TECHNISCHE UNIVERSITÄT ILMENAU

Fakultät für Informatik und Automatisierung
Institut für Theoretische und Technische Informatik
Fachgebiet Softwaresysteme / Prozessinformatik

Diplomarbeit

zur Erlangung des akademischen Grades Diplom-Informatiker

Traceability und Entwurfsentscheidungen für Software-Architekturen mit der Quasar-Methode

---

Autor:	Stephan Bode
Geboren am:	05. 02. 1983
Matrikelnummer:	34761
Verantwortlicher Hochschullehrer:	Priv.-Doz. Dr.-Ing. habil. Matthias Riebisch
Inventarisierungsnummer:	2008-01-23/015/IN02/2232

Eingereicht am: 23. 01. 2008

Zusammenfassung

Die Wartung von Software spielt heutzutage eine wichtige Rolle, da bestehende Systeme häufig an sich ändernde Bedürfnisse und Anforderungen angepasst werden müssen. Mit dem Konzept der Traceability können Entwicklungsschritte eines Softwaresystems nachvollzogen werden, indem die Artefakte der verschiedenen Schritte über Traceability-Links miteinander verknüpft werden.

In der vorliegenden Arbeit wird das Vorgehen nach der Architekturentwurfsmethode Quasar kritisch untersucht. Die einzelnen Aktivitäten von Quasar werden durch zusätzliche Aktivitäten erweitert, um ein lückenloses Vorgehen zu erreichen und die Zuordnung von Traceability-Links zu ermöglichen. Die Methode wird bei der Weiterentwicklung des SalesPoint-Frameworks eingesetzt.

Die Architektur der grafischen Benutzerschnittstelle einer Beispielanwendung wird unter Verwendung der erweiterten Quasar-Methode sowie auf Basis des SalesPoint-Frameworks neu entwickelt. Die getroffenen Entwurfsentscheidungen in diesem Prozess werden dokumentiert und Traceability-Links erstellt. Besondere Beachtung beim Design der Benutzeroberfläche wird auch der Einhaltung von softwareergonomischen Richtlinien geschenkt, um gute Usability zu erreichen.

Abstract

Nowadays software maintenance plays an important role, because existing systems have to adapt to frequently changing needs and requirements. The concept of traceability helps to understand the development steps of a software system by linking the artifacts of different steps via traceability links.

In this thesis the development steps of the architectural design method Quasar are examined critically. The several activities of Quasar are extended by additional activities to complete the method and to enable traceability. The method is used for the evolution of the SalesPoint framework.

The architecture of the graphical user interface is developed as an example for both the extended method and the SalesPoint framework. The design decisions in this process are documented and traceability links are established. Special emphasis during the design of the user interface is also placed on the consideration of software-ergonomic guidelines to achieve good usability.

Inhalt

1 Einleitung

1.1 Motivation

Softwaresysteme spielen heutzutage eine zunehmend wichtigere Rolle und werden immer komplexer. Sie müssen an die sich häufig ändernden Bedürfnisse und neuen Anforderungen angepasst werden, um ihren weiteren Einsatz gewährleisten zu können. Besonders in komplexen Umgebungen sind Änderungen mit großem Aufwand verbunden und tragen hohe Risiken in sich. Solche Risiken, die bei Änderungen an Softwaresystemen auftreten, sind nach Boehm [Boe81] und Broy et al. [BDP06] beispielsweise:

- unvollständige Implementierung,
- missverstandene Abhängigkeiten,
- fehlendes Verständnis oder
- Strukturverlust (englisch: *architectural decay*).

Zur Vermeidung daraus resultierender negativer Konsequenzen bei der evolutionären Weiterentwicklung oder Wartung von Softwaresystemen ist eine Unterstützung durch softwaretechnische Konzepte, Methoden und Werkzeuge notwendig. Ein Konzept, welches bei der Softwarewartung komplexer Systeme die Entwurfsentscheidungen erleichtert und das Programmverständnis erhöht, ist Traceability. Traceability ermöglicht es, Verbindungen über mehrere Abstraktionsebenen von den neuen oder geänderten Anforderungen zu ihren Realisierungen aufzubauen.

Traceability, Rückverfolgbarkeit, stellt die Fähigkeit dar, die Entwicklungsschritte eines Systems durch die Verbindung der Requirements mit den Ergebnissen eines jeden Entwicklungsschrittes nachverfolgen und wiederherstellen zu können [GF94]. Diese Verbindungen zwischen Artefakten der Anforderungsanalyse, des Entwurfs und der Implementierung werden üblicherweise Traceability-Links genannt. Vorteilhaft am Konzept der Traceability ist die explizite Formulierung von Abhängigkeiten und Bezügen zwischen den einzelnen Software-Artefakten [Rie04]. Traceability-Links erhöhen die Verständlichkeit und vereinfachen Änderungen an Softwaresystemen durch Unterstützung von Aufwandsschätzungen, Vollstän-

digkeitsanalysen und Konsistenzprüfungen. Doch ist es für eine effektive Nutzung der Traceability-Links notwendig, dass diese gültig, also korrekt und vollständig, sind [MRP06].

Trotz der positiven Effekte wird Traceability in der Praxis bisher selten verwendet. Ein Haupteinsatzgebiet für Traceability ist beim Requirements-Engineering, wie von Ramesh und Jarke [RJ01] vorschlagen, also beim Erfassen und Verwalten von Anforderungen. Jedoch sind Traceability-Links ebenso im Bereich Design und Implementierung von Softwaresystemen von Bedeutung. Mit steigender Komplexität der Systeme steigen oft auch bestimmte Anforderungen wie Flexibilität, Verfügbarkeit, Robustheit an, und der Änderungsaufwand sowie Risiken erhöhen sich. Traceability-Links können Design-Entscheidungen erleichtern und das Programmverständnis erhöhen, indem sie Abhängigkeiten im System offen legen. Für eine umfassende Unterstützung von Design- und Implementierungsaktivitäten durch Traceability müssen die Traceability-Links auf feingranularer Ebene erfasst werden. Unglücklicherweise erfordern das Management und die Wartung der Links einen hohen Aufwand aufgrund der Vielzahl an Links und der Komplexität der Traceability-Informationen. Viele der Tätigkeiten müssen derzeit manuell durchgeführt werden. Methoden- und Werkzeugunterstützung ist hier eine Voraussetzung für den praktischen Einsatz, aber die Unterstützung des Traceability-Konzeptes durch die Entwicklungsmethoden ist bisher dürftig. Die Methoden sind zu unpräzise in der Definition von Artefakten und der Beschreibung von Beziehungen und Aktivitäten, um darauf Techniken für die Aktualisierung von Traceability-Links festzulegen [MPR07].

Eine Vision ist es, die Aktivitäten zur Verwaltung und Aktualisierung der Traceability-Links in die Entwicklungsmethoden und -werkzeuge zu integrieren. Die Links sollen bei der eigentlichen Entwurfs- und Implementierungstätigkeit für den Entwickler transparent im Hintergrund gezogen werden, sodass diesem keine zusätzliche Belastung durch eine parallele Pflege der Links entsteht. Eine Herausforderung, die sich daraus ergibt, ist die Verfeinerung der Beschreibung der Entwicklungsmethoden. Denn für die Nachvollziehbarkeit der einzelnen Entwicklungsschritte ist es notwendig zu wissen, welche Aktivitäten und Gründe zu welchen Entscheidungen geführt haben. Dazu dürfen die Methoden nicht lückenhaft sein. Die Verfeinerung kann jedoch nicht für alle Methoden gleich erfolgen, denn Entwurfsentscheidungen werden methodenspezifisch getroffen. Somit muss ebenfalls ein Traceability-Modell jeweils für eine konkrete Methode erstellt werden. Weiterhin ist es für die Akzeptanz dieses Ansatzes wichtig, die Verfeinerung konkret an den in der Praxis häufig verwendeten Methoden durchzuführen. Der Autor möchte in dieser Arbeit mit der kritischen Untersuchung von Quasar und

den damit verbundenen und gegebenenfalls verfeinerten Aktivitäten einen Schritt in diese Richtung beitragen.

1.2 Aufgabenstellung und Ziele

> *Bei der Überarbeitung und Erweiterung eines Softwaresystems ist die Architektur-Entwicklungsmethode Quasar anzuwenden. Die einzelnen Entwurfsentscheidungen entsprechend dieser Methode sind zu dokumentieren und es sind entsprechende Traceability-Links zu erstellen. Ziel ist einerseits die Weiterentwicklung des SalesPoint-Frameworks, wobei Entwurfsentscheidungen und Traceability-Links aufgestellt werden. Dabei kann auf die Ergebnisse der Diplomarbeit von Kristian Herpel aufgesetzt werden. Andererseits erfolgt eine kritische Untersuchung der Entwurfsmethode Quasar und eine Verfeinerung ihrer Aktivitäten zwecks Zuordnung der Traceability-Links.*

Die Aufgabe der vorliegenden Arbeit umfasst die Anwendung von Vorgaben der Standardarchitektur Quasar bei der Weiterentwicklung eines Softwaresystems. Bei diesem System, welches gleichzeitig den Entwicklungsgegenstand darstellt, handelt es sich um das SalesPoint-Framework, genauer dessen grafische Benutzerschnittstelle. Ziel dieser Arbeit ist die Entwicklung einer Benutzerschnittstelle für eine Beispielanwendung auf Basis von SalesPoint unter Anwendung der Aktivitäten und Regeln, die Quasar vorsieht. Entwurfsentscheidungen, die dabei getroffen werden, sollen anhand von Traceability-Links zu den entsprechenden Vorgaben nachvollziehbar sein. Als Ausgangspunkt für die Weiterentwicklung dient eine Version des SalesPoint-Frameworks, die von Herpel [Herp07] bereits (u. a. mittels Quasar) komponentenorientiert überarbeitet wurde.

Zudem soll eine kritische Auseinandersetzung mit Quasar erfolgen. Dabei soll untersucht werden, welche Schritte nach dieser Methode anzuwenden sind. Schwierigkeiten, die bei der Anwendung des Vorgehens auftreten, zeigen Lücken in der Methode auf. Die Quasar-Aktivitäten an dieser Stelle zu verfeinern, um die Lücken zu schließen, ist ebenso ein Ziel dieser Arbeit. Dies ermöglicht letztendlich die Erstellung der Traceability-Links unter

Berücksichtigung der Entwurfsentscheidungen der (eventuell verfeinerten) Aktivitäten und spezifisch für diese Methode.

Bei der Erstellung einer grafischen Benutzerschnittstelle stellt die Software-Ergonomie einen entscheidenden Aspekt dar. Mit der Einhaltung von Normen und Richtlinien der Software-Ergonomie kann die Usability der zu erstellenden Anwendung für den Nutzer verbessert werden. Aus diesem Grund ist auch Ziel dieser Arbeit, solche Kriterien beim Entwurf der Benutzerschnittstelle einzuhalten. Dafür müssen geeignete Richtlinien und Ansätze der Software-Ergonomie untersucht werden. Darüber hinaus sind zur Dokumentation der Entwurfsentscheidungen, die nichtfunktionalen Anforderungen, die sich aus der Software-Ergonomie ergeben, und beim Entwurf der Architektur der Beispielanwendung verwendet werden, mit entsprechenden funktionalen Eigenschaften der Benutzerschnittstelle zu verbinden.

1.3 Aufbau der Arbeit

Die vorliegende Arbeit ist wie folgt untergliedert. Das einleitende Kapitel beschreibt die Motivation und enthält die Aufgabenstellung sowie Ziele dieser Arbeit.

Im zweiten Kapitel wird der Stand der Technik analysiert und bewertet. Dabei werden als Grundlage für die Entwicklung einer Benutzerschnittstelle zunächst einige Begriffe und Richtlinien der Software-Ergonomie erläutert. Anschließend wird entsprechend der Ziele die Architekturmethode Quasar in ihren einzelnen Aktivitäten analysiert und bewertet. Zudem werden weitere Ansätze untersucht, um die Lücken im Vorgehen zu schließen.

Einen groben Überblick über das SalesPoint-Framework gibt das dritte Kapitel. Es werden die Gründe erläutert, die eine Überarbeitung des Frameworks motivieren. Danach wird der Stand der Überarbeitung als Grundlage für die weiteren eigenen Verfeinerungen festgehalten und Vendorbase als Beispielanwendung zum SalesPoint-Framework vorgestellt.

Das vierte Kapitel systematisiert das methodische Vorgehen nach Quasar. Die bereits in Quasar vorhandenen Schritte zu Entwurf und Design von Softwaresystemen werden ange-

wendet und an bestimmten Stellen durch weitere Aktivitäten verfeinert. Die Illustration des Vorgehens erfolgt an einem durchgängigen Beispiel zur Neuentwicklung von Vendorbase.

Im fünften Kapitel wird ein Resümee gezogen und die erreichten Ergebnisse bei der Anwendung der Methode werden gegen die Ziele geprüft. Zum Abschluss wird ein Ausblick für zukünftige Arbeit gegeben.

2 Stand der Technik

In diesem Kapitel werden zunächst die Grundlagen für diese Arbeit erläutert. Dazu werden Aspekte zur Software-Ergonomie sowie die Entwicklungsmethode Quasar vorgestellt. Die charakterisierenden Gesichtspunkte von Quasar werden herausgestellt, wichtige Begriffe erläutert und das methodische Vorgehen nach Quasar analysiert und bewertet. Weiterhin werden ergänzende Methoden zum Design von Benutzerschnittstellen untersucht.

2.1 Software-Ergonomie

Der Gegenstand dieser Arbeit umfasst zum großen Teil die Erstellung einer neuen grafischen Benutzeroberfläche für eine Beispielanwendung zum SalesPoint-Framework. Bei der Erarbeitung eines neuen Konzeptes für eine grafische Benutzerschnittstelle spielt die Benutzerfreundlichkeit eine nicht unwesentliche Rolle. Informationen diesbezüglich, sind im Gebiet der Software-Ergonomie zu finden.

Die Software-Ergonomie (englisch: *Usability Engineering*) ist eine Wissenschaft auf dem Gebiet der Mensch-Computer-Kommunikation (englisch: *Human-Computer Interaction* (HCI)), die sich mit der Benutzbarkeit und Gebrauchstauglichkeit (englisch: *Usability*) von Computer-Programmen beschäftigt und Leitlinien sowie Rahmenbedingungen zur benutzergerechten Gestaltung bereitstellen soll. Es wird die Interaktion des Menschen mit Anwendungssystemen bzw. allgemein Computern betrachtet. Problemstellungen der Software-Ergonomie erfordern die Einbeziehung anderer Fachgebiete wie Psychologie, Physiologie oder Arbeitswissenschaften. Die nutzbaren Ergebnisse der Software-Ergonomie sind unterschiedlichste Hilfsmittel wie [Herc94, S. 4]:

- Normen (nationale und internationale Standards),
- Empfehlungen (Vorschläge),
- Design-Regeln (Richtlinien),
- Tools (Software-Bausteine und Entwicklungswerkzeuge).

Eine der wichtigsten Normen auf dem Gebiet der Software-Ergonomie ist die DIN EN ISO 9241 „Ergonomie der Mensch-System-Interaktion“. Sie ist als internationale, europäische sowie als deutsche Norm gültig. Für die Gestaltung von Dialogen in Anwendungssystemen ist der Teil 110 „Grundsätze der Dialoggestaltung“ maßgebend. Er wurde im Jahr 2006 eingeführt und ersetzt den bisherigen Teil 10.

Darin definiert ist der Begriff Benutzungsschnittstelle, auch Benutzerschnittstelle, als:

> *„Alle Bestandteile eines interaktiven Systems (Software oder Hardware), die Informationen und Steuerelemente zur Verfügung stellen, die für den Benutzer notwendig sind, um eine bestimmte Arbeitsaufgabe mit dem interaktiven System zu erledigen.“* [DIN9241, S. 7]

Da sich die vorliegende Arbeit mit Software-Architektur und nicht mit Hardware beschäftigt, soll sich der Begriff nur auf den Software-Anteil beziehen. Der englische Begriff User Interface wird synonym verwendet.

Der Begriff Benutzungsoberfläche oder Benutzeroberfläche bezeichnet den Teil der Benutzerschnittstelle, der nach außen hin sichtbar ist [VN98, S. 2]. Dies sind die visuelle Darstellung und das Verhalten dem Nutzer gegenüber. Im Zusammenhang mit Quasar fallen hier gleichberechtigt die Begriffe Maske und visuelle Präsentation.

Es werden folgende Arten von Benutzerschnittstellen unterschieden [Chl06, S. 40f.]:

- Kommandozeilen,
- zeichenbasierte Benutzerschnittstellen,
- grafische Benutzerschnittstellen,
- sprachbasierte Benutzerschnittstellen,
- haptische Benutzerschnittstellen.

In dieser Arbeit ist nur die grafische Benutzerschnittstelle von Bedeutung, die auch kurz mit dem Akronym GUI (englisch: *Graphical User Interface*) benannt wird.

Teil 110 der ISO 9241 definiert einige Mindestanforderungen und Gestaltungsgrundsätze an grafische Benutzerschnittstellen. Die in der Norm definierten Qualitätskriterien sind [DIN9241]:

- Aufgabenangemessenheit,
- Selbstbeschreibungsfähigkeit,
- Erwartungskonformität,
- Lernförderlichkeit,
- Steuerbarkeit,
- Fehlertoleranz,
- Individualisierbarkeit.

Sie „*dienen als eine Zusammenstellung allgemeiner Ziele für die Gestaltung und Bewertung von Dialogen*“ [DIN9241]. Die Kriterien sind eher unscharfe Empfehlungen und sehr abstrakt formuliert. Ein wenig konkreter und leichter anwendbar sind die „*Acht goldenen Regeln des Dialogdesigns*“ von Shneiderman [Shn92], die sich weitgehend in der ISO-Norm wiederfinden [Herc94, S. 103ff.].

1. Versuche Konsistenz zu erreichen.
2. Biete erfahrenen Benutzern Abkürzungen an.
3. Biete informatives Feedback.
4. Dialoge sollen abgeschlossen sein.
5. Biete einfache Fehlerbehandlung.
6. Biete einfache Rücksetzmöglichkeiten.
7. Unterstütze benutzergesteuerten Dialog.
8. Reduziere die Belastung des Kurzzeitgedächtnisses.

Diese Regeln können als Leitfaden für Entwickler beim Design von Benutzungsschnittstellen dienen.

Eine genaue Auflistung von Faktoren, welche die Usability beeinflussen (englisch: *usability factors*) gibt Lauesen [Lau05, S. 9] an.

a) „Fit for use (or functionality). The system can support the tasks that the user has in real life.
b) Ease of learning. How easy is the system to learn for various groups of users?
c) Task efficiency. How efficient is it for the frequent user?

d) Ease of remembering. How easy is it to remember for the occasional user?
e) Subjective satisfaction. How satisfied is the user with the system?
f) Understandability. How easy is it to understand what the system does?"

Diese Faktoren basieren ebenfalls auf den Regeln von Shneiderman und sollen es ermöglichen, die Usability von Softwaresystemen beispielsweise mithilfe von Usability-Tests messbar zu machen.

Von entscheidender Bedeutung für die Präsentation von Daten in Benutzungsschnittstellen sind auch die Gestaltgesetze. Sie beziehen sich auf die menschliche Wahrnehmung von Objekten. Beispiel hierfür sind [Lau05, S. 68ff.]:

- das Gesetz der Nähe,
- das Gesetz der Geschlossenheit,
- das Gesetz der guten Fortsetzung oder
- das Gesetz der Ähnlichkeit.

Diese Gesetze zur visuellen Wahrnehmung des menschlichen Gehirns überlagern sich gegenseitig und haben signifikanten Einfluss darauf, wie ein Dialog der Schnittstelle auf einen Nutzer wirkt. Daher lassen sich aus ihnen Prinzipien zur Dialoggestaltung und Positionierung von Bedienelementen auf Dialogen wie Schaltflächen, Werkzeugleisten, Auswahllisten sowie die Vor- und Nachteile von verschiedenen Präsentationsformaten ableiten.

Als Präsentationsformate ohne Anspruch auf Vollständigkeit nennt Lauesen [Lau05, S. 85ff.]:

- klassisches Formular mit Datenfeldern,
- Listenformat,
- Detailfenster,
- Matrixformat,
- Kartenformat,
- Hierarchien,
- Baum (*explorer tree*),
- verschachtelte Menüs,
- verschiedene Diagrammtypen.

Aus Präsentationsmöglichkeiten dieser Art müssen für eine konkrete Benutzeroberfläche die geeignetsten ausgewählt werden, um den unterschiedlichen Usability-Kriterien gerecht zu werden.

Eine Aufstellung von hilfreichen grundsätzlichen Prinzipien zum User Interface Design findet sich in [Gal02, S. 40ff.]. Zu diesen Grundsätzen zählen:

Aesthetically Pleasing	Directness	Recovery
Clarity	Efficiency	Responsiveness
Comprehensibility	Familiarity	Simplicity
Configurability	Flexibility	Transparency
Consistency	Forgiveness	Trade-Offs
Control	Predictability	

Die Prinzipien sollen hier nicht näher erläutert werden, dazu wird auf die Literatur verwiesen. Dort finden sich auch detaillierte Beschreibungen zu den Vor- und Nachteilen, Verwendungsmöglichkeiten und Auswirkungen der unterschiedlichen Bedienelemente und Beispiele für schlechtes und gutes Design von Benutzerschnittstellendialogen.

2.2 Quasar

Der Begriff Quasar steht für Qualitätssoftwarearchitektur. Er wurde bei sd&m (software design & management) geprägt und bezeichnet eine Standardarchitektur für die Entwicklung von Softwaresystemen [Sie04]. Quasar beschreibt Regeln und Mechanismen, die methodisch angewandt dem Design von hochqualitativen Informationssystemen dienen sollen. Dies umfasst bewährte Konzepte und Ideen der Softwaretechnik wie die Trennung der Zuständigkeiten (*separation of concerns*) durch Festlegung von Softwarekategorien, das konsequente Denken in Komponenten und Schnittstellen und die Definition wichtiger Begriffe für ein einheitliches Verständnis. Zur Beschreibung der Standardarchitektur gehört weiterhin die Defi-

nition von Standardschnittstellen für verschiedene Gesichtspunkte wie die grafische Benutzerschnittstelle oder die Persistenzschicht eines Softwaresystems.

Das methodische Vorgehen in Quasar zum Entwurf einer Komponentenarchitektur basiert auf der Spezifikation eines Daten-, Anwendungsfall- sowie Kategorienmodells [Ade07]. Nachfolgend werden die einzelnen Aspekte des durch Quasar definierten Vorgehens wie z. B. die Identifikation und Spezifikation von Komponenten mittels Aufteilung auf Softwarekategorien und die Festlegung von Schnittstellen analysiert und bewertet.

2.2.1 Softwarekategorien

Mit Quasar wird ein zu entwerfendes Softwaresystem zunächst auf Abhängigkeiten und Zuständigkeiten hin untersucht. Dies geschieht mithilfe der Softwarekategorien [Sie04, S. 73ff.]. Die Kategorien sind eine Vorstufe von Komponenten und bilden eine Richtlinie bei deren Entwurf sowie bei späteren Änderungen am System.

Quasar definiert fünf primäre Kategorien zur Trennung der Zuständigkeiten auf oberster Abstraktionsebene eines Anwendungssystems [Sie03a, S. 3]. Zu diesen Kategorien zählen 0-Software, die als Basis für alle anderen Kategorien dient, weiterhin A-Software, T-Software sowie die Mischformen AT-Software und R-Software.

- 0-Software ist komplett unabhängig vom eigentlichen Anwendungssystem, d. h. unabhängig von Anwendung und eingesetzter Technik. Dazu zählen alle Module, Klassen und Schnittstellen die Allgemeingültigkeit und einen hohen Grad an Wiederverwendung besitzen. Sie steht dem gesamten System zur Verfügung, erzeugt keine unerwünschten Abhängigkeiten, ist sorgfältig getestet und besitzt eine geringe Wahrscheinlichkeit für Änderungen. Beispiele sind Klassenbibliotheken wie die Laufzeitumgebung (JRE – Java Runtime Environment) bei Java-Programmen.
- A-Software ist anwendungsspezifisch, vollkommen technikunabhängig und enthält alle verfeinerten Softwarekategorien, Klassen und Schnittstellen, die sich mit der Anwendungslogik beschäftigen. Dazu zählen ebenso alle Entitäten wie Vertrag, Auto

usw. für die Umsetzung der Domainfunktionalität. Dies ist in der Regel der größte Teil des Softwaresystems.

- T-Software ist unabhängig von der Anwendung und behandelt die technischen Aspekte zur Realisierung. T-Software-Bausteine besitzen Implementierungen für Programmierschnittstellen (API – Application Programming Interface) z. B. zur Anbindung von Datenbanken.
- AT-Software als Mischform enthält sowohl anwendungsspezifische Teile als auch den technischen Code des Anwendungssystems. Diese unreine Art von Software verhindert eine klare Trennung der Zuständigkeiten, wodurch eine separate Betrachtung und Änderung von fachlichen und technologischen Anforderungen schwer fällt. AT-Software sollte vermieden werden, außer, sie wirkt als R-Software.
- R-Software dient nur der Transformation von fachlichen Bestandteilen in externe Repräsentationen oder umgekehrt und übernimmt keine sonstigen fachlichen Funktionen. Externe Darstellungsformen sind beispielsweise XML-Formate[1] oder Datenbanktabellen. R-Software als Vermengung der Kategorien A und T ist oft unvermeidlich aber dafür oft generierbar.

Die erläuterten Hauptkategorien können beim Systementwurf und der Analyse der Abhängigkeiten durch eigene Kategorien verfeinert werden. Als Ausgangspunkt dazu dienen jedes Mal die fünf primären Kategorien. Dargestellt werden die Softwarekategorien in einem zyklenfreien Kategoriengraphen, dessen Wurzel immer die Kategorie 0 ist. Die anwendungsspezifischen Kategorien können eine oder mehrere andere Kategorien verfeinern und werden demnach in reine und unreine Kategorien unterteilt. [Sie04, S. 78]

Sind die Kategorien definiert, können ihnen die Komponenten zugeordnet werden. Ziel dabei ist es, eine Komponente genau einer Kategorie zuzuordnen, um ungewollte Abhängigkeiten zwischen den Komponenten zu vermeiden. Für die Kommunikation zwischen Komponenten der einzelnen Kategorien im Graphen gibt es strenge Sichtbarkeitsregeln [Sie04, S. 80ff.]. Diese sollen eine Vermischung der Kategorien vermeiden und der Einhaltung der Trennung der Zuständigkeiten dienen.

- Komponenten in Kategorien höherer Ebene sehen Komponenten niedrigerer Kategorien. In umgekehrter Richtung gilt dies nicht. Für alle Kategorien ist damit die Kate-

[1] XML – Extensible Markup Language, siehe http://www.w3.org/XML/

gorie 0 sichtbar und alle Komponenten können auf 0-Software zugreifen, 0-Software selbst aber nur auf Komponenten ihrer eigenen Kategorie.

- Für Komponenten einer Kategorie gilt weiterhin, dass sie nur Schnittstellen der eigenen oder einer verfeinerten Kategorie importieren oder exportieren dürfen. Die Komponenten einer hohen Kategorie dürfen folglich mit Komponenten einer niedrigeren verfeinerten Kategorie nur über deren Schnittstellen kommunizieren. Die Verwendung von 0-Schnittstellen und 0-Komponenten ist dabei absolut problemlos möglich.
- Gibt es keine direkte Verbindung zwischen Komponenten verschiedener Kategorien im Kategoriengraphen, kann die Kommunikation zwischen diesen Kategorien nur über Schnittstellen einer dritten Kategorie erfolgen. Die Kategorie der Schnittstelle muss ein gemeinsamer Vorgänger der Kategorien der beiden kommunizierenden Komponenten sein, die also von beiden Kategorien verfeinert werden. Wenn kein solcher gemeinsamer Vorgänger existiert, ist eine Transformation mittels R-Software erforderlich.

Durch die Definition von Softwarekategorien für ein Anwendungssystem werden die Abhängigkeiten zwischen den einzelnen Softwarebausteinen untersucht. Dies fördert die Trennung nach Verantwortlichkeiten (*separation of concerns*), da die Zuständigkeiten durch die Festlegung der Kategorien explizit gemacht werden. Durch diese klare Aufteilung und die Kontrolle der Abhängigkeiten wird die Flexibilität bei Änderungen erhöht und leichter planbar. So gibt es beispielsweise keine unerwünschten Auswirkungen auf die Anwendungslogik, wenn technologische Aspekte wie das verwendete GUI-Framework geändert werden. Mit den Kategorien kann genau beurteilt werden, welche Teile des Systems betroffen sind. Die Trennung zwischen A- und T-Software ist wichtig, weil sich die fachlichen Anforderungen sowie die technischen Grundlagen ändern können, dies allerdings aufgrund anderer Innovationszyklen mit unterschiedlichem Rhythmus. Es werden Abhängigkeiten vermieden, die die Wartung be- oder verhindern können. Die Softwarekategorien sind damit ein Mittel, um die Wartung von Softwaresystemen, die als kritischer Punkt im ersten Abschnitt motiviert wurde, zu erleichtern.

Abgesehen von den Kriterien zur Zuordnung zu den Hauptkategorien und den Festlegungen zur Kommunikation zwischen einzelnen Kategorien, ist jedoch eines etwas unklar in der Quasar-Beschreibung zum Festlegen der Softwarekategorien. Wie genau werden diese Kate-

gorien aus den Anforderungen abgeleitet? Eine Möglichkeit dieses Vorgehen mithilfe eines Funktionsbaumes zu vervollständigen wird im Abschnitt 4 erläutert.

2.2.2 Datenmodell

Ein weiterer Punkt auf dem Weg zur Komponentenarchitektur ist zunächst die Erstellung des Datenmodells. Die Datenmodellierung wird in Quasar als klassische Entity-Relationship-(ER)-Modellierung betrachtet [Sie03, S. 56]. Das bedeutet die Entitäten einer Anwendung und deren Beziehungen stehen im Mittelpunkt. Die zentralen Begriffe der Anwendung wie *Lieferant* oder *Liefervertrag* werden also als Entitätstypen mit Attributen wie *Name* und *Lieferdatum* beschrieben. Die Beziehungen zwischen Entitäten werden durch Beziehungstypen mit Angabe von Kardinalitäten charakterisiert. Jedes Attribut hat entweder einen Standarddatentyp wie eine Zeichenkette oder einen fachlichen Datentyp wie Adresse, der auch zusammengesetzt sein kann. Jeder Entitätstyp hat einen fachlichen Schlüssel, Datentypen hingegen nicht. [Sie04, S. 176]

Bei der Datenmodellierung werden zunächst die wichtigsten Entitätstypen und die wichtigsten Attribute formuliert und als erstes Ergebnis veröffentlicht. Später erfolgt die Festlegung der Datentypen und ein Abgleich des Datenmodells mit anderen Bausteinen wie Anwendungsfällen, Dialogen, Nachbarsystemen. Die Vollständigkeit des Modells kann daran erkannt werden, dass jeder Entitätstyp und jedes Attribut mindestens einmal verwendet wird. [Sie03, S. 57]

Mittels UML[2] kann das Datenmodell als Klassenmodell notiert werden. Die Klassen entsprechen den Entitätstypen und dürfen keine Methoden enthalten, sondern nur Attribute. Beziehungen werden als Assoziationen oder Aggregationen modelliert. Da es für Datentypen in UML kein spezielles Element gibt, wird in [SSKK02] die Einführung von Stereotypen wie *<<enumeration>>*, *<<range>>* und *<<structure>>* zur Kennzeichnung von Klassen vorgeschlagen. Zum Vorgehen bei der Implementierung von Entitäts- und Datentypen am Beispiel von Java sei auf [Sie04, S. 178ff.] verwiesen, wo dies beispielhaft gezeigt wird. Da die Daten-

[2] UML – Unified Modeling Language, http://www.uml.org/

modellierung als ER-Modellierung schon lange Zeit angewendet wird und ausgereift ist, sind in diesem Bereich im Quasar-Vorgehen keine Schwächen auszumachen.

2.2.3 Anwendungsfallmodell

Zur Anforderungsanalyse und damit zur Erstellung des Anwendungsfallmodells hält Quasar nur sehr wenige Informationen bereit. Für die Ermittlung und Strukturierung der Anforderungen an ein zu entwickelndes Softwaresystem wird lediglich beispielhaft auf Volere verwiesen [Sie04, S. 1f.]. Volere umfasst eine Sammlung von Hilfsmitteln und Materialien wie etwa Schablonen (*Templates*) zur Anforderungsbeschreibung. Weiterhin wird dort ein *Requirements Process* vorgeschlagen [RR99]. Die Anwendungsfälle (englisch: *use cases*) werden von Quasar im Sinne der UML verwendet und können somit in Anwendungsfalldiagrammen dargestellt werden [Sie04, S. 167]. Sie finden sich später in den Methoden von Komponenten der Kategorie A wieder und werden aus Dialogkomponenten heraus aufgerufen.

2.2.4 Komponentenmodell

Der Begriff Komponentenmodell wird vom Autor in der Bedeutung verwendet, wie er in [Ade07] vorkommt. Das Komponentenmodell von Quasar beschreibt die Zerlegung des Anwendungssystems in einzelne Softwarebausteine. Diese Bausteine sind Komponenten, die über Schnittstellen ihre Funktionalität zur Verfügung stellen und miteinander interagieren. Es ist folglich als ein Modell zu sehen, welches aus Anwendungsfallmodell, Datenmodell sowie Kategorienmodell hervorgeht, und mit dem Komponenten und ihre Schnittstellen entworfen werden. Diese Bedeutung ist daher klar von derjenigen abzugrenzen, die ein Komponentenmodell als konkrete Ausprägung des Paradigmas der komponentenbasierten Entwicklung versteht, welche u. a. eine Infrastruktur zur Ausführung von Komponenten vorsieht. Beispiele hierfür wären Enterprise Java Beans oder das Distributed Component Object Model (DCOM).

Bei der Definition einer Komponente lehnt sich Quasar an verschiedene Definitionen anderer Autoren wie Szyperski [Szy02] sowie die Definition der UML an und extrahiert daraus sechs charakterisierende Merkmale [Sie04, S. 42f.]:

1. Eine Komponente exportiert eine oder mehrere Schnittstellen. Über diese Schnittstellen können andere Komponenten deren Funktionalität abrufen. Für jede Schnittstelle, die eine Komponente exportiert, ist sie auch eine Implementierung.
2. Eine Komponente kann andere Schnittstellen importieren, die sie benutzt, um ihre eigenen Dienste zu verrichten. Das bedeutet, die Komponente ist nur dann lauffähig, wenn sie die benötigten Schnittstellen von anderen Komponenten zur Verfügung gestellt bekommt. Die Verknüpfung ist Aufgabe der Konfiguration.
3. Die Implementierung einer Schnittstelle wird durch die Komponente versteckt. Dadurch kann die Komponente durch eine andere ausgetauscht werden, die dieselbe Schnittstelle exportiert.
4. Komponenten kennen ihre Umgebung nicht, machen darüber nur minimale Annahmen und sind deshalb als Einheit wiederverwendungsfähig.
5. Komponenten können Teil einer Komposition sein, d. h., sie können aus anderen Komponenten über mehrere Stufen zusammengesetzt werden.
6. Eine Komponente ist als Einheit für Entwurf, Implementierung und Planung zu sehen.

Für die Beschreibung von Komponenten empfiehlt Quasar ein schrittweises Vorgehen. Dabei werden der Reihe nach folgende Dokumentationen erstellt:

1. *Übersicht*: Hier wird die Idee wiedergegeben, auf der die Komponente basiert. Es werden für die Sicht eines Managers Kosten und Nutzen der Komponente erläutert.
2. *Außensicht*: Sie ist wie ein Benutzerhandbuch geschrieben und für die Entwickler sowie für die Benutzer der Komponente von Bedeutung. Hier werden die Schnittstellen der Komponente beschrieben und ihre Konfiguration.
3. *Innensicht*: Die Innensicht beschreibt für die Entwickler den inneren Aufbau der Komponente, wie sie komponiert ist und wichtige Implementierungsdetails.
4. *Variabilitätsanalyse*: Hierbei werden Änderungsszenarien aufgeführt. Es wird beschrieben:
   - für welche Änderungen die Komponente bereits vorbereitet ist,

- welche Änderungen konform zur Architektur sind und nur geringen Änderungsaufwand bedeuten (z. B. die Verwendung eines anderen Datenbankmanagementsystems) sowie
- die Änderungen, die nicht so leicht mit der Architektur vereinbar sind und einen kompletten Um- oder Neubau der Komponente bedeuten.

Gerade der letzte Punkt, die Variabilitätsanalyse, ist für die spätere Wartung eines Softwaresystems von enormer Bedeutung, da hier bereits bei der Entwicklung Informationen über mögliche Änderungen hinterlegt werden und somit deren Aufwand leichter einschätzbar ist. Darüber hinaus könnten in einer Wartungsphase durchgeführte Änderungen über Traceability-Links mit den beschriebenen Szenarien verknüpft werden. Außensicht und Innensicht sind weiterhin entscheidende Dokumentationen für Entwickler bzw. Nutzer der Komponente. Hingegen ergibt die Übersicht als eigenes Dokumentationsobjekt mit Kosten-Nutzen-Vergleich nur in größeren realen Projekten wirklich Sinn. Es sollte daher individuell über den Umfang und die Elemente der Komponentenbeschreibung entschieden werden. Für die vorliegende Arbeit mit der Beispielanwendung kann die Idee einer Komponente ebenso kurz in der Außensicht erläutert werden und die Übersicht kann entfallen.

Damit Außen- und Innensicht einer Komponente beschrieben werden können, muss diese zunächst entworfen werden. Die Aktivitäten beim Entwurf werden nachfolgend dargestellt. Dazu ist eine kurze Anmerkung zum Schnittstellenbegriff notwendig. Schnittstellen werden bei Quasar unterschieden nach Programmschnittstellen zwischen Komponenten untereinander und Benutzerschnittstellen zwischen Komponenten und dem Benutzer (siehe Spezifikation von Schnittstellen in Abschnitt 2.2.4). Im weiteren Verlauf ist im Zusammenhang mit Komponenten immer von Programmschnittstellen die Rede, wenn nicht anders angegeben.

Der Komponentenentwurf nach Quasar untergliedert sich in vier Teilschritte [Sie04, S. 57ff.]:

1. *Entwurf der Schnittstelle*

   Für eine Komponente müssen in diesem Schritt alle importierten und exportierten Schnittstellen festgelegt werden. Dabei wird oft zwischen zwei Arten von Schnittstellen unterschieden. So gibt es operative Schnittstellen, über welche die angebotenen Funktionalitäten der Komponente ausgeführt werden, und nach Bedarf administrative Schnittstelle zur Konfiguration.

2. *Validierung der Schnittstelle*

   In diesem Schritt ist sicherzustellen, dass die Schnittstelle so einfach wie möglich ist. Die Komponente wird auf minimale Abhängigkeiten hin überprüft. Das bedeutet, Importe und Exporte sollen minimal sein, sodass gerade alle Anforderungen erfüllt werden. Die Prüfung erfolgt mithilfe von formalen Reviews und Anwendungsszenarien.

3. *Validierung der Implementierung*

   Hier wird die Implementierbarkeit der Komponente überprüft. Ausschlaggebend sind technische und nichtfunktionale Faktoren. Bei der Prüfung können Dummy-Implementierungen hilfreich sein.

4. *Konfiguration*

   Bei der Konfiguration werden Importeur und Exporteur einer Schnittstelle miteinander verbunden. Dies kann geschehen durch direkten Konstruktoraufruf, indirekt über eine Fabrik (siehe [GHJV95]) oder über einen Namensdienst. Für jede Komponente gibt es einen Kompositionsmanager, der diese Konfiguration (und gegebenenfalls eine Ausnahmebehandlung) übernimmt. Die Konfiguration kann direkt programmiert sein oder über eine Konfigurationsdatei z. B. im XML-Format festgelegt werden. Sie entscheidet auch darüber, ob im konkreten Fall mit der Dummy-Implementierung oder der richtigen Implementierung gearbeitet wird.

