Institut für Softwaretechnologie
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3266

Integrierte Dokumentation für Software-Module

Michael Kircher

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Jochen Ludewig
Betreuer:	Dipl.-Inf. Ivan Bogicevic
begonnen am:	01. November 2011
beendet am:	30. April 2012
CR-Klassifikation:	D.2.2, D.2.7

Integrierte Dokumentation für Software-Module

Zusammenfassung

Ein umfangreiches und komplexes Software-System muss in mehrere Module unterteilt werden, damit dieses verständlich bleibt. Dabei hilft eine Moduldokumentation, welche die Details eines einzelnen Moduls beschreibt.

Existierende Werkzeuge zur integrierten Software-Dokumentation unterstützen die Dokumentation auf Modulebene nur unzureichend. So wird oftmals keine Integration in gängige Entwicklungsumgebungen geboten oder die Dokumentation eines Moduls wird zusammen mit der Dokumentation von Klassen und Methoden vermischt.

Ziel dieser Diplomarbeit war es daher *J-PaD*, ein Werkzeug zur integrierten Dokumentation von Software-Modulen, zu entwickeln. Es erlaubt eine Dokumentation von Java-Paketen über eine flexibel anpassbare, grafische Oberfläche. J-PaD wurde als Plugin für die Entwicklungsumgebung Eclipse konzipiert, wodurch Entwickler dieses Werkzeug in einer gewohnten Umgebung verwenden können. Die Dokumentation eines Moduls wird dabei in einer separaten Datei gespeichert, welche sich jedoch in direkter Nähe zum Code befindet.

Integrated Documentation for Software Modules

Abstract

A huge and complex software system has to be divided into several modules to maintain comprehensibility. Therefore a module documentation is necessary which describes the details of a single module.

Existing tools for integrated software documentation don't support the documentation on a module level very well. Many of them don't offer an integration into common development environments. Also the documentation on a module level is often mixed up with class- and method comments.

Thus, the goal of this diploma thesis was to develop *J-PaD*, a tool for an integrated documentation of software modules. It supports the documentation process of Java packages through a highly customizable, graphical user interface. J-PaD has been designed as a plugin for the development environment Eclipse, so developers can use this tool within their familiar environment. The documentation for a module is kept in a separate file which is located next to the code files of a module.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
CASE	Computer-Aided Software Engineering
CBSE	Component Based Software Engineering
COCOMO	Constructive Cost Model
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
J-PaD	Java Package Documentation
JDK	Java Development Kit
MSDN	Microsoft Developer Network
PDF	Portable Document Format
PHP	PHP Hypertext Preprocessor
RTF	Rich Text Format
SQL	Structured Query Language
SWT	Standard Widget Toolkit
UML	Unified Modeling Language
URL	Uniform Resource Locator
WYSIWYG	What You See Is What You Get
XML	Extensible Markup Language

Kapitel 1

Einleitung

Software spielt heutzutage in den meisten Firmen eine wichtige Rolle und stellt einen beachtlichen Teil deren Kapitals dar. Die häufige Einführung von neuer Software ist oftmals mit Ausfällen bestimmter Dienste und damit Kosten verbunden, weshalb in vielen Firmen zunehmend Alt-Systeme im Einsatz sind. Die Alterung von Software (engl. *Software-Aging*) wird daher ein ernstzunehmender Faktor in der Industrie [Par94]. Für dieses Software-Aging lassen sich die folgenden Ursachen identifizieren: Die Software-Hersteller versäumen es oftmals, ihre Produkte an sich ändernde Anforderungen anzupassen. Die Erwartungen der Benutzer ändern sich im Laufe der Zeit. Kann die Software damit nicht schritthalten, wird diese als veraltet betrachtet. Führen die Hersteller dagegen Änderungen an ihrer Software durch, so degeneriert die Struktur der Software zunehmend mit jeder Änderung. Dies liegt häufig daran, dass die Wartungsingenieure das ursprüngliche Entwurfskonzept nicht vollständig verstehen und dennoch versuchen die Software anzupassen. Nach mehrerer solcher Anpassungen verliert auch der ursprüngliche Architekt den Überblick über die Software, was dazu führt, dass niemand mehr ein umfassendes Verständnis über das Software-System besitzt. Dieser Effekt wird verstärkt, wenn die Wartungsingenieure aus Zeitmangel das Aktualisieren der Dokumentation vernachlässigen. Die Dokumentation wird dadurch inkonsistent und zukünftige Wartungen werden zusätzlich erschwert.

Viele Studien zeigen, dass die Dokumentation von Software nicht erst während der Wartung vernachlässigt wird, sondern bereits in der Entwicklung schlecht strukturiert, unvollständig und ungenau ist [Par94, KM01, LSF03]. Wird die Dokumentation durch einen *Technical Writer* [Par94] erstellt, so sind diesem die technischen Details oftmals nicht bekannt. Dies führt zu einer oberflächlichen Dokumentation, welche von den Wartungsingenieuren größtenteils ignoriert wird. Selbst wenn die Dokumentation die nötigen Informationen enthält, ist oftmals nicht klar, wo diese wiederzufinden sind, da keine einheitliche Struktur vorgegeben ist.

Kajko-Mattson [KM01] zeigte, dass nur 53% der untersuchten Software-Firmen eine vollständige und konsistente Dokumentation nach der Entwicklung an die Wartung übergeben. Weiterhin definierten und verfolgten nur 42% der Firmen Richtlinien für die integrierte Codedokumentation. Lethbridge et al. [LSF03] stellen fest, dass Entwickler nach durchgeführten Änderungen am Programmcode die Dokumentation nur sehr selten nachziehen. In den Fällen, in welchen die Dokumentation angepasst wurde, fand dies oftmals mehrere Wochen

nach der vorgenommenen Code-Änderung statt. Häufig hat der Entwickler dann jedoch die Details der Änderung vergessen, was zu einer ungenauen Dokumentation führt.

Aus oben genannten Studien lassen sich die Schlussfolgerungen ziehen, dass Software-Entwickler zu Dokumentation neigen, die sich auf einfache Weise erstellen, verwenden und anpassen lässt [LSF03]. Umfangreiche CASE-Tools (*Computer-Aided Software Engineering*) werden im Allgemeinen als zu komplex und zeitraubend eingestuft. Dagegen werden z. B. Bug-Tracking Systeme als wichtige Informationsquellen angesehen. Diese erlauben es, auf einfache Weise einen Kommentar zu einem Fehlereintrag zu notieren, wodurch die Information anschließend schnell wiedergefunden werden kann.

1.1. Motivation

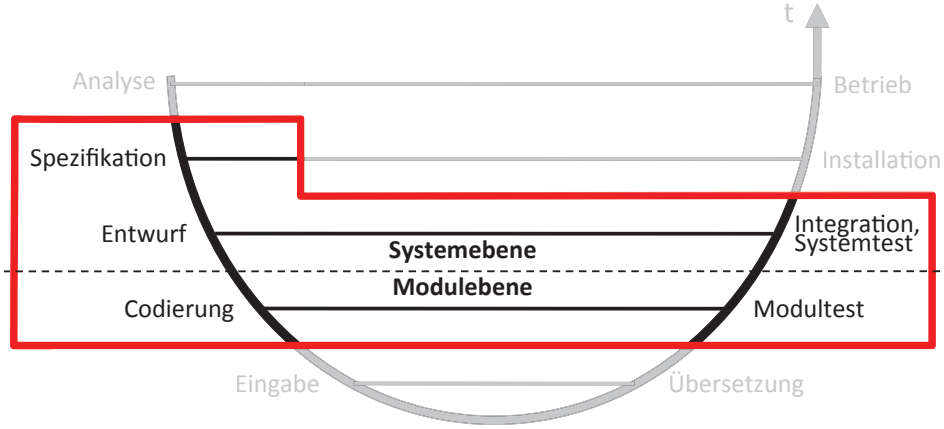

Abbildung 1.1.: Die Badewannenkurve – zeigt die verschiedenen Abstraktionsebenen während der Software-Entwicklung, nach [LL10, Boe79].

Die Badewannenkurve (Abb. 1.1) zeigt die zeitliche Abfolge der Aktivitäten bei der Software-Entwicklung. Die linke Hälfte beschreibt dabei einen top-down Ansatz, welcher ausgehend von der Arbeitsaufgabe bis zur Eingabe des Programmcodes reicht. Anschließend wird auf der rechten Hälfte daraus das Gesamtsystem bottom-up zusammengestellt. Aktivitäten desselben Abstraktionsniveaus sind durch horizontale Linien miteinander verbunden.

Der horizontale Schnitt in der Mitte der Kurve nimmt eine Unterteilung in die System- und die Modulebene vor. So werden für die Integration und den Systemtest die Vorgaben aus der Spezifikation und dem Entwurf herangezogen. Für den Modultest hingegen sollten die Vorgaben aus einer Modulspezifikation bzw. -dokumentation entnommen werden. Diese ist auf der Ebene der Codierung einzuordnen, wodurch sich gewisse Forderungen dafür ergeben: Die Dokumentation sollte eine Nähe zu den Code-Dateien eines Moduls aufweisen und mit gängigen Werkzeugen bei der Programmierung vereinbar sein. Dadurch ist die Wahrscheinlichkeit höher, dass die Dokumentation mit dem Programmcode nachgezogen wird und somit bei Änderungen konsistent bleibt.

Vorhandene Werkzeuge [Hee, Orab, Wol, Mar] unterstützen die integrierte Dokumentation von Software-Modulen nur sehr schlecht. Der Programmierer muss die Dokumentation als Kommentarblöcke innerhalb von Code-Dateien schreiben und diese nach gewissen Regeln formatieren. Dabei erwarten die meisten Werkzeuge ihre eigene Syntax, welche dem Entwickler jeweils bekannt sein muss, damit das Werkzeug richtig eingesetzt werden kann. Zudem existiert in den Code-Dateien oftmals keine Trennung zwischen der Dokumentation auf Klassen- und auf Modulebene. Dokumentation unterschiedlicher Granularität liegt somit gemischt vor. Will sich der Entwickler einen Überblick über ein Modul verschaffen, führt der Weg meist zu separater Dokumentation, welche aus den Kommentarblöcken generiert werden muss. Die ursprüngliche Nähe zum Programmcode geht dabei verloren und damit auch die Vorteile von integrierter Dokumentation.

1.2. Ziel

Das Ziel dieser Diplomarbeit ist daher die Entwicklung eines Werkzeugs zur integrierten Dokumentation von Software-Modulen. Die Beschreibung eines Moduls soll dabei in einer separaten Datei erfolgen, welche sich jedoch in der Nähe der Codierung eines Moduls befindet.

Dazu sollen zunächst bereits existierende Möglichkeiten zur integrierten Dokumentation von Modulen untersucht werden. Hierfür sind integrierte Entwicklungsumgebungen und andere Werkzeuge darauf zu überprüfen, ob diese eine Dokumentation auf Modulebene unterstützen. Zusätzlich soll überprüft werden, welche Usablility-Aspekte für ein Werkzeug zur integrierten Dokumentation beachtet werden müssen.

Das zu entwickelnde Werkzeug ist als ein Eclipse-Plugin zu implementieren, das eine integrierte Dokumentation für einzelne Pakete einer Java-Software ermöglicht. Die Darstellung und Eingabe der Daten soll dabei mit tabellarischen Formularen nach anpassbarem Schema erfolgen. Eine Anbindung der Software an JUnit soll möglich sein.

1.3. Gliederung der Arbeit

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen definiert zentrale Begriffe, die im weiteren Verlauf dieser Arbeit vorausgesetzt werden. Zusätzlich wird ein Überblick über die Erkenntnisse zur Dokumentation im Software-Engineering gegeben.

Kapitel 3 – Vorhandene Ansätze zeigt bereits existierende Möglichkeiten, welche das Dokumentieren während der Software-Entwicklung unterstützen. Dabei wird ein Überblick über Dokumentationskonzepte gegeben und es werden verschiedene Werkzeuge miteinander verglichen.

Kapitel 4 – Usability-Aspekte setzt sich mit der Usability von Software-Systemen auseinander. Daraus werden Usability-Anforderungen für ein Werkzeug zur Software-Entwicklung abgeleitet sowie existierende Dokumentationswerkzeuge in Bezug auf deren Usability untersucht.

Kapitel 5 – Anforderungen listet die detaillierten Anforderungen an das zu entwickelnde Werkzeug auf.

Kapitel 6 – Konzept stellt den eingeschlagenen Lösungsweg für das Werkzeug zur Dokumentation von Java-Paketen vor.

Kapitel 7 – Ergebnis präsentiert das entwickelte Werkzeug.

Kapitel 8 – Umsetzung geht auf technische Aspekte ein und beschreibt den gewählten Entwicklungsprozess.

Kapitel 9 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt mögliche Anknüpfungspunkte vor.

Kapitel 2

Grundlagen

Um dem Leser eine allgemeine Wissensbasis zu vermitteln, werden in diesem Kapitel die Grundlagen behandelt. Dafür werden zunächst Begriffe definiert, welche für das weitere Verständnis maßgeblich sind. Des Weiteren werden die Vorteile von Dokumentation in der Software-Entwicklung sowie die Auswirkungen von schlechter Dokumentation beleuchtet. Abschließend werden verschiedene Dokumentationsmöglichkeiten vorgestellt.

2.1. Begriffe und Definitionen

2.1.1. Dokumentation

Dokumentation ist in der Software-Entwicklung eine zentrale Tätigkeit. Jeder Schritt, wie die Planung, die Implementierung oder auch der Test, liefert mindestens ein Dokument, welches die Resultate festhält. Dabei lassen sich nach [LL10] folgende Begriffe unterscheiden:

Integrierte Dokumentation bezeichnet die im Programmcode enthaltenen Informationen. Dazu gehören Kommentare, die Namensgebung von Variablen und Methoden oder auch die Strukturierung des Programmcodes selbst.

Separate Dokumentation bezeichnet die außerhalb des Programmcodes existierenden Teile der Software. Dazu gehören unter anderem das Begriffslexikon, das Spezifikations- oder auch das Entwurfsdokument.

Dokumentation steht sowohl für integrierte als auch für separate Dokumentation.

Als ein **Dokument** wird im weiteren Verlauf dieser Arbeit jedes erstellte Artefakt während der Software-Entwicklung und -Wartung betrachtet. Die Code-Dateien stellen somit spezielle Dokumente dar.

2.1.2. Software-Module

Eine Definition für Software-Module findet sich in der IEEE-Norm [iee90]:

> ***module*** *– (1) A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to, or output from an assembler, compiler, linkage editor, or executive routine.*
>
> *(2) A logically separable part of a program.*
>
> IEEE Std 610.12 (1990)

Parnas [Par72] hat mit dem Information Hiding Kriterien festgelegt, nach denen Software-Systeme in Module zerlegt werden können. Dabei schlägt er vor, dass ein System nicht nach dessen Hauptfunktionalität zerlegt werden sollte, sondern primär anhand der schwierigsten und sich möglicherweise ändernden Entwurfsentscheidungen. Module verstecken dabei ihre interne Realisierung, sodass sich zukünftige Änderungen nur lokal auswirken. Dadurch ergeben sich folgende Vorteile:

- Die Entwicklungszeit verkürzt sich. Die Entwicklung kann entsprechend der Module auf mehrere Teams aufgeteilt werden, sodass diese nur wenig Kommunikationsaufwand untereinander haben.
- Die Software wird flexibler. Solange die Schnittstellen konstant bleiben, lassen sich größere Änderungen an einem Modul durchführen, ohne die anderen Teile des Systems zu beeinflussen.
- Der Programmcode ist einfacher zu verstehen, da die Module in sich abgeschlossen sind.

Soweit nichts anderes angegeben ist, werden in dieser Arbeit Module im Sinne von Java-Paketen betrachtet. Ein Modul ist somit Gegenstand eines Arbeitspaketes, das typischerweise von einem Programmierer oder einem Team aus Programmierern entwickelt und ausgeliefert wird.

2.2. Dokumentation im Software-Engineering

Während der Software-Entwicklung entstehen zahlreiche Dokumente. Diese lassen sich nach Abschnitt 2.1.1 der integrierten oder separaten Dokumentation zuordnen. Wenn möglich sollte die integrierte Dokumentation vorgezogen werden, da es dann bei zukünftigen Code-Änderungen wahrscheinlicher ist, dass diese mit angepasst wird. Dennoch ist die separate Dokumentation mindestens genauso wichtig. Alle Ergebnisse, die vor der Codierung entstehen, wie die Spezifikation oder das Begriffslexikon, werden in separaten Dokumenten festgehalten und sollten nicht in Code-Artefakten stehen.

Das Anfertigen von Software-Dokumentation erfordert einen nicht vernachlässigbaren Aufwand. Um dies wieder auszugleichen, müssen die Dokumente einen Nutzen haben, der

den Erstellungsaufwand übersteigt. Nach [LL10] lassen sich folgende Ziele mit einer guten Dokumentation erreichen:

- Entwickler können ihr Wissen über das entwickelte Programm mithilfe von Dokumenten untereinander austauschen. Dies ist vor allem für Personen, die die Software weiterentwickeln und warten müssen, besonders wertvoll. Gleichzeitig dienen die Dokumente zur Kommunikation zwischen Auftraggeber und Auftragnehmer.
- Werden die Informationen über die Software in Dokumenten festgehalten, können diese, z. B. über Reviews, systematisch überprüft werden. Dadurch ist es wahrscheinlicher, dass Fehler frühzeitig erkannt werden.
- Mithilfe von Dokumenten lässt sich der Projektfortschritt feststellen. Liegt beispielsweise ein Reviewprotokoll zu einer Spezifikation vor, kann damit nachgewiesen werden, dass diese überprüft wurde.
- Durch Dokumente kann nachgewiesen werden, dass gewisse Standards während der Entwicklung eingehalten wurden. Dies ist für Bereiche, wie Banken oder Versicherungen wichtig, da diese oft strengen Regeln unterworfen sind.

Dokumente werden dafür geschrieben, um dem Leser bestimmte Informationen zu übermitteln. Daher sollten die Dokumente so aufgebaut sein, dass diese sinnvoll benutzt werden können. [LL10] beschreibt daher folgende Eigenschaften, die für Dokumente gelten sollten:

- In den Dokumenten sind die Verfasser und die Prüfer festgehalten.
- Dokumente sollten nach Vorgaben und Richtlinien strukturiert und geordnet werden. Dies hat den Vorteil, dass sich ein fremder Leser schnell in einem Dokument zurechtfinden kann, da eine bereits bekannte Struktur verwendet wird. Zudem vereinfacht eine vorgegebene Struktur die Erstellung, da dadurch eine gewisse Vollständigkeit garantiert wird.
- Dokumente sind in elektronischer Form verfügbar.
- Dokumente werden in der Konfigurationsverwaltung archiviert.

2.2.1. Software-Wartung

Wird der gesamte Lebenszyklus einer Software betrachtet, so fallen für die Wartung weit mehr Kosten an als für die Entwicklung einer Software. Die Wartungskosten können dabei mehr als 65% der Gesamtkosten ausmachen [LL10, Fin90]. Ist die Dokumentation veraltet und unvollständig, so steigt der Einarbeitungsaufwand in der Wartung. Studien haben gezeigt, dass die meiste Zeit während der Wartung damit verbracht wird, ein bestehendes Software-System zu verstehen [Fin90]. Wird dagegen die Dokumentation aktuell gehalten, so kann der Aufwand und damit die Kosten für die Wartung deutlich reduziert werden.

COCOMO II (*Constructive Cost Model*) [BCH+00], ein Modell zur Aufwandsbestimmung in der Software-Entwicklung und -Wartung, berücksichtigt über den *Maintenance Adjustment*

Factor (MAF) Effekte bei der Wartung von Software. Dieser Faktor wirkt sich positiv auf den geschätzten Aufwand auf, wenn die Software eine hohe Modularität, aktuelle Dokumentation und eine hohe Übereinstimmung zwischen Programmcode und Anwendungsfällen aufweist. Dagegen steigt der geschätzte Aufwand, sobald die Software diese Eigenschaften nicht mehr erfüllt.

2.3. Klassifizierung von Dokumentationsmethoden

Software wird verstärkt aus bereits existierenden, wiederverwendbaren Komponenten entwickelt. Aktuelle Programmiersprachen, wie Java oder C#, enthalten bereits zahlreiche Bibliotheken, die einen hohen Grad an Wiederverwendung erlauben. In [Lea10] beschreibt Leavens die Notwendigkeit von Dokumentation für eine erfolgreiche Wiederverwendung von Software-Bibliotheken. Dabei gibt es für einen erfolgreichen Umgang mit Bibliotheken zwei Probleme zu lösen:

1. **Lernproblem:** Welche Teile der Software lassen sich mithilfe der Bibliothek realisieren? Vorausgesetzt wird dafür das Verständnis des Zwecks und der Fähigkeiten der Bibliothek.
2. **Referenzproblem:** Wie schnell lassen sich Antworten auf konkrete Fragen über die Bibliothek finden, um eine bestimmte Programmieraufgabe lösen zu können?

Um diese beiden Probleme anzugehen, beschreibt Leavens verschiedene Dokumentationsmethoden, wobei jede ihre Vor- und Nachteile besitzt. Die Methoden lassen sich in drei Hauptkategorien einteilen (siehe Abb. 2.1):

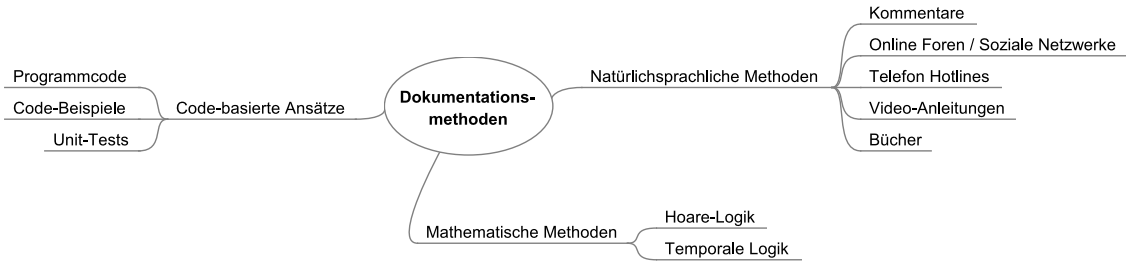

Abbildung 2.1.: Überblick über Dokumentationsmethoden für Software-Bibliotheken, nach [Lea10].

Code-basierte Ansätze – *Programmcode* ist eine mathematisch präzise Spezifikation einer Bibliothek. Dieser kann, abgesehen von der Namensgebung von Variablen oder Methoden, von Entwicklern unterschiedlicher Nationalität gleich verstanden werden. Dennoch ist der Programmcode meist zu detailliert, um das Lern- oder Referenzproblem damit beantworten zu können. Ein Nutzer einer Bibliothek sollte sich nicht auf

deren interne Datenstrukturen verlassen, da sich diese in neueren Versionen ändern können.

