Ruprecht-Karls-Universität Heidelberg

Institut für Informatik
Lehrstuhl Software Systeme
Prof. Dr. Barbara Paech

Softwareentwicklung mit dem TRAIN-Prozess

BACHELORARBEIT

Philipp Häfele

Universität Heidelberg
Institut für Informatik
Lehrstuhl Software Systeme
Prof. Dr. Barbara Paech

Bachelorarbeit

Softwareentwicklung mit dem TRAIN-Prozess

eingereicht von
Philipp Häfele
Anwendungsorientierte Informatik
Matrikelnummer 2222770

betreut von
Prof. Dr. Barbara Paech
Dipl. Inf. Lars Borner

Heidelberg, den 24.02.2005

Erklärung

Hiermit erkläre ich, Philipp Häfele, dass ich diese Arbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Heidelberg, den 24.02.2005

1 Einleitung

Im Sommersemester 2004 wurde am Institut für Informatik der Universität Heidelberg erstmals die Vorlesung „Software Engineering – Planung und Durchführung von Softwareentwicklungsprojekten“ gehalten. Inhalt der Vorlesung sind v. a. Techniken, Methoden, ingenieurmäßige Prinzipien und Werkzeuge des Software Engineering, die dazu dienen, die Komplexität der zu entwickelnden Systeme und deren Projekte beherrschbar zu machen. Im Rahmen der aufgezeigten Vorgehensweisen wird auf die Aufgaben der Softwareentwicklung (Anforderungsspezifikation, Architektur, Entwurf, Implementierung, Projekt- und Qualitätsmanagement) eingegangen, und als Unterstützung werden im objektorientierten Entwicklungsumfeld die Grundlagen der Unified Modeling Language (UML) vorgestellt.
Im Vordergrund der vorlesungsbegleitenden Übung steht ein Softwareentwicklungsprojekt, das anhand einer in der Vorlesung gelehrten, umfassenden Entwicklungsmethode namens TRAIN (**T**est, **Ra**tionale und **In**spektion unterstützender Softwareentwicklungsprozess) durchgeführt werden muss.

1.1 Inhalt der Arbeit

Die vorliegende Arbeit soll den TRAIN-Prozess mit seinen Entscheidungsarten, Aktivitäten, Artefakten, Modellierungstechniken und Qualitätssicherungsmaßnahmen zusammenfassen und anhand eines Beispiels veranschaulichen. Das Ziel ist, zukünftigen Studierenden eine kompakte, aus den Vorlesungs- und Übungsinhalten extrahierte Vorgehensbeschreibung an die Hand zu geben, um sich mit kurzem Zeitaufwand in den komplexen Prozess der Softwareentwicklung einarbeiten zu können und eine Orientierung bei der Bewältigung ihrer zukünftigen Entwicklungsaufgabe(n) zu erhalten.
Die Arbeit umfasst jedoch nicht alle Aspekte des Prozesses vollständig. So fehlt die Benutzungsschnittstellenentwicklung, die ab der Systemebene in die Entwicklung einbezogen wird (Kapitel 4.8.1) sowie Teile des Testprozesses. Die Benutzungsschnittstellenentwicklung wird weder im theoretischen noch im praktischen Teil der Arbeit weitergehend besprochen und der Testprozess wird nicht vollständig im Beispiel aufgezeigt. So wird auf Integrationstestbestandteile ganz verzichtet, und der Systemtest sowie ein Komponententest werden ausschließlich bis zu deren Spezifikation

dokumentiert. Weiterhin fehlt bei der Erarbeitung des Entwurfsmodells eine ausführliche Diskussion über Entwurfsalternativen.

1.2 Gliederung der Arbeit

Die Arbeit ist maßgeblich nach den Prozessphasen Requirements Engineering (Kapitel 4), Architekturdefinition (Kapitel 5), Feinentwurf (Kapitel 6) und Implementierung (Kapitel 7) gegliedert. Diesen Kapiteln vorangestellt ist ein zusammenfassender Überblick über den Prozess (Kapitel 2) und eine Beschreibung des Entwicklungsbeispiels, anhand dessen der Prozess in dieser Arbeit praktisch erläutert wird (Kapitel 3). Im Anhang findet man zusätzliche Informationen, die für das Verständnis der Arbeit von Nutzen sein können:
In Anhang A steht ein Prozessüberblick in Tabellenform zur Verfügung, der die Phasen, Entscheidungsebenen, Artefakte, das Rationale und die Qualitätssicherungsmaßnahmen des Prozesses auf wenigen Seiten zusammenfasst. Anhang B beinhaltet eine Zusammenstellung der wichtigsten Beschreibungsvorlagen (Templates) des Prozesses mit deren Erklärung und Anhang C eine Zusammenstellung dreier Checklisten für die Inspektion von Softwaredokumenten, die mit als Grundlage für die Entwicklung des Beispiels dienen. Da die Checklisten für den TRAIN-Prozess entwickelt wurden, können sie aber auch zur Inspektion einer Anforderungsspezifikation herangezogen werden, die mittels des Prozesses erstellt wurde. Anhang D enthält Ausschnitte des Java-Quellkodes, der im Beispiel entwickelten Klassen. Die gewählten Ausschnitte entsprechen Implementierungsaspekten, die während der Phase des Feinentwurfs angesprochenen wurden. Anschließend werden in der Arbeit vorhandene, jedoch nicht genauer erklärte Begrifflichkeiten – schwerpunktmäßig aus dem Bereich des Testens, der Inspektion und aus dem technischen Bereich – im Glossar erläutert. Abschließend wird auf die in der Arbeit verwendeten Quellen verwiesen; besonders ist auf die Literaturangaben zu achten, die auch dazu dienen, die in der Arbeit erläuterten Themen zu vertiefen.
Innerhalb der Kapitel, welche die Prozessphasen im Detail beschreiben (Kapitel 4-7), findet zuerst eine theoretische Aufarbeitung der Prozessphase statt. Hierbei soll die Aufmerksamkeit verstärkt auf die Frage gelenkt werden, wann und wozu die Entscheidungen und das Rationale explizit gemacht werden sollten und wofür die verschiedenen Modellierungshilfsmittel dienen. Auch die Möglichkeit der frühzeitigen

Integration von Qualitätssicherungsmaßnahmen soll aufgezeigt werden. Danach folgt die Erläuterung der Prozessphase anhand des Beispiels. Dieser Teil wird durch eine andere Schriftart gekennzeichnet, damit die LeserInnen den theoretischen (Schriftart: Times New Roman) und praktischen (Schriftart: Arial) Teil möglichst schnell unterscheiden können. Eingerückte, kursiv gedruckte Textabschnitte innerhalb des Beispielteils kennzeichnen zusammenfassende Reflexionen über Sachverhalte, die sich danach meist in einem Artefakt (z. B. Template, Diagramm) widerspiegeln. Normal gedruckte Abschnitte haben meist prozessbeschreibenden bzw. erläuternden Inhalt.

1.3 Annahmen

Bei der Verfassung der Arbeit wurde vorausgesetzt, dass die LeserInnen mit den Notationen der Modellierungstechniken vertraut sind. Diese wurden im Rahmen der Vorlesung und den zugehörigen Übungen bereits erörtert, können aber, da sie größtenteils zur Unified Modeling Language (UML) gehören, in einem entsprechenden Buch nachgeschlagen werden. Des Weiteren wird angenommen, dass die Lesenden mit den Grundkonzepten des Testens und – für das weitergehende Verständnis des Beispiels – mit den Grundlagen der Objektorientierung und der Programmiersprache Java vertraut sind.

2 Der TRAIN-Prozess im Überblick

Während des Softwareentwicklungsprozesses müssen Entscheidungen getroffen werden, die den Kontext des Systems und das System selbst betreffen. Werden diese erst bei der Implementierung gefällt – bei der sie spätestens gefällt werden müssen – ist die Gefahr groß, dass dies implizit und damit meist unbewusst geschieht. Das birgt v. a. bei komplexen, über lange Zeit entwickelten Systemen eine Reihe von Problemen. Beispiele dafür sind die fehlende Nachvollziehbarkeit von Entscheidungen und der fehlende Wissenstransfer zwischen den vielen Beteiligten des Prozesses. Dies wirkt sich später v. a. auf die Evolution (Weiterentwicklung, Wiederverwendung, Reengineering) des Systems negativ aus. Aber auch die Planbarkeit und damit die effiziente Organisation von Ressourcen wie Zeit, Kosten, Personal etc. ist schwer möglich. Ohne explizite Dokumentation fehlt den Qualitätssicherungsmaßnahmen – wie Inspektionen und Tests – außerdem die Grundlage zur systematischen prozessbegleitenden Entwicklung und Durchführung.

Daher versucht der TRAIN-Prozess eine Entwicklungsmethode aufzuzeigen, die diesen Problemen vorbeugt. Wie das im TRAIN-Prozess erreicht werden soll, fasst Kapitel 2.1 zusammen. Kapitel 2.2 geht auf die wichtigsten, am Prozess beteiligten Rollen ein, und in Kapitel 2.3 wird ein Überblick über die Gliederung bzw. die Inhalte des Prozesses gegeben.

2.1 Das Ziel des Prozesses

Zu Anfang des Kapitels wurde festgestellt, dass während des Softwareentwicklungsprozesses (Gestaltungs-) Entscheidungen getroffen werden müssen. Die Entwicklung eines speziellen Prozesses zielt nun darauf ab, eine Systematik in diesen Entscheidungen zu erlangen. Da es aber auch mittels eines Prozesses nicht möglich ist, *alle* Entscheidungen inklusive deren Begründungen zu erfassen, muss man sich auf die wichtigen Entscheidungen für die Kommunikation der Prozessbeteiligten und die Qualität des Produktes konzentrieren. Dies versucht der TRAIN-Prozess umzusetzen, indem Prozessphasen vorgegeben werden, innerhalb derer dann wiederum Hilfestellungen zur Entscheidungsfindung, Dokumentation, qualitätsorientierten Entwicklung und Begründungserfassung (Rationale) einbezogen werden. Zu diesen Hilfestellungen gehören:

- Ein konzeptionelles Modell für das Requirements Engineering, welches die für die Kommunikation im Entwicklungsprozess wichtigen Entscheidungstypen identifiziert und eine geeignete Notation zur Dokumentation festlegt sowie das Vorgehen der Phase beschreibt (Requirements Engineering Phase).
- Eine Methode, wie man von den spezifizierten Anforderungen der Requirements Engineering Phase zu einem Klassendiagramm für den objektorientierten Entwurf gelangen kann (Requirements Engineering Phase).
- Identifizierung der Sichten auf die Architektur und darauf aufbauend Vorgaben zur Architekturdokumentation (Architekturdefinitionsphase).
- Vorgaben zur geeigneten Dokumentation des objektorientierten Entwurfes, wie z. B. Klassendiagramme sowie Package-, Klassen und Operationsbeschreibungen (Feinentwurfphase).
- Standards und Konventionen für die Codierung (Implementierungsphase).
- Ein Vorgehensmodell, in dem festgelegt wird, wann und wie Qualitätssicherungsmaßnahmen in die Phasen des Prozesses integriert werden (phasenübergreifend).
- Vorgaben, auf welche Art und an welchen Stellen des Prozesses welche Begründungszusammenhänge (Rationale) unbedingt einbezogen werden sollen (phasenübergreifend).

Aus der starken Gewichtung der Qualitätssicherungsmaßnahmen Inspektion und Testen sowie der Begründungserfassung von Entscheidungen (Rationale) und deren kontinuierliche Integration in den Prozessablauf, resultiert die Namensgebung des TRAIN-Prozesses (**T**est, **Ra**tionale und **In**spektion unterstützender Softwareentwicklungsprozess).

2.2 Die Rollen des Prozesses

Im Softwareentwicklungsprozess gibt es zwei große Rollengruppen: Rollen aus dem Umfeld des Auftraggebers und Rollen aus dem des Auftragnehmers, meist einer Softwarefirma. Zu den Rollen aus dem Kontext des Auftraggebers gehören dessen Geschäftsleitung (Management) und die zukünftigen Nutzer des Systems. Im Geschäftsprozess des Auftragnehmers findet man ebenfalls die Geschäftsleitung und weiterhin die Systementwickler. Dazu gehören der Requirements Ingenieur, der die

Anforderungen an zu entwickelnde Softwaresysteme aufstellt, der Rationale Maintainer, der das Rationale überprüft, aufrecht erhält und strukturiert, der Inspektor, der die Qualität der Softwaredokumente gewährleisten soll, der Systemdesigner, der die Architekturdefinition und den Feinentwurf entwickelt, der Systemprogrammierer, der die Implementierung des Systems und der Testfälle durchführt und der Tester, der diverse Testaktivitäten wie Testplanung und -spezifikation sowie die Testauswertung übernimmt.
Natürlich sind noch weitere Rollen am Softwareentwicklungsprozess beteiligt; diese Auswahl genügt jedoch für die Betrachtungsperspektive der Arbeit.

2.3 Die Phasen des Prozesses

Der Prozess gliedert sich in mehrere Hauptphasen:

- das Requirements Engineering,
- die Architekturdefinition,
- den Feinentwurf,
- die Implementierung und
- die Qualitätssicherung.

Bevor im weiteren Verlauf des Kapitels die einzelnen Prozessphasen mit ihren Zielen und Entscheidungen überblicksweise erläutert werden, **ist sehr wichtig zu erwähnen, dass die Gliederung des Prozesses in Phasen und später auch in Abstraktionsebenen nicht eine ausschließlich starr sequentielle Abarbeitung zur Folge haben muss.** D. h. es kann durchaus sein, dass man zu einem Zeitpunkt phasenübergreifend Entscheidungen trifft. Als Beispiel seien Architekturentscheidungen angeführt, die nach der Prozessgliederung erst in der Architekturdefinitionsphase nach dem Requirements Engineering getroffen werden. Es ist jedoch verständlich, dass die Architektur maßgeblich von der mit dem System zu realisierenden Aufgabenstellung und damit von der Domäne des Systems bestimmt wird. So sollten/können Aktivitäten der Architekturdefinitionsphase – z. B. in Form der Dokumentation Architektur einschränkender Anforderungen (Qualitätskriterien an die Architektur) – bereits parallel zur Phase des Requirements Engineering stattfinden. **D. h. die auf den ersten Blick starr wirkende Gliederung des Prozesses in seine Phasen dient sehr einer strukturierenden Aufbereitung und kann/sollte durchaus an einigen Stellen aufgebrochen werden.**

2.3.1 Qualitätssicherung

Ziel der Qualitätssicherungsmaßnahmen ist, eine möglichst hohe Qualität des Produktes zu gewährleisten. Um dies optimal zu unterstützen, und um nicht etwa aus Zeitgründen am Projektende Abstriche bei der Qualität in Kauf nehmen zu müssen, werden die Qualitätssicherungsmaßnahmen im TRAIN-Prozess frühstmöglich und kontinuierlich in den Prozessverlauf integriert. Unterscheiden kann man die Maßnahmen in konstruktive und analytische Produktqualitätssicherung:

2.3.1.1 Konstruktive Produktqualitätssicherung

Die konstruktive Produktqualitätssicherung versucht durch systematische Entwicklung, die Produktqualität zu erhöhen. Hierzu zählen die Vorgabe von Aktivitätsabfolgen (Vorgehensmodell), Entscheidungen (Entscheidungsmodell), Notationen, Templates, Werkzeugen, Ausbildung usw.

2.3.1.2 Analytische Produktqualitätssicherung

Die analytische Produktqualitätssicherung versucht, die Qualität durch Prüfprozesse (Inspektion, Testen, Metriken, formale Analyse) sicherzustellen. Im TRAIN-Prozess werden Inspektion und Testen verwendet.

2.3.1.2.1 Inspektion

Durch die Inspektion sollen die Qualitätseigenschaften der Prozessprodukte überprüft werden. Ziel ist, die Qualität der Softwaredokumente zu verbessern, indem diese nach Fehlern durchsucht werden und dadurch ein besseres Endprodukt erreichen.

Im TRAIN-Prozess finden Inspektionen mindestens nach der Phase des Requirements Engineering, der Architekturdefinition sowie nach der des Feinentwurfes statt, um jeweils die fehlerfreie und eindeutige Weiterentwicklung in der nächsten Phase zu unterstützen. Da man die Phasen nicht als sequentielle Aneinanderreihung betrachten kann (vgl. Kapitel 2.3), sollte der Inspektionsprozess zyklisch nach größeren Änderungen in einer Phase und vor dem Eintritt in die folgende Phase wiederholt werden. Abbildung 1 zeigt diese Integration des Inspektionsprozesses in den TRAIN-Prozess.

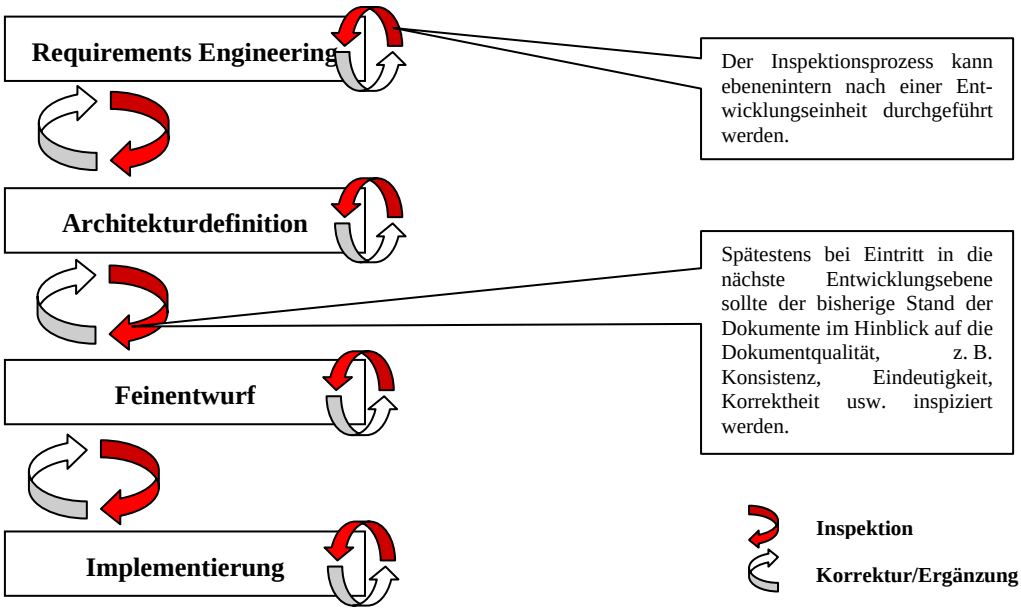


Abbildung 1: Integration der Inspektion in den Entwicklungsprozess

2.3.1.2.2 Testen

Ziel des Testens ist, in einem Programm Fehler aufzudecken. Hierzu zerlegt man es in kleine Einheiten, testet diese, testet dann deren Zusammenspiel und schließlich das komplette System. Hieraus ergeben sich folgende Teststufen:

- Komponententest
- Integrationstest
- Systemtest

Außerdem existieren noch weitere Testarten wie Last- und Akzeptanztest, die hier aber nicht weiter betrachtet werden.

Wie bereits erwähnt, ist eine Charakteristik des TRAIN-Prozesses, auch den Testprozess bereits in den frühen Phasen des Entwicklungsprozesses fest zu integrieren. Hierzu teilt man den Testprozess nochmals orthogonal zu den Teststufen in Testaktivitäten:

- Testplanung
- Testspezifikation
- Testimplementierung
- Testdurchführung, (-protokollierung, -auswertung)

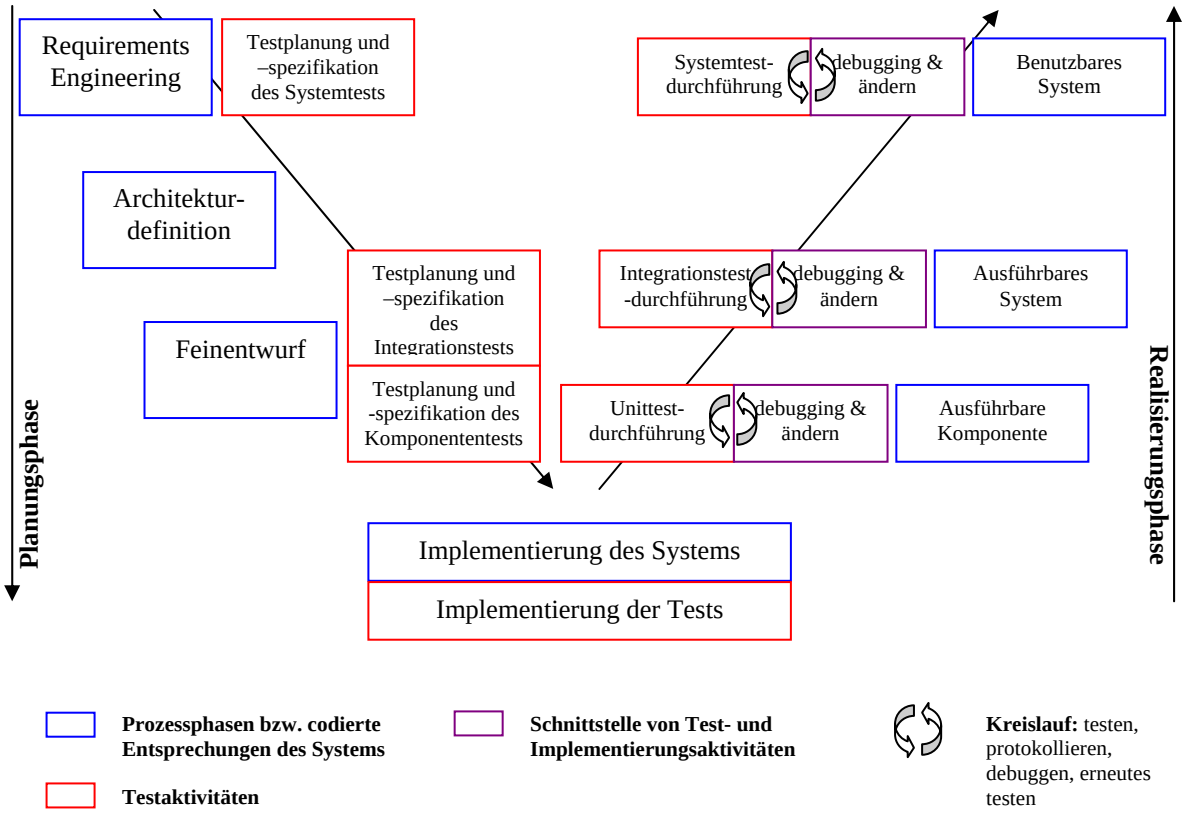


Abbildung 2: Die Integration des Testens in den Entwicklungsprozess

Abbildung 2[1] veranschaulicht die Integration des Testprozesses in den TRAIN-Prozess. In der Abbildung wird deutlich, dass es zu den Phasen des Prozesses jeweils Testentsprechungen gibt (Teststufen). Die Phase der Architekturdefinition hat dabei ihre Testentsprechung gemeinsam mit der des Feinentwurfs in den Integrationstestaktivitäten, wobei letztere Phase ihre Entsprechung ebenfalls in den Komponententestaktivitäten besitzt.

2.3.2 Requirements Engineering

Das Requirements Engineering beschäftigt sich mit der Akquisition von Wissen zu den Anforderungen des Systems und dessen Aufbereitung für die Prozessbeteiligten. Wofür die Software von wem genutzt werden soll, was man sich von dem System erhofft und welchen Bedingungen die Entwicklung unterworfen ist, sind Fragen, die gestellt und beantwortet werden müssen.

[1] Entwickelt auf Grundlage von: SPILLNER, ANDREAS: The W-MODEL – Strengthening the Bond Between Development and Test. S. 3.

Sowohl beim Erwerb, als auch bei der Aufbereitung der Informationen, ist den Bedürfnissen zweier Hauptgruppen an Beteiligten (vgl. Kapitel 2.2) gerecht zu werden:[2]

- Für die Auftraggeber (Management) des Systems muss das Requirements Engineering die Wertsteigerung, für die Nutzer des Systems die Anforderungen identifizieren **(Kundenanforderungen).**
- Für die Systementwickler muss die Basis für die Architekturdefinition, den Feinentwurf und die Implementierung geschaffen werden. D. h. es muss präzise und konsistent festgelegt werden, was und wie Systemdesigner und -programmierer entwickeln sollen und was in welcher Form von den Testern zu validieren ist **(Entwickleranforderungen).**

Die Anforderungsspezifikation, in der die Gestaltungsentscheidungen und Rahmenbedingungen zu den Anforderungen sowie deren Begründungen ihren Niederschlag finden, muss also für beide Gruppen Informationen zur Verfügung stellen und als Diskussionsgrundlage dienen. Da es unmöglich ist, beiden Gruppen dieselben Produkte des Anforderungsermittlungs- und Entscheidungsprozesses als Basis zur Diskussion bzw. Verarbeitung zur Verfügung zu stellen,[3] betreibt man das Requirements Engineering auf verschiedenen Abstraktionsebenen. Jede der niederen Ebenen verfeinert hierbei die nächst höhere. So gibt aber auch jede höhere Ebene den Realisierungsrahmen für die auf sie folgende (nächst tiefere) Ebene vor. D. h. die sich in den Spezifikationsprodukten widerspiegelnden Entscheidungen einer Ebene fließen beim Fortschreiten auf die nächst tiefere Ebene in deren Entscheidungen – und damit in deren Produkte – ein und bestimmen diese.

2.3.2.1 Das Entscheidungsmodell für das Requirements Engineering

Im TRAIN-Prozess unterscheidet man innerhalb des Requirements Engineering folgende Abstraktionsebenen (Entscheidungsebenen): Aufgabenebene (Task Level), Domänenebene (Domain Level), Interaktionsebene (Interaction Level) und Systemebene (System Level). Diese beinhalten die für die gesamte Phase wichtigen (Gestaltungs-) Entscheidungen, die

---

[2] Nach PAECH, BARBARA; KOHLER, KIRSTEN: Task-Driven Requirements in Object-Oriented Development. In: LEITE, JULIO; DOORN, JORGE (Hrsg.): Perspectives on Software Requirements. Dordrecht 2004.

[3] So kann ein Nutzer eines Systems kaum etwas mit einem komplexen Klassen- oder Sequenzdiagramm anfangen, ein Systemprogrammierer kann aus einer Aufgabenbeschreibung (User Task) noch kein Code entwickeln, und ein Tester kann aus einer UI-Struktur keinen Test spezifizieren.

in Hinblick auf die Architekturdefinition, den Feinentwurf und die Implementierung getroffen werden müssen. In Abbildung 3 sind die Abstraktionsebenen mit ihren Entscheidungstypen dargestellt. Diese werden in Kapitel 4 noch detaillierter beschrieben. In Kapitel 2.3 wurde bereits angemerkt, dass trotz der Gliederung des TRAIN-Prozesses in aufeinander folgende Phasen, diese nicht streng sequentiell abgearbeitet werden können/sollten. Dies gilt auch für die Gliederung der Requirements Engineering Phase in die Entscheidungsebenen. Im Folgenden sollen die Entscheidungsebenen kurz erläutert werden.

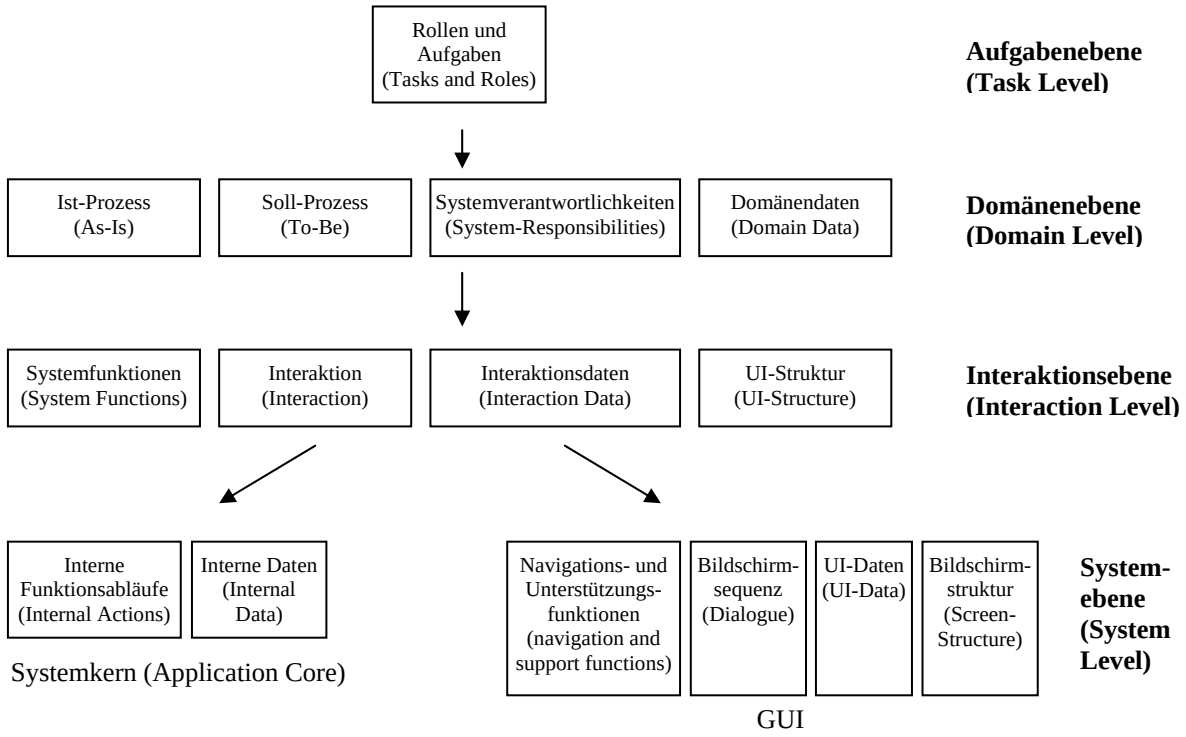


Abbildung 3: Entscheidungstypen des Requirements Engineering

2.3.2.1.1 Die Aufgabenebene

Auf der Aufgabenebene werden Entscheidungen zu Rollen und Aufgaben getroffen, die vom späteren System unterstützt werden sollen. Ausgegangen wird von den Aktoren (Rollen) des Geschäftsprozesses, über die man direkt zu den/deren Aufgaben gelangt. Hierzu greift man häufig auch auf Geschäftsprozessbeschreibungen, Beobachtungen oder Gesprächsprotokolle zurück.

2.3.2.1.2 Die Domänenebene

Die Domänenebene rückt das zukünftige System in den Vordergrund. Hier wird entschieden, wie sich der aktuelle Arbeitsprozess (Ist-Prozess) durch Einführung des Systems unter Einbeziehungen der IT-Technologiemöglichkeiten ändert (Soll-Prozess). Außerdem identifiziert man die Aktivitäten der Aufgaben/des Ist-Prozesses, die vom System übernommen werden sollen (Systemverantwortlichkeiten), sowie die in der Domäne vom System verarbeiteten Daten (Domänendaten).

2.3.2.1.3 Die Interaktionsebene

Die Interaktionsebene fokussiert die Gestaltung der Mensch-Maschinen Schnittstelle. Hier entscheidet man, wie der Mensch mit dem System in Kontakt tritt. Dafür wird bestimmt, in welcher Art die Arbeitsbereiche für den Benutzer strukturiert werden (UI-Struktur) und auf welche Weise der Benutzer seine Aktivitäten mit dem System ausführt (Interaktionsbeschreibung). Um mit dem Benutzer interagieren zu können, muss das System Funktionen bereitstellen, die ebenfalls auf der Interaktionsebene identifiziert und beschrieben werden (Systemfunktionsbeschreibung).

2.3.2.1.4 Die Systemebene

Die Systemebene teilt man in zwei Bereiche: den Systemkern und die graphische Benutzungsschnittstelle (GUI). Letztere beinhaltet z. B. Überlegungen zur Bildschirm-Struktur und Navigation, wohingegen im Bereich des Systemkerns der Übergang von der Anforderungsspezifikation zum objektorientierten Entwurf vollzogen wird. Resultat dieses Teilprozesses ist ein Analyseklassendiagramm, welches die Grundlage für den Feinentwurf und zum Teil für die Architekturdefinition bildet.

2.3.3 Architekturdefinition

Die Architektur eines Softwaresystems beschreibt dessen grundlegende Organisation, verkörpert durch dessen Komponenten – je nach Perspektive Teilsysteme, Packages oder Klassen. Es interessieren in dieser Phase des Prozesses also vor allem die Beziehungen der Komponenten untereinander und zu ihrer Umgebung. Ziel der Architekturdefinition ist, für die Systementwickler den (architektonischen) Rahmen zur Realisierung des (Fein-) Entwurfs festzulegen und für die späteren Anwender einen konzeptionellen Grobüberblick über das System, seine Bestandteile und seine Umgebung zu erarbeiten. Hierbei wird sehr

deutlich, dass sich die Architekturdefinition und der Feinentwurf sehr stark gegenseitig beeinflussen, da Entscheidungen in der einen Phase meist unmittelbare Auswirkungen auf die andere Phase haben.

Das Vorgehen in der Architekturdefinitionsphase besteht darin, über die Entwurfsziele und -prinzipien nachzudenken und darauf aufbauend, die Struktur der Komponenten, Externen Systeme, Ressourcen und Prozesse festzulegen. In diese Überlegungen fließen auch Architekturmuster mit ein, die zum einen Erfahrungswissen zugänglich machen und zum anderen als Grundlage zur Diskussion verschiedener Optionen dienen. Die Ausarbeitung eines Betriebskonzeptes beschreibt die Grenzfälle der Nutzung: (De-) Installation, Hoch- und Herunterfahren, Ressourcen- und Komponentenausfall.

2.3.4 Feinentwurf

Das Ziel des Feinentwurfes ist, aus dem im Requirements Engineering entwickelten Analyseklassendiagramm und den architektonischen Rahmenvorgaben der Architekturdefinition ein Entwurfsmodell zu entwickeln, das als Grundlage für die Implementierung dient.

Im (ab der Systemebene des Requirements Engineering) auf objektorientierte Softwareentwicklung ausgerichteten TRAIN-Prozess sind das in erster Linie ein ausführliches Klassendiagramm und zusätzlich die vollständige Beschreibung der Packages, Klassen und deren Mitglieder (Operationen und Variablen). Das Analyseklassendiagramm wird also zu einem Entwurfsklassendiagramm im Hinblick auf die Implementierung verfeinert, indem immer mehr Details (Sichtbarkeiten von Variablen und Operationen, vollständige Signaturen von Operationen etc.) bestimmt und der Implementierungskontext wie Programmiersprache und in der Architekturdefinition beschriebene bestehende Infrastruktur, Fremdsysteme, Middleware, Wiederverwendung etc. miteinbezogen werden.

2.3.5 Implementierung

Die Implementierung umfasst das Erstellen von Programmkode. Ziel dieses Prozessschrittes ist, qualitativ hochwertigen, d. h. funktionalen, verständlichen, wartbaren und effizienten Programmkode zu erzeugen. Nach den Testaktivitäten Testplanung und -spezifikation der vorangegangenen Phasen, fokussiert die analytische

Produktqualitätssicherung nun die Testaktivitäten Testimplementierung und -durchführung. Hier wird auch der Debuggingzyklus an der Schnittstelle des Testens und Implementierens vollzogen: Test durchführen, Test protokollieren, debuggen und erneutes Testen (vgl. Abbildung 2).

Im Bereich der konstruktiven Produktqualitätssicherung unterstützen im TRAIN-Prozess v. a. Programmierstandards, Werkzeugunterstützung und Vorgaben zu ausreichender Quellkodekommentierung die Kodierungstätigkeiten.

3 Das Entwicklungsbeispiel

In Kaptitel 1 wurde bereits erwähnt, dass die theoretische Erörterung des TRAIN-Prozesses in dieser Arbeit anhand eines Softwareentwicklungsbeispiels praktisch veranschaulicht wird. Das folgende Kapitel stellt dieses Beispiel vor: In Kapitel 3.1 wird die Aufgabenstellung und in Kapitel 3.2 das Softwaresystem beschrieben, in welches das Entwicklungsbeispiel integriert ist.

3.1 Das Beispiel

Das Beispiel für die praktische Erörterung des TRAIN-Prozesses basiert auf einer Aufgabenstellung aus der zu Beginn der Arbeit erwähnten vorlesungsbegleitenden Übung (vgl. Kapitel 1). Die Aufgabenstellung diente als Grundlage für ein Softwareentwicklungsprojekt, das von den Studierenden zu verwirklichen war. Dies umfasste die Realisierung einer Änderungsaufgabe im Softwaresystem Sysiphus. Um für die Studierenden einen gewissen, wenn auch künstlichen industriellen Projektkontext zu erzeugen, wurde die Änderungsaufgabe in ein kleines Szenario eingebettet: *„Unser" Unternehmen hat vor zwei Jahren ein Software-Projekt namens „Sysiphus" betreut. Dabei ist ein Softwaretool namens „REQuest" entstanden, das Softwareentwickler bei ihren Tätigkeiten unterstützt und entlastet. REQuest ermöglicht die Verwaltung von Softwaredokumenten und den dabei getroffenen Entscheidungen. Im Weiteren soll dieses Tool um neue Funktionalität zur Unterstützung des Inspektors erweitert werden.*

Die neue Funktionalität soll also den Inspektor unterstützen, auf welche Weise – etwa durch eine teilweise Automatisierung des Inspektionsprozesses wie in dieser Arbeit realisiert – sollte von den Studierenden mit Unterstützung des TRAIN-Prozesses diskutiert, spezifiziert und implementiert werden. Über Erfahrung in der Domäne verfügten sie insoweit, als sie kleinere Inspektionen anhand von Checklisten bereits selbst durchgeführt hatten. Somit konnte von folgendem Wissen ausgegangen werden:

Der Inspektor ist verantwortlich für die Inspektion der Softwaredokumente. Hierin soll er Fehler finden, zu diesen Fragen formulieren und Verbesserungsvorschläge unterbreiten. Die Arbeit der Inspektorenrolle sah bislang so aus, dass sie v. a. checklistenbasiert durchgeführt wurde. Zu den verschiedenen Anforderungselementen der Softwaredokumente gab es drei Checklisten (Anhang C), die Punkt für Punkt abgearbeitet

werden mussten. Wurde ein Problem gefunden, so wurden Problemlisten zu deren Protokollierung von Hand ausgefüllt und in einem darauf folgenden Treffen aller Inspektoren und des Dokumentenautors (meist der Requirements Ingenieur) besprochen. Wurde das Problem während des Treffens als Fehler festgehalten, musste der Autor diesen später korrigieren.

Beschränkt wurde die Änderungsaufgabe allerdings auf die Inspektion des Anforderungsspezifikationsdokuments. Weitere Dokumentationsbestandteile wie z. B. Architekturdokumentation oder Testfallspezifikation etc., die das Sysiphussystem ebenfalls unterstützt, sollten außen vor gelassen werden.

Die in der Arbeit als Beispiel herangezogene Realisierung der Änderungsaufgabe entstand innerhalb der vorlesungsbegleitenden Übung. Für diese Arbeit wurde sie lediglich aufbereitet und ergänzt, die meisten getroffenen Entscheidungen wurden jedoch nicht revidiert. Daraus resultieren an einigen Stellen Gestaltungsentscheidungen, die aufgrund der gewonnenen Erfahrungen bei einer Neuentwicklung anders getroffen worden wären. Die völlige Neuentwicklung der Änderungsaufgabe hätte jedoch den Rahmen der Arbeit gesprengt.

3.2 Das Softwaresystem Sysiphus[4]

Das in unserem Entwicklungsbeispiel verwendete Softwaresystem nennt sich Sysiphus. Dieses Werkzeug unterstützt die Durchführung von Softwareprojekten, indem es die Erstellung und Modifikation von Softwaredokumenten der verschiedenen Entwicklungsphasen und deren zugehörigen Modellen mit der Kommunikation zwischen den Beteiligten und der Erfassung von Entscheidungsbegründungen (Rationale) vereinigt. Gerade in dieser integrativen Unterstützung der Kommunikation und Begründungserfassung in die Systemmodelle unterscheidet sich das Tool von anderen CASE-Tools. Auftretende Fragen und Kommunikationsströme, wie aus Newsgroups bekannt, können direkt zu den einzelnen Modellelementen wie z. B. Aufgaben, Use Cases, Szenarien, aber auch Klassen oder Testfällen erstellt werden. So werden die Modelle mit

[4] Nach PAECH, BARBARA; BORNER, LARS; u. a.: Vom Kode zu den Anforderungen und wieder zurück: Software Engineering in sechs Semesterwochenstunden. In: LÖHR, KLAUS-PETER; LICHTER, HORST (Hrsg.): Software Engineering im Unterricht der Hochschulen. Heidelberg 2005.

explizitem Wissen angereichert, welches bei der Verwendung von herkömmlichen Kommunikationsmedien wie Email oder Telefon verloren gehen kann oder nicht für alle ProjektteilnehmerInnen zur Verfügung steht. Um dieses explizite Wissen abzurufen, sind die ProjektmitarbeiterInnen jederzeit in der Lage, von den Modellelementen zu den Kommunikationsströmen und Begründungserfassungen hin und zurück zu navigieren.

