Institut für Softwaretechnologie
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 24

Verbesserung eines Dokumentationswerkzeugs für Java-Pakete

Tobias Kuhn

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Jochen Ludewig
Betreuer:	Dipl.-Inf. Ivan Bogicevic
begonnen am:	01. Mai 2012
beendet am:	31. Oktober 2012
CR-Klassifikation:	D.2.2, D.2.7

Zusammenfassung

Java-Anwendungen werden in Module, im Java-Kontext „Pakete“ genannt, unterteilt. Dabei war es allerdings bislang eher umständlich Module einzeln zu dokumentieren. Aus diesem Grund wurde in einer Diplomarbeit das Werkzeug J-PaD – Java Package Documenter – entwickelt, welches direkt in der Entwicklungsumgebung Eclipse integriert ist. Es ermöglicht die gezielte integrierte Dokumentation von Java-Paketen in einer anpassbaren Benutzeroberfläche mit Hilfe eines Plugin-Systems für die darzustellenden Eingabefelder.

In dieser Bachelorarbeit wird die Software J-PaD zunächst sowohl auf vorhandene Mängel als auch auf Möglichkeiten für Erweiterungen untersucht. Hierfür wird ein brauchbares Vorgehen entwickelt, um möglichst viele Schwachstellen aufdecken zu können. Um die entstehende Befundliste zu priorisieren, wird auch ein Bewertungsschema definiert, welches die Befunde auf verschiedene Arten ordnet.

Aufbauend auf der priorisierten Befundliste werden dann konkrete Verbesserungen ausgesucht, deren Umsetzung im Kontext möglich und sinnvoll erscheint. Diese Verbesserungen und Erweiterungen werden dann an J-PaD umgesetzt und die entstehenden Änderungen dokumentiert.

Abstract

Java applications are divided in modules. These modules are called “packages“ in Java. Although it was rather cumbersome to document the modules one by one so far. For this reason the tool J-PaD – Java Package Documenter – was developed in a diploma thesis. J-PaD is directly integrated in the development environment Eclipse and allows for the systematic integrated documentation of Java packages in a flexible user interface with the help of a plugin system for the input fields to be displayed.

In this bachelor thesis the software J-PaD initially is analyzed both for existing defects and for possibilities for extensions. For this purpose a suitable procedure is developed in order to be able to expose as much flaws as possible. In order to prioritize the emerging list of findings an assessment scheme is defined which sorts the findings in different ways.

Based on the prioritized list of findings concrete improvements are chosen so that their realization seems possible and reasonable. These improvements and extensions for J-PaD are then implemented and the resulting changes are documented.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

1.1. Aufgabe

Heutzutage haben Software-Systeme typischerweise eine derartige Komplexität und einen derartigen Umfang erreicht, dass es nicht möglich wäre, sie als einen großen, monolithischen Block zu warten. Daher wird der Quellcode der Software üblicherweise in einzelne, kleinere Module zerlegt. Die Sicht auf den Code lässt sich mit der Sicht auf die Dokumentation vergleichen. Aus einer großen, unüberschaubaren Spezifikation müssten also auch viele einzelne Modulspezifikationen entstehen. Dies ist aber in der Praxis nicht der Fall. Ein Grund hierfür ist sicherlich der Mangel an Werkzeugen, die auch die Idee näher an die Entwickler bringen würden.

Um diesen Ansatz weiter zu verfolgen, wurde der „Java Package Documenter“ J-PaD im Rahmen einer Diplomarbeit erstellt. J-PaD gliedert sich nahtlos als Plugin in Eclipse ein. Dabei entsteht eine integrierte Dokumentation, die sehr nahe bei der Codebearbeitung verfügbar ist. Mit J-PaD können Pakete – Module in Java – dokumentiert werden. Dabei ist die Dokumentationsmaske frei konfigurier- und erweiterbar. Sie wird einerseits dynamisch mittels einer Konfigurations-XML erzeugt, andererseits stellt J-PaD selbst ein Pluginsystem dar, das beliebige Eingabekomponenten für die Dokumentation, sogenannte Widgets, nachladen kann.

Die Bachelorarbeit beinhaltet zwei Teilaspekte. Zunächst umfasst der erste Teil der Bachelorarbeit die Analyse des Werkzeugs J-PaD. Dabei liegt das Augenmerk einerseits darauf, welche Anwendungsfälle es gibt und aus welchen Schritten diese bestehen; andererseits werden auch Defizite des Werkzeugs hervorgehoben. Hierfür wird ebenso die Produktqualität analysiert. Ergebnis dieser Arbeit ist eine priorisierte Liste der Mängel und der fehlenden Funktionalität.

Der zweite Teil der Bachelorarbeit behebt ausgewählte, konkrete Defizite. Dies wird durch die Umsetzung von neuer Funktionalität und durch die Verbesserung der bestehenden Implementierung erreicht. Die konkreten Details, wie Umfang und Inhalt der Verbesserung, ergeben sich aus den Ergebnissen der ersten Teils.

Die einzelnen Schritte der Arbeit und die Ergebnisse werden in einem Zwischenvortrag und einer Abschlusspräsentation vorgestellt. Diese Ausarbeitung dokumentiert die Bachelorarbeit.

1.2. Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen erklärt die Ausgangslage der Bachelorarbeit und fasst die notwendigen Grundbegriffe zusammen, die nachfolgend verwendet werden.

Kapitel 3 – Analyse & Bewertung verwendet die Grundlagen, um J-PaD auf Erweiterungspotential und Mängel zu analysieren.

Kapitel 4 – Fokus wählt einzelne Entwicklungsaufgaben aus, deren Umsetzung in dieser Bachelorarbeit angemessen erscheint.

Kapitel 5 – Umsetzung beschreibt die Resultate der Umsetzung zuvor ausgewählten Entwicklungsaufgaben.

Kapitel 6 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2. Grundlagen

Diese Arbeit baut notwendigerweise auf die Diplomarbeit von Michael Kircher [Kir12] auf, in der sich die Grundlagen von J-PaD finden. Alle direkt relevanten Basisinformationen für diese Arbeit finden sich in diesem Kapitel.

2.1. Begriffe

Modul In modernen Programmiersprachen werden Programme und deren Quellcode in einzelne Komponenten bzw. Module zerlegt. Bedingt durch den Aufbau auf J-PaD werden Module im Sinne dieser Arbeit durch Java-Pakete dargestellt.

Usability Die ISO-Norm 9241 definiert Usability als „Extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use“ [ISO98]. Wörtlich übersetzt handelt es sich also um den Grad, zu dem bestimmte Benutzer bestimmte Ziele in einem bestimmten Nutzungskontext mit Effektivität und Effizienz zufriedenstellend erreichen können. Umformuliert könnte man salopp sagen: Benutzer sollen in einer bestimmten Situation ihre Ziele einfach, schnell und zufriedenstellend erreichen können.

Das für die Benutzung von Software zentrale Konzept „Usability“ lässt sich auch als Benutzbarkeit übersetzen, wobei dieser Begriff gerne zu eng interpretiert wird [LL10]. Für die Verwendung des griffigeren englischen Begriffs „Usability“ spricht, dass er auch im deutschen Sprachgebrauch etabliert ist [Rö12].

Befund Alle festgestellten Mängel der Analyse werden als „Befund“ bezeichnet. Dies hat den Vorteil, dass sie sich durchgängig nummerieren lassen. Eine Klasse von Befunden wird dabei aufsteigend mit arabischen Ziffern nummeriert. Ein Befund enthält dabei nicht nur die Beschreibung, sondern auch eine geplante Lösungsstrategie zur Behebung des Mangels.

Konfiguration, Schema Die Einstellung der Dokumentationsmaske in J-PaD nennt sich Konfiguration, Konfigurationsschema oder einfach kurz Schema. Dabei handelt es sich um die Benennung, Auswahl und Gruppierung der Eingabekomponenten, z. B. für „Name:“, „Autor:“, „Datum:“, usw.

2.2. Ziel

Eine Verbesserung bedeutet dem Wortsinn nach ein „*Besser*machen“; es handelt sich also um eine gezielte Aktion, deren Ergebnis die Erhöhung einer Qualität darstellt. In der ISO-9000-Norm wird Qualität als „degree to which a set of inherent characteristics fulfils requirements“ [ISO05] definiert. Wörtlich übersetzt ist eine Qualität also der Grad, zu dem eine Menge an inhärenten Eigenschaften Anforderungen erfüllt. Daraus folgt, dass die Eigenschaften objektivierbar sein müssen. Ansonsten wären sie mit hoher Sicherheit willkürlich zugewiesen statt inhärent, oder anders gesagt: Sie würden nicht dem zu bewertenden Objekt innewohnen. Weiterhin beziehen sich Anforderungen nicht nur auf Kundenanforderungen. Stattdessen hat jede am Projekt beteiligte Rolle eigene Anforderungen, wie z. B. der Wartungsingenieur oder der tatsächliche Benutzer.

Implizit ist mit jeder Verbesserung also eine Utopie verbunden. Dies ist der Zustand, in dem keine weitere Verbesserung durchgeführt werden könnte, da alle Qualitäten maximal sind. Das bedeutet wiederum, dass alle objektivierbaren Eigenschaften alle an sie gestellten Anforderungen erfüllen. Wendet man diese Idee auf das konkret zu verbessernde Objekt an, erhält man eine Vision. Obgleich diese praktisch unerreichbar ist, lässt sich der aktuelle Zustand hieran messen und der weitere Weg in Richtung der Vision planen. Angewandt auf J-PaD ergibt sich beispielsweise diese Vision:

(a) Die Software J-PaD ist die perfekte Lösung zur Dokumentation von Java-Paketen in der Eclipse-IDE. J-PaD kann vom Benutzer in minimaler Zeit für seine Bedürfnisse konfiguriert und anschließend zum effizienten Lesen und Schreiben von Paket-Dokumentationen benutzt werden.

(b) Da J-PaD keine Fehler enthält, fällt es nie aus und sorgt auch nicht für Ausfälle von Eclipse. Mit J-PaD ist ein unabsichtlicher Datenverlust genauso ausgeschlossen wie böswillige Einflüsse wie Daten-Manipulation, Spionage oder Verletzung von Persönlichkeitsrechten. J-PaD verstößt gegen keine Gesetze und hält sich an alle anwendbaren Richtlinien.

(c) Die Einführung von J-PaD ist so einfach möglich, dass prinzipiell nur die Installation nötig ist. J-PaD wurde gleichzeitig so gestaltet, dass selbst unerfahrene Anwender die Oberfläche intuitiv bedienen und die Ausgaben verstehen können. J-PaD verhält sich ohne Ausnahme in ähnlichen Fällen konsistent.

(d) J-PaD enthält selbst alle hilfreichen Dokumentationsdokumente, sowohl für Anwender als auch für Entwickler. Die Spezifikation ist zutreffend, übersichtlich und konform mit einer verbreiteten Norm. Das Handbuch von J-PaD erklärt alle tatsächlich vorhandenen Funktionalitäten und denkbaren Arbeitsflüsse.

(e) J-PaD benötigt die minimal notwendige Rechenzeit und den minimal notwendigen Speicherplatz, sodass auf allen gängigen Rechnern mit sofortiger Antwortzeit zu rechnen ist. J-PaD wurde so entworfen, dass es nur Module

mit minimaler Kopplung und maximalem Zusammenhalt gibt. Kein Modul übt eine Fernwirkung aus, es gibt keine unnötigen Redundanzen. Ebenso enthält der Code keine unnötigen Redundanzen, ist vollständig und für Anfänger verständlich kommentiert. Dabei werden alle Spezifikationsanforderungen vollständig bidirektional im Code referenziert, was eine Nachverfolgbarkeit ermöglicht. Auch der Quellcode ist vollkommen leserlich und verständlich gehalten. Alle Bezeichner sind zutreffend, verständlich und ästhetisch ansprechend gewählt. Sowohl die Architektur als auch der Code ist mit minimalem Aufwand erweiterbar. Weiterhin ist der Code absolut fehlerfrei, besitzt keine Geräteabhängigkeit und reagiert auf jede mögliche Fehlersituation angemessen.

(f) Die Spezifikation enthält nur die tatsächlichen Anforderungen aller Stakeholder, jeweils mit Bezug auf den Ansprechpartner. Die Spezifikation und J-PaD selbst erfüllen alle Anforderungen, die die Stakeholder an sie stellen.

Die Beschreibung wurde dabei in sechs Teile gegliedert: (a) Funktionalität, (b) nicht-funktionale Aspekte, (c) Usability, (d) Dokumentation, (e) Architektur und Code und (f) Umsetzung aller Anforderungen. Die (f) Umsetzung aller Anforderungen bezieht sich hauptsächlich auf Aspekte, die noch nicht in der (a) Funktionalität oder den (b) nicht-funktionalen Aspekten geklärt werden. Daher sind diese nicht deckungsgleich.

2.3. Vorgehen

Es muss zunächst ein Vorgehen entwickelt werden, um alle sinnvollen und möglichen Verbesserungen zu identifizieren und zu bewerten. Erst wenn man alle Ansatzpunkte kennt, können geeignete Befunde ausgewählt und J-PaD tatsächlich verbessert werden. Das Vorgehen ist dabei so gestaltet, dass es leicht auf ähnliche Projekte übertragbar sein dürfte.

Erstes Ziel ist dabei – auch aus der Aufgabenstellung heraus – zunächst eine Erfassung des Ist-Zustands. Hier ist gefordert, die gängigsten Anwendungsfälle zu dokumentieren und die Softwarequalität detailliert zu analysieren. Somit ergibt sich direkt die Vorgabe, dass Anwendungsfälle eingesetzt werden müssen. Diese sollen den bereits vorhandenen Teil der (a) Funktionalität abdecken. Dabei verbessert ihre Modellierung auch den Zustand der (e) Dokumentation.

Die Produktqualität einer Software besteht aus vielen Teilqualitäten; diese werden üblicherweise in einem Qualitätsstandard in einer hierarchischen Struktur aufgelistet, wie z. B. in der ISO-25010-Norm [ISO11]. Ebenfalls in Frage kamen der ISO-9126-Standard [ISO01] oder eine akademische Definition. Der ISO-9126-Standard scheidet allerdings aus, da er als veraltet gilt und durch den ISO-25010-Standard abgelöst wurde. Hinzu kommt, dass der ISO-9126-Standard auch nur auf einem akademischem Modell [BBK+78] basiert [Dro95]. Letztendlich ähneln sich alle Standards größtenteils, sodass sich eine akademische Definition anbietet. Diese sollte einerseits als Publikation verfügbar, andererseits auch nicht vollkommen aus der Luft gegriffen sein. Dabei wird eine menschliche Bewertung einer automatischen

Metrikensammlung gegenüber bevorzugt. Der Qualitätenbaum aus [LL10] ist nicht nur für den qualitativen Einsatz brauchbar formuliert, er wird auch im Studiengang Softwaretechnik an der Universität Stuttgart gelehrt. Dieser bewertet vor allem (e) Architektur und Code, (b) nicht-funktionale Aspekte und (c) Usability. Darüber hinaus wird allerdings auch (a) Funktionalität und (d) Dokumentation erfasst, also fast alle Aspekte der Vision.

Neben dem qualitativen soll auch ein quantitatives Vorgehen verwendet werden, um (e) Architektur und Code zu analysieren. Ein verbreiteter Ansatz ist die statische Code-Analyse mit Hilfe von Compiler und gebräuchlichen Analysewerkzeugen.

Um den Ist-Zustand zu dokumentieren, ist es notwendig, dass nicht nur der Code und die Architektur näher betrachtet werden; besonders die Ausführung steht im Mittelpunkt. J-PaD muss ausgiebig in den Rollen der Akteure benutzt werden, beispielsweise um die Anwendungsfälle zu dokumentieren. Das Gleiche gilt, wenn die Bedienbarkeit aus der Produktqualität bewertet wird.

Der zweite Teil stellt eher die Analyse des Soll-Zustands dar und deckt die (f) Umsetzung aller Anforderungen aus der Vision ab. Hier verlangt die Aufgabenstellung lediglich, dass neben Mängeln auch „noch fehlende Funktionen“ aufgelistet werden. Hierzu werden offene Entwicklungsansätze aus der Diplomarbeit [Kir12] übernommen und weitere Anforderungen des Kunden gesammelt. Dazu kommen eigene Ideen, die während der gesamten Benutzung von J-PaD erfasst werden. Damit werden die Ansätze aller zuvor an J-PaD beteiligten Entwickler und Benutzer einbezogen. Es erscheint nicht sinnvoll, mögliche zusätzliche Anforderungen zukünftiger Benutzer und Entwickler zu erfassen. Dies begründet sich einfach durch den entstehenden Aufwand, welcher im Rahmen dieser Arbeit nicht machbar anmutet.

Ein zentraler Wunsch des Kunden war auch die Umsetzung einer guten (c) Usability. Aus Entwicklersicht am aufwändigsten sind dabei die Aspekte der Usability, die auch die (a) Funktionalität betreffen. Idealerweise sollen diese nicht nur in (e) Architektur und Code, sondern auch sauber für die (d) Dokumentation spezifiziert werden. Es bietet sich somit an, sogenannte Usability Elicitation Patterns – zu deutsch etwa Usability-Erhebungsmuster – aus [JMSSD07] auf die Anwendungsfälle anzuwenden. Dies ist insbesondere lohnenswert, da die Anwendungsfälle bereits erfasst werden. Besonders hilfreich, um diese praktisch anzuwenden, ist die Weiterentwicklung und Katalogisierung solcher Patterns durch Holger Röder [Rö12].

