Interaktive Integritätsprüfung für komponentenbasierte Architekturen

Technische Unterstützung für Endanwender beim Anpassen komponentenbasierter Software

Dissertation

zur

Erlangung des Doktorgrads (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Markus Won

aus

Köln

Bonn 2004

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät
der Rheinischen Friedrich-Wilhelms-Universität Bonn.

1. Referent: Prof. Dr. Armin B. Cremers
2. Referent: Prof. Dr. Volker Wulf

Datum der Promotion: 11.2.2004

Danksagungen

Viele Menschen haben mich bei der Arbeit an dieser Dissertation unterstützt und waren mir eine wertvolle Hilfe. Zuerst möchte ich meinem Doktorvater, Professor Dr. Armin B. Cremers, für die wissenschaftliche Anleitung während der gesamten Zeit danken. Er gab mir wertvolle Hinweise, die meine Arbeit maßgeblich unterstützt haben. Weiterhin gilt mein Dank auch Professor Dr. Volker Wulf, der mir mit vielen Ratschlägen zur Seite stand und mit dem ich mich in Diskussionen intensiv mit dem hier behandelten Themenkomplex auseinandersetzte.

Weiterhin möchte ich allen meinen aktuellen und ehemaligen Kollegen aus dem Projektbereich ProSEC und aus der Arbeitsgruppe Software-Technologie bedanken, die mich in intensiven Diskussionen auf neue Aspekte der Thematik und Probleme hinwiesen.

Die praktischen Arbeiten wären ohne die Unterstützung durch mehrere Diplomanden nicht möglich gewesen. So wurde die Vorstudie „visuelles, komponentenbasiertes Anpassen“ von Sebastian Förster durchgeführt. Den TailorClient implementierte Michael Krüger. Matthias Krings konzipierte und implementierte die Event Flow Integrity und die Restricted Solutions Integrity. Auch ihnen möchte ich auf diesem Weg danken.

Schließlich gilt mein Dank auch Anja Keul, meinen Eltern und vielen guten Freunden, die mich während der gesamten Zeit motivierten und unterstützten.

Inhaltsverzeichnis

1 Einleitung

Viele moderne Applikationen bieten heutzutage Möglichkeiten zur Anpassung. Speziell bei Standard-Software und Groupware sind solche Anpassungsmöglichkeiten sehr wichtig (vgl. [Bentley und Dourish, 1995], [Henderson und Kyng, 1991]). Üblicherweise werden Anpassungen einer Software von den Benutzern als zusätzliche Aufgabe im Rahmen ihrer normalen Arbeit durchgeführt. Um diese Aktivitäten zu fördern und zu unterstützen, müssen Anpassungsumgebungen möglichst leicht zu erlernen sein und sollten den Anpassungsakt selbst technisch unterstützen.

Verschiedene Vorarbeiten stützen die These, dass sich komponentenbasierte Architekturen (vgl. [Szyperski, 1998]) besonders eignen, um hochgradig anpassbare Anwendungen zu entwickeln und auf diese Weise die oben geforderte Flexibilität zu erreichen. Die komponentenbasierte Software-Entwicklung basiert auf dem Wunsch nach erhöhter Wiederverwendbarkeit von Einzelkomponenten, die in unterschiedlichen Zusammenstellungen zu neuen Applikationen kombiniert werden können. Aus diesem Grund wird großer Wert auf Kapselung von Funktionalitäten, Vermeidung von Seiteneffekten und ausschließliche Kommunikation zwischen den Komponenten über wohldefinierte Schnittstellen gelegt. Dies resultiert bei der Entwicklung von komponentenbasierter Software in erhöhter Flexibilität, was spätere Änderungen angeht.

Zudem sind Komponentenarchitekturen aufgrund der Nähe dieses Ansatzes zu anderen Konstruktionstechniken auch für Endbenutzer leicht verständlich. Es hat sich gezeigt, dass Komponentenarchitekturen aufgrund aller dieser Eigenschaften eine gute Basis darstellen, um Systeme zu gestalten, die von Endbenutzern angepasst werden können (vgl. [Magee et al., 1995], [Stiemerling, 2000]).

Darüber hinaus zeigen verschiedene Studien, dass Unterstützungstechniken sowohl den ersten Zugang zur Software als auch die weitere Nutzung deutlich erleichtern. Die meisten solcher Konzepte zielen darauf ab, durch eigenes Erfahren, Explorieren oder durch kooperative Prozesse das Verständnis für die Software zu erhöhen und damit Anpassungen zu erleichtern [Engelskirchen, 2000], [Wulf, 2000]. Solche einschränkenden Hilfen, die auf den Anwendungskontext zugeschnitten sein sollten [Repenning und Summer, 1995],[Lewis und Olson, 1987], können regelbasiert definiert werden.

Im Rahmen dieses Promotionsvorhabens steht eine technische Unterstützung der Anpassungsaktivitäten, die in vielen Fällen einen explorativen Charakter [Engelskirchen, 2000] haben, im Vordergrund. Das Ziel ist es, durch zusätzliche Informationen Möglichkeiten zu finden, um technisch überprüfen zu können, inwieweit eine Anpassung sinnvoll im Sinne der Erwartungskonformität ist oder auch ob die resultierende Applikation lauffähig ist. Solche Integritätsprüfungen kommen in unterschiedlichen Bereichen der Informatik zum Einsatz. Im Gegensatz zu den beispielsweise im Bereich der Datenbanken eingesetzten Verfahren, die durch Reduktion des Anwendungsfeldes auf ein Modell ermöglichen, vollständig Integritätsbedingungen zu definieren, geht es hier nicht um die vollständige Erfassung der semantischen Korrektheit einer Applikation. Diese ist in den meisten Fällen zu komplex, um sie darzustellen. Eine Abstraktion des Codes auf seine Semantik (Modellbildung) schränkt auch hier (ebenso wie beim Übergang von der realen Welt auf die Modelle innerhalb der Datenbanken) die Möglichkeiten der Überprüfung ein. Semantische Korrektheit kann hier also nicht im mathematischen Sinne begriffen werden. Sie zielt vielmehr auf die Fragestellung ab, ob die Benutzer, die im Normalfall mathematisch ungeschult sind, durch Anpassungsoperationen Veränderungen im Sinne ihrer Erwartungen hervorrufen. Hier kann

eine Beschreibung der eingesetzten Komponenten helfen [Wulf, 1999]. Weiterhin lassen sich darauf basierend Integritätsbedingungen formulieren, die auch die Interaktionsmöglichkeiten von Komponenten untereinander einbeziehen.

Ebenso ist von Interesse, wie Hilfestellungen, Fehlermeldungen oder automatisch generierte Verbesserungsvorschläge helfen können, das Erlernen und entsprechend den eigenen Wünschen effektive Nutzen von Anpassungsoperationen zu erleichtern. Hinter all dem steckt eine Erleichterung bei der Bildung eines mentalen Modells einer Software. Dieser Ansatz ist insofern neu, als dass bisher keine technischen Unterstützungsmechanismen im Bereich der Endbenutzer-orientierten Anpassung angeboten wurden. Die Konzepte lassen sich ebenfalls in den Bereich der Applikationsentwicklung übertragen.

Diese Arbeit soll also Antworten auf folgende Fragen liefern:

1. Welche Eigenschaften und Anforderungen sind für eine Komposition und aus der Sicht einer Komponente zentral und sollten dementsprechend beschrieben werden? – Diese Frage zielt darauf ab zu untersuchen, was Kernfunktionalitäten einer Komponenten sind. Sie sind selbstverständlich abhängig von der untersuchten Komponente oder der Komponentenmenge. Allerdings sollten sich hier Gemeinsamkeiten herausarbeiten lassen, so dass darauf basierend ein Modell gebildet werden kann. Dieser Ansatz setzt auf die Arbeiten von Engelskirchen und Wulf auf [Engelskirchen, 2000], [Wulf, 2000]. Beide fügen den Komponenten natürlich-sprachliche Beschreibungen bei, die Anwendern helfen sollen, diese bei Anpassungen „richtig“ im Sinne des von den Entwicklern beschriebenen Kontexts zu nutzen.
2. Wie lassen sich diese Eigenschaften beschreiben? – Um eine automatische Prüfung der für eine Komposition relevanten Integritätsbedingungen durchführen zu können, müssen diese in einer für eine Maschine lesbaren Form vorliegen. Bei dieser Frage geht es vor allen Dingen um die Möglichkeiten der technischen Auswertung. Gleichzeitig sollten Integritätsbedingungen an und für Komponenten veränderbar bleiben. So muss eine Darstellungsform gewählt werden, die ebenso auch für Benutzer (Administratoren) handhabbar ist.
3. Wie lassen sich Kompositionen und die für sie relevanten Integritätsbedingungen im Sinne einer semantischen Korrektheit überprüfen? – Neben der Beschreibung eines ganzheitlichen Integritätskonzepts und darauf basierenden Definitionen von Integritätsbedingungen sollten diese auch maschinell auswertbar sein. Effiziente Prüfalgorithmen sind hier zu implementieren.
4. Kann eine so konzipierte Integritätsprüfung Benutzern helfen, Anpassungen einfacher und besser im Sinne der Erwartungskonformität anzupassen? – Die oben beschriebenen Integritätskonzepte sollen nicht in erster Linie genutzt werden, um automatisiert Kompositionen zu prüfen oder gar zu verändern. Sie werden im Hinblick der leichteren Nutzung von Anpassungssprachen für Benutzer entwickelt. Insofern sollten die Integritätskonzepte in einer Evaluierung auf Verständlichkeit und Lernförderlichkeit bei der Komposition geprüft werden.
5. Wie muss ein Anpassungswerkzeug beschaffen sein, um Benutzern dieses Konzept und die darauf basierenden Mechanismen geeignet als Ratgeber zur Verfügung zu stellen? – Weiterhin soll im Rahmen dieser Arbeit neben der Integritätsprüfung auf konzeptioneller Ebene auch eine Anpassungsumgebung entwickelt werden, die den Benutzern diese als Unterstützungsmechanismus zur Verfügung stellt. Diese muss daraufhin untersucht werden, ob sie das Anpassen und das Erlernen der Anpassungssprache erleichtert und das Verständnis für die eingesetzten Komponenten verbessert.

Der Fokus dieser Arbeit liegt dabei wie schon angesprochen auf der Unterstützung von Endanwendern beim Anpassen ihrer Applikationen. Die hier entwickelten Konzepte lassen sich jedoch auch allgemein im Bereich der komponentenbasierten Softwareentwicklung übertragen. Auch dort wird derzeit intensiv im Bereich von Unterstützungstechniken zur schnelleren und fehlerfreieren Entwicklung (z. B. Unit Testing) gesucht. Insofern lassen sich die in dieser Arbeit entwickelten Ergebnisse in diesen Bereich übertragen, wobei die größere Erfahrung im Umgang mit Entwicklungswerkzeugen sicherlich speziell am User Interface zu anderen Visualisierungskonzepten führen wird.

Um die oben genannten Fragen zu beantworten, ist die Arbeit wie folgt strukturiert (siehe Abbildung 1): Kapitel 2 gibt eine Einführung in die komponentenbasierte Anpassbarkeit. Dabei werden noch einmal die dafür benötigten Grundlagen aus dem Bereich der Software-Ergonomie und der Software-Technik aufgearbeitet. Vorarbeiten, die am Institut für Informatik III der Universität Bonn im Projektbereich für Software-Ergonomie und CSCW entwickelt wurden, werden vorgestellt und diskutiert. Im Mittelpunkt stehen hier die FREEVOLVE-Plattform [Stiemerling, 2000], ihre Anpassungssprache und verschiedene für sie entworfene Anpassungsumgebungen. Auf diese Komponentenarchitektur wird auch die prototypische Umsetzung des hier vorgestellten Ansatzes aufbauen. Weiterhin werden darauf aufbauend in Kapitel 3 allgemeine und speziell für den gewählten Ansatz relevante Faktoren und Konzepte beschrieben, die das Erlernen einer Anpassungssprache erleichtern. Sie orientieren sich an allgemeinen Kriterien für das Design ergonomischer Software, die speziell im Kontext von Anpassbarkeit diskutiert werden. Aus hier beschriebenen Schwächen und Problemen wird das weitere Vorgehen motiviert. Gleichzeitig ergeben sich so auch eine Reihe von grob formulierbaren Anforderungen, die es bei der Umsetzung eines Integritätskonzepts zu beachten gilt.

Das nachfolgende Kapitel 4 stellt Integritätskonzepte aus unterschiedlichen Teildisziplinen der Informatik vor. Dabei werden vor allem in den Bereichen der Informationssysteme und der Software-Technik existierende Ansätze untersucht. Sie werden vor dem Hintergrund diskutiert, sie als Basis für einen Integritätsbegriff für komponentenbasierte Architekturen verwenden zu können.

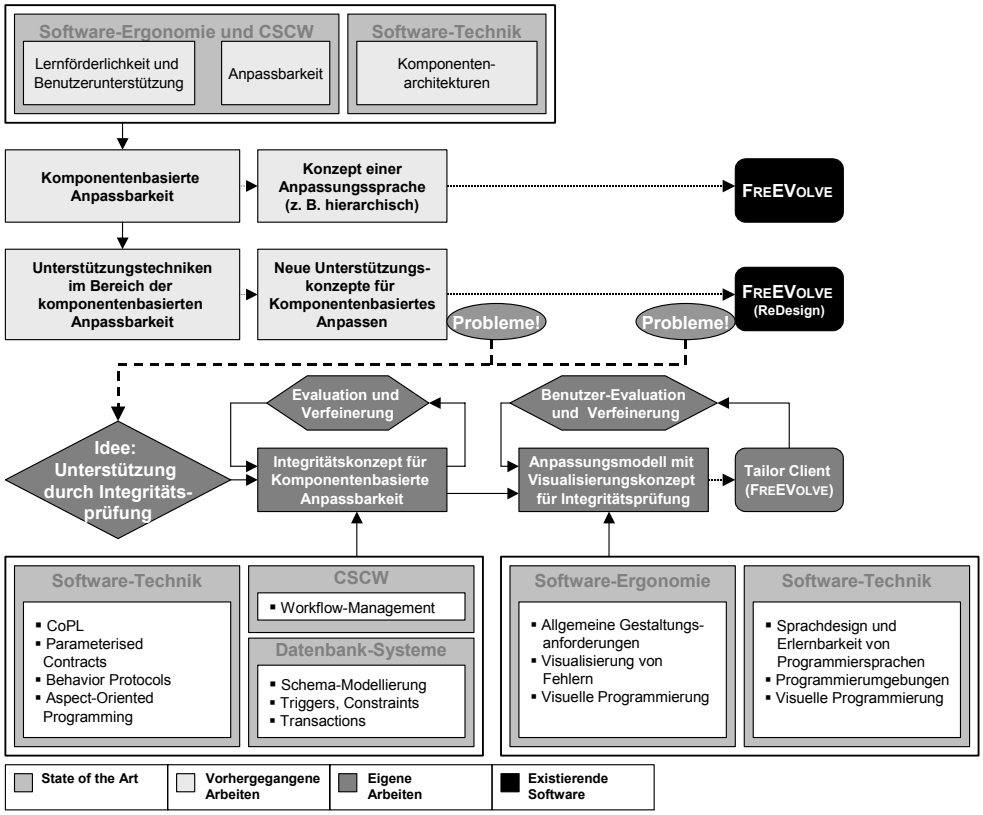


Abbildung 1: Überblick

Die Kapitel 5-10 stellen nun den in dieser Arbeit entwickelten Ansatz der Unterstützung von Anpassungsvorgängen durch Integritätsprüfungen vor. Dazu wird in Kapitel 5 eine neue Anpassungsumgebung (TailorClient) für die FREEVOLVE-Plattform vorgestellt. Sie soll vor allem unter dem Gesichtspunkt der Lernförderlichkeit konzipiert sein. Eine Evaluierung des Prototypen schließt das Kapitel. Im darauf folgenden Kapitel 6 werden die Anforderungen an ein zu entwickelndes Integritätskonzept noch einmal intensiviert dargestellt. In Kapitel 7 wird auf Basis von Constraints eine Integritätsprüfung entwickelt. Sie analysiert die grundlegenden Eigenschaften von Kompositionen (enthaltene Komponenten, deren Parameterisierung und Verbindungen zwischen ihnen). Kapitel 8 stellt eine weitere Prüfungsstrategie vor, in deren Fokus die Untersuchung von Nachrichtenflüssen innerhalb einer Komposition steht. Die grundlegende Idee hierbei ist zu sichern, dass Nachrichten, die von einer Komponente gesendet werden, Empfänger finden.

In vielen Fällen ist Anwendern von vorneherein klar, welche Art von Applikation sie konstruieren bzw. verändern wollen. In solchen Fällen ist es sinnvoll, Vorlagen (oder Schablonen) bereitzustellen, die möglichst nah am speziellen Anwendungskontext ausgerichtet sind. Aus diesem Grund werden in Kapitel 9 Application Templates vorgestellt. Sie vereinen eine Menge von Integritätsbedingungen, die sich so speziell auf einen Anwendungsfall maßschneidern lassen.

Kapitel 10 zeigt, wie die in den vorigen Kapiteln entwickelte Integritätsprüfung in den TailorClient integriert wird. Geeignete Visualisierungs- und Interaktionskonzepte stehen im Mittelpunkt des Interesses. Eine Evaluierung untersucht die Nützlichkeit dieses Ansatzes.

In einem abschließenden Kapitel werden schließlich noch einmal die wichtigsten Ergebnisse dieser Arbeit zusammengefasst. In diesem Teil der Arbeit wird dann auch ein Überblick über hier nicht beantwortete Fragen ebenso wie ein Ausblick auf mögliche Anschlussarbeiten gegeben.

2 Komponentenbasierte Anpassbarkeit

In den folgenden beiden Kapiteln soll die oben skizzierte Idee, die Lernförderlichkeit von anpassbaren Groupware-Systemen mit Hilfe einer interaktiven Integritätsprüfung zu verbessern, präzisiert und aus der Literatur heraus motiviert werden. Abbildung 2 skizziert die Zusammenhänge zwischen den unterschiedlichen Bereichen des State of the Art. Die Diskussion wird zeigen, wie diese Bereiche zusammenhängen.

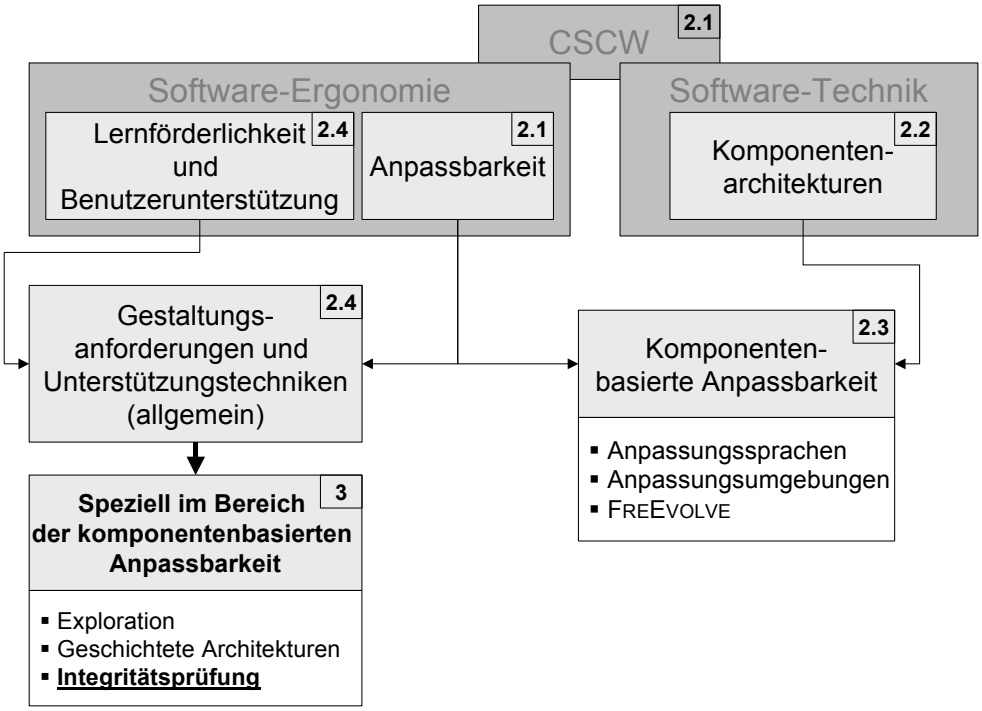


Abbildung 2: Kapitel 2 und 3 im Überblick

Zu Beginn wird gezeigt, inwieweit Anpassbarkeit als Eigenschaft von Software-Systemen deren Qualität deutlich erhöhen kann (siehe Abschnitt 2.1). Daraus ergeben sich zwei Fragen: „Welche Software-technische Basis eignet sich besonders gut, um mächtige Adaptionen zuzulassen?“ und „Wie können anpassbare Systeme so gestaltet werden, dass auch Endbenutzer leichten Zugang dazu haben?“ Eine mögliche Lösung auf die erste Frage bieten Komponentenarchitekturen (siehe Abschnitt 2.2). Sie wurden vor allem in jüngerer Zeit im Kontext höherer Wiederverwertbarkeit und damit schnellerer Entwicklungszyklen bei gleichzeitig sinkenden Kosten und verbesserter Qualität diskutiert. Es wird beschrieben, wie sich auf Basis dieses Ansatzes hochgradig flexible Anwendungen bei gleichzeitigem leichtem Verständnis für die darauf aufsetzende Anpassungssprache realisieren lassen (siehe Abschnitt 2.3).

Abschließend wird beantwortet, wie Anpassungsoperationen auch für unerfahrenere Endbenutzer handhabbar gestaltet werden können (siehe Abschnitt 3). Hier werden Untersuchungen im Bereich der allgemeinen Software-Ergonomie (Gestaltungskriterien) herangezogen und speziell im Kontext der Anpassbarkeit untersucht. Weiterhin werden verschiedene Ansätzen diskutiert, die sich mit der Frage beschäftigen, wie Programmieranfängern ein leichter Zugang in diese Materie ermöglicht werden kann. Hier gefundene Lösungen sollten berücksichtigt werden, wenn man versucht, Anpassungssprachen und -umgebungen lernförderlich zu gestalten. Die Diskussion fokussiert dabei auf den gewählten Ansatz, Anpassbarkeit auf Basis von komponentenbasierten Architekturen unterstützen zu wollen.

2.1 Anpassbarkeit von Software

Eine der wichtigsten Eigenschaften moderner Software ist es, flexibel in unterschiedlichen Kontexten einsetzbar zu sein. Dementsprechend bieten viele Applikationen Möglichkeiten zur Anpassung. Speziell bei Standard-Software und Groupware sind solche Anpassungsmöglichkeiten wichtig (vgl. [Bentley und Dourish, 1995], [Henderson und Kyng, 1991], [Kahler et al., 2000]). Beide Arten von Software werden in für die Entwickler nicht vollständig voraussehbarer Weise eingesetzt. Dementsprechend häufig treten im Verlauf der Nutzung Änderungswünsche auf. Diese resultieren vor allem aus geänderten Arbeitsbedingungen oder -aufgaben. Bei der Einführung von Groupware-Systemen in Organisationen ergeben sich Vorteile durch anpassbare Aspekte der Software vornehmlich dadurch, dass sich ein Teil des Entwicklungsprozesses in den Nutzungszeitraum verschieben lässt. Auf diese Weise können die vorher nur schwierig erfassbaren Anforderungen in der Praxis weiter analysiert werden. Die Software kann dann vom Entwicklerteam oder auch von den Benutzern selbst entsprechend den verfeinerten Anforderungen geändert werden. Dieses Vorgehen erweitert auch die Einflussmöglichkeiten der Nutzer auf den Entwicklungsprozess, wie es das *Participatory Design* (siehe auch [Nielsen, 1993]) vorsieht. Aus der Sicht der Software-Entwicklung stellen sich hier verschiedene Fragen:

- **Welche Aspekte der Software sollten anpassbar gestaltet sein?** Schon in der Analysephase sollte untersucht werden, welche Aspekte der späteren Software flexibel zu gestalten sind (vgl. [Dourish, 1992]).
- **Auf welcher technischen Basis lässt sich geeignet anpassbare Software entwickeln?** Diese Frage beschäftigt Software-Architekten schon seit langem. Im Folgenden (siehe 2.3) wird eine Möglichkeit aufgezeigt, wie sich auf Basis von Komponentenarchitekturen hochgradig flexible Anwendungen entwickeln lassen.
- **Wie kann das Erlernen und die tägliche Nutzung einer Anpassungssprache und -umgebung erleichtert werden?** Neben der Frage, welche Aspekte der Software flexibel gestaltet sein sollten, geht es hier darum, ein geeignetes Konzept für eine Anpassungssprache zu finden. Dies hat dann auch maßgeblichen Einfluss darauf, wie Anpassungsoperationen an bestimmten Elementen der Software verankert sind und zugänglich gemacht werden können. Weiterhin können auch bestimmte Unterstützungstechniken, die in Abschnitt 3 eingehender diskutiert werden, Hilfestellungen beim Anpassen geben.

2.1.1 Definition „Anpassbarkeit“ und Abgrenzung

Unter *Anpassbarkeit* wird die Eigenschaft technischer Systeme verstanden, Veränderungen des Systems als Reaktion auf unterschiedliche oder wechselnde Anforderungen zu ermöglichen (vgl. [Henderson und Kyng, 1991]). Diese zwar einfache, jedoch unscharfe Definition kann durch genauere Untersuchung verfeinert werden.

Adaptivität vs. Anpassbarkeit

In der Literatur werden die Begriffe *Adaptivität*[1] und *Anpassbarkeit* oder auch *Adaptierbarkeit*[2] häufig synonym verwandt. Eine genauere Untersuchung und Abgrenzung ist also nötig, um Missverständnisse zu vermeiden. Als Hauptunterscheidungsmerkmale zwischen Adaptivität und Adaptierbarkeit sind die Rollen des Initiators und die des Akteurs auszumachen [Haaks, 1999].

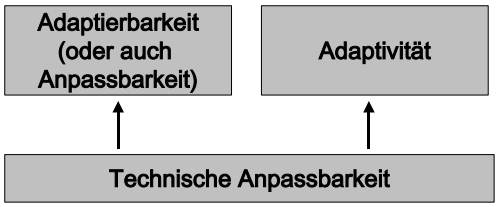


Abbildung 3: Anpassbarkeit

Unter *Anpassbarkeit* versteht man in der Literatur i. A. die Eigenschaft technischer Systeme, Benutzern Veränderungen am System selbst zu ermöglichen. Auf diese Weise lässt sich das System entsprechend der persönlichen Bedürfnisse und Anforderungen „maßschneidern”. Hier ist also der Benutzer sowohl Initiator als auch Akteur. Hier beschäftigt sich die aktuelle Forschung vor allem damit, wie geeignete Anpassungs-Interfaces konzipiert werden können.

Im Gegensatz dazu verfügen *adaptive Systeme* (auch selbst-adaptierbare) über eine Sensorik [Schneider-Hufschmidt et al., 1993]. Über die Sensoren eingehende Informationen werden ausgewertet und fließen in ein Situations- oder Umgebungsmodell ein. Auf Basis dieses Modells trifft das Kontrollsystem schließlich Entscheidungen, die zu einer Modifikation der Software führen. Auch im Falle adaptiver Interfaces verbleibt die Kontrolle über die Anpassungsmechanismen beim System selbst. Benutzerinteraktionen werden hier von den Sensoren aufgezeichnet und zu einem Benutzermodell verarbeitet. Im Falle adaptiver Systeme ist also das System selbst der Akteur einer Anpassungsaktion. Initiator bleibt jedoch (zumindest in dem hier vorgegebenen Kontext) ein Benutzer, dessen Verhalten durch Sensoren aufgezeichnet und verarbeitet wurde. Die technische Grundlage sowohl für Adaptierbarkeit als auch für Adaptivität eines Systems ist, dass das System anpassbare Aspekte bietet.

Beiden Ansätzen liegt zugrunde, dass die technische Basis grundsätzlich Veränderungen zulässt. Dies lässt kann man als *technische Anpassbarkeit* bezeichnen. Sie ist eine Eigenschaft des System, Flexibilität – wie sie gefordert wird – zu ermöglichen. Insofern bezieht sie sich auf die Wahl der Architektur, bestimmte Implementierungstechniken etc.

[1] Engl. *adaptivity*
[2] Engl. *tailorability*

Kühme et al. [Kühme et al., 1992] liefern eine feinere Klassifikation, anhand derer zwischen Adaptivität und Adaptierbarkeit unterschieden werden kann. Sie teilen einen einzelnen Anpassungsprozess in vier Teilschritte auf und untersuchen, ob das System oder der Benutzer Akteur sind:

1.	Initiative, eine bestimmte Funktionalität zu verändern
2.	Vorschlag über mögliche Veränderungen
3.	Entscheidung, eine bestimmte Alternative auszuwählen
4.	Ausführung einer gewählten Veränderung

Tabelle 1: Anpassungsprozess in vier Schritten

Auch hier bleibt jedoch das Problem der Entwicklung sinnvoller Benutzermodelle erhalten. Ob sich diese Modellbildung auf Basis von verwendeten Sensoren aus dem technischen Bereich wie z. B. Ferndiagnose von Telephonnetzen und damit verbundene Rekonfiguration auf menschliches Verhalten übertragen lässt, bleibt fraglich (siehe auch [Friedrich, 1990]). Probleme stellen hier u. a. die höhere Komplexität der Aktionen und die nur schwer analysierbare Motivation für eine bestimmte Aktion auf Seiten der Benutzer dar.

