Institut für Softwaretechnologie
Abteilung Software Engineering
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 39

Erfassung von Anforderungen an Software-Module

Patrick Strobel

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Jochen Ludewig
Betreuer:	Dipl.-Inf. Ivan Bogicevic
begonnen am:	01. Dezember 2012
beendet am:	27. Mai 2013
CR-Klassifikation:	D.2.2, D.2.7

Kurzfassung

In modernen Programmiersprachen werden Übersetzungseinheiten in Module zusammengefasst. Um Module dabei besser dokumentieren zu können, wurde das Werkzeug J-PaD entwickelt, das erweiterte Möglichkeiten für die integrierte Dokumentation auf Modulebene bietet. Der Einarbeitungsaufwand in J-PaD bei bestehenden Softwareprojekten ist momentan allerdings hoch, weil viele Informationen in das Werkzeug nachgetragen werden müssen. Viele der benötigten Informationen befinden sich jedoch bereits an anderen Stellen im Projekt wieder, wie etwa im Versionsverwaltungssystem.

In dieser Arbeit wird darum analysiert, wie sich die in einem Softwareprojekt anfallenden Daten verwenden lassen, um die in J-PaD verwaltete Modulbeschreibung automatisch ausfüllen zu können. Es wird untersucht, wo sich die benötigten Daten befinden und wie diese verarbeitet werden müssen. Anschließend wird beschrieben, wie J-PaD um eine Importfunktion erweitert wird, die das automatische Ausfüllen durchführt. Zuletzt wird die Importfunktion an einem Open-Source-Projekt demonstriert. Die hierbei gemachten Beobachtungen werden vorgestellt und die Resultate werden bewertet.

Abstract

In modern programming languages, compilation units are encapsulated in modules. To improve module documentation, the tool J-PaD has been developed, which gives enhanced capabilities for the integrated documentation one module-level. Unfortunately, the effort to use J-PaD in existing software projects is high at the moment, since a lot of information has to be added to the tool. However, a lot of information is already available in other places around the project, like in the version control system.

Therefore, this thesis analyses how the data that is accumulated in software projects can be used to fill out the module description held in J-PaD automatically. It is examined where the required data can be found and how it has to be processed. Afterwards it is described how J-PaD is enhanced with an import function that does automatically fill out the module description. At the end, the import function is demonstrated on an open-source project. Observations made thereby are presented and the results are evaluated.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

Moderne Programmiersprachen ermöglichen es, verschiedene inhaltlich zusammenhängende Software-Übersetzungseinheiten (z. B. Klassen und Schnittstellen) in Modulen zusammenzufassen. Wird dieses Konzept konsequent angewendet, lassen sich nicht nur Konflikte bei der Auswahl der Bezeichner vermeiden. Die Aufteilung in logisch zusammengehörende Module erleichtert die Aufgabenverteilung bei der Umsetzung des Projekts und bietet einen besseren Überblick über die Software. Der besser Überblick erleichtert wiederum die Wartung und Wiederverwendung.

Werkzeuge für die integrierte Dokumentation haben sich weit verbreitet und werden bei vielen Software-Projekten eingesetzt. Diese Werkzeuge integrieren die Dokumentation in den Quellcode und lassen sich leicht anwenden. Dadurch werden sie von den Entwicklern gut angenommen und rege genutzt. Das macht es wahrscheinlicher, dass Dokumentation und Quellcode konsistent bleiben. Im Java-Umfeld schreiben Oracles Programmierrichtlinien die Dokumentation des Programmcodes mit Javadoc vor [Ora99].

Javadoc und vergleichbare Werkzeuge bieten vielfältige Möglichkeiten, um Elemente auf der Ebene der Übersetzungseinheiten zu beschreiben. Sie zeigen jedoch Schwächen, wenn Software-Module beschrieben werden sollen: Während sich Klassen und Methoden mit vielfältigen Beschreibungselementen dokumentieren lassen, ist die Zahl der für die Modul-Dokumentation bereitgestellten Beschreibungselemente limitiert.

Um dieses Ungleichgewicht zu schließen, entwickelte Michael Kircher im Rahmen seiner Diplomarbeit das Werkzeug J-PaD [Kir12]. J-PaD nutzt das von Javadoc bereitgestellte Verfahren für die Modul-Dokumentation, erweitert es jedoch um weitere Beschreibungselemente. Die bei der Installation mitgelieferten Beschreibungselemente lassen sich den eigenen Bedürfnissen anpassen. In einer der Diplomarbeit folgenden Bachelorarbeit wurde J-PaD von Tobias Kuhn überarbeitet und erweitert, wobei das Grundkonzept aber erhalten blieb [Kuh12].

J-PaD wurde als Erweiterung für die Entwicklungsumgebung Eclipse realisiert. Um von der verwendeten Entwicklungsumgebung und der zu dokumentierenden Programmiersprache unabhängiger zu werden, wurde das Werkzeug von Ivan Bogicevic, Mitarbeiter am Institut für Softwaretechnologie, überarbeitet. Es entstand eine eigenständige J-PaD-Version, die sich unabhängig von Eclipse benutzen lässt. Um von der zu dokumentierenden Programmiersprache unabhängiger zu werden, wird die erstellte Modul-Dokumentation nun nicht mehr als Javadoc, sondern als XML-Datei gespeichert. Das langfristige Ziel sei Ivan Bogicevic zufolge, beide J-PaD-Versionen (eigenständige Version und Erweiterung) nach Abschluss dieser Bachelorarbeit wieder zusammenzuführen.

Besonders wenn J-PaD bei bereits bestehenden Projekten eingesetzt werden soll, ist der damit verbundene Aufwand momentan hoch; die oftmals nicht oder nicht ausreichend vorhandene Modul-Dokumentation muss manuell in J-PaD nachgetragen werden. Viele der benötigten Informationen sind allerdings bereits an anderen Stellen im Projekt verfügbar. So lassen sich beispielsweise Informationen über die Autoren eines Moduls aus dem Versionsverwaltungssystems beziehen.

Ziel dieser Bachelorarbeit ist es daher, J-PaD um eine Importfunktion zu erweitern, die die bereitgestellten Beschreibungselemente automatisch ausfüllt. Hierzu werden zunächst die vorhandenen Beschreibungselemente näher untersucht und Datenquellen ermittelt, die für die betreffenden Beschreibungselemente die benötigten Informationen liefern können. Im Anschluss an diese Analyse wird J-PaD um die Importfunktion erweitert und die implementierte Importfunktion an einem größeren Software-Projekt erprobt und demonstriert.

Gliederung

Die Gliederung der Arbeit orientiert sich an der Aufgabenstellung:

Kapitel 2 – Grundlagen gibt einen Überblick über das Thema und erläutert zentrale Begriffe.

Kapitel 3 – Beschreibungselemente gibt einen Überblick über die von J-PaD standardmäßig angebotenen Beschreibungselemente. Verfügbare Datenquellen, die zu diesen Beschreibungselementen die nötigen Informationen liefern können, werden vorgestellt. Schließlich wird für jedes Beschreibungselement näher untersucht, durch welche Datenquellen es automatisch ausgefüllt werden kann.

Kapitel 4 – Implementierung geht auf die Aspekte der Implementierung ein. Es wird ein Überblick über die Architektur und den Entwurf gegeben und auf die implementierten Anbindungen an Datenquellen eingegangen. Zuletzt wird darauf eingegangen, wie sich J-PaD um weitere Datenquellenanbindungen erweitern lässt.

Kapitel 5 – Demonstration: An einem größeren Software-Projekt wird die implementierte Importfunktion erprobt und demonstriert. Die dabei gemachten Beobachtungen und die Qualität der Ergebnisse werden vorgestellt.

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

2 Grundlagen

Starke und Hruschka entwickelten eine Vorlage für die Architekturdokumentation [SH11a]. Sie sehen eine Architektur dabei als das Zusammenspiel von Bausteinen an. Mit dem Begriff des Bausteins abstrahieren die Autoren von zum Teil programmiersprachen-abhängige Konstrukte wie Übersetzungseinheiten oder Module. Das erlaubt es, viele Aspekte der Vorlage auf J-PaDs Modulbeschreibung zu übertragen. Für jeden Baustein werden gemäß der Vorlage verschiedene Teile beschrieben. Die Beschreibung sieht beispielsweise Teile vor, in denen die Aufgabe, die Schnittstellen oder die Entwurfsentscheidungen für den jeweiligen Baustein beschrieben werden.

In einer Umfrage haben Forward und Lethbridge verschiedene Praktiker unter anderem befragt, welche Werkzeuge sie bei der Dokumentation als nützlich ansehen [FL02]. Dabei wurden Textverarbeitungswerkzeuge am häufigsten genannt. Etwa 54 % der befragten Teilnehmer empfanden diese Werkzeuge als nützlich. An zweiter Stelle der als nützlich angesehenen Werkzeuge folgen bereits Werkzeuge für die integrierte Dokumentation: 51 % der Teilnehmer nannten Javadoc und ähnliche Werkzeuge. In einer weiteren Frage wurden die Teilnehmer befragt, ob sie der Aussage zustimmen, die Dokumentation enthalte viele Informationen, die aus dem Quellcode extrahiert werden können. Dieser Aussage stimmten 22 % voll und 37 % teilweise zu.

Während Forward und Lethbridge ihre Umfrage unabhängig vom eingesetzten Prozessmodell durchführten, versuchten Stettina und Heijstek Informationen über die Dokumentation bei Agilen-Prozessen zu erhalten [SH11b]. Sie befragten Softwareentwickler, wie viel Zeit sie für die Dokumentation aufbringen, durch welche Werkzeuge sie bei der Dokumentation unterstützt werden und wie zufrieden sie mit ihrer Dokumentation sind. Die meisten der befragten Teilnehmer brachten täglich weniger als 15 Minuten für die Dokumentation auf. Gleichzeitig gaben beinahe die Hälfte der Teilnehmer an, dass die Dokumentation teilweise oder überwiegend nicht ausreichend sei. Mit 84 % bzw. 83 % wurden Versionsverwaltungssysteme und Issue-Tracker eingesetzt. Dabei wurden beide Werkzeuge von den Teilnehmern im Durchschnitt als sehr hilfreich angesehen, der Issue-Tracker jedoch als etwas hilfreicher. Als weitere unterstützende Werkzeuge wurden u. a. Terminplaner (62 %) und Scrum-Werkzeuge (48 %) angesehen. Die Autoren konnten zudem beobachten, dass Wiki-Systeme für die Dokumentation oft eingesetzt werden.

Entwurfsmuster, wie sie etwa im Katalog von Gamma et al. zusammengefasst wurden, werden oft verwendet. Sie erleichtern es, wiederverwendbaren Quellcode zu schreiben und bestehenden Quellcode zu verstehen [GHJV04]. Um Entwickler auf verwendete Entwurfsmuster hinweisen zu können, werden Verfahren erforscht, die in verschiedenen Dokumenten

nach Entwurfsmustern suchen. So stellten Bergenti und Poggi ein System vor, das in UML-Diagrammen Entwurfsmuster erkennt [BP00]. Ihnen zufolge sei die Mustererkennung jedoch nicht immer zuverlässig möglich, da Entwurfsmuster häufig leicht abgewandelt verwendet werden und sich nur schwer formalisieren lassen. Tsantalis et al. stellten hingegen einen Algorithmus vor, der den Code von Java-Programmen nach verwendete Entwurfsmuster untersucht [TCSH06]. Ihren Algorithmus evaluierten sie an drei Open-Source-Projekten und konnten dabei für alle unterstützten Muster eine Genauigkeit (Precision) von 100 % erzielen. Von den insgesamt 128 in den Open-Source-Projekten vorhandenen Mustern konnten lediglich sechs Muster nicht erkannt werden.

Unter dem Schlagwort „Mining Software Repositories“ werden Verfahren erforscht, mit denen sich aus s. g. Software Repositories gezielt Informationen extrahieren lassen, die den Softwareentwicklungsprozess unterstützen können. Der Begriff des Software Repositories bezeichnet dabei die bei der Softwareentwicklung anfallenden Daten. Das sind neben dem Quellcode und der separaten Dokumentation u. a. auch Tickets in einem Issue-Tracker oder die Historie im Versionsverwaltungssystem [XTLL09]. In diesem Bereich ist die Arbeit von Marcus und Maletic interessant. Sie untersuchten, wie sich Quellcode und separate Dokumentation nachträglich automatisch verknüpfen lassen [MM03]. Dabei wendeten sie Methoden aus der Informationsrückgewinnung an, um die Ähnlichkeit zwischen zwei Dokumenten – Quellcode und weitere Dokumente – zu messen.

2.1 Begriffe

In diesem Abschnitt werden einige Begriffe erläutert, die in der Ausarbeitung verwendet werden.

Beschreibungselement: Bezeichnet ein Element, mit dem eine bestimmte Information in einem Kopfkommentar klassifiziert und von anderen Informationen abgegrenzt wird. Beschreibungselemente haben in Javadoc die Form „`@Beschreibungskement`“.

Datenquelle: Oberbegriff für Werkzeuge und Daten, die bei einer Softwareentwicklung eingesetzt werden bzw. dabei entstehen. Demnach wäre das bei einem Projekt eingesetzte Versionsverwaltungssystem Subversion oder ein in Java geschriebener Quellcode jeweils eine Datenquelle.

Integrierte Dokumentation: Bezeichnet, angelehnt an [LL10, S. 463], die Teile einer Dokumentation, die in Form von (Kopf-)Kommentaren im Quellcode vorliegen.

Issue-Tracker: Ein Werkzeug, mit dem Aufgaben und gemeldete Fehler verwaltet werden können. Für jede Aufgabe oder jeden Fehler kann ein s. g. Ticket angelegt werden, das neben einer ausführlichen Beschreibung auch weitere Informationen (z. B. eine für die Bearbeitung verantwortliche Person oder die Priorität) enthält. Für das Werkzeug sind diverse Synonyme geläufig, wie z. B. „Bugtracker“ oder „Ticketing-System“.

J-PaD: Sofern nicht anderes gesagt, wird mit J-PaD die von Ivan Bogicevic überarbeitete, eigenständige Version gemeint (da diese Version die Grundlage dieser Arbeit bildet).

Modul: Bezeichnet, angelehnt an die IEEE-Definition, einen logisch abtrennbaren Teil eines Programms [IEE10, S. 229]. Module erlauben es demgemäß, mehrere Übersetzungseinheiten logisch zusammenzufassen. Wird in vielen Sprachen, so auch in Java, auch als „Paket“ bezeichnet.

Separate Dokumentation: Bezeichnet, angelehnt an [LL10, S. 463], die Teile einer Dokumentation, die sich außerhalb des Quellcodes befinden. Dazu gehören Textdokumente ebenso wie Einträge in einem Issue-Tracker.

Übersetzungseinheit: Bezeichnet die kleinste eigenständige Software-Einheit (compilation unit), die von einem Compiler übersetzt werden kann. Das entspricht typischerweise einer einzelnen Quellcode-Datei; in Java also einer einzelnen Klasse oder Schnittstelle.

Versionsverwaltungssystem: Meist mit „VCS“ abgekürzt. Ein Werkzeug, mit dem verschiedene Versionen oder Arbeitsstände von (Quellcode-)Dateien verwaltet werden können. Für jede im System verwaltete Änderung lassen sich zusätzliche Informationen (z. B. der die Änderung durchführende Benutzer, das Datum der Änderung oder eine Änderungsbeschreibung) speichern. Für das Werkzeug sind diverse Synonyme geläufig, wie z. B. „Quellcodeverwaltungssystem“ oder „Konfigurationsverwaltungssystem“.

3 Beschreibungselemente

Dieses Kapitel stellt die in J-PaD verfügbaren Beschreibungselemente vor und geht auf die Datenquellen ein, die zum automatischen Ausfüllen der Beschreibungselemente (d. h. zum Importieren eines bestehenden Projekts) herangezogen werden können. Hierfür wird zunächst jedes Beschreibungselement in eine von insgesamt sieben Kategorien eingeteilt. In Abschnitt 3.1 werden diese Kategorien vorgestellt und auf einige in der jeweiligen Kategorie enthaltene Beschreibungselemente kurz eingegangen. Anschließend werden in Abschnitt 3.2 die Datenquellen vorgestellt, die, unabhängig von konkreten Beschreibungselementen, als Datenlieferanten überhaupt in Frage kommen. In der Analyse in Abschnitt 3.3 wird dann jedes Beschreibungselement, nach Kategorie geordnet, detailliert vorgestellt und auf die für dieses Beschreibungselement tatsächlich in Frage kommenden Datenquellen untersucht.

3.1 Überblick

Angelehnt an die Dokumentation einiger Beschreibungselemente von Ivan Bogicevic [Bog12] und den im Quellcode von J-PaD enthaltenen Kommentaren, lassen sich die in J-PaD angebotenen Beschreibungselemente in folgende sieben Kategorien einordnen:

Die Beschreibungselemente der Kategorie **Inhalt & Struktur** informieren über den Namen und die Aufgabe des Moduls. In einem weiteren Beschreibungselement wird zudem der qualifizierte Modulname angegeben, über den sich die Modulbeschreibungen eindeutig identifizieren und deren Quellcode zuordnen lassen.

In der Kategorie **beteiligte Parteien** werden Personen und andere Module aufgelistet, die mit dem Modul in Verbindung stehen. Das sind zum einen Personen, die das Modul bearbeiten oder für das Modul Verantwortungen übernehmen. Zum anderen werden auch Personen und andere Module aufgelistet, die von dem Modul – dessen Entwicklung, Wartung oder (korrektem) Verhalten – abhängig sind.

Die Beschreibungselemente der Kategorie **Referenzen** verknüpfen die Modulbeschreibung mit den konkreten Artefakten. Diese Artefakte sind neben den Quellcode-Dateien des Moduls auch weitere Dokumente oder Internetseiten, die zusätzliche Informationen über das Modul beinhalten. Auf eingesetzte Hilfsmittel wird ebenso eingegangen.

Die Beschreibungselemente der Kategorie **Historie** geben einen kurzen Überblick über die Entwicklungsgeschichte des Moduls. Dieser Überblick beinhaltet das (geplante oder tatsächliche) Erstellungsdatum sowie Informationen über stattgefundene Reviews. Ferner lassen

sich in weiteren Beschreibungselementen offene Punkte und die Planung der zukünftigen Entwicklung beschreiben.

In der Kategorie **Umfeld** werden Informationen genannt, die auf die Verwendung des Moduls und dessen Stellung innerhalb der Software eingehen. Das beinhaltet die Beschreibung der Schnittstellen und Invarianten. Eine Angabe zur Priorität erlaubt den Vergleich mit anderen Modulen.

In der Kategorie **Status** lässt sich der Zustand der Reviews von Spezifikation und Modul speichern. Es lässt sich hier festlegen, ob ein Review noch nicht stattgefunden hat, momentan stattfindet oder bereits stattgefunden hat.

Als letzte Kategorie bietet **Metriken** Beschreibungselemente an, in denen die Resultate von Metriken abgelegt werden. Neben Metriken, die für die Berechnung auf dem Quellcode operieren, sind in dieser Kategorie auch Beschreibungselemente für geplante Werte vorhanden. Die geplanten Werte erlauben den Vergleich mit den berechneten Werten.

3.2 In Frage kommende Datenquellen

Um die Analyse allgemeingültig und unabhängig von konkreten Datenformaten und eingesetzten Werkzeugen zu halten, werden in diesem und dem nachfolgenden Abschnitt Datenquellenkategorien betrachtet. Dies ist nötig, da für eine bestimmte Aufgabenstellung verschiedene Werkzeuge verfügbar sind, die sich hauptsächlich in der internen Verarbeitung und Speicherung der Daten unterscheiden. Die Art der dem Benutzer angebotenen Funktionen und Daten sind hingegen gleich oder sehr ähnlich. Beispielsweise gibt es auf dem Markt sehr viele verschiedene Versionsverwaltungssysteme, bei denen aber die selben Grunddaten (Historie mit Autoren etc.) abgefragt werden können.