Sysiphus ist eine verteilte Client-Server Mehrbenutzeranwendung und bietet mit REQuest eine web- sowie mit RAT eine Java Swing-basierte Benutzungsschnittstelle als Client-Anwendung an. Beide Anwendungen arbeiten mit dem gleichen Server und können zeitgleich verwendet werden. Abbildung 4 zeigt die einzelnen Komponenten und deren Integration in eine Schichtenarchitektur. Eine weiterführende Erörterung der Architektur ist in Kapitel 5.6.1 zu finden.

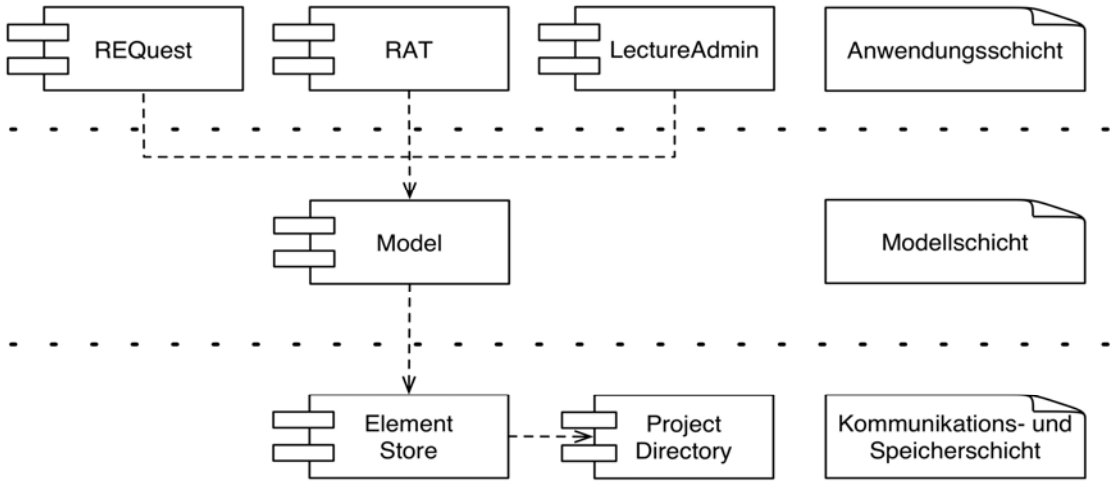


Abbildung 4: Architektur des Sysiphus Projektes

4 Requirements Engineering im TRAIN-Prozess

Das Requirements Engineering umfasst mehrere Teilbereiche:

1. Projektinitiierung
2. Analyse (Wissenserwerb und Konsensbildung)
3. Anforderungsspezifikation
4. Validierung
5. Anforderungsmanagement

Für die vorliegende Arbeit beschränken wir uns im Bereich des Requirements Engineering auf die Beschreibung der Anforderungsspezifikationserstellung, deren grundlegende Bestandteile in Kapitel 4.1 erläutert werden. Zur Erfassung der Anforderungen und deren Spezifikation ist/sind:

- (Gestaltungs-) Entscheidungen zu treffen,
- diese Entscheidungen zu dokumentieren,
- das entsprechende Rationale explizit zu erfassen und
- Qualitätssicherungsmaßnahmen zu planen/integrieren.

Wie und wann diese Aufgaben ausgeführt werden und wie man von Abstraktionsebene zu Abstraktionsebene vorgeht, wird im Folgenden anhand der einzelnen Ebenen und durch die jeweils anschließende Erarbeitung des Entwicklungsbeispiels aufgezeigt. Die Aufgabenebene wird in Kapitel 4.2 (Beispiel in Kapitel 4.3), die Domänenebene in Kapitel 4.4 (Beispiel in Kapitel 4.5), die Interaktionsebene in Kapitel 4.6 (Beispiel in Kapitel 4.7) und die Systemebene in Kapitel 4.8 (Beispiel in Kapitel 4.9) ausführlich beschrieben.

Für einen Überblick über die inhaltliche Gestaltung des Requirements Engineering im TRAIN-Prozess sei nochmals auf die Prozesstabelle in Anhang A und Abbildung 3 verwiesen.

4.1 Die Anforderungsspezifikation

Wie schon in Kapitel 2.3.2 erwähnt, ist das Ziel der Anforderungsspezifikation, das ermittelte Wissen und die getroffenen Entscheidungen den Bedürfnissen der Projektbeteiligten gerecht zur Verfügung zu stellen. Hierzu beinhaltet die Anforderungsspezifikation folgende Elemente:

- Funktionale Anforderungen
- Nicht-funktionale Anforderungen
- Begründungszusammenhänge (Rationale)
- Glossar

4.1.1 Funktionale Anforderungen

Funktionale Anforderungen (Functional Requirements) spezifizieren die Funktionalität des Systems. Sie halten fest, *was* das System und *wie* das System später den Benutzer unterstützt. Beispiele für die Dokumentation Funktionaler Anforderungen sind Aufgaben, Use Cases und Systemfunktionen.

4.1.2 Nicht-funktionale Anforderungen

Neben den Funktionalen Anforderungen werden Nicht-funktionale Anforderungen (Non-functional Requirements (NFR), Qualitätskriterien, Quality Constraints) spezifiziert. Nicht-Funktionale Anforderungen bestimmen *wie gut* die Funktionalen Anforderungen umgesetzt werden müssen.

Teilweise sind Nicht-funktionale Anforderungen untrennbar mit den Funktionalen Anforderungen verbunden. Ein Bespiel dafür ist die Reaktionszeit bei der Kontrolle chemischer Prozesse, d. h. die Nicht-funktionale Anforderung entsteht unmittelbar aus der Funktionalen Anforderung. Oft spielen aber ebenso externe Faktoren eine Rolle. Somit muss man sich z. B. damit beschäftigen, wie schnell ein System generell reagieren muss, so dass es für den Nutzer noch akzeptabel ist. Hierzu sollten auch konkrete Werte genannt werden, die später dann testbar sind.

Im TRAIN-Prozess werden auf den Abstraktionsebenen unterschiedliche Arten Nicht-funktionaler Anforderungen erfasst, die in den Kapiteln zu den Abstraktionsebenen noch genauer erläutert werden:

Auf Aufgabenebene

- Qualitätskriterien von Aufgaben (Quality Constraints of User Tasks)

Auf Domänenebene

- Domänenfaktoren (Domain Constraints)
- Globale Nicht-funktionale Anforderungen (Global Functional Constraints)

Auf Interaktionsebene

- Qualitätskriterien für Use Cases (Quality Constraints on Use Cases)
- Qualitätskriterien für Systemfunktionen (Quality Constraints on Services)

Ab der Domänenebene

- Qualitätskriterien an die Architektur (Architectural Constraints)
- Qualitätskriterien an das GUI (User Interface Constraints)

Die Eigenschaft der Nicht-funktionalen Anforderungen ist, – wie gesagt – dass sie die Güte Funktionaler Anforderungen bestimmen und dabei entweder unmittelbar aus einer Funktionalen Anforderung entstehen oder aber von externen Faktoren herrühren.

Ob die von einer Nicht-funktionalen Anforderung bedingte Güte einer Funktionalen Anforderung eingehalten wird, zeigt sich in der Gestaltung der von ihr abgeleiteten, verfeinerten Funktionalen Anforderungen. Z. B. sollten sich Qualitätskriterien von Aufgaben sowohl in der Gestaltung des Soll-Prozesses als auch der Interaktionsgestaltung (Use Cases) widerspiegeln.

Tendenziell verhält es sich also im TRAIN-Prozess so, dass die Nicht-funktionalen Anforderungen einer höheren Abstraktionsebene die Funktionalen Anforderungen einer tiefer liegenden Abstraktionsebene einschränken/bestimmen. Allerdings kann bei den Qualitätskriterien für Use Cases und Qualitätskriterien für Systemfunktionen der Fall auftreten, dass diese – falls sie Nicht-funktionale Anforderungen einer höheren Abstraktionsebene verfeinern/konkretisieren – ebenfalls die Gestaltungsfreiheit der Use Cases bzw. Systemfunktionen beeinflussen.

4.1.3 Rationale

Unter Rationale versteht man die Dokumentation der Begründungen für Gestaltungsentscheidungen während der Softwareentwicklung. Hierdurch sollen die getroffenen Entscheidungen für jedes Teammitglied nachvollziehbar sein und gewonnenes Wissen nicht verloren gehen. Im TRAIN-Prozess wird ein Großteil der Gestaltungsentscheidungen begründet, indem Gestaltungsalternativen (Optionen) nach Nicht-funktionalen Anforderungen (Kriterien) bewertet werden. Die Gesamtbewertungen aller Optionen werden dann abgewogen und eine Entscheidung für eine Gestaltungsalternative getroffen. Die Nicht-funktionalen Anforderungen, die die Gestaltungsfreiheit einer Funktionalen Anforderung einschränken, sind dabei die zur

Bewertung der Gestaltungsalternativen der Funktionalen Anforderung herangezogene Kriterien.

Die Bewertung von Gestaltungsalternativen Funktionaler Anforderungen gegen Nichtfunktionale Anforderungen findet im TRAIN-Prozess in Tabellenform statt (vgl. z. B. Rationaletabelle 3).

4.1.4 Glossar

Wichtig während der gesamten Softwaredokumentation ist, dass man Begriffe möglichst einheitlich und präzise verwendet. Der Aufbau eines Glossars mit Erklärung der wichtigen und häufig verwendeten Begriffe fördert ein gemeinsames Verständnis, beugt damit Missverständnissen vor und macht die Softwaredokumente einfacher erschließ- und lesbar. Man sollte sich deshalb auch mit den im Glossar eingeführten und erklärten Begriffen auseinandersetzen.

4.2 Die Aufgabenebene

Auf der Aufgabenebene (Task Level) betrachtet man Rollen und Aufgaben des Geschäftsprozesses, die später mit dem System zu tun haben werden. Ziel dieser Ebene ist, Wissen über diese Rollen und Aufgaben zu erhalten, um diese bestmöglich durch das System unterstützen zu können. Entscheidungen werden deshalb über

- den Kontext der Rollen im Geschäftsprozesses und
- den Kontext der Aufgaben im Geschäftsprozess

getroffen.

4.2.1 Funktionale Anforderungen

4.2.1.1 Rollen

Mit den Rollen (Actors) des Geschäftsprozesses wird sich beschäftigt, da man von diesen unmittelbar zu den/deren Aufgaben (Kapitel 4.2.1.2) gelangt. Aber auch die Interessen der Rollen sind für den Entwicklungsprozess von großer Bedeutung, da sie auf potentielle Konfliktursachen hinweisen und zur Erarbeitung von Konfliktlösungen dienen können. Somit werden die Rollenbeschreibungen (Beschreibungstemplate in Anhang B) erarbeitet, um die Verantwortlichkeiten und Interessen einer Rolle zu erfassen sowie deren (Vor-) Wissen, Erfahrungen und Fähigkeiten bezüglich ihrer Aufgaben und bezüglich des Umgangs mit Softwaresystemen festzuhalten.

4.2.1.2 Aufgaben

Die Aufgaben des Geschäftsprozesses bilden die Grundlage für die weiteren Gestaltungsentscheidungen innerhalb des TRAIN-Prozesses. Sie, bzw. Teile von ihnen, sollen später durch das System übernommen werden. Dabei spielt jedoch bei der Identifizierung und Beschreibung der Aufgaben das zu entwickelnde System noch keinerlei Rolle; von diesem wird noch abstrahiert.

Anhaltspunkte über die Aspekte einer Aufgabe, die dokumentiert werden sollten, liefert das Aufgabenbeschreibungstemplate in Anhang B. In diesem werden u. a. die initiierende/beteiligte(n) Rolle(n) der Aufgabe, deren Ursachen und Ziele sowie die benötigten Ressourcen und die Priorität der Aufgabe dokumentiert.

4.2.2 Nicht-funktionale Anforderungen

4.2.2.1 Qualitätskriterien von Aufgaben

Auf der Aufgabenebene ergeben sich Nicht-funktionale Anforderungen aus dem Kontext der identifizierten Aufgaben, sog. Qualitätskriterien von Aufgaben (Quality Constraints on User Tasks). Diese legen fest, wie gut eine Aufgabe durchgeführt werden soll. So beschränken sie die Gestaltungsentscheidungen der Artefakte, die die Aufgabe verfeinern (z. B. Soll-Prozess oder Use Cases). Qualitätskriterien von Aufgaben sind beispielsweise die angestrebte Effizienz einer Aufgabe oder deren Priorität im Vergleich zu anderen Aufgaben.

4.2.3 Qualitätssicherungsmaßnahmen

4.2.3.1 Inspektion

Die Qualitätssicherungsmaßnahmen beginnen auf der Aufgabenebene mit den Inspektionsaktivitäten (vgl. Kapitel 2.3.1.2.1). Anhaltspunkte zu Kriterien, die überprüft werden sollten, finden sich in Anhang C in der „Checkliste - Bereich Rollen, Aufgaben, NFR's, Rolleninstanzen".

4.2.4 Rationale

Nachvollziehbar für die Projektbeteiligten sollten auf dieser Ebene die Wahl der Rollen und Aufgaben sowie deren inhaltliche Gestaltung sein. Warum wurden bestimmte Rollen/Aufgaben des Geschäftsprozesses (nicht) aufgenommen bzw. Aspekte einer Rolle/Aufgabe (nicht) weggelassen?

Bei einer Änderungsaufgabe/Ergänzung eines bestehenden Projektes sollten die Änderungen in den Rollen/Aufgaben begründet werden. Warum wurden (keine) neue(n) Rollen/Aufgaben hinzugenommen bzw. Aufgaben zusammengelegt oder aufgesplittet?

Die Begründung von Gestaltungsentscheidungen anhand von Vorgaben (Nicht-Funktionalen Anforderungen) höherer Abstraktionsebenen ist hier nicht möglich, da die Rollen/Aufgaben keine Umsetzung einer höheren Ebene darstellen.

4.3 Die Aufgabenebene im Beispiel

4.3.1 Funktionale Anforderungen

4.3.1.1 Rollen

Das in Kapitel 3.1 vorgestellte Entwicklungsbeispiel soll das System um zusätzliche Funktionalität zur Unterstützung der Inspektorenrolle bei der Inspektion der Anforderungsspezifikation erweitern. Die Aufgabenstellung gibt also schon konkret die Rolle vor, über die Entscheidungen auf dieser Ebene getroffen werden. Da die Rolle bereits in der bestehenden Systemdokumentation vorhanden ist, muss untersucht werden, ob diese Beschreibung ausreicht bzw. ob sie aufgrund der Änderungsaufgabe ergänzt werden muss. In Rollenbeschreibung 1 stellen wir die folgenden Sachverhalte in Templateform zusammen:

> *Der Inspektor ist eine Rolle aus dem Umfeld des Software Engineering mit genügend technischen Kenntnissen. Bestenfalls hat er aber ebenso ausreichend Distanz zum Projekt, um fehlende Anforderungen oder unzureichend dokumentierte Entscheidungen nicht als implizit zu erachten. Dies ist notwendig, damit er sein Ziel, Fehler in den Softwaredokumenten zu finden, hierzu Fragen zu formulieren und Verbesserungsvorschläge zu unterbreiten, erreichen kann. Das Resultat der Arbeit eines Inspektors soll ein verbessertes[5] Softwaredokument sein.*

4.3.1.2 Aufgaben

Das Hauptziel der Aufgabenebene und deren Aktivitäten ist jedoch, die Aufgaben des Geschäftsprozesses zu identifizieren, da später Teile dieser Aufgaben (Aktivitäten) vom System unterstützt bzw. übernommen werden sollen. Nun gibt es grundsätzlich bei einer Änderungsaufgabe verschiedene Möglichkeiten, die Anforderungen zu erweitern. Entweder

- die in der Änderungsaufgabe erwünschten Aktivitäten sind bereits in den bestehenden Aufgabenbeschreibungen vorhanden und es ändert sich nichts,
- die beschriebene Aufgabe muss erweitert werden oder
- es muss eine neue Aufgabe hinzugefügt werden.

Rollenbeschreibung (Actor)	
Name:	**INSPEKTOR**
Verantwortlichkeiten:	Der Inspektor ist verantwortlich für die Inspektion der Softwaredokumente. Hierin soll er potentielle Fehler (Probleme) finden, zu diesen Fragen formulieren, und Verbesserungsvorschläge unterbreiten
Erfolgskriterien:	Verbessertes Softwaredokument
Aufgaben:	Überprüfen der Anforderungen
Kommunikationspartner:	Andere Inspektoren, Requirements Ingenieur (Autoren der Dokumente), Rationale Maintainer
Innovationsgrad:	Mittel, da noch nicht „flächendeckend“ in der Praxis etabliert
Rollenprofil	
Wissen/Erfahrung/Fähigkeiten bzgl.	
Aufgaben	Mittel bis hoch, da regelmäßige Durchführung der Aufgaben (aber: etwas Distanz zum Projekt, um fehlende Anforderungen oder nicht dokumentierte Entscheidungen nicht als implizit zu erachten)
Softwaresystem	Mittel, da i. a. aus technischem Umfeld kommend

Rollenbeschreibung 1: Inspektor

In der Dokumentation von Sysiphus existiert bereits eine Aufgabe „Überprüfen der Anforderungen“ des Inspektors. Im Fall unserer Änderungsaufgabe sollte diese Aufgabe erweitert werden, da sie den Inspektionsprozess bislang nur ausgerichtet auf eine herkömmliche Inspektion ohne Systemunterstützung beschreibt. Dies ist – wie wir in Kapitel 4.2.1.2 bereits angemerkt haben – auf der Aufgabenebene auch intendiert. Wir werden die Beschreibung aber vorausschauend dahingehend erweitern, dass wir auf dieser Abstraktionsebene den später vom System unterstützten Aufgabenteil bereits versuchen, in die Aufgabenbeschreibung mit einzubeziehen (selbstverständlich ohne das System bereits ins Spiel zu bringen):

> *Bei der Aufgabe, eine Inspektion durchzuführen hat der Inspektor mit den verschiedenen Reviewverfahren mehrere Vorgehensweisen zur Auswahl. Ziel all dieser Techniken ist, den Inspektor möglichst umfassend und effizient darin zu unterstützen, potentielle syntaktische und semantische Fehler (Probleme) in den Dokumenten zu finden.*
>
> *Syntaktische Fehler sind in diesem Zusammenhang Fehler, die gegen feste Regeln beim Aufstellen der Dokumente verstoßen. Sie betreffen v. a. die Struktur der Elemente eines Dokuments und deren Beziehungen zueinander. Beispiel eines syntaktischen Fehlers wäre ein nicht ausgefüllter Abschnitt eines Templates (z. B.*

[5] In IEEE Std. 830-1998 sind die Eigenschaften einer guten Anforderungsspezifikation festgelegt.

Verantwortlichkeit bei Rollenbeschreibung) oder eine Nicht-funktionale Anforderung für Systemfunktionen, die keine Systemfunktion einschränkt.

Semantische Fehler sind meist schwerer als syntaktische zu entdecken. Sie betreffen z. B. die Verständlichkeit, Eindeutigkeit oder Konsistenz in der Beschreibung eines Dokumentationselementes bzw. elementübergreifend. Weitere semantische Fragestellungen könnten sein, ob das Dokument vollständig ist oder alle Anforderungen realisierbar sind.

Wird ein Problem während der Inspektion gefunden, so wird es in eine Problemliste eingetragen und evtl. ein Verbesserungsvorschlag festgehalten.

Da die (Anforderungs-) Spezifikationsdokumente die Grundlage für den Entwurf und die Implementierung des Systems sind und diese Schritte meist von anderen Personen als von den Autoren der Dokumente vollzogen werden, muss deren höchste Qualität gewährleistet sein. Hierzu ist die Inspektion ein sehr wichtiges Mittel. Deshalb hat sie eine hohe Priorität und wird regelmäßig ausgeführt. Ihre Komplexität ist recht unterschiedlich: es hängt davon ab, ob es gilt, syntaktische oder semantische Probleme zu finden.

In der überarbeiteten Beschreibung der bestehenden Aufgabe, die in Templateform in Aufgabenbeschreibung 1 dargestellt wird, findet nun eine Trennung zwischen syntaktischen und semantischen Fehlern statt. Genau dieser Punkt arbeitet den folgenden Spezifizierungsschritten auf den kommenden Abstraktionsebenen zu. Hier wird zwar noch nichts von der Arbeit des Systems gesagt, es ist jedoch ersichtlich, dass das System genau die in der Beschreibung herausgearbeitete syntaktische Überprüfung der Dokumente realisieren kann.

Anstatt die Aufgabe „Überprüfen der Anforderungen“ beizubehalten und nur zu ergänzen, könnte man auch auf die Idee gekommen sein, diese bestehende Aufgabe in zwei Aufgaben „Anforderungselemente manuell überprüfen“ und „Anforderungselemente automatisch überprüfen“ aufzuspalten. Da auf der Aufgabenebene die Systemunterstützung jedoch in der Aufgabenidentifizierung noch keine Rolle spielt, sollte diese Trennung hier nicht erfolgen.

In der Aufgabenbeschreibung wurden neue Begriffe eingeführt. Um beispielsweise ein gemeinsames Verständnis von *syntaktischen Fehlern* und *semantischen Fehlern* in einem Softwaredokument unter den Beteiligten des Softwareentwicklungsprozesses zu erlangen, sollten zumindest diese Begriffe in das Glossar der Softwaredokumentation aufgenommen werden (vgl. Kapitel 4.1.4).

Aufgabenbeschreibung (User Task)	
Name:	**ÜBERPRÜFEN DER ANFORDERUNGEN**
Verantwortliche Rolle:	Inspektor
Beteiligte Rollen:	Keine
Aufgabenbewertung	
Ziel:	Alle potentiellen syntaktischen und semantischen Fehler (Probleme) in den (Anforderungs-) Spezifikationsdokumenten sollen gefunden, dokumentiert und Verbesserungsvorschläge erarbeitet werden
Eingriffsmöglichkeiten:	Ablauf wird von Reviewverfahren bestimmt
Ursachen:	Qualitätssicherung der Dokumente
Priorität:	Hoch, wegen höchster Qualitätsanforderungen der Dokumente für weitere Arbeit (Entwurf und Implementierung)
Aufgabendurchführung	
Durchführungsprofil (Häufigkeit, Kontinuität, Komplexität):	- Mittelmäßig häufig, aber regelmäßig - Unterbrechungen möglich - Mittlere bis hohe Komplexität
Ausgangssituation (Vorbedingung):	- Dokument vorhanden und Zugriff auf Dokument - Baseline der Dokumente
Info-In:	Zu inspizierende Softwaredokumente
Info-Out:	Problemliste
Ressourcen (z. B. Arbeitsmittel etc.):	Dokumente das Reviewverfahren betreffend

Aufgabenbeschreibung 1: Überprüfen der Anforderungen

4.3.2 Nicht-funktionale Anforderungen

4.3.2.1 Qualitätskriterien von Aufgaben

Für unsere veränderte Aufgabe „Überprüfen der Anforderungen“ besteht in der Softwarespezifikation des Sysiphussystems bereits ein Qualitätskriterium von Aufgaben, das für unsere Entwicklung relevant wird:

> ***Effizienz bei der Inspektion***
> *Die Inspektion des Anforderungsspezifikationsdokuments mit bis zu 40 Anforderungselementen darf nicht länger als 30 min dauern.*

Qualitätskriterium von Aufgaben 1: Effizienz bei der Inspektion

Neue Nicht-funktionale Anforderungen kommen zu diesem Zeitpunkt nicht hinzu.

4.3.3 Rationale

Als Rationale auf dieser Ebene sollte begründet werden, weshalb man für die Änderungsaufgabe keine neue Rolle bzw. Aufgabe hinzugefügt hat. Hierzu reicht die Anmerkung, dass durch die Änderungsaufgabe keine neue Rolle bzw. Aufgabe hinzugekommen ist, die nicht schon in der bestehenden Dokumentation vorhanden war. Ebenfalls nachvollziehbar sollte sein, dass die Aufgabe „Überprüfen der Anforderungen“ verfeinert wurde, um auf eine Systemunterstützung hinzuarbeiten (in erster Linie durch die Trennung von syntaktischen und semantischen Fehlern), und warum die Aufgabe nicht in zwei Aufgaben „Anforderungselemente manuell überprüfen“ und „Anforderungselemente automatisch überprüfen“ aufspaltet wurde. Wie weiter oben begründet, kommt aufgrund des Abstraktionsgrades die Aufspaltung zwar streng genommen erst gar nicht in Frage, es könnte jedoch vor allem neuen Teammitgliedern eine Hilfe sein, dies nochmals ausdrücklich zu erwähnen.

4.4 Die Domänenebene

Auf der Domänenebene (Domain Level) sollen die (potentiellen) Einflüsse des zukünftigen Systems auf den Geschäftsprozess und seine Beteiligten offen gelegt werden. Ziel ist, das System sinnvoll und mit Mehrwert für Auftraggeber und Nutzer in den Geschäftsprozess der Domäne zu integrieren. Um dies zu erreichen, werden auf der Domänenebene Entscheidungen getroffen zu

- den (IT-) Potentialen des Systems in der Domäne (Soll-Prozess),
- den Aktivitäten (der Aufgaben) an denen das System beteiligt sein wird (Systemverantwortlichkeiten) und
- den Daten und Konzepten, die vom System verarbeitet und manipuliert werden (Domänendaten).

Grundlage dieser Entscheidungen sind die auf der Aufgabenebene erstellten Artefakte sowie eine Beschreibung des aktuellen Geschäftsprozesses (Ist-Prozess), die zu Beginn der Betrachtungen auf der Domänenebene erstellt wird.

4.4.1 Funktionale Anforderungen

4.4.1.1 Ist-Prozess

Analysiert man die Ist-Prozesse (As-Is), so zerlegt man die Aufgaben in kleinschrittige Bestandteile (Aktivitäten). Dabei interessiert der momentane Ablauf der wesentlichen Aufgaben und Aktivitäten des Geschäftsprozesses ohne die zukünftige Systemunterstützung. Die Entscheidungen über die Ist-Aktivitäten bilden die Grundlage für die Entwicklung des potentiellen Soll-Prozesses mit seinen Soll-Aktivitäten und damit der Identifizierung der Systemverantwortlichkeiten.

Je nach Dokumentationsmethode werden die Betrachtungen vor allem textuell und/oder mit Aktivitätsdiagrammen festgehalten. Neben der Integration der Aktivitätsbeschreibungen in die Aufgabenbeschreibung ist es auch möglich, den Ist-Prozess im Glossar der Softwaredokumentation unter dem Stichwort der Aufgabe/Aktivität zu beschreiben.

4.4.1.2 Soll-Prozess

In diesem Schritt wird entschieden, wie sich die im Ist-Prozess ermittelten Aktivitäten durch den Einsatz neuer IT-Technologien ändern können. Hierzu müssen die Potentiale durch IT-Experten analysiert und die dadurch hervorgerufenen Änderungen des Arbeitsprozesses und deren Alternativen mit dem Auftraggeber und den Nutzern diskutiert werden. Die Dokumentation der Entscheidungen des Soll-Prozesses (To-Be) findet analog zu denen des Ist-Prozesses statt.

4.4.1.3 Systemverantwortlichkeiten

Nicht unbedingt alle im Ist-Prozess identifizierten und im Soll-Prozess weiterhin bestehenden Aktivitäten sollen/können später durch das System übernommen werden. Die Aktivitäten, für die das System später (mit-) verantwortlich sein soll, nennt man Systemverantwortlichkeiten (System Responsibilities). Diese werden in einer Systemverantwortlichkeitenübersicht (Use Case-Diagramm) dokumentiert, welche die Systemverantwortlichkeiten auflistet und die Beziehungen zu den ausführenden/beteiligten Rollen aufzeigt.

4.4.1.4 Domänendaten

Vom System unterstützte Aktivitäten verändern Daten. Diese und andere Konzepte, die im Kontext des Systems wichtig sind, fasst man mit deren Beziehungen als Entitäten und Attribute unter dem Begriff *Domänendaten* zusammen. Der Unterschied zwischen Entitäten und Attributen besteht bezüglich der Werte, mit denen sie belegt sind. Ein Attribut wird definiert über den Wert den es innehat. Ändert sich der Wert, so ist das Attribut nicht mehr dasselbe. Eine Entität dagegen besitzt eine Identität, welche sie auch behält, wenn sich zu ihr gehörige Attribute ändern. Ein Beispiel wäre ein Buch, das eine eigene Identität und mehrere Attribute (ISBN, Titel, Preis, Beschreibung etc.) besitzt. Ändert sich nun der Wert des Attributes *Preis* bzw. *Beschreibung*, so bleibt das Buch immernoch dasselbe.

Zur Identifizierung der Entitäten werden die Problem- und Aufgabenbeschreibungen sowie die Artefakte aus der Ist- und Soll-Prozessbetrachtung herangezogen. Die Entitäten mit ihren Beziehungen und gegebenenfalls Attributen werden dann mittels eines Domänendatendiagramms (Entity-Relationship-Diagramm, ER-Diagramm) modelliert und möglicherweise ergänzend eine ausführliche Beschreibung des Diagramms, z. B. im

Glossar hinzugefügt. Das Domänendatendiagramm bildet mit den noch zu entwickelnden Klassendiagrammen den statischen Teil der Systemmodellierung.
Im Zuge der Erarbeitung der Domänendaten sollte man die Eingangsartefakte nochmals auf Konsistenz und Vollständigkeit prüfen. Identifiziert man Daten bzw. Konzepte, die in keinem der Artefakte erwähnt werden, so muss man höchstwahrscheinlich Ergänzungen vornehmen.

4.4.2 Nicht-funktionale Anforderungen

4.4.2.1 Qualitätskriterien von Aufgaben

Aufgrund der Betrachtung des Ist-Prozesses ergeben sich auf der Domänenebene weiterhin Qualitätskriterien von Aufgaben. Dies resultiert daraus, dass die Beschreibung des Ist-Prozesses die wesentlichen Aspekte der Aufgaben des Geschäftsprozesses enthält (vgl. Kapitel 4.4.1.1). Somit ist sie eigentlich ein Teil einer Aufgabe. Daher werden aus den Ist-Prozessbetrachtungen entstehende Nicht-funktionale Anforderungen den Qualitätskriterien von Aufgaben zugeordnet.

4.4.2.2 Domänenfaktoren

Außerdem beginnt auf der Domänenebene die Identifizierung und Dokumentation der Domänenfaktoren (Domain Constraints). Diese ergeben sich aus der Anwendungsdomäne des Systems (meist bei den Ist- und Soll-Prozessbetrachtungen) und machen deshalb Vorgaben für alle weiteren Gestaltungsentscheidungen. Dazu können sie verfeinert/konkretisiert werden, wodurch sie an spezielle Funktionale Anforderungen (z. B Aufgaben, Use Cases) angepasst werden. Beispiel für einen Domänenfaktor ist die effiziente Gestaltung des Geschäftsprozesses, um einen Wettbewerbsvorteil zu erlangen. Dieser könnte z. B. an eine spezielle Aufgabe angepasst werden, indem man konkrete Werte zur Erlangung der Effizienz in dieser Aufgabe formuliert. Daraus ergäbe sich dann ein Qualitätskriterium von Aufgaben, das einen Domänenfaktor verfeinert.

4.4.2.3 Globale Nicht-funktionale Anforderungen

Globale Nicht-funktionale Anforderungen (Global Functional Constraints) sind übergreifende Nicht-funktionale Anforderungen, die die Artefakte auf der Interaktions- und Systemebene einschränken oder noch in detailliertere Nicht-funktionale Anforderungen zerlegt werden müssen. Sie entstehen meist bei der Soll-

Prozessbetrachtung auf der Domänenebene, aber auch noch zu Beginn der Interaktionsebene. Exemplarisch ist die Anforderung, dass innerhalb des Systems möglichst wenig Sichten auf Daten und Funktionen existieren sollen, um die Übersichtlichkeit für den Benutzer zu wahren.

4.4.2.4 Qualitätskriterien an die Architektur

Aus den identifizierten Qualitätskriterien an die Architektur ergeben sich Einschränkungen der Gestaltungsentscheidungen in der Phase der Architekturdefinition aber auch in der des Feinentwurfs.[6] Qualitätskriterien an die Architektur ergeben sich zu einem großen Teil in der Requirements Engineering Phase und besonders auf der Domänenebene, da in der Soll-Prozessbetrachtung u. a. die IT-Realisierungsmöglichkeiten in Betracht gezogen werden, die oft eng mit architektonischen Aspekten des Softwaresystems verknüpft sind. Somit Außerdem verfeinern die Qualitätskriterien an die Architektur auch die Domänenfaktoren, die ebenfalls aus den Ist- und Soll-Prozessbetrachtungen entstehen. Als Beispiel sei ein Qualitätskriterium an die Architektur genannt, das die Möglichkeit des weltweiten Zugriffs auf das System fordert. Dies könnte aus einem Domänenfaktor entstehen, der formuliert, dass an Projekten die das System verwaltet, weltweit tätige Mitarbeiter beteiligt sind.

4.4.3 Qualitätssicherungsmaßnahmen

Neben den Inspektionsaktivitäten können auf dieser Ebene die Testaktivitäten in Form der Systemtestplanung beginnen.

4.4.3.1 Inspektion

Die Inspektion auf dieser Ebene schließt die Inspektion der Aufgabenebene selbstverständlich mit ein. Zusätzlich muss vor allem die Konsistenz zwischen den einzelnen (neu hinzugekommenen) Artefakten gewährleistet sein (z. B. den Beschreibungen und Diagrammen).

4.4.3.2 Testen

4.4.3.2.1 Systemtestplan

Auf der Domänenebene beginnen die Testaktivitäten mit der Systemtestplanung. Im Systemtestplan wird festgelegt, wann das System aus Sicht eines Systemtests als getestet

anzusehen ist (Testendekriterien), auf welcher Grundlage die Systemtestspezifikation entwickelt wird und welche Testressourcen benötigt werden. Außerdem sollte im Systemtestplan eine Gliederung in Systemtestteile und deren Priorisierung stattfinden. Die Gliederung kann beispielsweise anhand der Systemverantwortlichkeiten (bzw. der daraus entwickelten Use Cases) stattfinden (vgl. Kapitel 4.5.3.1.1), so dass später der komplette Systemtest aus mehreren Systemtests pro Systemverantwortlichkeit (bzw. Use Case) besteht.

4.4.4 Rationale

Die Begründung der Gestaltung des Soll-Prozesses steht auf der Domänenebene im Mittelpunkt. Warum hat man sich entschieden, bestimmte Prozesse zu ändern und warum hat man bestimmte IT-Technologie bevorzugt? Zur Begründung der Artefakte, insbesondere des Soll-Prozesses, sollten u. a. die aus der Anwendungsdomäne und dem Ist-Prozess entstandenen Domänenfaktoren und Qualitätskriterien von Aufgaben herangezogen werden.
Des Weiteren sollte die Planung des Systemtests nachvollziehbar sein.

[6] Die Qualitätskriterien an die Architektur für die Phase des Feinentwurfs entstehen häufig aus den Entwurfszielen (Kapitel 5.1) bzw. den objektorientierten Entwurfsprinzipien (vgl. Kapitel 5.2).

4.5 Die Domänenebene im Beispiel

4.5.1 Funktionale Anforderungen

4.5.1.1 Ist-Prozess

Zur Dokumentation des Inspektionsprozesses ohne Systemunterstützung (Ist-Prozess) verwenden wir einen Glossareintrag in der Softwaredokumentation namens „Inspektionsprozess (Ist-Prozess)“. Hierin wird Schritt für Schritt der bisherige Ablauf der Überprüfung der Anforderungselemente mit Hilfe der Checklisten beschrieben. Auch etwaige Probleme oder Nachteile des Ist-Prozesses sollten diskutiert werden und sich in Nicht-funktionalen Anforderungen niederschlagen (vgl. Kapitel 4.5.2):

> ***Inspektionsprozess (Ist-Prozess)***
> *Der folgende Abschnitt beschreibt den Ablauf des Inspektionsprozesses, wie er ohne Systemunterstützung praktiziert wird.*
> *Ist eine Inspektion angesetzt, so beginnt jeder Inspektor anhand der vorher bestimmten Lesetechnik mit der Suche nach Problemen. Im vorliegenden Fall wird die Inspektion anhand des checklistenbasierten Lesens vollzogen. Das bedeutet, dass jeder Punkt der Checkliste (Prüfkriterium) für jedes einzelne Dokumentationselement (bislang entspricht dies jedem Anforderungselement) geprüft werden muss. Hierbei kann der Inspektor entweder punktweise oder elementweise vorgehen. Punktweises Vorgehen heißt, dass für ein Prüfkriterium jedes Element untersucht wird, bevor zum nächsten Kriterium der Checkliste weitergegangen wird. Elementweise bedeutet, jedes Element auf alle Kriterien hin zu untersuchen, bevor das nächste Element an die Reihe kommt.*
> *Hat der Inspektor ein Problem entdeckt, so trägt er dieses und gegebenenfalls einen Verbesserungsvorschlag in die Problemliste ein. Die Vorbereitung des Inspektors ist somit abgeschlossen, und es kommt zum Treffen der Inspektoren und der Dokumentenautoren. Hierbei werden die Probleme besprochen und gegebenenfalls als priorisierte Fehler in der Fehlerliste protokolliert. Die Autoren nehmen dann mittels dieser die Korrektur der Fehler vor.*
> *Ein großes Problem des praktizierten Verfahrens ist, dass es sehr stark auf syntaktische Probleme hinweist, die schwerer zu entdeckenden semantischen Fehler aber seltener aufgedeckt werden. Dies resultiert u. a. aus der schnell auftretenden Ermüdung und Nachlässigkeit der Inspektoren, wenn sie in sehr vielen Elementen die „lästigen“ syntaktischen Fehler suchen müssen.*

Glossareintrag 1: Inspektionsprozess (Ist-Prozess)

4.5.1.2 Soll-Prozess

Auf dem Ist-Prozess aufbauende Überlegungen der potentiellen Systemunterstützung (Soll-Prozess) und die Vorgaben der Nicht-funktionalen Anforderungen führen in unserem Beispiel zu der Entscheidung, möglichst das System die Überprüfung der Anforderungselemente auf syntaktische Probleme vornehmen zu lassen. Die Bewertung der Gestaltungsalternativen gegen die Nicht-funktionalen Anforderungen für den Soll-Prozess ist in Rationaletabelle 1 in Kapitel 4.5.4 festgehalten. Das genaue Vorgehen für den zukünftigen Soll-Prozess wird in einem Glossareintrag in der Softwaredokumentation mit Namen „Inspektionsprozess (Soll-Prozess)“ erläutert (Glossareintrag 2).

> ***Inspektionsprozess (Soll-Prozess)***
>
> *Der folgende Abschnitt beschreibt den Ablauf des Inspektionsprozesses, wie er mit Systemunterstützung praktiziert werden soll.*
>
> *Hierbei ändert sich der Inspektionsprozess ohne Systemunterstützung (Inspektionsprozess (Ist)) dahingehend, dass das System die Suche nach den syntaktischen Problemen in den Dokumentationselementen weitgehend selbstständig (nachdem sie vom Inspektor angestoßen wurde) durchführt und diese Probleme berichtet.*
>
> *Der Inspektor bekommt dadurch die Möglichkeit, stärker auf semantische Probleme in den Dokumenten zu achten.*
>
> *Der Ablauf der Aktivitäten des Inspektionsprozesses ändert sich nicht, außer dass gegebenenfalls neben den von den Inspektoren nun hauptsächlich untersuchten semantischen Probleme zusätzlich die vom System identifizierten syntaktischen Probleme beim Treffen der Inspektoren und der Dokumentenautoren diskutiert werden sollten.*
>
> *Wie genau die syntaktische Überprüfung der Anforderungselemente ausgeführt wird, ist unter dem Glossareintrag „Syntaktische Anforderungselementeüberprüfung (des Systems)“ in der Softwaredokumentation erfasst.*

Glossareintrag 2: Inspektionsprozess (Soll-Prozess)

Die Unterstützung des Inspektors soll sich laut Aufgabenstellung für das Beispiel auf die Anforderungsspezifikation beschränken. Die genaue Erarbeitung der syntaktischen Überprüfung der einzelnen darin enthaltenen Anforderungselemente basiert in erster Linie auf Überlegungen, wie diese Elemente in Beziehung stehen und welche Informationen in den Beschreibungstemplates nicht fehlen dürfen.