Die geschilderten Schritte zum Komponentenentwurf sind in ihrer Anwendung schlüssig. (Details zur Schnittstellenspezifikation folgen im nächsten Abschnitt.) Allerdings geht daraus noch nicht hervor, wie die richtigen Komponenten und ihre Schnittstellen eigentlich gefunden werden. Dies liegt daran, dass die Methodenbeschreibung deutlich mehr die Modelle an sich beschreibt als die Übergänge zwischen den Modellen. Für die Identifikation der Komponenten ist der Übergang vom Datenmodell, Anwendungsfallmodell und Kategorienmodell zum Komponentenmodell entscheidend. Hier helfen die in [Ade07] analysierten Konsistenzbedingungen zwischen den Modellen. Komponenten bzw. Schnittstellen[3] ergeben sich danach entsprechend folgenden Kriterien:

- Jeder Entitätstyp aus dem Datenmodell wird exklusiv einer Komponente zugeordnet. Für die Referenzierung von Entitätstypen anderer Komponenten wird daher eine Entitätsschnittstelle (siehe [Sie04, S. 178]) benötigt.

[3] Im Zusammenhang mit den Konsistenzbedingungen steht der Schnittstellenbegriff für beide Arten, Programm- und Benutzerschnittstellen, wenn nicht eine explizit genannt wird.

- Schnittstellen gehören zu genau einer Softwarekategorie, Komponenten sind dagegen mindestens einer Kategorie zugeordnet.
- Werden von einer Komponente Programmschnittstellen importiert oder exportiert, so muss die Komponente der gleichen oder einer spezielleren Softwarekategorie angehören.
- Jeder konkrete Anwendungsfall wird einer Schnittstelle exklusiv zugeordnet, aber eine Schnittstelle kann mehrere Anwendungsfälle behandeln.
- Hat ein Anwendungsfall eine direkte Verbindung zu einem Akteur, so werden dieser sowie daraus spezialisierte oder erweiterte Anwendungsfälle einer Benutzerschnittstelle zugeordnet.
- Für jede Assoziation (<<*include*>>, <<*extend*>>) zwischen konkreten Anwendungsfällen gibt es eine Abhängigkeit im Komponentenmodell. Bei einer include-Beziehung existiert eine Abhängigkeit zwischen der Schnittstelle des importierten Anwendungsfalles und der Komponente, welche die Schnittstelle des importierenden Anwendungsfalles realisiert. Eine extend-Beziehung führt zu einer Abhängigkeit zwischen der Schnittstelle für den erweiterten Anwendungsfall und der Komponente, welche die Schnittstelle des erweiternden Anwendungsfalls exportiert.

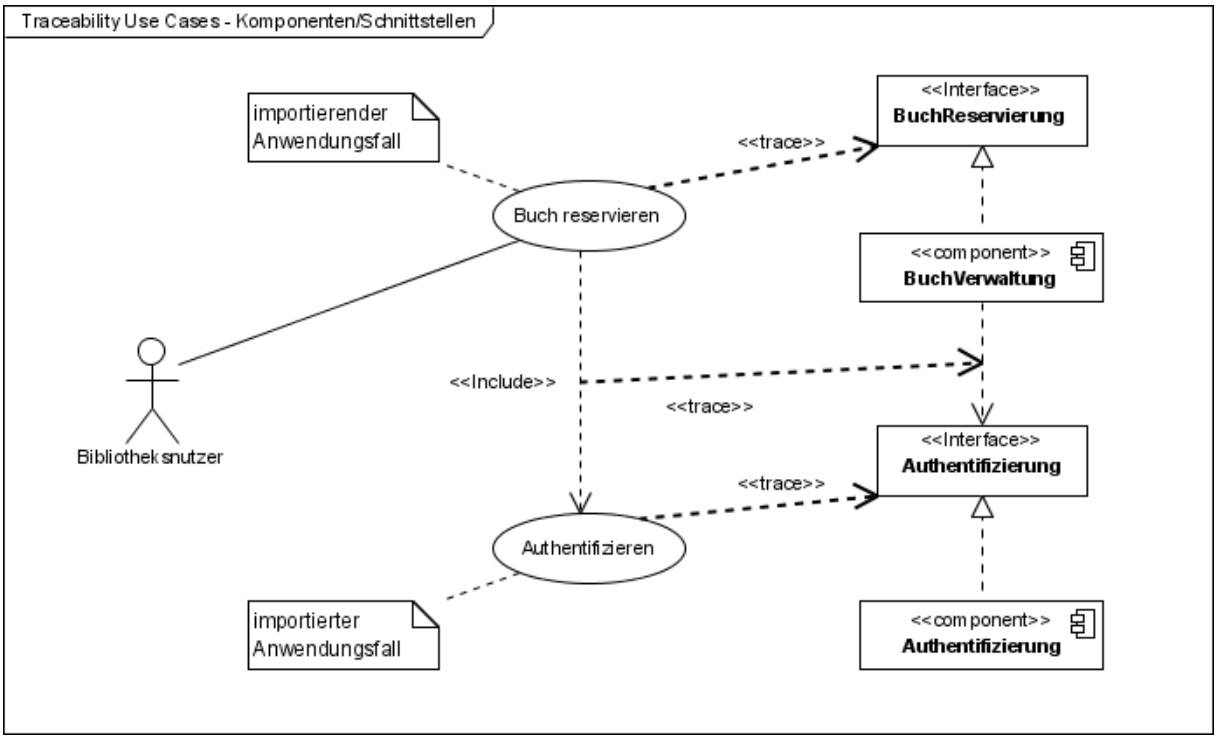

Abbildung 2-1: Traceability-Links vom Anwendungsfallmodell zum Komponentenmodell

Abbildung 2-1 zeigt ein Beispiel wie Anwendungsfälle in Schnittstellen abgebildet werden und welche Traceability-Links sich dabei ergeben. Es wird auch gezeigt, dass eine include-

Beziehung entsprechend der letztgenannten Konsistenzbedingung in einer Abhängigkeit zwischen der Schnittstelle *Authentifizierung* und der Komponente *BuchVerwaltung* resultiert.

Bei der Identifikation der Komponenten ist weiterhin Folgendes zu beachten. Nicht jeder einzelne Anwendungsfall wird durch eine eigene Komponente implementiert, denn Komponenten sollen funktionale Einheiten darstellen. Sie vereinigen ähnliche oder gemeinsam genutzte Funktionalitäten [Herp07, S. 79]. Daher müssen gegebenenfalls mehrere Anwendungsfälle zu Aufgabenblöcken zusammengefasst werden. Diese Aufgabenblöcke werden dann durch Komponenten und ihre Schnittstellen erfüllt.

Spezifikation von Schnittstellen

Dieser Abschnitt widmet sich noch einmal dem ersten Schritt beim Vorgehen zum Entwurf von Komponenten aus dem letzten Unterkapitel. Die Abfolge der Tätigkeiten beim Entwurf von Schnittstellen soll genauer dargelegt werden. Dazu wird zunächst der Schnittstellenbegriff, wie ihn Quasar verwendet, untersucht.

Schnittstellen werden in Quasar an verschiedenen Stellen nach unterschiedlichen Kriterien unterschieden. So werden zum einen Benutzerschnittstellen, bei denen ein Benutzer mit dem System interagiert, von Programmschnittstellen zur Kommunikation von Komponenten untereinander getrennt [Sie04, S. 44f]. Des Weiteren werden Programmschnittstellen danach kategorisiert, wer die Schnittstellen definiert [Sie04, S. 66ff.] oder welchen Grad der Kopplung die Beziehungen der Komponenten haben, die sie verwenden [Sie04, S. 184ff.].

Programmschnittstellen existieren für sich selbst unabhängig von Komponenten. Eine Unterscheidung erfolgt danach, woher sie kommen. Wird eine Programmschnittstelle von keiner Komponente spezifiziert, so handelt es sich um eine Standardschnittstelle. Legt der Exporteur die Schnittstelle fest, ist es eine angebotene. Erfolgt die Spezifikation in der importierenden Komponente, ist es eine angeforderte Schnittstelle. Definieren Importeur mit einer angeforderten und Exporteur mit einer angebotenen, zwei unterschiedliche Schnittstellen, so wird ein Adapter benötigt, der zwischen beiden vermittelt.

Bei der Spezifikation der Programmschnittstellen ist weiterhin danach zu differenzieren, ob zwischen den Komponenten, die sie verwenden, eine enge oder lose Kopplung herrschen soll. Bei Festlegung von enger Kopplung werden objektorientierte Schnittstellen erstellt, bei loser Kopplung dienstorientierte Schnittstellen. Die Unterschiede zwischen beiden Typen lassen sich an den Parametertypen festmachen. So dürfen in objektorientierten Schnittstellen Datentypen und Entitätsschnittstellen [Sie04, S. 178] verwendet werden, in dienstorientierten Schnittstellen hingegen nur Datentypen und Transportklassen. Zu beachten ist außerdem, dass Entitätstypen in keiner Parameterliste der beiden Schnittstellenarten auftreten dürfen. Quasar sieht vor, im Entwicklungsprozess zunächst nur objektorientierte Schnittstellen zu entwerfen und dienstorientierte bei Bedarf durch Adapter nachzuliefern. Die Entscheidung, ob Komponenten eng oder lose gekoppelt sind, ist jedoch explizit zu machen und nicht zufällig zu fällen.

Programmschnittstellen definieren eine Menge von Methoden. Diese Methoden haben verschiedene charakterisierende Eigenschaften. Dazu zählen:

- Syntax (Rückgabewerte, Parametertypen),
- Semantik (Wirkungsweise der Methoden),
- Protokoll (synchroner/asynchroner Ablauf) und
- nichtfunktionale Eigenschaften (z. B. Performance, Robustheit).

Wie die Schnittstellen spezifiziert werden können, wurde z. B. in [Herp07] untersucht, auch [Sie04, S. 119ff.] gibt einen Überblick. Ein allgemeingültiges Verfahren dafür gibt es noch nicht. Für die Syntax kann z. B. die Programmiersprache verwendet werden, in der implementiert wird. Ein kritischer Punkt bei der Spezifikation stellt aber die Semantik dar. Quasar verfolgt aus diesem Grund für die Spezifikation den Ansatz von Design-by-Contract, der auf Meyer [Mey97] zurückgeht. Demnach sind Schnittstellen Verträge, an die sich die Komponenten halten müssen, wenn sie die Schnittstellen verwenden wollen. Die Spezifikation von Programmschnittstellen erfolgt nach Quasar mittels QSL (Quasar Specification Language). QSL ist eine halbformale Notation, die zwar nicht so präzise aber leichter zu verwenden ist als beispielsweise OCL[4], jedoch eine genauere Spezifikation ermöglicht als vergleichsweise Java-Interfaces mit Javadoc-Kommentaren.

[4] OCL – Object Constraint Language ist Bestandteil der UML.

Mittels QSL kann die Semantik einer Schnittstelle festgelegt werden. Die Semantik gibt an, was genau die einzelnen Methoden der Schnittstelle bewirken. Dazu werden die Methoden unterschieden nach:

- einfachen Abfragen (zum Erfragen des Zustandes einer Schnittstelle),
- abgeleiteten Abfragen (verwenden einfache Abfragen und werden mit ihrer Hilfe definiert),
- Kommandos (ändern den Zustand einer Schnittstelle).

Diese Aufteilung ist wichtig für das Vorgehen bei der Spezifikation. Die einzelnen Elemente, die zur Spezifikation von Schnittstellen als Verträge genutzt werden, sind [Sie04, S. 123ff.]:

- Vor- und Nachbedingungen,
- Invarianten,
- Zustandsmodelle,
- Testfälle sowie
- sonstige Angaben wie mögliche Fehler.

Die einzelnen Elemente werden mittels QSL notiert und so die syntaktische Beschreibung einer Schnittstelle um Semantik angereichert. Als Reihenfolge beim Erarbeiten der Spezifikationselemente für eine Schnittstelle schlägt Quasar fünf Schritte vor [Sie04, S. 126]:

1. Spezifizieren der einfachen Abfragen mit ihren Vorbedingungen,
2. Spezifizieren der abgeleiteten Abfragen in Abhängigkeit der einfachen Abfragen,
3. Spezifizieren der Kommandos mit Vor- und Nachbedingungen,
4. Formulieren der Invarianten,
5. Formulieren von Testfällen.

Die Annotation der Spezifikationselemente zur Anreicherung von Schnittstellenbeschreibungen mit Semantik erfolgt mit QSL direkt in der Zielsprache. Bei Verwendung von Java bedeutet das, dass die Javadoc-Kommentare in den Schnittstellen um QSL ergänzt werden können und kein extra Dokument für die Schnittstellenspezifikation benötigt wird. Die QSL-Notation ist leicht zu erlernen und leichter zu verstehen als z. B. OCL, da sie den Sprachumfang der Zielsprache nutzt. Weiterhin ist es für den Entwickler von Vorteil die Annotationen direkt am Quellcode vorzunehmen, um den zusätzlichen Aufwand gering zu halten. Daher wird bei der Implementierung der Beispielanwendung die Quasar Specification Language verwendet.

Implementierung von Komponenten

Für die Vorgehensweise bei der Implementierung von Komponenten hält Quasar ebenfalls einige Hinweise bereit. [Sie04, S. 52ff.]

Für jede Komponente ist zu entscheiden, ob sie ein Singleton oder Multiton sein soll, d. h., ob nur ein Komponentenobjekt existieren darf oder mehrere. Die Entscheidung darüber kann eine Fabrik (siehe GHJV95) übernehmen, die der Komponente angehört und ihre Objekte erzeugt.

Komponenten dürfen keine eigenen Methoden definieren. Sie implementieren nur solche, die in Schnittstellen vorgegeben sind.

Vererbung über Komponentengrenzen hinweg ist nach Quasar nicht erlaubt. Dies soll vor unerwünschten Abhängigkeiten schützen, was bei Komponenten sinnvoll erscheint, die nur für eine konkrete Anwendung entwickelt werden. Bei dieser Regel bleibt aber unberücksichtigt, dass für die Verwendung von Frameworks wie SalesPoint der Einsatz von Vererbung notwendig ist. Hier muss wohl eine Ausnahme gemacht werden.

Wird eine Komponente in Java umgesetzt, besteht sie aus einer Menge an eng kooperierenden Klassen. In Java können Pakete als Einheit angesehen werden. Komponenten sollten daher als ein Paket realisiert werden [Sie04, S. 37]. Auf der obersten Ebene des Pakets befinden sich alle definierten Schnittstellen, die erstellten Nicht-Standarddatentypen, Fabriken sowie die Transportklassen[5]. Eine Ebene tiefer finden sich die implementierenden Klassen, die niemals von anderen Komponenten importiert werden.

2.2.5 Standardarchitektur der Benutzerschnittstelle

Quasar bietet für die Entwicklung grafischer Benutzerschnittstellen einen grundlegenden Rahmen. So erfolgt in [Sie03b, S. 28ff.] und [Sie04, S. 235ff.] die Definition wichtiger Be-

[5] Transportklassen werden für Entitätsklassen erstellt und in dienstorientierten Schnittstellen verwendet, um lose Kopplung zu erreichen (siehe [Siedersleben04, S. 179 u. 184]).

griffe. Dazu zählen beispielsweise Dialog, Dialogobjekt, Sitzung, Dialogwechsel, Unterdialog und Formular (Maske). Diese Begriffsbestimmungen tragen erheblich zum Verständnis der dort ebenfalls beschriebenen Konzepte bei. Weiterhin wird eine Standardarchitektur für eine GUI-Engine vorgeschlagen. Diese Standardarchitektur beschreibt den Aufbau der Benutzerschnittstelle aus verschiedenen Komponenten wie dem Dialograhmen und den Dialogen. Wichtige Themen wie Ereignisverarbeitung, Synchronisation, Sitzungsverwaltung oder Validierung von Benutzereingaben werden erläutert.

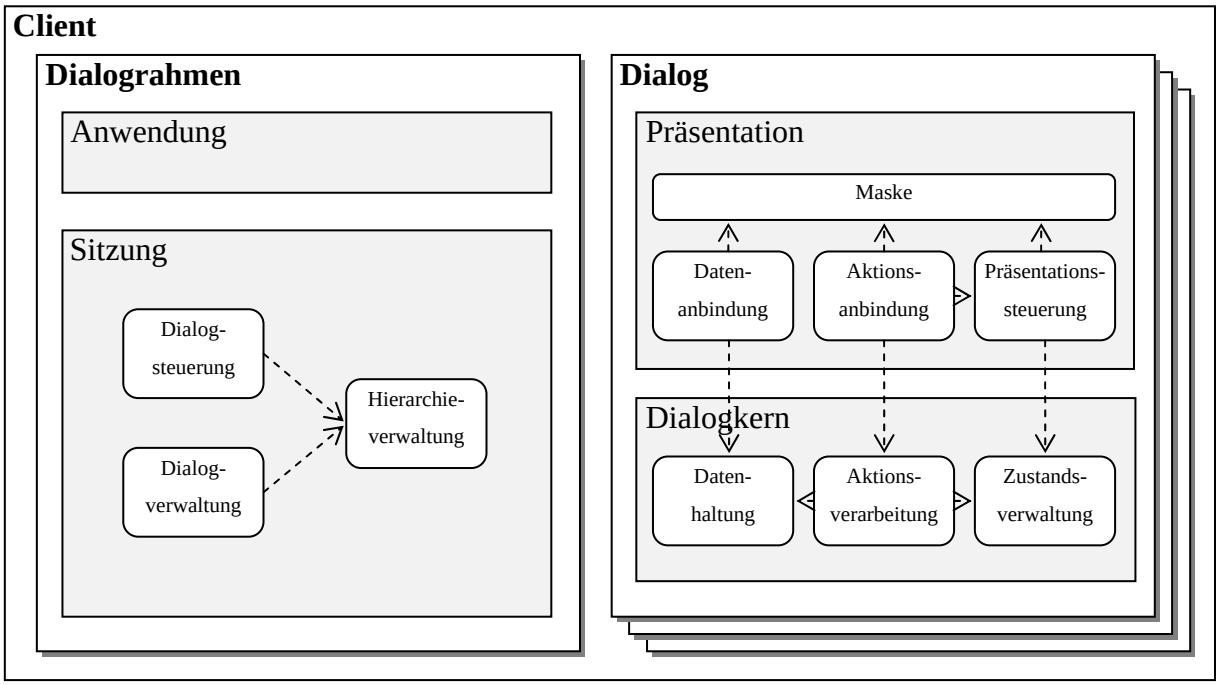

Abbildung 2-2: Komponentenbasierte Client-Architektur nach Quasar

Diese Standardarchitektur für User Interfaces wird in [HHS04, S. 15ff.] und [HHS05, S. 114ff.], wo sie als Referenzarchitektur für Clients bezeichnet wird, um die Definition von Standardschnittstellen von Dialogen und eines Dialograhmens in QSL ergänzt. Obendrein wird auf die Abläufe wie das Öffnen und Schließen von Dialogen mittels Sequenzdiagrammen genauer Bezug genommen.

Schließlich bietet [HO07] mit der „Komponentenbasierte[n] Client-Architektur“ eine nochmals detailliertere und ausgereiftere Beschreibung der Dialogarchitektur, wie sie Abbildung 2-2 zeigt. Dort wird eine weiter verfeinerte Darstellung der Aufteilung von Dialograhmen und Dialogen in Komponenten wie die Sitzung mit Dialogverwaltung, Dialogsteuerung und Hierarchieverwaltung sowie den Dialogkern und die Präsentation geliefert. Ferner wird der Aspekt der Einbettung bzw. Komposition von Dialogen vertieft betrachtet.

Die Quasar Client-Architektur macht die Entwicklung von Benutzerschnittstellen leichter beherrschbar. Der Ansatz ermöglicht bei konsequenter Anwendung eine strukturierte Vorgehensweise und sorgt für eine klare Trennung der Zuständigkeiten. Das Vorgehen zur Spezifikation von Schnittstellen kann auch für die Programmschnittstellen der Komponenten verwendet werden, die für die GUI zuständig sind. Jedoch kann das Vorgehen zur Spezifikation von Komponenten, welches eher für den Anwendungskern geeignet ist, nicht so ohne weiteres auf die Identifikation von Komponenten für die Benutzerschnittstelle übertragen werden. Dies liegt zum Teil daran, dass Benutzerschnittstellen aus der Spezifikation ausgeklammert werden, was sich natürlich derart auswirkt, dass die Methode keine konkreten Kriterien für GUI-Komponenten vorsieht. Trotz der durchdachten Konzepte und Prinzipien für den Anwendungskern und der vorgeschlagenen Client-Architektur bleibt ein Problem unbehandelt. Nach welchen Kriterien wird der Client in Dialoge unterteilt? Zum Vorgehen bei der Identifikation der benötigten Dialoge und der Komposition ihrer visuellen Darstellung hält Quasar keine Informationen bereit, was allerdings allgemein ein Problem von Softwareentwicklungsmethoden darstellt (siehe [vHa01]). Daher wurden hierfür einige andere Methoden untersucht, die Anleihen aus dem Fachgebiet Human-Computer Interaction (HCI) nehmen und im nächsten Abschnitt erläutert werden.

2.3 Ansätze zum Dialogdesign

Die Architekturentwicklungsmethode Quasar trifft zwar einige Aussagen über die Architektur der Benutzerschnittstelle, speziell über den Aufbau und das Zusammenwirken der einzelnen Dialog- und Dialograhmenkomponenten. Aber für den Prozess von Identifikation, Entwurf und Gestaltung der Dialogkomponenten mangelt es an Informationen (siehe letzter Abschnitt). Daher wurde nach alternativen Ansätzen zur Dialoggestaltung gesucht und diese auf Eignung zur Füllung dieser Lücke hin geprüft.

2.3.1 User Interface Design im Rational Unified Process

Der Rational Unified Process (RUP) ist eine Software-Entwicklungsmethode, die ursprünglich von der Firma Rational entwickelt und vertrieben wurde und nun zu IBM gehört. Die

Methode ist anwendungsfallgetrieben und basiert auf einem iterativen Softwareentwicklungslebenszyklus [RUP06].

Nachfolgend exzerpiertes Vorgehen entstammt [KAB01] und ist dort im Detail beschrieben. (Neuere Versionen des RUP können sich davon leicht unterscheiden. So ist in [RUP06] beispielsweise die Rolle *System Analyst* für das Artefakt *Storyboard* verantwortlich. Dies hat aber auf das beschriebene Konzept keinen entscheidenden Einfluss.)

Im Rational Unified Process ist die Rolle des *User Interface Designers* mit dem Modellieren und dem Prototyping für die Benutzerschnittstelle beschäftigt. Dazu werden die Artefakte *Use Case Storyboard* und *User Interface Prototype* in den Aktivitäten *User Interface Modeling* und *User Interface Prototyping* erstellt.

Ein Use Case Storyboard beschreibt logisch und konzeptionell, wie ein Anwendungsfall durch die Benutzerschnittstelle angeboten wird, einschließlich der Interaktionen zwischen dem Akteur und dem Anwendungssystem. Die Repräsentation in UML entspricht einer Kooperation (englisch: *collaboration* siehe [Fow05]) mit dem Stereotyp <<*use case storyboard*>>. Zu einem Use Case Storyboard gehören folgende Bestandteile:

- Flow-of-events-Storyboard (eine textuelle Beschreibung der Interaktion zwischen Nutzer und System angereichert um Usability-Aspekte),
- Interaktionsdiagramme,
- Klassendiagramme,
- Usability-Anforderungen,
- Referenzen zum User Interface Prototype sowie
- Traceability-Beziehungen zu den entsprechenden Anwendungsfällen.

Ein User Interface Prototype ist ein anfängliches rohes Modell der Benutzerschnittstelle und kann sich z. B. durch folgende Arten manifestieren:

- Papierskizzen oder Bilder,
- Bitmaps eines Zeichenwerkzeugs oder Design-Kompositionen,
- interaktiv ausführbare Prototypen.

Aktivität User Interface Modeling

Bei dieser Aktivität wird ein Modell der Benutzerschnittstelle erzeugt. Als Eingangsdaten dafür wird das *Use Case Model* mit den Anwendungsfällen des Systems verwendet. Jeder Anwendungsfall wird beim User Interface Modeling wie folgt behandelt:

1. Beschreibe die Charakteristik der in Beziehung stehenden Akteure.
2. Erstelle ein Use Case Storyboard.
3. Beschreibe das Flow-of-events-Storyboard.
4. Erfasse Usability-Anforderungen am Use Case Storyboard.
5. Finde boundary-Klassen,[6] die im Use Case Storyboard gebraucht werden.
6. Beschreibe Interaktionen zwischen boundary-Objekten und Akteuren.
7. Ergänze die Diagramme des Use Case Storyboards.
8. Verweise auf den User-Interface-Prototypen des Use Case Storyboards. Für alle boundary-Klassen führe folgende Schritte aus:
    a. Beschreibe die Verantwortlichkeiten der boundary-Klassen.
    b. Beschreibe die Attribute der boundary-Klassen.
    c. Beschreibe die Beziehungen zwischen den boundary-Klassen.
    d. Beschreibe die Usability-Anforderungen an die boundary-Klassen.
    e. Präsentiere die boundary-Klassen in globalen Klassendiagrammen.
9. Evaluiere die Ergebnisse.

Die einzelnen Schritte müssen dabei nicht strikt in dieser Reihenfolge ausgeführt werden und können in Abhängigkeit der Komplexität des Projektes auch ausgelassen werden.

Aktivität User Interface Prototyping

Für jedes Use Case Storyboard werden zur Erstellung eines Prototyps folgende Schritte durchlaufen:

1. Beschreibe den Prototyp.
    a. Identifiziere die primären Fenster.
    b. Gestalte die Visualisierung der primären Fenster.

[6] Boundary-Klassen sind Klassen, welche die Kommunikation zwischen der Systemumgebung und den inneren Bestandteilen modellieren, also auch Interaktionen zwischen ein oder mehreren Akteuren und dem System.

   c. Entwerfe die Operationen der primären Fenster.
   d. Gestalte die Eigenschaftsfenster (beinhalten die Attribute von Entitätstypen).
   e. Entwerfe die Operationen, die mehrere Objekte einbeziehen.
   f. Entwerfe verschiedene Merkmale (z. B. zum dynamischen Verhalten).
2. Implementiere den Prototyp.
3. Hole Feedback zum Prototyp ein.

Die Schritte beim Prototyping müssen nicht alle zwingend durchgeführt werden. Im Rational Unified Process werden die einzelnen Aktionen zum besseren Verständnis noch um einige Richtlinien (*Guidelines*) ergänzt. Darauf soll an dieser Stelle nicht detaillierter eingegangen werden.

Für die Aufgabenstellung im Rahmen dieser Arbeit war festzustellen, dass der Design-Ansatz aus dem Rational Unified Process geeignet zu sein scheint, die Lücke im Quasar-Vorgehen bei der Identifikation von Komponenten für die Benutzerschnittstelle zu schließen. User Interface Design im Rational Unified Process speziell mit den Use Case Storyboards bietet eine strukturierte, an definierten Schritten orientierte Vorgehensweise. Sie betrachtet die Funktionalität aus Nutzersicht und reichert Anforderungen, modelliert in Anwendungsfällen, um Informationen zur Usability an. Dies ermöglicht eine erleichterte Identifikation von Klassen oder Komponenten für das Graphical User Interface. Das Vorgehen ist anwendungsfallgetrieben und fügt sich so in den Prozess der GUI-Entwicklung für ein Beispielprojekt zum SalesPoint-Framework ein, da hier die Anforderungen und Anwendungsfälle bereits ermittelt wurden (siehe Kapitel 3.4). Bezüglich Quasar ist eine Anpassung im Vorgehen die boundary-Klassen betreffend nötig. Diese Änderung von boundary-Klassen zu Dialogkomponenten, die die gleiche Aufgabe übernehmen können, ist aber ohne weiteres möglich. Die Aktivität Prototyping mit den dazu gehörigen Richtlinien gibt entscheidende Hinweise zur Umsetzung der Benutzeroberfläche.

2.3.2 Ein benutzerzentrierter Ansatz für objektorientiertes User Interface Design

Gulliksen et al. stellen in [GGL01] eine Methode vor, die dem benutzerzentrierten Design, als essenziellem Prozess beim Entwickeln von Softwaresystemen mit guter Usability, besondere Bedeutung beimisst. Dabei soll der Nutzer stärker in den Entwicklungsprozess für die Benutzerschnittstelle eingebunden werden als bei anwendungsfallorientierten Vorgehensweisen. Dies soll die kreativen Aspekte des User Interface Design begünstigen. Die Arbeit mit den späteren Nutzern des Systems wird intensiviert und der Aspekt der Usability beim Schnittstellendesign betont. Das benutzerzentrierte Design wird in eine Entwicklungsmethode wie dem Rational Unified Process integriert.

Objektorientierte Ansätze zur Entwicklung von Anwendungen, welche allein auf Anwendungsfällen basieren, sind nützlich für die Analyse der Anforderungen des Systems und für die Entwicklung von Softwarekomponenten für Informationssysteme (den Anwendungskern, die A-Software nach Quasar). Sie unterstützen den Entwurf der Benutzerschnittstelle aber nur unzureichend, da sie beispielsweise keine Aspekte der Usability berücksichtigen und nur auf die Funktionalität des Systems fokussiert sind. Stattdessen laden solche prozessorientierten Vorgehensweisen, bei denen die Arbeit der Nutzer durch ein Datenmodell und definierte Prozesse spezifiziert wird, die Entwickler dazu ein, eine Benutzerschnittstelle zu entwerfen, die für jede Funktion oder jeden Anwendungsfall ein eigenes Fenster auf dem Bildschirm anzeigt. Das Ergebnis sind fragmentierte Interfaces, bei denen der Nutzer typischerweise mit einer Menge verschiedener Fenster arbeiten muss, um nur einen Arbeitsgang zu erledigen. [LOGS01]

Die von Gulliksen et al. entwickelte Methode des *User Interface Modeling* (UIM) soll Nutzeranforderungen sammeln, die direkt ins Benutzerschnittstellendesign einfließen und dieses somit verbessern. UIM ist als Ergänzung zur Anwendungsfallmodellierung und einem objektorientierten Softwareentwicklungsprozess zu sehen. Es spezifiziert ein *actor model*, ein *goal model* sowie ein *work model*. Als wichtigster Aspekt des UIM soll hier die Workspace-Metapher [LOGS01] erläutert werden. Einem Akteur werden im *work model* mehrere Arbeitssituationen zugeordnet. Die Hauptidee besteht darin, jede Arbeitssituation durch eine geeignete Benutzerschnittstelle mit allen benötigten Informationen und Aktionen zu unterstützen. Dies resultiert in einem Design, welches speziell darauf zugeschnitten ist, wie ein Nutzer

seine Arbeit verrichtet. Jede Arbeitssituation wird durch einen Workspace auf dem Bildschirm abgebildet. Dadurch muss der Nutzer im Gegensatz zum prozessorientierten Ansatz mit deutlich weniger Fenstern interagieren und gewinnt einen besseren Überblick. Darüber hinaus unterstützt die Workspace-Metapher nicht nur die Aufgaben innerhalb von Arbeitssituationen, sondern erleichtert auch den Wechsel zwischen unterschiedlichen Situationen. Somit kann manchmal ein und derselbe Anwendungsfall aus mehreren Workspaces aus aufgerufen werden.

Das Vorgehen von Gulliksen et al. konzentriert sich auf die Möglichkeiten für eine effektive Nutzerpartizipation im Softwareentwicklungsprozess. Dabei wird keine Reihenfolge von Schritten zur Erstellung von Benutzerschnittstellen mit guter Usability beschrieben. Der Grund dafür ist, dass das User Interface Design als kreativer Prozess angesehen wird, der sich nicht in eine solch strikte Abfolge pressen lässt. Der von den Autoren dargestellte Weg des benutzerzentrierten und objektorientierten GUI-Designs soll die Lücke zwischen den OO- und HCI-Methoden schließen.

Eine vollständige Anwendung dieser Methode wäre aufwendig und würde sich nicht ohne weiteres in das Quasar-Vorgehen und damit in diese Arbeit integrieren lassen, weshalb darauf verzichtet wird. Dagegen erscheint die Workspace-Metapher eine ausgezeichnete Hilfestellung bei der Gestaltung der Benutzeroberfläche zu sein und wird bei der Festlegung der Hauptfenster der zu erstellenden Beispielanwendung mit einfließen.

2.3.3 Virtual Windows

Lauesen adressiert mit seiner entwickelten Virtual-Windows-Methode die semantische Lücke, die sich zwischen User Interface Design nach klassischen Softwareengineering-Vorgehensweisen und den HCI-Methoden zeigt [Lau05]. Mit dem entwickelten Vorgehen wird ein systematischer Weg aufgezeigt, eine Benutzerschnittstelle zu erstellen, bei dem Praktiken aus beiden Welten adaptiert und kombiniert werden. Dieser Weg gliedert sich grob in sechs Teile. Zunächst wird eine Domain-Analyse durchgeführt. Anschließend folgt das Design der Virtual Windows, einer frühen grafischen Umsetzung der Datenpräsentation in der Anwendung. Darauf folgen das Funktionsdesign, schließlich Prototyping mit Usability-Tests, die Programmierung und abschließende Tests.

Zur Domain-Analyse gehören die Erstellung eines Datenmodells sowie die Task-Analyse. Task-Analyse ist das HCI-Vorgehen zum Erlangen des Verständnisses über die Problemdomäne vergleichbar mit der Anforderungsanalyse (*requirements elicitation*) [Lau05, S. 133ff.]. Task-Beschreibungen erklären im Detail, welche Aufgaben ein Anwendungsnutzer erledigen will. Sie werden dazu verwendet, um abzusichern, dass die künftige Benutzerschnittstelle alle Aufgaben erfüllen kann. Im Vergleich zu Task-Beschreibungen sind Anwendungsfälle eine Unterart, die im Gegensatz zu Tasks explizit angeben, was das System und was der Mensch macht. Use Cases, die in ihrer Beschreibung zu speziell formuliert sind, können bei exakt an der Anwendungsfallbeschreibung orientierter Implementierung zu einer Benutzerschnittstelle führen, die dem Nutzer jeden Schritt genau vorgibt und damit schlechte Usability aufweist. Tasks hingegen machen noch keine Annahme über die Aufteilung der Schritte zwischen Nutzer und System. Sie lassen somit mehr Spielraum für die spätere Gestaltung der Dialoge.

Die Virtual Windows sind eine benutzerorientierte Repräsentation der persistenten Daten, die sich der Nutzer hinter dem physischen Bildschirm vorstellen soll [Lau05, S. 167ff.]. Sie sind anfänglich Papierskizzen und zeigen Daten ohne Buttons, Menüs oder andere Funktionen. Später werden diese Elemente hinzugefügt und die reale Dialogmaske entsteht. In einem iterativen Prozess wird zunächst festgelegt, welche Daten in welchem Fenster sichtbar sein sollen. Danach wird ein detailliertes grafisches Design der Fenster erstellt. Daraufhin wird überprüft, ob die Nutzer das Design verstehen. Abschließend wird das Design gegen die Task-Beschreibungen und das Datenmodell abgeglichen. Für das Design der Virtual Windows werden die nachfolgenden Regeln beachtet [Lau05, S. 169]:

1. „Few window templates."
2. „Few window instances per task."
3. „Data in one window instance only."
4. „Rooted in one thing."
5. „Virtual windows close to final screen size."
6. „Necessary overview of data."
7. „Things – not actions."
8. „All data accessible."

Diese Regeln sind Richtlinien im Design-Prozess, die sich häufig widersprechen und gegeneinander abgewogen werden müssen, stellen aber hilfreiche Kriterien zum Finden adäquater

Dialoge dar. Für eine gute visuelle Gestaltung der einzelnen Dialoge mit ihren Bedienelementen werden weitere Prinzipien vorgeschlagen [Lau05, S. 178]:

1. „Follow platform."
2. „Alternative presentations."
3. „Reuse old data presentations."
4. „Use gestalt laws."
5. „Study other people's design."

Beim Schritt Funktionsdesign wird geplant, welche Buttons der Nutzer benötigt, um seine Aufgaben auszuführen, wie sich die Virtual Windows zu Bildschirmmasken zusammensetzen und wie zwischen diesen navigiert werden kann. Dabei gilt es, sich z. B. zwischen der Umsetzung als *full-screen workspace* oder *part-screen workspace* zu entscheiden. [Lau05, S. 244f.]

Im Anschluss können ein Prototyping und ein Usability-Test vorgenommen werden. Dabei werden die Probleme mit der Benutzbarkeit soweit wie möglich entfernt. Darauf folgt der Übergang zur Implementierung und den finalen Tests.

Zwischen den einzelnen Stufen kann iteriert werden und es besteht die Möglichkeit, Teile auszulassen.

Die Virtual Windows Methode von Lauesen kann rein Task-getriebenen und Datenbank-getriebenen Herangehensweisen zum Entwurf von Benutzerschnittstellen gegenübergestellt werden [Lau05, S. 208ff.]. Virtual Windows helfen dabei, die Balance zwischen Extremen zu halten. Dies sind zum einen Datenbank-orientierte Interfaces, wo es für jede Datenbanktabelle einen Dialog gibt, und zum anderen Schritt-für-Schritt-orientierte Interfaces, die sich aus rein Task-getriebenem Vorgehen ergeben können. In dieser Beziehung ähnelt die Methode der von Gulliksen et al. mit der Workspace-Metapher, die auch ein fragmentiertes Interface mit vielen Fenstern vermeiden will. Der Virtual-Windows-Ansatz basiert auf einer Task-Analyse und ist in vollem Umfang angewandt sehr umfangreich. Im Hinblick auf das schon vorhandene und zu verwendende Anwendungsfallmodell der Beispielanwendung (siehe Kapitel 3.4) wäre erheblicher zusätzlicher Aufwand nötig. Somit sieht der Autor von einer Kombination mit der Quasar-Methode zunächst ab, obwohl Virtual Windows nach Lauesens Vorgehen potenzielle Kandidaten für Dialogkomponenten darstellen und zu deren Identifikation beitragen könnten.

Lauesens Ansatz bietet allerdings die oben genannten überaus nützlichen Regeln und Prinzipien, die bei der visuellen Gestaltung der Dialoge der Beispielanwendung eine Hilfestellung sein werden.

3 Das SalesPoint-Framework

Das Framework SalesPoint wurde von der Technischen Universität Dresden und der Universität der Bundeswehr München entwickelt. Es wird an diversen Universitäten in praktischen Übungen für die Erstellung von Shop-Anwendungen eingesetzt.

Für die vorliegende Arbeit kamen die Ergebnisse der Diplomarbeit von Kristian Herpel [Herp07] zum Einsatz. Dies sind Java-Komponenten, die einem Refactoring des Frameworks SalesPoint entstammen. Als weitere Grundlage wird die Version 3.2 des SalesPoint-Frameworks verwendet, welche von den Entwicklern auf Java 1.5 aktualisiert worden ist.

3.1 SalesPoint im Original – ein kurzer Überblick

Das SalesPoint-Framework dient dazu, Shop-Anwendungen mit wenig Aufwand zu implementieren, sodass Projekte für Studenten in praktischen Übungen nicht zu umfangreich werden. Das Framework kann durch Simulation mehrerer Verkaufsstellen, den *SalesPoints*, in denen Verkaufsprozesse (*SalesProcess*) ablaufen, eine verteilte Anwendung mit Multi-User-Support simulieren.

SalesPoint besteht im Original aus zwei unabhängigen Teilen, der Datenverwaltung im Paket *data* und der dynamischen Applikationsverwaltung im Paket *sale*. Weitere kleinere Bestandteile sind Benutzermanagement (Paket *user*), Protokollverwaltung (Paket *log*) und die grafische Benutzerschnittstelle. Die Klassen, welche die Realisierung von Benutzeroberflächen für Shop-Anwendungen unterstützen, bauen direkt auf dem Swing-Framework[7] auf. Aufgrund der unmittelbaren Verknüpfung ergibt sich eine große Abhängigkeit von SalesPoint zu Swing. Weiterhin sind die einzelnen Klassen, die für die Erzeugung der Benutzeroberfläche verantwortlich sind, nicht von den anderen Bestandteilen getrennt und über die verschiedenen Pakete des Frameworks verteilt.