Bevor Datenquellen ausgewählt werden können muss geklärt werden, welche Daten bei möglichst allen Softwareprojekten vorhanden sind. Hier lässt sich mit hoher Sicherheit nur der Quellcode nennen. Er ist darum das einzige Dokument, aus dem sich die Struktur des Softwareprojekts und damit die Aufteilung in die einzelnen Module exakt und verlässlich ablesen lässt. Das automatische Ausfüllen der Beschreibungselemente muss daher immer vom Quellcode ausgehen, auch wenn für einige Beschreibungselemente Datenquellen verwendet werden, die mit dem Quellcode selbst nicht direkt in Verbindung stehen. Es sind somit nur Datenquellen geeignet, die sich mit dem Quellcode verknüpfen lassen und damit Rückschlüsse auf die Module erlauben, auf die sich die ausgelesenen Daten beziehen.

Diese Überlegungen lassen sich in drei Anforderungen zusammenfassen, die von den Datenquellen erfüllt werden müssen:

1. **Relevant:** Die von den Datenquellen gelieferten Informationen müssen für mindestens ein Beschreibungselement verwendbar sein oder zu einem besseren Ergebnis beitragen.
2. **Verfügbar:** Die den Datenquellen zu Grunde liegenden Daten müssen bei möglichst allen Softwareprojekten vorhanden sein. D. h. die Wahrscheinlichkeit, dass die benötigten Daten vorhanden sind, sollte möglichst hoch sein.

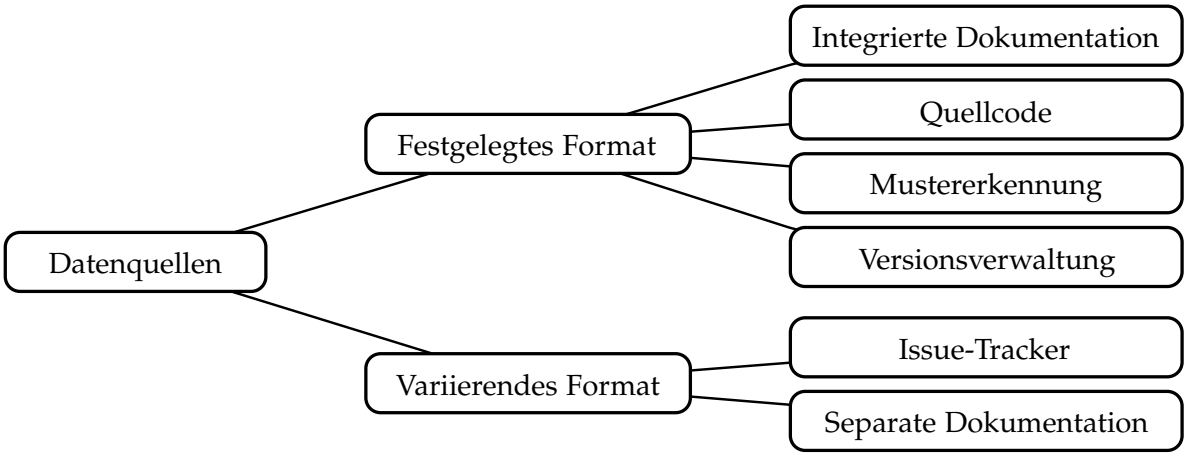

Abbildung 3.1: Übersicht über in Frage kommende Datenquellen.

3. **Nachverfolgbar:** Die von den Datenquellen bereitgestellten Informationen müssen sich mit Teilen des Quellcodes verknüpfen lassen.

Die Relevanz der ermittelten Datenquellen wird in der Analyse in Abschnitt 3.3 ausführlich dargestellt. Deshalb soll in den beiden folgenden Unterabschnitten nicht näher auf die Relevanz eingegangen und stattdessen auf den Analyseabschnitt verwiesen werden.

Abbildung 3.1 gibt einen Überblick über die nachfolgend näher betrachteten Datenquellen.

3.2.1 Festgelegtes Format

Viele Datenquellen bauen auf Daten auf, deren Format systembedingt oder durch allgemeine und projektübergreifende Richtlinien vorgegeben ist. Zu diesen Daten zählt etwa der Quellcode, dessen Format durch die verwendete Programmiersprache festgelegt wird. Durch die praktisch immer gegebene Verfügbarkeit und die ohnehin gegebene Nachverfolgbarkeit des Quellcodes, sind alle auf dem Quellcode aufbauende Datenquellen wichtige Datenlieferanten. Sie sollen daher nachfolgend als erstes betrachtet werden.

Die Richtlinien der eingesetzten Programmiersprachen (bei Java etwa [Ora99]) schreiben oft ein bestimmtes Format zur Dokumentation von Schlüsselemente (Klassen, Methoden u. ä.) vor. Mit speziellen Werkzeugen kann auf diese **integrierte Dokumentation** zugegriffen und eine separate Dokumentation generiert werden. In einer 2002 in der Industrie durchgeführten Umfrage gaben 51 % der befragten Mitarbeiter an, dass sie diese Werkzeuge und die mit ihnen erstellten Dokumente als besonders nützlich empfinden [FL02]. Für die integrierte Dokumentationen existieren verschiedene, teils programmiersprachen-spezifische Werkzeuge mit unterschiedlichem Funktionsumfang. Weit verbreitet ist das programmiersprachen-

unabhängige Doxygen[1] und das bei der Entwicklung in Java meist standardmäßig eingesetzte Javadoc[2]. Die Analyse soll sich daher auf die von diese beiden Werkzeugen angebotenen Daten beschränken.

Aus dem **Quellcode als Ganzes** lassen sich mit geringem Aufwand weitere Informationen extrahieren. Die Verzeichnisstruktur, in der die einzelnen Quellcodedateien abgelegt sind, spiegelt in vielen Programmiersprachen die Modulstruktur wider und lässt sich damit für weitere Untersuchungen heranziehen. Der Quellcode selbst lässt sich nach bestimmten Schlüsselwörtern durchsuchen, die Aufschluss über Zugriffe auf andere Übersetzungseinheiten geben oder dabei helfen, den Quellcode zu kategorisieren. Viele Basismetriken stützen sich auf Daten, die sich leicht durch Zählen bestimmter Elemente im Quellcode ermitteln lassen. Diese Basismetriken wiederum geben einen Einblick in das Modul und werden von weiteren (Pseudo-)Metriken als Grundlage verwendet.

Über eine genaue Analyse der statischen Struktur des Quellcodes und des Laufzeitverhaltens können **Mustererkennung**s-Werkzeuge auf verwendete Entwurfsmuster hinweisen. Viele Entwurfsmuster lassen den Entwicklern Freiraum bei bei der Umsetzung. Sie lassen sich daher nicht oder nur schwer formal spezifizieren und lassen sich deshalb im Quellcode nicht mit absoluter Sicherheit bestimmen (vgl. [BP00]). Weil Entwurfsmuster häufig zentrale Einheiten identifizieren (z. B. eine abstrakte Fabrik oder ein Erbauer), lässt sich die Information über verwendete Muster aber an vielen Stellen nutzen.

Nach persönlicher Einschätzung gehören **Versionsverwaltungssysteme** (VCS) zu den bei den meisten Softwareprojekten eingesetzten Werkzeugen. Eine von Stettina und Heijstek durchgeführte Studie zur Dokumentation und zum Werkzeugeinsatz in der agilen Softwareentwicklung [SH11b] bestätigt diese Einschätzung: Der Studie zufolge wurden Versionsverwaltungssysteme bei 84 % der untersuchten Projekte eingesetzt. Es ist also eine hohe Verfügbarkeit gegeben. Versionsverwaltungssysteme nehmen den Benutzern die Aufgabe ab, die verschiedenen Revisionen des Projekts manuell zu verwalten, und stellen damit einen einheitlichen Zugriffspunkt auf die verwalteten Daten bereit. Dadurch wird, ähnlich wie beim Quellcode, ein einheitliches Format der verwalteten und bereitgestellten Daten zur Verfügung gestellt.

Die von allen Versionsverwaltungssystemen verwaltete Historie erlaubt einen einheitlichen Zugriff auf die Entwicklungsgeschichte des Projekts. Jeder Eintrag dieser Historie stellt eine durchgeführte Änderung an einer Datei oder mehreren Dateien dar. Da hierüber die einzelnen Einträge den verschiedenen Dateien zugeordnet werden, ist die Nachvefolgbarkeit sichergestellt. Jedem Eintrag wird zusätzlich der die Änderung durchführende Benutzer, das Datum der Änderung und eine kurze Änderungsbeschreibung zugeordnet. Hier ist besonders die Zuordnung der einzelnen Benutzer zu den unter der Versionsverwaltung stehenden (Quellcode-)Dateien und das jeweilige Änderungsdatum interessant. Diese Informationen liefert von allen betrachteten Datenquellen nur das Versionsverwaltungssystem zuverlässig.

[1] http://www.doxygen.org
[2] http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html

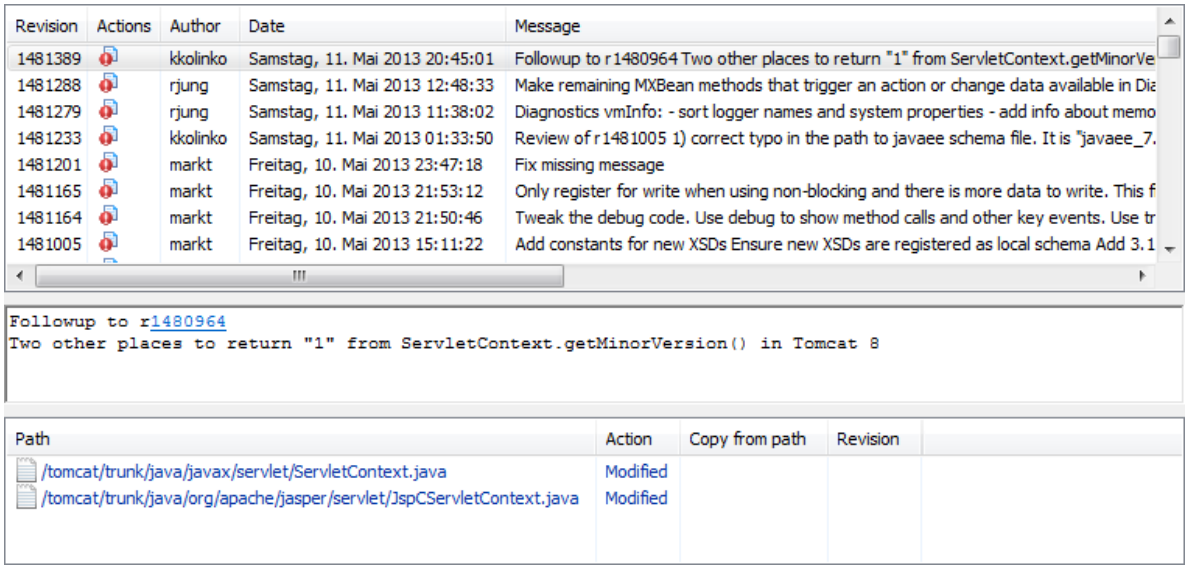

Abbildung 3.2: Ausschnitt aus der Historie eines im Versionsverwaltungssystem Subversion verwalteten Softwareprojekts.

Abbildung 3.2 zeigt einen Ausschnitt aus der Versionshistorie des Open-Source-Projekts Tomcat.

3.2.2 Variierendes Format

Die von Stettina und Heijstek durchgeführte Studie [SH11b] hat ebenfalls ergeben, dass **Issue-Tracker** mit 83 % sehr oft eingesetzt werden. Die Forderung nach der Verfügbarkeit ist damit erfüllt. Die Zahl der angebotenen Issue-Tracker ist sehr umfangreich. Online-Plattformen für quelloffene Anwendungen (z. B. Google-Code und SourceForge) bieten zudem oft Issue-Tracker an, die auf einer eigenen Implementierung basieren. Trotz der vielen angebotenen Issue-Tracker ist die Art der in den Tickets gespeicherten Informationen weitestgehend identisch. Für jedes Ticket werden mindestens folgende Informationen gespeichert: Fortlaufende Ticketnummer, Ersteller, Erstellungsdatum, Kategorie, Beschreibung, Priorität, Status und die für die Bearbeitung des Tickets als verantwortlich eingetragene Person. Abbildung 3.3 zeigt das Eingabeformular des u. a. beim Eclipse-Projekt verwendeten Issue-Trackers Bugzilla[3], über das sich ein neues Ticket anlegen lässt.

Das Format der durch die Issue-Tracker bereitgestellten Daten ist damit zwar festgelegt, die Granularität der Daten variiert hingegen stark zwischen den verschiedenen Softwareprojekten. Für welche Aufgaben Tickets angelegt und in wie weit mehrere (Teil-)Aufgaben zu einem einzelnen Ticket zusammengefasst werden, lässt sich nicht voraussagen und hängt von

[3] http://www.bugzilla.org

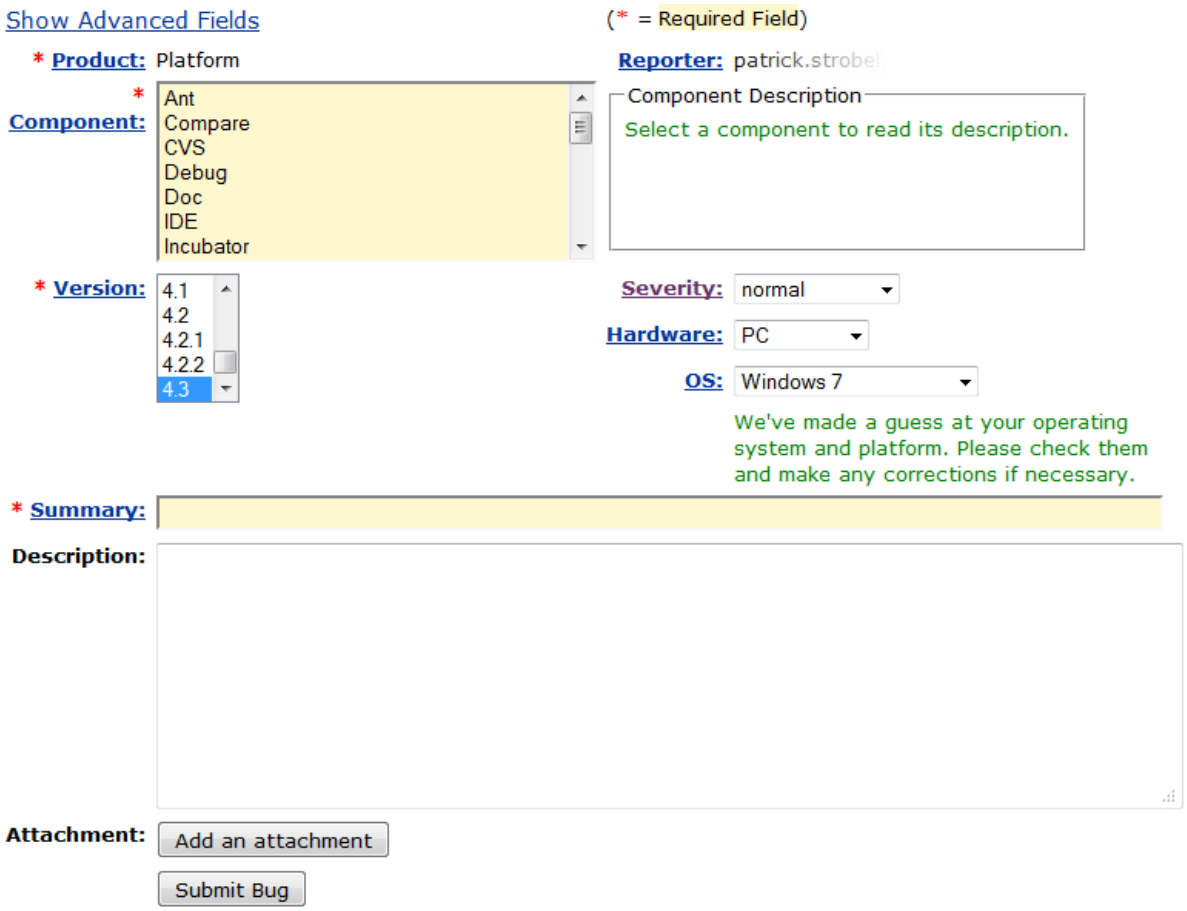

Abbildung 3.3: Beispielformular eines Issue-Trackers. Hier Bugzilla. Über die Schaltfläche „Show Advanced Fields“ lassen sich weitere Eingabefelder anzeigen (u. a. für den Status und den Verantwortlichen)

den Vorgaben der entsprechenden Organisation und der Mentalität der Projektbeteiligten ab: Zwischen *„ein Ticket für jeden größeren Arbeitsschritt“* und *„nur Tickets für Fehler“* sind beliebige Abstufungen anzutreffen.

Viele Issue-Tracker erlauben es, in das Versionsverwaltungssystem eingereichte Änderungen mit Tickets zu verknüpfen. Hierfür muss beim Einreichen die Ticketnummer in der Änderungsbeschreibung (in einem festgelegten Format) mit angegeben werden. Wird die Ticketnummer von den Benutzern konsequent mit angegeben, dann lassen sich die Tickets des Issue-Trackers über das Versionsverwaltungssystem dem Quellcode zuordnen – die Nachverfolgbarkeit ist damit gegeben.

Die **separate Dokumentation** lässt sich nur bedingt verwenden. Da Teile der separaten Dokumentation entstehen bevor Quellcode geschrieben wird (vgl. [LL10, S. 259 f.]), enthält sie Informationen, die von anderen hier betrachteten Datenquellen nicht bereitgestellt wer-

den können (unter der Annahme, dass redundante Informationen im Projekt vermieden werden). In welchem Format die separate Dokumentation angelegt wurde, wie die einzelnen Dokumente strukturiert und wie detailliert sie sind, hängt sehr stark von der Mentalität der am Projekt beteiligen Personen und den Vorgaben der Organisation ab. Besonders die stark variierende Struktur der separaten Dokumentation macht es schwer, gezielt auf bestimmte Informationen innerhalb eines Dokuments zuzugreifen.

Im Bereich der Informationsrückgewinnung (Information Retrieval) werden Techniken erforscht, mit denen sich der Zusammenhang zwischen Quellcode-Dateien und der separaten Dokumentation berechnen lässt. Marcus und Maletic stellen beispielsweise einen Technik vor, die anhand der im Quellcode vorhandenen Bezeichner und Kommentare die Ähnlichkeit zwischen Quellcode-Fragmenten und weiteren Dokumenten berechnet [MM03]. Da im Quellcode nur in den seltensten Fällen eine explizite Referenz auf die separate Dokumentation enthalten ist, ist nur durch den Einsatz solcher Techniken die separate Dokumentation nachverfolgbar.

3.3 Analyse

In den nachfolgenden Unterabschnitten werden die in J-PaD enthaltenen Beschreibungselemente näher vorgestellt.

Für jedes Beschreibungselement wird zunächst der Zweck des Elements erläutert. Dazu wird darauf eingegangen, welche Informationen in diesem Beschreibungselement gespeichert werden, welche Ziele mit diesen Informationen verfolgt werden und für welche Benutzergruppen (Entwickler, Tester oder Wartungsingenieure) diese Informationen besonders wichtig sind. Darauf folgend wird untersucht, welche Datenquellen die benötigten Informationen liefern können. Kommen mehrere Datenquellen in Frage, werden diese gegeneinander abgewogen und verglichen.