Aus den Checklisten des bisherigen Inspektionsprozesses (Anhang C) identifiziert man die Fragen, die auf syntaktische Fehler hinweisen[7] und legt daraus die genauen syntaktischen Überprüfungen durch das System im Soll-Prozess fest. Zur Dokumentation verwenden wir Glossareintrag 3 („Syntaktische Anforderungselementeüberprüfung (des Systems)“) in der Softwaredokumentation. Dieser wird durch ein ER-Diagramm ergänzt, das die durch die syntaktische Anforderungselementeüberprüfung untersuchten Beziehungen zwischen den Elementen darstellt.

Syntaktische Anforderungselementeüberprüfung (des Systems)

In diesem Abschnitt wird detailliert aufgelistet, welche Prüfkriterien bei der syntaktischen Überprüfung der Anforderungselemente zugrunde gelegt werden.

Bei der Prüfung werden folgende Elemente miteinbezogen:

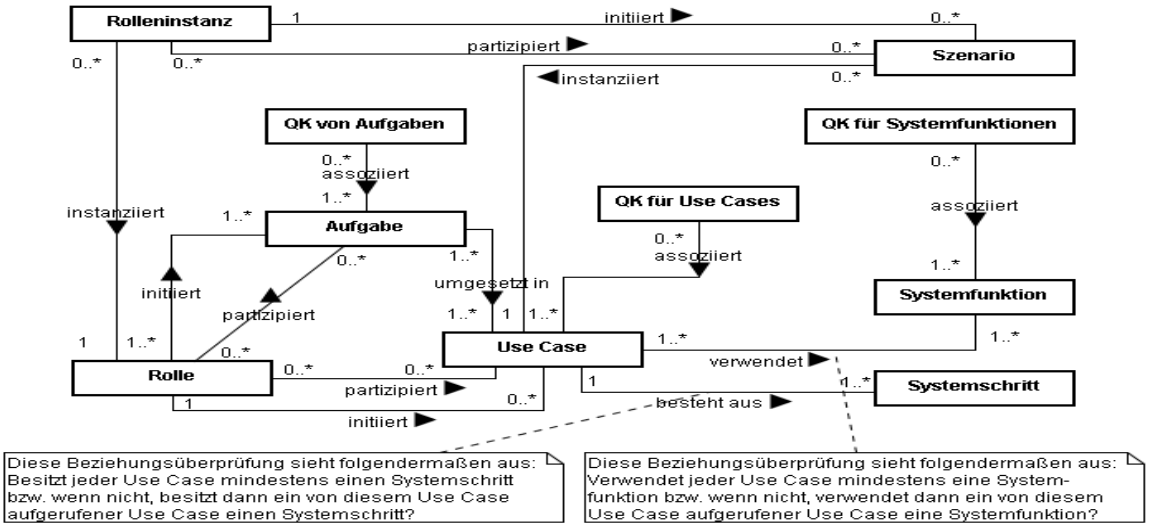
- *Aufgabenbeschreibungen (User Task)*
- *Rollenbeschreibungen (Actor)*
- *Rolleninstanzen (Actor Instance)*
- *Nicht-funktionale Anforderungen (NFR)*
- *Use Cases*
- *Szenarien (Szenario)*
- *Systemfunktionen (Service)*

Bei diesen Elementen werden folgende Kriterien (Prüfkriterien) geprüft:

- ***Alle Elemente***
  - *Besitzt jedes Element eine Beschreibung?*
- ***Rollenbeschreibungen***
  - *Ist jede Rolle mindestens für eine Aufgabe verantwortlich oder mindestens an einer Aufgabe beteiligt?*
- ***Aufgabenbeschreibungen***
  - *Hat jede Aufgabe eine verantwortliche Rolle?*
  - *Wird jede Aufgabe durch mindestens einen Use Case umgesetzt?*
- ***Use Cases***
  - *Realisiert jeder Use Case eine Aufgabe?*

[7] Auf syntaktische Fehler in den Anforderungsdokumenten weisen hin (vgl. Anhang C):
in der „Checkliste - Bereich Rollen, Aufgaben, NFR’s, Rolleninstanzen“ die Fragen 2, 9, 11, 12, 13,
in der „Checkliste - Bereich Use Cases“ die Fragen 2, 11, 12 und
in der „Checkliste - Bereich Systemfunktionen“ die Fragen 4 und 6.

  - *Besitzt jeder Use Case mindestens einen Systemschritt (System Step) bzw. wenn nicht, besitzt dann ein von diesem Use Case aufgerufener Use Case einen Systemschritt (rekursiv)?*
  - *Verwendet jeder Use Case mindestens eine Systemfunktion (Service) bzw. wenn nicht, verwendet dann ein von diesem Use Case aufgerufener Use Case eine Systemfunktion (rekursiv)?*
- ***Systemfunktionsbeschreibungen***
  - *Wird jede Systemfunktion in mindestens einem Use Case verwendet?*
- ***Nicht-funktionale Anforderungen (NFR, (Qualitäts-)Kriterium) auf jeder Ebene***
  - *Ist jede NFR in einer Frage (zur Abwägung von Entscheidungen) oder in Bezug auf ein Element (als Einschränkung) verwendet worden?*
- ***Qualitätskriterien von Aufgaben***
  - *Ist jedes Qualitätskriterium von Aufgaben mindestens einer Aufgabe zugeordnet?*
- ***Qualitätskriterien für Use Cases***
  - *Ist jedes Qualitätskriterium für Use Cases mindestens einem Use Case zugeordnet?*
- ***Qualitätskriterien für Systemfunktionen***
  - *Ist jedes Qualitätskriterium für Systemfunktionen mindestens einer Systemfunktion zugeordnet?*
- ***Rolleninstanz (Actorinstance)***
  - *Ist jede Rolleninstanz einer Rolle zugeordnet?*
- ***Szenarien***
  - *Ist jedes Szenario genau einem Use Case zugeordnet?*
  - *Hat jedes Szenario eine verantwortliche Rolleninstanz?*



Glossareintrag 3: Syntaktische Anforderungselementeüberprüfung (des Systems)

4.5.1.3 Systemverantwortlichkeiten

Aus der Beschreibung des Soll-Prozesses erhält man die zukünftigen Systemverantwortlichkeiten. Im Falle unserer Änderungsaufgabe ist die neue Systemverantwortlichkeit die syntaktische Überprüfung der Anforderungselemente und das Berichten der Probleme („Anforderungsdokumente automatisiert überprüfen“). Die Modellierung der neuen Systemverantwortlichkeit wird mittels einer Systemverantwortlichkeitenübersicht realisiert, die in Abbildung 5 dargestellt ist. Sie enthält die identifizierten Systemverantwortlichkeiten und deren Beziehungen zu den an ihnen beteiligten Rollen. In der Abbildung sind neben der neuen Systemverantwortlichkeit bereits in der Softwaredokumentation von Sysiphus bestehende Systemverantwortlichkeiten modelliert. Die Systemverantwortlichkeiten kreuzen die Systemgrenzen, um zu visualisieren, dass diese generell im Verantwortlichkeitsbereich des Systems sind, die genaue Mensch-Maschinen Schnittstelle aber erst auf der Interaktionsebene festgelegt wird.

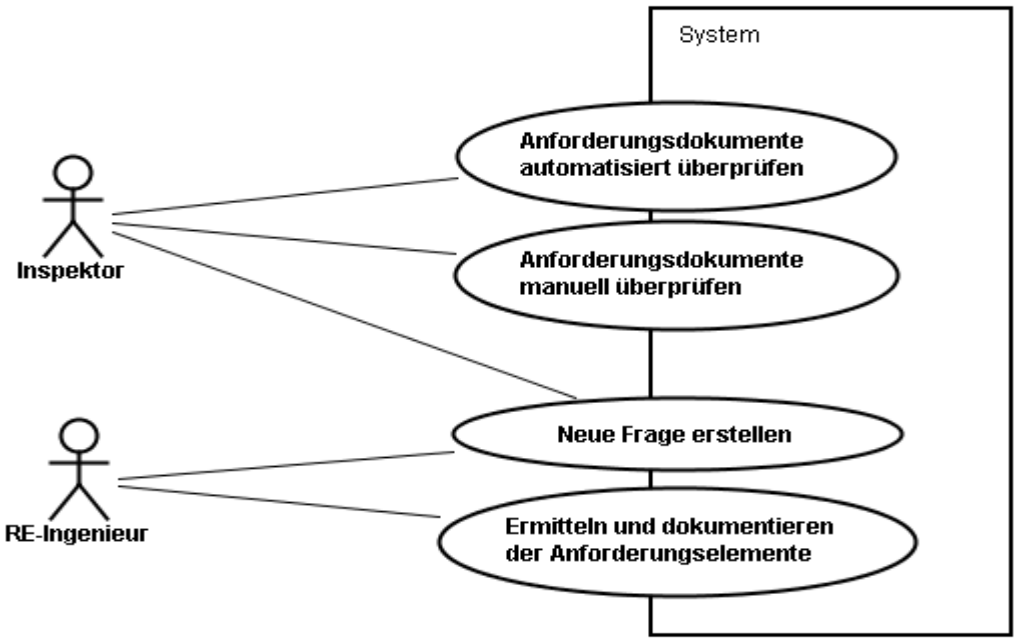


Abbildung 5: Systemverantwortlichkeitenübersicht

4.5.1.4 Domänendaten

Die Beschreibung der Domänendaten wird in Form eines Domänendatendiagramms geschehen. Zur Identifizierung der relevanten Daten müssen die bisher erstellten Produkte der Anforderungsermittlung (Problem-, Aufgabenbeschreibung, Glossareinträge der Softwaredokumentation etc.) betrachtet werden.

U. a. aus der Problembeschreibung entnehmen wir, dass das System der Dokumentation von Anforderungselementen dient, die zu einer bestimmten Anforderungselementekategorie gehören. Zur Unterstützung des Rationale können Fragen (zu den Anforderungselementen) gestellt und diskutiert werden. Aus den Überlegungen zum Soll-Prozess weiß man, dass das System in Zukunft die Anforderungselemente auf syntaktische Probleme hin untersuchen soll und diese berichten muss. Im Glossareintrag der Softwaredokumentation „Syntaktische Anforderungselementeüberprüfung (des Systems)" (Glossareintrag 3) werden Prüfkriterien festgelegt, auf die die Anforderungselemente inspiziert werden.

Die genannten Entitäten *Anforderungselement*, *Anforderungselementekategorie*, *Frage*, *Problem* und *Prüfkriterium* sind also erste Entitäten aus dem Kontext des Inspektionsprozesses, die zu modellieren sind (Abbildung 6). Die Entität *Anforderungselement* muss jedoch noch weiter in einzelne Elemente aufgeteilt werden, da die Granularitätsebene der Domänendaten der Granularitätsebene der im Soll-Prozesses betrachteten Daten und Konzepte entsprechen muss. Während des Soll-Prozesses werden – wie auch in der Abbildung zum Glossareintrag „Syntaktische Anforderungselementeüberprüfung (des Systems)" (Glossareintrag 3) zu sehen ist – die einzelnen Typen der Anforderungselemente für die Inspektion betrachtet. Daher ist das Domänendatenmodell aus Abbildung 6 noch um die von der automatisierten Inspektion zu prüfenden Anforderungselemente zu ergänzen (Abbildung 7).[8] Abbildung 6 und Abbildung 7 ergeben gemeinsam das Domänendatenmodell.

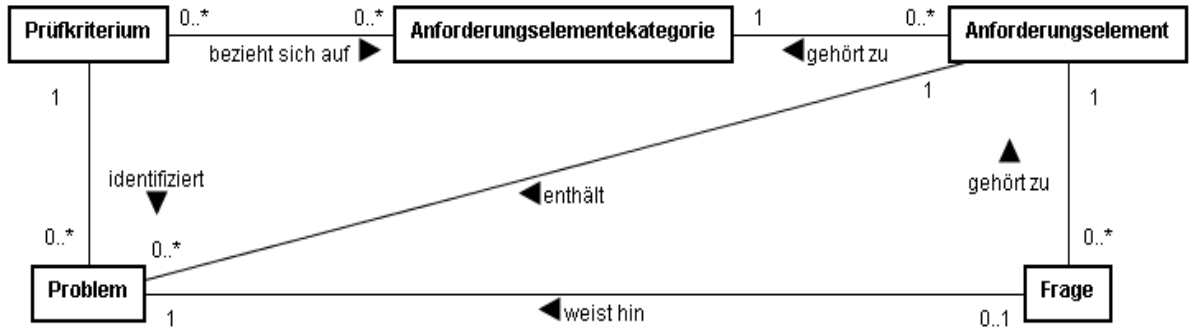


Abbildung 6: Domänendatendiagramm (Teil 1)

[8] Das „Domänendatendiagramm (Teil2)" (Abbildung 7) und das im Glossareintrag „Syntaktische Anforderungselementeüberprüfung (des Systems)" (Glossareintrag 3) eingefügte ER-Diagramm unterscheiden sich in den Multiplizitäten, da ersteres die möglichen Beziehungen im Softwaresystem Sysiphus darstellt, wohingegen letzteres die für eine konsistente und korrekte Softwarespezifikation gewünschten und von der syntaktischen Anforderungselementeüberprüfung zu prüfenden Beziehungen enthält.

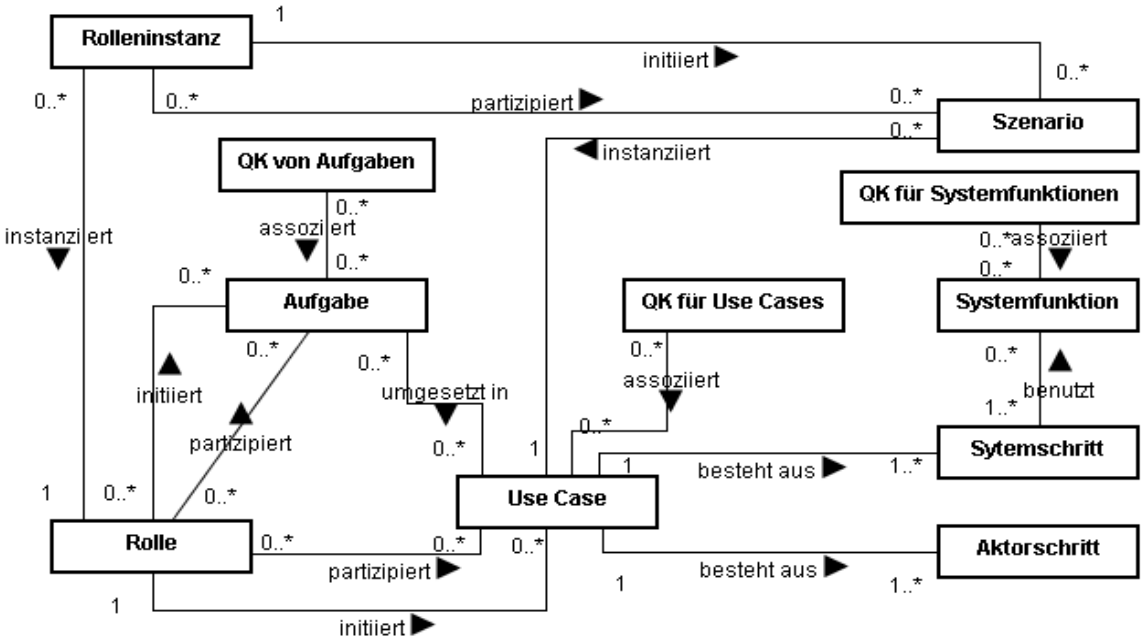


Abbildung 7: Domänendatendiagramm (Teil2)

4.5.2 Nicht-Funktionale Anforderungen

4.5.2.1 Qualitätskriterien von Aufgaben

In Glossareintrag 1 haben wir Probleme mit dem Ist-Prozess angesprochen und in Kapitel 4.5.1.1 angemerkt, dass diese sinnvoll als Nicht-funktionale Anforderungen dokumentiert werden sollten. Da die Beschreibung des Ist-Prozesses die wesentlichen Aspekte der Aufgaben des Geschäftsprozesses enthält (vgl. Kapitel 4.4.1.1), und somit eigentlich ein Teil einer Aufgabe ist, sind die daraus entstehenden Nicht-funktionalen Anforderungen als Qualitätskriterien von Aufgaben zu formulieren. So beeinflussen sie die von der Aufgabe abgeleiteten, verfeinerten Funktionalen Anforderungen. In unserem Fall u. a. die Entscheidungen für die Gestaltung des Soll-Prozesses. Aus den Problemen ergeben sich folgende Qualitätskriterien:

Konzentration des Inspektors auf semantische Fehler

Der Inspektor soll sich während der Inspektion verstärkt auf die Identifizierung von semantischen Fehlern in der Anforderungsspezifikation konzentrieren können.

Qualitätskriterium von Aufgaben 2: Konzentration des Inspektors auf semantische Fehler

Keine Ermüdung des Inspektors

Der Inspektor soll bei seiner Arbeit nicht ermüden, um die Dokumente voll konzentriert zu inspizieren.

Qualitätskriterium von Aufgaben 3: Keine Ermüdung des Inspektors

Außerdem soll der Inspektor weiterhin – so wie im Ist-Prozess beschrieben – selbst bestimmen können, auf welche Weise und in welcher Reihenfolge er seine Inspektion vollzieht. Daher formulieren wir noch eine weitere Nicht-funktionale Anforderung an die Aufgabe:

Feingranulare Inspektion

Der Inspektor soll die Möglichkeit haben, selbst detailliert zu bestimmen, auf welche Weise er bei der Inspektion vorgeht.

Qualitätskriterium von Aufgaben 4: Feingranulare Inspektion

Durch die neuen Nicht-funktionalen Anforderungen müssen also die Probleme und die positiven Aspekte des Ist-Prozesses in die weitere Entwicklungsarbeit einbezogen werden.

4.5.3 Qualitätssicherungsmaßnahmen

4.5.3.1 Testen

4.5.3.1.1 Systemtestplan

Im Systemtestplan werden von uns die Testendekriterien, die Grundlage des Systemtests und die benötigten Testressourcen festgelegt:

Testendekriterien für den Systemtest

- Ein ganzes System gilt als getestet, wenn alle Use Cases und alle Systemfunktionen getestet worden sind.
- Ein Use Case gilt als getestet, wenn alle Ausnahmen, mindestens ein gültiger Durchlauf und alle verwendeten Systemfunktionen getestet worden sind.
- Eine Systemfunktion gilt als getestet, wenn alle Exceptions und mindestens ein gültiger Ablauf getestet worden sind.

Grundlage für Systemtest

- Die Systemtests sollen in unserem Fall auf den Szenarien (vgl. Kapitel 4.6.1.3) basieren, die auf der Interaktionsebene für jeden Use Case erstellt werden. Falls in

den Szenarien nicht alle oben genannten Testendekriterien erfüllt sind, muss die Testspezifikation zusätzlich ergänzt werden, so dass diese erfüllt sind.

Systemtestressourcen

- Da wir in unserem Fall die webbasierte Benutzungsschnittstelle REQuest testen, verwenden wir das Testtool JWebUnit. Dieses kann einen Benutzer eines Webbrowsers simulieren.

Um die Übersichtlichkeit des Systemtests zu gewährleisten, wurde in der bestehenden Testspezifikation bereits angefangen, diesen in einzelne kleinere Tests zu gliedern. Als Gliederungsgrundlage werden die Systemverantwortlichkeiten bzw. die daraus abgeleiteten Use Cases verwandt und für jede(n) ein Teilsystemtest spezifiziert. In unserer Änderungsaufgabe wird daher im weiteren Verlauf der Arbeit für die Systemverantwortlichkeit „Anforderungsdokumente automatisiert überprüfen“ ein Systemtest entwickelt, der eine hohe Testpriorität erhalten und ausführlich getestet werden soll.

4.5.4 Rationale

Die Entscheidung für die Gestaltung des Soll-Prozess basiert auf den Qualitätskriterien von Aufgaben und den Domänenfaktoren. Bei den Qualitätskriterien von Aufgaben spielen allerdings nur die der Aufgabe „Überprüfen der Anforderungen“ eine Rolle. Auf welche Weise die Nicht-funktionalen Anforderungen in die Entscheidung einfließen, wird im Rationale dokumentiert. Rationaletabelle 1 enthält die Bewertungen, wie gut diese Kriterien im Soll-Prozess umgesetzt sind und die darauf basierende Entscheidung für eine Option (fett markiert). In den zur Bewertung herangezogenen Kriterien sind in diesem Fall keine Domänenfaktoren, da diese für die Optionen keine unterschiedliche Bewertung ergeben hätten.

Optionen		**Kriterien**			
		Effizienz bei der Inspektion	Konzentration des Inspektors auf semantische Fehler	Keine Ermüdung des Inspektors	Feingranulare Inspektion
1.	**System übernimmt syntaktische Überprüfung des Softwaredokuments**	**+**	**++**	**+**	**N.A.**
2.	System übernimmt keine Inspektionsaktivität	**--**	**-**	**--**	**++**

Bewertungen für die Berücksichtigung eines Kriteriums in der Gestaltungsalternative (Option):

++	Kriterium sehr gut umgesetzt	-	Kriterium weniger gut umgesetzt
+	Kriterium gut umgesetzt	--	Kriterium schlecht umgesetzt
N.A.	keine Angaben		

Rationaletabelle 1: Wie sollte der Soll-Prozess gestaltet werden?

4.6 Die Interaktionsebene

Wird auf der Aufgabenebene noch gänzlich vom System abstrahiert und vor allem die Rollen und Aufgaben des Geschäftsprozesses zwar im Hinblick auf das System, jedoch im Allgemeinen betrachtet und beschrieben, so wurde auf der Domänenebene bereits das System mit seinen zukünftigen Verantwortlichkeiten und manipulierten Daten fokussiert sowie IT-Realisierungsmöglichkeiten in Betracht gezogen.
Auf der Interaktionsebene (Interaction Level) rückt die konkrete Gestaltung der Schnittstelle zwischen Mensch und Maschine in den Vordergrund. Ziel dieser Ebene ist, die richtige Funktionalität benutzerfreundlich zur Verfügung zu stellen. Um dieses Ziel zu erreichen, werden Entscheidungen getroffen,

- in welchem Kontext der Benutzer welche Funktionen und Daten aufrufen können soll (UI-Struktur),
- in welcher Reihenfolge der Benutzer hierzu mit dem System interagiert bzw. kommuniziert (Use Case-Beschreibungen),
- welche Funktionen das System dafür zur Verfügung stellen muss (Systemfunktionsbeschreibungen),
- und wie die Daten, die der Benutzer und das System austauschen und manipulieren, in Verbindung stehen, sich zusammensetzen und durch Sichten zugänglich gemacht werden (Verfeinertes Datenmodell).

Als tendenzielles Vorgehen auf der Interaktionsebene hat sich bewährt:

1. Die Ableitung der UI-Struktur aus den Systemverantwortlichkeiten.
2. Die Erstellung von Use Cases für die Systemverantwortlichkeiten.
3. Die Erstellung von Systemfunktionsbeschreibungen für komplexere Operationen, d. h. mit komplizierten Berechnungen, Verarbeitung vieler verschiedener Daten oder komplexer Kontrolllogik.
4. Die Verfeinerung des Datenmodells.
5. Die Konsolidierung der UI-Struktur.

4.6.1 Funktionale Anforderungen

4.6.1.1 UI-Struktur

Die Erstellung der UI-Struktur beinhaltet die Gliederung zusammengehöriger Daten und Funktionen in Arbeitsbereiche (Workspaces, Virtual Windows) bzw. Sichten (Views), die dann von einem Arbeitsbereich gebündelt werden. Ziel hierbei ist, dem späteren Benutzer des Systems möglichst wenig Navigationsaufwand zwischen zusammengehörigen Arbeitsschritten mit ihren benötigten Daten und Funktionen zu bereiten und ebenfalls möglichst Arbeitsbereiche/Sichten wieder zu verwenden, um die Gesamtzahl der Arbeitsbereiche/Sichten zu reduzieren und die Übersichtlichkeit zu wahren.

Ein/e Arbeitsbereich/Sicht besteht aus einer Auflistung der benötigten Daten und Funktionen sowie einer Bestimmung des Ziels, das mittels des/der Arbeitsbereiches/Sicht in dem Geschäftsprozess erreicht werden soll.

Das Vorgehen beim Erstellen der UI-Struktur ist in drei Teilschritte gegliedert:

1. Im Grobentwurf wird für jede Systemverantwortlichkeit ein Arbeitsbereich erstellt. Dieser umfasst die Daten, mit denen der Benutzer arbeiten muss und die (in diesem Kontext wichtigen) Funktionen, die er auf den Daten ausführen kann. Des Weiteren sollte das Ziel des Arbeitsbereiches erfasst werden. Eine Aufgliederung der Arbeitsbereiche in Sichten ist zur weiteren Strukturierung der Daten sinnvoll, wenn z. B. die Systemverantwortlichkeit aus mehreren Arbeitsschritten besteht, die unterschiedliche Ziele verfolgen und auf unterschiedlichen Daten bzw. mit unterschiedlichen Funktionen arbeiten.
2. Im Feinentwurf sollte man nun versuchen, ähnliche Arbeitsbereiche/Sichten aus verschiedenen Systemverantwortlichkeiten zusammenzufassen. Dazu muss man evtl. Arbeitsbereiche/Sichten nochmals aufgliedern, um daraus Sichten wieder zu verwenden. Bedacht werden muss jedoch weiterhin, dass dies mit o. g. Ziel, möglichst wenig zwischen den Arbeitsbereichen navigieren zu müssen, abgewogen wird.
3. Eine Konsolidierung der Entscheidungen kann durch eine Systemverantwortlichkeiten/Arbeitsbereiche-Matrix unterstützt werden, die eine Überprüfung der sinnvollen Gewichtung beider Teilziele (Wiederverwendung, minimaler Navigationsaufwand) ermöglicht.

Die UI-Struktur und gegebenenfalls auf ihr basierende Prototypen eignen sich gut zur Diskussion mit dem Kunden und späteren Benutzern darüber, welche Daten und Funktionen wann und wo angeboten werden sollten. Damit beeinflussen sie die spätere Benutzerfreundlichkeit und Akzeptanz des Systems wesentlich.

4.6.1.2 Interaktionsbeschreibung

Die Interaktionsbeschreibung (Use Case) hält den zukünftigen Kommunikationsablauf zwischen Benutzer und System fest. Während einer Interaktion geben die Benutzer dem System (zielgerichtete) Anweisungen, worauf das System dann (System-) Funktionen ausführt, die Berechnungen anstellen und/oder Daten manipulieren.

Im TRAIN-Prozess werden Use Cases auf der Interaktionsebene zur Anforderungsspezifikation[9] als Vorgabe für den Entwurf erstellt (Beschreibungstemplate in Anhang B). Dabei ist es nicht einfach, den in dieser Ebene beabsichtigten Abstraktionsgrad der Beschreibung zu treffen. Der gewünschte Abstraktionsgrad sollte nicht zu allgemein (aufgabenorientiert) und auch nicht zu detailliert (GUI-orientiert) gewählt werden. Vielmehr ist in der Beschreibung auf Eingaben, Veränderungen und Ausgaben von Daten in den Systemfunktionen einzugehen. Nicht gemeint sind damit allerdings konkrete Datenaufschlüsselungen im Sinne aller Attribute eines Datums, sondern die Behandlung der Daten im Sinne einer Entität (vgl. Kapitel 4.4.1.4). Ausnahmen sollten die interne Verarbeitung einer Systemfunktion auf Grund eines fehlenden Datums betreffen (beispielsweise „Eintrag nicht im System vorhanden“) und nicht Ausnahmen auf Wertebene (beispielsweise „Die Eingabe für die Kundennummer liegt außerhalb des Gültigkeitsbereiches“). D. h auf dieser Ebene abstrahiert man im TRAIN-Prozess noch von konkreten Bildschirmdarstellungen, UI-Daten und Navigations- und Supportfunktionen, welche für die spätere Realisierung des GUI notwendig sind.

Das Vorgehen bei der Interaktionsbeschreibung sieht so aus, dass man für jede Systemverantwortlichkeit einen Use Case erarbeitet. Ob man als Zwischenschritt hierfür vorher Szenarien als Vorlage für den Use Case erstellt (vgl. Kapitel 4.6.1.3), muss fallgebunden entschieden werden.

---

[9] In der Literatur findet man häufig auch Use Cases auf anderen Abstraktionsebenen, z. B. zur Beschreibung Funktionaler Anforderungen auf Aufgabenebene, als Mittel zur Anforderungsermittlung vor der konkreten Spezifikation aber auch zu Interaktionsbeschreibung auf Benutzungsschnittstellenebene.

4.6.1.3 Szenarien

Szenarien (Scenarios) beschreiben einen möglichen, spezifischen Interaktionsablauf eines Use Cases (Beschreibungstemplate in Anhang B). Damit stellen sie quasi eine Instanz eines solchen dar. Umgekehrt ist ein Use Case eine abstrakte Beschreibung einer endlichen Menge von Szenarien.

Einerseits kann eine Auswahl an typischen Szenarien als Vorlage für die Use Case-Erstellung dienen, andererseits kann ein Use Case durch nachträgliche Erstellung von Szenarien an einem konkreten Handlungsablauf validiert werden. Außerdem können Szenarien später als Grundlage für den Systemtest im Bereich der Qualitätssicherung (vgl. Kapitel 4.6.3.2.1) sowie für die Erarbeitung von Sequenzdiagrammen während der Entwicklung des Analyseklassendiagramms dienen. Diese werden zur Kontrolle des notwendigen Kommunikationsaufwandes, insbesondere des Nachrichten- und Ereignisfluss zwischen Objekten (vgl. Kapitel 4.8.2.1.3), herangezogen.

4.6.1.4 Systemfunktionen

Systemfunktionen (Services) werden dokumentiert, um festzuhalten, welche Daten als Input in eine Funktion hineingesteckt und welche von dieser verändert werden, wie das Ergebnis berechnet werden soll, welche Ausnahmen auftreten können usw. Zu deren Beschreibung existiert im TRAIN-Prozess ein Template (Anhang B), welches den Requirements Ingenieur bei der Dokumentation der wichtigen Gesichtspunkte unterstützt.

4.6.1.5 Verfeinertes Datenmodell

Das Datenmodell der Domänenebene sollte nochmals überarbeitet werden, indem es in erster Linie durch sog. Sichten ergänzt wird. Unter Sichten versteht man den Blick des Benutzers auf die Daten und Funktionen, die in der Domäne relevant sind.

Im TRAIN-Prozess werden die Arbeitsbereiche/Sichten aus der UI-Struktur als Sicht in das Verfeinerte Datenmodell übernommen. Außer der Aufnahme von Sichten kann es weiterhin sinnvoll sein, Attribute zu Entitäten (oder umgekehrt) zu machen bzw. auch Beziehungen, Entitäten oder Attribute neu hinzuzunehmen oder wegzulassen.

4.6.2 Nicht-funktionale Anforderungen

Auf der Interaktionsebene kommen Qualitätskriterien für Use Cases (Quality Constraints on Use Cases), Qualitätskriterien für Systemfunktionen (Quality Constraints on Services)

und Qualitätskriterien an das GUI (User Interface Constraints) zu den Nicht-funktionalen Anforderungen des Projektes hinzu.

4.6.2.1 Qualitätskriterien für Use Cases und Qualitätskriterien für Systemfunktionen

Qualitätskriterien für Use Cases und Systemfunktionen sind Nicht-funktionale Anforderungen, die meist aus den gleichnamigen Artefakten entstehen oder für diese aus Nicht-funktionalen Anforderungen höherer Abstraktionsebenen verfeinert werden. Qualitätskriterien für Use Cases schränken die Gestaltung der Use Cases und Systemfunktionen ein, Qualitätskriterien für Systemfunktionen die Gestaltung der Systemfunktionen bzw. Gestaltungen im Feinentwurf. Ein Beispiel eines Qualitätskriteriums für Use Cases ist die Forderung nach einer maximalen Zeitspanne für die Ausführung einer Systemfunktion.

4.6.2.2 Qualitätskriterien an das GUI

Qualitätskriterien an das GUI betreffen den Dialog bzw. das Layout und können beispielsweise im Speziellen verlangen, dass eine bestimmte Funktionalität aus jedem Dialog erreichbar oder ausführbar sein muss. Sie werden aber in dieser Arbeit nicht weiter behandelt.

4.6.3 Qualitätssicherungsmaßnahmen

Auf der Interaktionsebene werden neben den Inspektionsaktivitäten Vorbereitungen für den System- und Usabilitytest getroffen.

4.6.3.1 Inspektion

Aspekte, die bei der Inspektion berücksichtigt werden sollten, sind in Anhang C in den Checklisten zur Inspektion festgehalten. Speziell auf Artefakte der Interaktionsebene beziehen sich die „Checkliste - Bereich Use Cases“ und die „Checkliste - Bereich Systemfunktionen“. Des Weiteren muss selbstverständlich auch die Konsistenz zwischen den einzelnen Artefakten (aller Ebenen) gewährleistet sein (z. B. den Beschreibungen und Diagrammen).

4.6.3.2 Testen

4.6.3.2.1 Systemtestspezifikation

Auf Basis des auf Domänenebene entwickelten Systemtestplans, den vorhandenen Nicht-funktionalen Anforderungen und den entwickelten Szenarien wird die

Systemtestspezifikation aufgestellt. In ihr wird festgelegt, wie das System getestet wird, um zu gewährleisten, dass es das gewünschte Soll und die Nicht-funktionalen Anforderungen erfüllt.

4.6.3.2.2 Systemtestimplementierung

Falls möglich, kann auch schon eine erste Rahmenimplementierung des Tests codiert werden. Hier ist natürlich zu bedenken, dass beispielsweise Klassen-, Funktions- und Variablennamen noch nicht (endgültig) festgelegt wurden, sondern erst nach dem Feinentwurf zur Verfügung stehen. Der Ablauf des Tests ist zu diesem Zeitpunkt jedoch bereits in der Testspezifikation vorgegeben.

4.6.3.2.3 Usabilitytestplan und -spezifikation

Aus den Use Case-Beschreibungen, der UI-Struktur und den Szenarien lässt sich der Usabilitytestplan und die -spezifikation entwickeln. Prototypen auf Grundlage der UI-Struktur eignen sich zur Diskussion mit den späteren Benutzern und damit der Kontrolle der Usability und Akzeptanz.

4.6.4 Rationale

Die Begründung der UI-Struktur- und der Use Case-Gestaltungen sollten anhand der auf den höheren Abstraktionsebenen erfassten Nicht-funktionalen Anforderungen stattfinden, die Use Case-Gestaltung weiterhin durch die Qualitätskriterien für Use Cases. Für die Bewertung der Systemfunktionen kommen dann nochmals die Qualitätskriterien für Systemfunktionen hinzu. Weiterhin sollten bei Änderungen die Neuerstellungen, Ergänzungen bzw. das Entfernen von Artefakten auf der Interaktionsebene nachvollziehbar sein.

Ebenso dürfen die Begründungen für die Qualitätssicherungsprodukte dieser Ebene nicht fehlen.

4.7 Die Interaktionsebene im Beispiel

4.7.1 Funktionale Anforderungen

4.7.1.1 UI-Struktur

In Abbildung 8 ist ein Ausschnitt der UI-Struktur des Sysiphus-Projektes dargestellt. Der Arbeitsbereich „Manuelle Inspektion“ mit seinen Sichten „Fragenauswahlansicht“, „Fragedetailansicht“, „Anforderungselementeauswahlansicht“ und „Anforderungselementedetailansicht“ ist im bestehenden Projekt bereits umgesetzt worden. Ausgehend von der Systemverantwortlichkeitenübersicht (Abbildung 5) entwickeln wir nun die UI-Struktur für die neue Systemverantwortlichkeit „Anforderungsdokumente automatisiert überprüfen“ weiter. Diese wird zu dem Arbeitsbereich „Automatisierte Inspektion“, der folgendes beinhaltet:

> *Will der Benutzer die automatisierte Inspektion durchführen, so sollte er bestimmen können, welche Prüfkriterien der einzelnen Elemente er untersuchen möchte. Nachdem er diese Angaben getätigt hat, muss er die Überprüfung starten und danach die Ergebnisse betrachten können.*

Die zusammengehörigen Daten und Funktionen, die der Benutzer benötigt, lassen sich in diesem Fall sinnvoll entlang der beiden Arbeitsschritte *Prüfkriterienwahl* und *Ergebnisbetrachtung* gliedern. Entsprechend werden wir auch zwei Sichten entwickeln: die Überprüfungsauswahlansicht und die Fragenauswahlansicht.[10] Das Ziel der Überprüfungsauswahlansicht ist die Wahl der Prüfkriterien für die automatisierte Überprüfung der Anforderungsdokumente und deren Start. Hierzu werden als Daten die zur Wahl stehenden Prüfkriterien und die zu prüfenden Anforderungselemente (i. a. alle des Projektes, für die Prüfkriterien existieren) benötigt. Als Funktion muss der Start der automatisierten Anforderungselementeüberprüfung möglich sein.

Wie in Kapitel 4.6.1.1 erwähnt, wird im Feinentwurf der UI-Struktur angestrebt, dass Arbeitsbereiche/Sichten möglichst wieder verwendet werden können. In unserem Beispiel wäre es denkbar, die Fragenauswahlansicht auch für die Betrachtung der Ergebnisse der

[10] Der Name Fragenauswahlansicht ergibt sich aus der Entscheidung, dass Fragen als „Report“ für die von der automatisierten Inspektion gefundenen Probleme verwendet werden und daher die Fragenauswahlansicht, die ja bereits in dem bestehenden Sysiphus-Projekt vorhanden ist, für die Betrachtung der Ergebnisse der automatisierten Anforderungselementeüberprüfung wieder verwendet werden kann (vgl. Kapitel 4.7.4).

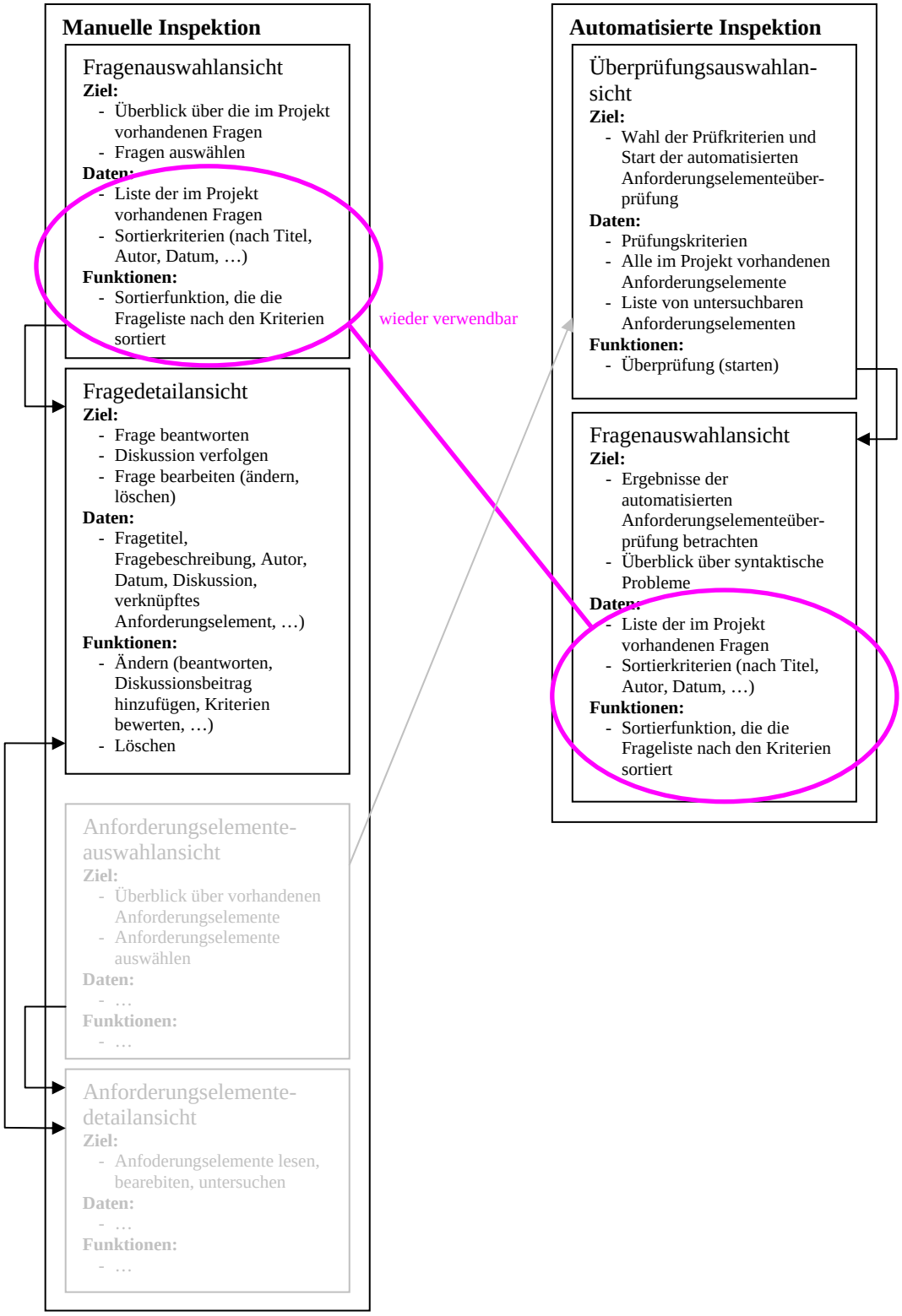


Abbildung 8: UI-Struktur

automatisierten Anforderungselementeüberprüfung wieder zu verwenden, wenn die Fragen als „Report“ für die gefundenen Probleme benutzt werden. Dieses Kriterium führt u. a. auch zu der Entscheidung, dass die Dokumentation der Probleme mit Hilfe der Fragen geschieht. In Kapitel 4.7.4 wird die Entscheidungsfindung in Rationaletabelle 3 dokumentiert.