Natürlich soll dieses Vorgehen nicht als abschließend oder vollständig verstanden werden. Es handelt sich viel mehr um ein zweckgewidmetes Vorgehen, das möglichst viele Aspekte aus dem in Abschnitt 2.2 dargestellten Ziel ansprechen soll. Die Hoffnung ist hierbei, dass das Ziel eher erreicht wird, wenn das Vorgehen sich auf die konkreten Aspekte der Vision konzentriert. Darüber hinaus werden die entwickelten Ansätze und die resultierenden Befunde in einem Zwischenvortrag präsentiert und zur Diskussion gestellt.

Hauptsächlich wird – auch weil vom Kunden so gewünscht – in der Analyse J-PaD betrachtet; die Drittkomponenten, wie der RichText-Editor, waren im Detail eher nebensächlich. Daher stellen die vorgeschlagenen Änderungen eher J-PaD in den Mittelpunkt. Alle Änderungen basieren jeweils auf einem oder mehreren der nachfolgenden Konzepte:

- Dokumentation der denkbaren Anwendungsfälle
- Analyse der Produktqualität mit dem Qualitätenbaum aus [LL10]
- Statische Analyse des Codes mit Werkzeugen
- Analyse der Usability mit dem Usability-Pattern-Katalog aus [Rö12]
- Einbringen eigener Ideen
- Übernahme von gezielten Anforderungen des Kunden und Ideen für die Weiterentwicklung aus [Kir12]

Die Punkte, die ein gewisses Hintergrundverständnis benötigen, werden dabei im Folgenden erläutert.

2.3.1. Nummerierung

Ein konkreter Vorschlag für eine Verbesserung oder ein Fehlerbefund wird in dieser Arbeit durchgängig nach dem Schema „Befundkategorie“ „Fortlaufende Zahl“ durchnummeriert. Dies sieht dann am Beispiel wie folgt aus:

(B1) Der Programmcode sollte im Schüttelreim verfasst werden, da dies die künstlerische Qualität hebt und den Leser erheitert.

Im weiteren Verlauf lässt sich dann beispielsweise sagen, dass die Motivation hinter Befund 1 gut gemeint, die praktische Relevanz jedoch nicht gegeben ist.

Befund 1 Nutzen: *Niedrig* Umfang: *Hoch*

2.3.2. Anwendungsfälle

Anwendungsfälle sind ein wichtiges Mittel, um funktionale Anforderungen an das Verhalten einer Software festzuhalten. Anwendungsfälle dienen dabei nicht nur dem Entwickler, sondern fast allen an der Entwicklung beteiligten Rollen. So kann z. B. der Tester das gewünschte Verhalten der Software beim Test daraus ableiten. [BS03] Die zentralen Anwendungsfälle von J-PaD sollen festgehalten werden. Dabei umfasst ein Anwendungsfall nicht nur einen Ablauf von Schritten, sondern auch eine Menge an Akteuren, ein bestimmtes Ziel, eine Vor- und Nachbedingung und festgehaltene Alternativabläufe, wie beispielsweise in [LL10] beschrieben.

2.3.3. Qualitätenbaum

Der Qualitätenbaum ist eine in [LL10] vorgeschlagene hierarchische Struktur, um die Gesamt-Qualität einer Software anhand diverser Teilaspekte zu bewerten. Er ist eine praktisch motivierte Wahl aus allen ähnlichen Qualitätsstandards, wie z. B. den ISO-Normen 9126 [ISO01] und 25010 [ISO11]. Dabei wird in dieser Arbeit nur auf die Produktqualität eingegangen. Der Prozess im universitären Umfeld hat sehr wechselhafte Züge, wie z. B. wechselnde Bearbeiter, die nichts voneinander wissen. Diese sind meist auch nur Einzelkämpfer in einer Abschlussarbeit, sodass auch die nötigen Freiräume zur Veränderung fehlen.

2.3.4. Analyse-Werkzeuge

Bei der Analyse des Quellcodes wurden verbreitete Werkzeuge benutzt:

- Der Java-Compiler selbst. Wird dieser passend konfiguriert, ist er bereits in der Lage diverse „Code Smells“ zu entdecken, z. B. potentielle `null`-Dereferenzierungen oder toten Code. Dazu kommen einige Überprüfungen, die die Einhaltung von Style-Guides überprüfen können.
- FindBugs [Uni], ein Eclipse-Plugin. Dieses ist darauf spezialisiert, typische Fehlermuster in Java-Programmen zu identifizieren, z. B. Fließpunkt-Vergleiche auf Gleichheit oder Probleme bei der Verwendung der bei Java mitgelieferten Bibliotheken.
- Geplant war auch ein Einsatz von Quamoco [Qua]. Allerdings zeigten sich Probleme auf unterschiedlichen Plattformen. Auch der Umfang der notwendigen Konfiguration erschien erheblich. Zudem funktionierte die Rückverfolgung von Qualitätsmerkmalen in Codebestandteile anscheinend nicht korrekt oder wäre nur sehr umständlich möglich gewesen, weswegen auf den Einsatz insgesamt verzichtet wurde.
- Zuletzt wurde noch das Eclipse-Plugin CodePro Analytix [Goo] eingesetzt. Hauptsächlich – wie der Name vermuten lässt – dient es zur Codeanalyse durch Berechnung von Metriken und zur Suche nach dupliziertem und totem Code.

2.3.5. Usability Patterns

Anwendungsfälle bieten den zusätzlichen Vorteil, dass sie mit Usability Patterns annotiert werden können, wie in [Rö12] beschrieben. Dabei kann mit dem Werkzeug „Tulip“ eine Auswahl von sinnvollen Usability Features getroffen werden, deren konkrete Ausprägung in den Anwendungsfällen dann detailliert spezifiziert werden kann. Es handelt sich dabei nicht um Design-Aspekte der grafischen Benutzeroberfläche, sondern um konkrete Funktionalität, die die Usability verbessern kann.

2.4. Bewertung

Da es weder zeitlich möglich noch effizient ist, alle erkannten Verbesserungen umzusetzen, wird jede Verbesserung mit drei Kategorien bewertet. Deren Ausprägung kann entweder *Niedrig*, *Mittel* oder *Hoch* aufweisen. Anhand dieser Einschätzung kann in Kapitel 4 eine Auswahl getroffen werden, bei welchen Verbesserungen eine Implementierung im Rahmen dieser Arbeit sinnvoll wäre.

2.4.1. Nutzen

Der Nutzen eines Befundes ist hoch, wenn durch die Umsetzung viel gewünschte Funktionalität erreicht oder viele schwere Fehler behoben werden. Allerdings muss beachtet werden, dass die Behebung eines Befundes nicht nur den Benutzern, sondern auch den Entwicklern nutzen kann.

Der Nutzen lässt sich nur schwer quantifizieren. Daher richtet sich die Einteilung einerseits nach der Einschätzung, wie häufig Benutzer oder Entwickler die Verbesserung bemerken werden. Andererseits spielt auch die Schwere der behobenen Einschränkung eine Rolle.

Eine Irritation liegt vor, wenn der Benutzer durch kleine Unstimmigkeiten, unnötige Schritte, fehlende Zusammenhänge oder kleinere Fehler behindert wird. Die gewünschte Funktionalität wird aber nach wie vor erbracht. Ein Ausfall liegt vor, wenn das System aufgrund des Fehlers eine Funktionalität gar nicht mehr oder nur in stark reduziertem Umfang erbringen kann. Ebenso liegt ein Ausfall vor, wenn ein Datenverlust eintritt. Der Totalausfall bedeutet schließlich, dass das gesamte System nicht mehr funktioniert, sondern „abstürzt“.

	Schwere		
Häufigkeit	Irritation	Ausfall	Totalausfall
eher selten	*Niedrig*	*Mittel*	*Hoch*
hin und wieder	*Niedrig*	*Mittel*	*Hoch*
meistens	*Mittel*	*Hoch*	*Hoch*

Tabelle 2.1.: Bestimmung der Bewertung des Nutzens eines Befundes

Hintergrund grade dieser Zuweisung ist zunächst die Idee, im Fall „hin und wieder“ die drei Schweregrade direkt auf die drei Ausprägungen des Nutzens abzubilden. Anschließend werden häufigere Vorfälle eine Stufe höher gewichtet, seltenere Vorfälle eine Stufe niedriger. Dabei kann es keine niedrigere Stufe als *Niedrig* und keine höhere Stufe als *Hoch* geben. Allerdings wurden zusätzlich die Bedingungen aufgenommen, dass die Behebung eines Ausfalls nie unnützlicher als *Mittel* und die Behebung eines Totalausfalls nie unnützlicher als *Hoch* bewertet werden darf.

2.4.2. Umfang

Der Umfang eines Befundes ist hoch, wenn die Umsetzung vermutlich viel Zeit in Anspruch nimmt. Dies kann daran liegen, dass sie besonders umfangreich oder besonders komplex ist.

Die Einteilung richtet sich daher nach der geschätzten Dauer, die für die Umsetzung im Rahmen dieser Arbeit vermutlich benötigt würde:

Ausprägung	*Niedrig*	*Mittel*	*Hoch*
geschätzter Aufwand	$\approx$ 1 Tag	$\leq$ 1 Woche	$>$ 1 Woche

Tabelle 2.2.: Bestimmung der Bewertung des Umfangs eines Befundes

2.4.3. Kundenansicht

Nachdem alle Befunde identifiziert wurden, wurde der Kunde zu seiner Einschätzung befragt, welche Befunde er für wichtig erachtet. Dabei hat er alle Befunde in einer dreistufigen Skala (*Wichtig*, *Mittel*, *Unwichtig*) bewertet. Aus dieser Bewertung lässt sich der tatsächliche Bedarf des Kunden abschätzen. Damit kann vermieden werden, dass die Umsetzung eines wichtig erscheinenden, umfangreichen Befundes dem Kunden konkret eigentlich gar keine Vorteile bringt.

3. Analyse & Bewertung

3.1. Anwendungsfälle

Basierend auf dem Anforderungskapitel aus [Kir12] wurden die gebräuchlichsten Anwendungsfälle aus Benutzersicht modelliert, um die Qualitätsmerkmale gezielt auf diese Abläufe hin analysieren zu können. Die Anwendungsfälle können dabei mit einer dreistelligen Zahl identifiziert werden. Das Schema entspricht dabei „Nummer der Kategorie“ „zweistellige Durchnummerierung in dieser Kategorie“, also z. B. 101. Die Anwendungsfälle wurden dabei in zwei Kategorien einsortiert:

1. die Produktion von Konfiguration und Dokumentation (1xx)
2. die Nutzung von Dokumentation (2xx)

Abläufe von Anwendungsfällen werden dabei als Flussdiagramme, wie beispielsweise in Abbildung 3.2, dargestellt. Blaue, runde Kästchen entsprechen dabei Vor- bzw. Nachbedingungen. Grüne, rechteckige Kästchen entsprechen den einzelnen Schritten des Anwendungsfalls. Tritt ein Sonderfall oder Alternativablauf auf, so verzweigt dieser von einer Raute. Die Bedingung für die Verzweigung ist dabei auf dem abgehenden Pfeil notiert.

3.1.1. Akteure

Konfigurateur

Der Konfigurateur trägt die Verantwortung dafür, das Dokumentationsschema zu erstellen, nach dem dokumentiert werden soll. J-PaD bringt zwar eine verwendbare Universalkonfiguration mit, allerdings lässt sich diese auf die individuellen Bedürfnisse anpassen. Hierfür ist Planung und Erfahrung hilfreich, da hier unnötige Einschränkungen zur Verschlechterung der Dokumentationsqualität und -effizienz führen können, beispielsweise wenn lediglich eine Zahl statt eines klärenden Kommentars angegeben wird oder umgekehrt. Typischerweise fällt diese eher autoritäre Rolle eher den Architekten, den Projektleitern oder den Qualitätssicherungsingenieuren zu.

Dokumentierer

Der hauptsächliche Benutzer, der einfache Entwickler, tritt zunächst in der Rolle des Dokumentierers auf: Er benutzt J-PaD mit einem Dokumentationsschema, um seine entwickelten Java-Pakete damit zu dokumentieren. Allerdings kann auch beispielsweise ein Architekt zunächst die Pakete ansatzweise dokumentieren, um so den Entwicklern ein klareres Bild über ihre konkrete Aufgabe im Code zu verschaffen. Ebenso kann auch ein Wartungsingenieur seine Änderungen an einem Paket in der Dokumentation notieren.

Leser

Die wichtigste Rolle bekleidet wieder hauptsächlich der einfache Entwickler. Die mit J-PaD geschriebene Dokumentation dient natürlich einem Zweck: Sie soll bei Bedarf gelesen werden können. Dies kann fast immer bei Verständnisproblemen der Fall sein, z. B. wenn ein Wartungsingenieur den Inhalt eines Paketes oder vorhergehende Wartungen nachvollziehen möchte. Es gibt aber auch reguläre Aufgaben, bei denen eine Paketdokumentation helfen kann, z. B. bei der Erstellung von Testfällen.

3.1.2. Anwendungsfall-Diagramm

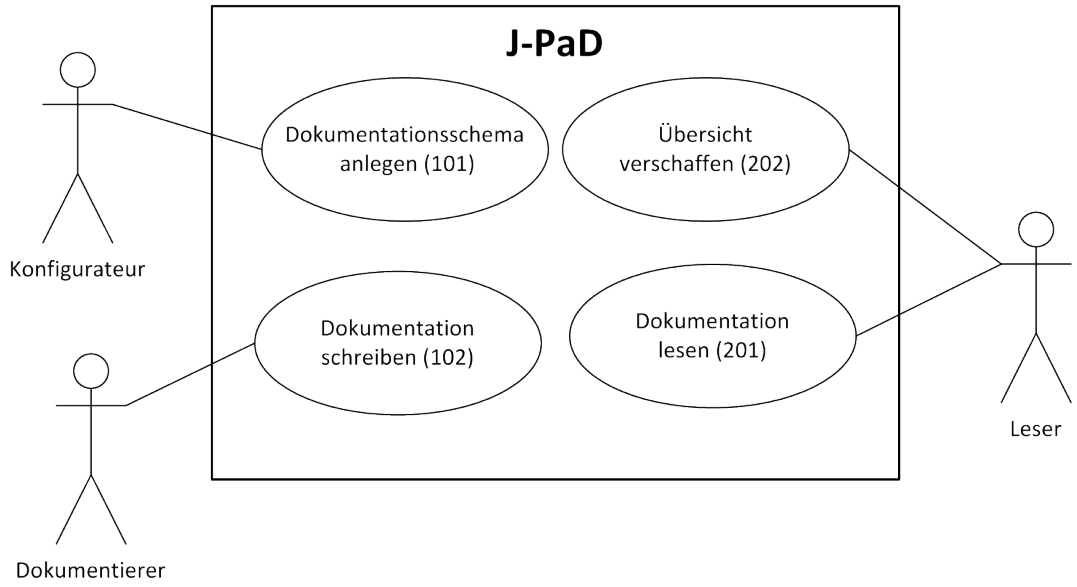

Abbildung 3.1.: Anwendungsfall-Diagramm für J-PaD

3.1.3. Dokumentationsschema anlegen (101)

Der Benutzer möchte ein neues Dokumentationsschema für ein Eclipse-Projekt anlegen. Dieser Anwendungsfall ist optional, da J-PaD ein vorkonfiguriertes Dokumentationsschema mitbringt, allerdings lässt sich J-PaD so an die individuellen Bedürfnisse konfigurieren. Wird er durchgeführt, so ist der Anwendungsfall der erste Schritt im Dokumentationsprozess mit J-PaD. Das dazugehörige Flussdiagramm findet sich in Abbildung 3.2.

Der Anwendungsfall ist zunächst so strukturiert, dass er sich auf das Hinzufügen und Konfigurieren eines einzelnen, neuen Elementes, d. h. Eingabe-Widgets, konzentriert. Sind die Daten von J-PaD übernommen, so gelangt der Anwendungsfall an eine Verzweigung, die es zunächst ermöglicht, erneut in den Normalablauf zu verzweigen, um weitere Elemente hinzuzufügen, da ein Dokumentationsschema praktisch immer aus mehreren Elementen besteht. Es besteht aber auch die Möglichkeit, einen Alternativablauf einzuschlagen, der dann das Ändern, das Verschieben (für die Darstellung in der späteren Dokumentationsmaske) und das Löschen von bestehenden Elementen ermöglicht.

Nach jedem Einzelschritt (Hinzufügen, Verschieben, usw.) kann erneut ein beliebiger anderer Einzelschritt durchlaufen werden. Für den Benutzer wird am Anfang nicht unbedingt die korrekte, minimale Bearbeitungsreihenfolge feststehen, wenn er nicht mit dem Arbeitsfluss vertraut ist oder mit seiner bisherigen Dokumentationsmaske unzufrieden ist und Verbesserungen vornehmen will.

Damit sind schließlich alle Elemente dem Benutzerwunsch nach erstellt. Anmerkend sei bemerkt, dass der Anwendungsfall auch für „Dokumentationsschema ändern“ verwendet werden kann, in diesem Fall kommt die zusätzliche Vorbedingung „Es gibt bereits ein Dokumentationsschema“ hinzu und Schritt 1 verzweigt nicht auf Schritt 2, sondern auf die erste, untere Verzweigung.