Tabelle 3.1 fasst die Ergebnisse der Analyse zusammen. Für jedes Beschreibungselement wird angegeben für welche Benutzergruppen die Informationen relevant sind und welche Datenquelle geeignet sind (○: irrelevant bzw. ungeeignet, ◐: teilweise relevant bzw. teilweise geeignet, ●: relevant bzw. geeignet). Kursiv dargestellte Beschreibungselemente sind in J-PaD ursprünglich nicht enthalten und werden vom Autor hinzugefügt.

		Benutzergruppen			Datenquellen					
Kategorie	**Beschreibungs-element**	***Entwick.***	***Test***	***Wartung***	***Int. Dok.***	***Code***	***Muster***	***VCS***	***Tracker***	***Sep. Dok.***
Inhalt & Struktur (3.3.1)	Qualif. Name	◐	●	●	○	●	○	○	○	○
	Name	◐	●	●	○	●	○	○	○	○
	Ausg. Name	◐	●	●	○	○	○	●	●	○
	Aufgabe	●	●	●	●	○	○	○	●	○
	Übersichtsdiag.	◐	◐	●	○	●	○	○	○	◐
Beteiligte Parteien (3.3.2)	Autor	◐	●	●	●	○	○	●	○	○
	Bearbeiter	◐	●	●	●	○	○	●	○	○
	Verantwortl.	●	●	●	◐	○	○	○	●	○
	Beeinflusst	●	◐	●	○	●	○	○	○	○
Referenzen (3.3.3)	Quellcode	○	●	●	○	●	○	○	○	○
	Einstiegspkt.	◐	◐	●	○	○	●	○	○	○
	Hilfsmittel	●	●	●	○	●	○	○	○	○
	Ursprung	●	●	●	●	○	○	○	◐	●
	Tests	○	◐	●	○	●	○	○	◐	●
	Extras	●	●	●	●	○	○	○	●	●
Historie (3.3.4)	Erst.-Datum	◐	○	○	○	○	○	●	○	○
	Reviews	◐	●	●	●	○	○	○	◐	●
	Zu erledigen	●	○	○	●	○	○	○	●	○
	Planung	○	○	●	●	○	○	○	○	●
Umfeld (3.3.5)	Eing. Schnitts.	●	●	●	○	●	○	○	○	○
	Ausg. Schnittst.	●	●	●	○	●	○	○	○	○
	Entwurfsmuster	◐	●	●	○	○	●	○	○	○
	Priorität	●	●	○	○	●	○	●	○	○
	Invarianten	●	●	●	◐	○	●	○	○	○

Tabelle 3.1: Übersicht über die Beschreibungselemente mit Angabe der Benutzergruppen und Datenquellen.

Kategorie	Beschreibungs-element	Benutzergruppen: Entwick.	Test	Wartung	Datenquellen: Int. Dok.	Code	Muster	VCS	Tracker	Sep. Dok.
Status (3.3.6)	Dokument	●	○	○	○	○	○	○	◐	◐
	Modul	○	●	●	◐	○	○	◐	●	◐
Metriken (3.3.7)	LOC	○	◐	◐	○	●	○	○	○	○
	LOC geplant	○	◐	◐	○	○	○	○	○	◐

Tabelle 3.1: Übersicht über die Beschreibungselemente mit Angabe der Benutzergruppen und Datenquellen *(Fortsetzung)*.

3.3.1 Inhalt & Struktur

Qualifizierter Modulname

Der qualifizierte Modulname erlaubt das eindeutige Identifizieren eines Moduls. Dazu spiegelt der qualifizierte Modulname die Schachtelung der Module wider: Der qualifizierte Modulname *„de.modulspezifikation.importer“* zeigt etwa an, dass sich das Modul *„importer“* innerhalb des Moduls *„modulspezifikation“* befindet. Das Modul *„modulspezifikation“* liegt wiederum im Modul *„de“*. Abbildung 3.4 zeigt diese Schachtelung anhand eines Verzeichnisbaums. In Java sowie in vielen anderen Programmiersprachen werden einzelne Übersetzungseinheiten durch die Verzeichnisstruktur den Modulen zugeordnet.

Der qualifizierte Modulname erleichtert es, den zugehörigen Quellcode zu finden. Liegt die Modulbeschreibung erst vor, nachdem die Module entwickelt wurden, dann ist der qualifizierte Modulname hauptsächlich für die mit dem Test oder der Wartung beauftragten Personen wichtig. Nachdem sie anhand der Modulbeschreibung die für den Test bzw. die Wartung relevanten Teile identifiziert haben, erleichtert der qualifizierte Modulname das Finden der relevanten Übersetzungseinheiten. Die im qualifizierten Namen mit angegebenen übergeordneten Module können auf weitere Module hinweisen, die beim Test oder der Wartung näher betrachtet werden sollten. Für die mit der Entwicklung des Moduls beauftragten Personen ist der qualifizierte Modulname nur von Bedeutung, wenn die Modulbeschreibung vor der Entwicklung vorliegt. Sie weißt die Entwickler dann darauf hin, wo (d. h. unterhalb welcher ggf. bereits vorhandener Module) sie ihr Modul anzulegen haben.

Die Übersetzungseinheiten werden durch eine Anweisung im Quellcode oder durch die Verzeichnisstruktur einem Modul zugeordnet. Somit lässt sich das Beschreibungselement leicht

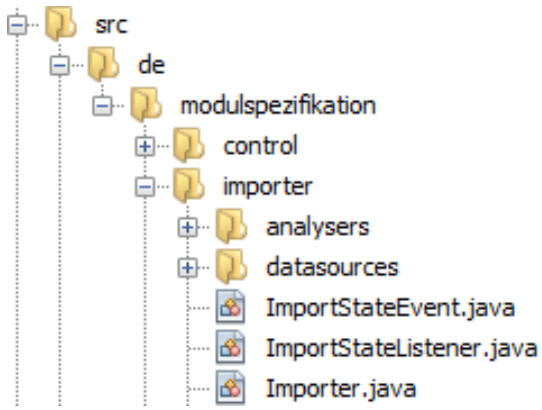

Abbildung 3.4: Modulstruktur eines in Java-Implementierten Projekts. Zu sehen ist die Schachtelung der verschiedenen Module.

und präzise ausfüllen, indem die einzelnen Quellcodedateien nach den genannten Anweisungen durchsucht werden. Spiegelt die Verzeichnisstruktur den qualifizierten Modulnamen wider, kann auf das Durchsuchen des Quellcodes verzichtet und das Beschreibungselement direkt anhand der Verzeichnisstruktur ausgefüllt werden.

Modulname

Der Modulname entspricht dem im qualifizierten Modulnamen (siehe vorhergehender Unterabschnitt) als letztes genannten Namen. Lautet der qualifizierte Modulname *„org.example.gui“*, dann heißt das Modul *„gui“*.

Da der Modulname im qualifizierten Modulnamen bereits enthalten ist, trägt er im Vergleich zum qualifizierten Modulnamen keine zusätzlichen Informationen. Der Modulname ist mehrdeutig – es können im Softwareprojekt mehrere Module mit dem selben Namen enthalten sein. Somit ist der Modulname für die Leser der Modulbeschreibung eher uninteressant. Ansonsten entsprechen die Benutzergruppen, für die der Modulname relevant ist, denen des qualifizierten Modulnamens (mit gleicher Begründung).

Nachdem das Beschreibungselement „Qualifizierter Modulname“ ausgefüllt wurde, kann der Modulname daraus extrahiert werden.

Ausgeschriebener Modulname

Die erlaubten Zeichen im (qualifizierten) Modulnamen sind beschränkt. Beispielsweise ist es nicht erlaubt, Leerzeichen in Modulnamen zu verwenden. Oft werden Modulnamen zudem abgekürzt, damit der qualifizierte Modulname nicht zu lang und unübersichtlich wird. Ob die gewählte Abkürzung für alle potentielle Leser verständlich ist, lässt sich nicht sicher

sagen. J-PaD enthält deshalb das Beschreibungselement „Ausgeschriebener Modulname“, in dem sich der vollständige und nicht gekürzte Modulname ablegen lässt.

Auch bei diesem Beschreibungselement gilt für die relevanten Benutzergruppen dasselbe wie für das Beschreibungselement „Qualifizierter Modulname“, weil der ausgeschriebene Modulname das Gleiche wie der Modulname aussagt.

Im Quellcode selbst wird zum Zugriff auf die in einem Modul enthaltenen Übersetzungseinheiten der qualifizierte Modulname verwendet. Typische Werkzeuge zur integrierten Dokumentation, wie Doxygen oder Javadoc, sehen für einen ausgeschriebenen Modulnamen keine Beschreibungselemente vor, weil diese Information in der ausführlichen Beschreibung des Moduls platziert werden kann. Das Format der ausführlichen Beschreibung aber ist dem Autor überlassen. Den ausgeschriebenen Modulname daraus auszulesen ist damit nicht möglich.

Die Änderungsbeschreibung innerhalb der Versionsverwaltung soll es den Benutzern erlauben, die von ihnen durchgeführten Änderungen zusammenzufassen und kurz zu beschreiben. Die Annahme liegt nahe, dass darum in vielen Beschreibungen auch der ausgeschriebene Modulname mit enthalten ist. Ein Ansatz beseht also darin, alle Beschreibungen der Historie-Einträge, die sich auf den Quellcode eines bestimmten Moduls beziehen, nach wiederkehrenden Mustern zu untersuchen. Wenn z. B. die Mehrzahl der am Modul *„org.example.gui“* durchgeführten Änderungen in der Beschreibung den Text *„Grafische Benutzeroberfläche“* enthält, lässt sich schlussfolgern, dass es sich dabei um den ausgeschriebenen Namen handelt.

Die Genauigkeit kann weiter erhöht werden, wenn die mit den Änderungen verknüpften Tickets im Issue-Tracker mit einbezogen werden. Das ist aber nur möglich, wenn die Benutzer für die Bearbeitung des Moduls zumindest ein Ticket angelegt haben. In diesem Fall kann in die Mustersuche der Titel des Tickets mit einbezogen werden.

Aufgabe und Zweck

Dieses Beschreibungselement gibt Aufschluss über die vom Modul gelösten Aufgaben. Das kann eine Beschreibung der vom Modul angebotenen Dienste oder der implementierten Anforderungen sein. Alles in allem soll das Beschreibungselement dem Leser einen kurzen Überblick darüber geben was das Modul leistet.

Damit ist das Beschreibungselement für alle der betrachteten Benutzergruppen relevant. Ist die Modulbeschreibung bereits vor der Implementierung vorhanden, dann informiert sie die Entwickler darüber, was und wie sie implementieren müssen. Für den Test kann die Beschreibung beim Bestimmen der Soll-Resultate helfen und für die Wartung bei der Auswahl der zu bearbeiten Module.

Eine allgemeine Modulbeschreibung erlauben alle Werkzeuge für die integrierte Dokumentation. Nach eigener Erfahrung werden die Module nur in sehr wenigen Softwareprojekten dokumentiert. Über die integrierte Dokumentation ist damit zwar ein einfacher Zugriff auf die Aufgabenbeschreibung möglich, die Information ist aber oft nicht verfügbar. Für

das Ausfüllen des Beschreibungselements sollte daher noch eine weitere Datenquelle mit einbezogen werden.

Bei einer planmäßigen Softwareentwicklung geht der Implementierung eine Spezifikation und ein Entwurf voraus, deren Ergebnisse in der separaten Dokumentation festgehalten werden. Durch Spezifikation und Entwurf wird u. a. festgelegt, was die einzelnen Module leisten sollen. Wie in Abschnitt 3.2.2 auf Seite 19 beschrieben, ist es schwer, die separate Dokumentation mit dem Quellcode zu verknüpfen. Hier wird dieser Versuch noch weiter erschwert. Welche Teile des Modulquellcodes für die Verknüpfung herangezogen werden können, lässt sich nicht allgemeingültig sagen. Das größte Problem ist aber, dass hier nicht nur zwei Dokumente miteinander verknüpft werden müssen, sondern drei: In der Spezifikation werden die Anforderungen gemeinsam und unabhängig von der späteren Umsetzung beschrieben. Im darauf aufbauenden Entwurf sind die Anforderungen nur noch implizit im Entwurf selbst enthalten, werden aber nicht mehr (oder zumindest nicht mehr vollständig) explizit genannt. Es müssten somit zunächst automatisiert die Anforderungen der Spezifikation den im Entwurf identifizierten Modulen zugeordnet werden. Anschließend würde der Entwurf mit dem Quellcode verknüpft werden. Da besonders die Verknüpfung zwischen Spezifikation und Entwurf automatisch praktisch nicht machbar sein dürfte, ist die separate Dokumentation als Datenquelle unattraktiv.

Als zusätzliche Datenquelle neben der integrierten Dokumentation kommt nur noch der Issue-Tracker in Frage. Dazu muss jedoch vorausgesetzt werden, dass von den Benutzern für jedes zu implementierende Modul ein Ticket angelegt wird. Die Ticketbeschreibung kann anschließend in das hier betrachtete Beschreibungselement übernommen werden. In der Beschreibung des Tickets wird typischerweise angegeben, was zu tun ist. Die Ticketbeschreibung gibt damit nur indirekt und unvollständig wieder, was das Ergebnis – hier das Modul – leisten soll. Der Inhalt des Beschreibungselements muss damit mit sehr hoher Wahrscheinlichkeit manuell überarbeitet werden, wenn der Issue-Tracker zum Ausfüllen mitverwendet wird.

Übersichtsdiagramm

Um einen Überblick über die Details der Softwarearchitektur zu haben, empfehlen Starke und Hruschka das Anlegen eines Übersichtsdiagramms für jedes Element, aus denen sich die Architektur zusammensetzt [SH11a]. Ziel des Diagramms ist es, die interne Struktur des jeweiligen Elements darzustellen. Betrachtet man die Architektur als Zusammenspiel der Module, dann handelt es sich bei diesen Elementen um Module. Um die Struktur eines Moduls darstellen zu können, bietet sich eine Kombination aus Komponenten- oder, wenn es sich bei den Übersetzungseinheiten um Klassen handelt, Klassen- und Paketdiagramm an. Für die Leser ist primär interessant, welche Übersetzungseinheiten sich im Modul befinden, zu dem sie die Beschreibung lesen, und wie diese miteinander interagieren. Übersetzungseinheiten aus anderen Modulen sollten darum im Übersichtsdiagramm nicht enthalten sein. Es bietet sich darum an, im Diagramm Übersetzungseinheiten, die sich im betrachteten Modul befinden, gemäß dem Komponenten- bzw. Klassendiagramm darzustellen. Wenn es Beziehungen zu Übersetzungseinheiten in anderen Modulen gibt, wird anstelle dieser

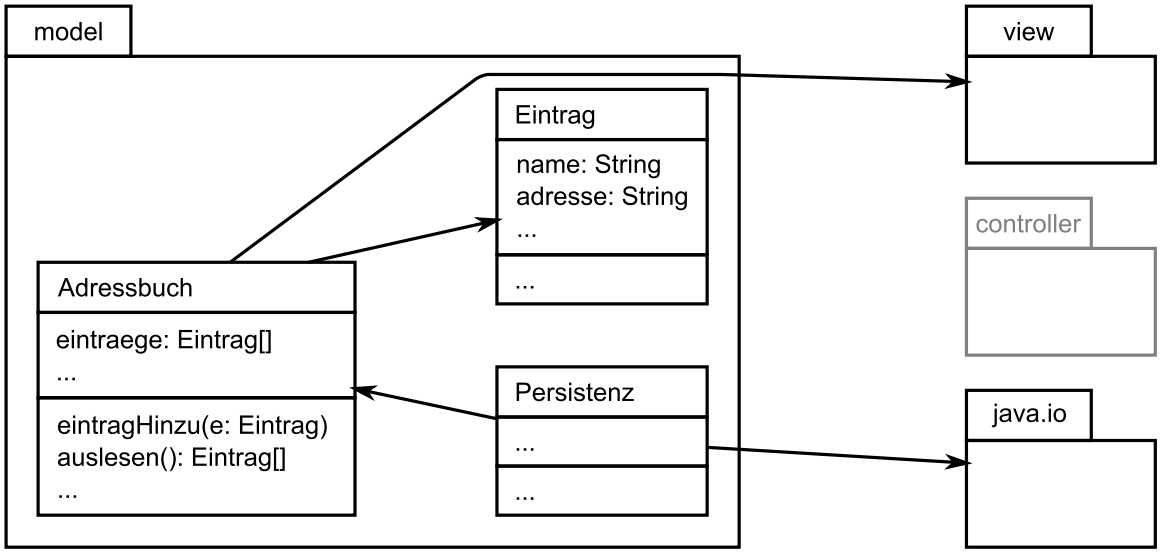

Abbildung 3.5: Beispiel eines Übersichtsdiagramms für das Modul *„model“* bei Verwendung des Model View Controller Musters und Zugriff auf Ein-Ausgabe-Funktionen. Das Modul *„controller“* wurde hier nur eingezeichnet, um das MFC-Muster deutlich zu machen und wäre in einem automatisch generierten Diagramm nicht enthalten.

Übersetzungseinheiten nur das Paket eingezeichnet. Abbildung 3.5 zeigt, wie ein solches Übersichtsdiagramm aussehen könnte.

Typischerweise wird von den Softwarearchitekten nur die grobe Struktur vorgegeben, um die Entwickler beim Feinentwurf nicht zu sehr einzuschränken (vgl. [SH11a]). Bei dieser groben Struktur kann es sich beispielsweise um die Module und deren Beziehungen handeln. Damit ist das Übersichtsdiagramm für die Entwickler weniger wichtig, da es erst aus dem Ergebnis ihrer Arbeit entsteht. Es gibt ihnen aber einen Überblick über Module, die von anderen Entwicklern erstellt wurden. Führen die Testingenieure nur Tests gegen die nach außen angebotenen Schnittstellen des Moduls durch, so ist das Übersichtsdiagramm für sie ebenfalls nur bedingt relevant (das Übersichtsdiagramm enthält auch nur Modulintern verwendete Übersetzungseinheiten). Das Zusammenspiel der einzelnen Übersetzungseinheiten ist für das Suchen und Beheben von Fehlern hingegen hilfreich.

Bei einem Komponenten-, Klassen oder Paketdiagramm handelt es sich um ein Diagramm, das die statische Struktur des Quellcodes widerspiegelt. Es kann damit ohne weiteres aus dem Quellcode generiert werden. Viele UML-Modellierungswerkzeuge bieten eine solche Funktion an. Alternativ kann sich ein Übersichtsdiagramm auch in der separaten Dokumentation befinden. Weil die Diagramme aber vergleichsweise einfach aus dem Quellcode generiert werden können und die separate Dokumentation keine sichere Quelle ist, wird der Quellcode als Datenquelle bevorzugt. Abschließend ist eine automatische Nachbear-

beitung nötig, die die Übersetzungseinheiten anderer Module zu den jeweiligen Paketen zusammenfasst.

3.3.2 Beteiligte Parteien

Autor

Beim Autor handelt es sich um die Person, die das Modul angelegt hat. Hier sollte mindestens ein Name angeben werden.

Die Entwickler eines Moduls arbeiten in der Regel eng zusammen. Damit sollte allen Entwicklern der Autor bekannt sein. Das Beschreibungselement kann ihnen allerdings helfen, wenn sie Fragen zu einem anderen Modul haben und einen Ansprechpartner suchen. Für Test- und Wartungsingenieure ist das Beschreibungselement aber relevant, denn hier kann davon ausgegangen werden, dass ihnen der Autor nicht bekannt ist.