Die getroffenen Entscheidungen über Wiederverwendung und Navigationsaufwand der Arbeitsbereiche/Sichten können durch eine Matrix veranschaulicht und überprüft werden. Hierzu trägt man in die eine Richtung die Systemverantwortlichkeiten (nehmen wir an vertikal) und in die andere die Arbeitsbereiche/Sichten (horizontal) ab. Somit weist eine starke vertikale Ausprägung der Matrix auf einen hohen Wiederverwendungsanteil der Arbeitsbereiche/Sichten in den verschiedenen Systemverantwortlichkeiten hin, wohingegen eine stark horizontale Ausprägung der Einträge innerhalb einer Systemverantwortlichkeit auf einen hohen Navigationsaufwand schließen lässt.

Betrachten wir die Matrix in unserem Beispiel (Tabelle 1), so ist zu erkennen, dass die Systemverantwortlichkeit „Anforderungsdokumente manuell überprüfen“ viel Navigation beansprucht. Dies lässt sich jedoch mit der steigenden Abgeschlossenheit und Übersichtlichkeit rechtfertigen, die durch die strukturierte Darstellung sinnverwandter Daten in den einzelnen Arbeitsbereichen entsteht. Dagegen benötigt die Systemverantwortlichkeit „Anforderungsdokumente automatisiert überprüfen“ wenig Navigationsaufwand, und es ist sogar möglich, die Fragenauswahlansicht in beiden Systemverantwortlichkeiten zu verwenden.

Arbeitsbereiche → ↓ Systemverantwortlichkeiten	Fragen-auswahl-ansicht	Fragedetail-ansicht	Anforderungs-elementeaus-wahlansicht	Anforderungs-elementedetail ansicht	Überprüfungs-auswahl-ansicht
Anforderungsdokumente manuell überprüfen	X	X	X	X	
Anforderungsdokumente automatisiert überprüfen	X				X

X: Markierung der Zugehörigkeit eines Arbeitsbereiches zu einer Systemverantwortlichkeit

Tabelle 1: Matrix zur Konsolidierung der UI-Struktur

4.7.1.2 Interaktionsbeschreibung

Der Use Case „Anforderungselemente automatisiert überprüfen“ entsteht als Interaktionsbeschreibung aus der Aufgabe „Überprüfen der Anforderungen“ und aus der Systemverantwortlichkeit „Anforderungsdokumente automatisiert überprüfen“, die von ihr abgeleitet wurde.

Der initiierende Aktor des Use Cases „Anforderungselemente automatisiert überprüfen“ ist der Inspektor und die unterstützenden Aktoren sind der Requirements Ingenieur und der

Rationale Maintainer. Ziel der Aktoren, die diesen Use Case durchführen, sind Hinweise über alle potentiellen syntaktischen Fehler (Probleme) in den Anforderungselementen zu erhalten.

Bevor nun die Beschreibung des Ablaufs (Flow of Event) und die während diesem möglicherweise auftretenden Ausnahmen (Exceptions) entwickelt werden, muss die im Flow of Events zu beschreibende Interaktion abgegrenzt werden. Hierzu gibt man den Zustand des Systems vor Eintritt in den Use Case an, der erfüllt seine muss, um dem Aktor seine Zielerreichung durch Durchführung des Use Cases zu ermöglichen (Vorbedingungen). In unserem Fall wären u. a. folgende Vorbedingungen zu erfüllen:

- eine Baseline, d. h. ein für die Inspektion geeigneter Zustand des Anforderungsspezifikationsdokumentes besteht,
- der Aktor muss eingeloggt sein und die notwendigen Rechte für die Durchführung der Inspektion besitzen,
- der Aktor muss sich in der Anforderungselementeauswahlansicht befinden, aus der er zur Überprüfungsauswahlansicht gelangen kann. In dieser kann er dann die syntaktische Überprüfungskriterien genauer bestimmen und die Inspektion starten.

Wird die Inspektion gestartet, so führt das System die Überprüfung aus. Die Überprüfung durch das System stellt eine Systemfunktion dar, deren Ablauf später gesondert festgehalten wird.

Der Zustand des Systems nach Beendigung des Use Case (Nachbedingung) ist davon gekennzeichnet,

- dass das System das Anforderungsspezifikationsdokument untersucht und auf jedes gefundene Problem durch eine Frage aufmerksam gemacht hat und
- das System in die Anforderungselementeauswahlansicht und die Fragenauswahlansicht zurückgekehrt ist.

Der Flow of Event wird also in „Rahmenbedingungen“ eingeschlossen, um ihn abzugrenzen.

Für die Realisierung des Flow of Events gibt es in unserem Beispiel mehrere Möglichkeiten:

1. Der Aktor startet die Inspektion, und das System führt daraufhin eine Inspektion nach festgelegten, vom Aktor nicht zu beeinflussenden Prüfkriterien aus. Anschließend gibt das System eine kurze Zusammenfassung der Probleme aus.
2. Analog zu 1, das System gibt jedoch keine Zusammenfassung der Probleme aus.

<table>
<tr><td colspan="3">Interaktionsbeschreibung (Use Case)</td></tr>
<tr><td>Name:</td><td colspan="2">ANFORDERUNGSELEMENTE AUTOMATISIERT ÜBERPRÜFEN</td></tr>
<tr><td>Verantwortliche Rolle (Initiating Actor):</td><td colspan="2">Inspektor</td></tr>
<tr><td>Beteiligte Rollen (Participating Actors):</td><td colspan="2">- Requirements Ingenieur
- Rational Maintainer</td></tr>
<tr><td>Ziel (Goal):</td><td colspan="2">Die Anforderungselemente des Softwaredokumentes sollen auf Fehler überprüft werden.</td></tr>
<tr><td>Vorbedingung (Preconditions):</td><td colspan="2">- Ein Baseline des Anforderungsspezifikationsdokumentes ist vorhanden
- Aktor ist in das System eingeloggt
- Aktor ist mit den entsprechenden Rechten ausgestattet
- Projekt ist geöffnet
- Anforderungselementeauswahlansicht ist geöffnet
Folgende Aktionen stehen zur Verfügung:
a) Automatisierte Inspektion durchführen</td></tr>
<tr><td rowspan="4">Beschreibung (Flow Of Events):</td><td>Aktor</td><td>System</td></tr>
<tr><td>1. Aktor wählt, dass er einen automatisierte Inspektion durchführen will</td><td></td></tr>
<tr><td></td><td>2. System zeigt ihm die Überprüfungsauswahlansicht
System ermöglicht:
2a) einzelne Prüfkriterien auswählen
2b) Automatisierte Inspektion starten</td></tr>
<tr><td>3.
- Falls Aktor ‚2a' wählt, weiter mit 2a oder 2b
- Falls Aktor ‚2b' wählt, weiter mit 4</td><td></td></tr>
<tr><td></td><td></td><td>4. System führt Systemfunktion „Automatisierte Prüfung der Anforderungselemente durchführen“ mit den gewählten Prüfkriterien aus</td></tr>
<tr><td>Ausnahmefälle (Exceptions):</td><td colspan="2">keine</td></tr>
<tr><td>Nachbedingungen (Postconditions):</td><td colspan="2">- System hat Anforderungsspezifikationsdokument untersucht und auf jedes gefundene Problem durch eine Frage aufmerksam gemacht
- System kehrt zur Anforderungselementeauswahlansicht zurück und zeigt die Fragenauswahlansicht</td></tr>
<tr><td>Regeln (Rules):</td><td colspan="2">keine</td></tr>
<tr><td>Qualitätsanforderung (Quality Constrains):</td><td colspan="2">- Geringe Anzahl von Sichten
- Geringe Navigation zwischen Sichten
- Überschaubarkeit der Sichten
- Einfache/unterstützte Zusammenstellung der Prüfkriterien
- Freiraum des Inspektors
- Übersichtlichkeit der Ergebnisse der automatisierten Inspektion

Vererbte:
- Qualitätskriterien von Aufgaben der Aufgabe „Überprüfen der Anforderungen“
- Domänenkriterien</td></tr>
<tr><td>Benutzte Funktionen (Used Services):</td><td colspan="2">Automatisierte Prüfung der Anforderungselemente durchführen</td></tr>
</table>

Interaktionsbeschreibung 1: Anforderungselemente automatisiert überprüfen

3. Der Aktor wählt aus vorgegebenen, fest definierten Checklisten mit festgelegten, vom Aktor nicht zu beeinflussenden Prüfkriterien aus. Das System führt die automatisierte Inspektion anhand der ausgewählten Checklisten durch. Anschließend gibt das System eine Zusammenfassung der Inspektion aus.
4. Analog zu 3, das System gibt jedoch keine Zusammenfassung der Probleme aus.
5. Der Aktor stellt sich eine Checkliste mit gewünschten Prüfkriterien aus einer Liste von möglichen Kriterien selbständig zusammen. Das System führt die automatisierte Inspektion anhand der zusammengestellten Checkliste aus. Abschließend gibt das System eine Zusammenfassung der Inspektion aus.
6. Analog zu 5, das System gibt jedoch keine Zusammenfassung der Probleme aus.
7. Der Aktor wählt aus drei vorgegebenen fest definierten Checklisten mit festgelegten, vom Aktor nicht zu beeinflussenden Prüfkriterien eine aus, kann diese jedoch zusätzlich noch nach seinen Wünschen anhand einer Liste von Prüfkriterien modifizieren. Das System führt anschließend die automatisierte Inspektion anhand der gewählten Checkliste durch. Abschließend gibt das System noch eine Zusammenfassung der Inspektion aus.
8. Analog zu 7, das System gibt jedoch keine Zusammenfassung der Probleme aus.

Nach Abwägung der Gesamtbewertung der obigen Alternativen (Kapitel 4.7.4, Rationaletabelle 4) gegen die beschränkenden Nicht-funktionalen Anforderungen, muss die Entscheidung für Vorschlag 8 fallen, da er dem Inspektor am meisten Freiraum bei seiner Kriterienwahl lässt. In unserem Beispiel entwickeln wir aber Vorschlag 6 weiter.[11] Den vollständig dokumentierten Use Case findet man in Interaktionsbeschreibung 1.

4.7.1.3 Systemfunktionen

Im Use Case „Anforderungselemente automatisiert überprüfen“ benötigen wir eine Systemfunktion.

Diese ist dafür zuständig, jedes Anforderungselement auf Probleme hin zu untersuchen und diese anschließend zu berichten. Als Eingabe erhält sie bestimmte Prüfkriterien, die sog. Checkliste. Daraufhin muss sie sich die Anforderungselemente aus dem Projekt holen und diese überprüfen. Die Ausgabe der Funktion sind die berichteten Probleme in Form von Fragen.

[11] Diese Entscheidung bestand bereits bei Beginn der Arbeit im Sysiphus-Projekt und wurde nicht mehr revidiert (vgl. Kapitel 3.1).

Nun gibt es mehrere Möglichkeiten, die gewählten Anforderungselemente zusammenzustellen:

1. Das System holt sich alle Anforderungselemente aus dem Projekt, gleicht anschließend jedes Element aus der Liste mit der vorhandenen Checkliste ab und prüft es auf die für das Element spezifizierten Kriterien.
2. Das System geht die Prüfkriterien der Checkliste Schritt für Schritt durch, holt sich für jedes Kriterium die zugehörigen Anforderungselemente und führt die automatisierte Inspektion für jeden einzelnen Checklistenpunkt einzeln durch (d.h. das System holt sich für jedes Prüfkriterium die Anforderungselemente die es braucht jedes Mal neu).
3. Analog zu Vorschlag 2. Jedoch werden die Anforderungselemente, die einmal aus dem Projekt geholt werden, für ein evtl. weiteres Prüfkriterium (indem sie benötigt werden) für die laufende Inspektion zwischengespeichert.
4. Das System überprüft die gesamte Checkliste, stellt sich eine Liste der benötigten Anforderungselementekategorien zusammen und holt anschließend nur die Elemente aus diesen Kategorien.

Nach Abwägung der Alternativen gegen die beschränkenden Nicht-funktionalen Anforderungen (Kapitel 4.7.4, Rationaletabelle 5) entscheiden wir uns für Vorschlag 3.

Die Gestaltungsentscheidung, die gefundenen Probleme mittels Fragen zu berichten (vgl. Kapitel 4.7.1.1 und Kapitel 4.7.4, Rationaletabelle 3), führt zu der Diskussion, wie man Probleme behandelt, die vom System wiederholt gefunden werden:

1. Eine zweite Frage wird erzeugt, die nochmals auf das Problem hinweist.
2. Die vorhandene Frage wird gelöscht und eine neue Frage erzeugt.
3. Es wird überprüft, ob die vorhandene Frage schon geschlossen ist. Wenn ja, diese Frage wieder öffnen und keine neue Frage erzeugen. Falls Frage noch geöffnet, keine weiter Aktion, da auf das Problem schon hingewiesen wird.

Die Entscheidung fällt für Vorschlag 3 (Kapitel 4.7.4, Rationaletabelle 6).

Fasst man alle Gestaltungsentscheidungen zusammen, so entsteht Systemfunktionsbeschreibung 1.

Systemfunktionsbeschreibung (Service)	
Name:	**AUTOMATISIERTE PRÜFUNG DER ANFORDERUNGSELEMENTE DURCHFÜHREN**
Beschreibung (Description):	1. System liest die gewählten Prüfkriterien (Checkliste) ein. [Ausnahme: Checkliste enthält keine (gültigen) Überprüfungen] 2. System holt sich Projekt dessen Elemente überprüft werden sollen 3. System geht die gewählten Kriterien sequenziell durch und a) holt sich die benötigten Anforderungselemente. Dabei überprüft es zuerst, ob die Anforderungselemente bereits zwischengespeichert sind aa) ist dies der Fall: Verwendung der zwischengespeicherten Elemente ab) ist dies nicht der Fall: Bezug der benötigten Elemente aus dem Projekt b) überprüft die Anforderungselemente c) erzeugt für jedes Anforderungselement, bei dem ein Problem entdeckt worden ist, eine Frage. [Ausnahme: eine Frage, die auf dieses Problem hinweist, ist schon im Projekt vorhanden]
Eingangsdaten (Inputs):	Keine
Ausgangsdaten (Outputs):	Keine
Ausnahmefälle (Exceptions):	[Ausnahme: eine Frage, die auf dieses Problem hinweist, ist schon im Projekt vorhanden]: System überprüft, ob die Frage bereits geschlossen ist. Wenn ja, so öffnet das System die Frage wieder. Anschließend fährt das System mit der automatisierten Inspektion fort. [Ausnahme: Checkliste enthält keine (gültigen) Überprüfungen]: System bricht die automatisierte Inspektion ab und gibt eine Fehlermeldung "Keine oder ungültige syntaktischen Überprüfungen ausgewählt"
Regeln (Rules):	Für die Bezeichnung der Probleme in den Fragen wird folgendes Schema für die Fragetitel festgelegt: - Einleitender ‚Tag': „SYSTEM:" zur schnellen Identifizierung der Systeminspektionsfragen - Kurzbeschreibung des Problemtyps, z. B.: „Why does the following User Task has no Initiating Actor?" - Hinweis auf das Element, bei dem das Problem gefunden wurde, z. B.: „User Task: *NAME*" Eine Frage zu dem User Task mit dem Namen „Aufgabe_01", welche keinen initiierenden Aktor besitzt hätte demnach folgenden Titel: „*SYSTEM: Why does the following User Task has no Initiating Actor? User Task: Aufgabe_01"*
Vorbedingungen (Preconditions):	Keine
Nachbedingungen (Postconditions):	Für jedes gefundene syntaktische Problem ist eine Frage dem Projekt hinzugefügt worden, die auf das syntaktische Problem hinweist
Qualitätsanforderung (Quality Constrains):	- Keine redundanten Fragen durch die automatisierte Inspektion - Kurze Dauer für automatisierte Inspektion **Vererbte**: - Domänenkriterien - Globale Nicht-funktionale Anforderungen
Use Cases:	Anforderungselemente automatisiert überprüfen

Systemfunktionsbeschreibung 1: Automatisierte Prüfung der Anforderungselemente durchführen

4.7.1.4 Verfeinertes Datenmodell

Im diesem Schritt wird das Domänendatendiagramm mittels Sichten zum Verfeinerten Datenmodell ergänzt. Jede Sicht aus der UI-Struktur wird mit Beziehung zu den über sie sichtbar werdenden Daten in das Domänendatenmodell einbezogen. In unserem Fall entspricht die Granularitätsebene der Sichten unserem gröberen Domänendatenmodell (Abbildung 6), weshalb wir dieses zu Abbildung 9 ergänzen.

Des Weiteren ist in der Systemfunktionsbeschreibung eine neue Entität relevant geworden, die nachträglich in das Domänendatendiagramm (Abbildung 6) eingearbeitet werden muss[12] und daher auch in unserem Verfeinerten Datenmodell auftaucht. Die neue Entität ist die Entität *Checkliste*, die als Bestandteile die Prüfkriterien beinhaltet.

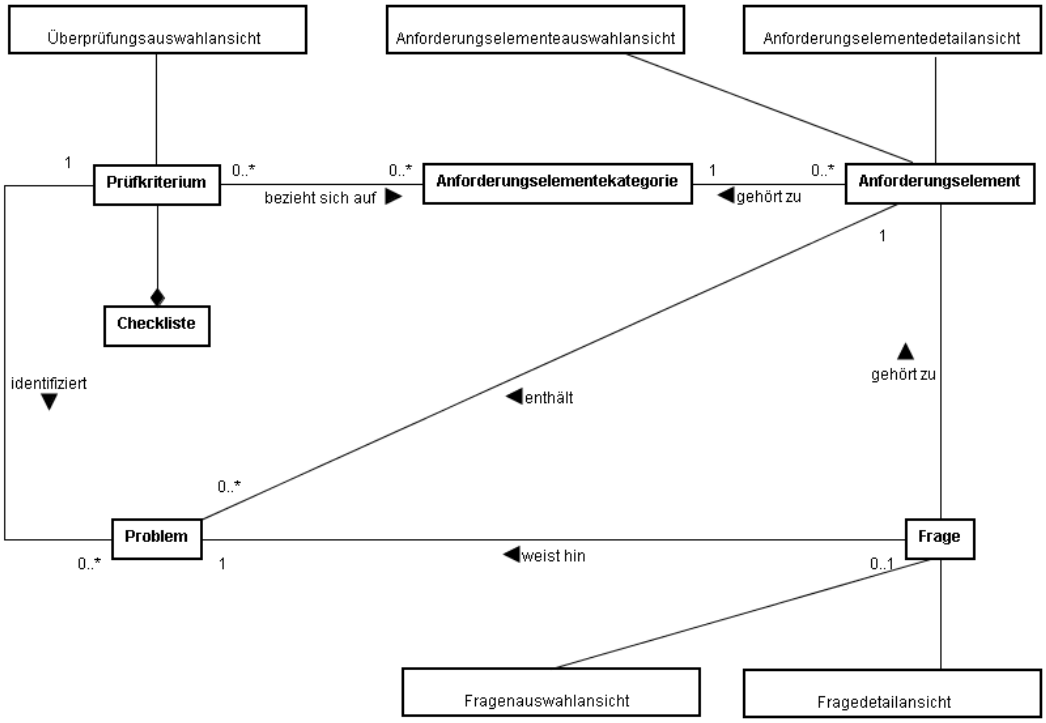


Abbildung 9: Verfeinertes Datenmodell

4.7.1.5 Szenarien

Ein mögliches Szenario unseres Beispiels ist in Szenario 1 „Jansen Automatisierte Inspektion“ ausgeführt. Dieses haben wir bereits auf die Anforderungen für einen Systemtest (Testendekriterien, vgl. Kapitel 4.5.3.1.1) ausgerichtet, um uns später die Systemtestspezifikation zu erleichtern.

---

[12] Wegen der Nachvollziehbarkeit werden wir in dieser Arbeit das Domänendatenmodell in Abbildung 6 jedoch nicht ändern.

Szenario (Scenario)	
Name:	**JANSEN AUTOMATISIERTE INSPEKTION**
Verantwortlicher Use Case (Initiated Use Case):	Anforderungselemente automatisiert überprüfen
Verantwortliche Rolleninstanz (Initiating Actor Instance):	Jansen
Beschreibung (Flow Of Events):	Das REQuest System, mit dem Herr Jansen arbeitet, wurde um die Funktionalität „automatisierte Inspektion“ erweitert. Um die neuen Möglichkeiten kennen zu lernen, simuliert Herr Jansen folgendes Vorgehen: Im aktuellen Projekt liegt eine Baseline der Anforderungsdokumente vor, so dass eine Inspektion der Dokumente anberaumt wurde. Herr Jansen hat die Aufgabe, neben einer Inspektion via perspektivenbasierter Lesetechnik zum Auffinden von semantischen Fehlern, die Vorbereitungen für die syntaktische Problemdiskussion im Inspektionsmeeting zu treffen. Hierfür verwendet er die vom System unterstützte automatisierte Inspektion. Er loggt sich mit seinem Nutzername („Jansen“) und seinem Passwort („jnsn“) ein und wählt danach das zu bearbeitende Projekt („Systemtestprojekt Inspektion“). Dieses beinhaltet bereits einen Aktor („Actor_01“), der keine Beschreibung besitzt. Herr Jansen ist mit den Rechten eines Inspektors ausgestattet und navigiert in die Anforderungselementeauswahlansicht. Hier wählt er die automatisierte Inspektion und das System zeigt ihm die Überprüfungsauswahlansicht. Herr Jansen erhält ein Anruf und startet daher die Inspektion, ohne sich auf seine Handlung zu konzentrieren. Da er vergessen hat anzugeben, welche Anforderungselementekategorien er überprüfen möchte, weist ihn das System auf die fehlenden Angaben hin. Weiterhin am Telefon navigiert er erneut zu der Überprüfungsauswahlansicht und wählt für alle Anforderungselementekategorien eine Überprüfung der Textfelder. Nachdem er das Telefonat abgeschlossen hat, navigiert er zur Fragenauswahlansicht, wählt die vom System erzeugte Frage zum fehlenden Textfeld bei dem Aktor („Actor_01“) und schließt sie mit einer Begründung („Ist schon recht so!“). Wieder in der Fragenauswahlansicht merkt er, dass keine Beziehungsüberprüfung stattgefunden hat. Somit wechselt er in die Überprüfungsauswahlansicht und wählt erneut eine Textfeld aber auch die Beziehungsüberprüfung für alle Elemente. Nachdem die automatisierte Inspektion beendet ist, navigiert er zur Fragenauswahlansicht und schaut, ob Fehler gefunden wurden. Hiernach loggt sich Herr Jansen aus dem System aus.

Szenario 1: Jansen Automatisierte Inspektion

4.7.2 Nicht-funktionale Anforderungen

4.7.2.1 Globale Nicht-funktionale Anforderungen

Für die Gestaltung der UI-Struktur wurden folgende Ziele formuliert:

- wenige Sichten u. a. durch Wiederverwendung,
- wenig Navigationsaufwand für den Benutzer und
- überschaubare Sichten.

Da diese die Gestaltungsentscheidungen (bei der Entwicklung der UI-Struktur und den Use Cases) einschränken, wurden aus ihnen bereits in der bestehenden Sysiphus-Dokumentation Nicht-funktionale Anforderungen formuliert (Globale Nicht-funktionale Anforderung 1, Globale Nicht-funktionale Anforderung 2, Globale Nicht-funktionale

Anforderung 3). Als Globale Nicht-funktionale Anforderungen schränken sie alle Gestaltungsentscheidungen ab der Interaktionsebene ein, die mit Sichten zu tun haben.

Geringe Anzahl von Sichten

Die Gesamtzahl der im System vorhandenen Sichten soll so gering wie möglich gehalten werden.

Globale Nicht-funktionale Anforderung 1: Geringe Anzahl von Sichten

Geringe Navigation zwischen Sichten

Es soll (innerhalb eines Use Cases) nur ein geringer Navigationsaufwand zwischen Sichten für den Aktor notwendig sein.

Globale Nicht-funktionale Anforderung 2: Geringe Navigation zwischen Sichten

Überschaubarkeit der Sichten

Die Sichten sollen die zusammengehörigen Daten und Funktionen überschaubar darstellen. D. h eine Sicht soll nicht zu viele und möglichst logisch zusammengehörige Informationen beinhalten..

Globale Nicht-funktionale Anforderung 3: Überschaubarkeit der Sichten

4.7.2.2 Qualitätskriterien für Use Cases

Des Weiteren entstehen mehrere Qualitätskriterien für Use Cases durch das Verfeinern von Nicht-funktionalen Anforderungen höherer Ebenen, die die Gestaltungsentscheidung auf der Interaktionsebene einschränken. Ein Qualitätskriterium für Use Cases entsteht aus einem bestehenden Domänenfaktor „Wissenstransfer zwischen den Projektmitarbeitern“, der fordert, dass der Wissenstransfer/Diskussionen zwischen den Mitarbeitern eines Projektes dauerhaft ermöglicht werden muss/müssen.

Gefundene Probleme diskutierbar (konkretisiert Domänenfaktor „Wissenstransfer zwischen den Projektmitarbeitern“)

Die vom System gefundenen Probleme einer Inspektion sollen zwischen den Inspektoren diskutierbar sein, und das gewonnene Wissen soll auch den anderen Projektmitarbeiten zur Verfügung stehen.

Qualitätskriterium für Use Cases 1: Gefundene Probleme diskutierbar

Die anderen verfeinern die von uns aufgestellten Qualitätskriterien von Aufgaben für die Aufgabe „Überprüfen der Anforderungen“:

Freiraum des Inspektors (konkretisiert Qualitätskriterium von Aufgaben 4: Feingranulare Inspektion)

Beim Zusammenstellen der zu überprüfenden Kriterien (Prüfkriterien) bei der automatisierten Inspektion muss der Inspektor frei Wahl haben und möglichst flexibel sein.

Qualitätskriterium für Use Cases 2: Freiraum des Inspektors

Wiederholbare Ausführung der Inspektion (konkretisiert Qualitätskriterium von Aufgaben 4: Feingranulare Inspektion)

Die automatisierte Inspektion muss beliebig oft hintereinander ausgeführt werden können.

Qualitätskriterium für Use Cases 3: Wiederholbare Ausführung der Inspektion

Einfache/unterstützte Zusammenstellung der Prüfkriterien (konkretisiert Qualitätskriterium von Aufgaben 1: Effizienz bei der Inspektion)

Das System soll den Inspektor beim Zusammenstellen der Prüfkriterien für die automatisierte Inspektion gut unterstützen und das Zusammenstellen so einfach wie möglich gestalten (z.B. durch Standardchecklisten)! So sollte eine (vollständige) Inspektion bereits mit wenigen Aktionen des Aktors angestoßen werden können.

Qualitätskriterium für Use Cases 4: Einfache/unterstützte Zusammenstellung der Prüfkriterien

Übersichtlichkeit der Ergebnisse der automatisierten Inspektion (konkretisiert Qualitätskriterium von Aufgaben 1: Effizienz bei der Inspektion)

Über die Ergebnisse einer automatisierten Inspektion muss der Inspektor schnell (für 25 berichtete Probleme innerhalb von 2 min) einen Überblick erhalten.

Qualitätskriterium für Use Cases 5 Übersichtlichkeit der Ergebnisse der automatisierten Inspektion

Kurze Dauer für automatisierte Inspektion (konkretisiert Qualitätskriterium von Aufgaben 1: Effizienz bei der Inspektion)

Die automatisierte Inspektion für 100 Anforderungselemente darf nicht länger als 5 sec dauern.

Qualitätskriterium für Use Cases 6: Kurze Dauer für automatisierte Inspektion

Das Qualitätskriterium für Use Cases 7 entsteht aus der Gestaltungsentscheidung zur Dokumentation der Probleme als Fragen, der Forderung nach einer mehrmals wiederholbaren Inspektion (Qualitätskriterium für Use Cases 3) und durch Beachtung des Qualitätskriteriums „Effizienz bei der Inspektion“ (Qualitätskriterium von Aufgaben 1).

Keine redundanten Fragen durch die automatisierte Inspektion

Keine zwei (oder mehr) vom System automatisch erzeugten Fragen dürfen auf ein und dasselbe gefundene Problem hinweisen.

Qualitätskriterium für Use Cases 7: Keine redundanten Fragen durch die automatisierte Inspektion

4.7.3 Qualitätssicherungsmaßnahmen

4.7.3.1 Testen

In diesem Beispiel werden wir uns auf die Systemtestspezifikation konzentrieren und die weiteren Qualitätssicherungsmaßnahmen dieser Ebene, die in Kapitel 4.6.3.2.1 erwähnt wurden, außen vor lassen.

4.7.3.1.1 Systemtestspezifikation

Im Systemtestplan wurden die Testendekriterien und die Grundlagen für den Systemtest festgelegt. Die Untersuchung, ob das Szenario alle Testendekriterien für einen Systemtest erfüllt (vgl. Kapitel 4.5.3.1.1), verläuft positiv: für den betroffenen Use Case „Anforderungselemente automatisiert überprüfen“ und dessen Systemfunktion „Automatisierte Prüfung der Anforderungselemente durchführen“ werden alle Ausnahmen überprüft, und sie werden einmal gültig durchlaufen. Somit kann das Szenario als Basis für den Systemtestteil der Inspektion dienen. Der Systemtest wird wie in Systemtestspezifikation 1 spezifiziert.

<table>
<tr><td colspan="4">Systemtestspezifikation</td></tr>
<tr><td>Name:</td><td colspan="3">SYSTEMTEST SZENARIO ‚JANSEN AUTOMATISIERTE INSPEKTION’</td></tr>
<tr><td>Typ (Type)</td><td colspan="3">Systemtestfall (System Test Case)</td></tr>
<tr><td>Ziel (Goal)</td><td colspan="3">Systemtest für die Inspektionsfunktionalität (Szenario “Jansen Automatisierte Inspektion”)</td></tr>
<tr><td>Vorbedingungen (Preconditions)</td><td colspan="3">- Elementstore ist gestartet
- Ein Projekt mit Namen „Systemtestprojekt Inspektion“ ist angelegt
- Das Projekt enthält als Anforderungselemente einen Aktor mit Name „Actor_01“ ohne Beschreibung und ohne Beziehungsangabe
- Ein Benutzer namens Jansen mit Passwort "jnsn" ist angelegt und die Rechte sind entsprechend (Inspektorenrechte) gesetzt
- Tomcat ist gestartet und das REQuest WAR - File ist eingebunden
- System zeigt Login Anmeldefenster</td></tr>
<tr><td rowspan="8">Testschritte (Steps)</td><td>Input</td><td>Expected Output</td><td>Exceptions</td></tr>
<tr><td>Login "Jansen" und richtigen Passwort "jnsn" in die gezeigten Felder eintragen -> Eingabe bestätigen</td><td>Einloggen war erfolgreich --> System zeigt Projektauswahlansicht</td><td>Einloggen war nicht erfolgreich -> System zeigt Fehlermeldung an (Testfall fehlgeschlagen)</td></tr>
<tr><td>Projekt „Systemtestprojekt Inspektion“ auswählen</td><td>System öffnet Projekt und zeigt Projektansicht</td><td>System öffnet Projekt nicht und zeigt Fehlermeldung an (Testfall fehlgeschlagen)</td></tr>
<tr><td>Öffnen der Überprüfungsauswahlansicht, keine Auswahl einer Anforderungselementekategorie tätigen und die Automatisierte Inspektion starten</td><td>System weist den Benutzer durch eine Fehlermeldung darauf hin, dass er keine Anforderungselementekategorie gewählt hat</td><td>System weist den Benutzer nicht auf die fehlende Auswahl der Anforderungselementekategorie hin oder erzeugt bereits Inspektionsfragen (Testfall fehlgeschlagen)</td></tr>
<tr><td>Erneutes Öffnen der Überprüfungsauswahlansicht und Wahl der Textfeldüberprüfung für alle Anforderungselementekategorien sowie Start der automatisierten Inspektion</td><td>System führt Inspektion durch, erzeugt Frage zu fehlendem Textfeld bei „Actor_01“, keine Fehlermeldung und Laden der Fragenauswahlansicht</td><td>System erzeugt keine Frage zu fehlendem Textfeld bei „Actor_01“, erzeugt eine Fehlermeldung oder lädt die Fragenauswahlansicht nicht (Testfall fehlgeschlagen)</td></tr>
<tr><td>Öffnen der Fragedetailansicht der Frage zu fehlendem Textfeld bei „Actor_01“, schließen der Frage mit der Begründung „Ist schon recht so!“</td><td>System erzeugt einen Eintrag bei Option und eine Decision für die Frage zu fehlendem Textfeld bei „Actor_01“. Damit ist die Frage als geklärt (Resolved) gekennzeichnet.</td><td>System erzeugt keinen Eintrag bei Option und keine Decision für die Frage zu fehlendem Textfeld bei „Actor_01“. Oder die Frage ist weiterhin als offen (Open) gekennzeichnet (Testfall fehlgeschlagen)</td></tr>
<tr><td>Erneutes Öffnen der Überprüfungsauswahlansicht und Wahl der Textfeldüberprüfung und der Beziehungsüberprüfung für alle Anforderungselementekategorien sowie Start der automatisierten Inspektion</td><td>System führt Inspektion durch, öffnet Frage zu fehlendem Textfeld bei „Actor_01“ (mit Open gekennzeichnet) und erzeugt Frage zu den fehlenden Beziehungen des Aktors, keine Fehlermeldung und Laden der Fragenauswahlansicht</td><td>System öffnet Fragen zu fehlendem Textfeld bei „Actor_01“ nicht, erzeugt keine Frage zu den fehlenden Beziehungen des Aktors, erzeugt eine Fehlermeldung oder lädt die Fragenauswahlansicht nicht (Testfall fehlgeschlagen)</td></tr>
<tr><td>Ausloggen</td><td>System loggt Benutzer aus</td><td>System loggt Benutzer nicht aus und gibt eine Fehlermeldung aus (Testfall fehlgeschlagen)</td></tr>
<tr><td>Nachbedingungen (Postconditions)</td><td colspan="3">- Nutzer wieder ausgeloggt
- Projekt „Systemtestprojekt Inspektion“ wieder auf dem Eingangsstand (ohne Fragen)</td></tr>
<tr><td>Testressourcen (Test Resources)</td><td colspan="3">JWebUnit</td></tr>
</table>

Systemtestspezifikation 1: Systemtest Szenario ‚Jansen automatisierte Inspektion’

4.7.4 Rationale

Als Rationale werden wir folgende Sachverhalte auf der Interaktionsebene festhalten. In den Rationaletabellen fett gedruckte Optionen kennzeichnen die Option, für die die Entscheidung gefallen ist:

1. Wie muss die UI-Struktur für die Systemverantwortlichkeit „Anforderungsdokumente automatisiert überprüfen“ ergänzt werden?

Diese Entscheidung steht an, da wir für jede Systemverantwortlichkeiten die UI-Struktur um einen Arbeitsbereich erweitern (vgl. Kapitel 4.6.1.1). Rationaletabelle 2 zeigt die Entscheidungsfindung. Für die Entscheidung werden vor allem die Globalen Nicht-funktionalen Anforderungen hinzugezogen. Im Hinblick auf die mögliche Wiederverwendung der Fragenauswahlansicht zur Präsentation der Ergebnisse der automatisierten Inspektion (Rationaletabelle 3) wird sich für Option 2 entschieden.

Optionen		**Kriterien**		
		Geringe Anzahl von Sichten	Geringe Navigation zwischen Sichten	Überschaubarkeit der Sichten
1.	Ein einziger neuer Arbeitsbereich für die Auswahl der Prüfkriterien der Inspektion, für den Start der Inspektion und die Präsentation der Ergebnisse	**+**	**+**	**-**
2.	**Getrennte Sichten für Prüfkriterienwahl sowie den Start der Inspektion (erste Sicht) und die Präsentation der Ergebnisse (zweite Sicht)**	**-**	**-**	**+**

Bewertungen für die Berücksichtigung eines Kriteriums in der Gestaltungsalternative (Option):

++	Kriterium sehr gut umgesetzt	-	Kriterium weniger gut umgesetzt
+	Kriterium gut umgesetzt	--	Kriterium schlecht umgesetzt
N.A.	keine Angaben		

Rationaletabelle 2: Wie muss die UI-Struktur für die Systemverantwortlichkeit „Anforderungs-dokumente automatisiert überprüfen“ ergänzt werden?

2. Wie sollten gefundene Probleme bei einer Inspektion dokumentiert werden?

Diese Entscheidung resultiert aus der Überlegung, dass man ja die Fragen auch als Dokumentationsmöglichkeit für gefundene Probleme verwenden könnte. Hierdurch ergäbe sich auch der Vorteil, dass man Arbeitsbereiche/Sichten in der UI-Struktur wieder verwendet (vgl. Kapitel 4.7.1.1), damit die Anzahl der Sichten reduziert und die Überschaubarkeit der Sichten im System steigert. Näheres zur Abwägung der Optionen enthält Rationaletabelle 3.

Optionen		Kriterien			
		Geringe Anzahl von Sichten	Geringe Navigation zwischen Sichten	Überschaubarkeit der Sichten	Gefundene Probleme diskutierbar
1.	Es sollte ein eigener Report für die von einer automatisierten Inspektion gefundenen Probleme zur Verfügung stehen (-> mit eigener Sicht).	-	+	+	--
2.	**Für das Berichten der von einer automatisierten Inspektion gefundenen Probleme kann man die Fragen verwenden (-> Wiederverwendung der Fragenauswahlansicht)**	-	-	+	++
3.	Option 1 und Option 2 sollten kombiniert werden. D. h. unmittelbar nach der automatisierten Inspektion soll ein Report als Übersicht dienen und die Fragen sollen als permanenter Hinweis auf ein Problem im Projekt bestehen bleiben.	--	--	+	++

Bewertungen für die Berücksichtigung eines Kriteriums in der Gestaltungsalternative (Option):

++	Kriterium sehr gut umgesetzt	-	Kriterium weniger gut umgesetzt
+	Kriterium gut umgesetzt	--	Kriterium schlecht umgesetzt
N.A.	keine Angaben		

Rationaletabelle 3: Wie sollten gefundene Probleme bei der automatisierten Inspektion dokumentiert werden?

3. Welche Use Cases müssen der Anforderungsspezifikation aufgrund der *Änderungsaufgabe zur besseren Unterstützung des Inspektors* hinzugefügt werden und warum?

Der Use Case „Anforderungselemente automatisiert überprüfen“ muss hinzugefügt werden, um die Aufgabe „Überprüfen der Anforderungen“ mit Systemunterstützung realisieren zu können.