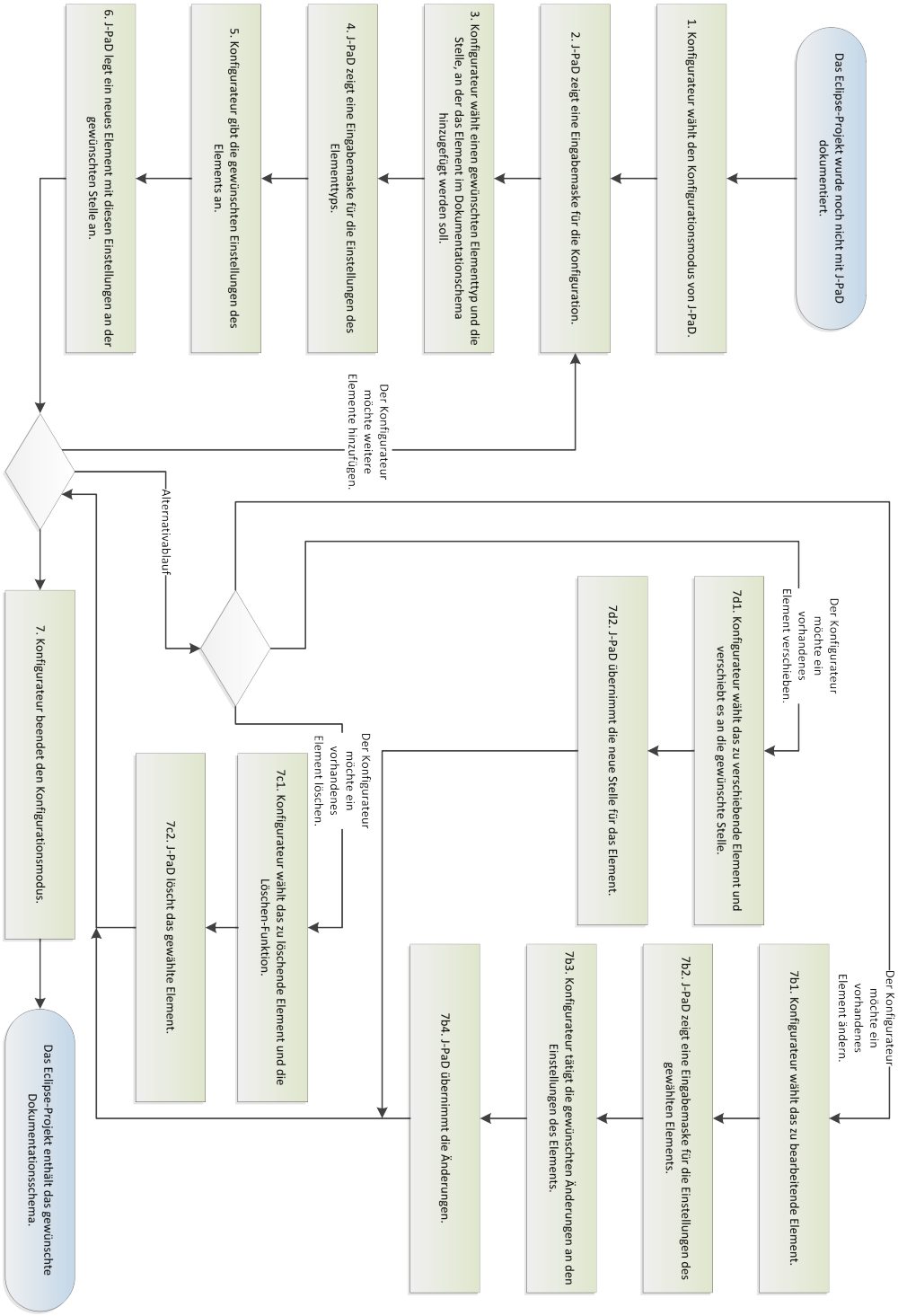

Abbildung 3.2.: Flussdiagramm für den Anwendungsfall „Dokumentationsschema anlegen“ (101)

3.1.4. Dokumentation schreiben (102)

Der Benutzer möchte mit einem vorhandenen Dokumentationsschema die Dokumentation anlegen, erweitern oder ändern. Hierbei handelt es sich um den ersten zentralen Schritt bei der Verwendung von Dokumentation im Allgemeinen: Jemand muss die Dokumentation erzeugen und aktuell halten. Das dazugehörige Flussdiagramm findet sich in Abbildung 3.3.

Dieser Anwendungsfall fällt eher einfach aus, was der (gewünschten) Simplizität und Intuitivität geschuldet ist.

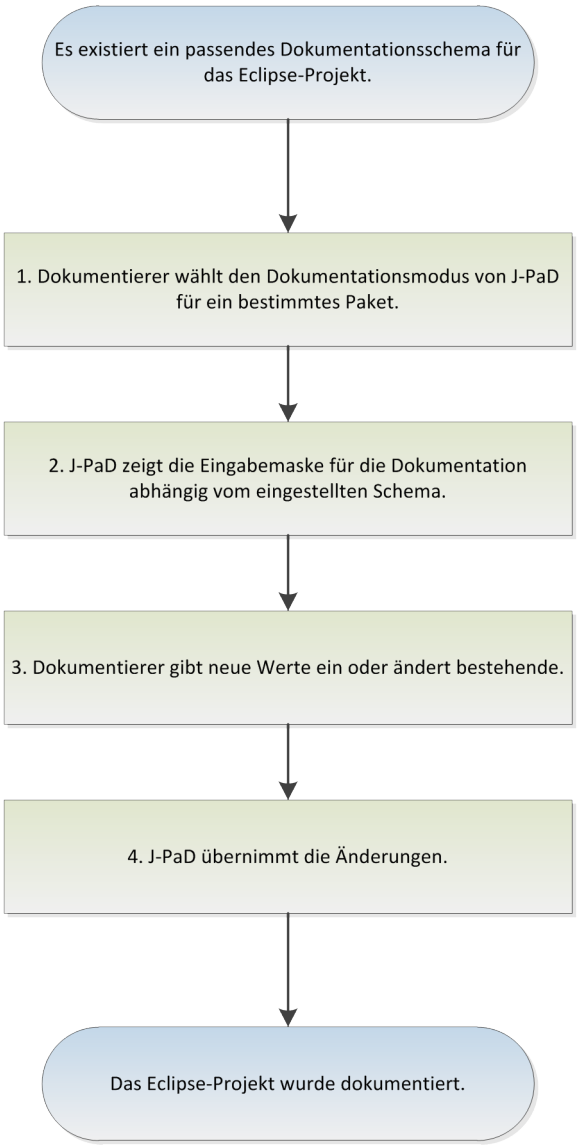

Abbildung 3.3.: Flussdiagramm für den Anwendungsfall „Dokumentation schreiben“ (102)

3.1.5. Dokumentation lesen (201)

Der Benutzer möchte für ein bestimmtes Paket die Dokumentation lesen. Dies ist der zweite zentrale Schritt bei der Verwendung von Dokumentation: Sie muss genutzt werden. Das dazugehörige Flussdiagramm findet sich in Abbildung 3.4.

Da es sich hierbei um den zentralen Aspekt in J-PaD handelt, fällt auch der Anwendungsfall denkbar knapp aus. Dem Benutzer soll es möglichst einfach sein, die Tätigkeit durchzuführen, die er am häufigsten mit J-PaD verrichtet.

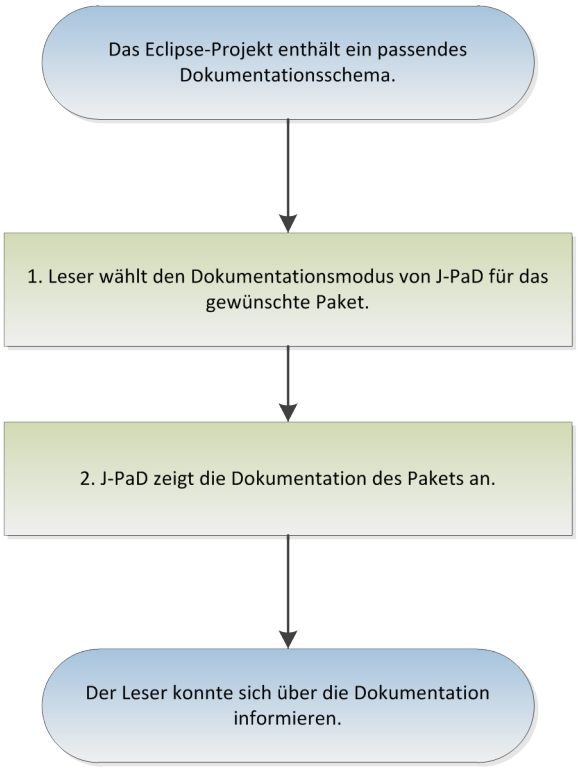

Abbildung 3.4.: Flussdiagramm für den Anwendungsfall „Dokumentation lesen“ (201)

3.1.6. Übersicht verschaffen (202)

Der Benutzer möchte sich einen Überblick über alle vorhandenen Paketdokumentationen verschaffen. In diesem Fall ist der Anwendungsfall keine Dokumentation des Ist-Zustandes, da er so nicht implementiert ist. Stattdessen handelt es sich eher um einen Vorschlag, um dem Benutzer zu ermöglichen, sich schneller in einer großen, unbekannten Architektur zurechtzufinden. Auch das Suchen nach bestimmten Paketen würde so vereinfacht. Das dazugehörige Flussdiagramm findet sich in Abbildung 3.5.

Im Prinzip handelt es sich um den Anwendungsfall „Dokumentation lesen“ (201), der allerdings um die Auswahl eines „Übersichtsmodus“ und eine Verzweigung für eine Schleife erweitert wurde. Gelangt der Ablauf zur Verzweigung, so kann er schnell ein weiteres Paket auswählen und betrachten, wenn das vorher betrachtete nicht das gesuchte war oder einen Querverweis enthalten hat.

Die Motivation für diesen Anwendungsfall findet sich in der Bedienung des „Package Explorers“ von Eclipse. Dieser stellt ein Projekt baumartig dar, allerdings ohne die Möglichkeit einen Unterbaum vollständig bis zu einer bestimmten Ebene zu entfalten. Stattdessen muss jedes (Unter-)Paket einzeln geöffnet werden und anschließend die Datei „package-info.java“ geöffnet werden. Da die ganzen Pakete auch die Klassen enthalten, werden in diesem Moment viele unerwünschte Informationen zusätzlich angezeigt. Dies kann sogar dazu führen, dass gescrollt werden muss, um ein anderes Paket zu betrachten.

Den Arbeitsablauf auf die Auswahl eines Pakets zu reduzieren würde dem Benutzer hier die ständige unnötige Arbeit ersparen, das Paket öffnen und die Datei auswählen zu müssen. Abgesehen von der Übersicht über alle Klassennamen, werden dem Benutzer keine neuen Informationen oder Wahlmöglichkeiten angeboten, weswegen dies ohne diesen Anwendungsfall eher als eine monotone Ausdauerübung wirken würde.

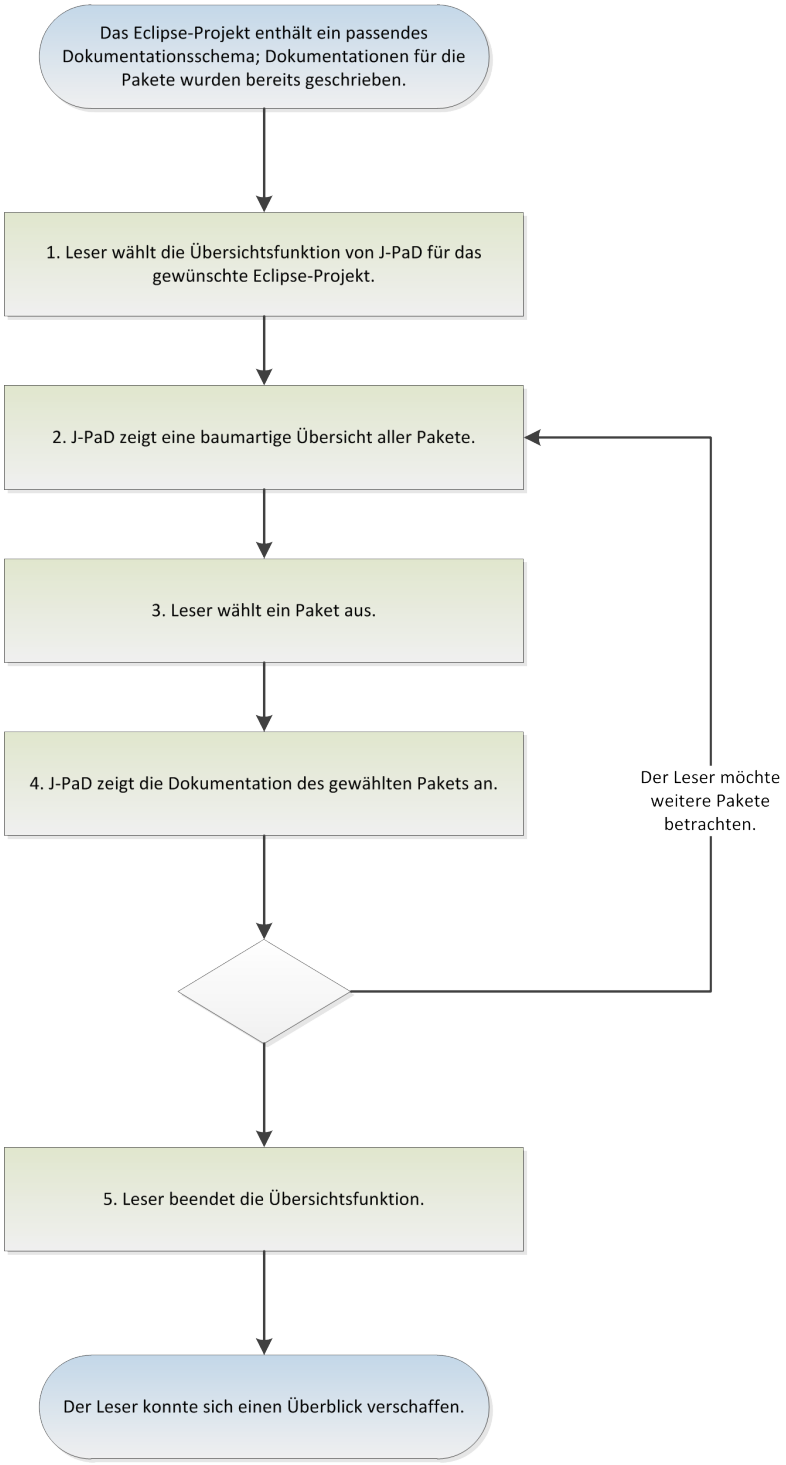

Abbildung 3.5.: Flussdiagramm für den Anwendungsfall „Übersicht verschaffen“ (202)

3.2. Qualitätenbaum

3.2.1. Wartbarkeit

Spezifikationsvollständigkeit

Die Spezifikation zu J-PaD findet sich hauptsächlich in [Kir12]. Allerdings handelt es sich hierbei nur um drei Seiten stichpunktartige Aufzählung, was lediglich die Grundzüge einer Spezifikation abbildet. Zudem ist die Tatsache, dass die Spezifikation nicht als Extra-Dokument sondern in einer wissenschaftlichen Publikation vorliegt, nachteilig für die Zugänglichkeit, Übersichtlichkeit und Aktualität.

(Q1) Verfassen einer ausführlichen Spezifikation

Es fehlt ein eigenständiges, zugängliches Spezifikationsdokument. Um einige Qualitäten abschließend zu bewerten und eine Soll-Basis für Tests zu haben, sollte eine umfangreiche Spezifikation verfasst werden. Dazu könnten auch die in Abschnitt 3.1 entwickelten Anwendungsfälle genutzt werden.

Eine Spezifikation für diesen Programmumfang mit Idealbedingungen als Einzelentwickler mit nur einem Kunden sollte sich innerhalb einer Woche schreiben lassen, da auch einige Teile bereits vorliegen. Insbesondere für die Wartbarkeit wäre eine zentrale Spezifikation eigentlich eine große Hilfe. Da der Programm- und Teamumfang sich jedoch in Grenzen hält und der Kunde einen softwaretechnischen Hintergrund hat, dürfte auch die Kommunikation eher leicht fallen. Daher wird der vermutlich gelegentliche Nutzen mittelmäßig eingestuft.

Qualitätenbaum-Befund 1 Nutzen: *Mittel* Umfang: *Mittel*

Lokalität

Schon beim Import der Projekte in Eclipse zeigt sich ein Problem mit den sogenannten „Access Rules“. Der Java-Compiler meldet: `Forbidden` bzw. `Discouraged access: The type ... is not accessible due to restriction on required library ...\plugins\....jar`.

„Access Rules“ werden in Java dazu genutzt, den Zugriff auf bestimmte Teile von eingebundenen Bibliotheken einzuschränken, obwohl er den Klassendefinitionen nach theoretisch möglich wäre. Das ist beispielsweise sinnvoll, wenn keine Abhängigkeit von internen Strukturen einer Bibliothek erzeugt werden soll; es handelt sich also um ein erweitertes Information Hiding. Wird gegen diese Access Rules verstoßen, so meldet der Compiler standardmäßig einen Fehler. Bei J-PaD scheint jedoch der Workaround bislang darin bestanden zu haben, den Compiler so einzustellen, dass er nur eine Warnung generiert.