Code-Beispiele über die Nutzung einer Bibliothek können für einen unerfahrenen Entwickler sehr hilfreich sein. Ebenso können *Unit-Tests* das Verhalten einer Bibliothek beschreiben. Meist jedoch sind über Tests nicht alle Fälle abgedeckt und umfangreiche Testfälle, die zuerst einen komplexen Zustand herstellen müssen, erschweren das Lesen deutlich.

Natürlichsprachliche Methoden – Die am häufigsten eingesetzte Variante, um Bibliotheken zu beschreiben, ist natürlichsprachliche Dokumentation. Diese lässt sich relativ günstig erstellen, da keine spezielle Ausbildung für das Schreiben erforderlich ist. Die Dokumentation kann beliebig strukturiert werden und ermöglicht die Beschreibung der Bibliothek auf einer abstrakten Ebene, um damit das Lernproblem zu adressieren. Natürlichsprachliche Dokumentation ist weniger gut geeignet, um Fragen des Referenzproblems zu beantworten. Dies ergibt sich daraus, dass natürliche Sprache oft ungenau und mehrdeutig ist. Zudem ist es schwierig, automatisch Informationen aus natürlicher Sprache zu extrahieren oder diese zu verifizieren.

Diese Form der Dokumentation findet sich unter anderem als *Kommentare* direkt im Programmcode. *Online Foren* oder *Soziale Netzwerke* sind vor allem bei der Fehlersuche sehr hilfreich, da hier Experten direkt um Rat befragt werden können. Eine Antwort kann jedoch einige Zeit in Anspruch nehmen. Für eine unmittelbare Antwort eignen sich vor allem *Telefon-Hotlines*, welche jedoch durch das Bereitstellen von ausgebildetem Personal mit zusätzlichen Kosten verbunden sind. Um einen allgemeinen Überblick über eine Bibliothek zu erhalten, sind *Anleitungen per Video* sehr nützlich. Allerdings lassen sich, durch die eingeschränkte Suche innerhalb eines Videos, schlecht detaillierte Fragen über die Nutzung einer Bibliothek beantworten. *Bücher*, vor allem elektronisch vorliegende, eignen sich sowohl für das Lernproblem als auch für das Referenzproblem. Diese können neben natürlichsprachlichem Text auch Anwendungsbeispiele auf Code-Ebene enthalten.

Mathematische Methoden – Mit formalen Methoden lassen sich eindeutige und präzise Spezifikationen einer Bibliothek erstellen. Diese sind meist kompakter als ein konkreter, ausformulierter Testfall. Allerdings erfordert das Erstellen und das Verständnis von formalen Spezifikationen speziell ausgebildetes Personal mit einem fundierten mathematischen Wissen.

Mit *Hoare-Logik* lassen sich formale Spezifikationen erstellen, wobei zu jeder Methode Invarianten, Vor- und Nachbedingungen angegeben werden können. Detaillierte Fragen, die nur eine Methode betreffen, lassen sich darüber gut beantworten. Allerdings ist es schwierig die Effekte von vielen Methodenaufrufen zu bestimmen. Hierfür sind *Temporale Logiken* besser geeignet, da diese es erlauben, die zeitlichen Abläufe zu beschreiben. Zudem können sie die Abwesenheit von Deadlocks zeigen.

In der Literatur finden sich weitere Untergliederungen. So werden in [LL10] zusätzlich grafische Darstellungen aufgeführt. Hier ist vor allem die UML-Notation (*Unified Modeling*

Language) zu nennen, welche in den vergangenen Jahren populär wurde. Darin sind einige Notationen enthalten, um eine Software auf unterschiedlichen Abstraktionsebenen zu beschreiben, wie z. B. über Paket-, Klassen- oder Sequenzdiagramme [Obj].

Diese Arbeit setzt den Schwerpunkt auf natürlichsprachlichen Dokumentationsmöglichkeiten, welche in Form von Kommentaren niedergeschrieben sind. Diese integrierte Dokumentationsform befindet sich in direkter Nähe zum Programmcode und ist somit stets verfügbar. Zudem ist es dadurch wahrscheinlicher, dass bei Code-Änderungen die Dokumentation mit angepasst wird.

Kapitel 3

Vorhandene Ansätze

In Kapitel 2 wurde die Bedeutsamkeit von Dokumentation während der Software-Entwicklung dargestellt. Es wurde gezeigt, welchen Nutzen eine Dokumentation mit sich bringt und welche Möglichkeiten zur Dokumentation von Software existieren. In diesem Kapitel sollen nun bestehende Ansätze zur Software-Dokumentation untersucht werden. Dazu werden Konzepte aus der Literatur vorgestellt sowie existierende Werkzeuge näher betrachtet.

3.1. Überblick über die Literatur

3.1.1. Literate Programming

Knuth beschreibt mit Literate Programming [Knu84] ein Konzept, bei dem die Verständlichkeit von Programmcode verbessert werden soll. Dabei betrachtet er ein Programm als ein literarisches Werk. Dieses soll nicht dafür geschrieben werden, um einen Computer zu instruieren, sondern es soll in erster Linie dem Leser des Programmcodes erklären, welche Aufgabe der Computer ausführen soll. Ein *literate* Programm besteht im Wesentlichen aus der Vermischung von Programmcode und Dokumentation. Die Grundidee ist, dass beide Elemente zusammen entstehen. Dies soll zu besser verständlichen und weniger fehleranfälligen Programmen führen.

Knuths Literate Programming System besteht aus drei Sprachen:

- Pascal als Programmiersprache
- TEX für die Formatierung der Dokumentation
- WEB als Verbindungssprache, welche hauptsächlich aus der Kombination der Programmier- und Formatierungssprache besteht.

Abbildung 3.1 zeigt das Vorgehen beim Literate Programming. Der Entwickler schreibt das Programm und die Dokumentation in der Sprache WEB, wobei hier zunächst Anweisungen der Programmier- und Formatierungssprache in einer *.web*-Datei vermischt sind. Es handelt sich hierbei also um ein *integriertes* Dokumentationskonzept (siehe Abschnitt 2.1.1). Anschließend folgen zwei Pfade: Einer erzeugt die Dokumentation und der andere das ausführbare Programm. Der erste Pfad *WEAVE* (engl. für *weben*) extrahiert die Dokumentation aus der WEB-Sprache und erzeugt daraus eine wohlgeformte *.tex*-Datei, welche als Eingabe für ein TEX-System dient. Dieses erstellt die formatierte Ausgabe in einer für den Menschen lesbaren Form. Im zweiten Pfad *TANGLE* (engl. für *verwickeln*) wird der Programmcode aus der Eingabe verwendet und an einen Pascal-Compiler weitergereicht. Dieser erzeugt anschließend das ausführbare Programm.

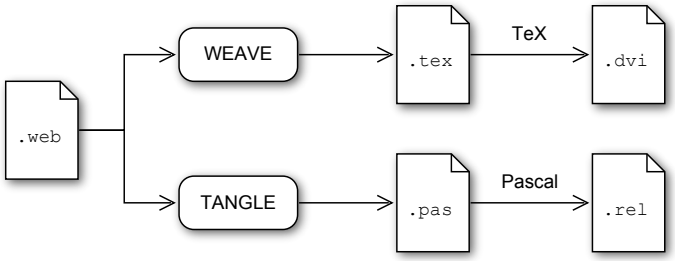

Abbildung 3.1.: Literate Programming Konzept – Der *WEAVE*-Prozess erzeugt die Dokumentation, während *TANGLE* das ausführbare Programm erstellt, nach [Knu84].

Ein WEB-Programm besteht aus mehreren Abschnitten, wobei ein Abschnitt in drei optionale Teile aufgeteilt ist. Der erste Teil dient dazu, den Abschnitt zu dokumentieren. Im zweiten Teil lassen sich Makrodefinitionen angeben, welche syntaktische Ersetzungen durchführen können, während der letzte Teil den Programmcode des Abschnitts enthält. Die Abschnitte werden so angeordnet, wie es für das menschliche Verständnis geeignet ist. Codeblöcke eines Abschnitts können zudem Verweise auf Codeblöcke eines anderen Abschnitts enthalten, wodurch sich unterschiedliche Abstraktionsebenen erreichen lassen.

Bewertung

Beim Literate Programming ist keine Entwicklungsrichtung wie top-down oder bottom-up vorgegeben. Das Programm wird hier als ein zusammenhängendes Netz aus Abschnitten gesehen. Diese werden nach psychologischen Aspekten angeordnet, wobei die Verständlichkeit des Codes im Vordergrund steht. Ein weiterer Vorteil ergibt sich aus der Tatsache, dass Programmcode und Dokumentation zusammen in einer Datei entstehen und somit eine hohe Wahrscheinlichkeit aufweisen, konsistent zu bleiben. Zudem stellt Knuth fest, dass sich die Zeit für das Fehlersuchen und -beheben reduziert, da sich der Entwickler durch das Ausformulieren der Dokumentation gedanklich mehr mit dem Programm auseinandersetzt und daher Fehler schon während der Eingabe erkennen kann.

Aguiar und David [AD05] argumentieren hingegen, dass Literate Programming in der Praxis einige Schwächen aufzeigt. So stellt sich die Kombination von drei Sprachen, wie Programmier-, Formatierungs- und Verbindungssprache als ungeeignet dar. Das Format eines *literate* Programms ist sehr komplex, wodurch die Lesbarkeit am Bildschirm und somit das Verständnis während der Entwicklung beeinträchtigt wird. Das Hauptproblem besteht jedoch darin, dass der vom Entwickler eingegebene Programmcode sich vom Code, der an den Compiler weitergereicht wird, unterscheidet. Der *TANGLE*-Prozess erzeugt aus dem *literate* Programm ein sekundäres Artefakt, welches nicht dafür gedacht ist, verändert zu werden. Durch die Unterscheidung der Code-Dateien kann es zu Integrationsproblemen mit Werkzeugen führen, welche direkt vom Programmcode abhängig sind. Die Benutzung von integrierten Entwicklungsumgebungen oder Debuggern stellt sich daher als schwierig heraus.

Shum und Cook [SC94] berichten von der Anwendung von Literate Programming in der Lehre. Gruppen von Studenten bewältigten dabei Programmieraufgaben einmal mit und einmal ohne den Literate Programming Ansatz. Das Ergebnis zeigte, dass mit Literate Programming die Dokumentation umfangreicher und besser ausfiel als mit einem traditionellen Ansatz. Die Probanden merkten jedoch auch an, dass sich die Fehlersuche im Programm als sehr schwierig darstellte.

3.1.2. Elucidative Programming

Elucidative Programming stellt eine von Nørmark [Nør00] entwickelte Variante zu Literate Programming dar. Diese versucht die oben beschriebenen Probleme des Literate Programming Konzepts zu beseitigen. Hierfür erfolgt die Dokumentation in einer separaten Datei, wodurch der Code nicht verändert werden muss. Verbinden lässt sich die Dokumentation mit dem Programmcode, indem Referenzen auf Elemente, wie Klassen oder Methoden gesetzt werden.

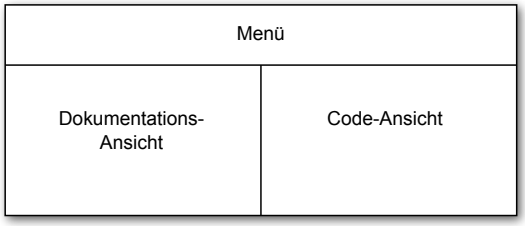

Abbildung 3.2.: Elucidative Programming Fensterlayout – Ansicht einer Elucidative Programming Entwicklungsumgebung, mit Dokumentation und Programmcode nebeneinander, nach [Nør00].

Die Benutzungsoberfläche der Entwicklungsumgebung stellt beim Elucidative Programming ein zentrales Element dar. Diese ist in zwei Hauptansichten aufgeteilt, wobei Dokumentation und Programmcode direkt nebeneinander bearbeitet werden können (siehe Abb.

3.2). Die Nähe von Dokumentation und Programmcode wird beim Elucidative Programming als *navigierbare Nähe* bezeichnet. Diese ist schwächer als die *physikalische Nähe* im Literate Programming, erlaubt es aber, mehrere Programmelemente aus unterschiedlichen Code-Einheiten zusammen in einem Abschnitt zu dokumentieren. Dadurch lassen sich z. B. Entwurfsmuster in objektorientierten Programmen übersichtlich beschreiben.

3.1.3. XSDoc Wiki

Auf der Grundidee von Elucidative Programming baut [AD05] mit dem Werkzeug XSDoc Wiki (*Extensible Software Documentation*) auf. Dabei wird ein separates Dokumentationskonzept verfolgt, um verschiedene Inhaltstypen, wie Texte, Bilder oder Codefragmente in der Dokumentation zu integrieren. Sämtliche Inhalte werden hierfür in das XML-Format transformiert und zentral gespeichert.

Die Bearbeitung der Dokumentation erfolgt über ein Wiki-System. Texte lassen sich damit einfach formatieren und es können zusätzliche Inhalte eingebunden werden. Die Einbindung erfolgt dabei nach dem Prinzip von Elucidative Programming, indem mithilfe von Schlüsselwörtern und dem vollständig qualifizierten Namen auf ein entsprechendes Codefragment verwiesen wird. Externe Inhalte werden jedoch erst beim Generieren der Anzeige in die Dokumentation eingebunden, um so die Aktualität aller Elemente zu gewährleisten. Auf analoge Weise können auch UML-Diagramme der Dokumentation hinzugefügt werden.

XSDoc Wiki wurde als Eclipse-Plugin für die Programmiersprache Java entwickelt. Damit lässt sich die Dokumentation direkt neben dem Programmcode anzeigen und bearbeiten. Leider ist dieses Werkzeug heute nicht mehr auffindbar.

3.2. Werkzeuge zur integrierten Codedokumentation

Das Internet fördert eine Vielzahl von Werkzeugen zur integrierten Dokumentation von Programmcode zu Tage. Einen guten Einstieg hierzu bietet Wikipedia [Wik]. Diesen Werkzeugen gemein ist, dass sie die enthaltenen Kommentare im Code extrahieren und diese zusammen mit Schnittstelleninformationen in eine formatierte, separate Dokumentation überführen. Im Folgenden werden einige der am weitesten verbreiteten und in Bezug auf die Themenstellung dieser Arbeit relevanten Werkzeuge näher betrachtet.

3.2.1. Doxygen

Doxygen [Hee] ist ein Dokumentationsgenerator, der hauptsächlich zur Erstellung von API-Dokumentation (*Application Programming Interface*) verwendet wird. Dieser unterstützt zahlreiche Programmiersprachen, wie C, C++, Java oder Fortran und kann die Dokumentation in mehrere Ausgabeformate überführen. Neben der PDF- oder RTF-Ausgabe sind

HTML-Seiten die gängigste Variante. Diese erlauben Ansichten mit unterschiedlichem Detailgrad und sind untereinander navigierbar gestaltet. So kann eine Übersicht mit allen Klassen dargestellt werden, welche die Beschreibungen aus deren Kopfkommentaren enthält. Weiterhin lassen sich zu einer bestimmten Klasse die darin enthaltenen Methoden und Attribute anzeigen, welche ebenfalls mit den entsprechenden Kopfkommentaren versehen sind. Auf der untersten Detailebene erlaubt es Doxygen, den Code direkt in die Dokumentation mit einzubinden. Kommentare können im Programmcode mit Schlüsselwörtern angereichert werden, um so z. B. die Parameter und den Rückgabewert einer Methode zu dokumentieren. Dadurch werden diese in der Ausgabe übersichtlich formatiert.

Bewertung

Doxygen lässt sich sehr umfangreich konfigurieren und bietet in der aktuellen Version über 200 Parameter an. Die Konfiguration kann wahlweise über eine grafische Oberfläche erfolgen, die es auch erlaubt, nur die wichtigsten Parameter zu setzen, oder direkt durch das Anpassen einer Konfigurationsdatei. Letztere Variante eignet sich vor allem, falls das Erstellen der Dokumentation mit einem Buildskript verbunden ist, sodass diese stets aktuell gehalten wird. Eine interessante Möglichkeit bietet Doxygen durch das Zusammenfassen von mehreren Elementen zu Modulen. Diese Module dienen der Gruppierung von mehreren Klassen oder Funktionen zu einer logischen Einheit, welche sich als Ganzes dokumentieren lässt. Die Strukturierung des Codes ist davon jedoch nicht betroffen. Zusätzlich kann über das Setzen bestimmter Schlüsselwörter in Kommentarblöcken eine allgemeine Hauptseite erstellt werden, welche einen Überblick über die dokumentierte Programmkomponente bietet.

Neben dem hohen Funktionsumfang von Doxygen fällt jedoch negativ auf, dass sämtliche Inhaltselemente der Dokumentation durch das Setzen bestimmter Schlüsselwörter innerhalb von Kommentaren gesteuert werden müssen. In einem Team von Entwicklern muss daher zunächst eine Konvention festgelegt werden, die vorschreibt, welche Elemente dokumentiert werden sollen. Dabei ist es wichtig, dass diese Konvention von den Entwicklern auch eingehalten wird. Zudem müssen die Schlüsselwörter einem Entwickler bekannt sein, damit das Werkzeug eine sinnvolle Dokumentation erstellen kann.

3.2.2. Javadoc

Javadoc [Orab] ist ein sehr ähnliches Werkzeug wie Doxygen, jedoch speziell auf die API-Dokumentation von Java-Programmen ausgelegt. Das Ausgabeformat der Dokumentation kann über sogenannte Doclets beeinflusst werden, wobei standardmäßig die Ausgabe in HTML-Dateien unterstützt wird. Mithilfe von Kommentarblöcken lassen sich Elemente unterschiedlicher Granularität, wie Methoden, Klassen oder Pakete dokumentieren. Diese Kommentare können mit vorgegebenen Schlüsselwörtern erweitert werden, um so z. B. Referenzen auf andere Elemente zu setzen oder direkt einen Beispielcode in die Dokumentation mit aufzunehmen.

Bewertung

Positiv fällt bei Javadoc auf, dass dies in der Entwicklungsumgebung Eclipse bereits integriert ist. In Eclipse lässt sich die Dokumentation zu einem Projekt über einen Assistenten erstellen. Zusätzlich werden die Kopfkommentare als Hilfestellung bei der Codierung genutzt, welche direkt bei der Verwendung von Methoden oder Klassen angezeigt wird. Ein weiterer interessanter Aspekt stellt die Dokumentation von Paketen dar. Diese lassen sich über eine separate Datei beschreiben, welche ebenfalls Javadoc-spezifische Schlüsselwörter enthalten kann. Javadoc erstellt daraufhin eine Übersichtsseite zu einem dokumentieren Paket. Weiterhin lässt sich eine Übersichtsseite über ein ganzes Projekt erstellen, welche als Hauptseite in der HTML-Ausgabe gesetzt wird.

Eine Schwierigkeit beim Umgang mit Javadoc, wie auch bei Doxygen, stellt die Verwendung der Schlüsselwörter und das direkte Einbinden von HTML-Formatierungselementen in den Kommentarblöcken dar. Diese machen die Kommentare innerhalb des Programmcodes schlecht lesbar und müssen von dem Entwickler erlernt und richtig eingesetzt werden.

3.2.3. AdaBrowse

AdaBrowse [Wol] stellt das Äquivalent zu Javadoc dar, jedoch ist dieses Werkzeug für die Dokumentation von Ada 95 Programmcode ausgelegt. Die Ausgabe der Dokumentation kann entweder in HTML-Seiten erfolgen, oder alternativ in ein XML-Format, das sich für die Weiterverarbeitung eignet, um die Konvertierung in beliebige Formate fortzusetzen. AdaBrowse extrahiert die Kopfkommentare zu Programmeinheiten, wie Pakete, Prozeduren oder Konstanten und schreibt diese zusammen mit den entsprechenden Signaturen in die HTML-Ausgabe. Anders jedoch als bei Javadoc kennt AdaBrowse standardmäßig keine Schlüsselwörter, um Elemente, wie Parameter oder Rückgabewerte separat in der Ausgabe zu beschreiben. Ist ein solches Verhalten gewünscht, so muss dies über eine Konfigurationsdatei definiert werden.

Bewertung

Die Dokumentation mit AdaBrowse bietet eine gute Übersicht über die dokumentierte Programmkomponente. In der HTML-Ausgabe werden alle Pakete hierarchisch aufgelistet und können jeweils für sich betrachtet werden. Die Programmiersprache Ada bietet mithilfe von Paketen ein interessantes Konzept, das den Modulbegriff nach Parnas [Par72] direkt umsetzt. Dabei lässt sich die Implementierung von der Spezifikation trennen und gleichzeitig durch öffentliche und private Bereiche das Information Hiding anwenden. AdaBrowse verwendet für die Erstellung der Dokumentation die Spezifikationsdateien und berücksichtigt standardmäßig nur deren öffentlichen Bereich. Ein Entwickler, der seine Programmkomponenten mit AdaBrowse dokumentieren möchte, muss sich nicht an vorgegebene Schlüsselwörter halten. Dafür sind jedoch die Elemente in der Ausgabe weniger strukturiert.

3.2.4. Document! X

Das Dokumentationswerkzeug Document! X [Inn] wird von der britischen Firma Innovasys entwickelt. Document! X bringt eine grafische Oberfläche mit, mit der sich bestehende Projekte laden lassen und die Dokumentation dazu im WYSIWYG-Stil (*What You See Is What You Get*) verfasst werden kann. Unterstützt werden dabei mehrere Programmiersprachen wie C#, Visual Basic .NET, Java oder auch SQL. Das Werkzeug erkennt abhängig von der Programmiersprache die dafür typischen Schlüsselwörter innerhalb von Kommentarblöcken und stellt diese gesondert in der Ausgabe dar. So werden für Java-Projekte Kommentare nach der Javadoc Konvention berücksichtigt, während bei .NET-Projekten die Kommentarblöcke mit XML-Elementen versehen werden. Kommentarblöcke lassen sich für Attribute und Methoden einer Klasse setzen. Bei letzteren können zudem weitere Informationen zu Parametern und zum Rückgabewert angegeben werden. Weiterhin lassen sich zu Klassen und Namensräumen jeweils allgemeine Beschreibungen vornehmen.

Bewertung

Im Gegensatz zu den bisher betrachteten Werkzeugen hebt sich Document! X durch die GUI-unterstützte Bearbeitung von Kommentarblöcken hervor. Für .NET-Sprachen bietet Document! X ein Visual Studio Plugin für das Erstellen der Dokumentation an. Dazu lassen sich Klassen in einem separaten Dokumentationsfenster öffnen und deren Elemente können bequem in einem WYSIWYG-Editor kommentiert werden. Die damit erzeugten Kommentare werden zusammen mit XML-Formatierungselementen direkt als Kopfkommentare an die entsprechenden Elemente im Programmcode geschrieben.