4. Wie sollte der Use Case „Anforderungselemente automatisiert überprüfen“ gestaltet werden?

Die Alternativen (Optionen) für eine Gestaltung des Use Cases wurden bereits in Kapitel 4.7.1.2 aufgezeigt. In Rationaletabelle 4 wird nun die Bewertung der Optionen gegen die Nicht-funktionalen Anforderungen, die Abwägung der Gesamtbewertung und die daraus resultierende Entscheidung für eine Option dokumentiert. Eine Use Case-Gestaltung sollte dabei gegen alle Qualitätskriterien von Aufgaben der übergeordneten Aufgabe, Globale Nicht-funktionale Anforderungen bzw. deren verfeinernden Qualitätskriterien für Use Case bewertet werden, die für die Gestaltungsentscheidung relevant sind.

Optionen	Kriterien				
	Geringe Anzahl von Sichten	Überschaubarkeit der Sichten	Freiraum des Inspektors	Einfache/unterstützte Zusammenstellung der Prüfkriterien	Übersichtlichkeit der Ergebnisse der automatisierten Inspektion
1. Der Aktor startet die Inspektion und das System führt daraufhin eine Inspektion nach festgelegten, vom Aktor nicht zu beeinflussenden Prüfkriterien aus. Anschließend gibt das System eine kurze Zusammenfassung der Probleme aus.	-	++	--	++	+
2. Analog zu 1), das System gibt jedoch keine Zusammenfassung der Probleme aus.	+	++	--	++	-
3. Der Aktor wählt aus drei vorgegebenen fest definierten Checklisten mit festgelegten, vom Aktor nicht zu beeinflussenden Prüfkriterien eine aus. System führt die automatisierte Inspektion anhand der ausgewählten Checkliste durch. Anschließend gibt System eine Zusammenfassung der Inspektion aus.	-	++	-	++	+
4. Analog zu 3), das System gibt jedoch keine Zusammenfassung der Probleme aus.	+	++	-	++	-
5. Der Aktor stellt sich eine Checkliste mit gewünschten Prüfkriterien aus einer Liste von möglichen Kriterien selbständig zusammen. Das System führt die automatisierte Inspektion anhand der zusammengestellten Checkliste aus. Abschließend gibt das System eine Zusammenfassung der Inspektion aus.	-	-	++	-	+
6. Analog zu 5), das System gibt jedoch keine Zusammenfassung der Probleme aus.	+	-	++	-	-
7. Der Aktor wählt aus drei vorgegebenen fest definierten Checklisten mit festgelegten, vom Aktor nicht zu beeinflussenden Prüfkriterien eine aus, kann diese jedoch zusätzlich noch nach seinen Wünschen anhand einer Liste von Prüfkriterien modifizieren. System führt anschließend die automatisierte Inspektion anhand der gewählten Checkliste durch. Abschließend gibt System noch eine Zusammenfassung der Inspektion aus.	-	+	++	++	+
8. Analog zu 7), das System gibt jedoch keine Zusammenfassung der Probleme aus.	+	+	++	++	-

Bewertungen für die Berücksichtigung eines Kriteriums in der Gestaltungsalternative (Option):

++	Kriterium sehr gut umgesetzt	-	Kriterium weniger gut umgesetzt
+	Kriterium gut umgesetzt	--	Kriterium schlecht umgesetzt
N.A.	keine Angaben		

Rationaletabelle 4: Wie sollte der Use Case „Anforderungselemente automatisiert überprüfen“ gestaltet werden?

5. Welche Systemfunktionen müssen der Anforderungsspezifikation aufgrund der *Änderungsaufgabe zur besseren Unterstützung des Inspektors* hinzugefügt werden und warum?

Die Systemfunktion „Automatisierte Prüfung der Anforderungselemente durchführen" muss hinzugefügt werden, um den Use Case „Anforderungselemente automatisiert überprüfen" umsetzen zu können.

6. Wie sollte die Systemfunktion „Automatisierte Prüfung der Anforderungselemente durchführen" gestaltet werden (Zusammenstellung der Anforderungselemente)?

Bei der Gestaltung der Systemfunktion muss eine Entscheidung getroffen werden, wie die Zusammenstellung der Anforderungselemente aus dem Projekt vollzogen wird (vgl. Kapitel 4.7.1.3). Die Entscheidungsfindung ist in Rationaletabelle 5 dokumentiert.

Optionen		**Kriterien**
		Kurze Dauer für automatisierte Inspektion
1.	Das System holt sich alle Anforderungselemente aus dem Projekt, gleicht anschließend jedes Element aus der Liste mit der vorhandenen Checkliste ab und prüft es auf die für das Element spezifizierten Kriterien.	**-**
2.	Das System geht die Prüfkriterien der Checkliste Schritt für Schritt durch, holt sich für jedes Kriterium die zugehörigen Anforderungselemente und führt die automatisierte Inspektion für jeden einzelnen Checklistenpunkt einzeln durch (d.h. das System holt sich für jedes Prüfkriterium die Anforderungselemente die es braucht jedes Mal neu).	**+**
3.	**Analog zu Vorschlag 2. Jedoch werden die Anforderungselemente, die einmal aus dem Projekt geholt werden für ein evtl. weiteres Prüfkriterium (indem sie benötigt werden) für die laufende Inspektion zwischengespeichert.**	**++**
4.	Das System überprüft die gesamte Checkliste, stellt sich eine Liste der benötigten Anforderungselementekategorien zusammen und holt anschließend nur die Elemente aus diesen Kategorien.	**N.A.**

Bewertungen für die Berücksichtigung eines Kriteriums in der Gestaltungsalternative (Option):

++	Kriterium sehr gut umgesetzt	-	Kriterium weniger gut umgesetzt
+	Kriterium gut umgesetzt	--	Kriterium schlecht umgesetzt
N.A.	keine Angaben		

Rationaletabelle 5: Wie sollte die Systemfunktionen „Automatisierte Prüfung der Anforderungs-elemente durchführen" gestaltet werden (Zusammenstellung der Anforderungselemente)?

7. Wie sollen von der automatisierten Inspektion wiederholt gefundene Probleme behandelt werden?

Diese Entscheidung steht an, da man sich für die Dokumentation der Probleme mittels Fragen entschieden hat (vgl. Rationaletabelle 3), und muss in die Gestaltung der

Systemfunktion „Automatisierte Prüfung der Anforderungselemente durchführen“ einbezogen werden. Zur Dokumentation dient Rationaletabelle 6.

Optionen		**Kriterien**			
		Keine redundanten Fragen durch die automatisierte Inspektion	Gefundene Probleme diskutierbar	Kurze Dauer für automatisierte Inspektion	Wissenstransfer zwischen den Projektmitarbeitern
1.	Eine zweite Frage wird erzeugt, die nochmals auf das Problem hinweist.	**--**	**+**	**++**	**N.A.**
2.	Vorhandene Frage wird gelöscht und eine neue Frage erzeugt.	**++**	**-**	**+**	**--**
3.	**Es wird überprüft, ob die vorhandene Frage schon geschlossen ist. Wenn ja, diese Frage wieder öffnen und keine neue Frage erzeugen. Falls Frage noch geöffnet, keine weiter Aktion, da auf das Problem schon hingewiesen wird.**	**++**	**++**	**N.A.**	**++**

Bewertungen für die Berücksichtigung eines Kriteriums in der Gestaltungsalternative (Option):

++	Kriterium sehr gut umgesetzt	-	Kriterium weniger gut umgesetzt
+	Kriterium gut umgesetzt	--	Kriterium schlecht umgesetzt
N.A.	keine Angaben		

Rationaletabelle 6: Wie sollen von der automatisierten Inspektion wiederholt gefundene Probleme behandelt werden?

4.8 Die Systemebene

Auf der Systemebene (System Level) steht die Konzeption für eine programmtechnische Realisierung der Anforderungen im Vordergrund. Ziel dieser Ebene ist, eine strukturierte, flexible Umsetzung der Spezifikation in einem Programmierparadigma (z. B. Objektorientierung) zu ermöglichen, d. h. eine Brücke zur Implementierung zu schlagen. Um dieses Ziel zu erreichen, werden Entscheidungen getroffen

- über die graphische Benutzungsschnittstelle (GUI) und
- den Systemkern (Application Core).

4.8.1 GUI

Die GUI-Betrachtung beinhaltet Entscheidungen zur Bildschirmstruktur, d. h. wie die einzelnen Bildschirmdarstellungen strukturiert werden und den Daten, die auf den jeweiligen Bildschirmdarstellungen präsentiert werden (UI-Daten). Außerdem muss bestimmt werden, wie die Abfolge der Bildschirmdarstellung, die sog. Bildschirmsequenz, geregelt ist. Hierzu werden Navigations- und Supportfunktionen spezifiziert, die der GUI Entwurf benötigt.
In der vorliegenden Arbeit wird auf diesen Bereich jedoch nicht detailliert eingegangen. Verwiesen sei zur weiteren Beschäftigung mit diesem Bereich auf das Buch „User Interface Design. A Software Engineering Perspective“ von Soren Lauesen.

4.8.2 Systemkern

Systemkernbetrachtungen beinhalten Entscheidungen, wie man die spezifizierten Daten und die spezifizierten Systemfunktionen in Systemobjekte (in der Objektorientierung repräsentiert durch Klassen) überführt. Hierbei gilt es, die „richtigen“ Klassen zu identifizieren und das Verhalten (in der objektorientierten Entwicklung durch Operationen repräsentiert) auf die Klassen zu verteilen. Dazu richtet man sich im TRAIN-Prozess nach einer Methode Ivar Jacobsons[13], die als Input die bereits erstellten Artefakte, insbesondere das Domänendatendiagramm (bzw. Verfeinerte Datenmodell), die Use Cases und Systemfunktionsbeschreibungen verwendet, um daraus ein sog. Analyseklassendiagramm

[13] JACOBSON, IVAR: Object-Oriented Software Engineering. Reading 1992.

zu entwickeln. Dieses erfasst die Klassen und deren Verhalten schematisch, aber noch nicht für die Implementierung optimiert. D. h. es legt grobe Strukturentscheidungen fest, die dann im Entwurfsklassenmodell (vgl. Kapitel 6.2) zu konkreten Implementierungsvorgaben ausgearbeitet werden.

4.8.2.1 Analyseklassendiagramm

Die o. g. Methode von Jacobson schlägt folgenden Ablauf zur Umsetzung der Use Cases, des Domänendatendiagramms (bzw. Verfeinerten Datenmodells) und der Systemfunktionsbeschreibungen in ein Analyseklassendiagramm vor:

Schritt 1: Bestimme die Klassenkandidaten aus den Use Cases, dem Domänendatendiagramm (bzw. Verfeinerten Datenmodell) und den Systemfunktionsbeschreibungen und lege geeignete Namen, Attribute und Assoziationen fest.

Schritt 2: Definiere die Grundoperationen, wie z. B. Zugriffe auf Klassenattribute.

Schritt 3: Definiere komplexe Operationen, wie z. B. Systemfunktionen aus den Use Cases und verteile dieses Verhalten auf die anderen Klassen.

Schritt 4: Nutze Vererbung und komplexe Assoziationen wie Aggregation zwischen den identifizierten Klassen.

Schritt 5: Konsolidiere das Analyseklassendiagramm, d. h. überprüfe z. B. die Verteilung von Verhalten auf die Klassen und deren Assoziationen.

4.8.2.1.1 Schritt 1: Identifizierung von Klassenkandidaten

Zur Definition der Klassen existieren verschieden Stereotype von Klassenrollen. Dies sind

- Dialog-Klasse (Boundary Class),
- Steuerungsklasse (Control Class) und
- Entitätsklasse (Entity Class).

Use Cases und Systemfunktionsbeschreibungen werden nun daraufhin untersucht, wie man die textuelle Beschreibung in die genannten Klassenrollen überführen kann. Hierzu existieren folgende Regeln:

Regeln für Klassen:

Regel 1: Entitätsklassen spiegeln Dinge oder Sachverhalte aus dem Anwendungsbereich wider. Typischerweise sind das Informationen bzw. Materialien, die verwaltet oder ausgetauscht werden.

Im Allgemeinen sind im TRAIN-Prozess diese Klassen im Domänendatendiagramm (bzw. Verfeinerten Datenmodell) bereits modelliert und können daraus übernommen werden.

Regel 2: Systemfunktionen werden im Analyseklassendiagramm zuerst durch Steuerungsklassen modelliert. Diese werden dann später in Schritt 3 auf die beiden anderen Klassenrollen verteilt.

Regel 3: Damit der Benutzer mit dem System interagieren kann, benötigt man Dialogklassen. Erwähnung von Benutzerinteraktion in Beschreibungstexten weist auf eine Dialogklasse hin.

Im TRAIN-Prozess können die Sichten aus dem Verfeinerten Datenmodell meist als Dialogklassen übernommen werden.

Regeln für Attribute:

Die zwei Grundregeln zur Unterscheidung von Klassen und Attributen lauten:

Regel 1: Klassen haben eine eigene Identität und mehrere Attribute. Diese Attribute werden für komplexere Berechnungen oder Überprüfungen gebraucht, die später als Teile komplexer Operationen modelliert werden.

Regel 2: Attribute modellieren einen (evtl. veränderlichen) Wert, der unabhängig von der Klasse nicht sinnvoll ist.

Ein anschauliches Beispiel ist das Datum:

- Ein Attribut *Datum* ist sinnvoll, wenn das Datum nur als zusammenhängender Wert abgespeichert, verändert und abgefragt wird und nur einfache Operationen damit durchgeführt werden.
- Eine Klasse *Datum* ist angemessen, wenn damit komplexere Operationen durchgeführt werden (z. B. Schaltjahrprüfung, Veränderungen einzelner Tage, Monate u. ä.).

Regeln für Assoziationen:

Regel 1: Assoziationen in Analysenklassendiagrammen spiegeln Zusammenhänge aus dem Anwendungsbereich wider. Oft entsprechen diesen Zusammenhängen Adjektive oder Verben im Use Case- bzw. Systemfunktionstext. Der Zusammenhang zwischen Buchung und Veranstaltung könnte sich z. B. aus

den Textteilen „die Veranstaltung ist gebucht“ oder „gebuchte Veranstaltung“ ergeben.

Für den TRAIN-Prozess sind die meisten Beziehungen für Entitätsklassen bereits im Domänendatendiagramm (bzw. Verfeinerten Datendiagramm) abzulesen.

Regel 2: Gibt es zwischen einer Gruppe von *Entitätsklassen* nur zweistellige Assoziationen (d. h. Assoziationen zwischen jeweils zwei Klassen dieser Gruppe), ist zu überlegen, ob man einige davon nicht aus anderen ableiten kann (z. B. Ableitung von Kunde–Veranstaltung aus Kunde–Buchung und Buchung–Veranstaltung).

4.8.2.1.2 Schritt 2: Grundoperationen definieren

Im zweiten Schritt werden die Grundoperationen definiert. Hierzu unterscheidet man zwei verschiedene Zugriffsarten auf Attribute durch Operationen:

1. *Direkter Zugriff*, d. h. die Methode eines Objektes greift auf dessen eigene Attribute zu.
2. *Indirekter Zugriff*, d. h. die Methode eines Objektes greift auf Attribute anderer Objekte durch Aufruf von Methoden dieser Objekte zu.

Operationen mit der ersten Eigenschaft werden *Grundoperationen* genannt, die zweiten *komplexe Operationen*.

Zur Identifikation von Grundoperationen gibt es folgende Regel:

Regel: Grundoperationen spiegeln Überprüfungen oder Berechnungen von und mit Attributen eines Objekts wider. Oft sind diese Überprüfungen oder Berechnungen mit entsprechenden Verben im Use Case-Text genannt. Teils müssen sie aber auch aus der Kenntnis der Attributbedeutung ergänzt werden.

4.8.2.1.3 Schritt 3: Komplexe Operationen definieren und auf andere Klassen verteilen

Im dritten Schritt werden die vorhandenen komplexen Operationen (Steuerungsklassen) auf die Entitäts- und Dialogklassen verteilt und gegebenenfalls komplexe Operationen aufgebrochen bzw. zusammengefasst. Um die Alternativen für die Verteilung zu erhalten, werden die Steuerungsklassen mit den Klassen assoziiert, deren Daten bzw. Beziehungen die Steuerungsklasse als Input bzw. Output benötigt. Hiernach kann die Verteilung erfolgen:

Regel 1: Verteile Steuerungsklassen auf Entitäts- und Dialogklassen.

- Falls Steuerungsklasse nur auf Attributen (und Beziehungen) *einer* anderen Klasse arbeitet: die Steuerungsklasse dieser Klasse zuordnen ➔ *Kanonische Lösung*
- Falls als Eingabe nur Attribute *einer* Klasse oder als Ausgabe nur Attribute *einer* Klasse: die Steuerungsklasse dieser Klasse zuordnen ➔ *Einfache Lösung*
- Wenn Attribute mehrerer Klassen als Eingaben oder als Ausgaben: Abwägung notwendig ➔ *keine „beste Lösung"*

Regel 2: Fasse evtl. Operationen zusammen oder teile sie weiter auf

- Operationen können mehrere Use Case-Schritte umfassen (Achtung: nur bei kanonischer Lösung)
- Use Case-Schritte evtl. noch in weitere Operationen aufspalten (falls auch eigenständig sinnvoll)

Die vorgenommene Verteilung der Operationen sollte durch Erstellung von Sequenzdiagrammen für jeden Use Case und für jede Systemfunktion überprüft werden. Als Grundlage zur Erstellung der Sequenzdiagramme können auch die auf der Interaktionsebene aufgestellten Szenarien dienen. Die Sequenzdiagramme sollte man dann auf folgende Merkmale hin untersuchen:

- Zu *hohe Kopplung*

  Sind für die Umsetzung sehr viele Objekte und Nachrichten notwendig?

  Falls ja: kann die Anzahl durch Umverteilung von Operationen verringert werden?
- Zu *niedrige Kohäsion*

  Gibt es Operationen einer Klasse, die auf getrennten Attributbereichen arbeiten?

  Falls ja: wird die Komplexität der Sequenzdiagramme durch eine Aufteilung dieser Operationen auf zwei Klassen erhöht?

Manchmal ist es auch sinnvoll, Steuerungsklassen in der Implementierung beizubehalten,

- um zu große Implementierungsklassen zu vermeiden,
- um die Austauschbarkeit von alternativen Vorgängen durch eine Vererbungshierarchie von Steuervorgängen zu gewährleisten,

- um Zwischenzuständen von lang andauernden Vorgänge speichern zu können oder
- um dieselbe Vorgangsklasse für unterschiedliche Implementierungen wieder zu verwenden.

Man sollte sich hierbei jedoch darüber bewusst sein, dass Objektifizierung von Verhalten zwar Flexibilität in der Implementierung und Entkopplung des Verhaltens von bestimmten Klassen bringt, jedoch einen hohen organisatorischen Zusatzaufwand nach sich zieht und der eigentlichen objektorientierten Idee – Verhalten den Klassen zuzuordnen auf deren Daten gearbeitet wird – zuwiderlaufen kann. Deshalb sollte die Objektifizierung von Verhalten nur eingesetzt werden, wenn sie sinnvoll begründbar ist.

4.8.2.1.4 Schritt 4: Vererbung und komplexe Assoziationen integrieren

In diesem Schritt wird überlegt, wie die Vererbungs- und Assoziationsstruktur der identifizierten Klassen aussehen wird.

4.8.2.1.5 Schritt 5: Konsolidierung des Analyseklassendiagramms

Regeln für den Umgang mit Dialogklassen:

Regel 1: Dialogklassen beibehalten, wenn sie eigene Operationen oder wichtige Attribute enthalten.

Regel 2: Andernfalls Dialogklassen einer Klasse zuordnen.

Regel 3: Dialogklassen in eine separate Einheit legen (Dialogschnittstelle).

Regeln um die Assoziationen zu konsolidieren:

Regel 1: Alle notwendigen Kommunikationswege müssen abdeckt sein. D. h. jede Klasse mit einer komplexen Operation *Op* muss eine Assoziation zu allen Klassen haben, deren (Grund-) Operationen bei der Ausführung von *Op* benötigt werden.

Regel 2: Unnötige Kommunikationswege sollen eliminiert werden. D. h. es ist zu überprüfen, ob Assoziationen, die in keinem Use Case als Kommunikationsweg genutzt werden, wirklich sinnvoll sind.

4.8.3 Nicht-funktionale Anforderungen

Auf dieser Ebene können sich weiterhin die auf den anderen Ebenen bereits genannten Nicht-funktionalen Anforderungen ergeben.

4.8.4 Qualitätssicherungsmaßnahmen

4.8.4.1 Inspektion

Auch auf dieser Ebene sollte die Softwaredokumentation inspiziert werden, um die Verständlichkeit, Vollständigkeit, Eindeutigkeit, Korrektheit und Konsistenz bei Eintritt in die weiteren Prozessphasen zu gewährleisten. Besonders wichtig sind die genannten Kriterien bei „Abschluss“ der Anforderungsspezifikation und vor Eintritt in die nächsten Phasen, da sich häufig die beteiligten Personen der verschiedenen Phasen unterscheiden.

4.8.5 Rationale

Wie auch auf den anderen Ebenen soll die Umsetzung von Entscheidungen vorangegangener Ebenen für jeden nachvollziehbar sein. Besonderes Gewicht liegt hier auf der Begründung der Entscheidungen im Analyseklassendiagramm.

4.9 Die Systemebene im Beispiel

4.9.1 Systemkern

4.9.1.1 Analyseklassendiagramm

Als Grundlage für die folgenden Schritte zur Entwicklung des Analyseklassendiagramms verwenden wir unseren Use Case „Anforderungselemente automatisiert überprüfen“ (Interaktionsbeschreibung 1), das Verfeinerte Datenmodell (Abbildung 9) und die Systemfunktion „Automatisierte Prüfung der Anforderungselemente durchführen“ (Systemfunktionsbeschreibung 1).

4.9.1.1.1 Schritt 1: Identifizierung von Klassenkandidaten

Aus dem Verfeinerten Datenmodell identifizieren wir als Entitätsklassen

- Anforderungselementekategorie
- Checkliste
- Problem
- Anforderungselement
- Prüfkriterium
- Frage

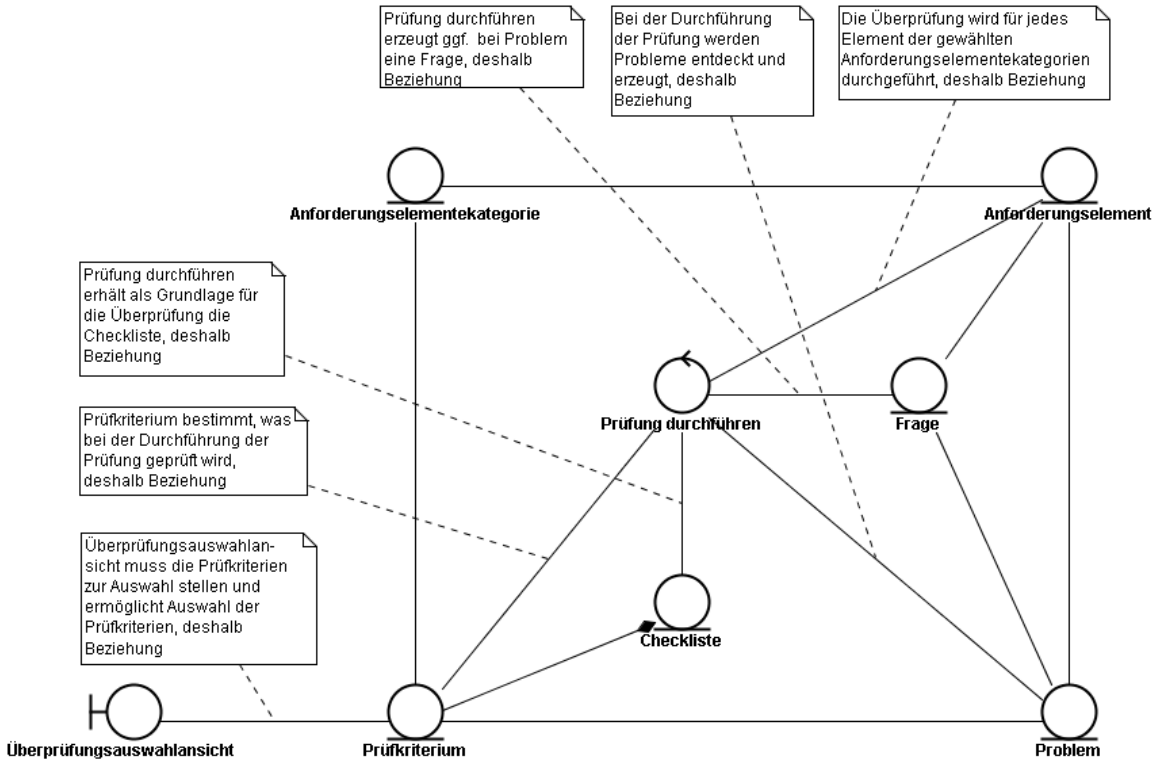


Abbildung 10: Analyseklassendiagramm nach Schritt 1

und als Dialogklasse

- Überprüfungsauswahlansicht.

Die Systemfunktion „Automatisierte Prüfung der Anforderungselemente durchführen“ ergibt die Steuerungsklasse

- Prüfung durchführen.

Behält man die Assoziationen des Verfeinerten Datenmodells bei und bezieht die im Use Case und in der Systemfunktion beschriebenen Assoziationen mit ein, ergibt sich Abbildung 10 (die neu hinzugekommenen Assoziationen werden begründet).

4.9.1.1.2 Schritt 2: Grundoperationen definieren

In unserem Beispiel spielt dieser Schritt keine Rolle.

4.9.1.1.3 Schritt 3: Komplexe Operationen definieren und auf andere Klassen verteilen

Die einzige komplexe Operation in der Änderungsaufgabe ist die Systemfunktion „Automatisierte Prüfung der Anforderungselemente durchführen“, die wir bereits als Steuerungsklasse „Prüfung durchführen“ modelliert haben. Betrachten wir deren Aufgaben, so stellen wir fest, dass man diese in mehrere Operationen aufteilen sollte.

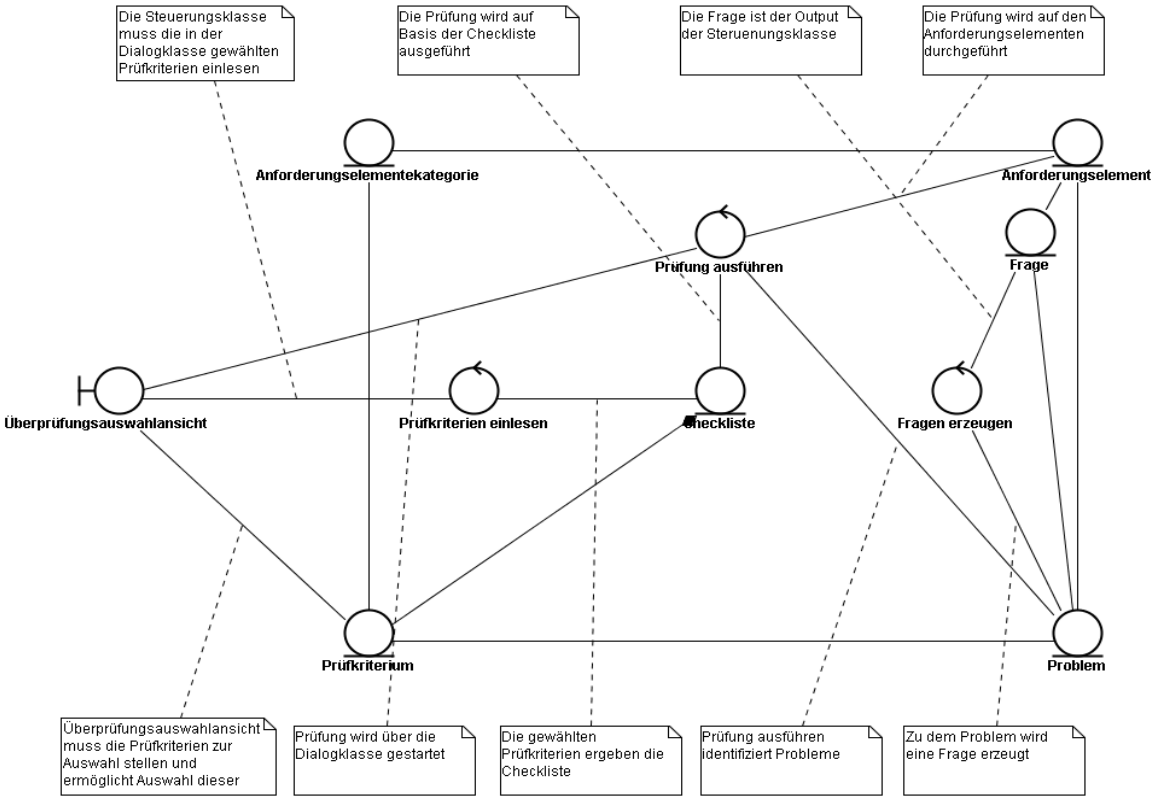


Abbildung 11: Das Analyseklassendiagramm mit den neu hinzugekommenen Assoziationen der Steuerungsklassen

Hierdurch kann man die einzelnen Operationen den Klassen zuordnen, auf deren Daten sie operieren:

- *Prüfkriterien einlesen:* Einlesen der Prüfkriterien und Zusammenstellen der Checkliste.
- *Prüfung ausführen:* Durchführen der eigentlichen Überprüfungen.
- *Fragen erzeugen:* Erzeugen der Fragen bei einem Problem eines Anforderungselementes.

In Abbildung 11 sind diese als Steuerungsklassen modelliert und mit Beziehungen zu den übrigen Klassen versehen. Auch die Beziehungsbegründungen zwischen den neuen Klassen sind in die Abbildung integriert.

Die Operationen (Steuerungsklassen) müssen weiter auf die Entitäts- und Dialogklassen aufgeteilt werden. Dies kann entlang der in Abbildung 11 enthaltenen Assoziationen der Steuerungsklassen vorgenommen werden. Die Aufteilung der Operationen und wiederum die Begründungen sind in Abbildung 12 zu sehen.

Die Aufteilung der Steuerungsklassen wird durch ein Sequenzdiagramm (Abbildung 13) basierend auf dem Use Case „Anforderungselemente automatisiert überprüfen“ und der Systemfunktion „Automatisierte Prüfung der Anforderungselemente durchführen“ auf

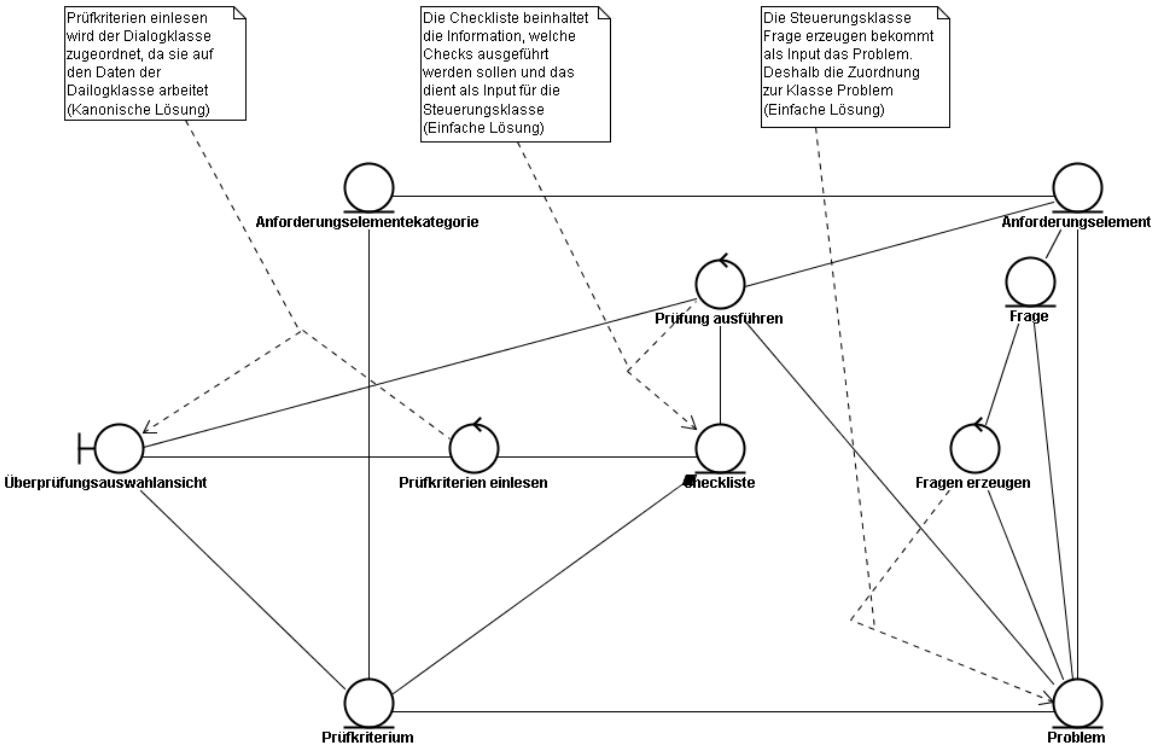


Abbildung 12: Aufteilung der Steuerungsklassen im Analyseklassendiagramm

Kopplung und Kohäsion überprüft. Auf eine zusätzliche Evaluierung weiterer Verteilungsmöglichkeiten, wie sie in der Praxis zu einer bestmöglichen Verteilung führen sollte, verzichten wir hier.

Während der Entwicklung des Sequenzdiagramms stellt man fest, dass man zum einen die Klasse *Project*, die bereits im bestehenden Programmkode vorhanden ist, mit einbeziehen sollte, um den Kontrollfluss klarer darzustellen. Die Klasse repräsentiert ein Projekt, bestehend aus Anforderungselementen und Fragen. Zum anderen erkennt man noch einige fehlende Operationen:

- eine Operation, die die Überprüfungsauswahlansicht anzeigt (showSelectionForm)
- eine Operation, die die Prüfung startet (startCheck)
- eine Operation, die die Frage stellt (createIssue)

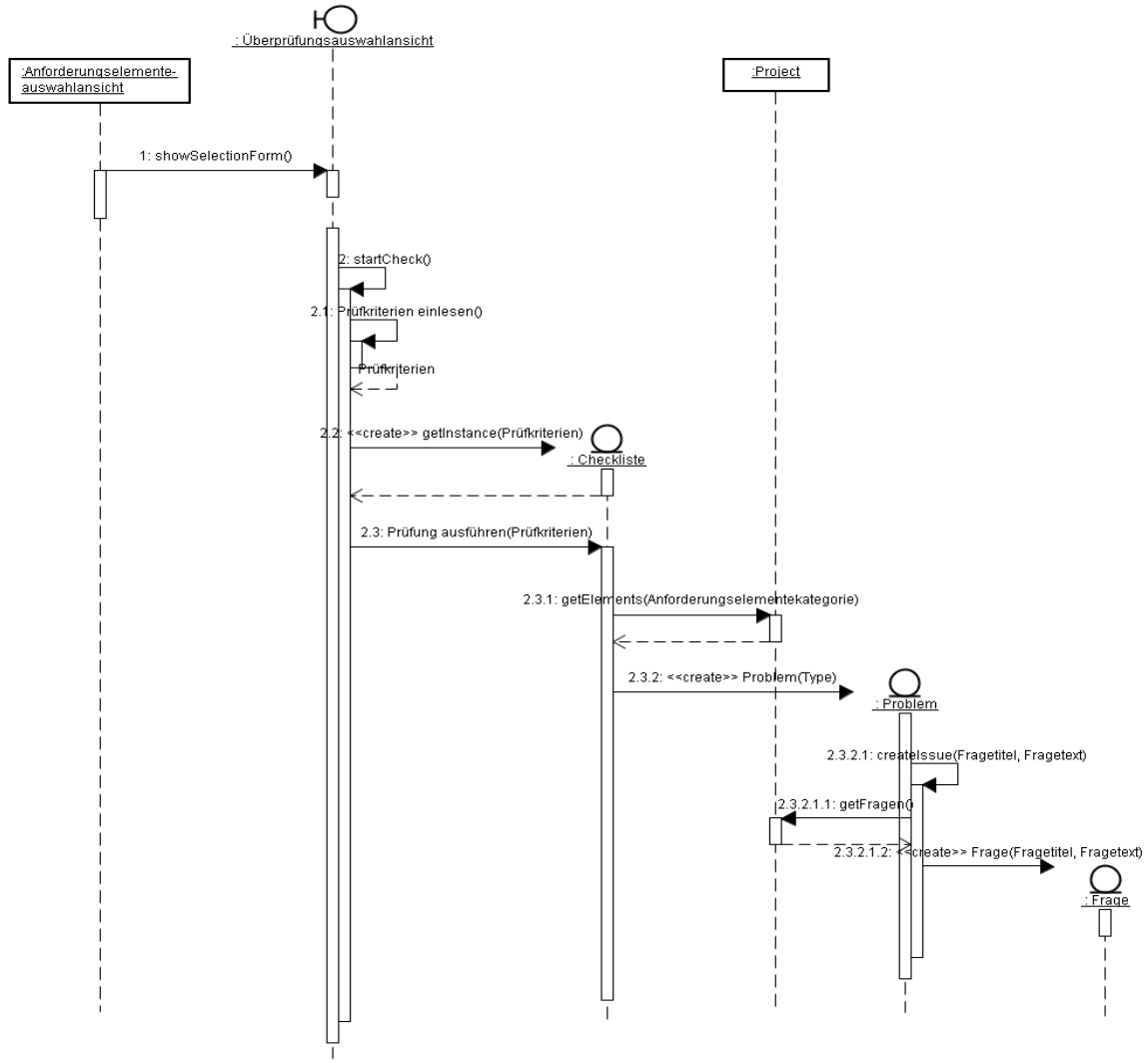


Abbildung 13: Prüfung der Kopplung und Kohäsion bei der im Analyseklassendiagramm gegebenen Verteilung der Operationen

4.9.1.1.4 Schritt 4: Vererbung und komplexe Assoziationen integrieren

In unserem Beispiel existiert zwischen den neu hinzugekommenen Klassen keine Vererbungsstruktur. Die Aggregationsbeziehung zwischen *Prüfkriterien* und *Checkliste* ist bereits in den erstellten Diagrammen enthalten. Deshalb verzichten wir auf eine erneute Erstellung eines Diagramms.

4.9.1.1.5 Schritt 5: Konsolidierung des Analyseklassendiagramms

Die Dialogklasse *Überprüfungsauswahlansicht* wird beibehalten, da sie eigene Operationen besitzt (vgl. 4.8.2.1.5 Regel 1). Nachdem auch alle Assoziationen überprüft wurden, sieht das Analyseklassendiagramm wie in Abbildung 14 aus; die weiterhin bestehenden Assoziationen sind wiederum integriert begründet.

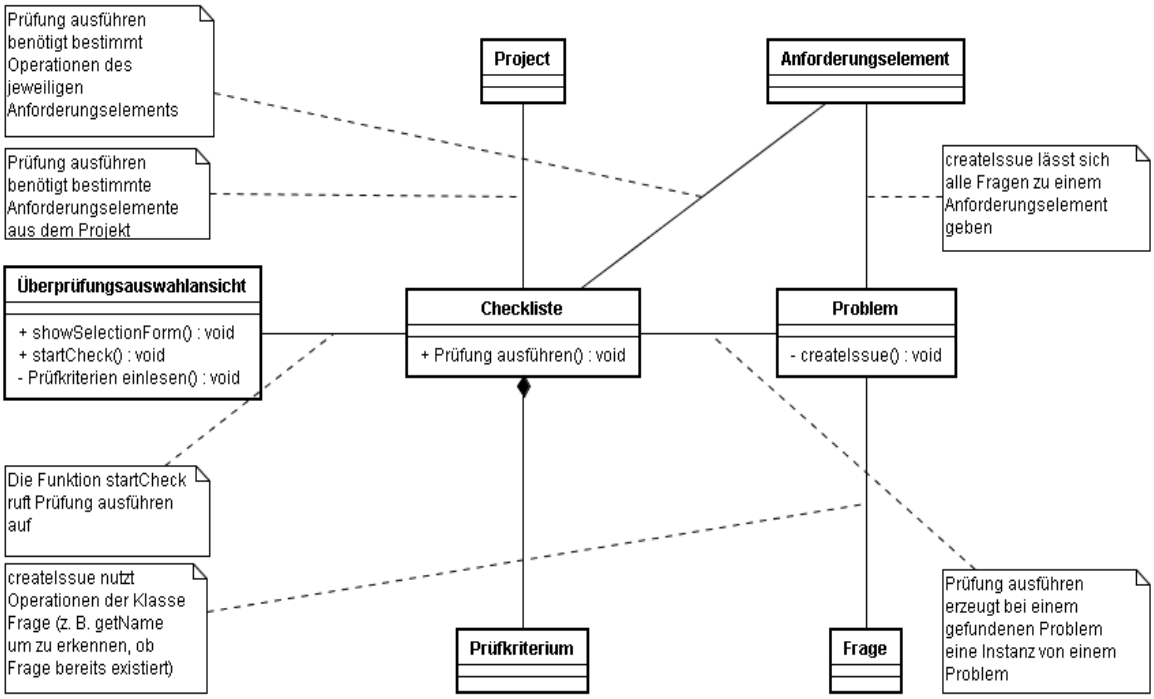


Abbildung 14: Analyseklassendiagramm nach dem letzten Schritt

4.9.2 Nicht-funktionale Anforderungen

4.9.2.1 Qualitätskriterien an die Architektur

Auf der Systemebene kommen in unserem Beispiel keine neuen Nicht-funktionalen Anforderungen mehr hinzu. Aus den in Kapitel 4.8.2.1.3 angesprochenen Zielen niedrige Kopplung und hohe Kohäsion sind in der bestehenden Sysiphus-Dokumentation jedoch

zwei Qualitätskriterien an die Architektur entstanden, die auch noch in unserem Beispiel relevant werden:

Niedrige Kopplung

Die Kopplung zwischen Komponenten soll möglichst gering sein.