Der Autor lässt sich aus zwei Datenquellen ermitteln. Mit einem festgelegten Beschreibungselement lässt sich in der Modulbeschreibung der integrierten Dokumentation ein Autor angeben. Diese Angabe entspricht direkt dem in J-PaD vorhandenen Beschreibungselement. Ist sie vorhanden, kann auf den Zugriff auf weitere Datenquellen verzichtet werden. Ist die Angabe nicht vorhanden, kann auf die integrierte Dokumentation der Übersetzungseinheiten des Moduls zurückgegriffen werden. Hier lassen sich die Autoren der jeweiligen Übersetzungseinheiten angeben. Die Liste der Übersetzungseinheiten-Autoren lässt sich nach Namen durchsuchen, die in allen Übersetzungseinheiten genannt werden. Aus der so ermittelten Schnittmenge lässt sich schließen, dass es sich um den Autor des Moduls handeln muss.

Neben der integrierten Dokumentation lässt sich noch das Versionsverwaltungssystem verwenden. Hier liegt die Vermutung zu Grunde, dass es sich beim Autor um die Person handelt, durch die das Modul-Verzeichnis angelegt wurde in dem sich die Übersetzungseinheiten befinden. Es muss darum eine Programmiersprache vorausgesetzt werden, bei der die Modulstruktur auf die Verzeichnisstruktur abgebildet wird (vgl. „Qualifizierter Modulname“ in Unterabschnitt 3.3.1). Ist das gegeben, kann die Historie nach dem Eintrag durchsucht werden, in dem das Modulverzeichnis erstellt wurde. Bei dem mit diesem Eintrag verknüpften Benutzer handelt es sich dann mit hoher Wahrscheinlichkeit um den Autor.

Bearbeiter und weitere Autoren

Neben dem Autor gibt es noch weitere Benutzer, die an dem Modul gearbeitet haben; diese Benutzer soll das Beschreibungselement auflisten. Neben den Entwicklern sind das auch die Wartungsingenieure.

Da dieses Beschreibungselement in seiner Aufgabe dem Beschreibungselement „Autor“ ähnlich ist, sind die Nutzer des Beschreibungselements identisch: Für Entwickler ist es nur

von Interesse, wenn sie Fragen zu einem anderen Modul habe. Test- und Wartungsingenieure hilft es, wenn sie Rückfragen zu einem bestimmten Modul haben.

Über die integrierte Dokumentation ist eine Abfrage der Bearbeiter möglich. In die integrierte Dokumentation der einzelnen Übersetzungseinheiten können sich in das Autoren-Beschreibungsfeld die Benutzer eintragen, die Änderungen durchgeführt haben. Bildet man nun die Vereinigungsmenge über alle angegebenen Autoren, erhält man alle Benutzer die am Modul gearbeitet haben.

Aus der Historie der Versionsverwaltung lassen sich die Bearbeiter noch zuverlässiger auslesen. Für jede Übersetzungseinheit und für das Verzeichnis, in dem sich das Modul befindet (vorausgesetzt es gibt einen Zusammenhang zwischen Modulen und Verzeichnissen), lassen sich alle Benutzer ermitteln, die daran Änderungen vorgenommen haben. Alle diese Benutzer zusammengefasst ergeben dann die Liste derer, die das Modul bearbeitet haben.

Verantwortliche Person

Das Beschreibungsfeld gib Aufschluss über die Person, die für das Modul verantwortlich ist. Der Dokumentation [Bog12] zufolge kann es sich hierbei um den Vorgesetzten des Autors, den Architekten oder den Autor selbst handeln.

Die verantwortliche Person kann den Nutzern der Moduldokumentation als Ansprechpartner bei weiteren Fragen dienen. Solche Fragen können z. B. auf den Entwurf oder auf die interne Struktur des Moduls gerichtet sein. Das Beschreibungselement ist damit für alle drei Benutzergruppen relevant.

Bei der verantwortlichen Person muss es sich nicht um eine Person handeln, die am Quellcode des Moduls gearbeitet hat. Damit ist das Versionsverwaltungssystem als zuverlässige Datenquelle nicht geeignet.

Wurde eine Modulbeschreibung in der integrierten Dokumentation angelegt, lässt sich daraus die verantwortliche Person ablesen. Das wird möglich, wenn davon ausgegangen wird, dass in der Modulbeschreibung ein Autor angegeben wurde der sich in der integrierten Dokumentation der Übersetzungseinheiten nicht wiederfindet. Wird eine solche Person gefunden, die zwar Autorin des Moduls aber keine Autorin einer Übersetzungseinheit ist, kann sie als Modul-Verantwortliche angesehen werden. Wie anfangs beschrieben, kann die verantwortliche Person aber auch eine Autorin sein. Daher ist die integrierte Dokumentation als Datenquelle nur bedingt geeignet.

In den Tickets des Issue-Trackers ist ein Eingabefeld vorhanden, in dem sich ein Benutzer angeben lässt, der für das Ticket verantwortlich ist. Wird von den Benutzern jeweils ein Ticket angelegt, wenn ein Modul implementiert werden soll, lässt sich das Ticket zum Ausfüllen des Beschreibungselements verwenden. Über das Versionsverwaltungssystem lassen sich die Tickets bestimmen, die mit den Änderungen am Modul verknüpft sind. Anschließend lässt sich der verantwortliche Benutzer aus dem Ticket auslesen und für das gleichnamige Beschreibungselement verwenden.

Beeinflusste Personen und Module

Jedes Modul wird meist von mehreren anderen Module verwendet. Wird die Schnittstelle oder das Verhalten eines Moduls verändert, werden dadurch andere Module beeinflusst und müssen möglicherweise überarbeitet werden. Neben anderen Modulen werden auch die Nutzer dieser Module beeinflusst. Diese beeinflussten Personen und Module werden in diesem Beschreibungselement gelistet.

Durch Änderungen werden hauptsächlich die Benutzer beeinflusst, die direkt an den Übersetzungseinheiten des Moduls arbeiten; das sind die Entwickler und die Wartungsingenieure. Die mit dem Test beauftragten Personen werden nur indirekt beeinflusst, wenn sich durch die Änderungen am Modul die Schnittstelle eines anderen Moduls ändert und einen neuen oder geänderten Test erfordert.

Um das Beschreibungselement ausfüllen zu können, müssen die Module ermittelt werden, die Übersetzungseinheiten enthalten die auf das Modul zugreifen. Diese Module lassen sich exakt bestimmten, indem der Quellcode aller Module nach Anweisungen durchsucht wird in denen Übersetzungseinheiten des betrachteten Moduls aufgerufen werden.

3.3.3 Referenzen

Quellcode

Das Beschreibungselement „Quellcode“ verweist auf das Verzeichnis, in dem sich die Implementierung des Moduls befindet.

Es kann davon ausgegangen werden, dass dieses Verzeichnis von den Entwicklern eigenständig und anhand des Modulnamens angelegt wird. Das Beschreibungselement ist damit für sie nicht relevant. Das Beschreibungselement ist aber hingegen für die mit dem Test und der Wartung beauftragten Personen von Bedeutung, wenn sie anhand der Modulbeschreibung das zu testende bzw. zu wartende Modul bestimmt haben. Insbesondere dann, wenn es in der verwendeten Programmiersprache keinen direkten Zusammenhang zwischen (qualifiziertem) Modulnamen und der Verzeichnisstruktur gibt.

Der Verweis auf das Quellcodeverzeichnis ist besonders wichtig, wenn der qualifizierte Modulname keine Rückschlüsse auf die Verzeichnisstruktur zulässt (siehe auch Beschreibungselement „Qualifizierter Modulname“). Daher sollte auch nicht der qualifizierte Modulname verwendet werden, um daraus den Pfad zum Quellcode abzuleiten. Das Beschreibungselement lässt sich vollständig ausfüllen, indem der gesamte Quellcode nach Übersetzungseinheiten durchsucht wird, die zum betrachteten Modul gehören. Für jede gefundene Übersetzungseinheit wird anschließend der Pfad zu deren Verzeichnis im Beschreibungselement abgelegt. Sollte es sich um eine Programmiersprache handeln, in der alle Übersetzungseinheiten eines bestimmten Moduls nur in einem einzigen Verzeichnis abgelegt werden dürfen, lässt sich die Suche vereinfachen. Es genügt dann ein einziger Treffer.

Einstiegspunkte

Mit Einstiegspunkt wird die Übersetzungseinheit bezeichnet, die für den Zugriff oder das Verwenden des Moduls von zentraler Bedeutung ist. Der Einstiegspunkt wird durch den internen Entwurf des Moduls bestimmt.

Starke und Hruschka empfehlen, beim Entwerfen der Softwarearchitektur den Entwicklern beim Detailentwurf einen gewissen Freiraum zu lassen [SH11a]. Auf Module und Übersetzungseinheiten übertragen bedeutet das, dass die Entwickler über den modulinternen Entwurf selbst bestimmen können. Für die Entwickler ist die Information über die Einstiegspunkte darum nur relevant, wenn sie auf andere Module zugreifen müssen. Da in der Regel alle von außen zugänglichen Übersetzungseinheiten des Moduls getestet werden, ist das Beschreibungselement für die Testingenieure ebenfalls von untergeordneter Bedeutung. Es hilft ihnen aber dabei zu entscheiden, welche Übersetzungseinheiten von ihnen besonders intensiv getestet werden müssen. Für Wartungsingenieure ist das Wissen über Einstiegspunkte wichtig, da es Aufschluss über die Übersetzungseinheiten gibt, die von einer Wartung mit hoher Wahrscheinlichkeit betroffen sind.

Zwischen Einstiegspunkten und dem Entwurf des Moduls gibt es eine enge Beziehung. Durch den Entwurf werden Übersetzungseinheiten festgelegt, denen eine zentrale Bedeutung zukommt. Diese Übersetzungseinheiten lassen sich leicht feststellen, wenn bekannte Entwurfsmuster verwendet werden. Handelt es sich bei den Übersetzungseinheiten um Klassen, dann lassen sich im Entwurfsmuster-Katalog der „Gang of Four“ aus der Beschreibung des jeweiligen Musters die zentralen Klassen ablesen [GHJV04]. Insbesondere die bei allen Mustern gegebene Darstellung der Struktur und die Beschreibung des Klienten in der Teilnehmerliste zeigen Einstiegspunkte auf. Ein Beispiel ist etwa das Entwurfsmuster „Abstrakte Fabrik“, bei der die zentrale Übersetzungseinheit diejenige Klasse ist, die diese abstrakte Fabrik implementiert. Diese Klasse ist damit ein Einstiegspunkt, wie Abbildung 3.6 an einem Ausschnitt aus der Java-API zeigt. Für jedes verwendete Entwurfsmuster muss bestimmt werden, welche Übersetzungseinheiten jeweils als Einstiegspunkte angesehen werden sollen.

Auf das automatische Ausfüllen des Beschreibungselements übertragen bedeutet das, dass sich die Einstiegspunkte über eine Entwurfsmustererkennung bestimmten lassen. Dazu muss zunächst für jedes Entwurfsmuster, das von dem Werkzeug erkannt wird, die zentrale Übersetzungseinheit – der Einstiegspunkt – bestimmt werden. Wird das Werkzeug dann auf dem Quellcode angewendet, können anhand der erkannten Entwurfsmuster die Einstiegspunkte bestimmt werden.

Hilfsmittel

Während das Modul implementiert wird, lassen sich verschiedene Hilfsmittel einsetzen, die die Arbeit erleichtern. Zum Beispiel werden häufig so genannte GUI-Builder verwendet, mit denen sich die Benutzeroberfläche grafisch erstellen lässt. Durch den Einsatz solcher Werkzeuge werden Teile des Quellcodes automatisch generiert.

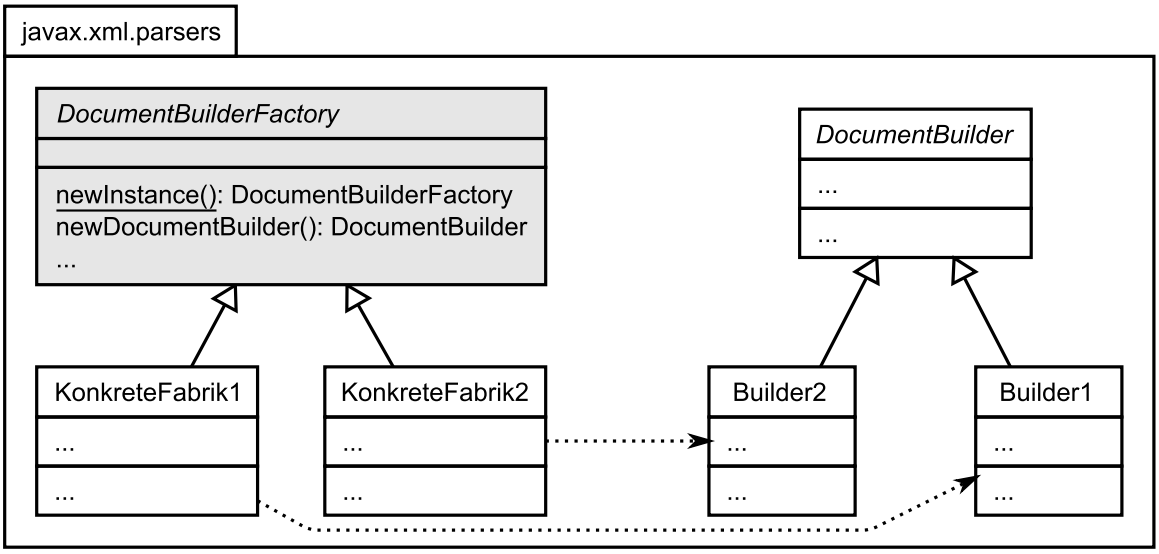

Abbildung 3.6: Ausschnitt aus der Java-API, der den Einsatz einer abstrakten Fabrik zeigt. Die abstrakte Klasse *„DocumentBuilderFactory“* bildet hier einen Einstiegspunkt für das Modul *„javax.xml.parsers“*.

Die Information über die verwendeten Hilfsmittel ist für alle betrachteten Benutzergruppen relevant. Die Entwickler wissen damit, wie sie mit dem generierten Quellcode umgehen müssen. Den Testern ist dadurch bekannt, dass der Quellcode automatisch erzeugt wurde und somit möglicherweise nicht getestet werden muss und den Wartungsingenieuren ist dadurch bekannt, welches Hilfsmittel sie verwenden müssen wenn sie Änderungen vornehmen möchten.

Da Quellcode durch die zu Hilfe genommenen Werkzeuge automatisch generiert wurde, enthält er oft Hinweise auf das eingesetzte Werkzeug. Solche Hinweise können als Kommentar vorliegen (Sinngemäß etwa *„Quellcode wurde durch Werkzeug X automatisch generiert. Nicht manuell bearbeiten.“*) oder durch eine bestimmte Struktur des Quellcodes (etwa durch ein bestimmtes Schema bei der Bezeichner-Wahl). Wie diese Hinweise im Detail ausfallen, ist stark vom jeweils eingesetzten Hilfsmittel abhängig. Damit variiert auch die Genauigkeit, mit der das jeweilige Hilfsmittel erkannt wird. Es muss also für jedes zu erkennende Hilfsmittel untersucht werden, wie es sich im Quellcode erkennen lässt.

Ursprung

Die Aufgabe des Beschreibungselements ist es, auf die Herkunft des Moduls zu verweisen. Stammt das Modul von einer dritten Person oder einer anderen Firma, sollte hierin auf diese Person bzw. Firma oder auf eine geeignete Webseite verwiesen werden. Stammt das Modul von der selben Organisation wie die Software, zu der es gehört, sollte das

Beschreibungselement auf die separate Dokumentation verweisen, aus der das Modul hervorgegangen ist. Das kann etwa die Spezifikation sein.

Das Beschreibungselement ist für alle Benutzergruppen von Bedeutung: Entwickler und Wartungsingenieure können daraus ablesen, wo sie weitere Informationen über das Modul erhalten und Tester können erkennen, ob Tests nötig sind oder wo sie Informationen über das Soll-Verhalten des Moduls erhalten.

Stammt das Modul aus einer externen Quelle, z. B. von einer anderen Firma, lässt sich die Information darüber am ehesten aus der integrierten Dokumentation ablesen. Doxygen und Javadoc bieten das Beschreibungselement `see` an, mit dem sich auf Dokumente mit weiteren Informationen verweisen lässt und das sowohl auf Modul- als auch Übersetzungseinheiten-Ebene unterstützt wird. Neben einem einfachen Text lässt sich hier auch eine URL angeben [Ora12, Hee13]. Dieses Beschreibungselement kann damit verwendet werden, um auf eine externe Webseite zu verweisen die weitere Informationen über das Modul enthält.

Dokumente, die als Grundlage für das Modul verwendet wurden, können ebenfalls mit dem `see`-Beschreibungselement angegeben werden. Alternativ lässt sich die zugehörige separate Dokumentation durch Analyse ihres Inhalts bestimmen. Ein Ansatz besteht darin, die separate Dokumentation nach Schlüsselwörtern, wie dem Modulnamen, zu durchsuchen. Abhängig von der Zahl der Treffer kann das Dokument dem Beschreibungselement „Ursprung“ hinzugefügt werden. Alternativ lassen sich Informationsrückgewinnungstechniken verwenden, um relevante Dokumente zu finden (siehe auch Unterabschnitt 3.2.2 auf Seite 19).

Auch in der Beschreibung der Tickets im Issue-Tracker können sich Informationen befinden, die bei der Entwicklung des Moduls als Grundlage (mit) verwendet wurden. Der Umfang der Ticket-Beschreibung wird meist kurz gehalten, so dass die Beschreibung für das betrachtete Beschreibungselement von geringer Bedeutung ist. Abhängig von der Art der angelegten Tickets kann es jedoch sinnvoll sein, dem Beschreibungselement einen Verweis auf das entsprechende Ticket hinzuzufügen.

Test

Den Modulen können über das Beschreibungselement „Test“ Dokumente zugeordnet werden, die mit Tests in Verbindung stehen. Diese Dokumente können Quellcode-Dateien mit Modultests sowie separate Dokumente mit Daten zu durchgeführten oder noch durchzuführenden Tests sein (z. B. Test-Protokolle oder Systemtestpläne).

Der Test findet nach der Entwicklung des Moduls statt; für die Entwickler sind die Verweise auf die Test-Dokumente deshalb irrelevant. Die Tester hingegen können die Information verwenden, wenn sie sich über bereits durchgeführte oder implementierte Tests informieren möchten. Die Verweise sind aber insbesondere für die Wartungsingenieure relevant, wenn sie die in den Tests entdeckten Fehler beheben müssen.

Modultests liegen als Quellcode vor und werden in der Regel mithilfe von Test-Frameworks erstellt. Diese Frameworks hinterlassen im Quellcode der Modultests Hinweise, über die

Listing 3.1 Ein für das Modultest-Rahmenwerk JUnit geschriebener Testfall. Mit der Annotation `Test` werden die beim Test auszuführenden Methoden gekennzeichnet.

```
public class Unittests {
	@Test
	public void testGui() {
		JPaD.main(null);
	}

	@Test
	public void executeScenarios() {
		ScenarioHandling.scenario1();
		ScenarioHandling.scenario2();
	}
}
```

ein Modultest erkannt werden kann (etwa Annotationen bei JUnit). Da sich die Modultests häufig entweder im gleichen Modul wie die getesteten Übersetzungseinheiten befinden oder den selben qualifizierten Modulnamen besitzen, ist auch eine Zuordnung zwischen einem Modul und dessen Tests möglich.

Informationen über Systemtests, wie Testpläne oder Testprotokolle, befinden sich in der separaten Dokumentation. Diese Dokumente können durch eine Analyse ihres Inhalts den jeweiligen Modulen zugeordnet werden. Dazu kann die Test-Dokumentation etwa nach dem qualifizierten Modulname durchsucht werden. Wie die Verknüpfung zwischen Modulen und Test-Dokumenten stattfindet, hängt allerdings stark vom Inhalt und Aufbau der separaten Dokumentation ab.