3.2.5. Doc-O-Matic

Das Dokumentationswerkzeug Doc-O-Matic [too] bringt ebenfalls eine grafische Oberfläche mit, in der sich die Dokumentation bearbeiten und in diverse Formate exportieren lässt. Darin können Projekte unterschiedlicher Programmiersprachen wie C/C++, C#, Java, PHP und einige weitere geladen werden. Die Projektstruktur wird in einer Hierarchie angezeigt und die einzelnen Elemente lassen sich über einen WYSIWYG-Editor kommentieren. Die Kommentare werden dabei direkt in die Code-Dateien zurückgeschrieben. Doc-O-Matic unterstützt dafür verschiedene Stile, um den Konventionen der entsprechenden Programmiersprache gerecht zu werden. So werden für Java Kommentare nach Javadoc-Regeln gesetzt, für .NET werden die Kommentare mit XML-Schlüsselwörtern erweitert und zusätzlich erlaubt es Doc-O-Matic die Kommentare in einem natürlichsprachlichen Format zu schreiben. Dabei werden die Kommentarblöcke anhand von Einrückungen und Trennzeichen formatiert.

Bewertung

Durch das Bearbeiten der Dokumentation in einem separaten Texteditor ermöglicht es Doc-O-Matic, ansprechendere Dokumentation als andere Werkzeuge zu erzeugen. Zudem bietet Doc-O-Matic eine Treemap-Ansicht des geladenen Projekts an, mit der der Fortschritt der Dokumentation verfolgt werden kann (siehe Abb. 3.3). Diese grenzt dokumentierte und undokumentierte Programmelemente farblich voneinander ab. Die Code-Dateien werden darin entsprechend der Programmstruktur geschachtelt, wobei die innersten Rechtecke Methoden darstellen. Damit lässt sich schnell erkennen, wie vollständig die Dokumentation bereits ist.

Leider bietet Doc-O-Matic bisher keine Integration in eine Entwicklungsumgebung an. Dadurch werden die Codierung und die Kommentierung über zwei separate Programme voneinander getrennt. Der Entwickler wird es daher vermutlich bevorzugen, die Kommentare während der Codierung weiterhin manuell einzugeben.

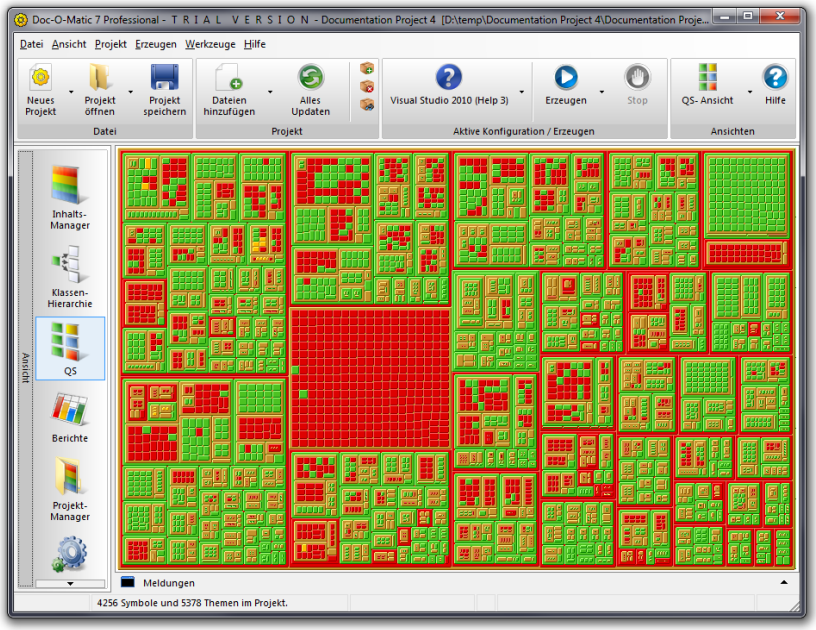

Abbildung 3.3.: Screenshot Doc-O-Matic Treemap – Gibt einen Überblick über dokumentierte und undokumentierte Programmelemente eines Projekts. *Rot* eingefärbte Stellen enthalten noch keine Kommentare, während *grüne* Bereiche bereits kommentiert sind. *Gelbe* Stellen zeigen vererbte Methoden an, die keine eigenen Kommentare besitzen, sondern diese von ihrer übergeordneten Methode erben.

3.2.6. Haddock

Haddock [Mar] ist ein Dokumentationsgenerator für die Programmiersprache Haskell. Dieser erzeugt aus den Kommentaren von Funktionen und Klassen eine übersichtliche Ausgabe in diversen Formaten wie HTML-Seiten oder auch LaTeX-Dateien. Kommentare werden dabei in einer natürlichsprachlichen Art geschrieben, welche ohne die Verwendung von Schlüsselwörtern auskommt. Es kann jedoch durch das Platzieren von Sonderzeichen Einfluss auf die Formatierung eines Kommentarblocks in der Ausgabe genommen werden. Damit lassen sich Listen, Texthervorhebungen oder auch Referenzen auf andere Programmelemente setzen. Es findet jedoch keine automatische Gliederung in mehrere Abschnitte statt, worüber sich z. B. Parameter oder Rückgabewerte separat beschreiben lassen.

Bewertung

Haddock stellt sich als interessant heraus, da es das Modulsystem der Haskell-Programmiersprache direkt unterstützt. Dabei ist es üblich, dass eine Programmkomponente zunächst in mehrere Module unterteilt wird. Für eine API ist es jedoch oft wünschenswert, dass diese aus nur genau einem öffentlichen Modul besteht. Dazu lässt sich in Haskell ein weiteres Modul erstellen, welches die relevanten Teile aller internen Module exportiert und nur diese dem Benutzer der API zugänglich macht. Dieses Prinzip übernimmt Haddock für die erzeugte Dokumentation, indem es dort die nicht exportieren Elemente und die interne Modulstruktur versteckt. Kommentare können dennoch direkt bei der Implementierung stehen, da Haddock diese an das öffentliche Modul weiter propagiert.

Im Vergleich zu anderen Dokumentationswerkzeugen fällt bei Haddock negativ auf, dass sich die Kommentarblöcke nicht automatisch in mehrere Abschnitte strukturieren lassen. Ist eine Untergliederung der Beschreibung einer Funktion nach Parametern und deren Rückgabewert erwünscht, so muss dies mit den vorhandenen Formatierungsoptionen pro Kommentarblock manuell definiert werden.

3.3. Vergleich der Werkzeuge

Tabelle 3.1 zeigt einige Merkmale der zuvor betrachteten Werkzeuge im Vergleich. Die Spalten besitzen dabei folgende Bedeutung:

Version – Die derzeit aktuelle Version.

Projekt – Gibt an, ob sich die dokumentierte Programmkomponente als Ganzes beschreiben lässt. Hierzu zählt jede Art von Übersichtsseite, die die enthaltenen Elemente des Projekts auflistet und eine allgemeine Beschreibung darstellen kann.

Module – Die Spalte beschreibt, ob es das Dokumentationswerkzeug erlaubt, Module zu dokumentieren. Als Module werden hierbei Pakete aus Java und Ada, Namensräume aus .NET-Sprachen und das Modulsystem aus Haskell betrachtet.

IDE-Integration – Zeigt an, ob es möglich ist, das Werkzeug aus einer Entwicklungsumgebung heraus zu verwenden. Hierbei zählt nur, ob dies von dem Werkzeug oder einer Entwicklungsumgebung direkt unterstützt wird. Plugins von Drittanbietern werden nicht berücksichtigt.

WYSIWYG – Gibt an, ob die Bearbeitung der Dokumentation durch eine grafische Benutzungsoberfläche unterstützt wird.

Status – Stellt dar, ob es eine Möglichkeit gibt, die Vollständigkeit der Dokumentation festzustellen.

Werkzeug	Version	Projekt	Module	IDE-Integration	WYSIWYG	Status
XSDoc Wiki	- [1)]	✓	✓	✓		
Doxygen	1.7.5.1	✓	✓			
Javadoc	5.0	✓	✓	✓		
AdaBrowse	4.0.3		✓			
Document! X	2011	✓	✓	✓	✓	
Doc-O-Matic	7.0.1		✓		✓	✓
Haddoc	2.9.4		✓			

1) Werkzeug nicht mehr verfügbar

Tabelle 3.1.: Vergleich der Dokumentationswerkzeuge

Als interessant stellt sich beim Vergleich heraus, dass jedes der hier betrachteten Werkzeuge eine Dokumentation auf Modulebene erlaubt. Die Möglichkeiten zur Gruppierung von mehreren Programmeinheiten zu Modulen unterscheiden sich jedoch zwischen den einzelnen Werkzeugen. So wird bei Javadoc ein Java-Paket als ein Modul aufgefasst, welches weitere Module enthalten kann. Die Dokumentation eines Pakets erfolgt über eine separate Datei, welche sich direkt im entsprechenden Paket befindet, wodurch ein hoher Zusammenhalt gewährleistet wird. Bei Doxygen dagegen können Module weit allgemeiner definiert werden. Dabei werden in Kopfkommentaren von Klassen und Funktionen Gruppen definiert und darüber die Zugehörigkeit von Programmeinheiten gesteuert. Die Programmeinheiten müssen hier keine physikalische Nähe aufweisen.

3.4. Konzepte in Entwicklungsumgebungen

Moderne integrierte Entwicklungsumgebungen (IDE) bieten heute zahlreiche Funktionen an, um den Softwareentwicklungsprozess zu unterstützen. Dies geht inzwischen weit über deren ursprüngliche Funktionalität hinaus: So waren die ersten Entwicklungsumgebungen lediglich eine Vereinigung von Editor, Compiler und Debugger [Boe03]. Heute bieten IDEs, neben diesen minimalen Anforderungen, weitere umfangreiche Werkzeuge an. Dazu zählen

beispielsweise Möglichkeiten zum Refactoring, automatische Codeerzeugung, Darstellung von API-Informationen während der Eingabe, Anbindung an die Konfigurationsverwaltung, Datenbankclients und viele mehr.

In diesem Abschnitt soll daher untersucht werden, welche Dokumentationsmöglichkeiten und Konzepte auf Modulebene von integrierten Entwicklungsumgebungen unterstützt werden.

Die Entwicklungsumgebung **IntelliJ IDEA** von JetBrains [Jet] bietet eine Möglichkeit, um die Abhängigkeiten zwischen Klassen verschiedener Projekte zu visualisieren. Dafür wird eine Matrix aufgebaut, in deren Zeilen und Spalten zunächst die Projekte aufgelistet werden. Ein Eintrag in einer Zelle der Matrix weist nun auf eine Abhängigkeit zwischen zwei Projekten hin. Dabei gilt für eine Zelle mit dem Zeilenindex i und dem Spaltenindex j: Falls $j < i$, existiert mindestens eine Abhängigkeit von Projekt j auf Projekt i. Ist $j > i$, so ist das Projekt i von Projekt j abhängig. Die Hauptdiagonale enthält somit keine Einträge. Befinden sich auf der oberen Dreiecksmatrix keine Einträge, so lässt sich daraus direkt ableiten, dass zwischen den Projekten keine zyklischen Abhängigkeiten bestehen. Die erste Ebene der Ansicht, die die Projektstruktur zeigt, lässt sich verfeinern, sodass dieses Prinzip auch auf der Paket- und Klassenebene angewendet werden kann.
Zusätzlich bietet IntelliJ IDEA einen Mechanismus an, um Abhängigkeiten explizit zu verbieten. Dafür lassen sich Regeln definieren, die z. B. eine Abhängigkeit von Paket *a* auf Paket *b* untersagen. Die Verletzung einer solchen Regel wird im Code direkt als Fehler angezeigt.

Visual Studio [Mic] bietet verschiedene Möglichkeiten, um Code-Komponenten zu visualisieren. So können z. B. zu einem C#-Projekt die Abhängigkeiten zwischen Namespaces über einen Graphen angezeigt werden. Weiterhin lassen sich UML-Klassendiagramme aus dem Code erzeugen, welche eine direkte Bearbeitung von Klassen- und Methodenbezeichnern erlauben.

Für **Eclipse** [Thea] existieren zahlreiche Erweiterungen, wodurch sich die Entwicklungsumgebung sehr vielseitig einsetzen lässt.
Die Erweiterung *SE-Editor* [Con] ermöglicht eine direkte Einbindung externer Inhalte in die Code-Ansicht. Hierfür muss im Code ein Kommentarblock mit einer bestimmten Syntax eröffnet werden, welcher eine URL auf den externen Inhalt spezifiziert. Dieser externe Inhalt wird anschließend geladen und in einem Bereich direkt über dem kommentierten Codefragment eingebunden. Auf diese Weise können neben einfachen textuellen Kommentaren auch Grafiken, Videos oder formatierte Texte in der Code-Ansicht dargestellt werden. Das Einbinden von kompletten Webseiten sollte jedoch nur bedingt eingesetzt werden, da keine harte visuelle Trennung zwischen Code und externen Inhalten erfolgt. Bei zu vielen eingebundenen Inhalten kann somit die Übersicht über den Programmcode sehr stark beeinträchtigt werden.
Die Eclipse-Erweiterung *JDocEditor* [Cer] erlaubt die Bearbeitung von Javadoc-Kommentaren in einem Richtext-Editor. Hierfür wird ein Kommentarblock in der Code-Ansicht ausgewählt, welcher anschließend in einer zweiten Ansicht über einen WYSIWYG-Stil bearbeitet werden kann.
Die Erweiterung *iDocIt!* [ME] bietet eine Möglichkeit, um Javadoc-Kommentare über eine

grafische Oberfläche für verschiedene Projektbeteiligte zu setzen. So kann z. B. ein Entwickler technische Details dokumentieren, während ein Tester Informationen für Testfälle festhält. Durch das Zusammenführen von Dokumentationen aus verschiedenen Sichtweisen soll die Kollaboration im Team verstärkt werden.

3.5. Weitere Beschreibungsmöglichkeiten für Software

Im folgenden Abschnitt wird die Dokumentation von Software auf höherer Ebene untersucht. Dafür wird von der Codeebene abstrahiert und die Software als Zusammenfassung von einzelnen Komponenten bzw. Bausteinen betrachtet. Es wird geprüft, ob sich hieraus Möglichkeiten ergeben, diese Ansätze auf die Dokumentation von Software-Modulen anzuwenden.

3.5.1. Software-Komponenten

Component Based Software Engineering (CBSE) macht sich die Wiederverwendung von bereits existierenden Software-Komponenten zu nutze. Diese können separat entwickelt und getestet werden und ermöglichen somit ein hohes Einsparpotential bei der Entwicklung neuer Software. Eine Definition für Software-Komponenten findet sich in [Szy99]:

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

Dabei wird explizit die Verwendung von Komponenten durch Dritte erwähnt, weshalb eine Dokumentation davon unerlässlich ist. Yacoub et al. [YAM99] beschreiben daher ein Klassifizierungsschema für Software-Komponenten, um diese gezielt in der Anwendungsentwicklung wiederzuverwenden. Abbildung 3.4 zeigt eine Übersicht dieses Schemas. Für eine detaillierte Beschreibung aller Eigenschaften wird auf [YAM99] verwiesen.

Die Klassifikation wird auf der ersten Ebene in folgende drei Hauptkategorien eingeteilt:

Informale Beschreibung enthält allgemeine Beschreibungen über die Komponente, wie das Alter, in welchem Kontext sich die Komponente nutzen lässt oder auch den Zweck der Komponente bzw. welche Probleme sich damit lösen lassen.

Externe Sicht beschreibt die Interaktion der Komponente mit anderen Artefakten und der Plattform, auf der die Komponente ausgeführt wird.

Interne Sicht enthält Realisierungsdetails über die Komponente, wie die Struktur oder Verhaltensaspekte.

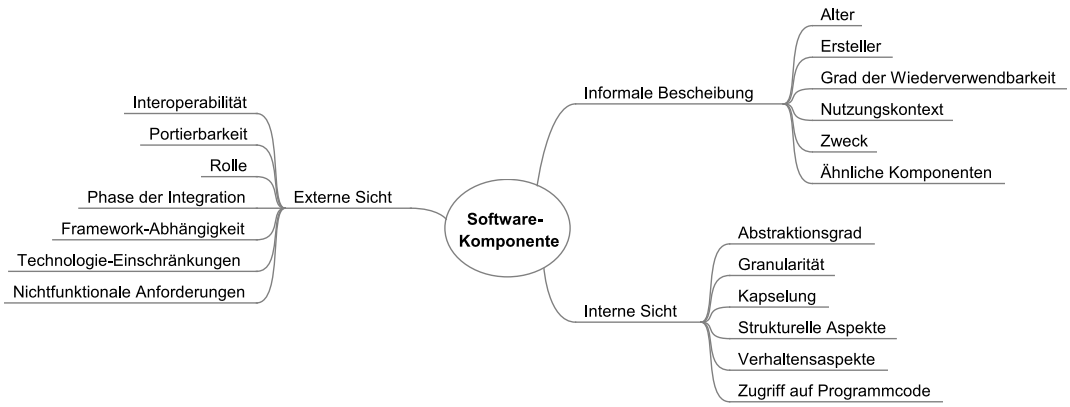

Abbildung 3.4.: Schema zur Klassifizierung von Software-Komponenten, nach [YAM99].

3.5.2. Software-Bausteine

Starke und Hruschka [SH09] beschreiben Software-Bausteine über ein Blackbox-Template. Ein Software-Baustein umfasst dabei Elemente unterschiedlicher Abstraktionsniveaus, wie

Bezeichnung	Bedeutung
Name	Wie heißt die Blackbox?
Zweck & Verantwortlichkeit	Welche Verantwortung übernimmt der Baustein in der Gesamtlösung?
Schnittstelle(n)	Welche Schnittstellen hat der Baustein?
Empfänger	Welcher Baustein empfängt die Ressource?
Ressource	Welche Ressourcen überträgt diese Schnittstelle?
Ablauf	Welche einzelnen Schritte sind zur Übertragung der Ressource nötig?
Fehlerszenarien	Welche Fehler können hier auftreten und wie werden sie behandelt?
Benutzungshinweise	Hinweise oder Beispiele zur Benutzung dieser Schnittstelle
Abhängigkeiten (optional)	Wovon ist dieser Bautein abhängig?
Erfüllte Anforderungen	Verweise zu den Anforderungen, die dieser Baustein erfüllt
Tests	Auf welche Weise wird der Baustein getestet?
Weitere Informationen	Beispielsweise Autor, Versions- und Änderungsinformationen

Tabelle 3.2.: Auszüge des Blackbox- und Schnittstellen-Templates nach [SH09].

Klassen, Pakete oder Komponenten. Die Blackbox-Sichtweise betrachtet einen Baustein von der Außensicht, wobei nichts über dessen interne Realisierung ausgesagt wird. Wichtige Punkte sind dabei der Zweck des Bausteins oder auch die Schnittstellen, d.h. wie der Baustein mit anderen Bausteinen interagiert. Tabelle 3.2 zeigt auf der ersten Ebene einige Eigenschaften des Blackbox-Templates. Die Beschreibung der Schnittstellen erfolgt in der zweiten Ebene über das Schnittstellen-Template. Für eine vollständige Beschreibung aller Eigenschaften wird auf [SH09] verwiesen.

Das Blackbox- und Schnittstellen-Template und das Klassifikationsschema für Software-Komponenten (siehe Abb. 3.4) zeigen einige Überschneidungen. Im Vergleich zu Software-Komponenten, befinden sich Module jedoch eine Abstraktionsebene tiefer, weshalb z. B. der *Abstraktionsgrad*, der unterscheidet, ob es sich bei der Komponente um ein Entwurfsmuster, eine Spezifikation oder eine ausführbare Anwendung handelt, weniger für die Beschreibung von Modulen geeignet ist. Dennoch sind einige der Merkmale, wie der *Nutzungskontext*, der *Zweck* oder auch die *nichtfunktionalen Anforderungen* für die Dokumentation von Modulen sehr hilfreich.

3.6. Fazit

Die in Abschnitt 3.2 betrachteten Werkzeuge zur integrierten Codedokumentation können nicht als Literate Programming Systeme eingestuft werden. Die Dokumentationsstruktur, die sich damit erstellen lässt, ist abhängig von der Struktur des Programmcodes. Die Programmiersprache Java erlaubt es beispielsweise, Methoden einer Klasse beliebig anzuordnen und damit auch deren Dokumentation nach eigenen Bedürfnissen zu strukturieren. Auf höherer Ebene folgt die Dokumentation jedoch der Klassenstruktur. Es ist somit keine psychologische Anordnung, wie es im Literate Programming gefordert wird, möglich [AD05].

Ein literate Programm, wie es von Knuth vorgeschlagen wurde, ist zum Lesen am Bildschirm schlecht geeignet. Durch den Einsatz der Formatierungssprache TEX wird der Inhalt der Dokumentation mit zusätzlichen Befehlen für die Formatierung aufgebläht, wodurch der Leser abgelenkt wird. Zum Lesen der Dokumentation muss daher zunächst ein TEX-System das literate Programm verarbeiten, um daraus eine gut lesbare Dokumentation zu erzeugen. Diese Dokumentation ist hilfreich, um eine stabile API einer Programmkomponente zu dokumentieren. Bei einem Programm, das häufigen Änderungen unterworfen ist, ist es jedoch fraglich, ob der Entwickler nach den Codeänderungen auch die Dokumentation mit aktualisiert und sich mit einer zusätzlichen Sprache wie TEX auseinandersetzt.

Trotz beschriebener Schwächen liefert Literate Programming wichtige Konzepte für eine bessere Dokumentation von Software. So wird durch die stärkere Fokussierung auf die Dokumentation der Entwickler dazu aufgefordert, seinen Programmcode zu dokumentieren. Durch die Unterteilung des Programms in mehrere Abschnitte mit unterschiedlichen Abstraktionsniveaus wird es Entwicklern erleichtert, fremden Programmcode schnell zu verstehen.

Im folgenden Kapitel soll nun untersucht werden, wie ein Werkzeug beschaffen sein sollte, um den Dokumentationsprozess zu unterstützen.

Kapitel 4

Usability-Aspekte

Moderne integrierte Entwicklungsumgebungen (IDEs) bieten eine Fülle von Funktionen an, um den Softwareentwicklungsprozess zu unterstützen. Diese vielfältige Unterstützung kann jedoch auch schnell zum Hindernis werden. So schreibt Raskin in [Ras03], dass durch die hohe Komplexität von IDEs viel Zeit verschwendet wird, um die Besonderheiten einer speziellen Entwicklungsumgebung zu verstehen. Der Entwickler kann sich dadurch weniger auf das eigentliche Problem konzentrieren.

Werden dagegen bessere Mensch-Maschine Schnittstellen verwendet, so führt dies zu höherer Produktivität, geringerer Einarbeitungszeit und weniger Belastung des menschlichen Gedächtnisses [Ras03]. Welche Faktoren sind jedoch für eine bessere Benutzbarkeit von Werkzeugen für die Softwareentwicklung entscheidend?