[7] Die Swing API ist Bestandteil der Java SE Plattform http://java.sun.com/javase/technologies/desktop/

Für die Arbeit mit dem SalesPoint-Framework stehen der von der TU Dresden bereitgestellte Quellcode sowie weitere Dokumentationen und Kommentare zur Verfügung. Dazu gehören:

- die im Internet abrufbare Online-Dokumentation[8] mit HowTos, Javadoc und technischen Dokumentationen,
- Antworten zu häufig gestellten Fragen (FAQ),
- sowie Beispielanwendungen und Tutorien.

3.2 Motivation zur Überarbeitung von SalesPoint

Das SalesPoint-Framework kommt an der Technischen Universität Ilmenau für Software-Praktika zum Einsatz. Dabei wurden verschiedene Kritikpunkte identifiziert, welche die Arbeit mit dem Framework für die studentischen Entwickler unnötig erschweren.

Ein hauptsächlicher Kritikpunkt am Framework ist die fehlende klare Aufteilung in Architekturschichten. Dies beruht auf der engen Kopplung an das Swing-Framework. So sind die Klassen für die Unterstützung bei der Realisierung einer Benutzeroberfläche nicht von den übrigen Bestandteilen des Frameworks getrennt, sondern über mehrere Pakete der anderen Teile verstreut.

Bei der Erstellung eigener Anwendungen mit dem SalesPoint-Framework wird prinzipiell Vererbung verwendet, denn eine Adaption der vorgegebenen Teile an die eigenen Bedürfnisse kann fast nur durch Unterklassenbildung erfolgen. Zudem ist die Funktionalität der Klassen des Frameworks teilweise schwer verständlich. Änderungen daran sind aufwendig und müssen intensiv getestet werden, um deren fehlerfreie Funktionalität weiterhin gewährleisten zu können. Dies muss als weiterer Schwachpunkt angesehen werden.

Eine Analyse der Klassen und Operationen in [Herp07] zeigte zusätzlich das Problem auf, dass die zentralen Klassen *Shop*, *SalesPoint* und *SalesProcess* für mehrere unterschiedlichste Aufgaben zuständig sind, die eigentlich voneinander getrennt bearbeitet werden sollten. Die Analyse dieser Zuständigkeiten motiviert also eine Überarbeitung des Frameworks nach dem

[8] Die gesamte Online-Präsenz zum SalesPoint-Framework mit allen Dokumentationen und Hinweisen ist zu finden unter der Webseite http://www-st.inf.tu-dresden.de/SalesPoint/v3.2/index.html

Prinzip Trennung der Verantwortlichkeiten (*separation of concerns*). Dies kann teilweise durch ein komponentenorientiertes Refactoring, wie es von Herpel beschrieben wird, geschehen.

3.3 Stand der Überarbeitung

Aufgrund der erkannten Schwächen wurde bereits in [Herp07] damit begonnen, eine Methode zur Identifikation und Extraktion von Komponenten auf das Framework SalesPoint anzuwenden. Dabei wurde SalesPoint mittels *Class-Responsibility-Collaboration-Cards* [BC89] analysiert, um die Abhängigkeiten offen zu legen. Anschließend erfolgte für das Framework die Ermittlung der Softwarekategorien nach der Quasar-Methode. Das Ergebnis ist ein Kategoriengraph für die Funktionsgruppen des Frameworks. Weiterhin wurde das Framework anhand der Kategorien in Komponenten unterteilt (siehe Abbildung 3-1).

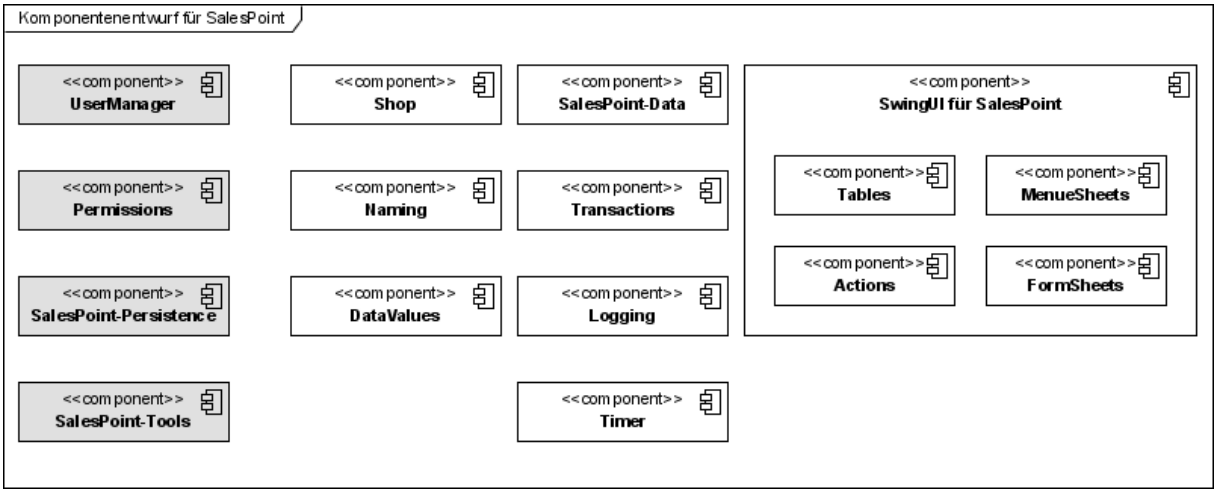

Abbildung 3-1: Komponentenentwurf für SalesPoint (vgl. [Herp07])

Für die Komponenten *UserManager*, *Permissions*, *SalesPoint-Persistence* sowie *SalesPoint-Tools* wurde bereits die Extraktion abgeschlossen, sodass für diese Komponenten eine Java-Implementierung vorliegt. Für die restlichen Komponenten steht eine Umsetzung noch aus. Jedoch wurde für die Bestandteile des SalesPoint-Frameworks, welche für die grafische Benutzeroberfläche zuständig sind, die Entscheidung getroffen, dass diese von Grund auf neu entwickelt werden sollen. Somit entfällt eine Extraktion der Komponente *SwingUI*. Diese Entscheidung beruht auf der Tatsache, dass die originalen Klassen des Frameworks zu eng

mit den anderen Teilen verbunden sind und der Aufwand für eine Trennung höher wäre, als eine strukturierte Neuentwicklung.

Der Entwurf einer komplett neuen Architektur für die Benutzeroberfläche ist eine umfangreiche und nicht ganz unkritische Aufgabe, da unter anderem viele mögliche Szenarien von Shop-Anwendungen berücksichtigt werden müssen. Daher soll in einem ersten Schritt anstelle einer API für das gesamte Framework zunächst mittels einer Beispielanwendung eine neue Benutzerschnittstellenarchitektur entwickelt werden, die später auf ein Framework übertragen werden kann. Dies geschieht im Rahmen der vorliegenden Arbeit anhand der Quasar-Methode.

3.4 Die Beispielanwendung: Vendorbase

Bei Vendorbase handelt es sich um eine Shop-Anwendung, die mithilfe des SalesPoint-Frameworks während einer praktischen Übung von Studenten an der TU Ilmenau entstanden ist. Sie hat die Aufgabe zu erfüllen, alle Lieferverträge einer Firma mitsamt den Produktteilen und entsprechenden Lieferanten zu verwalten.

Vendorbase gliedert sich in vier Hauptbestandteile. Diese sind die Lieferantenverwaltung, die Produktteilverwaltung, die Liefervertragsverwaltung und die Nutzerverwaltung. Für die einzelnen Domänenobjekte existieren Funktionen zum Anlegen, Bearbeiten sowie Löschen und die Objekte müssen gefiltert, sortiert und mit ihren Detaildaten angezeigt werden können. Weiterhin ist in Vendorbase ein Rollenmodell vorhanden, über das die Nutzer der Anwendung unterschiedliche Berechtigungen zugeteilt bekommen. Detaillierte Informationen zur Beispielanwendung sind der Projektdokumentation von Vendorbase zu entnehmen, die sich auf der CD im Anhang befindet.

Warum wurde Vendorbase als Beispielanwendung ausgewählt? Dies ist zum einen damit zu begründen, dass die Dokumentation zur Anwendung sehr ausführlich ist und z. B. das Pflichtenheft mit allen definierten Anforderungen als gute Grundlage für eine Neuentwicklung dient. Des Weiteren ist Vendorbase überschaubar, leicht verständlich und nicht so umfangreich wie vergleichsweise das „*Sohn&Sohn Großmarkt*“-Projekt aus der Online-Dokumenta-

tion zu SalesPoint der TU Dresden. Darüber hinaus werden in Vendorbase hauptsächlich die oben genannten Funktionen für Domänenobjekte benötigt, sodass die Anwendung nicht so prozesslastig ist wie beispielsweise *Großmarkt*. Dies ist hilfreich, da das Refactoring in Komponenten für SalesPoint, wie in Abschnitt 3.3 *Stand der Überarbeitung* erwähnt wurde, noch nicht vollständig umgesetzt ist. Von der fehlenden Umsetzung ist unter anderem die zentrale *Shop*-Klasse betroffen. Mit Vendorbase werden jedoch die Funktionalitäten dieser Klasse, wie z. B. die Prozesssteuerung, nicht zwingend benötigt.

4 Methodisches Vorgehen

Dieses Kapitel beschreibt das methodische Vorgehen nach Quasar. Die einzelnen Aktivitäten werden durch zusätzliche Schritte verfeinert, um ein lückenloses Vorgehen zu erreichen. Dies ist die Grundlage dafür, dass alle Entwurfsentscheidungen über Traceability-Links erfasst werden können. Zur Illustration der Aktivitäten wird das Vorgehen an der Neuentwicklung der Beispielanwendung Vendorbase mit komplett überarbeiteter grafischer Benutzeroberfläche verdeutlicht.

4.1 Anforderungsanalyse

Die Entwicklung eines Softwaresystems beginnt mit der Analyse der Anforderungen an die zukünftige Anwendung. Diese sind zu identifizieren, zu dokumentieren und zu strukturieren [PBG04]. Wie bereits in Kapitel 2.2.3 herausgestellt wurde, bietet Quasar leider keine speziellen Aktivitäten zur Anforderungsanalyse an, sondern es wird lediglich beispielhaft auf das Vorgehen nach Volere verwiesen.

Für den Autor lagen die Anforderungen der verwendeten Beispielanwendung Vendorbase bereits vor. Als Dokumentation wurde von den Entwicklern ein Pflichtenheft [FKR07] bereitgestellt. In ihm sind die benötigten Anwendungsdaten sowie die Funktionen beschrieben. Weiterhin sind detaillierte Anwendungsfall- und Aktivitätsdiagramme für die Funktionen der Anwendung vorhanden. Aus diesem Grund war eine Anforderungsanalyse für die Arbeit mit Vendorbase nicht weiter von Belang. Das Anwendungsfallmodell wurde als Input für das weitere Vorgehen nach Quasar als gegeben angesehen, und es wurden keine näheren Betrachtungen zur Vorgehensweise bei der Erstellung eines Anwendungsfallmodells angestellt.

Als durchgängiges Beispiel für die weitere Anwendung der Quasar-Aktivitäten sollen die Produktfunktionen zum Liefervertragsmanagement dienen, wenn eine vollständige Darstellung für Vendorbase zu umfangreich ist. Deshalb werden die Anwendungsfälle zum Liefervertragsmanagement hier überblickshalber dargestellt.

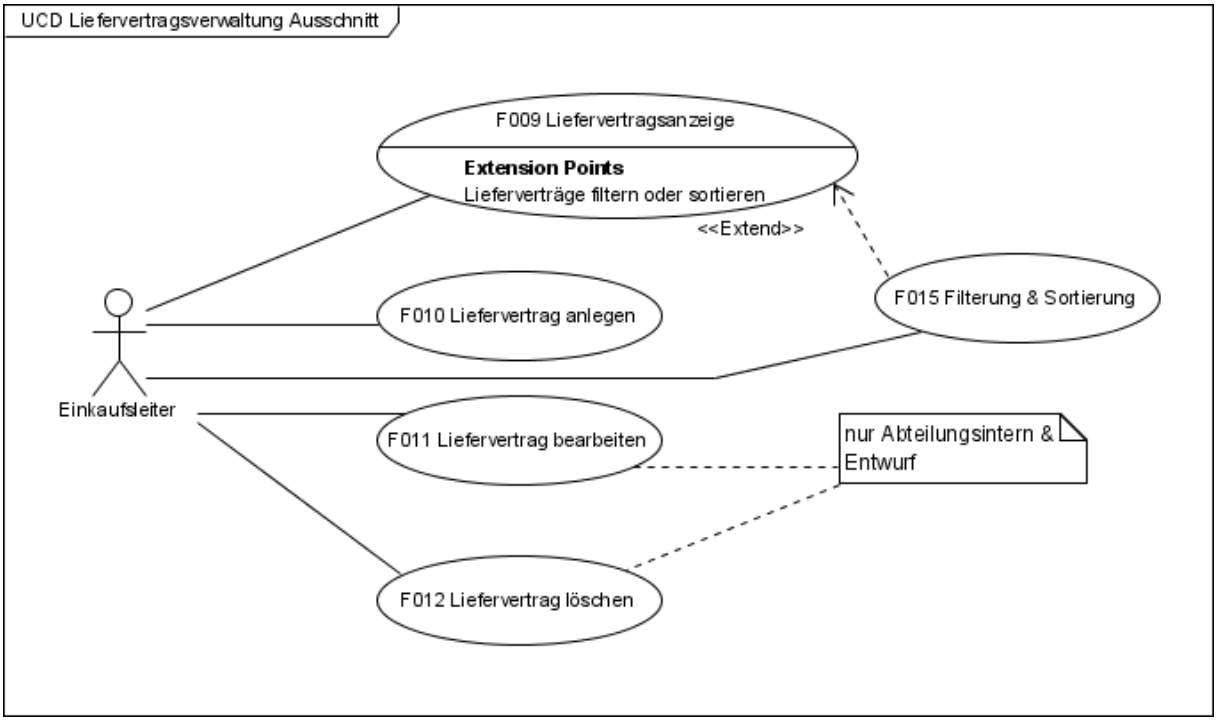

Abbildung 4-1: Anwendungsfalldiagramm zum Liefervertragsmanagement (vgl. [FKR07])

In Abbildung 4-1 sind alle für das Liefervertragsmanagement relevanten Anwendungsfälle zusammengetragen. Dazu gehören die Anzeige der Lieferverträge sowie die Funktionen zum Anlegen, Bearbeiten und Löschen von Verträgen. Der Anwendungsfall der Liefervertragsanzeige sieht optional eine Filterung oder Sortierung nach verschiedenen Kriterien vor. Für die Ausführung der einzelnen Anwendungsfälle sind bestimmte Rechte erforderlich, die von der Rolle des Benutzers abhängen. Hier ist als Akteur beispielhaft der Einkaufsleiter dargestellt, der alle gezeigten Anwendungsfälle durchführen darf. Ein Manager hätte im Vergleich dazu z. B. nicht die Möglichkeit, einen Liefervertrag zu bearbeiten oder zu löschen. Für detaillierte Beschreibungen der Anwendungsfälle sei auf das Pflichtenheft von Vendorbase verwiesen.

Ein Anforderungsmodell bietet eine Blackbox-Sicht auf das zu entwickelnde System. Bedingungen zur Konsistenz eines solchen Modells, die bei dessen Erstellung hilfreich sind, können beispielsweise in [Ade07] gefunden werden. Das Modell ist in Quasar grundlegend für die weitere Spezifikation, denn es geht direkt in die nachfolgenden Schritte der Erstellung des Kategorien- sowie Komponentenmodells ein. Es bildet damit später den Ausgangspunkt für Traceability-Links zu Elementen der anderen Modelle.

4.2 Erstellung des Datenmodells

Das Datenmodell für eine Anwendung wird nach Quasar, wie in Abschnitt 2.2.2 beschrieben, erstellt. Dabei werden zunächst die Entitäten und Beziehungen der Domäne identifiziert. Anschließend werden den Entitäten ihre Attribute zugeordnet. Die Entitäten, Attribute und Beziehungen für Vendorbase sind bereits im Pflichtenheft aufgeführt. Die zu modellierenden Entitätstypen sind *Lieferant*, *Produktteil*, *Liefervertrag* und *Nutzer*.

Das Datenmodell könnte nach Quasar in ER-Notation formuliert werden, da die Datenmodellierung als klassische Entity-Relationship-Modellierung betrachtet wird. Eine bestimmte Notation wird jedoch von Quasar nicht vorgegeben. Vom Autor wird eine UML-Notation mittels Klassendiagramm[9] bevorzugt. Diese ist vorteilhaft, weil sich die Konsistenzbedingungen, welche in [Ade07] für ein Datenmodell nach Quasar angegeben werden, ebenfalls auf eine UML-Notation beziehen. Auf Unterschiede der eigenen Notation zu dieser wird an geeigneter Stelle hingewiesen.

Bei der Erstellung des Datenmodells geben die erwähnten Konsistenzbedingungen nach [Ade07] hilfreiche Kriterien vor.

- Die modellierten Klassen sollen frei von Methoden sein.
- Die jeweiligen Attribute der Entitätstypen werden innerhalb der Klasse notiert.
- Dabei ist darauf zu achten, keine Attribute zu formulieren, die nur als technische Schlüssel dienen.
- Alle Attribute sind Standarddatentypen oder selbstdefinierte fachliche Datentypen. Referenzen auf Entitäten werden mittels Assoziationen modelliert, nicht mit Attributen.

Auf diesen Kriterien wird bei der Definition des Datenmodells in der vorliegenden Arbeit aufgebaut. Jede gefundene Entität einer Anwendung wird als Klasse modelliert. Diese Klassen enthalten keine Methoden, denn sie sollen keine Funktionalität sondern nur die benötigte Datenstruktur abbilden. Der Autor verzichtet im Gegensatz zu den Konsistenzbedingungen in [Ade07] auf die Definition von Feldlisten, welche die Typen der Attribute beschreiben sollen. Hingegen werden die Datentypen, wie in der UML vorgesehen, direkt an den Attributen notiert. Informationen zu Wertebereichen, wie die vorgesehene Länge von Strings, können wenn

[9] Sämtliche in der Arbeit verwendeten UML-Diagramme wurden mit Visual Paradigm for UML 6.0 (CE) erstellt. Das Programm ist verfügbar unter http://www.visual-paradigm.com/ und auf der CD zu dieser Arbeit.

nötig aus dem Pflichtenheft entnommen werden. Weiterhin sind die Attribute in den Klassen für Entitätstypen nicht öffentlich sondern privat. Dieses Vorgehen ist für die spätere Implementierung mit Java geeigneter, da bei Java nach Konvention der Zugriff auf Attribute über get- und set-Methoden erfolgt. Somit könnten die Klassen mit ihren typisierten Attributen aus dem UML-Modell auch generiert werden. Dies ist ein weiterer Vorteil, der für die UML-Notation spricht.

Die selbstdefinierten fachlichen Datentypen werden ebenfalls als Klassen notiert. Diese Klassen werden zusätzlich mit den Stereotypen *<<structure>>*, *<<enumeration>>* oder *<<range>>* versehen, je nachdem, ob sie einen zusammengesetzten Datentyp, eine Aufzählung oder einen Bereichsdatentyp repräsentieren. Bei Aufzählungen werden in der Regel keine Attribute benötigt, da Enumerationen nur eine fixe Menge an Werten beschreiben. Diese Wertemenge wird in der Klasse ähnlich den Attributen mit nicht spezifizierter Sichtbarkeit notiert. Wenn bei der Festlegung der Datentypen für Attribute eine Auswahlmöglichkeit besteht, dann ist die getroffene Entscheidung für eine der Alternativen als Traceability-Link festzuhalten.

Die Beziehungen zwischen den Entitäten werden in einem weiteren Schritt als Assoziationen (oder Aggregationen) zwischen den erstellten Klassen modelliert. Zu jeder Assoziation ist weiterhin die Kardinalität der teilnehmenden Klassen zu annotieren. Sind für einen Beziehungstyp ebenfalls Attribute vorgesehen, werden diese mittels einer Assoziationsklasse berücksichtigt.

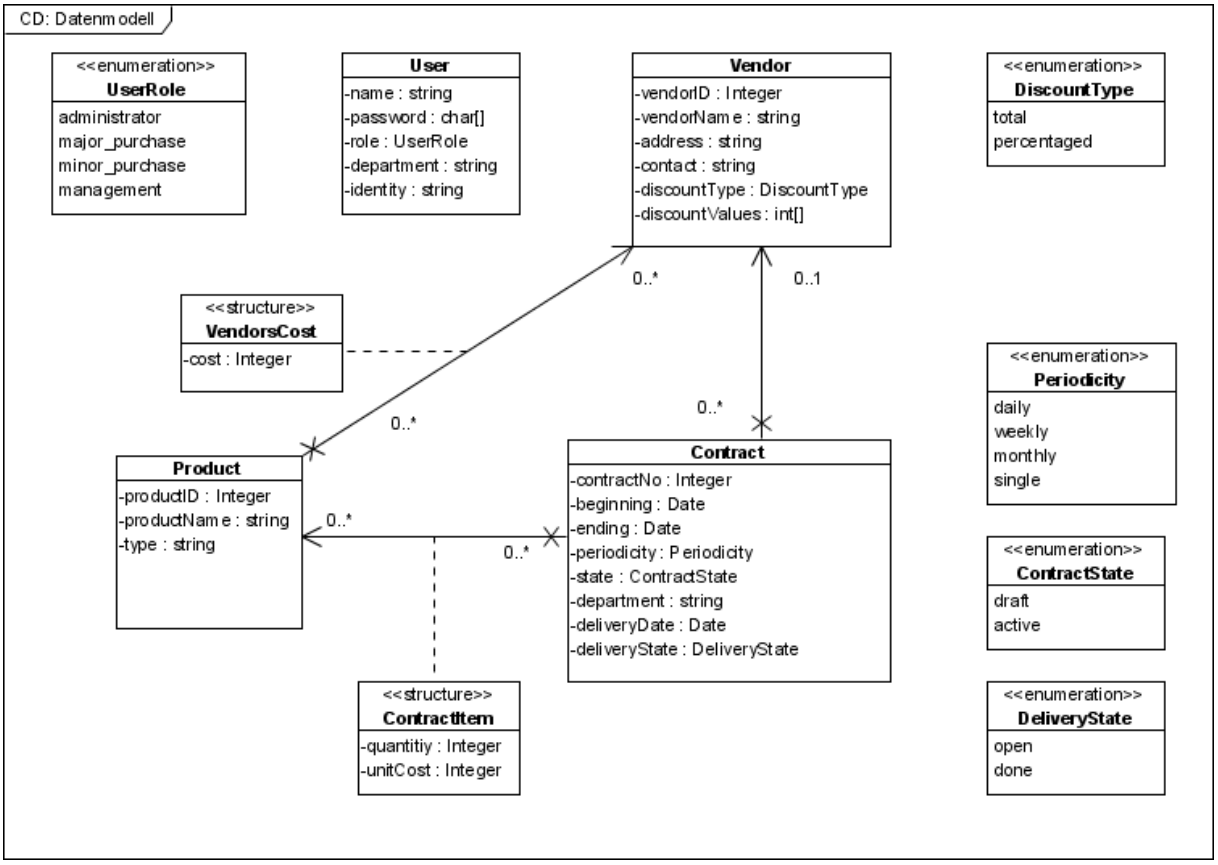

Abbildung 4-2: Datenmodell für Vendorbase

Abbildung 4-2 zeigt das Datenmodell für Vendorbase. Es sind alle Entitätstypen mit ihren Beziehungen untereinander abgebildet. Ebenso sind die einzelnen Attribute mit ihren Standarddatentypen sowie den zusätzlichen fachlichen Datentypen, gekennzeichnet durch Stereotypen, zu sehen.[10] Die Assoziationsklassen sind als strukturierte Datentypen markiert. Dies geschah bereits im Hinblick auf die spätere Implementierung. Die Datentypen können so leicht angepasst werden, dass die Beziehungen mittels loser Kopplung realisiert werden können. Dazu werden die Identifikatoren der Entitätsklassen in den strukturierten Datentyp mit aufgenommen.

Die bei der Erstellung des Datenmodells angewendeten Regeln bzw. getroffenen Entscheidungen lassen sich mittels Traceability-Links festhalten. Anhand der Links ist später erkennbar, welche Klassen mit ihren Attributen, welche Entitäten oder fachlichen Datentypen realisieren.

[10] Abweichend vom Pflichtenheft wurden an einigen Stellen Enumerationen anstatt Strings modelliert. Dies erhöht die Qualität des Modells sowie später des Quellcodes, da die möglichen Wertebelegungen offensichtlich werden.

Quelle	Ziel	Entwurfsentscheidung/ Quasar-Aktivität bzw. -Regel
Entitätstyp Liefervertrag	Klasse Contract	eine Klasse im Datenmodell für jeden Entitätstyp
Attribut Vertragsnummer	Attribut contractNo vom Typ Integer	jedes Attribut bekommt einen Standarddatentyp oder einen fachlichen Datentyp
…	…	
Attribut Periodizität	Attribut periodicity vom fachlichen Datentyp Periodizität	
fachlicher Datentyp Periodizität	Enumeration Periodicity (Klasse Periodicity mit Stereotyp <<enumeration>>)	fachliche Datentypen als Klassen mit Stereotypisierung abbilden; Enumeration anstatt String, da Aufzählung besser für begrenzte Wertemenge geeignet
…	…	…

Tabelle 4-1: Ausschnitt der Traceability-Links zum Datenmodell

Tabelle 4-1 zeigt einen Ausschnitt, wie die Traceability-Links tabellarisch dargestellt werden. Für einen Link ist die Quelle und das Ziel sowie die Entwurfsentscheidung oder Quasar-Aktivität bzw. -Regel angegeben, die für die Festlegung beim Entwurf verantwortlich ist. So ist beispielsweise im Nachhinein an einem Traceability-Link die Entscheidung nachzuvollziehen, warum für das Attribut *Periodizität* des Entitätstyps *Liefervertrag* im Datenmodell für das entsprechende Attribut der Klasse *Contract* ein Aufzählungsdatentyp anstelle eines Strings modelliert wurde. Alle erstellten Traceability-Links zum Datenmodell von Vendorbase sind im Anhang zu finden.

4.3 Entwurf des Kategorienmodells

Grundlegend für das Kategorienmodell sind die in Abschnitt 2.2.1 vorgestellten Hauptkategorien von Quasar, die Softwarekategorien 0, A, T, R und AT. Ausgehend von diesen primären Kategorien sollen beim Entwurf verfeinerte anwendungsspezifische Kategorien gefunden und nach festgelegten Kriterien in einen Kategoriengraphen eingeordnet werden. Zur Durchführung dessen sollte zunächst der Kategorienbegriff vollkommen klar sein. Daher folgt eine kurze Erläuterung, was unter dem Begriff Softwarekategorie zu verstehen ist.

Eine Kategorie vom Wortsinn her bezeichnet eine Gruppe oder Klasse, in die etwas eingeordnet wird. Softwarekategorien untergliedern eine Anwendung in Sachgebiete oder Themen

bzw. Zuständigkeiten. Sie sind wie *„Schubladen, in denen man gefundene Komponenten ablegt“* [Sie04, S. 74]. Komponenten, die gleiche oder ähnliche Zuständigkeiten haben bzw. Themen betreffen, werden somit einer Kategorie zugeordnet, ebenso die dazugehörigen Schnittstellen. Bevor die Komponenten einer Anwendung identifiziert werden, muss nach Quasar zunächst das Kategorienmodell aufgestellt werden, um unnötige Abhängigkeiten zwischen den Komponenten zu vermeiden.

Quasar definiert die primären Kategorien genau und legt auch die Regeln für die Kommunikation zwischen Kategorien sowie die Zuordnung von Komponenten zu Kategorien eindeutig fest (siehe Abschnitt 2.2.1). Woran es jedoch mangelt, ist eine allgemeine Beschreibung, wie die Softwarekategorien identifiziert werden können. Sicherlich bedarf die Identifikation der benötigten Softwarekategorien für eine Anwendung einiger Kreativität des Softwareingenieurs. Dennoch ist es für die Nachvollziehbarkeit der getroffenen Entscheidungen wichtig, ein definiertes Vorgehen zu haben. Nur so kann Traceability den Entwicklungsprozess vollkommen unterstützen. Daher werden im Folgenden die Anhaltspunkte, die Quasar zur Erstellung eines Kategorienmodells gibt, in einer eigenen Beschreibung des Vorgehens erweitert.

Als Ausgangspunkt und Input für das Kategorienmodell dient das Anwendungsfallmodell. Die Anwendungsfälle werden in der Regel weder 1:1 auf Komponenten noch auf Kategorien abgebildet. Dafür sind sie zu feingranular. Deshalb bietet es sich an, einen Zwischenschritt einzuführen. Als Zwischenergebnis von den Anwendungsfällen zu den Softwarekategorien bietet sich eine funktionale Zerlegung des Softwaresystems in Form eines Funktionsbaumes an, in dem die Funktionalitäten der Anwendungsfälle strukturiert und teilweise zusammengefasst werden.

4.3.1 Funktionsbaum als Zwischenschritt

Die Erstellung des Funktionsbaumes geschieht nach dem in [Herp07] vorgeschlagenen Verfahren. Dabei wird von den Anwendungsfällen und den Anwendungsfalldiagrammen ausgegangen. Die Abbildung des Anwendungsfallmodells auf einen Funktionsbaum ermöglicht eine übersichtliche Darstellung und die Illustration von Verfeinerungen zwischen Funktiona-

litäten. Funktionalitäten ergeben sich entweder direkt durch die Übernahme von Anwendungsfällen in den Funktionsbaum oder durch Zusammenfassung mehrerer Anwendungsfälle zu einer Funktionalität. Die Zuordnung der Anwendungsfälle zu Funktionen kann bei der Übernahme direkt mittels Traceability-Links für die spätere Nachvollziehbarkeit erfasst und dokumentiert werden.

Als Darstellungsmittel für den Funktionsbaum wird ein UML-Klassendiagramm mit eigenen Stereotypen verwendet. Jeder Baum bekommt als Wurzelelement eine Klasse mit dem Stereotyp *<<system>>*. Direkt darunter werden die Architekturschichten[11] des Softwaresystems, gekennzeichnet mit *<<architecture layer>>*, mittels Spezialisierungsbeziehung angeordnet. Alle aus den Anwendungsfällen ermittelten Anwendungsfunktionalitäten werden als Kind-Elemente der Architekturschichten bzw. Kind-Elemente anderer Funktionalitäten eingefügt und mit *<<system functionality>>* markiert. Neben den Generalisierungs- bzw. Spezialisierungsbeziehungen lassen sich im Funktionsbaum auch Assoziationen zwischen den Klassen verwenden, um Abhängigkeiten zwischen den verschiedenen Funktionen zu verdeutlichen.

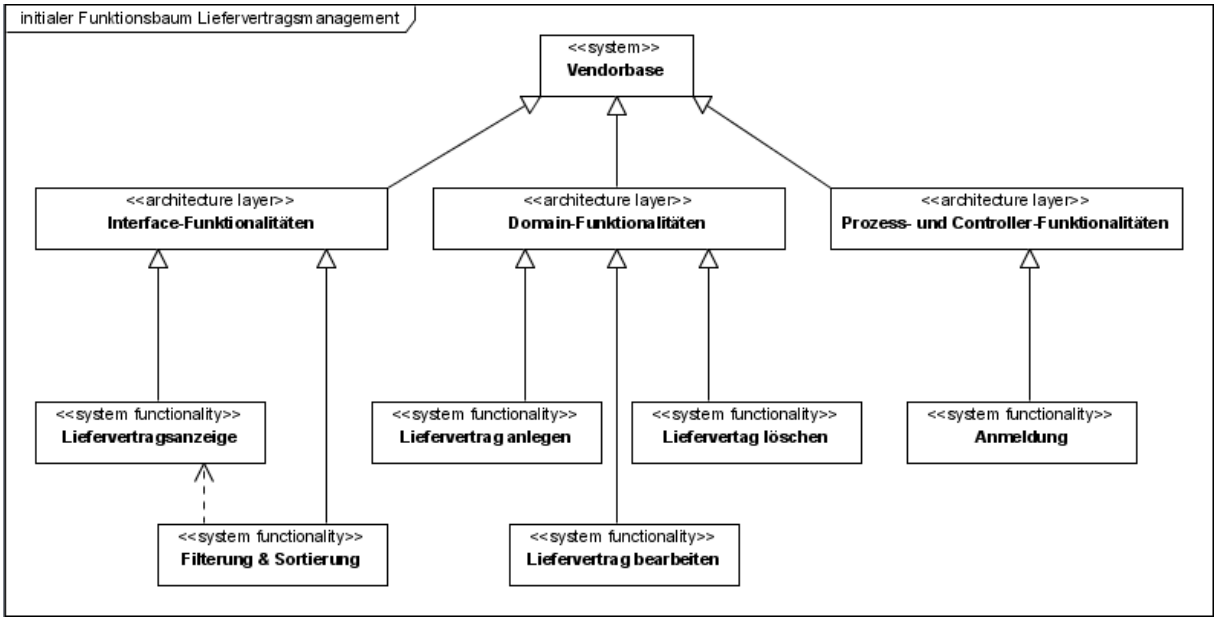

Abbildung 4-3: Ausschnitt aus dem initialen Funktionsbaum

Die Generierung des Funktionsbaumes erfolgt iterativ. In einem ersten Schritt werden zunächst alle Anwendungsfälle als Funktionalitäten übernommen und den einzelnen Architekturschichten zugeordnet. In Abbildung 4-3 ist ein Ausschnitt des initialen Funktionsbaumes für Vendorbase dargestellt. Aus Gründen der Übersicht zeigt die Abbildung nur den Bereich

[11] Zur Darstellung der verschiedenen Architekturschichten sei auf die Beschreibung in [Herp07] verwiesen.

Liefervertragsmanagement sowie die Funktionalität zur Anmeldung. Es ist die direkte Übernahme der Anwendungsfälle zu erkennen (vergleiche Abbildung 4-1 Anwendungsfalldiagramm zum Liefervertragsmanagement). Sie wurden auf die für sie zuständigen Architekturschichten aufgeteilt. Damit können Traceability-Links von den Anwendungsfällen zu den Systemfunktionalitäten gezogen werden. Abbildung 4-4 zeigt Traceability-Links zwischen den besprochenen Elementen. Zu beachten ist ebenfalls die Abhängigkeit zwischen den Funktionalitäten „Filterung & Sortierung“ sowie „Liefervertragsanzeige“. Diese ist auf die extend-Beziehung zwischen den Anwendungsfällen zurückzuführen. Die dargestellten Links werden, wie schon beim Datenmodell, in gleicher Art und Weise in tabellarischer Form festgehalten.

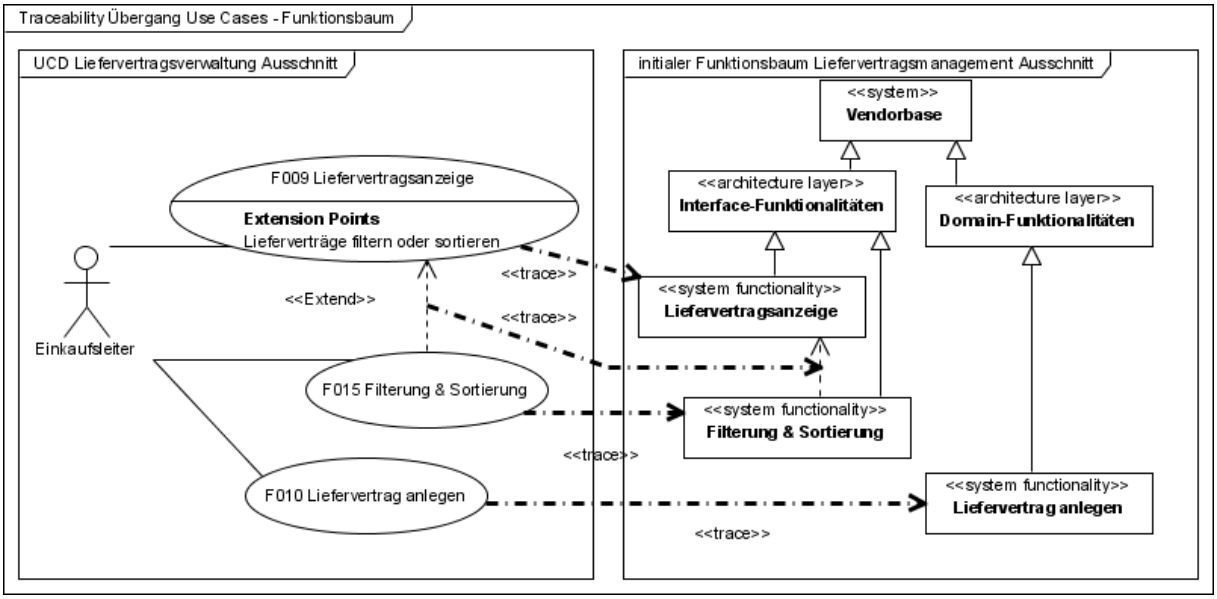

Abbildung 4-4: Traceability-Links zwischen Use Cases und Funktionalitäten

Bei weiteren Schritten werden anschließend Funktionalitäten, die mehrere Anwendungsfälle zusammenfassen, im Baum als Generalisierungen eingefügt. Die Funktionalitäten der Anwendungsfälle werden dabei nicht aus dem Baum entfernt, sondern bleiben unter den zusammenfassenden Funktionalitäten bestehen. Damit bleiben auch die Traceability-Links erhalten. Sollten Funktionalitäten nicht eindeutig einer Architekturschicht zuordenbar sein, müssen diese in Teilfunktionalitäten zerlegt werden. Abbildung 4-5 zeigt das Ergebnis einer Iteration des Vorgehens, bei der die Liefervertragsverwaltung als bündelnde Funktionalität eingeschoben wurde. Bei Schritten dieser Art können ebenfalls Traceability-Links zwischen den Funktionalitäten erstellt werden, welche die Entscheidungen zur Gruppierung dokumentieren.

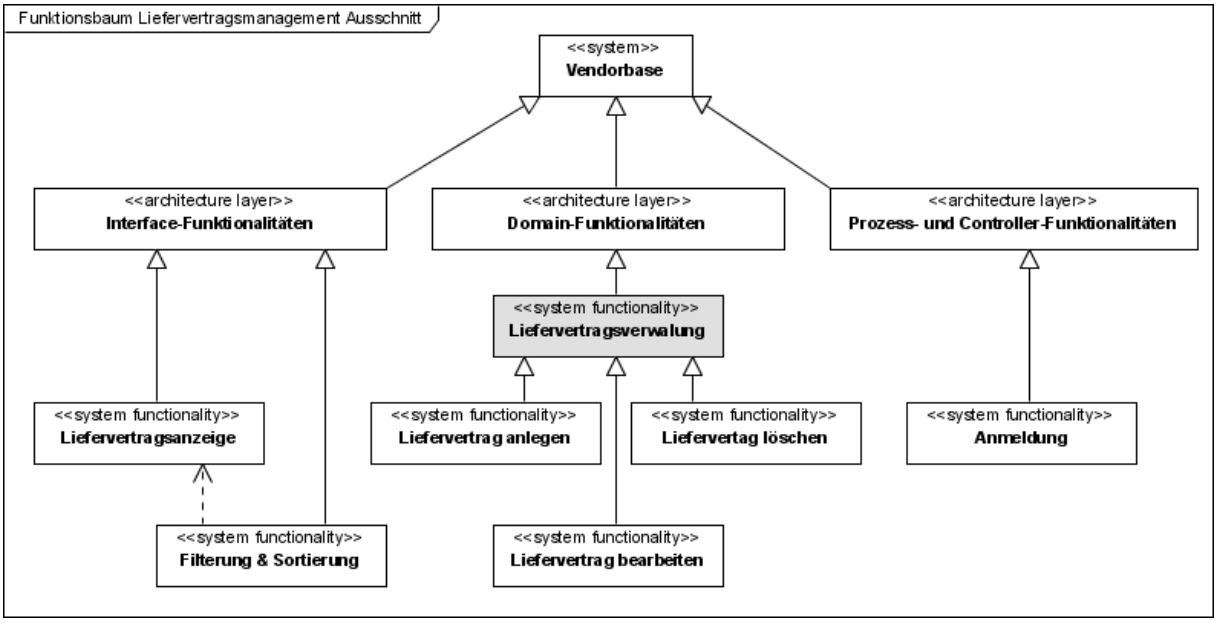

Abbildung 4-5: Ausschnitt mit Liefervertragsmanagement aus dem fertigen Funktionsbaum

Eine andere Möglichkeit des Vorgehens beim Erzeugen des Funktionsbaumes bestände darin, die Zusammenfassung von Anwendungsfällen schon bei der Übernahme als Funktionalitäten in den Funktionsbaum vorzunehmen. In diesem Fall wären nicht alle Anwendungsfälle direkt im Funktionsbaum wiederzufinden, sondern nur indirekt über die sie gruppierenden Funktionalitäten. Dies würde eine teilweise abstraktere und nicht so feingranulare Sicht auf den gesamten Funktionsumfang der Anwendung ermöglichen. Für die Traceability-Links hätte dies zur Folge, dass sie direkt von den Anwendungsfällen zu den Funktionalitäten, die die Anwendungsfälle zusammenfassen, erstellt werden müssten. Gleichzeitig müssten noch die Zuordnungen der Funktionalitäten zu den Architekturschichten dokumentiert werden. Hierbei bestände die Gefahr oder wäre nachteilig, dass Abhängigkeiten, die zwischen Anwendungsfällen bestehen, nicht in den Funktionsbaum übernommen werden oder dort nicht mehr sichtbar sind.