Trotzdem sieht auch Lieberman [Lieberman, 2001] einen fließenden Übergang zwischen Adaptivität und Benutzer-initiierter Anpassbarkeit. Schließlich werden viele Anpassungsoperationen von Menschen initiiert und auch primär ausgeführt. Die einzelnen Ausführungsschritte jedoch liegen in der Programmlogik begründet. Hier sind moderne Systeme in der Lage, sinnvolle Vorschläge zu unterbreiten oder Annahmen vorzunehmen und so bestimmte Alternativen auszublenden (s.o.).

Auch in der vorliegenden Arbeit soll keine scharfe Trennung dieser beiden Bereiche erfolgen. Grundlage hier sind von Benutzern initiierte Anpassungsvorgänge. Diese werden jedoch auf Basis von Modellannahmen über das anzupassende System unterstützt. Im Folgenden wird daher der Begriff Anpassbarkeit verwendet, wobei damit je nach Kontext sowohl der Begriff Adaptivität als auch die technische Anpassbarkeit gemeint sind.

Anpassung vs. Nutzung

Die Abgrenzung der Anpassung eines System von der Nutzung beleuchtet einen weiteren Aspekt des oben eingeführten *Anpassbarkeits*-Begriffs (vgl. [Oberquelle, 1994]): Anpassbarkeit wird verstanden als die Eigenschaft von Computersystemen, die „*Veränderbarkeit von während der Aufgabenerledigung stabilen Aspekten von Hilfsmitteln (Räumen, Verbindungen, Werkzeugen, Steuerungen) zu ermöglichen*“. Auf Basis dieser Definition kann man also den Unterschied zwischen Nutzung und Anpassung eines Computersystems leichter herausarbeiten.

Aspekt	**Nutzung**	**Anpassung**
Stabilität	Veränderung von während der Aufgabenerledigung unstabilen Aspekten	Veränderung von während der Aufgabenerledigung stabilen Aspekten
Gegenstand, auf den die Aktion wirkt	Veränderung des Dokuments	Veränderung des Werkzeugs, das zur Aufgabenerledigung dient
Zeitpunkt des Effekts	Wirkt sofort auf Aufgabenerledigung	Veränderung hat erst später (indirekt) einen Effekt auf die Aufgabenerledigung

Tabelle 2 : Nutzung vs. Anpassung (nach [Henderson und Kyng, 1991])

In obiger Tabelle wird auf verschiedene Aspekte Bezug genommen, bei denen der Unterschied zwischen der Anpassung eines Computersystems und Nutzung desselben deutlich wird. Allerdings ist diese Unterscheidung relativ zu sehen, da sie abhängig vom Benutzer und von seiner Aufgabe ist. Das Problem lässt sich an einem einfachen Beispiel veranschaulichen (siehe [Henderson und Kyng, 1991]).

Als Beispiel sei hier ein Benutzer genannt, der einen Text mit Hilfe eines Editors bearbeitet. Für diesen Benutzer sind alle Veränderungen, die er am Editor vornimmt, Anpassungen, wohingegen das Bearbeiten des Dokuments für ihn die Nutzung des Systems darstellt. Auf der anderen Seite kann man den Entwickler des Editors betrachten. Er verwendet einen Compiler zum Bearbeiten, d.h. zum Verändern des Editors. Für ihn sind Veränderungen, die er auf diese Weise am Editor durchführt, Nutzung (des Compilers), wohingegen Veränderungen am Compiler, wie z.B. das Ändern der Compiler-Einstellungen, für ihn Anpassungen sind. An diesem Beispiel lässt sich gut darstellen, dass nur eine gemeinsame Untersuchung des Artefakts und der daran vorgenommenen Operation Aufschluss darüber gibt, ob es sich um Nutzung oder Anpassung des Artefakts handelt.

End-User Development vs. Anpassung

Vom Bereich der Programmierung ist die Anpassbarkeit nur schwierig abzugrenzen, da – wie im Folgenden gezeigt wird – moderne Anpassungssprachen in ihrer Mächtigkeit zunehmen. Damit ist es für sehr erfahrene Benutzer möglich, mit Hilfe von Anpassungsoperationen große Teile der existierenden Software zu verändern. Gleichzeitig sind dafür Kenntnisse erforderlich, die es nicht erlauben, solche Personen als Endbenutzer zu bezeichnen. Sie brauchen ein weitgehend vollständiges Modell der Software, um die angesprochenen Änderungen durchzuführen. Geht es um Endbenutzer, so schränkt das nicht vollständige mentale Modell der Software die Möglichkeiten zur Anpassung ein. Zwei Definitionen, die im Rahmen eines Workshops zum Thema End-User Development [Lieberman et al., 2002] erarbeitet wurden, spiegeln dies wider:

- **„End-User Development is a set of activities that allows non-professional developers at some point to create or modify a software artefact".** End-User sind demnach keine Software-Entwickler und verfügen nicht über entsprechende Erfahrungen, um sich ein vollständiges mentales Modell der zu entwickelnden oder zu verändernden Software zu erarbeiten.
- **„End-User Development is a set of activities that allows for modifying an artefact which is used to perform a certain task."** Diese zweite Definition stellt die Aufgabe bzw. das dafür eingesetzte Werkzeug in den Vordergrund. Sie ist an der Definition des Begriffs Anpassbarkeit orientiert und berücksichtigt, dass eine Person das von ihr verwendete Werkzeug selbst verändert.

Beide Definitionen lassen sich durch entsprechende Beispiele so deuten, dass sie einem natürlichen Verständnis der Abgrenzung zwischen End-User Development und Anpassung entgegenstehen. Für die folgende Argumentation ist eine pragmatische Bestimmung jedoch ausreichend. Hier wird zugrunde gelegt, dass es sich bei den End-Benutzern im Normalfall nicht um erfahrene Programmierer handelt, sondern zumindest um Menschen, die kein vollständiges mentales Modell des zu verändernden Artefakts haben. Sie verändern ein Artefakt, dass sie selbst nutzen. D.h., die Anpassung oder Eigenentwicklung wird vorgenommen, um eine bessere Abstimmung des selbst genutzten Werkzeugs auf eine Arbeitsaufgabe zu erreichen. Insofern reicht es für den Kontext dieser Arbeit aus, beide Begriffe (End-User Development, Anpassung) synonym zu verwenden.

Problematik der wachsenden Komplexität

Üblicherweise werden Anpassungen einer Software von den Benutzern als zusätzliche Aufgabe im Rahmen ihrer normalen Arbeit durchgeführt. Um diese Aktivitäten zu fördern und zu unterstützen, müssen Anpassungsumgebungen möglichst leicht zu erlernen sein und sollten den Anpassungsakt selbst technisch unterstützen. Auf der anderen Seite steigt mit zunehmender Erfahrung der Benutzer mit ihrer Software der Wunsch nach immer weitreichenderen Anpassungsmöglichkeiten. Henderson und Kyng [Henderson und Kyng, 1991] unterscheiden drei grundsätzliche Komplexitätsniveaus:

Art der Anpassung	Erläuterung
Einfache Alternativenauswahl	Ein Programm wird angepasst, indem eine Alternative ausgewählt wird. Ein Beispiel dafür ist die Wahl eines sogenannten „Skins“ zur Veränderung des GUI[3].
Konstruktion neuen Verhaltens auf Basis existierender Elemente	Neues Programmverhalten wird erzeugt, indem man bestehende Elemente des Programms in einer neuen Art kombiniert. Dies kann sowohl auf graphischer Basis (Verändern der Menüleisten eines Word-Prozessors) als auch auf textueller Ebene (Makroprogrammierung) geschehen.
Re-Implementierung	Das Programmverhalten wird dadurch verändert, dass der Programmcode verändert wird. Hierunter fallen sowohl Änderungen am ausführbaren Code („Patch“) als auch Änderungen am Source-Code inkl. erneuter Kompilierung[4].

Tabelle 3: Anpassungsmechanismen nach Henderson und Kyng

Offensichtlich steigt mit wachsender Komplexität deren Ausdruckskraft. Anpassungssprachen müssen gezielt im Hinblick auf die erforderliche Ausdruckskraft entwickelt werden. Wenn ein hoher Grad an Anpassungskomplexität erforderlich ist, sollten Operationen auf unterschiedlichen Komplexitätsstufen für eine gleichmäßige Steigerung („gentle slope“) in dem Sinne sorgen, dass Benutzer mit einfachen Operationen erste Schritte unternehmen können und sich allmählich in Bezug auf die Komplexität ihrer Anpassungen steigern können [MacLean et al., 1990]. Dies bezieht sich in erster Linie auf das grundlegende Design der Anpassungssprache und der mit ihr zur Verfügung gestellten Operationen.

In Abschnitt 2.2 werden komponentenbasierte Architekturen diskutiert. Anschließend geht es darum, diese im Kontext anpassbarer Software genauer zu beleuchten (vgl. Abschnitt 2.3).

[3] Graphical User Interface (deutsch: Benutzungsschnittstelle)

[4] Der Unterschied zwischen der oben angesprochenen Makrosprache und einer hier verwendeten Programmiersprache besteht aus mathematischer Sicht nicht in einer größeren Mächtigkeit. Unterschiedlich ist aber die Umgebung, in der durch einen Code beschriebene Aktionen ablaufen. Bei Makrosprachen ist diese „Welt“ die Applikation selbst, während mit einer externen Programmiersprache die Umgebung durch Betriebssystem und Hardware beschrieben wird.

2.2 Komponentenarchitekturen im Überblick

Der Ansatz der komponentenbasierten Architektur [Szyperski, 1998] entstammt den klassischen Ingenieurswissenschaften. Hierbei versucht man, Komponenten zu konstruieren, die wiederverwendbar sind. Eine solche Komponente ist definiert durch ihre Funktionalität, ihre explizit beschrieben Schnittstellen zu anderen Komponenten und evtl. auftretende Abhängigkeiten. Auf diese Weise verkürzt man die Entwicklungszeiten bei der Konstruktion eines neuen Geräts, kann die Kosten senken und ein höheres Qualitätsniveau sichern. Dies kann dann durch Wiederverwendung oder Entwicklung durch Kombination existierender Komponenten erreicht werden. Beispiele hierfür finden sich im Maschinenbau oder in der Konstruktionstechnik, bei denen Standardbauteile (z. B. Schrauben, Muttern, Gewinde etc.) verwendet und zu größeren Einheiten zusammengefügt werden.

Übertragen in die Informatik bedeutet das, eine Applikation aus bestehenden Komponenten zusammenzusetzen. In Komponenten-Architekturen werden also Applikationen konstruiert, indem man einzelne Komponenten auswählt und miteinander verbindet. Es gab schon in den 1970er Jahren Stimmen [McIlroy, 1968], die im Zusammenhang mit der Softwarekrise[5] forderten, die Software-Entwicklung sollte sich die eben beschriebenen Ansätze der Ingenieurswissenschaften zum Vorbild nehmen, um qualitativ hochwertigere Software in kürzerer Zeit entwickeln zu können.

Im Folgenden soll der Begriff der Komponente ausgehend von der Definition erläutert werden, wobei unterschiedliche Komponentenmodelle untersucht werden. Dies führt zu einer Beschreibung der für die Praxis relevanten Eigenschaften von Komponenten bzw. Komponentenarchitekturen. Abschließend wird dann der Prozess der komponentenbasierten Softwareentwicklung, wie er in der Industrie stattfindet, in die oben angesprochene Anpassbarkeits-Diskussion überführt. Hier wird dann diskutiert, inwieweit sich auf Basis komponentenbasierter Architekturen hochgradig anpassbare und damit flexible Software-Systeme konzeptionieren lassen.

2.2.1 Definition „Komponente“

Der Begriff der Komponente ist von verschiedenen Autoren und zusätzlich in der praktischen Verwendung sehr unscharf und auch unterschiedlich verwendet worden. So ist beispielsweise eine Abgrenzung vom Begriff des Objekts oder der Klasse ebenso wenig sauber beschrieben wie auch eine Einordnung in die aus der Objekt-Orientierung bekannten Begrifflichkeiten nicht ohne weiteres möglich ist. Microsoft [Microsoft, 1996] bezeichnet eine Komponente als „eine kompilierte Softwareeinheit, die einen Dienst anbietet“. Auch die Beschreibung, es handelt sich bei einer Komponente um „eine kleinste Einheit eines Systems, die entfernt werden kann, ohne die Integrität des Systems zu berühren“ [Waskiewicz, 1995], ist nur wenig aussagekräftig.

Folgt man der Definition von Szyperski und Pfister [Szyperski und Pfister, 1996], die Szyperski [Szyperski, 1998] weiter ausformuliert und diskutiert, so lassen sich Komponenten als „Kompositionseinheiten mit vereinbarten (vertraglich festgelegten), spezifizierten Schnittstellen und ausschließlich explizit angegebenen Kontextabhängigkeiten beschreiben: „Eine Softwarekomponente kann unabhängig eingesetzt werden und ist Gegenstand der Komposition durch Dritte.“ Es handelt sich also üblicherweise um binäre Software-

5 Zu dieser Zeit wurde damit begonnen, größere zusammenhängende Software-Systeme zu entwickeln. Diese waren aufgrund vieler Abhängigkeiten einzelner „Module“ untereinander schlecht wartbar.

Einheiten, die unabhängig voneinander entwickelt, vertrieben und zusammengekauft wurden und die als ein System miteinander interagieren.
Daraus lassen sich verschiedene Grundannahmen ableiten:

- **Kontextfreiheit**: Eine Komponente ist unabhängig von anderen Komponenten in der Entwicklung. Abhängigkeiten zur Umwelt müssen dementsprechend berücksichtigt und genau beschrieben werden.
- **Komposition**: Eine Applikation wird aus mehreren Komponenten komponiert. Dabei sind die für die Komposition zuständigen Applikationsentwickler normalerweise nicht die Entwickler der Einzelkomponenten selbst.
- **Wohldefinierte Schnittstellen**: Sie dienen der Kommunikation zwischen einzelnen Komponenten und stellen somit die einzige Abhängigkeit zwischen Komponenten innerhalb einer Komposition dar.
- **Semantik einer Komponente**: Die Semantik einer Komponente muss durch Dokumentation und Verwendung der Schnittstellen detailliert beschrieben werden, um einen Einsatz in einer Komposition zu ermöglichen. Die exakte Funktionalität wird normalerweise nicht offengelegt, obwohl auch dies z. B. durch Distribution einer Komponente im Source-Code[6] geschehen kann.

Eine Komposition ist eine dann Auswahl von Komponenten, die sich teilweise parameterisieren lassen, und Verbindungen zwischen ihnen. Eine Komposition stellt dann eine Anwendung (Applikation) dar. Im Allgemeinen kommt es auf die gewählte Komponententechnologie an, welche Datenstruktur eine Komposition von Komponenten hat. Man kann sicherlich eine flache Struktur (eine Komposition besteht aus mehreren atomaren Komponenten) oder eine hierarchische Form (es gibt zusätzlich zu den atomaren Komponenten abstrakte Komponenten, die ihrerseits wiederum eine Komposition darstellen) unterscheiden.

2.2.2 Komponentenmodelle im Vergleich

Die in Szyperski [Szyperski, 1998] vorgestellte Definition eines Komponentenmodells ist zum Teil konzeptionell begründet, wird jedoch durch Beobachtung verschiedener bestehender Ansätze untermauert. Der folgende Unterabschnitt beschreibt anhand ausgewählter Komponentenmodelle die Unterschiede und deren Auswirkungen auf die Entwicklung von Applikationen ausgehend von diesen Modellen. Hier geht es nicht um eine vollständige Beschreibung der am Markt befindlichen und miteinander konkurrierenden Ansätze, vielmehr sollen Unterschiede und so auch eine sinnvolle Kategorisierung herausgearbeitet werden. Dabei muss die oben eingeführte zum Teil zu enge Definition wieder aufgeweicht werden.

[6] Dies entspricht nicht der von Szyperski eingeführten Definition einer Komponente, die aus seiner Sicht immer nur binär vertrieben werden kann. Da jedoch verschiedene Komponentenmodelle wie auch die JAVABEANS eine Distribution des Source-Codes nicht explizit ausschließen und sie zumindest für Dokumentationszwecke durchaus sinnvoll ist, wird hier eine weiter gefasste Definition von Komponenten verwendet.

	JAVA-BEANS	FLEXI-BEANS	COM	CCM	REGIS, DARWIN	Enterprise JAVA-BEANS	UNIX Shell-Scripts
Definition der Schnittstellen	Explizit	Explizit	implizit	Explizit in IDL	-	Explizit	implizit
Sprache für die Schnittstellendefinition	Java	Java	-	CORBA-eigene IDL	-	Java	-
Schnittstellennutzung	Events	Events, Shared Variables	Events (bidirektional)	Events (bidirektional über Facets und Receptangles)	Events	Facets (Methodenaufrufe)	Streams
Kompostionssprache	Java	CAT	p-Code	Corba-Kompositionssprache	Darwin	Java	Pipes
Zeitpunkt der Komposition	Während der Kompilierung	Zur Laufzeit	Zur Laufzeit	Während der Initialisierung	Zur Laufzeit	Zur Initialisierung	Zur Laufzeit
Persistenz der Komposition	Applikation wird monolithisch gespeichert	Applikation liegt als Plan vor	Applikation liegt als Plan vor	-	Applikation liegt als Plan vor	-	Applikation liegt als Plan vor
Distributionsform	Binär oder im Source-Code	Binär oder im Source-Code	Binär	Binär	Binär	Binär	Binär

Tabelle 4: Komponentenmodelle im Überblick

Tabelle 4 gibt einen grundsätzlichen Überblick über die hier betrachteten Komponentenmodelle. Die Kernunterschiede der einzelnen Komponentenmodelle ergeben sich vor allem durch den Kompositionsgedanken. Er führt zuerst zu einer Betrachtung der Schnittstellen:

- **Definition der Schnittstellen**: Hier stellt sich die Frage, wann und wie die Schnittstellen definiert werden. Dies kann innerhalb der Komponente geschehen oder extern (JAVABEANS, CORBA) [RAJ, 1998] formuliert werden. Erster Ansatz führt dazu, dass die Schnittstellen nicht mehr explizit beschrieben werden, sondern sich aus der Komponente ergeben. Nachteilig ist dann, dass eine zusätzliche Dokumentation beigefügt werden muss, um diese Schnittstellen zu beschreiben. Zweiter Ansatz ist laut Szyperski [Szyperski, 1998] eigentlich derjenige, der im Rahmen von Komponentenarchitekturen vorgesehen ist. Er legt nahe, dass eine explizite Schnittstellenbeschreibung existiert, auf Basis der Komponenten entwickelt werden. Die Verwendung der Schnittstellendefinition sichert dann, dass die Komponenten miteinander interagieren können.
- **Sprache für die Schnittstellendefinition:** Legt man zugrunde, dass eine externe Schnittstellendefinition existiert, lassen sich die unterschiedlichen Ansätze auch dahingehend unterscheiden, in welcher Sprache die Schnittstellen definiert werden. In CORBA [OMG, 1999]werden Schnittstellen in einer von der OMG definierten Interface Definition Language (IDL) beschrieben. Diese lassen sich dann kompilieren, wobei auch die Implementierung in verschiedenen Programmiersprachen erfolgen kann, ohne dass dies das Zusammenspiel der Komponenten beeinträchtigt. Da Java die Integration von Fremdkomponenten im Rahmen der Definition von JAVABEANS [JavaSoft, 1997] nicht vorsieht, werden hier die Schnittstellen ebenfalls in Java beschrieben.
- **Schnittstellennutzung:** Ein weiterer Unterschied ergibt sich durch die Form, wie Daten zwischen Komponenten über die definierten Schnittstellen ausgetauscht werden bzw. welche Datentypen zum Transfer zugelassen sind. Hier kann man unterscheiden, ob die Kommunikation zwischen den Komponenten mit (unidirektionaler) Versendung

von Nachrichten abläuft[7], ob es gemeinsam benutzte Variablen gibt (über deren Aktualität die Komponenten automatisch benachrichtigt werden) oder ob gar Streaming-Verbindungen möglich sind. Der Unterschied ist hier aufgrund von praktischen Überlegungen greifbar, auf konzeptueller Ebene zeigt Stiemerling [Stiemerling, 2000], dass die beiden erstgenannten gemeinsam alle notwendigen Kommunikationsmuster abdecken können.

- **Kompositionssprache:** Auch hier unterscheiden sich die einzelnen Komponentenmodelle maßgeblich voneinander. Während in einigen Modellen die Komposition mit der gleichen Sprache beschrieben wird wie die Komponenten selbst (z. B. [JavaSoft, 1997]), wird in anderen Ansätzen eine Komposition explizit in einer anderen Sprache vorgenommen [Magee et al., 1995], [Ng und Kramer, 1995]. Zum einen kann so die Verständlichkeit der Komposition erhöht werden, zum anderen kann dies konkreten Einfluss darauf haben, wann aus vorhandenen Einzelkomponenten und einem Kompositionsplan eine Applikation erzeugt wird. Teilweise wird zur Komposition ein sogenannter Glue-Code erzeugt, der Zuordnungen zwischen Schnittstellen festlegt und evtl. sogar Transformationen vornimmt, um Schnittstellen aufeinander abzustimmen.
- **Zeitpunkt der Komposition:** Verschiedene Komponentenarchitekturen lassen sich weiterhin danach unterscheiden, wann ein Kompositionsplan ausgewertet und zusammen mit den existierenden Komponenten eine Applikation erzeugt wird. Eine frühe Bindung bei den JavaBeans wird durch einen Kompilierungsvorgang abgeschlossen. Danach liegt die Applikation grundsätzlich monolithisch vor. Ein extremes Gegenbeispiel stellen die Shell-Scripte unter UNIX [Newham und Rosenblatt, 1998] dar. Die einzelnen Komponenten sind kleine Programme (z. B. less), die per Pipes miteinander verbunden werden. Hier ist die Kompositionssprache die jeweilige Shellsprache. Die Interpretation des Scripts, also der Kompositionsvorgang, findet erst zur Laufzeit statt.
- **Persistenz der Komposition**: Aus dem Zeitpunkt der Komposition und der Kompositionssprache ergibt sich weiterhin, ob und wie eine Komposition persistent gemacht wird. Während kompilierte Kompositionen normalerweise als eine Applikation speicherbar sind, werden in anderen Modellen (wie oben am Beispiel der Shell-Scripte oder einer DARWIN-Applikation gezeigt) Kompositionen erst zur Laufzeit gebildet.

Unter dem Gesichtspunkt, komponentenbasierte Architekturen zu verwenden, um anpassbare Software zu entwickeln, lassen sich aus den o. g. Merkmalen Schlüsse ziehen, inwieweit die diskutierten Architekturen dafür geeignet sind. Hier spielen vor allem die drei letzten Punkte eine wichtige Rolle. Sie entscheiden, inwieweit eine Anpassung (Veränderung der Komposition) flexibel vorgenommen werden kann. In einigen Fällen führt dies dazu, dass die Applikation neu kompiliert werden muss, in anderen wird die Änderung sofort nach einem Neustart der Applikation umgesetzt. Optimalerweise (siehe FLEXIBEANS) können Änderungen an der Komposition sofort durchgeführt werden und sind dann auch unmittelbar verfügbar. In dieser Arbeit soll es darum gehen, die Veränderung von Kompositionen (Anpassung von Applikationen) mit einem Integritätsprüfungsmechanismus zu unterstützen. Insofern ist es sinnvoll, möglichst alle Möglichkeiten der Interaktion einer Komponente mit ihrer Umwelt zu untersuchen, um die Machbarkeit dieses Ansatzes zu sichern. Aus diesem Grund sollte eine solche Anpassungsumgebung auf einer Komponentenarchitektur basieren, die viele der Interaktionstechniken unterstützt.

[7] In Java wird hier ein Benachrichtigungsmodus verwendet, der dem Observer-Pattern nach Gamma [1995] entspricht.

Weitere Unterschiede ergeben sich durch folgende Aspekte, die jedoch im Kontext des komponentenbasierten End-User Development nicht maßgeblich wichtig sind:

- **Distributionsform**: Komponenten können (wenn man die Definition von Szyperski etwas aufweicht) sowohl im Source-Code als auch in binärer Form ausgeliefert werden. Diese Ablehnung gegenüber einer Distribution im Source-Code rührt vor allem daher, dass Szyperski eine nicht rein technische Perspektive auf Komponenten einnimmt. Sie führt dazu, auch Komponentenmärkte zu berücksichtigen, die sich nur dann gut etablieren können, wenn nicht das gesamte Innenleben einer Komponente offenbart wird und so leicht kopiert werden kann.
- **Kompositionswerkzeuge**: In den meisten Fällen wird die Komposition durch entsprechende Werkzeuge unterstützt. In vielen Fällen handelt es sich dabei um graphische Oberflächen, mit Hilfe derer sich Kompositionen direkt in einer WYSIWIG-ähnlichen Form (im Fall von GUI-Entwicklungen) anfertigen und bearbeiten lassen. Da im Rahmen dieser Arbeit ein komplett neues Anpassungswerkzeug konzipiert werden soll, ist dieser Punkt nicht ausschlaggebend für die Wahl einer Komponentenarchitektur.

Verschiedene Punkte sprechen hier für den Einsatz der FLEXIBEANS als Komponentenmodell. Weiterhin können so Vorarbeiten genutzt werden, die sich mit der Nutzung von Komponentenarchitekturen im Kontext von Anpassbarkeit beschäftigten (vgl. [Won, 1998], [Stiemerling, 2000], [Hinken, 1999]). Aus diesem Grund sollen im Folgenden nun weitere Vorteile dieses Ansatzes diskutiert werden:

- **Mehrfachverwendung von Schnittstellen gleichen Typs**: Neben der oben angesprochenen Erweiterung der Interaktionsmöglichkeiten bieten die FLEXIBEANS zusätzlich auch die Möglichkeit, mehrere Schnittstellen gleichen Typs zu verwalten. Diese werden dann anhand eines zu vergebenden Namens unterschieden.
- **Verteilte Kommunikation**: Weiterhin ist es möglich, verteilte Applikationen zu entwickeln, wobei die Komponenten-Interaktion über Rechnergrenzen hinweg transparent bleibt. Hier wird auf Funktionalität der RMI API [SUN, 1998] aufgesetzt.
- **Externe Komposition und Komposition zur Laufzeit**: Kompositionen werden explizit beschrieben und erst zur Laufzeit aus Einzelkomponenten zusammengesetzt. Da die FLEXIBEANS dediziert im Hinblick auf anpassbare Softwareentwicklung konzipiert wurden, werden Kompositionen der CAT-Sprache (Component Architecture for Tailorability) [Stiemerling, 1998] formuliert. Eine Komposition kann hier als eine Komponente ohne äußere Schnittstellen betrachtet werden. In ihr sind weitere Komponenten (atomare oder auch abstrakte) [Won, 1998] enthalten.
- **Hierarchische Komposition**: Die FREEVOLVE-Plattform basiert auf der Idee einer hierarchischen Komponentenarchitektur [Won, 1998]. Neben einfachen Komponenten, die als Klassen implementiert sind, besteht die Möglichkeit, in CAT vorgefertigte, zusammengesetzte Komponenten zu definieren. Diese enthalten dann entweder einfache Komponenten oder andere zusammengesetzte Komponenten. Außerdem ist es innerhalb dieser sogenannten abstrakten Komponenten möglich, Verbindungen festzulegen und Ports nach außen zu definieren.

Wie hier gezeigt unterscheiden sich die verschiedenen diskutierten Komponentenmodelle zum Teil gravierend voneinander. Im Kontext dieser Arbeit schließen damit einige Einschränkungen von vornherein bestimmte Modelle aus, die sich nicht für die Konzeption hochgradig anpassbarer Software-Systeme eignen. Deswegen – und aus Gründen der Praktikabilität – basieren die hier vorgestellten Implementierungen auf dem FLEXIBEANS-Komponentenmodell. Dies bedeutet für die Konzepte bezüglich Integritätsprüfung und damit einhergehender Visualisierung keine Einschränkung.

Nachdem nun unterschiedliche Modelle vorgestellt wurden, soll es im Folgenden darum gehen, den praktischen Einsatz von Komponentenarchitekturen in der Softwareentwicklung grob zu diskutieren und diese Erfahrungen in den Bereich des End-User Developments zu transferieren.

2.2.3 Komponentenbasierte Software

In der Software-Entwicklung lassen sich, wenn es um komponentenbasierte Software geht, zwei Rollen unterscheiden: Den Komponenten(framework)-Entwickler und den Applikationsentwickler. In der folgenden Beschreibung soll der Applikationsentwickler im Vordergrund stehen, da es in dieser Arbeit weniger um die Entwicklung von Komponenten geht, als vielmehr darum, wie Kompositionen von Anwendern selbst erzeugt oder verändert werden können. Dementsprechend ähneln diese Aufgaben denen der Applikationsentwickler. Betrachtet man die grundlegenden Operationen zur Komposition einer Applikation – unabhängig vom gewählten Komponentenmodell, so sind dies:

- **Auswählen von Komponenten**: Zu Beginn des Kompositionsprozesses steht die Auswahl der für die Applikation benötigten Komponenten. Diese wird bestimmt durch die Zielvorgabe und die Funktionsbeschreibung der Komponenten.
- **Parametrisieren der Komponenten**: Teilweise lassen sich Komponenten in ihrer Funktionalität verändern, indem man von außen zugängliche Parameter setzt.
- **Verbinden der Komponenten**: Anschließend lassen sich die ausgewählten und parameterisierten Komponenten entsprechend ihrer Schnittstellendefinitionen miteinander verknüpfen. Auf diese Weise wird die Kommunikation zwischen Einzelkomponenten innerhalb einer Applikation ermöglicht. Unter Umständen müssen nun noch einige Komponenten hinzugefügt werden, die als Adapter zwischen nicht zueinander passenden Komponenten dienen.