4.1. Definition

Bevor auf die Usability-Aspekte für ein Werkzeug zur Softwareentwicklung eingegangen wird, soll zunächst der Begriff *Usability* geklärt werden. Der Standard ISO 9241-11 schlägt dazu folgende Definition vor:

> ***Usability*** *ist das Ausmaß, in dem ein Produkt durch bestimmte Benutzer in einem bestimmten Nutzungskontext genutzt werden kann, um bestimmte Ziele effektiv, effizient und zufriedenstellend zu erreichen.*
>
> ISO 9241-11

Im deutschen Sprachgebrauch lässt sich Usability am ehesten mit *Gebrauchstauglichkeit* übersetzen.

Um die Usability eines Produkts zu bestimmen, sind folgende Informationen nötig [iso98]:

- Beschreibung der beabsichtigten Ziele des Produkts.
- Beschreibung der Benutzer. Diese kann Charakteristiken wie Erfahrung, Ausbildung oder motorische und sensorische Fähigkeiten der Benutzer beinhalten.

- Beschreibung der Aufgaben. Diese geben die Aktivitäten an, um ein bestimmtes Ziel zu erreichen.
- Beschreibung der Ausstattung und Umgebung bei Durchführung der Usability-Bestimmung.
- Bestimmung von Messgrößen, die sich auf Effektivität, Effizienz und Zufriedenheit beziehen (z. B. Genauigkeit, benötigte Zeit oder Auswertung von Fragebögen).

Wichtig ist dabei eine ausreichend genaue Beschreibung des Kontextes. Da durch das Verändern der Benutzergruppen oder der Umgebung die Usability-Messwerte signifikant verändert werden können.

4.2. Design-Richtlinien

Die Norm ISO-9241-110 [iso06] liefert Grundsätze für die ergonomische Gestaltung von interaktiven Systemen. Darin sind Richtlinien enthalten, die die Analyse, den Entwurf und die Bewertung von Benutzungsschnittstellen unterstützen. Diese sollen den Endbenutzer des Systems vor typischen Nutzungsproblemen, wie irreführende Informationen oder eine ineffiziente Behebung von Fehlern, schützen. Durch das Anwenden der Norm in frühen Phasen der Software-Entwicklung können gebrauchstaugliche und konsistente Benutzungsschnittstellen entwickelt werden, die zu einer hohen Produktivität führen.

Die Norm definiert dazu die folgenden sieben Grundsätze:

- **Aufgabenangemessenheit** liegt vor, wenn das System den Benutzer bei seiner durchzuführenden Aufgabe unterstützt. Die Gestaltung des Systems sollte auf den charakteristischen Eigenschaften der Arbeitsaufgabe basieren und nicht auf den Besonderheiten der verwendeten Technologie. Damit verbunden ist auch, dass keine Informationen angezeigt werden, die für die aktuelle Aufgabe nicht relevant sind.
- **Selbstbeschreibungsfähigkeit** zeigt dem Benutzer zu jeder Zeit an, welche Handlungen unternommen werden können und wie diese auszuführen sind.
- **Erwartungskonformität** ist vorhanden, sofern das System aus dem aktuellen Nutzungskontext heraus den Erwartungen des Benutzers und allgemein anerkannten Konventionen entspricht. Darunter fällt z. B., dass Formate für Ein- und Ausgaben den kulturellen und sprachlichen Gewohnheiten angepasst sind.
- **Lernförderlichkeit** unterstützt den Benutzer beim Erlernen des Umgangs mit dem interaktiven System.
- **Steuerbarkeit** ermöglicht dem Benutzer die Richtung und Geschwindigkeit des Dialogablaufs zu beeinflussen, bis er sein Ziel erreicht hat.

- **Fehlertoleranz** liegt vor, wenn fehlerhafte Eingaben mit minimalem Aufwand durch den Benutzer korrigiert werden können. Das System sollte den Benutzer über den gemachten Fehler informieren und auf eine mögliche Korrektur hinweisen, wie z. B. welches Format erwartet wird.
- **Individualisierbarkeit** ist vorhanden, wenn der Benutzer die Darstellung der Informationen beeinflussen kann, um diese an seine individuellen Fähigkeiten anzupassen.

Diese Grundsätze sind noch sehr allgemein gehalten und müssen für jedes System erneut überprüft werden, wie sie im Konkreten umzusetzen sind. Jedoch schaffen sie eine gute Grundlage, um die Usability eines Software-Systems zu verbessern. Weitere Richtlinien zielen auf ähnliche Aspekte ab, wie z. B. Shneiderman's *Eight Golden Rules of Dialog Design* [Shn98] oder weit allgemeiner die Gestaltgesetze von Wertheimer [Wer23].

4.3. Kognitive Aspekte beim Programmverstehen

In der Literatur finden sich zahlreiche Modelle, die versuchen zu erklären, wie Programmierer einen gegebenen Programmcode verstehen [MV95]. Diesen Modellen gemein ist, dass der Programmierer sein vorhandenes Wissen nutzt, um daraus Erkenntnisse aus dem vorliegenden Code zu gewinnen. Daraus erstellt er ein mentales Modell, wobei Hypothesen über die Software aufgestellt, überprüft oder auch wieder verworfen werden. Das mentale Modell stellt eine interne Repräsentation des betrachteten Programms dar.

Programmierer weisen dabei zwei Arten von Wissen auf: *Allgemeines Wissen* und *softwarespezifisches Wissen*. Software-spezifisches Wissen beschreibt den Grad des Verständnis über den betrachteten Programmcode. Dieses nimmt im Laufe des Verstehensprozesses zu. Das allgemeine Wissen unterstützt den Verstehensprozess und umfasst Elemente wie Kenntnisse über die verwendete Programmiersprache, allgemeine Programmierprinzipien oder Wissen über Algorithmen.

Das Verstehen von gegebenem Programmcode umfasst die meisten Aktivitäten beim Arbeiten an Software. Offensichtlich erforderlich ist dies während der Wartung. Jedoch erfordern auch Aktivitäten während der Entwicklung das Lesen von Programcode, wie z. B. Code-Reviews, Debugging oder das Erstellen von Testfällen. Der Verstehensprozess kann dabei entweder *top-down* oder *bottom-up* stattfinden [MV93, MV95]. Im Folgenden soll dazu jeweils ein Modell vorgestellt werden. Anschließend wird überprüft, ob sich daraus Anforderungen für ein Werkzeug zur Unterstützung dieses Prozesses ergeben.

4.3.1. Top-down Ansatz nach Brooks

Brooks beschreibt in [Bro83] eine Theorie, in welcher sich ein Programmierer ein mentales Modell nach einem top-down Ansatz erstellt. Grundlage der Theorie ist, dass während des Programmierens Wissensbereiche unterschiedlicher Ebenen durchlaufen werden. Dies beginnt auf der Ebene des Anwendungsbereichs und führt über mehrere Zwischenebenen

bis auf den Wissensbereich der verwendeten Programmiersprache. Als Beispiel kann ein Routenplanungsproblem aus der Logistik betrachtet werden. Dabei sind die Objekte des Anwendungsbereichs die Transportgüter, welche ein bestimmtes Lieferziel besitzen und damit verbundene Kosten aufweisen. Ein Programm, das eine mögliche Route berechnet, erfordert auf einer Zwischenebene mathematisches Wissen, wie z. B. Wissen über Matrizeninvertierung, während auf der untersten Ebene Wissen über die konkrete Programmiersprache erforderlich ist.

Der Verstehensprozess zu einem gegebenen Programmcode stellt nun die Rekonstruktion und Verbindung dieser Wissensbereiche dar. Dabei stellt der Programmierer Hypothesen über den fremden Programmcode auf, die im Laufe des Verstehensprozesses hierarchisch verfeinert werden. Ausgangspunkt ist eine primäre Hypothese, welche zunächst noch recht vage und allgemein ist. Der Programmierer stellt diese bereits auf, sobald er den Namen oder eine kurze Beschreibung des Programms erfährt. Diese erste Hypothese wird nun top-down, nach dem Tiefensuche-Prinzip, anhand entsprechender Stellen im Code oder in der Dokumentation weiter ausgebaut. Dieser Prozess wird fortgesetzt, bis das Verständnis über den Code ausreicht, um z. B. eine Änderung darin durchzuführen.

4.3.2. Bottom-up Ansatz nach Pennington

Pennington zeigt in [Pen87], dass der Verstehensprozess von Programmcode bottom-up verläuft. Der Programmierer beginnt mit dem Lesen der Codezeilen des Programms und gewinnt dabei ein immer besseres Verständnis über dessen Funktionalität und Struktur. Pennington teilt das mentale Modell in zwei aufeinanderfolgende Stufen ein: Das *Program Model* und das *Situation Model* [MV95].

Program Model – Ist der Programmcode komplett neu für einen Programmierer, so erstellt dieser als erste mentale Repräsentation eine Kontrollfluss-Abstraktion des Codes. Dieses Program Model wird über elementare Codestellen (engl. *Beacons*) bottom-up erstellt. Sogenannte Beacons beschreiben dabei die Ausdrucksfähigkeit des Codes, z. B. wie stark eine Notation oder Konvention den Programmierer unterstützt, die Bedeutung des Codes zu erfassen [CSW02]. Um das Program Model zu erstellen, muss der Programmierer somit Wissen über die Syntax der verwendeten Programmiersprache besitzen.

Situation Model – Sobald eine Repräsentation im Program Model vorliegt, entwickelt der Programmierer auf dieser Grundlage in der nächsten Stufe das Situation Model. Dieses wird ebenfalls bottom-up erstellt, stellt jedoch eine Abstraktion des Datenflusses und der Funktionalität dar. Dafür ist Wissen aus dem Anwendungsbereich erforderlich, mit welchem der Code hierarchisch in Objekte aus der realen Welt untergliedert wird.

Eine Dokumentation auf Modulebene lässt sich meiner Meinung nach am ehesten dem Situation Model zuordnen. Hierbei ist es förderlich, wenn die Klassen innerhalb eines Moduls einen hohen Zusammenhalt aufweisen. Dadurch kann die Dokumentation eines Moduls eine abgeschlossene Funktionalität beschreiben, die durch die enthaltenen Klassen

bereitgestellt wird. Durch eine hierarchische Organisation der Module nimmt der Grad der Abstraktion auf höheren Ebenen zu, wodurch sich Objekte des Anwendungsbereichs beschreiben lassen. Die Informationen werden somit stets auf der gedanklichen Ebene des Programmierers festgehalten. Auf der untersten Ebene kann die Bildung des Program Models durch Kommentare direkt im Programmcode unterstützt werden.

Crosby et al. [CSW02] zeigen, dass der Verstehensprozess stark von der Erfahrung des Programmierers abhängt. So machen Anfänger weniger Unterscheidungen zwischen den einzelnen Anweisungen, während Experten komplexe Codestellen als Beacons identifizieren und den Code dadurch schneller wahrnehmen können.

Mayrhauser und Vans [MV93] stellten in ihren Untersuchungen fest, dass der Verständnisprozess eine Kombination aus top-down und bottom-up Aktivitäten erfordert. Ist die Codestruktur oder der Anwendungsbereich bekannt, so kann theoretisch der Code vollständig top-down erfasst werden. Dagegen findet bei komplett unbekanntem Code das bottom-up Modell nach Pennington Anwendung. In der Praxis stellten Mayrhauser und Vans fest, dass Wartungsingenieure bei ihrer Arbeit häufig zwischen beiden Ansätzen wechseln. Das Verständnis wird dabei unterstützt, sofern die Programmierer Wissen über den Anwendungsbereich und die verwendete Programmiersprache aufweisen.

4.3.3. Abgeleitete Anforderungen an ein Werkzeug

Die betrachteten Modelle geben zunächst keine konkrete Hinweise darauf, wie ein Werkzeug beschaffen sein sollte, um den Verstehensprozess sinnvoll zu unterstützen. Zusätzlich kritisieren Singer et al. [SLVA10], dass Studien zur Verifikation dieser Modelle oft Studenten heranzogen und Programme mit einer nur sehr geringen Anzahl an Codezeilen verwendeten. Dabei ist unklar, wie sich diese Aussagen auf die industrielle Software-Entwicklung anwenden lassen.

In [MV93] und [SLVA10] wurden daher Untersuchungen in der Industrie durchgeführt, die die Aktivitäten der Programmierer beobachteten. Darin wurden die folgenden, teilweise auch recht allgemein gehaltenen Anforderungen an ein Werkzeug abgeleitet:

- Beim Verstehensprozess von umfangreichem Programmcode spielen kognitive Einschränkungen eine große Rolle. Das Werkzeug sollte daher die Belastungen des Kurzzeitgedächtnisses reduzieren, da dies nur eine geringe Kapazität aufweist.
- Der Programmierer sollte über das Werkzeug schnell und prägnant Antworten auf seine Fragen erhalten. Informationen sollten dabei auf der jeweiligen gedanklichen Ebene des Programmierers präsentiert werden, also z. B. auf der Ebene des Program- und Situation Models (siehe Abschnitt 4.3.2).
- Das Werkzeug sollte in der Lage sein, mit sehr großen Software-Systemen, die aus mehreren Millionen Zeilen Code bestehen, umgehen zu können. Dadurch kann der Entwickler das Werkzeug mit jedem beliebigen Software-System einsetzen.

- Das Werkzeug sollte Anfragen ohne wahrnehmbare Verzögerungen ausführen. Falls der Entwickler längere Zeit auf das Ergebnis einer Anfrage warten muss, unterbricht dies seinen aktuellen Gedankengang und erfordert möglicherweise das Wiederholen von vorangegangenen Tätigkeiten.
- Um die Akzeptanz eines Werkzeugs bei Programmierern zu erhöhen, ist es wichtig, dass sich das Werkzeug mit bestehenden Lösungen vereinen lässt. In einer Umfrage [SLVA10] kritisierten Entwickler, dass Werkzeuge häufig nicht in einer einheitlichen Umgebung integriert oder untereinander inkompatibel sind.

4.4. Usability existierender Werkzeuge

Basierend auf den Usability-Grundlagen, wie sie in diesem Kapitel vorgestellt wurden, sollen nun die betrachteten Werkzeuge zur Codedokumentation aus Kapitel 3.3 analysiert werden. Dazu werden Auffälligkeiten in Bezug auf die Usability herausgearbeitet.

Mit diesen Werkzeugen lassen sich allgemein zwei mögliche Ziele verfolgen. Ein Benutzer kann damit entweder die Dokumentation erstellen und anpassen oder Informationen in der erzeugten Dokumentation nachlesen. Dabei nehmen die Werkzeuge maßgeblich Einfluss darauf, wie der Benutzer die Kommentare zu erstellen hat, damit diese von einem Werkzeug verarbeitet werden können. Zudem ist meist die Struktur der erzeugten Dokumentation durch das Werkzeug vorgegeben. Diese lässt sich jedoch in vielen Fällen anpassen.

Als Benutzer dieser Werkzeuge werden Software-Entwickler angenommen, welche bereits Erfahrungen im Programmieren aufweisen.

XSDoc Wiki – Bedingt dadurch, dass dieses Werkzeug nicht mehr verfügbar ist, lassen sich nur begrenzte Aussagen in Bezug auf die Usability machen. Diese werden aus [AD05] abgeleitet. Bei XSDoc Wiki ist die Dokumentation nicht an die Strukturierung des Codes gebunden. Dadurch wird ermöglicht, die Dokumentation nach psychologischen Aspekten anzuordnen, um so den Leser zu unterstützen. Die Individualisierbarkeit kann daher als hoch angesehen werden, da der Benutzer selbst für die Strukturierung der Dokumentation verantwortlich ist. Durch das verwendete Wiki-System ist die Ansicht zum Lesen und Bearbeiten der Dokumentation auf sehr ähnliche Weise aufgebaut. Als WYSIWYG-Stil kann dies jedoch nicht betrachtet werden.

Doxygen – Doxygen extrahiert die Dokumentation von Programmcode aus zahlreichen Programmiersprachen. Dabei werden Kommentarblöcke nach den Standards der jeweils verwendeten Sprache erkannt. Der Benutzer kann somit die Kommentare entsprechend seinen Erwartungen formatieren.
Doxygen lässt sich sehr umfangreich konfigurieren. Die wichtigsten Einstellungen können dabei über einen Assistenten vorgenommen werden, wodurch sich die Ausgabe individuell anpassen lässt. In der erzeugten Ausgabe lassen sich die Elemente nach unterschiedlichen Kriterien gruppieren, sodass der Leser der Dokumentation nur die aktuell relevanten Informationen, entsprechend seiner Aufgabe, angezeigt bekommt.

Javadoc – Javadoc wird beispielsweise von der Entwicklungsumgebung Eclipse unterstützt. Dabei werden Javadoc-Kommentare farblich hervorgehoben und mögliche Schlüsselwörter werden über eine direkte Hilfe während der Eingabe vorgeschlagen. In Verbindung mit Eclipse kann daher die Selbstbeschreibungsfähigkeit von Javadoc höher als bei vergleichbaren Werkzeugen eingestuft werden.

AdaBrowse – Dieses Werkzeug ist sehr umfangreich konfigurierbar. Schlüsselwörter und Formatierungsregeln müssen hier jedoch manuell in Konfigurationsdateien definiert werden. Dies erfordert einen zusätzlichen Einarbeitungsaufwand für den Entwickler.

Document! X – In der grafischen Benutzungsoberfläche zeigt Document! X eine Dokumentationsvorlage an, die die Dokumentation in mehrere Abschnitte gliedert. Der Benutzer wird dadurch angeleitet, welche Elemente dokumentiert werden sollten. Daher kann die Selbstbeschreibungsfähigkeit beim Erstellen der Dokumentation als hoch angesehen werden. Zugleich wird dadurch eine einheitliche Dokumentation gewährleistet.
Da die Dokumentation über einen WYSIWYG-Stil bearbeitet wird, ist die erzeugte Ausgabe dazu gleich aufgebaut. Der Benutzer sieht somit eine vertraute Struktur und kann sich schneller darin zurechtfinden. Zusätzlich bietet die Ausgabe eine Suchfunktion, damit schnell Informationen zur aktuellen Arbeitsaufgabe gefunden werden können.

Doc-O-Matic – Die Erstellung der Dokumentation über einen WYSIWYG-Editor bietet mehr Komfort als die direkte Bearbeitung in Code-Dateien. In Doc-O-Matic kann der Benutzer den Inhalt der Kommentarblöcke individuell gestalten, jedoch wird er dabei nicht angeleitet, die Dokumentation mit speziellen Feldern zu versehen.
Die erzeugte Dokumentation orientiert sich dabei an gängigen Vorlagen, wie z. B. Stile von Visual Studio- und MSDN-Hilfe (*Microsoft Developer Network*). Ein Benutzer findet somit eine bekannte Dokumentationsstruktur vor.

Haddoc – Um die Dokumentation mit Haddoc zu formatieren werden bestimmte Sonderzeichen verwendet. Diese muss der Entwickler zunächst erlernen, wodurch auch hier zusätzlicher Einarbeitungsaufwand erforderlich ist.

Allgemein lässt sich festhalten, dass die betrachteten Werkzeuge, bis auf wenige Ausnahmen, eine nur geringe Selbstbeschreibungsfähigkeit aufweisen. Bei Werkzeugen ohne grafischen Editor müssen die Kommentare nach speziellen Regeln formatiert werden, damit diese erkannt werden. Zudem müssen die Schlüsselwörter, die innerhalb den Kommentaren anwendbar sind, dem Entwickler bekannt sein. Die Werkzeuge erlauben es, die Struktur der erzeugten Dokumentation nach individuellen Bedürfnissen anzupassen. Jedoch muss dies über die Modifikation von gegebenen Vorlagen bewerkstelligt werden, wofür häufig Fachwissen erforderlich ist, wie z. B. HTML-Kenntnisse. Des Weiteren findet eine Trennung zwischen den Ansichten zum Lesen und zum Bearbeiten der Dokumentation statt. Wird die Dokumentation in HTML-Seiten exportiert, so verwendet der Entwickler in der Regel einen Browser, um diese zu lesen. Dagegen müssen für das Aktualisieren der Dokumentation direkt Code-Dateien bearbeitet werden.

Kapitel 5

Anforderungen

Ziel dieser Arbeit ist die Entwicklung von J-PaD (*Java Package Documentation*), mit dem sich Software-Module, im konkreten Java-Pakete, dokumentieren lassen. Damit soll ein Entwickler einen detaillierten Überblick über ein dokumentiertes Modul gewinnen können.

5.1. Nichtfunktionale Anforderungen

Eine Analyse zu Beginn des Projekts ergab die folgenden nichtfunktionalen Anforderungen an das Werkzeug:

- Es soll als Plugin in die Entwicklungsumgebung Eclipse integriert sein und in der Programmiersprache Java realisiert werden.
- Die Daten der Dokumentation sollen mithilfe von tabellarischen Formularen angezeigt und bearbeitet werden können.
- Die gespeicherten Daten in der Projektstruktur dürfen den Programmcode nicht von dem Werkzeug abhängig machen. Ein Projekt, welches mit diesem Werkzeug dokumentiert wurde, muss sich übersetzen und ausführen lassen, auch wenn das Werkzeug nicht installiert ist oder das Projekt in einer anderen Entwicklungsumgebung bearbeitet wird.
- Die Software soll eine hohe Wartbarkeit aufweisen und sich einfach erweitern lassen.
- Der Programmcode von J-PaD soll gut dokumentiert werden und in Form einer Selbstanwendung Dokumentationen auf Modulebene aufweisen.
- Eine gute Usability der Software wird angestrebt, um eine intuitive Bedienung zu ermöglichen.

5.2. Funktionale Anforderungen

Die Hauptfunktionalität des zu entwickelnden Werkzeugs soll darin bestehen, Java-Pakete eines Projekts über eine grafische Benutzungsoberfläche zu beschreiben. Dabei soll mindestens die folgende Funktionalität bereitgestellt werden:

- Eine Übersichtsseite des Pakets, auf der sich eine Beschreibung der Funktionalität und des Zwecks des Pakets angeben lässt.
- Eine Anbindung an JUnit. Es soll eine Zuordnung von Testfällen zu entsprechenden Paketen dargestellt werden können.
- Es soll möglich sein, die zu dokumentierenden Eigenschaften eines Pakets projektspezifisch anzupassen. Dafür soll pro Projekt eine Datei angelegt werden, die die Konfiguration der Datenfelder vorgibt.
- Die Daten für die Konfiguration und für die Beschreibung der Pakete sollen innerhalb der Projektstruktur in einem Textformat (wie z. B. XML) gespeichert werden. Das Format ist dabei so zu wählen, dass Versionsverwaltungssysteme wie Subversion sinnvoll eingesetzt werden können.

5.3. Optionale Anforderungen

Als optionale Anforderung wurde eine grafische Benutzungsoberfläche für die Konfiguration genannt. Darin sollen die möglichen Datenfelder der Paketdokumentation in einem grafischen Editor organisiert und die dafür notwendigen Einstellungen vorgenommen werden können.

