil

Braces-Stil	Beispiel

K&R / Egyptian Brackets / 1TBS	`if (a == b) {`
	`    printf("hello");`
	`}`

Allman-Stil / Stroustrup	`int f(int x, int y, int z)`
	`{`
	`    if (x < foo(y, z))`
	`    {`
	`        qux = bar[4] + 5;`
	`    }`
	`    else`
	`    {`
	`        while (z)`
	`        {`
	`            qux += foo(z, z);`
	`            z--;`
	`        }`
	`        return ++x + bar();`
	`    }`
	`}`

Horstmann-Stil	`int f(int x, int y, int z)`
	`{   if (x < foo(y, z))`
	`    {   qux = bar[4] + 5;`
	`    }`
	`    else`
	`    {   while (z)`

	`{ qux += foo(z, z); z--; } return ++x + bar(); } }`
Whitesmith-Stil	`int f(int x, int y, int z) { if (x < foo(y, z)) { qux = bar[4] + 5; } else { while (z) { qux += foo(z, z); z--; } return ++x + bar(); } }`

Abbildung 7. Vier verbreitetste Stile für Klammern-Setzung

Line-Wrapping (wenn eine lange Zeile „gebrochen“ werden soll)

Whitespace (vertical, horizontal)

Comments

Beautyfier / Quellcodeformatierer

Prettyprint / Quelltextformatierung

1.4. Strukturierung und Logik

Anti-Pattern / Pitfalls / „Antimuster“

- Big Ball Of Mud (große Matschkugel)
- God Object / Gottobjekt
- Software rot / Code rot / dt. Softwareerosion / Softwarezerfall

Spaghetti Code

noch ein paar …

Magic Number

```
drawSprite(53, 320, 240);
```

Abbildung 8. Magische Zahlen im Code

Für denjenigen, der den Code nicht geschrieben hat, ist es schwierig zu verstehen, was diese Zahlen bedeuten.

```
final int SCR_WIDTH = 640;
final int SCR_HEIGHT = 480;
final int SCR_X_CENTER = SCR_WIDTH/2;
final int SCR_Y_CENTER = SCR_HEIGHT/2;
final int SPRITE_CROSSHAIR = 53;
drawSprite(SPRITE_CROSSHAIR, SCR_X_CENTER, SCR_Y_CENTER);
```

Abbildung 9. Magische Zahlen durch Konstanten ersetzt

Jetzt wird es klar, dass es um die Mitte eines Bildschirms geht.

1.5. Google Java Style

Java Code Standards:

Sun Standards

http://www.oracle.com/technetwork/java/codeconvtoc-136057.html

Abbildung 10. Sun-Conventions für Java Code werden nicht gewartet

Google Java Style

https://google-styleguide.googlecode.com/svn/trunk/javaguide.html

Abbildung 11. Google-Java-Konvention

Das Projekt ‚Style Guideline' bietet Code-Standards für Open-Source-Projekte aus dem Hause Google an. Die Standards können auch für andere Projekte genommen werden. Neben Google-Java-Style, sind folgende Guidelines zu finden:

- C++ Style Guide
- Objective-C Style Guide
- Python Style Guide
- Shell Style Guide
- HTML/CSS Style Guide
- JavaScript Style Guide
- AngularJS Style Guide
- Common Lisp Style Guide
- Vimsrcript Style Guide
- XML Document Format Style Guide

Letzte Änderung in Java Style wurde am 21. März 2014 vorgenommen.

Google-Java-Style Guide enthält folgende Kapitel:

- Source file basics
- Source file structure
- Formatting

- Naming
- Programming Practices
- Javadoc

Formatierungskonventionen sind in einer XML-Datei zusammengefasst. Hier die HTML-Version vom File:

https://code.google.com/p/google-styleguide/source/browse/trunk/eclipse-java-google-style.xml

Das XML-File wurde am 30. August 2011 angelegt und am 3. Februar zum letzten Mal geändert.