Weiterhin ist es auch möglich, das Verhalten einer existierenden Komponente zu verändern, indem Änderungen am Source-Code – falls dieser vorliegt – (mit anschließender Compilierung) oder am Maschinencode vorgenommen werden. Dies ist aber nicht die eigentliche Aufgabe des Applikationsentwicklers und sei deswegen nur der Vollständigkeit halber erwähnt. Probleme ergeben sich dabei vor allem dadurch, dass neue Versionen einer veränderten Komponente nicht verwendet werden können. Die Aktualisierungen berücksichtigen normalerweise nur die expliziten Schnittstellen, dementsprechend kann das komplette Innenleben einer Komponente sich verändern. Schon aus diesem Grund sollte diese Möglichkeit der Applikationsentwickler für die weitere Betrachtung grundsätzlich ausgeschlossen werden.

Wie beschrieben besteht die Arbeit eines Applikationsentwicklers also vorrangig darin, bestehende Software-Elemente zu komponieren. Sein Arbeitsergebnis ist also ein Plan für eine Applikation, dessen Einzelelemente von anderen entworfen wurden. Wann dieser Plan übersetzt wird, ist für die Arbeit des Applikationsentwicklers zweitrangig. Es resultiert aus dem verwendeten Komponentenmodell (s.o.). Weiterhin verändern sich so die Anforderungen an die Laufzeitumgebung[8] für die Komposition und damit auch evtl. das Verhalten der Applikation beim Start oder zur Laufzeit. Betrachtet man nun die Erstellung und Veränderung eines Kompositionsplans, so basiert diese Arbeit besteht auf den o. g. drei Techniken.

[8] Die eventuell auch für die Auswertung des Kompositionsplans zuständig ist (siehe FREEVOLVE).

In ähnlicher Weise wie Kompositionen von Anwendungsentwicklern modelliert werden, können auch die Anpassungsvorgänge eines Anwenders betrachtet werden, wenn die anzupassende Applikation eine Komposition aus Komponenten ist und die Anpassungsumgebung entsprechende Funktionen zur Anpassung bereitstellt. Insofern lassen sich hier (anpassende) Anwender und Applikationsentwickler miteinander vergleichen. Unterschiede ergeben sich vor allem in der Erfahrung mit den Kompositionstechniken, die in erster Linie mit der überwiegenden Tätigkeit (Nutzung einer Applikation vs. Gestalten und Verändern einer Applikation) zusammenhängen. So lässt sich ein fließender Übergang feststellen, wenn man die Erfahrungsachse um die Nutzertypen „Power-User“ und „Administrator“ erweitert.

Als Fazit bleibt also, dass sich die Erfahrungen bei der Unterstützung von Applikationsentwicklern durchaus übertragen lassen, wenn es darum geht, Endbenutzern das Anpassen von Software zu erleichtern. Im Kontext dieser Arbeit geht es dabei um die Analyse, wie ein Kompositionswerkzeug gestaltet sein sollte und welche Unterstützungstechniken (inkl. deren Visualisierung) geeignet sind, um das Verständnis für die Applikation zu erhöhen und damit einhergehend den Anpassungsvorgang bestmöglich zu erleichtern.

2.2.4 Entwicklung komponentenbasierter Software

Betrachtet man die mittlerweile recht gängige Arbeit von Softwarehäusern, komponentenbasiert zu entwickeln (Komponenten zu verwenden, um eigene Applikationen zu entwerfen), dann fällt vor allem auf, dass der Anteil der selbst entwickelten und wiederverwendeten Komponenten hoch, wohingegen der Anteil fremd zugekaufter Komponenten recht niedrig ist [Reussner, 2001]. Komponentenbasierte Softwareentwicklung unterscheidet sich also in der Praxis recht deutlich von der von Szyperski beschriebenen. Eine Antwort darauf versuchen Hau und Mertens [Hau und Mertens, 2002] zu geben. Sie übertragen die für Software im Allgemeinen gültigen Qualitätsanforderungen [ISO-9126, 2001] auf Komponentenarchitekturen und beschreiben eine Kategorisierung von Qualitäten, die für den Einsatz von Komponenten entscheidend sind. Nachfolgende Tabelle veranschaulicht dies:

Qualitätsmerkmal	Ausprägung
Zuverlässigkeit	Ausreifungsgrad, Fehlertoleranz und Wiederanlauf
Benutzungsfreundlichkeit	Verständlichkeit, Erlernbarkeit und Hilfsmittel
Effizienz	Durchsatz, Antwortzeiten und Ressourcenschonung
Wartbarkeit	Analysierbarkeit, Änderbarkeit, Stabilität und Testbarkeit
Portabilität	Anpassbarkeit, Installierbarkeit und Ersetzbarkeit

Tabelle 5: Allgemeine Eigenschaften von Komponenten [Hau und Mertens, 2002]

Diese Qualitätsmerkmale sind mitentscheidend für die Auswahl oder den Einsatz von Fremdkomponenten. Abstriche werden dann verglichen mit der Kostenersparnis beim Zukauf einer Komponente gegenüber der (potenziell optimalen) Eigenentwicklung. Teilweise sind diese Merkmale auch beim Design von Komponentenframeworks, die vordringlich durch Endbenutzer komponiert werden sollen, zu berücksichtigen. Dies soll im Folgenden diskutiert werden. Während grundlegende Merkmale wie Portabilität und Wartbarkeit für die Verwendung von Komponenten innerhalb einer Anpassungsumgebung keine große Rolle spielen, sind ein hohes Maß an Zuverlässigkeit oder Effizienz einzelner Komponenten

grundsätzlich immer von Vorteil. Letztere Qualitäten entscheiden immer über den Erfolg einer Applikation oder die Nutzung einer Komponente, wenn Alternativen existieren.

Speziell aber die Aspekte rund um die Benutzungsfreundlichkeit sollten bei der Entwicklung sowohl von Komponenten als auch von einer Anpassungsumgebung, die auf die Endbenutzer-Anpassbarkeit ausgelegt ist, Beachtung finden:

- **Verständlichkeit**: Kritikpunkt an externer Software ist häufig die Verständlichkeit der Funktionalität und damit genaue Vorgaben, wie sich eine Komponente in eine bestehende Komposition integrieren lässt. Je höher hier die Hürde, desto höher ist der Aufwand, der dann gegen Kosten für eine Eigenentwicklung gegengerechnet wird. Auch die Angst, Komponenten „falsch" zu verwenden und so die gesamte Applikation in Funktionalität und Performance negativ zu beeinflussen, schreckt vom Einsatz ab.
- **Erlernbarkeit**: Das Verstehen und Erlernen von Funktionalität einer Komponente lässt sich nicht klar voneinander trennen. Wohl aber lassen sich die Art und Weise, wie Informationen darüber zur Verfügung gestellt werden, aufspalten. Diese können sowohl durch den Aufbau der Komponente selbst, die Namen der Schnittstellen und verwendeten Typen als auch durch externe Dokumentation bereitgestellt werden. Während die erste Variante tendenziell eher dazu geeignet ist, eine Komponente durch Experimente zu erforschen, dient zweite Variante eher zum klassischen „Erlernen" der Funktionalität durch Studium der Dokumentation.
- **Nutzbarkeit**: Werkzeuge zur Komposition spielen eine wichtige Rolle bei der Verwendung von Komponentenarchitekturen. Solche Hilfsmittel sind nicht so sehr abhängig von einer Komponente oder einem Komponenten-Framework. Vielmehr sind sie abhängig von einem Komponentenmodell. Nichtsdestotrotz ist die Existenz und die Güte einer verwendeten Entwicklungsumgebung eine maßgebliche Einflussgröße bei der Wahl einer Komponentenarchitektur und damit auch eines konkreten Komponenten-Sets für ein Entwicklungsprojekt.

Sowohl der Wunsch nach einer Komponente, die sich in ihrer Funktionalität leicht verstehen lässt oder eine gute Dokumentation selbiger als auch die Forderung nach leicht bedienbaren Kompositionsumgebungen lassen sich auch auf den Bereich des End-User Development übertragen. Hier ist die Wahl des Architekturmodells noch weniger entscheidend und wird stark dominiert von Aspekten rund um die Benutzbarkeit, da hier auch die größte Hemmschwelle aufgrund fehlenden Verständnisses für eine Applikation zu suchen ist.

2.2.5 Flexible Softwareentwicklung auf Komponentenbasis

Die grundlegende Idee, mit Hilfe von Komponentenarchitekturen die Wiederverwendbarkeit von Software-Elementen zu erhöhen und damit qualitativ hochwertigere Software in kürzer Zeit produzieren zu können, geht einher mit dem Gedanken, dass sowohl die einzelnen Komponenten als auch das Komponentennetz flexibel verändert werden können. Damit wohnt dem Grundkonzept der Komponentenarchitekturen inhärent der Gedanke an Anpassbarkeit (aus Sicht der Applikationsentwickler) inne.

Wie schon oben beschrieben ist der Softwareentwicklungs-Prozess mit Hilfe von Komponenten für den Applikationsentwickler geprägt durch die Auswahl, das Konfigurieren und das Verbinden existierender Komponenten. Diese Arbeitsschritte lassen sich auch auf einer schon existierenden Anwendung (Komponentennetz) wiederholen. Auf diese Weise können Applikationen recht schnell veränderten Bedürfnissen angepasst werden. Wie hoch der Aufwand dabei im Einzelfall ist hängt von verschiedenen Parametern wie dem Umfang der Änderung und auch der zugrundeliegenden Komponentenarchitektur, die diese Arbeiten

fördern oder auch behindern kann (siehe Abschnitt 2.2.2 und 2.2.4), ab. Weiterhin kann eine Software auch recht schnell aus Effizienzgründen oder zur Behebung von Fehlern optimiert werden, wenn eine Komponente durch eine neuere, schnellere oder stabilere Komponente mit gleicher Funktionalität ausgetauscht wird.

Diese schon oben angesprochenen Aspekte auf die komponentenbasierte Software-Entwicklung legen die Vermutung nahe, dass solche Arbeiten auch von Anwendern selbst durchgeführt werden können. Voraussetzung dafür ist aus technischer Sicht, dass sich Änderungen möglichst leicht durchführen lassen und diese Änderungen unmittelbar sichtbar werden. Dafür sollte ein für Endbenutzer handhabbares Anpassungswerkzeug zur Verfügung stehen.

2.3 Komponenten im Kontext von End-User Development und Anpassbarkeit

In verschiedenen Arbeiten (siehe z. B. [Stiemerling et al., 2000]) wurde gezeigt, dass sich Komponentenarchitekturen gut eignen, um für Endbenutzer anpassbare Software zu entwickeln. Eine Studie [Baster et al., 2001] enthält eine Definition für die Begriffe *Komponente* und *Komponentenbasierte Entwicklung* aus dem Bereich der Geschäftsprozesse: „Predefined components allow business users to be self-reliant and to assemble their own applications. We define components as abstract, self-contained packages of functionality performing a specific business function within a technology framework. These business components are reusable with well-defined interfaces“. Auch diese Definition sieht die Beteiligung der Benutzer bei der Entwicklung ihrer Applikationen vor.

Wie oben gezeigt, sind zwei der Hauptprobleme beim Design anpassbarer Software, sowohl eine technische Basis zu finden, die Flexibilität unterstützt, als auch eine Anpassungssprache zu entwickeln, die auch unerfahrenen Benutzern den Zugang ermöglicht.

Um diese Probleme anzugehen, wurden in der Vergangenheit Anpassungsumgebungen für komponentenbasierte Software konzipiert und implementiert (vgl. [Won, 1998], [Hinken, 1999]). Dieser Ansatz zeigt, wie die Prinzipien komponentenbasierter Software-Entwicklung (vgl. [Szyperski, 1998]) auf die Gestaltung von Anpassungsumgebungen und -sprachen zu übertragen sind.

Ng und Kramer [Ng und Kramer, 1995] sowie Fossa und Sloman [Fossa und Sloman, 1996] verfolgten diesen Ansatz zur Konstruktion eines Werkzeugs, mit dem es möglich ist, hierarchisch strukturierte Komponentenarchitekturen zu visualisieren. Beiden Werkzeugen liegt die Kompositionsbeschreibungssprache DARWIN zugrunde. Schwerpunkt dieser Arbeiten sind die Darstellung und Veränderung von Komponentenarchitekturen in verteilten Systemen. Zielgruppe sind Entwickler solcher Systeme, denen ein hoher Kenntnisstand bezüglich der einzelnen Komponenten und der Anpassungsoperationen abverlangt wird.

Ein weiterer Ansatz wurde von Malone et al. [Malone et al., 1994] entwickelt. Auf Basis eines objektorientierten Modells ging es hier darum, ein möglichst anpassbares Werkzeug (OVAL) für den CSCW-Bereich zu entwickeln, mit dem sich auf relativ einfache Weise Applikationen erstellen lassen. Der Benutzer hat hier vier verschiedene Objektklassen, Objects, Views, Agents und Links, zur Verfügung. Durch Kombinieren und Verbinden einzelner Instanzen dieser Klassen ist es möglich, komplette CSCW-Applikationen zu konstruieren. Weiterhin lassen sich von den bestehenden Klassen neue ableiten, um so die Funktionalität zu erweitern. An verschiedenen Beispielen wurde gezeigt, wie mächtig dieses Werkzeug ist, indem verschiedene bestehende CSCW-Applikationen wie zum Beispiel Lotus Notes [Knäpper und Perc, 2000] oder gIBIS [Conklin und Begeman, 1988] in OVAL nachempfunden wurden. Allerdings ist ein Werkzeug wie OVAL, mit dem vollkommen

verschiedene Programme konstruiert werden können, nicht leicht zu bedienen, d.h. auch dieser Ansatz ist eher für Entwickler als für Benutzer interessant [Kahler, 1995].

Die beiden hier diskutierten Ansätze beschränken sich auf die grundsätzliche technische Machbarkeit und stellen dar, wie flexible Software komponentenbasiert entwickelt werden kann. In verschiedenen Studien (vgl. [Won, 1998] [Wulf, 1999]) wurde weiter gezeigt, dass die Idee von Software bestehend aus Einzelkomponenten nebst den dazugehörigen Anpassungsoperationen auch für Endbenutzer verständlich gemacht werden kann. Entscheidend hierfür ist eine leicht erlernbare Anpassungssprache bestehend aus Komponenten, die die Operanden dieser Sprache bilden, und darauf ausführbaren Operationen.

2.3.1 Operanden der Anpassungssprache: Komponentenarchitekturen als flexible Basis

Komponentenarchitekturen eignen sich – wie schon oben gezeigt – hervorragend, um flexible Softwaresysteme zu entwickeln. Durch die weitgehende Kontextfreiheit lassen sich Einzelkomponenten in unterschiedlichen Verwendungsformen einsetzen. Durch sinnvolle Namensgebung und Beschreibungen wird dies weiter unterstützt. Auf diese Weise lassen sich nicht nur theoretisch alle Bereiche der Software sehr flexibel konzeptionieren. Sogar vorher nicht antizipierbare Anpassungen können unter Umständen ermöglicht werden. Dies kann beispielsweise durch Verwendung von verschiedenartigen Komponenten-Sets geschehen, die vorher nicht gemeinsam konzeptioniert und entwickelt wurden.

Entscheidend für das Verständnis und die Bedienbarkeit der Anpassungsoperationen ist also die Gestaltung der Einzelkomponenten, ihrer Schnittstellen sowie auch Designentscheidungen im Hinblick auf offenes Zusammenspiel zwischen den Komponenten. Ein weiterer Erfolgsfaktor ist, die richtige Granularität beim Design der Komponenten zu wählen. Sie geht bisher üblicherweise aus der Zerlegung einer (fiktiven) Anwendung in Einzelkomponenten hervor, wobei dieser Arbeitsschritt selbst in der Forschung bisher nicht tiefgehend untersucht wurde.

Es zeigt sich, dass ein Trade-Off besteht zwischen maximaler Flexibilität des Komponenten-Sets auf der einen Seite und Bedienbarkeit und Verständlichkeit der Einzelkomponenten auf der anderen Seite. Einen Weg, mit diesem Problem umzugehen, bietet das in Abschnitt 2.2 eingeführte FLEXIBEANS-Komponentenmodell. Es erlaubt mit der Möglichkeit der hierarchischen Komposition, die Komplexität mit Hilfe weniger großer abstrakter Komponenten zu Beginn der Lernphase der Anpassungssprache oder für unerfahrenere Benutzer im Allgemeinen gering zu halten. Mit zunehmender Erfahrung und dem häufig damit einhergehenden Wunsch nach erhöhter Flexibilität können dann diese auch vollständig erforscht und in ihrem Inneren verändert werden. Auch dieser Ansatz lässt sich mit Vergleichen aus der realen Welt leicht veranschaulichen: So besteht ein Auto aus der Sicht eines Laien aus wenigen großen Komponenten (Reifen, Motor, Innenraum etc.). Ein Experte sieht den Motor als zusammengesetzte Komponente bestehend aus einer Vielzahl von Einzelkomponenten. Zusammenfassend lässt sich also sagen, dass die Auswahl an vorhandenen Komponenten und deren Granularität einen großen Einfluss auf die Erlernbarkeit der Anpassungssprache haben. Sie sollten demnach sorgfältig gewählt werden.

2.3.2 Operationen der Sprache

Das Design einer Anpassungssprache ist maßgeblich mitentscheidend für den Erfolg der darauf basierenden Software. Komponentenarchitekturen kommen – wie in Abschnitt 2.2 eingehend dargestellt – mit nur sehr wenigen Konstruktionsoperationen (auswählen, parameterisieren und verbinden von Komponenten) aus. Die gleichen Operationen können nun auch für den Anpassungsprozess verwendet werden (siehe Abbildung 4).

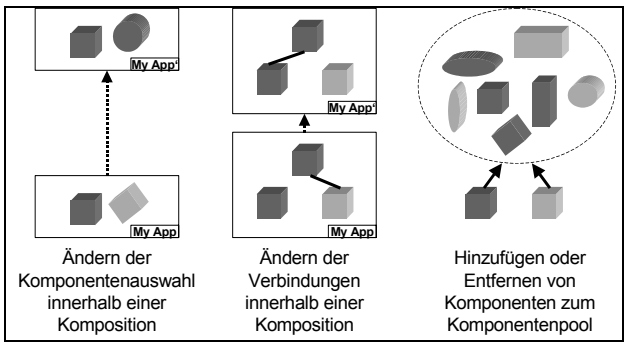


Abbildung 4: Anpassungsmechanismen in Komponentenarchitekturen

Dementsprechend lässt sich nun die in Abschnitt 2.1 von Henderson und Kyng eingeführte Kategorisierung verfeinern, wenn man sie in den Kontext komponentenbasierter Anpassung setzt.

Art der Anpassung	Im Kontext komponentenbasierter Architekturen	Erläuterung
Einfache Alternativenauswahl	Ändern der Parameter einer Komponente	Durch Ändern der Parameter einer Komponente lässt sich deren Funktionalität oder deren Aussehen verändern.
Konstruktion neuen Verhaltens auf Basis existierender Elemente	**Ändern der Komponentenzusammenstellung**	Verändern der Funktionalität einer Komposition (Applikation) durch Hinzufügen oder Entfernen einzelner Komponenten.
	Ändern der Verbindungen zwischen Komponenten	Verändern der Funktionalität einer Komposition durch Änderung der Verbindungen zwischen den enthaltenen Komponenten.
Re-Implementierung	Ändern der Implementierung einer Komponente	Durch Re-Programmierung einer Komponente kann deren Verhalten verändert werden.[9]

[9] Dieses Vorgehen ist problematisch und sollte deswegen sowohl bei der komponentenbasierten Applikationsentwicklung als auch bei der Anpassung vermieden werden. Es ist hier nur der Vollständigkeit halber aufgeführt.

	Ändern der zur Verfügung stehenden Komponenten für eine Komposition	Durch Erweitern/Verkleinern des Komponentenpools wird die potenzielle Funktionalität einer daraus zu bildenden Applikation verändert.

Tabelle 6: Kategorisierung der Anpassungsmechanismen

Wie man sieht, bietet die Wahl von Komponentenarchitekturen im Kontext anpassbarer Software vor allem auf der nach Henderson und Kyng definierten zweiten Stufe sehr einfache und trotzdem mächtige Anpassungsoperationen. Auf diese Weise bieten schon die ersten drei Operationen (zweite Spalte) Möglichkeiten, eine Anwendung komplett umzugestalten.

Zudem erleichtern komponentenbasierte Architekturen auch Entwicklern das Hinzufügen neuer Funktionalität durch Erweiterung des Komponentenpools. Neben der Re-Programmierung – hier die Veränderung des Codes einer einzelnen Komponente – ist es nun möglich, neue Komponenten, die weitgehend unabhängig[10] von vorhandenen sind, zu entwickeln.

2.3.3 Anpassungsumgebungen für Komponentenarchitekturen

In den vorigen beiden Unterabschnitten wurde mit der Beschreibung von Operanden und Operationen eine leicht verständliche Anpassungssprache konzeptualisiert. Die Einfachheit begründet sich hier sowohl durch die mit wenigen Operationen auskommende Anpassungssprache als auch durch die leichten Übertragbarkeit des Komponentenansatzes in bekannte Felder der Konstruktion.[11] Maßgeblich mitentscheidend für den Erfolg oder Misserfolg einer Anpassungssprache ist die dazugehörige Anpassungsumgebung. Diese kann von einem einfachen Texteditor, mit der sich Kompositionen verändern lassen, bis hin zu graphischen Werkzeugen reichen. Die meisten solcher Werkzeuge sind als Front-End zu verstehen, sie verbergen lediglich die textuelle Komposition bzw. stellen diese graphisch dar und erlauben so eine einfachere Bearbeitung.

Textuelle Beschreibung von Komposition

Basis der meisten textuellen Kompositionsbeschreibungssprachen ist objekt-orientierter Quellcode. Auch hier werden Komponenten (hier Objekte) instanziiert. Weiterhin werden über die Zuweisung von Methodenaufrufen, die als Nachrichten zwischen Objekten fungieren, Verbindungen zwischen Objekten beschrieben. Das JavaBeans-Komponentenmodell [JavaSoft, 1997] verwendet entsprechend dieser Idee die Sprache Java als Kompositionssprache.

[10] Ausgenommen sind hier gewünschte Schnittstellen zu vorhandenen Komponenten.

[11] In durchgeführten Workshops wurde das Konzept der Komponenten fast immer zu Beginn mit einem Vergleich zu den Spielzeugen „Lego“ oder „Playmobil“ eingeführt.

```
s_component VerySimpleSearchTool {
 subcomponent buttonSearch ControlButton;
 subcomponent engine SearchEngine;
 subcomponent switch ResultSwitch;
 subcomponent Output1 ResultList;
 subcomponent Output2 ResultList;
 subcomponent buttonLink1 ControlButton;
 subcomponent buttonLink2 ControlButton;

 bind buttonSearch.out → engine.in;
 bind engine.out →switch.dataIn;
 bind switch.dataOutGood →Output1.data;
 bind switch.dataOutBad →Output2.data;
 bind buttonLink1.out → Output1.L;
 bind buttonLink2.out → Output2.L;
}
```

Abbildung 5: Codebeispiel für eine Komposition in CAT

Abstraktere Beschreibungssprachen versuchen, einen Teil der Komplexität von „klassischen“ Programmiersprachen, zu verbergen. In ADL [Allen, 1997], DARWIN [Magee et al., 1995] oder auch CAT [Stiemerling, 1997], das sich an der DARWIN-Syntax orientiert, werden Kompositionen nach einem ähnlichen Muster beschrieben. Eine Oberkomponente enthält eine Menge typisierter Unterkomponenten. Diese können mit dem bind-Operator miteinander über entsprechende Ports verbunden werden. Zudem besteht die Möglichkeit offen zugängliche Parameter der Einzelkomponenten zu setzen. Abbildung 5 zeigt eine Komposition bestehend aus sieben Komponenten und einigen Verbindungen zwischen ihnen.

Graphische Anpassungsumgebungen

In modernen Entwicklungsumgebungen finden sich fast immer graphische Entwicklungswerkzeuge. Hier muss darauf geachtet werden, die Komplexität weitestgehend zu minimieren. Sie gelten als leichter verständlich als textuelle (z. B. [Green et al., 1991], [Myers, 1990]). Berücksichtigt werden muss dabei auch die Gestalt des resultierenden Programms. Ein linearer Programmablauf mit wenigen Ausnahmebehandlungen lässt sich einerseits recht gut textuell darstellen und auch verstehen [Green et al., 1991]. Andererseits können graphische Darstellungsformen dem Verständnis vor allem dann sehr zuträglich sein, wenn Programmflüsse gekennzeichnet sind durch Bedingungen, Verzweigungen, Schleifen und Nebenläufigkeiten. Hier eignen sich vor allem *Data Flow Visual Programming Language*s [Hils, 1992], bei denen die Darstellung des Programmablaufs in gerichteten azyklischem Graphen erfolgt.
In Bezug auf die Mächtigkeit der Entwicklungswerkzeuge kann unterschieden werden zwischen graphischen Programmier(unterstützungs)werkzeugen im Allgemeinen und graphischen Programmiersprachen. Während erstere lediglich als Zusatzwerkzeug für die Generierung von GUIs o. ä. zu verstehen sind, bieten graphische Programmierumgebungen darüber hinaus auch die Möglichkeit, den Programmablauf (s.o.) visuell zu gestalten. Dies ist umso leichter möglich, je eher sich die Sprache in einzelne, weitgehend unabhängige Elemente aufbrechen lässt, wie dies beispielsweise bei den komponentenbasierten Sprachen der Fall ist.

So gibt es im Bereich der objektorientierten Programmierung und auch in der komponentenbasierten Softwareentwicklung eine Reihe von integrierten Programmierumgebungen (IDE), die Entwickler bei dieser Arbeit unterstützen. So genannte CASE-Tools ermöglichen die Entwicklung von Programmen mit Hilfe von graphischen Elementen. Beispiele dafür sind Visual Age [IBM, 2002] oder auch Together [TogetherSoft, 2002]. Die graphische Darstellung einer GUI-Komponente ist häufig ähnlich der Komponentensicht zur Laufzeit (Visual Age). Andere Werkzeuge abstrahieren von dieser Sichtweise. Sie können

sich beispielsweise an der UML als mögliche Repräsentation von Komponenten orientieren. Hier wird der Wiedererkennungseffekt zu Gunsten detaillierterer Informationen über Schnittstellen oder wählbarer Parameter eingeschränkt.

Sie unterstützen die Konstruktion dadurch, dass Komponenten mit der Maus in eine bestehende Komposition eingefügt werden können. Verbindungsmöglichkeiten werden angezeigt. Die Parameterisierung einzelner Komponenten kann in Auswahlfenstern geschehen. Abbildung 6 zeigt eine mögliche Entwicklungsumgebung.

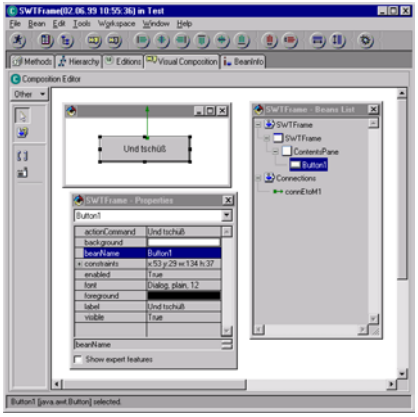


Abbildung 6: Beispiel für ein Kompositionswerkzeug (IBM Visual Age [Hawlitzek, 1999])

Die Eingaben des Benutzers werden hier interpretiert und es wird automatisch Java-Code erzeugt. Dieser kann anschließend nachbearbeitet und verfeinert werden. Auch diese Techniken lassen sich in den Bereich der Anpassungsumgebungen recht gut übertragen. Hier spielt das Design der Anpassungssprache und der gewählten Programmierparadigmen eine ebenso wichtige Rolle.

Die Wahl für ein komponentenbasiertes Design und die Verwendung von CAT als Kompositionssprache erleichtern die Entwicklung einer leicht zugänglichen graphischen Anpassungsumgebung. Die komplexe Programmlogik bleibt hier in den Einzelkomponenten verborgen und ist lediglich durch Parameterisierung änderbar. Eine hochgradig flexible Anpassung wird dadurch ermöglicht, dass sich der komplexe Kontrollfluss zwischen austauschbaren Komponenten ändern lässt.

Die schon in 2.2.2 beschriebene FREEVOLVE-Plattform wurde dementsprechend in verschiedenen Arbeiten genutzt, um anpassbare Software zu entwickeln. Dabei ging es fast immer auch darum, die Anpassungsschnittstelle Endbenutzer-gerecht zu gestalten. In [Won, 1998] wurde ein Suchwerkzeug für ein Groupware-System entwickelt. Die Anpassungsschnitte (siehe Abbildung 7) war sehr einfach gestaltet. Hier wurde geprüft, ob es grundsätzlich sinnvoll ist, komponentenbasiert Software zu entwickeln, die dann von Benutzern selbst angepasst werden könnte. Die Evaluierung belegte damals, dass der Ansatz grundsätzlich vielversprechend ist, jedoch die graphische Anpassungsumgebung um Unterstützungstechniken speziell für Endbenutzer erweitert werden sollte. Positiv gewertet wurde vor allem die Nähe zwischen der Sicht auf die Applikation zur Anpassungszeit und der Laufzeit-Applikation. Auch der schnelle und einfache Wechsel zwischen beiden Modi (die Applikation bleibt während der Anpassung vollständig funktionsfähig) erleichtern das Anpassen und Explorieren der Anwendung.