(Q2) Einstellen sinnvoller Access Rules

Bei Eclipse-Plugin-Abhängigkeiten werden zunächst alle Abhängigkeiten auf ein verbotenes Niveau gesetzt. Es wäre allerdings problemfrei möglich, wenigstens alle Abhängigkeiten zu erlauben statt den Compilerfehler zu ignorieren. Am schönsten wäre natürlich eine zweckmäßige „so viel wie nötig, so wenig wie möglich“-Einstellung.

Dies nutzt vor allem Entwicklern, denen die Plugin-Entwicklung unter Eclipse unbekannt ist, da sie zunächst die Fehlerursache ergründen müssen. Dies tritt zwar nur einmal auf bis der Compiler umkonfiguriert wurde, ändern sie diese Einstellung allerdings nicht, so ist das Plugin prinzipiell unausführbar, was einem Totalausfall gleichkommt. Die notwendigen Einstellungen sollten in wenigen Stunden ausfindig gemacht und eingetragen werden können.

Qualitätenbaum-Befund 2 Nutzen: *Hoch* Umfang: *Niedrig*

Testbarkeit

Die Testbarkeit ist prinzipiell innerhalb der Programmgrenzen gut gegeben, da es sich um eine vergleichsweise kleine Einzelplatzsoftware handelt und alle sinnvollen Zustände determinstisch reproduzierbar sind.

Problematisch gestaltet sich die Testbarkeit allenfalls über die Grenzen des Plugins hinaus, da Eclipse doch relativ umfangreich ist. Dabei stellen allerdings auch hier Verbindungen mit anderen Systemen oder (pseudo-)zufälliges Verhalten eher die Ausnahme dar.

Strukturiertheit

J-PaD ist prinzipiell gut in die Unterprojekte „Kernsystem - Editor für Schema konfigurieren“, „Kernsystem - Editor für Dokumentation schreiben“, „Kernsystem - Widget API“ und „Mitgelieferte Widgets“ strukturiert. Dazu gesellen sich Projekte für die Bibliothek, die den Rich-Text-Editor zur Verfügung stellt. Auch bei der Aufteilung in Klassen fanden sich keine offensichtlichen Auffälligkeiten.

Allerdings wie bereits im Qualitätenbaum-Befund 1 erwähnt, liegt auch hier eine vollständige Architektur-Dokumentation nur in [Kir12] vor. Zwar ist J-PaD mit sich selbst dokumentiert, allerdings fehlt die Übersichtlichkeit über das „große Ganze“, wie in Eigene Idee 1 oder Eigene Idee 2 vorgeschlagen.

(Q3) Vollständiger Architekturentwurf fehlt

Es fehlt ein eigenständiges, zugängliches Architektur-Entwurfsdokument. Um einen schnellen Überblick über das System gewinnen zu können, der auch in der Wartung nachhaltig verändert werden kann, sollte ein separater Architekturentwurf verfasst werden. Dazu könnte J-PaD womöglich selbst genutzt werden.

Da gewisse Teile eines Entwurfs und Paketdokumentation vorliegen, dürfte der Aufwand, diese zu extrahieren und in einem Dokument zu konzentrieren, sich in Grenzen mehrerer Tage halten. Obwohl sich der Programmumfang in Grenzen hält und die bisherige Strukturierung ausbaufähig erscheint, bietet eine Architekturdokumentation den Vorteil, dass neue Entwickler schneller einen Einstiegspunkt finden als vor einem großen unbekannten, monolithisch wirkenden System zu stehen. Daher wird der vermutlich gelegentliche Nutzen mittelmäßig eingestuft.

Qualitätenbaum-Befund 3 Nutzen: *Mittel* Umfang: *Mittel*

Simplizität

Beim exemplarischen Betrachten des Quellcodes fällt keine unnötige Komplexität sowohl auf Modul- als auch auf Architekturebene auf.

Knappheit

Sowohl bei händischem Durchlesen als auch bei der Analyse mit dem Eclipse-Plugin „CodePro Analytix“ lässt sich innerhalb der Projekte so gut wie keine unnötigen Redundanzen finden.

Lesbarkeit

Generell ist der Code des Projektes ausreichend formatiert und Bezeichner sinnvoll benannt, sodass ein lesbarer Eindruck entsteht.

Eine Auffälligkeit findet sich darin, dass das Basis-Paket `de.j_pad` benannt ist. Ich halte dies für eher unansehnlich und auch der Java Coding Style Guide [Redoo] sagt hierzu:

> Generally, package names should use only lower-case letters and digits, and *no underscore*.

Es ist zwar nachvollziehbar, dass die Intention darin bestanden haben könnte, den Domain-Namen `www.j-pad.de` direkt auf die Java-Paket-Struktur abzubilden; allerdings wirkt dies eher unschön. Hinzu kommt, dass diese Abbildung lediglich eine Empfehlung darstellt und sich eher auf das Prinzip der Ländercodes und hierarchische Domainstruktur bezieht als auf eine exakte Abbildung der Domains. [Sun99]

(Q4) Ersetzen der Namensgebung `j_pad` durch `jpad`

In der Abwägung zwischen Abbildung des Domainnamens und dem Verzicht auf Unterstriche würde ich letztere Variante bevorzugen.

Dies löst eine gelegentliche Irritation, sodass der Nutzen nur gering ist. Allerdings sollte dies auch zügig mit Refactoring zu lösen sein.

Qualitätenbaum-Befund 4 Nutzen: *Niedrig* Umfang: *Niedrig*

Wie sich bei weiterem Durchsehen zeigt, richtet sich der Code zwar grundsätzlich nach dem Java-Styleguide [Sun99], allerdings nicht überall und der Styleguide ist nirgends festgehalten. Dies mündet in entsprechenden Compilerwarnungen, wenn dieser entsprechend restriktiv eingestellt ist.

(Q5) Erstellung und Umsetzung eines einheitlichen Styleguides

Der Code ist für den Entwickler in der Implementierung stets präsent, da die Sprache der zentrale Werkstoff in der Software schlechthin ist. Da durch irritierende Formatierung schwerwiegende Fehler entstehen können, wie z. B. beim Zugriff auf die falsche Variable oder fehlende Block-Einklammerung, wird dies als Ausfall bewertet. Durch die in Eclipse integrierten Funktionen lässt sich relativ einfach ein umfangreicher Styleguide erstellen. Damit lässt sich im Anschluss der Code automatisch formatieren.

Qualitätenbaum-Befund 5 Nutzen: *Hoch* Umfang: *Niedrig*

Geräteunabhängigkeit

Da J-PaD in Java geschrieben ist, weist es naturgemäß eine geringe Geräteabhängigkeit auf. Allerdings gibt es drei Projekte mit jeweils zwei Klassen, die der Namensgebung nach offensichtlich plattformabhängig sind.

(Q6) Ersetzen der drei plattformabhängigen Projekte durch ein plattformunabhängiges Projekt

Die verwendete Rich-Text-Editor-Komponente bringt die drei Projekte `com.onpositive-.swt.extension.linux`, `....macosx` und `.win32` mit. Diese stellen nur eine Teilmenge der mit Java kompatiblen Plattformen dar und können womöglich vereinheitlicht werden.

Da nicht klar ist, ob es sich nur um eine Irritation handelt, oder ob dies zu einem Ausfall der Rich-Text-Formatierbarkeit auf anderen Plattformen führt, wird sicherheitshalber ein mittlerer Nutzen angenommen. Es ist prinzipiell nicht klar ist, in welcher Zeit das Problem vollständig zu lösen wäre, deswegen wird sicherheitshalber ein hoher Umfang angenommen.

Qualitätenbaum-Befund 6 Nutzen: *Mittel* Umfang: *Hoch*

Abgeschlossenheit

Als Plugin, das gleichzeitig Kern eines Plugin-Systems ist und zudem Javadoc erweitert, kann starke Abgeschlossenheit prinzipbedingt kaum erreicht werden. Ich sehe hier auch kein Verbesserungspotential, da sich dieser integrierte Zustand des erweiterbaren Systems explizit als Anforderung in [Kir12] ergeben hat.

3.2.2. Brauchbarkeit

Korrektheit

Sowohl nach meiner Einschätzung als auch nach der des Kunden wird die Spezifikation erfüllt. Da diese allerdings, wie in Qualitätenbaum-Befund 1 erwähnt, nicht sonderlich umfangreich ist, bleibt die Nützlichkeit dieser Aussage unklar.

Ausfallsicherheit

Während des gesamten Betriebes zur Analyse sind keine Ausfälle aufgetreten. Es scheint somit eine ausreichende Ausfallsicherheit vorzuliegen.

Genauigkeit

Da eher organisatorische als rechnerische Tätigkeiten vom Code ausgeführt werden, lässt sich kaum eine Genauigkeit bestimmen. Innerhalb der textuellen Operationen scheinen die Resultate aber nicht von den Eingaben abzuweichen.

Effizienz

Die Effizienz während des Betriebes ist unauffällig. Es finden sich auch keine auffälligen Stellen im Code, die besondere Rechenzeit benötigen würden. Es wird von einer ausreichend guten Effizienz ausgegangen.

Sparsamkeit

Die Sparsamkeit während des Betriebes ist unauffällig. Es finden sich auch keine auffälligen Stellen im Code, die besonderen Speicherplatz benötigen würden. Es wird von einer ausreichend guten Sparsamkeit ausgegangen.

Leistungsvollständigkeit

Nach den Aussagen des Kunden erfüllt die Software prinzipiell die geforderten Leistungen, abgesehen von den in Abschnitt 3.6 erwähnten neuen Kundenanforderungen.

Handbuchvollständigkeit

Ähnlich wie Spezifikation und Entwurf findet sich ein kleines Handbuch in [Kir12]. Allerdings ist es nicht realistisch, dass dem Benutzer dieses Dokument vorliegt und er darin gezielt nach Hilfe zur Bedienung suchen wird.

(Q7) Verfassen eines umfangreichen Handbuchs

Es fehlt ein eigenständiges, zugängliches Handbuch. Um einen schnellen Überblick über die Bedienung des Systems gewinnen zu können und gezielte Hilfestellungen zu finden, sollte ein separates Handbuch verfasst werden.

Da gewisse Teile bereits vorliegen, dürfte der Aufwand, diese zu extrahieren und in einem Dokument zu konzentrieren, sich in Grenzen mehrerer Tage halten. Obwohl sich der Programmumfang in Grenzen hält und die Bedienung intuitiv erscheint, bietet ein Handbuch den Vorteil, dass für Benutzer auch selten benutzte Funktionalitäten erläutert oder offene Fragen beantwortet werden. Daher wird der vermutlich gelegentliche Nutzen mittelmäßig eingestuft.

Qualitätenbaum-Befund 7 Nutzen: *Mittel* Umfang: *Mittel*

Konsistenz

Im Rahmen der Analyse sind einige Mängel aufgefallen, die eine nahtlose Integration in Eclipse und einer erwartungsgemäßen Bedienung entgegenstehen.

(Q8) "Document Package" und „Edit Configuration" sind zu arg versteckt

Die Funktionalitäten zum Anlegen der Dokumentation und Bearbeiten der Konfiguration finden sich nicht an allen Orten, an denen Eclipse-Benutzer diese vermutlich suchen würden.

Zum Anlegen der Dokumentation muss auf das Paket rechtsgeklickt werden, wobei sich der Menüpunkt weit unten im Popup-Menü befindet, wie in Abbildung 3.6 dargestellt. In Eclipse könnte man auch den Weg über die Menüpunkte „Datei"→ „Neu" erwarten.

Auch sinnvoll wäre im Zusammenhang mit Anwendungsfall 202 aus Abschnitt 3.1.6 wäre eine Erweiterung der „Link with Editor"-Funktion von Eclipse, sodass die Auswahl eines Pakets in der Baumansicht des Package Explorers direkt die dazugehörige Paketdokumentation anzeigt.

Ebenso ist die Konfiguration nur in einer Dokumentation über ein Icon erreichbar. Dies widerspricht schon dem zeitlichen Arbeitsablauf, da man üblicherweise die Konfiguration zuvor anlegen möchte. Da die Konfiguration projektbezogen ist, ließe sie sich womöglich auch im Eclipse-Assistenten für „Projekteigenschaften" erreichbar machen.

Prinzipiell gibt es keinen Grund, den Zugang zur Funktionalität nur an einem Ort anzubieten. Da dies gelegentlich zu Irritationen führen dürfte, wird ein niedriger Nutzen angenommen. Die konkrete Umsetzung dürfte sich in wenigen Tag erledigen lassen.

Qualitätenbaum-Befund 8 Nutzen: *Niedrig* Umfang: *Mittel*

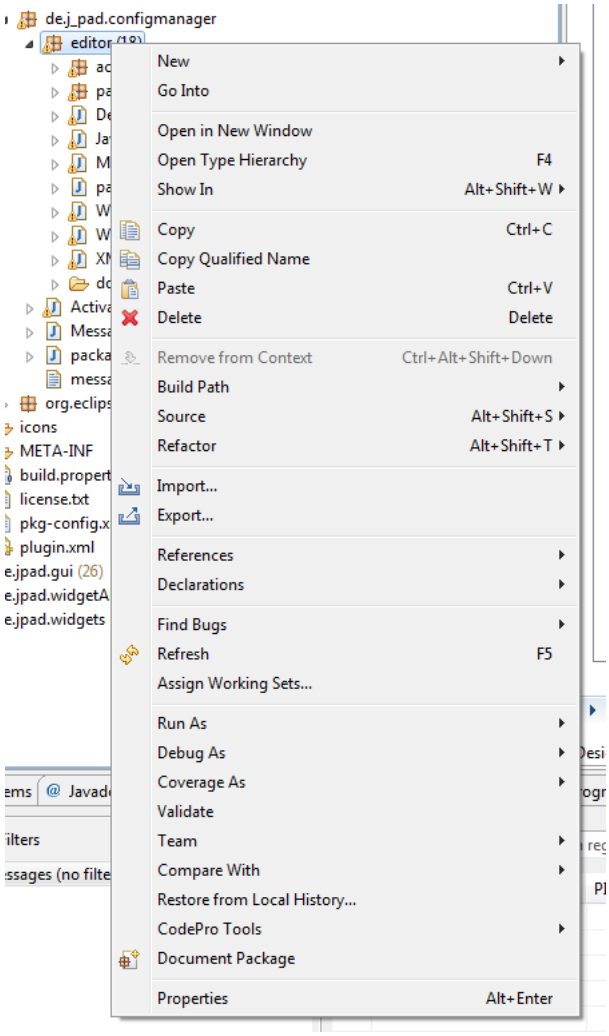

Abbildung 3.6.: Beispielhafte Anzeige des Popup-Menüs, das für jedes dokumentierte Paket aufgerufen werden muss

(Q9) Datumswidget kann kein Pflichtfeld sein

Jedes Widget kann als Pflichtfeld verwendet werden, d. h. es wird ein kleiner Warnhinweis dargestellt, wenn das Feld nicht dokumentiert wurde. Da das Datumswidget als Standardwert den aktuellen Tag anwählt, ist die erreichte Eingabe nicht automatisch abzuschätzen.

Es sollte ein leerer Standardwert verwendet werden, damit das Datum ebenfalls ein Pflichtfeld sein kann. Es dürfte sich um eine gelegentliche Irritation handeln. Die Behebung dürfte sich nicht so einfach gestalten, da das Fehlen eines leeren Standardwertes für diese Komponente seit 5 Jahren als Bug geführt wird [Ecl12] und somit entweder die vorhandene Komponente in großem Maßstab angepasst werden oder eine weitere Bibliothek eingebunden werden müsste.

Qualitätenbaum-Befund 9 Nutzen: *Niedrig* Umfang: *Hoch*

(Q10) Plugin-Beschreibungen in Eclipse sind fehlerhaft

Eclipse bietet in den „Installationsdetails“ eine Übersicht über alle Plugins, die derzeit installiert sind. Diese werden mit Angaben von Hersteller, Name, Version und ID angezeigt. Bei J-PaD findet sich jedoch kein Hersteller und die Plugins sind lediglich als „Gui“, „ConfigManager“, „Widget API“ und „Widgets“ benannt. Stattdessen sollten diese wenigstens ihre Zugehörigkeit mit einem „J-Pad“-Präfix klarstellen.

Es handelt sich hierbei um eine geringfügige Irritation, die auch schnell zu beheben sein sollte.

Qualitätenbaum-Befund 10 Nutzen: *Niedrig* Umfang: *Niedrig*

(Q11) Konfigurationstab hat keinen wiedererkennbaren Namen

In Eclipse werden alle Datei-Editoren als Tabs angezeigt. Alle J-PaD-Konfigurationen werden dabei als „pkg-config.xml“ statt ihrem Projektnamen geführt. Es ist prinzipiell möglich, den Tab anders zu benennen.

Im eher seltenen Falle mehrerer Projekte gleichzeitig zu bearbeiten, wie in Abbildung 3.7 dargestellt, kann dies zu Irritationen führen, weswegen der Nutzen eher niedrig ist. Allerdings dürfte auch die Umsetzung keine größeren Probleme bereiten.