Ähnlich zum Beschreibungselement „Ursprung“ können die Tickets im Issue-Tracker Informationen zu den Tests enthalten, wobei auch hier der Informationsgehalt gering ist. Abhängig von der Organisation kann es darum sinnvoll sein dem Beschreibungselement Verweise auf Tickets hinzuzufügen, die mit Tests im Zusammenhang stehen.

Extras

In diesem Beschreibungselement lassen sich Verweise auf weitere Dokumente ablegen, die nicht in die beiden zuvor genannten Beschreibungselemente „Ursprung“ oder „Test“ passen. Das können etwa Dokumente sein, die weitere Informationen über die Implementierung oder zu Reviews enthalten.

Auf welche Art von Dokumenten im Beschreibungselement verwiesen wird hängt von der Organisation ab und variiert damit. Darum ist das Beschreibungselement potentiell für alle Benutzergruppen relevant.

Für die verwendbaren Datenquellen gilt dasselbe wie beim Beschreibungselement „Ursprung“. Die integrierte Dokumentation kann in einem speziellen Beschreibungselement (`see` in Doxygen und Javadoc) auf separate Dokumente mit weiteren Informationen verweisen. Der Inhalt und Aufbau der separaten Dokumentation kann ebenfalls analysiert werden,

um einen Zusammenhang zwischen dem Modul oder dessen Übersetzungseinheiten und der Dokumentation zu bestimmen. In jedem Fall müssen die Dokumente jedoch auf ihre Art untersucht werden, um unterscheiden zu können, ob der Verweis im Beschreibungselement „Ursprung“ oder „Extras“ abgelegt werden muss.

Die Tickets des Issue-Trackers können ebenfalls Informationen enthalten, die für das Verständnis des Moduls wichtig sind oder Aufschluss auf dessen Zustand geben. Darum sollen in diesem Beschreibungselement Verweise auf die dem Modul oder dessen Übersetzungseinheiten zugeordneten Tickets hinzugefügt werden.

3.3.4 Historie

Erstellungsdatum

Das Erstellungsdatum gibt an, wann das Modul angelegt wurde oder, sofern die Modulbeschreibung bereits vor der Implementierung vorliegt, wann es angelegt werden soll.

Die Relevanz des Beschreibungselements ist gering und nur gegeben, wenn die Beschreibung vorhanden ist bevor mit dem Implementieren begonnen wird. Das Beschreibungselement unterstützt die Entwickler dann bei der Entscheidung darüber, welches Modul zuerst implementiert werden sollte oder eine größere Aufmerksamkeit erhält.

Das Versionsverwaltungssystem speichert die gesamte Historie des Projekts und gibt damit auch Aufschluss darüber, wann eine bestimmte Übersetzungseinheit angelegt wurde. Um das Erstellungsdatum eines Moduls bestimmen zu können genügt es darum, die Historie vorwärts (d. h. von der ersten durchgeführten Änderung zur letzten) zu durchlaufen. Dabei werden für jeden Eintrag die neu angelegten Übersetzungseinheiten ermittelt und deren Modul mit dem hier betrachteten Modul verglichen. Stimmen beide überein, wurde der Eintrag in der Versionsverwaltung gefunden, an dem das Modul angelegt wurde. Das diesem Historie-Eintrag zugeordnete Datum entspricht dem Erstellungsdatum des Moduls.

Reviews

In diesem Beschreibungselement werden Reviews und deren Resultate gelistet, die mit dem Modul im Zusammenhang stehen. Das können Verweise auf Protokolle durchgeführter Reviews sein, es können aber auch Beschreibungen offener Punkte sein, die in einem Review festgestellt wurden.

Inwieweit das Beschreibungselement für die einzelnen Benutzergruppen relevant ist, hängt von der Organisation im Projekt ab – insbesondere davon, welche Gruppen für das Beheben der im Review festgestellten Mängel zuständig sind. Hier kann man annehmen, dass dies hauptsächlich die Aufgabe der Wartungsingenieure ist. Da die Entwickler möglicherweise aber ebenfalls mit der Überarbeitung beauftragt werden, ist das Beschreibungselement für diese Gruppe teilweise von Bedeutung. Für die Tester sind die Resultate der Reviews relevant, wenn sie die Tests daran anpassen müssen.

Moderne Entwicklungsumgebungen (IDEs) durchsuchen die Kommentare im Quellcode nach Schlüsselwörtern wie `TODO` oder `FIXME`. Die Kommentare, die diese als „Task Tags“ oder „TODO Kommentare“ bezeichneten Schlüsselwörter enthalten, werden dem Benutzer anschließend in einer Liste angezeigt. Die dahinter stehende Idee ist, dass diese Kommentare auf Stellen im Quellcode hinweisen die einer Überarbeitung bedürfen. Hujer und Holly greifen dieses Konzept auf und erweitern es um das Schlüsselwort `REVIEW` [HH11]. Mit dem Schlüsselwort `REVIEW` lassen sich Beanstandungen von Review-Gutachtern kennzeichnen, die im Quellcode als Kommentar vermerkt wurden. Wird von den Gutachtern das Schlüsselwort konsequent verwendet, lässt sich das Beschreibungselement durch Durchsuchen der Übersetzungseinheiten des Moduls nach Kommentaren, die dieses Schlüsselwort enthalten, ausfüllen.

Weitere Informationen über Reviews lassen sich möglicherweise im Issue-Tracker finden, wenn von den Benutzern für durchzuführende Reviews Tickets angelegt werden. Diese Tickets lassen sich bestimmen, in dem der Titel oder Beschreibungstext aller mit dem Modul oder dessen Übersetzungseinheiten verknüpften Tickets nach dem Schlüsselwort „Review“ durchsucht werden. Dem Beschreibungselement können anschließend Referenzen auf die gefundenen Tickets hinzugefügt werden.

Für systematisch durchgeführte Reviews wird ein Protokoll erstellt und typischerweise im Versionsverwaltungssystem abgelegt. Die separate Dokumentation sollte darum ebenfalls nach relevanten Dokumenten durchsucht werden. Die genaue Vorgehensweise hierfür variiert allerdings je nach Verzeichnisstruktur: Werden die Review-Protokolle beispielsweise in einem eigenen Verzeichnis (d. h. getrennt von der restlichen Dokumentation) abgelegt, können die Protokolle den jeweiligen Modulen durch Durchsuchen ihres Inhalts nach dem Modulnamen oder den Namen der im Modul enthaltenen Übersetzungseinheiten zugeordnet werden. Andernfalls müssen die Protokolle zuvor von der restlichen Dokumentation getrennt werden (z. B. durch Analyse des Dateinamens oder des Inhalts).

Zu erledigen

Offene und noch zu überarbeitende Punkte lassen sich im Beschreibungselement „Zu erledigen“ ablegen. Dazu gehören Stellen im Quellcode, die einer weiteren Überarbeitung bedürfen.

Es lässt sich annehmen, dass Notizen zu zu erledigenden Punkten während oder vor der Entwicklung angelegt werden. Mit Beginn der Tests sollten alle notierten Punkte erledigt sein. Damit richtet sich die Bedeutung des Beschreibungselements an die Entwickler, die diese Notizen angelegt haben und auch für deren Bearbeitung zuständig sind.

Wie bereits beim Beschreibungselement „Reviews“ erwähnt, durchsuchen moderne Entwicklungsumgebungen den Quellcode nach Kommentaren, die bestimmte Schlüsselwörter enthalten. Zumindest die beiden verbreiteten Entwicklungsumgebungen Eclipse und NetBeans unterstützen die beiden Schlüsselwörter `TODO` und `FIXME`. Hier liegt die Intention dahinter, mit `TODO` Kommentare zu versehen, die allgemein auf Stellen im Quellcode hinwiesen die überarbeitet werden sollten. `FIXME` weißt hingegen auf Stellen im Quellcode

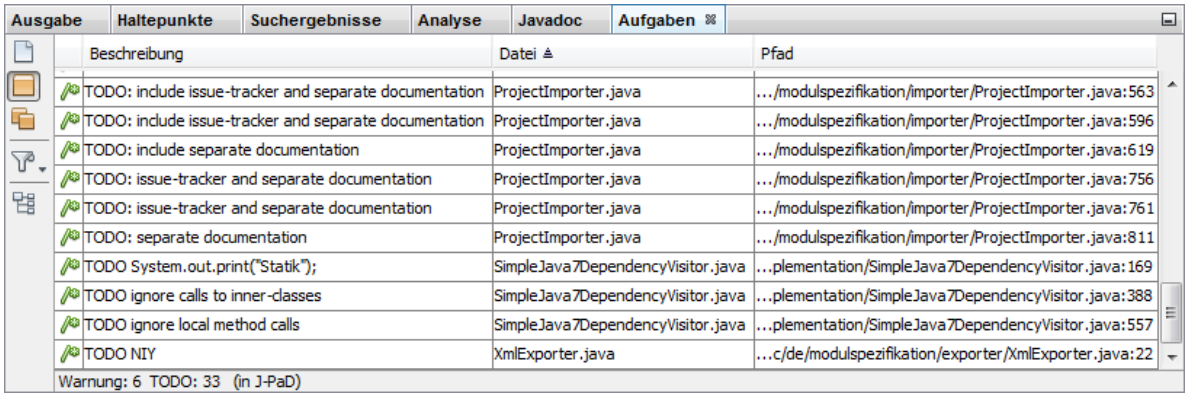

Abbildung 3.7: In NetBeans listet das Aufgaben-Fenster alle Kommentare auf, die ein `TODO` oder `FIXME` enthalten.

hin, die entdeckte und zu beseitigende Fehler enthalten. Neben diesen Schlüsselwörtern enthält der restliche Kommentar detailliertere Angaben und weitere Begründungen [HH11]. Darum lässt sich das Beschreibungselement durch Durchsuchen der Kommentare nach den beiden Schlüsselwörtern `TODO` und `FIXME` ausfüllen. Abbildung 3.7 zeigt, wie die mit den Schlüsselwörtern versehenen Kommentare in NetBeans angezeigt werden.

Je nach Projekt-Organisation werden im Issue-Tracker für zu erledigende Implementierungsaufgaben Tickets angelegt. Diese Tickets enthalten in ihrer Beschreibung weitere Informationen über noch zu implementierenden Funktionalitäten. Darum sollen dem hier betrachteten Beschreibungselement Referenzen auf die mit dem Modul oder dessen Übersetzungseinheiten verknüpften Tickets hinzugefügt werden.

Planung

Das Beschreibungselement enthält Planungen zur zukünftigen Entwicklung des Moduls. Nach [Bog12] stehen hierin Informationen zu geplanten Umbauten oder zum Ersatz des Moduls.

Damit richtet sich das Beschreibungselement vornehmlich an die Wartungsingenieure des Softwareprojekts, die ihre Arbeit anhand der abgelegten Informationen planen und durchführen können.

Doxygen und Javadoc ermöglichen es mit dem Beschreibungselement `deprecated` Elemente im Quellcode zu markieren, die veraltet sind und darum nicht mehr verwendet werden sollten. In der mit anzugebenden Beschreibung können die Benutzer die Entscheidung, warum sie das entsprechende Quellcode-Element als veraltet markiert haben, begründen und auf Alternativen hinweisen [Ora12, Hee13]. Die Beschreibung kommt damit der Intention des hier betrachteten Beschreibungselements „Planung“ sehr nahe. Es kann somit durch

Auslesen der im Modul und dessen Übersetzungseinheiten enthaltenen `deprecated`-Elemente automatisch ausgefüllt werden.

Es liegt die Annahme nahe, dass geplante Umbauten und Ersetzungen in der separaten Dokumentation festgehalten werden. Sie sollte darum als Datenquelle mit einbezogen werden. Wie die Dokumentation durchgeführt wird hängt von der Organisation ab. Es dürfte jedoch zielführend sein, wenn die in Frage kommenden Dokumente (d. h. Dokumente, die Planungen enthalten) nach dem (Qualifizierten-)Namen des Moduls oder dessen Übersetzungseinheiten durchsucht werden. Auf Dokumente, die diese Namen enthalten, kann anschließend im Beschreibungselement verwiesen werden.

3.3.5 Umfeld

Eingehende Schnittstellen

Unter eingehenden (oder auch *benötigten*) Schnittstellen werden die Schnittstellen verstanden, die ein Modul für die ordnungsgemäße Arbeit benötigt [SH11a, S. 35]. Das Beschreibungselement definiert damit auch die Abhängigkeiten des Moduls zu anderen Modulen und deren Übersetzungseinheiten. Eine detaillierte Beschreibung aller Schnittstellen befindet sich bereits in der integrierten Dokumentation der Übersetzungseinheiten. Darum genügt es, die vom Modul benötigten Schnittstellen in diesem Beschreibungselement aufzulisten und für weitere Informationen auf die detaillierte Dokumentation der jeweiligen Übersetzungseinheit zu verweisen.

Die Informationen über die eingehenden Schnittstellen eines Moduls sind prinzipiell für alle drei Benutzergruppen relevant. Entwickler können damit nachvollziehen, welche Module die von ihnen implementierten Schnittstellen verwenden und Wartungsingenieure erhalten eine Überblick über Module, die von einer Änderung an einer Schnittstelle betroffen sind. Von den Testingenieuren kann das Beschreibungselement verwendet werden um zu testende Schnittstellen ermitteln zu können.

Die Liste der eingehenden Schnittstellen lässt sich durch analysieren des Quellcodes der im Modul enthaltenen Übersetzungseinheiten aufbauen. Dazu müssen aus dem Quellcode alle Zugriffe auf Schnittstellen extrahiert werden. Alle extrahierten Schnittstellen, die sich in einem fremden Modul befinden, werden dann der Liste hinzugefügt.

Ausgehende Schnittstellen

Unter ausgehenden (oder auch *angebotenen*) Schnittstellen werden die Schnittstellen verstanden, die ein Modul für den Zugriff auf die implementierten Dienste und Funktionalitäten anderen Modulen und Übersetzungseinheiten zur Verfügung stellt [SH11a, S. 35]. Auch hier gilt wie bei den „Eingehende Schnittstellen“, dass sich detaillierte Informationen über die Schnittstellen bereits in der integrierten Dokumentation der Übersetzungseinheiten befinden. Daher sollen in diesem Beschreibungselement ebenfalls nur Schnittstellen aufgelistet und für

weiterführende Informationen auf die Dokumentation der jeweiligen Übersetzungseinheit verwiesen werden.

Die Informationen über ausgehende Schnittstellen sind ebenfalls für alle Benutzergruppen relevant. Entwickler und Wartungsingenieure werden über Schnittstellen informiert, die sie verwenden können oder überarbeiten müssen, und Testingenieure erhalten Informationen über zu testende Schnittstellen.

Die ausgehenden Schnittstellen eines Moduls entsprechen allen öffentlichen Schnittstellen der im Modul enthaltenen Übersetzungseinheiten. In Java sind damit alle öffentlichen Methoden und Attribute Teil der ausgehenden Schnittstelle. Die Liste der ausgehenden Schnittstellen lässt sich somit direkt aus dem Modul-Quellcode auslesen.

Entwurfsmuster

Entwurfsmuster stellen Lösungen für oft wiederkehrende Problemstellungen dar (vgl. [GHJV04]). Es ist darum sehr wahrscheinlich, dass beim Implementieren des Moduls bekannte Entwurfsmuster eingesetzt wurden. Die Benutzer finden sich in einem Modul schneller und bessert zurecht, wenn sie über diese eingesetzten Entwurfsmuster Bescheid wissen. Deshalb soll dieses Beschreibungselement eine Liste der verwendeten Entwurfsmuster enthalten.

Da den Modul-Entwicklern beim Feinentwurf „ihres“ Moduls oft Freiheiten gelassen werden (vgl. [SH11a]), ist davon auszugehen, dass dieses Beschreibungselement vor dem Implementieren unvollständig ist. Somit ist es für die Entwickler nur bedingt von Interesse, nämlich wenn sie auf andere Module zugreifen und deren Verhalten erschließen möchten (siehe auch Beschreibungselement „Einstiegspunkte“). Von größerer Bedeutung ist das Beschreibungselement für die mit Tests und der Wartung beauftragten Personen, da einige Entwurfsmuster (insbesondere die Verhaltensmuster nach [GHJV04]) zentrale, für den Test relevante Übersetzungseinheiten sowie Erweiterungspunkte definieren.

Das Beschreibungselement kann direkt anhand des Ergebnisses der Mustererkennung ausgefüllt werden. Die Mustererkennung analysiert den Quellcode und bestimmt die – mit einer gewissen Wahrscheinlichkeit – verwendeten Entwurfsmuster.

Priorität

Im Beschreibungselement „Priorität“ lässt sich angeben, wie wichtig das Modul im Vergleich zu den anderen Modulen ist.

Die Priorität ist für Entwickler und Testingenieure wichtig. Sie können anhand der Priorität entscheiden, welches Modul die größte Aufmerksamkeit bei der Implementierung bzw. beim Durchführen der Tests erhalten sollte.

Wie wichtig ein Modul von den Benutzern subjektiv wahrgenommen wird, lässt sich automatisch nicht bestimmen. Es kann nur nach Merkmalen Ausschau gehalten werden, die

der Wahrnehmung nahe kommen. Ein Ansatz besteht darin, die Zahl der Zugriffe auf Übersetzungseinheiten des Moduls zu bestimmen. Um den nötigen Aufwand zu verringern, können dafür näherungsweise die Anweisungen im Quellcode gezählt werden, mit denen der Namensraum eines Moduls importiert wird (in Java über die `import`-Anweisung). Es wird also ein Zusammenhang zwischen Zahl der Modulzugriffe und der Priorität angenommen.

Einen weiteren Hinweis auf die Priorität eines Moduls kann die Zahl der durchgeführten Änderungen geben. Hier liegt die Vermutung zu Grunde, dass im System wichtige Module öfters bearbeitet oder angepasst werden müssen. Die Zahl der Änderungen lässt sich aus der Historie im Versionsverwaltungssystem bestimmen, indem die am Modul oder an dessen Übersetzungseinheiten vorgenommen Änderungen gezählt werden.

Invarianten

Invarianten, die über die gesamten Lebensdauer oder Nutzung des Moduls hinweg gelten, lassen sich in diesem Beschreibungselement nennen.

Das Wissen über Invarianten ist für Entwickler wichtig, wenn sie auf fremde Module zugreifen und deren Verhalten erschließen möchten. Um während dem Testen überprüfen zu können, ob Invarianten eingehalten werden, müssen diese den Testingenieuren ebenfalls bekannt sein. Ähnliches gilt für die Wartungsingenieure, die beim Durchführen von Änderungen darauf achten müssen, dass die Invarianten weiterhin erhalten bleiben.

Doxygen unterstützt im Gegensatz zu Javadoc das Beschreibungselement `invariant`, das die eben genannte Aufgabe erfüllt und die Beschreibung von Invarianten ermöglicht [Ora12, Hee13]. Wurde die integrierte Dokumentation mit Doxygen durchgeführt, kann das hier betrachtete Beschreibungselement „Invarianten“ direkt mit den Daten der integrierten Dokumentation befüllt werden. Wie erwähnt, unterstützt Javadoc die Dokumentation von Invarianten nicht. Somit muss diese Information von den Benutzern in der allgemeinen Beschreibung abgelegt werden. Es ist darum ohne weiteres nicht möglich, aus Javadoc die Invarianten gezielt auszulesen.