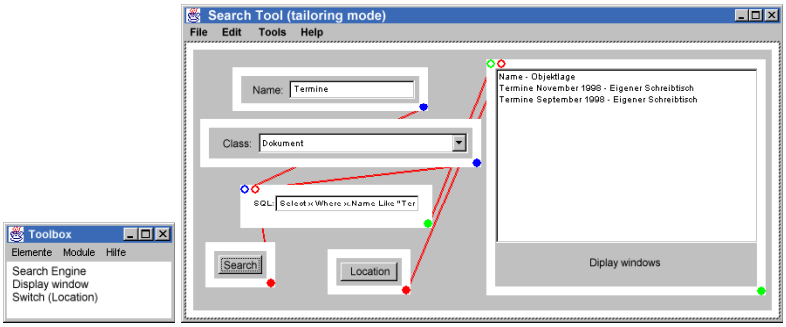


Abbildung 7: Komponentenbasiertes Suchwerkzeug

Engelskirchen [Engelskirchen, 2000] erweiterte diese Arbeit schließlich um verschiedene Techniken wie einen Explorationsmodus, Annotationsmöglichkeiten, bessere Beschreibungen und leichtere Bedienbarkeit. Auf diese Techniken wird in Abschnitt 3 detaillierter eingegangen.

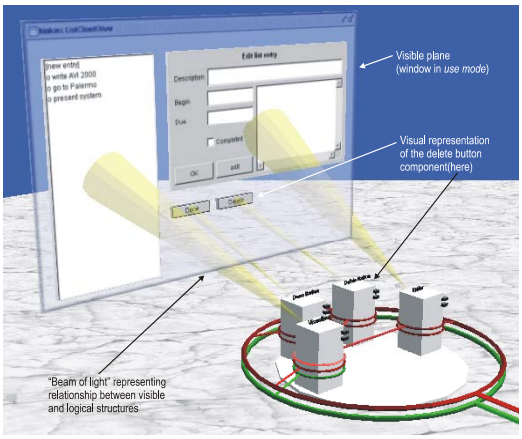


Abbildung 8: 3D-Anpassungsumgebung für FREEVOLVE

Schließlich implementierte Hinken [Hinken, 1999] die Client-Server-basierte Erweiterung von FREEVOLVE. Sie verfügt über eine Anpassungs-API, die es ermöglicht, alle wichtigen Methoden von außen zuzugreifen. Auf Basis dieser Schnittstelle entwickelte Hallenberger [Hallenberger, 2000] schließlich ein 3D-Interface (siehe Abbildung 8). Die Kompositionseinheiten sind hier als Quader in einem Raum dargestellt, Verbindungen durch Linien zwischen ihnen. Benutzer können durch diesen Raum navigieren und so die Applikation erforschen und verändern. Diese Benutzerschnittstelle ist recht abstrakt, die resultierende Applikation nur schwierig zu erkennen, was es speziell unerfahreneren Benutzern erschwerte, den Bezug zwischen dieser Darstellung und ihrer Applikation zur Laufzeit zu finden. Aus diesem Grund wurde dieser Ansatz nicht weiter verfolgt.

2.4 Diskussion – FREEVOLVE im Kontext komponentenbasierter Anpassbarkeit

Die Diskussion in diesem Kapitel hat gezeigt, dass sich komponentenbasierte Architekturen gut eignen, um hochgradig flexible Anwendungen zu realisieren. Die dabei verwendete Anpassungssprache eignet sich aufgrund der wenigen Operatoren und intuitiv verständlichen Operanden besonders gut für auch für unerfahrenere Benutzer.

MacLean et al. [MacLean et al., 1990] fordern, dass ein schrittweises Erlernen der Anpassungssprache möglich sein sollte, um Einsteigern den Zugang zum System zu erleichtern. Dieses Konzept wurde auch in verschiedenen anderen Anpassungssprachen umgesetzt.[12] Die FREEVOLVE-Plattform nutzt die vom FLEXIBEANS-Komponentenmodell zur Verfügung gestellte hierarchische Schachtelung. Auf diese Weise lassen sich die Komponenten in ihrer Komplexität und Granularität verschachteln. Anfänger haben die Möglichkeit, mit wenigen intern sehr komplexen und mit viel Funktionalität ausgestatteten Komponenten eine Applikation zu entwickeln. Mit wachsender Erfahrung und damit einhergehendem wachsenden Wunsch nach Optimierung lassen sich diese Komponenten dann selbst wiederum erforschen. Sie enthalten eine größere Anzahl kleinerer Komponenten. Dieser Ansatz erlaubt es zu Beginn, Komplexität vor den Anpassenden zu verbergen, ohne später die Mächtigkeit der Anpassungssprache einzuschränken. Ein weiterer Vorteil besteht darin, dass die Anpassungssprache bei steigender Komplexität gleich bleibt, lediglich die Operanden verändern sich. Auch die Visualisierung oder die Interaktionsmöglichkeiten verändern sich hier nicht. Am Beispiel eine Suchwerkzeugs für ein Groupware-System wurde dieses Konzept umgesetzt [Stiemerling und Cremers, 1998]. Weiterreichende Evaluierungen zeigten, dass dieser Ansatz für Endbenutzer sehr geeignet ist, um das Erlernen einer Anpassungssprache zu erleichtern [Wulf, 1999].

Die durch Komponentenarchitekturen ermöglichte Flexibilität wird maßgeblich erreicht durch die Zerlegung monolithischer Einheiten in Einzelkomponenten, die dann unter Einhaltung gewisser Regeln miteinander kombiniert werden können. So geht dann auch höhere Flexibilität bei der Anpassung immer einer mit einer höheren Modularisierung auf Komponentenebene.

Dadurch wird die Komplexität durch die Verwendung einer größerer Anzahl kleiner Komponenten bei der Anpassung erhöht. Dieses Problem lösen Stiemerling und Cremers [Stiemerling und Cremers, 1998] durch die Nutzung hierarchischer Kompositionsstrukturen (siehe Abschnitt 2.2.2). Weiterhin sind aber speziell für unerfahrenere Nutzer die Funktionsweisen, der passende Kontext und der Nutzen der einzelnen, sehr speziellen Komponenten nicht mehr ohne weiteres nachvollziehbar. Semantische Integritätsbedingungen können dann helfen, diese dennoch im Sinne der Entwickler zu nutzen.

[12] MS Word erlaubt beispielsweise das einfache Verändern der Menüleisten per „Drag and Drop“, das Aufzeichnen von Funktionsabläufen (Macro-Recording) oder auch das Entwickeln neuer Funktionalität mit Hilfe einer teilweise visuellen, integrierten Programmiersprache (VBA).

Ein weiteres Problem stellt das Design der Anpassungsschnittstelle dar. Anpassungsmechanismen – gerade wenn sie für Endbenutzer geeignet sein sollen – dürfen eine gewisse Komplexität in der Bedienung nicht überschreiten. Dies steht häufig in direktem Gegensatz zu der gewünschten Mächtigkeit der Anpassungsoperationen. Aus diesem Grund beschäftigt sich die Software-Ergonomie seit längerem damit, einfache, leicht zugängliche Anpassungssprachen zu entwerfen. Die bisherigen Werkzeuge sind insofern nicht zufriedenstellend, als dass bisher die Brücke zwischen einfacher Nutzung und Mächtigkeit nie vollständig geschlagen wurde. Allerdings erlaubt es die oben angesprochene API, neue Anpassungsschnittstellen einzubinden. Hier sollen existierende Visualisierungs- und Unterstützungskonzepte die Basis für ein neu zu entwickelndes Anpassungswerkzeug darstellen (siehe Kapitel 5).

3 Gestaltungsanforderungen an und Unterstützungstechniken für anpassbare Systeme

Nachdem Kapitel 2 die Grundlagen im Bereich der Anpassbarkeit und speziell der komponentenbasierten Anpassbarkeit eingeführt hat, soll es nun darum gehen, wie die das Erlernen und die Benutzung von Anpassungsmechanismen erleichtert werden kann. Wie sich in Abschnitt 2.3 gezeigt hat, ist der Erfolg einer Anpassungssprache und -umgebung wesentlich abhängig von deren Gestaltung. Aus diesem Grund sollen in diesem Abschnitt eben solche Gestaltungsrichtlinien aus der Software-Ergonomie diskutiert und in den Kontext des End-User Development gesetzt werden.

Verschiedene Autoren haben sich mit der Gestaltung ergonomischer Software beschäftigt. Sie stellen Leitfäden für die ergonomische Entwicklung von Applikationen zusammen [Nielsen, 1993], [Dix et al., 1998], [Raskin, 2000]. Dem hat auch die ISO Rechnung getragen mit der ISO 9241 [DIN/ISO, 1996]. In Teil 10 dieser Norm werden Grundsätze zur Dialoggestaltung beschrieben. Auch bei der Entwicklung von Anpassungsumgebungen für Benutzer sollten diese ergonomischen Konzepte berücksichtigt werden. Darüber hinaus ist es auch empfehlenswert, eine konsistente Bedienung zu ermöglichen (vgl. [MacLean et al., 1990], [Nardi, 1993], [Wulf, 1999]). Teilweise lassen sich Ansätze und Konzepte aus dieser Perspektive ebenso wie geschilderte Probleme oder Hemmschuhe beim praktischen Einsatz in der Software-Entwicklung auf den Bereich des End-User Development übertragen.

Pane und Myers beschreiben weiterhin in einem Übersichtsartikel [Pane und Myers, 1996] verschiedene Probleme, die Einsteigern das Erlernen von Programmiersprachen und -umgebungen erschweren, und daran angeknüpfte Lösungen. Die hier beschriebenen Ansätze konzentrieren sich darauf, wie existierende Programmierparadigmen (imperativ, objektorientiert) sich mit sinnvollen Änderungen oder Erweiterungen auf den Bereich des Erlernens von Programmiersprachen übertragen lassen. Dabei liegt die Annahme zugrunde, dass ein so entwickeltes Sprachkonzept als Einstieg in die „richtige“ Programmierung zu verstehen ist. Trotzdem lassen sich viele der hier vorgestellten Ansätze und Lösungen gut auf das Design der vorgestellten Anpassungssprache und die Anpassungsschnittstelle übertragen, die auf einer Komponentenarchitektur basiert.

Weiterhin soll auch der Bereich des End-User Development berücksichtigt werden. Er beschäftigt sich hauptsächlich aus kognitivistischer Sicht mit dem Design von Anpassungssprachen, die von Endanwendern bedienbar sind, und damit einhergehenden Problemen. Hier sind vor allem Arbeiten zu nennen, die untersuchen inwieweit visuelle Programmierungssprachen geeignet sind für das End-User Development [Green und Petre, 1996]. Andere Untersuchungen zielen mehr auf das sinnvolle Design von Programmiersprachen in Bezug auf deren Verständlichkeit für Programmieranfänger (vgl. [Bonar und Soloway, 1985], [Bruckman und Edwards, 1999], [Pane et al., 2001], [Soloway et al., 1983]). Vor allem, wenn man das Verhalten von Kindern bei der Programmierung untersucht [Bruckman und Edwards, 1999], [Smith et al., 1994] lassen sich grundsätzliche Erkenntnisse gewinnen, wie Programmieranfänger versuchen, natürlich sprachliche formulierte Gedanken in Programmiersprachen (textuell oder graphisch) umzusetzen und welche Probleme dabei entstehen. Dies untersuchen auch Repenning und Summer [Repenning und Summer, 1995], die eine visuelle Programmiersprache für eine 3D-Welt (Agentsheets) entwickelten. Die in dieser Welt aktiven Agenten können sowohl mittels direkter Manipulation als auch mit Delegationsbeschreibungen gesteuert werden. Auch die Agentsheets dienen vordringlich zur

Untersuchung, wie Anfänger Programme entwickeln und welche Sprachelemente am geeignetsten sind, hier einen leichten Zugang zu gewähren.

Sehr übersichtlich beschreibt [Nielsen, 1994], welche Aspekte bei der ergonomischen Gestaltung von Software beachtet werden sollten. Er bildet dazu zehn Kategorien: Sichtbarkeit des Systemstatus**,** Abgleich zwischen System und realer Welt, Benutzerkontrolle und -freiheit, Konsistenz und Standards, Fehlervermeidung, Bestätigung statt Erinnerung, Flexibilität und Effizienz, ästhetisches und minimalistisches Design, Unterstützung der Benutzer bei der Erkennung und Behebung von Fehlern, Hilfe und Dokumentation. Diese Kategorisierung ist recht allgemein gehalten und kann gut als Obermenge anderer Arbeiten betrachtet werden. Aus diesem Grunde soll sie für die weitere Diskussion (Abschnitte 3.1 bis 3.10) als Orientierung dienen. Hierbei sollen weitere Arbeiten integriert und im Kontext des End-User Programming auf Basis von Komponentenarchitekturen beleuchtet werden.

3.1 Sichtbarkeit des Systemstatus

Um ein System sinnvoll bedienen zu können, ist es unerlässlich, den aktuellen Systemstatus zu kennen. Dementsprechend sollten Signale und Hinweise den semantischen Status des Systems anzeigen (vgl. [Gellenbeck und Cook, 1991], [Green, 1990]). Weiterhin weisen Green et al. [Green et al., 1991] darauf hin, dass von Benutzern sämtliche visuelle Informationen als relevant eingestuft werden. Insofern sollte darauf geachtet werden, unwichtige Informationen auszublenden.

Auch für Anpassungsumgebungen gelten diese Forderungen. Demnach lassen sich verschiedene Ansätze aus dem Bereich der integrierten Programmierumgebungen, die Entwickler bei dieser Arbeit unterstützen, übertragen. So genannte CASE-Tools ermöglichen die Entwicklung von Programmen mit Hilfe von graphischen Elementen. Beispiele dafür sind Visual Age [IBM, 2002] oder auch Together [TogetherSoft, 2002]. Die graphische Darstellung einer GUI-Komponente ist häufig ähnlich der Komponentensicht zur Laufzeit (Visual Age). Hier werden die elementaren Bestandteile der Komposition visualisiert. Fenster, die zusätzliche Detailinformationen über die Einzelkomponenten (Parameter), deren Funktionsweise (Code) oder die zwischen Komponenten festgelegten Verbindungen enthalten, können bei Bedarf ein- und ausgeblendet werden.

Aus der Sicht dieses Promotionsvorhabens geht es allerdings nicht darum, die Arbeit von Entwicklern zu erleichtern, sondern die Benutzer eines Systems durch eine geeignete Anpassungsumgebung (für komponentenbasierte Architekturen) zu unterstützen. Deshalb sind diese Sichten weiter zu vereinfachen und die Sichten auf den Source-Code der Komponenten durch angemessene Benennung und Beschreibungen der Funktionalität zu ersetzen. Außerdem ist zu prüfen, welche Detailinformationen zusätzlich benötigt werden, um den Benutzern zu ermöglichen, sich ein adäquates mentales Modell der Software zu bilden.

Ein anderes Problem ergibt sich, wenn es darum geht, eine Applikation zu verändern, in der die Abhängigkeiten zwischen verschiedenen Elementen nicht klar sind (side effects). In diesem Zusammenhang weisen Green und Petre [Green und Petre, 1996] auf die Schwierigkeiten beim Verstehen von Code mit verteilten Referenzen und versteckten Abhängigkeiten hin. Insofern stellen Komponentenarchitekturen auch aus dieser Perspektive betrachtet eine günstige Möglichkeit dar, um das End-User-Programming zu erleichtern. Die Abhängigkeit zwischen Komponenten ist hier auf die Schnittstellen beschränkt.

An die im Folgenden zu entwickelnde Anpassungsumgebung stellen sich so verschiedene Anforderungen. Also sollten alle für das zu bearbeitende Artefakt, in diesem Fall eine Komposition, relevanten Aspekte dargestellt werden. Dies bezieht sich auf das Komponentennetz, die Darstellung an der Benutzungsschnittstelle und auch auf die eingestellten Parameter. Weiterhin sollten auch durch die Integrationsprüfung bemerkte Fehler markiert und in Abhängigkeit von ihrem Kontext angezeigt werden.

3.2 Erkennbarer Abgleich zwischen realer Welt und dem Computersystem

Für viele Programmieranfänger stellen ein hohes Abstraktionsniveau und der nur schwierig herstellbare Abgleich zwischen einem Problem und der programmierbaren Lösung große Hindernisse beim Erlernen einer Programmiersprache dar [Pane und Myers, 1996]. Insofern spielt die Natürlichkeit einer Programmiersprache eine wichtige Rolle bei deren Erlernbarkeit [Curtis et al., 1988].

Dies sollte auch bei der Konzeption einer Anpassungssprache beachtet werden. In dem hier beschriebenen Ansatz besteht die Anpassungssprache wie in Abschnitt 2.3 beschrieben aus Operanden und Operatoren. Die Operatoren sind mit Hilfe der „Lego-Metapher“ leicht verständlich zu machen, wie in [Won, 1998] gezeigt wurde. Die Operanden (Komponenten) haben zwar keine Entsprechung in der realen Welt, orientieren sich jedoch stark an der visuellen Darstellung der resultierenden Applikation. Hier wird auf die von verschiedenen Autoren (z. B. [du Boulay, 1989]) geforderte geeignete Metaphernwahl Bezug genommen. Die Zerlegung einer Applikation in Komponenten sollte sich demnach an der Perspektive des Nutzers auf deren Funktionalität orientieren und nicht an technischen Zusammenhängen. Auf diese Weise fällt es leichter, die Funktionsweise einer Einzelkomponente anhand ihres Namens oder einer beigefügten Kurzbeschreibung zu verstehen.

Weiterhin weisen verschiedene Autoren (z. B. [Nardi, 1993], [Lewis und Olson, 1987]) darauf hin, dass es für Endbenutzer häufig schwierig ist, mit Low-Level Primitiven zu programmieren. Diese sind in unüberschaubarer Weise der zu entwickelnden Anwendung nicht mehr zuzuordnen. Auch die daraus konstruierbaren hierarchisch aufgebauten Artefakte lassen sich für Endbenutzer nicht mehr mit der Aufgabe, die zu lösen ist, verbinden. Um Programmiereinsteigern den Zugang zu erleichtern, verwenden viele Trainer aus diesem Grund nur eine Teilmenge des gesamten Sprachschatzes, den sie sukzessive erweitern. Der Nachteil besteht hier darin, dass dennoch die Nähe zu Anwendungsebene fehlt und so die Motivation nicht klar wird. Der in FREEVOLVE verfolgte Ansatz geht ebenso von einer hierarchischen Struktur aus. Jedoch sind hier die zusammengesetzten Komponenten das Einstiegs-Level. Sie müssen nicht erst erzeugt werden. Stattdessen können Benutzer mit wenigen komplexen Komponenten, die dicht an der Arbeitsaufgabe orientiert sind, Applikationen entwerfen. Mit zunehmender Erfahrung können diese komplexen Komponenten, die selbst wieder Kompositionen darstellen, eingesehen und verändert werden [Won, 1998]. Schließlich gelten – wie schon in Abschnitt 2.3.3 gezeigt – speziell visuelle Programmierumgebungen als besonders verständnisfördernd.

Für die Gestaltung einer Anpassungsschnittstelle (siehe Kapitel 5) sind solche Aspekte ebenso zu beachten. Hier kommt als Schwierigkeit hinzu, dass eine Anpassungsumgebung als Applikation nicht dazu dient, um ein aus der realen Welt bekanntes Artefakt wie einen Brief (Textverarbeitung) o. ä. zu verändern, sondern eine weitere Computerapplikation. Umso mehr sollte darauf geachtet werden, mit der sinnvollen Metaphernwahl und einer verständlichen Darstellung einen Bezug zu Bekanntem herzustellen.

3.3 Benutzerkontrolle und Freiheit

Nach Nielsen [Nielsen, 1994] stellt die Kontrolle, die Benutzer über ihr zu bearbeitendes Dokument haben, eine sehr wichtige Eigenschaft dar. Diese Kontrollmöglichkeit sollte auch durch das System nicht weiter als nötig eingeschränkt werden. So ist auch die Freiheit der Anwender bei der Nutzung eine Dimension zur Bemessung der Güte der Software.

Diese beiden Forderungen lassen sich auch auf Anpassungsumgebungen gut übertragen. Hier sollte es Benutzern ebenso möglich sein, in breit gesteckten Grenzen Änderungen am Artefakt vornehmen zu können und über diese Änderungen die Kontrolle zu behalten.

Wie schon oben beschrieben bietet die FREEVOLVE-Umgebung eine gute Basis, um diesen Forderungen gerecht zu werden. In den bisher entwickelten Anpassungsumgebungen ist es in den durch die Anpassungssprache definierten Grenzen erlaubt, sämtliche im System zur Verfügung stehenden Komponenten zu nutzen. Das Verbinden und auch das Parameterisieren werden hier nur durch die Wahl der zueinander passenden Interfaces und Datentypen eingeschränkt.

Je höher der Freiheitsgrad bei der Anpassung von Software wird, desto wichtiger wird es, Funktionen bereitzustellen, die schnelles Testen erlauben. Nur so haben die Benutzer die Möglichkeit, die vollzogenen Änderungen am System mit den von ihnen intendierten abzugleichen [Lewis und Olson, 1987]. Dies kann beispielsweise durch schnelle Wechsel zwischen der Anpassungs- und der Laufzeitumgebung gewährleistet werden.

Auch in diesem Fall sollte auf die positiven Erfahrungen im Zusammenhang mit den Anpassungsumgebungen der FREEVOLVE-Plattform hingewiesen werden. Hier ist es unmittelbar möglich, zwischen Anpassungs- und Laufzeitfunktionalität zu wechseln [Won, 1998]. Darüber hinaus bleiben die Komponenten auch im Zeitraum der Anpassung vollständig funktionsfähig. Auf diese Weise wird man hier dem Wunsch nach einfacher Testbarkeit gerecht. Problematisch werden solche Tests, wenn man die voraussichtliche Funktionalität unvollständiger Kompositionen testen möchte. Hier können Integritätsprüfungstechniken helfen, die auch unvollständige Kompositionen auf ihr Verhalten testen und wertvolle Hinweise zur Vervollständigung liefern. Die Nutzung von Debuggern als Medium zur Verständnisbildung über eine Software beschreiben so auch Goldenson und Wang [Goldenson und Wang, 1991]. In einer Studie zeigte sich, dass mächtige Debugger von Anfängern häufig nicht nur zur Fehlersuche, sondern auch zum Verstehen von Programmen eingesetzt werden. Die Simulation eines Event-Flusses stellt eine solche Simulation ähnlich der Debugging-Funktionalität moderner Entwicklungsumgebungen dar. Hier lässt sich unter Zuhilfenahme detaillierterer Informationen über den Programmablauf die resultierende Funktionalität von Programmen prüfen.

Paul [Paul, 1994] stellt als Testumgebung zum sofortigen Explorieren einer Anwendung eine Szenarium-Maschine vor. Eine künstliche Systemumgebung erlaubt es, mit exemplarischen Daten Anwendungsszenarien durchzuspielen. Besonders geeignete Daten erleichtern dabei die Exploration.

Weiterhin wird von verschiedenen Autoren (z. B. [du Boulay, 1989], [Nardi, 1993]) gefordert, dass Änderungen, die an einem System vorgenommen werden, möglichst gut im laufenden Programm erkennbar sein sollten, um den Lernprozess zu unterstützen. Diesen Sachverhalt untersuchte du Boulay [du Boulay, 1989] genauer, als er sich mit dem Erlernen von Programmiersprachen beschäftigte. Dabei beobachtete er das Verhalten von Programmieranfängern, die häufig vor dem Problem standen, Änderungen im Code vorgenommen zu haben und diese dann im erneut compilierten Programm nicht wiederzufinden. Auch Nardi [Nardi, 1993] und Lewis und Olson [Lewis und Olson, 1987] stützen diese These. Sie untersuchen in diesem Zusammenhang auch den Erfolg von Tabellenkalkulations-

Anwendungen im Bereich des End-User Programming. Hier sind Änderungen am „Code“[13] ebenso wie in interpretierten Sprachen sofort sichtbar.

Für die Konzeption einer auf Lernförderlichkeit ausgerichteten Integritätsprüfung für eine komponentenbasierte Anpassungsumgebung sind solche Aspekte ebenso relevant. Zwar sind Seiteneffekte in Komponentenarchitekturen weitgehend ausgeschaltet, jedoch gelten auch hier gewisse Abhängigkeiten. Diese sollten sich durch Simulation visualisieren lassen. Eine schrittweise Ausführung einer Kontrollfluss-Analyse innerhalb des Komponentennetzes könnte also sehr hilfreich sein.

3.4 Konsistenz und Standards

Die konsistente Gestaltung spielt eine wichtige Rolle für die Erlernbarkeit und das leichte Verständnis. Konsistenz in Bezug auf die Bedienung – nicht nur innerhalb einer Applikation, sondern auch bei der Benutzung verschiedener Anwendungen – kann dadurch erreicht werden, dass die Style Guides oder Design Guidelines der Rechner- bzw. Betriebssystemhersteller beachtet werden (siehe beispielsweise [Apple, 1987], [Microsoft, 1995], [SUN, 2001]. Diese ähnliche Bedienung unterschiedlicher Anwendungen erleichtert die Einarbeitung in neue Applikationen. Die hier vorliegenden Beschreibungen gehen weit über das reine GUI-Design hinaus. Sie beschreiben zudem auch erwartungskonforme Reaktionen des Systems auf Benutzereingaben oder -fehler. Solche Anforderungen gelten selbstverständlich ebenso für die Gestaltung einer Anpassungsumgebung.

3.5 Fehlervermeidung

Besser als aussagekräftige Fehlermeldungen und die Unterstützung bei der Korrektur (siehe Abschnitt 3.9) ist es, das System so zu gestalten, dass Bedienungsfehler von vorneherein ausgeschlossen sind. Falls beispielsweise nicht sinnvolle Optionen bei der Bedienung gar nicht erst zur Verfügung stehen, werden Bedienungsfehler stark eingeschränkt und verhelfen so zu einer effizienteren Nutzung. Selbstverständlich gilt dies ebenso für Anpassungsumgebungen.

Solche Erfahrungen wurden auch bei der Entwicklung der FREEVOLVE-Plattform gesammelt. Möchte man zwei Komponenten miteinander verbinden, so klickt man auf den Ausgangsport der ersten Komponente. Nun kann man eine Verbindung zu einer Komponente durch Zeichnen einer Linie zu einem anderen (Eingangs-)port herstellen. Dabei werden Ports, die von der Typisierung her nicht zum vorher gewählten Ausgangsport passen, nicht als gültige Endpunkte der Verbindung akzeptiert. In Benutzerstudien hat sich gezeigt, dass diese Einschränkung durchaus sinnvoll ist, und die Bedienung gerade für weniger erfahrenere Benutzer deutlich erleichtert [Won, 1998]. Allerdings sind darüber hinaus führende Beschränkungen nicht immer hilfreich, sondern wirken gerade im Bereich der Anpassungsfunktionen oft sehr beschränkend. In vielen Fällen kann die Funktion einer Komponente für eine Applikation nicht im vorhinein vollständig spezifiziert werden. Damit sind Integritätsbedingungen, die für eben diese Komponente formuliert wurden und auf Basis derer das System über die Gültigkeit einer Anpassung entscheidet, per se nur unzurei-

[13] Tabelleneinträge können hier sowohl Werte als auch Formeln enthalten. Insofern lassen sich mit modernen Tabellenkalkulationen komplexe Berechnungen durchführen und man kann in diesem Zusammenhang durchaus von Programmierung sprechen.

chend. Sie können deswegen nicht als letzte Instanz zur Bewertung herangezogen werden. Hier bewegt man sich zwischen dem Wunsch nach möglichst exakter Fehlerkontrolle und -vermeidung und dem Wunsch nach größtmöglicher Freiheit beim Anpassen.

3.6 Bestätigung statt Erinnerung

Für die Bearbeitung komplexerer Aufgaben ist es wichtig, die bisher getroffenen Entscheidungen und die sich daraus ergeben Konsequenzen im Auge zu behalten. Das System sollte so gestaltet sein, dass die Nutzer dabei unterstützt werden. Beispielsweise kann dies der Fall sein, wenn die Bearbeitung einer Aufgabe erfordert, mehrere Bildschirmmasken, deren Inhalte voneinander abhängen, zu bearbeiten. In solchen Fällen sollten die wichtigsten Informationen permanent zur Verfügung stehen. In Anpassungsumgebungen sollte dieser Aspekt berücksichtigt werden.

3.7 Flexibilität und Effizienz

Speziell für Benutzer, die erste Erfahrungen mit einem Computersystem bereits gesammelt haben, ist es wichtig, eine möglichst optimale Unterstützung für ihre (häufig wiederkehrenden) Aufgaben zu bekommen. Durch Anpassungen an der Oberfläche können häufig verwendete Funktionen leichter zugänglich gemacht werden. Auch Tastaturkürzel können hier die Effizienz erhöhen. Ein gutes Beispiel hierfür sind die Produkte des Microsoft Office Pakets [Cook et al., 2001]. In diesen Applikationen lassen sich die Menüleisten vollständig rekonfigurieren. Auch der Austausch dieser Anpassungen innerhalb einer Arbeitsgruppe, die ähnliche Aufgaben zu bearbeiten hat, hat sich als sehr nützlich herausgestellt [Kahler et al., 1999].

Ähnliche Gestaltungsmöglichkeiten sollten ebenso für Anpassungsumgebungen gelten. Allerdings lassen sich hier weniger gut bestimmte Operationen, die besonders häufig benutzt werden, ausmachen. Green und Petre [Green und Petre, 1996] untersuchen hier auch visuelle Programmiersprachen. Ihnen geht es vor allem darum zu zeigen, wie kleinere Änderungen an Programmen einfach möglich sind und auch so von der Anpassungsschnittstelle unterstützt werden müssen.

Speziell wenn es um eine für die FREEVOLVE-Plattform entwickelte Anpassungsumgebung geht, sind nur wenige Operationen zu unterstützen. Viel entscheidender ist die Auswahl und Darstellung der verwendeten oder zu verwendenden Komponenten. Hier sollten Konzepte entwickelt werden, die die Bedienung erleichtern.

3.8 Ästhetisches und minimalistisches Design

Ebenso wie das oben angesprochene konsistente Design und eine einheitliche Benutzerführung ist auch die Reduktion der Informationsbereitstellung auf das Wesentliche ein wichtiges Kriterium für die leichte Erlernbarkeit einer Software. Informationen, die die Benutzer verwirren oder von der eigentlichen Arbeitsaufgabe ablenken, erschweren den Zugang zur Software [Pane und Myers, 1996].