5.4. Zielgruppe

Zu der Zielgruppe von J-PaD zählen in erster Linie die Entwickler von Java-Anwendungen. Es kann also allgemein von Nutzern ausgegangen werden, die bereits erste Erfahrungen im Programmieren und im Umgang mit der Entwicklungsumgebung Eclipse aufweisen. Neben dieser Zielgruppe lassen sich noch weitere Sichtweisen auf das Werkzeug identifizieren. Abbildung 5.1 zeigt die Benutzerrollen von J-PaD im Überblick:

Dokumentation – Die Zielgruppe, die das Werkzeug zur Dokumentation von Java-Paketen einsetzt, stellt die Hauptanwendergruppe dar. Dabei kann das Werkzeug von Benutzern in unterschiedlichen Positionen eingesetzt werden. Zu Beginn der Software-Entwicklung kann der *Software-Architekt* die Paketstruktur anlegen und eine Spezifikation pro Paket durch dieses Werkzeug vorgeben. Die *Programmierer* ziehen diese Modulspezifikation heran, um daraus den Programmcode zu erstellen. Zudem können diese ihre Implementierung auf Paketebene damit dokumentieren. Anschließend

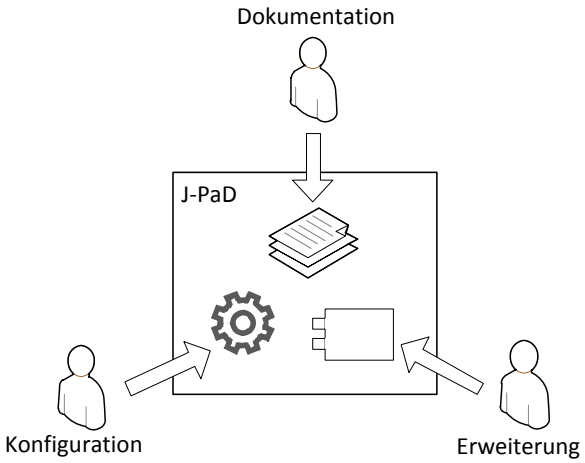

Abbildung 5.1.: Benutzerrollen von J-PaD

können *Testingenieure* aus dieser Dokumentation die Sollergebnisse für Testfälle ableiten. Zusätzlich hilft die Paketdokumentation den *Wartungsingenieuren* während der Wartung, um sich einen detaillierten Überblick über dokumentierte Pakete zu verschaffen.

Konfiguration – Die Konfiguration gibt ein einheitliches Schema für ein Projekt vor, das festlegt, welche Eigenschaften über ein Paket dokumentiert werden sollen. Diese Rolle sollte von einer zentralen Stelle übernommen werden, wie z. B. dem *Software-Architekten*, dem *Projektleiter* oder der Abteilung zur *Qualitätssicherung*.

Erweiterung – Als dritte Gruppe können im weitesten Sinn die Entwickler angesehen werden, die Erweiterungen für das Werkzeug erstellen. Dies wird im Allgemeinen durch eine *Open-Source-Community* bewerkstelligt.

Kapitel 6

Konzept

Basierend auf den Anforderungen aus Kapitel 5 und den vorangegangenen Kapiteln, soll nun im Folgenden daraus ein Konzept erarbeitet werden. Dazu wird beschrieben, wie die Anforderungen an das Werkzeug J-PaD (*Java Package Documentation*) umgesetzt werden.

6.1. Javadoc Paketdokumentation

Wie bereits in Abschnitt 3.2 vorgestellt, erlaubt es Javadoc Java-Pakete gesondert zu dokumentieren. Dafür wird eine separate Datei angelegt und im entsprechenden Paket platziert. In der Pakethierarchie kann damit für jedes Paket eine eigene Dokumentation erstellt werden.

Um eine Paketdokumentation zu erstellen, werden von Javadoc prinzipiell zwei Ansätze unterstützt. Die Dokumentation kann entweder in einer Datei mit dem Namen *package.html* oder *package-info.java* platziert werden. Beiden Ansätzen gemein ist, dass diese direkt im dokumentierten Paket gespeichert werden und so eine Nähe zur Codierung des Pakets aufweisen. Zudem wird bei beiden Ansätzen, beim Exportieren der Dokumentation mit Javadoc, die gleiche Art von Übersichtsseite erstellt. Bei Verwendung der *package.html* Datei wird die Dokumentation in HTML-Elemente eingeschlossen. Diese kann aus einer Beschreibung und einer Menge an Javadoc Tags bestehen. Seit JDK 5.0 (*Java Development Kit*) ist es jedoch auch möglich, die Paketdokumentation in eine Java-Datei zu schreiben. Die Verwendung dieser Datei wird seitdem empfohlen und daher auch als Grundlage bei der Entwicklung von J-PaD verwendet. J-PaD gewährleistet dadurch eine Kompatibilität zu Javadoc, wodurch die Einstiegshürde für das Werkzeug verringert wird (siehe Abschnitt 4.3.3).

6.1.1. Aufbau der Datei package-info.java

Die Datei *package-info.java* besteht im Wesentlichen aus Paket-Annotationen, Paket-Kommentaren, Javadoc Tags und einer Paket-Deklaration. Es kann keinen Code für eine Klasse `package-info` enthalten, da dies kein gültiger Klassenbezeichner ist. Technisch ist es

möglich, darin paketinterne Klassen zu implementieren. Dies sollte jedoch bei einem guten Programmierstil vermieden werden.

Genauer betrachtet stellt dieses Konzept eine Mischform aus integrierter und separater Dokumentation dar (siehe Abschnitt 2.1.1). Die Dokumentation wird in einer separaten Datei erstellt, welche keinen ausführbaren Code enthält. Gleichzeitig befindet sich die Datei jedoch in direkter Nähe zum Code und wird auch zusammen mit diesem der Versionsverwaltung unterstellt. Da außerdem diese Datei auf die gleiche Art und Weise wie der Code bearbeitet wird, wird sie in dieser Arbeit der integrierten Dokumentation zugeordnet.

```
/**
 * Provides the classes necessary to create an applet and the classes an applet uses
 * to communicate with its applet context.
 * <p>
 * The applet framework involves two entities:
 * the applet and the applet context. An applet is an embeddable window (see the
 * {@link java.awt.Panel} class) with a few extra methods that the applet context
 * can use to initialize, start, and stop the applet.
 *
 * @since 1.0
 * @see java.awt
 */
package java.applet;
```

Ausschnitt 6.1: Aufbau einer *package-info.java* Datei, nach [Orab].

Ausschnitt 6.1 zeigt ein Beispiel einer package-info.java Datei. Darin enthalten ist ein Dokumentationskommentar nach Javadoc-Konventionen, welcher direkt der Paket-Deklaration vorausgeht. Der Dokumentationskommentar besteht aus einer allgemeinen Beschreibung über das Paket und einem Abschnitt mit beliebig vielen Javadoc Tags. Die allgemeine Beschreibung startet direkt nach den einleitenden Zeichen des Kommentarblocks (/**) und endet mit dem Beginn des Tag-Abschnitts. Der erste Satz der Beschreibung stellt eine kurze Zusammenfassung des Pakets dar. Diese wird von Javadoc in einer Übersichtsseite verwendet und sollte daher möglichst prägnant und aussagekräftig ausfallen. Der führende Stern (*) zu Beginn jeder Zeile innerhalb des Kommentarblocks (ausgenommen die letzte Zeile) ist optional und dient lediglich der Formatierung des Blocks, er ist nicht Teil der Dokumentation. Nach der allgemeinen Beschreibung folgt ein Abschnitt mit Javadoc Tags.

Javadoc Tags

Bestimmte Eigenschaften eines Pakets lassen sich mit Javadoc über *Tags* beschreiben. Tags sind vordefinierte Schlüsselwörter und werden direkt in einem Dokumentationskommentar platziert. Sie dienen dazu, die Dokumentation besser zu strukturieren und zu formatieren. Javadoc unterscheidet zwei Arten von Tags: Block-Tags und Inline-Tags. Erstere können nur im Tag-Abschnitt jeweils am Zeilenanfang verwendet werden. Diese haben die Form *@tag*, gefolgt von einem Textabschnitt, welcher die entsprechende Eigenschaft beschreibt. Der Textabschnitt eines Block-Tags kann sich über mehrere Zeilen erstrecken. Er endet mit dem

Beginn eines neuen Block-Tags. Inline-Tags dagegen können im Beschreibungstext von Block-Tags und in der allgemeinen Beschreibung des Dokumentationskommentars verwendet werden. Diese sind in geschweiften Klammern eingeschlossen, wie z. B. *{@tag}*. Allgemein können Beschreibungstexte mit HTML-Elementen erweitert werden, um die Dokumentation weiter zu formatieren.

Jeder Javadoc Tag hat einen bestimmten Gültigkeitsbereich. So können unterschiedliche Tags z. B. auf Paket- oder Klassenebene verwendet werden. Im Folgenden werden nur die Tags, die auf Paketebene unterstützt werden, näher betrachtet. Für eine Beschreibung der weiteren Tags wird auf [Oraa] verwiesen. Block-Tags lassen sich teilweise mehrfach innerhalb eines Dokumentationskommentars verwenden. Die nachfolgenden Beschreibungen enthalten daher hinter jedem Tag eine Bereichsangabe in eckigen Klammern, welche die mögliche Häufigkeit angibt. Eine Verletzung dieser Angaben stellt jedoch keinen Fehler für Javadoc dar. Für den Fall, dass mehrere Tags in einem Kommentar enthalten sind, als Javadoc unterstützt, werden diese zusätzlichen Tags beim Export der Dokumentation ignoriert.

Folgende Tags werden von Javadoc auf Paketebene unterstützt:

@author *Beschreibungstext*, [0..*]
: Beschreibt die Autoren des Pakets. Werden mehrere @author-Tags angegeben, so konkateniert Javadoc beim Erzeugen der Dokumentation die Werte jedes Tags und trennt diese durch ein Komma voneinander ab. Dabei wird empfohlen, nur einen Namen pro Tag zu verwenden, um eine einheitliche Trennung zu erreichen. Ist nur ein @author-Tag spezifiziert, so wird dessen Wert ohne Weiterverarbeitung in die Dokumentation übernommen.

@deprecated *Beschreibungstext*, [0..1]
: Gibt an, dass die Codierung des Pakets veraltet ist und nicht mehr verwendet werden sollte. Der Beschreibungstext sollte dem Leser mitteilen, seit wann die Codierung veraltet ist und welche Alternative in Zukunft zu verwenden ist.

@see *Referenz*, [0..*]
: Ermöglicht der Dokumentation Referenzen hinzuzufügen. Als *Referenz* können dabei drei verschiedene Arten verwendet werden:

  - *@see "Beschreibungstext"* – Erzeugt einen Texteintrag in der generierten Dokumentation, ohne dass ein Link dafür erstellt wird. Dies kann z. B. verwendet werden, um ein Buch anzugeben, welches nicht über eine URL erreichbar ist.
  - *@see Beschriftung* – Erstellt aus der angegebenen *Beschriftung* einen Link. Die *URL* kann dabei absolut oder relativ sein.
  - *@see Paket.Klasse#Mitgliedsname Beschriftung* – Erlaubt das Einfügen von Verweisen auf Java-Elemente. Java-Elemente können dabei Pakete, Klassen, Interfaces, Methoden oder Felder sein. Die Beschriftung ist optional. Wird diese weggelassen, so wird der Name des referenzierten Java-Elements verwendet.

@serial `include | exclude`, [0..1]
Auf Paketebene ist nur einer der beiden Werte `include` oder `exclude` zulässig. Dieser gibt an, ob für die Klassen des Paketes, die das Interface `java.io.Serializable` erweitern, jeweils eine Übersichtsseite über serialisierbare Felder erstellt werden soll. Der @serial-Tag ist sowohl auf Paket- als auch auf Klassenebene anwendbar. Wird dieser auf beiden Ebenen verwendet, überschreibt der @serial-Tag auf der Klassenebene den entsprechenden Tag auf der Paketebene.

@since *Beschreibungstext*, [0..*]
Der Beschreibungstext gibt an, seit wann eine bestimmte Implementierung verfügbar ist. Werden mehrere @since-Tags verwendet, so werden diese analog wie mehrere @author-Tags behandelt. Mehrere Tags können eingesetzt werden, falls das Paket von mehr als einer API verwendet wird.

@version *Beschreibungstext*, [0..*]
Hält die aktuelle Version eines Pakets. Mehrere @version-Tags werden auf analoge Weise wie mehrere @author-Tags behandelt.

{@code *Text***}** Inline-Tag, welcher den angegebenen *Text* als Code formatiert. Der Inhalt wird dabei nicht als HTML interpretiert, wodurch sich direkt die Zeichen < und > z. B. in Vergleichen verwenden lassen. Alternativ lässt sich ein Text über die HTML-Elemente `<code>...</code>` einschließen, um dieselbe Formatierung zu erzielen. Enthaltene Spitzklammern müssen dabei jedoch, entsprechend der HTML-Konvention, umcodiert werden.

{@docRoot} Stellt den relativen Pfad zum Wurzelverzeichnis der generierten HTML-Dokumentation dar. Damit lassen sich z. B. Logos oder Copyright-Hinweise von jeder beliebigen Unterseite aus referenzieren.

{@link *Paket.Klasse#Mitgliedsname Beschriftung***}** Setzt einen Hyperlink auf die Dokumentation des angegebenen Java-Elements. Der Tag ist dabei sehr ähnlich wie der entsprechende @see-Tag, jedoch mit dem Unterschied, dass dieser innerhalb von Beschreibungstexten verwendet werden kann. Die Beschriftung wird als Code formatiert.

{@linkplain *Paket.Klasse#Mitgliedsname Beschriftung***}** Analog zu {@link}, jedoch wird hier die Beschriftung nicht als Code formatiert.

{@literal *Text***}** Der angegebene *Text* wird nicht als HTML oder geschachtelte Tags interpretiert.

6.1.2. Erweiterung von Javadoc

Wie oben beschrieben stellt Javadoc nur eine begrenzte Anzahl an Tags zur Verfügung, um bestimmte Eigenschaften eines Pakets zu dokumentieren. Daher muss eine Möglichkeit gefunden werden, um diese für die Dokumentation mit J-PaD zu erweitern. Wird die Paketdokumentation nur intern in der Entwicklungsumgebung verwendet, so könnten weitere Informationen einfach durch beliebige eigene Tags in die Dokumentation mit aufgenommen

werden. Jedoch meldet dann Javadoc beim Exportieren der Dokumentation für jeden unbekannten Tag eine Warnung. Eine Möglichkeit, dies zu verhindern, ist die Implementierung eines Taglets für jeden neuen Tag. Ein Taglet ist ein Java-Programm, das die Taglet API von Javadoc ausnutzt. Diese erlaubt es, eigene Tags zu definieren und eine bestimmte Formatierung dafür zu hinterlegen. Dieser Ansatz weißt sich jedoch als sehr unflexibel aus, da für jeden neuen Tag zuerst ein Taglet implementiert werden müsste.

In J-PaD wird daher ein Ansatz verwendet, wie er in Ausschnitt 6.2 angedeutet ist. Dabei wird zwischen Javadoc Tags und individuell erstellten Tags unterschieden. Die Javadoc Tags werden dabei in den Kopfkommentar der package-info.java Datei geschrieben, sodass Javadoc damit weiterhin verwendet werden kann. Individuell erstellte Tags werden dagegen in einen eigenen Kommentarblock unterhalb der Paket-Deklaration geschrieben (*Fußkommentar* in Ausschnitt 6.2). Javadoc ignoriert diesen, da es Dokumentationskommentare stets über einem entsprechenden Element – hier die Paket-Deklaration – erwartet. Dadurch lässt sich Javadoc ohne Einschränkungen verwenden und die Dokumentation kann sehr einfach erweitert werden.

```
/**
 * Javadoc Kopfkommentar
 */
package java.applet;

/**
 * Fußkommentar
 */
```

Ausschnitt 6.2: Erweiterung der *package-info.java* Datei.

Die Zuordnung eines Tags zu Kopf- oder Fußkommentar soll dabei flexibel gestaltet sein. Falls ein Benutzer von J-PaD die Taglet API bereits erweitert hat, so soll er diese Tags dem Kopfkommentar zuordnen können, sodass Javadoc diese beim Export berücksichtigen kann. Außerdem ist es möglich, dass durch zukünftige Javadoc Versionen weitere Tags unterstützt werden (siehe [Orac]).

6.1.3. Externe Dateien

Javadoc kann beim Erzeugen der separaten Dokumentation Inhalte aus vier verschiedenen Kategorien berücksichtigen. Darunter fallen die Dokumentationskommentare aus Java-Codedateien (**.java*) sowie die Inhalte aus Paketdokumentationsdateien (*package.html* oder *package-info.java*). Weiterhin können eine HTML-Übersichtsseite (z. B. *overview.html*) für eine projektweite Dokumentation und externe Dateien, deren Inhalt nicht von Javadoc verarbeitet wird, eingebunden werden. Diese externen Dateien können z. B. Grafiken, Beispiel-Code oder auch zusätzliche separate Dokumentation sein, welche von Javadoc beim Exportieren unverändert in das Ausgabeverzeichnis kopiert werden. Damit Javadoc diese Dateien berücksichtigt, müssen diese in einem Ordner mit dem Namen ***doc-files*** platziert werden. Dieser kann als Unterordner für jedes Paket erstellt werden, wodurch sich die externen Dateien nahe bei der Codierung eines Pakets befinden. Ausführbarer Code kann darin nicht

enthalten sein, da der Name des Ordners keinem gültigen Paketnamen entspricht. Innerhalb von Dokumentationskommentaren kann nun über eine relative Pfadangabe auf eine externe Datei verwiesen werden. Soll z. B. die Grafik *classDiagram.png* in die Dokumentation eingebunden werden, so lässt sich dies über die folgende HTML-Schreibweise erreichen:

```
<img src="doc-files/classDiagram.png">
```

Die durch Javadoc generierten HTML-Seiten zeigen daraufhin an entsprechender Stelle die Grafik an. Andere Dateitypen lassen sich z. B. über Hyperlinks referenzieren.

6.2. Umsetzung weiterer Anforderungen

Im Normalfall soll der Benutzer von J-PaD die Paketdokumentation nicht über Javadoc-Kommentare bearbeiten müssen. Dennoch soll diese Option für eine erweiterte Dokumentationsmöglichkeit offen gehalten werden. Die Dokumentation soll sich standardmäßig über eine grafische Oberfläche bearbeiten lassen, welche nach tabellarischen Formularen aufgebaut ist (siehe Abschnitt 5.1). Beide Stile werden dabei in einem Editor vereint, welcher diese über mehrere Tabs voneinander trennt.

Für die direkte Bearbeitung der Javadoc-Kommentare soll die gleiche Ansicht verwendet werden, wie sie auch bei der Bearbeitung von Java-Codedateien eingesetzt wird. Diese bietet neben Syntaxhervorhebungen auch eine direkte Hilfestellung über mögliche Schlüsselwörter während der Eingabe an. Der Dokumentationseditor und dessen grafische Oberfläche sollen nach folgenden Kriterien gestaltet werden:

- Der Dokumentationseditor soll sich standardmäßig öffnen, sobald der Benutzer eine `package-info.java` Datei aus der Projektstruktur auswählt. Die Erwartungskonformität (siehe Abschnitt 4.2) soll dadurch erhöht werden, dass der Dokumentationseditor sich am Design und Verhalten an gängigen grafischen Editoren von Eclipse orientiert.
- Die projektspezifische Konfiguration (siehe Abschnitt 5.2) ermöglicht eine Individualisierung der Dokumentation für unterschiedliche Java-Projekte. Oberflächenelemente können dabei dem Dokumentationseditor flexibel hinzugefügt oder darin umsortiert werden. Zudem können diese mit einem eigenen Vokabular versehen werden, welches typisch für den entsprechenden Geschäftsbereich ist.
- Um kognitiven Einschränkungen entgegenzuwirken, sollen die Informationen in der Paketdokumentation nicht in einer einzigen Ansicht dargestellt werden (siehe Abschnitt 4.3.3). Die grafische Oberfläche soll es daher ermöglichen, einzelne Oberflächenelemente zu Gruppen zusammenzufassen und unterschiedliche Elemente in Tabs zu organisieren.

- Die Konfiguration soll ermöglichen, einzelne Oberflächenelemente als Pflichtfelder zu spezifizieren. Schreibt der Benutzer keine Dokumentation zu einem entsprechend markierten Feld, so soll der Dokumentationseditor mit einer Warnung darauf hinweisen. J-PaD soll hierauf jedoch fehlertolerant reagieren und das Abspeichern dennoch ermöglichen. Streng genommen widerspricht dies der Semantik eines Pflichtfeldes. Mangels einer passenderen Bezeichnung wird in dieser Arbeit der Begriff jedoch weiterverwendet.
- Länger andauernde Operationen sollen im Hintergrund ausgeführt werden, damit die grafische Oberfläche weiterhin auf Benutzereingaben reagieren kann. So soll z. B. das Parsen der package-info.java- und der Konfigurations-Datei in einen eigenen Thread ausgelagert werden.

6.2.1. Grafische Oberflächenelemente

Die Hauptanwendung des Dokumentationseditors besteht in der natürlichsprachlichen Dokumentation von Java-Paketen. Dennoch sollen in der grafischen Oberfläche verschiedene Felder unterstützt werden, um unterschiedliche Inhaltstypen zu dokumentieren. J-PaD soll mindestens die folgenden Oberflächenelemente zur Verfügung stellen:

Text – Stellt ein einzeiliges Textfeld bereit, über welches sich kurze Informationen zu einem Paket festhalten lassen. Dies eignet sich beispielsweise für Versionsangaben oder den Beschreibungstext einer *@deprecated*-Markierung.

HTML – Für umfangreiche Beschreibungen wird ein HTML-Editor zur Verfügung gestellt, welcher eine Bearbeitung von HTML-Syntax im WYSIWYG-Stil erlaubt. Diese Form eignet sich in erster Linie für die allgemeine Beschreibung eines Pakets.

Listen – Sollen mehrere Werte für einen Tag angegeben werden, so kann dies über eine Liste erfolgen. Darüber können z. B. mehrere Autoren verwaltet werden, ohne dass diese zusammen in einem Textfeld bearbeitet werden müssen.

Datum – Stellt ein Datumsauswahlfeld bereit.

Ressourcen-Tabelle – Ermöglicht es, separate Dokumentation in die Dokumentation eines Pakets einzubinden. Dabei können die Ressourcen entweder referenziert oder direkt in den *doc-files* Ordner (siehe Abschnitt 6.1.3) eines Pakets importiert werden. Jede Ressource kann zusätzlich mit einer Beschreibung versehen werden.

Modul-Testfälle – Bietet eine Verwaltung der Testfälle zu den Klassen eines Pakets an. Dabei können neben JUnit-Testfällen auch beliebige Testmethoden eingetragen werden. Eine Implementierung der Testmethoden wird nicht vorausgesetzt, wodurch sich Tests bereits im Voraus spezifizieren lassen.

Kapitel 7

Ergebnis

Nachdem in Kapitel 6 das Konzept vorgestellt wurde, soll nun das Ergebnis – *J-PaD*, das im Rahmen dieser Diplomarbeit entwickelt wurde – präsentiert werden. Dazu wird zunächst auf die Installation des Werkzeugs eingegangen, während anschließend die Benutzung von J-PaD aus Anwendersicht beschrieben wird. Die Beschreibung orientiert sich dabei an den in Abschnitt 5.4 festgelegten Zielgruppen.