Ein weiterer vorstellbarer Schritt beim Aufbau eines Funktionsbaumes wäre nicht nur die Zusammenfassung von Funktionalitäten sondern auch deren Zerlegung. Dies ist im Hinblick auf den Zweck, den der Baum erfüllen soll, die Unterstützung bei der Identifikation der Softwarekategorien, eher kontraproduktiv und findet daher hier nicht statt. Eine weitere Aufteilung in Teilfunktionalitäten wird deshalb dem Entwurf der Komponenten und Schnittstellen überlassen.

Zur Vervollständigung des Quasar-Vorgehens ist die zuerst beschriebene Vorgehensweise gut geeignet. Werden zunächst alle Anwendungsfälle in den Funktionsbaum übernommen, hat dies den Vorteil einer gewissen Vollständigkeit. Bei der weiteren Arbeit mit dem Baum muss nicht auf das Anwendungsfallmodell zurückgegriffen werden, um zu erkennen, welche Anwendungsfälle zu einer zusammenfassenden Funktionalität gehören. Andererseits ist nach Bedarf zudem die Ausblendung detaillierter Funktionalitäten, die genau den Anwendungsfällen entsprechen, oder von Funktionalitäten auf verschiedenen Stufen im Baum denkbar.

Der vollständig nach dem beschriebenen Vorgehen aufgebaute Funktionsbaum für Vendorbase sowie die dabei erstellten Traceability-Links sind dem Anhang zu entnehmen. Sie stellen den Output dieser Aktivität dar. Der Funktionsbaum geht weiterhin direkt als Input in die nächste Aktivität der Festlegung der Softwarekategorien ein.

4.3.2 Festlegung der Softwarekategorien

Die Softwarekategorien sollen das in der Anwendung vorhandene Wissen und die Funktionalitäten widerspiegeln. Einen Überblick über die Funktionalitäten des Systems bietet der im vorigen Schritt erstellte Funktionsbaum und unterstützt deshalb bei den nächsten Schritten zur Identifikation der Kategorien.

Der Ausgangspunkt für die Ermittlung der Softwarekategorien der zu entwickelnden Anwendung stellt die 0-Software dar. Darauf aufbauend werden zunächst Kategorien auf der höchsten Abstraktionsebene des Softwaresystems gesucht. Anschließend erfolgt mithilfe der Informationen der vorangegangen Aktivitäten eine Verfeinerung. Mit der Erstellung der Softwarekategorien sind für alle Bausteine des Systems *„alle Abhängigkeiten […] explizit und verbindlich“* zu machen [Sie04, S. 73].

Für die Notation des Kategoriengraphen werden von Quasar keine Vorgaben gemacht. Aus diesem Grund verwendet der Autor wie in [Herp07] bei der Untersuchung der Kategorien für das SalesPoint-Framework ein UML-Klassendiagramm. Die einzelnen Kategorien werden als Klassen dargestellt. Entsprechend ihrer Zuordnung zu den Quasar-Hauptkategorien werden ihnen die Stereotypen *<<0-Software>>*, *<<A-Software>>*, *<<T-Software>>*,

<<*AT-Software*>> oder <<*R-Software*>> zugewiesen. Beziehungen zwischen Kategorien werden als Generalisierungsbeziehungen dargestellt.

Für die Ermittlung der Softwarekategorien dient nun der Funktionsbaum als wesentliches Hilfsmittel. Die enthaltenen Funktionalitäten werden entweder zusammengefasst und in eine Softwarekategorie überführt, oder einzelne Funktionalitäten werden direkt als eigene Softwarekategorie modelliert. Weiterhin bieten die im Datenmodell gefundenen Beziehungen zwischen den Entitäten Unterstützung bei der Festlegung der Kategorien und ihrer Abhängigkeiten.

Zunächst betrachten wir den Fall, die Beispielanwendung Vendorbase würde komplett neu entwickelt werden und nicht das SalesPoint-Framework verwenden. Dies ist der Fall, der auch in der Quasar-Beschreibung berücksichtigt ist. Denn die Quasar-Methode trifft keine Aussagen zu ihrer Anwendung bei der Entwicklung von Softwaresystem, die auf vorhandenen Softwarebausteinen aufbauen. Ausgehend von Kategorie 0 werden entsprechend der Architekturschichten verschiedene Kategorien vorgesehen. So sollten die Interface-Funktionalitäten in einer anderen Kategorie zu finden sein als die Domain- oder Prozess- und Controller-Funktionalitäten. Der Autor hält für Vendorbase die Kategorien für notwendig, welche im Kategoriengraphen in Abbildung 4-6 zu sehen sind.

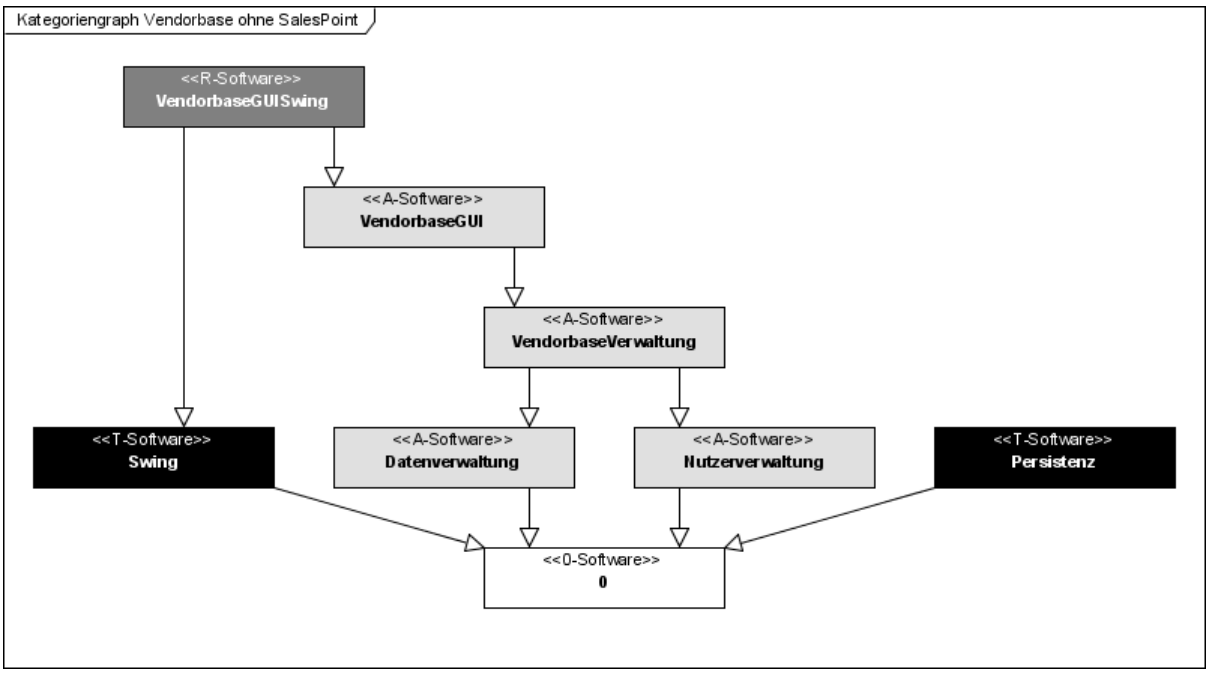

Abbildung 4-6: Kategoriengraph für Vendorbase ohne SalesPoint

Zunächst werden nun die einzelnen Kategorien mit ihrer Bedeutung erläutert. Auf die Traceability wird im Anschluss eingegangen.

Aus den Domain-Funktionalitäten ergeben sich die A-Kategorien *Datenverwaltung* und *Nutzerverwaltung*. Zur Kategorie Datenverwaltung gehören die System-Funktionalitäten der Liefervertrags-, Lieferanten- sowie Produktteilverwaltung. Eine weitere Trennung in Unterkategorien wird hier nicht vorgenommen. Dies ist aus dem Datenmodell ableitbar. Da die Entitäten Liefervertrag, Lieferant und Produktteil über Beziehungen miteinander verknüpft sind, erscheint eine Aufteilung der Funktionalitäten auf verschiedene Kategorien wenig sinnvoll. Für die Nutzerverwaltung hingegen ist eine eigene Kategorie angebracht, da Nutzer laut Datenmodell unabhängig von den anderen Entitäten existieren. Weiterhin wird in der Nutzerverwaltung auch die Berechtigungsverwaltung für die verschiedenen Nutzerrollen mit untergebracht.

Die Kategorien Datenverwaltung und Nutzerverwaltung werden von der Kategorie *VendorbaseVerwaltung* verfeinert. In dieser Kategorie werden alle Domain-Funktionalitäten der Daten- und Nutzerverwaltung zum Anwendungskern gebündelt. Weiterhin wird hier die Prozess- und Controller-Funktionalität zum An- und Abmelden der Nutzer untergebracht. So wäre es bei einem Multi-User-System beispielsweise möglich, entsprechend dem sich anmeldenden Nutzer die nötigen Komponenten zur Datenverwaltung zur Verfügung zu stellen.

Die letzte Softwarekategorie vom Typ Anwendung ist die Kategorie *VendorbaseGUI*. Ihr werden sämtliche Interface-Funktionalitäten zugeordnet. Diese Funktionalitäten werden vom Rest der Anwendung getrennt, da sie später durch Dialogkomponenten realisiert werden, die auf den Anwendungskern der Kategorie *VendorbaseVerwaltung* zugreifen. Damit ergibt sich die Abhängigkeitsbeziehung zwischen *VendorbaseGUI* und *VendorbaseVerwaltung*.

Die technische Kategorie *Swing* folgt aus der Tatsache, dass die Benutzerschnittstelle mit Java Swing realisiert werden soll. Sie enthält alle von Java bereitgestellten Swing-Komponenten sowie eigene Implementierungen.

Die Client-Architektur nach Quasar sieht, wie in Kapitel 2.2.5 vorgestellt, innerhalb der Dialoge eine Trennung in Dialogkern und Präsentation vor. Die Dialogkerne umfassen den fachlichen Teil der Dialoge und gehören bereits zur Kategorie *VendorbaseGUI*. Die Präsentatio-

nen hingegeben nehmen die Umsetzung der Fachlichkeit auf die technischen Swing-Komponenten vor. Damit haben wir es hier mit Repräsentationssoftware zu tun. Daraus resultiert die Softwarekategorie *VendorbaseGUISwing* als zulässige Vermischung von A- und T-Software zur Repräsentation von fachlichen Daten mittels Swing-Komponenten. Auf diese Weise ist es später z. B. möglich, die Swing-Benutzerschnittstelle durch ein Web-Interface zu ersetzen, indem einfach die Präsentationen geeignet ersetzt werden.

Zu guter Letzt bleibt noch die Kategorie *Persistenz*. Sie ist für die dauerhafte Speicherung der Anwendungsdaten verantwortlich. Persistenzkomponenten sollen unabhängig von fachlichen Dingen sein. Sie beschäftigen sich nur mit der technischen Realisierung der Datenspeicherung. Daraus resultiert die Kennzeichnung als T-Software.

Wie gezeigt wurde, ergeben sich die Kategorien relativ leicht unter Zuhilfenahme des Funktionsbaumes. Die Zuordnung der einzelnen Funktionalitäten und dabei getroffene Entscheidungen können infolge mittels Traceability-Links verfolgt werden. So wird beispielsweise die Aufteilung der Domain-Funktionalitäten auf die Datenverwaltung und die Nutzerverwaltung dokumentiert.

Das bisher beschriebene Vorgehen betrifft jedoch den Fall der Durchführung ohne Verwendung bereits existierender Softwarebausteine wie dem SalesPoint-Framework. Ziel dieser und zukünftiger Arbeiten ist jedoch eine Überarbeitung der Benutzerschnittstelle des Frameworks SalesPoint, und die Beispielanwendung Vendorbase basiert auf dem selbigen. Aus diesem Grund muss bei der Erstellung des Kategorienmodells für Vendorbase im Gegensatz zum hypothetischen ersten Fall das Framework berücksichtigt werden.

Für Vendorbase wurde von Herpel bereits ein geeignetes Kategorienmodell erstellt, dessen Kategoriengraph Abbildung 4-7 präsentiert. Den Kategorien in diesem Modell werden in der überarbeiteten Version von SalesPoint extrahierte Komponenten zugeordnet. So gehören beispielsweise die bereits fertiggestellten Komponenten *UserManager* und *Permissions* der A-Kategorie *Nutzerverwaltung* an. Für eine detaillierte Beschreibung der SalesPoint-Kategorien sei auf [Herp07] verwiesen.

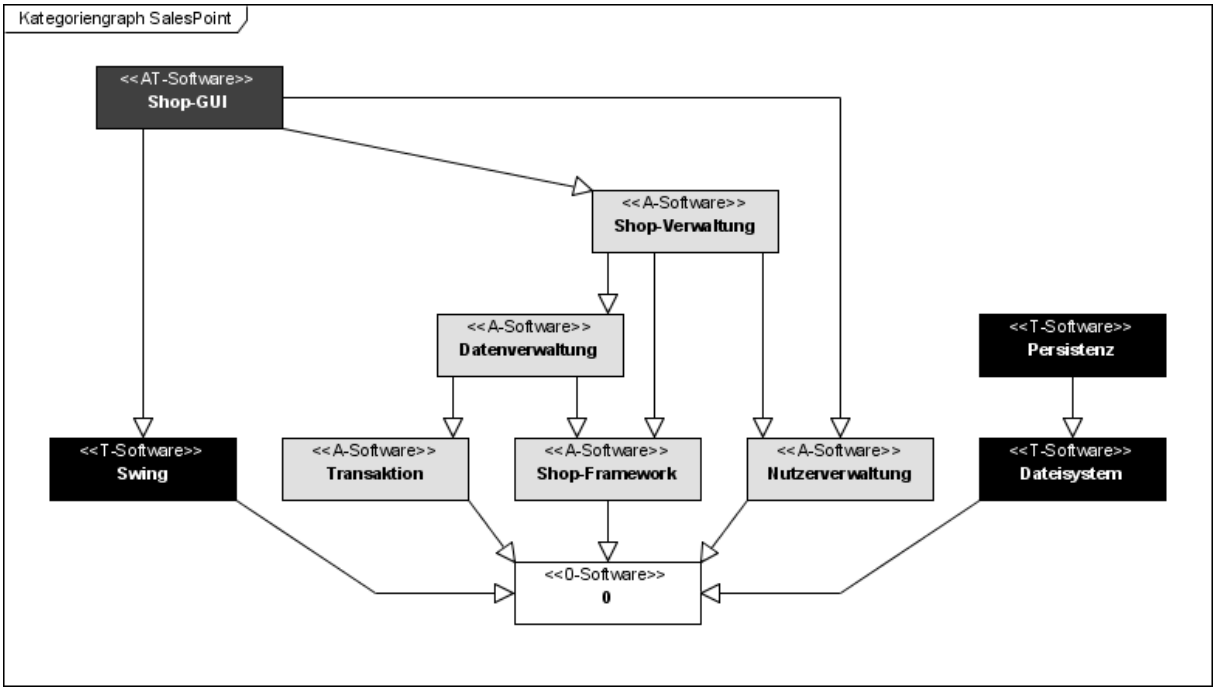

Abbildung 4-7: Kategorienmodell für das SalesPoint-Framework (vgl. [Herp07])

Quasar hinterlässt wie schon erwähnt keinen Hinweis zur Vorgehensweise bei Verwendung bereits existierender Softwarekomponenten. Daher schlägt der Autor im Weiteren folgendes Vorgehen bei Verwendung von Software mit bekanntem Kategoriengraphen, wie es auf SalesPoint zutrifft, vor.

Der vorhandene Kategoriengraph des Frameworks wird zunächst in das neu für Vendorbase zu erstellende Kategorienmodell übernommen. Dies folgt aus dem Fakt, dass Vendorbase soweit wie möglich die SalesPoint-Komponenten verwenden soll und diese anhand des existierenden Kategorienmodells erstellt wurden. Anschließend muss der übernommene Graph für die Belange von Vendorbase adaptiert werden.

Bei Betrachtung der Kategoriengraphen für SalesPoint und aus dem ersten Fall für Vendorbase fallen sofort einige Übereinstimmungen ins Auge. So existieren in beiden Graphen die technischen Kategorien *Swing* und *Persistenz*. Für den neuen Kategoriengraphen von Vendorbase auf Basis von SalesPoint existieren diese Kategorien bereits aufgrund der oben beschriebenen Übernahme der SalesPoint-Kategorien. Hier ist keine Änderung notwendig, da die genannten T-Kategorien anwendungsneutral sind.

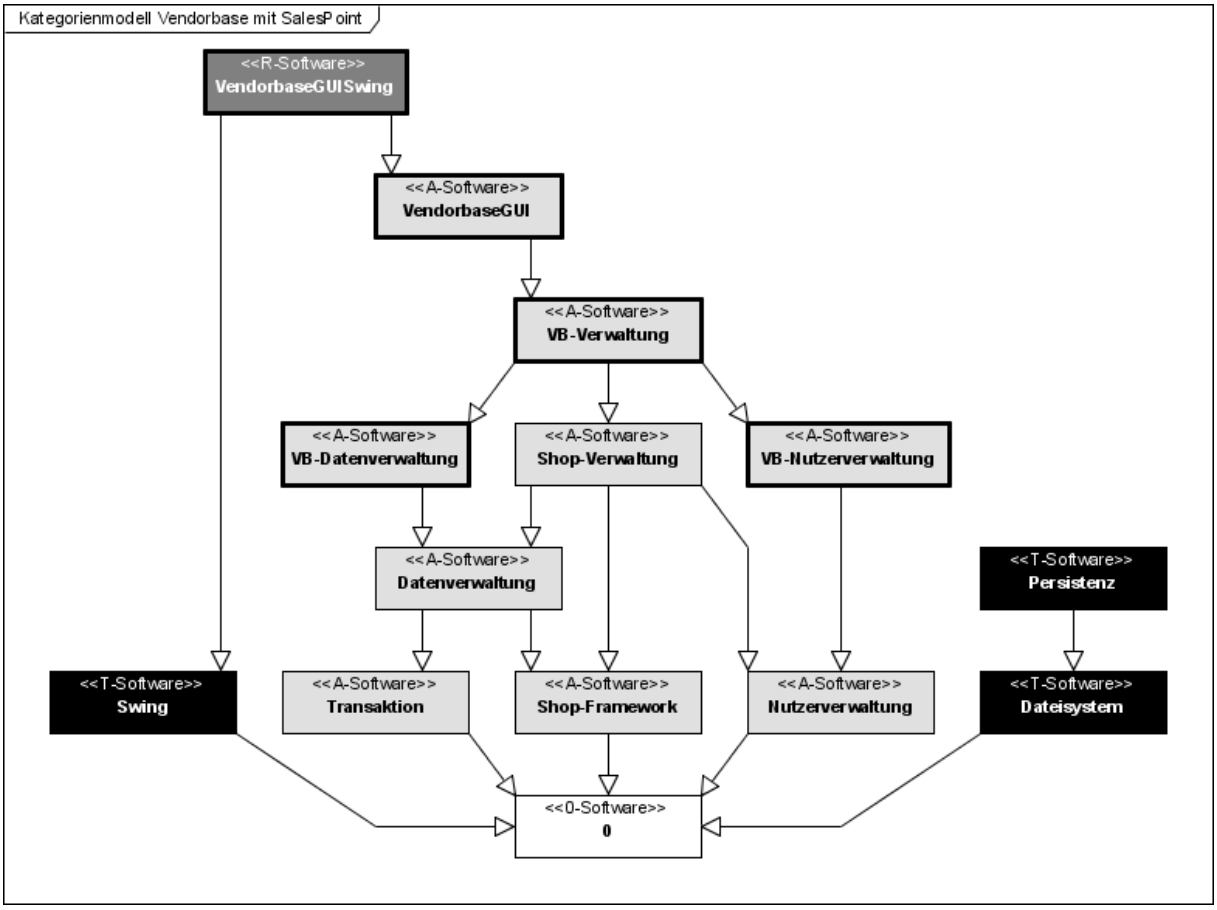

Abbildung 4-8: Kategoriengraph für Vendorbase mit SalesPoint

Bei den anwendungsspezifischen Kategorien ist in den Graphen eine Entsprechung bei *Datenverwaltung*, *Nutzerverwaltung* und *Shop-Verwaltung* bzw. *VB-Verwaltung* zu erkennen. Da die Vendorbase-Komponenten auf den SalesPoint-Komponenten basieren sollen, liegt an dieser Stelle die Verfeinerung der SalesPoint-Kategorien durch die entsprechenden Vendorbase-Kategorien nahe. Somit werden die übernommenen Kategorien wie in Abbildung 4-8 dargestellt durch die fett umrandeten Kategorien *VB-Datenverwaltung*, *VB-Nutzerverwaltung* und *VB-Verwaltung* ergänzt, welche in ihrer Funktion den Kategorien *Datenverwaltung*, *Nutzerverwaltung* und *VendorbaseVerwaltung* aus dem ersten Fall entsprechen (siehe Abbildung 4-6).

Die AT-Kategorie *Shop-GUI* von SalesPoint stellt eine unzulässige Vermischung von A- und T-Software dar. Wie in Kapitel 3.3 dargelegt, soll die zugehörige Komponente *SwingUI für SalesPoint* durch eine Neuentwicklung ersetzt werden. Daher wird die in den Kategoriengraphen für Vendorbase übernommene Kategorie *Shop-GUI* aus dem Graphen entfernt. An ihre Stelle treten die Kategorien *VendorbaseGUI* und *VendorbaseGUISwing*, wie sie auch im Modell ohne SalesPoint vorhanden sind.

Der vollständige Graph mit allen Softwarekategorien für Vendorbase ist in Abbildung 4-8 zu sehen. Die bei der Modellierung getroffenen Entscheidungen wurden mittels Traceability-Links dokumentiert und sind im Anhang enthalten. Das fertige Kategorienmodell als Output dieser Aktivität wird im weiteren Quasar-Vorgehen zusammen mit den anderen bisherigen Ergebnissen zur Modellierung der Komponenten und Schnittstellen verwendet.

4.4 Spezifikation der Komponenten und Schnittstellen

Nach den bisherigen Aktivitäten zur Erstellung der Modelle für die Anwendungsfälle, Daten und die Softwarekategorien folgt nun gemäß der Quasar-Methode der Aufbau des Komponentenmodells. Damit ist die Identifikation und Spezifikation der Komponenten und ihrer Schnittstellen gemeint. Die einzelnen Begriffe im Kontext von Quasar wurden bereits in Kapitel 2.2.4 zum Komponentenmodell erläutert, und zum Verständnis dieses Abschnitts sei dorthin verwiesen. Ebenfalls wurden dort bereits die einzelnen Schritte zum Design von Komponenten und Schnittstellen dargelegt. Diese sollen hier im Folgenden am Entwicklungsbeispiel Vendorbase angewendet werden.

Wie in Kapitel 2.2.5 schon erklärt wurde, eignen sich die Quasar-Schritte zur Identifikation von Komponenten nur bedingt für die Erstellung der grafischen Benutzerschnittstelle, sondern sind eher für die Komponenten des Anwendungskernes gedacht, weshalb alternative Ansätze zum GUI-Design untersucht wurden. Aus diesem Grund wird die nachfolgende Illustration des Vorgehens zur Festlegung der Komponenten und Schnittstellen für Vendorbase dementsprechend in zwei Teile untergliedert. Zunächst folgt die Beschreibung für den Anwendungskern.

An dieser Stelle soll noch kurz erläutert werden, warum überhaupt die Entwicklung des Anwendungskerns für Vendorbase betrachtet wird, obwohl aus den Zielen der Aufgabenstellung nur die Neuentwicklung der grafischen Benutzerschnittstelle hervorgeht. Die Abhängigkeiten zwischen Domain-Funktionalitäten und Interface-Funktionalitäten sind aufgrund der Verwendung des originalen SalesPoint-Frameworks in der bisherigen Vendorbase-Beispielanwendung sehr groß. Dies ist auf den in Kapitel 3.2 *Motivation zur Überarbeitung von SalesPoint* erläuterten Kritikpunkt der engen Verzahnung des SalesPoint-Frameworks mit Java-Swing

zurückzuführen. Aus diesem Grund ist eine separate Entwicklung der Benutzerschnittstelle nicht möglich und es müssen zugleich die Domain- und Prozess-Funktionalitäten überarbeitet werden. Dies ermöglicht darüber hinaus die Demonstration der Anwendung der Quasar-Methode für einen Anwendungskern am Beispiel Vendorbase.

4.4.1 Komponenten und Schnittstellen des Anwendungskerns

In diesem Abschnitt wird für Vendorbase der Teil des Komponentenmodells erstellt, der den Anwendungskern des Systems betrifft. Als Input dienen alle bisher erstellten Artefakte: Anwendungsfallmodell, Datenmodell, Funktionsbaum und Kategorienmodell. Zunächst einmal erfolgt nach dem Quasar-Vorgehen die Identifikation der Komponenten, bevor anschließend die Komponenten mit ihren Schnittstellen beschrieben und parallel die Schnittstellen spezifiziert werden. Eine anschauliche Darstellung wird durch Verwendung von Komponenten-, Kompositionsstruktur- und Paketdiagrammen der UML erreicht.

Identifikation der Komponenten

Als Ausgangspunkt dafür, die Komponenten des Anwendungskernes zu finden, dienen zunächst einmal das Datenmodell und der Funktionsbaum. So muss jede Entität aus dem Datenmodell entsprechend der Konsistenzbedingung zwischen Daten- und Komponentenmodell exklusiv einer Komponente zugeordnet werden. Das bedeutet, es dürfen sich nicht verschiedene Komponenten direkt mit derselben Entität beschäftigen. Des Weiteren gibt der Funktionsbaum eine Übersicht über die im System umzusetzenden Funktionalitäten. Er bietet eine hierarchisch gegliederte Struktur, die durch die Zusammenfassung von Funktionalitäten auf verschiedenen Ebenen im Baum bereits auf eine mögliche Aufteilung von Funktionen auf Komponenten hinweist. Ferner sind an der Untergliederung des Funktionsbaumes nach den Architekturschichten leicht die Funktionalitäten identifizierbar, welche die Benutzerschnittstelle und nicht den Anwendungskern betreffen.

Für Vendorbase bedeutet dies konkret, dass für den Anwendungskern Komponenten gefunden werden müssen, welche die Domain- sowie Prozess- und Controller-Funktionalitäten behan-

deln. Hier helfen die Systemfunktionalitäten, die in den Funktionsbaum eingefügt wurden, um von Anwendungsfällen abstammende Einzelfunktionalitäten zusammenzufassen (siehe Abschnitt 4.3.1). Dies sind *Liefervertragsverwaltung*, *Lieferantenverwaltung*, *Produktteilverwaltung* und *Nutzerverwaltung*. Daraus hervor gehen in diesem Fall eins zu eins die Komponenten *ContractManager*, *VendorManager*, *ProductManager* sowie *Authorization*. Den Komponenten werden im gleichen Atemzug die Entitäten *Contract*, *Vendor*, *Product* und *User* sowie die von ihnen verwendeten fachlichen Datentypen aus dem Datenmodell zugeordnet. Abbildung 4-9 veranschaulicht dies. Die dabei vorgenommenen Zuordnungen von Funktionalitäten bzw. Entitäten zu Komponenten werden mit Traceability-Links dokumentiert.

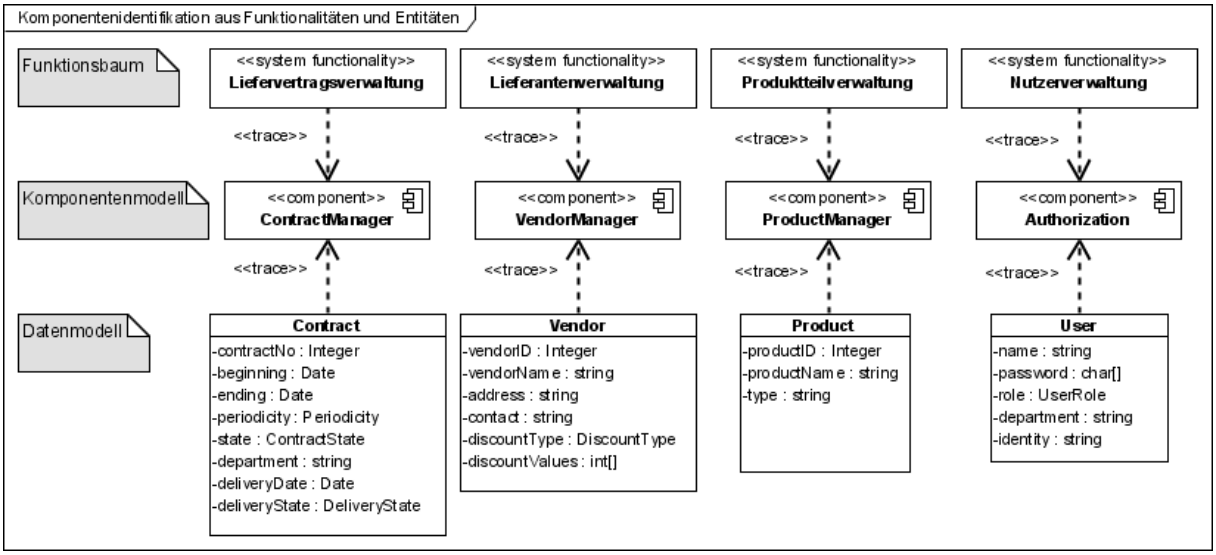

Abbildung 4-9: Komponentenidentifikation aus Funktionalitäten und Entitäten

Der Name *Authorization* für die Komponente zur Nutzerverwaltung deutet bereits darauf hin, dass hier nicht nur die Nutzer erzeugt, bearbeitet und gelöscht werden, sondern auch Berechtigungen verwaltet und überprüft werden können. Sie orientiert sich dabei an der in [Sie04] für Quasar-Anwendungen vorgeschlagenen Spezifikation.

Sind die Komponenten identifiziert, müssen sie entsprechend den Konsistenzbedingungen noch den erstellten Softwarekategorien zugeordnet werden. Die Zuordnung der Komponenten zu den Kategorien ergibt sich implizit aus der Abhängigkeit zwischen Funktionalitäten und Softwarekategorien. Hier wird eindeutig ein Vorteil des Funktionsbaumes als eingefügtem Zwischenschritt im Quasar-Vorgehen offensichtlich. Wäre der Kategoriengraph ohne Hilfe des Funktionsbaumes erstellt worden, müssten nicht nur die Komponenten direkt aus den Anwendungsfällen gefunden werden, was bedeutend schwerer fällt. Sondern die Festlegung

der Kategorie, in die eine Komponente gehört, müsste erst noch explizit getroffen werden. Mit dem Zwischenschritt liegt dies schon aufgrund der Verbindung zwischen Systemfunktionalität und Komponente auf der Hand. Die Verteilung der Komponenten auf die Softwarekategorien kann jedoch mit expliziten Traceability-Links dokumentiert werden.

Die für Vendorbase identifizierten Komponenten *ContractManager*, *VendorManager*, und *ProductManager* gehören demnach der Kategorie *VB-Datenverwaltung* an. Die Komponente *Authorization* ist der Kategorie *VB-Nutzerverwaltung* zugeordnet.

Als letzte Funktionalität, die für den Anwendungskern relevant ist und bisher nicht betrachtet wurde, bleibt noch die Systemfunktionalität *Anmeldung* übrig. Die Anmeldung eines Nutzers erfolgt am gesamten System. Sie stellt daher eine übergeordnete Funktionalität dar. Deshalb wird für sie eine Komponente in der Kategorie *VB-Verwaltung* erstellt. Diese Softwarekategorie verfeinert die Kategorien *VB-Datenverwaltung*, *VB-Nutzerverwaltung* und *Shop-Verwaltung*. In ihr wird die in Abbildung 4-10 dargestellte Komponente *Shop* identifiziert. Diese Komponente realisiert nicht nur die Funktionalität *Anmeldung*, sondern ist auch für die Erstellung und Verwaltung der Komponentenobjekte der anderen Komponenten *ContractManager*, *VendorManager*, *ProductManager* und *Authorization* verantwortlich. Die Komponente *Shop* agiert für sie als Kompositionsmanager, übernimmt die Konfiguration und repräsentiert als Einheit den Anwendungskern. Über den Anwendungskern greifen später die Dialogkomponenten auf die einzelnen Anwendungsfälle respektive die korrespondierenden Systemfunktionalitäten zu.

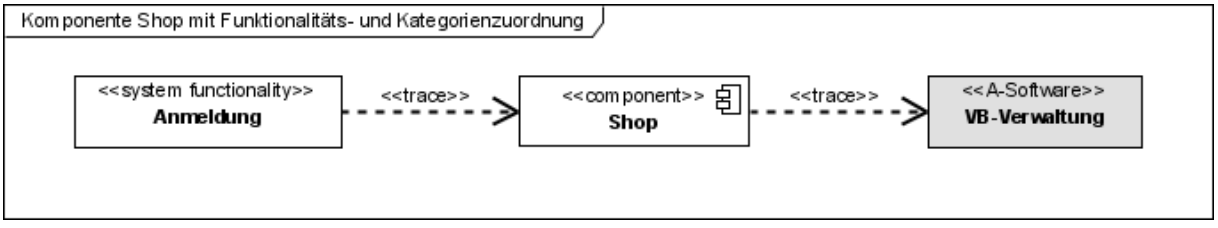

Abbildung 4-10: Identifikation der Komponente Shop

Entwurf der Komponentenschnittstellen

Nachdem die benötigten Komponenten identifiziert worden sind, müssen anschließend gemäß dem Vorgehen zur Komponentenbeschreibung die Schnittstellen identifiziert und spezifiziert

werden. Bei der Identifikation der Schnittstellen helfen wieder, wie bei den Komponenten, die Konsistenzbedingungen (siehe Kapitel 2.2.4). So muss z. B. jeder Anwendungsfall durch eine Schnittstelle abgedeckt und jede Schnittstelle genau einer Kategorie zugeordnet sein.

Für die einzelnen Komponenten *ContractManager*, *VendorManager* und *ProductManager* bedeutet dies, dass sie eine Schnittstelle bieten müssen, die das Anlegen, Bearbeiten und eventuell Löschen der jeweiligen Entität unterstützt. Diese Schnittstellen werden zunächst als objektorientierte Schnittstellen für enge Kopplung innerhalb der Softwarekategorie *VB-Datenverwaltung* entworfen, wie dies von Quasar vorgeschlagen wird. Dienstorientierte Schnittstellen für lose Kopplung können dann durch sogenannte Object-to-Service- oder kurz O2S-Adapter [Sie04, S. 189] nachgereicht werden. Für die objektorientierten Schnittstellen werden weiterhin Entitätsschnittstellen benötigt, da in ihnen die Entitäten nicht direkt als implementierende Klassen auftreten dürfen.

In Abbildung 4-11 sind die objektorientierten Programmschnittstellen für die einzelnen Systemfunktionalitäten sowie die Entitätsschnittstellen der verschiedenen Anwendungskernkomponenten dargestellt. Bei der Komponente *ContractManager* beispielsweise ist die Schnittstelle *IContractManager* für die Systemfunktionalitäten der *Liefervertragsverwaltung*, also *Liefervertrag anlegen*, *Liefervertrag bearbeiten* und *Liefervertrag löschen*, zuständig. Für die Entität *Contract* aus dem Datenmodell wurde die Entitätsschnittstelle *IContract* erstellt. Außerdem benötigt die Komponente *ContractManager* die Schnittstellen von *VendorManager*, was auf die entsprechende Assoziation im Datenmodell zurückzuführen ist. Solche Verknüpfungen werden mit Traceability-Links nachvollziehbar dokumentiert. Die Komponenten von Vendorbase sollen durch Unterstützung des SalesPoint-Frameworks erstellt werden. Daher importiert die Komponente *ContractManager* das Interface *Catalog* von SalesPoint aus der Kategorie *Datenverwaltung*. Diese benötigte Schnittstelle muss später durch die Konfiguration zur Verfügung gestellt werden.

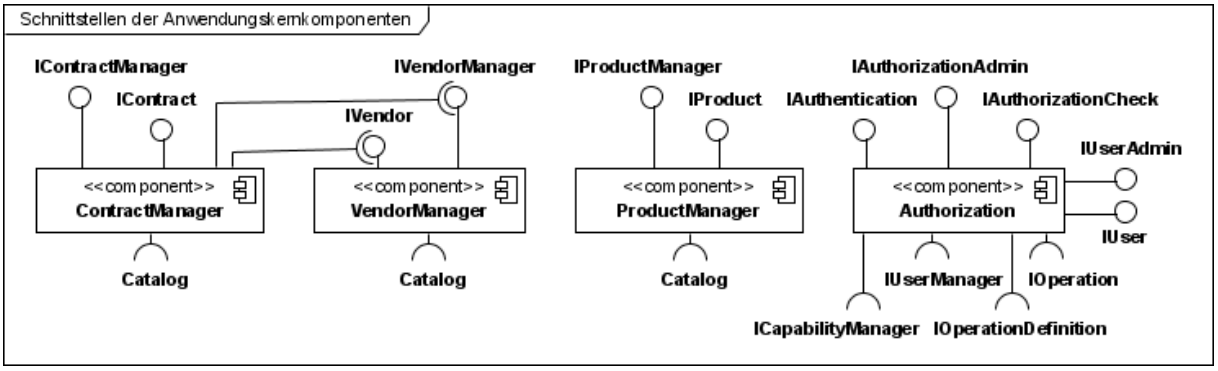

Abbildung 4-11: Schnittstellen der Anwendungskernkomponenten

Die Komponenten *VendorManager* und *ProductManager* gleichen der Komponente *ContractManager* und ihre angebotenen Schnittstellen gehören derselben Kategorie *VB-Datenverwaltung* an. *VendorManager* und *ProductManager* müssen ebenfalls jeweils mit einer SalesPoint-Implementierung der Schnittstelle *Catalog* konfiguriert werden. Die Komponente *Authorization* hingegen ist etwas umfangreicher, da sie daneben Funktionen zur Berechtigungsadministration und -prüfung anbietet. Ihre angebotenen Schnittstellen gehören zur Kategorie *VB-Nutzerverwaltung*. Die Schnittstelle *IUser* ist eine Entitätsschnittstelle, wie auch *IContract* bei *ContractManager*. Die anderen angebotenen Programmschnittstellen orientieren sich an den in [Sie04] und [HHS04] vorgeschlagenen Standardschnittstellen für eine Autorisationskomponente und werden für Vendorbase adaptiert. *IAuthentication* und *IAuthorizationCheck* sind operative *IUserAdmin* und *IAuthorizationCheck* administrative Schnittstellen. *IOperationDefinition* stellt eine benötigte Callback-Schnittstelle dar, über die die Operationen definiert werden, die im System verfügbar sind und bestimmten Berechtigungen unterliegen. *IOperation* ist lediglich ein Marker-Interface für diese Operationen. Darüber hinaus importiert die Autorisationskomponente ebenfalls Schnittstellen von SalesPoint. Dies sind *IUserManager* und *ICapabilityManager* aus der Softwarekategorie *Nutzerverwaltung*, die durch die Konfiguration bereitgestellt werden müssen.