3.9 Unterstützung der Benutzer bei der Erkennung und Korrektur von Fehlern

Sowohl während der Nutzung von Software als auch während eines Anpassungsvorgangs kann es zu Bedienungsfehlern kommen. Dieser Problematik kann auf zwei Ebenen entgegengetreten werden: Durch Fehlervermeidung und durch Fehlermanagement (Erkennung und Korrektur von Fehlern) [Frese et al., 1991]. Aktuell steht bei der Entwicklung der meisten Software-Systeme der Aspekt der Fehlervermeidung im Fokus. Die Software wird nach ergonomischen Kriterien so gestaltet, dass sie möglichst fehlerfrei handhabbar ist. Wie Frese et al. [Frese et al., 1991] jedoch weiterhin darstellen, steigt die Anzahl der Fehler, die ein Benutzer macht, mit der Erfahrung im Umgang mit dem System. Dies lässt darauf schließen, dass die explorative Handhabung und damit die bewusste Fehlbedienung Teil der eigentlichen Nutzung ist. Deswegen sollten nicht nur Entwicklungsumgebungen, sondern auch Softwaresysteme im Allgemeinen und in Bezug auf Anpassungsumgebungen im Besonderen über ein integriertes Fehlermanagement verfügen.

Hutchins et al. [Hutchins et al., 1986] untersuchen in diesem Zusammenhang, wie Benutzer üblicherweise die Reaktion eines Computersystems auf ihr Nutzungsverhalten hin evaluieren. Dieser sechsstufige Prozess beginnt mit der Formulierung der Intention, der Spezifikation und der anschließenden Ausführung einer Aktion. Dies führt zu einer Reaktion des Computers. Darauf folgt wiederum beim Benutzer die Interpretation des Wahrgenommenen und eine abschließende Evaluierung. Erst in der letzten Phase kann ein Vergleich zwischen wahrgenommenem Ergebnis und der initialen Intention durchgeführt werden, wobei Fehler jedoch in allen vorhergegangenen Schritten möglich sind.

Ein geeignetes Fehlermanagement sollte also nicht nur für das System erkennbare Fehler markieren, sondern darüber hinaus auch die Benutzer möglichst gut bei der Umsetzung ihrer Intention führen. Eine dreistufige Fehlerbehandlung (entdecken, erklären, beheben) kann dies leisten. Insofern dient das Fehlermanagement zum einen zur Unterstützung bei Korrekturen, zum anderen sollte es in dem Sinne lernförderlich ausgelegt sein, dass die Benutzer aus ihren Fehlern lernen, also sich ein genaueres mentales Modell von der Software entwickeln können.

Hier ist auch der Zusammenhang zwischen der Hürde, Anpassungen am System selbst vorzunehmen, und einem guten Fehlermanagement offensichtlich. Die Führung, die das Fehlermanagement gibt, erleichtert das Anpassen, führt zu größerer Sicherheit und damit zu Bereitschaft zum Anpassen.

Umgekehrt kann gesagt werden, dass ein schlechtes oder nicht vorhandenes Fehlermanagement Benutzer häufig zu Recht davon abhält, Anpassungsoperationen zu nutzen, da – wie Henderson und Kyng [Henderson und Kyng, 1991] darlegen – falsch vorgenommene Anpassungen die Nützlichkeit eines Systems deutlich senken können.

Verschiedene Arbeiten untersuchen, auf welche Weise die Hemmschwelle bei der Nutzung von Software allgemein oder beim Anpassen von Software im speziellen gesenkt werden kann. Sie alle tragen dazu bei, gemachte Fehler leicht korrigieren und erkennen zu können. Paul [Paul, 1994] beschreibt in diesem Zusammenhang verschiedene mögliche Unterstützungsmechanismen, von denen die wichtigeren im Folgenden kurz diskutiert werden:

- **Wiederaufsetzpunkt** (freezing point): Systemzustände können gespeichert werden. Änderungen können dann rückgängig gemacht werden, indem der zuvor gespeicherte Zustand wieder geladen wird. Die Nutzer müssen sich also vor Ausführung einer „kritischen“ Operation darüber bewusst sein und den aktuellen Systemstatus zur späteren Wiederherstellung speichern.

- **Stornierung**: Die Stornierung setzt voraus, dass Aktionen aufgezeichnet und invertiert ausgeführt werden können. Auf diese Weise lassen sich durchgeführte Operationen zurücksetzen. Anders als beim Wiederaufsetzpunkt müssen die Benutzer sich nicht im Vorfeld für die Nutzung der Funktion entscheiden. Diese Funktion wird häufig als „Undo“ bezeichnet.
- **Neutralmodus**: Er ermöglicht die Selbsterklärung von Systemfunktionen. Aufgerufene Funktionen werden nicht durchgeführt, sondern ihre Wirkung wird erläutert.
- **Szenarium-Maschine/Explorationsmodus**: Eine künstliche Systemumgebung erlaubt es, mit exemplarischen Daten Anwendungsszenarien durchzuspielen. Besonders geeignete Daten erleichtern dabei die Exploration. Wie schon oben gezeigt, lassen sich einige der Techniken zum Explorieren sowohl von Funktionalität als auch von Anpassungsmöglichkeiten nicht beliebig in gruppenbezogene Arbeitskontexte übertragen. Um dennoch exploratives Lernen von anpassbarer Groupware zu unterstützen, wurde das Konzept der Explorationsumgebungen entwickelt [Wulf, 2000], [Wulf und Golombek, 2001]. In Explorationsumgebungen wird das Systemverhalten an der Benutzungsschnittstelle des Aktivators in den für das Erlernen relevanten Aspekten simuliert. So wird es möglich, die Benutzungsschnittstelle anderer Nutzer anstelle der des Aktivators darzustellen. Durch Wechseln der Perspektiven zwischen verschiedenen möglichen Nutzern und die Aktivierung anderer in der Explorationsumgebung implementierter Funktionen kann der Aktivator die Effekte der Ausführung der zu explorierenden Funktion auf die Schnittstelle anderer Nutzer erkunden.

Alle diese Mechanismen bauen auf das weitgehend selbständige Erlernen von Anpassungssprachen. Das Computersystem liefert eine Unterstützung, die auf der einen Seite erleichtert, mit Hilfe von Dokumentationen oder Beispielen den Umgang sowohl mit den Anpassungsoperationen als auch mit den Anpassungsartefakten (in unserem Kontext Komponenten) umzugehen. Auf der anderen Seite ermöglichen sie eine Kontrolle des angepassten Artefakts mit dem mentalen Modell. Auf Basis von Exploration lässt sich spielerisch die Funktionstüchtigkeit und Funktionalität prüfen und mit der intendierten abgleichen. Hier findet also eine Prüfung statt, die weitgehend ohne technische Unterstützung vonstatten geht und lediglich dadurch gestützt wird, dass Beschreibungen oder Testumgebungen zur Verfügung gestellt werden. Einen Schritt weiter soll das Konzept der semantischen Integritätsprüfung führen, das Thema dieser Arbeit ist.

Ein weiterer Schwachpunkt der Explorationstechnik ist die sehr informelle Beschreibung der Funktionen, die vieles sowohl in der Funktionsbeschreibung als auch in der technischen Realisierung unbeantwortet lässt: Beispielsweise stellt sich fast zwangsläufig die Frage, wie sinnvolle Explorationsdaten generiert werden können. Werden diese von den Entwicklern der Software mitgeliefert? Sollten diese in Abhängigkeit von der Anwendungsdomäne definiert werden? Eignen sich bestimmte Daten besser, kritische Momente im angepassten Artefakt auszumachen? Auch hier kann eine Integritätsprüfung helfen, eben diese Fragen zu beantworten.

3.10 Hilfe und Dokumentation

Mackay [Mackay, 1990] zeigt, dass das Fehlen einer Dokumentation die Nutzung bzw. das Anpassen verhindert. Der Umkehrschluss, dass gute Dokumentationen förderlich sind für Nutzung bestimmter Techniken, lässt sich ebenso herleiten. Diese grundsätzlichen Gedanken gelten selbstverständlich ebenso für Anpassungsumgebungen. Hier kommt noch hinzu, dass Anpassungen im Verhältnis zur allgemeinen Nutzung einer Applikation seltener vorgenommen werden. Der Bedarf an Unterstützung durch das System ist also größer.

Ein Beispiel für Hilfestellungen können mit Interaktionshistorien aufgezeichnete Animationen von Arbeitsschritten sein. Mit ihnen können leicht Anleitungen für die Nutzung bestimmter Funktionen erzeugt und bereitgestellt werden. (vgl. [Paul, 1994]). Zudem können Beispiele und Erklärungen, die andere Nutzer zusammengestellt haben, wichtige „Anreize" zur Anpassung sein. Hier können sowohl einfache Texte helfen als auch die oben angesprochenen animierten Anleitungen. Die Möglichkeit, dass Dokumentationsmöglichkeiten als Teil der Systemfunktionalität angeboten werden, ist in vielen Entwicklungswerkzeugen integriert. Hier ist es sowohl möglich, den entwickelten Code als auch Teile der Programmierumgebung zu kommentieren. Diese Funktionalität kann auch dauerhaft während der Benutzung der Software oder deren Anpassungsfunktionalität zur Verfügung stehen. Dies scheint vor allem vor dem Hintergrund sinnvoll, dass so Erklärungen von den Benutzern selbst erstellt und erweitert werden, die dann im konkreten Arbeitsbezug stehen. In Nutzungstests [Engelskirchen, 2000] zeigte sich, dass auch die ersten Anpassungsumgebungen der FREEVOLVE-Plattform optimierungswürdig waren, wenn es darum ging, Nutzern den Einstieg zu erleichtern. Aus diesem Grunde erweiterte Engelskirchen eine Anpassungsumgebung um Beschreibungen (siehe auch [Carroll und Carrithers, 1984]) der Anpassungsoperanden und -operationen, einen Neutralmodus und eine Explorationsumgebung. Darüber hinaus wurde kooperatives Lernen in der Form ermöglicht, dass Benutzer eigene Anmerkungen oder auch Fragen zu den jeweiligen Anpassungsartefakten formulieren und anhängen konnten [Wulf, 1999]. Eine sich daran anschließende qualitative Evaluierung zeigte deutlich verbesserte Ergebnisse. Parallel dazu entwickelten Wulf und Golombek [Wulf und Golombek, 2001] eine Explorationsumgebung für ein Filtersystem für den Austausch von Dokumentformatvorlagen innerhalb einer Gruppe. Eine sehr detaillierte quantitative Analyse zeigte auf, wie wertvoll diese Möglichkeit der Exploration für das leichtere Erlernen eines Software-Systems ist.

3.11 Fazit

Es wurde gezeigt, dass die Nutzung einer Software durch die Beachtung diverser Gestaltungsrichtlinien und auch durch verschiedene Unterstützungskonzepte erleichtert werden kann. Diese allgemeinen Gestaltungsanforderungen gelten für die Nutzbarkeit von Software ebenso wie für die Bedienbarkeit und die Erlernbarkeit von Anpassungsumgebungen.

Der Wunsch nach möglichst flexibler und von Endbenutzern selbst dem Arbeitskontext anpassbarer Software führt hier zur Entwicklung komplexer Anpassungssprachen. Diese Komplexität lässt sich durch die oben beschriebene Nutzung von hierarchischen Komponentenarchitekturen reduzieren. Mit ihnen lassen sich recht einfach hochgradig anpassbare Software-Systeme entwickeln. Dennoch bleiben Anpassungen an Systemen für die meisten Nutzer komplizierte und deswegen ungern durchgeführte Aufgaben.

Unterstützungskonzepte können die Nutzbarkeit von Anpassungsumgebungen weiter verbessern. Die in Abschnitt 3.9 beschriebenen Unterstützungstechniken haben bisher allerdings auf konzeptioneller Ebene verschiedene Schwächen:

- **Keine aktive Nutzerunterstützung**: Alle hier vorgestellten Unterstützungstechniken stellen die Endbenutzer bei der Fehlersuche oder bei Optimierungsmöglichkeiten während des Anpassungsvorgangs klar in den Mittelpunkt (z. B. Explorationsmodus zur Fehlersuche, Dokumentationen für die richtige Verwendung von Komponenten). Hemmschwellen beim Anpassen lassen sich jedoch vor allem dadurch abbauen, dass Fehlerquellen eingeschränkt werden bzw. Fehler leicht zu finden und zu korrigieren sind. Hier kann ein aktiv führendes Anpassungssystem eine wertvolle Hilfe sein.

- **Technische Perspektive auf Unterstützungskonzepte bleibt unbeleuchtet**: Alle hier vorgestellten Ansätze (Dokumentationen, kontextsensitive Hilfen, Explorationsmodi) sind bisher vordringlich von der Anwendungsseite aus betrachtet worden. Wie aber lassen sich effizient und sinnvoll Komponenten oder Komponenten-Sets anwendungsspezifisch um Annotationen (Erläuterungstexte) anreichern? Und wie können diese einfach kontextsensitiv angezeigt werden? Eine solide technische Basis ist hier nötig, um Entwicklern solcher anpassbarer Softwaresysteme ein geeignetes Handwerkszeug zur Verfügung zu stellen.
  Auch die Idee, Explorationsumgebungen zu entwickeln, wurde diskutiert. Hier kommen „Spieldaten" zu Einsatz, mit deren Hilfe sich in einem vereinfachten Abbild der realen Systemumgebung die Funktionalität testen lässt. Die Güte dieses Unterstützungsmechanismus ist maßgeblich abhängig von der Wahl geeigneter Spieldaten. Bisher wird nicht dargestellt, welche Spieldaten sich für welche Testzwecke eignen und wie diese zu kombinieren sind.
- **Keine Berücksichtigung der veränderten Möglichkeiten durch Einsatz von Komponentenarchitekturen**: Alle vorgestellten Techniken gehen implizit von monolithischen Architekturen aus oder nehmen auf diese technische Basis nur wenig Bezug. Legt man als Basis für eine hochgradig anpassbare Software Komponentenarchitekturen zugrunde (siehe z. B. [Engelskirchen, 2000]), so lassen sich verschiedene dieser technischen Basis und der damit einhergehende Mikrosicht auf die Applikation inhärent gegebenen Eigenschaften nutzen. Beispielsweise lässt die Zerlegung einer Applikation in voneinander weitestgehend unabhängige Komponenten zu, dass Beschreibungen für jede Einzelkomponente erstellt werden. Diese kann ebenso unabhängig vom Rest des Komponenten-Sets konzipiert werden. Engelskirchen führt Beispiele als Erläuterungen zur Funktionalität einer Komponenten ein [Engelskirchen, 2000]. Auch hier liegt dann der Schwerpunkt wieder auf der Betrachtung des gesamten Komponenten-Sets auf der Ausführungsebene.
  Mit Hilfe der stärkeren Ausrichtung an den Einzelkomponenten kann die Beschreibung deutlich vereinfacht werden. Nicht notwendige Abhängigkeiten werden eliminiert.

Eine Lösung für die hier angesprochenen Probleme bieten Integritätsprüfungssysteme. Sie basieren auf der Idee, dass allen Einzel-Komponenten zusätzliche (semantische) Informationen bzgl. ihrer Nutzungsmöglichkeiten beigefügt sind. Diese Informationen können in mehrerlei Hinsicht genutzt werden. So stellen sie selbst eine erweiterte Dokumentation dar, die mit Hilfe von automatischen Textkompositionen auch für Endanwender hilfreich ist. Eine Beschreibung der Integritätsbedingungen sollte ebenso wie das zugrunde liegende Komponentenmodell eine hierarchische Konstruktion erlauben. Auf diese Weise können für Einzelkomponenten zur Verfügung gestellte semantische Informationen im Falle einer Komposition zu einer Information für die resultierende abstrakte Komponente aggregiert werden.

Weiterhin können den Komponenten Bedingungen für ihre Nutzung und auch Korrekturvorschläge beigefügt werden. Eine auf diesen Bedingungen basierende Prüfung einer Komposition und der in ihr enthaltenen Einzelkomponenten kann genutzt werden, um Fehler aufzuzeigen, Verbesserungsvorschläge zu unterbreiten oder sogar automatisch Korrekturen vorzunehmen.

Schließlich können diese Informationen genutzt werden, um effektiver „kritische" Spieldaten für Explorationsumgebungen zu erzeugen. Schließlich können sie dazu dienen, direkt mit Hilfe eines Integritätsprüfungssystems analysiert zu werden. Auf diese Weise lässt sich eine Komposition auf semantische Korrektheit im Sinne der vorhandenen Integritätsbedingungen und -strategien hin prüfen.

4 Integrität in Informationssystemen

Wie schon oben angesprochen scheint eine Unterstützung auf Basis von Integritätsprüfungsmechanismen bei der Komposition von Anwendungen sinnvoll. In verschiedenen Bereichen der Informatik spielt eine teilweise automatische Integritätsprüfung von Informationssystemen eine wichtige Rolle. Dieses Kapitel wird die dort vorhandenen Konzepte beschreiben und in den Kontext der Komponentenarchitekturen setzen.

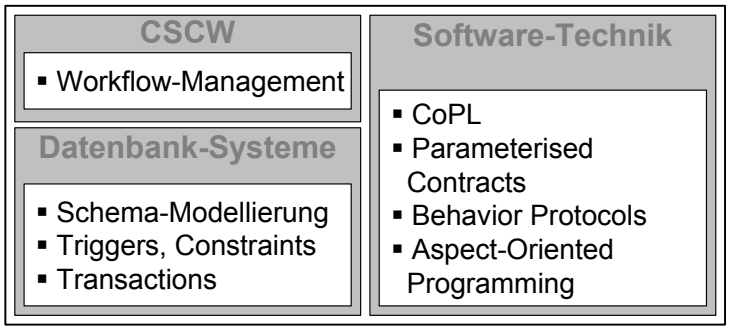


Abbildung 9:Der Integritätsbegriff in verschiedenen Bereichen der Informatik

Zu Beginn werden Workflow-Management-Systeme (WFMS) eingeführt. Sie unterstützen die Gestaltung und Steuerung halbautomatisierter Arbeitsabläufe. Hier werden während der Konzeptionsphase häufig Integritätsprüfungsmechanismen eingesetzt. Weiterhin werden Datenbank-Management-Systeme untersucht. Sie nutzen sowohl in der Konzeptionsphase als auch während der Nutzung Integritätsprüfungen, um die Konsistenz der Datenbestände zu gewährleisten. Schließlich beschäftigt sich auch der Bereich der Software-Technologie seit langem damit, wie sich die Qualität von Software durch automatische Prüfungsmechanismen, die über reine Syntax-Prüfungen hinausgehen, verbessern lässt. Hier wird der Bereich der vertragsbasierten Entwicklung[14] diskutiert. Auch neuere Konzepte, die sich dediziert mit Komponentenarchitekturen beschäftigen, werden angesprochen.

Eine Diskussion darüber, wie sich die Konzepte in den Kontext der komponentenbasierten Anpassbarkeit integrieren lassen, schließt jeden Abschnitt. Für eine Integritätsprüfung, die Benutzer bei Anpassungsvorgängen unterstützen sollen, müssen die Anpassungsoperationen im Zentrum der Überlegungen stehen, wie sich ein Integritätskonzept zusammensetzen sollte. Weiterhin ist auch zu beachten, dass die einzelnen Integritätskonzepte im Sinne der Lernförderlichkeit verständlich sein müssen.

4.1 Validitätsprüfung in Workflow-Management-Systemen und im Bereich des VLSI-Designs

Workflow-Management-Systeme lassen sich den Groupware-Systemen zuordnen. Ihre spezielle Aufgabe besteht darin, strukturierte Arbeitsabläufe, in denen menschliche Akteure zusammenarbeiten, zu unterstützen [Jablonski et al., 1997]. Allerdings beschäftigen sich auch andere Teilbereiche der Informatik wie die Forschungsgebiete um Datenbanksysteme

[14] Engl: Design By Contract

mit der Unterstützung von Workflows [Silberschatz et al., 2001]. Sie betrachten hier vordergründig die gemeinsame Informationsbasis, die durch verschiedene Schnittstellen (menschliche und technische) strukturiert verändert wird. Beide Perspektiven des Workflow-Management sehen jedoch die strukturierte Bearbeitung eines mehrstufigen Vorgangs als zentral an.

Ein Beispiel für einen Workflow können Antrags- oder Genehmigungsvorgänge in öffentlichen Verwaltungen sein. Sie laufen mit wenigen Ausnahmen nach immer gleichen Mustern ab: Ein Antrag wird eingereicht (*Empfang*), wird nach der Zuständigkeit geordnet und einem Sachbearbeiter zugewiesen (*Verzweigung*). Dieser bearbeitet den Antrag, wobei evtl. Rücksprachen mit anderen Dienststellen (*Sub-Workflow*) nötig werden. Schließlich muss der bearbeitete Antrag vom Vorgesetzten unterzeichnet und innerhalb einer gesetzten Bearbeitungsfrist an den Antragsteller zurückgeschickt werden.

Dieses recht einfache Beispiel zeigt die zentralen Aspekte bei der Modellierung von Workflows. Neben den in Klammern eingeführten Begriffen, sind weitere Basiskonzepte notwendig, um einen Geschäftsprozess zu modellieren:

- **Rollen**: Mitglieder einer Organisation erhalten Rollen oder werden Gruppen zugeordnet. Auf diese Weise lassen sich beispielsweise Aufgaben und auch Rechte innerhalb eines WFMS modellieren.
- **Aufgaben**: Sie werden an jeder Station von den Rollenträgern bearbeitet. Dabei ist für das WFMS, das die Güte der Bearbeitung nicht prüfen kann und soll, lediglich von Bedeutung, inwieweit die Aufgabe eine Bestätigung seitens des Benutzers erfordert. Beispielsweise kann man die bloße Kenntnisnahme auf der einen Seite von der signierten Zustimmung scharf trennen.
- **Bedingte Verzweigungen**: Sie werden in Abhängigkeit von Ereignissen, wie zum Beispiel eine positive oder negative Aufgabenbestätigung, bearbeitet.
- **Parallele Teilprozesse**: Auf Basis von Verzweigungen können Workflows aufgespaltet werden. In solchen Fällen können Teilaufgaben parallel bearbeitet werden. Beide Sub-Workflows fließen anschließend wieder zusammen.

Da die meisten der für den Arbeitsvorgang benötigten Informationen sowieso schon elektronisch zumeist in Datenbanken vorliegen, bietet es sich an, die Arbeitsflüsse ebenso durch IT zu unterstützen. Aus diesem Grund betrachten Silberschatz et al. [Silberschatz et al., 2001] Workflow-Management auch als eine auf Datenbanksystemen aufsetzende Integrationstechnik. Diese Integration kann zentral durch einen Scheduler koordiniert werden. Den Gegenpol dazu stellt die vollständig distributierte Bearbeitung eines Workflows dar. Dies wird beispielsweise bei Email-basierten Workflow-Systemen realisiert. Eine einzelne Station ist hier vollständig sowohl für die Bearbeitung der Teilaufgabe als auch für das Auffinden der nächsten Stationen des Workflows zuständig.

Die Modellierung eines Workflows beginnt üblicherweise mit einer intensiven Analyse des vorhandenen Geschäftsprozesses. Anschließend wird nach Optimierungsmöglichkeiten gesucht. Die systematische Beschreibung eines Workflows, der dann vom WFMS bearbeitet werden kann, wird häufig von Prozessmodellierungseditoren unterstützt. Häufig wird die Arbeit darüber hinaus durch Verifikationssysteme erleichtert.

In WFMS gibt es diverse Verfahren, Integrität zu prüfen. Nach [van der Aalst und van Hee, 2002] hat die Soundness eines Workflows oberste Priorität. Mit Soundness sind grundsätzliche Eigenschaften eines Workflows gemeint, die sinnvollerweise erwartet werden müssen. Dazu gehören beispielsweise die Endlichkeit und die Deadlock-Freiheit.

Ein solches automatisches Verifikationssystem stellen van der Aalst et al. [van der Aalst et al., 1997] mit Woflan vor. Hier werden Workflows als gerichtete Graphen betrachtet und mit Hilfe von Petri-Netzen untersucht. Woflan kann neben Soundness noch weitere Integritätsbedingungen von Workflows testen. Dazu gehören auch rein syntaktische Prüfungen, wie das Auffinden von Aufgaben ohne Eingabe- oder Ausgabebedingungen. Petri-Netze eignen sich aus diversen Gründen sehr gut zur Modellierung von Workflows. Wie in [van der Aalst und van Hee, 2002] genauer beschrieben wird, lassen sich auf diese Weise Tests mit Standardmethoden durchführen. Auch die Darstellung des Arbeitsprozesses als Graph erleichtert das Verständnis und hilft, vom System angemerkte Fehler zu verstehen.

Auch im VLSI-Design werden Verfahren zur Analyse angewendet, die auf Petri-Netzen basieren [Yakovlev et al., 1996], [Shapiro, 1991]. Im Gegensatz zu den Workflows im Bereich der Groupware sind hier alle Stellen innerhalb des Netzes vollständig formalisierbar. Ein Problem hier ergibt sich eher in der Komplexität, die sich bei der vollständigen Abbildung ergeben würde. Auch hier muss man aus diesem Grund ein geeignetes Abstraktionsniveau einnehmen. Durch hierarchische Konstruktion der gefärbten Petri-Netze soll weiterhin eine [Shapiro, 1991] geschichtete Validierung ermöglicht werden.

Petri-Netz-basierte Analyse in komponentenbasierten Architekturen

Aufgrund der nicht vollständig vorhersehbaren „menschlichen Faktoren“ kann mit Hilfe der graphenbasierten Analyse, wie sie in WFMS betrieben wird, die semantische Korrektheit eines Arbeitsablaufs nur bedingt überprüft werden. Für die Integritätsprüfung von Komponentennetzen bietet sie sich trotzdem an. Hier sind Funktionsweisen der einzelnen Stellen (Teilaufgaben) durch Komponenten deterministisch vorgegeben. Werden die für eine semantische Prüfung relevanten Informationen bereitgestellt, können die Kommunikationsflüsse zwischen den Komponenten mit ähnlichen Mechanismen, wie sie bei der Workflow-Analyse zum Einsatz kommen, untersucht werden.

Syntaktische Analysen, die zumeist von Compilern vorgenommen werden, prüfen lediglich die korrekte Verbindung zweier Komponenten miteinander. Dabei können nur falsch miteinander verbundene Komponenten entdeckt werden. Nicht genutzte Schnittstellen können so nicht als Fehler erkannt werden, obwohl dies unter semantischen Aspekten einer Komposition von großer Wichtigkeit sein kann. Weiterhin wird bisher immer nur eine Komponente und ihre Verbindung mit anderen betrachtet. Erst die Betrachtung der Komposition als Graphen erlaubt eine tiefgehende Analyse der vollständigen Kommunikationsflüsse. Diese Beobachtung macht auch Heinlein [Heinlein, 2000] der die oben angesprochenen Workflows ebenfalls als Graphen betrachtet und sie mit Hilfe von Interaktionsausdrücken und -graphen analysiert.

Bei allen Workflow-Analyse-Ansätzen wird die Dynamik eines Arbeitsablaufes untersucht. Hier spielen Nebenläufigkeiten eine wichtige Rolle bei der Frage, wann eine Station im Workflow mit der Arbeit beginnen kann. Diese dynamischen Ansätze lassen sich auch mit Hilfe von Petri-Netz-basierten Analysen nicht ohne weiteres auf Komponentennetze übertragen. Es fehlen hier Informationen darüber, wann und in welcher Reihenfolge Kommunikationsschnittstellen angesprochen werden. Dies ist häufig auch abhängig von den unterschiedlichen Situationen, die sich bei der Nutzung (Benutzerinteraktion) ergeben können. Ein Weg, dieses Problem zu lösen, wird weiter unten (siehe Abschnitt 4.3) diskutiert, wenn es um die Darstellung von Behavior-Protokollen [Plásil et al., 2000] geht.

4.2 Datenbank-Integrität

Datenbanken dienen der Speicherung und schnellen Bereitstellung großer Informationsmengen; sie werden vielfältig eingesetzt. Eine wichtige Anforderung an die Daten ist dabei ihre Konsistenz und die Integrität des Datenbestandes. Prüfungstechniken existieren in diesem Kontext schon seit langer Zeit. Während zur Designzeit Schema-Transformationen helfen, sinnvolle (redundanzfreie) Datenmodelle zu entwerfen [Kemper und Eickler, 1997], werden zur Laufzeit mit Hilfe von Transaktionsmechanismen [Silberschatz et al., 2001] und Constraints oder Triggers [ANSI, 1996] Änderungen der Datenbasis überwacht.

Während der Modellierung der späteren Datenbank werden verschiedene Techniken eingesetzt, um die Konsistenz des späteren Datenbestands zu gewährleisten. Eine der wichtigsten Arbeiten ist hier die Normalisierung, die Redundanzen verhindern soll. Diese Redundazen führen vor allem bei Änderungen (Einfügen, Verändern oder Löschen von Datensätzen) zu Problemen [Kemper und Eickler, 1997].

Zur Laufzeit werden verschiedene Techniken genutzt, um die Konsistenz des Datenbestands und dessen semantische Korrektheit zu erhalten. Transaktionsmechanismen [Silberschatz et al., 2001] spielen hier eine wichtige Rolle. Sie sorgen dafür, dass zusammengehörige komplexe Änderungen an der Datenbasis atomar, d.h. vollständig oder gar nicht, durchgeführt werden können. Dies können beispielsweise Änderungen sein, die an mehreren Tabellen gleichzeitig durchgeführt werden sollen und nur dann sinnvoll sind, wenn alle durchgeführt werden können. Für Fälle, in denen dies nicht möglich ist, können Rollback-Mechanismen die Teilaktionen einer Transaktion rückgängig machen, so dass der Zustand der Datenbasis vor Beginn der Transaktion wiederhergestellt ist.