7.1. Installation

J-PaD ist auf der Webseite `http://www.j-pad.de` in der Version 0.1 veröffentlicht. Darauf werden allgemeine Informationen über das Werkzeug sowie die notwendigen Schritte zur Installation vorgestellt.

Um J-PaD zu verwenden, wird eine Eclipse Installation der Version 3.7.1 oder höher vorausgesetzt. Darin kann J-PaD, über den in Eclipse eingebauten Update-Mechanismus, installiert werden. Folgende Schritte sind dafür notwendig:

1. Im Menü von Eclipse »`Help`« → »`Install New Software...`« auswählen.
2. Im Dialog die Update-Seite `http://www.j-pad.de/update` eintragen.
3. *J-PaD* aus der Liste auswählen und dem Dialogverlauf folgen.
4. Nach Abschluss der Installation Eclipse neu starten.

7.2. Dokumentation

In diesem Abschnitt wird die Verwendung von J-PaD aus Sicht der Hauptanwendergruppe beschrieben. Diese setzt das Werkzeug zur Dokumentation von Java-Paketen ein.

Nachdem J-PaD in Eclipse installiert wurde und bereits erste Pakete in einem Projekt angelegt sind, kann mit der Dokumentation von Paketen begonnen werden. Dazu wählt der Benutzer

aus dem Kontextmenü eines Pakets den Eintrag »`Document Package`« aus. Diese Aktion veranlasst, dass die `package-info.java` Datei (siehe Abschnitt 6.1) im gewählten Paket angelegt und der Dokumentationseditor dafür geöffnet wird. Ist die `package-info.java` Datei bereits vorhanden, so kann der Dokumentationseditor direkt über einen Doppelklick auf diese Datei geöffnet werden.

Im Folgenden wird die grafische Oberfläche des Dokumentationseditors beschrieben. Diese ist nach der Konfiguration, die in J-PaD standardmäßig verwendet wird, aufgebaut (siehe dazu auch Abschnitt 3.5.2). Im Anhang A.1 ist die entsprechende XML-Konfigurationsdatei beigefügt. Beliebige andere Konfigurationen sind jedoch auch denkbar (siehe Abschnitt 7.3).

Paketübersicht

Wird die Dokumentation zu einem Paket geöffnet, so wird im ersten Tab eine Übersicht dazu angezeigt (siehe Abb. 7.1). Darin sind allgemeine Informationen über das Paket aus der Außensicht enthalten.

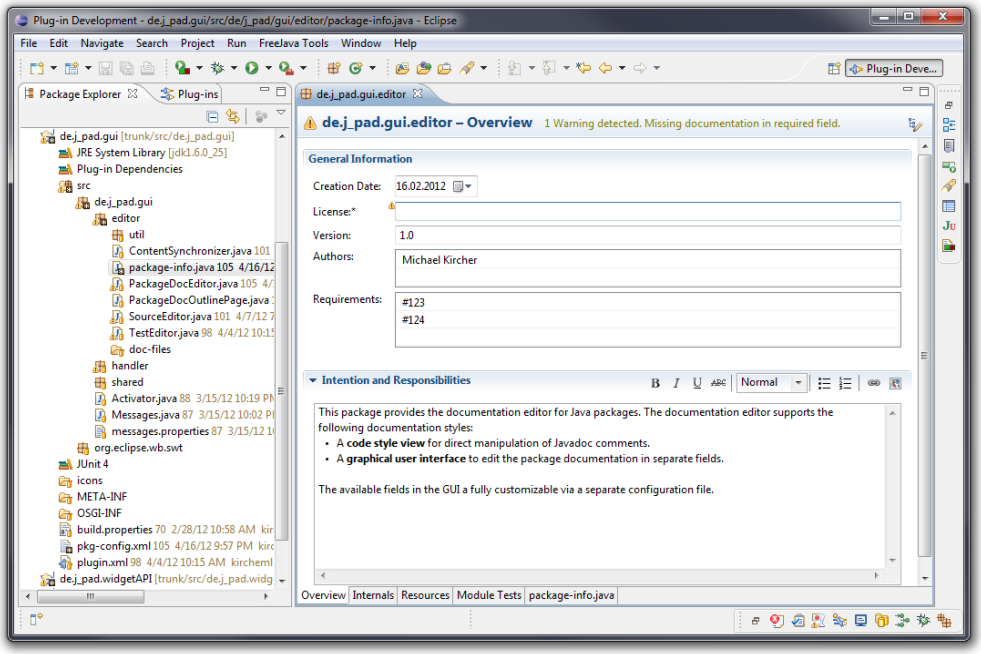

Abbildung 7.1.: Screenshot Paketübersicht – Allgemeine Informationen über ein Paket.

In Abbildung 7.1 werden Oberflächenelemente aus vier verschiedenen Typen verwendet (siehe Abschnitt 6.2.1): Für das *Erstelldatum* wird ein Datumsauswahlfeld benutzt. Dieses stellt über eine Dropdown-Box eine Kalenderansicht zur Verfügung, in welcher sich der Tag, der Monat und das Jahr auswählen lassen. Die Lizenz- und Versionsinformationen werden über einzeilige Textfelder eingegeben. In die Felder *Autoren* und *Verweise zu den*

Anforderungen werden in der Regel mehrere Werte eingetragen. Daher werden für diese Felder Listen eingesetzt, sodass in jede Zeile ein eigener Eintrag eingefügt werden kann. Die allgemeine Beschreibung des Pakets kann über einen HTML-Editor bearbeitet werden. Darin lässt sich der Text ansprechend strukturieren und formatieren. Diese Beschreibung sollte den Zweck und die Verantwortlichkeit des Pakets kurz und präzise darstellen [SH09].

Abbildung 7.1 zeigt zusätzlich die Verwendung eines Pflichtfelds (siehe Abschnitt 6.2). Dabei wurde in der Konfiguration das Feld *License* als obligatorisch vorgegeben. Dies bewirkt, dass die Beschriftung des Felds mit einem Stern versehen wird, um auf ein Pflichtfeld hinzuweisen. Solange in einem solchen Feld kein Inhalt eingetragen wurde, erscheint neben diesem ein Warnhinweis. Zusätzlich wird beim Abspeichern der Paketdokumentation eine Warnmeldung im Kopfbereich des Dokumentationseditors angezeigt.

Paketinterne Aspekte

Details über die Implementierung eines Pakets werden im Tab *Internals* aufgeführt (siehe Abb. 7.2). Für die Beschreibung des Zusammenspiels der Klassen eines Pakets wird ebenfalls ein HTML-Editor verwendet. Darin können neben Text auch Grafiken, wie z. B. UML-Diagramme, eingebunden werden. Zusätzlich werden weitere Oberflächenelemente bereitgestellt, um die Abhängigkeiten aufzulisten oder um zu beschreiben, auf welche zukünftigen Änderungen die Implementierung des Pakets vorbereitet ist. Der Bereich *Interface* dokumentiert die ein- und ausgehende Schnittstelle des Pakets. Dabei kann angegeben werden, was das Paket von anderen Komponenten erwartet und was es an andere Komponenten liefert.

Separate Dokumentation

Bei der Software-Entwicklung entstehen zahlreiche Dokumente bereits vor dem Beginn der Implementierung [LL10, S. 260]. Um die Software-Dokumentation redundanzfrei und damit einfacher wartbar zu machen, unterstützt J-PaD das Einbinden von separater Dokumentation in die Dokumentation eines Pakets. Dafür wird eine Tabelle bereitgestellt, welche die Verwaltung von externen Ressourcen erlaubt (siehe Abb. 7.3). Darin können verschiedene Typen von separater Dokumentation hinzugefügt werden. Dokumentation kann über eine URL oder über einen absoluten Pfad referenziert werden. Pfadangaben können z. B. in einem Unternehmen eingesetzt werden, in dem Dokumente auf einem Netzlaufwerk gespeichert sind. Zusätzlich können Dateien in die Paketstruktur importiert werden. Diese werden im *doc-files*-Ordner eines Pakets gespeichert, wodurch diese in direkter Nähe zur Paketdokumentation liegen (siehe Abschnitt 6.1.3). In Abbildung 7.3 wurde beispielsweise das Klassendiagramm aus der vorherigen Ansicht importiert.

Zu jeder eingebundenen Ressource lässt sich eine Beschreibung angeben. Diese kann direkt in der Tabelle bearbeitet werden.

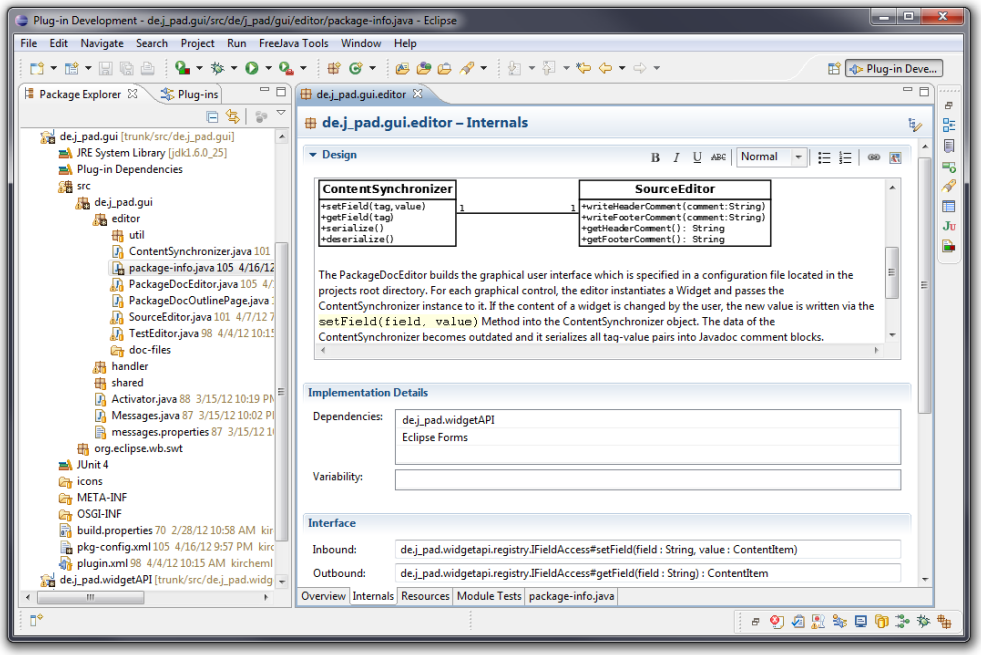

Abbildung 7.2.: Screenshot paketinterne Aspekte – Details zur Implementierung.

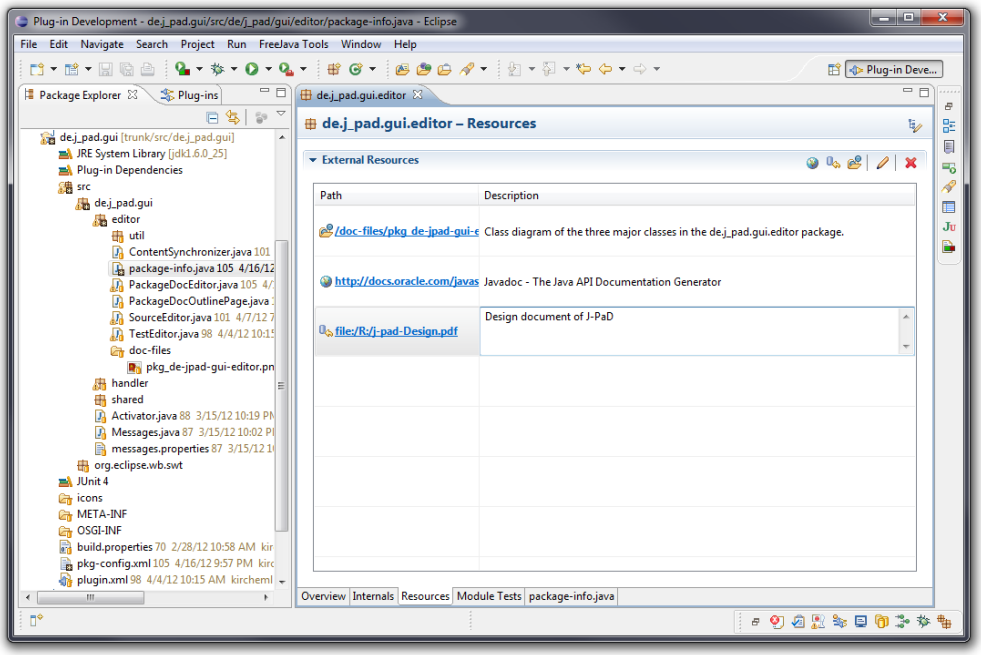

Abbildung 7.3.: Screenshot Ressourcen-Tabelle – Auflistung separater Dokumentation.

Modul-Tests

Eine weitere tabellarische Darstellung zeigt Abbildung 7.5 mit einer Verwaltung von Modul-Testfällen. Darin lassen sich Methoden eintragen, die die Implementierung eines Pakets testen. Zusätzlich kann zu jeder Methode eine Beschreibung angegeben werden.

Zu der Tabelle lassen sich beliebige Methoden hinzufügen. Diese müssen noch nicht im Code vorhanden sein (siehe Abschnitt 6.2.1). Ein automatisierter Import wird für JUnit-Testmethoden bereitgestellt. Über einen Assistenten (siehe Abb. 7.4) können dabei alle JUnit-Testfälle eines Projekts aufgelistet werden. In dieser Liste sind standardmäßig alle Testfälle ausgewählt, die dem aktuellen Paket des Dokumentationseditors zugeordnet sind. Testfälle aus anderen Paketen können jedoch auch selektiert werden. Diese werden nach Abschluss des Assistenten in die Tabelle der Modul-Testfälle übernommen.

Eine Methode wird als JUnit-Testfall erkannt, sobald diese die Annotation `@Test` des JUnit-Frameworks aufweist. Ebenfalls werden Methoden berücksichtigt, deren Klasse von `junit.framework.TestCase` ableitet und deren Name mit *test* beginnt. Zudem müssen diese Methoden eine leere Parameterliste aufweisen. Ist einer Methode ein Javadoc Dokumentationskommentar zugeordnet, so wird aus diesem die allgemeine Beschreibung extrahiert und als initialer Wert dem Beschreibungsfeld in der Tabelle gesetzt. Eine anschließende Synchronisation zwischen dem Dokumentationskommentar und dem Beschreibungsfeld findet jedoch nicht statt.

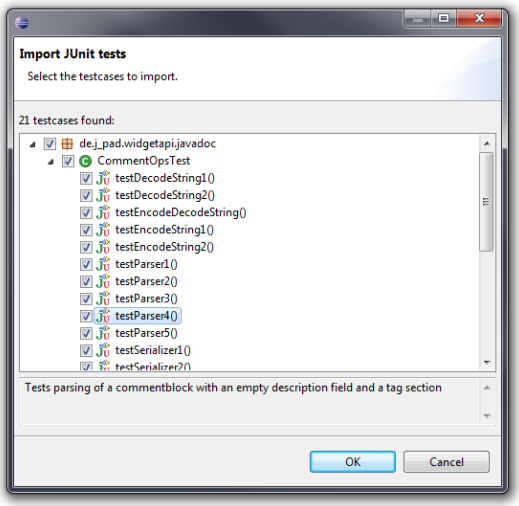

Abbildung 7.4.: Screenshot JUnit-Testfälle – Assistent zur Suche nach JUnit-Testfällen.

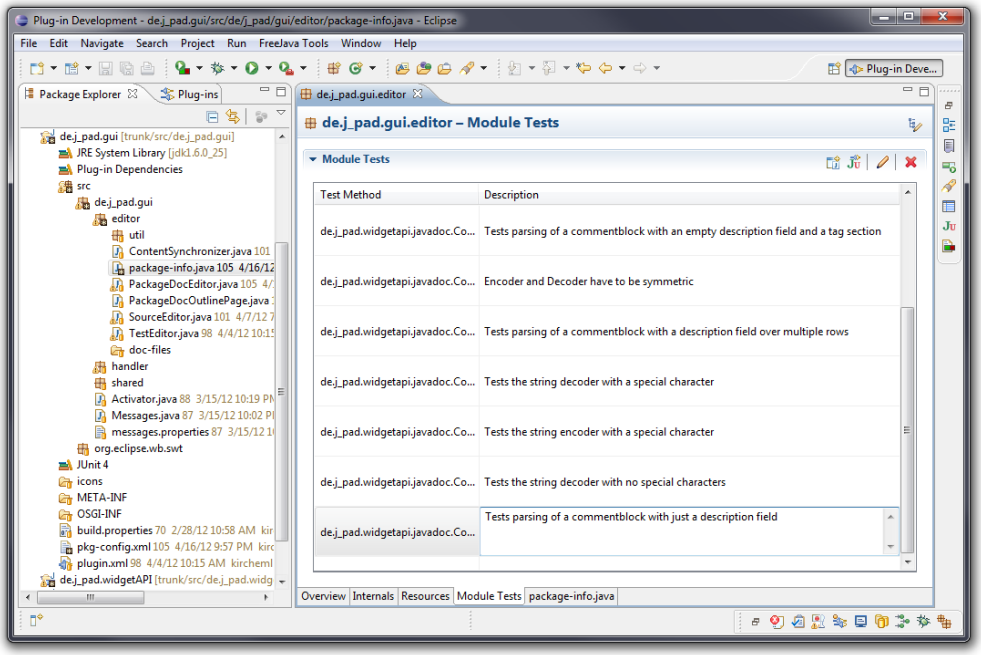

Abbildung 7.5.: Screenshot Modul-Testfälle – Auflistung von Testfällen zu den Klassen eines Pakets.

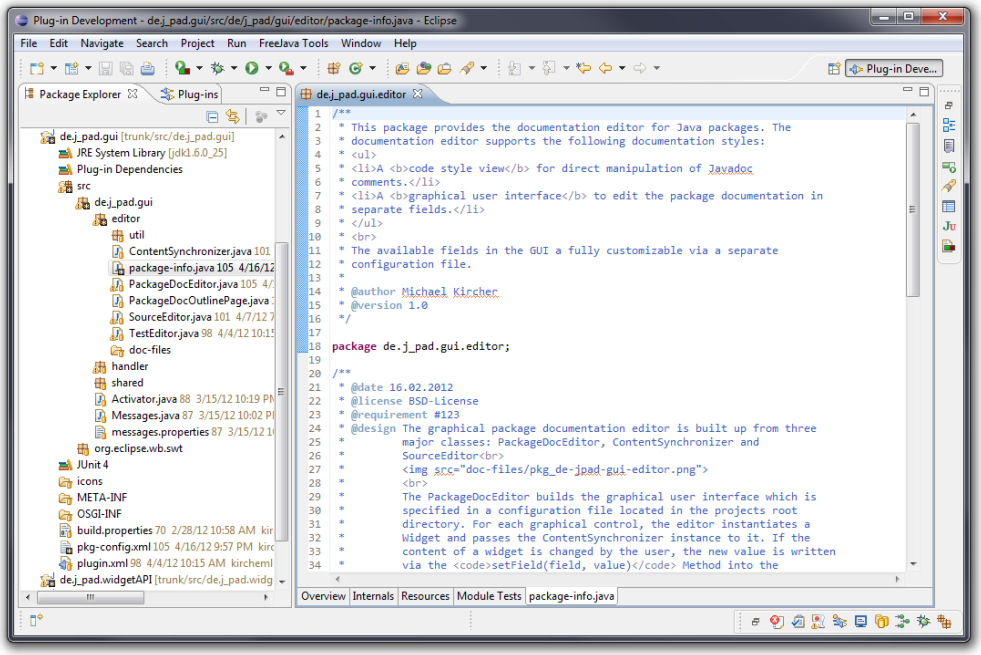

Abbildung 7.6.: Screenshot Code-Ansicht – Integration der Code-Ansicht in den grafischen Dokumentationseditor.

Code-Ansicht

Im letzten Tab des Dokumentationseditors ist eine Code-Ansicht integriert (siehe Abb. 7.6). Hierfür wird der Standard-Editor für Java-Dateien verwendet. Darin werden die Inhalte aller Oberflächenelemente nach Javadoc Konventionen in Kommentarblöcken gespeichert. Die Code-Ansicht ist dabei mit den Oberflächenelementen synchronisiert, sodass beide Darstellungen stets dieselben Informationen anzeigen (siehe Abschnitt 8.2.3).

7.3. Konfiguration

In diesem Abschnitt wird die Konfiguration von J-PaD beschrieben. J-PaD stellt dafür einen eigenen Editor bereit, um die möglichen Datenfelder des Dokumentationseditors zu konfigurieren. Dabei wird eine XML-Datei erzeugt, die als Vorlage für alle Dokumentationseditoren eines Projekts dient. Der Konfigurationseditor ist in zwei Tabs unterteilt, in welchen die Konfiguration entweder über eine grafische Oberfläche oder direkt über einen XML-Editor bearbeitet werden kann. Abbildung 7.7 zeigt den grafischen Konfigurationseditor.

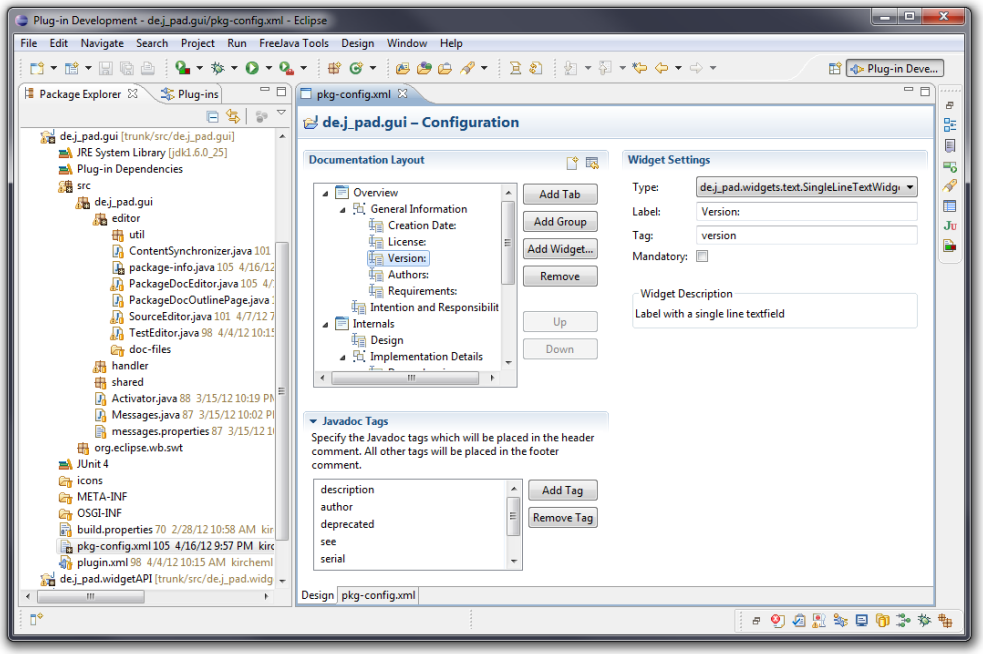

Abbildung 7.7.: Screenshot Konfigurationseditor – Definition der Eigenschaften für alle Paketdokumentationen eines Projekts.