Qualitätskriterium an die Architektur 1: Niedrige Kopplung

Hohe Kohäsion

Die Kohäsion innerhalb einer Komponente soll möglichst hoch sein.

Qualitätskriterium an die Architektur 2: Hohe Kohäsion

4.9.3 Rationale

Das Rationale für das Analyseklassendiagramm haben wir weitgehend in die Diagramme integriert. Alternativdiskussionen zu dessen Gestaltung werden in dieser Arbeit nicht geführt, sollten jedoch in der Praxis dazugehören.

5 Architekturdefinition im TRAIN-Prozess

Die Architektur eines Softwaresystems beschreibt nach IEEE Standard 1471-2000 die grundlegende Organisation eines Systems, verkörpert durch seine Komponenten, deren Beziehungen zueinander und zu deren Umgebung. Das Ziel der Architekturdefinition ist, die Architektur für das zu entwickelnde System festzulegen und zu dokumentieren.
Hierzu werden Entwurfsziele (Kapitel 5.1) erarbeitet, die dann in den Architekturentwurf (Kapitel 5.2) einfließen. Wie in jeder Phase des Prozesses werden auch in die Architekturdefinition Qualitätssicherungsmaßnahmen (Kapitel 5.4) und Rationale (Kapitel 5.5) integriert. Kapitel 5.6 veranschaulicht die Phase im Beispiel.
Da in dieser Arbeit der Bereich Architekturdefinition nur angerissen wird, sei an dieser Stelle zur ausführlichen Beschäftigung mit diesem Thema auf das Buch „Object-Oriented Software Engineering“ von Bernd Bruegge und Allen H. Dutoit verwiesen.

5.1 Entwurfsziele

Entwurfsziele beschreiben die durch den Entwurf angestrebten Systemeigenschaften. Diese werden anfangs sehr allgemein gefasst, wie beispielsweise:

- hohe Performanz (z. B. durch starke Hardware oder Verringerung der Kommunikation)
- hohe Flexibilität (z. B. durch Abstraktion bzw. Kapselung)
- Persistenz (z. B. durch Datenbanksystem oder Filesystem)
- Integration (z. B. Einbezug von Altsystemen)
- Ausfallsicherheit (z. B. durch Verteilung)

Unter Einbeziehung der während des Requirements Engineering aufgestellten Qualitätskriterien an die Architektur werden diese jedoch zu konkreten Vorgaben für den Architekturentwurf.

5.2 Architekturentwurf

Im Architekturentwurf wird unter Berücksichtigung der Entwurfsziele, der objektorientierten Entwurfsprinzipien (hohe Kohäsion, niedrige Kopplung, hierarchische Zerlegung bzw. Schichtung und Wiederverwendbarkeit) und der (Architektur-) Muster

sowie der Nicht-funktionalen Anforderungen die Entscheidungen zur Architektur des Systems gefällt. Hierzu zählen Entscheidungen zur Zerlegung des Systems in Subsysteme, zu deren Verteilung auf die Hardware (Hardware/Software Mapping), zur Datenpersistenz, zu Sicherheit und Zugangskontrolle sowie zum globalen Kontrollfluss des Systems. Festgehalten werden die Entscheidungen in der Architekturdokumentation (System Design Document). Eine Gliederung[14] wird vom TRAIN-Prozess vorgeschlagen:

5.2.1 Gliederung der Architekturdokumentation

1. Einleitung

   Beinhaltet das Ziel der Software, die Entwurfsziele, Definitionen und Abkürzungen; außerdem Referenzen auf andere Dokumente und Annahmen, die zugrunde gelegt werden.

2. Soll-Architektur (gegebenenfalls aufbauend auf Ist-Architektur) unter Angabe von
   a. Überblick

      Kurzüberblick über die Soll-Architektur mit Kurzbeschreibung der Verantwortlichkeiten der einzelnen Subsysteme (Komponenten).

   b. Komponenten und Schnittstellen (Subsysteme)

      Beschreibung der Zusammensetzung des Gesamtsystems aus den Komponenten sowie deren Verantwortlichkeiten, deren Schnittstellen und der externer Systeme.

   c. Ressourcen und Abbildung der Komponenten auf die Ressourcen (Hardware/Software Mapping)

      Beschreibung der Verteilung der Komponenten auf die Hardware und Überlegungen zu benötigter Hardware.

   d. Betriebskonzept

      Beschreibung der Grenzfälle während der Nutzung des Systems, wie z. B. Installation, De-Installation, Hochfahren usw.

3. Hervorgehobene Aspekte
   a. Datenverwaltung (Persistenz)

      Beschreibung, wie Daten dauerhaft gespeichert werden sollen. Beinhaltet ebenfalls Überlegungen zu Datenbank- oder Dateisystemen bzw. Datenschemata.

   b. Zugangskontrolle und Sicherheit

      Beschreibung, wie Zugangskontrolle, Authentifizierung, Verschlüsselung usw. erfolgen soll.

---

[14] Nach BRUEGGE, BERND; DUTOIT, ALLEN H.: Object-oriented Software engineering. Upper Saddle River 2004. S. 283f.

c. Allgemeiner Kontrollfluss

Beschreibung der Anfragestruktur an das System bzw. zwischen den Komponenten.

4. Komponentenbeschreibung

Beschreibung der Dienste, die jede Komponente bereitstellt (nahe an Interfacebeschreibung).

Diese Gliederung beinhaltet die von den am Softwareentwicklungsprozess beteiligten Gruppen eingenommenen Sichten auf die Architektur. Dies sind:

5.2.2 Sichten auf die Architektur

5.2.2.1 Konzeptionelle Sicht

Die konzeptionelle Sicht ermöglicht in erster Linie dem Auftraggeber und dem Nutzer des Systems einen Grobüberblick über das System mit seinen Komponenten, externen Systemen und gegebenenfalls seinen benötigten Ressourcen. Als Darstellungsmittel eignen sich besonders Konfigurationsdiagramme und eine textuelle Beschreibung.

5.2.2.2 Komponentensicht

Die Komponentensicht betrachtet die Architektur aus dem Blickwinkel eines Systemdesigners bzw. -programmierers. Welche Komponenten beinhaltet das System, wie stehen diese in Beziehung (innere Struktur), worin liegen deren Verantwortlichkeiten und wie sehen deren Schnittstellen aus? Diese Darstellung ermöglichen sowohl Block- als auch Implementierungsdiagramme.

5.2.2.3 Physikalische Sicht

Die physikalische Sicht liefert einen Strukturüberblick für den Auftraggeber und den Systemprogrammierer. Diese benötigen einen Überblick über die vom System in Anspruch genommenen Ressourcen und die Verteilung der Systemkomponenten auf diese. Zur Illustration werden Konfigurations- und Verteilungsdiagramme verwendet.

5.2.2.4 Laufzeitsicht

Integratoren und Systemprogrammierern bietet die Laufzeitsicht einen Strukturüberblick über Prozesse, die während der Laufzeit des Systems aktiv sein können und koordiniert werden müssen. Für deren Dokumentation eignet sich meist eine textuelle Beschreibung. Auch der Kontrollfluss innerhalb des Systems spielt für den Entwickler eine entscheidende Rolle und wir meist mittels Sequenzdiagrammen oder textuellen Beschreibungen festgehalten.

5.2.2.5 Betriebssicht

Die Betriebssicht beinhaltet ein Konzept zum Betrieb des Systems für den Auftraggeber bzw. den Operator. In ihr werden die Grenzfälle der Nutzung wie z. B. Installation, De-Installation, Hochfahren, Herunterfahren, Ressourcen-Ausfall und Komponenten-Ausfall beschrieben, sowie Maßnahmen zur Gewährleistung eines reibungsfreien Ablaufs und in Problemsituationen diskutiert.

Tabelle 2 stellt für jede der genannten Sichten den Bezug zu o. g. Gliederung der Architekturdokumentation her. Die Spalte „Bereich in der Architekturdokumentation“ zeigt, in welchem Kapitel der Architekturdokumentation die Sichten repräsentiert sind. Des Weiteren umfasst die Tabelle die Zielgruppe, den Zweck und die zur Dokumentation verwendeten Artefakte der jeweiligen Sichten.

Sicht	Beteiligte/Zielgruppe	Zweck	Bereich in der Architektur-dokumentation	Artefakte
Konzeptionelle Sicht	Auftraggeber, Nutzer	Grobüberblick	Externe Systeme, Komponenten, Ressourcen	Konfigurations-diagramm
Komponentensicht	Systemdesigner, Systemprogrammierer	Bündelung und Konkretisieren der Analyseklassen, Vorlage zur Implementierung	Komponenten, Schnittstellen	Blockdiagramm, Implementierungs-diagramm
Physische Sicht	Auftraggeber, Systemprogrammierer	Strukturüberblick	Ressourcen und Verteilung der Komponenten	Konfigurations-diagramm, Verteilungsdiagramm
Laufzeitsicht	Integratoren, Systemprogrammierer	Strukturüberblick	Allgemeiner Kontrollfluss, Prozesse	Sequenzdiagramm
Betriebssicht	Auftraggeber, Operator	Anleitung zum Betrieb des Systems	Betriebskonzept	Betriebskonzept

Tabelle 2: Sichten der Architekturdefinition

5.3 Nicht-funktionale Anforderungen

5.3.1 Qualitätskriterien an die Architektur

Während der Architekturdefinitionsphase (und weiter auch im Feinentwurf) werden für die Gestaltungsentscheidungen die Entwurfsziele und die objektorientierten Entwurfsprinzipien relevant. Daher ist es möglich, diese als Qualitätskriterien an die Architektur zu formulieren.

5.4 Qualitätssicherungsmaßnahmen

5.4.1 Inspektion

Die Qualität der Artefakte der Architekturdefinition wird wiederum durch die Integration des Inspektionsprozesses sichergestellt.

5.5 Rationale

In der Architekturdefinition sind, wie in jeder Phase des Prozesses, die getroffenen Entscheidungen über die Artefakte mittels der Nicht-funktionalen Anforderungen zu begründen. In erster Linie dienen hierzu die im Requirements Engineering aufgestellten Qualitätskriterien an die Architektur. Beispiele für das Rationale in dieser Phase wären die Begründung des Betriebskonzeptes, der Schichtenaufteilung bzw. -zuordnung der Klassen, des Kontrollflusses für das System oder der Verwendung bestimmter Architekturmuster (z. B. von Client-Server Architekturen).

5.6 Architekturdefinition im Beispiel

Im Grunde besteht die Architekturdefinition für Sysiphus bereits. Eine systematische Vorstellung und Diskussion der bestehenden Architekturdefinition über die bereits in Kapitel 3.2 gemachten Anmerkungen zur Architektur von Sysiphus hinaus würde den Rahmen der Arbeit sprengen. Deshalb werden wir im Folgenden nur auf die für die Änderungsaufgabe relevanten Aspekte eingehen. Somit werden wir einige Gesichtspunkte, die bei der Neuentwicklung eines Systems wichtig sind, außer Acht lassen.

Unter Architekturgesichtspunkten ist nun wichtig:

- die Integration der neu zu entwickelnden Klassen in die bestehende Schichtenarchitektur des Systems (Komponentensicht, Kapitel 5.6.1) und
- die Einbindung der neuen Inspektionsfunktionalität in den allgemeinen Kontrollfluss des Systems (Laufzeitsicht, Kapitel 5.6.2).

5.6.1 Zuordnung der neuen Klassen in die Schichtenarchitektur von Sysiphus[15]

Sysiphus ist in mehrere Schichten gegliedert (vgl. Abbildung 4). Die Kommunikations- und Speicherschicht wird durch den *ElementStore* und das *ProjectDirectory* repräsentiert. Der ElementStore ist für die Speicherung der Systemmodell- und Rationaledaten und für die Kommunikationsinfrastruktur von Sysiphus verantwortlich. Das ProjectDirectory dient der Zuweisung von Benutzerrechten und der Authentifizierung der Benutzer bei Zugriff auf ein Projekt. Die Software Engineering Modelle und deren Beziehungen werden in der Modellschicht realisiert. Das bedeutet, dass es hier beispielsweise Elemente wie Aufgaben, Use Cases und Systemfunktionen gibt, die in Beziehung zueinander gesetzt werden können. Aber auch Elemente zur Modellierung von Dokumenten und Modelle zur Kommunikation und Begründungserfassung sind in der Modellschicht vorhanden. Die Anwendungsschicht beinhaltet die Benutzeranwendungen *REQuest*, *RAT* und *LectureAdmin*. REQuest ermöglicht den webbasierten Zugriff auf die Dokumente und Modelle sowie deren Änderung. RAT – eine funktionsreichere Benutzeroberfläche programmiert in JavaSwing – tut dies ebenfalls und ermöglicht zusätzlich u. a. das

[15] Nach PAECH, BARBARA; BORNER, LARS; u. a.: Vom Kode zu den Anforderungen und wieder zurück: Software Engineering in sechs Semesterwochenstunden. In: LÖHR, KLAUS-PETER; LICHTER, HORST (Hrsg.): Software Engineering im Unterricht der Hochschulen. Heidelberg 2005.

Zeichnen und automatisierte Generieren von UML-Diagrammen. LectureAdmin dient der Verwaltung der Projekte und deren Nutzer in der Lehre.
Mit diesen Informationen können für die Änderungsaufgabe Überlegungen darüber angestellt werden, welcher Schicht man die neuen Klassen zuordnet: Die Wahl fällt für die Klassen *Checkliste*, *Prüfkriterium* und *Problem* auf die Modellschicht, da die Klassen das Modell ergänzen, genauer gesagt das Inspektionsmodell darstellen. Die Dialogklasse *Überprüfungsauswahlansicht* ist allerdings der Anwendungsschicht und hierin der Anwendung REQuest zuzuordnen.

5.6.2 Einbindung der neuen Funktionalität in den allgemeinen Kontrollfluss von REQuest

Um unsere neue Funktionalität in den Kontrollfluss von REQuest einbinden zu können, muss dieser bekannt sein. Grob lässt er sich folgendermaßen darstellen: Jede Anfrage an den Server wird an das Java Servlet *REQ* geleitet. Von der Java Servlet Spezifikation ist festgelegt, dass eine http-Anfrage an ein Servlet in dessen Methode doPost() bzw. doGet() behandelt wird. In unserem REQ-Servlet ruft die Methode doPost() die Methode doGet() auf, so dass diese die Kontrolle sowohl über eintreffende post- als auch get-Anfragen übernimmt. Steuerungsgrundlage sind im Falle von REQuest mit der http-Anfrage übergebene Parameter. Der Kontrollfluss der Methode sieht wie folgt aus:

- Es wird überprüft, ob eine Verbindung zum ElementStore vorhanden ist, gegebenenfalls Fehlermeldung.
- Die Authentifizierung des Nutzers wird überprüft, gegebenenfalls Weiterleitung zum Login.
- Ist der Benutzer berechtigt auf das System zuzugreifen, serialisierte Weiterleitung der Anfrage an die Methode doAction() der Klasse *ProjectIntf*.

In der Klasse ProjectIntf übernimmt die Operation doAction() die Steuerung und entscheidet wiederum aufgrund der übergebenen Parameter, was mit der Anfrage weiter geschieht. Beispielsweise kann ein Methodenaufruf einer weiteren Benutzungsschnittstellenklasse erfolgen. Welche Methode aufgerufen wird, bestimmt der in der http-Anfrage des Clients übermittelte Anfrageparameter *action*. Ein spezifischer Kontrollfluss für die Erzeugung eines neuen Aktors, unter der Annahme, dass der Nutzer bereits im System eingeloggt und ein Projekt geöffnet ist, ist in Abbildung 15 dargestellt.
In dieser wird ersichtlich, dass die Operationen doAction() der Benutzungsschnittstellenklassen, die Weiterleitungen der Anfragen an die entsprechenden Klassen (auch sich selbst) bzw. deren Operationen übernehmen. Somit

muss später im Feinentwurf darauf geachtet werden, dass bei Wahl der automatisierten Inspektion die doAction() Operationen derart angepasst werden, dass über die Parameter der http-Anfrage die entsprechenden Operationen der neuen Benutzungsschnittstellenklassen aufgerufen werden.

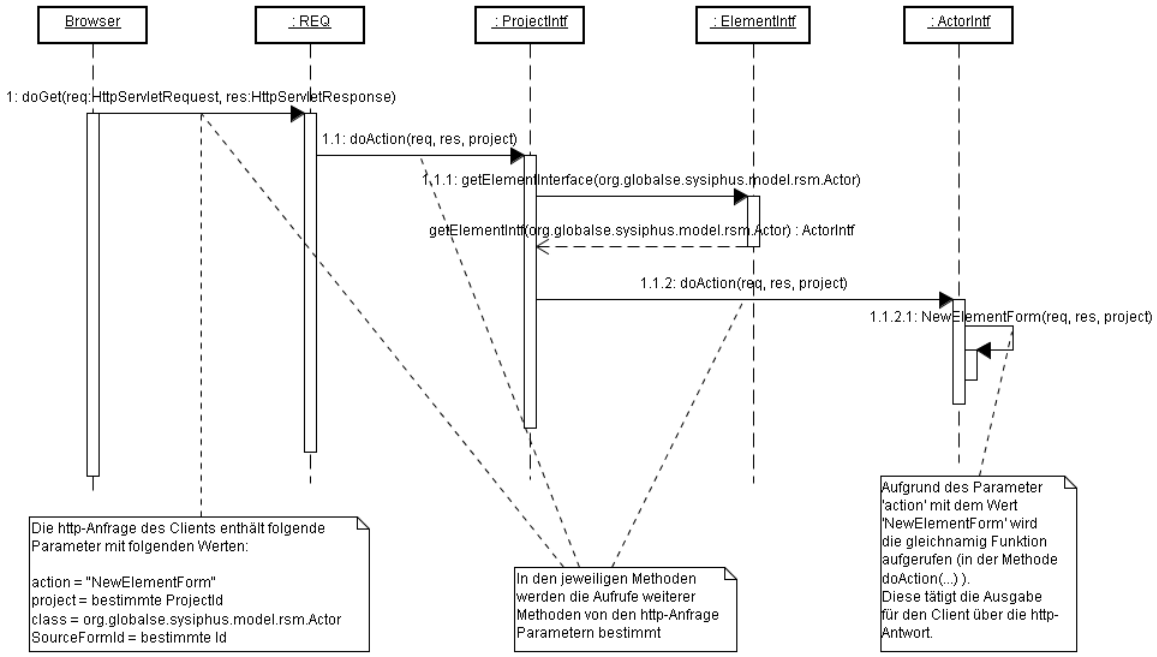


Abbildung 15: Kontrollfluss einer http-Anfrage an REQuest zur Erzeugung eines neuen Aktors

5.6.3 Nicht-funktionale Anforderungen

5.6.3.1 Qualitätskriterien an die Architektur

Wie in Kapitel 5.3 angesprochen, können aus den Entwurfszielen der Architekturdokumentation und den objektorientierten Entwurfsprinzipien, Qualitätskriterien an die Architektur formuliert werden. Diese sind in der bestehenden Sysiphus-Dokumentation bereits vorhanden. Für unser Beispiel relevant werden folgende:

Realisierung einer Schichtenarchitektur (konkretisiert bestehende Qualitätskriterien an die Architektur „Hohe Flexibilität“ und „Gute Wartbarkeit“)

Zur Austauschbarkeit einzelner Systemteile (z.B. GUI, Persistenzschicht) soll eine Schichtenarchitektur realisiert werden.

Qualitätskriterium an die Architektur 3: Realisierung einer Schichtenarchitektur

Kapselung

Daten und Funktionalität sollen möglichst gekapselt werden.

Qualitätskriterium an die Architektur 4: Kapselung

5.6.4 Rationale

Bezüglich der Komponentensicht ist die Zuordnung der neuen Klassen zu den bestehenden Schichten zu begründen. Die Begründung ist bereits in Kapitel 5.6.1 erwähnt, soll jedoch nochmals explizit in einer Rationaletabelle dokumentiert werden (Rationaletabelle 7).

Optionen		**Kriterien**		
		Niedrige Kopplung	Hohe Kohäsion	Realisierung einer Schichtenarchitektur
1.	**Überprüfungsauswahlansicht zur Anwendungsschicht REQuest**	**+**	**+**	**++**
2.	Überprüfungsauswahlansicht zur Modellschicht	-	-	--
3.	Überprüfungsauswahlansicht zur Kommunikations-/Speicherschicht	-	-	--
4.	Checkliste zur Anwendungsschicht REQuest	-	-	--
5.	**Checkliste zur Modellschicht**	**+**	**+**	**++**
6.	Checkliste zur Kommunikations-/Speicherschicht	-	-	--
7.	Prüfkriterium zur Anwendungsschicht REQuest	-	-	++
8.	**Prüfkriterium zur Modellschicht**	**+**	**+**	**--**
9.	Prüfkriterium zur Kommunikations-/Speicherschicht	-	-	++
10.	Problem zur Anwendungsschicht REQuest	-	-	--
11.	**Problem zur Modellschicht**	**+**	**+**	**++**
12.	Problem zur Kommunikations-/Speicherschicht	-	-	--

Bewertungen für die Berücksichtigung eines Kriteriums in der Gestaltungsalternative (Option):

++	Kriterium sehr gut umgesetzt	-	Kriterium weniger gut umgesetzt
+	Kriterium gut umgesetzt	--	Kriterium schlecht umgesetzt
N.A.	keine Angaben		

Rationaletabelle 7: Wie soll die Verteilung der im Analyseklassendiagramm identifizierten Klassen auf die Schichtenarchitektur von Sysiphus gestaltet werden?

6 Feinentwurf im TRAIN-Prozess

Ziel des Feinentwurfs ist, konkrete Vorgaben für eine Implementierung des Systems aus dem gesammelten Wissen und den spezifizierten Entscheidungen der vorangegangenen Phasen zu entwickeln.
Hierzu werden unter Einbeziehung der Architekturinfrastruktur die Implementierungsziele (Kapitel 6.1) festgelegt. Daraufhin wird das Analysemodell in Form des Analyseklassendiagramms (vgl. Kapitel 4.8.2.1) zu einem Entwurfsmodell (Kapitel 6.2) – repräsentiert durch das Entwurfsklassendiagramm – weiterentwickelt. Dieses dient als Grundlage für die Implementierung und schließt die Feinentwurfsphase ab. Kapitel 6.4 behandelt das Beispiel.

6.1 Implementierungsziele

Die Implementierungsziele legen fest, welche Programmiersprachen zur Implementierung gewählt, welche Frameworks, Klassenbibliotheken oder Komponenten wieder verwendet bzw. weiterentwickelt und welche Middleware für welche Aufgaben herangezogen werden sollen. Hiermit konkretisieren sie die Überlegungen der Architekturdefinition insofern, als sie detaillierte Vorschläge für die Gestaltungen in der Implementierungsphase machen.

6.2 Entwurfsmodell

Das Ziel der Entwurfsmodellerstellung ist die Vorbereitung der Implementierung unter Berücksichtigung der Implementierungsziele. Hierzu wird das Analyseklassendiagramm durch Anpassung und Verfeinerung zum Entwurfsklassendiagramm entwickelt. Dies geschieht v. a. durch das Anreichern des Analysemodells mit mehr Struktur und Details:

- Listen der Attribute und Operationen werden vervollständigt,
- Spezifikationen der Operationen, Klassen, Paketen, Komponenten werden erstellt (die Spezifikation der Operationen, Klassen, Pakete und Komponenten erfolgt meist durch Text in natürlicher Sprache, oft bereits ausgerichtet auf eine (Teil-) Wiederverwendung als Kommentar im Kode, z. B. als Javadoc),
- Assoziationen und Aggregationen werden konsolidiert (z. B. durch Qualifikation und Ordnung),

- Mehrfachvererbung wird vermieden,
- Komponenten, Klassenbibliotheken, Frameworks werden einbezogen und
- Anpassungs- und Entkopplungsschichten (z. B. für Datenbankzugriff, CORBA, XML usw.) werden hinzugezogen.

In Tabelle 3 ist der konzeptionelle Rahmen der beiden Modelle gegenübergestellt: Werden im Analysemodell noch Fachgegenstände als Objekte betrachtet, so treten an deren Stelle im Entwurfsmodell Softwareeinheiten. Klassen, die Konzepte, Daten und Fachbegriffe aus der Analyse repräsentierten, werden zu Klassenschemata. Vererbungen aufgrund von Beziehungsstrukturen zwischen den Begriffen repräsentieren nun die Programmableitung. Auch der Betrachtungswinkel der Technologie ändert sich: Wurde im Analysemodell noch die Annahme einer perfekten Technologie gemacht, wird im Entwurfsmodell eine konkret vorhandene mit den für diese geltenden Implementierungszielen einbezogen. Ist die Entwicklung des Analysemodells noch projektspezifisch, so finden sich in den Entwurfmodellen Ähnlichkeiten zwischen den verwandten Projekten, die dann auch die Einbeziehung von Entwurfsmustern ermöglichen. Diese stellen eine gute Möglichkeit dar, von den Erfahrungen bei der Lösung für eine Menge ähnlicher Probleme zu profitieren.

	Analysemodell	**Entwurfsmodell**
Objekte	Fachgegenstände	Softwareeinheiten
Klassen	Fachbegriffe	Schemata
Vererbung	Begriffsstruktur	Programmableitung
	Annahme perfekter Technologie	Erfüllung konkreter Implementierungsziele
	Funktionale Essenz	Gesamtstruktur des Systems
	Meist projektspezifisch	Ähnlichkeiten zwischen verwandten Projekten
	➔ **Grobe Strukturskizze**	➔ **Genaue Strukturdefinition mehr Struktur & mehr Details**

Tabelle 3: Analyse- und Entwurfsmodell

6.3 Qualitätssicherungsmaßnahmen

In der Phase des Feinentwurfs werden Testvorbereitungen für Integrations- und Komponententests getroffen, sowie die Dokumente erneut Inspektionen unterzogen.

6.3.1 Inspektion

Inspektionen werden in dieser Phase ausgeführt, um zu prüfen, ob die Vorgaben der vorigen Phasen nachvollziehbar umgesetzt wurden und ob die Artefakte die notwendigen

Informationen enthalten, die für deren Weiterverarbeitung notwendig sind. Auch die Konsistenz der Artefakte untereinander sollte einer Überprüfung unterzogen werden.

6.3.2 Testen

6.3.2.1 Integrations- und Komponententests

Zu Beginn der Testaktivitäten steht die Testplanung. Gemäß der Qualitätsziele werden hier – wie schon beim Systemtestplan – die Testendekriterien, Teststrategien und -methoden, die Grundlagen der Tests sowie die benötigten Ressourcen festgelegt. Anfangs werden für die Planung der Komponententests das Analyseklassendiagramm und für die Integrationstests zusätzlich noch die Architekturdefinition hinzugezogen. Zur weiteren Konkretisierung der Pläne, beispielsweise zur Festlegung der Testobjekte oder der Testpriorisierung, muss jedoch das Entwurfsmodell weiter fortgeschritten sein. Abgeschlossen werden die Testaktivitäten auf dieser Ebene möglichst mit den Spezifikationen der Integrations- und Komponententests und gegebenenfalls schon mit einer Implementierung der Rahmenabläufe und der benötigten Testtreiber und Stubs. Dies ist jedoch nicht immer möglich, da z. B. zur Spezifikation eines White-Box Komponententests der Komponentenquellkode bereits zur Verfügung stehen muss.

6.4 Feinentwurf im Beispiel

In unserem Beispiel werden wir im Folgenden für die Phase des Feinentwurfs den Fokus auf die Entwicklung des Entwurfsklassendiagramms (aus dem Analyseklassendiagramm und der Architekturdefinition) richten (Kapitel 6.4.1). Daher werden wir beispielsweise die Aufstellung konkreter Implementierungsziele außer Acht lassen. Abschließend wird stellvertretend für die Qualitätssicherungsmaßnahmen dieser Phase die Erstellung des Komponententestplans und der Komponententestspezifikation vorgestellt (Kapitel 6.4.2).

6.4.1 Entwurfsmodell

Beginnen werden wir mit der Verfeinerung des bestehenden Analyseklassendiagramms aus Kapitel 4.9.1.1.5. Hierzu werden wir:

1. Die Klassenstruktur des Analyseklassendiagramms an die bestehende Klassenstruktur des Systems anpassen (Kapitel 6.4.1.1).
2. Die Klassenmitglieder identifizieren (Kapitel 6.4.1.2).
3. Das Erarbeitete konsolidieren (Kapitel 6.4.1.3).
4. Die Beschreibung der Komponenten erarbeiten (Kapitel 6.4.1.4).

6.4.1.1 Klassenstruktur anpassen

Für die Weiterentwicklung des Analyseklassendiagramms muss man erkennen, dass die aus den Domänendaten abgeleitete Klasse *Anforderungselement* in der bisherigen Umsetzung des Systems durch die Klasse *LinkableElement* repräsentiert ist. Diese bildet eine Oberklasse für die konkreten Anforderungselemente, wie beispielsweise Use Case oder Aktor.

Außerdem müssen die Bezeichner der Klassen nun endgültig festgelegt werden:

- `CheckSelectionIntf` (ehemals Überprüfungsauswahlansicht)
- `Problem`
- `Checklist` (ehemals Checkliste)
- `Checkcriteria` (ehemals Prüfkriterium).

6.4.1.2 Klassenmitglieder identifizieren

Eine Schwierigkeit besteht in der vollständigen Identifikation aller benötigten Klassenmitglieder. Zur Identifikation benötigter Operationen helfen vor allem Szenarien (vgl. Kapitel 4.6.1.3), da sie exemplarisch die Interaktion des Benutzers mit dem System aufzeigen. Deckt das Szenario alle möglichen Interaktionsabläufe und Ausnahmen ab, so

sind durch das Nachvollziehen des programmtechnischen Ablaufs anhand des Szenarios (und ggf. das Aufstellen eines zugehörigen Sequenzdiagramms) alle beteiligten Operationen und Klassen zu identifizieren. Zur Identifikation der Klassenattribute überlegen wir uns, auf welchen bzw. mit welchen Daten die Operationen arbeiten.

6.4.1.2.1 Operationen identifizieren

Unser Szenario (Szenario 1) werden wir für die Identifikation der Operationen heranziehen. Wir beginnen allerdings erst zu dem Zeitpunkt der Interaktionsfolge, an dem die Überprüfungsauswahlansicht angezeigt wird:

1. Wir benötigen eine Operation zum Anzeigen der Überprüfungsauswahlansicht. Diese soll kurz Informationen zu der Ansicht und ihrem Zweck geben und die Auswahlmöglichkeiten für die Prüfkriterien (mit Kurzbeschreibung) zur Verfügung stellen (→ neue Operation `showSelectionForm()`).
2. Die möglichen Prüfkriterien und deren Kurzbeschreibungen sollen durch eine Operation zugänglich gemacht werden (→ neue Operation `getPossibleChecks()`).
3. Die Auswahl der Prüfkriterien muss nach der Übermittlung an den Server von diesem ausgelesen und auf getätigte Auswahl überprüft, gegebenenfalls ein Hinweis auf nicht getätigte Wahl von Prüfkriterien ausgegeben und bei getätigter Auswahl die Inspektion durchgeführt werden. Eine neue Operation `startCheck()` soll diesen Ablauf steuern: die übermittelten Prüfkriterien auslesen (→ neue Operation `readCheckBoxes()`), je nach getätigter Auswahl von Prüfkriterien den Hinweis ausgeben (→ neue Operation `showHintNoChecks()`) bzw. die Inspektion durchführen (→ neue Operation `performCheck()`).
4. Die Operation `performCheck()` soll auf Basis der Benutzerauswahl die gewählten Kriterien überprüfen. Für jedes Kriterium soll es eine neue Operation geben, die als Bezeichner das Präfix „check“ erhält und weiter eine eindeutige Bezeichnung für das Prüfkriterium.

   Beispiel: `checkUseCase_SystemStep()` zur Überprüfung der Use Case-Systemschritt Beziehung.
5. Wird in einer der `check…()`-Operationen ein Problem gefunden, so wird eine Instanz der Klasse `Problem` erzeugt, die sich um das Erstellen einer problemspezifischen, dem Problemelement zuzuordnenden Frage kümmert (→ neue Operation `createIssue()`).

6.4.1.2.2 Attribute identifizieren

1. Zentrales Element ist die Klasse *Prüfkriterium* aus dem Analyseklassendiagramm. Für den Umgang mit dieser überlegen wir uns folgende Möglichkeiten:
   a. Eine eigene Klasse `Checkcriteria`, die dann auch für den jeweiligen Prüfpunkt die Checkoperation enthält.
   b. Die Prüfkriterien und auch die zugehörigen Checkoperationen in die Klasse `Checklist`, der dadurch alle ausführbaren Kriterien und Checks unmittelbar bekannt sind.
   c. Die Prüfkriterien in die Klasse `CheckSelectionIntf`, die diese auch für ihre Anzeige benötigt.

   Nach Abwägung der Alternativen (vgl. Kapitel 6.4.3, Rationaletabelle 8) fällt die Entscheidung für Option b). Zu deren Gestaltung werden wir in der Klasse `Checklist` statische, finale, eindimensionale Array-Attribute vom Typ `String` für jedes Prüfkriterium definieren. Der erste Eintrag des Arrays soll das Prüfkriterium eindeutig identifizieren und der zweite eine Beschreibung zur Information des Benutzers liefern. Für eine statische Deklaration entscheiden wir uns, da dieses Attribut für alle möglichen Instanzen dasselbe ist, für eine finale, da es nicht änderbar sein wird. Beispiel:

```
private static final String[] ELEMENT_DESCRIPTION_REQ
      = {"ElementDescription_REQ", "Checks if each element in the
      Requirements Section has a description."};

private static final String[] ACTOR_USERTASK
      = {"Actor_UserTask", "Checks if each actor initiates/paticipates in
      at least one User Task."};
```

2. Um diese Liste der Prüfkriterien zu gliedern, werden wir diese den Kapiteln „Requirements“, „Specification“ und „Example“ zuordnen. Dies entspricht der Anforderungsdokumentenstruktur in Sysiphus. Für jede dieser Gliederungseinheiten fügen wir eine weitere Variable der Klasse `Checklist` hinzu, die den Namen für die Einheit, jedoch keine Beschreibung enthält. Beispiel:

```
private static final String[] LIST_HAEDER_REQ = {"Requirements", null};
```

3. Um nun letztlich die einzelnen Prüfkriterien den Gliederungseinheiten zuzuordnen, fügen wir der Klasse `Checklist` weitere Attribute hinzu, die jeweils die Prüfkriterien einer Einheit umfasst. Beispiel:

```
private static final String[][] REQ_LIST
      =    {LIST_HAEDER_REQ,    ELEMENT_DESCRIPTION_REQ,    ACTOR_USERTASK,
      USERTASK_ACTOR, QUALITY_USERTASK};
```

4. In Systemfunktionsbeschreibung 1 wurden die Titel für die Fragen spezifiziert, die auf Probleme hinweisen. Diese Hinweistexte für jeden (zu einem Prüfkriterium gehörenden) Problemtyp werden wir als statisches, finales Attribut der Klasse `Problem` zuweisen. Beispiel:

```
public static final String NO_DESCRIPTION
      = "Why does the following Element has no description? Element: ";
public static final String NO_USERTASK_FOR_ACTOR
      = "Why does the following Actor does not initiate/participate in an
      User Task? Actor: “;
```

6.4.1.3 Konsolidierung des Entwurfs[16]

Die Klassenattribute sind noch mit Sichtbarkeiten bzw. Parameter zu versehen. Aber auch über die Anwendung von Entwurfsmustern sollte nachgedacht werden. Exemplarisch seien einige Festlegungen aufgezeigt:

1. `showSelectionForm()`

public:	um Aufruf von außen zu ermöglichen
void:	da nur Ausgabe an den Client in dieser Operation
HttpServletRequest req:	zur Kommunikation mit dem Client
HttpServletResponse res:	zur Kommunikation mit dem Client
Project project:	benötigte projektspezifische Information

2. `performCheck()`

public:	um Aufruf von außen zu ermöglichen (beispielsweise durch die Operation `startCheck()`)
void:	da „Steuerungsfunktion“ und daher keine Rückgabe notwendig
String[] checksToDo:	die Bezeichner der zu absolvierenden Prüfkriterien
Project project:	benötigte projektspezifische Information

3. Wie alle Benutzungsschnittstellenklassen werden wir auch `CheckSelectionIntf` mittels des Entwurfsmusters Singleton realisieren. Hierzu

---

[16] Während wir hier den Entwurf erarbeitet haben, stellt der Lesende vermutlich fest, dass einige Entscheidungen nicht unbedingt objektorientiert getroffen wurden. So ist beispielsweise die Entscheidung, die Prüfkriterien als Attribute der Klasse Checklist zuzuordnen nicht objektorientiert gedacht (zumal sie im Analyseklassediagramm schon als Klasse identifiziert wurden). Vielmehr hätte man eine eigene Klasse für diese beibehalten bzw. eine Zuordnung der Checkoperation zu den Unterklassen von LinkableElement (die sich dann selbst untersuchen) in Betracht ziehen sollen. Bei der Beibehaltung einer eigenen Klasse für die Prüfkriterien hätte diese dann die Informationen (Daten) zu diesem Prüfkriterium (Kurzbeschreibung, Kategoriezuordnung, Fragetitel für das durch dieses Kriterium gefundenen Problems usw.) und die jeweilige

benötigen wir ein privates, statisches Klassenattribut `instance` vom Typ der Klasse, einen privaten Konstruktor und eine öffentliche, statische Operation `getInstance()`, die die einzige Instanz der Klasse zurückliefert.

4. Für die Klasse `Checklist` haben wir bereits gesagt, dass es eine Methode `performCheck()` gibt. Diese führt auf Basis der ausgelesenen Prüfkriterien die Inspektion der Elemente für ein spezielles Projekt durch. Um zu verhindern, dass die Inspektion mehrmals parallel ausgeführt wird, werden wir diese synchronisieren. Außerdem werden wir die Klasse `Checklist` ebenfalls als Singleton umsetzen.

6.4.1.4 Beschreibung der Komponenten

Nachdem wir nun die Mitglieder der Klassen identifiziert haben, muss die Paket-, Klassen-, Operations- und gegebenenfalls Attributbeschreibung erarbeitet werden. Diese werden im Folgenden ebenfalls nur exemplarisch aufgezeigt:

1. *req.im* (package)

   Dieses Package (**i**nspection **m**odel) beinhaltet die Benutzungsschnittstellenklassen, die in der Anwendung REQuest mit der Inspektionsaktivität in Beziehung stehen. In unserem Fall ist das nur die Klasse `CheckSelectionIntf`, die die Benutzungsschnittstelle für die Auswahl der Prüfkriterien generiert.

2. *model.im* (package)

   Dieses Package umfasst das Modell des Inspektionsprozesses. Im Beispiel bedeutet dies, die Zugehörigkeit der Klasse `Checklist` und `Problem`.

3. `CheckSelectionIntf`

   Diese Klasse ist für die Präsentation der mit dem Inspektionsprozess verbundenen Benutzungsschnittstelle (insbesondere der Überprüfungsauswahlansicht) verantwortlich. Des Weiteren nimmt sie die Auswahl des Benutzers entgegen und entscheidet über einen Teil der Prozesssteuerung.