Die Komponente *Shop* bietet eine Schnittstelle *IShop* an, über welche die Funktionalität *Anmeldung* aufzurufen ist. Da sie den gesamten Anwendungskern repräsentiert, ist sie weiterhin dafür verantwortlich, die anderen Anwendungskernkomponenten zu konfigurieren. Deshalb definiert sie die gesamten durch Berechtigungen geschützten Operationen und stellt für die *Authorization*-Komponente eine Implementierung für das Callback-Interface *IOperationDefinition* zur Verfügung. Darüber hinaus muss sie für den Zugriff der Dialogkomponenten auf die Anwendungsfallfunktionen serviceorientierte Schnittstellen der Komponenten *ContractManager*, *VendorManager* und *ProductManager* sowie *Authorization* expor-

tieren, welche von den Subkomponenten bereitgestellt werden. Die Komponente *Shop* mit ihren Schnittstellen wird in Abbildung 4-12 verdeutlicht.

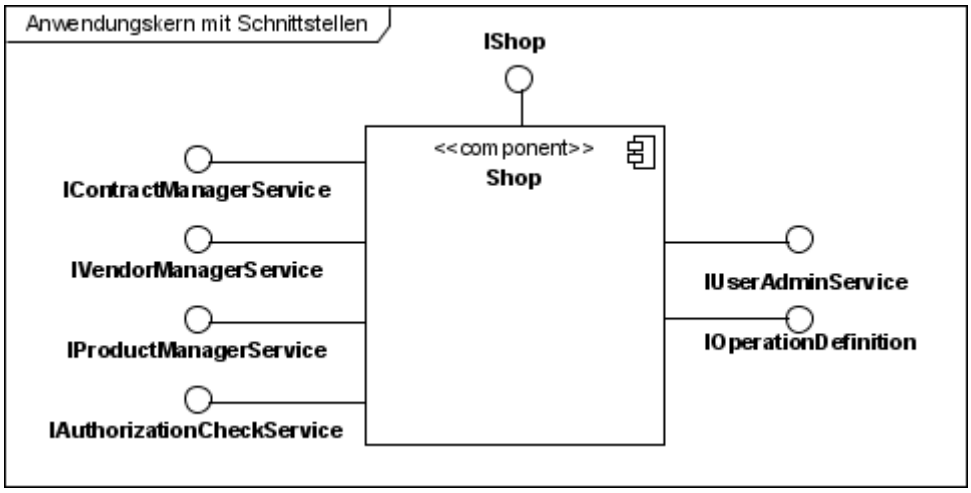

Abbildung 4-12: Komponente Shop als Anwendungskern mit Schnittstellen

An diesem Punkt verzichtet der Autor für die Komponente *Shop*, den Anwendungskern von Vendorbase, auf eine Verwendung weiterer Teile des SalesPoint-Frameworks wie der Shop-Verwaltung. Der Grund liegt darin, dass das Refactoring zu Komponenten für SalesPoint noch nicht abgeschlossen ist (siehe Kapitel 3.3). So wurde z. B. die Abgrenzung der SalesPoint-Komponente *Shop* noch nicht vollzogen. Durch eine Verwendung der Klassen *Shop* oder *SalesPoint* des Frameworks, welche die Prozess- und Controller-Funktionalitäten unterstützen, würden nicht gewollte Abhängigkeiten in Vendorbase eingeführt. Deshalb werden keine weiteren Bausteine des Frameworks eingesetzt. Eine Folge dessen ist, dass die Anwendungskernimplementierung keinen Multi-User-Support bieten wird, da die Prozess- und Transaktionsverwaltung keine Verwendung findet. Jedoch war auch die ursprüngliche Beispielanwendung Vendorbase nicht dazu geeignet, mehrere angemeldete Nutzer zu simulieren, sodass durch die Nichtunterstützung dieser Funktionalität kein Qualitätsverlust im Vergleich zur vorherigen Anwendung besteht. Darüber hinaus liegt das Augenmerk bei der Überarbeitung, wie zu Beginn dieses Kapitels herausgestellt, auf der Neuentwicklung der Benutzerschnittstelle von Vendorbase. Aus diesem Grund, und um den Aufwand im Rahmen zu halten, wird es ebenfalls unterlassen, eine persistente Speicherung der einzelnen Daten wie den Lieferverträgen und Lieferanten zu realisieren. Dies war in SalesPoint mittels Serialisierung vorgesehen. In einer späteren Stufe der Überarbeitung des Frameworks könnte dafür eine andere Lösung wie z. B. ein Persistenz-Framework eingesetzt werden, mit dem die Fachklassen auf Relationen einer Datenbank abgebildet werden.

Nachdem für die Vendorbase Komponenten alle Schnittstellen identifiziert worden sind, müssen sie genauer spezifiziert werden. Dazu gehört die Beschreibung all ihrer Methoden mit charakterisierenden Eigenschaften wie Syntax und Semantik. Dies erfolgt mithilfe der Quasar Specification Language (QSL) wie im Abschnitt zur *Spezifikation von Schnittstellen* in Kapitel 2.2.4 aufgezeigt wurde.

Für Vendorbase soll auszugsweise die Programmschnittstelle der Komponente *ContractManager* dargestellt werden, um beispielhaft die einzelnen QSL-Spezifikationselemente zu veranschaulichen. Die gesamten Schnittstellenspezifikationen für alle Komponenten von Vendorbase befinden sich im Anhang dieser Arbeit.

Listing 4-1 zeigt die Schnittstellenspezifikation für die Schnittstelle *IContractManager*. Wie zu erkennen ist, sind die QSL-Ausdrücke mit den semantischen Informationen der Interface-Spezifikation in den Javadoc-Kommentaren der Quellcode-Datei untergebracht. So sind die Informationen nicht nur im Quelltext sondern auch später in der Java-Dokumentation ersichtlich. In den Zeilen 9-26 des Listings sind die QSL-Angaben zu finden, die das gesamte Interface betreffen:

- eine Angabe, ob die Schnittstelle exportiert oder importiert wird (`export`),
- ob sie operativ oder administrativ ist (`operation`),
- die als Parameter genutzten Schnittstellen und Klassen (`uses`),
- die verwendeten Variablen für das Design-by-Contract (`variables`) sowie
- die Invarianten (`invariants`).

Bei den jeweiligen Methoden wird im Javadoc-Kommentar angegeben, ob es sich um

- eine einfache Abfrage (`basicQuery`),
- eine abgeleitete Abfrage (`derivedQuery`) oder
- ein Kommando (`command`) handelt.

Wenn sinnvolle Vor- und Nachbedingungen für eine Methode möglich sind, wurden diese mit den Schlüsselworten `pre` und `post` in den einzelnen Kommentaren der Methoden notiert.

Die Testfälle für die Schnittstelle hingegen sind nicht mit QSL formuliert worden. Hier überwogen die Vorteile, die Tests direkt als JUnit-Testcases[12] zu programmieren. Die Tests wurden so implementiert, dass sie die Vor- und Nachbedingungen sowie Invarianten aus der QSL-Spezifikation kontrollieren. Die JUnit-Tests können auf diese Weise immer wieder gestartet, und damit die Implementierungen der Komponenten gegen die Schnittstellenspezifikationen geprüft werden. Der Testfall für die Schnittstelle *IContractManager* und die implementierende Komponente *ContractManager* wird in Listing 4-2 präsentiert. Bei genauer Betrachtung der Schnittstellenspezifikation in Listing 4-1 und des JUnit-Tests in Listing 4-2 wird die Korrespondenz der `assert` Anweisungen zu den QSL-Notationen für die Vor- und Nachbedingungen sowie Invarianten deutlich. Die Zuordnung der JUnit-Testcase-Klassen zu den Schnittstellen und ihren implementierenden Klassen wird mit Traceability-Links dokumentiert.

[12] JUnit ist ein Framework für automatisierte Tests in Java, http://www.junit.org/

```
 1 package vendorbase.shop.contractmanager;
 2
 3 import java.util.Date;
 4 import java.util.List;
 5
 6 import vendorbase.shop.vendormanager.IVendor;
 7
 8 /**
 9  * [export, operation] Interface for contract management functionality.
10  * <pre>
11  * [uses
12  *             EContractState
13  *             EDeliveryState
14  *             EPeriodicity
15  *             IContract
16  *             IVendor
17  *             ContractItem
18  *
19  * variables
20  *             Integer contractNo, IContract contract
21  *
22  * invariants
23  *             getContract(contractNo) in getContracts()
24  *             updateContract(contract) in getContracts()
25  *             not exits IContract x,y : x.getContractNo() == y.getContractNo()
26  * ]
27  * </pre>
28  *
29  * @author Stephan Bode
30  * @version 1.0 03.10.2007
31  */
32 public interface IContractManager {
33
34     // basicQueries
35
36     /**
37      * [basicQuery] Returns all contracts in a {@link List} or an empty
38      * {@link List} if there is none.
39      *
40      * @return a {@link List} of {@link IContract} objects
41      */
42     List<IContract> getContracts();
43
44     /**
45      * [basicQuery] Returns a {@link IContract} object by its contract number.
46      *
47      * @param contractNo    the number of the contract
48      * @return the contract object for the given contract number
49      *
50      * <pre>
51      * [pre  exists contract : contract.getContractNo() == contractNo
52      * post  result.getContractNo() == contractNo
53      * error unknownContractNo
54      * ]
55      * </pre>
56      */
57     IContract getContract(Integer contractNo);
58
```

```
59      /**
60       * [basicQuery] Returns the next number to be assigned with a new contract.
61       *
62       * @return the next number for a new contract
63       *
64       * <pre>
65       * [post forall contract in getContracts():contract.getContractNo() < result
66       * ]
67       * </pre>
68       */
69      int getNextContractNo();
70
71      // commands
72      /**
73       * [command] Creates a new contract without items. Only to use if contract's
74       * state is {@link EContractState}<code>.DRAFT</code> because it must be
75       * changeable after creation.
76       *
77       * @param beginning             the contract's date of beginning
78       * @param ending                the contract's date of ending
79       * @param periodicity           the contract's periodicity of delivery
80       * @param contractState         the contract's state
81       * @param creationDepartment    the department which creates the contract
82       * @param deliveryDate          the contract's date of delivery
83       * @param deliveryState         the contract's delivery state
84       * @return a {@link IContract} object for the newly created contract
85       *
86       * <pre>
87       * [pre beginning <= deliveryDate <= ending
88       * post result.getBeginning().equals(beginning)
89       *      result.getEnding().equals(ending)
90       *      result.getPeriodicity().equals(periodicity)
91       *      result.getContractState().equals(contractState)
92       *      result.getDepartment().equals(creationDepartment)
93       *      result.getDeliveryDate().equals(deliveryDate)
94       *      result.getDeliveryState().equals(deliveryState)
95       *      result.getItems().isEmpty()
96       *      getContracts().size() == 'getContracts().size() + 1
97       * ]
98       * </pre>
99       */
100     IContract createContract(Date beginning, Date ending,
101             EPeriodicity periodicity, EContractState contractState,
102             String creationDepartment, Date deliveryDate,
103             EDeliveryState deliveryState);
104
```

```
105     /**
106      * [command] Creates a new contract immediately with items. Must be used if
107      * contract is created with {@link EContractState}<code>.ACTIVE</code>
108      * because it's unchangeable afterwards.
109      *
110      * @param beginning             the contract's date of beginning
111      * @param ending                the contract's date of ending
112      * @param periodicity           the contract's periodicity of delivery
113      * @param contractState         the contract's state
114      * @param creationDepartment    the department which creates the contract
115      * @param deliveryDate          the contract's date of delivery
116      * @param deliveryState         the contract's delivery state
117      * @param vendor                the contract's vendor
118      * @param items                 the contract's items
119      * @return a {@link IContract} object for the newly created contract
120      *
121      * <pre>
122      * [pre beginning <= deliveryDate <= ending
123      * post result.getBeginning().equals(beginning)
124      *      result.getEnding().equals(ending)
125      *      result.getPeriodicity().equals(periodicity)
126      *      result.getContractState().equals(contractState)
127      *      result.getDepartment().equals(creationDepartment)
128      *      result.getDeliveryDate().equals(deliveryDate)
129      *      result.getDeliveryState().equals(deliveryState)
130      *      result.getVendor().equals(vendor)
131      *      result.getItems().equals(items)
132      *      getContracts().size() == 'getContracts().size() + 1
133      * ]
134      * </pre>
135      */
136     IContract createContract(Date beginning, Date ending,
137             EPeriodicity periodicity, EContractState contractState,
138             String creationDepartment, Date deliveryDate,
139             EDeliveryState deliveryState, IVendor vendor,
140             List<ContractItem> items);
141
142     /**
143      * [command] Sets the items of a contract.
144      *
145      * @param contractNo    the number of the contract to set the items for
146      * @param items         the contract's items
147      *
148      * <pre>
149      * [pre  exists contract : contract.getContractNo() == contractNo
150      * post  getContract(contractNo).getItems().equals(items)
151      * error unknownContractNo
152      * ]
153      * </pre>
154      */
155     void setItems(Integer contractNo, List<ContractItem> items);
156
```

```
157     /**
158      * [command] Updates a contract's data.
159      *
160      * @param contractNo            the number of the contract to update
161      * @param beginning             the beginning date to update to
162      * @param ending                the ending date to update to
163      * @param periodicity           the periodicity to update to
164      * @param contractState         the state of the contract to update to
165      * @param creationDepartment    the creation department to update to
166      * @param deliveryDate          the date of delivery to update to
167      * @param deliveryState         the delivery state to update to
168      * @param vendor                the contract's vendor to update to
169      * @param items                 the contract's items to update to
170      * @return a {@link IContract} object for the updated contract
171      *
172      * <pre>
173      * [pre getContract(contract.getContractNo()) in getContracts()
174      * post result.getBeginning().equals(beginning)
175      *      result.getEnding().equals(ending)
176      *      result.getPeriodicity().equals(periodicity)
177      *      result.getContractState().equals(contractState)
178      *      result.getDepartment().equals(creationDepartment)
179      *      result.getDeliveryDate().equals(deliveryDate()
180      *      result.getDeliveryState().equals(deliveryState()
181      *      result.getVendor().equals(vendor)
182      *      result.getItems().equals(items)
183      *      result.equals(getContract(contractNo))
184      * ]
185      * </pre>
186      */
187     IContract updateContract(Integer contractNo, Date beginning, Date ending,
188             EPeriodicity periodicity, EContractState contractState,
189             String creationDepartment, Date deliveryDate,
190             EDeliveryState deliveryState, IVendor vendor,
191             List<ContractItem> items);
192
193     /**
194      * [command] Deletes a contract by number.
195      *
196      * @param contractNo    the number of the contract to delete
197      *
198      * <pre>
199      * [pre exits contract : contract.getContractNo() == contractNo
200      * post not exists contract : contract.getContractNo() == contractNo
201      *      getContracts().size() == 'getContracts().size() - 1
202      * ]
203      * </pre>
204      */
205     void deleteContract(Integer contractNo);
206 }
```

Listing 4-1: Spezifikation der Schnittstelle IContractManager

```java
package vendorbase.shop.contractmanager.test;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import data.ooimpl.CatalogImpl;

import junit.framework.TestCase;
import vendorbase.shop.contractmanager.ContractItem;
import vendorbase.shop.contractmanager.EContractState;
import vendorbase.shop.contractmanager.EDeliveryState;
import vendorbase.shop.contractmanager.EPeriodicity;
import vendorbase.shop.contractmanager.IContract;
import vendorbase.shop.contractmanager.IContractManager;
import vendorbase.shop.contractmanager.impl.ContractManager;
import vendorbase.shop.vendormanager.EDiscountType;
import vendorbase.shop.vendormanager.IVendor;
import vendorbase.shop.vendormanager.IVendorManager;
import vendorbase.shop.vendormanager.impl.VendorManager;

/**
 * Test case belonging to the {@link IContractManager} interface specification.
 *
 * @author Stephan Bode
 * @version 1.0 24.11.2007
 */
public class ContractManagerTest extends TestCase {

    private IContract c;
    private IContractManager cm;
    private Date beginning = new Date();
    private Date deliveryDate = new Date();
    private Date ending = new Date();
    private EPeriodicity periodicity = EPeriodicity.SINGLE;
    private EContractState contractState = EContractState.DRAFT;
    private String creationDepartment = "department";
    private EDeliveryState deliveryState = EDeliveryState.OPEN;
    private List<ContractItem> items = new ArrayList<ContractItem>();
    private ContractItem ci = new ContractItem(100, 22, 9);
    private IVendor vendor;

    @Override
    protected void setUp() throws Exception {

        items.add(ci);
        IVendorManager vm = new VendorManager(new CatalogImpl(
                VendorManager.CATALOG_NAME));
        vendor = vm.createVendor("Vendor1", "Adresse", "Kontakt",
                EDiscountType.TOTAL, new int[] {});
        cm = new ContractManager(new CatalogImpl(ContractManager.CATALOG_NAME));
    }

    public void testGetContracts() {

        assertTrue(cm.getContracts().isEmpty());
        cm.createContract(beginning, ending, periodicity, contractState,
                creationDepartment, deliveryDate, deliveryState);
        assertFalse(cm.getContracts().isEmpty());
    }

```

```
62      public void testGetContract() {
63
64          c = cm.createContract(beginning, ending, periodicity, contractState,
65                  creationDepartment, deliveryDate, deliveryState);
66          assertEquals(c, cm.getContract(c.getContractNo()));
67          assertTrue(cm.getContracts()
68                  .contains(cm.getContract(c.getContractNo())));
69          assertTrue(cm.getContracts().contains(c));
70      }
71
72      public void testGetNextContractNo() {
73
74          cm.createContract(beginning, ending, periodicity, contractState,
75                  creationDepartment, deliveryDate, deliveryState);
76          cm.createContract(beginning, ending, periodicity, contractState,
77                  creationDepartment, deliveryDate, deliveryState);
78          int id = -1;
79          for (IContract c : cm.getContracts()) {
80              assertTrue(id != c.getContractNo());
81              id = c.getContractNo();
82              assertTrue(c.getContractNo() < cm.getNextContractNo());
83          }
84      }
85
86      public void testCreateContractDateDateEPeriodicityEContractState
                                                        StringDateEDeliveryState() {
87
88          int oldSize = cm.getContracts().size();
89          c = cm.createContract(beginning, ending, periodicity, contractState,
90                  creationDepartment, deliveryDate, deliveryState);
91          assertEquals(beginning, c.getBeginning());
92          assertEquals(ending, c.getEnding());
93          assertEquals(periodicity, c.getPeriodicity());
94          assertEquals(contractState, c.getState());
95          assertEquals(creationDepartment, c.getDepartment());
96          assertEquals(deliveryDate, c.getDeliveryDate());
97          assertEquals(deliveryState, c.getDeliveryState());
98          assertTrue(cm.getContracts().size() == oldSize + 1);
99          assertTrue(cm.getContracts().contains(c));
100     }
101
102     public void testCreateContractDateDateEPeriodicityEContractState
                                        StringDateEDeliveryStateListOfContractItem() {
103
104         int oldSize = cm.getContracts().size();
105         c = cm.createContract(beginning, ending, periodicity, contractState,
106                 creationDepartment, deliveryDate, deliveryState, vendor, items);
107         assertEquals(beginning, c.getBeginning());
108         assertEquals(ending, c.getEnding());
109         assertEquals(periodicity, c.getPeriodicity());
110         assertEquals(contractState, c.getState());
111         assertEquals(creationDepartment, c.getDepartment());
112         assertEquals(deliveryDate, c.getDeliveryDate());
113         assertEquals(deliveryState, c.getDeliveryState());
114         assertEquals(vendor, c.getVendor());
115         assertEquals(items, c.getItems());
116         assertTrue(cm.getContracts().size() == oldSize + 1);
117         assertTrue(cm.getContracts().contains(c));
118     }
119
```

```
120     public void testSetItems() {
121
122         c = cm.createContract(beginning, ending, periodicity, contractState,
123                 creationDepartment, deliveryDate, deliveryState);
124         assertNotSame(items, c.getItems());
125         cm.setItems(c.getContractNo(), items);
126         assertEquals(items, c.getItems());
127         assertTrue(cm.getContracts().contains(c));
128     }
129
130     public void testUpdateContract() {
131
132         c = cm.createContract(beginning, ending, periodicity, contractState,
133                 creationDepartment, deliveryDate, deliveryState);
134         int id = c.getContractNo();
135         Date beginning = new Date();
136         Date deliveryDate = new Date();
137         Date ending = new Date();
138         String department = "purchasing";
139         c = cm.updateContract(id, beginning, ending, EPeriodicity.MONTHLY,
140                 EContractState.ACTIVE, department, deliveryDate,
141                 EDeliveryState.DONE, vendor, items);
142         assertEquals(beginning, c.getBeginning());
143         assertEquals(ending, c.getEnding());
144         assertEquals(EPeriodicity.MONTHLY, c.getPeriodicity());
145         assertEquals(EContractState.ACTIVE, c.getState());
146         assertEquals(department, c.getDepartment());
147         assertEquals(deliveryDate, c.getDeliveryDate());
148         assertEquals(EDeliveryState.DONE, c.getDeliveryState());
149         assertEquals(vendor, c.getVendor());
150         assertEquals(items, c.getItems());
151         assertTrue(c.getContractNo().equals(id));
152         assertTrue(cm.getContracts().contains(c));
153     }
154
155     public void testDeleteContract() {
156
157         c = cm.createContract(beginning, ending, periodicity, contractState,
158                 creationDepartment, deliveryDate, deliveryState);
159         assertTrue(cm.getContracts().contains(c));
160         cm.deleteContract(c.getContractNo());
161         assertFalse(cm.getContracts().contains(c));
162     }
163
164 }
```

Listing 4-2: Spezifikation des Testfalles ContractManagerTest

Validierung und Konfiguration

An diesem Punkt nach dem Entwurf der Komponentenschnittstellen angelangt, ist der wohl schwierigste Teil bei der Erstellung des Komponentenmodells geschafft. In nachfolgenden Schritten sollen nach Quasar die Schnittstellen validiert und ihre Implementierbarkeit geprüft werden. Außerdem ist die Konfiguration der Komponenten zu beschreiben. Die entworfenen Komponenten des Anwendungskerns sind relativ klein und in ihrer Funktionalität überschau-

bar, was daran liegt, dass es sich bei Vendorbase um ein eher akademisches Beispiel handelt. Aus diesem Grund können die genannten Schritte recht kurz gefasst werden.

Für die Validierung der Schnittstellen wurde vom Autor zunächst kontrolliert, ob alle vorgesehenen Anwendungsfälle durch die Schnittstellen repräsentiert werden. Dabei sind die Traceability-Links von Nutzen, die nach den Konsistenzbedingungen von jedem konkreten Anwendungsfall zu einer Schnittstelle existieren müssen. Diese ergeben sich allerdings durch den eingefügten Zwischenschritt, der Erstellung des Funktionsbaums, zur Identifikation der Komponenten aus den Anforderungen nur indirekt über die einzelnen Funktionalitäten. Es gibt demnach Links von den Anwendungsfällen zu den Funktionalitäten und weiter zu den Schnittstellen. Die Vollzähligkeit dieser Links muss nachgeprüft werden. Weiterhin müssen die Schnittstellenexporte und -importe der Komponenten gegen die im Kategorienmodell festgelegten Abhängigkeiten geprüft werden. Dazu wurden die Schnittstellenspezifikationen auf unnötige `import`-Anweisungen hin untersucht. Auch die Zuordnung der Schnittstellen auf die Java-Pakete wurde einer Begutachtung unterzogen, sodass das in Kapitel 2.2.4 erwähnte Kriterium zur Aufteilung von Schnittstellen und Klassen auf Pakete eingehalten wird.

Die Implementierung der Schnittstellen stellt keine besondere Herausforderung dar, da sie frei von technischen Aspekten sind. Für die Komponente *Authorization* basieren die definierten Schnittstellen auf den von Quasar empfohlenen Standardschnittstellen bzw. werden durch Komponenten von SalesPoint bereitgestellt. Die angebotenen Schnittstellen von *ContractManager*, *VendorManager* sowie *ProductManager* umfassen lediglich Funktionalität zur Arbeit mit Entitäten und können dazu auf dem SalesPoint-Framework mit der Schnittstelle *Catalog* aufbauen. Die Schnittstelle *IShop* soll die Anmeldung realisieren, was mithilfe der Autorisierungsschnittstellen umgesetzt werden kann. Die einzelnen dienstorientierten Schnittstellen, welche der Anwendungskern bereitstellt, können durch O2S-Adapter bewerkstelligt werden. Somit steht die Implementierbarkeit der gesamten Schnittstellen nicht in Frage, und es wurde auf die von Quasar empfohlenen Dummy-Implementierungen verzichtet.

Die Konfiguration der Anwendungskernkomponenten gestaltet sich recht einfach. Die Komponente *Shop* dient als Konfigurationsmanager für *ContractManager*, *VendorManager*, *ProductManager* sowie *Authorization*. Er versorgt die Komponenten mit Implementierungen ihrer benötigten Schnittstellen aus dem SalesPoint-Framework. So wird *Authorization* beispielsweise mit den Komponenten *UserManager* und *Permissions*, die aus der bisherigen

Überarbeitung des Frameworks stammen, konfiguriert. Für eine Implementierung der Schnittstelle *Catalog* wird auf das originale Framework zurückgegriffen. Die Instanziierung der Komponentenobjekte für Vendorbase erfolgt einfach per direkten Konstruktoraufruf. Eine Alternative dazu wären z. B. Fabriken. Diese würden mehr Flexibilität bieten, jedoch auch den Aufwand erhöhen. Der Anwendungskern selbst, also die *Shop*-Komponente, wird als Singleton realisiert und benötigt keine weitere Konfiguration, da sie keine angeforderten Schnittstellen besitzt.

Mit Abschluss der Aktivität zur Beschreibung der Konfiguration ist die Außensicht, der wichtigste Teil der Komponentenbeschreibung, vollständig. Daran schließen sich die Darstellung der Innensicht der Komponenten sowie eine Variabilitätsanalyse an.

Innensicht der Komponenten

Die Innensicht beschäftigt sich mit dem inneren Aufbau, der Komposition, der Komponenten. Daher soll zunächst die Zusammensetzung der Komponente *Shop* für Vendorbase skizziert werden. Im Anschluss wird noch eine Ebene tiefer beispielhaft auf den Aufbau der Komponente *ContractManager* eingegangen.

Die Komponente *Shop* ist nach Quasar im Wesentlichen der Kompositionsmanager für *ContractManager*, *VendorManager*, *ProductManager* sowie *Authorization*. Damit setzt sich *Shop* im Inneren aus diesen Komponenten zusammen. Sie versorgt die Importschnittstellen der komponierten Komponenten und bietet eine beliebige Teilmenge der Exportschnittstellen nach außen hin an.

Abbildung 4-13 zeigt die Komposition der Komponente *Shop*. Die für die GUI exportierten dienstorientierten Schnittstellen *IAuthorizationCheckService*, *IUserAdminService*, *IContractManagerService*, *IVendorManagerService* sowie *IProductManagerService* werden mittels O2S-Adaptern aus den objektorientierten Schnittstellen der Subkomponenten gewonnen. Das Callback-Interface *IOperationDefinition* und die Operationen für die *Authorization*-Subkomponente werden von der *Shop*-Implementierung selbst bereitgestellt. Für die restlichen angeforderten Schnittstellen der Subkomponenten, welche nicht intern bereit stehen,

werden SalesPoint-Implementierungen genutzt. Außerdem bietet die Komponente selbst noch die Realisierung für das Interface *IShop,* welches die Funktionalität Anmeldung repräsentiert.

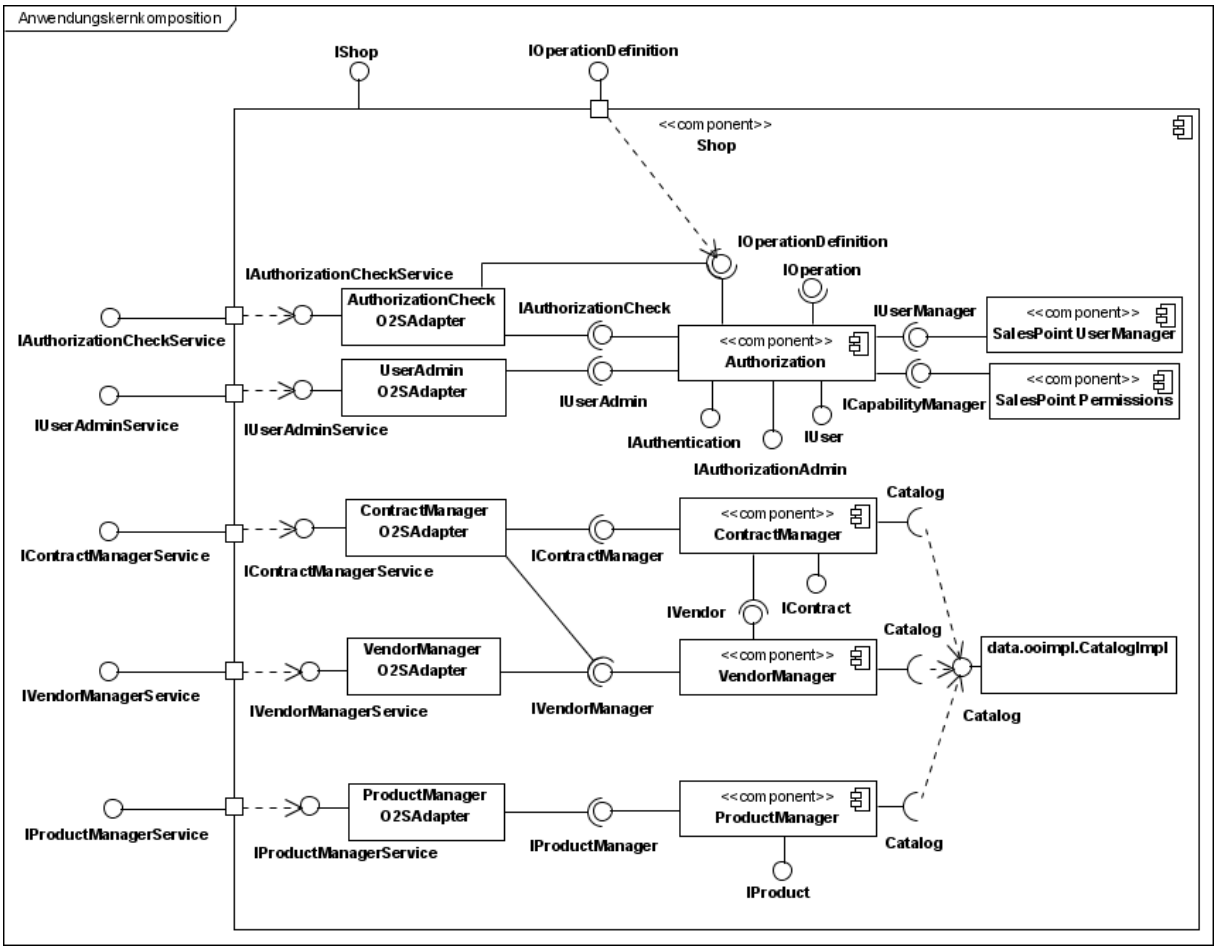

Abbildung 4-13: Kompositionsstrukturdiagramm für die Komponete Shop

Die Entitätsschnittstellen der Subkomponenten werden nicht für die Benutzerschnittstelle, welche den Anwendungskern nutzt, nach außen delegiert. Entitätsschnittstellen sind nur in objektorientierten Programmschnittstellen erlaubt. An ihre Stelle treten Transportklassen für die einzelnen Entitäten in den exportierten dienstorientierten Schnittstellen. Diese Transportklassen sind *UserRecord, ContractRecord, VendorRecord* sowie *ProductRecord*. Sie werden von den O2S-Adaptern bei der Transformation der Schnittstellen genutzt. Die einzelnen Adapter sind hier außerhalb der Subkomponenten von *Shop* dargestellt, um die Transformation zu verdeutlichen. Sie werden aber in der Paketstruktur genauso wie die Transportklassen den einzelnen Subkomponenten des Anwendungskerns zugeordnet.

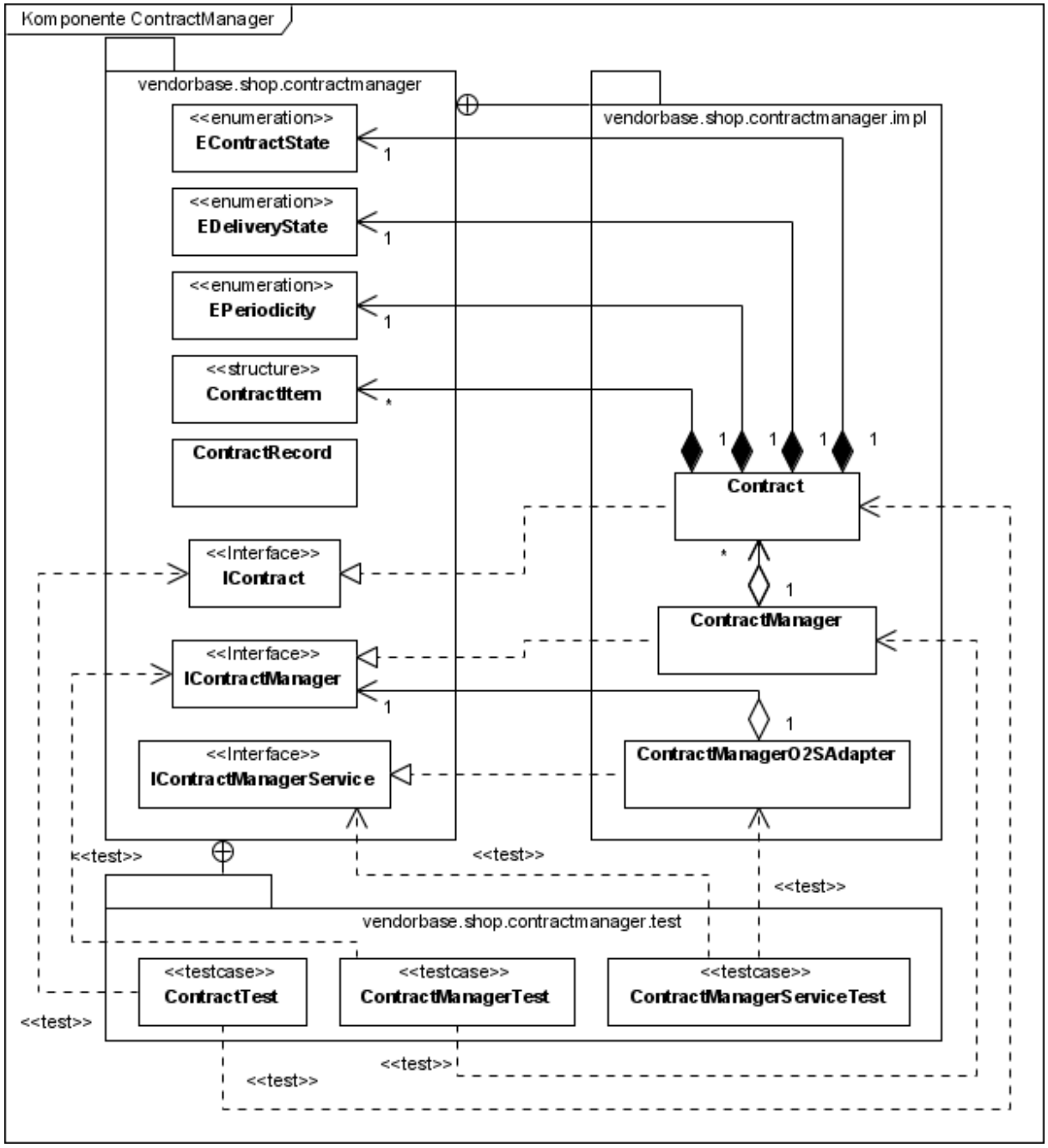

Abbildung 4-14: Aufbau der Komponente ContractManager

Zur Verdeutlichung des Aufbaus der Subkomponenten wird in Abbildung 4-14 beispielhaft die Komponente *ContractManager* skizziert. Es ist eine Aufteilung der Klassen und Schnittstellen auf mehrere Java-Pakete erkennbar. Das Paket *vendorbase.shop.contractmanager* enthält nur die nach außen hin sichtbaren Schnittstellen und Datentypen der Komponente sowie die Transportklasse *ContractRecord*. Die implementierenden Klassen sowie die Testfälle befinden sich jeweils eine Ebene tiefer. Abhängigkeiten nach außen zu importierten Schnittstellen sind nicht dargestellt. Für weitere Darstellungen zu den anderen Anwendungskernkomponenten *Authorization, VendorManager* und *ProductManager*, die dieser ähneln, sei auf die Modelldiagramme im Anhang verwiesen.

Eine Anmerkung zur Implementierung der Komponenten muss noch gemacht werden. Nach Quasar ist Vererbung über Komponentengrenzen hinweg verboten [Sie04, S. 55], da sonst eine sehr enge Beziehung geschaffen wird. Das hieße für Vendorbase, dass keine Klasse des SalesPoint-Frameworks spezialisiert werden dürfte. Die Verwendung von Frameworks im Allgemeinen und SalesPoint im Besonderen sind jedoch darauf ausgerichtet, dass eigene Klassen von den Framework-Klassen erben, um die Funktionalität der Frameworks zu nutzen. In der Vendorbase Implementierung sind die Klassen *Contract*, *Vendor*, *Product* z. B. Spezialisierungen von *CatalogItem* aus SalesPoint, da die Unterstützung durch SalesPoint nur so möglich ist. In diesem Punkt musste demnach eine Ausnahme gemacht und entgegen der Quasar-Regel eine Vererbungsbeziehung über die Grenze der Komponenten von Vendorbase hin zu SalesPoint erstellt werden. Der Autor bewertet dies jedoch als unkritisch, weil Framework-Klassen in der Regel so konzipiert sein sollten, dass sie eine Vielzahl an Szenarien unterstützen, ohne geändert werden zu müssen.

Variabilitätsanalyse

Bei der Variabilitätsanalyse sind für die Komponenten Änderungsszenarien aufzustellen und diese Szenarien nach dem Änderungsaufwand zu klassifizieren. Quasar empfiehlt dafür das Vorgehen nach der *Software Architecture Analysis Method* (SAAM, siehe [CKK02]). Auf eine Anwendung dieser Methode hat der Autor im Rahmen dieser Arbeit verzichtet. Gründe dafür sind, dass SAAM umfangreich und aufwendig ist und von mehreren Personen durchgeführt werden sollte. Außerdem können einzelne Änderungsszenarien auch ohne SAAM betrachtet werden.

Ein technischer Aspekt, der bezüglich Änderungen gern untersucht wird, ist die Art und Weise der persistenten Speicherung. So stellt sich die Frage nach dem Änderungsaufwand von einer Speicherung der Daten im Dateisystem hin zu einer Speicherung in einer relationalen Datenbank. Für die überarbeitete Version von Vendorbase wurde, wie eingangs des Kapitels erläutert, die persistente Speicherung nicht realisiert. Daher können bezüglich dieses Szenarios in der Architektur die Wege noch frei eingeschlagen werden. Allerdings muss über eine Anpassung der technischen Softwarekategorien im Kategorienmodell nachgedacht

werden, wenn z. B. ein Framework für objektrelationales Mapping wie Hibernate[13] eingesetzt werden sollte. Außerdem muss zusätzliche R-Software eingeführt werden, welche die Entitätsdaten aus den Kategorien der Domainfunktionalitäten auf die technischen Schnittstellen transformiert.