Benutzen in Eclipse

Der in Eclipse geschriebene Code lässt sich auch nach Google-Java-Standard formatieren. Davor soll man aber die Google-Formatierung in die Liste möglicher Standards in Eclipse hinzufügen. Dazu braucht man ein XML-File mit Beschreibung der angenommenen Konventionen. Dieses wird vom unten angeführten erwähnten Link (rohes File) durch Rechtsklick im Browser heruntergeladen.

Rohes File:

https://google-styleguide.googlecode.com/svn/trunk/eclipse-java-google-style.xml

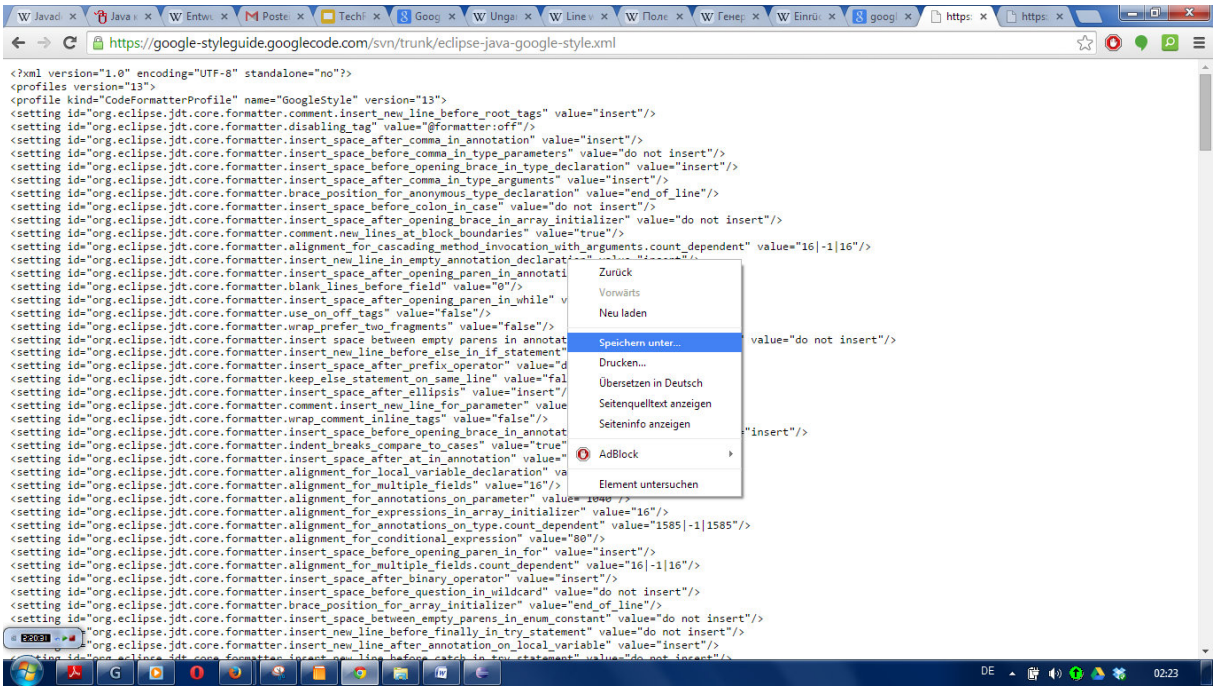

Abbildung 12. Herunterladen der XML-Datei mit Google-Formatierung

Durch Anklicken von ‚Window – Preferences – Java – Code Style – Formatter' gelangt man zum Menü, wo eigene Formating-Standards erstellt oder importiert werden können. Man drückt die Schaltfläche ‚Import' und wählt die heruntergeladene Datei aus. Alle Aktionen mit OK bestätigen.