Qualitätenbaum-Befund 11 Nutzen: *Niedrig* Umfang: *Niedrig*

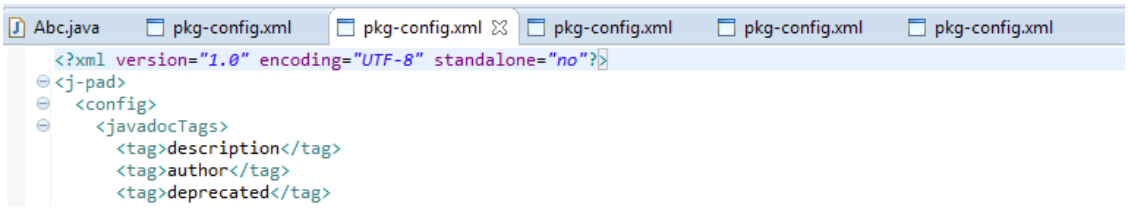

Abbildung 3.7.: Beispielhafte Anzeige multipler Konfigurationstabs

(Q12) Gleiche Tags in der Konfiguration werden verarbeitet

Die Elemente der Dokumentation sollten jeweils unter einem einzigartigen Javadoc-Tag gespeichert werden, z. B. `@author`. Andernfalls würde immer eine Eingabe in ein Widget verloren gehen, da ja der Tag nur einmal existieren kann. Daher warnt J-PaD, wenn zwei gleiche Tags in der Konfiguration vorhanden sind.

Allerdings ist es möglich die Warnung vollständig zu ignorieren, die Konfiguration zu speichern und das erwartete Fehlverhalten ohne Warnung im Dokumentationsmodus herbeizuführen. Wünschenswert wäre es, dem Benutzer gar nicht zu ermöglichen, diesen Fehler zu begehen, indem entweder das Speichern verweigert wird oder der Dokumentationsmodus eine entsprechende Warnung zeigt und das Eingeben in betroffene Widgets blockiert.

Auch wenn dieses Vorgehen eher selten der Fall sein dürfte, führt es doch zu einem Ausfall, da Eingaben verloren gehen. Somit ist ein mittlerer Nutzen gegeben. Die Umsetzung sollte relativ einfach möglich sein.

Qualitätenbaum-Befund 12 Nutzen: *Mittel* Umfang: *Niedrig*

(Q13) Vor leerem Tag in der Konfiguration wird nicht gewarnt

Wird einem Element der Dokumentation ein leerer Javadoc-Tag zugewiesen, so warnt J-PaD nicht über diesen Umstand. Tatsächlich werden alle Eingaben in dieses Element aber gar nicht gespeichert.

Es handelt sich somit um einen gelegentlichen Ausfall, dessen Behebung als mittlerer Nutzen zu bewerten ist. Die Umsetzung dürfte relativ einfach machbar sein.

Qualitätenbaum-Befund 13 Nutzen: *Mittel* Umfang: *Niedrig*

(Q14) Gleicher Name in Konfiguration wird nicht gewarnt

Die Konfiguration des Dokumentationsschemas in J-PaD ist als Baumstruktur hierarchisch aufgebaut. Es ist allerdings möglich, auf einer bestimmten Hierarchieebene zwei gleich benannte Elemente anlegen, ohne dass J-PaD hiervor warnt.

Bei der Dokumentation können zwei gleiche Felder jedoch Verwirrung stiften und für unnötige doppelte Datenhaltung sorgen. Es handelt sich hierbei um eine gelegentliche Irritation, die leicht zu lösen sein sollte.

Qualitätenbaum-Befund 14 Nutzen: *Niedrig* Umfang: *Niedrig*

Verständlichkeit

Als neuer Benutzer in einem neuen Projekt wird man in J-PaD zunächst die Dokumentationsansicht sehen, die dazu auffordert, in die leeren Eingabe-Felder Werte einzutragen. Diese sind zwar durch Namen wie „Datum“ oder „Autor“ gekennzeichnet, allerdings ist durch ein Stichwort nicht unbedingt der Zusammenhang klar, was genau hier eingetragen werden sollte.

(Q15) Erklärung der Eingabe-Felder fehlt

Die zu tätigenden Eingaben könnten Benutzern unklar sein, da die Felder nur mit einem Stichwort beschriftet sind. Daher wäre eine Erklärungsfunktion wünschenswert, die zu einzelnen Feldern einen längeren Hinweistext bereitstellt.

Es handelt sich hierbei vermutlich um häufige Irritationen bis gelegentliche Ausfälle, wenn lieber nichts eingetragen wird als etwas falsches. Dies führt zur Einstufung mittleren Nutzens. Die Umsetzung dürfte etwas aufwändiger werden, allerdings im Rahmen einer Woche bleiben.

Qualitätenbaum-Befund 15 Nutzen: *Mittel* Umfang: *Mittel*

Einfachheit

Obwohl J-PaD generell eher eine einfache Bedienoberfläche aufweist, gab es doch einige Auffälligkeiten, die die Bedienung unnötig erschwerten. Einige Probleme werden hier nicht als Befund erwähnt, da sie sich in Abschnitt 3.4 ergeben haben.

(Q16) Zu wenig Bearbeitungsmöglichkeiten beim Konfigurationsbaum

Um das Dokumentationsschema zu konfigurieren, verwendet J-PaD eine Baumansicht der Elemente. Diese ist allerdings nicht sonderlich intuitiv gestaltet, da beispielsweise kein Drag&Drop innerhalb des Baumes möglich ist, sondern Elemente nur mit „Up“- und „Down“-Buttons bewegt werden können.

Unglücklich ist auch die optische Vermischung von Tabs und tatsächlichen sichtbaren Elementen, da alle als Baumelemente, lediglich mit abweichenden Icons, dargestellt werden. Es kann auch keine mehrspaltige Formatierung gewählt werden. Möglich wäre auch eine Integrierung des „Add-Widget“-Dialogs direkt in die Benutzeroberfläche durch eine Tool-Bar am Rand oder eine direkte Vorschau des Layouts.

Es handelt sich hierbei vermutlich um eine häufige Behinderung, sodass ein mittlerer Nutzen unterstellt wird. Der Umfang der vorgeschlagenen Änderungen bewegt sich allerdings eher in einem größeren Rahmen, weswegen mit mehr als einer Woche gerechnet wurde.

Qualitätenbaum-Befund 16 Nutzen: *Mittel* Umfang: *Hoch*

(Q17) Bearbeiten im Konfigurationsbaum störend für Arbeitsfluss

Selbst wenn man die vorhandene Baumansicht zur Konfiguration des Dokumentationsschemas verwendet, ist der Arbeitsfluss leicht behindert. So wird beim Hinzufügen eines Elements das derzeit selektierte Element deselektiert (unter welches das neue Element eingehangen wird) und der entstandene Unterbaum mit dem Kindelement nicht ausgeklappt. Wünschenswert wäre es, den Baum auszuklappen, das neue Widget zu selektieren und den Tastatur-Fokus auf ein zu bearbeitendes Eingabeelement zu richten.

Es handelt sich hierbei um eine Irritation, die bei Neu-Erstellung häufig, bei reiner Änderung nur selten auftreten kann. Daher wird nur ein geringer Nutzen angenommen. Allerdings dürften sich die Änderungen relativ schnell erledigen lassen.

Qualitätenbaum-Befund 17 Nutzen: *Niedrig* Umfang: *Niedrig*

(Q18) Neue Konfiguration löscht Javadoc-Tags

J-PaD verwendet in der Konfiguration eine Liste von Tags, die auch von Javadoc erkannt werden und unterscheidet so die Platzierung des Tags in der `.java`-Datei. Benutzt man im Konfigurationsmodus von J-PaD jedoch das Icon „neue Konfiguration“, so löscht J-PaD alle Einstellungen inklusive der Javadoc-Tags. Tatsächlich wird man allerdings in der überwiegenden Mehrzahl der Fälle diese Tags auch verwenden wollen, um eine Integration mit Javadoc zu gewährleisten.

Der Klick auf das Icon sollte somit diese Tags nicht löschen. Als häufige Irritation wird ein mittlerer Nutzen zugewiesen, die Umsetzung sollte vergleichsweise einfach vonstatten gehen.

Qualitätenbaum-Befund 18 Nutzen: *Mittel* Umfang: *Niedrig*

(Q19) Fehlende Übernahme der Daten wenn Tag-Name geändert wird

Wird in einem vorhandenen Dokumentationsschema ein Javadoc-Tag-Name geändert, so zeigt J-PaD zwar eine Warnung, da die gewünschte Aktion nicht eindeutig ist. Allerdings bestehen die Möglichkeiten nur aus „Lösche die Dokumentation unter dem alten Tag“ und „Behalte die Dokumentation unter dem alten Tag bei“. Prinzipiell wäre aber wohl „Übernehme die Dokumentation unter dem alten Tag und speichere sie unter dem neuen Tag“ die gewünschte Funktionalität, da der Tag ja nur umbenannt, nicht gelöscht wird. Ebenfalls lässt sich zum Zeitpunkt der Warnung die Aktion gar nicht mehr abbrechen.

Der Kernpunkt der Lösung dürfte die Implementierung des Umbenennens sein. Da die Warnung gar nicht erst von der Bearbeitungsaktion, sondern von der Konsistenzprüfung ausgelöst wird, dürfte diese einen größeren Umfang aufweisen und Eingriffe in die Architektur erfordern, was vermutlich einige Tage in Anspruch nehmen wird. Es gibt keine andere Möglichkeit, den Dokumentationstag umzubenennen. Da es sich somit um einen seltenen bis gelegentlichen Ausfall der Funktionalität handelt, wird der Nutzen als mittelmäßig bewertet.

Qualitätenbaum-Befund 19 Nutzen: *Mittel* Umfang: *Mittel*

(Q20) Warnung beim Löschen eines Element fehlt

Die bereits unter Qualitätenbaum-Befund 19 erwähnte Warnung wird ebenfalls ausgelöst, wenn man Elemente aus der Konfiguration löscht und dann speichert. Ebenfalls lässt sich die Aktion nicht mehr abbrechen.

Die Bewertung gestaltet sich ebenfalls analog zu Qualitätenbaum-Befund 19.

Qualitätenbaum-Befund 20 Nutzen: *Mittel* Umfang: *Mittel*

(Q21) „MultiValue“-Widget kann nicht mit Tastatur bearbeitet werden

In J-PaD kann man wie gewohnt durch das restliche Dokumentationsfeld mit der Tastatur navigieren, z. B. mit Hilfe der Tab-Taste. Dadurch kann der Eingabefokus schnell gewechselt werden ohne zuvor die Maus bedienen zu müssen.

Im „MultiValue“-Widget ist dies allerdings nicht möglich, da mit der Tastatur hier zwar die einzelnen Zeilen angewählt werden können, aber keine Möglichkeit besteht, vorhandene Zeilen zu editieren oder zu löschen bzw. existierende Zeilen zu löschen.

Es dürfte sich hierbei um eine häufige Irritation handeln, was mittelmäßigen Nutzen erkennen lässt. Die Umsetzung dürfte innerhalb einiger Tage möglich sein, insofern sie nicht ähnlichen Einschränkungen wie Qualitätenbaum-Befund 9 unterworfen ist.

Qualitätenbaum-Befund 21 Nutzen: *Mittel* Umfang: *Mittel*

3.3. Werkzeug-Ergebnisse

Statische Code-Analysen erzeugen auf höchster Feinheitseinstellung naturgemäß eine große Zahl von falsch-positiven oder zumindest nur nebensächlichen Meldungen [WJKT05]. Allerdings war in dieser Projektgröße abzusehen, dass sich die Fehlerzahl im vergleichsweise 'überschaubaren', dreistelligen Bereich (mit Wiederholungen) ausprägen wird. Somit halte ich die Verwendung mit maximalen Einstellungen für sinnvoll, da die Fehleranzahl bearbeitet werden kann und bestimmte Fehlerklassen im Nachhinein gezielt nicht als Fehler anerkannt werden müssen.

Die Tabelle 3.1 illustriert die Verteilung von Einzel-Befunden (mit Wiederholungen) der einzelnen verwendeten Analysewerkzeugen.

	Werkzeug	
Paket	javac	FindBugs
`de.jpad`	—[a]	—[a]
`de.jpad.configManager`	328	37
`de.jpad.gui`	182	2
`de.jpad.widgetAPI`	141	21
`de.jpad.widgets`	416	79
`com.onpositive.richtext.model`	1700	291
`com.onpositive.richtexteditor`	1304	223
`com.onpositive.richtexteditor.swt.extension`	535	52
`com.onpositive.swt.extension.linux`	18	1
`com.onpositive.swt.extension.macosx`	10	3
`com.onpositive.swt.extension.win32`	5	0

Tabelle 3.1.: Anzahl der Werkzeug-Befunde der statistische Codeanalyse-Werkzeuge

[a] enthält keinen Quelltext

Eine erste Sichtungen der einzelnen Befunden der Werkzeuge zeigt erwartungsgemäß mehrheitlich eher formatierungsbedingte und unkritische Probleme bei den J-PaD-Paketen, wie z. B. unqualifizierter Zugriff, unnötige Null-Pointer-Überprüfungen oder unter Umständen von der Lokalisierung abhängige Funktionalität. Das Ergebnis deutet jedoch neben den gleichen Fehlern im Rich-Text-Editor auch auf einige schwerwiegendere Fehler hin, wie z. B. Gleichheitsvergleiche auf Strings und Gleitkommazahlen, ungenutzte Variablen oder mögliche Null-Dereferenzierungen.

(W1) Werkzeug-Befunde bewerten und gegebenenfalls beheben

Die etwa 5000 gefundenen Befunde müssen als falsch-positiv oder richtig-positiv bewertet werden und anschließend alle richtig-positiven Befunde behoben werden. Die vergleichsweise große Zahl kommt hauptsächlich durch Formatierungsprobleme zustande, sodass in Zusammenhang mit Qualitätenbaum-Befund 5 bereits eine großer Prozentsatz erledigt sein könnte.

Der Nutzen lässt sich in diesem Fall nur abschätzen, da kein wirklicher Ausfall derzeit in Verbindung mit diesen Fehlern gebracht werden kann. In Anbetracht der doch relativ gravierenden Fehlern in der Rich-Text-Komponente wird allerdings wenigstens ein mittlerer Nutzen angenommen.

Die Mehrzahl der Fehler ist vergleichsweise einfach mit einer Standardlösung zu beheben, da die Werkzeuge auch die richtige Codestelle sofort anzeigen. Jedoch dürften sich insbesondere durch den Umfang der Fehlerzahl der Rich-Text-Komponente auch einige komplexere Fehler darunter finden, sodass die Behebung einige Tage in Anspruch nehmen dürfte.

Werkzeug-Befund 1 Nutzen: *Mittel* Umfang: *Mittel*

(W2) Rich-Text-Editor ersetzen

Als weitere Alternative kommt ein Ersetzen der Rich-Text-Komponente in Betracht, da eine separate Wartung einer Alternative der Komponente unnötigen Aufwand erzeugt.

Der Nutzen lässt sich in diesem Fall nur abschätzen, da kein wirklicher Ausfall derzeit in Verbindung mit diesen Fehlern gebracht werden kann. In Anbetracht der doch relativ gravierenden Fehlern in der Rich-Text-Komponente wird allerdings wenigstens ein mittlerer Nutzen angenommen.

Der Aufwand hierfür wird als Hoch eingeschätzt, da zunächst andere Komponenten gefunden und evaluiert werden müssen. Wäre eine entsprechende Funktionalität und Fehlerfreiheit gegeben, müsste zudem an allen Schnittstellen die Verbindung verändert werden, was womöglich Architektur-Probleme oder neue Fehler nach sich zieht.

Werkzeug-Befund 2 Nutzen: *Mittel* Umfang: *Hoch*

3.4. Usability Patterns

(U1) Objektbezogenes Rückgängigmachen für Konfigurationseditor

In J-PaD wird das Dokumentationsschema in einem eigenen Konfigurationseditor bearbeitet, wie in Anwendungsfall 101 aus Abschnitt 3.1.3 dargelegt. Während für den normalen Dokumentationseditor Eclipse eine Rückgängig-Funktion (Undo/Redo) bereitstellt, ist dies im Konfigurationseditor nicht der Fall. Allerdings ist grade das Ändern der Konfiguration ein vergleichsweise aufwändiger Prozess aus mehreren Schritten, die nicht unbedingt sehr einfach rückgängig zu machen sind, wie z. B. das Löschen eines ganzen Tabs mit einem Dutzend Elementen oder das Verschieben von Elementen. Ebenso existieren 2 Icons mit unterschiedlicher Funktionalität, die ohne Warnung die gesamte Konfiguration löschen.

Es bietet sich daher an, für den Konfigurationseditor ebenfalls ein objektbezogenes Undo/Redo zu implementieren. Das entstehende Fehlverhalten wird als häufige Irritation bis gelegentlichen Ausfall eingestuft und somit ein mittlerer Nutzen unterstellt. Stünde eine vorhandene Eclipse-Einbindung zur Verfügung, ließe sich dies wohl in einer Woche implementieren. Da aber auch der interne Aufbau des Models nicht ganz klar ist und derartige Systeme, wenn sie nachträglich eingeführt werden, oft zu unerwünschten Nebenwirkungen tendieren, wird eher eine längere Zeitdauer angenommen.