Eine weitere zu betrachtende Änderung bezieht sich auf die Unterstützung mehrerer parallel arbeitender Nutzer im Anwendungskern. Hier wäre in den Komponenten der Kategorie *VB-Verwaltung* zusätzlicher Aufwand nötig, um eine Transaktionsverwaltung einzuführen. Auf eine Unterstützung dessen seitens des SalesPoint-Frameworks wurde bisher aus schon genannten Gründen verzichtet. Jedoch ist auch dieses Szenario insofern mit der Architektur vereinbar, dass lediglich die Aufrufe der Methoden der Subkomponenten des Anwendungskerns mit einer Transaktionsverwaltung verpackt werden müssen.

Auf weitere Änderungsszenarien soll an dieser Stelle nicht eingegangen werden. Dennoch bleibt festzuhalten, dass identifizierte und nach Aufwand untersuchte Szenarien mittels Traceability-Links den Softwarekategorien zugeordnet werden können. Zudem können sich später durchgeführte Änderungen an der Software auf die bereits mit ihren Auswirkungen beschriebenen Szenarien beziehen.

Mit Abschluss der Variabilitätsanalyse vervollständigt sich die Komponentenbeschreibung und damit endet der Prozess der Erstellung des Komponentenmodells nach Quasar zunächst für den Anwendungskern. Als Ergebnisse liegen die Beschreibung von Komponenten in ihrer Außen- und Innensicht sowie zusätzliche Änderungsszenarien vor. Diese fließen nachfolgend schließlich in die Phase der Implementierung ein. Darüber hinaus wurden Traceability-Links zwischen den Elementen der verschiedenen Modelle, wie z. B. zwischen Softwarekategorien und Komponenten oder Schnittstellen, erstellt. Die Traceability-Links für Vendorbase sind in tabellarischer Form im Anhang verfügbar. In weiteren Aktivitäten sind neben der Implementierung noch die Komponenten der grafischen Benutzerschnittstelle zu modellieren.

[13] Hibernate ist ein objektrelationales Mapping-Framework für Java und zu finden unter der Web-Adresse http://www.hibernate.org/

4.4.2 Architektur der Benutzerschnittstelle für Vendorbase

In diesem Abschnitt soll die Architektur der Benutzerschnittstelle mit ihren Komponenten und Programmschnittstellen[14] für die Beispielanwendung Vendorbase und das Vorgehen bei der Erstellung erläutert werden. Dabei kommt die in Kapitel 2.2.5 vorgestellte Client-Architektur von Quasar zur Anwendung. Das Vorgehen zum Entwurf der Dialogkomponenten wird mithilfe der in Kapitel 2.3 vorgestellten Ansätze zum Dialogdesign vervollständigt. Doch zunächst muss zum Verständnis des Vorgehens noch ein Aspekt der Quasar-Methode erklärt werden, der bisher nicht betrachtet wurde – die Untergliederung der Systemarchitektur [Sie04, S. 145ff.].

Systemarchitektur nach Quasar

Die Architektur eines Systems wird nach Quasar untergliedert in die technische Infrastruktur (kurz TI-Architektur) und die Softwarearchitektur. Zur TI-Architektur gehören die physischen Geräte, Systemsoftware und ebenso die verwendeten Programmiersprachen. Die Softwarearchitektur beschreibt das Zusammenwirken von Komponenten und Schnittstellen. Sie wird aus zwei Sichten betrachtet, der Anwendungssicht und der technischen Sicht. Demnach wird zwischen Anwendungsarchitektur (A-Architektur) und Technikarchitektur (T-Architektur) unterschieden. Tabelle 4-2 verdeutlicht die Untergliederung. Die A-Architektur beschreibt fachliche Abläufe, die Anwendungskomponenten. Sie ist frei von Technik. Die T-Architektur stellt die Verbindung zwischen der TI-Architektur und der A-Architektur her. Sie legt fest, wie die A-Komponenten auf der technischen Infrastruktur ablaufen.

[14] In diesem Kapitel wird für den Begriff Programmschnittstelle häufig der Einfachheit halber auch Schnittstelle verwendet. Die Abgrenzung in der Beschreibung zu Benutzerschnittstellen erfolgt dadurch, dass letztere immer explizit als Benutzerschnittstellen bezeichnet werden.

Systemarchitektur		
Softwarearchitektur		**Technische Infrastruktur (TI-Architektur)**
Anwendungsarchitektur (A-Architektur)	Technikarchitektur (T-Architektur)	
- Anwendungskomponenten - fachliche Abläufe - Entitäten (Vertrag, Lieferant) - keine Abhängigkeiten von Technik wie API (z. B. JDBC)	- von Anwendung unabhängig - Verbindung zwischen A und TI - Ablaufumgebung für A-Komponenten (virtuelle Maschine) - z. B. GUI-Rahmen, Zugriffsschicht - Einheiten (jar, dll) und Verteilung der Software auf Prozesse	- physische Geräte (Rechner, Drucker) - Systemsoftware (Betriebssystem, Anwendungsserver) - Programmiersprache (Java, C++)
- Standard-A-Architektur (nicht bei Individualsoftware)	- Standard-T-Architekturen für spezielle TI-Architekturen	- Standard-TI-Architekturen (EJB, .Net)

Tabelle 4-2: Systemarchitektur nach Quasar

Darüber hinaus können Standardarchitekturen definiert werden. Die Client-Architektur nach Quasar (Kapitel 2.2.5) ist eine Standardarchitektur, die unabhängig von Anwendung und Technik wiederverwendet werden kann und somit keiner Kategorie angehört. Standard-TI-Architekturen beschreiben immer wiederkehrende technische Infrastrukturen wie EJB-Server oder .Net. Standard-T-Architekturen sind Spezialisierungen von Standardarchitekturen für bestimmte Standard-TI-Architekturen. Standard-A-Architekturen können bestimmte wiederkehrende Anwendungen oder Anwendungsteile wie eine Kundenverwaltung beschreiben.

Die drei Architekturen A, T und TI werden nach Quasar im Entwicklungsprozess parallel erstellt, sodass eine arbeitsteilige Entwicklung sogar unter Berücksichtigung der Trennung von Anwendung und Technik erfolgen kann [Sie04, S. 158f.].

Für die Beschreibung des Anwendungskernes im letzten Abschnitt wurde die Standardarchitektur des Anwendungskernes angewendet. Eine gesonderte Betrachtung der Architekturen A, T und TI war nicht erforderlich, da der Anwendungskern von Vendorbase, wie er beschrieben wurde, nur auf A-Kategorien beruht. Technische Aspekte wie eine persistente Speicherung wurden nicht betrachtet, sodass eine Umsetzung auf eine konkrete API wie JDBC nicht berücksichtigt werden musste. Aus Sicht der Infrastruktur war weiterhin nur die Verwendung von Java relevant, was keine besonders bemerkenswerten Auswirkungen auf die Gestaltung der Anwendungsarchitektur mit ihren Komponenten und Schnittstellen hat. Für die weiteren Betrachtungen zur Benutzerschnittstelle von Vendorbase ist jedoch die Unterscheidung der verschiedenen Begriffe Standardarchitektur, A-Architektur, T-Architektur und TI-Architektur von Bedeutung, da nun zwingend eine konkrete technische Umsetzung für grafische Benutzersteuerelemente benötigt wird. Darüber hinaus liegt der Schwerpunkt für den Autor auf der Beschreibung des Entwurfs der Benutzerschnittstelle. In den nächsten Abschnitten werden der Reihe nach die einzelnen, eigentlich parallel erarbeiteten Architekturen für die Benutzerschnittstelle von Vendorbase beschrieben.

TI-Architektur

Zunächst wird mit der Beschreibung der TI-Architektur begonnen, da diese zum großen Teil schon vorher feststand. Bei der Beschreibung der technischen Infrastruktur müssen nach und nach die unterschiedlichen dazugehörigen Elemente (siehe Tabelle 4-2) in ihrer konkreten Ausprägung für das System beschrieben werden. Die grundsätzlichen Eckpunkte der TI-Architektur sind für Vendorbase durch das Pflichtenheft bereits vorgegeben. Als Programmiersprache ist Java vorgegeben. Dies resultiert aus der Tatsache, dass das zu verwendende SalesPoint-Framework in Java implementiert ist. Daher ist die Java-Laufzeitumgebung als Systemsoftware eine Grundvoraussetzung, um das Programm später ablaufen zu lassen. Der Autor bevorzugt bei der Entwicklung die neueste Version 1.6 des Java-Development-Kits, da diese in einigen Details gegenüber der Version 1.5 Vorteile bietet. Diese Version 1.6 wird damit für die spätere Verwendung des Programms vorausgesetzt. Eine konkrete Komponententechnologie wie EJB[15] wird nicht verwendet. Weiterhin wird vom Pflichtenheft mindestens das Betriebssystem Windows XP mit einer Bildschirmauflösung von 1024x768 Pixel vorgesehen. Die Auflösung ist für das Design der Benutzerschnittstelle von zentraler Bedeutung, da

[15] Enterprise Java Beans

sie den für das Dialogdesign zur Verfügung stehenden Platz auf dem Bildschirm vorgibt. Die Beispielanwendung Vendorbase wird als Rich-Client erstellt, wobei die Swing-API zur Anwendung kommt. Swing wird vom Autor aus mehreren Gründen z. B. im Vergleich zu SWT[16] bevorzugt. Zum einen basiert die GUI-Unterstützung durch SalesPoint auf Swing. Ein Austausch der GUI-Bibliothek würde die Hürde zum Verständnis beim Wechsel von der alten SalesPoint-Version zum neu entwickelten Benutzerschnittstellenkonzept nur unnötig vergrößern. Zum anderen möchte der Autor zur Umsetzung der softwareergonomischen Richtlinien auf weitere Java-Bibliotheken zurückgreifen, die auf Swing beruhen. Dazu zählen SwingX[17] und JGoodies[18]. Bei SwingX handelt es sich um eine Bibliothek von verbesserten und neuen Benutzersteuerelementen für Swing, die direkt von Sun entwickelt wird. Zu JGoodies gehören mehrere Bibliotheken, von denen *Forms* und *Looks* zum Einsatz kommen. Mit *Forms* und *Looks* wird der Aufbau von Dialogmasken mit guter Usability vereinfacht, indem Steuerelemente besser ausgerichtet und das Look&Feel besser an das vom Betriebssystem gewohnte Aussehen angepasst werden können. Für die Entwicklung von Vendorbase werden die Versionen SwingX 0.9.1 sowie Forms 1.1.0 und Looks 2.1.4 verwendet. Diese werden damit ebenfalls Voraussetzung für den Betrieb der Anwendung. Jedoch ist anzumerken, dass die Verwendung von Swing und der darauf aufbauenden Bibliotheken nicht in Stein gemeißelt ist. Durch die Anwendung der Quasar-Standardarchitektur für Clients, wie in den nächsten Abschnitten beschrieben, wird später durchaus ein Wechsel beispielsweise hin zu einer Web-Oberfläche möglich. Die TI-Architektur-Beschreibung soll damit für die Beispielanwendung abgeschlossen sein und die T-Architektur kann darauf aufbauen.

T-Architektur

Bei der Beschreibung der T-Architektur wird unter Berücksichtigung der technischen Infrastruktur der technologische Rahmen definiert. In unserem Fall muss die T-Architektur die technischen Grundlagen für den Ablauf der grafischen Benutzeroberfläche festlegen. Dazu wird die Standardarchitektur für eine GUI verwendet, wie sie von Quasar definiert wird und in Kapitel 2.2.5 kurz vorgestellt wurde. Diese Standardarchitektur muss für Vendorbase mit ihren abstrakten Konzepten, wie der Trennung der GUI in Dialograhmen und Dialoge, auf die

[16] Standard Wigdet Toolkit
[17] Die Webseite von SwingX ist verfügbar unter http://swinglabs.org/
[18] Die Webseite von JGoodies ist verfügbar unter http://www.jgoodies.com/

konkrete Technologie, also Java und im besonderen Swing, abgebildet werden. Die GUI-Standardarchitektur, die wohlgemerkt weder der Kategorie A noch der Kategorie T angehört, wird technisch gefärbt und damit zu einer Standard-T-Architektur.

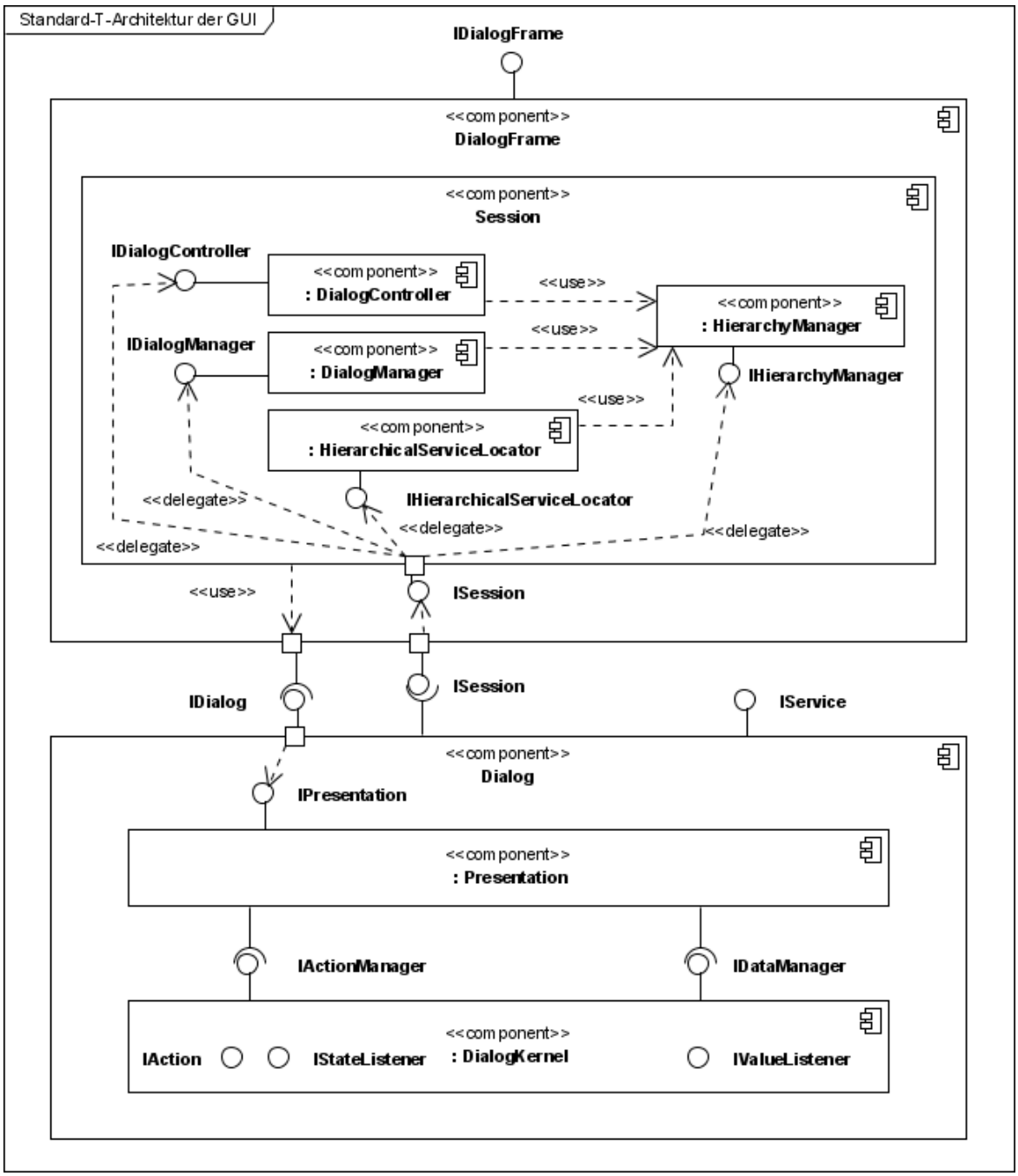

Abbildung 4-15: Standard-T-Architektur für Benutzerschnittstellen

Der Autor hat im Rahmen der vorliegenden Arbeit die in den verschiedenen Dokumentationen zu Quasar ([Sie03b, S. 28ff.], [Sie04, S. 235ff.], [HHS04, S. 15ff.], [HHS05, S. 114ff.] und [HO07]) beschriebene Standardarchitektur für Benutzerschnittstellen in Java umgesetzt. Dabei wurden die Konzepte und teilweise exemplarisch vorhandenen Programmschnittstellen für Dialograhmen, Dialog, Dialogkern und Präsentation übernommen, verändert und vervoll-

ständigt, damit sie einen geschlossenen GUI-Rahmen bilden. Abbildung 4-15 zeigt die technische Architektur der Benutzerschnittstelle mit Dialograhmen und Dialog und ihren Programmschnittstellen. Die Schnittstellen wurden vollständig mit QSL spezifiziert wie auch in Kapitel 4.4.1 für den Anwendungskern. Sie gehören fast ausschließlich der Kategorie 0 an. Eine Ausnahme davon ist die Methode `java.awt.Component getVisualRepresentation()` in den Schnittstellen *IDialog* und *IPresentation* sowie die Methoden zur Einbettung von Masken aus Unterdialogen in *IPresentation*. Listing 4-3 zeigt beispielhaft die QSL-Spezifikation für *IPresentation*. Hier wird Bezug auf die konkrete visuelle Darstellung, auch Maske genannt, genommen. An dieser Stelle zeigt sich unumgänglich die Verwendung einer speziellen GUI-Bibliothek. Da im Fall Vendorbase Swing zur Anwendung kommt, wird als Parameter in diesen Methoden `java.awt.Component` als gemeinsamer Vorfahre für alle grafischen Elemente der GUI-Bibliothek verwendet. Mit Ausnahme dieser Methoden verwenden alle Schnittstellen des GUI-Rahmens lediglich Standarddatentypen oder 0-Schnittstellen.

```
 1 package quasar.gui.dialog.presentation;
 2
 3 import java.awt.Component;
 4
 5 import quasar.gui.dialog.dialogkernel.IActionManager;
 6 import quasar.gui.dialog.dialogkernel.IDataManager;
 7
 8 /**
 9  * [export, operation] The <code>IPresentation</code> interface
10  * defines the services, call-backs and a life cycle of a presentation.
11  * <pre>
12  * [uses
13  *             IActionManager,
14  *             IDataManager,
15  *             Component   // as visualRepresentation
16  * ]</pre>
17  *
18  * @author Stephan Bode
19  * @version 1.0 05.10.2007
20  */
21 public interface IPresentation {
22
23     // basicQueries
24
25     /**
26      * [basicQuery] The visual representation is returned.
27      *
28      * @return  the visual representation of the dialog if it has been
29      *          initialized, null otherwise
30      */
31     Component getVisualRepresentation();
```

```
32
33      // commands
34
35      /**
36       * [command] The presentation is initialized with its resources and can be
37       * used by the dialog. Particularly, its visual representation is provided.<br>
38       * initialize() must be invoked before the first use of the visual
39       * representation
40       *
41       * @param dataManager
42       *            IDataManager to exchange (read and write) dialog data with
43       * @param actionManager
44       *            IActionManager that manages the dialog's actions
45       *
46       * <pre>
47       * [post getComponent() != null
48       *    // a visual representation is provided
49       * ]
50       * </pre>
51       */
52      void initialize(IDataManager dataManager, IActionManager actionManager);
53
54      /**
55       * [command] The presentation has been initialized before and now is called
56       * to embed another visual representation as its sub-presentation if such a
57       * thing exists and suits to be embedded.
58       *
59       * @param visualRepresentation  the sub-presentation to integrate
60       *
61       * <pre>
62       * [pre getComponent() != null
63       *   // a visual representation is provided
64       * ]
65       * </pre>
66       */
67      void embedSubPresentation(Component visualRepresentation);
68
69      /**
70       * [command] Removes a previously embedded sub-presentation.
71       *
72       * @param visualRepresentation  the sub-presentation to remove
73       *
74       * <pre>
75       * [pre  exists Component visualRepresentation, Time t : t < now :
76       *       invoke(this.embedSubPresentation(visualRepresentation)
77       *    // the visual representation to be removed must have been embedded before
78       * ]
79       * </pre>
80       */
81      void removeEmbeddedSubPresentation(Component visualRepresentation);
82
83      /**
84       * [command] The presentation including its visual representation is
85       * destroyed and cannot be used within the dialog any more. <br>
86       * destroy() must be invoked on releasing a dialog.
87       */
88      void destroy();
89
90      /**
91       * [command] The presentation is called to update itself (observer pattern) e.g.,
92       * due to changed data in the dialog kernel
93       */
94      void update();
95
96  }
```

Listing 4-3: Spezifikation der Schnittstelle IPresentation

Mit der vom Autor erstellten Standard-T-Architektur können die A-Dialogkomponenten für Vendorbase mit minimalen Abhängigkeiten zur Technik gegen die spezifizierten einfachen Programmschnittstellen der Kategorie 0 implementiert werden. Darüber hinaus ist die Architektur prinzipiell für ähnliche Projekte wiederverwendbar, die mit Java implementiert werden und eine Rich-Client-Oberfläche benötigen. Neben der Spezifikation der Schnittstellen wurden weiterhin abstrakte Klassen für den Dialograhmen mit Sitzungs- und Dialogverwaltung sowie für den Dialog mit Dialogkern und Präsentation erstellt, welche die Schnittstellen soweit wie möglich anwendungsneutral implementieren. Sie dienen als Ausgangspunkt für die in der A-Architektur zu erstellenden Dialogkomponenten, die für die konkreten anwendungsspezifischen Dialoge der Benutzerschnittstelle verantwortlich sind. Darüber hinaus wurden für einzelne in Vendorbase verwendete Benutzersteuerelemente wie Schaltfläche, Textfeld und Tabelle Adapter erstellt, welche die Daten- und Aktionsanbindung der Swing-Komponenten an den Dialogkern vornehmen. Diese Adapter sind frei von fachlichen Aufgaben und können somit in anderen auf Swing basierenden Anwendungen wiederverwendet werden.

Soll die entwickelte Standard-T-Architektur für eine andere GUI-Bibliothek als Swing benutzt werden, sind geringfügige Änderungen nötig. Die Bibliothek ist jedoch prinzipiell austauschbar. Als gemeinsamer Vorfahre für alle Masken muss `java.awt.Component` in den entsprechenden Interfaces durch eine andere Schnittstelle der neuen GUI-Bibliothek ersetzt werden. Außerdem ist selbstverständlich die Entwicklung neuer Adapter für die Daten- und Aktionsanbindung nötig, da die bestehenden nur für Swing-Komponenten gedacht sind. Diese Adapter gehören zur Kategorie R. Sie müssen deshalb ersetzt werden, wenn eine andere Transformation als zu Swing benötigt wird.

Die QSL-Spezifikation für sämtliche Programmschnittstellen der T-Architektur der GUI, sowie der Java-Quellcode für die abstrakten Implementierungen und die Swing-spezifischen Adapter zur Daten- und Aktionsanbindung sind vollständig im Anhang dieser Arbeit zu finden. Sie wurden auf die Pakete `quasar.gui.dialogframe` sowie `quasar.gui.dialog` und deren Unterpakete aufgeteilt. Eine kompilierte Version ist ebenfalls beigefügt. Dazu wurden als Auslieferungseinheiten die zwei Archive `dialog.jar` sowie `dialogframe.jar` generiert. Eine ähnliche Untergliederung wird ebenfalls für die A-Komponenten von Vendorbase vorgesehen. So gibt es für den Dialograhmen von Vendorbase ein jar-Archiv und gleichfalls für jeden Hauptdialog. Auf eine Unterteilung in selbstständige Archive für einzelne Unterdialoge wird zugunsten der Übersichtlichkeit verzichtet.

Zum Vorgehen bei der Erstellung der T-Architektur eines Anwendungssystems soll noch Folgendes angemerkt werden. Die Beschreibung von T-Komponenten erfolgt gleichermaßen wie bei den A-Komponenten aufgeteilt in die verschiedenen Sichten [Sie04, S. 157]. Die Schilderung der Standard-T-Architektur wurde nicht nach Sichten aufgegliedert, da es sich bei den eigentlichen Dialogkomponenten um A-Komponenten handelt und die erstellten Schnittstellen im Vordergrund stehen. Für eine detailliertere Beschreibung der einzelnen Bestandteile der GUI-Standardarchitektur von Quasar, wie Dialograhmen, Sitzung, Dialog und Dialogkern sowie deren inneren Aufbau, sei auf [HO07] verwiesen. Die Darstellung der anwendungsspezifischen Dialogkomponenten für Vendorbase ist im nächsten Abschnitt zur A-Architektur der Benutzerschnittstelle zu finden.

A-Architektur der Benutzerschnittstelle

Die A-Architektur beschreibt die anwendungsspezifischen Teile des Softwaresystems. Für die Benutzerschnittstelle bedeutet dies, dass die einzelnen A-Komponenten mit ihren Schnittstellen spezifiziert werden müssen. Wie schon bei der Erstellung des Komponentenmodells des Anwendungskernes dienen auch hier die Artefakte Anwendungsfallmodell, Datenmodell, Funktionsbaum und Kategorienmodell als Input für die Aktivität zur Erstellung der A-Komponenten und ihrer Schnittstellen. Für die Benutzerschnittstelle kommen noch die Client-Standardarchitektur von Quasar sowie die darauf aufbauende T-Architektur aus dem vorigen Abschnitt hinzu. Mithilfe dieser Eingangsdaten müssen im Folgenden die Komponenten der GUI mit ihren Schnittstellen identifiziert und spezifiziert werden. Wie Kapitel 2.2.5 bereits herausstellt, bietet Quasar bei der Entwicklung der Benutzerschnittstelle außer der Standardarchitektur, die sich eher um organisatorische Aspekte der Verwaltung und der Kommunikation der Dialoge kümmert, wenig Unterstützung für die Aufteilung der Benutzeroberfläche in einzelne Dialogkomponenten oder das konkrete Design der Dialoge. Aus diesem Grund werden diesbezüglich in diesem Kapitel die vorgestellten Ansätze zum Dialogdesign sowie ergonomische Richtlinien Verwendung finden.

Bereits vor der Identifikation der einzelnen Komponenten lässt sich aufgrund der verwendeten GUI-Standardarchitektur die Aussage treffen, dass alle Dialogkomponenten zur Kategorie A gehören, denn sie präsentieren die Daten der Anwendung für den Benutzer und stellen fachliche Operationen zur Interaktion bereit. Ebenso gehört der Dialograhmen, den die Standard-

architektur vorsieht, zur Kategorie A. Denn die Verwaltung der verschiedenen anwendungsspezifischen Dialoge durch den Dialograhmen ist ebenfalls vom entwickelten System und der Art seiner Dialoge abhängig. Für die Komponenten von Vendorbase bedeutet dies von vorn herein, dass sämtliche Komponenten der Benutzerschnittstelle der Kategorie *VendorbaseGUI* im Kategorienmodell (siehe Abschnitt 4.3.2) zugeordnet werden. Da sich die fachlichen Dialogkomponenten in die beschriebene T-Architektur einfügen müssen, werden sie die dort definierten Schnittstellen verwenden, die fast ausschließlich zur Kategorie 0 gehören. Darüber hinaus werden fachliche Schnittstellen für die Dialoge spezifiziert und die Dialogkomponenten werden auf die dienstorientierten Schnittstellen des Anwendungskernes zugreifen.

Identifikation der Dialogkomponenten

Als ein Ausgangspunkt zur Identifikation der Dialogkomponenten kann der Funktionsbaum aus Kapitel 4.3.1 genommen werden. So sind die Systemfunktionalitäten der Architekturschicht Interface-Funktionalitäten bisher nicht betrachtet worden und müssen noch zu Komponenten zugeordnet werden. Folglich werden Dialogkomponenten benötigt, welche die Funktionalitäten *Lieferantenanzeige*, *Nutzeranzeige*, *Produktteilanzeige* und *Liefervertragsanzeige* sowie *Filterung & Sortierung* umsetzten. Von der Anzeige ausgehend werden die anderen Systemfunktionalitäten, wie *Liefervertrag anlegen* oder *Liefervertrag bearbeiten*, durch den Nutzer des Systems aufgerufen. Einen weiteren Hinweis zur Identifikation von Dialogkomponenten bieten noch die Konsistenzbedingungen. Danach muss jeder konkrete Anwendungsfall, der mit einem Akteur verbunden ist, einer Benutzerschnittstelle zugeordnet werden (siehe Kapitel 2.2.4). Daraus folgt für Vendorbase, dass es für den Nutzer möglich sein muss, alle Anwendungsfälle des Systems und damit alle daraus abgeleiteten Systemfunktionalitäten über die Benutzerschnittstelle aufzurufen. An diesem Punkt im Vorgehen angelangt, endet die Unterstützung durch Quasar. In weiteren Schritten werden deshalb vom Autor die im Kapitel 2.3 vorgestellten Ansätze zur Unterteilung der GUI in Dialogkomponenten sowie die Ergonomie-Richtlinien zur Gestaltung der Oberfläche zurate gezogen.

Zunächst wird die Methode des User Interface Designs nach dem Rational Unified Process (siehe Kapitel 2.3.1) angewendet und adaptiert. Dabei werden in den vorgestellten Schritten zunächst für die Anwendungsfälle Use Case Storyboards erstellt. Dies wird nachfolgend für

Vendorbase am Beispiel der Liefervertragsverwaltung demonstriert. Die logisch zusammengehörenden Anwendungsfälle zur Anzeige, zum Anlegen, Bearbeiten und Löschen von Lieferverträgen (vergleiche Abbildung 4-1 auf Seite 42) werden in einem Storyboard behandelt. Weil die Anwendungsfallbeschreibungen für Vendorbase im Pflichtenheft sehr an der GUI-Unterstützung durch SalesPoint orientiert sind, wird von diesem Aspekt abstrahiert und der Fokus auf die fachlichen Abläufe gelegt. Als Akteure für die Anwendungsfälle werden alle Nutzer von Vendorbase gleichberechtigt unabhängig ihrer Rolle und Berechtigungen betrachtet, da dies für die visuelle Dialoggestaltung keine besondere Bedeutung hat.

Nachfolgendes Flow-of-events-Storyboard wurde für die Liefervertragsverwaltung erstellt:

1. Der Nutzer wählt die Liefervertragsanzeige, in der die vorhandenen Verträge aufgelistet werden. [Verträge werden nach Vertragsnummer aufsteigend sortiert angezeigt.] {Es werden ungefähr 10 bis 15 Verträge gleichzeitig aufgelistet.} „*MainDialog*“ „*ContractDialog*“
2. Optional führt der Nutzer die Filterung & Sortierung auf der Anzeige aus. [Bezug zwischen Filter- bzw. Sortierkriterium und Ergebnisliste soll ersichtlich sein.] „*ContractDialog*“
3. Der Nutzer wählt anschließend eine oder mehrere der folgenden Aktionen aus:
    a) Auswählen eines Vertrages und Ansehen der Detaildaten. (in mehr als 90 % der Fälle) „*ContractDialog*“ „*ContractDataDialog*“
    b) Löschen eines Vertrages: „*ContractDialog*“ „*ContractDataDialog*“
        i. Vertrag mit Status „Entwurf“ auswählen und löschen.
        ii. Bestätigen oder Abbrechen des Löschvorgangs.
    c) Bearbeiten eines Vertrages: „*ContractDialog*“ „*ContractDataDialog*“ „*ContractProductChoiceDialog*“
        i. Vertrag mit Status „Entwurf“ auswählen und Vertragsdaten bearbeiten (Status, Beginn, Ende, Periodizität, Lieferdatum, Lieferstatus) und optional Lieferant sowie Produktteile mit Menge auswählen. [nur Produktauswahl zu einem Lieferanten möglich, Auflistung aller Produkte des Lieferanten] {Datenlängen: (<=256 Zeichen, 8 Stellen, 8 Stellen, <=256 Zeichen, 8 Stellen, <=256 Zeichen)}
        ii. Bestätigen oder Abbrechen der Bearbeitung.

d) Anlegen eines Vertrages: „*ContractDataDialog*“
„*ContractProductChoiceDialog*“

i. Vertrag anlegen und Daten eingeben (Status, Beginn, Ende, Periodizität, Lieferdatum, Lieferstatus) und optional Lieferant sowie Produktteile mit Menge auswählen. [nur Produktauswahl zu einem Lieferanten möglich, Auflistung aller Produkte des Lieferanten] {Datenlängen: (<=256 Zeichen, 8 Stellen, 8 Stellen, <=256 Zeichen, 8 Stellen, <=256 Zeichen)}

ii. Bestätigen oder Abbrechen der Vertragserstellung.

4. Nutzer wechselt Anzeige zu Lieferanten, Produkten oder Nutzern. „*MainDialog*“

Dies stellt die finale Version des Flow-of-events-Storyboards dar, bei dem die einzelnen Schritte bereits um Angaben angereichert wurden, die hilfreich bei der Dialoggestaltung sind. Initial werden nur die einzelnen Schritte aufgeführt. Anschließend erfolgt die Annotation mit Hinweisen für die Dialoggestaltung in [ ], Angaben zu dargestellten Datenwerten in { } oder Aussagen zur Häufigkeit der gewählten Schritte in ( ). Die Vervollständigung der textuellen Beschreibung durch Nennung der involvierten Dialogkomponenten in Anführungszeichen „“ geschieht erst nach den nächsten beiden Schritten. Dabei werden Diagramme erstellt, welche die identifizierten Dialogkomponenten darstellen.

Nach der Erstellung des Flow-of-events-Storyboards sollen im Rational Unified Process die boundary-Klassen identifiziert und ihre Interaktionen beschrieben werden. Für Quasar werden die boundary-Klassen vom Autor durch Komponenten für Dialoge, die miteinander in Beziehung stehen, ersetzt. Damit werden keine Klassen, sondern Komponentendiagramme erstellt. Abbildung 4-16 zeigt das Komponentendiagramm für das Use Case Storyboard zur Liefervertragsverwaltung. Es wurden anhand der Schritte im Flow-of-events-Storyboard die Dialogkomponenten *MainDialog*, *ContractDialog*, *ContractDataDialog* sowie *ContractProductChoiceDialog* identifiziert. Sie werden zusammen mit einem Akteur im Diagramm dargestellt. Die Aggregationsbeziehungen zwischen den Komponenten sind so zu verstehen, dass die aggregierten Komponenten Unterdialoge darstellen. Sie sind keine direkten Subkomponenten, ohne die der übergeordnete Dialog nicht existieren kann, denn dieser ist nicht direkt von den aggregierten Komponenten abhängig. Die Kommunikation läuft über die von der T-Architektur zur Verfügung gestellte Diensthierarchie (vergleiche [HO07]).

Unterdialoge werden in den jeweiligen Dialog mitsamt ihrer visuellen Darstellung eingebettet. Ein Nutzer interagiert mit allen Dialogkomponenten in einer Aggregationshierarchie.

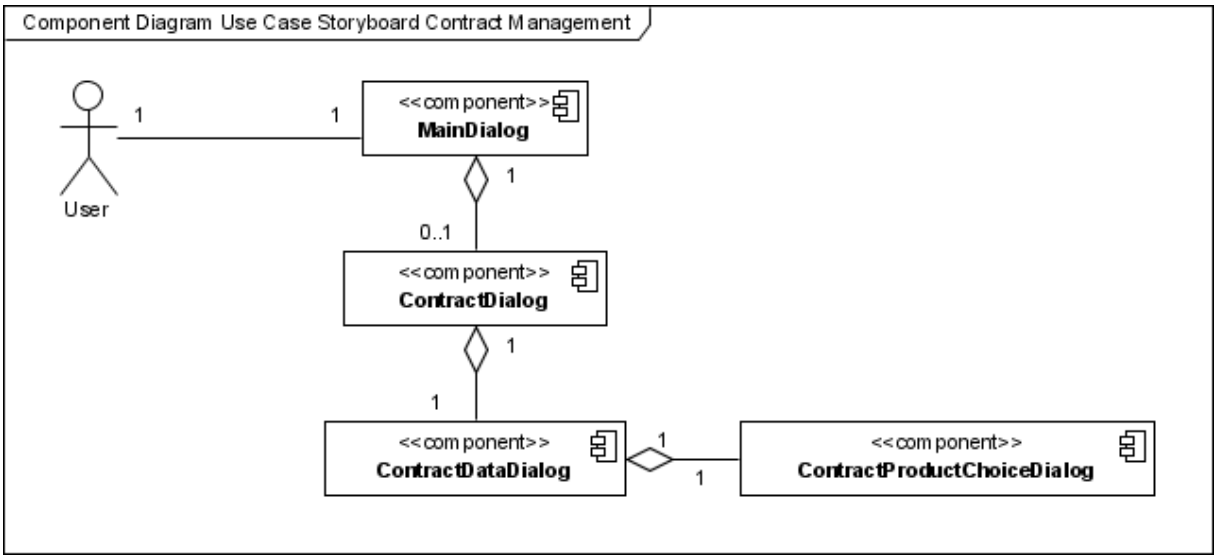

Abbildung 4-16: Komponentendiagramm zum Storyboard Liefervertragsverwaltung

Als zentrale Dialogkomponente des Systems wurde *MainDialog* identifiziert. Diese zentrale Dialogkomponente repräsentiert das Hauptfenster der Vendorbase-Anwendung, mit dem der Nutzer interagiert. Sie wird für die anderen Use Case Storyboards wiederverwendet und verwaltet die Dialogkomponenten für die Interface-Funktionalitäten wie hier den *ContractDialog* für die *Liefervertragsanzeige*. Die verschiedenen Anzeigen werden nur entsprechend der Berechtigung des interagierenden Nutzers bereitgestellt, worin die Multiplizität 1 zu 0..1 von *MainDialog* zu *ContractDialog* resultiert. So bekommt beispielsweise nur ein Administrator Zugriff auf die Nutzerverwaltung in Vendorbase.

ContractDialog listet die im System vorhanden Lieferverträge auf und verwaltet den Unterdialog *ContractDataDialog*, der für die Anzeige und Bearbeitung der Detaildaten verantwortlich ist. *ContractDataDialog* nutzt wiederum *ContractProductChoiceDialog* als Unterdialog für die Verwaltung der Auswahl des Lieferanten und dessen Produkten als Vertragsgegenstände. Auf einen Filterdialog wie in der bisherigen Vendorbase-Anwendung wurde aus ergonomischen Gründen verzichtet. Stattdessen wird die Sortierung und Filterung in der visuellen Darstellung von *ContractDialog* in direkter Nähe der tabellarischen Auflistung der Verträge vorgenommen. Ein Nutzer kann so besser den Zusammenhang zwischen Filter- bzw. Sortierkriterium und dem Ergebnis herstellen. Für eine ausführliche Diskussion zum Thema Usability von Dialogen mit Such- oder Filterfunktion wird auf [Lau05, S. 202ff.] verwiesen. In Abbildung 4-17 ist ein Kommunikationsdiagramm zu sehen, welches die Interaktionen

zwischen den Dialogkomponenten der Liefervertragsverwaltung und dem Nutzer beschreibt. Die einzelnen Schritte entsprechen dem Ablauf im Flow-of-events-Storyboard. Eckige Klammern stellen optionale Schritte dar. Das Flow-of-events-Storyboard kann nach der Identifikation der Dialogkomponenten nun in den einzelnen Schritten mit den jeweils involvierten Komponenten vervollständigt werden, wie oben gesehen.