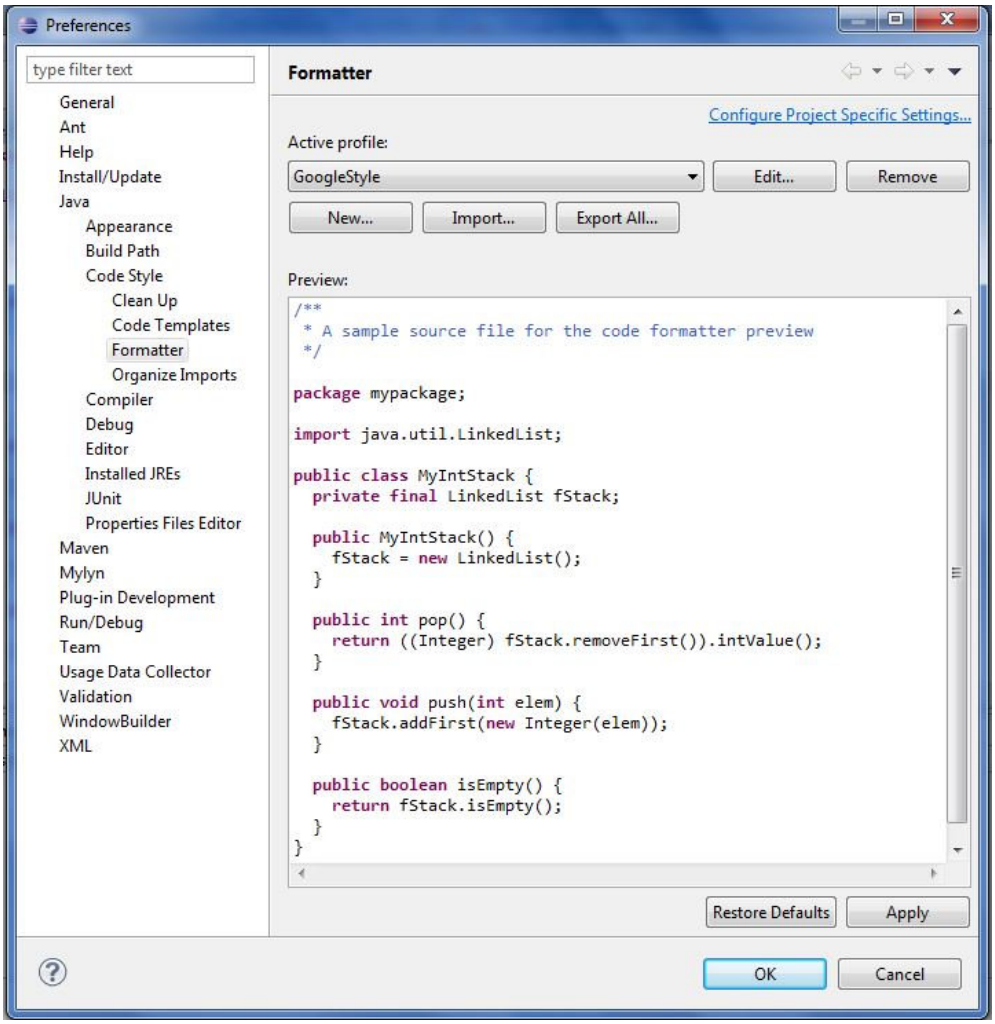

Abbildung 13. Hinzufügen von GoogleStyle in Eclipse

2. Konsistenter Code durch Kommentieren

2.1. Einleitung

Dokumentwerkzeuge helfen, Dokumentation über eine Software für mitarbeitende Entwickler oder für Endnutzer zu erstellen. Dabei werden Module des Programms analysiert und auf besondere Weise gestaltete Kommentare des Quellcodes genutzt. Die Dokumentation wird dann in einem festgelegten Standardformat ausgegeben (HTML, PDF, RTF).

Für verschiedene Sprachen gibt es auch eigene Dokumentationswerkzeuge. Hier ein paar Beispiele:

- Doxygen für C++, C, Java, PHP, C#
- Javadoc für Java

- Epydoc und Sphinx für Python
- JSDoc für JavaScript
- PhpDocumentor und PHPDoc für PHP

Weiter geht es um das Werkzeug Javadoc.