Schließlich bieten speziell aktive Datenbanksysteme verschiedene Mechanismen an, um Änderungen an der Datenbasis an Nebenbedingungen und sogar damit verknüpfte Operationen binden zu können. Stonebraker [Stonebraker, 1975] und Bernstein [Bernstein et al., 1980] stellen Konzepte dar, mit denen sich Integritätsbedingungen an die Datenbasis formulieren lassen. Sie können dann dazu genutzt werden, um Teile der für ein Informationssystem relevanten Business-Logik im Datenbanksystem umzusetzen. So können solche Änderungen an der Datenbasis verhindert werden, die mit dem Schema durchaus vereinbar sind, aber der Semantik des Informationssystems widersprechen.

Schon frühzeitig wurde das Konzept von Events, Conditions und Actions (ECA-Regeln, siehe z. B. [Griefahn, 1997]) beschrieben. Auch SQL bietet neben Constraints [ANSI, 1992] auch Triggers zur Forcierung der Einhaltung von Integritätsbedingungen. Beide Konzepte sind sich aus der hier eingenommenen Perspektive recht ähnlich, in der Form, dass Bedingungen an die Datenbasis gestellt werden. Diese werden bei Änderungen überprüft. Triggers sehen außerdem auch vor, Aktionen bei Nichteinhaltung bestimmter Bedingungen durchzuführen. Gertz und Lipeck [Gertz und Lipeck, 1996] zeigen, dass sich beide Konzepte ineinander überführen lassen.

Auch in objekt-orienterten Datenbank-Management-Systemen (OODBMS), in denen die enthaltenen Objekte eigene Methoden mitführen, sind solche „externen“ Kontroll- und Modifikationsmöglichkeiten sinnvoll [Heuer, 1992]. Wie auch in der hier vorliegenden Arbeit sind die Integritätsbedingungen in OODBMS deswegen häufig extern formuliert.

Datenbank-Schemata und Komponentenarchitekturen

Eine Möglichkeit, einen Vergleich zwischen der Datenbank-Welt und der der komponentenbasierten Architekturen zu ziehen, ist, das Datenbankschema und das für eine Komposition zur Verfügung stehende Komponenten-Set gegenüberzustellen. Beide stellen nur einen Rahmen dar. Allerdings kann eine gute Schema-Modellierung helfen, die Integrität des Datenbestands zu sichern. Ebenso spielt die richtige Entwicklung bzw. Auswahl von Kompo-

nenten, die dann die Basis für mögliche Kompositionen darstellen, eine wichtige Rolle für die Güte und Funktionsweise der resultierenden Applikationen. Hier ist neben der impliziten Funktionsweise der Komponenten natürlich ebenso entscheidend, welche Schnittstellen sie haben und wie sie sich parameterisieren lassen. Dies führt zu der Frage, inwieweit eine geeignete Zerlegung einer Applikation in Einzelkomponenten oder auch die eigenständige Entwicklung von Komponenten und Auswahl für ein domänen-spezifischen Komponenten-Set Einfluss auf die semantische Integrität späterer Applikationen haben. Sie ist jedoch nicht Teil dieser Arbeit und wird dementsprechend nicht weiter verfolgt.

Constraints zur Sicherung von Eigenschaften einer Komponente oder einer Komposition

In (aktiven) Datenbanksystemen werden häufig zusätzliche Bedingungen an die Datenbasis beschrieben. Sie können – wie oben beschrieben – bei Nichterfüllung automatisch durchzuführende Aktionen nach sich ziehen. Zu jeder Komponente lassen sich extern formulierte Nebenbedingungen festlegen. Beispielsweise kann eine Forderung sein, dass die Komponente mit mindestens einer weiteren Komponente verbunden sein muss (also nicht isoliert in einer Komposition steht). Weiterhin können Bedingungen auch an einen bestimmten (vor der Kompositionsaktivität auszuwählenden) Applikationstypus gekoppelt sein. Solche Bedingungen sind dann nicht nur bezogen auf eine einzelne Komponente, sondern treffen Aussagen über eine Komponentengruppe. Sie können sich ebenso auf die Existenz einzelner Komponenten(typen), Verbindungen zwischen ihnen oder Parameter beziehen.

Es können auch Aktionen an diese Bedingungen geknüpft sein. So können beispielsweise aufwendige Unterstützungsfunktionen angeboten werden (textuelle Hilfen, Starten eines Agenten, der den Fehler visualisiert und gleichzeitig kontextabhängig Verbesserungsvorschläge unterbreitet) oder die Integrität automatisch wiederhergestellt werden, sei es durch ein einfaches Rollback oder automatische Korrekturen. Welcher Ansatz hier sinnvoll und machbar ist, wird in den folgenden Kapiteln thematisiert.

Als Transaktionen zusammengefasste Anpassungsoperationen

Zur Laufzeit einer Datenbank erhalten vor allem Transaktionskontrollen [Silberschatz et al., 2001] die Integrität der Datenbasis. Sie sichern bei komplexeren Änderungsoperationen zu, dass Änderungen atomar, d.h. vollständig oder gar nicht, durchgeführt werden. Hier gilt also die ACID-Eigenschaften, nach denen die Transaktion atomar, konsistent, isoliert und dauerhaft durchgeführt wird.

Ähnlich können auch die Änderungsoperationen beim Anpassungsvorgang zusammengefasst werden, die dann als eine gemeinsame Anpassungs-Transaktion betrachtet wird. Eine solche Änderung wird nur vollständig oder gar nicht durchgeführt. Dies ist von großer Bedeutung, wenn die Laufzeitumgebung nur integere Komposition im Sinne der vorher festgelegten Regeln erlaubt. Während des Anpassungsvorgangs kann es hier häufiger zu Integritätsverletzungen kommen. Mit einer Zusammenfassung des Anpassungsvorgangs zu einer atomaren Aktion solche Zustände vermieden werden.[15] Auf diese Weise können Anpassungen, die in einer Mehrbenutzerapplikation durchgeführt werden, leichter kontrolliert werden, und inkonsistente Zwischenzustände werden vermieden. Dies ist also speziell für Groupware-Applikationen wichtig. Wenn also Anpassungen zur Laufzeit des Systems durchgeführt werden, sollten inkonsistente Zustände bzgl. der Komposition vermieden werden. Auch hier kann eine Transaktionskontrolle hilfreich sein. Sie stellt sicher, dass ein Anpassungsvorgang immer als Transaktion atomar durchgeführt und so die Applikation von einem konsistenten (integeren) Zustand in den nächsten überführt wird.

[15] Die Integritätsprüfung kann und sollte jedoch trotzdem während des Anpassungsvorgangs aktiv bleiben. Sie kann wie oben angesprochen Hilfestellungen geben oder Korrekturen vornehmen.

4.3 Software-Technik

In der jüngeren Vergangenheit haben sich verschiedene Forschungsgruppen damit beschäftigt, die Qualität von Software dadurch zu erhöhen, dass Zusatzinformationen, die einzelnen Modulen beigefügt wurden und die Aussagen über Verhalten oder Schnittstellen machen, von Compilern ausgewertet wurden. Die hier beschriebenen Ansätze sind vor allem im Bereich der objekt-orientierten Programmierung entwickelt worden. Insofern sind sie in den meisten Fällen ohne größere Schwierigkeiten auf den Ansatz der komponentenbasierten Architekturen übertragbar. Sie lassen sich als semantische Nebenbedingungen, die beim Einsatz eines Moduls gelten müssen, begreifen.

4.3.1 Verträge zwischen Komponenten

Die von Hoare [Hoare, 1972] erstmals erwähnte vertragsbasierte Software-Entwicklung (Design by Contract, siehe auch [Meyer, 1988]) knüpft Bedingungen, an die Ausführung von Methoden. Sie müssen erfüllt sein, damit die Methode aufgerufen werden kann. Diese Vorbedingung wird ergänzt um eine Nachbedingung, die das Objekt der aufgerufenen Methode zusichert. Invarianten beschreiben Eigenschaften, die eine Klasse grundsätzlich immer erfüllt. Während einer Methodenausführung kann eine Invariante verletzt werden. Das aufrufende Objekt und das aufgerufene werden so zu Vertragspartnern. Die Vor- und Nachbedingungen und auch die Invarianten gelten dann als Gegenstand des Vertrags. Hier wird also die Typsicherheit als Prüfung, ob ein Methodenaufruf zulässig ist, erweitert um statisch festgelegte Nebenbedingungen, die zum Zeitpunkt des Aufrufs gelten müssen.

Praktisch umgesetzt wurde dieses Konzept beispielsweise bei der objekt-orientierten Programmiersprache Eiffel [Eiffel, 2002], [Meyer, 1992]. Hier werden die Vor-, Nachbedingungen und Invarianten im Source-Code deklariert. Eine Überprüfung auf Einhaltung findet während der Laufzeit statt. Vertragsbasierte Entwicklung ist eine Möglichkeit, um die Zuverlässigkeit von Software zu erhöhen, sie dient der Softwarespezifikation und der Dokumentation.

Auch für Java gibt es verschiedene Werkzeuge wie iContract [Kramer, 1998], die die vertragsbasierte Entwicklung unterstützen. Allen ist gemein, dass auf Basis eines Mehraufwands bei der Programmierung die anschließende Testphase und die Suche von Fehlern erleichtert wird. Werkzeuge wie JUnit [*Griffiths*, 2001] wurden zum Test von einzelnen Programmmodulen entwickelt. Mit ihnen lassen sich ebenso Vor- und Nachbedingungen innerhalb eines Programmablaufs prüfen. Sane und Harris [Sane und Harris, 2003] diskutieren in diesem Zusammenhang, inwieweit die beim Design by Contract definierten Zusicherungen sich mit externen Tests vergleichen und überprüfen lassen.

Erweiterung der Schnittstellenbeschreibungen um geforderte Vor- und garantierte Nachbedingungen

Die Schnittstellen von Komponenten lassen sich grundsätzlich ebenso wie die von Objekten um zusätzliche Informationen erweitern. Daher ist das Konzept der vertragsbasierten Entwicklung grundsätzlich übertragbar. Die Beschreibungen der Vor- und Nachbedingungen können helfen, die Funktionalität einer Schnittstelle zu begreifen. Ein Anpassungswerkzeug kann zudem die Bedingungen analysieren und in eine für Endanwender verständlichere Form übersetzen und mit Erläuterungen anreichern. Auch kann die auf diese Weise gegenüber der einfachen Typprüfung weiter eingeschränkte Möglichkeit, Komponenten miteinander zu verbinden, graphisch in Konstruktionswerkzeugen anschaulich visualisiert werden.

Eine große Schwäche ergibt sich aus der Tatsache, dass die Prüfung auf Einhaltung der Vor- und Nachbedingungen erst zur Laufzeit möglich ist. Damit kann zwar die Integrität der laufenden Komposition gesichert werden, jedoch steht in dieser Arbeit die Unterstützung der Benutzer während des Anpassungsvorgangs im Vordergrund. Insofern sollte dieser Ansatz – da er sich nicht statisch realisieren lässt – nicht in das im folgenden zu entwickelnde Integritätskonzept einfließen. Auch mit Hilfe von Simulationen durch Testdaten ist dies nur bedingt möglich, da viele Zustände abhängig sind von Benutzereingaben, die nur bedingt vorhersehbar sind.

4.3.2 Beschreibung des Nutzungsverhaltens

Auch die Behavior-Protokolle, die von Pospísil [Pospísil, 1999] und Plásil [Plásil et al., 2000] vorgestellt werden, erweitern die Schnittstellenbeschreibungen um eine Nutzungssemantik. In diesem Kontext ist unter der Schnittstelle einer Komponente das gesamte bereitgestellte Interface (Verbund von Schnittstellen) zu verstehen. Die Nutzungssemantik wird in Form von Protokollen beschrieben. Hier werden bestimmte sinnvolle Verwendungsmöglichkeiten der Schnittstelle definiert. Dabei geht es vorrangig um die Reihenfolge, in der die einzelnen Methoden angesprochen werden. Die Beschreibung eines solchen Protokolls wird als regulärer Ausdruck beschrieben. Ebenso existiert auch zu der Komponente, die eben diese Schnittstelle nutzen möchte, ein Protokoll, das beschreibt, wie die Schnittstelle angesprochen werden soll. Bei einer Verbindung zwischen zwei Komponenten muss also geprüft werden, ob die beiden Verhaltensprotokolle zueinander passen.

Diese Art der Beschreibung und Prüfung der genutzten bzw. angebotenen Schnittstellen schränkt auf der einen Seite die möglichen Verbindungen zwischen Komponenten ein, auf der anderen Seite verhindert sie gerade bei der Verwendung komplexerer Komponenten Fehler, die dadurch auftreten, dass Komponentenschnittstellen in nicht vorgesehener Art angesprochen werden.

Im Gegensatz zur vertragsbasierten Entwicklung geht es hier nicht darum, welche Bedingungen an den Aufruf einer Methode geknüpft sind. Stattdessen steht das Zusammenspiel mehrerer Methoden im Vordergrund. Beispielsweise könnte eine Datenbank-Komponente als Protokoll vorsehen, dass zu Beginn die Methode open(...) aufzurufen ist, anschließend beliebige Anfrage-Methoden; die Kommunikation muss mit dem Aufruf der close()-Methode abgeschlossen werden. Man kann hier also eher von einer semantischen Beschreibung der Funktionalität einer Komponente sprechen, wenn man das Verhaltensprotokoll entsprechend interpretiert.

Erweiterung der bloßen Schnittstellentypisierung um Verhaltensprotokolle

Die Verhaltensprotokolle scheinen eine interessante Erweiterung zu sein, wenn es darum geht, das Verhalten einer Komponente auf Basis ihrer Schnittstellen zu beschreiben und damit einen Teil des Laufzeitverhaltens statisch zu beschreiben. Allerdings ist nicht immer leicht festzulegen, welche Kommunikations-Interaktionen erlaubt sein sollten. Hier ist viel Erfahrung nötig. Außerdem schränken die Verhaltensprotokolle die Nutzung von Komponenten unter Umständen stark ein. Auch das Zusammenspiel von Komponenten unterschiedlicher Entwickler wird erschwert.

Nichtsdestotrotz sollten Protokolle, die die erlaubte Nutzung einer Komponente definieren, als Teil eines Gesamtintegritätskonzepts beachtet werden. Die oben skizzierten Schwierigkeiten lassen sich dadurch beheben, dass die Protokolle extern beschrieben werden. Damit sind sie von den Administratoren der Anpassungsumgebung entsprechend den Anforderungen im Anwendungsfeld änderbar. Die technische Prüfung, inwieweit die Pro-

tokolle zweier Komponenten zueinander passen, ist im Gegensatz zu den Verträgen im Design-by-Contract-Ansatz, leicht zur Anpassungszeit vorzunehmen.

Für den in diesem Vorhaben verfolgten Ansatz der Endbenutzerunterstützung bei der Komposition ist der Ansatz insofern interessant, als dass er durch Einschränkungen bezüglich der Nutzung der Komponentenschnittstellen Fehler bei der Komposition verhindert. Zu diskutieren ist jedoch, inwieweit die Darstellung eines Protokolls für Benutzer verständlich gemacht werden kann. Weiterhin bleibt auch offen, wie sich die Beschreibung eines Protokolls zur Nutzung bestimmter Komponenten in einem definierten Kontext für ein Komponentennetz erweitern lässt.

4.3.3 Komponenten und ihre Anforderungen an ihre Umwelt

Betrachtet man eine Komposition aus der Sicht einer einzelnen, darin enthaltenen Komponente, so kann man von einer Komponente und ihrer Umwelt sprechen. Diese Komponente kann ihrer Umwelt bestimmte Dienste anbieten. Umgekehrt benötigt sie (dafür) andere Dienste, die von der Umwelt zur Verfügung gestellt werden müssen. Die Interaktion mit der Umwelt ist der Komponente möglich über ihre Schnittstellen.

Reussners Ansatz der Parameterized Contracts [Reussner, 2001] basiert auf der Idee, dass eine Komponente vertraglich festgelegt bestimmte Dienste über ihre Schnittstellen zur Verfügung stellt. Kontrakte dienen dann dazu, die Eigenschaften von Software-Komponenten zu definieren. Ein Kontrakt bestimmt, wie eine Komponente unter bestimmten Voraussetzungen agiert. Laut Reussner [Reussner, 2001a] ist diese Sicht jedoch zu eng. Sie ermöglicht einer Komponente nur, in genau einer sehr speziellen Nutzungssituation eingesetzt zu werden. Typischerweise könnten Komponenten jedoch unter verschiedenen ähnlichen Situationen genutzt werden. Also wird diese grobe Sicht dadurch verfeinert, dass ebenso die Eingangsschnittstellen der Komponente berücksichtigt werden. Der Vertrag ist also auch abhängig von dem, was die Komponente von ihrer Umwelt erhält, er wird parameterisiert.

Werden zwei Komponenten miteinander verbunden, wird also ähnlich wie beim Design-by-Contract (siehe Abschnitt 4.3.1) geprüft, inwieweit Vor- und Nachbedingungen zueinander passen. Dieses Konzept wird jedoch so aufgeweicht, dass aus bestimmten Vorbedingungen jeweils individualisierte Nachbedingungen folgen. Vor- und Nachbedingungen einer Komponente sind dann miteinander verknüpft, die Nachbedingungen werden in Abhängigkeit der erfüllten oder nicht erfüllten Vorbedingungen bestimmt. Auf diese Weise können auch nicht exakt zueinander passende Komponenten miteinander verbunden werden, wenn die aufrufende Komponente (in Abhängigkeit der nicht erfüllten Vorbedingungen) mit weniger scharfen Nachbedingungen auskommt.

Weiterhin verwendet auch Reussner Schnittstellenprotokolle als weiterreichendes Integritätskriterium ähnlich den Behavior-Protokollen. Er beschreibt die Schnittstellenprotokolle mit Hilfe von deterministischen, endlichen Automaten [Reussner, 2001a]. Auf diese Weise lassen sich auch komplexe Protokolle für Entwickler einfach darstellen. Auch die Überprüfbarkeit, inwieweit ein genutztes mit einem angebotenen Protokoll übereinstimmt, lässt sich technisch einfach vornehmen.

Parameterisierte Verhaltensprotokolle als Kompatibilitätsprüfung zwischen Komponenten

Ähnlich wie auch das zugrundeliegende Design-by-Contract-Konzept stellt sich auch hier die Frage, inwieweit sich Vor- und Nachbedingungen zur Anpassungszeit ermitteln lassen. Ungeachtet dessen scheint der Gedanke, die sehr strikten Contracts aufzuweichen, beachtenswert. Die Idee, die Nachbedingungen in Abhängigkeit von Vorbedingungen zur formulieren, zeigt, inwiefern Komponenten und ihre (mögliche) Funktionalität voneinander abhängen.

Die Vor- und Nachbedingungen können im statischen Sinne die Existenz von Schnittstellen und damit verbunden Diensten sein, die eine Komponente bereitstellt. Auch diese sind häufig abhängig von weiteren Diensten. Beispielsweise kann eine Komponente zur Sortierung von Daten nur dann sinnvoll sortierte Daten liefern, wenn sie eine (unsortierte) Menge von Daten von einer anderen erhält. Insofern ist der Dienst, sortierte Daten abhängig von weiteren außerhalb der Sortier-Komponente liegenden Einflüssen anbieten zu können.

Betrachtet man darüber hinaus solche Abhängigkeiten transitiv, so kann man über eine Kette (oder ein Netz) von Komponenten Aussagen treffen. Dieser Ansatz kann dann ähnlich wie bei den in Abschnitt 4.1 diskutierten Workflows mit Hilfe von Petri-Netzen analysiert werden. Hier kann also eine Verbindung gezogen werden. Die mit Hilfe von Petri-Netzen analysierten Komponentennetze (siehe Abschnitt 4.1) können auf diese Weise erweitert werden um Stellen, die in Abhängigkeit vom Status anderer Komponenten und Verbindungen unterschiedliche Tokens enthalten und verarbeiten können.

4.3.4 Aspekt-orientierte Programmierung

Die objekt-orientierte Entwicklung hat an verschiedenen Stellen konzeptuelle Erweiterungen erfahren. Eine dieser Strömungen ist die aspekt-orientierte Programmierung (AOP).[16] Durch die zunehmende Komplexität aktueller Software-Projekte wird die Kapselung in Klassen immer schwieriger. Häufig kommt dazu, dass einzelne Aspekte der Funktionalität quer zur Klassenhierarchie liegen. Die Verwaltung solcher Aspekte nimmt entsprechend großen Raum ein.

Wagner und Mehner [Wagner und Mehner, 2000] definieren einen Aspekt als eine (meist technische) Anforderung an eine Software. Sie lässt sich mit den anderen Anforderungen, die Grundlage für die Dekomposition und den daraus resultierenden Klassenbaum sind, nur schlecht vereinbaren. In der Literatur (siehe auch [Elrad et al., 2001]) spricht man deswegen von „crosscutting concerns“. Solche Aspekte lassen sich dementsprechend nicht getrennt kapseln, sondern durchziehen den funktionalen Code mehrerer Objekte oder innerhalb eines Objekts den unterschiedlicher Methoden. Dies hat zur Folge, dass die Verständlichkeit des Codes abnimmt und die Wiederwendbarkeit der Klassenstruktur gemindert wird. Solche Aspekte sollen mit der Hilfe aspekt-orientierter Programmiersprachen separiert werden. Der Code eines Aspekts wird dann nicht mehr in verschiedenen Klassen verteilt, sondern als Aspekt getrennt definiert. In den Klassen, die dann einen Aspekt benötigen, wird dies durch spezielle Sprachkonstrukte (z. B. als join points) kenntlich gemacht.

Aspekt-orientierte Programmiersprachen wurden zu Beginn zumeist als Pre-Compiler zu objekt-orientierten Programmiersprachen realisiert. Eine aspekt-orientierte Programmiersprache zur Programmiersprache Java stellt AspectJ [AspectJ, 2002] dar.

---

[16] Das Konzept der aspekt-orientierten Programmierung ist nicht ausschließlich an objekt-orientierte Konzepte gebunden. Jedoch sind hier die ersten Ansätze diskutiert worden.

Ein einfaches Beispiel für die sinnvolle Anwendung von AOP-Konzepten liegt im Bereich Debugging. Der gemeinsame Aspekt liegt dabei oft in Schnittstellen, deren Nutzung zu unerwartetem Fehlverhalten führt. Der Wunsch liegt dann nahe, vor jedem Methodenaufruf den aktuellen Status der beteiligten Objekte zu erfahren. Mit OOP-Methoden müsste dann in jeder Klasse, die diese Schnittstelle benutzt, nach dem entsprechenden Aufruf der Schnittstelle gesucht und Debug-Code eingefügt werden. In AOP ist dieser Aspekt zentral erfassbar. Ähnlich ist Protocolling- oder Logging-Unterstützung leicht zu implementieren.

Als weiteres Beispiel für Anwendungen sei Change-Monitoring genannt. Mit AOP können Änderungen an einem Objekt zentral erfasst und behandelt werden, so dass nicht jede Methode, die auf das Objekt zugreift, selbst dafür Sorge tragen muss.

Integritätserhaltung als Aspekt

Wie schon in obigen Beispielen angeklungen, lassen sich Aspekte u. a. sehr gut für die Überwachung der Status von Objekten nutzen. Dies kann leicht auf die komponentenbasierten Architekturen übertragen werden. Eine Komponente kann dann globale Eigenschaften, die erhalten werden müssen, von einem Aspekt prüfen lassen. Die Ankerpunkte für den Aspekt verändern die Komponente nicht wesentlich. Auf diese Weise lassen sich Integritätsprüfungen zu den Komponenten hinzufügen, ohne Änderungen an deren Code vornehmen zu müssen. Darüber hinaus können diese Prüfalgorithmen auch unabhängig von den Komponenten, die sie benutzen, geändert werden. Mit Aspekten lassen sich also sehr flexibel dynamische Integritätsprüfungssysteme realisieren. Aus diesem Grund sind in AspectJ auch schon Mechanismen vorgesehen, um während der Kompilierung Fehlermeldungen zu generieren, wenn ein Aspekt nicht anwendbar ist. Auch wurden schon Versuche unternommen, das Design-by-Contract in Aspekten umzusetzen.

4.3.5 Kompositionspläne

Applikationsentwickler oder sogar Endanwender können mit Hilfe Agenten-gestützter Kompositionswerkzeuge Applikationen selbst zusammenstellen. Die von Birngruber und Hof [Birngruber und Hof, 2000] konzipierten BeanPlans [Birngruber, 2000a] sind Applikations-Pläne, die aus einer Sammlung von Bedingungen bestehen.

Birngruber zeigt, wie ein Applikations-Plan mit Hilfe der Component Plan Language (CoPL) definiert wird. Ein solcher beschreibt beispielsweise Anforderungen an die konkrete Instanz einer Applikation, welche Arten von Komponenten enthalten sein müssen. Hier können auch Abhängigkeiten beschrieben werden („Wenn Komponente A enthalten ist, dann wird Komponente C benötigt“). Diese Applikations-Pläne werden dann mit Hilfe eines Agenten, dem CoML-Generator, ausgewertet. Das Ergebnis dieser Interaktion zwischen CoML-Generator und Benutzer ist dann schließlich ein Plan für eine konkrete Applikation. Er wird in CoML (Component Markup Language) [Birngruber, 2001], einer Sprache ähnlich CAT (siehe Abschnitt 2.3), beschrieben.

Der (interaktive) Agent erzeugt dann gemeinsam mit dem Benutzer also die Beschreibung einer konkreten Applikation. Dabei spielen Alternativen, die sogenannten Decision Spots, eine wichtige Rolle. Die Applikation schließlich entsteht, indem dieser Entscheidungsbaum mit Hilfe von Benutzern abgearbeitet wird. Das Ergebnis ist eine in jedem Fall korrekte (im Sinne der im Plan beschriebenen Integritätsbedingungen) Applikation. Das zentrale Element dieser Sprache sind also die Decision Spots. Sie können lokale oder auch globale Bedeutung haben. Lokale Entscheidungen beziehen sich beispielsweise auf die Auswahl einer Komponente an einer bestimmten Stelle innerhalb der Applikation. Globale Entscheidungen beschreiben Abhängigkeiten zwischen Einzelentscheidungen und stellen

die Verbindung zwischen einer Einzelkomponente und dem Komponentennetz her. Auf diese Weise lassen sich domänen-spezifische Schablonen für eine Gruppe von Applikationen erstellen [Zdun, 2002]. Eine solche Schablone kann als sehr stark eingeschränkte domänen-spezifische Anpassungssprache gesehen werden, in der Benutzer jedoch de facto nicht frei agieren, sondern nur die vom CoML-Generator generierten Entscheidungswege ablaufen können.

.

Pläne als Sammlungen von Integritäts-Bedingungen

Mit den BeanPlans, die in CoPL formuliert werden, werden prinzipiell Bedingungen an eine Anwendung beschrieben. Diese Bedingungen lassen Freiheiten, die dann von einem Agenten angefragt werden. Insofern lässt sich dieser Ansatz als „automatische Applikationsgenerierung mit Benutzerbeteiligung“ beschreiben. Trotzdem kommt er der hier verfolgten Idee recht nahe. Bedingungen müssen eingehalten werden und regeln die Komposition. Die Bedingungen haben untereinander Abhängigkeiten, so dass durch die Entscheidungen, die ein Benutzer an einem Decision Spot trifft, immer neue Teil-Entscheidungsbäume erzeugt werden. Die Regeln beziehen sich dabei auf die Existenz von Komponenten innerhalb der Komposition, voneinander abhängige Parametereinstellungen oder Verbindungen. Ähnliche Regeln sind auch nötig, wenn Benutzer durch Integritätsprüfung bei Anpassungsvorgängen unterstützt werden sollen. Sie sollen dann allerdings nicht von einem Agenten ausgewertet und in einen Entscheidungsbaum transformiert werden. Stattdessen werden die Regeln genutzt, eine frei erstellte Komposition zu prüfen. Dies impliziert dann auch, dass die Regeln weniger hart ausgelegt sein müssen, da sich dadurch ja kein automatischer Kompositionsprozess mit nur wenigen Einflussmöglichkeiten steuern lassen soll. Im Gegenteil geht es darum, den Benutzern möglichst viele Freiheiten bei der Komposition ihrer Anwendungen einzuräumen und nur dann einzugreifen, wenn Fehler auftreten.

4.4 Fazit

Dieses Kapitel hat gezeigt, wie in den unterschiedlichen Teildisziplinen mit dem Problem der Integritätserhaltung von Informationssystemen umgegangen wird. Einige dieser Ansätze lassen sich mit nur wenig Modellierungsarbeit für ein Integritätskonzept für komponentenbasierte Anpassbarkeit übertragen. Dazu gehört in jedem Fall die Möglichkeit, mit Hilfe von Constraints und Triggers, Bedingungen an die Verwendung von Einzelkomponenten oder auch ganzen Kompositionen zu knüpfen. Dieser aus dem Bereich der Datenbanken stammende Ansatz wurde schon für die semi-automatische Applikationsgenerierung auf Basis der oben beschriebenen BeanPlans angewandt.

Wichtig scheint aber neben der Beschränkung der Verwendung von Komponenten und deren Parameterisierung auch zu sein, wie sich ein Komponentennetz als ganzes mit seinen Kontrollflüssen beschreiben lässt. Hier kann auf Arbeiten im Bereich des Workflow-Management aufgesetzt werden. Ebenso sollten verschiedene Konzepte, die die Verwendung von Schnittstellen zwischen Komponenten aus Sicht der Software-Technik betrachten, untersucht und – wenn sinnvoll – in das Gesamtkonzept integriert werden.