Insbesondere die Verhaltensmuster (vgl. [GHJV04]) legen einige Invarianten fest, weil durch die Entwurfsmuster die Aufgaben und das Verhalten bestimmter Übersetzungseinheiten definiert wird. Beispielsweise wird durch das Entwurfsmuster Befehl der Auslöser einer Aktion von der Reaktion auf das Ereignis getrennt. Die Invariante besteht nun darin, dass die die Aktion auslösende Übersetzungseinheit die Reaktion stets einer anderen (austauschbaren) Übersetzungseinheit überlässt. Das Grundverhalten des Auslösers, nämlich das Weiterreichen von Aktionen, bleibt also immer gleich, auch wenn sich die tatsächliche Reaktion unterscheiden kann. Abbildung 3.8 zeigt dieses Verhalten an einem Beispiel aus der Java-API. Das Beschreibungselement kann deshalb ausgefüllt werden, indem im Modul nach Entwurfsmustern gesucht wird. Zuvor müssen für jedes Entwurfsmuster die Invarianten, inklusive einer Beschreibung, bestimmt werden. Die Beschreibung der Motivation, Interaktion und der Konsequenzen im Entwurfsmusterkatalog von Gamma et al. [GHJV04] geben dabei Anhaltspunkte.

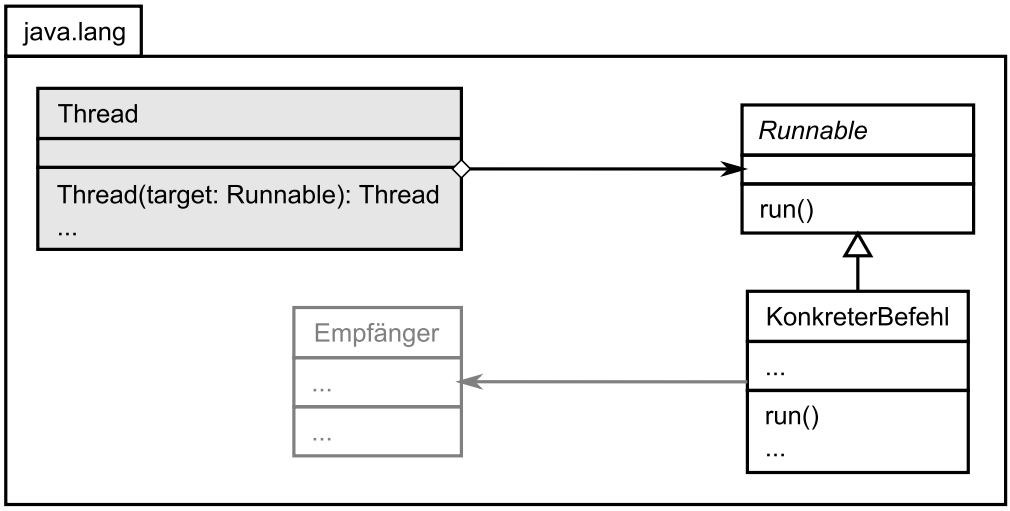

Abbildung 3.8: Ausschnitt aus der Java-API, der eine Invariante am Entwurfsmuster Befehl zeigt: Die im *„Thread“* auszuführende Tätigkeit wird in ein *„Runnable“* ausgelagert, das den variierenden Implementierungsteil enthält. Das Verhalten von *„Thread“* bleibt somit gleich, auch wenn sich die tatsächlichen Aktionen unterscheiden.

3.3.6 Status

Dokumente

Das Beschreibungselement speichert den Review-Status der Modul-Dokumentation. Das Beschreibungselement kann dabei einen der folgenden drei Zustände annehmen: „unreviewed“ (noch kein Review durchgeführt), „in_review“ (Review wurde geplant aber noch nicht abgeschlossen) oder „reviewed“ (Review wurde durchgeführt).

Die Dokumentation enthält typischerweise die Spezifikation und den Entwurf des Moduls. Daher ist der Review-Status der Dokumentation hauptsächlich für die Entwickler relevant, weil sie geneigt sein dürften mit dem Implementieren erst zu beginnen, wenn die Dokumentation geprüft wurde. Alle weiteren Phasen (Test und Wartung) finden erst später statt, so dass die zugehörigen Benutzergruppen zu diesem Zeitpunkt bereits auf ein geprüftes Dokument zurückgreifen können.

Werden für durchzuführende Reviews Tickets im Issue-Tracker angelegt, können diese zum automatischen Ausfüllen des Beschreibungselements herangezogen werden. Dafür muss zunächst das Ticket bestimmt werden, das mit den Reviews im Zusammenhang steht. Dabei ist es hilfreich, wenn die Benutzer für durchzuführende Dokumenten-Reviews eine separate Ticket-Kategorie angelegt haben. Wurde das relevante Ticket bestimmt, kann anhand dessen Status das Beschreibungselement gesetzt werden: Befindet sich das Ticket im Status „in Bearbeitung“, wird das Beschreibungselement auf „in_review“ gesetzt. Befindet sich das

Ticket im Status „geschlossen“, wird das Beschreibungselement auf „reviewed“ gesetzt. In allen anderen Fällen (etwa kein Ticket vorhanden oder noch im Status „neu“) wird das Beschreibungselement auf „unreviewed“ gesetzt.

Für durchgeführte Reviews wird im Versionsverwaltungssystem gewöhnlich ein Protokoll angelegt. Die Herausforderung besteht darin, die Modul-Dokumentation mit den Protokollen zu verknüpfen. Die genaue Vorgehensweise hierfür variiert je nach Organisation und dem Benennungsschema der Dateien und Verzeichnisse. Werden z. B. alle Protokolle der Dokumenten-Reviews in einem ausgewählten Verzeichnis angelegt, lassen sich die Protokolle den Modulen durch Durchsuchen des Inhalts nach dem Modulnamen zuordnen. Ist für das betrachtete Modul ein Protokoll vorhanden, lässt sich der Dokumenten-Status auf „reviewed“ setzen. Werden zusätzlich für die Planung der Reviews Dokumente angelegt, ist zusätzlich eine Unterscheidung zwischen „unreviewed“ und „in_review“ möglich.

Modul

Das Beschreibungselement speichert den Review-Status der Modul-Implementierung. Ähnlich dem Beschreibungselement „Dokumente“ sind drei Zustände möglich: „unreviewed“ (noch kein Review durchgeführt), „in_review“ (Review wurde geplant aber noch nicht abgeschlossen) oder „reviewed“ (Review wurde durchgeführt).

Da Code-Reviews erst nach der Implementierung durchgeführt werden, betrifft der Review-Status die nach der Entwicklung folgenden Phasen. Damit ist der Modul-Status hauptsächlich für die Test- und Wartungsingenieure von Bedeutung.

Wie beim Beschreibungselement „Reviews“ beschrieben, können Kommentare die während dem Modul-Review im Quellcode angelegt werden mit dem Schlüsselwort `REVIEW` markiert werden. Der Modul-Quellcode lässt sich nach diesen Kommentaren durchsuchen. Sind solche Kommentare vorhanden, wurde ein Review durchgeführt und der Modul-Status kann auf „reviewed“ gesetzt werden. Eine Unterscheidung zwischen „unreviewed“ und „in_review“ ist hiermit aber nicht möglich.

In diesem Zusammenhang kann weiter angenommen werden, dass die durch das Hinzufügen der Review-Kommentare durchgeführten Änderungen mit einer charakteristischen Beschreibung in das Versionsverwaltungssystem eingestellt werden (z. B. *„Review des Moduls x durchgeführt“*). Die in der Versionsverwaltung verfügbare Modul-Historie kann in diesem Fall nach charakteristischen Schlüsselwörtern – wie „Review“ – durchsucht werden. Wurden Treffer in der Historie gefunden, kann davon ausgegangen werden, dass bereits Reviews durchgeführt wurden. Das Beschreibungselement kann anschließend ebenfalls auf „reviewed“ gesetzt werden.

Werden für alle Aufgaben (darunter auch durchzuführende Reviews) Tickets im Issue-Tracker angelegt, lassen sich diese zum Ausfüllen des Beschreibungselements verwenden. Weil angenommen werden kann, dass die Review-Gutachter Befunde direkt in Quellcode-Kommentaren ablegen, ist eine Verknüpfung zu eventuell vorhandenen Review-Tickets gegeben (da die Ticketnummer in der Beschreibung mit angegeben werden kann, wenn die

Änderungen ins Versionsverwaltungssystem eingestellt werden). Der Status des Review-Tickets kann direkt auf den Modul-Status abgebildet werden: Der Ticket-Status „neu“ entspricht dem Modul-Status „unreviewed“, „in Bearbeitung“ entspricht „in_review“ und „geschlossen“ entspricht „reviewed“.

Analog zum Beschreibungselement „Dokumente“ kann die separate Dokumentation nach Review-Protokollen und -Planungen durchsucht werden. Anhand der gefundenen Dokumente ist es anschließend möglich, den Modul-Status zu setzen. Weitere Informationen zu einem möglichen Vorgehen und der Schwierigkeiten hierbei finden sich in der Analyse des Beschreibungselements „Dokumente“.

3.3.7 Metriken

Anzahl Code-Zeilen

Das Beschreibungselement speichert die Anzahl aller Quellcode-Zeilen im Modul. Das heißt, es werden alle Code-Zeilen der einzelnen Übersetzungseinheiten des Moduls aufaddiert. Kommentare und Leerzeilen werden dabei mitgezählt .

Die Anzahl der Code-Zeilen sind für die Test- und Wartungsingenieure teilweise relevant, weil sie einen groben Überblick über die Größe des Moduls bieten. Die Größe kann bei der Planung und Priorisierung der Tests bzw. der Wartung mit einbezogen werden. Für die Entwickler ist das Beschreibungselement weniger interessant, da die Zahl der Code-Zeilen erst nach der Entwicklung feststeht und sie mehr am Verhalten der Module interessiert sind als an deren Umfang.

Die Anzahl der Quellcode-Zeilen lässt sich durch einfaches Zählen der Zeilenumbrüche im Code bestimmen.

Geplante Anzahl Code-Zeilen

Die geplante Anzahl der Quellcode-Zeilen spiegelt die geplanten Größe des Moduls wider.

Das Beschreibungselement ist wie das Beschreibungselement „Anzahl Code-Zeilen“ für Entwickler wenig interessant. Es gibt den Testern aber einen groben Einblick über den Fortschritt der Modulentwicklung und den Wartungsingenieuren die Information, ob die aktuelle Größe der Planung entspricht oder (bei Überschreitung) Gegenmaßnahmen nötig sind.

Die betrachteten Werkzeuge für die integrierte Dokumentation (Doxygen, Javadoc) unterstützen keine Beschreibungselemente, in denen die geplanten Code-Zeilen notiert werden können; vermutlich, weil diese Planungen bereits stattfinden, bevor implementiert wird. Um das hier betrachtete Beschreibungselement automatisch ausfüllen zu können, kann darum nur die separate Dokumentation verwendet werden. Da diese Dokumente keinem

vorgegebenen Aufbau folgen (und sich die geplante Anzahl zudem im Fließtext „verstecken“ dürfte), lässt sich an dieser Stelle keine allgemeine Vorgehensweise nennen.

4 Implementierung

Dieses Kapitel geht auf die Umsetzung der Analyseergebnisse und die Anpassung von J-PaD ein. Abschnitt 4.1 beschreibt die zentralen Entwurfsentscheidungen. In Abschnitt 4.2 werden anschließend die Datenquellen vorgestellt, die von der erweiterten J-PaD-Version unterstützt werden. Diese beiden Abschnitte sind damit insbesondere für Entwickler interessant, die weitere Datenquellenkategorien hinzufügen möchten.

Schließlich gehen die Abschnitte 4.3 und 4.4 darauf ein, wie sich J-PaD an die eigenen Bedürfnisse anpassen und um Anbindungen an weitere Datenquellen erweitern lässt. Für Entwickler, die die bereits vorhandenen Datenquellenkategorien um weitere Anbindungen erweitern möchten, ist damit Abschnitt 4.4 interessant. Abschnitt 4.3 enthält zudem Informationen, die auch für die Benutzer von J-PaD interessant sind.

4.1 Architektur

Wie in Abschnitt 3.2 erläutert, existieren für bestimmte Aufgabenstellungen verschiedene Werkzeuge, deren angebotene Funktionen sich jedoch stark ähneln. Darum muss auch beim Entwurf zwischen konkreten Datenquellen und Datenquellenkategorien unterschieden werden. Eine Datenquellenkategorie ist beispielsweise *„Versionsverwaltungssystem“*, während eine konkrete Datenquelle das Versionsverwaltungssystem „Subversion“ ist. Für einen Überblick über die verschiedenen Datenquellenkategorien sei an dieser Stelle auf Abbildung 3.1 auf Seite 17 verwiesen.

Da die Zahl der konkreten Datenquellen einer gegebenen Kategorie umfangreich und schwer vorhersagbar ist (es existieren etwa sehr viele verschiedene Versionsverwaltungssysteme), sollte das spätere Anbinden an weitere konkrete Datenquellen mit wenig Aufwand möglich sein. Dieser Aufwand lässt sich verringern, wenn das Anbinden an konkrete Datenquellen und das darauf folgende Verarbeiten der ausgelesenen Daten getrennt werden.

Hierfür sieht der Entwurf für jede Datenquellenkategorie drei Komponenten vor. Abbildung 4.1 zeigt die drei Komponenten und deren Interaktionen am Beispiel der Datenquellenkategorie *„Versionsverwaltungssystem“*:

Für jede Datenquellenkategorie existiert eine *Schnittstelle*, die alle für die Anbindung an konkrete Datenquellen benötigten Funktionen definiert. Diese Schnittstelle bildet damit eine Schnittmenge über alle von den Datenquellen der jeweiligen Kategorie angebotenen Funktionen.

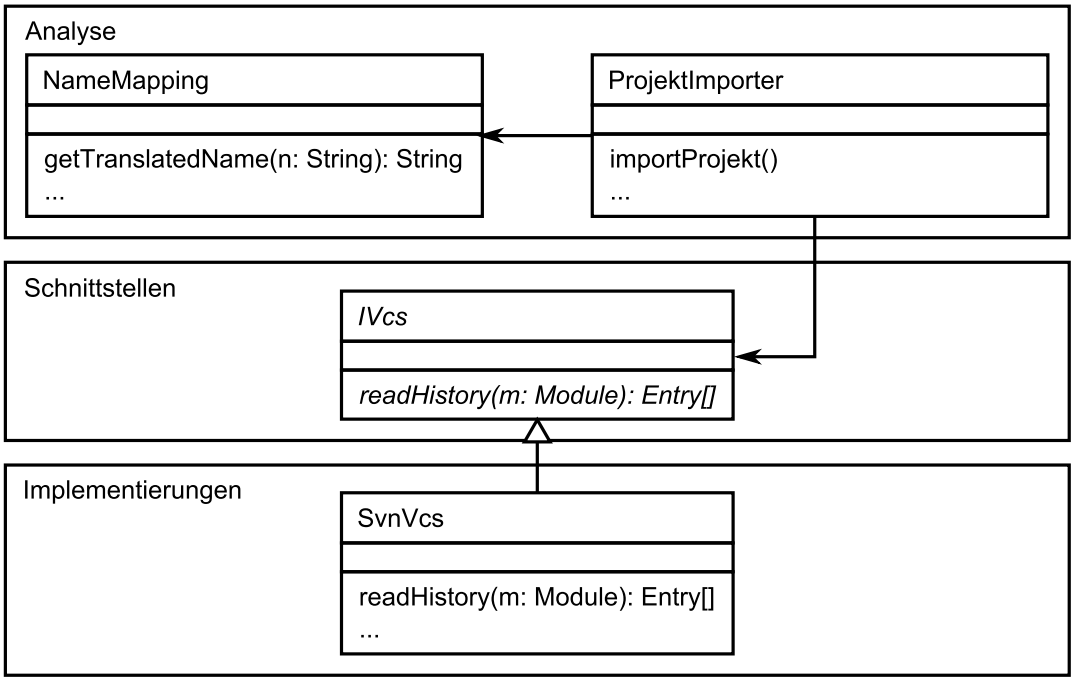

Abbildung 4.1: Ausschnitt aus der Architektur, der die Anbindung an Datenquellen am Beispiel der Datenquellenkategorie *„Versionsverwaltungssystem“* mit der konkreten Datenquelle „Subversion“ zeigt.

Wird eine konkrete Datenquelle angebunden, wird die Schnittstelle der zugehörigen Datenquellenkategorie implementiert (*Implementierung*). Damit der Aufwand dabei möglichst gering ist, ist die Schnittstelle so ausgelegt, dass die Roh-Daten der Datenquelle möglich wenig aufbereitet werden müssen. Da sich die Roh-Daten der verschiedenen Datenquellen einer Kategorie zum Teil jedoch unterscheiden, lässt es sich nicht immer vermeiden, dass die Daten auf ein einheitliches Format gebracht werden müssen.

Die *Analyse*-Komponente nimmt schließlich die (aufbereiteten) Roh-Daten entgegen und verarbeitet sie wie in Abschnitt 3.3 beschrieben. Der Analyse-Komponente sind dabei nur die Schnittstellen der Datenquellenkategorien bekannt. Damit ist die Analyse unabhängig von den konkreten Datenquellen bzw. den Datenquellenanbindungen. Die Analyse-Komponente muss also nicht angepasst werden, wenn eine weitere Datenquellenanbindung hinzugefügt werden soll.

Für die Verwaltung der Datenquellenanbindungen steht ein so genanntes Repository zur Verfügung. Dieses bietet für jede Datenquellenkategorie eine Registrierungsfunktion, über die sich die Datenquellenanbindungen beim System anmelden. Soll anschließend ein bestehendes Projekt in J-PaD importiert und dabei analysiert werden, können aus dem Repository die für das jeweilige Projekt benötigten Datenquellenanbindungen ausgewählt werden. In Abbildung 4.2 wird dieser Vorgang anhand einer Anbindung an das Versionsverwaltungssystem Subversion gezeigt.

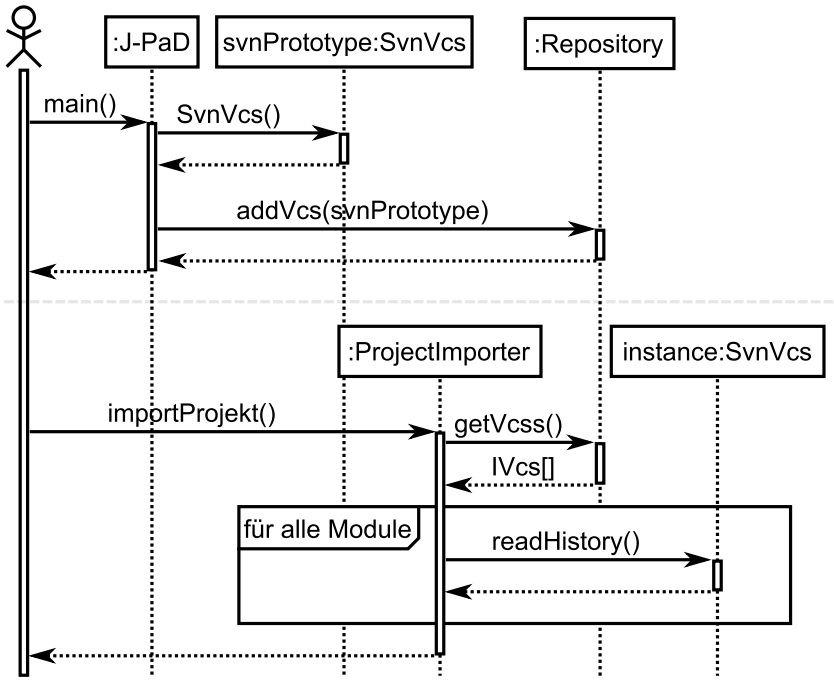

Abbildung 4.2: Datenquellenanbindungen (hier *„SvnVcs“*) einer bestimmten Datenquellenkategorie (hier *„Versionsverwaltungssystem“*) melden sich beim Repository an (obere Hälfte). Aus dem Repository können dann die für das Importieren eines Projekts benötigten Datenquellenanbindungen ausgewählt werde (untere Hälfte).