2.2. Javadoc

Javadoc wurde von Sun Microsystems entwickelt und ist in der Mehrheit der Entwicklungsumgebungen enthalten.

Die dokumentierenden Kommentare werden für Dokumentieren von Klassen, Konstruktoren, Methoden und Datentypen eingesetzt.

Der Kommentar soll vor dem zu dokumentierenden Element stehen.

Ein Doc-Kommentar soll wie folgt gestaltet werden:

```
/**
* Dieser Kommentar wird als Doc Comment interpretiert.
*/
```

Abbildung 14. Javadoc-gerechter Kommentar

Javadoc-Tags

@author name @version version @since jdk-version @see reference
@serial @serialField @param name description @return description
@exception classname description @throws classname description
@deprecated description {@inheritDoc} {@link *reference*}
{@linkPlain*reference*}{@value} {@docRoot} {@code} {@literal}

Die Rolle von Doc-Kommentaren bei Packeten spielen die html-Dateien, die im Workspace von entsprechenden Packages abgelegt werden. Der Text im <body> wird von Javadoc in die Dokumentation integriert.

How to write Doc Comments for the Javadoc Tool

http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

2.3. Beispiele aus Eclipse

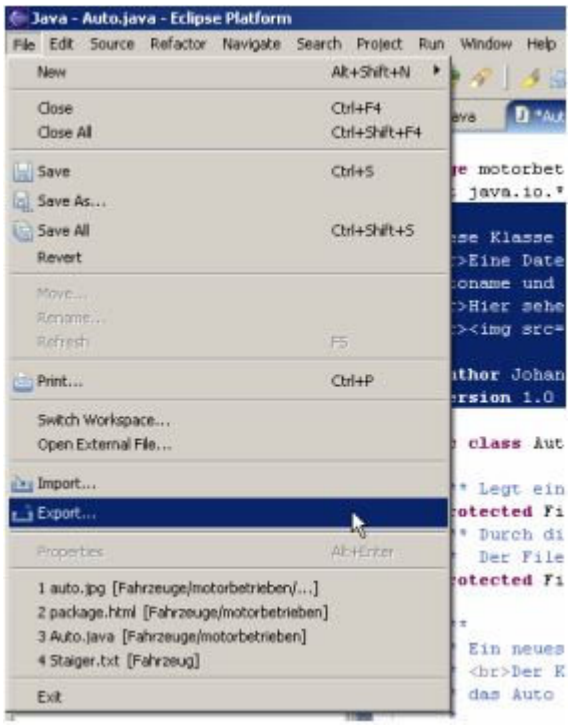

Abbildung 15. Dokumenation erstellen

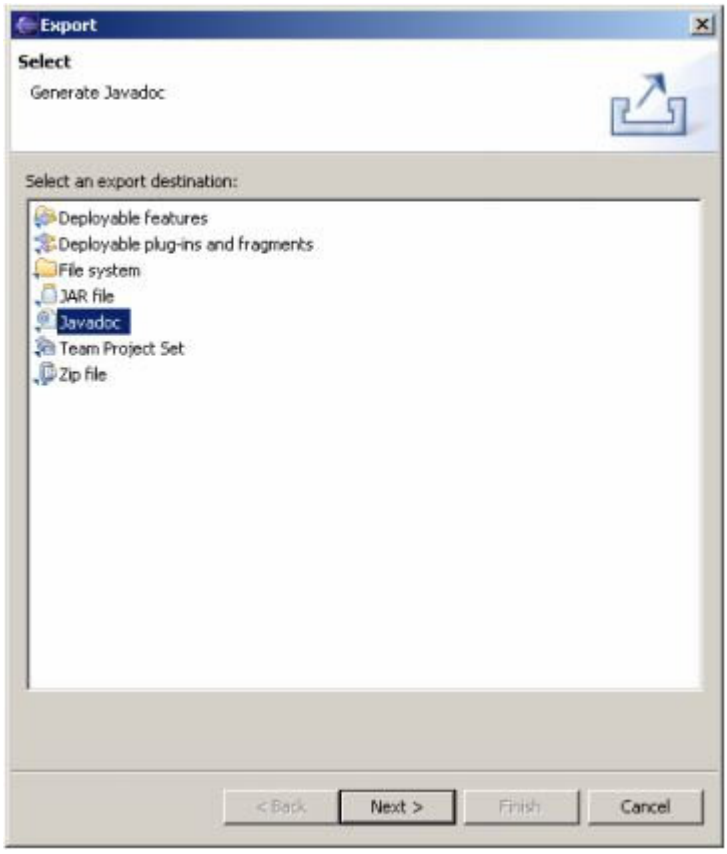

Abbildung 16

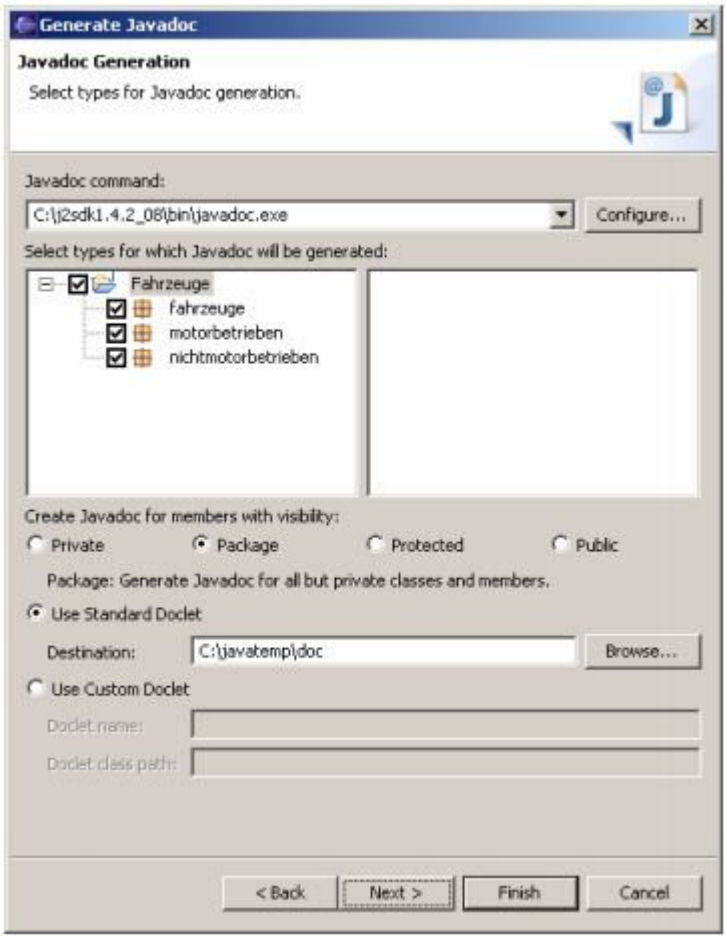

Abbildung 17.