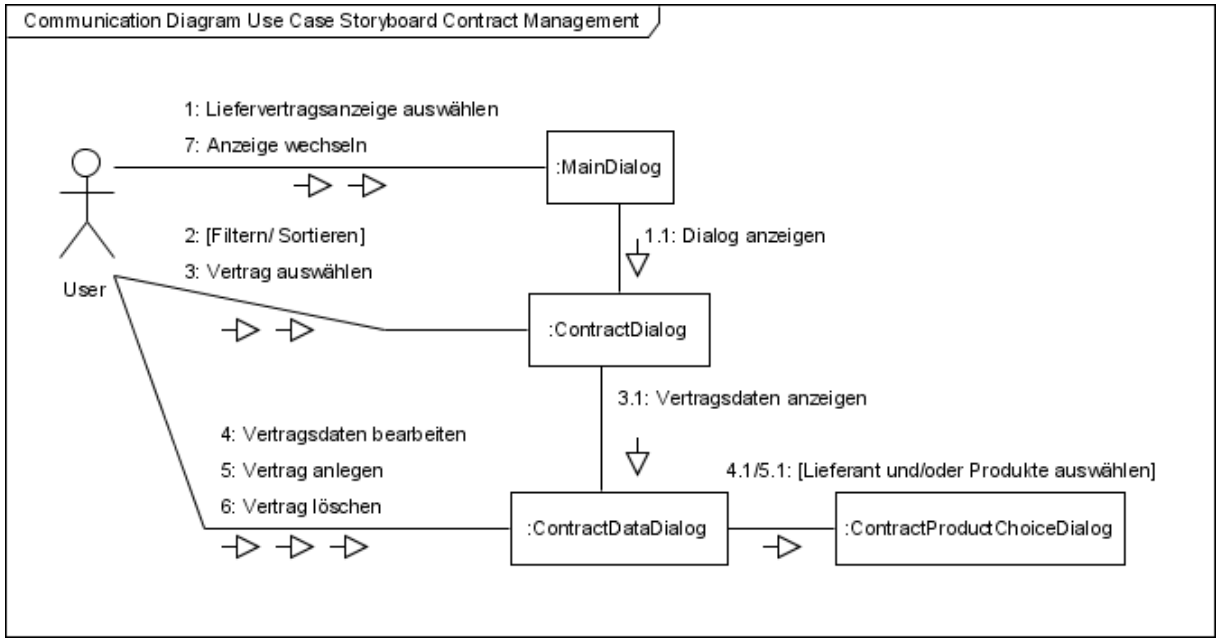

Abbildung 4-17: Kommunikationsdiagramm zum Storyboard Liefervertragsverwaltung

Damit ist die Aktivität Use Case Modeling abgeschlossen. Die Use Case Storyboards für die restlichen hier nicht dargestellten Anwendungsfälle sind im Anhang enthalten. Der Autor möchte herausstellen, dass die Identifikation von Komponenten für die Benutzerschnittstelle ein sehr kreativer Prozess ist. Die Anwendung des Use Case Storyboarding stellt jedoch eine Unterstützung bei der Festlegung der Verantwortlichkeiten für einzelne Bausteine dar und mit ihrer Hilfe wurden die benötigten Dialogkomponenten für die GUI identifiziert. Die Quasar-Standardarchitektur für grafische Benutzerschnittstellen gibt weiterhin den Rahmen vor, in dem die Dialogkomponenten ablaufen. Daher muss noch erwähnt werden, dass zu den einzelnen Komponenten der Dialoge die Bausteine für die Umsetzung des Dialograhmens hinzu kommen. Die Komponente *DialogFrame* ist demnach für die Verwaltung von nutzergebundenen Sitzungen zuständig. Ihre Subkomponente *Session*, eine für Vendorbase spezifische Umsetzung einer Sitzung, hat die Aufgabe die einzelnen Dialogkomponenten wie *ContractDialog* zu erzeugen und ihren Lebenszyklus zu verwalten.

Festlegung der Komponentenschnittstellen

Nach der Identifikation der Dialogkomponenten sind ihre Schnittstellen festzulegen. Eine allgemeine Programmschnittstelle, die alle Dialoge anbieten müssen, um vom Dialograhmen verwaltet werden zu können, ist von der Standard-T-Architektur bereits vorgegeben. Dies ist die Schnittstelle *IDialog*. Im Gegenzug benötigen die Dialoge vom Dialograhmen die Schnittstelle *ISession*. Sie stellt Dialogsteuerung, Hierarchieverwaltung und Dienstverwaltung bereit. Des Weiteren fordern die Dialogkomponenten Schnittstellen an, die der Anwendungskern, demnach bei Vendorbase die Komponente *Shop*, anbietet. Für jede Dialogkomponente ist anzugeben, welche Anwendungskernschnittstellen sie benötigt, damit sie mit diesen Schnittstellen konfiguriert werden kann. Darüber hinaus müssen die Dialoge untereinander kommunizieren. Dies geschieht über die von der Standardarchitektur festgelegte hierarchische Dienstverwaltung. Jeder Dialog stellt demzufolge seine Funktionen über Dienste zur Verfügung, damit sie von anderen Dialogen genutzt werden können. Diese Dienste müssen als Programmschnittstellen für die einzelnen Dialoge spezifiziert werden.

Abbildung 4-18 zeigt die Dialogkomponenten der Liefervertragsverwaltung mit ihren bereitgestellten und angeforderten Programmschnittstellen. Wie zu sehen ist, implementieren alle Komponenten die Schnittstelle *IDialog*. Weiterhin muss jeder Dialog mit einer Sitzung (Schnittstelle *ISession*) initialisiert werden. Ferner ist zu erkennen, welche Anwendungskernschnittstellen die einzelnen Dialoge anfordern. So benötigt der *ContractProductChoiceDialog* z. B. *IContractManagerService*, *IVendorManagerService* und *IProductManagerService*. Die nicht abgebildete Komponente *Session* des Dialograhmens fungiert für alle Dialogkomponenten als Konfigurationsmanager. Sie verbindet die Komponenten bei deren Erzeugung mit den dienstorientierten Schnittstellen, die vom Anwendungskern bereitgestellt werden. Die Schnittstellen der Dienste für die Kommunikation der Dialoge untereinander müssen nicht konfiguriert werden, denn die Dialogkomponenten fragen diese selbst bei der hierarchischen Dienstverwaltung ihrer Sitzung ab, wenn sie benötigt werden.

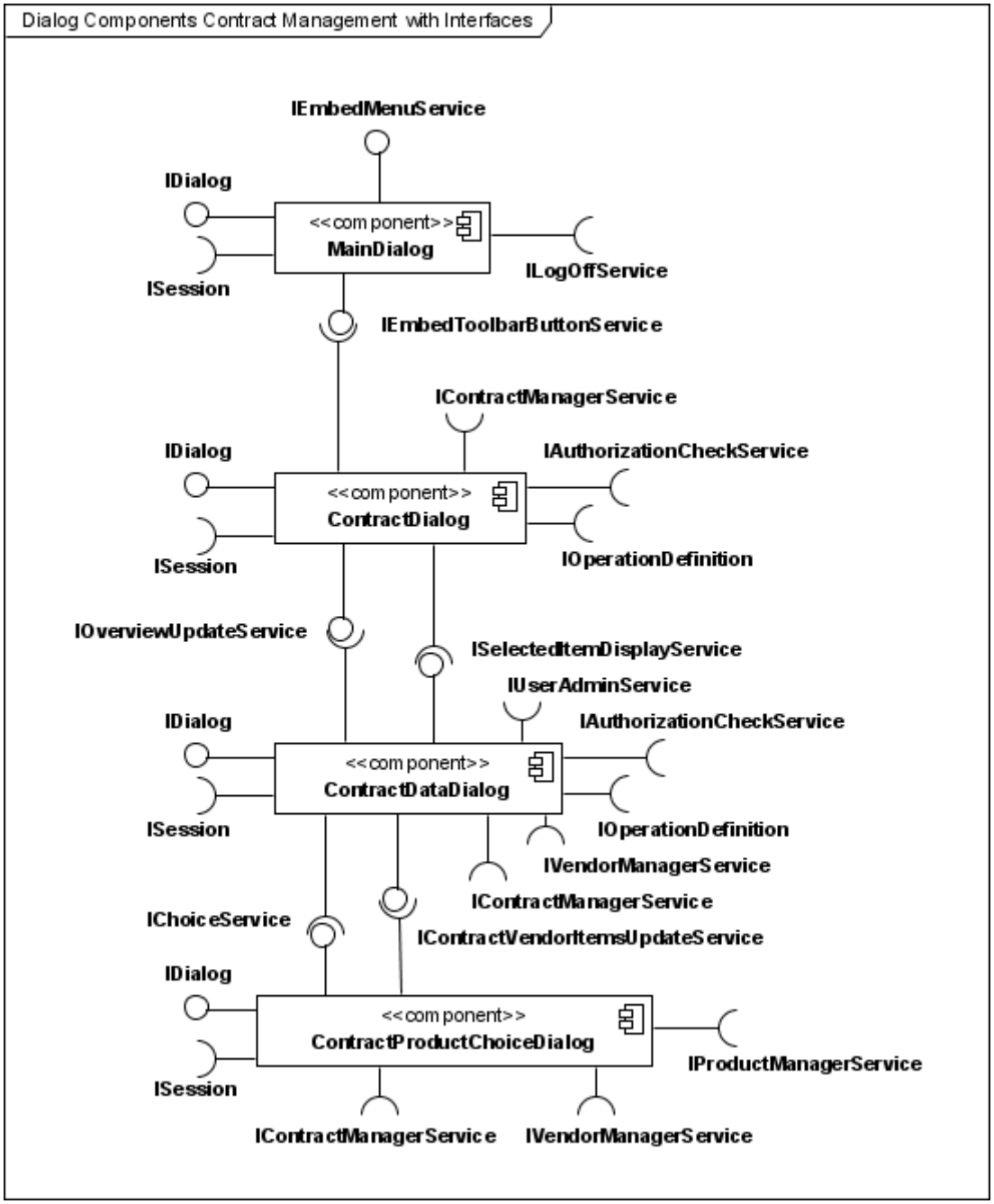

Abbildung 4-18: Dialogkomponenten und Schnittstellen der Liefervertragsverwaltung

Die Services, welche von den Dialogen untereinander genutzt werden, dienen hauptsächlich der Synchronisation der in den Dialogen angezeigten Daten, z. B. bei *IOverviewUpdateService* oder aber zum Einbetten von grafischen Benutzersteuerelementen in einen übergeordneten Dialog wie bei *IEmbedToolbarButtonService*. Mit *IEmbedToolbarButtonService* wird vom *ContractDialog* bei dessen Einbettung in den *MainDialog* ein Button in der Toolbar integriert, über welchen der *ContractDialog* aktiviert werden kann. *IOverviewUpdateService* wird dafür benutzt, die im *ContractDialog* angezeigte Liste aller Lieferverträge zu aktualisieren, sobald über den *ContractDataDialog* ein neuer Vertrag erstellt oder die Daten eines Vertrages geändert wurden. *ISelectedItemUpdateService*

dient der Aktualisierung der Detaildatenanzeige in *ContractDataDialog* bei Auswahl eines anderen Liefervertrages in der Auflistung von *ContractDialog* durch den Nutzer. *IChoiceService* wird für die Weiterleitung dieser Auswahl an den *ContractProductChoiceDialog* genutzt. *IContractVendorItemsUpdateService* wird in der Rückrichtung dazu verwendet, die für einen Liefervertrag ausgewählten Vertragspositionen (ContractItems) und dem dazugehörigen Lieferanten dem *ContractDataDialog* zu melden, welcher die Informationen gemeinsam mit den anderen Vertragsdaten speichern kann.

Die QSL-Spezifikation für die einzelnen in Abbildung 4-18 dargestellten Komponentenschnittstellen der Liefervertragsverwaltung ist im Anhang dieser Arbeit zu finden. Sie wurde nach dem gleichen Schema durchgeführt, wie beim Anwendungskern und für ein Beispiel sei auf Abschnitt 4.4.1 mit Listing 4-1 verwiesen. Weitere Modelle für die Dialogkomponenten der Lieferantenverwaltung, Produktteilverwaltung, Nutzerverwaltung sowie der Anmeldung mit ihren Programmschnittstellen sind ebenfalls dem Anhang zu entnehmen.

In Bezug auf Traceability wurde vom Autor bei der Anwendung der beschriebenen Aktivitäten folgendes festgestellt. Die beim Use Case Storyboarding beschriebenen Interaktionen zwischen den einzelnen Dialogkomponenten, die im Kommunikationsdiagramm in Abbildung 4-17 dargestellt sind, werden unter Anwendung der GUI-Standardarchitektur bei der Spezifikation der Schnittstellen als Dienste abgebildet. Die Dienste werden entsprechend den in Abbildung 4-16 dargestellten Aggregationsbeziehungen von den Dialogkomponenten angeboten bzw. über die hierarchische Dienstverwaltung aufgerufen. Die Kommunikationsbeziehungen und Aggregationen können daher mit den Serviceschnittstellen über Traceability-Links verknüpft werden. Diese Links sind in tabellarischer Form im Anhang enthalten. Weitere Links ergeben sich aus der Verknüpfung der Systemfunktionalitäten des Funktionsbaums, welche den Anwendungsfällen entsprechen, mit den Dialogkomponenten der Benutzerschnittstelle. Damit wird der Konsistenzbedingung genüge getan, die für jeden Anwendungsfall, der mit einem Akteur verbunden ist, eine Benutzerschnittstelle vorschreibt.

Innensicht der GUI-Komponenten und Variabilitätsanalyse

Nach der Identifikation der Dialogkomponenten und der Spezifikation ihrer Schnittstellen kann die Innensicht der Komponenten beschrieben werden. Viel ist jedoch dazu nicht zu sagen, denn der innere Aufbau der Dialogkomponenten entspricht den Festlegungen der Standardarchitektur. Jeder Dialog besteht aus einem Dialogkern und einer Präsentation. Diese können je nach Bedarf als selbstständige Subkomponenten oder zusammen mit dem Dialog selbst betrachtet werden. Für die Umsetzung der fachlichen Dialoge von Vendorbase werden die abstrakten Klassen der vom Autor entwickelten Standard-T-Architektur genutzt. So dienen *AbstractDialog*, *AbstractDialogKernel* und *AbstractPresentation* als Ausgangspunkte für die Realisierung der Dialogkomponenten. Die anwendungsspezifischen Dialoge wie *ContractDialog* müssen sich somit nicht mehr um die Erfüllung der technischen Schnittstellen kümmern. Die zu erledigenden Aspekte konzentrieren sich auf die Festlegung fachlicher Aktionen, die Verwaltung von Datenwerten zur Anzeige in der Maske sowie die Implementierung der Dienste zur Kommunikation zwischen den Dialogen. Der Dialogkern greift dabei über die dienstorientierten Schnittstellen des Anwendungskernes auf die einzelnen Anwendungsfälle zu. Die Präsentation stellt die fachlichen Daten mit Swing-Steuerelementen grafisch dar und nimmt die Daten- und Aktionsanbindung zum Dialogkern mit den Adaptern vor, die von der T-Architektur bereitgestellt werden.

Abbildung 4-19 zeigt einmal beispielhaft den inneren Aufbau der Komponente *ContractDialog*. Oben im Bild sind die Schnittstellen und abstrakten Klassen der T-Architektur zu sehen. Sie gehören der Kategorie 0 bzw. aufgrund der bei der Standard-T-Architektur beschriebenen Abhängigkeit zu *JComponent* der Kategorie T an. In der Mitte befinden sich die für die Liefervertragsverwaltung spezifischen A-Klassen und Schnittstellen, sowie die Präsentation und die verwendeten Adapter der Kategorie R. Am unteren Rand der Abbildung befinden sich die verwendeten dienstorientierten Schnittstellen des Anwendungskernes. Trotz der Vererbung z. B. zwischen *AbstractDialog* und *ContractDialog* haben wir es nicht mit einer unzulässigen Vermischung der Kategorien A und T zu tun, denn die Verbindung wird an genau definierten Punkten innerhalb der Präsentation mit den R-Adaptern hergestellt.

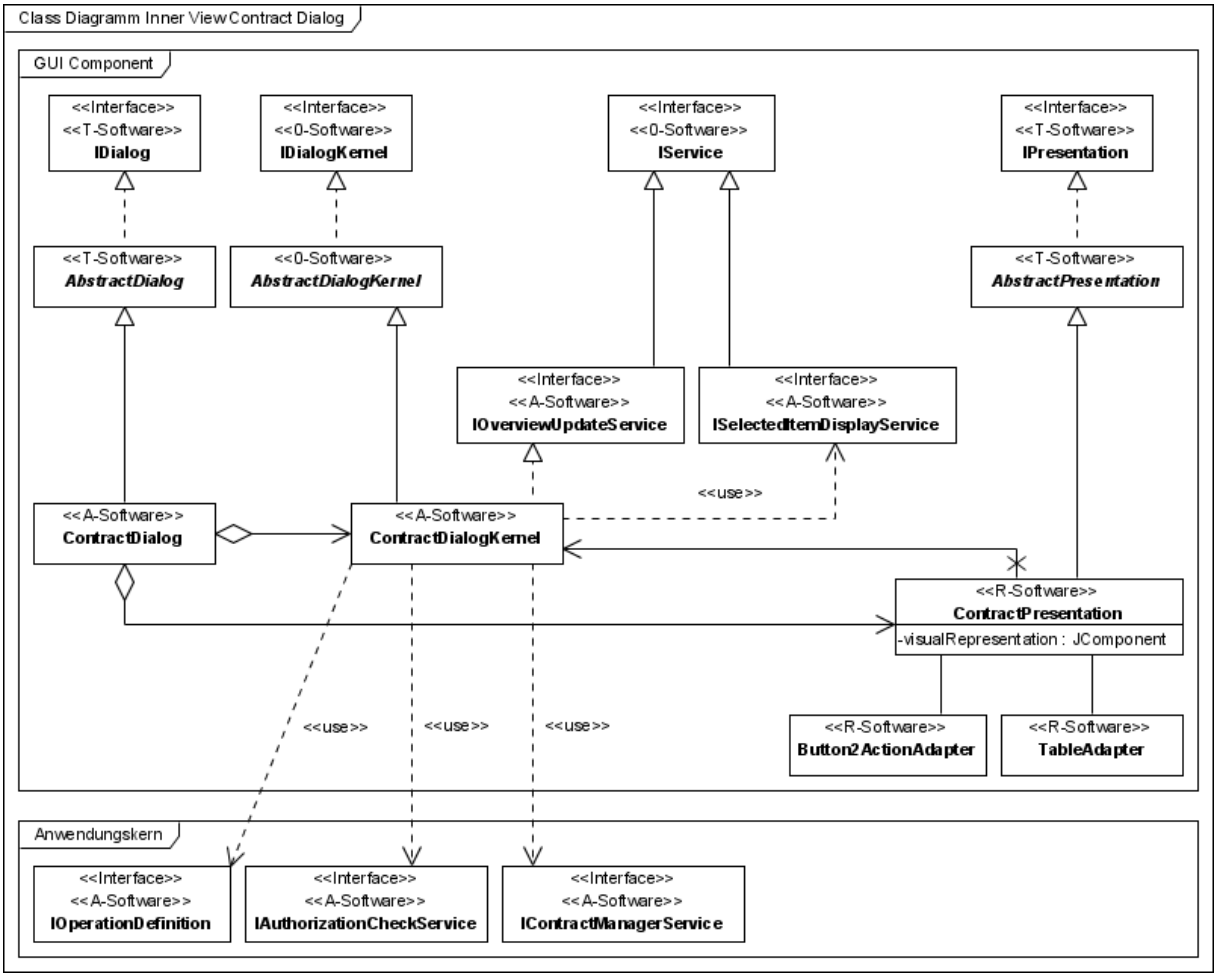

Abbildung 4-19: Innensicht der Komponente *ContractDialog*

Über die genaue Zuordnung der Dialogkomponenten und -klassen wie z. B. der Klasse *ContractDialog* zur Kategorie A ließe sich streiten, da die Präsentation R-Software ist, und die Kategorien ansteckend sind. Dadurch könnte der Dialog gleichermaßen der Kategorie R zugeordnet werden. Im Kategorienmodell verfeinert die R-Kategorie *VendorbaseGUISwing* die A-Kategorie *VendorbaseGUI* und ein Zugriff ist eigentlich nur von den oberen zu den unteren Kategorien erlaubt. Der Autor vertritt jedoch die in der Abbildung dargestellte Zuordnung, da so die Verantwortlichkeiten klar ausgedrückt werden. *ContractDialog* beispielweise muss entsprechend der Fachlichkeit der Liefervertragsverwaltung gestaltet werden und agiert nur zweitrangig als Konfigurationsmanager für die Präsentation und greift lediglich für die visuelle Darstellung direkt auf diese zu. Darüber hinaus ist die Trennung der Kategorien als Hilfsmittel für die Identifikation der Komponenten und für die klare Zuordnung von Verantwortlichkeiten anzusehen und soll nicht bei der Implementierung der Komponenten durch Klassen im Wege stehen.

Aus Gründen der Vollständigkeit sei noch auf die Variabilitätsanalyse verwiesen. Für Vendorbase wird diese nur kurz betrachtet, da es sich nur um eine Beispielanwendung handelt. Der technische Aspekt der Austauschbarkeit der GUI-Bibliothek Swing wurde bei der T-Architektur schon herausgestellt. In der Innensicht der Komponenten wird deutlich, dass für diesen Fall in allen Dialogen lediglich die Präsentationsklassen inklusive der Adapter geändert werden müssen. Die Dialogkerne bleiben unberührt. Ein weiterer Aspekt ist die Austauschbarkeit der Dialoge selbst. Hier ist die Architektur sehr flexibel. Durch die Einhaltung der Standardarchitektur könnten beispielsweise leicht neue Verwaltungsdialoge erstellt und in *MainDialog* eingebettet werden. Eine Anpassung bestehender Software für diesen Fall ist nur in der Komponente *Session* vorzunehmen, die die Dialoge verwaltet. Als Änderungsszenario vorstellbar ist zudem die Verwendung der *Gates* und *Transitions* von SalesPoint für die Zustandsverwaltung in den Dialogkernen. In den erstellten Dialogkomponenten wurden diese nicht eingesetzt, da die komponentenbasierte Überarbeitung von SalesPoint noch nicht abgeschlossen ist. Stattdessen wurden Zustandsvariablen verwendet. Die Verwendung der *Gates* und *Transitions* ist in der Architektur der einzelnen Dialoge bisher nicht vorgesehen. Eine Änderung wäre deshalb mit hohem Aufwand verbunden. Für neu zu entwickelnde Dialoge bzw. in anderen Projekten ist dennoch die Unterstützung einer überarbeiteten SalesPoint-Version denkbar.

Entwurf der visuellen Darstellung

Nachdem die Dialogkomponenten für die Benutzerschnittstelle identifiziert und ihre Schnittstellen spezifiziert wurden, muss noch die visuelle Darstellung, die Benutzeroberfläche, entworfen werden, bevor die Beispielanwendung Vendorbase vollständig in der Zielsprache implementiert werden kann. Hierfür werden die verschiedenen Ansätze zum Dialogdesign (Kapitel 2.3) sowie Ergonomie-Richtlinien (Kapitel 2.1) verwendet. Dies ist der Punkt in der Beschreibung des Entwurfs, an dem entsprechend des in Kapitel 1.2 genannten Zieles, die funktionalen Eigenschaften der Beispielanwendung mit den nichtfunktionalen Anforderungen aus der Software-Ergonomie verknüpft werden.

Der Rational Unified Process sieht für die visuelle Gestaltung der Dialoge die Aktivität User Interface Prototyping vor (Kapitel 2.3.1). Dabei können verschiedenartige Prototypen der Benutzeroberfläche wie Papierskizzen, Bilder eines Grafikprogramms oder ausführbare Pro-

totypen erstellt werden. Für Vendorbase beschränkt sich der Autor auf handgezeichnete Skizzen als Prototypen. Trotz dessen die GUI neu entwickelt wird, steht die frühere Version von Vendorbase zur Verfügung, die als Hinweis dafür genommen werden kann, wie die Anwendung im Betrieb aussehen kann. Nachfolgend wird das Vorgehen beim Prototypdesign für Vendorbase beschrieben.

Zunächst müssen alle primären Fenster identifiziert werden, mit denen der Nutzer interagiert. Jede im Use Case Storyboard beschriebene Dialogkomponente (im Original boundary-Klasse) ist ein Kandidat für ein primäres Fenster. Jedoch sollte die Länge des Navigationspfades nicht zu groß werden und alle primären Fenster sollten von einem Hauptdialog aus zugreifbar sein [KAB01, S. 178f]. Für Vendorbase ist dieser Hauptdialog, der für den Nutzer sichtbar wird, wenn die Anwendung gestartet ist, der an der Spitze der Aggregationshierarchie stehende *MainDialog*. Vom *MainDialog* aus können die anderen primären Fenster *ContractDialog, VendorDialog, ProductDialog, UserDialog* sowie *LogInDialog* aufgerufen werden. Alle anderen Dialoge wie *ContractDataDialog* werden nicht als eigenständige Fenster vorgesehen. Diese Unterdialoge werden in die primären Fenster wie *ContractDialog* integriert. *ContractDialog*, *ContractDataDialog* und *ContractProductChoiceDialog* erscheinen demnach dem Nutzer als ein zusammengesetztes Fenster. Auf diese Art und Weise kann der Navigationspfad kurz gehalten werden, was die Übersichtlichkeit des gesamten Systems erhöht.

Dieses Vorgehen entspricht auch den Regeln von Lauesens Ansatz (Kapitel 2.3.3). Es werden möglichst wenige Fenster erstellt (*few window templates*). Speziell die Anzahl an Fenstern pro Aufgabe (*task*) wie beispielsweise die Liefervertragsverwaltung ist minimal (*few window instances per task*). Der Nutzer benötigt nur ein Fenster, um einen Liefervertrag anzulegen, zu bearbeiten oder zu löschen. Ferner sind alle relevanten Daten für einen Liefervertrag in einem Dialog verwurzelt (*rooted in one thing*). Darüber hinaus bietet das Fenster für die Liefervertragsverwaltung mit dem *ContractDialog* sogar dann eine Übersicht über alle Lieferverträge, wenn die Daten eines einzelnen Vertrages im *ContractDataDialog* bearbeitet werden (*necessary overview of data*). Dies war in der alten Version von Vendorbase z. B. nicht der Fall. Sobald dort ein einzelner Vertrag bearbeitet wurde, war es unmöglich für den Nutzer eine Übersicht über die anderen Lieferverträge zu bekommen, was auf die Verwendung der Benutzerschnittstellenunterstützung von SalesPoint zurückzuführen ist. Die neue Benutzeroberfläche soll diesbezüglich besser benutzbar sein und orientiert sich an der achten Regel von Shneiderman (siehe Kapitel 2.1), die besagt, dass die Belastung des Kurzzeitgedächtnis-

ses des Nutzers reduziert werden soll. Im Beispiel geschieht dies dadurch, dass möglichst alle interessanten Daten für eine Aufgabe auf einen Blick verfügbar sind und selbst dann eine Auflistung als Übersicht angeboten wird, wenn gerade eine einzelne Entität bearbeitet wird.

Neben der Übereinstimmung mit den bisher beschriebenen Regeln korrespondiert die Aufteilung der Dialogkomponenten auf Fenster darüber hinaus mit der von Gulliksen et al. bei ihrem Vorgehen beschriebenen Workspace-Metapher (Kapitel 2.3.2). Die vom Nutzer ausführbaren Aktionen werden in die Arbeitssituationen Liefervertragsverwaltung, Lieferantenverwaltung, Produktteilverwaltung sowie für den Administrator die Nutzerverwaltung aufgeteilt. Dem folgend stehen mit den primären Fenstern von *ContractDialog, VendorDialog, ProductDialog* und *UserDialog* Workspaces zur Verfügung, die den Arbeitssituationen entsprechend aufgeteilt sind. Die Workspaces werden in *MainDialog* organisiert und sind für den Benutzer unabhängig voneinander zugreifbar. Somit wird der Wechsel zwischen verschiedenen Arbeitssituationen optimal unterstützt. Hierin spiegelt sich z. B. auch ein Faktor zur Messung der Usability nach Lauesen wider, die *„task efficiency“* (Kapitel 2.1).

Sind die Hauptfenster identifiziert, ist ihre Visualisierung zu beschreiben und Operationen und Eigenschaften der primären Fenster müssen festgelegt werden. Der Entwurf der primären Fenster hat entscheidenden Einfluss auf die Usability der Anwendung. Für *MainDialog* wird zur Realisierung mit Swing ein *JFrame* mit einer *DesktopPane* verwendet. Darin werden die anderen primären Fenster, oder Workspaces, als interne Fenster realisiert. Dies ist ein unter Windows von vielen Anwendungen wie PaintShop, Photoshop und Ähnlichen bekannter Standard. Abbildung 4-20 zeigt beispielhaft die internen Fenster zur Bearbeitung von Bildern in PaintShop. Zwischen den internen Fenstern kann beliebig hin und her gewechselt werden. Sie können minimiert oder maximiert werden. Für Vendorbase tritt an die Stelle des Hauptfensters der Anwendung der *MainDialog*. Die anderen primären Fenster für Liefervertrags-, Lieferanten-, Produktteil- und Nutzerverwaltung entsprechen im Screenshot den internen Fenstern für die Bilddateien. Sie werden als *JInternalFrame* realisiert und sind über eine Werkzeugleiste im *MainDialog* zugreifbar.

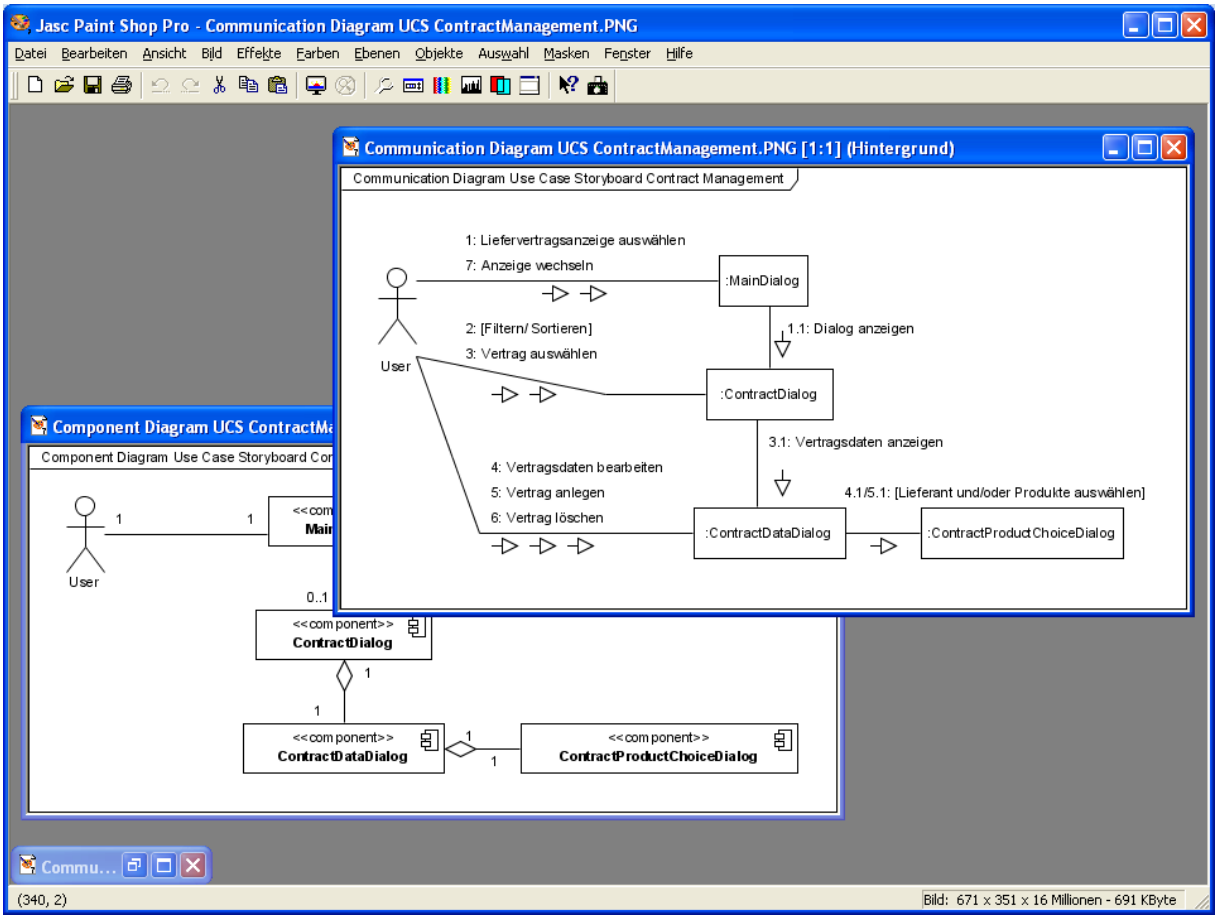

Abbildung 4-20: PaintShop als Beispiel einer Desktop-Anwendung mit internen Fenstern

Nach dem Entwurf der Maske des Hauptfensters verbleibt das Design der anderen primären Fenster. Beim Maskenentwurf müssen für die anzuzeigenden Daten die Präsentationsformate ([Lau05, S. 85ff.], siehe Kapitel 2.1) und damit die einzelnen Steuerelemente festgelegt werden. Dafür wurden zunächst handgezeichnete Prototypen angefertigt. Abbildung 4-21 zeigt die Papierskizze für die Maske des Dialogs zur Liefervertragsverwaltung. Sie zeigt das interne Fenster des *ContractDialog*, in dessen oberer Hälfte sich eine Tabelle befindet, welche die Lieferverträge des Systems anzeigt und die gefiltert werden kann. Die Sortierung soll über Klicks auf die Tabellenspalten vorgenommen werden. In der unteren Bildhälfte ist der *ContractDataDialog* zu sehen, welcher in den *ContractDialog* integriert ist. Der *ContractDataDialog* bietet Schaltflächen, mit denen die Anwendungsfälle angesteuert werden können, sowie mit Labeln ausgezeichnete Text- sowie Comboboxen zur Darstellung der Detaildaten eines Liefervertrages. In den *ContractDataDialog* wiederum integriert ist auf der rechten Seite der *ContractProductChoiceDialog*. In diesem werden Tabellen zur Auswahl des Lieferanten sowie der einzelnen Produkte als Vertragsgegenstände geboten. Beim Entwurf der Masken für die Datendialoge ist vor allem zu beachten, dass jedes Attribut einer darzustellenden Entität eine visuelle Repräsentation bekommt (*all data accessible* – 8. Design-Regel nach

Lauesen). Nur so ist gewährleistet, dass alle Daten vom Nutzer bearbeitet werden können. Die Dialoge der Lieferanten-, Produktteil- und Nutzerverwaltung werden ähnlich gestaltet, um Konsistenz zu erreichen (erste Regel nach Shneiderman). Sie unterscheiden sich lediglich in der Anzahl der benutzbaren Schaltflächen und der Steuerelemente für die Darstellung der Detaildaten. Die handgezeichneten Skizzen dienen selbstverständlich nur als erster Entwurf. Die fertige Maske kann später im Detail abweichen. Vor allem die räumliche Verteilung der Steuerelemente kann sich vom Papier zur fertigen Maske deutlich unterscheiden. Ferner können andere Steuerelemente für bestimmte Dateneingaben verwendet werden. So wird aus Gründen der Benutzerfreundlichkeit beispielsweise für die Eingabe einer Datumsangabe im Programm später kein Textfeld sondern ein *JXDatePicker* als spezielles Steuerelement verwendet, welches mit der GUI-Bibliothek SwingX mitgeliefert wird. Diese Entscheidung ist auf die von Galitz beschriebenen grundsätzlichen Prinzipien der Dialoggestaltung wie Einfachheit (s*implicity*) und Effizienz (*efficiency*) zurückzuführen (siehe Kapitel 2.1).

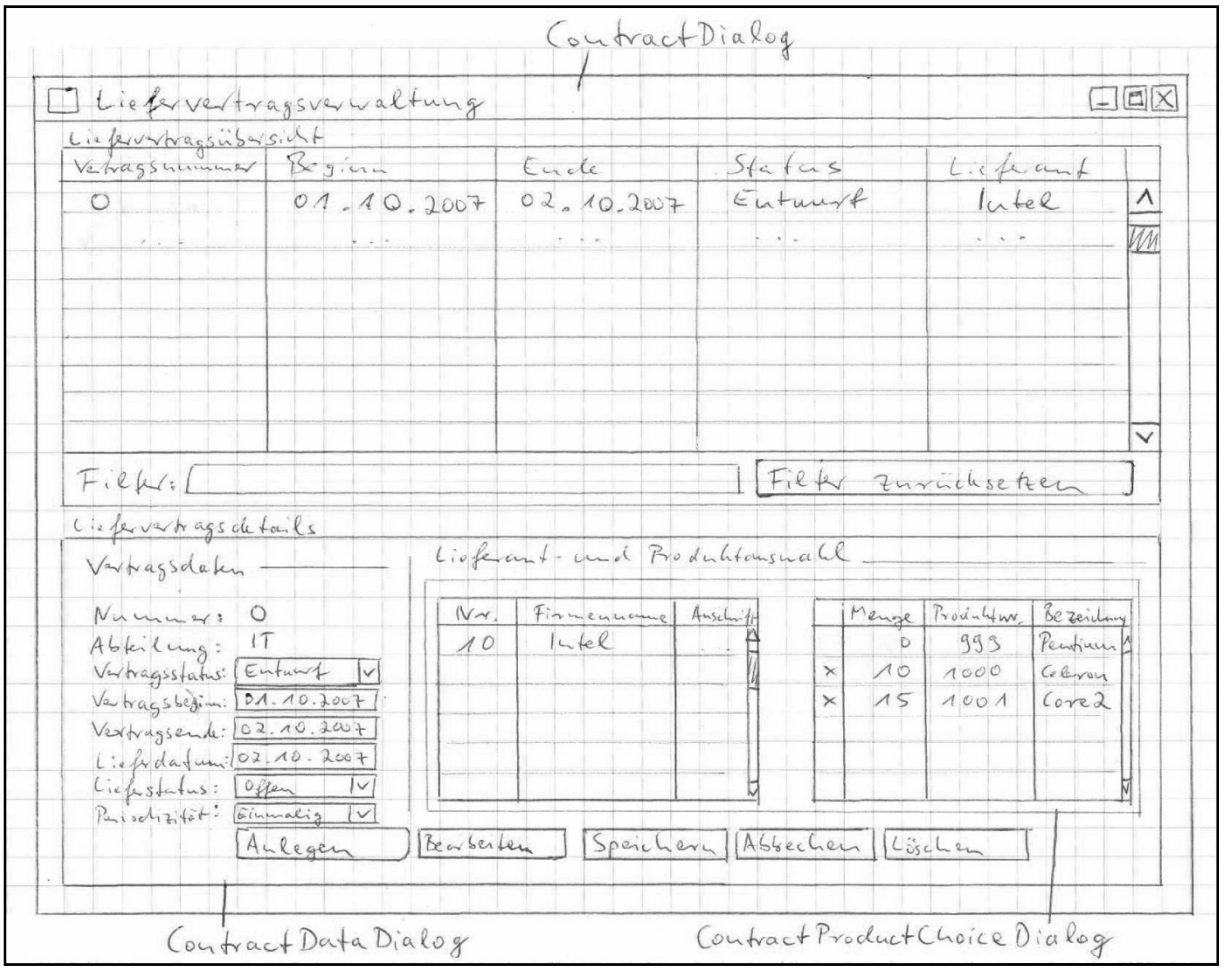

Abbildung 4-21: Skizze für die Maske zur Liefervertragsverwaltung

Nach dem Prototypentwurf können die Dialoge mit Swing in Java programmiert werden. Für Vendorbase wurde zusätzlich die Bibliothek JGoodies verwendet. Mit JGoodies Forms kön-

nen die Steuerelemente ganz einfach über ein Layout-Gitter positioniert werden. In Abbildung 4-22 ist die Skizze für den *LogInDialog* mit eingezeichnetem Gitter zu sehen. Die Steuerelemente wie Label, Textfelder und Schaltflächen können mithilfe von JGoodies Forms in die einzelnen Gitterzellen gesetzt werden und sind somit optimal ausgerichtet. Eine exakte Ausrichtung der Steuerelemente bedient Design-Kriterien wie Klarheit (*clarity*) und ästhetische Anmutung (*aestetically pleasing*) der Benutzeroberfläche.

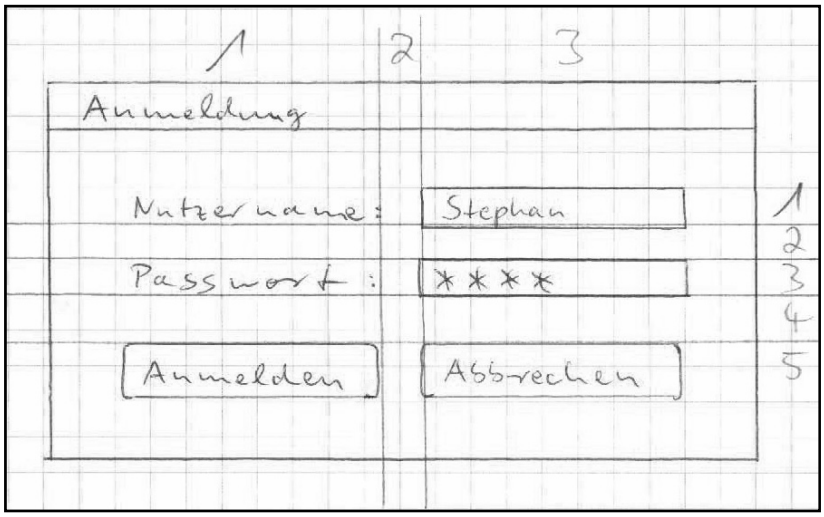

Abbildung 4-22: Skizze zum *LogInDialog* mit Layout-Gitter

Abbildung 4-23 zeigt schließlich einen Screenshot der endgültigen Version der Benutzeroberfläche für die Neuentwicklung von Vendorbase. Vom Autor wurde versucht, das Design so verständlich wie möglich für den Nutzer zu machen (Usability-Faktor *„ease-of-learning“*). Dazu gehört es, Muster wiederzuverwenden, die dem Nutzer bekannt sind. So wurde als Tabelle beispielsweise eine *JXTable* aus dem SwingX-Paket verwendet. Deren Spalten sind standardmäßig durch einen Mausklick auf den Spaltentitel sortierbar, wie dies auch im Windows-Explorer der Fall ist (*reuse old data presentations*). Weiterhin wurde Wert darauf gelegt, das Look&Feel der Anwendung so nah wie möglich am Betriebssystem auszurichten (*follow platform*). Unterstützung hierbei bieten vor allem JGoodies Looks und Forms. Die Filterfunktion wurde so implementiert, dass sie die Ergebnismenge direkt bei Tasteneingabe anpasst (Live-Search), und dem Nutzer somit informatives Feedback bietet (3. Regel von Shneiderman). Ferner wurde z. B. das Gestaltgesetz der Nähe berücksichtigt. Es findet Anwendung, um grafische Elemente über räumliche Nähe visuell zu gruppieren. So bekommen zusammengehörende Steuerelemente wie die Schaltflächen in der Button-Leiste einen geringeren Abstand zueinander als zu benachbarten gruppierten Elementen. Unterstützung bei der Gruppierung bieten außerdem die speziellen Möglichkeiten der SwingX-Bibliothek mit der *JXTitledPane* und der Hinterlegung von Steuerelementen mit Schatten. Alternativ hätten

zur Gruppierung von Elementen die Standardrahmen der Swing-Bibliothek genutzt werden können. Diese sind jedoch nicht so elegant und bei Schachtelung für das Design sogar als kritisch anzusehen [Len07].

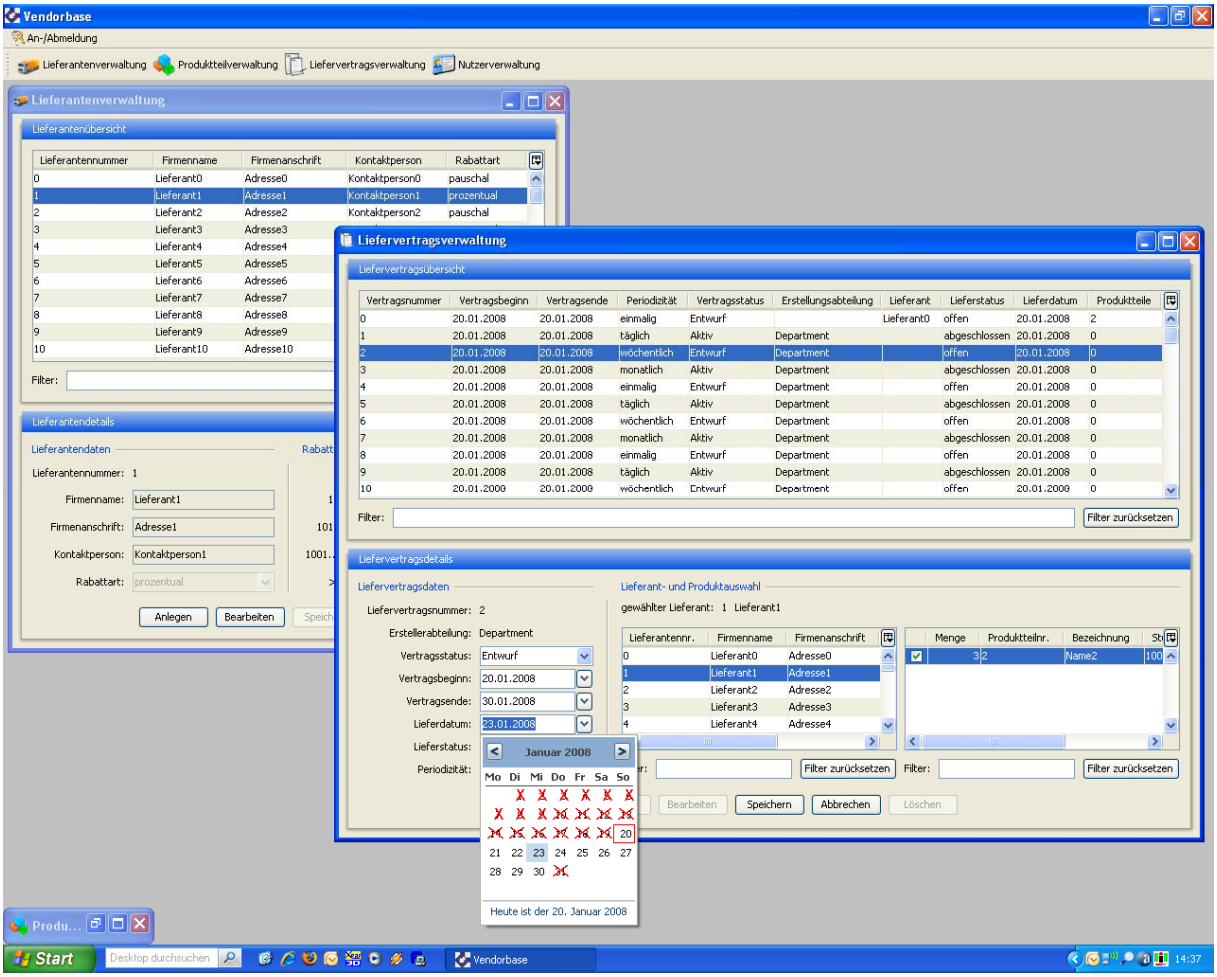

Abbildung 4-23: Screenshot der finalen Benutzeroberfläche von Vendorbase

Damit ist das Vorgehen zur Erstellung einer neuen grafischen Benutzerschnittstelle abgeschlossen. Traceability-Beziehungen wurden dabei für das Design der Benutzeroberfläche dokumentiert. So sind beispielsweise die Festlegungen der verwendeten Steuerelemente in den Dialogen auf die Beschreibungen im Datenmodell zurückführbar. Die erstellten Traceability-Links sind dem Anhang zu entnehmen. Mit Abschluss der GUI-Modellierung und dem Design der Masken endet auch das gesamte methodische Vorgehen bei der Beschreibung der Anwendungsarchitektur.

5 Resümee und Ausblick

Ziel dieser Arbeit war es, die Architekturentwicklungsmethode Quasar bei der Weiterentwicklung des SalesPoint-Frameworks anzuwenden. Einerseits sollten die getroffenen Entwurfsentscheidungen dokumentiert und entsprechende Traceability-Links erstellt werden. Andererseits sollte die Methode Quasar kritisch untersucht und an den nötigen Stellen verfeinert werden, sodass sich ein lückenloses Vorgehen für die Zuordnung von Traceability-Links ergibt.

Das methodische Vorgehen nach Quasar wurde in der vorliegenden Arbeit am SalesPoint-Beispiel Vendorbase angewendet. Das Hauptaugenmerk lag dabei auf der Neuentwicklung einer GUI-Architektur, denn die GUI-Unterstützung von SalesPoint wird als besonders kritisch angesehen und wurde bisher nicht überarbeitet. Bei der Bearbeitung der Aufgabe wurde festgestellt, dass die GUI bei Vendorbase nicht losgelöst von den Domainfunktionalitäten betrachtet werden kann. Dieser Umstand wurde vom Autor genutzt, um das methodische Vorgehen nach Quasar nicht nur für die GUI sondern auch für den Anwendungskern und damit im Ganzen anzuwenden.

Bei der Untersuchung von Quasar stellte sich heraus, dass die Methode recht gut für den Entwurf des Anwendungskernes eines Softwaresystems geeignet, jedoch unvollständig ist. Sie definiert Softwarekategorien, die als Grundlage für eine komponentenbasierte Architektur dienen. Aber Quasar selbst umfasst keine Aktivitäten, welche die Festlegung der Anforderungen, das Requirements-Engineering, unterstützen. Für Vendorbase wurde dies nicht näher untersucht, da die Anforderungen bereits aufgestellt waren. Des Weiteren ist das Quasar-Vorgehen in der Hinsicht unschlüssig, wie im Detail die Softwarekategorien und Komponenten für das zu entwickelnde System zu identifizieren sind. In der Arbeit wurde gezeigt, wie mithilfe eines Funktionsbaumes, der als Konzept schon von Herpel genutzt wurde, die Entscheidungen für die Festlegung des Kategorien- und Komponentenmodells erleichtert werden können. Es wurde weiterhin dargestellt, wie die Aktivitäten von Quasar zur Beschreibung von Komponenten und ihren Schnittstellen anzuwenden sind.

Die Lücken in der Quasar-Methode bezüglich des GUI-Designs wurden vom Autor mit Aktivitäten aus dem Rational Unified Process geschlossen. Auf Basis des Anwendungsfallmodells und des Funktionsbaumes wurden Use Case Storyboards erstellt, mit denen die Komponenten der Benutzerschnittstelle und ihre Programmschnittstellen spezifiziert werden können. Der Entwurf der visuellen Darstellung erfolgte mithilfe von User Interface Prototypen.

Die Entwurfsentscheidungen, die bei der Anwendung der Quasar-Methode getroffenen wurden, wurden dokumentiert und mittels Traceability-Links nachvollziehbar abgelegt. Die Traceability-Links wurden nach den verschiedenen Schritten des Vorgehens bzw. den Übergängen zwischen den erstellten Modellen des Systems untergliedert.

Für die Entwicklung der grafischen Benutzerschnittstelle fiel ferner auf, dass Quasar eine Standardarchitektur bietet. Jedoch wird weder thematisiert, wie die benötigten Dialogkomponenten durch den Architekten gefunden werden können, noch wie die visuelle Darstellung der Dialoge zu entwerfen ist. Aus diesem Grund wurden vom Autor unterschiedliche Ansätze analysiert, die sich mit dem Dialogdesign befassen. Überdies wurde untersucht, welche Kriterien und Richtlinien die Software-Ergonomie für die Gestaltung von Benutzeroberflächen bereithält. Mit diesen Informationen ausgestattet, wurde die grafische Benutzerschnittstelle für Vendorbase neu entwickelt. Die Benutzeroberfläche wurde an software-ergonomischen Kriterien ausgerichtet, sodass das Design möglichst gute Usability aufweist.

Die Standard-T-Architektur, die vom Autor entwickelt wurde, ist eine Realisierung der Client-Standardarchitektur nach Quasar. Bei der Entwicklung wurden die technologischen Bedingungen von Vendorbase auf die Standardarchitektur angewendet. Dazu wurden die von Quasar beschriebenen Konzepte und Schnittstellen übernommen und mussten beispielsweise bezüglich der Verwaltung und Einbettung von Dialogen vervollständigt werden. Das Ergebnis ist eine technische Architektur, die auch für andere Java-Projekte wiederverwendet werden kann.

Insgesamt wurden die Ziele der Arbeit erreicht. Die auf SalesPoint basierende Anwendung Vendorbase wurde mit der Quasar-Methode neu entwickelt. Das Vorgehen wurde dazu systematisiert und erweitert, um Lücken zu füllen, und Entwurfsentscheidungen sowie Traceability-Beziehungen wurden dokumentiert. Dennoch bleiben einige Aspekte für weitere Forschungsarbeit offen.

Für die weitere Überarbeitung des SalesPoint-Frameworks könnte überdacht werden, wie aus dem Framework eine Quasar-konforme Standard-A-Architektur für Shop-Anwendungen entstehen kann. Quasar selbst bietet hier wenig Hinweise, da Standard-A-Architekturen bei Individualsystemen, wie sd&m sie baut, keine Anwendung finden [Sie04, S. 146]. Es gibt jedoch zwei Aspekte, die mindestens berücksichtigt werden müssen. Zum einen ist zu überlegen, welche Persistenzlösung für SalesPoint-Anwendungen später eingesetzt werden soll. Die persistente Speicherung der Entitätsobjekte wurde für die Beispielanwendung vorerst weggelassen. Zum anderen sollte die komponentenbasierte Überarbeitung für das Framework zunächst vervollständigt werden, sodass dessen Konzepte wie die Prozessverwaltung mit Zuständen und Transitionen ohne unerwünschte Abhängigkeiten genutzt werden können. Dann kann auch die Idee von SalesPoint, Multi-User-Anwendungen zu simulieren, umgesetzt werden. Ein Multi-User-Betrieb mit Vendorbase ist bisher nicht möglich. In der Beispielanwendung werden zwar mehrere Nutzer verwaltet, jedoch können sich diese nicht parallel anmelden und arbeiten. Hier wären Änderungen an der Anwendung und eben obendrein an SalesPoint nötig, um Unterstützung durch das Framework zu bekommen.

Das methodische Vorgehen bei der Erstellung des Anwendungskernes könnte in Zukunft weitergehend untersucht werden. So wäre interessant, inwiefern sich das Vorgehen übertragen lässt, oder ob es angepasst werden muss, wenn eine andere T-Architektur benötigt wird. Ein geändertes Szenario könnte z. B. die Einführung einer Transaktionsverwaltung, einer persistenten Speicherung oder gar einer speziellen technologischen Infrastruktur (EJB, .Net usw.) umfassen. Überdies haben andere Architekten möglicherweise eine Sichtweise, die sich von der des Autors unterscheidet, und würden bestimmte Aktivitäten der Methode adaptieren oder weiter verfeinern. Zusätzlich müssten in anderen Projekten unter Umständen weitere Aktivitäten für die Erstellung des Anwendungsfallmodells berücksichtigt werden. Daneben ist die verwendete halbformale Sprache QSL für die Spezifikation der Schnittstellen sicherlich nicht das erreichbare Optimum. Wünschenswert wäre eine Sprache, die so einfach anzuwenden ist wie Quasar, bei der jedoch ähnlich wie bei JML (bis Java 1.4) die Implementierung der Schnittstellen gegen die Spezifikation verifizierbar ist. So könnte die Einhaltung der Schnittstellenverträge automatisch überprüft werden (siehe [LC06]).

Im Hinblick auf das Vorgehen zur GUI-Erstellung könnte eine Integration anderer HCI-Methoden erwogen werden. Aktivitäten solcher Methoden, beispielsweise der Virtual-Windows-Methode von Lauesen, könnten anstelle oder ergänzend zu den verwendeten Ent-

wurfsschritten aus dem RUP integriert werden. Betrachtenswert wäre außerdem, wie die erstellte Standard-T-Architektur adaptiert oder nach der GUI-Standardarchitektur von Quasar neu aufgesetzt werden muss, wenn sich die Infrastruktur ändert. So könnte die Erstellung eines Web-Front-end genauer untersucht werden.

Künftig ist eine Verallgemeinerung der Benutzerschnittstelle von Vendorbase nötig, um eine Grundlage für alle Shop-Anwendungen zu schaffen, die von der Unterstützung durch das SalesPoint-Framework profitieren sollen. Die Dialoge, die für Vendorbase entworfen wurden, sollten zu Standard-A-Komponenten verallgemeinert werden. Speziell für SalesPoint ist es angebracht, allgemeine wiederverwendbare Standard-Dialoge zur Verfügung zu haben. Weiterhin muss der Dialograhmen für die Simulation eines Multi-User-Betriebs anwendungsspezifisch geändert werden. Denn für Vendorbase wird nur eine Sitzung erstellt, in der auch der Dialog zur Anmeldung läuft. Dieser müsste sitzungsunabhängig gemeinsam mit einem übergeordneten Dialog zur Verwaltung der Hauptfenster der verschiedenen Nutzer existieren. Unabhängig davon ist beim Design der visuellen Darstellung noch Spielraum für die Umsetzung weiterer Software-Ergonomie-Richtlinien. So könnten, wie von Shneiderman in seinen Regeln gefordert, Rücksetzmöglichkeiten (Undo/Redo) oder zusätzliche Abkürzungen für erfahrene Benutzer eingebaut werden.

Schließlich sind weitere Forschungen anzustreben, um ein allgemeines Traceability-Modell für Quasar zu entwickeln. Zusätzliche Aktivitäten und Traceability-Links ergeben sich bei Berücksichtigung der Anforderungserhebung und Modellierung der Anwendungsfälle. Die Links zwischen den verschiedenen Artefakten sind noch genauer auf Typen hin zu untersuchen und es sollte aus dem Modell von vornherein ersichtlich sein, welche Links entsprechend den Aktivitäten zu erstellen sind. Nur so kann der Architekt oder Entwickler bei der Anwendung der Quasar-Methode, sei es bei der Neuentwicklung eines Systems oder partiell bei der Wartung, in seiner Tätigkeit optimal unterstützt werden.

A Anhang

Im Anhang dieser Arbeit werden zusätzliche Arbeitsergebnisse aufgelistet, auf die im Hauptteil hingewiesen wurde. Dazu zählen die Traceability-Links, Modelldiagramme sowie die Use Case Storyboards. Die teils umfangreichen Tabellen und Darstellungen werden nicht direkt im Anhang dargestellt und sind nur auf der beiliegenden CD verfügbar. Für diese Resultate werden das Verzeichnis und der Dateiname auf der CD angegeben.

A.1 Traceability-Links

Die Traceability-Links, die bei der Anwendung der erweiterten Quasar-Methode erstellt wurden, sind auf der beiliegenden CD im Ordner `\traceability\` zu finden. Die Zusammenstellung der Traceability-Links wurde entsprechend der Verknüpfung zwischen den verschiedenen Artefakten untergliedert. Die Tabellen mit den Links liegen sowohl im PDF-Format als auch als Excel-Sheet vor. In Tabelle A-1 sind die Dateinamen der Tabellen ohne Endung und eine Beschreibung der Art der enthaltenen Traceability-Links aufgelistet.

Dateiname	Beschreibung
Traceability-Links Anforderungen - Datenmodell	Verknüpfung der im Pflichtenheft beschriebenen Datenstrukturen mit den Elementen des Datenmodells (S. 43ff.)[19]
Traceability-Links Datenmodell - Komponentenmodell AWK	Zuordnung der Entitäten des Datenmodells zu Komponenten des Anwendungskerns nach den Konsistenzbedingungen (S. 58ff.)
Traceability-Links Datenmodell - visuelle Darstellung	Verbinden der Elemente des Datenmodells mit den Benutzersteuerelementen in der visuellen Darstellung, die für die Präsentation verantwortlich sind (S. 98ff.)

[19] Die Seitenzahlen beziehen sich auf die Abschnitte der Arbeit, in denen die entsprechenden Aktivitäten beschrieben sind, während denen die Links erstellt wurden.

Traceability-Links Funktionsbaum - Kategorienmodell	Erstellung der Softwarekategorien mithilfe der Funktionalitäten (S. 51ff.)
Traceability-Links Funktionsbaum - Komponentenmodell AWK	Verknüpfung der im Anwendungskern spezifizierten Komponenten und deren Schnittstellen mit den Funktionalitäten, aus denen sie hervorgehen (S. 58ff.)
Traceability-Links Funktionsbaum - Komponentenmodell GUI	Verlinken der für die Benutzerschnittstelle erstellten Komponenten und deren Programmschnittstellen mit den zugrunde liegenden Funktionalitäten (S. 88ff.)
Traceability-Links Kategorienmodell - Komponentenmodell AWK	Entwurf der Komponenten des Anwendungskerns und ihrer Schnittstellen in den Softwarekategorien (S. 58ff.)
Traceability-Links Kategorienmodell - Komponentenmodell GUI	Zuordnung der Komponenten und Programmschnittstellen der GUI mit den Softwarekategorien, denen sie angehören (S. 88ff.)
Traceability-Links Schnittstellen - Testfälle	Erstellen von Testfällen für die Programmschnittstellen und ihre Implementierungen entsprechend den Schnittstellenverträgen (S. 60ff. und S. 93ff.)
Traceability-Links UseCaseModell - Funktionsbaum	Erfassen der Funktionalitäten im Funktionsbaum anhand der Anwendungsfälle (S. 47ff.)
Traceability-Links UseCaseModell - UseCaseStoryboards	Entwurf der Use Case Storyboards für die Anwendungsfälle (S. 88ff.)
Traceability-Links UseCaseStoryboards - Komponentenmodell GUI	Identifikation und Spezifikation der GUI-Komponenten mithilfe der Use Case Storyboards (S. 88ff.)

Tabelle A-1: Auflistung der Tabellen mit den Traceability-Links

A.2 Modelldiagramme

Die einzelnen Modelle und dazugehörige Dateien, die mit Anwendung der Methode entstanden sind, werden in diesem Abschnitt aufgeführt. Auf der CD im Verzeichnis `\modelle\` sind in den in Tabelle A-2 angegebenen Unterverzeichnissen jeweils Projektdateien von Visual Paradigm for UML[20] sowie daraus erzeugte Bilder im PNG-Format enthalten.

Dateiname	Beschreibung
Unterverzeichnis `\anwendungsfallmodell\`	
UCD Liefervertragsverwaltung	Dies ist ein Anwendungsfalldiagramm zu den Use Cases der Liefervertragsverwaltung. Die übrigen Anwendungsfälle sind dem Pflichtenheft [FKR07] zu entnehmen.
Unterverzeichnis `\datenmodell\`	
Datenmodell	Ein Klassendiagramm, welches die Entitäten ihre Attribute und Assoziationen beschreibt.
Unterverzeichnis `\funktionsbaum\`	
Funktionsbaum gesamt	Das Klassendiagramm, welches den Funktionsbaum für Vendorbase darstellt.
Funktionsbaum Liefervertragsverwaltung initial.png Funktionsbaum Liefervertragsverwaltung iteriert.png Funktionsbaum Liefervertragsverwaltung.vpp	Die Dateien enthalten eine Darstellung mit den Funktionalitäten der Liefervertragsverwaltung. Die Bilder „initial“ und „iteriert“ stellen verschiedene Phasen bei der Erstellung dar.
Unterverzeichnis `\kategorienmodell\`	
Kategoriengraph SalesPoint	Der Kategoriengraph von SalesPoint, der als Ausgangspunkt dient.
Kategoriengraph Vendorbase mit SalesPoint	Dies ist ein Klassendiagramm mit den für Vendorbase verwendeten Kategorien.

[20] Die Installationsdatei für die Visual Paradigm Suite 3.0, in der Visual Paradigm for UML 6.0 enthalten ist, wird auf der CD beigefügt. Zur Verwendung der Projektdateien ist eine Lizenz nötig und die Projekte müssen in einen Workspace kopiert werden.

Kategoriengraph Vendorbase ohne SalesPoint	Das Klassendiagramm enthält eine Darstellung der für Vendorbase benötigten Kategorien ohne Verwendung von SalesPoint (siehe Abschnitt 4.3.2).
Unterverzeichnis `\komponentenmodell\AWK\`	
Komponente Authorization	Enthält ein Paketdiagramm für die Komponente Authorization.
Komponente ContractManager	Enthält ein Paketdiagramm für die Komponente ContractManager.
Komponente ProductManager	Enthält ein Paketdiagramm für die Komponente ProductManager.
Komponente Shop	Enthält ein Paketdiagramm für die Komponente Shop mit Integration der Pakete der Subkomponenten.
Komponente VendorManager	Enthält ein Paketdiagramm für die Komponente VendorManager.
Komponenten SalesPoint	Darstellung des Komponentenentwurfs für SalesPoint (vgl. Herp07).
Komposition AWK	Ein Kompositionsstrukturdiagramm für die Komponente Shop, die den Anwendungskern repräsentiert.
Schnittstellen AWK Komponenten.vpp Schnittstellen AWK Subkomponenten.PNG Schnittstellen Komponente Shop.PNG	Darstellung der Komponenten des Anwendungskerns mit ihren angeforderten und bereitgestellten Schnittstellen.
Unterverzeichnis `\komponentenmodell\GUI\`	
Dialogs ContractManagement	Komponentendiagramm mit den Dialogkomponenten und Programmschnittstellen der Liefervertragsverwaltung.
Dialogs LogIn	Komponentendiagramm mit den Dialogkomponenten und Programmschnittstellen für die Anmeldung.
Dialogs ProductManagement	Komponentendiagramm mit den Dialogkomponenten und Programmschnittstellen der Produktteilverwaltung.

Dialogs UserManagement	Komponentendiagramm mit den Dialogkomponenten und Programmschnittstellen der Nutzerverwaltung.
Dialogs VendorManagement	Komponentendiagramm mit den Dialogkomponenten und Programmschnittstellen der Lieferantenverwaltung.
Inner View ContractDialog	Ein Klassendiagramm, welches beispielhaft den inneren Aufbau einer Dialogkomponente mit den verwendeten Schnittstellen der Standard-T-Architektur und des Anwendungskernes darstellt.
T-Architektur GUI	Kompositionsstrukturdiagramm für den Aufbau eines Dialogs nach der Standardarchitektur.
UCS ContractManagement.vpp UCS Communication Diagram ContractManagement.PNG UCS Component Diagram ContractManagement.PNG	Kommunikations- und Komponentendiagramm für das Use Case Storyboard zur Liefervertragsverwaltung.
UCS LogIn.vpp UCS Communication Diagram LogIn.PNG UCS Component Diagram LogIn.PNG	Kommunikations- und Komponentendiagramm für das Use Case Storyboard zur Anmeldung.
UCS ProductManagement.vpp UCS Communication Diagram ProductManagement.PNG UCS Component Diagram ProductManagement.PNG	Kommunikations- und Komponentendiagramm für das Use Case Storyboard zur Produktteilverwaltung.
UCS UserManagement.vpp UCS Communication Diagram UserManagement.PNG UCS Component Diagram UserManagement.PNG	Kommunikations- und Komponentendiagramm für das Use Case Storyboard zur Nutzerverwaltung.

UCS VendorManagement.vpp UCS Communication Diagram VendorManagement.PNG UCS Component Diagram VendorManagement.PNG	Kommunikations- und Komponentendiagramm für das Use Case Storyboard zur Lieferantenverwaltung.

Tabelle A-2: Modelldiagramme

A.3 Use Case Storyboards

Die Use Case Storyboards, die in Abschnitt 4.4.2 zur Identifikation der Dialogkomponenten erstellt wurden, sind auf der CD im Verzeichnis `\storyboards\` zu finden. In Tabelle A-3 sind die Dateinamen mit Beschreibung aufgelistet.

Dateiname	Beschreibung
Use Case Storyboard Anmeldung	Das Storyboard umfasst die Funktionalität zum Anmelden am Anwendungskern.
Use Case Storyboard Lieferantenverwaltung	Dieses Storyboard betrifft die Funktionalitäten der Lieferantenverwaltung.
Use Case Storyboard Liefervertragsverwaltung	Das Storyboard ist identisch zur Beschreibung der Liefervertragsverwaltung in Kapitel 4.4.2.
Use Case Storyboard Nutzerverwaltung	Dieses Storyboard enthält die Nutzerverwaltung.
Use Case Storyboard Produktteilverwaltung	Das Storyboard zur Produktteilverwaltung.

Tabelle A-3: Use Case Storyboards für den GUI-Entwurf

A.4 Quellcode

Der Quellcode für die Beispielanwendung Vendorbase und für die Schnittstellen und Klassen der Standard-T-Architektur, der mit Anwendung der Methode entstanden ist, befindet sich auf der CD im Verzeichnis `\quellcode\`. Der Code ist als Eclipse-Projekt verfügbar und setzt das JDK 1.6 voraus. Bei Import des Projektes in die Entwicklungsumgebung Eclipse muss unter Umständen der Erstellungspfad zu den Bibliotheken angepasst werden. Die Java-Quelldateien befinden sich im Unterverzeichnis `\src\`, die benötigten Bibliotheken in `\lib\`. Für die Übersetzung der Quelldateien und die Erstellung der JAR-Archive sowie der Java-Dokumentation wird ein ANT-Build-Skript mitgeliefert. Mit diesem können auch die erstellten JUnit-Tests gestartet werden.

B Literaturverzeichnis

[Ade07] **Adersberger, Josef:** *Konsistenzbedingungen bei der Entwicklung einer Softwarearchitektur nach der QUASAR Methode.* Lehrstuhl für Software Engineering, Friedrich-Alexander-Universität Erlangen-Nürnberg. 2007.

[BC89] **Beck, Kent; Cunningham, Ward:** *A laboratory for teaching object oriented thinking.* In: Meyrowitz, Norman (Hrsg.): Proceedings on Object-oriented programming systems, languages and applications (OOPSLA ´89), New Orleans, Louisiana, USA, 1989. ACM, 1989, S. 1-6, 0-89791-333-7

[BDP06] **Broy, Manfred; Deissenboeck, Florian; Pizka, Markus:** *Demystifying Maintainability.* In: Proceedings of the 2006 international Workshop on Software Quality (WoSQ '06), Shanghai, China, 21. Mai 2006. New York, NY : ACM, 2006, S. 21-26, 1-59593-399-9

[Boe81] **Boehm, Barry W.:** *Software Engineering Economics.* 1st ed. Englewood Cliffs, N.J. : Prentice-Hall, 1981. 0138221227

[Chl06] **Chlebek, Paul:** *User Interface-orientierte Softwarearchitektur.* Wiesbaden : Vieweg Verlag, 2006. 3-8348-0162-3

[CKK02] **Clements, Paul; Kazman, Rick; Klein, Mark:** *Evaluating software architectures.* Boston : Addison-Wesley, 2002. 0-201-70482-X

[DIN9241] *Ergonomie der Mensch-System-Interaktion – Teil 110: Grundsätze der Dialoggestaltung (ISO 9241-110:2006); Deutsche Fassung EN ISO 9241-110:2006.* DIN Deutsches Institut für Normung e.V., Berlin, März 2007.

[FKR07] **Fiedler, Patrique; Klinger, David und Rühle, Johannes:** *Pflichtenheft Lieferantenverzeichnis.* Softwareprojekt Ingenieurinformatik, TU Ilmenau. Ilmenau, 2007.

[Fow05] **Fowler, Martin:** *UML distilled: a brief guide to the standard object modeling language.* 3. ed. Boston, MA : Addison-Wesley, 2005. 0-321-19368-7

[Gal02] **Galitz, Wilbert O.:** *The essential guide to user interface design: an introduction to GUI design principles and techniques.* 2nd ed. New York : Wiley, 2002. 0-471-08464-6

[GF94] **Gotel, O. C. Z. und Finkelstein, A. C. W.:** *An analysis of the requirements traceability problem.* In: First International Conference on Requirements Engineering (ICRE'94), S. 94-101, Colorado Springs, Apr. 1994. IEEE Computer Society Press.

[GGL01] **Gulliksen, Jan; Göransson, Bengt und Lif, Magnus:** *A User-Centered Approach to Object-Oriented User Interface Design.* In: van Harmelen, Mark (Ed.): Object Modeling and User Interface Design. Boston, Mass. : Addison-Wesley, 2001, 8, S. 283-312.

[GHJV95] **Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John:** *Design patterns: elements of reusable object-oriented software.* Boston, MA : Addison-Wesley Longman Publishing Co., Inc., 1995. 0-201-63361-2

[Herc94] **Herczeg, Michael:** *Software-Ergonomie: Grundlagen der Mensch-Computer-Kommunikation.* Bonn; Paris; Reading, Mass. : Addison-Wesley, 1994. 3-89319-615-3

[Herp07] **Herpel, Kristian:** *Refactoring und Identifikation von Komponenten.* Fakultät für Informatik und Automatisierung - Fachgebiet Softwaresysteme / Prozessinformatik, TU Ilmenau. Ilmenau, 2007. Diplomarbeit. Inventarisierungsnummer: 2007-01-17/029/IN97/2232

[HHS04] **Haft, Martin; Humm, Bernhard und Siedersleben, Johannes:** *Quasar Reference Interfaces for Business Information Systems.* sd&m Research. München, 2004.

[HHS05] **Haft, Martin; Humm, Bernhard und Siedersleben, Johannes:** *The Architect's Dilemma – Will Reference Architectures Help?* In: R. Reussner et al. (Eds.): Quality of Software Architectures and Software Quality (QoSA-SOQUA 2005). Berlin, Heidelberg : Springer-Verlag, Lecture Notes in Computer Science 3712, 2005, S. 106-122

[HO07] **Haft, Martin und Olleck, Bernd:** Komponentenbasierte Client-Architektur. *Informatik-Spektrum.* 24. Juni 2007, Volume 30 Issue 3, S. 143-158.

[KAB01] **Kruchten, Philippe; Ahlqvist, Stefan und Bylund, Stefan:** User Interface Design in the Rational Unified Process. In: van Harmelen, Mark (Ed.): Object Modeling an User Interface Design. Boston, Mass. : Addison-Wesley, 2002, 5, S. 161-196.

[Lau05] **Lauesen, Soren:** *User interface design: a software engineering perspective.* Harlow : Pearson/Addison-Wesley, 2005. 0-321-18143-3

[LC06] **Leavons, Gary T. und Cheon, Yoonsik:** *Design by Contract with JML.* [Online] 28. September 2006. [Zitat vom: 8. Januar 2008] ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf

[Len07] **Lentzsch, Karsten:** *JGoodies: First Aid for Swing UIs.* [Online] 26. April 2007. [Zitat vom: 8. Januar 2008.] http://www.jgoodies.com/articles/first%20aid%20for%20swing-75min_de.pdf

[LOGS01] **Lif, Magnus; Olsson, Eva; Gulliksen, Jan; Sandblad, Bengt:** Workspaces Enhance Efficiency - Theories, Concepts and a Case Study. *Information Technology & People.* 2001, Volume 14 Number 3, S. 261-272.

[Mey97] **Meyer, Bertrand:** *Object-oriented software construction.* 2nd ed. Upper Saddle River, NJ, USA : Prentice Hall, 1997. 0-13-629155-4

[MPR07] **Mäder, Patrick; Philippow, Ilka; Riebisch, Matthias:** *Customizing Traceability Links for the Unified Process.* In: Proceedings Third International Conference on the Quality of Software-Architectures (QOSA2007), Medford MA, USA, 12.-13. Juli, 2007. Springer: LNCS, 2007 (post-conference proceedings in press)

[MRP06] **Mäder, Patrick; Riebisch, Matthias und Philippow, Ilka:** *Traceability for Managing Evolutionary Change - A Roadmap.* In: Proceedings 15th International Conference on Software Engineering and Data Engineering (SEDE-2006), 6.-8. Juli, 2006, Los Angeles, California, USA. International Society for Computers and their Applications, 2006, S. 1-8.

[PBG04] **Posch, Torsten; Birken, Klaus und Gerdom, Michael:** *Basiswissen Softwarearchitektur: verstehen, entwerfen, bewerten und dokumentieren.* Heidelberg : dpunkt.verlag, 2004. 3-89864-270-4

[Rie04] **Riebisch, Matthias:** *Unterstützung evolutionärer Softwareentwicklung durch Merkmalmodelle und Traceability-Links.* In: Proceedings Modellierung 2004 - Praktischer Einsatz von Modellen. 23.-26. März 2004, Marburg, Germany. LNI, Gesellschaft für Informatik, 2004, S. 315-316

[RJ01] **Ramesh, Bala und Jarke, Matthias:** *Toward reference models of requirements traceability.* IEEE Transactions Software Engineering, 27(1), S. 58-93, 2001.

[RR99] **Robertson, Suzanne und Robertson, James:** *Mastering the requirements process.* Harlow : Addison-Wesley, 1999. 0-201-36046-2

[RUP06] *Rational Unified Process Version 7.0.1.E.* [CD] : IBM Corporation, 1987, 2006.

[Shn92] **Shneiderman, Ben:** *Designing the user interface: strategies for effective human-computer interaction.* 2nd ed. Reading, Mass. : Addison Wesley, 1992. 0-201-57286-9

[Sie03] **Siedersleben, Johannes (Hrsg.):** *Softwaretechnik: Praxiswissen für Software-Ingenieure.* München : Hanser, 2003. 3-446-21843-2

[Sie03a] **Siedersleben, Johannes (Hrsg.):** *Quasar: Die sd&m Standardarchitektur Teil 1.* sd&m Research. München, 2003.

[Sie03b] **Siedersleben, Johannes (Hrsg.):** *Quasar: Die sd&m Standardarchitektur Teil 2.* sd&m Research. München, 2003.

[Sie04] **Siedersleben, Johannes:** *Moderne Software-Architektur: umsichtig planen, robust bauen mit Quasar.* Heidelberg : dpunkt.verlag, 2004. 3-89864-292-5

[SSKK02] **Stutz, Christiane; Siedersleben, Johannes; Kretschmer, Dörthe; Krug, Wolfgang:** *Analysis beyond UML.* In: Proceedings IEEE Joint International Conference on Requirements Engineering (RE 2002). 9.-13. September, 2002, Essen, Germany. IEEE Computer Society, S. 215-218.

[Szy02] **Szyperski, Clemens:** *Component software: beyond object-oriented programming.* London : Addison-Wesley, 2002. 0-201-74572-0

[vHa01] **van Harmelen, Mark (Ed.):** *Object Modeling and User Interface Design.* Boston, Mass : Addison-Wesley, 2001. 0-201-65789-9

[VN98] **Voss, Josef und Nentwig, Dietmar:** *Entwicklung von graphischen Benutzungsschnittstellen: Modelle, Techniken und Werkzeuge der User-Interface-Gestaltung.* München; Wien : Carl Hanser Verlag, 1998. 3-446-19089-9

C Abbildungsverzeichnis

D Tabellenverzeichnis

E Listingverzeichnis

F Thesen

1. Durch die Erstellung von Traceability-Links werden Entwurfsentscheidungen nachvollziehbar.
2. Die vollständige Unterstützung durch Traceability bei der Entwicklung eines Softwaresystems ist nur für Architektur- bzw. Entwicklungsmethoden möglich, deren Aktivitäten lückenlos definiert sind.
3. Quasar ermöglicht die konsequente Trennung von Anwendung und Technik und unterstützt Separation of Concerns.
4. Die Softwarekategorien von Quasar legen Abhängigkeiten im Softwaresystem offen und unterstützen die Identifikation von Komponenten.
5. Die Verwendung eines Funktionsbaumes als Zwischenschritt in der Quasar-Methode erleichtert die Erstellung des Kategorien- und Komponentenmodells.
6. Die Schnittstellen der Komponenten sollten nach dem Design-by-Contract-Ansatz spezifiziert werden.
7. Die Definition von Standardschnittstellen für eine Client-Architektur wie bei Quasar verbessert die Wiederverwendbarkeit.
8. Use Case Storyboards helfen, die Anforderungen an das Benutzerschnittstellendesign festzulegen und die Dialogkomponenten zu finden.
9. Die Ausrichtung des Designs der Benutzeroberfläche nach Richtlinien der Software-Ergonomie verbessert die Usability der Anwendung.
10. Ansätze aus dem Bereich Human-Computer Interaction (HCI) können auch in andere Architekturmethoden wie Quasar integriert werden.

_____

Ilmenau, den 23. 01. 2008 Stephan Bode

G Eidesstattliche Erklärung

Ich erkläre an Eides Statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Ich versichere, dass ich dieses Diplomarbeitsthema bisher weder im In- noch im Ausland in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

_____

Ilmenau, den 23. 01. 2008 Stephan Bode