4.2 Implementierte Datenquellenanbindungen

Um den Implementierungsaufwand zu reduzieren, werden nur Datenquellenanbindung an Datenquellen mit festgelegtem Format (vgl. Abschnitt 3.2.1 auf Seite 17) implementiert. Das sind Anbindungen an Datenquellen der Kategorien „integrierte Dokumentation“, „Quellcode“, „Mustererkennung“ und „Versionsverwaltungssysteme“. Beim Implementieren nicht berücksichtigt werden die beiden Datenquellenkategorien „Issue-Tracker“ und „Separate Dokumentation“.

Nachdem die Kategorien der zu implementierenden Datenquellenanbindungen festgelegt sind, muss für jede Datenquellenkategorie noch die konkrete Datenquelle bestimmt werden, an die angebunden werden soll. Hier bietet es sich auch im Hinblick auf die spätere Demonstration an, an sehr verbreitete Datenquellen anzubinden.

Da J-PaD selbst in Java implementiert ist und auch viele Projekte in dieser Programmiersprache realisiert werden, bietet sich eine Anbindung an **Java** an. Bei der Entwicklung in Java wird quasi standardmäßig die integrierte Dokumentation in **Javadoc** durchgeführt. Damit

sind die Anbindungen für die beiden Kategorien „Quellcode“ und „integrierte Dokumentation“ bestimmt.

Die in Frage kommenden Mustererkennungswerkzeuge beschränken sich auf Werkzeuge, die Muster in Java-Quellcode erkennen können. Zusätzlich ist es wichtig, dass das Mustererkennungswerkzeug plattformunabängig ist und sich aus J-PaD heraus ansprechen lässt. Die Zahl der verfügbaren Mustererkennungswerkzeuge ist momentan noch sehr überschaubar, da die Erkennung von Entwurfsmustern noch ein offenes Forschungsgebiet zu sein scheint. Nach einer Recherche fällt die Wahl schließlich auf ein von Tsantalis et al. [TCSH06] entwickeltes Mustererkennungswerkzeug[1], weil es als eines der wenigen Werkzeuge die genannten Anforderungen erfüllt. In der Kategorie „Mustererkennung“ wird somit an das von **Tsantalis et al.** entwickelte Mustererkennungswerkzeug angebunden.

Es existieren sehr viele verschiedene Versionsverwaltungssysteme mit zum Teil unterschiedlichem internen Aufbau [JKS12]. Dennoch greifen viele Open-Source-Projekte auf das Versionsverwaltungssystem Subversion zurück, das darum sehr verbreitet ist. Deshalb soll in der Kategorie „Versionsverwaltungssysteme“ ebenfalls an **Subversion** angebunden werden.

4.3 Konfiguration

Zu J-PaD werden vier Konfigurationsdateien hinzugefügt, über die sich das Verhalten beim Importieren eines Projekts anpassen lässt.

Gerade beim Versionsverwaltungssystem Subversion entspricht der den Benutzern beim Einreichen einer Änderung in das Systeme zugewiesene Name nicht mit ihrem ausgeschriebenen Namen überein. Stattdessen wird der Name oft abgekürzt (z. B. *„strobepk“* statt *„Patrick Strobel“*). Das stellt insbesondere dann ein Problem dar, wenn die aus der Historie des Versionsverwaltungssystems ausgelesenen Benutzernamen mit den in der integrierten Dokumentation verwendeten Namen (dort sind sie typischerweise ausgeschrieben) verknüpft werden müssen. Um diesem Problem zu begegnen, lassen sich in der Konfigurationsdatei `names.properties` die abgekürzten Namen auf die ausgeschriebenen Namen abbilden.

Mit den sogenannten „Task Tags“ lassen sich Quellcode-Kommentare klassifizieren (vgl. Beschreibungselemente „Reviews“ und „Zu erledigen“ in Unterabschnitt 3.3.4). Alle Kommentare die einen solchen „Task Tag“ enthalten, werden von vielen Entwicklungsumgebungen auf Wunsch aufgelistet. Zwar unterstützen die meisten Entwicklungsumgebungen diese Konzept, die tatsächliche Form der „Task Tags“ unterscheidet sich jedoch. Das Erkennen der „Task Tags“ lässt sich in J-PaD darum über die Konfigurationsdatei `tasks.properties` an die in einem Projekt verwendeten anpassen.

Es gibt eine Vielzahl bei der Softwareentwicklung verwendeter Werkzeuge. Für das Importieren eines Projekts sind besonders Quellcode generierende Werkzeuge (z. B. Parsergeneratoren) und Werkzeuge, die ein Rahmenwerk für Quellcode-Komponenten bieten

[1] Verfügbar unter http://java.uom.gr/~nikos/pattern-detection.html

Listing 4.1 XML-Datei `tools.xml`, über die sich die von J-PaD erkannten Werkzeuge definieren lassen. Dabei wird zwischen Codegeneratoren (`generator`) und Modultest-Rahmenwerke (`unittest`) unterschieden.

```
<?xml version="1.0" encoding="UTF-8"?>
<tools>
    <generator name="JavaCC">
        <comment>Generated By:JavaCC</comment>
        <comment>Generated By:JJTree</comment>
    </generator>
    <unittest name="JUnit">
        <annotation>Test</annotation>
        <annotation>org.junit.Test</annotation>
    </unittest>
    <!-- ... -->
</tools>
```

Listing 4.2 XML-Datei `patterns.xml`, über die sich Einstiegspunkte und Invarianten zu erkannten Entwurfsmustern definieren lassen. Dem Entwurfsmuster *„Template-Method“* lässt sich kein Einstiegspunkt zuordnen, weshalb `entryPoint` der Wert `none` zugeordnet wurde.

```
<?xml version="1.0" encoding="UTF-8"?>
<patterns>
    <pattern name="observer">
        <entryPoint>subject</entryPoint>
        <invariant>Observers added to the subject are notified by the subject.</invariant>
    </pattern>
    <pattern name="template_method">
        <entryPoint>none</entryPoint>
        <invariant>Subclasses can change the parent-class' behaviour.</invariant>
    </pattern>
    <!-- ... -->
</patterns>
```

(z. B. Modultest-Rahmenwerke), interessant. Die von J-PaD erkannten Werkzeuge lassen sich in der XML-Datei `tools.xml` festlegen. Für jedes Werkzeug, das erkannt werden soll, lassen sich charakteristische Kommentare und Annotationen angeben. Listing 4.1 zeigt, wie Kommentare und Annotationen zu Quellcodegeneratoren und Modultest-Rahmenwerke zugeordnet werden können.

J-PaD bestimmt anhand der erkannten Entwurfsmuster Einstiegspunkte und Invarianten (vgl. Beschreibungselemente „Einstiegspunkte“ und „Invarianten“ in Unterabschnitt 3.3.3 bzw. 3.3.5). Über die XML-Datei `patterns.xml` lassen sich für jedes erkannte Entwurfsmuster die am Muster teilnehmenden Übersetzungseinheiten festlegen, die zentrale Rolle spielen und damit Einstiegspunkte darstellen. Ebenfalls lässt sich in der Konfigurationsdatei für jedes Entwurfsmuster die geltende Invariante beschreiben. In Listing 4.2 werden für zwei Entwurfsmuster die Einstiegspunkte und Invarianten angegeben.

Listing 4.3 Auszug aus der Javadoc-Anbindung. Die Anbindung wird mit `@tool` benannt und das Attribut *„rootPackage“* durch die Annotation `@AdditionalProperty` als durch den Benutzer änderbar gekennzeichnet.

```
@Tool("Javadoc")
public class JavadocIntegratedDoc implements IIntegratedDoc {

    @AdditionalProperty(name = "Root Packages", description = "Javadoc comments are read from
        classes located in this package and it's subpackages only.")
    private String rootPackages = "com:de:net:org";

    public String getRootPackages() {
        return this.rootPackages;
    }

    public void setRootPackages(String rootPackages) {
        // Eingabepruefungen
        this.rootPackages = rootPackages;
    }

    // Restliche Implementierung inkl. Implementierung der Schnittstelle
}
```

4.4 Erweiterung

Soll J-PaD um eine weitere Datenquellenanbindung erweitert werden, muss zunächst die Schnittstelle der Kategorie, zu der die anzubindende Datenquelle gehört, implementiert werden. Informationen zu den zu implementierenden Methoden befinden sich in der integrierten Dokumentation der jeweiligen Schnittstelle. Auf das verlangte Format der von der Datenquelle eingelesenen Daten wird in der integrierten Dokumentation ebenfalls eingegangen. Darum sollen diese beiden Punkte an dieser Stelle nicht näher beschrieben werden.

Um den Benutzern einen aussagekräftigen Namen für jede Datenquellenanbindung anzeigen zu können, muss jede Datenquellenanbindung mit der Klassen-Annotation `@language` oder `@tool` versehen werden. Beide Annotationen erwarten als einzigen Parameter die Angabe eines Textes, der dem Namen der Datenquelle entspricht. Die Annotation `@language` muss für Datenquellen-Anbindungen der Kategorie „Quellcode“ verwendet werden und gibt die Programmiersprache an, an die angebunden wird (z. B. *„Java“*). Die Annotation `@tool` muss für alle verbleibenden Kategorien verwendet werden und gibt den Namen des angebundenen Werkzeugs an (z. B. *„Subversion“*).

In manchen Fällen ist es notwendig, dass sich bei einer Datenquellenanbindung weitere Daten durch den Benutzer angeben lassen. So verlangt z. B. das Werkzeug Javadoc die Angabe eines Java-Pakets, ab dem die integrierte Dokumentation eingelesen wird. Wird ein solches Datenfeld benötigt, wird zunächst ein (privates) Attribut angelegt und mit der Annotation `@AdditionalProperty` als durch den Benutzer änderbares Attribut gekennzeichnet. Damit später der Wert des Attributes ausgelesen und verändert werden kann, muss eine `get`- und eine `set`-Methode hinzugefügt werden. Beide Methoden müssen nach dem get-

bzw. set-Präfix den gleichen Namen wie das zugehörige Attribut haben. Eine Prüfung der Benutzereingabe ist innerhalb der `set`-Methode möglich.

Listing 4.3 zeigt einen Auszug aus der Anbindung an Javadoc, das Teil der Datenquellenkategorie „integrierte Dokumentation“ ist.

Nachdem die Datenquellenanbindung implementiert wurde, muss sie bei J-PaD angemeldet werden. Dazu dient das Repository, das für jede Datenquellenkategorie eine Methode `addX(...)` anbietet. Das X steht hierbei für die jeweilige Kategorie (z. B. kann mit *„`addVcs(...)`“* eine Anbindung an ein Versionsverwaltungssystem angemeldet werden).

5 Demonstration

Dieses Kapitel stellt die in J-PaD hinzugefügte Import-Funkion vor und geht auf die Qualität der beim Import erzielten Ergebnisse ein. Zunächst wird in Abschnitt 5.1 die Benutzeroberfläche vorgestellt, über die Angaben zum zu importierenden Projekt getätigt werden und der eigentliche Import gestartet wird. In Abschnitt 5.2 wird dann das für die Demonstration verwendete Projekt „Tomcat“ vorgestellt. Abschnitt 5.3 beschreibt schließlich, was beim Importieren aufgefallen ist und wie die Ergebnisse ausgefallen sind (d. h. wie gut die Beschreibungselemente automatisch ausgefüllt wurden).

5.1 Benutzeroberfläche

Die Benutzeroberfläche von J-PaD war zum Zeitpunkt der Implementierung nur rudimentär. Darum wird der Import eines Projekts beim Starten von J-PaD automatisch angestoßen und die eingelesenen Beschreibungselemente nach Abschluss des Imports direkt im HTML-Format exportiert. Sobald jedoch die Benutzeroberfläche fertiggestellt ist, wird der Importvorgang über einen Menü-Eintrag gestartet und der Benutzer kann nach Abschluss des Imports selbst bestimmen, wie mit den importierten Daten weiter verfahren werden soll.

Durch den Benutzer müssen einige Daten angegeben werden, bevor ein Projekt importiert werden kann. Diese Daten lassen sich im Import-Fenster eingeben, das wie oben erwähnt beim Start von J-PaD automatisch erscheint (später wird es über einen entsprechenden Menü-Eintrag geöffnet). Abbildung 5.1 zeigt das Import-Fenster, das hier auf das Open-Source-Projekt *„Tomcat“* eingestellt wurde. Das Import-Fenster unterteilt sich in drei Bereiche:

Im oberen Bereich werden alle projektrelevanten Verzeichnispfade angegeben. Das sind neben den Verzeichnissen, in denen sich die Quellcode-Dateien („Source-code“) und der Quellcode der Modultests („Test-code“) befinden auch das Verzeichnis mit übersetztem Programmcode („Compiled-code“) und ein Arbeitsverzeichnis („Working“). Das Modultest-Verzeichnis ist optional und kann ausgefüllt werden, wenn in dem zu importierenden Projekt der Quellcode der Geschäftslogik und der Quellcode der Modultests in einer getrennten Verzeichnisstruktur abgelegt wurden. Das Arbeitsverzeichnis sollte sich außerhalb des zu importierenden Projekts befinden. In diesem werden Dateien, die während des eigentlichen Importvorgangs generiert werden (z. B. die List der erkannten Entwurfsmuster), abgelegt.

Im mittleren Bereich werden alle projektrelevanten Datenquellenanbindungen selektiert. Über die Auswahlfelder werden damit also die Datenquellen und Werkzeuge angegeben, die im zu importierenden Projekt verwendet wurden. Dies betrifft die vier unter Abschnitt

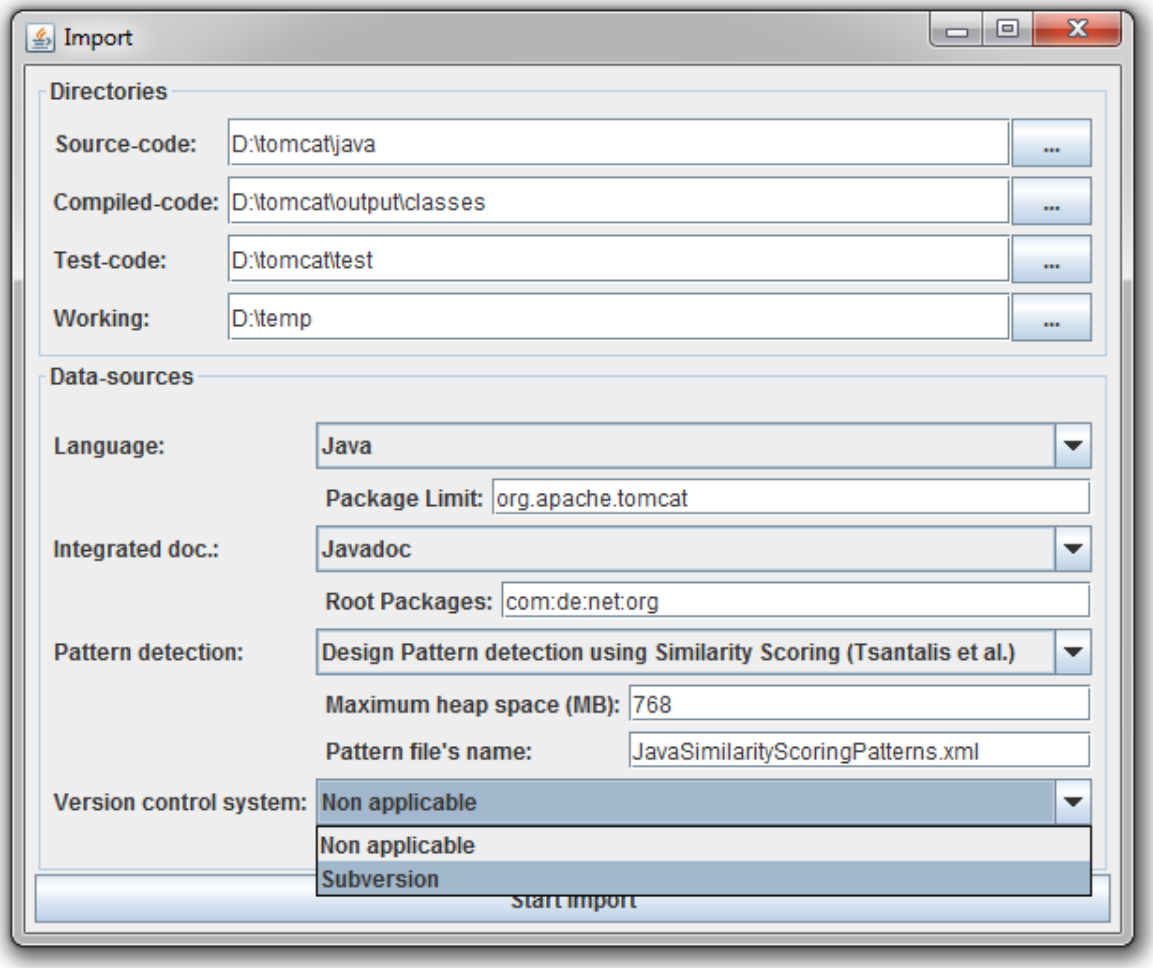

Abbildung 5.1: Benutzeroberfläche, über die Projekt-Verzeichnisse und Datenquellen ausgewählt werden.

4.2 erwähnten Datenquellenkategorien „Quellcode“ („Language“), „integrierte Dokumentation“ („Integrated doc.“), „Mustererkennung“ („Pattern detection“) sowie „Versionsverwaltungssystem“ („Version control system“). Mit Ausnahme des Quellcodes lässt sich bei den verbleibenden drei Kategorien als Datenquellenanbindung der Eintrag „nichts zutreffend“ („non applicable“) auswählen, wenn keine zum Projekt passende Datenquellenanbindung verfügbar ist. Bei einigen Datenquellenanbindungen sind zusätzlich spezifische Angaben nötig (vgl. Abschnitt 4.4 auf Seite 50). Diese lassen sich, nachdem die Anbindung ausgewählt wurde, über zusätzliche Eingabefelder unterhalb des Auswahlfeldes anpassen.

Im unteren Bereich lässt sich über die Schaltfläche „Import starten“ („Start Import“) der eigentliche Importvorgang durchführen. Dabei wird zunächst die Projektstruktur (die vorhandenen Module, Übersetzungseinheiten und deren Beziehungen untereinander) eingelesen,

die integrierte Dokumentation aus dem Quellcode ausgelesen und über die Mustererkennung die verwendeten Entwurfsmuster bestimmt. Anschließend wird für jedes Modul die Historie aus dem Versionsverwaltungssystem ausgelesen, das Modul wie unter Abschnitt 3.3 beschrieben analysiert und die Beschreibungselemente ausgefüllt. Über einen Status-Dialog wird der Import-Fortschritt angezeigt.

5.2 Demonstrationsprojekt

Die Demonstration soll an einem größeren Open-Source-Projekt durchgeführt werden. Das Projekt sollte zudem eine praktische Relevanz haben, also von möglichst vielen Anwendern verwendet werden. Unter Berücksichtigung dieser Punkte habe ich mich für den Webserver Tomcat[1] als Demonstrationsprojekt entschieden.

Tomcat ist ein Webserver, mit dem in Java geschriebene Webanwendungen ausgeführt werden können und der selbst in Java implementiert ist. Die integrierte Dokumentation wird in Javadoc durchgeführt. Sowohl der Quellcode von Tomcat selbst als auch die zugehörige separate Dokumentation wird im Versionsverwaltungssystem Subversion verwaltet. Als Issue-Tracker wird Bugzilla eingesetzt.

Quellode: Java (`http://svn.apache.org/repos/asf/tomcat/trunk`)

Integrierte Dokumentation: Javadoc

Versionsverwaltungssystem: Subversion

Issue-Tracker: Bugzilla (`https://issues.apache.org/bugzilla/describecomponents.cgi?product=Tomcat7`

Separate Dokumentation: HTML & PDF (`http://svn.apache.org/repos/asf/tomcat/site/trunk/docs`)

Tomcat besteht momentan aus rund 328.000 Zeilen Java-Code, die sich auf etwa 1.170 Übersetzungseinheiten verteilen.