Usability-Pattern-Befund 1 Nutzen: *Mittel* Umfang: *Hoch*

(U2) Gute Standardwerte

Der Anwendungsfall 101 aus Absatz 3.1.3 enthält die Möglichkeit, Widgets zur Konfiguration hinzuzufügen. Fügt man also im Konfigurationseditor von J-PaD ein solches neues Element hinzu, so wird als Standardname „New Widget“ vergeben, alle anderen Felder bleiben leer. Es sollte möglich sein, je nach Widget sinnvollere Standardwerte für z. B. Name oder Tag zu entwickeln und zu implementieren. Ein Beispiel wäre „Datum“ als Name für ein Datumswidget und einen abhängig von der derzeitigen Konfiguration garantiert einmaligen Tag (“datum1“, „datum2“, etc.).

Es handelt sich hierbei um eine Irritation, die bei Neu-Erstellung häufig, bei reiner Änderung nur selten auftreten kann. Daher wird nur ein geringer Nutzen angenommen. Der Umfang einer Lösung sollte eine Woche nicht überschreiten.

Usability-Pattern-Befund 2 Nutzen: *Niedrig* Umfang: *Mittel*

(U3) Widget-Vorschau

Benutzer des Konfigurationseditors suchen nach bestimmten Eingabemöglichkeiten (Text, Datum, Ressourcen, etc.), wenn sie in Anwendungsfall 101 aus Absatz 3.1.3 ein neues Widget hinzufügen wollen. Derzeit werden nur die Klassen- und Paket-Namen der Widget-Fabriken angezeigt, also z. B. `de.j_pad.widgets.text.SingleLineTextWidgetFactory`. Obwohl die Entwickler wahrscheinlich mit derartigen Namenskonventionen und mit dem Factory-Pattern vertraut sein werden, so wäre eine direkte Vorschau des Widgets doch sinnvoller, da so dem Benutzer das Angebot schneller ersichtlich wird.

Falls dies nicht möglich sein sollte, so wäre es wenigstens nach der Auswahl eine sinnvolle Option, da derzeit selbst mit Auswahl nur ein einzeiliger Beschreibungstext bereitgestellt wird.

Es handelt sich hierbei um eine Irritation, die bei Neu-Erstellung häufig, bei reiner Änderung nur selten auftreten kann. Daher wird nur ein geringer Nutzen angenommen. Der Umfang einer Lösung sollte im Bereich um eine Woche liegen. Sicherheitshalber wird daher eher mit einer längeren Zeitspanne gerechnet.

Usability-Pattern-Befund 3 Nutzen: *Niedrig* Umfang: *Hoch*

(U4) Auto-Vervollständigung bei Class#Method

Eclipse bietet dem Benutzer eine Auto-Vervollständigung bei Eingaben von Klassennamen und Methodennamen. Eine Übernahme dieser Funktionalität in die Editoren von J-PaD würde dafür sorgen, dass diese leichter eingegeben werden können. Sinnvoll wäre dieses Verhalten vor allem für die Dokumentation bei Anwendungsfall 102 aus Absatz 3.1.4.

Es handelt sich hierbei um eine Irritation, die auch nicht allzu häufig auffällig sein sollte. Die konkrete Dauer zum Einbau einer derartigen Funktionalität ist bedingt durch die Rich-Text-Editor-Komponente nicht wirklich abzusehen und wird daher mit hoch angenommen.

Usability-Pattern-Befund 4 Nutzen: *Niedrig* Umfang: *Hoch*

(U5) Assistent für Konfiguration erstellen

Momentan muss zum Durchführen von Anwendungsfall 101 aus Absatz 3.1.3 und Erstellen eines Dokumentationsschemas ein umständlicher Editor bedient werden, der Kenntnisse über den konkreten Aufbau von J-PaD voraussetzt (z. B. Tabs, Gruppen, Widgets, Tags). Es wäre wünschenswert, dass Benutzer ein eigenes Dokumentationsschema entwerfen können ohne Kenntnisse über den Aufbau von J-PaD besitzen zu müssen.

Hierfür bietet sich der Einbau eines Assistenten an, der, basierend auf allgemeinen Eingaben, ein zutreffendes Dokumentationsschema automatisch erstellen kann. Selbst wenn die Formatierung oder Anordnung noch bearbeitet werden muss, würde ein solcher Assistent doch dem unerfahrenen Benutzer viel Arbeit abnehmen.

Es handelt sich hierbei um eine Irritation, die nur bei der Neu-Erstellung eines Schemas gelegentlich bis selten auftreten kann. Die konkrete Dauer zum Einbau einer derartigen Funktionalität würde wahrscheinlich in den Rahmen einer Woche fallen, allerdings müsste zunächst die Verarbeitungsschritte des Assistenten modelliert werden, wie z. B. wie der Tag bestimmt wird; daher wird sicherheitshalber ein hoher Umfang angenommen.

Usability-Pattern-Befund 5 Nutzen: *Niedrig* Umfang: *Hoch*

3.5. Eigene Ideen

Zu den genannten Punkten finden sich auch eigene Ideen zur Weiterentwicklung, die sich nicht direkt den bisherigen Analysetechniken zuordnen lassen. Allen ist dabei gemein, dass sie als eine fehlende Funktionalität angesehen werden und somit allen Ideen ein mittlerer Nutzen unterstellt wird.

(E1) Modulübersicht

Da Java-Pakete baumartig strukturiert sind, würde sich anbieten eine Baumübersicht aller Pakete mit kurzer Zusammenfassung anzubieten, sodass man schnell einen Überblick über die gesamte Architektur des Projekts gewinnen kann. Ein derartiges Vorgehen ist auch in Anwendungsfall 202 aus Absatz 3.1.6 erwähnt.

Momentan existiert eine derartige Übersicht zwar mit dem „Package Explorer“ von Eclipse, allerdings müssen hier zunächst Unterbäume ausgeklappt werden und es finden sich auch alle Klassen darin, sodass dies eher einen umständlichen Weg darstellt.

Es ist zwar denkbar, dass sich dies unter Umständen auch in wenigen Tagen erledigen lässt, allerdings ist eine Entwicklungszeit von wenigstens einer Woche eine realistischere Annahme.

Eigene Idee 1 Nutzen: *Mittel* Umfang: *Hoch*

(E2) Report erzeugen

Es ist zwar möglich, die Paket-Dokumentation teilweise über Javadoc in HTML zu erzeugen. Allerdings werden dabei nur die bereits vorhandenen Javadoc-Tags berücksichtigt, sodass viele Eingaben ignoriert werden. Es sollte daher möglich sein, gezielt *alle* Daten der Paketdokumentation zu exportieren; wahlweise in ein Fein-Architektur-Entwurfsdokument oder eine interaktive HTML-Übersicht.

Eine solche Funktionalität weist naturgemäß viele Komplikationen bezüglich der Formatierung des Textes auf, sodass mit einem Zeitrahmen von mindestens einigen Wochen gerechnet werden sollte.

Eigene Idee 2 Nutzen: *Mittel* Umfang: *Hoch*

(E3) Motivation durch Verknüpfung wecken

J-PaD sollte in der Lage sein, gezielt die Motivation nach Dokumentation zu wecken. Dazu könnte ein Ansatz gewählt werden, der vergleichbar zur vorhandenen Javadoc-Dokumentation von Methoden ist:

- Der Benutzer muss zu einem gewissen Zeitpunkt die Nachteile von fehlender Dokumentation erleben. Im bereits vorhandenen Ansatz von Javadoc stellt er z. B. fest, dass eine Methode nicht dokumentiert ist, wie in Abbildung 3.8 gezeigt.
- J-PaD muss den Benutzer in diesem Moment darauf hinweisen, dass Dokumentation die Arbeit jetzt erleichtert hätte. Bei Javadoc wird dies erreicht, indem der Benutzer übliche Anzeigen kennt, die ihm normalerweise die für ihn hilfreichen Informationen angezeigt hätten. Da er aber nur ein leeres Feld vor sich hat, bemerkt er den Unterschied zu fehlender Dokumentation direkt.
- Wenn der Benutzer dann mit J-PaD dokumentiert hat, muss J-PaD in einem ähnlichen Moment nun die hilfreiche Dokumentation anzeigen. Im Javadoc-Beispiel wäre nun die Methode dokumentiert und würde den normalerweise hilfreichen Kommentar in der Anzeige darstellen.

Bei der Dokumentation von Methoden ist klar, dass (fehlende) Dokumentation als Hinweis angezeigt werden soll, wenn der Mauszeiger den Methodenbezeichner überfährt. Es lässt sich aber schlecht erahnen, wann der Benutzer die Dokumentation von Modulen bzw. Paketen benötigt hätte. Ein möglicher Ansatz wäre die Einblendung ausgewählter Dokumentationselemente, wenn der Mauszeiger den Paketbezeichner oder einen Klassenbezeichner innerhalb des Pakets überfährt.

Auch hier ist der Umfang nicht wirklich ersichtlich, es kann allerdings durch den Eingriff in Javadoc von einer größeren Komplexität ausgegangen werden, weswegen ebenfalls mindestens eine Woche angesetzt wird.

Eigene Idee 3 Nutzen: *Mittel* Umfang: *Hoch*

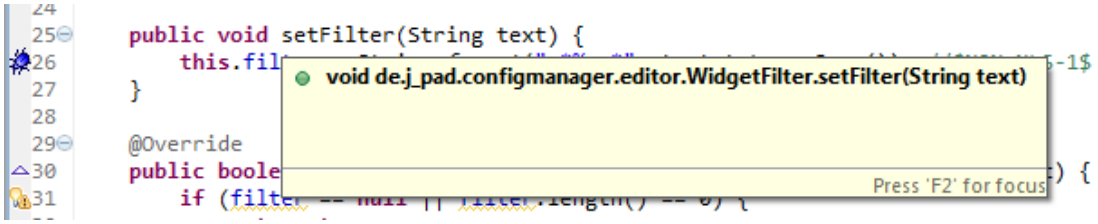

Abbildung 3.8.: Beispielhafte Anzeige einer undokumentierten Methode in Javadoc

3.6. Kundenanforderungen

Im Rahmen der Analyse sind auch vom Kunden Anforderungen genannt worden, die allerdings größtenteils Schwachstellen benennen, die auch in der Analyse aufgetreten sind und daher nur der Vollständigkeit und Nachvollziehbarkeit halber aufgeführt werden. Nichtsdestotrotz finden sich auch neue Anforderungen darunter.

(K1) Knoten des Konfigurationsbaums mit Drag&Drop verschieben

Der Befund fällt inhaltlich teilweise mit Qualitätenbaum-Befund 16 zusammen, wo er bereits erläutert wurde.

Da es sich aber hier explizit nur um die Drag&Drop-Funktionalität handelt, wird lediglich mittlerer Umfang von einigen Tagen angenommen.

Kundenanforderung 1 Nutzen: *Mittel* Umfang: *Mittel*

(K2) Bei der Auswahl eines Widgets eine Vorschau anzeigen

Der Befund fällt inhaltlich mit Usability-Pattern-Befund 3 zusammen, wo er bereits erläutert wurde.

Kundenanforderung 2 Nutzen: *Niedrig* Umfang: *Hoch*

(K3) Standardwerte für Konfigurationen einstellbar machen

Öffnet man die Dokumentationsmaske von J-PaD für ein bislang undokumentiertes Paket so zeigen alle Widgets leere Eingabedaten bzw. das Datumswidget den heutigen Tag. Das bedeutet, dass Angaben, die über den Großteil eines Projektes gleich bleiben und nur in einzelnen Stellen angepasst werden müssten (wie z. B. die Lizenz) für jedes Paket erneut eingegeben werden müssen.

Es wäre hier sinnvoll ein Feld in der Konfiguration anzubieten, das es ermöglicht, einen konkreten Standardwert für ein Widget für alle Pakete einstellbar zu machen. Dadurch würde, immer wenn keine anderslautende Eingabe vorliegen würde, automatisch der Standardwert benutzt und in die Paketdokumentation eingetragen.

Es handelt sich hierbei um eine Irritation, die bei Neu-Erstellung häufig, bei reiner Änderung nur selten auftreten kann. Daher wird nur ein geringer Nutzen angenommen. Der Umfang einer Lösung sollte eine Woche nicht überschreiten.

Kundenanforderung 3 Nutzen: *Niedrig* Umfang: *Mittel*

(K4) Anzeige für relativen Anteil der ausgefüllten Dokumentation

Um in einem Projekt eine schnelle Übersicht über die Quantität der vorhandenen Dokumentation gewinnen zu können wäre eine projektweite Übersicht über den Dokumentationsstand bezüglich der ausgefüllten Widgets hilfreich. Diese ließe sich um eine Aufschlüsselung pro Paket erweitern, sodass eventuelle Missverhältnisse zwischen einzelnen Paketen offensichtlich werden würden.

Es handelt sich hierbei allenfalls um eine Irritation, die gelegentlich auftritt und somit mit niedrigem Nutzen gewertet wird. Die konkrete Umsetzung sollte sich in einigen Tagen bewerkstelligen lassen.

Kundenanforderung 4 Nutzen: *Niedrig* Umfang: *Mittel*

(K5) Nicht dokumentierte Pflichtfelder in der „Problems View“ von Eclipse anzeigen

Nicht dokumentierte Pflichtfelder werden in der Ansicht J-PaD zwar als Warnung angezeigt, allerdings bietet Eclipse eine eigene Übersicht über alle Warnungen und Fehler in Projekten. Es wäre hilfreich für die Übersichtlichkeit, wenn die fehlende Pflichtdokumentation also auch dort bemängelt würde.

Es handelt sich hierbei allenfalls um eine Irritation, die gelegentlich auftritt und somit mit niedrigem Nutzen gewertet wird. Die konkrete Umsetzung sollte sich in einigen Tagen bewerkstelligen lassen.

Kundenanforderung 5 Nutzen: *Niedrig* Umfang: *Mittel*

(K6) Ein weiteres Ressourcen-Widget hinzufügen

J-PaD stellt ein Widget bereit, das eine Auflistung von externen Ressourcen ermöglicht. Gewünscht ist nun ein Widget sehr ähnlich zu `de.j_pad.widgets.resources`, welches jedoch nur alle Ressourcen aus dem doc-files-Ordner anzeigt und bei dem nur die Beschreibungstexte geändert werden können. Es handelt sich also um ein Widget für die Auflistung von internen Ressourcen.

Da es sich bei dem neuen Widget lediglich um eine Einschränkung des vorhandenen Widgets handelt, sollte der Nutzen eher gering sein, da es sich hier lediglich um eine gelegentliche Irritation handelt. Ebenso dürfte der Umfang kaum einen Tage überschreiten, da lediglich Einschränkungen statt Erweiterungen getroffen werden müssen.

Kundenanforderung 6 Nutzen: *Niedrig* Umfang: *Niedrig*

(K7) Widget für exklusive Auswahl hinzufügen

In einer Dokumentation möchte man manchmal eine exklusive Auswahl aus mehreren vorgegebenen Punkten treffen, wie z. B. die Angabe der Priorität eines Pakets. Hierfür soll ein Widget umgesetzt werden, das aus Radio-Buttons besteht, mit denen eine derartige Auswahl getroffen werden kann.

Da diese Funktionalität noch gar nicht in J-PaD bereitsteht, wird dies als gelegentlicher Ausfall gewertet. Der Umfang dürfte zwar nur einige Tage beanspruchen, allerdings ist nicht unbedingt klar, inwiefern die Architektur eine Bereitstellung zusätzlicher Daten in einem Dokumentationsschema unterstützt. Falls die zu erwartenden Änderungen viel weiteren Aufwand erzeugen würden, müsste gegebenenfalls die freie Einstellbarkeit aufgegeben werden und sich auf lediglich z. B. fünf feste Radio-Buttons beschränken.

Kundenanforderung 7 Nutzen: *Mittel* Umfang: *Mittel*

(K8) Ersetzen der drei plattformabhängigen Module durch ein einziges, plattformunabhängiges Modul

Der Befund fällt inhaltlich mit Qualitätenbaum-Befund 6 zusammen, wo er bereits erläutert wurde.

Kundenanforderung 8 Nutzen: *Mittel* Umfang: *Hoch*

(K9) Hilfetext für Widgets verfügbar machen

Der Befund fällt inhaltlich mit Qualitätenbaum-Befund 15 zusammen, wo er bereits erläutert wurde.

Kundenanforderung 9 Nutzen: *Mittel* Umfang: *Mittel*

(K10) Arbeitsfluss beim Widget hinzufügen verbessern

Derzeit wird beim Erstellen eines Dokumentationsschemas beim Einfügen eines Widgets lediglich ein Dialog geöffnet, der eine Auswahl des Widgets und eine Text-Eingabemaske zur Filterung der bereitstehenden Widgets beinhaltet. Die konkreten Einstellungen wie Name, Tag, etc. können nach dem Schließen des Dialogs eingestellt werden. Bei der konkreten Benutzung zeigte sich, dass dieser Ansatz wenig intuitiv ist und dazu führt, dass Benutzer den gewünschten Namen in die Eingabemaske zur Filterung eingeben.

Daher soll bei der Erstellung eines neuen Widgets im sich dafür öffnenden Dialog bereits der Name angegeben werden können. Für das Feld „Tag“ soll zudem als Standardwert der Name des Widgets in Kleinbuchstaben eingetragen werden.

Bezüglich des Standardwerts fällt der Befund mit Usability-Pattern-Befund 2 zusammen, wo dieser Teil bereits erläutert wurde. Die Änderung des Dialogs erscheint nicht gravierend und macht somit keine Änderung an der getroffenen Einschätzung nötig.

Alternativ ließe sich auch Qualitätenbaum-Befund 16 umsetzen.