4. `public void showSelectionForm(HttpServletRequest req, HttpServletResponse res, Project project)`

   Diese Operation übernimmt das Anzeigen der Prüfkriterien (Überprüfungsauswahlansicht) für die Inspektion. Die Parameter *req* bzw. *res*

Checkoperation beinhaltet. Die Entscheidungen bestanden jedoch bereits bei Beginn der Arbeit im Sysiphus-Projekt und wurden nicht mehr revidiert (vgl. Kapitel 3.1).

werden für die Kommunikation mit dem Client benötigt, *project* für den Erhalt von Projektinformationen.

5. `Checklist`

   Diese Klasse beinhaltet alle Informationen über die möglichen Prüfkriterien (Checks), d. h. deren Beschreibungen, Zuordnung zu den Rubriken sowie die Operationen zu deren Durchführung.

6. `public void checkUseCase_SystemStep(Project project)`

   Diese Operation untersucht die Use Case-Systemschritt-Beziehung für alle Use Cases des übergebenen Projekts, d. h. besitzt jeder Use Case, bzw. wenn nicht, einer seiner inkludierten Unter-Use Cases mindestens einen Systemschritt?

7. `Problem`

   Diese Klasse dient dazu, bei einem gefundenen Problem eine Frage zu erzeugen, die mit einem problem- und elementspezifischen Titel versehen und dem Element zugeordnet ist. Danach wird die Instanz der Klasse Problem dem Garbage Collector überlassen.

   Alle Problemtypbeschreibungen werden als finale, statische Variablen in dieser Klasse verwaltet.

8. `private void createIssue(LinkableElement element, String problemType)`

   Diese Operation ist dafür zuständig, die Erzeugung einer Frage für ein bestimmtes Problem (*problemType*) vorzunehmen und diese dann dem Problemelement (*element*) zuzuordnen. Dabei besteht die Möglichkeit, dass dieses Problem bereits bei einem früheren Inspektionsdurchlauf berichtet wurde und als Frage bereits vorhanden ist. Deshalb wird in der Operation wie folgt vorgegangen (vgl. Systemfunktionsbeschreibung 1):

   1. Es wird geschaut, ob diese Frage (*problemType*) bereits für das Element (*element*) vorhanden ist.
   2. Ist dies der Fall, so wird diese Frage wieder zur Diskussion gestellt.
   3. Ist die Frage zu diesem Problem (*problemType*) und Element (*element*) nicht vorhanden, so wird sie erzeugt und dem Element hinzugefügt.

Das erarbeitete, vollständige Entwurfsklassendiagramm ist in Abbildung 16 dargestellt.

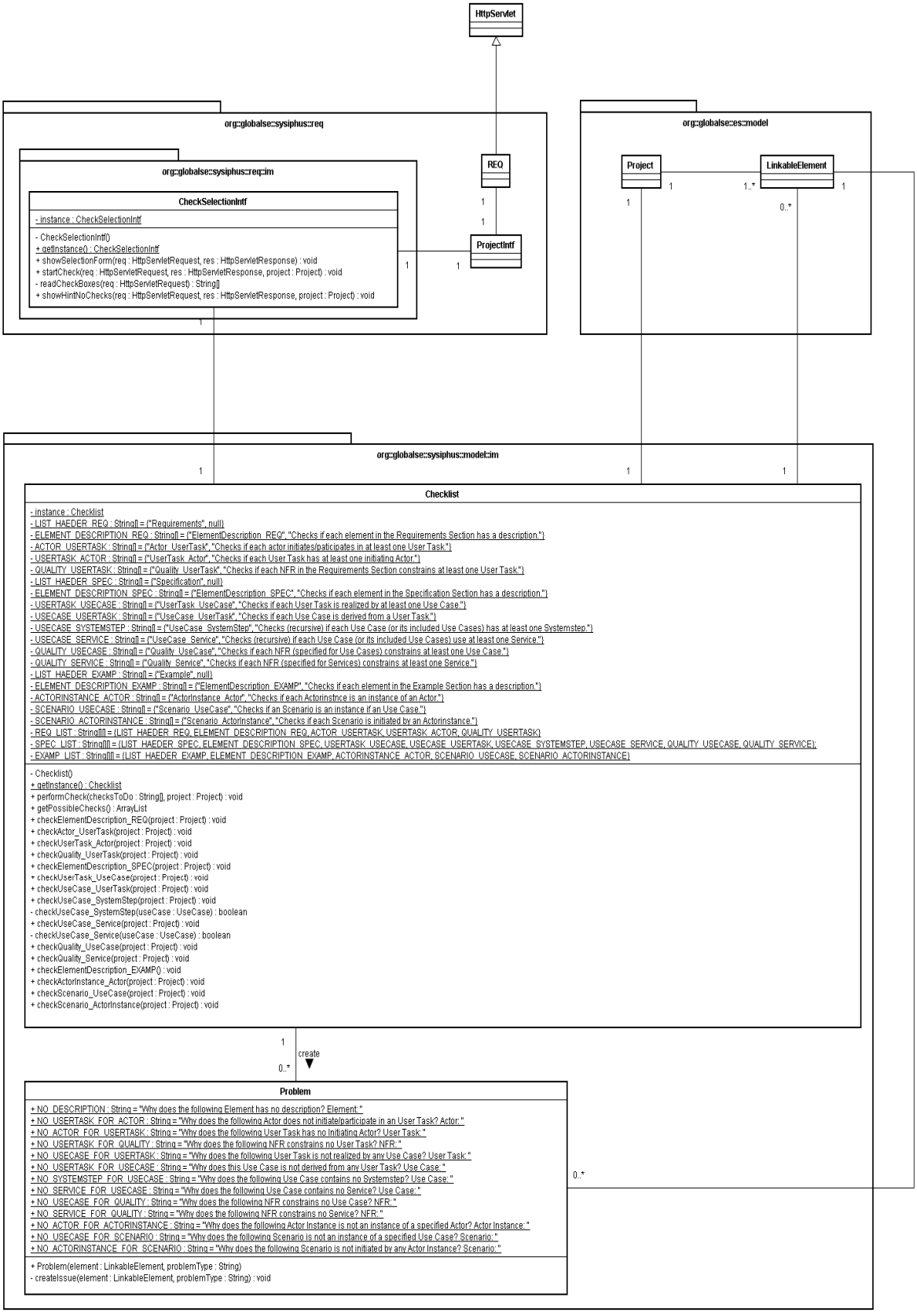


Abbildung 16: Entwurfsklassendiagramm

6.4.2 Qualitätssicherungsmaßnahmen

Wie bereits eingangs in Kapitel 6.4 erwähnt, werden wir stellvertretend für die Qualitätssicherung die Erarbeitung der Komponententestspezifikation vorstellen. Hierzu benötigen wir allerdings einen Testplan für die Komponententests.

6.4.2.1 Testen

6.4.2.1.1 Komponententestplan

Der Komponententestplan beinhaltet in unserem Beispiel folgende Festlegungen:

Testmethoden:	Black-Box Test, White-Box Test
Testendekriterien:	Testobjekte wurden mit mindestens einem Repräsentanten aus jeder Äquivalenzklasse getestet, eine vollständige Zweigüberdeckung wurde erreicht
Testobjekte:	Alle Konstruktoren und Operationen der neuen Klassen
Testgrundlage:	Objektspezifikation (Black-Box Test) bzw. Programmkode und Kontrollflussgraph der Objekte (White-Box Test)
Testressourcen:	JUnit

6.4.2.1.2 Komponententestspezifikation (Black-Box Test)

Bei den White-Box Tests ist es erst nach der Implementierung der Testobjekte möglich, die Testspezifikation zu schreiben. Deshalb werden wir an dieser Stelle erst einmal für die Operation `createIssue()` der Klasse `Problem` die Black-Box Komponententestspezifikation ausarbeiten.[17] Hierzu identifizieren wir die Äquivalenzklassen:

Die Operation `createIssue()` hat zwei Übergabeparameter,

- einmal das Element, für das die Frage erzeugt werden soll und
- zum anderen den Problemtyp, der den Hinweistext für die Frage bestimmt.

Für das Element kommt

1. (eine Referenz auf) ein Element oder

---

[17] Es kann hier diskutiert werden, ob für die Operation `createIssue()` ein Komponententest spezifiziert werden kann, da diese Operation keinen Rückgabewert im Sinne einer return-Anweisung besitzt. In unserem Fall ist der „Rückgabewert“ eine auf ein Problem hinweisende Frage bzw. keine neue Frage, wenn ungültige Parameter übergeben wurden. Diese „Rückgabewerte“ werden aber durch Aufruf anderer Komponenten realisiert. Dies ist der Grund, dass man bei alleiniger Betrachtung der zu testenden Komponente diese „Rückgabewerte“ nicht überprüfen kann. Um diese „Rückgabewerte“ zu testen, benötigt man Stubs von den beteiligten Komponenten. Das Zusammenspiel mehrerer Komponenten ist aber nach dem Buch „Basiswissen Softwaretest“ von Andreas Spillner (S.43) Gegenstand eines Integrationstests. Dessen sind wir uns bewusst, werden diesen Test aber zur Vorstellung der Methode der Äquivalenzklassenbildung weiterentwickeln.

2. der Wert *null* als Übergabe in Frage.

Für einen Problemtyp sind

1. als gültige Äquivalenzklasse ein in der Klasse `Problem` als statisch, finale Variable deklarierter Problemtyp und
2. als ungültige Äquivalenzklasse irgendein String (jedoch keiner, der in der Klasse `Problem` als statisch, finale Variable deklarierten) oder
3. der Wert *null* denkbar.

Diese Werte bilden also die Äquivalenzklassen der Operation und sind in Tabelle 4 nochmals zusammengefasst.

Parameter	Äquivalenzklasse	Repräsentant
LinkableElement element	gÄK1: nicht *null* uAK1: *null*	*Wahl findet später statt*
String problemType	gÄK1: deklarierter Problemtyp der Klasse Problem uÄK1: String, nicht deklarierter Problemtyp der Klasse Problem uÄK2: *null*	*Wahl findet später statt*

gÄK: Äquivalenzklasse mit gültigen Repräsentanten

gÄK: Äquivalenzklasse mit ungültigen Repräsentanten

Tabelle 4: Äquivalenzklassen der Operation `createIssue()`

Zur Testdurchführung muss man die Äquivalenzklassen der beiden Parameter der Operation kombinieren. Hierdurch ergeben sich die Testfälle für den Black Box-Test, die in Tabelle 5 aufgeführt sind. Die Tabelle enthält außerdem das erwartete Ergebnis jedes Testfalles.

Testfall	Äquivalenzklassenkombinationen		Parameter		Erwartetes Ergebnis
	Parameter 1 (element)	Parameter 2 (problemType)	Parameter 1 (element)	Parameter 2 (problemType)	
1	gÄK1	gÄK1			Frage zu Element
2	uÄK1	gÄK1			keine neue Frage
3	gÄK1	uÄK1	Wahl findet später statt		keine neue Frage
4	gÄK1	uÄK2			keine neue Frage

gÄK: Äquivalenzklasse mit gültigen Repräsentanten

gÄK: Äquivalenzklasse mit ungültigen Repräsentanten

Tabelle 5: Testfälle der Operation `createIssue()` bei Äquivalenzklassenbildung

Für die Testfallspezifikation muss man aus diesen Äquivalenzklassen noch konkrete Repräsentanten wählen. Diese werden wir aber erst nach Betrachtung des Kontrollflussgraphen der White-Box Testspezifikation wählen (vgl. Kapitel 7.2.2.1.2), um durch geeignete Repräsentantenwahl eventuell mit diesen vier Testfällen gleichzeitig auch

die angestrebte Zweigüberdeckung bei den White-Box Tests erlangen zu können. So ist es möglich, mit geringer Anzahl von Testfällen eine maximale Testabdeckung zu erreichen.

6.4.3 Rationale

Das Rationale zur Gestaltung der Prüfkriterien (Checkcriteria) im Feinentwurf (vgl. Kapitel 6.4.1.2.2) wird in Rationaletabelle 8 dokumentiert. Nach objektorientierten Kriterien hätte man sich für Option 1 entscheiden müssen. Option 2 wurde jedoch gewählt, um Änderungen an den Prüfkriterien (und deren Operationen) nur in einer Klasse vornehmen zu müssen.[18]

Optionen		**Kriterien**			
		Niedrige Kopplung	Hohe Kohäsion	Realisierung einer Schichtenarchitektur	Kapselung
1.	Eine eigene Klasse `Checkcriteria`, die dann auch für den jeweiligen Prüfpunkt die Checkoperation enthält.	**-**	**N.A.**	**+**	**++**
2.	**Die Prüfkriterien und auch die zugehörigen Checkoperationen in die Klasse `Checklist`, der dadurch alle ausführbaren Kriterien und Checks unmittelbar bekannt sind**	**+**	**+**	**+**	**N.A.**
3.	Die Prüfkriterien in die Klasse `CheckSelectionIntf`, die diese auch für ihre Anzeige benötigt	**-**	**N.A.**	**-**	**N.A.**

Bewertungen für die Berücksichtigung eines Kriteriums in der Gestaltungsalternative (Option):

++	Kriterium sehr gut umgesetzt	-	Kriterium weniger gut umgesetzt
+	Kriterium gut umgesetzt	--	Kriterium schlecht umgesetzt
N.A.	keine Angaben		

Rationaletabelle 8: Wie werden die Prüfkriterien (Checkcriteria) im Feinentwurf gestaltet?

[18] Diese Entscheidung bestand bereits bei Beginn der Arbeit im Sysiphus-Projekt und wurde nicht mehr revidiert (vgl. Kapitel 3.1).

7 Implementierung im TRAIN-Prozess

Ziel der Implementierungsphase ist, den Entwurf in ablauffähigen, korrekten und dem Anspruch der (Qualitäts-) Anforderungen gerecht werdenden Programmkode umzusetzen. Dabei sollten idealerweise nur noch die Umsetzungen der Vorgaben des Feinentwurf stattfinden und keine größeren Entscheidungen mehr getroffen werden müssen. Eine geeignete Werkzeugunterstützung in der Implementierungsphase hilft systematischer und effizienter, den Implementierungsprozess zu bewältigen.
Kapitel 7.1 stellt die Qualitätssicherungsmaßnahmen während der Implementierungsphase vor, und Kapitel 7.2 behandelt die Umsetzung dieser Phase anhand des Beispiels.

7.1 Qualitätssicherungsmaßnahmen

Bei der Umsetzung des Entwurfsmodells wird viel Wert auf die Qualität und Korrektheit des erzeugten Programmkodes gelegt. Ersteres beinhaltet die

- Funktionalität,
- Verständlichkeit und Wartbarkeit,
- Effizienz und
- Eleganz

des Kodes und kann durch Konventionen (Namen, Bezeichner, Formatierung), Standards, Dokumentationen, aber auch Vergleiche bzw. Analysen (Effizienz), Pair Programming und Tests (Funktionalität) erreicht werden. Die Korrektheit wird vor allem durch Tests gewährleistet. Hierfür findet die Implementierung der Tests statt, die in den vorangegangenen Phasen spezifiziert wurden. Daraufhin führt man den Debuggingzyklus (Kapitel 2.3.5) durch.
Die Implementierungs- und Testaktivitäten (Realisierungsphase) finden in umgekehrter Reihenfolge korrespondierend zur Planungsphase statt (vgl. Abbildung 2):

- aus den einzelnen codierten Komponenten werden ausführbare Komponenten, deren Anforderungskonformität durch Komponententests verifiziert werden (Pendant in der Planungsphase: Feinentwurf),
- durch Zusammensetzen der Komponenten und Durchführung der Integrationstests entsteht ein ausführbares System (Pendant: Feinentwurf),
- durch weitere Evaluierung in der Praxis und Durchführung der Systemtests entsteht ein benutzbares System (Pendant: Requirements Engineering/Interaktionsebene).

7.2 Implementierung im Beispiel

In unserem Beispiel werden wir nun auf die Implementierung der drei neuen Klassen eingehen (Kapitel 7.2.1), sowie – exemplarisch für die Qualitätssicherungsmaßnahmen – die Identifizierung der Testfälle für den White-Box Komponententest der Operation `createIssue()` vorstellen (Kapitel 7.2.2.1.1). Weiterhin werden wir die Zusammenführung der White-Box Testfälle mit den Black-Box Testfällen (aus Kapitel 6.4.2.1.2) der Operation aufzeigen (Kapitel 7.2.2.1.2).

7.2.1 Implementierungen der entwickelten Klassen

Für die Implementierung werden das Modellierungswerkzeug JUDE und Eclipse als Entwicklungsumgebung verwendet. Neben der Modellierung wird JUDE auch zur automatischen Generierung des Kodeskeletts aus unserem Entwurfsmodell (Abbildung 16) herangezogen und in Eclipse das Testwerkzeug JUnit integriert.
Ausschnitte des Quellkodes der implementierten Klassen `CheckSelectionIntf`, `Checklist` und `Problem` sind in Anhang D vorhanden.[19]

7.2.2 Qualitätssicherungsmaßnahmen

Zur Qualitätssicherung bei den Codierungsaktivitäten wird sich großenteils an die Java Code Conventions[20] gehalten. Außerdem wird vereinbart, dass bis auf die Klassenattribute alle Operationen, Klassen und Packages eine JavaDoc-Kommentierung[21] erhalten sollen.
Um die Funktionalität des Programmkodes zu verifizieren, werden die Tests ausgeführt, protokolliert und daraufhin die Fehler im Programmkode behoben.

7.2.2.1 Testen

7.2.2.1.1 Komponententestspezifikation (White-Box Test)

Da wir uns bei der Aufstellung des Komponententestplans auch für einen White-Box Test als Testmethode entschieden haben, können wir die Testspezifikation erst nach der Implementierung der Testobjekte entwickeln. Stellvertretend für alle weiteren Komponenten werden wir die Testspezifikation für die Operation `createIssue()` der

[19] Die gewählten Ausschnitte beziehen sich auf Aspekte (Operationen, Variablen), die für das Beispiel der Arbeit, vor allem in der Feinentwurfsphase, erwähnt wurden.
[20] http://java.sun.com/docs/codeconv/
[21] http://java.sun.com/j2se/javadoc/writingdoccomments/

Klasse `Problem` vorstellen. Wie bereits in Kapitel 6.4.2.1.2 erwähnt, werden wir versuchen, diese so zu entwickeln, dass wir gemeinsame Testfälle für den Black- und White-Box Test erhalten. Hierzu erstellen wir erst einmal den Kontrollflussgraphen der Operation `createIssue()` (Abbildung 17).

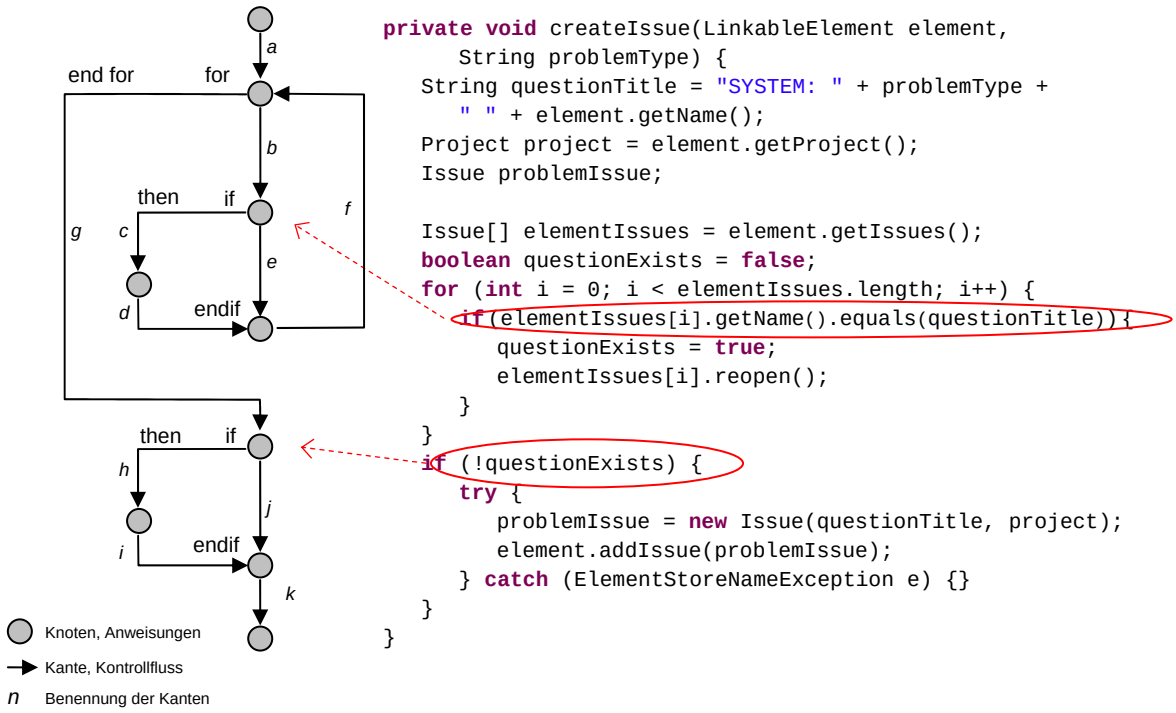


Abbildung 17: Kontrollflussgraph der Operation `createIssue()`

Aus dem Kontrollflussgraph ist ersichtlich, dass zwei Testfälle für den White-Box Test benötigt werden, um eine Zweigüberdeckung in dieser Operation zu erhalten:

1. Wird ein Element-Problemtyp „Paar“ übergeben, zu dem noch keine auf das Problem hinweisende Frage existiert, so werden mindestens die Zweige a, (g), h, i, k abgedeckt (Testfall 1).
2. Wird ein Element und ein Problemtyp übergeben und es existiert zu diesem „Paar“ bereits eine auf das Problem hinweisende Frage, so werden die Zweige a, b, c, d, (e), (f), (g), (j), k abgedeckt (Testfall 2).

Um diese beiden Testfälle umzusetzen, müssen nun konkrete Werte für die Übergabeparameter gewählt werden. Wir benötigen einmal ein Element-Problemtyp „Paar“, für das noch keine korrespondierende Problemfrage existiert (Testfall 1), und ein „Paar“, für das eine korrespondierende Problemfrage existiert (Testfall 2). Das Element

muss als Stub implementiert werden, für den Problemtyp werden wir `NO_DESCRIPTION` (vgl. Abbildung 16) wählen.

7.2.2.1.2 Kombination von Black- und White-Box Tests

Die beiden White-Box Testfälle entsprechen in Hinblick auf die Repräsentantenwahl Testfall 1 der Black-Box Tests (vgl. Tabelle 5). Somit erreicht man bei einer geschickten Kombination mit vier Testfällen einen vollständigen Black-Box Test und einen zweigüberdeckenden White-Box Test der Operation `createIssue()`.

Für die Testimplementierung werden allerdings noch einige Stubs benötigt:

- konkretes Element der Klasse `LinkableElement` zur Übergabe an die Operation `createIssue()`
- Stub der Klasse `Issue`, da man die Fragetitel vergleichen und eine Frage erzeugen muss

Die Testfälle für den Komponententest der Komponente `createIssue()` sind für die Implementierung folgende:

Testfall	ursprüngliche Testfälle		Äquivalenzklassen-kombinationen		Parameter		Erwartetes Ergebnis
	White-Box	Black-Box	Parameter 1 (element)	Parameter 2 (problemType)	Parameter 1 (element)	Parameter 2 (problemType)	
1	1, 2	1	gÄK1	gÄK1	*LinkableElement-Stub*	`NO_DESCRIPTION`	Frage zu Element
2		2	uÄK1	gÄK1	null	`NO_DESCRIPTION`	keine neue Frage
3		3	gÄK1	uÄK1	*LinkableElement-Stub*	`NON_SENSE`	keine neue Frage
4		4	gÄK1	uÄK2	*LinkableElement-Stub*	null	keine neue Frage

gÄK: Äquivalenzklasse mit gültigen Repräsentanten

gÄK: Äquivalenzklasse mit ungültigen Repräsentanten

Tabelle 6: Testfälle der Operation `createIssue()` bei Kombination von Black- und White-Box Tests

7.2.2.1.3 Weitere Tests

Um die Änderungsaufgabe abzuschließen, müssten von uns alle Komponententests, Integrationstests und Systemtests spezifiziert, implementiert und dem Debuggingzyklus zugeführt werden. Nach Abschluss dieser Aktivitäten ist das System dann zur Übergabe an den Auftraggeber bereit.

A Prozessübersichtstabelle

Management	Phasen	Entscheidungsebenen	Dokumentation: Artefakte IN	Dokumentation: Artefakte OUT	Dokumentation: Rationale	QS-Tätigkeiten
Projektmanagement Änderungsmanagement Konfigurationsmanagement	Analyse, Wissenserwerb, Konfliktlösung					
	Requirements-engineering	Aufgaben-ebene	- Erhobene Defizite des bisherigen Geschäftsprozesses (z. B. auch Probleme mit externen Systemen) - Geschäftsprozessmodelle - Protokolle	- Rollenbeschreibungen - Aufgabenbeschreibungen - Glossareinträge - Qualitätskriterien (NFR) von Aufgaben	Begründung der Aufgabenwahl und der NFRs	
		QS	- Softwaredokumentation - Lesetechnik	- Problem-/Fehler-/Korrekturliste - Korrigierte SW-Dokumentation		**Inspektion**
		Domänen-ebene	- Rollenbeschreibungen - Aufgabenbeschreibungen - NFRs	- Domänendaten-Diagramm - Datenbeschreibung - Systemverantwortlichkeiten-übersicht (in Form eines UC Diagramms) - Aktivitätsdiagramme für IST und SOLL - Ist-, Sollbeschreibung - Glossareinträge - Qualitätskriterien (NFR) für die Domäne - Qualitätskriterien (NFR) an die Architektur - Globale Qualitätskriterien (NFR) für Funktionale Anforderungen	Begründung des SOLL und der NFRs (aber auch aller weiteren Artefakten) aus den Artefakten und v. a. den NFRs der Aufgaben-ebene	
		QS	- Softwaredokumentation - Lesetechnik - Systemverantwortlichkeiten - NFRs	- Problem-/Fehler-/Korrekturliste - Korrigierte SW-Dokumentation - QS-Plan (z. B. was, wie intensiv, Priorisierung) - Systemtestplan	Begründung des QS- und Testplans	**Inspektion** **QS-/Testplan-erstellung**

Management	Phasen		Entscheidungsebenen	Dokumentation			QS-Tätigkeiten
				Artefakte		Rationale	
				IN	OUT		
Projektmanagement Änderungsmanagement Konfigurationsmanagement	**Analyse, Wissenserwerb, Konfliktlösung**	**Requirements-engineering**	Interaktions-ebene	- Systemverantwortlichkeiten-übersicht - Sollbeschreibung - Aktivitätsdiagramme (SOLL) - NFRs - Systemverantwortlichkeiten-beschreibungen (d. h. Aufgabenbeschreibungen, die zu Systemverantwortlichkeiten geworden sind) - Domänendaten	- Use Case Beschreibungen - Systemfunktionsbeschreibungen - Nutzungsdiagramm - Szenarien - Globale Qualitätskriterien (NFR) für Funktionale Anforderungen - Qualitätskriterien (NFR) für Use Cases - Qualitätskriterien (NFR) für Systemfunktionen - Qualitätskriterien (NFR) an die Architektur - UI-Struktur-Diagramm - Interaktionsdaten-Diagramm (Verfeinertes Datendiagramm)	Begründung der Artefakte aus den Artefakten und v. a. den NFRs der Domänenebene	
			QS	- Softwaredokumentation - Lesetechnik	- Problem-/Fehler-/Korrekturliste - Korrigierte SW-Dokumentation		**Inspektion**
				- Systemtestplan - Funktionale und Nicht-funktionale Anforderungen - Szenarien	- Systemtestspezifikation - Erstimplementierung (d. h. Rahmenablauf in der Testumgebung implementieren)	Begründung der Testspezifikation	**Testspezifikations-erstellung**
				- Use Case Beschreibungen - UI-Struktur-Diagramm	- Usabilitytestplan - Usabilitytestspezifikation - Prototypes	Begründung des Testplans und der Testspezifikation	**Testplan-erstellung/ -spezifikation Prototyping**

Management	Phasen		Entscheidungsebenen		Dokumentation: Artefakte IN	Dokumentation: Artefakte OUT	Dokumentation: Rationale	QS-Tätigkeiten
Projektmanagement Änderungsmanagement Konfigurationsmanagement	**Analyse, Wissenserwerb, Konfliktlösung**	**Requirements-engineering**	System-ebene	Kern	- ERD, UC, Systemfunktions-beschreibungen - Szenarien - NFRs	- Analyseklassendiagramm (AKD) - Sequenzdiagramme - NFRs	Begründung der Artefakte aus den Artefakten und v. a. den NFRs der Interaktionsebene	
				GUI		- Dialog Text - Dialog Zustandsdiagramm - Hilfsfunktionen		
			QS		- Softwaredokumentation - Lesetechnik	- Problem-/Fehler-/Korrekturliste - Korrigierte SW-Dokumentation		**Inspektion**

Management	Phasen		Entscheidungsebenen	Dokumentation: Artefakte IN	Dokumentation: Artefakte OUT	Dokumentation: Rationale	QS-Tätigkeiten
Projektmanagement Änderungsmanagement Konfigurationsmanagement	**Analyse, Wissenserwerb, Konfliktlösung**	**Architektur-definition**		- Soll-Prozess - Analyseklassendiagramm - (Architektur-) NFRs - Ressourcen - Externe Systeme - Architekturmuster	- Entwurfziele - Konfigurationsdiagramm - Komponentendiagramm/ Blockdiagramm - Implementierungsdiagramm - Verteilungsdiagramm - Einsatzdiagramm - Betriebskonzept	Begründung der Artefakte (Architekturmuster, Betriebskonzept usw.) aus den Architektur NFRs	
			QS	- Softwaredokumentation - Lesetechnik	- Problem-/Fehler-/Korrekturliste - Korrigierte SW-Dokumentation		**Inspektion**

Management	Phasen		Entscheidungsebenen	Dokumentation			QS-Tätigkeiten
				Artefakte		Rationale	
				IN	OUT		
Projektmanagement Änderungsmanagement Konfigurationsmanagement	**Analyse, Wissenserwerb, Konfliktlösung**	**Feinentwurf**	QS	- QS-Plan - Architekturdefinition - Analyseklassendiagramm	- Integrationstestplan - Komponententestplan	Begründung des Testplans	**Testplan-erstellung**
				- Analyseklassendiagramm - Architekturdefinition - Entwurfsmuster	- Entwurfklassendiagramm (EKD) - Package-, Klassen-, Operationsbeschreibungen	Begründung des Übergangs von AKD zu EKD Begründung der Implementierungs-vorschläge	
			QS	- Softwaredokumentation - Lesetechnik - Integrationstestplan - Sequenzdiagramme - Package-, Klassen-, Operationsbeschreibungen - Entwurfsklassen-Diagramm - Komponententestplan - Systemfunktionsbeschreibung, Package-, Klassen-, Operationsbeschreibungen und Operationssignatur aus Entwurfsklassen-Diagramm	- Problem-/Fehler-/Korrekturliste - Korrigierte SW-Dokumentation - Integrationstestspezifikation - Erstimplementierung (d. h. Rahmenablauf in der Testumgebung implementieren) - Komponententestspezifikation - Erstimplementierung (d. h. Rahmenablauf in der Testumgebung implementieren)	 Begründung der Testspezifikation Begründung der Testspezifikation	**Inspektion** **Testspezifi-kation** **Testimple-mentierung** **Testspezifi-kation** **Testimple-mentierung**

Management	Phasen		Entscheidungsebenen	Dokumentation			QS-Tätigkeiten
				Artefakte		Rationale	
				IN	OUT		
Projektmanagement Änderungsmanagement Konfigurationsmanagement	Analyse, Wissenserwerb, Konfliktlösung	**Implementierung**		- Entwurfsklassendiagramm - Package-, Klassen-, Operationsbeschreibungen - Implementierungsmuster	- Klassencode - Teilsystemcode - Systemcode	Begründung der Implementierung	
			QS	- Komponententestspezifikation - Erstimplementierung des Komponententests	- Komponententestcode	Begründung der Testimplementierungen	**Testimplementierung**
				- Klassencode - Komponententestcode	- Fehlerprotokoll		**Testdurchführung**
				- Fehlerprotokoll	- Fehlerursache - Geänderter Operations-/ Klassencode		**Debugging/ Ändern des Codes**
				- Integrationstestspezifikation - Erstimplementierung des Integrationstests	- Integrationstetcode		**Testimplementierung**
				- Teilsystemcode - Integrationstestcode	- Fehlerprotokoll		**Testdurchführung**
				- Fehlerprotokoll	- Fehlerursache - Geänderter Teilsystemcode		**Debugging/ Ändern des Codes**
				- Systemtestspezifikation - Erstimplementierung des Systemtests	- Systemtestcode		**Testimplementierung**
				- Systemcode - Systemtestcode	- Fehlerprotokoll		**Testdurchführung**
				- Fehlerprotokoll	- Fehlerursache - Geänderter Systemcode		**Debugging/ Ändern des Codes**

B Beschreibungstemplates[22] des TRAIN-Prozesses

Rolle (Actor)

Rollenbeschreibung (Actor)	
Name:	*Kurzbezeichnung.*
Verantwortlichkeiten:	*Kurzbeschreibung der Aufgabenbereiche der Rolle.*
Erfolgskriterien:	*Welche Bedingungen müssen erfüllt sein, damit die Rolle seine Aufgaben erfolgreich abschließen kann?*
Aufgaben:	*Welche spezifischen Aufgaben hat diese Rolle?*
Kommunikationspartner:	*Mit welchen anderen Rollen arbeitet diese Rolle zusammen, damit sie ihre Aufgaben erfüllen kann?*
Innovationsgrad:	*Ist diese Rolle schon in den Ist-Prozessen etabliert (Hoher Innovationsgrad, wenn nicht vorhanden)?*
Rollenprofil	
Wissen/Erfahrung/Fähigkeiten bzgl.	
Aufgaben	*Wissen/Erfahrung/Fähigkeiten der Rolle in Bezug auf die von ihr zu erfüllenden Aufgaben.*
Softwaresystem	*Wissen/Erfahrung/Fähigkeiten der Rolle in Bezug auf den Umgang mit Softwaresystemen im Allgemeinen.*

Aufgabe (User Task)

Aufgabenbeschreibung (User Task)	
Name:	*Kurzbezeichnung der Aufgabe.*
Verantwortliche Rolle (Initiating Actor):	*Welche Rolle löst diese Aufgabe aus?*
Beteiligte Rollen (Participating Actors):	*Welche Rollen sind weiterhin als Hilfe beteiligt?*
Aufgabenbewertung	
Ziel (Goal):	*Was ist das Ziel dieser Aufgabe.*
Eingriffsmöglichkeiten:	*Entscheidungsfreiräume bei der Durchführung der Aufgabe*
Ursachen:	*Wofür/Warum wird diese Aufgabe benötigt?*
Priorität:	*Wie wichtig ist diese Aufgabe?*
Aufgabendurchführung	
Durchführungsprofil (Häufigkeit, Kontinuität, Komplexität):	*Wie oft muss diese Aufgabe ausgeführt werden?* *Wie ist der Ablauf (Unterbrechungen Ja/Nein)? Wie komplex ist die Aufgabe?*
Ausgangssituation (Vorbedingung):	*Unter welchen Bedingungen macht die Durchführung dieser Aufgabe Sinn?*
Info-In:	*Welche Informationen fließen in die Aufgabenlösung mit ein?*
Info-Out:	*Was kommt am Ende dabei raus?*
Ressourcen (z. B. Arbeitsmittel etc.):	*Material/SW-Systeme…*

[22] Beschreibung des Templateinhalts von Lars Borner (http://www-swe.informatik.uni-heidelberg.de/people/borner.shtml) (27.01.2005).

Interaktion (Use Case)

Interaktionsbeschreibung (Use Case)		
Name:	*Kurzbezeichnung.*	
Verantwortliche Rolle (Initiating Actor):	*Durchführender des Use Cases (Rolle oder externe Soft-/Hardware).*	
Beteiligte Rollen (Participating Actors):	*Welche Rollen werden weiterhin benötigt, um Ziel des Use Cases zu erreichen.*	
Ziel (Goal):	*Was soll – aus Sicht des Aktors – durch diesen Use Case erreicht werden?*	
Vorbedingung (Preconditions):	*Zustand des Systems und der Umgebung aus Sicht des Aktors. Die Voraussetzung dafür ist, dass die Zielerreichung möglich ist.*	
Beschreibung (Flow Of Events):	**Aktor**	**System**
	Folge von Interaktionen zwischen Aktor und System, beschreibt Normalfall und kennzeichnet Ausnahmefälle.	
Ausnahmefälle (Exceptions):	*Folge von Interaktionen zwischen Aktor und System im Ausnahmefall*	
Nachbedingungen (Postconditions):	*Zustand des System aus Sicht des Aktors nach erfolgreicher Beendigung des Use Cases (ohne Ausnahmefälle).*	
Regeln (Rules):	*Komplexe funktionale und kausale Zusammenhänge, die das Verhalten des Aktors oder Systems genauer erklären.*	
Qualitätsanforderung (Quality Constrains):	*Übergreifende Eigenschaften, die sich auf die Gestaltung des Use Cases beziehen.*	
Benutzte Funktionen (Used Services):	*Systemfunktionen, die vom Use Case betroffen sind (von diesem genutzt werden).*	

Systemfunktion (Service)

Systemfunktionsbeschreibung (Service)	
Name:	*Kurzbezeichnung.*
Beschreibung (Description):	*Wie soll das Ergebnis normalerweise „berechnet“ werden bzw. wie ist der grobe Ablauf der Funktion?*
Eingangsdaten (Inputs):	*Welche Daten verarbeitet die Funktion?*
Ausgangsdaten (Outputs):	*Welche Daten erzeugt oder verändert die Funktion?*
Ausnahmefälle (Exceptions):	*Welche Ausnahmen gibt es? Was soll dann passieren?*
Regeln (Rules):	*Komplexe funktionale oder kausale Zusammenhänge bei der Berechnung.*
Vorbedingungen (Preconditions):	*Zustand von System und Umgebung aus Sicht des Aktors bevor die Funktion ausgeführt wird.*
Nachbedingungen (Postconditions):	*Zustand des Systems aus Sicht des Aktors nachdem die Funktion erfolgreich beendet ist.*
Qualitätsanforderung (Quality Constrains):	*Übergreifende Eigenschaften, die sich auf die Gestaltung der Systemfunktion beziehen.*
Use Cases:	*Use Cases in denen die Funktion auftaucht.*

Szenario (Scenario)

Szenario (Scenario)	
Name:	*Kurzbezeichnung.*
Verantwortlicher Use Case (Initiated Use Case):	*Use Case auf dem das Szenario beruht (von dem es eine „Instanz“ bildet).*
Verantwortliche Rolleninstanz (Initiating Actor Instance):	*Die Rolleninstanz, die dieses Szenario ausführt.*
Beschreibung (Flow Of Events):	*Detaillierte Beschreibung des Ablaufs des Szenarios.*

C Checklisten zum Checklistenbasierten Lesen (Inspektion)

Checkliste - Bereich Rollen, Aufgaben, NFR's, Rolleninstanzen

Checkliste - Bereich Rollen, Aufgaben, NFR's, Rolleninstanzen Stellen Sie sich vor, Sie sind ein Auftraggeber und möchten überprüfen, ob ihr Auftrag richtig erfasst worden ist. Überprüfen Sie anhand der Checkliste, wie ihr Auftrag im Dokument umgesetzt worden ist.	
Nr.	**Frage**
1.	**Vollständigkeit:** Sind alle Anforderungen aus der Problembeschreibung durch die Rollen, Aufgaben und NFR's vollständig umgesetzt?
2.	**Vollständigkeit:** Sind für alle Rollen, Aufgaben und NFR's die Textfelder mit sinnvollem Text gefüllt?
3.	**Konsistenz:** Sind die vorhandenen Rollen und Aufgaben konsistent zu der Problembeschreibung?
4.	**Korrektheit:** Sind alle beschriebenen Rollen und Aufgaben durch die Problembeschreibung gefordert?
5.	**Eindeutigkeit:** Sind die beschreibenden Texte einfach, eindeutig und verständlich formuliert?
6.	**Eindeutigkeit:** Sind alle Fachbegriffe im Glossar erläutert?
7.	**Korrektheit:** Ist jede NFR korrekt und sinnvoll verlinkt?
8.	**Prüfbarkeit:** Ist jedes NFR realistisch und eindeutig messbar?
9.	**Vollständigkeit:** Wirkt sich jede NFR auf mindestens ein Anforderungselement (Aufgabe, Use Case, Systemfunktion) aus?
10.	**Interne Konsistenz:** Leistet jede Rolle, jede Aufgabe und jede NFR einen wichtigen, neuen und einmaligen Beitrag zur Anforderungsspezifikation?
11.	**Vollständigkeit:** Ist jede Rolle mindestens für eine Aufgabe verantwortlich oder an mindestens einer Aufgabe beteiligt?
12.	**Vollständigkeit:** Besitz jede Aufgabe eine verantwortliche Rolle?
13.	**Vollständigkeit:** Ist jede Rolleninstanz ein Beispiel für einer Rolle?
14.	**Konsistenz:** Sind die Beschreibungen der Beziehungen zwischen Rollen und Aufgaben konsistent mit den Beschreibung im Nutzungsdiagramm?