Der grafische Konfigurationseditor (siehe Abb. 7.7) ist in drei Bereiche unterteilt: Im Bereich links oben lassen sich die Oberflächenelemente für die Paketdokumentation strukturieren und auf der rechten Seite dazu Einstellungen vornehmen. Der untere Bereich dient zur Konfiguration der Javadoc Tags.

Der Bereich zur Strukturierung der Oberflächenelemente zeigt eine Hierarchie, die auf jeder Ebene konfigurierbar ist. Auf der obersten Ebene lassen sich beliebig viele Elemente spezifizieren, die im Dokumentationseditor als jeweils ein Tab dargestellt werden. Diese können eingesetzt werden, um verschiedene Elemente auf einer höheren Ebene zu gruppieren. Der Inhalt eines Tabs kann mit Gruppen (via »`Add Group`«) oder direkt mit Oberflächenelementen (via »`Add Widget...`«) befüllt werden. Eine Gruppe kann benannt werden und dient dazu, zusammengehörige Oberflächenelemente visuell zu gruppieren. Abbildung 7.1 gruppiert z. B. mehrere Eingabefelder über die Gruppe *General Information*. Auf der untersten Ebene lassen sich aus einer Menge an verfügbaren Oberflächenelementen die zu dokumentierenden Eigenschaften eines Pakets festlegen. Dabei können, abhängig von der Eigenschaft, verschiedene Oberflächenelemente, wie z. B. Textfelder, Listen oder Tabellen ausgewählt werden (siehe Abschnitt 6.2.1 für eine vollständige Übersicht).

Entsprechend des ausgewählten Elements in der Hierarchie, werden im rechten Bereich die Einstellungen dafür angezeigt. Hierfür lassen sich für ein Oberflächenelement die Eigenschaften, wie eine Beschriftung oder eine Assoziation mit einem Tag, festlegen. Ein Tag dient dazu, die dokumentierten Inhalte eindeutig einem Oberflächenelement zuzuordnen. Dabei können Javadoc Tags verwendet oder eigene Tags definiert werden.

Im Bereich links unten kann spezifiziert werden, welche Tags Javadoc unterstützt. Die Inhalte aller Oberflächenelemente, deren assoziierter Tag in dieser Liste enthalten ist, werden in den Kopfkommentar der package-info.java Datei geschrieben (siehe Abschnitt 6.1.2). Ist ein Tag nicht in dieser Liste enthalten, so wird die Dokumentation dem Fußkommentar zugeordnet. Eine Sonderrolle spielt dabei der Tag *description*. Dieser ist standardmäßig nicht in Javadoc enthalten. Er wird jedoch verwendet, um die allgemeine Beschreibung in einem Kommentarblock zu setzen (siehe Abschnitt 6.1.1). Dabei wird der Tag selbst nicht in die package-info.java Datei geschrieben, sondern nur dessen Inhalt.

7.4. Erweiterung

Die Architektur von J-PaD wurde so konzipiert, dass sich dessen Kernsystem ohne Anpassungen mit zusätzlichen Oberflächenelementen erweitern lässt. Hierfür definiert die Komponente `Widget API` einen Erweiterungspunkt, über den neue Oberflächenelemente hinzugefügt werden können (siehe auch Abschnitt 8.2.1 und 8.2.2).

Um den Erweiterungspunkt verwenden zu können, muss in einem Plugin-Projekt, welches die Implementierung der neuen Oberflächenelemente enthält, eine Abhängigkeit auf das Plugin `de.j_pad.widgetAPI` gesetzt werden. Anschließend kann der Erweiterungspunkt *de.j_pad.widgetAPI.widgets* in der Datei `plugin.xml` hinzugefügt werden. An diesem lassen sich nun die eigenen Oberflächenelemente registrieren.

Kapitel 8

Umsetzung

Nachdem in Kapitel 7 J-PaD vorgestellt wurde, sollen nun die technischen Details genauer betrachtet werden. Dazu wird die Architektur von J-PaD beschrieben und es werden zentrale Elemente der Implementierung erläutert. Im Anschluss wird ein Überblick über das Vorgehen während der Entwicklung sowie ein Bericht über die Selbstanwendung von J-PaD gegeben.

8.1. Eingesetzte Technologien

J-PaD baut auf den folgenden Technologien auf. Die Versionsangaben beziehen sich dabei auf die minimal erforderliche Version.

- *Java 6.0*
- *Eclipse 3.7.1* [Thea]
  Soll J-PaD weiterentwickelt werden, so ist die RCP-Variante erforderlich, ansonsten reicht die Java-Variante aus.
- *Eclipse Forms* [Dejo5]
  Stellt Oberflächenelemente analog zu Web-Formularen dar. Wird standardmäßig in Eclipse für grafische Editoren eingesetzt.
- *Eclipse Delta Pack* [Theb]
  Nur für die Weiterentwicklung erforderlich. Ermöglicht es, J-PaD für unterschiedliche Plattformen zu exportieren.
- *Onpositive Richtext Editor* [Ecl]
  Ein HTML-Editor, der es ermöglicht, HTML-Syntax in einem WYSIWYG-Stil zu bearbeiten. Der Programmcode dieser Komponente wurde bei der Entwicklung von J-PaD erweitert.

8.2. Implementierung

8.2.1. Architektur

Die Architektur von J-PaD unterteilt die Software in die vier Komponenten `Widgets`, `Widget API`, `Documentation GUI` und `Config Manager` (siehe Abb. 8.1). Dabei kommt das Plugin-Architekturmuster [LL10] in mehreren Stufen zum Einsatz. Das Kernsystem stellt die Entwicklungsumgebung Eclipse dar, in welchem sich J-PaD als Plugin registriert. Gleichzeitig ist J-PaD selbst ein Kern, d.h. es lässt sich mit zusätzlichen Oberflächenelementen erweitern. Die beiden Komponenten `Widget API` und `Documentation GUI` übernehmen die Rolle des Kernsystems in J-PaD.

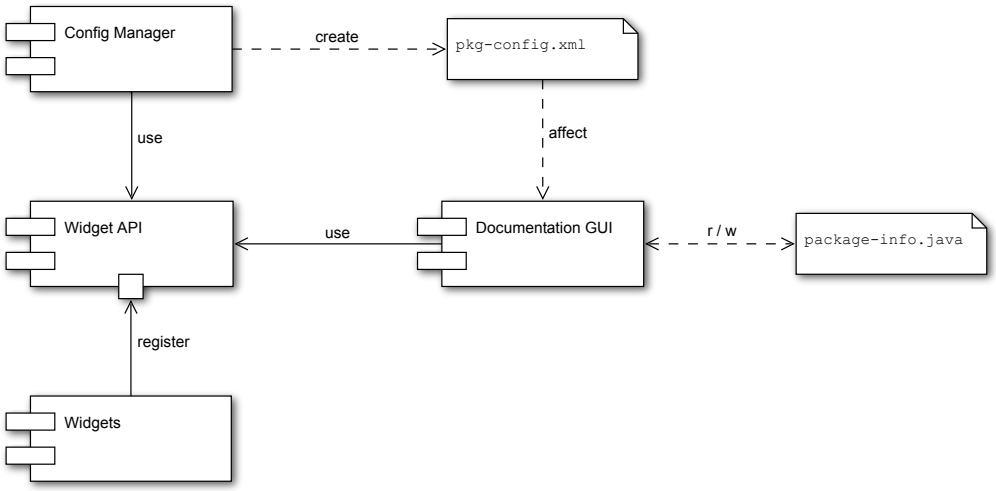

Abbildung 8.1.: Architektur von J-PaD

Die vier Komponenten von J-PaD übernehmen die folgenden Aufgaben:

Widgets stellt eine Menge an implementierten Widgets bereit. Ein Widget ist ein grafisches Oberflächenelement, welches sich im Editor zur Dokumentation eines Pakets einsetzen lässt. Im Unterschied zu den Standardelementen aus SWT (*Standard Widget Toolkit*) besitzen Widgets weitere Eigenschaften, wie eine Beschriftung und einen zugeordneten Tag. Dieser Tag gibt an, welche Eigenschaft eines Pakets sich über das entsprechende Widget dokumentieren lässt (siehe Abschnitt 6.1.1). J-PaD enthält beispielsweise Widgets für einzeilige Texteingaben, Datumseingaben oder einen mehrzeiligen HTML-Editor (siehe Abschnitt 6.2.1).

Widget API ist eine zentrale Komponente von J-PaD. Diese ermöglicht es, über einen Erweiterungspunkt zusätzliche Widgets der Software hinzuzufügen. Dafür wird das Konzept von *Extension Points* aus Eclipse verwendet, welche es erlauben, Funktionalität durch andere Plugins bereitzustellen. Der Erweiterungspunkt dieser Komponente definiert

einen Vertrag zwischen Plugins, die zusätzliche Widgets implementieren und dem Plugin Widget API. Im Plugin Widget API wird dafür eine Registrierung angelegt, welche Informationen über alle vorhandenen Widgets hält. Interessierte Komponenten können darauf zugreifen, um diese Widgets zu verwenden.

Config Manager erlaubt eine grafische Bearbeitung des Schemas für den Dokumentationseditor. Dafür greift die Komponente auf die Registrierung der Widget API zu und liest die vorhandenen Widgets aus. Diese können vom Benutzer von J-PaD verwendet werden, um das Layout des Dokumentationseditors zu spezifizieren. Die Konfiguration wird anschließend in der Datei `pkg-config.xml` gespeichert und im Wurzelverzeichnis eines Projekts platziert. Diese XML-Datei dient somit als Vorlage für alle Pakete innerhalb eines konfigurierten Projekts. Enthält ein Projekt keine solche Datei, so wird eine Standardvorlage für den Dokumentationseditor verwendet.

Documentation GUI stellt die grafische Oberfläche bereit, um Pakete zu dokumentieren. Die Oberfläche wird dabei anhand der Vorgaben aus der `pkg-config.xml` Datei aufgebaut. Darin ist spezifiziert, in welcher Anordnung welche Widgets verwendet werden sollen. Die Komponente Documentation GUI greift nun mit diesen Informationen auf die Komponente Widget API zu, um die entsprechenden Widgets in der grafischen Oberfläche des Dokumentationseditors zu instanziieren. Die Inhalte der Oberflächenelemente werden dabei synchron mit den Daten der `package-info.java` Datei gehalten (siehe Abschnitt 8.2.3).

8.2.2. Widget Implementierung

Mithilfe der Widget-Registrierung aus der Komponente `Widget API` lassen sich auf einfache Weise neue Widgets zu J-PaD hinzufügen. Die Registrierung wird beim Laden der Plugins befüllt und kann daher noch nicht die Instanzen der Oberflächenelemente enthalten. Einerseits nicht, da jedes Oberflächenelement in eine Layouthierarchie der grafischen Oberfläche eingeordnet werden muss. Diese ist jedoch zum Zeitpunkt des Ladens der Plugins noch nicht erstellt. Andererseits soll es möglich sein, mit einem Widget-Typ mehrere, sich unterscheidende Eigenschaften eines Pakets zu dokumentieren. Ein Widget-Typ soll sich also mehrfach in der Dokumentation eines Pakets verwenden lassen.

In J-PaD werden daher Widgets nach dem Entwurfsmuster *Abstrakte Fabrik* implementiert [GHJV95]. Abbildung 8.2 zeigt dies exemplarisch für einen Widget-Typ. Die linke Spalte ordnet dabei die Klassen den Komponenten aus Abbildung 8.1 zu. Jedes Widget, das in J-PaD eingebunden wird, muss somit aus mindestens zwei Klassen bestehen. Eine erweitert die Klasse `AbstractWidget`, welche das Erstellen der grafischen Oberflächenelemente übernimmt. Die zweite Klasse leitet von `AbstractWidgetFactory` ab und ist für die Instanziierung von Widget-Objekten verantwortlich, welche die Klasse `AbstractWidget` erweitern. In der Widget-Registrierung werden daher lediglich Objekte der Factory-Klassen gehalten. Diese haben nur geringe Speicheranforderungen, da sie noch keine Oberflächenelemente enthalten. Zudem reicht eine Factory pro Widget-Typ aus, da sich damit beliebig viele Widgets erzeugen lassen.

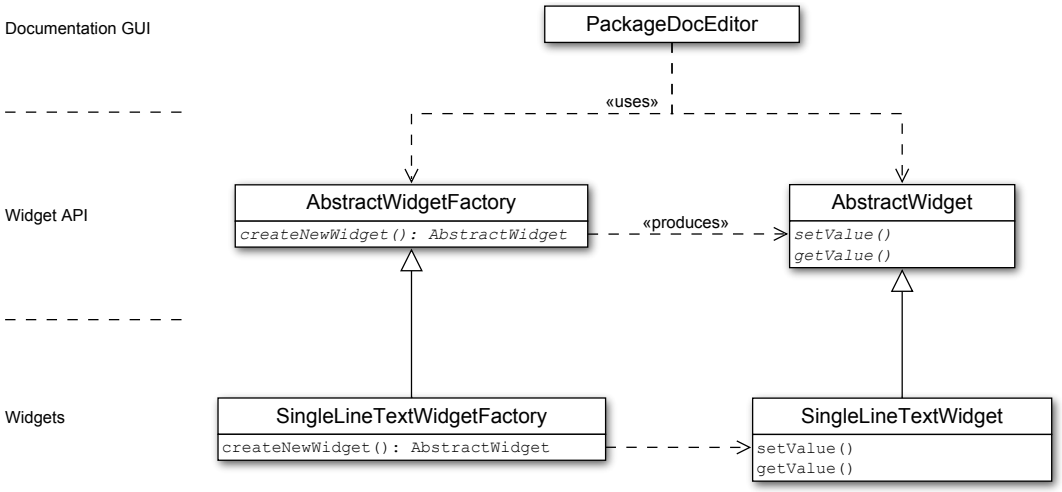

Abbildung 8.2.: Entwurfsmuster *Abstrakte Fabrik* für die Instanziierung von Widgets.

Die Klasse `PackageDocEditor` ist für das Erstellen der grafischen Oberfläche der Paketdokumentation verantwortlich. Das Layout des Dokumentationseditors wird über die Datei `pkg-config.xml` spezifiziert (siehe Abb. 8.1). Diese gibt vor, welche Factory-Klassen verwendet werden sollen. Der Editor greift damit auf die Registrierung zu und liest die notwendigen Factory-Objekte aus. Über diese Factorys können nun die konkreten Widgets instanziiert und in die Oberfläche eingebunden werden. Dem Editor ist dabei nur die Ebene der `AbstractWidgetFactory` und `AbstractWidget` bekannt, die konkrete Implementierung eines Widgets bleibt ihm verborgen. Somit ist jedes Widget selbst verantwortlich, wie die Inhalte aus dessen Oberflächenelementen ausgelesen und darin gesetzt werden.

8.2.3. GUI Synchronisation

Die Ansicht zur Dokumentation von Paketen bietet zwei unterschiedliche Dokumentationsstile. Eine Möglichkeit wird durch die grafische Oberfläche bereitgestellt, über die sich gezielt einzelne Felder eines Pakets beschreiben lassen. Zusätzlich wird ein erweiterter Modus angeboten, welcher die direkte Bearbeitung der Dokumentationskommentare über eine Code-Ansicht ermöglicht.

Beide Arten verwenden dabei dieselben Daten. Wird die Dokumentation in einem Modus geändert, so muss die andere Darstellung aktualisiert werden. Die trivialen Ansätze für dieses Problem sind, die Daten entweder bei jeder Tastatureingabe oder erst beim Speichern zu synchronisieren. Diese stellen jedoch in Bezug auf die Usability keine zufriedenstellende Lösung dar. So erfordert der erste Ansatz viel Rechenaufwand, da für jedes eingegebene Zeichen die Daten serialisiert bzw. die Kommentare geparst werden müssen. Dies führt vor allem bei einer umfangreichen Dokumentation zu schlechter Performanz. Der zweite Ansatz führt dazu, dass beide Ansichten über längere Zeit unterschiedliche Inhalte darstellen.

Dadurch kann der Benutzer schnell den Überblick verlieren und es können Konflikte auftreten, falls sich vorgenommene Änderungen widersprechen.

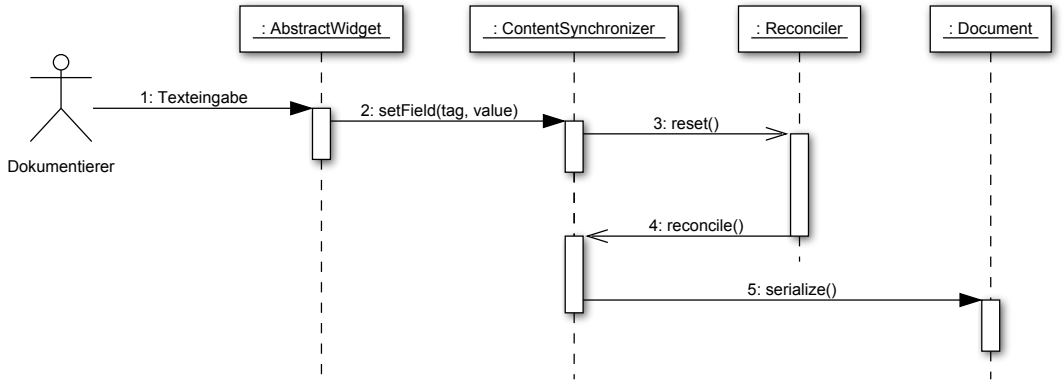

Abbildung 8.3.: Synchronisation zwischen GUI und Code-Ansicht.

In J-PaD wird daher ein Reconciler (engl. für *in Einklang bringen*) verwendet. Dieser nimmt die Synchronisation erst nach einer gewissen Zeitverzögerung vor, nachdem der Benutzer einen Text eingegeben hat. Das Sequenzdiagramm in Abbildung 8.3 zeigt den Ablauf der Synchronisation. Der Benutzer gibt dabei Daten in der grafischen Oberfläche innerhalb eines Widgets ein. Diesem Widget ist ein Tag zugeordnet, welcher zusammen mit der Eingabe an die Klasse `ContentSynchronizer` weitergeleitet wird. Diese Klasse stellt die Verbindung zwischen dem grafischen Dokumentationseditor und der Code-Ansicht her und sorgt dafür, dass beide Ansichten synchron gehalten werden. Die Klasse `ContentSynchronizer` nutzt dazu einen `Reconciler`, welcher in einem Hintergrundthread einen Zähler startet. Die Schritte 1 – 3 in Abbildung 8.3 werden für jede Zeicheneingabe des Benutzers ausgeführt, wobei der Zähler des Reconcilers immer wieder zurückgesetzt wird. Erst wenn der Benutzer eine Zeit lang keine Eingabe mehr vornimmt, überschreitet der Zähler einen bestimmten Schwellenwert, wodurch die eigentliche Synchronisation durchgeführt wird. Dazu werden alle Tag-Wert-Paare in einen Dokumentationskommentar serialisiert und in das Modell der Code-Ansicht geschrieben. Dieses Modell ist durch ein `Document`-Objekt realisiert, das die Code-Ansicht bei Änderungen aktualisiert.

Die Rückrichtung funktioniert dabei analog. Ändert der Benutzer direkt die Dokumentationskommentare in der Code-Ansicht, so werden erst nach einer gewissen Verzögerung die Inhalte der Widgets in der grafischen Oberfläche angepasst.

Die Bibliothek JFace enthält bereits Implementierungen eines Reconcilers. Jedoch können diese nur in Verbindung mit einem Texteditor, wie der Editor zur Bearbeitung von Java-Code, verwendet werden. Das oben beschriebene Prinzip soll auch für Widgets in J-PaD verwendet werden. Diese erfüllen in der Regel nicht die Anforderungen eines umfangreichen Texteditors, weshalb hier eine eigene Implementierung eines Reconcilers verwendet wird. Sinnvolle Werte für den Schwellenwert des Zählers liegen bei etwa 500 ms, bis die Synchronisation gestartet wird. Der Benutzer nimmt somit keine Performanzeinbußen während Tastatureingaben

wahr und erhält, abgesehen von einem kurzen Zeitintervall, stets synchrone Daten in beiden Ansichten.

8.3. Vorgehen

Bevor ich die Implementierung von J-PaD aufnahm, erstellte ich zunächst einen Demonstrationsprototypen [LL10] über den möglichen Aufbau der grafischen Oberfläche. Dieser zeigte mögliche Ausprägungen und konkretisierte die Anforderungen. Anschließend entwarf ich die Software-Architektur für J-PaD (siehe Abschnitt 8.2.1).

Aufgrund des Forschungscharakters dieses Projekts habe ich mich für eine inkrementelle Software-Entwicklung [LL10] entschieden. Dabei wird das System in mehreren aufeinanderfolgenden Ausbaustufen entwickelt. Im ersten Drittel der mir zur Verfügung stehenden Zeit entwickelte ich zunächst das Kernsystem von J-PaD. Dieses besteht aus den Komponenten `Widget API` und `Documentation GUI`. Das Kernsystem übernimmt in erster Linie den Aufbau der grafischen Oberfläche sowie die Synchronisation der Daten mit der `package-info.java` Datei.

Nachdem das Kernsystem realisiert war, begann ich mit der Entwicklung der übrigen Komponenten. Dazu zählt der `Config Manager`, der eine grafische Bearbeitung der Konfigurationsdatei erlaubt, sowie Erweiterungen in Form von Widgets. Hierbei wählte ich zunächst einfache Widgets aus, wie beispielsweise ein Widget für ein einzeiliges Textfeld und eines für eine Liste von Werten (siehe Abschnitt 6.2.1). Somit konnte ich mich von der Realisierbarkeit meines Konzepts vergewissern und es entstand bereits sehr früh ein einsatzbereites System.

Die Erweiterbarkeit von J-PaD ermöglichte es mir, weitere Anregungen, die im Laufe des Projekts aufkamen, einfach zu integrieren. Zudem konnten so durch regelmäßige Rücksprachen noch ungeklärte Details umgesetzt werden.

8.3.1. Evaluation

Durch die inkrementelle Software-Entwicklung entstand schon sehr früh ein einsatzbereites System. Diesen Vorteil nutzte ich aus, um die Module von J-PaD in einer Selbstanwendung zu dokumentieren. Gleichzeitig wurde die Software dabei unter realen Bedingungen getestet und konnte währenddessen kontinuierlich verbessert werden. Bei der Selbstanwendung wurden die folgenden Beobachtungen gemacht:

- Werden Pakete eines Projekts dokumentiert, so führt dies zu einer gedanklichen Auseinandersetzung über die Funktionalität der Pakete. Während des Dokumentationsprozesses werden somit Klassen erkannt, welche einen nur geringen Zusammenhalt zu anderen Klassen eines Pakets aufweisen. Diese können daraufhin umstrukturiert werden, wodurch das Projekt eine insgesamt bessere Paketstruktur erhält.