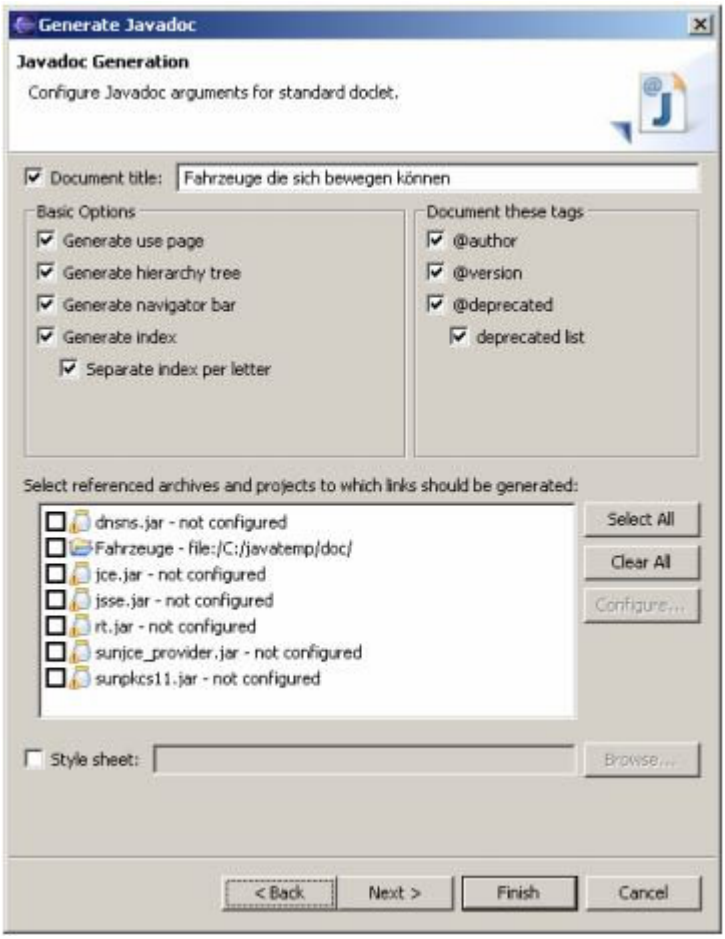

Abbildung 18.

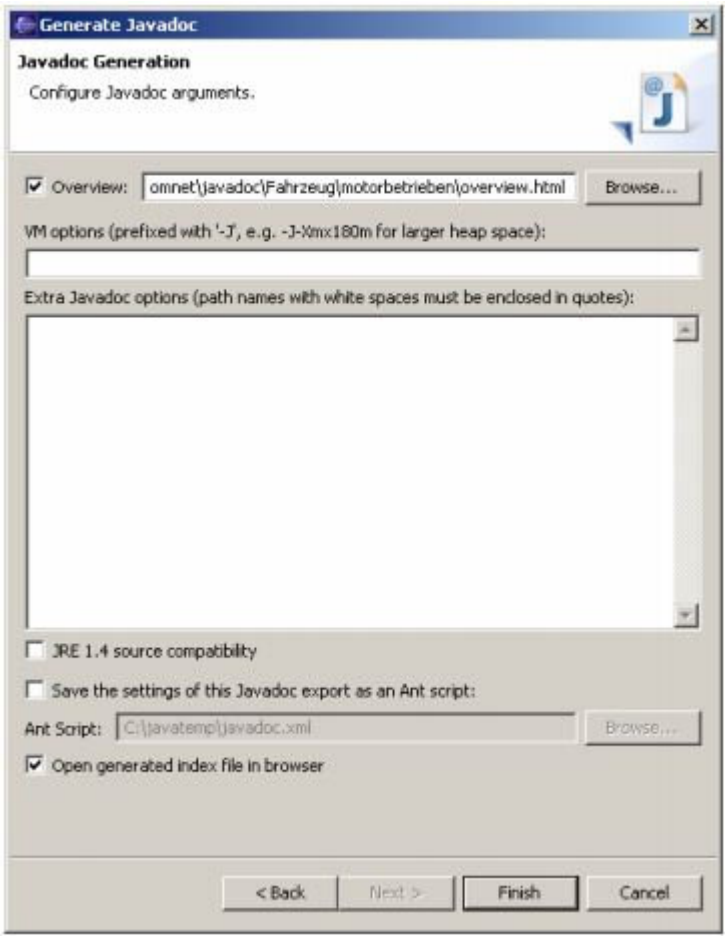

Abbildung 19.

Abbildung 20. Dokumentation in Browser-Ansicht

Field Summary

protected java.io.File	**AutoDatei** Legt eine neue Datei an.
protected java.io.FileWriter	**AutoDateiSchreiber** Durch diesen Filewriter kann in eine Datei geschrieben werden.

Constructor Summary

Auto(java.lang.String Name, int AnzahlGaenge)
Ein neues Auto wird angelegt.

Method Summary

boolean	**sichereAuto**(java.lang.String Autoname, int Gaenge) Sichert Auto in eine Datei.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Abbildung 21.