Kundenanforderung 10 Nutzen: *Niedrig* Umfang: *Mittel*

(K11) Löschen der Dokumentationseinträge bei Widget umbenennen verhindern

Das Umbenennen eines Widgets löscht das vorhandene Widget und legt es neu an, sodass alle Dokumentationseinträge mitgelöscht werden. Stattdessen soll das Umbenennen keine unerklärlichen Folge-Effekte zeigen. Der Befund fällt inhaltlich mit Qualitätenbaum-Befund 19 zusammen, wo er bereits erläutert wurde.

Kundenanforderung 11 Nutzen: *Mittel* Umfang: *Mittel*

(K12) Leeren Standardwert und Pflichtfeld für Date-Widget

Das Date-Widget soll als Standardwert nicht heute, sondern einen leeren Wert enthalten. Es soll auch als Pflichtfeld gesetzt werden können. Der Befund fällt inhaltlich mit Qualitätenbaum-Befund 9 zusammen, wo er bereits erläutert wurde.

Kundenanforderung 12 Nutzen: *Niedrig* Umfang: *Hoch*

(K13) Seltsames Verhalten des Multi-Value-Widgets unter MacOS

Das Multi-Value-Widget von J-PaD ermöglicht es eine beliebige Anzahl von Textzeilen einzufügen, z. B. eine Liste von Autoren. Hierzu muss immer eine zusätzliche, leere Zeile bereitstehen, um eine neue Zeile eingeben zu können. Unter MacOS scheint hier allerdings die Darstellung nicht befriedigend zu funktionieren, sodass Änderung zu spät dargestellt oder eine Leerzeile zu viel dargestellt wird.

Das auffällige Verhalten hängt womöglich mit Kundenanforderung 8 zusammen. Für MacOS-Benutzer dürfte es sich hierbei um eine häufige Irritation handeln, weswegen ein mittlerer Nutzen angenommen wird. Das konkrete Problem könnte mit der genutzten Bibliothekskomponente zusammenhängen, deren Änderung sich als schwierig erweisen würde, weswegen ein hoher Umfang zur Beseitigung erwartet wird.

Kundenanforderung 13 Nutzen: *Mittel* Umfang: *Hoch*

(K14) Auto-Complete für Paket.Klasse#Member

Der Befund fällt inhaltlich mit Usability-Pattern-Befund 4 zusammen, wo er bereits erläutert wurde.

Kundenanforderung 14 Nutzen: *Niedrig* Umfang: *Hoch*

(K15) Testfalldokumentation mit Kopfkommentaren synchronisieren

J-PaD bietet einen Widget an, mit dem Testfälle in die Paketdokumentation aufgenommen werden können. Dort können die Testfälle einzeln dokumentiert werden. Dabei wird zwar beim Importieren der Testfälle die Dokumentation aus den Kopfkommentaren der JUnit-Testfälle ausgelesen; allerdings wird bei Änderungen in der J-PaD-Dokumentation keine rückwirkende Änderung in den JUnit-Testfällen vorgenommen.

Es dürfte sich hierbei um eine gelegentliche bis häufige Irritation handeln. Da allerdings insbesondere kein Datenverlust auftritt und Testfälle vermutlich seltener nachträglich dokumentiert werden, wird lediglich ein geringer Nutzen angenommen. Die konkrete Umsetzung dürfte innerhalb einiger Tage in ähnlicher Weise wie die Synchronisierung mit der Paketdokumentation machbar sein.

Kundenanforderung 15 Nutzen: *Niedrig* Umfang: *Mittel*

4. Fokus

Während der Arbeit soll auch eine konkrete Verbesserung an J-PaD erreicht werden. Da aber die Zahl der gefundenen Befunde die Möglichkeiten weit übersteigt, werden in diesem Kapitel diejenigen Befunde ausgewählt, die direkt behoben werden sollen.

4.1. Befundübersicht

Alle identifizierten Befunde lassen sich in Abbildung 4.1 zusammenfassen. Die durch die Aufwand- und Nutzen-Kategorie entstehende 3x3-Matrix wurde dabei ampelartig eingefärbt. Das heißt Befunde mit wenig Aufwand aber viel Nutzen sind grün, während Befunde, die viel Arbeit aber nur geringen Nutzen bewirken, rot markiert sind. Dazu kommt die vom Kunden getroffene dreistufige Einschätzung: Ein fettgedruckter Befund bedeutet die Einstufung „*Wichtig*“, ein normal gedruckter „*Mittel*“ und ein grau gedruckter Befund „*Unwichtig*“. Allgemein lag der Fokus des Kunden eher auf der Umsetzung der Dokumentationsperspektive aus der Rolle des Dokumentierers, der Rest wurde eher als „nice to have“ empfunden. Auffällig ist insbesondere, dass die Einschätzung „hoher Nutzen“ bei Befunden eher selten vertreten ist und keine großen Ausfälle aufgetreten sind. Der Großteil der Befunde wurde mit wenigstens mittleren Aufwand bewertet, da die Zeit-Grenze für mittleren Aufwand (1 Woche) auch wegen der begrenzten Zeit der Arbeit vergleichsweise klein ausgefallen ist.

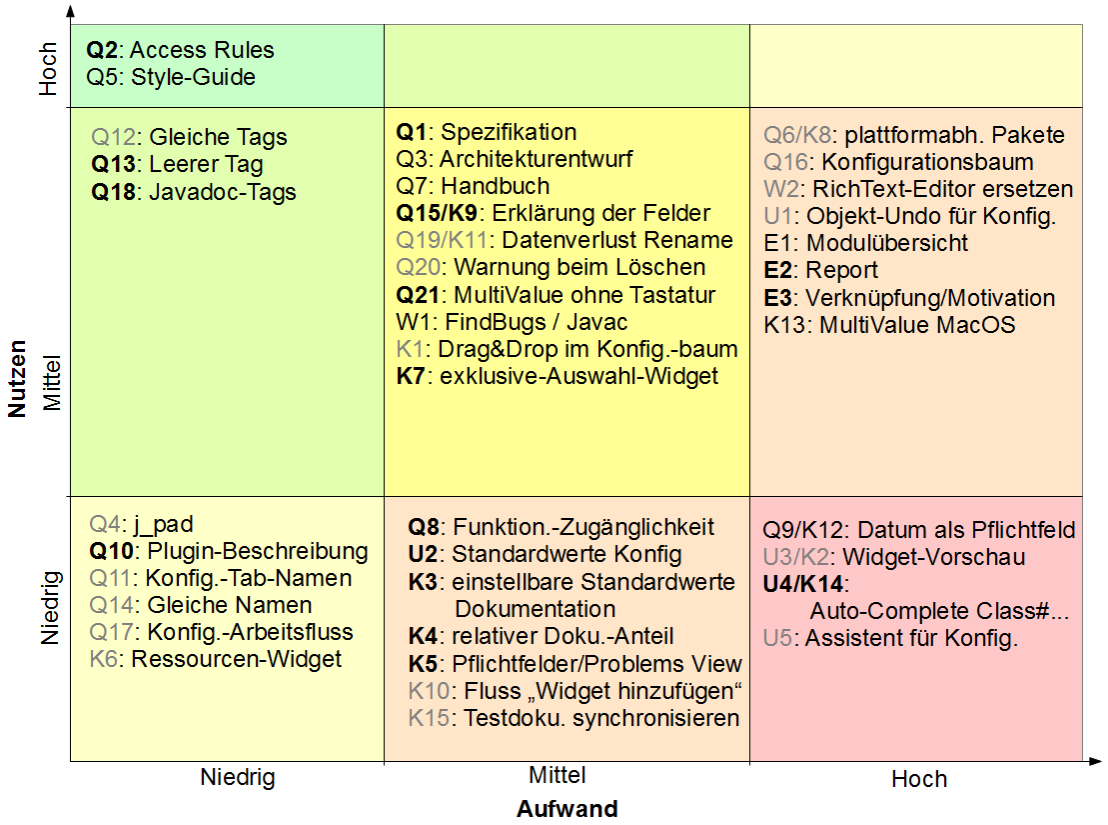

Abbildung 4.1.: Übersicht über alle Befunde und ihre Einschätzung

4.2. Auswahl

Nun muss eine sinnvolle Auswahl an Befunden getroffen werden, die im Rahmen dieser Arbeit umsetzbar sind. Hierzu wird die Implementierung auf zwei Phasen verteilt:

1. Kernimplementierung à etwa 20 Arbeitstage
2. Optionales à etwa 10 Arbeitstage

Um diese Zeit zu füllen, wurden möglichst „grüne“ Befunde ausgewählt, d. h. das Verhältnis Nutzen zu Aufwand sollte möglichst groß sein. Sehr einfach zu behebende und die vom Kunden als eher wichtig eingestuften Befunde wurden dabei bevorzugt. Die ausgewählten Befunde werden hier kurz aufgelistet:

4.2.1. Kernimplementierung

- Qualitätenbaum-Befund 2: Access Rules
- Qualitätenbaum-Befund 5: Style-Guide

- Qualitätenbaum-Befund 13: Leerer Tag
- Qualitätenbaum-Befund 18: Javadoc-Tags
- Qualitätenbaum-Befund 4: j_pad
- Qualitätenbaum-Befund 10: Plugin-Beschreibung
- Qualitätenbaum-Befund 17: Konfiguration-Arbeitsfluss
- Qualitätenbaum-Befund 1: Spezifikation
- Qualitätenbaum-Befund 15/Kundenanforderung 9: Erklärung der Felder
- Qualitätenbaum-Befund 21: MultiValue ohne Tastatur
- Werkzeug-Befund 1: FindBugs- / javac-Befunde
- Kundenanforderung 7: Widget für exklusive Auswahl
- Qualitätenbaum-Befund 8: Zugänglichkeit zur Funktionalität
- Kundenanforderung 3: einstellbare Standardwerte für Dokumentation

4.2.2. Optionales

- Kundenanforderung 4: relativer Dokumentationsanteil
- Eigene Idee 3: Verknüpfung/Motivation

5. Umsetzung

Im Nachfolgenden werden die fokussierten Befunde nochmals aufgelistet. Hier werden die resultierenden Änderungen erläutert, die während dieser Arbeit entwickelt wurden.

5.1. Kernimplementierung

5.1.1. Einstellungskorrekturen

Die Befunde Qualitätenbaum-Befund 2: Access Rules, Qualitätenbaum-Befund 4: j_pad, Qualitätenbaum-Befund 10: Plugin-Beschreibung und Qualitätenbaum-Befund 5: Style-Guide ließen sich erwartungsgemäß leicht umsetzen, da sie sich lediglich auf Änderungen von Einstellungen beziehen. Die durchgeführten Verbesserungen sind dabei allerdings nicht für den Benutzer sichtbar und auch nur schwer in Auszügen darstellbar. Da es sich ja auch nur um einige geänderten Einstellungen handelt, wird hier auf eine nähere Darstellung verzichtet.

5.1.2. Kleinere Befunde

Qualitätenbaum-Befund 18: Javadoc-Tags und Qualitätenbaum-Befund 21: MultiValue ohne Tastatur wurden korrigiert. Die Javadoc-Tags verschwinden nun nicht mehr einfach und das MultiValue-Widget ist ohne Tastatur benutzbar. Dieses geänderte Verhalten ist allerdings nur schwer darstellbar, weswegen auch hierauf verzichtet wird.

5.1.3. Qualitätenbaum-Befund 13: Leerer Tag

Es wurde der Fehler behoben, dass ein leerer Tag trotz möglicher Folge eines Datenverlusts nicht gewarnt wird. Hierzu zeigt Abbildung 5.1 die Konfigurationsansicht vor der Verbesserung und Abbildung 5.2 die Konfigurationsansicht nach der Verbesserung.

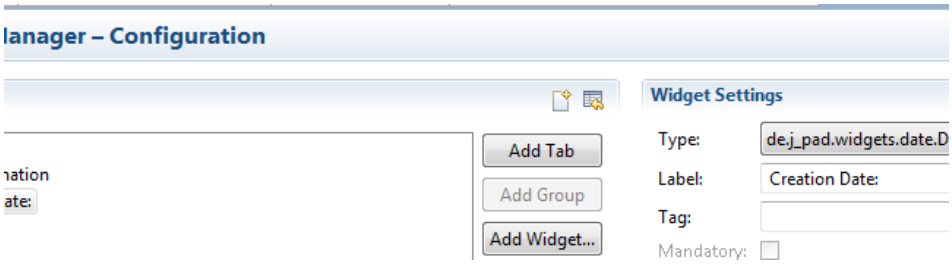

Abbildung 5.1.: Darstellung eines leeren Tags zuvor, ohne Warnung

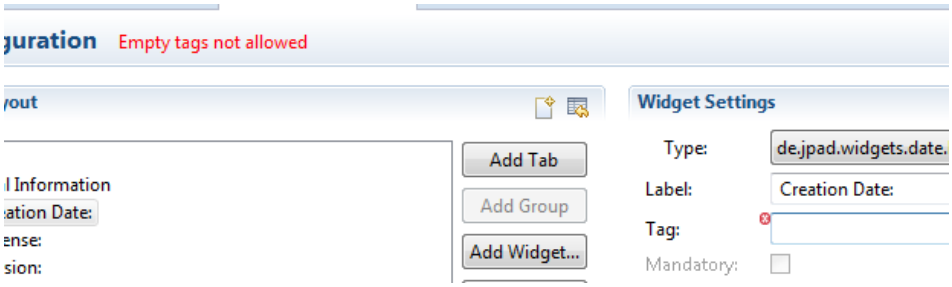

Abbildung 5.2.: Darstellung eines leeren Tags nach der Verbesserung, mit Warnung

5.1.4. Qualitätenbaum-Befund 17: Konfiguration-Arbeitsfluss

Es wurde die Irritation behoben, dass beim Hinzufügen von Widgets der Konfigurationsbaum nicht ausgeklappt und das neue Widget nicht selektiert wird. Hierzu zeigt Abbildung 5.1 die Konfigurationsansicht vor der Verbesserung und Abbildung 5.2 die Konfigurationsansicht nach der Verbesserung.

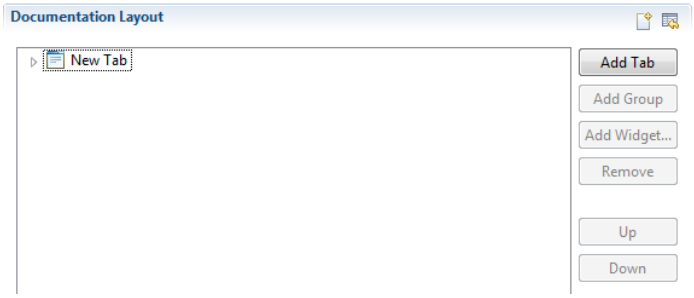

Abbildung 5.3.: Darstellung der Konfigurationsansicht nach dem Hinzufügen eines Widgets vor der Verbesserung

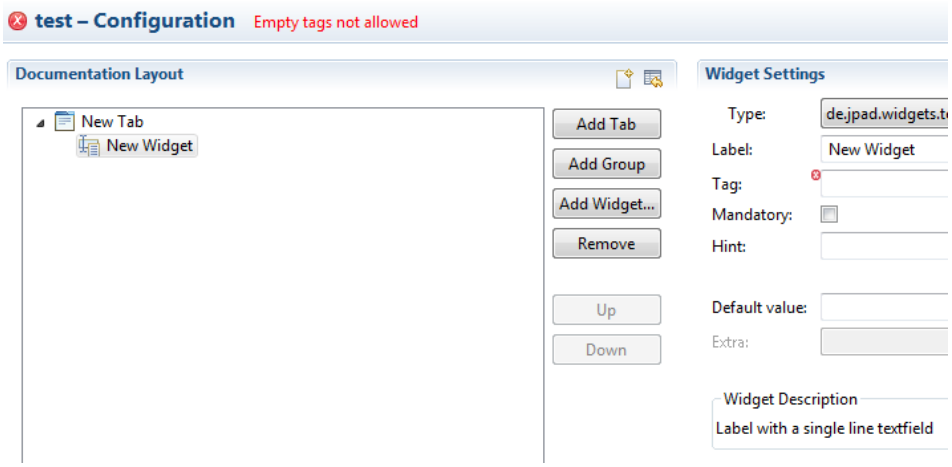

Abbildung 5.4.: Darstellung der Konfigurationsansicht nach dem Hinzufügen eines Widgets nach der Verbesserung

5.1.5. Qualitätenbaum-Befund 1: Spezifikation

Es wurde ein leicht zugängliches Spezifikationsdokument in HTML verfasst und die passenden Inhalte sowohl aus [Kir12] und dieser Arbeit übernommen. Dabei bietet diese Lösung für die Projektgröße von J-PaD gleich mehrere Vorteile; HTML ist leicht änderbar, viele Entwickler sind mit dem Bearbeiten vertraut und das Inhaltsverzeichnis kann wie beim Textsatz automatisch generiert werden. Ein kurzer Auszug aus dem Inhaltsverzeichnis wird in Abbildung 5.5 dargestellt.

Abbildung 5.5.: Beispielhafter Auszug aus dem Inhaltsverzeichnis der Spezifikation

5.1.6. Qualitätenbaum-Befund 15/Kundenanforderung 9: Erklärung der Felder

Es wurde die Möglichkeit geschaffen, zusätzlich zu einer Bezeichnung eines Eingabe-Widgets auch eine Erklärung in Form eines Hints einzustellen. Dieser wird immer dann angezeigt, wenn man mit der Maus eine kurze Zeit lang auf ein Element zeigt. Um die Konfigurierbarkeit zu ermöglichen, musste zusätzlich ein Datenfeld „Hint“ in die Konfigurationsmaske und die Schema-XML eingefügt werden.