Ein Integritätskonzept basiert also auf den Grundkonstrukten formulierter Bedingungen an Komponenten und Kompositionen (Constraints) und der Analyse der Kommunikationsflüsse innerhalb einer Komposition. Die Idee der Constraints kann grundsätzlich aus dem Bereich der Datenbanken übernommen werden. Allerdings ist zu prüfen, welche Bedingungen formulierbar sein sollten. Sie können erweitert werden um Prädikate, die eine Prüfung

von Verträgen oder Verhaltensprotokollen zulassen. Im Gegensatz zu den Ansätzen aus dem Bereich der Datenbanken sind aber auch Einschränkungen vorzunehmen. Da die hier zu entwickelnde Integritätsprüfung vordringlich das Erlernen von Anpassungssprachen erleichtern soll, steht die Verständlichkeit von Bedingungen im Vordergrund. Weiterhin sollten die Bedingungs- wie auch die Ausführungsbereiche eines Triggers so konzipiert sein, dass sie sich problemlos automatisieren lassen.

Auch der aus dem Bereich des Workflow-Management stammende Ansatz, Kommunikationsflüsse auf deren Integrität zu untersuchen, kann übernommen werden. Hier ist genau zu untersuchen, inwieweit die Analysetechniken helfen, Fehler in einer Applikation zu erkennen und auch hier den Benutzern eine Hilfestellung bei der Anpassung ihrer Komposition zu geben. Dies soll in Kapitel 8 vertieft werden.

5 Eine Anpassungsumgebung für die FREEVOLVE-Plattform

Nachdem die Grundlagen im Bereich Software-Ergonomie und End-User Development (Kapitel 3) gelegt wurden, geht es nun darum, die im Rahmen dieses Vorhabens neu entwickelte Anpassungsumgebung für die FREEVOLVE-Plattform zu beschreiben. Dabei ist das Vorgehen wie folgt: Zu Beginn werden spezielle Anforderungen, die sich an eine solche Anpassungsschnittstelle ergeben, aufgeführt und unter Berücksichtigung der früher beschriebenen Anforderungen aus der Literatur diskutiert. Anschließend wird die prototypische Entwicklung der Software vorgestellt und anhand von Beispielen deren Funktionsweise beschrieben. Das Kapitel schließt mit zwei unterschiedlichen Evaluierungen. Erstere analysiert die Güte der Software im Hinblick auf die aus der Literatur entlehnten Anforderungskriterien. Zweitere wurde im Rahmen dieser Arbeit als qualitative Benutzungsstudie durchgeführt. Hier werden vordringlich Schwachstellen, aber auch positive Kritik am Software-Prototypen beschrieben.

Wie schon in Kapitel 2 in einer Übersicht dargestellt, wird bei der Software-Entwicklung im Rahmen dieses Vorhabens ein zyklischer Ansatz (vgl. [Floyd et al., 1989], [Boehm, 1988]) zugrunde gelegt. Der in diesem Kapitel beschriebene Prototyp basiert also auf einer Reihe von Vorarbeiten, die in Kapitel 2 beschrieben wurden. Weiterhin sind Erkenntnisse aus einer Vorstudie integriert, die sich damit beschäftigte, wie Benutzer mit Hilfe komponentenbasierter Email-Filter zur Koordinierung ihre verteilte Gruppenarbeit optimieren [Förster, 2002]. Dabei wurde ebenfalls prototypisch eine visuelle Anpassungsumgebung entwickelt, in der Benutzer Filterkomponenten auswählen, parameterisieren und verbinden konnten. Ein Explorationsmodus unterstützte hier die Fehlerkorrektur. Der hier beschriebene Prototyp stellt im Rahmen dieses Vorhabens die erste Implementierung dar. Er wurde weiter verfeinert und um eine Integritätsprüfung ergänzt.

5.1 Anforderungen an eine Anpassungsumgebung für FREEVOLVE

Innerhalb der Client-Server-Plattform FREEVOLVE lassen sich komponentenbasierte Applikationen (Kompositionen) starten. Der Fokus liegt hier zum einen auf der weitest gehenden Unterstützung verteilter Applikationen, zum anderen auf möglichen Veränderungen an den Kompositionen zur Laufzeit (nähere Informationen über die Funktionsweise von FREEVOLVE finden sich in Anhang A). Anpassungen sollten von den Benutzern selbst durchführbar sein. Dabei ist es wichtig, dass die Anpassungsumgebung alle Anpassungsfunktionen unterstützt und zugleich möglichst einfach zu bedienen ist. In Kapitel 2.3 wurden in diesem Zusammenhang die im Rahmen der Neuentwicklung oder Änderung einer Komposition relevanten Operationen herausgearbeitet. Bei der Anpassung einer möglichen Client-Server-Plattform müssen also vor allem folgende Punkte beachtet werden:

Darstellung der Komponenten (Operanden)

Beim Entwurf der Anpassungsumgebung muss darauf geachtet werden, dass alle in einer Komposition enthaltenen Komponenten sichtbar und von den Benutzern veränderbar sind. Diese Unmittelbarkeit der zu bearbeitenden Objekte erleichtert das Verständnis für eine Applikation enorm. Die nicht vollständige Darstellung einer Applikation und damit der

Verlust der Übersicht war eine der Schwächen der 3D-Anpassungsschnittstelle, die weiter oben beschrieben wurde [Hallenberger, 2000]. Auf dieses Problem weisen auch Gellenbeck und Cook [Gellenbeck und Cook, 1991] hin, die darlegen, dass der semantische Status des Systems durch Signale und Hinweise im Blickfeld der Benutzer liegen muss. Um Benutzer nicht zu überfordern, sollte darauf geachtet werden, dass eine Abstufung der möglichen Informationen in Bezug auf ihre Relevanz erfolgt. Davon abhängig sollte bei der Auswahl der angezeigten Informationen Bezug genommen werden [Green et al., 1991].
Die FREEVOLVE-Plattform bietet von sich aus schon die Möglichkeit der hierarchischen Komposition. Auch dies sollte bei der Darstellung der Komponenten berücksichtigt und genutzt werden. Sie erleichtern für unerfahrenere Benutzer den Zugang dadurch, dass ihre Semantik näher an der Nutzungssemantik der Applikation liegt. Dies ist bei Low-Level-Komponenten, die einen sehr speziellen Aspekt des Systems bedienen, häufig nicht der Fall [Lewis und Olson, 1987].

Weiterhin sollte noch berücksichtigt werden, dass nicht alle Komponenten einer Komposition für die Anwender zur Laufzeit sichtbar sind. Das ist häufig bei Controller-Komponenten der Fall (in Kapitel 2.2 wurde hierfür als Beispiel eine SQL-Suchmaschine eingeführt). Weiterhin gilt dies speziell im FREEVOLVE-Kontext für alle Server-Komponenten. Für solche Komponenten muss im Anpassungsmodus eine geeignete Darstellungsmöglichkeit gefunden werden, die den Benutzern hilft, die Semantik der Komponenten intuitiv zu verstehen.

Schließlich sollten alle für das Verständnis und die Konfiguration notwendigen Eigenschaften einer Einzelkomponente dargestellt werden und so im Zugriffsbereich der Anpassenden liegen. Dabei ist darauf zu achten, sie so zu strukturieren, dass Benutzer nicht von der Vielzahl der Möglichkeiten überfordert werden.

Daraus ergibt sich, eine Anpassungsumgebung zu konzipieren, die im wesentlichen auf der schon vorher erwähnten 2D-Plattform aufsetzt. Allerdings muss ein Weg gefunden werden, auch verteilte Applikationen darstellen zu können.

Bereitstellung der Anpassungsfunktionalität (Operationen)

Operationen beziehen sich auf die grundlegenden Eigenschaften einer Komposition. Sie beziehen sich auf ganze Komponenten, spezielle Aspekte von ihnen und die Beziehung von Komponenten untereinander.

- **Operationen auf einer Komponente**: Komponenten sollten sich sowohl einer Komposition hinzufügen lassen als auch wieder entfernt werden können. In zweitem Fall ist darauf zu achten, dass automatisch auch alle Abhängigkeiten zu anderen Komponenten entfernt werden. Dies kann eine Verbindung zwischen der entfernten und einer anderen Komponente sein. Weiterhin ist es aber auch möglich, dass eine Komponente Teil einer sie umgebenden abstrakten Komponente war. In solchen Fällen müssen auch Änderungen an dieser vorgenommen werden.
  Um Komponenten von einer in eine andere abstrakte Komponente zu verschieben, sind zusätzlich Möglichkeiten zum Ausschneiden und Einfügen vorzusehen. Sie erleichtern das Anpassen insofern, als dass die Komponente dann nicht wie beim Hinzufügen in Bezug auf ihre Parametereinstellungen (s.u.) verändert werden muss, sondern die schon vorhandenen Parametereinstellungen erhalten bleiben. In diesem Kontext sollte es auch möglich sein, Komponenten direkt zu kopieren.
  Komponenten können unabhängig von ihrer Typenbeschreibung benannt sein. Dies erhöht unter Umständen das Verständnis für eine Komposition, da geänderte Beschreibungen zumeist die Funktionalität der Komponente genauer im Kontext der speziellen Komposition darstellen. Namensänderungen – evtl. erweitert um zusätzliche Erläuterungstexte – sollten einfach an einer Komponente durchgeführt werden können.

Abstrakte Komponenten haben grundsätzlich keinen automatisch generierten Namen. Sie müssen also immer von den anpassenden Benutzern benannt werden.

- **Parameter einer Komponente**: Viele Komponenten lassen sich in ihrem Aussehen oder Verhalten dadurch verändern, dass Parameter verändert werden. Diese Möglichkeit sollte den Benutzern leicht zugänglich sein. Weiterhin ist es hier sinnvoll, durch Beispiele und automatische Überprüfung der eingegebenen Werte durch das System eine aktive Fehlervermeidung bereitzustellen.
  Abstrakte Komponenten verfügen nicht über eigene Parameter. Sie können diese jedoch von en in ihnen enthaltenen Komponenten übernehmen. Durch auf diese Weise hergestellte Verknüpfungen zwischen einem Parameter einer abstrakten Komponente und Parametern in ihr enthaltener Komponenten lassen sich dann auch Eigenschaften gebündelt verändern.
  Aus diesem Grund sollten das Anlegen, Benennen und Löschen von Parametern einer abstrakten Komponente als Operationen bereitgestellt werden. Weiterhin sollte es auch eine einfache Möglichkeit geben, Parameter auf die nächsthöhere Hierarchie-Ebene innerhalb der Komposition „hochzureichen“ (siehe Abbildung 10).

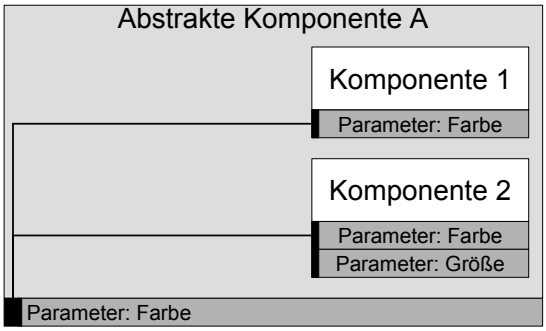


Abbildung 10: Parameter von abstrakten Komponenten

- **Ports eine Komponente**: Ebenso wie die Parameter einer Komponente von der nächsthöheren abstrakten Komponente verwaltet werden können, ist es auch möglich, abstrakte Komponenten mit Ports auszustatten. Auch hier werden diese Ports mit den innerhalb der abstrakten Komponente vorhandenen Komponenten verbunden. Als Operationen müssen demnach das Hinzufügen von typisierten Ports, das Entfernen ebensolcher, die Benennung und das Hochreichen unterstützt werden.
- **Verbindungen zwischen Komponenten**: Anpassen von Kompositionen besteht neben der Auswahl von Komponenten und dem Parameterisieren darin, Verbindungen zwischen den Komponenten aufzubauen. Sie ermöglichen die Interaktion zwischen ihnen. Dabei sollten typkonforme Schnittstellen ausgewählt und eine Verbindung aufgebaut werden. Als wesentliche Operationen sind hier das Hinzufügen und Entfernen von Ports zu unterstützen. Wie auch bei der Parameterisierung von Komponenten kann eine aktive Fehlervermeidung das Anpassen unterstützen.

5.2 TailorClient: Eine graphische Anpassungsumgebung

Nachdem die wesentlichen Anforderungen im letzten Abschnitt übersichtsartig dargestellt wurden, geht es nun darum, die erste prototypische Implementierung einer graphischen Anpassungsumgebung für FREEVOLVE zu beschreiben.

Grundsätzlich wurde der neue TailorClient so konzipiert, dass unterschiedliche Sichten auf die anzupassende Komposition jeweils unterschiedliche Aspekte betonen. So ist eine Sicht vorgesehen, die dediziert zur Positionierung der visuellen Komponenten geeignet ist. Eine weitere stellt die Verbindungen innerhalb der Komposition in den Vordergrund. Im folgenden Abschnitt werden die unterschiedlichen Sichten detaillierter beschrieben.

Der Anpassungsclient wird aus der FREEVOLVE-Laufzeitumgebung (TailorClient) heraus gestartet. Dabei wird angenommen, dass die aktuell laufende Applikation angepasst werden soll. Ist keine Applikation geladen, so wird eine leere Komposition in der Anpassungsumgebung angezeigt. Die Komposition kann nun unter Nutzung verschiedener miteinander synchronisierter Werkzeuge, die die Komposition jeweils aus unterschiedlichen Perspektiven betrachten, verändert werden. Die Werkzeuge werden in den folgenden Unterabschnitten ausführlich beschrieben.

Die Laufzeitumgebung bleibt während des gesamten Anpassungsvorgangs aktiv, wobei die Applikation erhalten bleibt, in der sie sich vor dem Start des Anpassungsmodus befand. Erst die Übernahme der Änderungen verändert die Applikation vollständig. Dies entspricht den Vorstellungen eines Neutralmodus (vgl. [Paul, 1994]), in dem Änderungen wirkungslos bleiben. Die Übernahme der Änderungen kann während des Anpassungsvorgangs explizit durchgeführt werden, beispielsweise zum Test der Anpassungen. In jedem Fall werden sie beim Verlassen des Anpassungsmodus übernommen. Hier schützt eine Sicherheitsabfrage vor unbeabsichtigten Änderungen (Änderungen übernehmen, Applikation unter anderem Namen speichern, Änderungen verwerfen).

Änderungen an den Serverkomponenten werden beim aktuellen Prototypen nicht übernommen. Sie haben Auswirkungen auf die Applikationen Anderer und können zu einem Fehlverhalten des gesamten Systems führen.

5.2.1 Komponenten und ihre Parameter

Für die Darstellung einer Komposition in der Übersicht und damit verbunden aller Parameter, die zu den jeweiligen Komponenten gehören, wurde der sogenannte Komponenten-Explorer entworfen (siehe Abbildung 11). Die Komposition wird hier mit allen ihren Hierarchieebenen als Baum dargestellt. Damit orientiert der Komponenten-Explorer sich an der weitreichend bekannten Baum-Metapher, die auch Basis von Standard-Werkzeugen, wie dem Windows Datei-Manager (Explorer) ist.

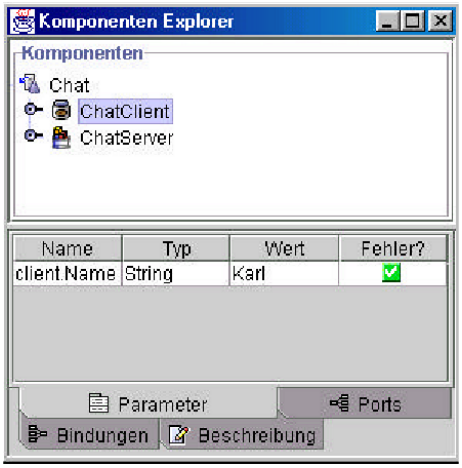


Abbildung 11: Komponenten-Explorer

Im oberen Bereich der Anwendung wird die Komposition als hierarchische Struktur dargestellt. Dabei werden für die verschiedenen Ebenen innerhalb der hierarchischen Struktur der Übersicht halber unterschiedliche Icons verwendet (siehe Tabelle 7):

Icon	Komponentenart	Bedeutung
	Applikation	Die Applikation stellt immer die Wurzel des Kompositionsbaums dar. Sie enthält alle weiteren Komponenten und stellt damit das Herz der Anwendung dar.
	Einfache Komponente	Eine einfache Komponente ist eine programmierte Software-Einheit. Sie stellt die Basis der Kompositionen dar und befindet sich an den Wurzeln eines Baums. Eine Kaffeebohne (Bean) wurde in Anlehnung an die von Sun vorgeschlagene Ikonographie gewählt.
	Serverkomponente	Serverkomponenten sind grundsätzlich ebenso einfache, d.h. instanziierbare Komponenten. Da sie im Kontext dieser Komposition jedoch auf dem Server instanziiert werden, kommt ihnen eine besondere Bedeutung zu. Sie können von mehreren Benutzern gleichzeitig verwendet werden. Insofern ist bei Anpassungen an ihnen große Vorsicht geboten.
	Abstrakte Komponente	Abstrakte Komponenten stellen die inneren Knoten des Kompositionsbaums dar. Sie enthalten andere abstrakte oder einfache Komponenten. Sie haben auch keine originären Schnittstellen oder Parameter, sondern verwalten die an sie von einer niedrigeren Kompositionsebene übergebenen („Hochreichen“ von Parametern oder Schnittstellen, siehe oben).

Tabelle 7: Komponentenarten im Überblick

Die Baumansicht dient in erster Linie zum Markieren einzelner Komponenten. Weiterhin ist es aber auch möglich, über das Kontextmenü (mit der rechten Maustaste erreichbar) Komponenten zu entfernen oder abstrakten Komponenten (wie auch der Applikationskomponente) neue hinzuzufügen. Hier kann also relativ leicht die komplette Baumstruktur der Komposition verändert werden.

Im unteren Bereich der Ansicht des Komponenten-Explorers finden sich Detailinformationen zur aktuell markierten Komponente. Sie beziehen sich auf die veränderbaren Parameter einer Komponente, Bindungen zu anderen Komponenten, verfügbare Ports und einer Beschreibung der Komponente. Letztere Beschreibung ist eine textuelle Beschreibung der Komponenten, ihrer Einsatzmöglichkeiten und ihrer Funktionalität. Sie ist nicht änderbar, sondern wurde der Komponente von ihren Entwicklern beigefügt (näheres dazu findet sich in Anhang A). Der Übersichtlichkeit halber werden nicht alle der oben genannten Informationen auf einmal dargestellt. Stattdessen sind die einzelnen Bereiche über entsprechende Reiter anwählbar. Mit dem Komponenten-Explorer können grundsätzlich alle oben genannten Anpassungen durchgeführt werden. Dabei ist eine Operation immer an exakt eine Komponente geknüpft (die aktuell ausgewählte).

Für die Veränderung von Parametern ist dies der geeignete Weg. Hier können Werte in einer Tabelle in textueller Form verändert werden. Abbildung 11 zeigt die Parameter der Komponente „ChatClient“. Nur ein Parameter „clientName“ ist änderbar. Durch Klicken in die entsprechende Werte-Zelle kann der aktuelle Wert „Karl“ beliebig geändert werden. Ist ein Parameter statt als beliebige Zeichenkette als Zahl (Ganzzahl oder reelle Zahl) typisiert, so würde die Eingabe einer Zeichenkette (z. B. „Karl“) einen Fehler erzeugen. Die Änderung wird dann nicht übernommen. Das Hinzufügen von neuen Parametern erfolgt durch die Eingabe eines entsprechenden Wertes in eine leere Tabellenzeile. Da solche Parameter dann nicht mit den physikalischen Eigenschaften der Komponente verknüpft sind, muss eine Verknüpfung mit einem Parameter einer anderen Komponente hergestellt werden. Wie schon weiter vorne erläutert kann es sich dabei um eine Komponente auf der gleichen hierarchischen Ebene innerhalb der Komposition oder um eine aus der darunter liegenden Ebene (im Falle einer abstrakten Komponente) stammenden handeln.

Auch Operationen an Ports wie das Hinzufügen neuer und Entfernen alter Ports können durchgeführt werden. Hierfür ist eine ähnliche Vorgehensweise wie beim Hinzufügen und Entfernen von Parametern vorgesehen. Die Bindungen einer Komponente zu einer anderen lassen sich ebenso anzeigen. Allerdings ist der Komponenten-Explorer zum Verbinden zweier Komponenten eher ungeeignet, da immer nur die aktuell ausgewählte Komponente angezeigt wird. Hierfür eignet sich der im folgenden beschriebene Komponenten-Editor besser.

5.2.2 Verbindungen zwischen Komponenten

Wie schon diskutiert ist der Komponenten-Explorer für die Bearbeitung der Verbindungen zwischen Komponenten ungeeignet. Was benötigt wird ist eine Sicht auf die Komposition, die alle Komponenten gemeinsam darstellt. Auf diese Weise lassen sich Verbindungen leicht visualisieren und die Zusammenhänge zwischen den Komponenten so schnell im Überblick erfassen. Zu diesem Zweck dient der Komponenten-Editor (siehe Abbildung 12).

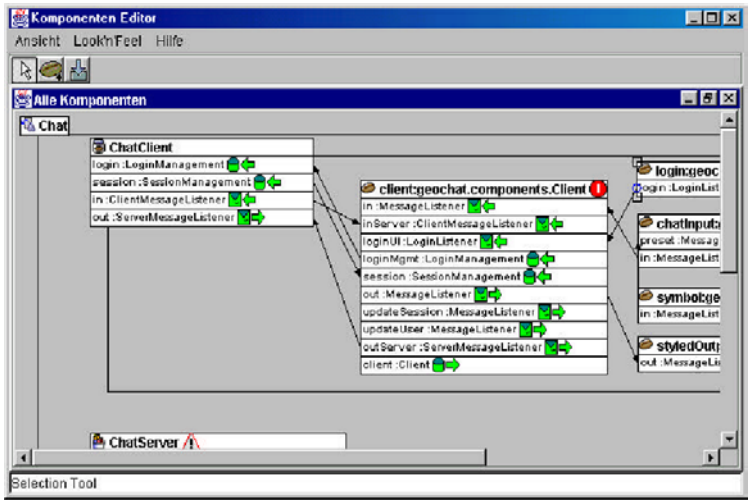


Abbildung 12: Komponenten-Editor

Im Komponenten-Editor werden die einzelnen Komponenten als Kästen schematisch dargestellt. Zur Orientierung steht in der ersten Zeile einer Komponente ihr Name. Die Darstellung ist angelehnt an die von UML-Klassendiagrammen, die explizit für Diskussionen zwischen Entwicklern und Domain-Experten im Rahmen der Software-Entwicklung konzipiert wurden [Fowler und Scott, 1999]. Abstrakte Komponenten sind als solche gekennzeichnet. Durch Doppelklick auf eine abstrakte Komponente wird diese geöffnet und alle in ihr enthaltenen Komponenten werden dargestellt. Dabei wird die Zusammengehörigkeit (zur umgebenden, abstrakten Komponente) durch einen Rahmen signalisiert. In Abbildung 12 ist die Komponente „ChatClient“ eine abstrakte Komponente, sie enthält die vier in dem Bildschirmausschnitt dargestellten Komponenten (Client-Komponente, Login-Komponente, Eingabe-Komponente und Ausgabekomponente).

Die möglichen Ports, die eine Komponente anbietet, sind in den Zeilen unterhalb ihres Namens dargestellt. Wichtigstes Erkennungszeichen eines Ports ist sein Name. Er wird am Anfang der Zeile dargestellt. Anschließend wird durch kleine Symbole (Icons) signalisiert, ob es sich um einen Eingangs- oder Ausgangsport handelt. Weiterhin ist gekennzeichnet, ob der Port eine gemeinsam verwendete Variable darstellt oder ein Nachrichtenport ist. Tabelle 8 gibt einen Überblick über die verschiedenen graphischen Symbole.

		Polarität	
		(Eingangsport (Required)	Ausgangsport (Provided)
Protokoll	Gemeinsam benutzte Variable (Shared Object)		
	Nachrichtenport (Event)		

Tabelle 8: Symbole für unterschiedliche Verbindungsarten

Die Verbindungen lassen sich nun einfach per Drag and Drop (Ziehen der Maus bei gehaltener Maustaste) zwischen zwei typkonformen Ports herstellen. Eine Linie zeigt dann die Verbindung an. Die Verbindungen können gelöst werden, indem man mit der rechten Maustaste auf eine Verbindung klickt und im Kontextmenü den Punkt „Verbindung entfernen“ auswählt.

Auch in dieser Ansicht lassen sich Komponenten hinzufügen. Sie werden als neue Rechtecke angezeigt und lassen sich durch Verschieben einer abstrakten Komponente zuordnen.

5.2.3 Spezielle Parameter sichtbarer Komponenten

Eine spezielle Art von Parametern stellen im Kontext der Anpassung Größe und Position der visuellen Komponenten dar. Wie andere Parameter auch lassen sie sich leicht im Komponenten-Editor verändern. Allerdings bietet es sich gerade für diese Parameter an, eine einfache Editierfunktionalität zur Verfügung zu stellen. Zu diesem Zweck verfügt der TailorClient über eine dritte Sicht auf die Applikation.

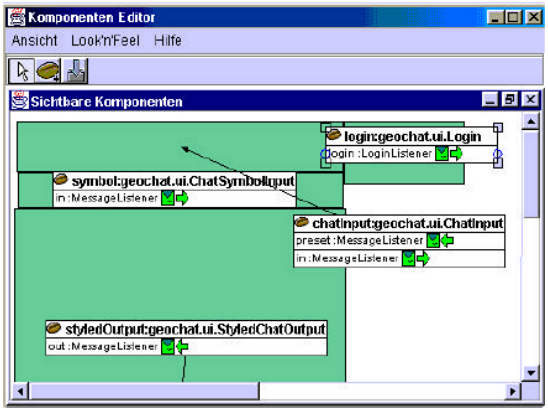


Abbildung 13: Visuelle Komponenten

In Abbildung 13 sieht man diese Darstellung. Hier werden alle visuellen Komponenten mit Platzhaltern angezeigt. Sie lassen sich verschieben oder in der Größe verändern. Zur Laufzeit nicht-sichtbare Komponenten sind in dieser Ansicht ausgeblendet ebenso wie Komponenten, die auf dem Server laufen.

5.2.4 Zusammenspiel der verschiedenen Ansichten

Die obigen drei Ansichten ermöglichen ein effizientes Anpassen. Dabei unterstützt jede Ansicht spezielle Operationen. Benutzer haben je nach Bedarf die Möglichkeit, bestimmte Ansichten ein- und auszublenden.

Abbildung 14: Wahl zwischen verschiedenen Sichten

Wie in Abbildung 14 dargestellt sind unterschiedliche Ansichten über die Menüstruktur des TailorClient auswählbar. Neben den schon beschriebenen Sichten gibt es hier darüber hinaus auch die Möglichkeit, dediziert nur die Teilapplikationen, die auf dem Server bzw. auf dem Client ablaufen, anzeigen zu lassen. Hinzu kommt eine Übersicht über aktuell verwen-

dete Komponenten. Der Komponenten-Explorer (siehe Abschnitt 5.2.1) steht hier nicht zur Auswahl. Er ist ständig aktiv und stellt das Kernstück der Anpassungsumgebung dar.

Entscheidend für die Bedienbarkeit der Anpassungsschnittstelle ist, dass alle Ansichten zu jedem Zeitpunkt die identische Komposition und ihre aktuelle Konfiguration darstellen. Zu diesem Zweck werden alle aktuell geöffneten Ansichten bei jeder Änderung miteinander synchronisiert. Die unterschiedlichen Ansichten benachrichtigen sich also wechselseitig über Änderungen an der Komposition. Solche Änderungen werden jedoch nicht unmittelbar in die Komposition innerhalb der Laufzeitumgebung übernommen. Dies geschieht erst nach expliziter Ausführung des Befehls „Änderungen übernehmen“, der sowohl über die Menüstruktur des TailorClient als auch über die Kontextmenüs in den einzelnen Ansichten zur Verfügung steht. Diese Vorgehensweise entspricht der Idee, dass ein Anpassungsvorgang immer als eine Transaktion zu sehen ist. Dementsprechend ist eine durchgeführte Anpassungsoperation für sich betrachtet wenig sinnvoll (z. B. das Entfernen einer Verbindung zwischen zwei Komponenten). Erst nachdem alle für einen sinnvollen Anpassungsvorgang nötigen Operationen durchgeführt wurden, sollten sie in die tatsächliche Applikation übernommen werden. Außerdem dient diese Vorgehensweise selbstverständlich auch der Erhaltung der Systemstabilität.

5.3 Evaluierung des Prototypen

Um Aussagen über die Ergonomie des Prototypen gewinnen zu können, wurden verschiedene Studien durchgeführt. Eine erste Studie beschäftigte sich mit der grundlegenden Frage, inwieweit Benutzer die Semantik komponentenbasierter Software erfassen können. Einfache Anpassungsaufgaben zur Änderung einer Komposition sollten hierbei selbstständig durchgeführt werden. Aufbauend auf diesen Erfahrungen wurde der in Abschnitt 5.2 vorgestellte Prototyp untersucht. Die Studie bildet die Grundlage für eine sinnvolle Einbindung eines Integritätsprüfungskonzepts.

In beiden Studien wurde die Thinking Aloud-Methode nach Nielsen [Nielsen, 1992], [Nielsen, 1993] unter Laborbedingungen gewählt. Unter „Thinking Aloud“ versteht man die handlungsbegleitende Verbalisierung von Gedanken. Diese Methode bietet sich bei kognitiven Aufgaben und beim Problemlösen an und stellt eine effiziente und effektive Technik dar (siehe auch [Jørgensen, 1989], [Denning et al., 1990], [Nielsen, 1994]), um Benutzerschnittstellen schon in der Entwicklungsphase zu bewerten.