5.3 Beobachtungen

Was zunächst auffällt ist, dass für das Erkennen der Entwurfsmuster sehr viel Zeit benötigt wird. Während die Projektstruktur und die integrierte Dokumentation von Tomcat (auf einer Intel Core2Duo Mobil-CPU mit 2,13 GHz) in unter 2 Minuten vergleichsweise schnell eingelesen werden, werden für die Musterkennung über 30 Minuten benötigt. Für das Erkennen der meisten Entwurfsmuster muss laut Tsantalis et al. ein großer Teil des Projekts analysiert werden, was zu einer langen Laufzeit führt [TCSH06]. Vermutlich dürfte es sich

[1] `http://tomcat.apache.org`

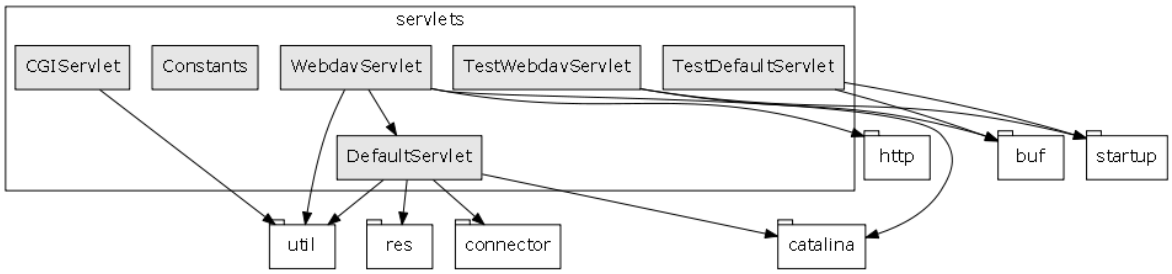

Abbildung 5.2: Aus der DOT-Beschreibung mit GraphViz erstelltes Übersichtsdiagramm des Moduls *„org.apache.catalina.servlets“*.

zudem bei dem Musterkennungswerkzeug um eine prototypische Implementierung handeln, die hinsichtlich der Laufzeit nicht optimiert wurde.

In den nachfolgenden Unterabschnitten werden die Ergebnisse des Tomcat-Imports betrachtet. Dafür wird exemplarisch die automatisch für das Modul *„org.apache.catalina.servlets“* erstellte Beschreibung näher betrachtet: In Unterabschnitt 5.3.1 werden einige Beschreibungselemente vorgestellt, die auch bei vielen anderen Modulen besonders zufriedenstellend ausgefüllt werden. In Unterabschnitt 5.3.2 wird auf Beschreibungselemente eingegangen, die bei manchen Modulen zufriedenstellend ausgefüllt werden. Abschließend werden in Unterabschnitt 5.3.3 einige Beschreibungselemente vorgestellt, die überwiegend nicht oder nur ungenügend automatisch ausgefüllt werden.

5.3.1 Gute Ergebnisse

Die Beschreibungselemente des Moduls werden besonders zutreffend ausgefüllt, wenn die benötigten Daten bereits in ähnlicher Form in den Datenquellen vorliegen. Das Beschreibungselement „Qualifizierter Modulname“ und das davon abgeleitete Element „Modulname“ werden etwa immer exakt ausgefüllt, weil sich die benötigten Informationen direkt im Quellcode befinden. Das Gleiche gilt für die Referenzen auf den Quellcode („Quellcode“) und die Codezeilen („Anzahl Code-Zeilen“).

Ebenfalls sehr gut wird das Übersichtsdiagramm aus dem Quellcode generiert. Das gleichnamige Beschreibungselement wird dabei in der Diagrammbeschreibungssprache DOT ausgefüllt, aus der sich mit dem Werkzeug GraphViz[2] ein Diagramm erstellen lässt. Abbildung 5.2 zeigt das erstellte Übersichtsdiagramm des Moduls.

Zwischen dem Übersichtsdiagramm und dem Beschreibungselement „Eingehende Schnittstellen“ gibt es einen engen Zusammenhang: Übersetzungseinheiten, die Teil der eingehenden Schnittstellen sind, befinden sich in Modulen, auf denen im Diagramm referenziert wird (z. B. *„startup“*, *„util“*). Das Beschreibungselement wird zuverlässig ausgefüllt, da sich alle

[2] http://www.graphviz.org

Listing 5.1 Auszug aus den für das Modul *„org.apache.catalina.servlets“* ermittelten eingehenden Schnittstellen. Tatsächlich enthält die Liste Referenzen auf 18 Übersetzungseinheiten, von denen hier aber nur fünf gezeigt werden.

```
org.apache.catalina.startup.Tomcat:
  org.apache.catalina.startup.Tomcat#addServlet,
       org.apache.catalina.startup.Tomcat#addWebapp,
       org.apache.catalina.startup.Tomcat#aseTest.getUrl,
       org.apache.catalina.startup.Tomcat#start
org.apache.catalina.startup.TomcatBaseTest:
  org.apache.catalina.startup.TomcatBaseTest#getUrl
org.apache.catalina.util.URLEncoder:
  org.apache.catalina.util.URLEncoder#addSafeCharacter,
       org.apache.catalina.util.URLEncoder#encode
org.apache.tomcat.util.buf.ByteChunk:
  org.apache.tomcat.util.buf.ByteChunk#recycle, org.apache.tomcat.util.buf.ByteChunk#toString
org.apache.catalina.WebResource:
  org.apache.catalina.WebResource#delete, org.apache.catalina.WebResource#exists,
       org.apache.catalina.WebResource#getContentLength,
       org.apache.catalina.WebResource#getCreation, org.apache.catalina.WebResource#getETag,
       org.apache.catalina.WebResource#getInputStream,
       org.apache.catalina.WebResource#getLastModified,
       org.apache.catalina.WebResource#getName, org.apache.catalina.WebResource#isDirectory,
       org.apache.catalina.WebResource#isFile
```

benötigten Informationen im Quellcode befinden. Allerdings wird der Inhalt des Beschreibungselementes schnell sehr umfangreich, weil für jede verwendete Übersetzungseinheit die verwendeten Operationen mit aufgelistet werden. In Listing 5.1 wird ein Teil der erkannten eingehenden Schnittstelle des Moduls gezeigt.

In Tomcat ist eine integrierte Dokumentation innerhalb einzelner Übersetzungseinheiten praktisch immer vorhanden. Beschreibungselemente, bei denen die Daten hieraus bezogen und zudem Historie-Einträge (Versionsverwaltungssystem) mitverwendet werden können, werden dadurch zufriedenstellend ausgefüllt. Das betrifft die Beschreibungselemente „Autor“ und „Bearbeiter und weitere Autoren“. Es fällt mir allerdings schwer zu beurteilen, in wie weit insbesondere der ermittelte Modul-Autor mit dem tatsächlichen Autor des Moduls übereinstimmt. Detaillierte Informationen über die Module und die daran arbeitenden Personen konnte ich außerhalb der integrierten Dokumentation und des Versionsverwaltungssystems leider nicht finden.

Das Bestimmen der Einstiegspunkte, ausgehend von den erkannten Entwurfsmustern, funktioniert gut. So werden für das Modul *„org.apache.catalina.servlets“* als Einstiegspunkte die drei Übersetzungseinheiten *„WebdavServlet“*, *„CGIServlet“* und *„DefaultServlet“* ermittelt.

5.3.2 Mittelmäßige Ergebnisse

Eine integrierte Dokumentation auf Modulebene findet in Tomcat nur teilweise statt. Lediglich 33 der insgesamt 116 Module (rund 28 %) sind dokumentiert. Da die Datenquelle

„Issue-Tracker“ in der Implementierung nicht angebunden ist, kann das Beschreibungselement „Aufgabe und Zweck“ momentan nur anhand der integrierten Moduldokumentation ausgefüllt werden (vgl. „Aufgabe und Zweck“ in Unterabschnitt 3.3.1). Bei Modulen, die keine integrierte Dokumentation enthalten, bleibt das Beschreibungselement somit leer und die Qualität des Inhalts entspricht der Qualität der integrierten Moduldokumentation.

5.3.3 Schlechte Ergebnisse

Die Annahme, das Beschreibungselement „Ausgeschriebener Modulname“ ließe sich anhand wiederkehrender Muster in der Historie ausfüllen, stellt sich als falsch heraus. Es gibt keine Muster, die aus mehr als drei Zeichen bestehen und in allen Historie-Einträgen vorhanden sind. Um überhaupt ein Ergebnis zu erhalten, musste der Anteil der Historie-Einträge, in denen sich das Muster befinden soll, auf 20 % reduziert werden. Das Ergebnis bleibt allerdings ernüchternd: Als Muster werden meist wiederkehrende Wörter wie *„and“*, *„the“* oder die Adresse des Issue-Trackers (*„http://issues.apache.org/bugzilla/show_bug.cgi?id=“*) erkannt.

In der momentanen Implementierung kann das Beschreibungselement „Verantwortliche Person“ aufgrund der fehlenden Anbindung an einen Issue-Tracker nur anhand der Autoren-Angabe in der integrierten Moduldokumentation ausgefüllt werden (vgl. „Verantwortliche Person“ in Unterabschnitt 3.3.2). Leider wird aber bei Tomcat in der integrierten Dokumentation der Module nie ein Autor erwähnt. Somit kann das Beschreibungselement momentan nicht ausgefüllt werden.

Aufgrund der fehlenden Anbindung an die Datenquelle „Separate Dokumentation“ lassen sich zudem alle Beschreibungselemente nicht oder nur ungenügend ausfüllen, die von dieser Datenquelle stark abhängen. Dazu zählen die Beschreibungselemente „Ursprung“, „Reviews“, „Dokumente“ (Status) und „Geplante Anzahl Code-Zeilen“.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurde die in J-PaD verwaltete Modulbeschreibung analysiert und das Werkzeug anhand der Analyseergebnisse erweitert.

Für die Analyse wurden die in J-PaD für die Modulbeschreibung angebotenen Beschreibungselemente zunächst näher betrachtet und jedes Beschreibungselement in eine von sieben Kategorien eingeteilt. Anschließend wurden Datenquellenkategorien ermittelt, aus denen sich die in den einzelnen Beschreibungselementen zu speichernden Daten auslesen lassen. Schließlich wurde jedes Beschreibungselement detailliert betrachtet und untersucht, über welche Datenquellenkategorien das jeweilige Beschreibungselement ausgefüllt werden kann und wie die ausgelesenen Daten dafür aufbereitet werden müssen.

Nachdem die Beschreibungselemente analysiert wurden, konnte mit der Implementierung begonnen werden. Dabei wurde J-PaD um eine Importfunktion erweitert, die die Beschreibungselemente automatisch ausfüllt. Hierbei stellten sich die zuvor dokumentierten Analyseergebnisse als große Hilfe heraus. Anhand der Ergebnisse konnte die Architektur sorgfältig geplant werden und auch die von den Datenquellenkategorien benötigten Daten waren frühzeitig bekannt. So waren größere Überarbeitungen während der Implementierung nicht erforderlich.

Die in einem Softwareprojekt verwendeten Werkzeuge und die anfallenden Daten – kurz die Datenquellen – variieren je nach Projekt. Darum wurde die Architektur so ausgelegt, dass sich Anbindungen an weitere Datenquellen (z. B. weitere Versionsverwaltungssysteme) leicht hinzufügen lassen. Aufgrund des begrenzten Umfangs der Bachelorarbeit konnten nicht alle Datenquellenkategorien in die Importfunktion integriert werden. Weitere Datenquellenkategorien sollten sich jedoch bei Bedarf leicht hinzufügen lassen, weil alle Datenquellen auf ähnliche Weiße in J-PaD integriert werden.

Zuletzt wurde die zu J-PaD hinzugefügte Importfunktion an dem Open-Source-Projekt Tomcat demonstriert und erprobt. Einige der dabei gemachten interessanten Beobachtungen wurden vorgestellt. Es zeigte sich, dass sich besonders die Beschreibungselemente, die keine größeren Datenaufbereitungen erforderten, gut automatisch ausfüllen lassen. Insbesondere Beschreibungselemente, die von den nicht implementierten Datenquellenkategorien abhängen, konnten jedoch nicht oder nicht zufriedenstellend ausgefüllt werden. Diese Beschreibungselemente ließen sich womöglich besser ausfüllen, wenn die noch fehlenden Datenquellenkategorien hinzugefügt werden.

Abschließend kann aber gesagt werden, dass der Einarbeitungsaufwand durch die implementierte Importfunktion stark erleichtert wird. Ein nachträgliches, manuelles Überarbeiten der automatisch ausgefüllten Beschreibungselemente lässt sich zwar nicht vermeiden, den

Benutzern wird aber ein guter Einstieg geboten: Die vorausgefüllten Beschreibungselemente bieten den Benutzern Vorschläge, an denen sie sich orientieren und die sie nach Bedarf erweitern können. Das erleichtert es nicht nur, die Intention hinter den verschiedenen Beschreibungselementen zu verstehen, sondern motiviert auch dazu, die Modulbeschreibung zu erweitern und zu überarbeiten.

Ausblick

Einige Beschreibungselemente lassen sich durch Datenquellenkategorien ausfüllen, die in J-PaD gegenwärtig nicht implementiert sind. Das führt dazu, dass einige Beschreibungselemente nur unzureichend ausgefüllt werden. Die Qualität der Modulbeschreibung ließe sich darum erhöhen, indem die noch verbleibenden Datenquellenkategorien zu J-PaD hinzufügt werden.

Andererseits stellten sich einige Annahmen, wie sich die Beschreibungselemente aus den Datenquellen ausfüllen lassen, als zu gewagt heraus (etwa das Suchen von Mustern beim Beschreibungselement „Ausgeschriebener Modulname“). Bei diesen Beschreibungselementen gehe ich davon aus, dass sie sich automatisch nicht ausfüllen lassen, weil sich die dafür benötigten Informationen in Softwareprojekten praktisch nicht finden lassen. Ich bezweifle auch, dass sich die benötigten Informationen aus anderen Daten zuverlässig ableiten lassen. Zufriedenstellende Ergebnisse bei allen Beschreibungselementen zu erhalten, ist in meinen Augen also nicht möglich.

Manche Beschreibungselemente, wie das Übersichtsdiagramm oder die eingesetzten Hilfsmittel, lassen sich sehr genau automatisch ausfüllen. Gleichzeitig ist es für die Benutzer aber wenig attraktiv, diese Informationen manuell zu aktualisieren; sie müssten bei jeder Quellcodeänderung überarbeitet werden. Darum ist eine Synchronisationsfunktion aus Sicht der Benutzer wünschenswert. Eine solche Funktion erlaubt es, einzelne Beschreibungselemente gezielt erneut automatisch auszufüllen. So kann beispielsweise das Beschreibungselement „Übersichtsdiagramm“ nach jeder größeren Änderung am Quellcode erneut automatisch ausgefüllt werden, wodurch es ohne einen größeren Aufwand aktuell bliebe. Damit würde sich die Chance erhöhen, dass Dokumentation und Quellcode konsistent bleiben.

Die in Softwareprojekten verfügbaren Datenquellen variieren auch in Quantität und Qualität der vorhandenen Daten. Der Umfang der integrierten Dokumentation unterscheidet sich etwa zwischen verschiedenen Projekten. Es wäre darum interessant, die Resultate der Importfunktion zwischen verschiedenen Softwareprojekten zu vergleichen.

Literaturverzeichnis

[Bog12] I. Bogicevic. Aktuelle Dokumentation einiger Beschreibungselemente, 2012. Zusatzmaterial. (Zitiert auf den Seiten 15, 29 und 37)

[BP00] F. Bergenti, A. Poggi. Improving UML Designs Using Automatic Design Pattern Detection. In *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering*, SEKE '00, S. 336–343. 2000. (Zitiert auf den Seiten 12 und 18)

[FL02] A. Forward, T. C. Lethbridge. The Relevance of Software Documentation, Tools and Technologies: A Survey. In *Proceedings of the 2002 ACM symposium on Document engineering*, DocEng '02, S. 26–33. ACM, New York, NY, USA, 2002. doi:10.1145/585058.585065. URL http://doi.acm.org/10.1145/585058.585065. (Zitiert auf den Seiten 11 und 17)

[GHJV04] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 5 Auflage, 2004. (Zitiert auf den Seiten 11, 31, 39 und 40)

[Hee13] D. van Heesch. Doxygen Manual, 2013. URL http://www.stack.nl/~dimitri/doxygen/manual. (Zitiert auf den Seiten 33, 37 und 40)

[HH11] C. Hujer, R. Holly. TODO Syntax 1.0 Specification, 2011. URL http://www.riedquat.de/TR/TODO_Syntax. (Zitiert auf den Seiten 36 und 37)

[IEE10] IEEE. Systems and software engineering – Vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, 2010. doi:10.1109/IEEESTD.2010.5733835. (Zitiert auf Seite 13)

[JKS12] J. Jarosch, T. Kuhn, P. Strobel. Marktanalyse Quellcodeverwaltung, 2012. URL http://elib.uni-stuttgart.de/opus/volltexte/2012/7964. (Zitiert auf Seite 48)

[Kir12] M. Kircher. *Integrierte Dokumentation für Software-Module*. Diplomarbeit, Universität Stuttgart, 2012. URL http://elib.uni-stuttgart.de/opus/volltexte/2012/7804. (Zitiert auf Seite 9)

[Kuh12] T. Kuhn. Optimierung eines Dokumentationswerkzeugs für Java-Pakete, 2012. Bachelorarbeit, Universität Stuttgart. (Zitiert auf Seite 9)

[LL10] J. Ludewig, H. Lichter. *Software Engineering*. dpunkt.verlag GmbH, Heidelberg, 2. Auflage, 2010. (Zitiert auf den Seiten 12, 13 und 20)

[MM03] A. Marcus, J. I. Maletic. Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, S. 125–135. Portland, Oregon, 2003. doi:10.1109/ICSE.2003.1201194. (Zitiert auf den Seiten 12 und 21)

[Ora99] Oracle Corporation. Code Conventions for the Java Programming Language, 1999. URL http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html. (Zitiert auf den Seiten 9 und 17)

[Ora12] Oracle Corporation. javadoc - The Java API Documentation Generator, 2012. URL http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html. (Zitiert auf den Seiten 33, 37 und 40)

[SH11a] G. Starke, P. Hruschka. *Software-Architektur kompakt*. Spektrum Akademischer Verlag, Heidelberg, 2. Auflage, 2011. (Zitiert auf den Seiten 11, 26, 27, 31, 38 und 39)

[SH11b] C. J. Stettina, W. Heijstek. Necessary and Neglected? An Empirical Study of Internal Documentation in Agile Software Development Teams. In *Proceedings of the 29th ACM international conference on Design of communication*, SIGDOC '11, S. 159–166. ACM, New York, NY, USA, 2011. doi:10.1145/2038476.2038509. URL http://doi.acm.org/10.1145/2038476.2038509. (Zitiert auf den Seiten 11, 18 und 19)

[TCSH06] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S. T. Halkidis. Design Pattern Detection Using Similarity Scoring. *IEEE Trans. Softw. Eng.*, 32(11):896–909, 2006. doi:10.1109/TSE.2006.112. URL http://dx.doi.org/10.1109/TSE.2006.112. (Zitiert auf den Seiten 12, 48 und 55)

[XTLL09] T. Xie, S. Thummalapenta, D. Lo, C. Liu. Data Mining for Software Engineering. *Computer*, 42(8):55–62, 2009. doi:10.1109/MC.2009.256. (Zitiert auf Seite 12)

Alle URLs wurden zuletzt am 26. 05. 2013 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschift