Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Markus Alexander Kuppe

Komponentenbasierte Software-Entwicklung im Rahmen einer Konsolidierung am Beispiel von Eclipse

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Markus Alexander Kuppe

Komponentenbasierte Software-Entwicklung im Rahmen einer Konsolidierung am Beispiel von Eclipse

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Olaf Zukunft
Zweitgutachter: Prof. Dr. rer. nat. Jörg Raasch

Abgegeben am 18.01.2007

Markus Alexander Kuppe

Thema der Bachelorarbeit
Komponentenbasierte Software-Entwicklung im Rahmen einer Konsolidierung am Beispiel von Eclipse
Stichworte
Java, Eclipse, OSGi, Komponente, CBSE, Konsolidierung, Komponentifizierung
Kurzzusammenfassung
Diese Arbeit befasst sich mit der Konsolidierung einer Bestands-Software unter dem Gesichtspunkt der komponentenbasierten Software-Entwicklung (CBSE). Zu Beginn werden deshalb die Grundlagen einer Konsolidierung und des CBSE erläutert, bevor ein Konzept zur Umwandlung einer Bestands-Software anhand eines Prototyps dargestellt wird. Dieser Prozess umfasst die Wiedergewinnung von Anwendungswissen, ein Herauslösen von Komponenten und deren Integration in ein Komponentenmodell. Die technische Implementierung erfolgt mit Eclipse und dem OSGi-Standard, die mit dem CBSE in Bezug gesetzt werden.
Die Anwendung des CBSE als Entwicklungsparadigma auf Bestands-Software ist, neben der Darlegung von dessen Nutzen, Kern dieser Arbeit.

Markus Alexander Kuppe

Title of the paper
Component-based software engineering applied to a consolidation using Eclipse
Keywords
Java, Eclipse, OSGi, Component, CBSE, Consolidation, Componentization
Abstract
This bachelor thesis deals with the consolidation of a pre-existing software application framed in the context of Component-based Software Engineering (CBSE). The aspects associated with software consolidation and the basic concepts behind CBSE are explained. A process for the conversion of an existing software application in the form of a prototype is then presented. This process covers the discovery of application knowledge in order to effectively extract components and integrate these into a component model. The implementation is accomplished with Eclipse and the OSGi standard, using both in correlation with CBSE.
The primary principles guiding this thesis are the presentation of the advantages provided by CBSE and the application of the CBSE paradigm to a consolidation.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Heutige Software-Entwicklungsprojekte überschreiten regelmäßig Zeitvorgaben, Budgetgrenzen und entsprechen nicht den Anforderungen. Sie scheitern entweder vollauf oder sind unwirtschaftlich. Unklare Ziele und wechselnde Anforderungen machen es den Projektteilnehmern schwer. Diese Misere liegt aber nicht in der Verantwortung der Entwickler. Welcher Entwickler hat sich nicht bereits die Frage gestellt, ob das aktuelle Problem nicht schon von jemand anderem gelöst wurde und ob diese Lösung nicht wiederverwendet werden könnte. Eine erneute Nutzung schlägt wegen Integrationsschwierigkeiten jedoch fehl oder ist zumindest ungeplant und unsystematisch.
In einer ähnlichen Situation befand sich die Automobilindustrie zu Beginn des 19. Jahrhunderts. Jedes Fahrzeug wurde handgefertigt, jedes Bauteil einzeln hergestellt und eingepasst. Die Herstellung ähnelte eher einer Kunst als einer Industrie. Die Preise für ein Auto waren astronomisch und der Bau benötigte viele Arbeitsstunden. Dieser Umstand änderte sich mit der Umstellung auf die Fließbandfabrikation durch Henry Ford, in deren Folge vorgefertigte Bauteile montiert wurden. Das Ergebnis waren sinkende Produktionspreise bei steigenden Verkaufszahlen.
Die komponentenbasierte Software-Entwicklung versucht, die Erkenntnisse der Automobilindustrie auf die Software-Technik anzuwenden. Vorgefertigte Bauteile – Komponenten – werden bei Zulieferern eingekauft und zu Software-Systemen zusammengesetzt. Ändern sich die Anforderungen an das Endprodukt, kann die Zusammensetzung verändert oder einzelne Komponenten ausgetauscht werden. Die Systeme sind flexibler in Bezug auf wechselnde Anforderungen und können billiger entwickelt werden.
Ob sich die komponentenbasierte Software-Entwicklung auf bestehende Software-Systeme anwenden lässt, um ihre Vorteile auszunutzen, ist weitgehend ungeklärt.

Der Verfasser beschäftigt sich seit einigen Jahren mit Eclipse[1], zuletzt als offizieller Entwickler des Dali-Projekts der Eclipse-Foundation. Bei diesen Arbeiten ist das Interesse des Autors an der komponentenbasierten Software-Entwicklung geweckt worden. Die Idee zu dieser Bachelorarbeit kam ihm, als die Versant Corp. eine Möglichkeit zur Erneuerung bestehender Anwendungen unter der Einschränkung von begrenzten Ressourcen suchte.

Die Versant Corp. entwickelt und vertreibt seit 1988 objektorientierte Datenbanksysteme. Zum Kundenkreis, des an der NASDAQ gelisteten Unternehmens, zählen unter anderem Firmen wie AT&T, Siemens AG, NEC oder ESA. Zum Zeitpunkt der Arbeit kommen Versant-Systeme weltweit in mehr als 50.000 Installationen zum Einsatz.
Im Jahr 2001 wurde die 1991 gegründete POET GmbH durch die Versant Corp. übernommen. Heute bietet Versant zwei Datenbanksysteme als Produktlinie an: das Versant Objekt-Datenbanksystem und das FastObjects Datenbank-System. Das Angebot wird durch verschiedene Application Programming Interfaces und Werkzeuge ergänzt, die einen Zugriff aus Java, C++ und C auf die Datenbank-Systeme erlauben. Die Produkte werden von rund 100 Mitarbeitern betreut, die an den drei Standorten in Fremont (U.S.A), Pune (Indien) und Hamburg (Deutschland) arbeiten. Diese Arbeit entstand im Hamburger Entwicklungszentrum, dem Hauptsitz der Firma.

1.2 Aufgabe und Ziel

Diese Arbeit untersucht die Anwendung der komponentenbasierten Software-Entwicklung im Zuge einer Konsolidierung auf ein bestehendes Software-System, um dessen Vorteile auszunutzen. Zu diesem Zweck soll ein Konzept entwickelt werden, das eine Identifikation von Komponentenkandidaten in einem Altsystem liefert, eine Dekomposition dieser Kandidaten aus dem Altsystem in Komponenten erlaubt, um die gewonnenen Komponenten anschließend innerhalb eines Komponentenmodells zu integrieren. Im Anschluss an die Anwendung dieses Konzepts soll das Altsystem komponentifiziert sein. Dieses Konzept wird auf der Unterstützung von Werkzeugen basieren. Die Überprüfung des Konzepts erfolgt anhand eines Altsystems, das als Prototyp dient. Die empirische Auswertung soll quantifizierbare Ergebnisse in Bezug auf das Konzept als auch den Nutzen der komponentenbasierten Software-Entwicklung liefern.

Den Rahmen der Arbeit bildet das Vitness-Konsolidierungsprojekt der Versant Corp. (Versant) Ziel ist die Konsolidierung einer Altanwendung mit Eclipse. Innerhalb der ersten Stufe dieses Projekts soll geprüft werden, ob sich Eclipse als Ziel einer Konsolidierung und als Komponentenmodell eignet. Ist das oben beschriebene Konzept zur Konsolidierung tauglich

[1] Eine Abgrenzung dieses Begriffs findet sich in Kapitel 3.

und kann der Prototyp damit erfolgreich konsolidiert werden, wird Versant das Vitness-Projekt auf weitere Altanwendungen ausdehnen. Diese Arbeit soll demnach Aussagen über die Nützlichkeit von Eclipse als Komponentenmodell und als Ziel einer Konsolidierung liefern.
Die Vorgehensweise dieser Arbeit zielt darauf ab, die komponentenbasierte Software-Entwicklung in ihrer Breite darzustellen. Dazu werden Begriffe wie Komponenten, Komponentenmodell, Komponenten-Rahmenwerk und Komponentenmarkt eingeführt, die während der einführenden Recherchen zu dieser Arbeit erarbeitet wurden. Die Konzepte der komponentenbasierten Software-Entwicklung werden danach mittels einer Konsolidierung des Prototyps auf Eclipse vertieft. Eine Einarbeitung in die Funktionsweise und die Architektur des Prototyps ist für diese Konsolidierung notwendig.

1.3 Gliederung der Arbeit

Diese Arbeit ist dazu ausgelegt, in einem Stück gelesen zu werden. Auf diese Weise wird der Leser in sechs aufeinander aufbauenden Kapiteln den einfachsten Einblick in diese Arbeit bekommen.
Im Anschluss an diese Einleitung wird in einem Grundlagenteil das nötige Hintergrundwissen zum Verständnis dieser Arbeit geliefert (2. Kapitel). Darauf folgt der Vergleich zwischen Eclipse und der komponentenbasierten Software-Entwicklung (3. Kapitel). In der folgenden Anforderungsanalyse sind der Ist- sowie Soll-Zustand des Prototyps zusammengefasst (4. Kapitel), die zum Verständnis der Umsetzung der Hypothese erforderlich sind (5. Kapitel). Das abschließende Kapitel beinhaltet eine Auswertung und Zusammenfassung der Resultate und gibt einen Ausblick auf die an diese Arbeit anschließenden Arbeiten (6. Kapitel).
Die praktischen Arbeiten, von denen der schriftliche Teil der Arbeit abstrahiert, befinden sich auf der beiliegenden CD-Rom (Anhang C). Der dortige Prototyp gilt als Dokumentation dieser Arbeiten.

Kapitel 2

Grundlagen

Die folgenden Grundlagen gliedern sich in drei Abschnitte. In diesen Abschnitten wird eine Definition des Konsolidierungsbegriffs gegeben (Abschnitt 2.1), eine ausführliche Erläuterung der komponentenbasierten Software-Entwicklung (Abschnitt 2.2) und eine Zusammenführung dieser beiden Begriffe (Abschnitt 2.3). Die Zusammenführung beschreibt dabei die Arbeiten, die bei einer Dekomposition einer Bestandsanwendung auf ein Komponentenmodell nötig werden.

2.1 Konsolidierung anstatt Neuentwicklung

Der folgende Abschnitt hat das Ziel, dem Leser den Begriff der Konsolidierung näher zu bringen, indem die Dimensionen und Eigenschaften einer Konsolidierung dargelegt werden. Am Ende des Abschnitts soll ein Verständnis entstanden sein, was Konsolidierung in dieser Arbeit bedeutet und inwieweit sie sich von der Software-Neuentwicklung unterscheidet.

2.1.1 Software-Konsolidierung

Um in dieser Arbeit sinnvoll mit dem Begriff der Konsolidierung arbeiten zu können, ist ein grundsätzliches Verständnis über die Dimensionen des Begriffs nötig. Konsolidierung ist in verschiedenen Fachbereichen mit unterschiedlichen, aber auch überschneidenden Bedeutungen belegt. Es lässt sich beispielsweise eine wirtschaftliche Erklärung wiedergeben, wonach Konsolidierung „[...] einen Prozess der Kostensenkung durch Streichung unprofitabler Bereiche innerhalb der Betriebswirtschaft bezeichnet. Dies kann auch viele Betriebe in einem Marktsegment betreffen, in dem insgesamt Überkapazitäten bestehen“ [Wikipedia 2006a].
Da eine wirtschaftliche Auslegung in dieser Arbeit nicht passend ist, soll damit begonnen werden, den Begriff der Konsolidierung für die Software-Entwicklung zu verfestigen. Eine vollständige und umfassende Definition ist aber nicht möglich und für diese Arbeit auch un-

nötig. Deshalb soll der Begriff nur im Sinne einer partiellen Definition für einen bestimmten Bereich umrissen werden.

2.1.1.1 Kontext der Konsolidierung

Konsolidierung bezieht sich im Rahmen dieser Arbeit ausschließlich auf die Software-Technik, die ein Fachgebiet innerhalb der Informatik darstellt. Es wird also explizit nicht die Konsolidierung von Hardware-Ressourcen wie Servern verstanden, die in aktuellen Fachartikeln der Informatik häufig Thema der Diskussion sind. Die Konsolidierung von Anwendungen innerhalb eines Betriebes, deren Funktionalität nach Jahren des betrieblichen Wachstums teilweise obsolet und veraltet sind, ist ebenfalls nicht gemeint. Jedoch können die Ursachen für solche Konsolidierungen ähnlich bis identisch mit den Gründen der hier thematisierten Software-Konsolidierung sein (vgl. Abschnitt 2.1.2).

2.1.1.2 Gegenstand der Konsolidierung

Beim Gegenstand der Konsolidierung, nachfolgend Altsystem oder Bestandsanwendung genannt, handelt es sich um ein Produkt, das innerhalb eines Software-Entwicklungsprozesses entstanden ist. Die Betrachtung erfolgt aus der Perspektive des Altsystemherstellers. Die Lebensdauer des Altsystems erstreckt sich gegebenenfalls über mehrere Jahre, in denen unter Umständen verschiedene Produktversionen des Altsystems veröffentlicht wurden.

2.1.1.3 Teilbereiche der Konsolidierung

Nachdem der Kontext sowie der Gegenstand der Konsolidierung abgesteckt sind, können einzelne Teilbereiche der Konsolidierung eines Altsystems beleuchtet werden. Diese können auch als Arbeitsschritte innerhalb der Konsolidierung verstanden werden. Zumindest handelt es sich aber um Begriffe, die in der Software-Technik eine feste Bedeutung haben oder für die eine klare Beschreibung existiert und hier wiedergegeben werden kann.
Die folgende Aufzählung trägt eine vermeintliche, zeitliche Ordnung, man könnte also auch von Phasen[2] der Software-Konsolidierung sprechen.

- Das ***Reengineering*** beschäftigt sich mit der Gewinnung von Informationen über das Altsystem. Unter Informationen sind in diesem Zusammenhang die Spezifikation und Dokumentation eines Software-Systems zu verstehen. Inhalte der Spezifikation und Dokumentation sind beispielsweise die fachliche und technische Systemarchitektur, Funktionsumfang, algorithmische Abläufe oder Details der Implementierung.
  In [Chikofsky u. II 1990] wird Reengineering als "[...] the examination and alteration

[2] Phasen erheben vor dem Hintergrund der Kenntnis der Schwächen des Wasserfall-Modells keinen Anspruch auf zeitliche Richtigkeit [Beck 2004]

of a subject system to reconstitute it in a new form and the subsequent implementation of the new form [...]" definiert. Somit umfasst Reengineering das Untersuchen und Verändern des Altsystems, um es in eine neue Struktur zu transformieren. Diese Definition beinhaltet die Veränderung des Altsystems, was das folgende Refactoring konkretisiert.

- ***Refactorings*** und ***Big Refactorings*** verändern die interne Struktur des Altsystems, um das System verständlicher zu gestalten und die Wartung zu erleichtern. Dabei wird das beobachtete Verhalten des Systems gleichzeitig aber nicht verändert. Die Änderung bewegen sich auf Klassen- oder Methodenebene, manchmal aber auch auf der Vererbungsebene in objektorientierten Konstruktionen [Fowler u. a. 2004; Roock u. Lippert 2004].

- Während die Refactorings das Verhalten des Altsystems ausdrücklich nicht verändern sollen, ist dieses bei der ***Migration*** üblich. Änderungen an der technischen Gesamtarchitektur, wie das Portieren einer Anwendung von einer Mainframe-Architektur auf eine Client-Server-Architektur, verändern das Verhalten der Anwendung zumindest in technischer[3] Hinsicht. Die Migration ist eine Veränderung des Altsystems oberhalb der Implementierungsebene und daher die dritte Komponente im Rahmen der Konsolidierung.

- Bisher wurden lediglich erhaltende Arbeiten der Konsolidierung erwähnt, die keine neuen fachlichen Funktionalitäten in die Anwendung einführen. Das würde den Tätigkeiten der Software-Konsolidierung allerdings nicht gerecht werden, denn eine Weiterentwicklung – eine ***Evolution*** – des Bestandsystems ist ebenfalls Teil der Konsolidierung. Unter Evolution werden alle Neuerungen des Bestandssystems zusammengefasst, die über die ursprüngliche Funktionalität des Systems hinausgehen und es erweitern. [Software-Evolution-Group 2003] definiert (Software-) Evolution als: „[...] the set of activities, both technical and managerial, that ensures that software continues to meet organisational and business objectives in a cost effective way." Vereinfachend ausgedrückt, beinhaltet Evolution demnach die technische und fachliche Modifikation von Systemen, um sie an die (wechselnden) Anforderungen des Markts anzupassen. Diese Definition umfasst somit einen wirtschaftlichen Aspekt, der von den anderen Teilbereichen bisher nicht beachtet wurde.

Die Summe der obigen Teilbereiche und die folgende Definition aus [Fischer 2002], wonach Konsolidierung die „Elimination von Widersprüchen und Redundanzen" ist, stellt die Bedeutung der Konsolidierung umfassend dar: (Software-) Konsolidierung ist die systematische Analyse und Modifikation der fachlichen und technischen Architektur eines oder mehrerer

[3] Eine fachliche Veränderung versucht man jedoch meist zu vermeiden.

Systeme mit dem Ziel, Widersprüche und Redundanzen innerhalb der Systeme zu beseitigen und sie (veränderten) wirtschaftlichen Anforderungen anzupassen.

2.1.2 Gründe für eine Konsolidierung

Nachdem im vorherigen Abschnitt die Bedeutung der Konsolidierung hergeleitet wurde, sollen in diesem Teil die Gründe erläutert werden, die eine Konsolidierung nötig werden lassen. Jedoch machen Gründe einzelner Teilbereiche eine Konsolidierung nicht zwangsläufig erforderlich (vgl. Abschnitt 2.1.1.3). Eine Konsolidierung wird erst dann nötig, wenn Gründe aller Teilbereich auftreten.
Die Liste der Gründe, die als Indikatoren für eine Konsolidierung anzusehen sind, erhebt keinen Anspruch auf Vollständigkeit. Trotzdem wurden die Gründe nach technischen und wirtschaftlichen Gesichtspunkte sortiert, um einen besseren Überblick zu erzielen.

2.1.2.1 Technische Gründe

Technische Gründe, die für eine Software-Konsolidierung sprechen, lassen sich wiederum als erweiternd und erhaltend charakterisieren.
Unter erweiternden Gründen versteht man das Hinzufügen neuer Funktionalitäten, um Anforderungen der Kunden und Nutzern der Anwendungen umzusetzen. So ist die Unterstützung von neuen Hardware-Plattformen eine Erweiterung der Anwendung. Neue Funktionalität wird aber nicht ausschließlich zur Zufriedenstellung des Kunden oder zur Neukunden-Gewinnung in die Anwendung integriert, sondern auch, um von technischen Innovationen Nutzen zu ziehen und damit die Weiterentwicklung zu beschleunigen. Technische Innovationen sind Neuentwicklungen von Technologien, Konzepten, Prozessen und ganz generell Fortschritten innerhalb der Informatik.
Die Hauptarbeit im heutigen Software-Entwicklungsprozess entfällt aber auf Pflege und Wartung von Bestandsanwendung.[4] Diese erhaltenden Maßnahmen sind weitere Gründe, die eine Konsolidierung einer Bestandsanwendung rechtfertigen.

2.1.2.2 Wirtschaftliche Gründe

Eine Konsolidierung wird aus wirtschaftlichen Gründen sinnvoll, wenn ein Schwellwert für die Pflege und Wartung des Altsystems überschritten wird. Dieser Schwellwert wird erreicht, wenn die Wartungskosten die Aufwände einer Konsolidierung übersteigen. Andere Szenarien, die eine vertriebliche Sichtweise darstellen, sind rückläufige Verkaufszahlen der Anwendung oder unzufriedene Kunden. In diesem Fall kann eine Konsolidierung die Anwendung attraktiver gestalten oder die Akzeptanz beim Kunden erhöhen.

[4] 60% bis 80% des Aufwands im Software-Entwicklungsprozess werden auf Software-Wartung verwendet [Nosek u. Palvia 1990]

Ein weiterer organisationsinterner Faktor für eine Konsolidierung entsteht durch Restrukturierungen der Organisationsformen und wechselnden Prozessen im Entwicklungsprozess. Eine Verschmelzung von Produktlinien, die durch den Zusammenschluss verschiedener Firmen notwendig wurde, kann hierfür als Beispiel angeführt werden.
Konsolidierungen werden somit durch organisationsinterne Prozesse verursacht und entstehen nicht (direkt) auf Grund von externen Auslösern.

2.1.3 Probleme bei der Konsolidierung

Die möglichen Probleme, die den Erfolg einer Konsolidierung gefährden können, sind vielschichtig. In dieser Arbeit wird deshalb der Versuch unternommen, die unterschiedlichen Probleme der Teilbereichen der Konsolidierung in drei Kategorien zu verdichten (siehe Abschnitt 2.1.1.3).

- Unzureichendes Wissen über das Altsystem: Das Wissen über die fachliche und technische Architektur der Bestandsanwendung ist nicht umfassend vorhanden. Manche Aspekte der Bestandsanwendung sind beispielsweise auf Grund von fehlender Dokumentation, der Abwanderung der ursprünglichen Entwickler oder fehlendem Quelltext unbekannt. In diesem Fall muss diese Wissenslücke durch Reengineering geschlossen werden (vgl. Abschnitt 2.1.1.3).

- Unvereinbarkeit der Paradigmen des Alt- und Neusystems: Ein Wechsel der Paradigmen zwischen den Alt- und Neusystemen führt zu Unvereinbarkeiten, infolge derer eine Abbildung der Funktionalität des Altsystems innerhalb des Neusystems nicht möglich ist. So besteht zwischen relationalen Datenbanksystemen und objektorientierter Programmierung ein „impedance mismatch", wonach konzeptionelle und technische Schwierigkeiten in der Abbildung von Klassen und Objekte auf Relationen im relationalen Modell bestehen.

- Wiedererreichung der Qualität: Das Neusystem erreicht nicht dieselbe Qualität in Bezug auf die Qualitätsmerkmale des Altsystems (vgl. [Balzert 1998, S. 256ff]). Ein Qualitätsverlust tritt ein. Qualitätsmerkmale sind die Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit und Übertragbarkeit eines Softwaresystems.

2.2 Komponentenbasierte Software-Entwicklung

Nachdem die Konsolidierung hinreichend eingegrenzt ist, soll im Folgenden die komponentenbasierte Software-Entwicklung („Component-based Software Engineering") (CBSE) dargestellt werden. Im Gegensatz zur Konsolidierung bietet die Literatur des CBSE diverse Definitionen und Erläuterungen, die zu einer Definition zusammengestellt werden können. Diese

Definition basiert grundsätzlich auf [Atkinson 2002; Atkinson u. a. 2003; Cheesman 2000; Heineman 2001; McIlroy 1968; Schmidt 2001; Szyperski 2002].

2.2.1 Komponentenbasierte Software-Entwicklung

„Components aren't rocket science" [Cheesman 2000, S. 1]: Erklärt man einem Nicht-Informatiker die grundlegenden Konzepte des CBSE, werden sie ihn kaum beeindrucken. Andere Ingenieursdisziplinen nutzen dieselben Konzepte seit Jahren. Zwar wurde das CBSE bereits 1968 von M.D. McIlroy in Form von „software integrated circuits" postuliert, um die anhaltende Krise der Software-Industrie zu lösen [McIlroy 1968; Brooks 1995; Heineman 2001, S. xxii]. Ungeachtet dessen lässt eine Umsetzung des CBSE, die alle Aspekte beinhaltet, weiterhin auf sich warten [Szyperski 2002].
Im Gegensatz zum klassischen Software-Engineering ist das CBSE nicht auf die Lösung eines spezifischen Problems durch eine Sonderanfertigung ausgerichtet, sondern ähnelt eher einer Fabrik, in der Maschinen nach dem Baukastenprinzip zusammengesetzt werden: „CBSE is an assembly-based product delivery system." [Heineman 2001, S. 674]. So ist CBSE eine Subdisziplin des Software-Engineering, dessen Wurzeln in der industriellen Produktion liegen [Heineman 2001, S. 1].

Wiederverwendung: Die vorherige Analogie zum Baukastenprinzip impliziert, dass identische Teile – die Komponenten – mehrfach Wiederverwendet werden. Laut [McIlroy 1968] ist diese Wiederverwendung das zentrale Ziel des CBSE. [Schmidt 2001] sieht Komponenten und deren Wiederverwendung sogar als die zwei Seiten einer Medaille.
Wie eingangs erwähnt, ist Wiederverwendung kein verblüffendes Konzept. Im Entwicklungsalltag werden Ideen, Methoden, Algorithmen und Quelltexte[5] permanent wiederverwendet. Man spricht auch von ad hoc-Wiederverwendung. Diese Wiederverwendung ist allerdings unstrukturiert und nur innerhalb eines kleinen Rahmens umsetzbar. Wiederverwendung im CBSE verfolgt dagegen einen systematischen Ansatz und skaliert beliebig.
Software-Lösungen, beziehungsweise deren technische Realisierung als Quelltext, werden in Komponenten gekapselt, um in einem neuen Kontext ohne Anpassung (vgl. Abschnitt 2.3.4.1) erneut nutzbar zu sein.[6] Die Wartung und Pflege dieser Komponenten liegt nicht in der Verantwortung des Komponentennutzers, sondern in der des Komponentenherstellers.

Flexibilität: Obgleich Wiederverwendung als Hauptmerkmal des CBSE gewertet wird, ist die entstehende Flexibilität ein ebenso wichtiges Kennzeichen [Heineman 2001, S. 502], [Cheesman 2000, S. 2]. Die Konstruktion von Anwendungen unter sich ständig ändernden Anforderungen erfordert ein Höchstmaß an Flexibilität [Hunt u. Thomas 2005]. Im CBSE ist

[5] Quelltexte werden in bestehenden Systemen kopiert. Eine Kopie birgt allerdings Redundanzprobleme.
[6] In Bezug auf Computer-Hardware ist ein ähnliches Konzept als „Plug and Play" bekannt.

ein Austausch von Komponenten und eine veränderte Zusammensetzung von Komponenten durchgehend möglich. Dies wird nicht nur durch einen modularen Aufbau der Anwendung erreicht, sondern auch durch die Reduzierung der Komplexität (Entropie) des Software-Entwicklungsprozesses [Schmidt 2001]. In [Cheesman 2000] wird dieser Sachverhalt als „divide and conquer" bezeichnet. Dies meint die Aufspaltung von Problemen, um sie handhabbar und lösbar zu machen.

Bevor in den kommenden Abschnitten die Merkmale des CBSE inhaltlich adressiert werden, stellt die folgende Abbildung diesen Sachverhalt in schematischer Form einleitend dar. Die Begriffe des Diagramms werden in den über die Nummerierung verwiesenen Abschnitten erläutert.

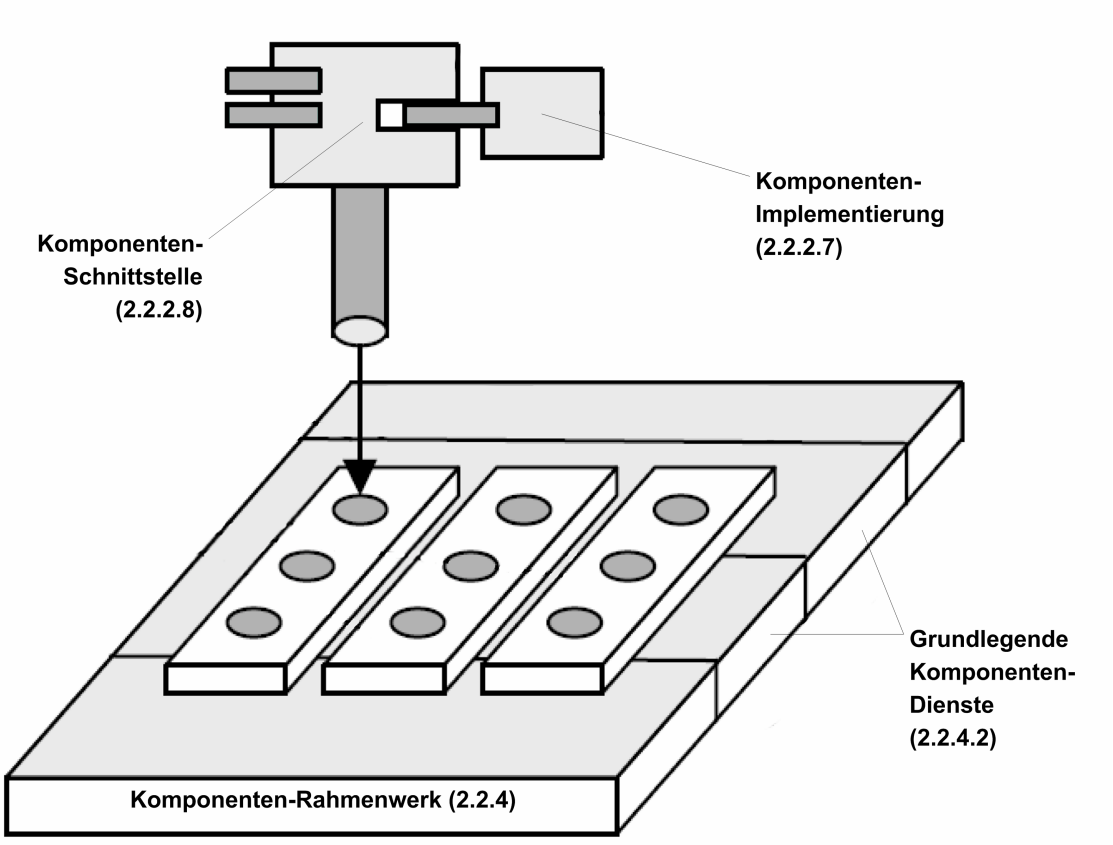

Abbildung 2.1: Schematische Darstellung eines Komponentenmodells

2.2.2 Komponente

[Szyperski 2002, S. 41] definiert (Software-) Komponenten als: „[...] a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties".
Eine Software-Komponente ist demnach eine Einheit mit wohl definierten Schnittstellen die in Abhängigkeit zu einem Kontext steht. In diesem Kontext setzen Dritte verschiedene unabhängige Komponenten zu einer Komposition zusammen (vgl. [Gamma u. a. 1995]). Ein weiterer Aspekt, der aus dieser Definition nicht deutlich wird, ist das Ziel der Wiederverwendbarkeit von Komponenten in unterschiedlichen Kontexten (vgl. Abschnitt 2.2.1). Gerade dieser zusätzliche Aspekt begründet die geforderte Unabhängigkeit von Komponenten.

2.2.2.1 Unabhängigkeit von Komponenten

Um Komponenten in verschiedenen Kontexten einsetzen zu können, darf die Komponente keine Annahmen über den Kontext machen. Annahmen verursachen Abhängigkeiten zum Kontext (vgl. Abschnitt 2.2.2.1). Daneben lässt sich die Forderung nach Unabhängigkeit auch mit der Möglichkeit zur Komposition von Komponenten begründen. Soll eine Komponente mit anderen Komponenten innerhalb einer Komposition interagieren, so darf die Komponente keine Annahmen über diese Komponenten machen. Jede Abhängigkeit würde die Möglichkeit zur Komposition einschränken.
Unabhängigkeit hat technische, fachliche, wirtschaftliche und juristische Aspekte. Demnach kann eine Komponente aus technischer Sicht unabhängig sein, die Verwendung auf Grund von juristischen Unvereinbarkeiten mit dem Kontext jedoch trotzdem unmöglich.[7]

2.2.2.2 Abgeschlossenheit von Komponenten

Eine andere Facette neben der Unabhängigkeit (siehe Abschnitt 2.2.2.1) ist die Abgeschlossenheit einer Komponente gegenüber ihrem Kontext. Damit eine Komponente in einer Komposition oder einem unbekannten, neuen Kontext funktionieren kann, muss sie alle Abhängigkeiten zur Übersetzungs- und Laufzeit beinhalten oder Annahmen über den Kontext machen (vgl. Unterabschnitt 2.2.2.10). Eine Besonderheit der Abgeschlossenheit muss in Bezug zur Übersetzungszeit erwähnt werden. Da die Komposition durch Dritte durchgeführt wird und die Komposition den Übersetzungsschritt einschließt, muss die Komponente ihr Build-Management enthalten. Eine Ausnahme dazu stellt jedoch die Black-Box-Komponente dar (siehe Unterabschnitt 2.2.2.9).
Da eine vollständige Komponentenabgeschlossenheit die Fähigkeit einer Komponente zur Interaktion mit ihrem Kontext zerstören würde, muss jedoch ein Mechanismus existieren, über den die Komponente mit ihrem Kontext kommunizieren kann. Diese Mechanismen sind

[7] Ein Beispiel wären inkompatible Lizenzen.

die Schnittstellen der Komponente (siehe Unterabschnitt 2.2.2.8).
Zur Laufzeit äußert sich die Abgeschlossenheit dadurch, dass eine Komponente entweder im Kontext eingebunden ist, oder nicht. Eine Komponente ist nicht fähig, nur teilweise im Kontext eingebunden zu sein [Szyperski 2002, S. 36].

2.2.2.3 Zustandslosigkeit von Komponenten

Wenn Komponenten in einem Kontext eingebunden sind, existiert zur Laufzeit, ähnlich dem Singleton der Objektorientierung (vgl. [Gamma u. a. 1995]), eine einzige Instanz der Komponente. Dieses begründet sich in der geforderten Zustandslosigkeit von Komponenten. Hätten Komponenten einen (externen) Zustand, so würden sie die Forderungen nach Unabhängigkeit und Abgeschlossenheit verletzten und darüber hinaus ein dynamisches Komponentenmodell ausschließen (siehe Abschnitt 2.2.4.1). Diese Zustandslosigkeit impliziert, dass verschiedene Instanzen einer Komponente von außen nicht unterscheidbar und somit identitätslos sind. Das Konzept der Zustandslosigkeit kann in manchen Situationen allerdings aus technischen Gründen, wie zum Beispiel zur Performanzsteigerung, aufgebrochen werden [Szyperski 2002, S. 36].

2.2.2.4 Komponenten und Objektorientierung

Laut [Heineman 2001, S. 36] ist das CBSE ein Nachfolger der objektorientierten Programmierung. Teilweise werden die Begriffe „Komponente" und „Objekt" sogar mit gleicher Bedeutung verstanden. Werden die Begriffe aber im Zusammenhang des CBSE verwendet, unterscheiden sich die dahinter stehenden Konzepte grundsätzlich [Cheesman 2000, S. 8]. Obwohl die Implementierung von Komponenten häufig mit der Objektorientierung erfolgt[8], überwiegt die Anzahl der Unterschiede die Zahl der Gemeinsamkeiten.
Objekte als auch Komponenten interagieren und kollaborieren innerhalb ihres Kontextes miteinander und stellen Funktionalität über Schnittstellen bereit. Unterschiedlich sind hingegen die Singleton-Charakteristika der Komponenten (vgl. Abschnitt 2.2.2.1), die Identitätslosigkeit (vgl. Abschnitt 2.2.2.3) und die Abgeschlossenheit (vgl. Abschnitt 2.2.2.2) der Komponenten. Diese Einschränkungen sind in der Objektorientierung nicht existent [Szyperski 2002, S. 37–38].

2.2.2.5 Kennzeichnung von Komponenten

Komponenten sind identitätslos (siehe Abschnitt 2.2.2.3), trotzdem besitzt jede Komponente eine eindeutige Kennzeichnung. Über diesen Bezeichner kann die Komponente von anderen Komponenten unterschieden werden. Die Bezeichnung wird somit zur Unterscheidung von

[8] CBSE ist auf keine Implementierungssprache begrenzt. Beispiele können für funktionale, prozedurale, maschinennahe und aspektorientierte Programmierung geliefert werden.

verschiedenen Komponenten aber nicht von Instanzen einer Komponente genutzt. Inhaltlich dient die Kennzeichnung zur Unterscheidung der Komponente in Bezug auf Funktionalität, aber auch zur Unterscheidung von funktional identischen Komponenten verschiedener Hersteller. Funktional identische Komponenten realisieren dieselben Schnittstellen und bieten daher gleiche Dienste an (siehe Abschnitte 2.2.2.8 und 2.2.4.2).
Als technische Realisierung sind zwei Mechanismen bekannt [Heineman 2001, S. 40]:

- Global eindeutige Bezeichner („Unique Universal Identifier"): Diese Kennzeichnungen können entweder von einer zentralen Instanz an Komponentenhersteller vergeben werden[9], oder durch technische Konzepte eindeutig sein.
- Hierarchische Namensräume: Einem Hersteller wird durch eine zentrale Vergabestelle ein bestimmter Präfix zugewiesen, unterhalb dessen die Kennzeichnung der Komponenten eingehängt ist.

Globale Bezeichner haben somit den Vorteil, ohne zentrale Instanz eindeutig zu sein.

2.2.2.6 Versionierung von Komponenten

Komponenten müssen die Fähigkeit zur Weiterentwicklung aufweisen. Um die resultierenden Unterschiede dieser Weiterentwicklung kenntlich zu machen, ist eine Versionierung nötig. Die Version einer Komponente bezieht sich auf deren Schnittstellen. Eine Änderung der Komponente, die deren (explizite) Schnittstellen nicht beeinflusst, macht wegen (impliziten) Schnittstellen oft eine neue Version der Komponente nötig (vgl. Abschnitt 2.2.2.8).
Eine einfache Lösung zur Versionierung ist die Nutzung von Versionsnummern in der „Major.Minor"-Notation. In Verbindung mit der Komponentenkennzeichnung (siehe Abschnitt 2.2.2.5), ist die Komponente dadurch im Kontext eindeutig identifizierbar. Ein vergleichbares Konstrukt ist die International Standard Book Number (ISBN) die aus einer Länderkennzeichnung, einer Herausgeberkennzeichnung sowie einer Version des Buches besteht.

2.2.2.7 Abhängigkeiten zwischen Komponenten

Abhängigkeiten bestehen exklusiv zwischen Komponenten und werden durch die Angabe der Komponentenkennzeichnung (siehe Abschnitt 2.2.2.5) und der Komponentenversion (siehe Abschnitt 2.2.2.6) angegeben. Ist die Anzahl der Komponenten entsprechend groß und die Abhängigkeiten umfangreich, entsteht ein starres Netz aus Abhängigkeiten. In diesem Fall ist ein flexibler Austausch von Komponenten nicht mehr möglich, was eingangs als Ziel des CBSE gefordert wurde. Die Kopplung zwischen den Komponenten steigt, wohingegen die Modularität abnimmt [Heineman 2001, S. 44].

[9] Ein Beispiel sind MAC-Adressen

Um dieses Problem zu lösen, sollten Komponenten nicht nur eine einzige mögliche Abhängigkeit, sondern eine minimale und maximal unterstützte Version angeben.[10] Die Angabe der unterstützten Version erfolgt dabei in Form eines Schiebefensterverfahrens, womit die Fortentwicklung der Abhängigkeiten sichergestellt wird. Veraltete, nicht mehr benötigte Abhängigkeiten, wandern über die Zeit aus dem Fenster, neue Abhängigkeiten in das Fenster hinein. Zusätzlich zur Angabe von unterstützten Versionen, können alternative Komponenten von anderen Herstellern als Abhängigkeit definiert werden.[11]
Sind zwei Komponenten von einer dritten Komponente abhängig und besteht keine Überschneidung zwischen den erlaubten Versionen der referenzierten Komponente, können diese beiden Komponenten nicht im selben Kontext existieren. Um diesen Fall zu lösen, muss der Kontext das Laden von Komponenten mit identischer Kennzeichnung aber unterschiedlichen Versionen zulassen und eine zeitgleiche Koexistenz erlauben [Szyperski 2002, S. 52-53]. Dadurch sollte sofort die Forderung nach Zustandslosigkeit der Komponenten ersichtlich werden (siehe Abschnitt 2.2.2.3). Hätte die referenzierte Komponente einen (externen) Zustand, würden Widersprüchlichkeiten zwischen den Zuständen der beiden Komponenten verschiedener Versionen auftreten.
Eine weitere Notwendigkeit zum flexiblen Austausch von Komponenten sind Abhängigkeiten auf Komponenten ausschließlich über deren spezifizierte Schnittstellen zu zulassen. Würde eine Komponente Annahmen über Interna einer abhängigen Komponente machen, entstünde eine untrennbare Kopplung (vgl. Abschnitt 2.2.2.1). Demnach muss die Implementierung einer Komponente, wie bei der Objektorientierung, hinter den Schnittstellen verborgen sein [Cheesman 2000, S. 5]. Verborgen kann eine technische Trennung der Komponentenimplementierung von der Komponentenschnittstelle bedeuten, wie es in Abbildung 2.1 schematisch dargestellt wird.

2.2.2.8 Schnittstellen von Komponenten

Über Schnittstellen stellen Komponenten ihre Funktionalität – ihre Dienste – im Kontext bereit. Somit bietet eine Komponente eine oder mehrere Schnittstellen und damit eine Menge an Diensten an. Die Art der Schnittstellen und Dienste muss zur Erreichung von Universalität und damit Wiederverwendbarkeit ein gewisses Niveau an Abstraktion erreichen. Umso konkreter und spezieller die Schnittstelle der Komponente ist, desto schwerer kann die Komponente in einem anderen Kontext eingesetzt werden.
Konzeptionell gliedert sich die Schnittstellenspezifikation einer Komponente in zwei Arten:

- Angebotene Schnittstellen („provided interface”): Angebotene Schnittstelle definieren die Dienste einer Komponente, die sie im Kontext bereitstellen kann.

[10] Eine sinnvolle Angabe der Maximalversion kann nur für bekannte Versionen getroffen werden.
[11] Ein Beispiel ist die Verwendung von Glühbirnen verschiedener Hersteller in einer Lampe.

- Geforderte Schnittstellen („required interface”): Geforderte Schnittstellen spezifizieren die Kontextabhängigkeiten der Komponente und damit Abhängigkeiten auf andere Komponenten. Werden die Anforderung dieser Schnittstelle nicht erfüllt, kann die Komponente auf Grund fehlender Funktionalität ihre angebotenen Schnittstellen nicht zur Verfügung stellen.

Die Spezifikation dieser Schnittstellen muss strikt und präzise sein. Da die abhängigkeitsauflösenden Komponenten zur Entwicklungszeit nicht bekannt sind, stellt die Schnittstelle die einzige Gemeinsamkeit zwischen den Komponenten dar. Die Schnittstelle liegt daher logisch zwischen den Komponenten und verbindet sie.
Die Genauigkeit der Spezifikation muss zum einen technisch von der Ebene der Sprachmittel abstrahieren um eine Unabhängigkeit vom Programmiermodell zu erreichen. Dies geschieht durch den Einsatz einer Interface Definition Language (IDL). Zum anderen reicht die ausschließliche Spezifikation des technischen Application Programming Interface (API) nicht aus. So beeinflusst beispielsweise die Laufzeit eines Algorithmus in einer Komponente das gesamte System. Diese Abhängigkeiten werden als indirekte Schnittstellen bezeichnet. [Szyperski 2002, S. 54] empfiehlt daher eine Spezifikation auf der Dienstebene der Komponente. Diese Spezifikation umfasst das technische API der Komponente und des erwarteten Kontexts, aber auch eine nicht funktionale Spezifikation und Dokumentation. Eine beispielhafte Darstellung findet sich in [Heineman 2001, S. 49–64]. Durch die nicht technischen Teile der Spezifikation wird das automatische Abprüfen auf Einhaltung der Spezifikation unmöglich.
Die Qualität dieser Spezifikation schlägt sich wiederum maßgeblich auf die Fähigkeit der Komponente zur Wiederverwendbarkeit nieder. Sind Verhalten und Eigenschaften einer Komponente nicht hinreichend bekannt, fällt der Einsatz dieser Komponente aus Sicht Dritter schwer.

2.2.2.9 Arten von Komponenten

Komponenten können anhand von zwei Eigenschaften unterteilt werden. Stammt eine Komponente aus eigener Fabrikation, wird von Eigenkomponenten gesprochen. Stammt sie aus fremder Produktion, sind es Fremdkomponenten (vgl. Abschnitt 2.2.5). Alternativ werden Fremdkomponenten als Commercial-Off-The-Shelf (COTS)-Komponenten bezeichnet.
Die zweite, aus technischer Sicht interessantere Eigenschaft, ist die Art der Komponente. Die Art bezieht sich auf die Verfügbarkeit der Komponentenimplementierung und die Möglichkeit zur Modifikation dieser Implementierung durch Dritte. Dieses resultiert im CBSE in vier verschiedenen Komponentenarten [Heineman 2001, S. 38–39] [Szyperski 2002, S. 40–41]:

- Black-Box-Komponente: Bei der Black-Box-Komponente ist die Implementierung der Komponente vollständig verborgen. Die Auslieferung der Komponente erfolgt ausschließlich in Binärform.

- Grey-Box-Komponente: Eine Spezialisierung der Black-Box-Komponenten ist die Grey-Box-Komponente. Auf die gesamte Komponentenimplementierung ist kein Zugriff möglich. Es werden allerdings bestimmte Teile der Implementierung nach außen gegeben. Ein Einsatzbeispiel ist die Herausgabe von Komponenteninterna zur Performanzsteigerung.

- Glass-Box-Komponente: Im Gegensatz zu Black-Box- und Grey-Box-Komponente ist die Komponentenimplementierung bei der Glass-Box-Komponente ein Bestandteil der an Dritte ausgegebenen Komponente. Die Implementierung kann zur Analyse der Komponente heran gezogen werden. Eine Modifikation der Implementierung ist jedoch nicht möglich.

- White-Box-Komponente: Die White-Box-Komponente weist in Bezug auf die Implementierung die größte Flexibilität auf. Die gesamte Implementierung der Komponente ist verfügbar und kann durch Dritte beliebig modifiziert und an spezielle Anforderungen angepasst werden.
  Aus Sicht des CBSE ist eine Modifikation der Komponente gleichwohl problematisch und stellt zumindest bei Anpassung der Komponentenschnittstellen in jedem Fall eine neue Komponente dar (vgl. Abschnitt 2.2.2.8). Unter dem Verweis auf indirekte Schnittstellen kann aber auch schon die Variation von Implementierungsinterna zur Entstehung einer neuen Komponente führen.

White-Box-Komponenten bieten im Rahmen des CBSE im Gegensatz zu den anderen Komponentenarten die größte Flexibilität zur Anpassung an die eigenen Anforderungen. Die Vorteile einer solchen Anpassung müssen aber mit den Nachteilen abgewogen werden. Durch die Erzeugung einer neuen Komponente aus einer Fremdkomponente verschieben sich die Zuständigkeiten für Wartung und Pflege. Wenn neue Komponentenversionen der ursprünglichen Komponente erscheinen und eingesetzt werden sollen, wird zusätzlich der spätere Austausch der Komponente erschwert. In diesem Fall müssen die Modifikationen neu eingepflegt werden.

2.2.2.10 Granularität von Komponenten

Die optimale Komponente bietet eine Menge von Diensten an (angebotene Schnittstellen), die disjunkt zueinander und zu anderen Diensten im Kontext sind.[12] Zur Erfüllung dieser Dienste benötigt sie keine Kontextabhängigkeiten (geforderte Schnittstellen). Dadurch ist die Komponente innerhalb jedes Kontextes einsatzfähig. Das Erreichen dieses Zustands ist allerdings für nicht triviale Komponenten unmöglich und widerspricht anderen Komponentenanforderungen des CBSE.

[12] Alternative Komponenten erfüllen jedoch identische Dienste.

Um eine Komponente vollständig kontextunabhängig zu gestalten, müssen alle benötigten Dienste in die Komponente integriert werden. Dieser Ansatz zur Behandlung der geforderten Schnittstellen verhält sich konträr zur CBSE-Forderung nach Wiederverwendung. Identische Dienste würden innerhalb des Kontextes vielfach vorkommen. Eine Alternative ist die Aufspaltung der Komponente in mehrere Komponenten, die jeweils nur einen Dienst realisieren. Dadurch ließen sich Redundanzen innerhalb der Komponenten vermeiden. Neben den offensichtlichen Vorteilen muss bei dieser Lösung jedoch die Explosion der Kontextabhängigkeiten (geforderten Schnittstellen) berücksichtigt werden.
Infolge dieser Unvereinbarkeit muss eine Balance zwischen der Anzahl der Kontextabhängigkeiten und der Menge der angebotenen Dienste gefunden werden. [Szyperski 2002, S. 45] folgert dazu: „Maximizing reuse minimizes use", wonach die Steigerung der Wiederverwendung die Möglichkeit zum Einsatz einschränkt. Ein ähnliche Aussage kann in Bezug auf den Umfang der Komponentendienste gemacht werden. Mit steigender Komponentengranularität (Anzahl der angebotenen Schnittstellen), sinkt die Fähigkeit zur modifikationsfreien Wiederverwendung einer Komponente [Atkinson u. a. 2003, S. 139]. Demnach lassen sich Komponenten, die beispielsweise einen Stack oder Queue realisieren, umfangreicher wiederverwenden, als eine Komponente, die eine vollständige Fachlogik kapselt.
Wegen diesen grundsätzlichen Problemen kann keine allgemein gültige Ober- und Untergrenze für die Komponentengranularität angesetzt werden. Allerdings können konzeptionelle Überlegungen und empirische Erfahrungswerte aufgeführt werden. Die Granularität einer Komponente in Bezug auf ihre Dienste soll aus Sicht der Dienste abgeschlossen sein. Setzt eine Komponente eine bestimmte Fachlogik um, so soll die Komponente alle Aspekte dieser Fachlogik beinhalten. Einschränkend muss allerdings erwähnt werden, dass die technische Realisierung dieser Komponente durch eine Komposition aus mehreren Komponenten erfolgen kann.
Im Zusammenhang der technischen Implementierung der Komponente durch die Objektorientierung wird in [Heineman 2001, S. 36] die Anzahl der Klasse, welche die Granularität der Komponente festlegen keine exakte Unter- und Obergrenze zur Klassenanzahl gegeben. Eine Komponente kann aus einer oder etlichen Klassen bestehen.

2.2.3 Komponentenmodell

Die Summe der bisher genannten Komponenteneigenschaften ist eine Konvention, die durch das Komponentenmodell spezifiziert wird. Der bisher abstrakt bezeichnete Kontext einer Komponente und dessen Charakteristika, ist das logische Äquivalent des Komponentenmodells. Es ist offensichtlich, dass das Komponentenmodell den theoretischen und konzeptionellen Rahmen angibt. In [Heineman 2001, S. 47–48] bilden die Implementierung der Komponenten, Komponentenkennzeichnung, Komponentenuniversalität, Komponentenanpassbarkeit, Komponentenkomposition, Komponentenevolution und die Komponenteneinbettung die Hauptaspekte des Komponentenmodells (vgl. Abschnitte 2.2.2.1–2.2.2.8).

Ein Komponentenstandard spezifiziert das Komponentenmodell formal. Ein Standard dient dazu, die Verbreitung eines Komponentenmodells zu verstärken, da sich Nutzer des Standards auf garantierte Eigenschaften verlassen können [Szyperski 2002, S. 27].

2.2.4 Komponenten-Rahmenwerke

Nach vorgenannter Definition aus [Heineman 2001, S. 47-48] bietet eine Implementierung eines Komponentenmodells folgende Funktionen an:

- Eine Laufzeitumgebung („a run-time environment").
- Dienste mit universellem Charakter („basic services").
- Horizontale Dienste, dessen Nützlichkeit sich über alle Fachbereiche erstreckt („horizontal services that are useful across multiple domains").
- Dienste mit vertikaler Relevanz, was bedeutet, dass sie sich für eine oder mehrere Fachbereiche eignen („vertical services providing functionality for a particular domain for software components").

Bei [Szyperski 2002] wird die Implementierung des Komponentenmodells als Komponenten-Rahmenwerk bezeichnet. Beide Definitionen gehen daher über eine Klassenbibliothek und Sammlung von grundlegenden Diensten hinaus [Szyperski 2002, : 425]. Eine besondere Bemerkung muss hier zur Abhängigkeit der Komponenten von den Diensten des Rahmenwerks gemacht werden. Im Abschnitt 2.2.2.7 wurde die Aussage getroffen, dass Abhängigkeiten exklusiv zwischen Komponenten bestehen. Dies trifft trotz der eingeführten Rahmenwerksdienste zu, weil Rahmenwerke selbst eine Komposition aus Komponenten darstellen.

2.2.4.1 Laufzeitumgebung

Das Komponenten-Rahmenwerk stellt eine Laufzeitumgebung für Komponenten bereit. Darin können Komponenten aus unterschiedlicher Produktion, die dem Komponentenstandard des Rahmenwerks entsprechen, zur Laufzeit (dynamisch) in Kompositionen zusammengesetzt werden. Komponenten haben einen Lebenszyklus innerhalb des Kontexts. Daraus leiten sich bestimmte Anforderungen an eine Laufzeitumgebung und somit das Rahmenwerk ab:

- Sicherheit: Ein Fehlerfall innerhalb einer Komponente oder in der Interaktion zwischen Komponenten darf sich nicht auf den restlichen Kontext (andere Komponente) auswirken. Das Rahmenwerk muss den Fehler in einer Komponente abfangen und behan-

deln.[13] Diese Art der Fehlerbehandlung wird als „Sandboxing" bezeichnet.
Ein vorheriger Anschluss des Fehlerfalls durch Testen und Verifizieren der abgeschlossenen und unabhängigen Komponente (vgl. Abschnitt 2.2.2.1, 2.2.2.2) ist vor dem Hintergrund der unvermeidbaren Kontextabhängigkeiten (vgl. Abschnitt 2.2.2.10) nicht möglich. Zur Entwicklungszeit ist die Menge der möglichen Kontextabhängigkeiten nicht bekannt [Szyperski 2002, S. 565].

- Zugriffsrechte von Komponenten: Die Laufzeitumgebung des Komponenten-Rahmenswerks reglementiert die Abhängigkeiten zwischen Komponenten anhand spezifizierter Zugriffsrechte.

- Lebenszyklen: Die Zusammensetzung von Komponenten in Kompositionen erfordert, dass Komponenten im Kontext eingebunden sind. Der Lebenszyklus definiert den Zustand einer Komponente im Kontext.

2.2.4.2 Grundlegende, horizontale und vertikale Dienste

Laut [Heineman 2001, S. 47–48] stellt das Rahmenwerk Dienste bereit. Diese Dienste werden nach grundlegenden, horizontalen und vertikalen Aspekten klassifiziert, obwohl Dienste prinzipiell nicht unterscheidbar sind. Grundlegende Dienste sind Dienste, die zur Umsetzung des Komponentenmodells definitiv nötig sind und ohne deren Existenz keine Interaktion zwischen Komponenten möglich ist (siehe Abschnitt 2.2.4.1). Die Menge dieser Dienste hängt vom Komponentenmodell ab und ist innerhalb der Komponentenspezifikation festgelegt. Horizontale Dienste sind keine grundlegenden Dienste, ihrer Art nach aber in allen Fachbereichen verwendbar. Vertikale Dienste sind spezifisch auf den Fachbereich zugeschnitten und realisieren Fachlogik sowie spezielle Anforderungen des Fachbereichs.
Horizontale und vertikale Dienste unterscheiden sich dahingehend, dass horizontale Dienste technische Aspekte abdecken und vertikale Dienste für spezifische Fachwelten konzipiert sind. Abbildung 2.2 illustriert diese Verhältnisse und die Beziehungen zwischen den Diensten.

2.2.5 Komponentenmärkte

An dieser Stelle soll ein Rückgriff auf das einleitende Zitat erfolgen (siehe Abschnitt 2.2.2), wonach eine Komponente: „[...] subject to composition by third parties [is]." Komponenten

[13] Ein prominentes Beispiel verdeutlicht dieses Problem. Beim Start der Ariane 5 im Jahr 1996 explodierte die Rakete nach wenigen Sekunden. Eine anschließende Untersuchung der Katastrophe ergab, dass die Ursache in einem wiederverwendeten Software-Modul zur Flugbahnberechnung entstand [Lions 1996]. Ein simpler Konvertierungsfehler hatte das gesamte System zum Absturz gebracht. Wäre das System mit Komponenten realisiert gewesen, hätte ein Komponenten-Rahmenwerk den Fehler innerhalb der Komponente erkennen müssen und das System in einen definierten Zustand überführt.

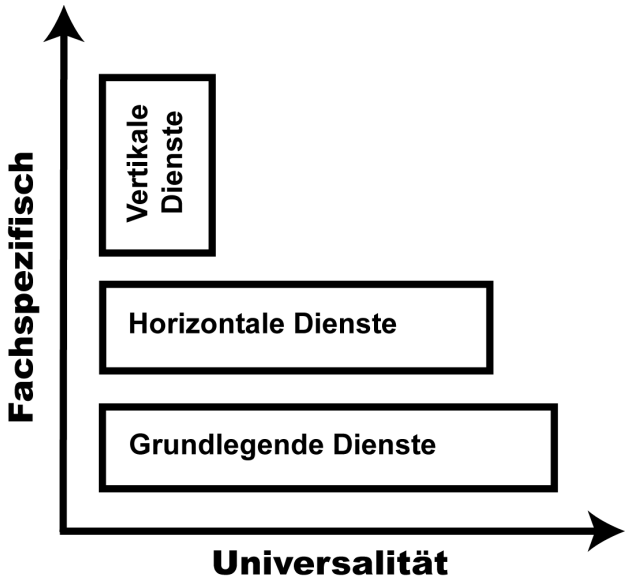

Abbildung 2.2: Grundlegende, horizontale und vertikale Komponentendienste

(Angelehnt an [Heineman 2001])

werden demnach durch Dritte zusammengesetzt. Dieses deutet den wirtschaftlichen Aspekt des CBSE an.
Komponenten werden nicht nur zur Wiederverwendung innerhalb einer Organisation entwickelt, sondern auch, um außerhalb der Organisation genutzt zu werden.[14] Somit wird ein Komponentenmarkt nötig, auf dem Komponenten gehandelt werden können. An diesen Komponentenmärkten partizipieren in Konkurrenz zueinander stehende Organisationen, die einen Handel mit Komponenten betreiben und Dienstleistungen rund um Komponenten anbieten [Szyperski 2002, S. 495ff]. Dabei impliziert der Handeln von Komponenten nicht zwangsläufig einen Verkauf und damit eine Währung. Komponenten können ebenso auf dem Markt getauscht werden (vgl. Abschnitt 3.2). Es muss lediglich ein Nutzen für alle teilnehmenden Parteien entstehen, da die Hersteller ihr Engagement ansonsten einstellen würden.
Entstammt eine Komponente aus fremder Herstellung, so gelten diese Komponenten als COTS-Komponenten [Heineman 2001, S. 100]. Entwickelt ein Unternehmen COTS-Komponenten, so sind sie auf horizontale oder vertikale Märkte zugeschnitten (vgl. Abschnitt 2.2.4.2).

[14] Es wäre zwar auch eine rein betriebsinterne Verwendung vorstellbar, diese könnte aber nicht die Vielfalt und damit Qualität eines heterogenen Markts erlangen.

2.2.5.1 Komponenten-Repository

Komponenten-Repositories entsprechen dem Marktplatz eines Komponentenmarkts. Sie werden allerdings auch beim organisationsinternen CBSE eingesetzt. Ein Komponenten-Repository ist ein (globales) Verzeichnis der verfügbaren Komponenten. Dabei kann das Komponenten-Repository eine Komposition aus mehreren Verzeichnissen sein [Heineman 2001, S. 516ff]. Laut [Szyperski 2002, S. 471] müssen Komponenten innerhalb des Repository umfangreich dokumentiert sein (insbesondere die Schnittstellen), damit ein Auffinden passender Komponente möglich wird und Wiederverwendung entstehen kann.

2.2.5.2 Zertifizierung von Komponenten

[Heineman 2001] begründet, warum eine Zertifizierung von Komponenten für den Erfolg eines Kompontenmodells essentiell wichtig ist. Der Einsatz von COTS-Komponenten erfordert demnach vom Nutzer Vertrauen in die Qualität der Komponente in zweierlei Hinsicht. Zum einen muss eine problemlose Komponentenkomposition in bestehenden Systemen möglich sein (siehe Abschnitt 2.2.2.1), zum anderen muss die Komponente den Anforderungen des Einsatzgebiets entsprechend und ausreichend robust[15] sein (siehe Abschnitt 2.2.4.1). Zur Lösung dieser Ansprüche können COTS-Komponenten für den Einsatz in einem Komponentenstandard nach definierten Qualitätsstandards zertifiziert werden. Die Rolle der Zertifizierungsinstanz muss allgemein vom Markt anerkannt sein. Da eine Zertifizierung einzelner Komponenten aufwändig ist, kann anstelle der Komponente der Herstellungsprozess der Komponente zertifiziert werden. Eine umfangreiche Übersicht über den Stand der Zertifizierung von Software-Komponenten findet sich in [Alvaro u. a. 2005].

2.2.6 Beispiele für Komponenten, Komponentenmodelle und Komponenten-Rahmenwerke

Ein erstes naives Beispiel für Komponenten, Komponentenmodelle und Komponenten-Rahmenwerke wäre die Klassifikation von Betriebssystemen als Komponenten-Rahmenwerke innerhalb derer Anwendungen die Komponenten darstellen, die auf dem Software-Markt gehandelt werden. Wird diese Aussage anhand obiger Erläuterungen betrachtet, kann sie allerdings als Falschannahme aufgedeckt werden.
Ein Betriebssystem definiert keine Schnittstellen für Anwendungen, über die eine Interaktion der Anwendungen erfolgen kann (vgl. Abschnitt 2.2.2.8).[16] Ferner besteht keine Wiederverwendung zwischen Anwendungen, wie vom CBSE gefordert (siehe Abschnitt 2.2.1). Die Ursache liegt in der Granularität der Anwendungen, die nicht den Anforderungen des CBSE entsprechen (sieh Abschnitt 2.2.2.10) [Heineman 2001, S. 34].

[15] Robustheit bedeutet, dass eine Komponente keinen Fehlerfall der Komposition verursacht.
[16] Interprozesskommunikation (IPC) stellt keine spezifizierte Schnittstelle im Sinne des CBSE dar.

Die Literatur des CBSE führt Common Object Request Broker Architecture (CORBA) der Object Management Group, Microsofts Component Object Model (COM) und Enterprise Java Beans (EJB) von SUN Microsystems als Beispiele für Komponentenstandards an [Heineman 2001, S. 557–607][Szyperski 2002, S. 231–381]. Da in späteren Teilen dieser Arbeit auf ein Java-basiertes Komponentenmodell eingegangen wird, soll an dieser Stelle ein Hinweis im Zusammenhang mit dem EJB-Modell gegeben werden, wonach entsprechend der Definition aus [Szyperski 2002, S. 292] ein Java-Archiv (JAR) im EJB-Komponentenstandard als rudimentäre Komponente gilt.
Ein umfangreicheres Beispiel kann in Kapitel 3 gefunden werden. Dort findet eine Betrachtung von Eclipse aus Sicht des CBSE statt.

2.2.7 Fazit Komponentenbasierte Software-Entwicklung

Die einleitende Analogie zur Automobilindustrie und Henry Ford aufgreifend (siehe Kapitel 1), steht die Komponentenbasierte Software-Entwicklung „[...] somewhere between a Model T and a 1950 Ford." [Heineman 2001, S. 750]. CBSE ist nach dieser Metapher bisher nicht am Ende der Entwicklung angelangt. Einige Probleme sind komplett oder zumindest teilweise ungelöst.

2.2.7.1 Probleme der komponentenbasierten Software-Entwicklung

Ein konzeptionelles Problem des CBSE stellt die Granularität der Komponenten dar (siehe Abschnitt 2.2.2.10). Das Paradoxon zwischen Wiederverwendung und Universalität ist grundlegend.
Neben diesem Problem, sind die weiteren Schwierigkeiten des CBSE jedoch lösbar. Um der Forderung nach Wiederverwendbarkeit nachzukommen, müssen die Schnittstellen einer Komponente rigoros nach einem strikten, aber auch allgemein akzeptierten Standard dokumentiert sein (vgl. Abschnitt 2.2.2.8). Treffen unterschiedliche Interessen zusammen, wird ein Konsens prekär. Bevor über den Erfolg eines Standards entschieden werden kann, bedarf es einem unter marktwirtschaftlichen Gesichtpunkten funktionierenden Markts. Ohne diesen Markt, der eine Auswahl an Komponenten anbietet, wäre CBSE nicht möglich, zumindest aber auf einen organisationsinternen Komponentenmarkt beschränkt. Komponentenbasierte Software-Entwicklung kann ihr komplettes Potenzial lediglich ab einer bestimmten Verbreitung ausspielen. Eine unbegrenzte Verbreitung ist jedoch nicht möglich [Szyperski 2002, S. 46ff].
Für die Nutzer des CBSE muss sich ein genereller Vorteil durch den Einsatz des CBSE bieten, damit eine Bereitschaft zum Einsatz sowie Produktion von Komponenten besteht. Sollte sich kein (messbarer) Vorteil zeigen, stellen sie ihre Teilnahme am CBSE ein. Ein Kritikpunkt am CBSE ist die geringe Anzahl an (empirischen) Untersuchungen zu dessen Nutzen. Die aus Sicht der Wissenschaften noch junge CBSE-Disziplin hat bisher keine umfangreichen

Beweise ihrer Nützlichkeit liefern können. Jedoch gehen CBSE-Fachleute davon aus, dass das CBSE seine Vorteile in den kommenden Jahren belegen wird [Heineman 2001, S. 773–774].

2.2.7.2 Vorteile der komponentenbasierten Software-Entwicklung

In [Heineman 2001, S. xxii] wird der komponentenbasierten Software-Entwicklung die Fähigkeit zur Lösung der Software-Krise zugesprochen. Diese Fähigkeit basiert auf den grundlegenden Zielen des CBSE, Wiederverwendung von Komponenten und Flexibilität bei der Komposition von Komponenten zu liefern (siehe Abschnitt 2.2.1).
Durch Wiederverwendung von (Fremd-)Komponenten können Software-Systeme aus Bestehendem zusammengesetzt, anstatt von Grund auf neu entwickelt zu werden, wodurch die Kosten für Entwicklung, Wartung und Pflege des Software-Systems sinken. Zusätzlich verkürzt sich der Entwicklungsprozess, weshalb die Arbeiten am Software-System früher abgeschlossen werden können [Heineman 2001, S. 502]. Die Entwicklung von Software wird wie in der Ford-Analogie wirtschaftlicher.
Die Flexibilität der komponentenbasierten Software-Entwicklung erlaubt daneben einen einfachen und schnellen Austausch von Komponenten durch Alternativen. Dadurch wird nicht nur eine Steigerung der Qualität erzielt, sondern auch eine Möglichkeit zur Anpassung an sich ständig ändernde Anforderungen geschaffen.

2.3 Konsolidierung auf ein Komponentenmodell

Nachdem in den beiden vorherigen Abschnitten die Begriffe Konsolidierung (siehe Abschnitt 2.1) und komponentenbasierte Software-Entwicklung (siehe Abschnitt 2.2) getrennt betrachtet wurden, werden die Begriffe in diesem Abschnitt gemeinsam untersucht und zusammengeführt. Der Schwerpunkt dieses Abschnitts ist demnach die Dekomposition[17] einer Bestandsanwendung in Komponenten (siehe Abschnitt 2.3.2) und deren Integration innerhalb eines Komponentenmodells (siehe Abschnitt 2.3.4). Abbildung 2.3 stellt diese Sachverhalte schematisch dar.

2.3.1 Vorstudie

Bevor eine Konsolidierung beginnen kann, muss eine Vorstudie durchgeführt werden [Brössler 2000, S. 212]. Diese Vorstudie dient der Untersuchung, ob eine Dekomposition des Altsystems in Komponenten kosteneffektiv durchgeführt werden kann oder ob eine vollständige Neuimplementierung des Fach- und Anwendungswissens wirtschaftlicher wäre. Ob eine

[17] Dekomposition meint nicht die Umkehroperation zur Komposition des CBSE, sondern die Zerlegung eines Altsystems.

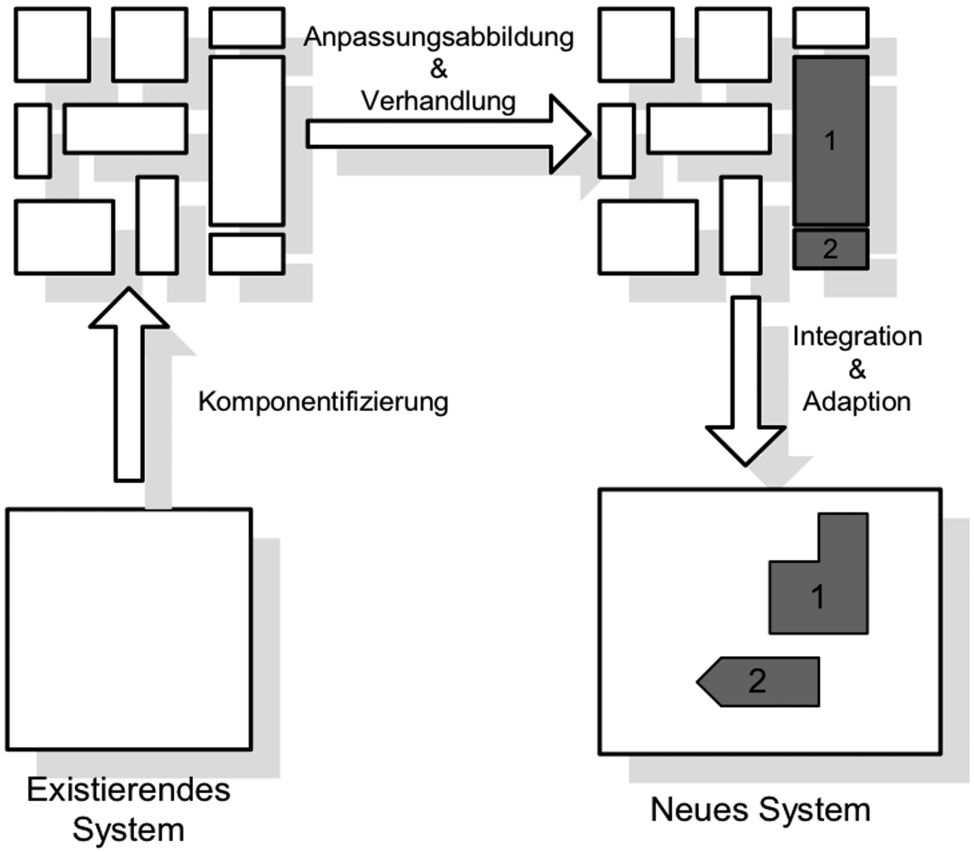

Abbildung 2.3: Schematische Darstellung der Integration vorhandener Quelltextteile

(Angelehnt an [Atkinson u. a. 2003, S. 159])

Konsolidierung sinnvoll umsetzbar ist, hängt vom Komponentenmodell und der Bestandsanwendung ab. Demnach muss in der Vorstudie eine Vorauswahl des Komponentenmodells getroffen werden und ausreichend Anwendungswissen bestehen, um eine objektive Einschätzung über die Dekomposition zu treffen.
Die Wahl des Komponentenmodells orientiert sich nach der fachlichen Architektur der Bestandsanwendung und stellt einen Abgleich mit den vertikalen Diensten des Komponenten-Rahmenwerks dar (vgl. Abschnitt 2.2.4.2). Die Integrationsfähigkeiten von gewonnen Komponenten aus dem Altsystem sind umso höher, je mehr sich Komponentenmodell und Anwendungsspezifikation ähneln. Beispielsweise sind die Anforderungen einer Echtzeitanwendung an ein Komponentenmodell offensichtlich unterschiedlich zu denen einer betriebswirt-

schaftlichen Anwendung.
Ob die Dekomposition der Altwendung wirtschaftlich ist, hängt von der technischen und fachlichen Architektur der Bestandsanwendung ab. Konkreter sind dies die Aspekte der Kapselung (Kohäsion), Kopplung und Modularität der Architektur. Zusätzlich eignen sich bestimmte Programmierkonzepte auf Grund von Ähnlichkeiten der Paradigmen besser zur Umwandlung in Komponenten als andere (vgl. Abschnitt 2.2.2.4).

2.3.2 Komponentifizierung

Unter Komponentifizierung versteht man den Prozess der Erzeugung von Komponenten aus einer Bestandsanwendung (Dekomposition der Bestandsanwendung). Dieser umfasst bei [Atkinson u. a. 2003, S. 160] die Kandidatenidentifikation sowie die Umwandlung dieser Kandidaten in Komponenten des Komponentenmodells. Im Gegensatz dazu definieren [Bauer u. a. 2000] die Komponentifizierung lediglich als die Dekomposition der Kandidaten in Komponenten. Die Kandidatenidentifikation ist dort ein gesonderter Prozess. Diese Arbeit folgt der Definition aus [Atkinson u. a. 2003] und zählt die Identifikation der Komponentenkandidaten zur Komponentifizierung.

2.3.2.1 Auswahl der Komponentenkandidaten

Komponentenkandidaten sind Entitäten, die entweder als Komponenten vorliegen oder in der Komponentifizierung entstehen. Dabei kann das Auffinden beziehungsweise Bestimmen von Kandidaten laut [Atkinson u. a. 2003, : 139] mit Hürden verbunden sein. Diese lassen sich klassifizieren:

- Geeignete Komponentenkandidaten sind nicht auffindbar.
- Kandidaten sind nicht in ein Komponentenmodell integrierbar.
- Es existieren keine geeigneten Kandidaten.

[Atkinson u. a. 2003, : 139] nennt drei wichtige Quellen, in denen Komponentenkandidaten gefunden werden können. Als erstes werden organisationsinterne Komponenten-Repositories angeführt, die eine Sammlung von Komponenten aus verschiedenen Projekten enthalten (vgl. Abschnitt 2.2.5.1). Externe Komponentenmärkte stellen eine zweite Quelle dar, auf denen Fremdkomponenten aus COTS-Produktion gehandelt und erworben werden können (vgl. Abschnitt 2.2.5). Die dritte Quelle sind Bestandsanwendungen, die entweder im Ganzen in Komponenten „verpackt" oder durch eine Komponentifizierung zerlegt werden.
Die Fähigkeit des Komponentenkandidats in ein Komponentenmodell integrierbar zu sein, ist eine Forderung des CBSE (vgl. Abschnitt 2.2.2.8). Da die Universalität von Komponenten jedoch einer idealisierten Vorstellung entspricht und komponentifizierte Komponenten in der

Praxis keine perfekte Übereinstimmung in Bezug der Schnittstellenkompatibilität mit dem Komponentenmodell aufweisen, wird die Auswahl anhand des Aufwands, sie in das Komponentenmodell zu integrieren, getroffen (siehe Abschnitt 2.3.4). Zusätzlich zum Aufwand erwähnt [Atkinson u. a. 2003, S. 149] „[...] nichttechnische Auflagen und Zwänge, die [die] Komponentenauswahl beeinflussen.". Dazu werden „[...] politische und geschäftliche Überlegungen [gezählt], die [die] Verwendung bestimmter Komponenten diktieren."
Falls keine Komponenten auf internen und externen Komponentenmärkten oder Bestandsanwendungen gefunden werden können, entsteht der Zwang zur Neuimplementierung der Komponente.

2.3.2.2 Top-Down- und Bottom-Up-Komponentifizierung

Im CBSE entstehen Komponenten nach einer Bottom-Up-Strategie, in der Komponenten sukzessiv dem Baukastenprinzip folgend in Kompositionen zusammengesetzt werden (siehe Abschnitt 2.2.2.1). Dieses Vorgehen erlaubt die Wiederverwendung von Komponenten. Diese Vorgehensweise kann bei einer Konsolidierung jedoch nicht erfolgen. Dort müssen (monolithische) Anwendungen mit einer rekursiven Anwendung von Arbeitsschritten aufgespaltet und in Komponenten aufgelöst werden. Die Arbeiten enden, wenn die Komponentengranularität (vgl. Abschnitt 2.2.2.10) den Anforderungen entspricht [Atkinson u. a. 2003, S. 138].

2.3.3 Verhandlung

Sind alle Komponentenkandidaten identifiziert und eine Auswahl zusammengestellt, wird in einem Auswahlschritt – der Verhandlung – mit allen Kandidaten eine Untersuchung der Aufwände zur Integration der jeweiligen Komponente in die Neuanwendung geführt. Die Aufwände ergeben sich aus den notwendigen Anpassungen (siehe Abschnitt 2.3.4) der Kandidaten, um sie in das Komponentenmodell zu integrieren, aber auch um sie zur Interaktion mit anderen Komponenten zu befähigen.
Sind die Aufwände zur Integration für sämtliche Kandidaten nicht vertretbar (d.h. kosteneffektiv), sollte ebenfalls eine Entscheidung zur Neuentwicklung einer Komponente getroffen werden [Atkinson u. a. 2003, S. 150].

2.3.4 Integration von Komponenten

Die Wiederverwendung von Komponenten in Neuanwendungen ist ein Stützpfeiler des CBSE. In [Atkinson u. a. 2003, S. 140] wird die Möglichkeit zur Wiederverwendung allerdings eingeschränkt: „... die Zahl der Komponenten, die direkt (d.h. ohne Modifikation) wieder verwendet werden können, [ist] im Regefall eher klein". Die Aussage impliziert, dass in den

meisten Fällen eine Anpassung der Komponente nötig ist, um sie im Komponentenmodell zu integrieren.

2.3.4.1 „Plug and Play" und adaptive Integration

Komponenten, die ohne Modifikation in einem Komponentenmodell wiederverwendet werden können, werden als „plug and play"-Komponenten bezeichnet. Nach [Atkinson u. a. 2003, S. 139] sind es vor allem kleine Komponenten (vgl. Abschnitt 2.2.2.10), die sich zur „plug and play"-Integration eignen.[18]
Bei Komponenten, die nicht ohne Modifikation in ein Komponentenmodell integrierbar sind, ist eine Anpassung – eine Adaption – der Komponente nötig. Eine Adaption lässt sich in eine inversive Adaption und in eine Adaption durch Klebecode unterteilen [Atkinson u. a. 2003, S. 163]. Inversive Adaption verändert die Komponentenimplementierung oder Komponentenschnittstelle, wohingegen der Klebecode zwischen zwei oder mehr Komponenten eingefügt wird und die Komponenten zur Interaktion befähigt. Eine inversive Adaption ist nur bei White-Box-Komponenten möglich. Klebecode eignet sich für alle Komponentenarten (siehe Abschnitt 2.2.2.9). Für detaillierte technische Umsetzungen der inversiven wie auch der Adaption durch Klebecode siehe [Atkinson u. a. 2003, S. 157ff].

[18] Als Beispiele lassen sich abstrakte Datentypen oder Datencontainer anführen.

Kapitel 3

Eclipse und komponentenbasierte Software-Entwicklung

Auf den Seiten dieses Kapitels wird Eclipse unter dem Aspekt des CBSE betrachtet. Dazu werden die Begriffe OSGi (siehe Abschnitt 3.1), Bundle (siehe Abschnitt 3.1.3) und Rich Client Platform (siehe Abschnitt 3.1.2.2) eingeführt.
Eine erste Definition für Eclipse lautet: „Eclipse is an open (IDE) platform for anything, and for nothing in particular" [Eclipse 2006d]. Demnach ist Eclipse eine universelle, offene Plattform, ohne Ausrichtung auf eine bestimmte Fachwelt. Diese Definition beschreibt die technische Seite von Eclipse, die Eclipse-Plattform. Die Definition von Eclipse reicht für diese Arbeit allerdings nicht aus. In dieser Arbeit fasst der Begriff Eclipse die Eclipse-Plattform und die von der Eclipse-Foundation geführten Projekte zusammen.
Die Eclipse Integrated Development Environment (IDE) ist nicht Gegenstand dieser Arbeit, kann aber als Referenz-Implementierung einer Anwendung auf der Eclipse-Plattform angesehen werden. Zusätzlich stellt die IDE Werkzeuge zur Entwicklung von Komponenten des Eclipse-Komponentenmodells bereit.

3.1 OSGi als Komponentenmodell

Das in Eclipse eingesetzte Komponentenmodell basiert auf dem OSGi-Standard der OSGi-Allianz. Die OSGi-Allianz ist ein Standardisierungsgremium, das sich aus verschiedenen Firmen zusammensetzt [OSGi-Alliance 2006a].

3.1.1 Der OSGi-Standard

Der OSGi-Standard spezifiziert ein auf Java basierendes, plattformunabhängiges, dynamisches Komponentenmodell, das eine definierte Anzahl an grundlegenden Diensten anbietet (vgl. Abschnitt 2.2.4). In Abbildung 3.1 wird der konzeptionelle Aufbau von OSGi wiederge-

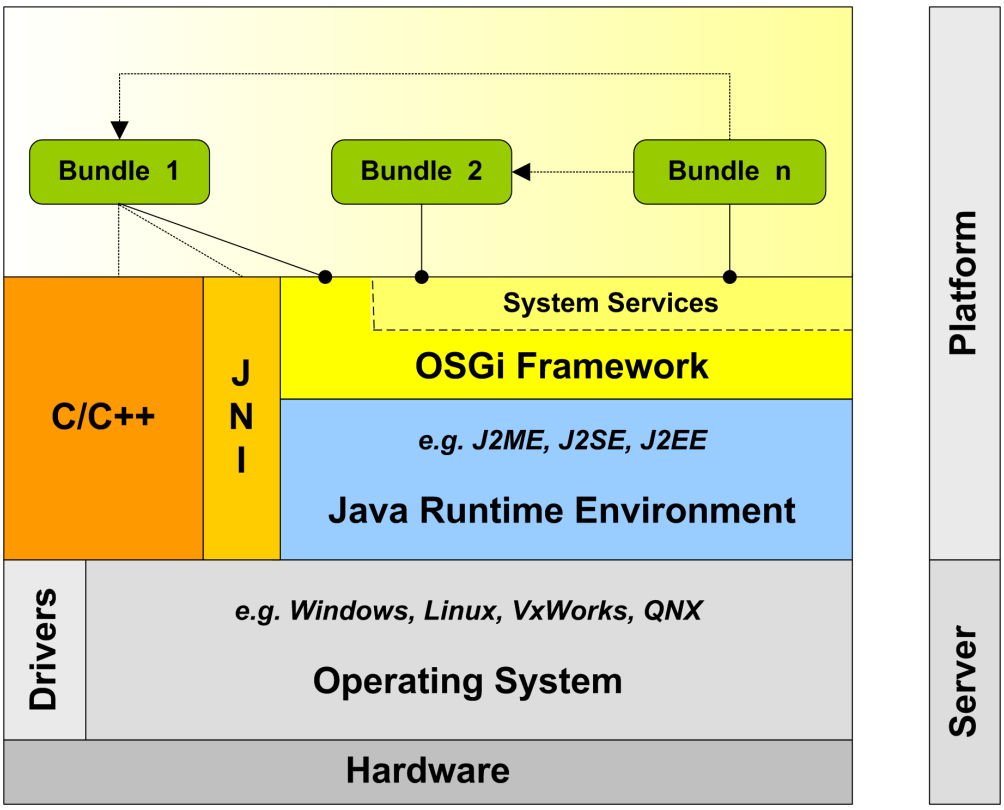

Abbildung 3.1: Konzeptioneller Aufbau des OSGi-Komponentenmodells

(Aus [Wikipedia 2006b])

geben, der in den folgenden Abschnitten erläutert wird.

3.1.2 Equinox als Komponenten-Rahmenwerk

Equinox ist die Bezeichnung der Referenz-Implementierung der vierten Version des OSGi-Standards und bildet gleichzeitig den Kern der Eclipse-Plattform. Die Implementierung bietet eine Laufzeitumgebung und Dienste an und kann daher als Komponenten-Rahmenwerk bezeichnet werden (vgl. Abschnitt 2.2.4).

3.1.2.1 Laufzeitumgebung

Der OSGi-Standard unterteilt die Laufzeitumgebung in vier Schichten [OSGi-Alliance 2006a, S. 9–12]. Die Schichten gliedern sich nach technischen Kriterien:

- Sicherheitsschicht („Security Layer"): Die Sicherheitsschicht setzt die Anforderungen eines Komponentenmodells unter Sicherheitsaspekten um. Sie reglementiert den Zugriff zwischen Komponenten und realisiert darüber hinaus einen Zertifizierungsmechanismus, womit die Authentizität von Komponenten sichergestellt wird [OSGi-Alliance 2006a, S. 19–33].

- Modulschicht („Module Layer"): Die Modulschicht erbringt die Forderung nach Abgeschlossenheit und Unabhängigkeit (vgl. Abschnitt 2.2.2.2 und 2.2.2.1) von Komponenten im Kontext. Diese Schicht spezifiziert darüber hinaus den konzeptionellen Aufbau von Komponenten [OSGi-Alliance 2006a, S. 34–84].

- Lebenszyklusschicht („Life Cycle Layer"): Die Lebenszyklusschicht basiert auf der (optionalen) Sicherheitsschicht und der Modulschicht und definiert den Lebenszyklus einer Komponente. Zum Lebenszyklus zählen das Installieren und Deinstallieren von Komponenten im Komponentenmodell, das dynamische Laden, Entladen und Auffinden von installierten Komponenten und das Aktualisieren von Komponenten [OSGi-Alliance 2006a, S. 85–108].

- Dienstschicht („Service Layer"): Die Dienstschicht bestimmt die Interaktion von Komponenten im Kontext. Sie spezifiziert den Aufbau von Schnittstellen und die Abhängigkeiten zwischen Komponenten. Referenzen auf Komponentendienste können zur Laufzeit bei der `ServiceRegistry` erhalten werden, wobei verschiedene Versionen einer Komponente verwaltet und aufgelöst werden können [OSGi-Alliance 2006a, S. 109–124].

3.1.2.2 Horizontale und vertikale Dienste in Form einer Rich Client Platform

Neben den vier Schichten der Laufzeitumgebung (siehe Abschnitt 3.1.2.1), spezifiziert der OSGi-Standard eine Reihe von grundlegenden Komponentendiensten (vgl. Abschnitt 2.2.4.2), welche die genannten Schichten der Laufzeitumgebung des OSGi-Rahmenwerks realisieren. Eine Beschreibung dieser Dienste findet sich in [OSGi-Alliance 2006a, S. 201–272]. Zusätzlich enthält der Standard verschiedene horizontale Dienste [OSGi-Alliance 2006b]. Vertikale Dienste sind im OSGi-Standard jedoch nicht enthalten.
Die auf OSGi aufbauende Eclipse-Plattform bündelt für Desktop-Anwendungen vertikale Dienste und Komponenten innerhalb der Rich Client Platform (RCP). Die zur RCP zählenden Komponenten realisieren eine Fensterbibliothek (Standard Widget Toolkit (SWT)), auf SWT basierende Benutzungselemente (JFace) und einen Desktop-Anwendungsrahmen, der Sichten und Perspektiven anbietet. „The minimal set of plug-ins needed to build a rich client application is collectively known as the Rich Client Platform" [Eclipse 2006a]. Die Grundmenge der beschriebenen RCP-Komponenten ist somit lediglich eine Vorauswahl. Eine RCP kann um beliebige Komponenten ergänzt werden [McAffer u. Lemieux 2005].

3.1.3 Bundle als Komponente

In der OSGi-Terminologie wird eine Komponente als Bundle bezeichnet.[19] Ein Bundle setzt sich aus einer JAR-Datei und Meta-Informationen, dem Bundle-Manifest, zusammen [OSGi-Alliance 2006a, S. 33ff]. JAR-Dateien implizieren, dass Bundle in Java realisiert sind und somit das Konzept der Objektorientierung nutzen. Es ist jedoch möglich, mit anderen Programmiersprachen und Konzepten entwickelte Programmlogik in Bundle zu „verpacken". Ein Beispiel wird in Abbildung 3.1 durch das Java Native Interface (JNI) angedeutet.
Die Kennzeichnung einer Komponente erfolgt durch Bezeichner aus hierarchischen Namensräumen (vgl. Abschnitt 2.2.2.5), eine Versionierung durch Versionsnummern nach der „Major.Minor.Micro"-Notation (vgl. Abschnitt 2.2.2.6) [McAffer u. Lemieux 2005, S. 456].
Ein Zugriff auf Klassen einer Komponente ist nur über die angebotenen Schnittstellen der Komponente möglich. Angebotene, aber auch geforderte Schnittstellen, können im OSGi-Bundle entweder deklarativ in XML-Dateien oder programmatisch spezifiziert werden. Die folgende Tabelle (siehe Tabelle 3.1) listet diesen Sachverhalt schematisch auf.

	Geforderte Schnittstellen	Angebotene Schnittstellen
deklarativ	Erweiterungen („Extensions")	Erweiterungspunkte („Extension Points")
programmatisch	Geforderte Pakete/-Bundle („Required Package/Bundle")	Angebotene Pakete („Exported Packages")

Tabelle 3.1: Aufteilung von geforderten und angebotenen Schnittstellen im OSGi-Standard

Die Spezifikation der programmatischen, geforderten Schnittstellen kann als Abhängigkeiten auf andere Bundle oder in Form eines Java-Pakets angegeben werden. Die Angabe als Java-Paket ist flexibler, da keine Abhängigkeiten auf andere Komponenten entstehen, sondern lediglich auf Komponentenschnittstellen (vgl. Abschnitt 2.2.2.8) [McAffer u. Lemieux 2005, S. 470]. Darüber hinaus erlauben es Paket-Abhängigkeiten, die Aufteilung von Java-Paketen über verschiedene Bundle vorzunehmen, ohne die abhängige Komponente anzupassen.
Beide Abhängigkeiten können als optional markiert werden. Dadurch kann die Komponente von der Laufzeitumgebung aktiviert werden, auch wenn diese Abhängigkeit nicht erfüllt ist. Abhängigkeiten sind ausschließlich auf die angebotenen Pakete einer Komponente erlaubt und möglich. Ist ein angebotenes Paket als intern gekennzeichnet, so signalisiert dieses, dass auf Interna der Komponente zugegriffen wird und implizite Schnittstellen genutzt werden. Dieser Zugriff ist aus Sicht des CBSE nicht erwünscht (siehe Abschnitt 2.2.2.7), wurde

[19] Die Bezeichnung Plug-In wird äquivalent verwendet.

aber aus verschiedenen Gründen aufgebrochen (vgl. [McAffer u. Lemieux 2005, S.471]). Zugriffsrechte erlauben Schnittstellenzugriffe auf bestimmte Komponenten zu beschränken.
Im Gegensatz zu deklarativen Schnittstellen können Abhängigkeiten auf programmatische Komponentenschnittstellen versioniert werden. Zur Versionierung wird dieselbe Notation wie bei der oben beschriebenen Komponentenversionierung verwendet. Die Versionen der Abhängigkeiten können durch ein Schiebefenster eingeschränkt werden.
Eine technisch erzwungene Trennung zwischen Komponentenschnittstelle und Komponentenimplementierung erfolgt nicht (vgl. Abschnitt 2.2.2.7). Dies kann für programmatische und deklarative Schnittstellen jedoch durch eine Aufspaltung der Komponente in eine Schnittstellen-Komponenten und eine Implementierungs-Komponente erfolgen.

3.1.3.1 Fragmente und Bundle

Eine Besonderheit in Bezug zum CBSE sind Fragmente des OSGi-Standards. Fragmente haben denselben Aufbau eines Bundle, können durch Bundle jedoch nicht als Abhängigkeiten spezifiziert werden. Fragmente erweitern ein Bundle und sind diesem einen Bundle immer zugeordnet. [McAffer u. Lemieux 2005, S. 452] nennen zwei Einsatzszenarien für Fragmente:

- Plattformabhängige Inhalte: Obwohl Java eine plattformunabhängige Programmiersprache ist, kann die Notwendigkeit entstehen, plattformabhängige Programmteile in eine Komponente zu integrieren.[20] Damit die Komponentengranularität minimal ist, kann für jede Plattform ein Fragment geliefert werden.
- Lokalisierung: Dieselbe Argumentation trifft auf Programmübersetzungen zu. Verschiedene Sprachpakete werden auf Fragmente verteilt. Nicht benötigte Sprachpakete entfallen.

Das erste Szenarium impliziert, dass die durch das Fragment unterstützte Plattform spezifizierbar sein muss. Dieses ist sowohl für Fragmente, als auch für Bundle möglich.

3.2 Eclipse als Komponentenmarkt

Neben dem im ersten Abschnitt dieses Kapitels dargestellten technischen Aspekt von Eclipse in Bezug auf CBSE, soll in diesem Abschnitt der Eclipse-Komponentenmarkt betrachtet werden (vgl. Abschnitt 2.2.5).
Die Entwicklung von Eclipse wird durch die Eclipse-Foundation, eine Non-Profit Stiftung, gesteuert. Diese Stiftung bildet einen Zusammenschluss von verschiedenen Firmen. Die

[20] Wie beispielhaft in Abbildung 3.1 für Bundle 1 dargestellt.

Stiftung betreut mehrere Top-Level-Projekte, die horizontalen Märkten zugeordnet werden.[21] Die innerhalb dieser Projekte entstehenden Bundle unterliegen der Eclipse Public License (EPL), welche die Nutzung der Bundle reglementiert. Die kommerzielle Nutzung der Bundle ist von der EPL erlaubt. Ein Bundle dieser Projekte ist als Glass-Box-Komponente einzustufen (siehe Abschnitt 2.2.2.9). Der Quelltext ist verfügbar, eine Veränderung allerdings nur in einem definierten Prozess möglich. Damit verschiebt sich die Wartungs- und Pflegeverantwortlichkeiten nicht. Aus technischer Sicht ist eine Komponente hingegen auch als Black-Box-Komponente realisierbar.
Im Gegensatz zu den von [Heineman 2001] beschriebenen Komponentenmärkten, findet im Eclipse-Komponentenmarkt kein kommerzieller Komponentenhandel statt. Ein Kauf und Verkauf von Komponenten erfolgt praktisch nicht.[22]
Ein Komponenten-Repository, das die verfügbaren Komponenten und Komponentenschnittstellen dokumentiert und zur Nutzung durch Dritte sammelt und bereit stellt (vgl. Abschnitt 2.2.5.1), existiert für Eclipse zum Zeitpunkt dieser Arbeit mit dem Bundle-Repository der OSGi-Allianz.[23] Darin werden kommerzielle und freie Bundles referenziert und auf technischer Ebene dokumentiert. In diesem Repository finden sich sowohl Black-Box-Komponenten als auch White-Box-Komponenten.
Eine Zertifizierung von OSGi-Komponenten ist momentan nicht möglich (vgl. Abschnitt 2.2.5.2). Die OSGi-Allianz zertifiziert lediglich Implementierungen des Komponenten-Rahmenwerks nach dem OSGi-Standard. Dieses schließt aber keine Bundle ein.[24] Somit ist lediglich eine Herstellerauthentifizierung von Komponenten durch digitale Signaturen möglich (vgl. Abschnitt 3.1.2.1).

3.3 Fazit von Eclipse in Bezug auf CBSE

Eclipse und OSGi liefern ein Komponentenmodell und einen Komponentenmarkt und ermöglichen dadurch die Wiederverwendung von Komponenten (vgl. Abschnitt 2.2). Aus Sicht des CBSE finden sich allerdings einige Kritikpunkte:

- Trennung von Komponentenimplementierung und Schnittstelle: Da Bundle nicht explizit zwischen der Komponentenschnittstelle und Komponentenimplementierung trennen, ist ein Austausch durch alternative Komponentenimplementierungen erschwert (siehe Abschnitt 3.1.3). Technische Mechanismen, die eine solche Trennung realisieren könnten, sind mit Fragmenten zwar vorhanden, aber als solche nicht vorgesehen (vgl. Abschnitt 3.1.3.1).

[21] siehe `http://www.eclipse.org/projects/`
[22] Trotzdem ergibt sich ein finanzieller Vorteil für Komponentenhersteller durch Wiederverwendung.
[23] siehe `http://bundles.osgi.org/browse.php`
[24] siehe `http://www.osgi.org/osgi_technology/compliance.asp?section=2`

- Versionierung von Erweiterungspunkten: Die fehlende Versionierung der deklarativen Schnittstellen (Erweiterungspunkten) bereitet Schwierigkeiten bei einer Komponentenevolution (siehe Abschnitt 3.1.3). Bietet eine Komponente Erweiterungspunkte an, so entfällt die Fähigkeit, verschiedene Versionen einer Komponente simultan zu laden (vgl. Abschnitt 3.1.2.1). Zum Zeitpunkt dieser Arbeit scheinen keine Pläne zu existieren, eine Versionierung für Erweiterungspunkte aufzunehmen.

- Zertifizierung von Komponenten: Neben digitalen Signaturen (siehe Abschnitt 3.2) ist keine Zertifizierung von Komponenten möglich. Allgemeine und umfassende Qualitätsstandards fehlen (vgl. Abschnitt 2.2.5.2).[25]

- Sprachunabhängigkeit: Komponenten des OSGi-Standards werden auf einer Java Runtime Environment (JRE) ausgeführt und sind daher auf die unterstützten Sprachen dieser Plattform limitiert (vgl. Abbildung 3.1). Andere Programmiersprachen können jedoch durch JNI „verpackt" werden, womit eine prinzipielle Universalität der Komponenten in Bezug zu Programmiersprachen und Programmiermodellen besteht.

[25] Die Eclipse-Foundation definiert für ihre Top-Level-Projekt lediglich Programmierstandards.

Kapitel 4

Anforderungsanalyse

Bevor im nächsten Kapitel die Umsetzung der grundlegenden Erkenntnisse auf den in der Einleitung erwähnten Prototypen erfolgen kann (siehe 5. Kapitel), wird in diesem Kapitel die Ausgangssituation und damit der Zustand des Prototypen zu Beginn der Konsolidierung dargelegt. Die Kenntnis des Prototyps ist zum Verständnis des Umsetzungskapitels nötig. Im zweiten Teil dieses Kapitels wird der Soll-Zustand und somit das Ziel der Konsolidierung nach Projektabschluss festgelegt.
Diese Anforderungsanalyse unterscheidet sich vom üblichen Aufbau einer Anforderungsanalyse. Im Kontext einer Konsolidierung liegt der Fokus nicht auf der Implementierung von funktionalen Anforderungen, sondern verstärkt auf technischen, nicht-funktionalen Aspekten des Systems.

4.1 Ist-Zustand

Der Ist-Zustand setzt sich aus der Historie der VMC (Abschnitt 4.1.1), der Erläuterung der fachlichen und technischen Gesamtarchitektur (Abschnitt 4.1.2 und 4.1.3), einer Konkretisierung der fachlichen und technischen Architektur für die Benutzungsoberfläche (Abschnitt 4.1.4 und 4.1.5) und einer Übersicht über die Quelldateien zusammen (Abschnitt 4.1.6). Abschließend wird ein Fazit des Ist-Zustands gezogen (Abschnitt 4.1.7).

4.1.1 Historie der VMC

Die zur Konsolidierung vorgesehene Bestandsanwendung trägt die Bezeichnung Versant Monitoring Console (VMC) und wurde zwischen 2001 und 2003 durch ein italienisches Unternehmen im Auftrag der Versant Corp. realisiert [Versant 2003]. In diesem Kontext entstanden zwei Versionen der VMC, wobei die zweite Version eine erweiterte Ausbaustufe gegenüber der ersten Version darstellt. Seit 2003 wurden keine weiteren Arbeiten in jeglicher

Form an der VMC vorgenommen. Die Anwendung wurde Kunden weder zum Kauf angeboten, noch anderweitig vertrieben. Eine Anwendung mit ähnlichem, aber eingeschränkten Funktionsumfang liegt dem Versant Objekt-Datenbanksystem (VOD) bei. Im Rahmen dieser Arbeit konnte keine abschließende Entstehungsgeschichte der VMC und der mitgelieferten Monitoring Console rekonstruiert werden.

4.1.2 Fachliche Gesamtarchitektur der VMC

Die VMC ist eine Arbeitsplatzanwendung, die zur Beobachtung verschiedener Versant Objekt-Datenbanksysteme genutzt werden kann. Zu jedem Datenbanksystem lassen sich die Eigenschaften der jeweiligen Datenbanken anzeigen. Unter Eigenschaften sind hier alle Arten von Meta-Informationen über den Zustand der Datenbank zu verstehen. Einige Beispiele wären der Füllstand der Datenbank, die Anzahl der momentanen Verbindungen zur Datenbank oder andere Statistiken über den Laufzeitzustand der Datenbank. Über all diese Informationen lassen sich grafische Auswertungen erstellen, welche die Werte zueinander und in einen zeitlichen Kontext versetzen.
Neben der reinen Beobachtung einer Datenbank beinhaltet die Anwendung die Möglichkeit, Warnmeldungen über kritische Zustände zu erzeugen und an einzelne Benutzer oder Benutzergruppen zu verschicken. Mit diesem Funktionsumfang richtet sich die Anwendung in erster Linie an Datenbankadministratoren, die damit die Verfügbarkeit und Performance des Datenbanksystems überwachen.

4.1.3 Technische Gesamtarchitektur der VMC

Die Anwendung unterteilt sich in zwei Bestandteile: Eine Benutzungsoberfläche und einen Agenten. Die Benutzungsoberfläche dient zur grafischen Visualisierung der Informationen. Die Bezeichnung „Agent” stammt aus der Terminologie des Simple Network Management Protocol (SNMP) und ist eine in Java geschriebene Kommandozeilen-Anwendung, die auf demselben Host wie das Datenbanksystem gestartet sein muss. Der Grund für diese Einschränkung liegt darin, das Datenbanksystem möglichst wenig zu beeinflussen und Messungen nicht zu verfälschen. Daher erfolgt der Zugriff auf das Datenbanksystem einzig über das (lokale) Dateisystem und macht so den lokalen Agenten nötig. Besonders erwähnenswert ist die Tatsache, dass nur eine Untermenge der gesamten Meta-Informationen des Datenbanksystems aus dem Dateisystem entnommen werden können. Einige Informationen werden ausschließlich im flüchtigen Speicher des Datenbanksystems gehalten. Außerdem erlaubt der Agent das Beobachten der verfügbaren Datenbankinformationen, während die Benutzungsoberfläche inaktiv ist. Es ermöglicht damit eine persistente, ununterbrochene Beobachtung des Datenbanksystems.
Die bereits erwähnten Warnungen (vgl. Abschnitt 4.1.2) werden ebenfalls vom Agenten

erzeugt und versandt. Neben E-Mail wird die SNMP-Schnittstelle als Kanal für Warnungen angeboten. In diesem Zusammenhang wird von SNMP-Traps gesprochen. Der Agent ist einerseits eigenständig lauffähig, andererseits aber auch im Verbund als Sub-Agenten eines Master-Agenten. Neben SNMP stellt der Agent außerdem eine Hypertext Transfer Protocol (HTTP)-Schnittstelle zur Verfügung, über die ein Zugriff auf die Daten der SNMP-Schnittstellen per Webbrowser möglich ist.
Als Kommunikationsprotokoll zwischen Agenten und Benutzungsoberfläche kommt Remote Method Invocation (RMI) zum Einsatz. Hierbei ist die Tatsache zu erwähnen, dass der Agent als auch die Benutzungsoberfläche gegenseitige RMI-Referenzen aufeinander halten. Somit besteht eine bidirektionale Abhängigkeit zwischen Agent und Benutzungsoberfläche. Diese Besonderheit wird im Abschnitt 4.2.2 aufgegriffen.

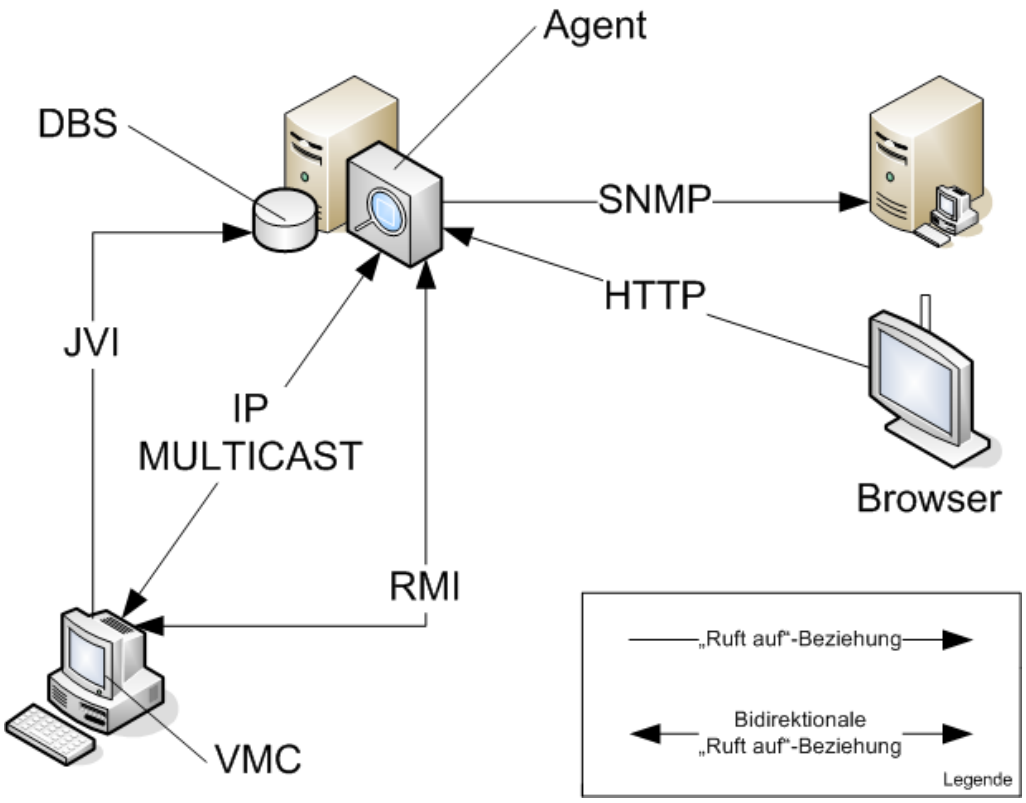

Abbildung 4.1: Netzwerk-Diagramm der VMC

Die Benutzungsoberfläche ist eine Java Swing-Anwendung und somit plattformunabhängig. Allerdings nutzt sie die Schnittstelle des Java Versant Interface (JVI), um eine Verbindung mit dem Datenbanksystem herzustellen. JVI ist keine rein in Java realisiertes API. Es basiert auf einem plattformspezifischen Anteil, der nicht für alle Betriebssysteme verfügbar ist. Dadurch

verliert die VMC die plattformunabhängigkeit. Neben RMI und JVI wird IP Multicast (MULTICAST) als drittes Kommunikationsmedium verwendet. In der Benutzungsoberfläche dient es dem dynamischen Auffinden von Agenten. Im Agenten wird es zur Benachrichtigung von Benutzungsoberflächen verwendet. Das Netzwerk-Diagramm (siehe Abbildung 4.1) stellt die grundsätzlichen Kommunikationskanäle der VMC-Architektur grafisch dar.

4.1.4 Fachliche Architektur der Benutzungsoberfläche der VMC

Die Benutzungsschnittstelle ist eine Arbeitsplatzanwendung, die sich in drei Hauptfenster aufteilen lässt. Im folgenden Screenshot (vgl. Abbildung 4.2) sind diese drei Fenster dargestellt und durchnummeriert.

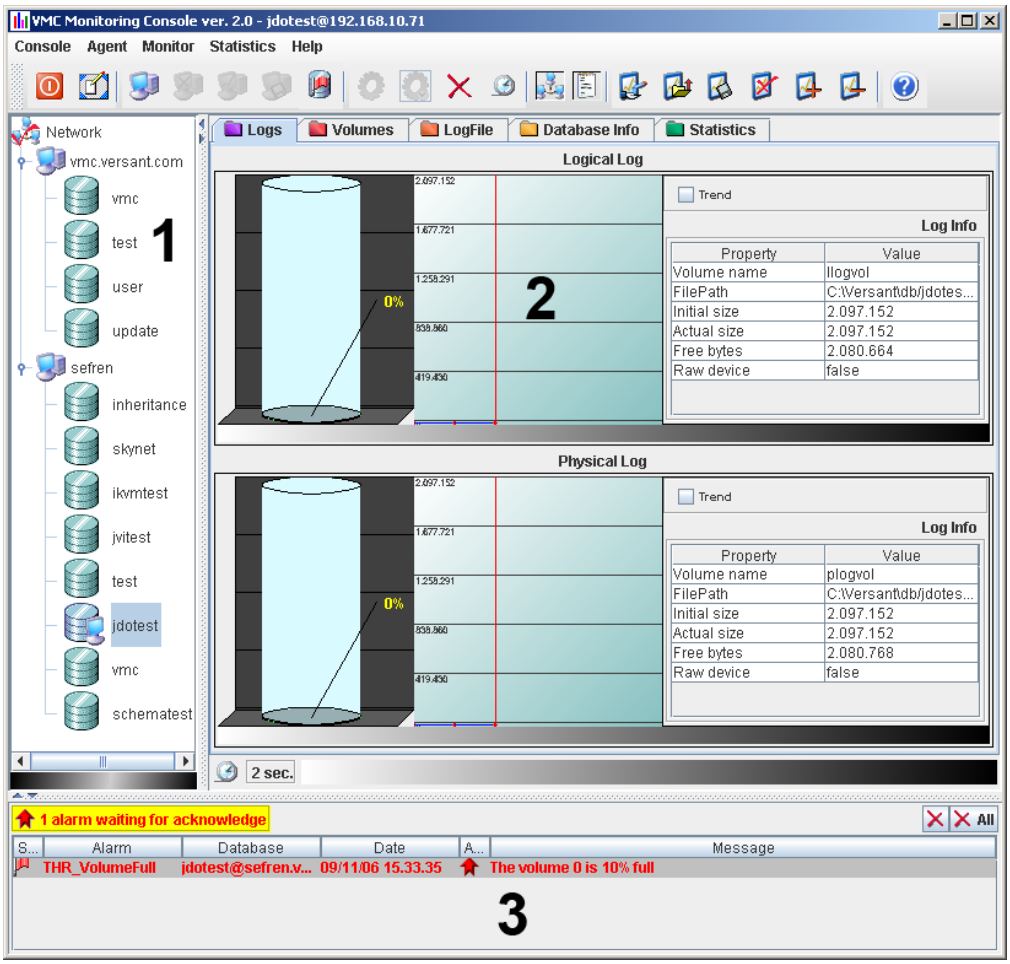

Abbildung 4.2: Screenshot der ursprünglichen Benutzungsoberfläche des Altsystems

Fenster 1 stellt Datenbanksysteme in einer Baumstruktur dar, dessen Kind-Knoten die einzelnen Datenbanken zeigen. Für jede dieser Datenbanken kann das Beobachten aktiviert werden. Ist das Beobachten für eine Datenbank aktiv, so werden in Fenster 2 die Informationen grafisch und textbasiert visualisiert. Fenster 3 zeigt Warnungen an, die durch Regeln definiert sind.

4.1.5 Technische Architektur der Benutzungsoberfläche der VMC

Die Fenster der Benutzungsoberfläche wurden als Subklassen der Klasse JPanel der Swing-Fensterbibliothek realisiert. Zur Entkopplung der Fenster (Präsentation) vom Datenmodell der VMC wird das Vermittler-Entwurfsmuster genutzt [Gamma u. a. 1995, S. 273-282]. Damit wird das Verhalten zwischen Präsentation und Datenmodell im Vermittler gekapselt.

4.1.6 Ausgangszustand der Quelldateien

Die vorherigen Abschnitte wurden auf Grundlage des Ausgangszustands der Bestandsanwendung erarbeitet. Deshalb soll in diesem Abschnitt ein Überblick über den vorgefundenen Zustand der zur Anwendung gehörenden Quellen und Dokumentation gegeben werden.
Wie eingangs erwähnt, wurde die Anwendung im Auftrag von Versant erstellt. Im Versionsverwaltungssystem wurden Quellen für die Versionen 1.1 und 2.0 aufgefunden. Alle Betrachtungen in dieser Arbeit beziehen sich auf die zweite Version der Versant Monitoring Console, da diese Version eine erweiterte Ausbaustufe der Anwendung repräsentiert. Eine ersten Prüfung ergab, dass keine Historie zu den Quelltext-Dateien im Versionsverwaltungssystem vorhanden ist. Die Ursache begründet sich in der externen Entwicklung der VMC (siehe Abschnitt 4.1.1). Die Quelltexte wurden nach der Übergabe an Versant einmalig ins Versionsverwaltungssystem importiert. Dadurch ist eine Gewinnung von zusätzlichen Informationen über die Bestandsanwendung aus dem Versionsverwaltungssystem ausgeschlossen (vgl. [Wu u. a. 2004]).
Die Anwendung wurde in der Programmiersprache Java entwickelt und nutzt somit das Programmierkonzept der Objektorientierung. Das zur Entwicklung genutzte SUN Microsystems Java Development Kit (JDK) liegt in der Version 1.3 vor. Der gesamte Quelltext befindet sich unterhalb der Java-Paket-Struktur `it.versant`. Neben den Quellen besteht die Anwendung aus verschiedenen JAR-Dateien, die von diversen Drittanbietern stammen (eine vollständige Liste findet sich in Tabelle 5.1). Die Lizenzen dieser JAR unterteilen sich in kommerzielle und freie Modelle.
Neben den eigentlichen Quelltexten wurde ein Benutzerhandbuch vorgefunden, das die Funktionalität, und in Auszügen auch architektonische Aspekte der VMC, aus Benutzersicht beschreibt. Neben dieser sehr abstrakten und informalen Dokumentation, enthielt das Versionsverwaltungssystem Fragmente von Unified Modeling Language (UML)-Diagrammen. Als dritte Quelle für technische Informationen kann die initial erzeugte JavaDoc-Dokumentation

betrachtet werden.
Eine Dokumentation, die über diese drei genannten Aspekte hinausgeht, besteht nicht. Auf eine Nachdokumentation der Anwendungsfunktionalität durch formale Anwendungsdiagramme („Use Cases") wird nach einer Kosten-Nutzenschätzung verzichtet. Für Fragen kann auf die ursprünglichen Entwickler der Anwendung nicht zurückgegriffen werden. Die Zusammenarbeit ist beendet. Eine Wiedergewinnung von Anwendungswissen durch eine Begleitung der Benutzer beim Einsatz der Anwendung, ist aus Mangel an Nutzern nicht möglich.
Unit-Tests oder funktionale Tests, die eine Prüfung der Funktionalität oder des Verhaltens der Anwendung zugelassen hätten, sind ebenfalls nicht vorhanden. Ein Testplan zur manuelle Kontrollen steht ebenfalls nicht zur Verfügung.
In der Distribution der VMC werden Agent und Benutzungsoberfläche in einem gemeinsamen JAR verpackt. Somit wird, ganz gleich ob der Agent oder die Benutzungsoberfläche genutzt werden soll, immer beides installiert. Da die beiden Teile aber einzeln lauffähig sind, stellt dies, abgesehen von erhöhtem Speicherplatzverbrauch, vorerst kein Problem dar.

4.1.7 Fazit des Ist-Zustands der VMC

Zusammenfassend kann der Ist-Zustand der VMC als typisch für eine gewachsene Bestandsanwendung angesehen werden. Die Architektur verletzt diverse Entwurfsparadigmen, verschiedene Aspekte entsprechen nicht mehr aktuellen Standards (vgl. Abschnitt 4.1.3 und 4.1.4). Aus Sicht des CBSE stellt die starke Kopplung zur Übersetzungs- sowie zur Laufzeit ein Problem dar (siehe Kapitel 2.2.2). Des Weiteren bündelt das Vermittler-Entwurfsmuster die Kommunikation zwischen Präsentation und Datenmodell in einer einzigen (monolithischen) Klasse. Dieser hohe Grad an Kapselung muss bei einer Dekomposition aufgebrochen werden (siehe Abschnitt 4.1.5).
Darüber hinaus ist die Dokumentation der Anwendung lückenhaft oder veraltet, was ein Verständnis des Altsystems erschwert. Dieser Zustand ist bei der Erstellung des Projektplans jedoch berücksichtigt, weshalb Schwierigkeiten durch zusätzliche Pufferzeit abgemildert sind.

4.2 Soll-Zustand

Der Soll-Zustand der Anforderungsanalyse gliedert sich in einen generellen (projektweiten) Soll-Zustand (Abschnitt 4.2.1), der Beschreibung der angestrebten Gesamtarchitektur (Abschnitt 4.2.2), der Architektur für Agent und Benutzungsoberfläche (Abschnitt 4.2.3 und 4.2.4) und einem abschließenden Fazit (Abschnitt 4.2.5).
Bevor die Festlegung des Soll-Zustands aufgezeigt wird, soll auf die Ziele (siehe Kapitel 1.2) dieses Konsolidierungsprojekts hingewiesen werden, da sie den Soll-Zustand maßgeblich beeinflussen.
Manche Forderungen gehen deutlich über den Rahmen dieser Arbeit und die dokumentier-

ten Änderungen des Umsetzungskapitels (siehe 5 Kapitel) hinaus und sind für spätere Stufen (vgl. [Zukunft u. Raasch 2004]) der Konsolidierung angesetzt. Trotzdem werden sie hier der Vollständigkeit halber mit aufgeführt.

4.2.1 Genereller Soll-Zustand der VMC

In diesem Abschnitt werden generelle Anforderungen an die Anwendung nach Abschluss der Konsolidierung festgelegt. Oberste Ziele sind eine Anpassungsfähigkeit des Anwendungszuschnitts auf Kundenwünsche und eine nahtlose Einbettung in das Vitness (VITNESS) Konsolidierungsprojekt zu nennen (vgl. Unterabschnitt 2.2.7.2). Dabei sollen keine umfangreichen Änderungen an der fachlichen Architektur der Bestandsanwendung durchgeführt werden. Beide Ziele lassen sich durch die Auflösung der Bestandsanwendung in Komponenten erreichen (siehe Abschnitt 2.2.2). Da bereits eine Auslieferung der ersten Stufe an Kunden als Meilenstein vorgesehen ist, ist eine Komponentenaktualisierung über das Internet nötig.
Der richtungsweisende Entschluss für das gesamte Konsolidierungsvorhaben ist die Entscheidung zur Nutzung des OSGi-Standards sowie der Eclipse-Plattform. Eine Nutzung von vorgefertigten Komponenten dieses Komponentenmodells ist erstrebenswert. Der Wiederverwendungsgedanke des CBSE ist somit ein grundlegendes Ziel im Konsolidierungsprojekt. Alle Arbeiten sollen vor diesem Hintergrund unternommen werden und müssen auf dieses Ziel abgestimmt sein.
Eine weitere Forderung der Vertriebsabteilung ist die Verbesserung der Anwendung in Bezug auf Benutzungsfreundlichkeit und Design. Eine Anpassung an Standards ist nötig. Eine genaue Spezifikation dieser Forderung findet sich im Abschnitt 4.2.4.
In Bezug auf den gesamten Produktionsprozess ist eine Einbettung in den Entwicklungs-, Übersetzungs-, Paketierungs- und Qualitätssicherungsprozess von Versant erforderlich. Da diese Teilprozesse bei der Produktion stark automatisiert sind, stellt sich diese Anforderung auch an die zu schaffenden Einbindungen. Eclipse bietet die Möglichkeit, die Implementierung des Build-Managements innerhalb der IDE als auch aus einer automatisierten Umgebung zu nutzen. Deshalb soll sie als Grundlage für den Übersetzungs- und Paketierungsprozess genutzt werden. In der Folge entfallen Redundanzen, die durch einen gesonderten Übersetzungs- und Paketierungsprozess entstehen würden. Die Qualitätssicherung ist hingegen völlig offen und muss in späteren Stufen definiert werden. Für die erste Stufe hat man sich lediglich auf eine manuelle Qualitätssicherung verständigt, die anhand eines Testplans zum Ende dieses Entwicklungszyklusses anläuft. Diese Entscheidung führt zum Bruch mit grundlegenden Prinzipien der Qualitätssicherung (vgl. [Balzert 1998]), ist vor dem Hintergrund beschränkter Ressourcen jedoch akzeptiert.

4.2.2 Gesamtarchitektur der VMC

Agent und Benutzungsoberfläche sollen im Anschluss an die Konsolidierung mit Ausnahme von JVI weiterhin plattformunabhängig sein. Ein Wechsel des Programmierkonzepts oder der Programmiersprache ist nicht geplant. Es wird weiterhin das SUN Microsystems JDK genutzt. Ein Wechsel auf die Version 1.4 wird durchgeführt, da diese die momentan vorgeschriebene Minimalversion für Versant-Produkte darstellt.
Durch die Nutzung der Eclipse-Plattform kommen neue Abhängigkeiten in der Benutzungsoberfläche hinzu. Die aktuelle Version der Eclipse-Plattform liegt in der Version 3.2 vor. Diese Version stellt bis zum Erscheinungstermin der Nachfolgeversion die minimale Abhängigkeit dar. Da allerdings ein Wechsel auf eine neuere Version (3.3) gegebenenfalls Probleme verursachen könnte – und definitiv Zeit in Anspruch nimmt, um das reibungslose Zusammenspiel zu verifizieren – wird ein Wechsel der minimalen Abhängigkeit mit dem Erscheinen des ersten Release-Kandidaten der neuen Version vorgesehen. Der voraussichtliche Erscheinungstermin dafür liegt im Frühjahr 2007.[26]
Da die Eclipse-Plattform nicht nur aus einem einzigen Projekt besteht (siehe Abschnitt 3.2), ist eine genauere Angabe der Abhängigkeit nötig. In dieser Konsolidierung dürfen ausschließlich Komponenten der vertikalen Fachwelt eingesetzt werden, die unter dem Eclipse Top-Level-Projekt zusammengefasst sind. Weitere Eclipse-Abhängigkeiten sind in der ersten Stufe nicht geplant. Zu einem späteren Zeitpunkt werden weitere Projekte auf ihre Vitness-Tauglichkeit evaluiert, um die Auswahl an COTS-Komponenten zu erhöhen.
Architektonische Abhängigkeiten zwischen dem Agent und der Benutzungsoberfläche sind in der ersten Stufe nicht erlaubt und müssen entfernt werden. Agent sowie Benutzungsoberfläche müssen eigenständig übersetzbar sein, das heißt, sie dürfen keine Abhängigkeiten zum Kompilierungszeitpunkt aufweisen. Die Kommunikation zwischen Agenten und Benutzungsoberfläche wird in der ersten Stufe allerdings weiterhin auf RMI basieren. In späteren Projektzyklen soll diese Technik, sowie JVI, durch Cobra (COBRA) als einziges Protokoll abgelöst werden, um die Anzahl der Schnittstellen zum Datenbanksystem, und somit die Komplexität der Anwendung, zu minimieren. COBRA ist eine weiteres JAVA-API des VOD, das Remote Procedure Calls (RPC) bereit stellt und als Grundlage für die Java Data Objects (JDO)-Implementierung dient. COBRA ist rein in JAVA realisiert und damit plattformunabhängig.

4.2.3 Der Agent der VMC

Wie im letzten Abschnitt erwähnt, sollen die Abhängigkeiten zwischen Agent und Benutzungsoberfläche beseitigt werden. Der Agent wird dazu in eigenständige Komponenten umgewandelt, die keine Abhängigkeiten zur Benutzungsoberfläche aufweisen dürfen. Die Kopplung des Agenten über RMI an die Benutzungsoberfläche wird in der erste Stufe entfernt

[26] Die Erscheinungstermine lagen in den Jahren seit der ersten Version immer im zweiten Quartal (vgl. `http://www.eclipse.org`).

(siehe Abschnitt 4.1.3).
JVI wird als eine neue Abhängigkeit im Agenten eingeführt. Diese Entscheidung weicht allerdings von der ursprünglichen Entwurfsentscheidung ab, dem Agenten keinen Zugriff auf das Datenbanksystem zu erlauben (vgl. Abschnitt 4.1.3). Das JVI-API wird lediglich zur Prüfung auf eine gültigen Lizenz beim Programmstart genutzt und erzeugt daher nur einmalig Last. Eine Alternative zu dieser Methode besteht darüber hinaus nicht.

4.2.4 Die Benutzungsoberfläche der VMC

Bei der VMC handelt es sich zum großen Teil um eine Benutzungsoberfläche. In dieser Fachwelt hat die Eclipse-Plattform ihren Ursprung und stellt daher spezifische horizontale Dienste bereit (siehe Abschnitt 2.2.4.2 und 3.1.2.2). Deshalb wurde diese Plattform als Grundlage der Vitness-Anwendung ausgewählt.
Die Swing-Fensterbibliothek der Benutzungsoberfläche soll durch die alternative SWT-Bibliothek ersetzt werden, um eine native Handhabung und natives Aussehen der Benutzungsoberflächen-Elemente für verschiedenen Betriebssystems zu erreichen (siehe Abschnitt 4.2.1). Da der vollständige Austausch der Fensterbibliothek den Rahmen der ersten Stufe überschreitet, werden die JPanel vorerst in SWT eingebettet (siehe Kapitel 4.1.4). In späteren Konsolidierungs-Stufen werden die Benutzungsoberflächen-Elemente dann sukzessive in reine SWT-Elemente umgewandelt. Während dieser Ersetzungsschritte soll die Anwendung in Hinsicht auf die Benutzbarkeit überprüft werden. Änderungen an der Oberfläche, wie die Aufteilung der Fenster oder die Anordnung anderer Elemente, sind unter dem Gesichtspunkt der Software-Ergonomie zu prüfen (vgl. [Balzert 1996]). Eine Anpassung an Benutzungsstandards ist nötig (siehe [Eclipse 2006b, c; Creasey 2006]). Ein Meilenstein der ersten Stufe ist der Austausch der Illustration der Benutzungsoberfläche mit neue Grafiken um die Benutzbarkeit zu verbessern.
Eine stärkere Ausrichtung der Benutzungsoberfläche in Bezug auf Komponenten ist nötig, um in späteren Stufen die Möglichkeit zur Wiederverwendung von ganzen Benutzungsoberflächen-Elemente zu gewinnen. So wird der Datenbank-Browser in der konsolidierten Anwendung eine zentrale Komponente sein, von der nicht nur das Beobachten einer Datenbank gestartet werden kann, sondern auch die Administration eines Datenbanksystem oder eine Schema-Ansicht möglich ist.
Da eine Aktualisierung über das Internet als Anforderung besteht, soll der Update-Manager der Eclipse-Plattform genutzt werden. Eine dadurch nötige Update-Site besteht bereits für andere Zwecke und muss daher lediglich um die neuen Komponenten erweitert werden.
Um der Popularität von Web-Anwendungen gerecht zu werden, die ein sehr einfaches Ausliefern von Anwendung beim Benutzer erlauben, soll die VMC JAVA Webstart unterstützen. JAVA Webstart ermöglicht das Starten einer Anwendung aus dem Webbrowser. Alle nötigen Komponenten werden dann automatisch zum Arbeitsplatz des Benutzers transferiert, so dass eine manuelle Installation beim Benutzer nicht mehr nötig ist. In der ersten Stufe stellt

JVI allerdings ein Hindernis für diese Funktionalität dar, da nicht alle nötigen Teile von JVI innerhalb der VMC verpackt werden können.[27] Dies ist ein weiterer Grund, die Abhängigkeit auf JVI zu entfernen.

4.2.5 Fazit des Soll-Zustands der VMC

Zusammenfassend können zwei maßgebliche Ergebnisse des Soll-Zustands genannt werden. Das wichtigste Ergebnis ist die Konsolidierung der Bestandsanwendung auf das OSGi-Komponentenmodell (vgl. Abschnitt 2.2.3 und 3.1). Dies wird zur Erreichung eines flexiblen Productline Engineerings, vereinfachter Wartbarkeit sowie Weiterentwicklung und zur Steigerung der Wiederverwendbarkeit unternommen (vgl. Abschnitt 2.2).
Da die Auswahl des Komponentenmodells auf OSGi gefallen ist, wird bei Verhandlungen (vgl. Abschnitt 2.3.3) außerdem ein Rückgriff auf Fremdkomponenten des Eclipse Komponentenmarkts (vgl. Abschnitt 2.2.5 und 3.2) möglich. Allerdings ist die Nutzung von Fremdkomponenten nur unter der Einschränkung erlaubt, dass kein zusätzlicher Aufwand zur Einbindung nötig ist. Der Schwerpunkt des gesamten Projekts ist die Konsolidierung der Bestandsanwendung.

Als zweites Ergebnis ist die Verbesserung der Benutzbarkeit definiert. Die Umsetzung dieser Forderung wird allerdings erst nach der erwähnte Ersetzung der Benutzungsoberflächen-Element möglich und sinnvoll. Somit ist sie für spätere Stufen vorgesehen.

[27] Eine Installation des VOD ist auf dem Arbeitsplatz des Benutzers nötig, damit JVI funktionieren kann.

Kapitel 5

Umsetzung

Dieses Kapitel behandelt die Beschreibung und Anwendung einer Komponentifizierung auf einen Prototyp (Abschnitt 5.1). Im Anschluss daran wird eine inhaltliche und quantifizierende Auswertung der Komponentifizierung durchgeführt (Abschnitt 5.2).

5.1 Komponentifizierung, Verhandlung und Integration

Ein Ziel dieser Arbeit ist die Erzeugung von Komponente aus einer Bestandsanwendung für ein zuvor ausgewähltes Komponentenmodell (siehe Abschnitt 2.2.3). Diese Erzeugung soll systematisch durchführbar sein, sodass der Prozess bei beliebigen Bestandsanwendungen zur wiederholten Anwendung kommen kann. Zum Zeitpunkt dieser Arbeit lag ein solcher Prozess nur als sehr abstrakte Beschreibung vor [Bauer u. a. 2000]. Darauf aufbauend wurde deshalb der nachfolgend beschriebene Prozess entwickelt und anhand einer Fallstudie verifiziert. Die Beschreibung beinhaltet daher die konzeptionellen Schritte sowie die empirischen Ergebnisse der Prozessumsetzung auf den in Kapitel 4 beschriebenen Prototyp.
Zur Vereinfachung des Prozesses wurden verschiedene Werkzeuge zur Unterstützung der Arbeiten herangezogen. Ein UML-Werkzeug hilft durch Abstraktion von Implementierungsdetails beim architektonischen Verständnis und Erkennen von Entwurfsmustern (siehe [Gamma u. a. 1995]). Daneben wird ein Software-Analysewerkzeug zur architektonischen Abstraktion auf Komponenten- und Schichtenebene eingesetzt. Die Auswahl der genannten Werkzeuge folgt vorrangig der Verfügbarkeit in Bezug auf eine wissenschaftliche Arbeit. Dafür wurde in einer einführenden Internetrecherche eine Auswahl zusammengestellt und als Grundlage des Auswahlprozesses genutzt.[28]
Die Liste der UML-Kandidaten umfasste Borland Together 2006, IBM Rational Software Architect 6.0, Omondo EclipseUML 2.1.0, MagicDraw UML 11.5 sowie Poseidon 4.2.1 von

[28] Die Webseite `http://www.jeckle.de/umltools.htm` bietet einen Überblick diverser UML-Werkzeuge.

Gentleware. Letztendlich fiel die Wahl auf das Werkzeug Poseidon, da eine preiswerte Studentenlizenz verfügbar ist.
Die Auswahl der Software-Analysewerkzeuge ist hingegen geringer. Neben dem letztlich ausgewählten Sotograph stehen SonarJ von Hello2Morrow und Bauhaus während dieser Arbeit zur Disposition. Da für den Sotograph keine kostenfreie oder preiswerte Lizenz verfügbar ist, wurde eine spezielle Lizenz von der Firma Software-Tomography GmbH für den Zeitraum der Arbeit bereitgestellt.

5.1.1 Vorbereitende Arbeiten

Bevor die komponentifizierenden Arbeiten an der Bestandsanwendung beginnen können, sind einige vorbereitende Arbeiten nötig, um die Quelltexte in einen nutzbaren Zustand für Entwicklungstätigkeiten zu überführen. So wird als Erstes überprüft, ob der Anwendungsquelltext vollständig zur Verfügung steht. Falls eine Rekonstruktion des Quelltextes nicht möglich ist, so müssen diese Teile der Anwendung in eine Black-Box-Komponente umgewandelt werden (vgl. Abschnitt 2.2.2.9).
Im Anschluss an die Vollständigkeitsprüfung der Anwendung und einen erfolgreichen Übersetzungsschritt, muss eine generelle funktionale Prüfung der Anwendung durchgeführt werden, um den Erfolg der Arbeiten in den später nötigen Refactorings zu prüfen (vgl. [Fowler u. a. 2004]).

Die Vorbereitung des Prototyps ergab, dass Teile der Quellen fehlten. Eine Suche in der Historie des Versionsverwaltungssystems zeigte, dass für einige Paket zu keinem Zeitpunkt Quelltextdateien vorhanden waren. Da allerdings ein Übersetzen der ursprünglichen Quellen möglich war, lagen zumindest Kompilate der Klassen vor. Mit Hilfe eines Decompilers konnten die Quellen aus den Kompilaten wieder hergestellt werden. Anschließend wurden die identifizierten Funktionen des Prototyps manuell getestet (vgl. Abschnitt 4.1.2). Die Test-Ergebnisse zeigten, dass eine Verbindung zum VOD nicht möglich war. Ein Austausch des JVI-JAR auf die passende Version des VOD löste dieses Problem. Dieser Austausch führte zu keinen programmatischen Problemen.[29] Weitere Einschränkungen im Funktionsumfang konnten durch diese (manuelle) Kontrolle nicht entdeckt werden.

Nachdem die Quellen in einen nutzbaren Zustand überführt sind, können im Folgenden die Arbeitsschritte zur Erzeugung der Komponenten unter Zuhilfenahme der Werkzeuge beginnen. Im nächsten Abschnitt werden vorher einige nötige Bemerkungen zum Sotograph gegeben.

[29] Eine Testumgebung des Prototyps zum Zeitpunkt der Arbeit fehlte, Aussagen zur korrekten Funktionalität müssen differenziert betrachtet werden (siehe Abschnitt 4.1.6).

5.1.2 Vorgehen mit dem Sotograph

Da der Sotograph eine zu dieser Arbeit abweichende Terminologie in Bezug auf Komponenten aufweist, müssen diese unterschiedlichen Begriffswelten zusammengeführt werden. Der Sotograph kennt die logischen Einheiten Klassen, Pakete, Subsysteme und architektonische Schichten. Diese Liste ist monoton steigend sortiert. Ein Paket wird beispielsweise aus einer oder mehreren Klassen gebildet. Physikalisch bilden sich die Einheiten auf Dateien, Namensräume, JAR- oder DLL-Dateien sowie Schichten eines Schichtenmodells ab [Sotograph 2006]. Da sich das Subsystem in dieser Definition auf dieselben technischen Konstrukte wie die Komponente abbildet, können die beiden Begriffe als äquivalent angesehen werden (siehe Abschnitt 2.2.6). Im Folgenden wird daher der Begriff Komponente verwendet, obwohl er im Sotograph unbekannt ist.
Neben obiger Begriffserklärung muss außerdem darauf hingewiesen werden, dass der Sotograph ausschließlich auf der Basis der Sprachmittel der unterstützen Programmiersprachen Analysen durchführen kann. Die Sprachmittel von JAVA reichen aber keineswegs aus, um ein umfassendes Verständnis der Anwendung zu gewinnen. Somit muss bei der Erzeugung der Komponenten Wissen über architektonische Besonderheiten einfließen. Es kann daher vorgreifend festgehalten werden, dass die nachfolgend dokumentierten Schritte nicht vollständig deterministisch sind, sondern auf der Basis von zusätzlichem Wissen beruhen.
Weiterhin muss auf die Einschränkung des Sotograph im Rahmen der Schnittstellenspezifikation hingewiesen werden. Aus Sicht des CBSE bieten Komponenten zwei Schnittstellenarten an (vgl. Abschnitt 2.2.2.8). Im Sotograph ist eine solche Spezifikation nicht möglich. Hier sind Schnittstellen lediglich Aufrufbeziehungen auf bestimmte Pakete/Namensräume und daher semantisch eine Vermischung dieser beiden Schnittstellenarten.

5.1.3 Schritt 1: Identifikation von Komponentenkandidaten

Zur reproduzierbaren Identifikation von Komponentenkandidaten sind zum Zeitpunkt der Arbeit keine Mechanismen auffindbar. Daher besteht die Notwendigkeit unter folgenden Annahmen eine Kandidatenidentifikation durchzuführen: Programmiersprachen wie Java oder C++ bieten das Konzept der Namensräume an. Diese Namensräume dienen neben der Beseitigung von Namenskollisionen meist der logischen Strukturierung von Programmen. Diese Annahme wird als Grundlage des Identifikationsschritts herangezogen. Ausgehend von der Wurzel des Namensraums, werden Grenzen von Komponentenkandidaten an den Verzweigungen des Namensraums vermutet. Dieser Vermutung folgend, werden Kandidaten identifiziert und spezifiziert. Dies geschieht in dem Bewusstsein, dass die Definition der Namensräume in der Bestandsanwendung vollkommen willkürlich erfolgt sein kann. Eine hohe Zahl an Abhängigkeitsverletzungen würde allerdings in späteren Schritten (siehe Abschnitt 5.1.5) eine mögliche Falschannahme aufdecken.
Ein konkreter Schwellwert für die Anzahl der Abhängigkeitsverletzungen kann nicht angege-

ben werden. Solch ein Schwellwert muss in weiteren Arbeiten hergeleitet werden. Sicherlich beeinflussen aber neben der reinen Anzahl der Abhängigkeitsverletzungen, die Größe der Komponenten, die Größe der Gesamtanwendung und das eingesetzte Programmierkonzept eine solche Metrik. Mit obigem Konzept werden die Komponentenkandidaten im Sotograph

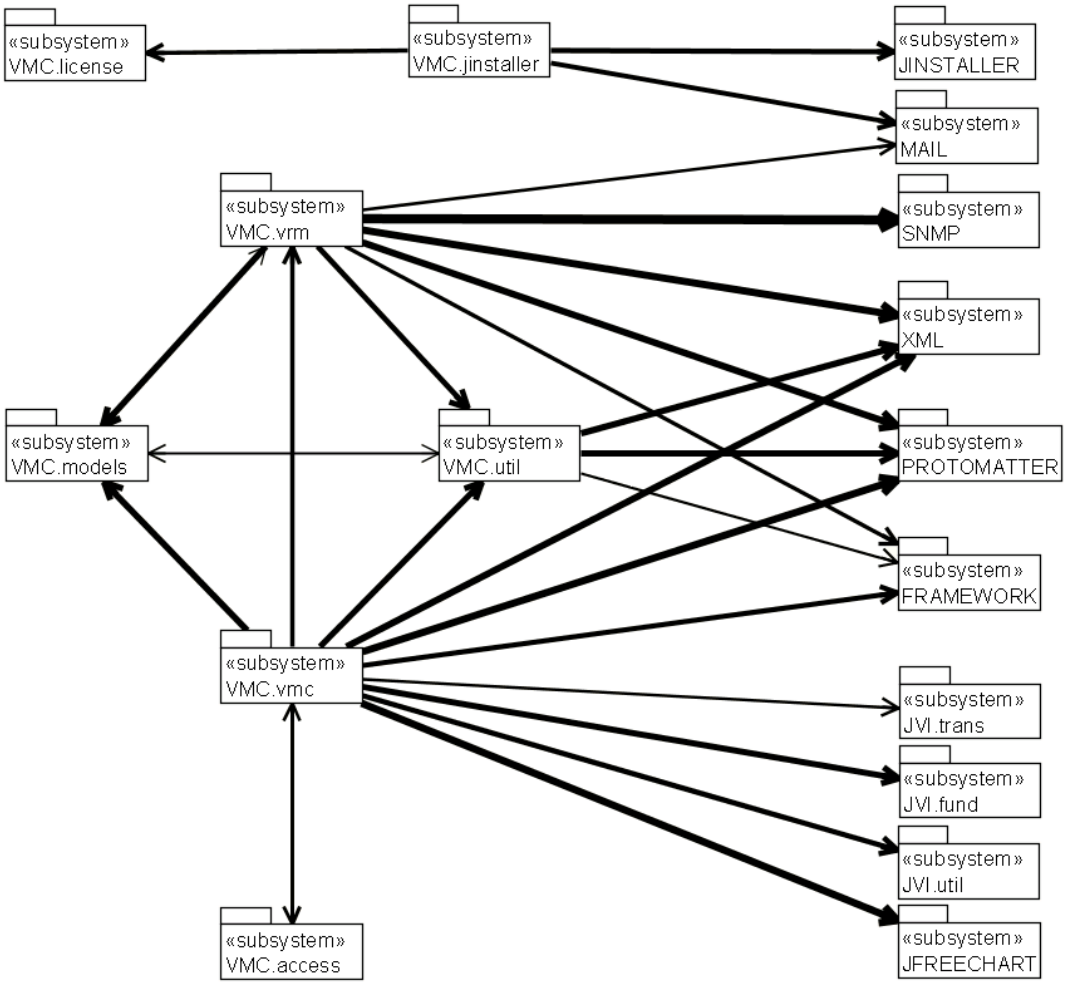

Abbildung 5.1: Abhängigkeitsdiagramm der identifizierten Komponentenkandidaten

als Subsysteme definiert. Im folgenden Beispiel des Prototyps wird die Definition dieser Subsysteme schematisch dargestellt. Mit einem regulären Ausdruck werden alle Sub-Pakete von `it.versant.dbmonitor` als Subsysteme definiert.

```
public void generateSubsystemModel(IModelFactory f,ISubsystemModel sm){
...
sm.createSubsystemGroupByRegexp("VMC", "it.versant.dbmonitor.'.*'",
sm.cHasInterface, sm.cRootPackageIsInterface, sm.cAddInterfacePackages);
...
}
```

Vorgreifend wird erwähnt, dass diese Subsysteme eine explizite Schnittstelle definieren. Eine genaue Beschreibung findet sich im Abschnitt 5.1.4.
Mit dieser Spezifikation lässt sich eine Visualisierung der Komponenten im Sotograph erzeugen (siehe Abbildung 5.1). Dieses Diagramm wird als Ausgangsbasis einer Abhängigkeitsanalyse genutzt. Dabei werden die Abhängigkeiten auf Plausibilität in Bezug auf architektonische Entwurfsmuster geprüft.[30] Das bisher gewonnene Gesamtwissen über die Bestandsanwendung kommt bei dieser Analyse zum Einsatz. Die Notwendigkeit dieser Abhängigkeitsprüfung zeigt sich im folgenden Schritt, in dem eine Spezifikation der Schnittstellen und Architektur erstellt wird.
Im Prototyp konnten auf diese Weise neun grundlegende Komponentenkandidaten identifiziert und im Sotograph spezifiziert werden. Da das Interesse der VMC-Komponente galt, wurde eine detailliertere Einteilung vorgenommen. Ebenso wurde für die JVI-Komponente verfahren.[31] Manche JAR-Dateien konnten nach einer Durchsicht der beiliegenden Dokumentation ohne weitere Analyse als Komponentenkandidaten zusammengefasst werden.
In Tabelle 5.1 werden die identifizierten Komponentenkandidaten mit ihrer jeweiligen Abbildung auf Java-Pakete und JAR-Dateien aufgelistet. Wenn sich ein Komponentenkandidat aus mehreren Paketen oder Archiven zusammensetzt, so wird dieses durch eine Aufzählung in der jeweiligen Spalte wiedergespiegelt.

5.1.4 Schritt 2: Komponentenschnittstellen und architektonische Schichten

Nachdem Komponentenkandidaten im vorigen Schritt identifiziert und ein Verständnis über deren gegenseitige Abhängigkeiten gewonnen wurde, werden in diesem Schritt die externen Schnittstellen jedes Kandidaten festgelegt, über die er Dienste nach außen bereitstellen kann. Die formale Spezifikation der Inter-Komponentenschnittstellen erfolgt im Sotograph ausschließlich auf Paketebene und soll beispielhaft wiedergegeben werden. Diese Definition erlaubt einen exklusiven Zugriff auf die `VMC.util`-Komponente über das `it.versant.dbmonitor.util`-Paket. Zugriffe auf Klassen außerhalb dieses Pakets werden durch den Sotograph als Verletzung der Komponentenschnittstelle gewertet und

[30] Die einleitend genannten UML-Werkzeuge bieten sich zur Erkennung von Entwurfsmustern an.

[31] JVI stellt verschiedene Abstraktionsebenen für den Zugriff auf das Datenbanksystem bereit und beeinflusst damit die Architektur.

Komponente	Java-Pakete	JAR
VMC.access	it.versant.dbmonitor.access	liegt im Quelltext vor
VMC.jinstaller	it.versant.dbmonitor.jinstaller	"
VMC.models	it.versant.dbmonitor.models	"
VMC.util	it.versant.dbmonitor.util	"
VMC.vmc	it.versant.dbmonitor.vmc	"
VMC.vrm	it.versant.dbmonitor.vrm	"
JVI.trans	com.versant.trans	jvi6.0.5-jdk1.3.jar
JVI.fund	com.versant.fund	"
JVI.util	com.versant.util	"
FRAMEWORK	it.versant.framework	vutil.jar
JINSTALLER	it.versant.jinstaller	installer.jar
JFREECHART	com.jrefinery com.keypoint	jfreechart-0.9.6.jar jcommon-0.7.2.jar
MAIL	javax.mail com.sun.mail javax.activation	mail.jar activation.jar
SNMP	com.sun.jdmk javax.management util	jdmkrt.jar jsnmpapi.jar
PROTOMATTER	com.protomatter	protomatter-1.1.7.jar
XML	org.apache.xml org.xml org.w3c javax.xml	xerces.jar

Tabelle 5.1: Abbildung zwischen Komponenten, Java-Paketen und JAR-Dateien

dementsprechend markiert.

```
public void generateSubsystemModel(IModelFactory f,ISubsystemModel sm){
...
sub= sm.getSubsystem("VMC.util");
sub.addInterfacePackage(
    sub.getPackageByPath("it.versant.dbmonitor.util"));
...
}
```

Neben den Schnittstellen der einzelnen Komponentenkandidaten wird auf der höchsten Abstraktionsebene des Sotographs eine Architekturspezifikation erstellt. Diese Spezifikation erlaubt eine Einteilung der vorher erzeugten Komponenten in einer Schichten- oder Matrixarchitektur und eine Festlegung der erlaubten (oder verbotenen) Abhängigkeiten zwischen den Komponenten. Für diese Spezifikation wird die Matrixarchitektur gewählt, da sich eine Schichtenarchitektur wegen starren Regeln in Bezug auf Abhängigkeiten als weniger geeignet erwiesen hat.[32] Wie die vorherigen Komponentenschnittstellen-Spezifikationen, soll auch die formale Architekturspezifikation im Sotograph schematisch dargestellt werden.

```
IGraphArchitectureModel am= f.createGraphArchitectureModel(
f.getCurrentModelScriptName(), sm);
...
am.addRelationship("VMC.vmc","JVI.trans");
am.addRelationship("VMC.vmc","JVI.fund");
am.addRelationship("VMC.vmc","JVI.util");
am.addRelationship("VMC.vmc","VMC.models");
am.addRelationship("VMC.vmc","VMC.util");
...
```

Die Spezifikation stellt die erlaubten, unidirektionalen Abhängigkeiten der `VMC.vmc`-Komponente dar. Weitere Abhängigkeiten werden durch diese Schnittstellenspezifikation für die Komponenten ausgeschlossen und vom Sotograph als Architekturverletzung gewertet.

In Bezug auf den Prototypen wurden die Schnittstellenspezifikation der Komponentenkandidaten wegen eingeschränktem Wissen über die Bestandsanwendung unrestriktiv gesetzt. Die Matrixarchitektur wurde für `VMC.vmc, VMC.vrm, VMC.model` und `VMC.model` angegeben.

5.1.5 Schritt 3: Abhängigkeiten zwischen Komponentenkandidaten

Durch die vorangegangene Spezifikation der Komponenten, deren Schnittstellen und zusätzliche architektonische Aspekte, können in diesem Schritt deren Verletzungen erkannt und an-

[32] Auf eine Utility-Komponente wird aus allen Schichten zugegriffen. Dies lässt sich mit einer Schichtenarchitektur nicht sinnvoll darstellen.

schließend beseitigt werden. Zur Unterstützung dieses Arbeitsschritts erlaubt der Sotograph automatische Überprüfungen auf Einhaltung der Komponentenkandidatenschnittstellen und korrekten Komponentenabhängigkeiten auf unterschiedlichen Abstraktionsebenen. Da auf den höheren Abstraktionsebenen des Sotographs gearbeitet wird, werden die Möglichkeit der Schnittstellen- und Abhängigkeitsprüfung genutzt.
Die Beseitigung der genannten Verletzungen kann nicht mit dem Sotograph erfolgen. Dazu wird die IDE als Werkzeug herangezogen, deren Unterstützungen für Refactorings in Form von Assistenten solche Arbeiten erleichtert.

Die Prüfung auf zulässige Schnittstellenbenutzung identifizierte am Prototyp zwei Verletzungen. Die `VMC.models`-Komponente nutzte nicht die spezifizierten Schnittstellen. Gleiches galt für die Abhängigkeit zwischen `VMC.vmc` und der `VMC.vrm`-Komponente. Weitere Verletzungen konnten durch die Validierungsfähigkeiten auf Schnittstellenebene nicht erkannt werden.
Auf der Ebene der architektonischen Schichten konnten durch den Sotograph vier verschiedene Verletzungen markiert werden. Dabei handelte es sich um verbotene Abhängigkeiten der `VMC.vmc`-Komponente auf `VMC.vrm` und `VMC.access`. Daneben bestand ein unerlaubter Zugriff von `VMC.util` auf `VMC.models`. Schließlich ist die Abhängigkeit der `VMC.access` Komponenten zur `VMC.vmc`–Komponente zu nennen. Im aufgeführten Abhängigkeitsdiagramm werden die Abhängigkeitsverletzungen des Prototypen rot gekennzeichnet (siehe Abbildung 5.2). Die Arbeiten zur Beseitigung dieser Verletzungen waren im Prototyp allerdings marginal. Hauptsächlich waren einfache Verschiebungen von Klassen und Interfaces in der Paketstruktur nötig.[33]

5.1.6 Zwischenschritt: Plausibilitätsprüfung der Kandidaten

In diesem Zwischenschritt wird der eingangs erwähnte Vorbehalt zur Komponentenidentifzierung aufgegriffen (siehe Abschnitt 5.1.2). Wie dargelegt, würde eine hohe Anzahl an Abhängigkeitsverletzungen im vorherigen Schritt auf eine falsche Identifikation der Komponentenkandidaten hinweisen.
[Bauer u. a. 2000] empfiehlt darüber hinaus eine Validierung der Komponenten durch unabhängige fachliche und technische Gutachter in Form von Design-Reviews.

Im Prototyp begrenzten sich die Verletzungen auf fünf Abhängigkeiten. Diese geringe Zahl wurde als Bestätigung der Komponentenkandidaten gewertet. Eine Validierung durch Dritte entfiel.

[33] Eine detaillierte Dokumentation würde den Rahmen dieser Bachelorarbeit überschreiten. Eine Rekonstruktion der Arbeiten ist über einen Vergleich der Version im Versionsverwaltungssystem jederzeit möglich.

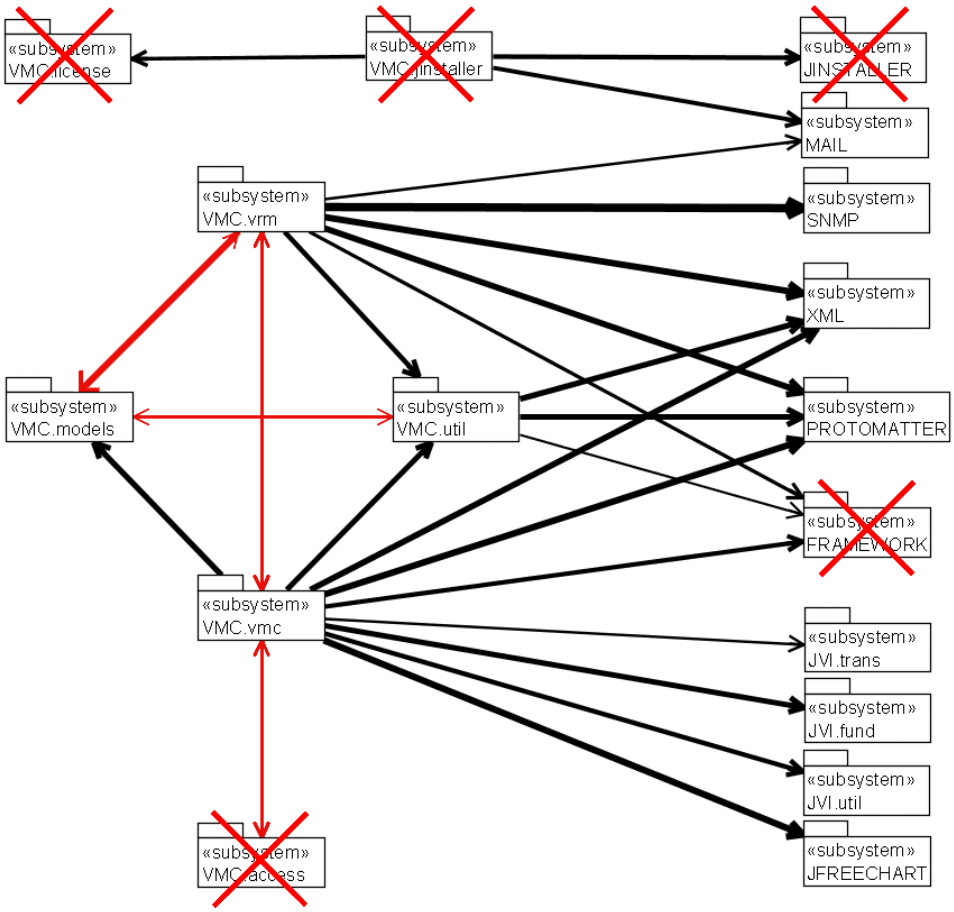

Abbildung 5.2: Abhängigkeitsverletzungen zwischen Komponentenkandidaten

5.1.7 Schritt 4: Verhandlung zur Kandidatenauswahl

Im Anschluss an die Beseitigung der unerlaubten Abhängigkeiten wird die Verhandlung (vgl. Abschnitt 2.3.3) über die Ersetzung einzelner Komponenten geführt. Sie basiert auf der Grundlage des gesammelten Anwendungswissens und den verfügbaren Fremdkomponenten. Hierbei müssen neben der Kenntnis der Komponentenschnittstellen auch die Dienste der jeweiligen Komponente mit einbezogen werden, um einen Abgleich unter diesen beiden Gesichtspunkten zwischen dem Komponentenkandidaten und der Fremdkomponente durchzuführen.

Für den Prototyp wurde die Entscheidung getroffen, die drei Komponenten `VMC.jinstaller`, `VMC.license` und `JINSTALLER` zu ersetzen. Deren Funktionalitäten (Abprüfen einer gültigen Lizenz (vgl. Abschnitt 4.2.3) und Bereitstellung einer

Installationsroutine) konnten durch andere Komponenten übernommen werden. Dabei ist es erwähnenswert, dass die Ersetzung auf Grund geringer Komponentenkopplung mit anderen VMC-Komponenten und Kohäsion besonders einfach erfolgte. Der Komponentenkandidat `VMC.access` wurde nach Beseitigung der Abhängigkeit aus dieser Betrachtung entfernt. Die Funktionalität dieses Kandidaten ging nicht über ein Quelltextbeispiel hinaus. Neben den Genannten wurden keine weiteren Komponentenersetzungen durchgeführt. Im Abhängigkeitsdiagramm (siehe Abbildung 5.2) sind die zur Entfernung vorgesehenen Kandidaten mit einem roten Kreuz entsprechend markiert.

5.1.8 Schritt 5: Erzeugung der identifizierten Komponenten

Alle zur Komponentifizierung vorgesehenen Komponenten, die nicht durch die vorherige Verhandlung entfallen sind, werden in diesem Schritt in Komponenten des verwendeten Komponentenmodells umgewandelt. Dabei sind die Arbeiten und deren Aufwände durch die vorherige Beseitigung der Abhängigkeiten (siehe Abschnitt 5.1.5) nur vom eingesetzten Komponentenmodell und der Verhandlung abhängig. Falls ein Austausch durch Fremdkomponenten erfolgte (siehe Abschnitt 5.1.7), können an den Schnittstellen zwischen den Bestandskomponenten und den Fremdkomponenten Modifikationen nötig werden. Diese Modifikation sind entweder Änderungen an der Schnittstelle der Bestandskomponente oder eine Adaption zwischen Bestands- und Fremdkomponente. In jedem Fall sind Änderungen an der Komponentendokumentation vorzunehmen, welche die neuen Eigenschaften der Komponente reflektieren (siehe Abschnitt 2.2.2.8).

Die Umwandlung der Komponentenkandidaten des Prototyps in Bundle des OSGi-Komponentenmodells waren aus zwei Gründen von geringem Aufwand (siehe Abschnitt 3.1). Zum einen unterstützt die IDE die Arbeiten mit Assistenten, zum anderen ist der Aufbau eines Bundle des OSGi-Komponentemodells ähnlich mit den ursprünglichen JAR-Dateien (vgl. Abschnitt 3.1.3). Im folgenden Diagramm werden die umgewandelten Kandidaten abschließend dargestellt (siehe Abbildung 5.3). Im Diagramm sind die modifizierten Abhängigkeiten sichtbar. So kamen neben der Bereinigung von unerlaubten Abhängigkeiten auch neue Abhängigkeiten zwischen Komponenten hinzu. Die `VMC.models`-Komponente benötigt nach der Komponentifizierung die `XML` und `PROTOMATTER`-Komponente, die `VMC.vrm`-Komponente die `JVI.license`-Komponente. Diese Abhängigkeiten sind durch die Verschiebe-Refactorings (vgl. Abschnitt 5.1.5) sowie der erwähnten Lizenzprüfung (vgl. Abschnitt 4.2.3) entstanden. Außerdem wurde die `FRAMEWORK`-Komponente in die `VMC.util`-Komponente integriert, da sich die jeweiligen Komponentendienste ähnelten und die Komponentengranularität gering war (vgl. Abschnitt 2.2.2.10).
Eine Erkenntnis der empirischen Anwendung ist, dass Technologien wie Remote Method Invocation weniger zur Komponentifizierung geeignet. Sie verletzen Forderungen des CBSE. So widerspricht das im Prototyp verwendete Remote Method Invocation der Komponen-

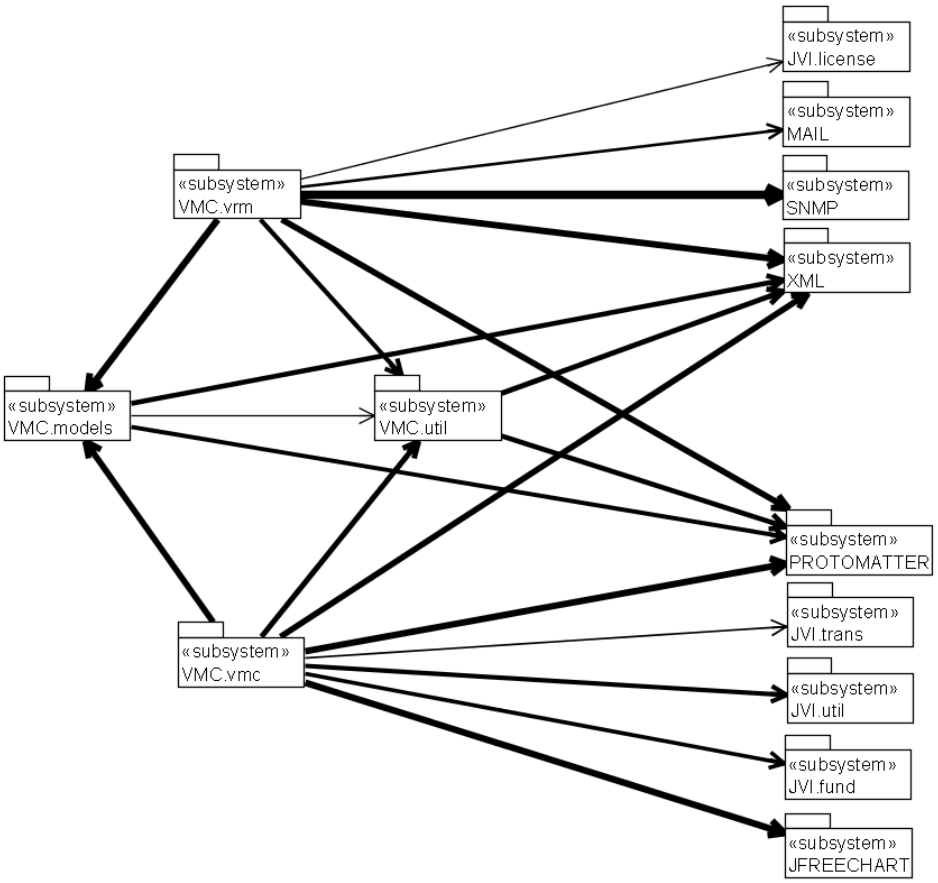

Abbildung 5.3: Komponenten und Abhängigkeiten nach der Komponentifizierung

tenanforderung nach Komponentenunabhängigkeit zur Übersetzungszeit (vgl. Abschnitt 2.2.2).[34]

5.1.9 Schritt 6: Rekursive Komponentifizierung

Der in den vorherigen Schritten dargestellte Prozess eignet sich zur rekursiven Anwendung. Eine formale Abbruchbedingung, die bei einem rekursiven Prozess nötig ist, bestünde, wenn die Unteilbarkeit der zugrunde liegenden Programmiersprachenkonstrukte erreicht wird. Diese formale Abbruchbedingung eignet sich aber nur bedingt, da die resultierenden Komponenten zu klein wären und ggf. atomarer als die Strukturierungskonzepte der Programmiersprache ausfallen würde (siehe Abschnitt 2.2.2.10). Somit muss eine praxisnahe Abbruchbedingung für den rekursiven Komponentifizierungsprozess entwickelt werden.

[34] Eine Erzeugung der RMI Stub-Klassen kann nicht aus den Interfaces erfolgen.

In dieser Arbeit wurde diese Abbruchbedingung am Fokus der Konsolidierung und der zur Verfügung stehenden Ressourcen festgemacht und daher nicht wissenschaftlich auf Metriken fundiert.
Der Komponentifizierungsprozess wurde erneut angewandt und endete demnach nicht nach dem ersten Durchlauf. An dieser Stelle wird jedoch nicht auf diesen Rekursionsschritt eingegangen, da diese Rekursionsdurchläufe keine neuen Erkenntnisse für den eigentlichen Kompontifizierungsprozess liefern. Die Ergebnisse dieser Rekursionsdurchläufe finden sich im Anhang C.

5.2 Fazit der Komponentifizierung

Abschließend sollen Schlüsse aus dieser Komponentifizierung gezogen werden. Dieses Fazit gliedert sich in eine Betrachtung der Komponentifizierung anhand des Prototyps, einer quantifizierenden Bewertung der empirischen Umsetzung der Komponentifizierung, einer Bewertung der Werkzeugunterstützung und einer generellen Folgerung des beschriebenen Prozesses.

5.2.1 Der Prototyp im Anschluss an die Komponentifizierung

Die Ergebnisse der Komponentifizierung entsprechen den Forderungen des Soll-Zustands (siehe Abschnitt 4.2). Eine Dekomposition des Prototyps in Komponenten ist geschehen. Insbesondere besteht eine strikte Trennung zwischen dem Agenten und der Benutzungsoberfläche (vgl. Abschnitt 4.2.2). Eine Integration in das OSGi-Komponentenmodell ist erfolgt, Komponentenkandidaten sind durch COTS-Komponenten ersetzt. Darüber hinaus wurde neue Funktionalität durch den Einsatz von Fremdkomponenten integriert (vgl. Abschnitt 4.2.4).

Einige Komponentenkandidaten wurden während der Verhandlung zur Ersetzung durch Fremdkomponenten ausgewählt. Jedoch ließen knappe Ressourcen und die Nicht-Existenz eines internen Komponenten-Repositories die hauptsächliche Komponentenauswahl bei den Verhandlungen auf Bestandskomponenten fallen. Beachtet man die Dienste der ersetzten Komponentenkandidaten, ist festzustellen, dass keine Fachlogik der Anwendung ersetzt ist. Es sind ausschließlich Komponenten mit Rahmenwerk-Diensten ausgetauscht worden (vgl. Unterabschnitt 5.1.7 und 2.2.4.2). Dies begründet sich in dem Fehlen von Alternativen für die Fachlogik. Auch der externe Komponentenmarkt bietet keine Komponenten dieser Fachwelt an.
Die nötigen Arbeiten zur Erreichung dieser Ergebnisse begrenzen sich im Prototyp auf einfache Refactorings. Dies ist auf den positiven Ist-Zustand des Prototyps zu Beginn der

Komponentifizierung zurückführbar (siehe Abschnitt 4.1). So zeigt sich eine geringe architektonische Degeneration, die sich in wenigen Abhängigkeitsverletzungen zwischen den Komponentenkandidaten manifestiert. Weiterhin erleichtert die geringe Größe des Prototyps die Handhabbarkeit und somit den Komponentifizierungsprozess. Darüber hinaus steht der Quelltext aller Komponentenkandidaten bereit (White-Box-Komponenten), weshalb ein inversive Adaption möglich ist und keine Adaption durch Klebecode, wie bei Black-Box-Komponenten oder Glass-Box-Komponenten nötig, vorgenommen wird (siehe Abschnitt 2.2.2.9).

5.2.1.1 Quantifizierende Auswertung der Komponentifizierung

In diesem Abschnitt wird eine quantifizierbare Aussage über den Erfolg der Komponentifizierung getroffen. Diese Aussage basiert auf Metriken, die interpretierbare Maßzahlen über ein Software-System liefern. In der Software-Technik existieren dazu diverse Metriken zur Erfassung unterschiedlicher Aspekte eines Software-Systems, die sich jedoch nicht auf das CBSE anwenden lassen. In [Gill u. Grover 2003] findet sich eine Begründung, warum viele traditionelle Metriken für das CBSE ungeeignet sind. Als Hauptgründe werden die Abhängigkeiten zum Quelltext genannt, der für Black-Box-Komponenten (siehe Abschnitt 2.2.2.9) nicht verfügbar ist und die Konzentration auf, aus der Sicht des CBSE, uninteressante Aspekte eines Software-Systems [Gill u. Grover 2003, S. 3].
Für diese Auswertung werden daher Metriken benötigt, die im Kontext des CBSE anwendbar sind und Aussagen über den Erfolg einer Komponentifizierung zulassen. Zum Zeitpunkt dieser Arbeit konnten jedoch keine speziellen Metriken für die Bewertung einer Komponentifizierung in der Literatur gefunden werden, weshalb auf allgemeine Metriken zurückgegriffen werden muss. Diese Metriken werden in dieser Arbeit im Rahmen der Komponentifizierung interpretiert. Dabei wurden ausschließlich Metriken ausgewählt, die automatisch oder zumindest effizient erhebbar, in jedem Fall aber reproduzierbar sind. Effizient bedeutet in diesem Zusammenhang, dass der Aufwand zur Erhebung der Metrik in wenigen Arbeitsschritten umsetzbar und deshalb regelmäßig durchführbar ist.
Ein weiteres Problem stellt der Zeitpunkt der Auswertung dar. Die Erhebung der Metriken erfolgt vor der Auslieferung der Anwendung an Kunden. Daher können Metriken, welche die Kundenzufriedenheit messen, nicht in die Auswertung einfließen (siehe [Heineman 2001, S. 440ff]). Somit werden vorerst folgende Metriken erhoben:

- Maßzahl der Wiederverwendung in Prozent („Measures of reuse level"): [Heineman 2001, S. 446] definiert eine Maßzahl der Wiederverwendung als den Anteil der durch Dritte erstellten und gewarteten Komponenten in Bezug zur Gesamtanwendung. Die Bestimmung der Metrik erfolgt mit folgender Formel, wobei die Wiederverwendung *W*, die Anzahl der wiederverwendeten Lines of Code (LOC) *Rloc* („Reused LOC"*)* und die

Gesamt-LOC *Tloc* („Total LOC") einfließen:

$$W = (Rloc/Tloc) * 100$$

Da jedoch, wie oben erörtert, der Quelltext von Fremdkomponenten zur Bestimmung der Metrik nicht bereitsteht, werden die LOC-Variablen in dieser Arbeit durch die Anzahl der Symbole einer Komponente substituiert. Symbole definieren sich als die Anzahl an Klassen, Methoden und Attribute einer Komponente. Die Erhebung der Symbolanzahl kann durch den Sotograph auch für Black-Box-Komponenten reproduzierbar und automatisch durchgeführt werden.

- Komponentenkosten („Component Cost Metrics (CCM)"): Die Komponentenkosten werden von [Gill u. Grover 2003] als die Gesamtkosten definiert, die durch den Einsatz von Fremdkomponenten entstanden sind. Dazu zählen Kosten, die für den Komponentenerwerb und durch die Komponentenintegration entstanden sind.

Die Erhebung der Messwerte der Metriken wurde mit dem Sotograph durchgeführt. Die Angaben zu Komponentenkosten entstammen organisationsinternen Berechnungen. Zu den durch Dritte erzeugten Komponenten wurden alle nicht in den Namensräumen `com.versant` und `it.versant` liegende Quelltexte gezählt. Eine Ausnahme von dieser Regel ist die JVI-Komponente. Sie wird als Fremdkomponenten gewertet. Komponente vor Beginn der Komponentifizierung sind alle in Tabelle 5.1 aufgeführten Komponenten. Das zugrunde liegende JDK ist in der Auswertung nicht enthalten. Komponentenkosten entstehen für die SNMP-Komponenten.
In der folgenden Tabelle werden die Messwerte, sowie die kalkulierten Metriken jeweils zu Beginn und nach Abschluss der Komponentifizierung angegeben. Die Zahlen belegen, dass die Komponentifizierung zu einer Steigerung der Wiederverwendung führt. Gleichzeitig muss allerdings auf das Anwachsen der Gesamtanwendung hingewiesen werden. Zwar ist dieser

	vorher	nachher	Trend (gerundet)
Gesamt			
Symbole	29306	58699	+100%
Eigenkomponenten			
Symbole	5771	5166	-10%
Fremdkomponenten			
Symbole	23557	53583	+127%
Wiederverwendung W in %	~80%	~91%	+11%
Komponentenkosten (CCM)	~33k EUR	~7k EUR	-79%

Tabelle 5.2: Quantifizierende Auswertung der Komponentifizierung

Anstieg durch Fremdkomponenten entstanden, wird aber der Aufwand zur Komponentenintegration, der auch für Fremdkomponenten anfällt, in der Aufwandserfassung berücksichtigt, äußert sich der gesteigerte Umfang negativ. Die Ursache für den Anstieg sind die durch die Konsolidierung auf Eclipse verursachten, neuen Abhängigkeiten (vgl. Abschnitt 2.1 und 3).
Die Komponentenkosten sind im Anschluss an die Komponentifizierung stark gesunken. Dies lässt sich auf die Verfügbarkeit von alternativen Komponenten zurückführen, die mit geringem Aufwand integriert werden können.

5.2.2 Bewertung der Werkzeugunterstützung

Die eingesetzten Werkzeuge unterstützen zwei Belange im Komponentifizierungsprozess: Den Prozess der Komponentifizierung und die Gewinnung von Wissen über die Bestandsanwendung. Obwohl der Fokus dieses Fazits auf dem Komponentifizierungsprozess liegt, soll vorher auf die Nützlichkeit der Werkzeuge im Bezug auf die Wissensgewinnung über die Bestandsanwendung eingegangen werden. Dieses Wissen ist im Komponentifizierungsprozess von Bedeutung und fließt in diesen ein (vgl. Abschnitt 5.1.2).
Die Gewinnung von Wissen ist nur im Rahmen der eingesetzten Programmiersprache und dem zugrunde liegenden Programmierkonzept möglich. Eine automatische Wissensgewinnung über die Grenzen der Sprachmittel hinaus, scheitert. Trotzdem helfen die Werkzeuge bei einer systematischen Analyse, da in einem Prozess der kleinen Schritte die Informationsflut durch Wechsel der Abstraktionsebenen handhabbar gemacht wird. Jedoch muss darauf hingewiesen werden, dass manche architektonischen Aspekte in den Analysen unberücksichtigt bleiben. So wird nicht nur Netzwerkkommunikation außer Acht gelassen, sondern auch Zugriffe auf Ressourcen wie Dateien. Hierzu müssten weitere Werkzeuge herangezogen werden, die diese Aspekte einer Architektur einbeziehen.
Ähnlich gestaltet sich die Situation der Werkzeugunterstützung im Bezug auf den Komponentifizierungsprozess. Die Werkzeuge ermöglichen einen systematischen Prozess, sind mit ihrem Funktionsumfang aber offenbar nicht als Komponentifizierungswerkzeuge entwickelt worden. Lediglich das UML-Komponentendiagramm und die Assistenten in der Entwicklungsumgebung sind speziell für die Komponentifizierung geeignet. Diese fehlende Werkzeugunterstützung erschwert den Komponentifizierungsprozess, macht ihn aber nicht unmöglich. Eine umfangreichere Unterstützung wäre allerdings wünschenswert (siehe Abschnitt 6.2).

5.2.3 Tauglichkeit des Komponentifizierungsprozesses

Der vorgestellte Prozess zur systematischen Komponentifizierung unter Zuhilfenahme von Werkzeugen hat sich im Rahmen dieser Arbeit als praxistauglich erwiesen (vgl. Unterabschnitt 5.2.1). Allerdings lassen sich technische sowie konzeptionelle Kritikpunkte an diesem

Prozess äußern: Technisch basiert der Prozess auf dem Programmierkonzept der Objektorientierung und abstrahiert demnach nicht ausreichend, um auf andere Programmiermodelle anwendbar zu sein. Eine Abbildung auf äquivalente Sprachkonstrukte aus anderen Programmiermodellen könnte diese Einschränkung jedoch beseitigen. Eine Konkretisierung dieses Problems ist die direkte Abhängigkeit von Namensräumen. Bietet das Programmierkonzept der Bestandsanwendung keine Namensräume, muss eine neue Methode zur Kandidatenidentifikation entwickelt werden (siehe Abschnitt 5.1.2). Falls das Programmierkonzept Namensräume unterstützt, ist die fehlende Metrik zur Plausibilitätsprüfung der Komponentenkandidaten ein weiteres Problem, das in dieser Arbeit durch einen Zwischenschritt im Prozess gelöst wurde. Die im Zwischenschritt erwähnte Validierung durch Dritte ist wegen dem menschlichen Einflussfaktor problematisch (vgl. Abschnitt 5.1.6). Neben den Einschränkungen durch Sprachkonstrukte, muss auf Probleme mit manchen Technologien hingewiesen werden. Nicht jede Technologie lässt sich mit den Forderungen des CBSE vereinbaren und erschwert die Komponentifizierung (vgl. Abschnitt 5.1.8).
Konzeptionell ist der Hauptkritikpunkt die eingeschränkte Vorhersagbarkeit des Prozesses. Zum einen basiert der Prozess auf informellem Wissen über die Bestandsanwendung, zum anderen wird dieses Wissen innerhalb des Prozesses gewonnen und fließt im Anschluss in einen folgenden Schritt ein. Im schlechtesten Fall entstehen Folgefehler (siehe Abschnitt 5.1.2). Außerdem existiert keine formal sinnvolle Abbruchbedingung für die Rekursion des Komponentifizierungsprozesses. Eine Metrik, welche die Anforderungen beachtet, muss in weiterführenden Arbeiten hergeleitet werden (vgl. Abschnitt 5.1.9).

Kapitel 6

Zusammenfassung und Ausblick

Dieses Kapitel präsentiert die Ergebnisse der Arbeit und zieht aus ihnen Schlussfolgerungen. Die Schlussfolgerungen beziehen sich einerseits auf das Vitness-Konsolidierungsprojekt der Versant Corp. Andererseits abstrahieren sie, um die allgemeine Methodik zu bewerten. Im Ausblick werden abschließend sinnvolle Anschlussarbeiten an diese Bachelorarbeit aufgezeigt.

6.1 Fazit

Die Aufgaben und Ziele dieser Bachelorarbeit konnten mehrheitlich erfüllt werden. Aufbauend auf den Begrifflichkeiten im Grundlagenkapitel, konnte Eclipse und das OSGi-Komponentenmodell mit CBSE in Bezug gesetzt (siehe Kapitel 3), eine Konsolidierung des Prototyps auf Eclipse umgesetzt (siehe Kapitel 4 und 5) und ein Prozess zur Umwandlung von Altanwendungen auf ein Komponentenmodell entwickelt werden (siehe Kapitel 5).
Prinzipiell sind die abstrahierenden Konzepte des CBSE, von Eclipse und OSGi sowie der Prozess zur Umwandlung im schriftlichen Teil der Arbeit dokumentiert. Die CD-Rom enthält die technische Dokumentation der Arbeiten am Prototyp im Rahmen des Konsolidierungsprojekts.

Die Konsolidierung einer (monolithischen) Altanwendung auf Eclipse ist gelungen. Die im Soll-Zustand beschriebene erste Stufe des Projekts wurde erreicht und ist mit der Auslieferung von VOD 7.0.1.3 an Kunden abgeschlossen. Die Altanwendung wurde in Komponenten zergliedert, die Wiederverwendung konnte mit Fremdkomponenten gesteigert werden. Eine quantifizierbare Auswertung der Konsolidierung ist erfolgt. Die Vorgehensweise zur Konsolidierung in diesem Projekt lässt sich auf andere Projekte übertragen.

Ein Prozess zur Umwandlung von Altanwendungen auf ein Komponentenmodell konnte entwickelt und anhand eines Prototyps auf Tauglichkeit verifiziert werden. Dieser Prozess

ist mit Metriken auswertbar und liefert quantifizierbare Ergebnisse. Der Prozess abstrahiert soweit von der zugrunde liegenden Altanwendung, dass er für alle Programmiersprachen einsetzbar ist, die Namensräume anbieten. Java-Anwendungen eigenen sich in diesem Zusammenhang besonders für die Umwandlung auf das OSGi-Komponentenmodell. Andere Technologien haben sich auf Grund ihrer Beschaffenheit jedoch als weniger geeignet herausgestellt. In manchen Bereichen weist der vorgestellte Prozess gleichwohl Unzulänglichkeiten auf. Die Abhängigkeit von Anwendungswissen und die damit verbundenen Probleme bei der Komponentenidentifikation machen den Prozess schwer vorhersagbar.

6.2 Ausblick

Auf Grundlage der erfolgreichen ersten Stufe der Vitness-Konsolidierung, kann Versant das Projekt auf weitere Altanwendungen ausdehnen. Damit die Wiederverwendung von Komponenten nicht nur in den Grenzen dieses Projekts möglich ist, sollte ein organisationsweites Komponenten-Repository installiert werden. Um die Auswahl an Fremdkomponenten zu steigern, müssen externe Komponenten-Repositories und Komponentenmärkte evaluiert und erschlossen werden. Ein potenzieller Komponentenkandidat für das Vitness-Projekt, ist das Datatools-Projekt. Zur weiteren Bewertung der Ausrichtung auf CBSE müssen diverse Metriken im Anschluss an die Produktauslieferung erfasst werden. Mit dieser Datenbasis ließen sich weitere Daten zum Nutzen des CBSE berechnen. Aus den Metriken sollte eine Cost-Database erzeugt werden, die als Schätzgrundlage für zukünftige Projekte dient. Aufwände für die Erstellung von wiederverwendbaren Komponenten sollten nicht aus dem Projektbudget stammen, sondern als organisationsweite Kosten gewertet werden.
Eclipse weist in Bezug auf komponentenbasierte Software-Entwicklung verschiedene Schwachpunkte auf, deren Beseitigung für die Verbesserung dieses Komponentenmodells nötig sind. Zu nennen sind insbesondere die fehlenden Qualitätsstandards und der eingeschränkte Komponentenmarkt. Eine Ausweitung der Komponentenbasiertheit in die Java-Programmiersprache steht mit ”Dynamic Component Support for JavaSE” bevor.[35]
Eine Verbesserung des Prozesses kann in zwei Richtungen erfolgen. Zum einen kann durch eine erweiterte Werkzeugunterstützung die Gewinnung von Altanwendungswissen und damit die Identifikation von Komponentenkandidaten erleichtern werden. Zum anderen zeigen [Li u. Tahvildari 2006] einen Ansatz zur Komponentenidentifikation durch die Repräsentation der Altanwendung als Graphen auf.[36] Die Integration dieser Erkenntnisse kann die Kandidatenidentifikation verbessern.
Die Komponentendienste („Component-Services”) deuten die Nähe zu Service-orientierten Architekturen (SOA) an. Inwiefern CBSE und SOA zusammenhängen, könnte der Kern einer weiteren Untersuchung sein.

[35] Das dazugehörige JSR findet sich unter `http://jcp.org/en/jsr/detail?id=291`
[36] Das dort genannte Werkzeug JComp stand zum Zeitpunkt dieser Arbeit nicht zur Verfügung.

Literaturverzeichnis

[Alvaro u. a. 2005] ALVARO, Alexandre ; ALMEIDA, Eduardo S. ; LEMOS MEIRA, Silvio R.: Software Component Certification: A Survey. In: *euromicro* 00 (2005), S. 106–113. `http://dx.doi.org/10.1109/EURMIC.2005.53`. – DOI 10.1109/EURMIC.2005.53. ISBN 0–7695–2431–1

[Atkinson 2002] ATKINSON, Colin: *Component-based product line engineering with UML*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002. – ISBN 0–201–73791–4

[Atkinson u. a. 2003] ATKINSON, Colin ; BÄR, Holger ; BAYER, Joachim ; BUNSE, Christian ; GIRARD, Jean-Francois ; GROSS, Hans-Gerhard ; KETTEMANN, Stefan ; KOLB, Ronny ; KÜHNE, Thomas ; ROMBERG, Tim ; SENG, Olaf ; SODY, Peter ; TOLZMANN, Enno: *Handbuch zur komponentenbasierten Softwareentwicklung*. Fraunhofer Institut Experimentelles IESE Software Engineering, 2003

[Balzert 1996] BALZERT, Helmut: *Lehrbuch der Software-Technik - Software-Entwicklung*. Spektrum Akademischer Verlag Heidelberg Berlin, 1996 (Lehrbücher der Informatik)

[Balzert 1998] BALZERT, Helmut: *Lehrbuch der Software-Technik - Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Spektrum Akademischer Verlag Heidelberg Berlin, 1998 (Lehrbücher der Informatik)

[Bauer u. a. 2000] BAUER, Markus ; NEUMANN, Rainer ; SCHULZ, B.: Von Anwendungen zu Komponentensystemen. In: *Proceedings of the Second German Workshop on Software-Reengineering*, 2000

[Beck 2004] BECK, Kent: *Extreme Programming Explained. Embrace Change*. 2. Addison-Wesley Professional, 2004

[Brooks 1995] BROOKS, Frederick P.: *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley Professional, 1995. – ISBN 0201835959

[Brössler 2000] BRÖSSLER, Peter ; SIEDERSLEBEN, Johannes (Hrsg.): *Softwaretechnik - Praxiswissen für Software-Ingenieure*. Hanser, 2000

[Cheesman 2000] CHEESMAN, John: *UML Components. A Simple Process for Specifying Component-based Software*. Addison Wesley, 2000. – 144 S.

[Chikofsky u. II 1990] CHIKOFSKY, Elliot J. ; II, James H. C.: Reverse Engineering and Design Recovery: A Taxonomy. In: *IEEE Software* 07 (1990), Nr. 1, S. 13–17. http://dx.doi.org/10.1109/52.43044. – DOI 10.1109/52.43044. – ISSN 0740–7459

[Creasey 2006] CREASEY, Tod: *Designing Accessible Plug-ins in Eclipse*. http://www.eclipse.org/articles/Article-Accessibility/index.html. Version: 2006. – [Online; letzter Zugriff 08.12.2006]

[Eclipse 2006a] ECLIPSE: *Rich Client Platform*. http://wiki.eclipse.org/index.php/RCP. Version: 2006. – [Online; letzter Zugriff 22.12.2006]

[Eclipse 2006b] ECLIPSE: *UI Best Practices v3.x*. http://wiki.eclipse.org/index.php/UI_Best_Practices_v3.x. Version: 2006. – [Online; letzter Zugriff 08.12.2006]

[Eclipse 2006c] ECLIPSE: *User Interface Guidelines*. http://wiki.eclipse.org/index.php/User_Interface_Guidelines. Version: 2006. – [Online; letzter Zugriff 08.12.2006]

[Eclipse 2006d] ECLIPSE: *What is Eclipse*. http://wiki.eclipse.org/index.php/FAQ_What_is_Eclipse%3F. Version: 2006. – [Online; letzter Zugriff 22.12.2006]

[Fischer 2002] FISCHER, Peter: *Lexikon der Informatik*. Swisscom, 2002

[Fowler u. a. 2004] FOWLER, Martin ; BECK, Kent ; BRANT, John ; OPDYKE, William ; ROBERTS, Don: *Refactoring - Improving the design of existing code*. Addison-Wesley Professionals, 2004

[Gamma u. a. 1995] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns - Elements of reusable object-oriented software*. Addison-Wesley Professionals, 1995

[Gill u. Grover 2003] GILL, N. S. ; GROVER, P. S.: Component-based measurement: few useful guidelines. In: *SIGSOFT Softw. Eng. Notes* 28 (2003), Nr. 6, S. 4–4. http://dx.doi.org/10.1145/966221.966237. – DOI 10.1145/966221.966237. – ISSN 0163–5948

[Heineman 2001] HEINEMAN, George: *Component-based Software Engineering. Putting the Pieces Together*. Addison-Wesley Longman, 2001. – 416 S.

[Hunt u. Thomas 2005] HUNT, Andrew ; THOMAS, David: *The Pragmatic Programmer*. http://www.pragmaticprogrammer.com/ : Addison-Wesley, 2005. – 320 S.

[Li u. Tahvildari 2006] LI, Shimin ; TAHVILDARI, Ladan: A Service-Oriented Componentization Framework for Java Software Systems. In: *wcre* 0 (2006), S. 115–124. http://dx.doi.org/10.1109/WCRE.2006.7. – DOI 10.1109/WCRE.2006.7. – ISSN 1095–1350

[Lions 1996] LIONS, J. L.: *ARIANE 5 Flight 501 Failure*. http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html. Version: 1996. – [Online; letzter Zugriff 12.12.2006]

[McAffer u. Lemieux 2005] MCAFFER, Jeff ; LEMIEUX, Jean-Michel: *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications*. Addision Wesley, 2005. – 552 S.

[McIlroy 1968] MCILROY, M.D.: "Mass Produced" Software Components. In: NAUR, P. (Hrsg.) ; RANDELL, B. (Hrsg.): *Software Engineering*. Brussels : Scientific Affairs Division, NATO, 1968, 138–155. – Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968

[Nosek u. Palvia 1990] NOSEK, J. T. ; PALVIA, P.: Software maintenance management: changes in the last decade. In: *Journal of Software Maintenance* 2 (1990), Nr. 3, S. 157–174. – ISSN 1040–550X

[OSGi-Alliance 2006a] OSGI-ALLIANCE: *OSGi Service Platform Core Specification*. Release 4, Version 4.0.1. San Ramon, CA 94583 USA: The OSGi Alliance, 7 2006. http://www.osgi.org/osgi_technology/license_agreement.asp

[OSGi-Alliance 2006b] OSGI-ALLIANCE: *OSGi Service Platform Service Compendium*. Release 4, Version 4.0.1. San Ramon, CA 94583 USA: The OSGi Alliance, 7 2006. http://www.osgi.org/osgi_technology/license_agreement.asp

[Roock u. Lippert 2004] ROOCK, Stefan ; LIPPERT, Martin: *Refactorings in großen Softwareprojekten - Komplexe Restrukturierungen erfolgreich durchführen*. dpunkt.verlag, 2004

[Schmidt 2001] SCHMIDT, Douglas C.: *Why Software Reuse has Failed and How to Make It Work for You*. http://www.dre.vanderbilt.edu/~schmidt/reuse-lessons.html. Version: 2001. – [Online; letzter Zugriff 27.10.2006]

[Software-Evolution-Group 2003] SOFTWARE-EVOLUTION-GROUP: *Research Institute for Software Evolution*. http://www.dur.ac.uk/RISE/. Version: 2003. – [Online; letzter Zugriff 15.12.2006]

[Sotograph 2006] SOTOGRAPH: *Sotograph - User's Guide and Reference Manuals*. 2.4. Cottbus, 03044 Deutschland, 2006

[Szyperski 2002] SZYPERSKI, Clemens: *Component Software. Beyond Object-Oriented Programming*. Addison Wesley, 2002. – 448 S.

[Versant 2003] VERSANT: *VMC Monitoring Console User Manual*. 2.0. Fremont, CA 94555 USA, 2003

[Wikipedia 2006a] WIKIPEDIA: *Konsolidierung — Wikipedia, Die freie Enzyklopädie*. http://de.wikipedia.org/w/index.php?title=Konsolidierung&oldid=23540430. Version: 2006. – [Online; letzter Zugriff 16.11. 2006]

[Wikipedia 2006b] WIKIPEDIA: *OSGi — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=OSGi&oldid=95203213. Version: 2006. – [Online; letzter Zugriff 20.12.2006]

[Wu u. a. 2004] WU, Xiaomin ; MURRAY, Adam ; STOREY, Margaret-Anne ; LINTERN, Rob: A Reverse Engineering Approach to Support Software Maintenance: Version Control Knowledge Extraction. In: *wcre* 00 (2004), S. 90–99. http://dx.doi.org/10.1109/WCRE.2004.8. – DOI 10.1109/WCRE.2004.8. – ISSN 1095–1350

[Zukunft u. Raasch 2004] ZUKUNFT, Olaf ; RAASCH, Jörg: *Folien zu Software-Engineering 15. Projektmanagement*. Hochschule Angewandte Wissenschaften Hamburg. http://www.informatik.haw-hamburg.de/213.html. Version: 2004. – [Online; letzter Zugriff 27.10.2006]

Glossar

API Application Programming Interface (API), Programmierschnittstelle.

Bauhaus Hersteller eines Werkzeugs zur Software-Analyse (http://www.bauhaus-stuttgart.de).

COBRA Implementierung des JDO Standards für VOD.

COM Das Component Object Model (COM) ist ein von Microsoft geschaffener Komponentenstandard (http://www.microsoft.com/com/default.mspx).

CORBA Die Common Object Request Broker Architecture (CORBA) ist ein Komponentenstandard der Object Management Group (http://www.omg.org/corba/).

Dali-Projekt Ein Eclipse-Projekt zur Entwicklung von Entwicklungswerkzeugen für EJB3-Anwendungen (http://www.eclipse.org/dali/).

Datatools-Projekt Das Datatools-Projekt (DTP) zielt auf die Entwicklung einer Werkzeugplattform für Datenbanksysteme (http://www.eclipse.org/datatools).

Decompiler Ein Decompiler führt die Umkehroperation zu einem Compiler aus. Er erzeugt damit aus einem Kompilat die Quellen.

DLL Eine Dynamic Link Library kapselt Funktionalität und kann von mehreren Anwendungen eingebunden werden.

EJB Enterprise Java Beans (EJB) sind ein Komponentenstandard von SUN Microsystems (http://java.sun.com/products/ejb/).

EPL Die Eclipse Public License (EPL) ist eine von der Free Software Foundation (FSF) und der Open Source Initiative (OSI) akzeptierte Lizenz für Komponenten der Eclipse-Foundation. Komponenten können demnach (kommerziell) genutzt, verändert und vertrieben werden.

Gentleware Hersteller eines UML-Werkzeugs (http://www.gentleware.com).

Hardware-Plattform Zeichnet sich durch eine Computer- bzw. Prozessor-Architektur aus.

Hello2Morrow Hersteller eines Werkzeugs zur Software-Analyse (`http://www.hello2morrow.com`).

HTTP Das Hypertext Transfer Protokoll dient zum zum Transport von HTML-Seiten im Internet.

IDE Eine Integrated Development Environment (IDE) ist ein Werkzeug, das die Entwicklung von Computeranwendung unterstützt.

IDL Die Interface Definition Language (IDL) ist eine programmiersprachenneutrale Schnittstellenspezifikation (siehe API).

IP Multicast Bei IP Multicast werden Nachrichten an eine bestimmte Gruppe in einem IP-Netzwerk versandt.

JAR Ein Java Archive (JAR) bündelt 1 bis n Klassen.

Java Development Kit Das Java Development Kit (JDK) ist die Laufzeitumgebung und Klassenbibliothek der Java-Programmiersprache.

JavaDoc JavaDoc ist ein Werkzeug um Dokumentation über Java-API zu erzeugen.

JDO Java Data Objects (JDO) ist ein Persistenzstandard zum transparenten Speichern von Objekten in Datenbanksystemen (`http://java.sun.com/products/jdo/`).

JNI Das Java Native Interface (JNI) ist ein API um Aufrufbeziehungen zwischen Java und anderen Programmiersprachen herzustellen.

JRE Die Java Runtime Environment (JRE) ist eine von SUN bereitgestellte Laufzeitumgebung für die Java-Programmiersprache.

JVI Das Java Versant Interface (JVI) ist ein API, das den Zugriff auf VOD aus Java erlaubt.

Klebecode Klebecode bezeichnet Teile einer Anwendung, die zwei oder mehr Komponenten zur Interaktion befähigen.

LOC Die Lines of Code (LOC) ist eine Metrik, welche die Quelltextzeilen einer Anwendung angibt.

MagicDraw Hersteller eines UML-Werkzeugs (`http://www.magicdraw.com/`).

Meilenstein Ereignis mit besonderer Bedeutung im Projektmanagement. Dort sind diese Ereignisse Unter- bzw. Zwischenziele eines Projektes.

Model T Das Modell T war das erste massentauglich hergestellte Auto zwischen 1908 und 1927.

Namensraum Stellt einen Kontext für die darin enthaltenen Klassen einer Programmiersprache bereit.

OMG Object Management Group (`http://www.omg.org`).

Omondo Hersteller eines UML-Werkzeugs (`http://www.omondo.com/`).

Productline Engineering Fasst ähnliche Software-Systeme in einem gemeinsamen Produktionsprozess zusammen (`http://www.sei.cmu.edu/productlines/index.html`).

Produktversion Stellt eine technische Ausbaustufe eines Software-Systems dar.

Release-Kandidaten Stellt eine Produktversion im Entwicklungsprozess dar, die zur Veröffentlichung vorgesehen ist.

RMI Remote Method Invocation ist ein Java-Protokoll und erlaubt einen Aufruf von Methoden entfernter Objekte.

RPC Remote Procedure Call ist ein Protokoll, das den Aufruf von Prozeduren oder Subroutinen auf entfernten Rechnern erlaubt.

Schiebefensterverfahren Eine Schiebefensterverfahren oder sliding window ist aus der Datenflusskontrolle bekannt und bewegt sich dabei über einen Strom von Daten. Daten die innerhalb des Fensters liegen, haben einen besonderen Zustand.

SNMP Das Simple Network Management Protocol (SNMP) dient der Administration und Überwachung von Netzwerk-Geräten.

Software-Tomography GmbH Hersteller eines Werkzeugs zur Software-Analyse (`http://www.software-tomography.com`).

Sub/Master-Agent Ein SNMP-Agent ist die überwachende Instanz eines Netzwerk-Geräts.

Swing Eine von SUN entwickelte Fensterbibliothek für Java, die ohne einen spezifischen Betriebssystemanteil auskommt (`http://java.sun.com/products/jfc/`).

SWT Das Standard Widget Toolkit ist eine alternative Fensterbibliothek für die Programmiersprache Java (`http://www.eclipse.org/swt/`).

Unique Universal Identifier UUID ist ein Standard der Open Software Foundation, um ohne zentrale Instanz eindeutige Bezeichner zu erzeugen.

Update-Manager Eine OSGi-Komponente mit der Funktionalität Komponenten über das Internet zu aktualisieren oder zu installieren.

Update-Site Beinhaltet alle Komponenten und zusätzliche Informationen, die vom Update Manager benötigt werden.

Vitness Das in dieser Arbeit beschriebene Konsolidierungsprojekt.

XML Extensible Markup Langauge (XML) ist ein Standard zur Modellierung von halbstrukturierten Daten in Form einer Baumstruktur.

Die CD-Rom

Dieser Bachelorarbeit ist eine CD beigelegt, die als weitere Dokumentation dient. Im Wurzelverzeichnis dieser CD findet sich eine Liesmich-Datei, die den Inhalt und die CD-Struktur erläutert.

Erklärung zur Eigenständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) bzw.§24(4) ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 18.01.2007 Markus Alexander Kuppe