- Das Anlegen einer Paketdokumentation ist mit J-PaD auf sehr einfache Weise möglich. Dazu reichen lediglich zwei Klicks aus, woraufhin die notwendige Datei angelegt und der entsprechende Dokumentationseditor geöffnet werden. Durch diese Einfachheit ist der Entwickler eher dazu geneigt, Pakete zu dokumentieren (siehe dazu auch Kapitel 1).
- Durch das vorgegebene Schema im Dokumentationseditor sieht der Entwickler direkt, welche Felder über ein Paket zu dokumentieren sind. Dadurch wird er beim Dokumentieren angeleitet und führt somit die eher unbeliebteren Dokumentationsaufgaben mit einer höheren Wahrscheinlichkeit durch.
- Ändert sich die Struktur des Programmcodes, so muss die Paketdokumentation dafür zusätzlich angepasst werden. J-PaD überwacht den Code nicht eigenständig auf Änderungen. Der Entwickler muss daher selbst für die Konsistenz zwischen Dokumentation und Code sorgen. Er wird jedoch dabei unterstützt, da J-PaD direkt in der Entwicklungsumgebung verwendet werden kann. Darin lassen sich Code und Dokumentation über eine einheitliche Art und Weise bearbeiten [CAFF10].

8.3.2. Empfohlenes Dokumentationsschema

J-PaD wird standardmäßig mit einer bereits einsatzbereiten Konfiguration ausgeliefert. Dadurch können Einsteiger dieses Werkzeug direkt verwenden, ohne zuerst ein eigenes Schema entwerfen zu müssen. Gleichzeitig wird fortgeschrittenen Benutzern die Möglichkeit gegeben, das Schema flexibel anzupassen.

Während der Selbstanwendung von J-PaD wurde eine Konfiguration verwendet, welche sich an dem Blackbox- und Schnittstellen-Template nach Starke und Hruschka [SH09] (siehe Abschnitt 3.5.2) orientiert. Dieses Schema wird in J-PaD als Standardkonfiguration eingesetzt und ist auch im Anhang im Ausschnitt A.1 beigefügt.

Kapitel 9

Zusammenfassung und Ausblick

Dieses Kapitel fasst die Ergebnisse dieser Arbeit – und insbesondere der entstandenen Software J-PaD – zusammen. Dazu werden die Konzepte von J-PaD nochmals aufgegriffen und anschließend dessen Vorteile aufgezeigt. Zum Abschluss werden mögliche Erweiterungspunkte für J-PaD vorgestellt.

9.1. Zusammenfassung

Diese Arbeit beschäftigt sich mit der integrierten Dokumentation von Software-Modulen. Die Dokumentation befindet sich dabei in direkter Nähe zum Programmcode und kann somit bei Code-Änderungen einfacher aktualisiert werden.

Es wurden existierende Werkzeuge zur integrieren Dokumentation von Software untersucht und deren Besonderheiten herausgearbeitet. Der Entwickler kann damit Code-Einheiten mithilfe von Kommentarblöcken dokumentieren. Häufig muss hierfür jedoch eine bestimmte Konvention eingehalten werden, welche durch das eingesetzte Werkzeug vorgegeben wird. Für einen Entwickler ist das Dokumentieren von Code daher mit zusätzlichem Einarbeitungsaufwand in das Dokumentationswerkzeug verbunden. Des Weiteren geben viele Werkzeuge kein Schema für die Dokumentation vor. Die Entscheidung, welche Informationen über eine Code-Einheit dokumentiert werden sollen, bleibt somit dem Entwickler überlassen. Eine einheitliche Dokumentationsstruktur kann dadurch nicht gewährleistet werden.

Das Ziel dieser Diplomarbeit war daher, ein Werkzeug zur integrierten Dokumentation von Java-Paketen zu entwickeln. Dabei entstand das Werkzeug *J-PaD*. Dieses wurde in der Programmiersprache Java geschrieben und als Plugin für die Entwicklungsumgebung Eclipse konzipiert. Bei der Entwicklung von J-PaD flossen die Erkenntnisse aus der Untersuchung existierender Werkzeuge mit ein. Mit der Betrachtung von Usability-Aspekten wurden Eigenschaften an ein Werkzeug für die Software-Entwicklung identifiziert, um dessen Akzeptanz bei den Entwicklern zu erhöhen. J-PaD wurde während der Entwicklung auf sich selbst angewendet, wodurch das Werkzeug bereits an einem realen Projekt getestet

wurde. Als Resultat dieser Diplomarbeit entstand somit ein einsatzbereites Werkzeug, das im Internet veröffentlicht ist.

9.1.1. Vorteile von J-PaD

Mit der Verwendung von J-PaD ergeben sich die folgenden Vorteile:

- Durch die Integration in Eclipse stellt sich die Verwendung von J-PaD als sehr einfach dar. Die Dokumentation kann dabei in der gleichen Umgebung bearbeitet werden, in der auch der Programmcode eingegeben wird. Ein Entwickler muss somit nicht zwischen unterschiedlichen Werkzeugen wechseln, um Pakete zu dokumentieren. Zudem orientiert sich die Gestaltung der grafischen Oberfläche an gängigen Editoren von Eclipse, wodurch ein Entwickler eine bekannte Struktur und ein gewohntes Verhalten vorfindet.
- J-PaD baut auf einem bestehenden Standard zur Dokumentation von Paketen auf und bleibt kompatibel zu Javadoc. Durch die Auftrennung der Dokumentation in einen Kopf- und einen Fußkommentar können mehr Eigenschaften über ein Paket beschrieben werden, als von Javadoc bereitgestellt werden. Die Informationen werden dennoch zusammen in einer Datei gespeichert. Die Zuordnung einer Eigenschaft zu Kopf- oder Fußkommentar lässt sich individuell konfigurieren. Damit ist J-PaD für zukünftige Javadoc-Versionen vorbereitet, die möglicherweise weitere Tags unterstützen.
- Die Einstiegshürde für dieses Werkzeug wird durch die Kompatibilität zu Javadoc minimiert. Wurden die Pakete eines Projekts bereits über eine `package-info.java` Datei dokumentiert, so kann dieses Werkzeug auch verwendet werden, um die Dokumentation nur zu lesen. Die Code-Dateien eines Projekts werden dadurch nicht verändert.
- Das Dokumentationsschema der Paketdokumentation lässt sich flexibel konfigurieren. Dabei können die zu dokumentierenden Eigenschaften eines Pakets vollständig ausgetauscht und die grafische Oberfläche kann nach eigenen Präferenzen strukturiert werden. Das Schema lässt sich beispielsweise auch so anpassen, dass nur zu Javadoc kompatible Eigenschaften benutzt werden.
- Durch das Dokumentationsschema wird eine einheitliche Struktur für alle Paketdokumentationen eines Projekts vorgegeben. Ein Entwickler muss sich somit nicht für jedes Paket erneut Gedanken über die Strukturierung machen. Zudem können mit einem einheitlichen Schema benötigte Informationen schneller wiedergefunden werden.
- Die Architektur von J-PaD erlaubt es, die Software auf einfache Weise um neue Widgets zu erweitern. Zusätzliche Erweiterungen können über einen Erweiterungspunkt registriert werden und müssen den bestehenden Code von J-PaD nicht verändern.

9.2. Ausblick

J-PaD ist durch die Nutzung der `package-info.java` Datei an die Dokumentation von Java-Paketen gebunden. Soll eine integrierte Moduldokumentation auch für andere Programmiersprachen genutzt werden, so muss J-PaD zunächst um ein entsprechendes Konzept erweitert werden.

Bei der Anwendung von J-PaD identifizierten wir weitere Aspekte, die für eine Paketdokumentation sinnvoll erscheinen. Diese können in einer zukünftigen Weiterentwicklung in J-PaD integriert werden:

- Um den Dokumentationsfortschritt festzustellen, kann eine grafische Darstellung hilfreich sein. Diese kann beispielsweise den Anteil der dokumentierten Felder eines Pakets darstellen oder global aufzeigen, welche Pakete noch nicht dokumentiert wurden. Durch eine derartige Übersicht lässt sich möglicherweise die Motivation des Entwicklers steigern, die Dokumentation von Paketen zu vervollständigen.
- In dieselbe Richtung zielt eine mögliche Erweiterung der Pflichtfelder ab. Eclipse bietet über die *Problems View* eine Ansicht, die Warnungen und Fehlermeldungen zu allen Projekten auflistet. Darin lassen sich nicht ausgefüllte Pflichtfelder eintragen, um eine Dokumentation dafür zu erwirken.
- Javadoc erlaubt es, in den Dokumentationskommentaren Referenzen auf andere Code-Einheiten einzufügen. Diese sind nach dem Schema *Paket.Klasse#Mitgliedsname* aufgebaut (siehe Abschnitt 6.1.1). Die grafische Oberfläche könnte den Benutzer bei der Eingabe unterstützen, indem über eine automatische Vervollständigung die möglichen Code-Einheiten vorgeschlagen werden.
- Werden die Code-Dateien eines Pakets geändert, so findet bisher keine Synchronisierung mit der Paketdokumentation statt. Hierfür könnte beispielsweise die Tabelle der Modultestfälle erweitert werden, sodass eine Änderung in der Beschreibungs-Spalte der Tabelle die Aktualisierung des Dokumentationskommentars der entsprechenden Methode veranlasst.

Anhang A

Anhang

A.1. Konfigurationdatei pkg-config.xml

Der folgende Ausschnitt zeigt ein Beispiel für die Konfigurationsdatei `pkg-config.xml`. Diese schreibt den Aufbau des Dokumentationseditors für alle Paketdokumentationen innerhalb eines Projekts vor. Sie ist in J-PaD als Standardkonfiguration hinterlegt.

```
<?xml version="1.0" encoding="UTF-8"?>

<j-pad xmlns="http://www.j-pad.de/template" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.j-pad.de/template template.xsd">

        <config>
                <javadocTags>
                        <tag>description</tag>
                        <tag>author</tag>
                        <tag>deprecated</tag>
                        <tag>see</tag>
                        <tag>serial</tag>
                        <tag>since</tag>
                        <tag>version</tag>
                </javadocTags>
        </config>

        <tab>
                < title >Overview</title>
                <group>
                        < title >General Information</ title >
                        <widget>
                                <class>de.j_pad.widgets.date.DateWidgetFactory</class>
                                <label>Creation Date:</label>
                                <tag>date</tag>
                                <mandatory>false</mandatory>
                        </widget>
                        <widget>
                                <class>de.j_pad.widgets.text.SingleLineTextWidgetFactory</class>
                                <label>License:</label>
                                <tag>license</tag>
                                <mandatory>true</mandatory>
                        </widget>
                        <widget>
                                <class>de.j_pad.widgets.text.SingleLineTextWidgetFactory</class>
```

```
                        <label>Version:</label>
                        <tag>version</tag>
                        <mandatory>false</mandatory>
                </widget>
                <widget>
                        <class>de.j_pad.widgets.multivalue.MultiValueWidgetFactory</class>
                        <label>Authors:</label>
                        <tag>author</tag>
                        <mandatory>false</mandatory>
                </widget>
                <widget>
                        <class>de.j_pad.widgets.multivalue.MultiValueWidgetFactory</class>
                        <label>Requirements:</label>
                        <tag>requirement</tag>
                        <mandatory>false</mandatory>
                </widget>
        </group>
        <widget>
                <class>de.j_pad.widgets.html.HtmlWidgetFactory</class>
                <label>Intention and Responsibilities</label>
                <tag>description</tag>
                <mandatory>false</mandatory>
        </widget>
</tab>

<tab>
        < title >Internals</ title >
        <widget>
                <class>de.j_pad.widgets.html.HtmlWidgetFactory</class>
                <label>Design</label>
                <tag>design</tag>
                <mandatory>false</mandatory>
        </widget>
        <group>
                < title >Implementation Details</ title >
                <widget>
                        <class>de.j_pad.widgets.multivalue.MultiValueWidgetFactory</class>
                        <label>Dependencies:</label>
                        <tag>dependency</tag>
                        <mandatory>false</mandatory>
                </widget>
                <widget>
                        <class>de.j_pad.widgets.multivalue.MultiValueWidgetFactory</class>
                        <label> Variability: </label>
                        <tag> variability </tag>
                        <mandatory>false</mandatory>
                </widget>
        </group>
        <group>
                < title >Interface</ title >
                <widget>
                        <class>de.j_pad.widgets.text.SingleLineTextWidgetFactory</class>
                        <label>Inbound:</label>
                        <tag>inboundInterface</tag>
                        <mandatory>false</mandatory>
                </widget>
                <widget>
                        <class>de.j_pad.widgets.text.SingleLineTextWidgetFactory</class>
                        <label>Outbound:</label>
                        <tag>outboundInterface</tag>
                        <mandatory>false</mandatory>
                </widget>
                <widget>
```

```
                        <class>de.j_pad.widgets.multivalue.MultiValueWidgetFactory</class>
                        <label>Errors:</label>
                        <tag>interfaceErrors</tag>
                        <mandatory>false</mandatory>
                </widget>
        </group>
        <widget>
                <class>de.j_pad.widgets.html.HtmlWidgetFactory</class>
                <label>Interface  Resources and Usage</label>
                <tag>interfaceUsage</tag>
                <mandatory>false</mandatory>
        </widget>
</tab>

<tab>
        < title >Resources</title>
        <widget>
                <class>de.j_pad.widgets.resources.ResourcesWidgetFactory</class>
                <label>External Resources</label>
                <tag>res</tag>
                <mandatory>false</mandatory>
        </widget>
</tab>

<tab>
        < title >Module Tests</title>
        <widget>
                <class>de.j_pad.widgets.resources.ModuleTestWidgetFactory</class>
                <label>Module Tests</label>
                <tag>test</tag>
                <mandatory>false</mandatory>
        </widget>
</tab>
</j-pad>
```

Ausschnitt A.1: Beispiel einer `pkg-config.xml` Datei

Literaturverzeichnis

[AD05] A. Aguiar, G. David. WikiWiki weaving heterogeneous software artifacts. In *Proceedings of the 2005 international symposium on Wikis*, WikiSym '05, pp. 67–74. ACM, New York, NY, USA, 2005. (Zitiert auf den Seiten 21, 22, 32 und 40)

[BCH+00] B. W. Boehm, Clark, Horowitz, Brown, Reifer, Chulani, R. Madachy, B. Steece. *Software Cost Estimation with Cocomo II with Cdrom*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st Auflage, 2000. (Zitiert auf Seite 15)

[Boe79] B. W. Boehm. Guidelines for Verifying and Validating Software Requirements and Design Specifications. In P. A. Samet, editor, *Euro IFIP 79*, pp. 711–719. North Holland, 1979. (Zitiert auf den Seiten 7 und 10)

[Boe03] C. Boekhoudt. The Big Bang Theory of IDEs. *Queue*, 1:74–82, 2003. (Zitiert auf Seite 28)

[Bro83] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543 – 554, 1983. (Zitiert auf Seite 37)

[CAFF10] F. F. Correia, A. Aguiar, H. S. Ferreira, N. Flores. Patterns for consistent software documentation. In *Proceedings of the 16th Conference on Pattern Languages of Programs*, PLoP '09, pp. 12:1–12:7. ACM, New York, NY, USA, 2010. (Zitiert auf Seite 69)

[Cer] Certiv Analytics. JDocEditor. URL http://www.certiv.net/projects/jdoceditor.html. (Zitiert auf Seite 29)

[Con] Concordia University. SE-Editor. URL http://aseg.cs.concordia.ca/seeditor/. (Zitiert auf Seite 29)

[CSW02] M. E. Crosby, J. Scholtz, S. Wiedenbeck. The Roles Beacons Play in Comprehension for Novice and Expert Programmers. In *Programmers, 14th Workshop of the Psychology of Programming Interest Group, Brunel University*, pp. 18–21. 2002. (Zitiert auf den Seiten 38 und 39)

[Dej05] Dejan Glozic, IBM Canada Ltd. Eclipse Forms: Rich UI for the Rich Client, 2005. URL http://www.eclipse.org/articles/Article-Forms/article.html. (Zitiert auf Seite 63)

[Ecl] Eclipse Labs. Rich Text Editor. URL http://code.google.com/a/eclipselabs.org/p/richtext/. (Zitiert auf Seite 63)

[Fin90] B. P. Finkelstein. A hypertext-based documentation workbench for Ada-language systems. In *Proceedings of the seventh Washington Ada symposium on Ada*, WADAS '90, pp. 331–338. ACM, New York, NY, USA, 1990. (Zitiert auf Seite 15)

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. (Zitiert auf Seite 65)

[Hee] D. van Heesch. Doxygen. URL http://www.stack.nl/~dimitri/doxygen/. (Zitiert auf den Seiten 11 und 22)

[iee90] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, p. 1, 1990. (Zitiert auf Seite 14)

[Inn] Innovasys Ltd. Document! X. URL http://www.innovasys.com/products/dx2011/overview.aspx. (Zitiert auf Seite 25)

[iso98] ISO 9241-11 Ergonomic requirements for office work with visual display terminals (VDT), Part 11: Guidance on usability. 1998. (Zitiert auf Seite 35)

[iso06] ISO 9241-110: Ergonomie der Mensch-System-Interaktion – Teil 110: Grundsätze der Dialoggestaltung. 2006. (Zitiert auf Seite 36)

[Jet] JetBrains. IntelliJ IDEA. URL http://www.jetbrains.com/idea/. (Zitiert auf Seite 29)

[KM01] M. Kajko-Mattsson. The State of Documentation Practice within Corrective Maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01, pp. 354–. IEEE Computer Society, Washington, DC, USA, 2001. (Zitiert auf Seite 9)

[Knu84] D. E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97 – 111, 1984. (Zitiert auf den Seiten 7, 19 und 20)

[Lea10] G. T. Leavens. The future of library specification. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pp. 211–216. ACM, New York, NY, USA, 2010. (Zitiert auf den Seiten 7 und 16)

[LL10] J. Ludewig, H. Lichter. *Software Engineering – Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag Heidelberg, 2. Auflage, 2010. (Zitiert auf den Seiten 7, 10, 13, 15, 17, 57, 64 und 68)

[LSF03] T. C. Lethbridge, J. Singer, A. Forward. How Software Engineers Use Documentation: The State of the Practice. *IEEE Softw.*, 20:35–39, 2003. (Zitiert auf den Seiten 9 und 10)

[Mar] S. Marlow. Haddock. URL http://www.haskell.org/haddock/. (Zitiert auf den Seiten 11 und 27)

[ME] D. Meier-Eickhoff. iDocIt! URL http://code.google.com/p/idocit/. (Zitiert auf Seite 29)

[Mic] Microsoft. Visual Studio. URL http://www.microsoft.com/germany/visualstudio/. (Zitiert auf Seite 29)

[MV93] A. von Mayrhauser, A. Vans. From program comprehension to tool requirements for an industrial environment. In *Program Comprehension, 1993. Proceedings., IEEE Second Workshop on*, pp. 78 –86. 1993. (Zitiert auf den Seiten 37 und 39)

[MV95] A. von Mayrhauser, A. M. Vans. Program Comprehension During Software Maintenance and Evolution. *Computer*, 28:44–55, 1995. (Zitiert auf den Seiten 37 und 38)

[Nøroo] K. Nørmark. Elucidative programming. *Nordic J. of Computing*, 7:87–105, 2000. (Zitiert auf den Seiten 7 und 21)

[Obj] Object Management Group. Unified Modeling Language. URL http://www.uml.org/. (Zitiert auf Seite 18)

[Oraa] Oracle. Javadoc – The Java API Documentation Generator. URL http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javadoc.html. (Zitiert auf Seite 49)

[Orab] Oracle. Javadoc 5.0 Tool. URL http://docs.oracle.com/javase/1.5.0/docs/guide/javadoc/index.html. (Zitiert auf den Seiten 11, 23 und 48)

[Orac] Oracle. Proposed Javadoc Tags. URL http://java.sun.com/j2se/javadoc/proposed-tags.html. (Zitiert auf Seite 51)

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15:1053–1058, 1972. (Zitiert auf den Seiten 14 und 24)

[Par94] D. L. Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pp. 279–287. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994. (Zitiert auf Seite 9)

[Pen87] N. Pennington. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, 19:395–341, 1987. (Zitiert auf Seite 38)

[Ras03] J. Raskin. The Woes of IDEs. *Queue*, 1:8–11, 2003. (Zitiert auf Seite 35)

[SC94] S. Shum, C. Cook. Using literate programming to teach good programming practices. *SIGCSE Bull.*, 26:66–70, 1994. (Zitiert auf Seite 21)

[SH09] G. Starke, P. Hruschka. *Software-Architektur kompakt – angemessen und zielorientiert*. Spektrum Akademischer Verlag Heidelberg, 2009. (Zitiert auf den Seiten 8, 31, 32, 57 und 69)

[Shn98] B. Shneiderman. *Designing the User Interface – Strategies for Effective Human-Computer-Interaction*. Addison-Wesley Longman Inc., 3 Auflage, 1998. (Zitiert auf Seite 37)

[SLVA10] J. Singer, T. Lethbridge, N. Vinson, N. Anquetil. An examination of software engineering work practices. In *CASCON First Decade High Impact Papers*, CASCON '10, pp. 174–188. ACM, New York, NY, USA, 2010. (Zitiert auf den Seiten 39 und 40)

[Szy99] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing / ACM Press, 1999. (Zitiert auf Seite 30)

[Thea] The Eclipse Foundation. Eclipse. URL http://www.eclipse.org. (Zitiert auf den Seiten 29 und 63)

[Theb] The Eclipse Foundation. Eclipse Delta Pack. URL http://www.eclipse.org/downloads/download.php?file=/eclipse/downloads/drops/R-3.7.1-201109091335/eclipse-3.7.1-delta-pack.zip. (Zitiert auf Seite 63)

[too] toolsfactory software inc. Doc-O-Matic. URL http://www.doc-o-matic.de/. (Zitiert auf Seite 25)

[Wer23] M. Wertheimer. Laws of organization in perceptual forms (Untersuchungen zur Lehre von der Gestalt). *Psychologische Forschung*, 4:301–350, 1923. (Zitiert auf Seite 37)

[Wik] Wikipedia. Comparison of documentation generators. URL http://en.wikipedia.org/wiki/Comparison_of_documentation_generators. (Zitiert auf Seite 22)

[Wol] T. Wolf. AdaBrowse. URL http://home.datacomm.ch/t_wolf/tw/ada95/adabrowse/index.html. (Zitiert auf den Seiten 11 und 24)

[YAM99] S. Yacoub, H. Ammar, A. Mili. Characterizing a Software Component. In *In Proceedings of the 2nd Workshop on Component-Based Software Engineering, in conjunction with ICSE'99*. 1999. (Zitiert auf den Seiten 7, 30 und 31)

Alle URLs wurden zuletzt am 25.04.2012 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Michael Kircher)