Ein Beispielszenario zeigt die Einstellung eines Hints in der Konfigurationsansicht in Abbildung 5.6 und die daraus resultierende Dokumentationsansicht in Abbildung 5.7.

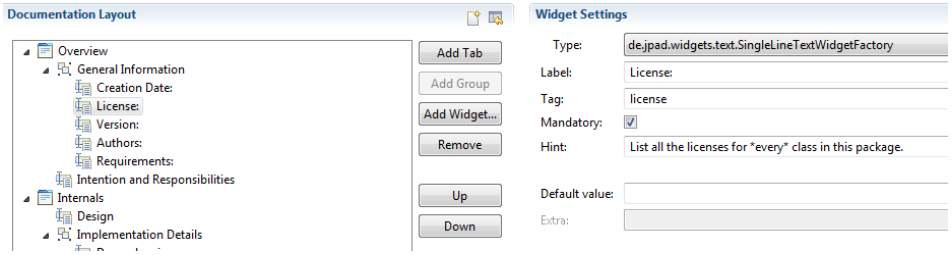

Abbildung 5.6.: Konfiguration eines Hints von einem Widget

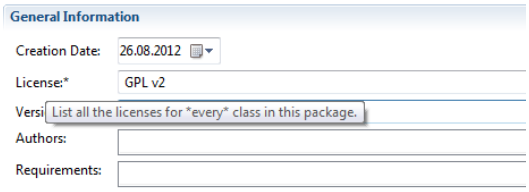

Abbildung 5.7.: Darstellung eines wie in Abbildung 5.6 konfigurierten Widgets mit Hint

5.1.7. Kundenanforderung 3: einstellbare Standardwerte für Dokumentation

Es wurde zusätzlich noch die Möglichkeit geschaffen, einem Widget einen Standardwert zuzuweisen. Immer wenn keine Eingabe für ein Element erfolgt, wird dieser Standardwert automatisch in der Dokumentationsansicht und der Javadoc-Paketinformation eingefügt. Um die Konfigurierbarkeit zu ermöglichen, musste zusätzlich ein Datenfeld „Standardwert“ in die Konfigurationsmaske und die Schema-XML eingefügt werden.

Ein Beispielszenario zeigt die Einstellung eines Standardwerts in der Konfigurationsansicht in Abbildung 5.6 und die daraus resultierende Dokumentationsansicht in Abbildung 5.7.

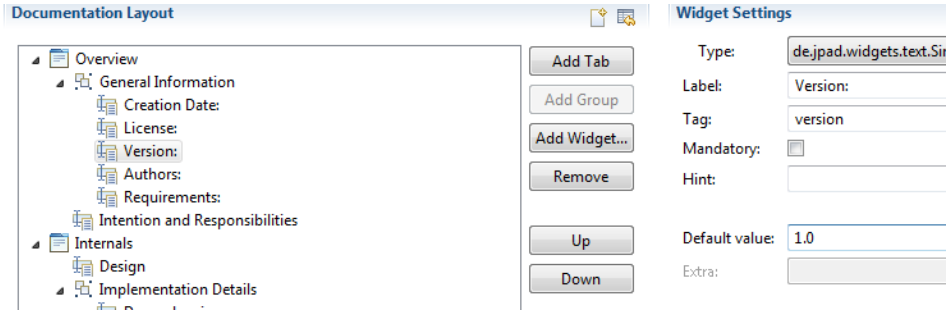

Abbildung 5.8.: Konfiguration eines Standardwerts von einem Widget

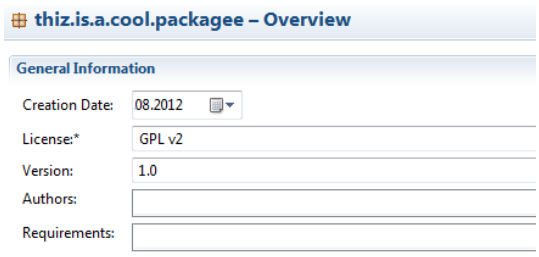

Abbildung 5.9.: Darstellung eines wie in Abbildung 5.8 konfigurierten Widgets ohne vorherige Eingabe mit Standardwert

5.1.8. Werkzeug-Befund 1: FindBugs- / javac-Befunde

In Anlehnung an Tabelle 3.1 wurden die Werkzeug-Befunde in Tabelle 5.1 erneut zusammengetragen und mit dem Zustand vor der Bearbeitung verglichen. Pakete, an denen keine relevanten Änderungen stattgefunden haben, wurden dabei nicht betrachtet. Es zeigt sich in fast allen Fällen eine Reduktion der Befunde. Insbesondere die Warnungen des Java-Compilers haben dank des durchgängigen Style-Guides stark abgenommen. Für die meisten restlichen Befunde gilt, dass sie entweder hier nicht relevant (z. B. wird Derefenzierung ohne `null`-Prüfung bemängelt, obwohl die Programmlogik dies ausschließt) oder nicht schwerwiegend (z. B. undokumentierte leere Codeblöcke) sind.

	javac			FindBugs		
Paket	vorher	nachher	Differenz	vorher	nachher	Differenz
`de.jpad.configManager`	328	24	−304	37	25	−12
`de.jpad.gui`	182	6	−176	2	2	±0
`de.jpad.widgetAPI`	141	29	−112	21	10	−11
`de.jpad.widgets`	416	17	−399	79	62	−17

Tabelle 5.1.: Anzahl der Werkzeug-Befunde im Vorher-Nachher-Vergleich

5.1.9. Kundenanforderung 7: Widget für exklusive Auswahl

Es wurde ein zusätzliches Widget neu entwickelt. Dieses ermöglicht eine exklusive Auswahl aus mehreren, vorher festgelegten Optionen mittels sogenannter „Radio Buttons“. Um die freie Konfigurierbarkeit zu erreichen, musste zusätzlich ein Datenfeld „Extra“ für zusätzliche Konfigurationsinformationen in die Konfigurationsmaske und die Schema-XML eingefügt werden.

Ein Beispielszenario zeigt die Konfigurationsansicht des neuen Widgets in Abbildung 5.10 und die daraus resultierende Dokumentationsansicht in Abbildung 5.11.

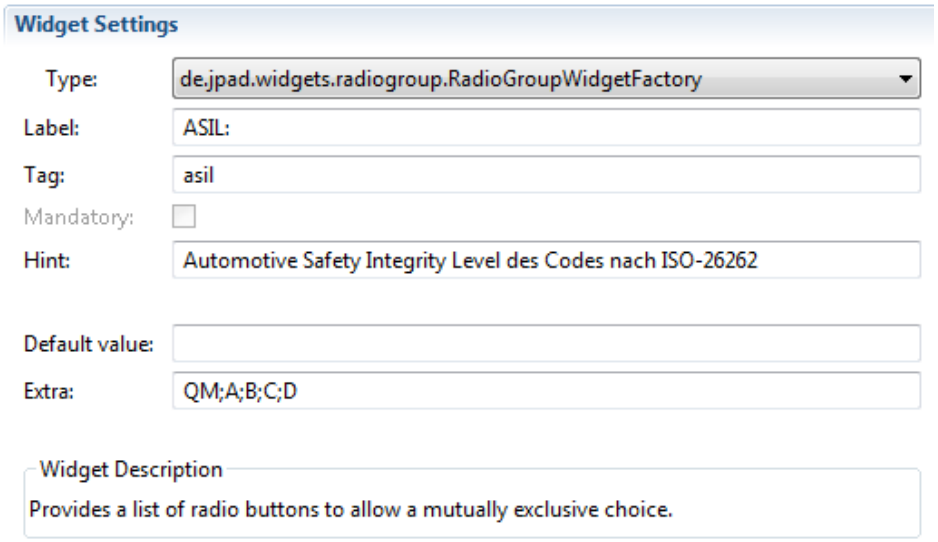

Abbildung 5.10.: Konfiguration eines Widgets für exklusives Auswahl

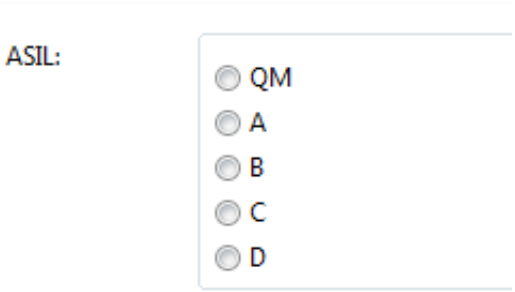

Abbildung 5.11.: Darstellung eines wie in Abbildung 5.10 konfigurierten Widgets für exklusives Auswahl in der Dokumentationsmaske

5.2. Optionales

5.2.1. Kundenanforderung 4: relativer Dokumentationsanteil

Es wurde eine Anzeige für den relativen (und absoluten) Dokumentationsanteil entwickelt. Diese bietet einerseits die Übersicht, wie viele Pakete in einem Projekt tatsächlich dokumentiert wurden; andererseits, wie viele Eingabefelder im gesamten Projekt tatsächlich Text enthalten. Diese Informationen sind natürlich mit Vorsicht zu genießen, da hierbei nicht die Qualität der vorliegenden Dokumentation bewertet werden kann.

Abbildung 5.12 zeigt beispielhaft eine Anzeige für den Dokumentationsfortschritt.

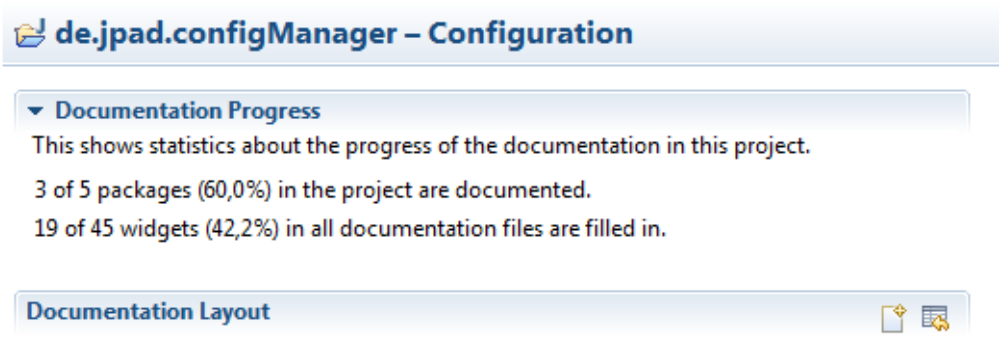

Abbildung 5.12.: Darstellung eines relativen Dokumentationsanteil in der Konfigurationsansicht

5.3. Übrig gebliebene Befunde

Die etwas ambitionierteren Befunde „Qualitätenbaum-Befund 8: Zugänglichkeit zur Funktionalität“ und „Eigene Idee 3: Verknüpfung/Motivation“ konnten aufgrund des umfangreicheren Eingriffs und der Zeitgrenze nicht zufriedenstellend gelöst werden.

6. Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Werkzeug zur Paket-Dokumentation in Java verbessert. Es wurde zunächst eine Vision des gewünschten Ergebnisses festgehalten. Darauf aufbauend wurde ein allgemeines, möglichst umfangreiches Verfahren für die Analyse der Software entwickelt. Dieses bezieht sich sowohl auf Ist- und Soll-Zustand als auch auf die Aspekte der Vision. Zusätzlich wurde ein Bewertungsschema entwickelt, um die extrahierten Befunde nach mehreren Gesichtspunkten gewichten zu können. Ebenso wurde eine Form zur übersichtlichen Darstellung gewählt. Neben einer Darstellung aller Befunde wurde auch eine durchgängig verwendete Darstellungsform jedes einzelnen Befundes entwickelt. Ähnlich wie in einem Ticketsystem, ermöglicht dies die Befunde übersichtlich darzustellen und zu identifizieren.

Das abstrakte Vorgehen wurde auf das Werkzeug J-PaD angewandt. Die resultierenden Befunde wurden näher erläutert, wozu auch die Beschreibung des Problems und möglicher Lösungen zählt. Des Weiteren wurden alle Befunde zusammengefasst; anhand eines Zeitplans wurde eine möglichst sinnvolle und machbare Auswahl an Befunden getroffen, die in dieser Arbeit behoben werden sollten.

Danach wurde eine umfangreiche Erweiterung durchgeführt, die zahlreiche kleinere Fehler behoben und alle in Abschnitt 2.2 angesprochenen Aspekte verbessert hat. Zu den wichtigsten konkreten Punkten zählt die Erstellung einer Spezifikation, die Behebung von Befunden der statischen Code-Analyse und das Hinzufügen von einstellbarem Standardwert und konfigurierbarer Hint-Anzeige. Zuletzt wurden die durchgeführten Verbesserungen dargelegt und erläutert.

Ausblick

Offensichtlich beinhaltet diese Arbeit diverse Anknüpfungspunkte:

- Es sind zahlreiche Befunde offen geblieben, die zunächst als wenig nützlich oder sehr aufwendig eingeschätzt wurden. Dazu kommen zwei Befunde, die in der Implementierung zu aufwändig geworden wären. Diese Befunde könnte man direkt umsetzen oder weiterentwickeln. Dabei ließe sich womöglich eine nützlichere oder weniger aufwändige Form finden.
- Das entwickelte Verfahren zur Verbesserung kann auf andere Software angewendet, verbessert oder auf seine Allgemeingültigkeit hin analysiert werden.
- Es kann versucht werden, Qualitäten von J-PaD oder den Qualitätsunterschied durch die Verbesserung zu messen, wie z. B. die Usability in einer Benutzerstudie.

A. Anhang

Befundliste

Literaturverzeichnis

[BBK+78] B. Boehm, J. Brown, H. Kaspar, M. Lipow, G. MacLeod, M. Merrit. *Characteristics of software quality*, Band 1. North-Holland Publishing Company, 1978. (Zitiert auf Seite 15)

[BS03] K. Bittner, I. Spence. *Use case modeling*. Addison-Wesley Professional, 2003. (Zitiert auf Seite 17)

[Dro95] R. Dromey. A model for software product quality. *Software Engineering, IEEE Transactions on*, 21(2):146–162, 1995. (Zitiert auf Seite 15)

[Ecl12] Eclipse. Bug 178362: Need ability to deselect on DateTime calendar. https://bugs.eclipse.org/bugs/show_bug.cgi?id=178362, 2007–12. (Zitiert auf Seite 37)

[Goo] Google. CodePro Analytix. https://developers.google.com/java-dev-tools/codepro/doc/. (Zitiert auf Seite 18)

[ISO98] ISO 9241-11:1998 – Ergonomic requirements for office work with visual display terminals (VDTs) - Part 11 : Guidance on usability. Norm, ISO, Geneva, Switzerland, 1998. (Zitiert auf Seite 13)

[ISO01] ISO/IEC 9126-1:2001 – Software engineering – Product quality – Part 1: Quality model. Norm, ISO, Geneva, Switzerland, 2001. (Zitiert auf den Seiten 15 und 18)

[ISO05] ISO 9000:2005 – Quality management. Norm, ISO, Geneva, Switzerland, 2005. (Zitiert auf Seite 14)

[ISO11] ISO/IEC 25010:2011 – Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. Norm, ISO, Geneva, Switzerland, 2011. (Zitiert auf den Seiten 15 und 18)

[JMSSD07] N. Juristo, A. Moreno, M. Sanchez-Segura, A. Davis. Guidelines for Eliciting Usability Functionalities. *Software Engineering, IEEE Transactions on*, 33(11):744–758, 2007. (Zitiert auf Seite 16)

[Kir12] M. Kircher. *Integrierte Dokumentation für Software-Module*. Diplomarbeit, Universität Stuttgart, 2012. (Zitiert auf den Seiten 13, 16, 17, 21, 30, 31, 34, 35 und 61)

[LL10] J. Ludewig, H. Lichter. *Software Engineering – Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag, Heidelberg, 2. Auflage, 2010. (Zitiert auf den Seiten 13, 16, 17 und 18)

[Qua] Quamoco-Konsortium. Quamoco. https://quamoco.in.tum.de/. (Zitiert auf Seite 18)

[Rö12] H. Röder. *Usability Patterns - Eine Technik zur Spezifikation funktionaler Usability-Anforderungen*. Dissertation, Universität Stuttgart, Göttingen, 2012. (Zitiert auf den Seiten 13, 16, 17 und 18)

[Red00] A. Reddy. Java™ Coding Style Guide. http://www.cs.bilgi.edu.tr/pages/standards_project/java_CodingStyle.pdf, 2000. (Zitiert auf Seite 32)

[Sun99] Sun. Java Code Conventions. http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html, 1999. (Zitiert auf den Seiten 32 und 33)

[Uni] University of Maryland. FindBugs. http://findbugs.sourceforge.net/. (Zitiert auf Seite 18)

[WJKT05] S. Wagner, J. Jürjens, C. Koller, P. Trischberger. Comparing Bug Finding Tools with Reviews and Tests. In F. Khendek, R. Dssouli, Herausgeber, *Testing of Communicating Systems*, Band 3502 von *Lecture Notes in Computer Science*, S. 316. Springer Berlin / Heidelberg, 2005. (Zitiert auf Seite 44)

Alle URLs wurden zuletzt am 26.10.2012 geprüft.

Erklärung

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

(Tobias Kuhn)