Checkliste - Bereich Use Cases

Checkliste - Bereich Use Cases Stellen Sie sich vor, Sie sind ein zukünftiger Benutzer des neu zu entwickelnden Systems und möchten überprüfen, ob die Arbeitprozesse und -abläufe für sie richtig umgesetzt worden sind. Überprüfen Sie anhand der Checkliste, ob die Aufgaben in den Use Cases korrekt umgesetzt worden sind.	
Nr.	**Frage**
1.	**Eindeutigkeit:** Sind die beschreibenden Texte einfach, eindeutig und verständlich formuliert?
2.	**Vollständigkeit:** Sind alle Aufgaben durch Use Cases umgesetzt?
3.	**Vollständigkeit:** Sind die Aufgaben vollständig umgesetzt, d.h. dass alle Aktionen aus den Aufgaben sich im Use Case widerspiegeln?

4.	**Korrektheit:** Sind alle beschriebenen Use Cases durch die Aufgabenbeschreibung gefordert?
5.	**Interne Konsistenz:** Leistet jeder Use Case einen wichtigen, neuen und einmaligen Beitrag zur Anforderungsspezifikation?
6.	**Korrektheit:** Sind die Use Cases korrekt und sinnvoll miteinander verlinkt?
7.	**Konsistenz:** Haben die Fachbegriffe in den Aufgaben und in den Use Cases die gleiche Bedeutung?
8.	**Eindeutigkeit:** Ist jeder verwendete Fachbegriff im Glossar erklärt?
9.	**Vollständigkeit:** Gibt es zu jeder Ausnahme eine entsprechende Beschreibung des Verhaltens, für den Fall, dass diese Ausnahme eintritt?
10.	**Prüfbarkeit:** Lassen sich aus den beschriebenen Use Cases sinnvolle Systemtestfälle ableiten?
11.	**Korrektheit:** Besitzt jeder Use Case bzw. einer seiner Unter - Use Cases mindestens einen Rollen- und einen Systemschritt?
12.	**Korrektheit:** Verwendet jeder Use Case bzw. einer seiner Unter – Use Cases mindestens eine Systemfunktion?
13.	**Konsistenz:** Sind die Beschreibungen der Beziehungen zwischen Aufgaben und Use Cases mit der Beschreibung aus dem Nutzungsdiagramm konsistent?

Checkliste - Bereich Systemfunktionen

Checkliste - Bereich Systemfunktionen	
Stellen Sie sich vor, Sie sind ein Entwickler des neu zu entwickelnden Systems und möchten überprüfen, ob die Systemfunktionen die Informationen enthalten, die Sie zur Entwicklung benötigen. Überprüfen Sie anhand der Checkliste, ob die Use Cases in den Systemfunktionen korrekt umgesetzt worden sind.	
Nr.	**Frage**
1.	**Eindeutigkeit:** Sind die beschreibenden Texte einfach, eindeutig und verständlich formuliert?
2.	**Eindeutigkeit:** Ist jeder verwendete Fachbegriff im Glossar erklärt?
3.	**Konsistenz:** Haben die Fachbegriffe in den Use Cases und den Systemfunktionen die gleiche Bedeutung?
4.	**Vollständigkeit:** Sind für alle Systemfunktionen die Textfelder mit sinnvollem Text gefüllt?
5.	**Vollständigkeit:** Sind die in den Use Cases genannten Systemfunktionen vollständig umgesetzt, d.h. stellen sie das Verhalten bereit, das in dem Use Cases an dieser Stelle notwendig ist.
6.	**Korrektheit:** Wird jede Systemfunktion von mindestens einem Use Case verwendet?
7.	**Korrektheit:** Sind die Systemfunktionen korrekt und sinnvoll mit den Use Cases verlinkt?
8.	**Interne Konsistenz:** Leistet jede Systemfunktion einen wichtigen, neuen und einmaligen Beitrag zur Anforderungsspezifikation?
9.	**Vollständigkeit:** Gibt es zu jeder Ausnahme eine entsprechende Beschreibung des Verhaltens, für den Fall, dass diese Ausnahme eintritt?
10.	**Vollständigkeit:** Besitzt jede Systemfunktion einen eindeutigen Input und Output?
11.	**Korrektheit:** Liegt die Vorbedingung und die Nachbedingung innerhalb der Systemgrenzen?
12.	**Prüfbarkeit:** Lassen sich aus den beschriebenen Systemfunktionen sinnvolle Testfälle ableiten?
13.	**Konsistenz:** Sind die Beschreibungen der Beziehungen zwischen Use Cases und Systemfunktion mit der Beschreibung aus dem Nutzungsdiagramm konsistent?

D Auszüge des Quellkodes der neu implementierten Klassen

Die Klasse `CheckSelectionIntf`

```
package org.globalse.sysiphus.req.im;

import …

/**
 * This class cares about the selection interface of the inspection. In
 * addition it receives and checks the users selection.
 *
 * @author Lars Borner
 * @author Philipp Haefele
 */
public class CheckSelectionIntf {

	private static CheckSelectionIntf instance;

	/**
	 * Constructor of the class (private because of the singleton pattern)
	 */
	private CheckSelectionIntf() {
		// nothing to do
	}

	/**
	 * Returns the instance of the class (singleton pattern). If ther is no
	 * instance when calling the function an instance will be created
	 *
	 * @return the only instance if the class
	 */
	public static CheckSelectionIntf getInstance() {
		if (instance == null) {
			instance = new CheckSelectionIntf();
		}
		return instance;
	}

	/**
	 * Shows the user interface for the selection of the checkcriteria.
	 *
	 * @param req The request object of the client.
	 * @param res The response object to the client.
	 *
	 * For more information about the checkcriteria
	 * @see Checklist
	 */
	public void showSelectionForm(HttpServletRequest req, HttpServletResponse res,
	                              Project project)
		throws IOException {

		String errorMsg = (String)req.getAttribute("Error");
		String automaticInspectionHint
			= "Use the form below to specify the automatic inspection you " +
			  "want to run. " +
			  "After specifying serveral checks, start the inspection.";

		res.setContentType("text/html");
		PrintWriter out = res.getWriter();

		…

		out.println(View.beginForm("REQ", project, "startCheck"));
		out.println(HTML.beginTable());

		ArrayList possibleChecks = Checklist.getInstance().getPossibleChecks();
```

```
        for (int i = 0; i < possibleChecks.size(); i++) {
                String[][] checkpointList = (String[][]) possibleChecks.get(i);
                for (int j = 0; j < checkpointList.length; j++) {
                        String[] checkpoint = checkpointList[j];
                        // Checkpoint is not an checkpoint
                        // (includes only the Section Haeder) see Class: Checklist
                        if (checkpoint[1] == null) {
                                out.println(View.showAttribute(checkpoint[0], ""));
                        }
                        // Checkpoint is a "normal" checkpoint
                        else {

out.println(View.showAttribute(HTML.checkbox(checkpoint[0], checkpoint[0], false),
                checkpoint[1]));
                        }
                }
        }

        out.println(View.showAttribute("", HTML.resetButton("Reset Choices")));
        out.println(View.showAttribute("", HTML.submitButton("Start Inspection")));

        out.println(HTML.endTable());
        out.println(HTML.endForm() + HTML.footer());
}

/**
 * Starts the inspection-process.
 *
 * @param req The request object of the client.
 * @param res The response object to the client.
 * @param project The project of which the elements should be checked.
 */
public void startCheck(HttpServletRequest req, HttpServletResponse res,
                        Project project) {
        String[] checksToDo = readCheckBoxes(req);

        try {
                if(checksToDo.length == 0) {
                        showHintNoChecks(req, res, project);
                }
                else {
                        PrintWriter out = res.getWriter();

                        …

                        out.println(HTML.center(View.hint("Performing the Auto - "
                                + "Inspection! Please wait!")));
                        // Flush the buffer to let the user know that the Inspection
                        // is running
                        res.flushBuffer();
                        Checklist.getInstance().performCheck(checksToDo, project);
                        // reload view: in right frame the ReqSpec view, in left
                        // frame the Question view
                        //

                        out.println("Auto - Inspection successfully done ");

                        …

                        out.println(HTML.footer());

                }

        } catch (Exception e) {
                e.printStackTrace();
        }
}

/**
 * Reads the selected checkcriteria given by request object of the client
 * and returns them in a string array.
 *
```

```
         * @param req The request object of the client.
         * @param res The response object to the client.
         *
         * @return The selected checkcriteria.
         */
        private String[] readCheckBoxes(HttpServletRequest req) {
                ArrayList checksToDo = new ArrayList();
                ArrayList possibleChecks = Checklist.getInstance().getPossibleChecks();
                for (int i = 0; i < possibleChecks.size(); i++) {
                        String[][] sectionList = (String[][]) possibleChecks.get(i);
                        for (int j = 0; j < sectionList.length; j++) {
                                String parameter = sectionList[j][0];
                                if (req.getParameter(parameter) != null) {
                                        checksToDo.add(req.getParameter(parameter));

                                }
                        }
                }
                return (String[]) checksToDo.toArray(new String[checksToDo.size()]);
        }

        /**
         * Shows the hint, that no selection of checkcriteria where made.
         *
         * @param req The request object of the client.
         * @param res The response object to the client.
         * @param project The current project.
         */
        private void showHintNoChecks(HttpServletRequest req, HttpServletResponse res,
                                        Project project)
                        throws IOException {

                PrintWriter out = res.getWriter();

                …

                out.println("<DIV style=\"color: red;\">");
                out.println(HTML.center(View.hint("The Inspection can't be performed. "
                        + "No Checks specified.")));
                out.println("</DIV>");

                …

                out.println(HTML.footer());
        }
}
```

Die Klasse `Checklist`

```
package org.globalse.sysiphus.model.im;

import …

/**
 * The class Checklist contains all the information about the possible checks
 * which could be made during an inspection supported by the system. It
 * includes all the operations necessary for checking.
 *
 * @author Lars Borner
 * @author Philipp Haefele
 */
public class Checklist {

        /* <<Haedderinformation>> of the requirements section (Requirements
         * Specification Document) checkcriteria
         */
        private static final String[] LIST_HAEDER_REQ = {"Requirements", null};
        /* Checkcriteria of elements of the requirements section
         *
```

```
 * 1st entry unique name (references to the check functions further down)
 * 2nd entry description of the checkcriteria (i.e. to show to the user
 * for selection)
 */
private static final String[] ELEMENT_DESCRIPTION_REQ
       = {"ElementDescription_REQ",
              "Checks if each element in the Requirements Section has a
              description."};
private static final String[] ACTOR_USERTASK = {"Actor_UserTask",
              "Checks if each actor initiates/paticipates in at least one User
              Task."};
private static final String[] USERTASK_ACTOR = {"UserTask_Actor",
              "Checks if each User Task has at least one initiating Actor."};
private static final String[] QUALITY_USERTASK = {"Quality_UserTask",
              "Checks if each NFR in the Requirements Section constrains at least
              one User Task."};

/* <<Haedderinformation>> of the specification section (Requirements
 * Specification Document) checkcriteria
 */
private static final String[] LIST_HAEDER_SPEC = {"Specification", null};
/* Checkcriteria of elements of the specification section
 *
 * 1st entry unique name (references to the check functions further down)
 * 2nd entry description of the checkcriteria (i.e. to show to the user
 * for selection)
 */
private static final String[] ELEMENT_DESCRIPTION_SPEC
       = {"ElementDescription_SPEC",
              "Checks if each element in the Specification Section has a
              description."};
private static final String[] USERTASK_USECASE = {"UserTask_UseCase",
              "Checks if each User Task is realized by at least one Use Case."};
private static final String[] USECASE_USERTASK = {"UseCase_UserTask",
              "Checks if each Use Case is derived from a User Task."};
private static final String[] USECASE_SYSTEMSTEP
       = {"UseCase_SystemStep",
              "Checks (recursive) if each Use Case (or its included Use Cases) has
              at least one Systemstep."};
private static final String[] USECASE_SERVICE = {"UseCase_Service",
              "Checks (recursive) if each Use Case (or its included Use Cases) use
              at least one Service."};
private static final String[] QUALITY_USECASE = {"Quality_UseCase",
              "Checks if each NFR (specified for Use Cases) constrains at least one
              Use Case."};
private static final String[] QUALITY_SERVICE = {"Quality_Service",
              "Checks if each NFR (specified for Services) constrains at least one
              Service."};

/* <<Haedderinformation>> of the example section (Requirements
 * Specification Document) checkcriteria
 */
private static final String[] LIST_HAEDER_EXAMP = {"Example", null};
/* Checkcriteria of elements of the example section
 *
 * 1st entry unique name (references to the check functions further down)
 * 2nd entry description of the checkcriteria (i.e. to show to the user
 * for selection)
 */
private static final String[] ELEMENT_DESCRIPTION_EXAMP
       = {"ElementDescription_EXAMP",
              "Checks if each element in the Example Section has a description."};
private static final String[] ACTORINSTANCE_ACTOR
       = {"ActorInstance_Actor",
              "Checks if each Actorinstnce is an instance of an Actor."};
private static final String[] SCENARIO_USECASE = {"Scenario_UseCase",
              "Checks if an Scenario is an instance if an Use Case."};
private static final String[] SCENARIO_ACTORINSTANCE
       = {"Scenario_ActorInstance",
              "Checks if each Scenario is initiated by an Actorinstance."};

/**
```

```
 * List that contains all the checkcriteria of the Requirements Section
 * of the Requirements Specification Document
 */
private static final String[][] REQ_LIST = {LIST_HAEDER_REQ,
               ELEMENT_DESCRIPTION_REQ, ACTOR_USERTASK, USERTASK_ACTOR,
               QUALITY_USERTASK};

/**
 * List that contains all the checkcriteria of the Specification Section
 * of the Requirements Specification Document
 */
private static final String[][] SPEC_LIST = {LIST_HAEDER_SPEC,
               ELEMENT_DESCRIPTION_SPEC, USERTASK_USECASE, USECASE_USERTASK,
               USECASE_SYSTEMSTEP, USECASE_SERVICE, QUALITY_USECASE,
               QUALITY_SERVICE};

/**
 * List that contains all the checkcriteria of the Example Section
 * of the Requirements Specification Document
 */
private static final String[][] EXAMP_LIST = {LIST_HAEDER_EXAMP,
               ELEMENT_DESCRIPTION_EXAMP, ACTORINSTANCE_ACTOR, SCENARIO_USECASE,
               SCENARIO_ACTORINSTANCE};

private ArrayList nowCheckedUseCases = new ArrayList();

private static Checklist instance;

/**
 * Constructor of the class (private because of the singleton pattern)
 */
private Checklist() {
       // nothing to do
}


/**
 * Returns the instance of the class (singleton pattern). If ther is no
 * instance when calling the function an instance will be created
 *
 * @return the only instance if the class
 *
 */
public static Checklist getInstance() {
       if (instance == null) {
              instance = new Checklist();
       }
       return instance;
}

/**
 * Returns a string-array of string-arrays that contain the possible
 * checks to perform
 *
 * @return string-array containing Lists of possible checkcriteria
 *
 * @see #REQ_LIST
 * @see #SPEC_LIST
 * @see #EXAMP_LIST
 *
 *
 */
public ArrayList getPossibleChecks() {

       ArrayList possibleChecks = new ArrayList();
       possibleChecks.add(0, REQ_LIST);
       possibleChecks.add(1, SPEC_LIST);
       possibleChecks.add(2, EXAMP_LIST);

       return possibleChecks;
}
```

```
/**
 * Performs the inspection with the given checkcriteria in the given
 * project
 *
 * @param checksToDo string-array with the names of the checkcriteria to
 *                   perform
 * @param project    id of the project in which the elements shoud be
 *                   checked
 */
public synchronized void performCheck(String[] checksToDo, Project project)
                throws Exception {

        for (int i = 0; i < checksToDo.length; i++) {
                Class[] parameterTypes = new Class[] {Project.class};
                Object [] operands = new Object[] {project};
                getClass().getMethod("check" + checksToDo[i],
                        parameterTypes).invoke(this, operands);
        }
}


/*       ************** Check functions **************       */
/**
 * Checks if any element of the requirements section of the Requirements
 * Specification Document has a description
 */
public void checkElementDescription_REQ(Project project) {
        …
}

/**
 * Checks (recursive) if each Use Case (or its included Use Cases) has
 * at least one Systemstep.
 */
public void checkUseCase_SystemStep(Project project) {
        UseCase[] useCases
                = (UseCase[]) project.getElementsByClass(UseCase.class);
        for (int i = 0; i < useCases.length; i++) {
                nowCheckedUseCases.clear();
                checkUseCase_SystemStep(useCases[i]);
        }
}

private boolean checkUseCase_SystemStep(UseCase useCase) {
        SystemStep[] systemSteps = useCase.getSystemSteps();
        // if the Use Case has an System Step everything is ok
        if (systemSteps.length > 0) {
                return true;
        }

        // if the Use Case has no System Step, get all Actor Steps and check
        // if they include any Use Cases
        else {
                ActorStep[] actorSteps = useCase.getActorSteps();
                ArrayList includedUCes = new ArrayList();
                for (int i= 0; i < actorSteps.length; i++) {
                        if (actorSteps[i].getIncludedUseCase() != null) {
                                includedUCes.add(actorSteps[i].getIncludedUseCase());
                        }
                }

                // if the UC includes no other UC report the problem
                if (includedUCes == null || includedUCes.size() == 0) {
                        new Problem(useCase, Problem.NO_SYSTEMSTEP_FOR_USECASE);
                        return false;
                }
                // else do the check for the included UC but be aware of cycles
                else {
                        nowCheckedUseCases.add(useCase);
                        boolean foundSystemStep = false;
                        for (int i = 0; i < includedUCes.size(); i++) {
                                if(!nowCheckedUseCases.contains(includedUCes.get(i))){
```

```
                                        foundSystemStep = foundSystemStep
                                                || checkUseCase_SystemStep(
                                                        (UseCase) includedUCes.get(i));
                                        }
                                }
                                if (foundSystemStep) {
                                        return true;
                                }
                                else {
                                        new Problem(useCase, Problem.NO_SYSTEMSTEP_FOR_USECASE);
                                        return false;
                                }
                        }
                }
        }

        …
}
```

Die Klasse `Problem`

```
package org.globalse.sysiphus.model.im;

import …

/**
 * This class represents a problem of an element. Because the documentation
 * of problems is realised through questions (issues) an instance of an issue
 * is generated for each problem and the problem instance is left to carbage
 * collector.
 *
 * @author Lars Borner
 * @author Philipp Haefele
 */
public class Problem {

        /* Problemtypes and their description (correspond to the checkcriteria
         * in class Checklist)
         */
        public static final String NO_DESCRIPTION
                        = "Why does the following Element has no description? Element: ";
        public static final String NO_USERTASK_FOR_ACTOR
                        = "Why does the following Actor does not initiate/participate in an
                                User Task? Actor: ";
        public static final String NO_ACTOR_FOR_USERTASK
                        = "Why does the following User Task has no Initiating Actor? User
                                Task: ";
        public static final String NO_USERTASK_FOR_QUALITY
                        = "Why does the following NFR constrains no User Task? NFR: ";
        public static final String NO_USECASE_FOR_USERTASK
                        = "Why does the following User Task is not realized by any Use Case?
                                User Task: ";
        public static final String NO_USERTASK_FOR_USECASE
                        = "Why does this Use Case is not derived from any User Task? Use
                                Case: ";
        public static final String NO_SYSTEMSTEP_FOR_USECASE
                        = "Why does the following Use Case contains no Systemstep? Use Case: ";
        public static final String NO_SERVICE_FOR_USECASE
                        = "Why does the following Use Case contains no Service? Use Case: ";
        public static final String NO_USECASE_FOR_QUALITY
                        = "Why does the following NFR constrains no Use Case? NFR: ";
        public static final String NO_SERVICE_FOR_QUALITY
                        = "Why does the following NFR constrains no Service? NFR: ";
        public static final String NO_ACTOR_FOR_ACTORINSTANCE
                        = "Why does the following Actor Instance is not an instance of a
                                specified Actor? Actor Instance: ";
        public static final String NO_USECASE_FOR_SCENARIO
                        = "Why does the following Scenario is not an instance of a specified
                                Use Case? Scenario: ";
```

```
	public static final String NO_ACTORINSTANCE_FOR_SCENARIO
			= "Why does the following Scenario is not initiated by any Actor
				Instance? Scenario: ";

	/**
	 * Constructor of the class. It only calls createIssue to add an issue to
	 * the project to hint at the problem.
	 *
	 * @param element Element which has a problem
	 * @param problemType Type of the problem
	 */
	public Problem(LinkableElement element, String type) {
		createIssue(element, type);
	}

	/**
	 * Creates an Issue of the specified problemtype and adds it to the
	 * specified element. Before it composes the text of the issue.
	 *
	 * @param element Element which has a problem
	 * @param problemType Type of the problem
	 */
	private void createIssue(LinkableElement element, String problemType) {
		String questionTitle = "SYSTEM: " + problemType + " " + element.getName();
		Project project = element.getProject();
		Issue problemIssue;

		Issue[] elementIssues = element.getIssues();
		boolean questionExists = false;
		for (int i = 0; i < elementIssues.length; i++) {
			if (elementIssues[i].getName().equals(questionTitle)) {

				questionExists = true;
				elementIssues[i].reopen();
			}
		}
		if (!questionExists) {
			try {
				problemIssue = new Issue(questionTitle, project);
				element.addIssue(problemIssue);
			} catch (ElementStoreNameException e) {}
		}
	}
}
```

Glossar

Ad-hoc Lesen

Hier liest der Inspektor die Dokumente ohne spezielle Vorgaben. Das Verfahren benötigt einen hohen Erfahrungsschatz des Inspektors, damit er weiß, worauf während der Inspektion geachtet werden muss und nicht nur „oberflächliche“ Fehler aufgedeckt werden.

Aktivität

(Innerhalb einer Methode) Handlungseinheit einer Rolle mit einem bestimmten angestrebten Ziel.

Anforderung

- Bedingung oder Fähigkeit, die ein Anwender stellt bzw. benötigt, um ein Problem zu lösen oder ein Ziel zu erreichen.
- Bedingung oder Fähigkeit, die ein System erfüllen bzw. besitzen muss.

Anforderungsanalyse

Soll ein neues System erstellt werden, so muss man beginnen, sich mit dessen Arbeitskontext und dessen Zweck auseinanderzusetzen. Möglichst früh sollte man sich auch den Interessen der an der Entwicklung beteiligten Gruppen bewusst werden, um den Konsensfindungsprozess voranzutreiben und die Entwicklung nicht in eine falsche Richtung zu lenken.

Zur Anforderungsanalyse (Wissenserwerb) finden verschiedenste Techniken Anwendung. Von Interviews mit den Zielgruppen des künftigen Systems (Management, Benutzer, usw.), über Beobachtungen des Geschäftsprozesses, bis hin zu Literaturrecherchen zu vorhandenen Prozessbeschreibungen. Zur Dokumentation des erlangten Wissens hat es sich bewährt, auf Geschäftsprozessmodellierung zurückzugreifen, die dann als Grundlage für die unterschiedlichsten Softwareentwicklungsphasen herangezogen werden können.

Wird ein bestehendes System erweitert, sieht der Wissenserwerb etwas anders aus. Neben den bereits genannten Methoden zur Wissensermittlung ist es notwendig, auch das bestehende System kennenzulernen, indem man die Anforderungs- und Spezifikationsdokumente durcharbeitet, um die Art und Weise zu erfahren, wie die bisherige Unterstützung des Arbeitsprozesses durch das System aussieht. Auch über eine Durchführung von Quellkodeinspektionen sowie Usability Tests kann man zur Einarbeitung in das System nachdenken. Dies setzt allerdings voraus, dass bereits eine

Auseinandersetzung mit der Domäne und dem Nutzungskontext des Systems stattgefunden hat.
Die Anforderungsanalyse ist nicht als Phase vor dem Beginn des Requirements Engineering zu sehen. Vielmehr erfolgt die Anforderungsanalyse im TRAIN-Prozess parallel zum Requirements Engineering, zur Architekturdefinition, zum Feinentwurf und zur Implementierung.

Anforderungselement
Ein Anforderungselement ist eine Hilfe, um die Wünsche des Auftragsgebers für das SW-System geeignet beschreiben zu können. Ein Anforderungselement besteht aus einem Namen und einer Beschreibung. Die Beschreibung für ein Anforderungselement ist abhängig von der Art des Anforderungselementes. Es gibt verschiedene Arten von Anforderungselementen, z. B. Aufgaben, Rollen, Use Cases, NFRs, Systemfunktionen usw.

Äquivalenzklasse
„Eine Menge von Werten, bei der jeder Wert

- als Eingabe für ein Testobjekt gleichartiges Sollverhalten zeigt (Äquivalenzklasse von Eingabewerten)
- als Ergebnis eines Testlaufs gleichartiges Sollverhalten aufzeigt (Äquivalenzklasse von Ausgabewerten)“[23]

Artefakt
(Dokumentations-) Produkt einer ↑Aktivität (z. B. UML-Diagramm).

Black-Box (Test-) Verfahren
„Alle Verfahren, die zur Herleitung oder Auswahl der Testfälle keine Information über die innere Struktur des Testobjektes benötigen.“[24]

CASE-Tool
„CASE steht für **C**omputer **A**ided **S**oftware **E**ngineering, also die Computer-seitige Unterstützung im gesamten Prozess der Software-Entwicklung.
CASE-Tools sind Programme, die den Software-Ingenieur bei der Planung, dem Entwurf, der Implementierung und der Dokumentation unterstützen. […]“[25]

---

[23] SPILLNER, ANDREAS, LINZ,TILO: Basiswissen Softwaretest. Heidelberg 2004. S. 197.
[24] Ebd., S. 198.

Checklistenbasiertes Lesen

Diese Technik fasst die bei vergangenen Inspektionen gewonnenen und reflektierten Erfahrungen in Checklisten zusammen. Anhand der in diesen Listen genannten Checkpunkte, geht der Inspektor die Dokumente systematisch durch. Jeder Checkpunkt weist auf potentiell vorhandene Fehler in den Dokumenten hin. Diese Technik ermöglicht eine recht schnelle, objektive und wiederholbare Inspektion.

Debuggen

Lokalisieren und Beheben eines ↑Defekts.

Debuggingzyklus

siehe Kapitel 2.3.5

Defekt

↑Fehler

Domäne

Kontext (Umgebung, Bereich, Anwendungsgebiet) in dem ein Softwaresystem eingesetzt wird.

Domänendaten

Domänendaten beschreiben Entitäten (Dinge und Konzepte), die im Kontext (↑Domäne) eines Systems wichtig sind (z. B. in der Domäne verarbeitete Daten).

Dokumentationselement

Ein Dokumentationselement ist ein Element in einem Softwaredokument, im speziellen ein ↑Anforderungselement. Als Dokumentationselement sind aber nicht nur Elemente des Anforderungsdokumentes zu verstehen, sondern auch von anderen Softwaredokumenten wie beispielsweise der ↑Testspezifikation. Hierin gehört z. B. ein Testplan oder eine ↑Testspezifikation zu den Dokumentationselementen.

Fehler

- (In einem Softwaresystem) Nichterfüllung einer Anforderung. Zu erkennen in einer Abweichung zwischen Ist- und Soll-Verhalten. Genauer ist aber zu unterscheiden zwischen Defekt (Fehlerursache) und Fehlerwirkung.
- (In einem Softwaredokument) Nichterfüllung der formalen (syntaktischen) oder inhaltlichen (semantischen) Anforderungen an ein Softwaredokument.

---

[25] http://de.wikipedia.org/wiki/Computer_Aided_Software_Engineering (25.01.2005).

Fehlerliste

Dient der Dokumentation identifizierter Fehler während des Inspektionsprozesses (vgl. auch ↑Problemliste).

http

Kommunikationsvereinbarungen (Protokoll) in Computernetzwerken (z. B. im Internet)

http-Anfrage Header

In einem Header befinden sich bei einer http-Anfrage eines Clients für den Server relevante Informationen.

http get Anfrage

http-Anfragemethode eines Clients an einen Server, die die Anfrageparameter in Form eines Strings mit einem vorangestellten Fragezeichen an das Ende des URLs anhängt.

http post Anfrage

http-Anfragemethode eines Clients an einen Server, die die Anfrageparameter in Anschluss an die http-Anfrage Haeder (und eine trennende Leerzeile) sendet.

Integrationstest

„Test mit dem Ziel, Fehler in Schnittstellen und im Zusammenspiel zwischen integrierten Komponenten zu finden.“[26]

Java Servlet-Technologie

Technologie zum Aufbau von web-basierten Anwendungen.

Java Swing

Framework zum Entwickeln von graphischen Benutzungsschnittstellen (GUI).

Klassenmitglieder

Variablen, Operationen und Konstruktoren einer Klasse.

Kohäsion

Die Kohäsion ist ein Maß für die Zusammengehörigkeit der Bestandteile einer Komponente. Eine hohe Kohäsion bedeutet eine starke Abhängigkeit zwischen den Elementen einer Komponente.

Komponententest

Test einer Softwareeinheit (z. B. Operation, Klasse).

---

[26] SPILLNER, ANDREAS, LINZ,TILO: Basiswissen Softwaretest. Heidelberg 2004. S. 202.

Kontrollfluss

„Abstrakte Repräsentation von allen möglichen Reihenfolgen von Ereignissen während einer Programmausführung, Repräsentation der Kontrollstruktur (meist graphisch dargestellt in einem Kontrollflussgraphen).“[27]

Kontrollflussgraph

„Kontrollflussgraphen dienen zur Darstellung der Kontrollstruktur von Komponenten“,[28] z. B. von Operationen.

Kopplung

Die Kopplung ist ein Maß für die Abhängigkeit zwischen Komponenten. Eine niedrige Kopplung bedeutet eine geringe Abhängigkeit zwischen Komponenten.

Lesetechnik

Lesetechniken sind Bestandteil des Inspektionsprozesses. Sie sind für den Inspektor bei der Fehlersuche ein Hilfsmittel, da sie ihm eine Systematik vorgeben, auf welche Weise er die Softwaredokumente inspizieren soll.
Es stehen folgende Lesetechniken zu Verfügung:

- ↑Ad-hoc Lesen
- ↑Checklistenbasiertes Lesen
- ↑Perspektivenbasiertes Lesen

Mitglieder

↑Klassenmitglieder

Pair Programming

Programmiermethode, bei der zwei Entwickler gemeinsam implementieren, daher Entwurfsumsetzungen im Programmkode dem „Partner“ direkt begründet werden müssen und von diesem unmittelbar überprüft werden können. Zielt auf qualitativ hochwertigeren Programmkode ab.

Perspektivenbasiertes Lesen

Diese Technik versetzt den Inspektor in die Perspektive einer am Softwareentwicklungsprozess beteiligten Rolle, indem eine Aufgabe aus der Sicht dieser Rolle mit dem Dokument bewältigt werden muss. Während der Abarbeitung der Aufgabe

---

[27] Ebd., S. 203.

ist eine Frageliste zu beantworten, die auf potentielle Fehler hinweist. Diese Technik ist besonders gut geeignet für die Entdeckung semantischer Mängel des Dokumentes, setzt jedoch vom Inspektor die Fähigkeit voraus, sich in die verschiedenen Rollen hineinzuversetzen. Eine bessere Möglichkeit ist natürlich, eine Person, die diese Rolle innehat, mit der Inspektion zu betrauen.

Problem

Vermuteter, potentieller ↑Fehler in einem Softwaredokument

Problemliste

Dient der Dokumentation von ↑Problemen, Fragen, Verbesserungsvorschlägen während der Fehlersuche einer Inspektion.

Produkt

↑Artefakt

Rationale

Erfassung der Begründungen für Gestaltungsentscheidungen während der Softwareentwicklung.

Reviewverfahren

Das Reviewverfahren bestimmt die Vorgehensweise beim Inspektionsprozess. Jedes Reviewverfahren beinhaltet für die Aktivität der Fehlersuche eine ↑Lesetechnik, die den Inspektor systematisch bei seiner Arbeit unterstützen soll.

Sollverhalten

„Erwartete Ausgabewerte und erwartetes Verhalten des Testobjektes (für jeden Testfall festzulegen)“[29]

Stub

- Stubs (Platzhalter) „werden beim [↑]Komponenten- und [↑]Integrationstest benötigt, um noch nicht implementierte Komponenten für die Testdurchführung zu ersetzen bzw. zu simulieren.“[30]
- Stubs ersetzen nicht vorhandene Kodeteile.

---

[28] Ebd., S. 204.
[29] Ebd., S. 208.
[30] Ebd., S. 206.

Systemtest

„Test eines integrierten Systems, um sicherzustellen, dass es spezifizierte Anforderungen erfüllt“[31]

Test

Systematische Aufdeckung von Fehlerwirkungen (vgl. ↑Fehler).

Testimplementierung

Implementierung der Testfälle gemäß ↑Testspezifikationen inkl. ↑Testtreiber und ↑Stubs.

Testplanung

Die Testplanung diskutiert Qualitätsziele, zu testende Objekte/Operationen, Testmethoden, Abdeckungsgrad der Testfälle, Priorisierung der Testobjekte/-fälle, benötigte Ressourcen, Verantwortlichkeiten, Zeitplanung und Risiken.

Testrahmen

Umsetzung des in der Testspezifikation festgelegten Ablaufs im Programmkode.

Testspezifikation

Definition der Testobjekte und ihrer Testfälle, Festlegung des ↑Sollverhaltens und Abbruchverhaltens bei Durchführung der Tests.

Testtreiber

- „Programm […], das es ermöglicht, ein Testobjekt ablaufen zu lassen, mit Testdaten zu versorgen und Ausgaben/Reaktionen des Testobjektes entgegenzunehmen.“[32]
- Testtreiber liefern Testdaten, stoßen die Ausführung an und protokollieren Ausgaben.

UML

↑Unified Modelling Language

Unified Modelling Language (UML)

Eine Sprachspezifikation zur Modellierung von Struktur, Verhalten und Architektur eines Softwaresystems sowie von Geschäftsprozessen.

---

[31] Ebd., S. 209.

Validierung

- Validierung ist die Frage, ob man *das Richtige* gemacht hat. Daher kann man eine Entscheidung nur unmittelbar gegen die Benutzererwartungen validieren.
- „Prüfung der Anwendbarkeit einer Problemlösung (eines Produkts bzw. Systems) in seiner Umwelt, i. a. durch den Benutzer.“[33]

Verifikation

- Verifikation ist die Frage, ob man *es richtig* gemacht hat, z. B. die Umsetzung einer Anforderung im Entwurf. Daher findet man die Antwort auf die Frage meist in der Dokumentation von Entscheidungen einer früheren Phase.
- „Vergleich des Produktes einer Entwicklungsphase mit seinen Vorgaben (z. B. Baustein gegen seine Spezifikation).“[34]

White-Box (Test-) Verfahren

„Alle Verfahren, die zur Herleitung oder Auswahl der Testfälle Information über die innere Struktur des Testobjektes benötigen.“ [35]

Wissenserwerb

↑Anforderungsanalyse

Zweig

Eine Kante im ↑Kontrollflussgraphen.

Zweigüberdeckung

„Dynamisches, kontrollflussbasiertes Testverfahren, fordert die Überdeckung aller ↑Zweige des ↑Kontrollflussgraphen einer Komponente“.[36]

---

[32] Ebd., S. 213.
[33] http://www.fbe.hs-bremen.de/spillner/begriffe/start.html (27.01.2005) – „Begriffsdefinitionen im Testbereich“ einer Arbeitsgruppe innerhalb der Gesellschaft für Informatik (GI) Fachgruppe „Testen, Analysieren und Verifizieren von Software“ (TAV).
[34] Ebd.
[35] SPILLNER, ANDREAS, LINZ,TILO: Basiswissen Softwaretest. Heidelberg 2004. S. 216.
[36] Ebd., S. 216.

Quellenverzeichnis

Bibliographie

CARBON, RALF; CIOLKOWSKI MARCUS; u. a.	Praktika in der AG Software Engineering – Kurzbeschreibung der Artefakte und Aktivitäten	Software Engineering Research Group (AGSE) Technische Universität Kaiserslautern
	wwwagse.informatik.uni-kl.de/teaching/se1lab/ss2004/Einarbeitung/Prozess.pdf (27.01.2005)	
BRUEGGE, BERND; DUTOIT, ALLEN H.	Object-Oriented Software Engineering. Using UML, Patterns, and Java	(second edition) Upper Saddle River: Prentice Hall 2004
	Homepage des Buches mit Materialien: http://oose.globalse.org/ (27.01.2005)	
GAMMA, ERICH; HELM, RICHARD; u. a.	Design patterns - elements of reusable object-oriented software	Reading: Addison-Wesley 2000
JACOBSON, IVAR; CHRISTERSON, MAGNUS ; u. a.	Object-Oriented Software Engineering. A Use Case Driven Approach	Reading: Addison-Wesley 1992
JECKLE, MARIO; RUPP, CHRIS; u. a.	UML 2 glasklar	München; Wien: Hanser 2004
LAUESEN, SOREN	User Interface Design. A Software Engineering Perspective	Reading: Addison-Wesley 2004
PAECH, BARBARA	Software Engineering - Planung und Durchführung von Softwareentwicklungsprojekten	Software Engineering Group (SEG) Universität Heidelberg
	Vorlesungsfolien WS2003/04 SS2004	
	http://www-swe.informatik.uni-heidelberg.de/teaching/ss2004/ss2004SWE/swe.shtml (27.01.2005)	
PAECH, BARBARA; KOHLER, KIRSTIN	Task-Driven Requirements in Object-Oriented Development	In: LEITE, JULIO; DOORN, JORGE (eds.): Perspectives on Software Requirements. Dordrecht: Kluwer Academic Publishers 2004
PAECH, BARBARA; BORNER, LARS; u. a.	Vom Kode zu den Anforderungen und wieder zurück: Software Engineering in sechs Semesterwochenstunden	In: LÖHR, KLAUS-PETER; LICHTER, HORST (Hrsg.): Software Engineering im Unterricht der Hochschulen. SEUH 9. Aachen 2005. Heidelberg: dpunkt-Verlag 2005
SPILLNER, ANDREAS; LINZ, TILO	Basiswissen Softwaretest. Aus- und Weiterbildung zum Certified-Tester	(1. Auflage) Heidelberg: dpunkt-Verlag 2004
SPILLNER, ANDREAS	The W-MODEL – Strengthening the Bond Between Development and Test	ZIMT - Zentrum für Informatik und Medientechnologien Hochschule Bremen
	http://www.stickyminds.com/getfile.asp?ot=XML&id=3572&fn=XDD3572filelistfilename1%2Epdf (27.01.2005)	

Internet

www.junit.org	Webseite des Testwerkzeugs JUnit
http://jude.esm.jp/	Webseite des UML-Modellierungswerkzeugs JUDE
www.eclipse.org	Webseite der Entwicklungsumgebung Eclipse
www.uml.org	Webseite der Unified Modeling Language (UML)
http://java.sun.com/	Webseite von Sun für Java
http://java.sun.com/docs/codeconv/	Webseite von Sun für Java Codierungsrichtlinien (Codeconventions)
http://java.sun.com/j2se/javadoc/	Webseite von Sun für JavaDoc
http://java.sun.com/products/servlet/	Webseite von Sun für JavaServlets
http://jakarta.apache.org/tomcat/	Webseite des JavaServlet Containers Tomcat
http://www-swe.informatik.uni-heidelberg.de/	Website des Lehrstuhls für Software Systeme an der Universität Heidelberg

(Verweise geprüft am 27.01.2005)