Nielsen fordert für die Thinking-Aloud Methode, dass diese anhand von realistischen Szenarien durchgeführt wird. Die Methode sieht vor, dass den Testpersonen konkrete und sinnvolle, mit Bezug zum realen Nutzungskontext nachvollziehbare Aufgaben gestellt werden. Diese werden durch die Testpersonen unter lautem Verbalisieren der Aktionen und Eindrücke bearbeitet. Die Probanden beschreiben hierfür die Darstellung auf dem Bildschirm, welche Handlungsmöglichkeiten sie zur Erfüllung der gestellten Aufgabe sehen, warum sie eine bestimmte Handlungsalternative auswählen und wie sie das Systemverhalten deuten. Die getroffenen Aussagen und auftretenden Probleme werden protokolliert. So ist es möglich, in einer Ursachenanalyse die Schwachstellen im Design der Benutzungsschnittstelle zu identifizieren. In einem anschließenden Interview haben die Testpersonen Gelegenheit – unabhängig von konkret zu bearbeitenden Aufgaben – ihre Eindrücke bei der Nutzung des Prototypen wiederzugeben und Verbesserungsvorschläge zu formulieren. Diese Gespräche liefern sehr häufig wertvolle Hinweise, die in die Folgeversion des Prototypen einfließen können.

5.3.1 Vorstudie: Komponentenbasierte Anpassbarkeit nicht-visueller Komponenten

Im Gegensatz zu den schon erwähnten Prototypen für eine Anpassungsschnittstelle für das FREEVOLVE-System (siehe Abschnitt 2.3.3), ist die hier vorgestellte in ihrer Sicht auf die Komposition deutlich abstrakter. Sie entfernt sich damit teilweise weit von der zur Laufzeit sichtbaren Benutzungsschnittstelle der Komposition. Aufgrund dessen ist es für Benutzer deutlich schwieriger, diese beiden Perspektiven miteinander zu vergleichen und Zusammenhänge zu erkennen. Anpassungen erfordern im gleichen Maße eine größere Abstraktionsfähigkeit. Insofern ist zu erfragen, inwieweit der Ansatz der komponentenbasierten Architekturen auch dann noch gut geeignet ist, Anpassbarkeit für Endbenutzer zu unterstützen.

Komponentenbasierte Email-Filter

Im Rahmen dieses Projekts wurde vor diesem Hintergrund ein Software-System entwickelt, mit dem Benutzer innerhalb einer Arbeitsgruppe (oder innerhalb einer virtuellen Organisation, vgl. [Picot et al., 1998]) ihren Email-Eingang organisieren können [Förster, 2002]. Die an den Email-Server gebundene Applikation verwaltet zu diesem Zweck verschiedene Filterregeln. Ein graphischer Anpassungsclient ermöglicht es den Benutzern, Filter komponentenbasiert zu konstruieren. Abbildung 15 zeigt diesen Client, in dem ein Filter dargestellt wird, der in Abhängigkeit von verschiedenen Bedingungen (Bedingungs- oder Weichen-Komponenten werden als Rauten dargestellt) eingehende Mails an verschiedene Empfänger weiterleitet. Es stehen verschiedene Komponenten zur Auswahl. Sie können parameterisiert werden, beispielsweise lassen sich die Weichen-Komponenten in Bezug auf ihre Bedingung, unter der sie den weiteren Fluss bestimmen, konfigurieren. Die Anpassungsmöglichkeiten entsprechen weitgehend den oben beschriebenen allgemeinen Anpassungsoperationen im Rahmen der komponentenbasierten Anpassbarkeit. Weiterhin bietet der Anpassungs-Client die Möglichkeit, zu den Filtern oder eingesetzten Teil-Komponenten Beschreibungen zu hinterlegen. Sie werden angezeigt, wenn Benutzer mit der Maus über die entsprechenden Komponenten fahren (sogenannte Quickinfos). Die Benutzer haben zusätzlich die Möglichkeit, das Verhalten von Filtern über einen Explorationsmodus zu testen. Dabei lässt sich per Drag'n'Drop eine Email aus einem Mailprogramm auf eine Komponente legen. Sie wird dann schrittweise durch die einzelnen Abschnitte der Komposition geführt. Die Benutzer können so genau nachvollziehen, wie ein Filter in Bezug auf eine bestimmte Mail funktionieren würde.

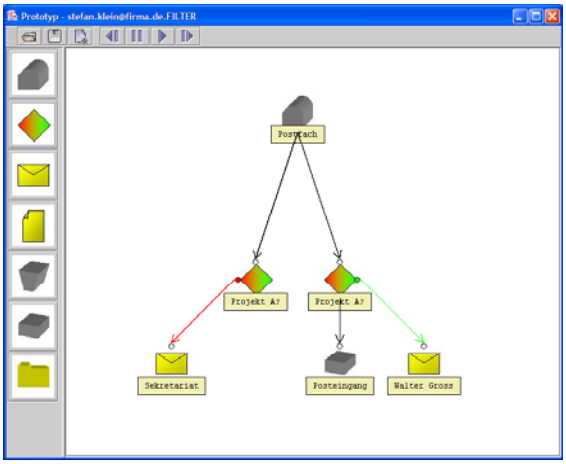


Abbildung 15: Email-Konfiguration

Für die weitere Untersuchung ist entscheidend, dass Email-Filter keine visuelle Darstellung im Rahmen eines Laufzeitsystems haben. Insofern ist zu erfragen, ob der Ansatz, Filter und ihr Zusammenspiel miteinander durch den Komponentenansatz zu veranschaulichen, die Verständlichkeit für Endbenutzer ausreichend deutlich machen kann.

Das Testszenarium – Aufgaben, Probanden, Ablauf

Die Testaufgaben sollten Erkenntnisse über den Prototypen und das durch diesen umgesetzte Konzept der visuellen Programmierung für Email-Filterregeln liefern. Zum einen war von Interesse, ob der im Rahmen der Aufgabe vorgestellte Email-Filter durch die vorhandenen Mechanismen semantisch durch Nutzer erfasst werden kann. Zum anderen sollte untersucht werden, inwiefern Nutzer in der Lage sind, Regeln an geänderte Bedürfnisse anzupassen. Zu diesem Zweck wurden Testaufgaben entworfen, die Regeln zur automatischen Behandlung von Emails widerspiegeln. Diese Regeln sind insofern realistisch, als dass sie Szenarien, wie sie durch eine vorhergegangene Nutzerbefragung gewonnen wurden, modellieren und ihre Anwendung in einem realen Arbeitsumfeld nachvollziehbar ist. Ausgangsbasis der Testaufgaben sind vorbereitete Email-Filterregeln, die durch die Nutzer erläutert und modifiziert werden sollten.

Der erste Teil der Tests bestand darin, vorhandene Filterkompositionen zu verstehen und damit verbundene Abläufe erläutern zu können. Anschließend sollte erprobt werden, inwieweit die Probanden in der Lage sind, Anpassungen an vorhandenen Kompositionen selbstständig vorzunehmen. Die Aufgaben bezogen sich darauf, Parameter in Weichen und Verbindungen zwischen Komponenten zu verändern und Komponenten auszutauschen. Dabei war der Schwierigkeitsgrad der einzelnen Aufgaben gestaffelt.

Die Auswahl der Versuchspersonen wurde repräsentativ entsprechend unterschiedlichen Kenntnisständen vorgenommen. Es wurden drei verschiedene Klassen von Nutzern (Anwender, Power-User und Administrator) gebildet. Aus jeder Klasse führte ein Proband die Testaufgaben mit dem Prototypen durch. Um eine qualitativ hochwertige Usability-Bewertung durchzuführen, ist weniger die Anzahl der Probanden als vielmehr die korrekte Anwendung der Thinking Aloud-Methode entscheidend. In diesem Fall lassen sich bereits mit 3 Testpersonen 80% der Schwachstellen identifizieren [Nielsen, 1992].

Der Typus „Anwender“ ist dadurch charakterisiert, dass er Grundkenntnisse in der Bedienung eines Betriebsystems und der für die Evaluierung wichtigen Anwendungen besitzt. Für den Test des Prototypen sind dafür Kenntnisse im Umgang mit MS Windows und ei-

nem beliebigen Email-Client erforderlich. Der Power-User hat darüber hinaus grundlegende Kenntnisse von formalen Konzepten wie Regeln und Flussdiagrammen und hat Erfahrung in deren Anwendung. Zusätzlich beherrscht er eine beliebige Programmiersprache. Der Administrator ist mit der Pflege eines Email-Systems vertraut. Er kennt den Aufwand, der mit der Einführung neuer Systeme verbunden ist, und kann die Schwierigkeiten bei der Betreuung der vorgestellten Software abschätzen. Den drei Nutzergruppen wurden jeweils die gleichen Testaufgaben gestellt.

Den Probanden wurden das Testszenarium, die Ziele des Tests und der Ablauf erläutert. War der Testperson die Thinking Aloud-Methode unbekannt, so wurde diese erklärt und anhand eines Beispiels vorgeführt. Die Funktionsweise des Prototypen wurde den Testpersonen kurz dargestellt, damit die gestellten Aufgaben gelöst werden konnten. Insbesondere wurden die Bedienelemente und die Bedeutung der verschiedenen Regelelemente erklärt. Nachdem die Testpersonen über das Evaluierungsverfahren unterrichtet waren, wurden die bereits beschriebenen Testaufgaben gestellt. Während der Bearbeitung der Testaufgaben wurden die beobachteten Probleme schriftlich festgehalten. Jedes Problem wurde dabei mit den Informationen über

6. die Situation, in der das Problem auftauchte,
7. das Bedienelement, das das Problem verursachte,
8. das Problem selbst (mit einer kurzen Beschreibung)
9. und die vermutliche Ursache des Problems

festgehalten.

War ein Problem derart gelagert, dass eine weitere Durchführung des Tests behindert wurde, wurde den Testpersonen weitergeholfen. Nachdem die Testaufgaben abgearbeitet waren, wurde ein semi-strukturiertes Interview durchgeführt, in dem es darum ging, die Erfahrungen festzuhalten, die der Proband bei der Arbeit mit dem Prototyp gemacht hatte. Außerdem ging es darum, Verbesserungsvorschläge sofort identifizieren zu können.

Ausgewählte Ergebnisse der Vorstudie

Die Auswertung der Befragungen und Benutzertests brachte verschiedene interessante Ergebnisse. Für die weitere Diskussion ist dabei weniger wichtig, welche Filterregeln als sinnvoll eingestuft wurden, welche Filterkomponenten fehlen, um das gemeinsame Arbeiten in einer virtuellen Organisation effizient zu unterstützen etc. Vielmehr sollen im Folgenden die Punkte erläutert werden, die sich mit der Bedienung einer komponentenbasierten Anpassungssprache beschäftigen.

- **Verständnis für vorhandene Kompositionen**: Die erste Testaufgabe beschäftigte sich damit, vorhandene Filterkompositionen zu erläutern. Dabei zeigte sich, dass keiner der Probanden damit Schwierigkeiten hatte. Die Bedeutungen und Funktionsweisen der teilweise parameterisierten Einzelkomponenten konnten erläutert werden. Das Zusammenspiel der Komponenten innerhalb der (hier immer baumartig verzweigten) Kompositionen wurde von allen Teilnehmern richtig erkannt und gut beschrieben.
- **Anpassung vorhandener und Konstruktion eigener Kompositionen**: Die zweite und dritte Aufgabe erforderten das selbstständige Entwickeln von Filtern, deren Funktionsweise mündlich beschrieben wurde. Hier zeigte sich, dass System-Administratoren und Probanden mit Programmiererfahrung damit kaum Probleme hatten. Beide waren in der Lage, die Aufgaben korrekt zu bearbeiten, wobei sich die Lösungen und auch die Herangehensweise im Detail unterschied. Nicht-programmiererfahrene Benutzer hatten mit dieser Aufgabe jedoch deutliche Probleme. Sie konnten die Aufgaben schließlich unter Nutzung des Explorationsmodus lösen. Dabei wurden die selbst entwickelten Filter sukzessive verfeinert, bis sie der gestellten

Aufgabe entsprachen. Trotzdem gaben alle Benutzer an, dass diese Art der visuellen Programmierung vom Ansatz her sehr vielversprechend sei.
Keine der Testpersonen nutzte von sich aus die Möglichkeit, den eigenen Kompositionen Beschreibungen hinzuzufügen, und das, obwohl diese bei der ersten Aufgabe als sehr hilfreich empfunden wurden.

Diskussion der Ergebnisse

Die Ergebnisse der Vorstudie zeigen, dass der Ansatz, komponentenbasierte Architekturen als Basis einer Anpassungssprache für Endbenutzer einzusetzen, von den Testpersonen grundsätzlich verstanden und positiv aufgenommen wurde. Es zeigt sich auch, dass trotz der vermeintlichen Einfachheit Unterstützungsmechanismen benötigt werden. In der Vorstudie wurden sowohl natürlich-sprachliche Erklärungen als auch ein Explorationsmodus bereitgestellt. Beide wurden besonders von den weniger erfahreneren Benutzern gerne und effizient eingesetzt. Weiterhin zeigte sich, dass die hier getesteten Benutzer jedoch die Möglichkeit der Beschreibung einer Komposition oder einer von ihnen parametrisierten Komponente nicht oder nur verhalten nutzten. Als Alternative böten sich automatisch generierte Beschreibungen an, die jedoch nur in einfachen Fällen aussagekräftig sind. Stiemerling et al. [Stiemerling et al., 2000] beschreiben, wie ein Zugriffskontrollsystem für Groupware diesen Mechanismus nutzen kann, um Benutzern zu erläutern, welche Zugriffsregeln für sie gelten und was sie bedeuten.

Wie schon oben (siehe Kapitel 3) angedeutet, eignen sich beide Techniken jedoch nur bedingt, um komplexere Kompositionen zu erläutern. Ein Explorationsmodus mag für ein Email-Filter-System, das nur aus wenigen im Vorfeld bekannten Komponententypen besteht, angemessen sein. Eine „beliebige“ Applikation lässt sich jedoch nur schwierig in ihrem Verhalten analysieren. Eine komplette Testumgebung ist nötig, in der die Komposition getestet werden kann. Weiterhin ist ein solcher Test auch nur dann aussagekräftig, wenn für alle kritischen Fälle geeignete Testdaten zur Verfügung stehen. Beispielsweise wären im Falle eines Email-Filters, der Nachrichten nach Eingangszeit (Vormittag vs. Nachmittag) zwei unterschiedlichen Personen zustellt, Test-Emails nur dann sinnvoll, wenn sie sich anhand dieses Kriteriums unterscheiden ließen. Auch natürlich-sprachliche Beschreibungen – speziell, wenn sie automatisch erzeugt werden – stoßen bei komplexeren Kompositionen an ihre Grenzen.

5.3.2 Evaluierung des TailorClient

Nachdem gezeigt wurde, dass die komponentenbasierten Anwendungen grundsätzlich von Anwendern verstanden werden können, soll nun der TailorClient in Bezug auf seine Handhabung geprüft werden.

Das Testszenarium – Aufgaben, Ablauf, Probanden

Zu diesem Zweck wurde ein Fragekatalog, wie er in [Nielsen, 1993] beschrieben wird, entwickelt. Er hat zum Ziel zu erfragen, wo Probleme bei der Bedienung des TailorClient auftreten. Im Rahmen der ca. 1,5 h dauernden Befragungen mussten Anpassungsaufgaben unterschiedlicher Art gelöst werden. Parallel dazu wurden die Fragen beantwortet.

Die Probandengruppe setzte sich aus zehn Personen zusammen. Sie ließ sich in zwei Nutzertypen einteilen: Computer-Experten mit Programmiererfahrung und Nutzer ohne Programmiererfahrung. Die erste Gruppe wurde gewählt, um auf Basis einer heuristischen Evaluierung durch Expertenbefragung (siehe [Nielsen, 1993]) mögliche Nutzungsprobleme aufzudecken. Dabei spielte die Erfahrung der Benutzer im Umgang mit Computersystemen

eine maßgebliche Rolle. Die zweite Teilgruppe „Nutzer“ diente im Rahmen dieser Evaluierung in erster Linie als Kontrollgruppe und sollte die Aussagen der „Experten“ prüfen.

Das Testszenarium sah vor, verschiedene Anpassungsaufgaben zu lösen und die Arbeit mit dem Prototypen zu erläutern. Nach der Einführung der Benutzer in Thema und Ziele der Studie waren zwei Aufgabenblöcke zu bearbeiten. Im ersten Block ging es darum, eine fiktive Komposition „der Computer“ (siehe Anhang C) zu bearbeiten. Der zweite Block beschäftigte sich mit einer realen Anwendung „GeoChat“ (siehe ebenfalls Anhang C). Die einzelnen Aufgaben sahen vor, bestimmte Aspekte an den Kompositionen zu erklären, zu verändern oder Funktionalität hinzuzufügen. Abschließend konnte noch einmal Kritik (positiv wie negativ) an der Gestaltung und Bedienung des Prototypen geäußert werden.

Ergebnisse der Interviews und Abgleich mit den von Nielsen beschriebenen Anforderungen

Im Folgenden werden nun die Ergebnisse der geführten Benutzertests und Interviews zusammengefasst und diskutiert. Dabei orientiert sich die Struktur an den in Kapitel 3 eingeführten Kategorien nach Nielsen. Da dort schon grundlegende Argumente für komponentenbasierte Architekturen im Allgemeinen und das FREEVOLVE-System im Speziellen genannt wurden, soll hier der Schwerpunkt auf der Betrachtung des TailorClient liegen.

- **Sichtbarkeit des Systemstatus**: Um ein System sinnvoll bedienen zu können, ist es unerlässlich, dass der aktuelle Systemstatus dargestellt wird. Im FREEVOLVE-System werden die verschiedenen Sichten, die gemeinsam den kompletten Systemstatus darstellen, ständig miteinander synchronisiert. Auf diese Weise haben Benutzer zu jedem Zeitpunkt eine Übersicht über den aktuellen Systemstatus. Diese Technik wurde von verschiedenen Testpersonen lobend erwähnt. Weiterhin wird im Komponenten-Explorer komprimiert die gesamte Applikation dargestellt. Nach der ersten Eingewöhnung kamen auch die weniger erfahreneren Benutzer gut mit dieser Ansicht zurecht.
  Um Benutzer nicht zu überfordern, sollten weniger wichtige Informationen ausgeblendet werden. Im TailorClient besteht die Möglichkeit, für die Anpassung einer Komposition nicht relevante Ports oder Parameter einer Komponente auszublenden („hide“).
  Einer der Experten merkte an, dass die im TailorClient verwendete Sprache überarbeitungsbedürftig sei. Sie ist nicht auf die Belange von Benutzer ausgerichtet, die wenig Erfahrung mit Software-Systemen haben. Hier besteht also noch Verbesserungsbedarf.
- **Erkennbarer Abgleich zwischen realer Welt und dem Computersystem**: Ein zu hohes Abstraktionsniveau stellt für viele Benutzer eine unüberwindbare Hürde beim Verstehen eines Computersystems dar. Aus diesem Grund sollten Systeme so gestaltet sein, dass Benutzer ihre Arbeitsaufgaben durch Wahl einer geeigneten Sprache im System wiederfinden, dass Module des Systems mit der realen Welt korrelieren etc. Der TailorClient bewegt sich hier auf einer Meta-Ebene, ist seine Aufgabe doch nicht, eine Arbeitsaufgabe zu unterstützen, sondern Software anzupassen, die dann als Werkzeug eingesetzt werden kann. Insofern ist die reale Welt in diesem Fall die komponierte Applikation zur Laufzeit. Hier hat sich (wie auch schon in der Vorstudie) gezeigt, dass der Ansatz, eine Applikation aus Komponenten zusammenzusetzen, durchaus geeignet ist, um den Zugang zur Anpassung zu erleichtern bzw. umgekehrt, ein Verständnis für die resultierende Applikation zu entwickeln. Hier zeigte sich, dass fast alle Benutzer sich verständlichere Namen (die die Funktionalität aus Sicht der Anwender beschreiben) gewünscht hätten. Schwierigkeiten bei der Auswahl geeigneter Komponenten ergaben sich lediglich dadurch, dass die Namen der Komponenten zu wenig aussagekräftig waren. Dieses Problem lässt sich im FREEVOLVE-System leicht beheben, da allen Komponenten textuelle Beschreibungen der Eigenschaften (CAT-Files) beigegeben sind.

Diese lassen sich verändern, so dass hier nur geeignetere Namen ausgewählt werden sollten.

- **Benutzerkontrolle und Freiheit**: Nielsen [Nielsen, 1994] sieht die Kontrolle, die Benutzer über die von ihnen im System zu bearbeitenden Objekte haben, als eine sehr wichtige Eigenschaft von Software. Im TailorClient sind den Benutzern nur durch die Typisierung von Ports und Parametern Grenzen gesetzt. So ist es nicht möglich, nichttypkonforme Ports miteinander zu verbinden. Auch die Eingabe von Werten, die nicht einem Parametertyp entsprechen, wird vom System unterbunden. Diese Einschränkungen waren aus der Benutzersicht während der Evaluierung eher hilfreich und wurden nicht als Einschränkung verstanden.
  Freiheiten haben Benutzer bei der Arbeit mit dem TailorClient nicht nur in Bezug auf die schon in Kapitel 2 erwähnten Möglichkeiten der komponentenbasierten Anpassbarkeit. Durch die unterschiedlichen Sichten ist es möglich, Anpassungen auf verschiedene Weisen vorzunehmen. Beispielsweise nutzten unerfahrenere Nutzer gerne die Sichten des Komponenten-Editors zum Verbinden von Komponenten (dessen Funktionsweise ausdrücklich lobend erwähnt wurde), wohingegen Experten verstärkt versuchten, Aufgaben ausschließlich mit Hilfe des Komponenten-Explorers zu lösen.
- **Konsistenz**: Wichtig für die Bedienbarkeit eines Computersystems ist Konsistenz in der Darstellung und Bedienung. Aus diesem Grund wurden im TailorClient ansichtenübergreifend durchgängige und einheitliche graphische Symbole für bestimmte Eigenschaften und Funktionalitäten des Systems verwendet. Weiterhin ist auch die Maussteuerung in allen Ansichten gleich, alle wichtigen Operationen sind über Kontextmenüs zu erreichen. In der Testphase hat sich gezeigt, dass alle Benutzer die Kontextmenüs zu schätzen lernten. Sie ermöglichen einen einfachen Zugriff auf mögliche und sinnvolle Operationen.
- **Fehlervermeidung**: Der hier vorgestellte TailorClient ist nicht mit expliziten Fehlervermeidungs- oder -korrekturmechanismen ausgestattet. Einzig die schon oben erwähnte Typprüfung und Kontextmenüs, die nicht zulässige Operationen grundsätzlich nicht zur Verfügung stellen, können der Fehlervermeidung zugerechnet werden. Trotzdem schätzten die Probanden gerade diese sehr einfachen Mechanismen. Sie schlossen Fehlbedienungen in vielen Fällen schon im Vorhinein aus.
- **Bestätigung statt Erinnerung**: Für die Bearbeitung komplexerer Aufgaben ist es wichtig, die bis dahin getroffenen Entscheidungen und die sich daraus ergebenen Konsequenzen im Auge zu behalten. Das System sollte dementsprechend so gestaltet sein, dass die Nutzer dabei unterstützt werden. Im Falle des TailorClient sind alle Änderungen sofort sichtbar. Zur Kontrolle können diese Änderungen bei Bedarf in die laufende Anwendung übernommen werden. Da der Status des Systems also immer vollständig dargestellt wird, ist es allen Benutzern jederzeit möglich, durchgeführte Änderungen zu erkennen.
- **Flexibilität und Effizienz**: Speziell für Benutzer, die bereits erste Erfahrungen mit einem Computersystem gesammelt haben, ist es wichtig, eine möglichst optimale Unterstützung für ihre Aufgaben zu bekommen. Hier zeigt sich, dass die unterschiedlichen Sichten im Anpassungsmodus gerade sehr hilfreich sein können. Während die weniger erfahreneren Probanden für jede Aufgabe die dafür optimierte Sicht wählten, nutzten einige der Experten im späteren Verlauf der Tests fast ausschließlich den Komponenten-Explorer, der alle Operationen zur Verfügung stellt. Auf diese Weise konnten sie einige Anpassungen sehr schnell vornehmen.

- **Ästhetisches und minimalistisches Design**: Ebenso wie das oben angesprochene konsistente Design und eine einheitliche Benutzerführung ist auch die Reduktion der Informationsbereitstellung auf das Wesentliche ein wichtiges Kriterium für die leichte Erlernbarkeit einer Software. Informationen, die die Benutzer verwirren oder von der eigentlichen Arbeitsaufgabe ablenken, sind nicht nur hinderlich, sondern erschweren den Zugang zur Software. Der TailorClient, der wie schon oben gezeigt, sehr viele Informationen auf wenig Raum darstellen muss, kann nicht als minimalistisch bezeichnet werden. Dennoch wurde versucht, der Komplexität gerecht zu werden und sie so weit wie möglich zu reduzieren. Eine Möglichkeit ist beispielsweise die schon oben erwähnte Eigenschaft, nicht benötigte Informationen auszublenden. Dies kann auch schon vor der ersten Nutzung durch einen Administrator geschehen, so dass diese Aufgabe nicht mehr den Benutzern überlassen wird und sie direkt zu Beginn ihrer Arbeit mit einer zu großen Informationsmenge konfrontiert werden.
- **Unterstützung der Benutzer bei der Erkennung und Korrektur von Fehlern**: Der TailorClient verfügt in der ersten Version über keinerlei Mechanismen zur Fehlerkorrektur. Sie werden erst im Folgenden integriert und dann erprobt (siehe Kapitel 6 bis 10). Schon in dieser Evaluierung zeigte sich jedoch, dass einfache Techniken wie ein Undo-Mechanismus von den Benutzern vermisst wurden. Solche Mechanismen erlauben es schnell, erkannte Fehler zu korrigieren.
- **Hilfe und Dokumentation**: Auch auf Hilfen und Dokumentationen wurde im ersten Prototypen verzichtet. Da die Benutzertests als Labortests ausgelegt waren, standen immer fachkundige Interview-Leiter zur Verfügung, so dass schriftliche Informationen zur Bedienung oder Funktionalität nicht benötigt wurden.

5.4 Fazit

Dieses Kapitel stellt den ersten Prototypen einer graphischen Anpassungsumgebung für FREEVOLVE, den TailorClient, vor. Nach einer eingehenden Anforderungsbeschreibung wird der TailorClient in Bezug auf Funktionalität und Darstellung beschrieben.

Eine Vorstudie zeigte, dass der grundlegende Ansatz, Komponenten visuell zu einer Applikation zu komponieren, geeignet ist, um Endbenutzern das Anpassen komplexer Applikationen zu ermöglichen. Weiterhin wurde auch der TailorClient selbst auf seine Funktionalität und Bedienbarkeit hin überprüft. Grundsätzlich war die Resonanz positiv. So konnten alle Probanden die ihnen gestellten Anpassungsaufgaben weitgehend selbstständig lösen. Schwächen zeigten sich vor allem in Bezug auf die hohe Komplexität und die sehr technisch orientierte Sprache in den Bedienelementen und bei der Beschreibung der Komponenten und ihrer Funktionalität. Dies lässt sich jedoch leicht beheben, da sowohl der TailorClient als auch die FREEVOLVE-Plattform über verschiedene auch für Administratoren leicht zugängliche Einstellungsfunktionen verfügt, die hier Änderungen zulassen.

Die Kritik bezüglich der hohen Komplexität, die auch durch die geforderte Mächtigkeit bei den Anpassungsmöglichkeiten induziert wird, lässt sich allerdings durch die Betrachtung der von den Experten erzielten Ergebnisse bei den Anpassungsaufgaben einschränken. Es zeigte sich, dass lediglich sehr unerfahrene Benutzer große Probleme mit der Bedienung des TailorClient hatten. Selbige hatten aber auch Schwierigkeiten, einige der Komponenten und ihre Funktionalität zu beschreiben. Hier zeigt sich, dass der TailorClient eher für durchschnittlich oder auch sehr erfahrene Benutzer ein geeignetes Werkzeug zur Anpassung darstellt. Betrachtet man aber allgemein die Entwicklung bezüglich der Erfahrung mit Computersystemen in den letzten fünf bis zehn Jahren, so wird offensichtlich, dass Kenntnisse im Bereich der IT deutlich zugenommen haben. So lässt sich prognostizieren, dass der

TailorClient für zukünftige Benutzer-Generationen ebenso wenig problematisch sein wird, wie die Nutzung einer Textverarbeitung oder einer Multimediadaten-Abspielsoftware für heutige.

Zusammenfassend lässt sich sagen, dass der erste Prototyp des TailorClient eine gute Basis für die Integration einer semi-automatischen Integritätsprüfung darstellt. Diese wird formal im folgenden Kapitel entwickelt. Im übernächsten Kapitel wird dann die zweite Version des TailorClient mit Integritätsprüfung beschrieben und diskutiert.

6 Ein Integritätskonzept für Komponentenarchitekturen – Anforderungen und Grundgedanken

In Kapitel 5 wurde eine Anpassungsumgebung für FREEVOLVE vorgestellt. Die Evaluierung hat verschiedene Schwachstellen offengelegt, wenn es darum geht, Benutzern das Anpassen von Software zu erleichtern. Einer der kritischen Punkte ist die mangelnde Fehlererkennung oder Fehlerkorrektur seitens des Systems und damit einhergehend eine aktive Führung der Benutzer beim Anpassen. In diesem Kapitel wird nun ein Konzept zur Integritätsprüfung auf komponentenbasierten Architekturen vorgestellt.

Im Bereich der Entwicklung eines Integritätskonzepts müssen im Vorfeld zwei unterschiedliche Sichtweisen geklärt werden: Die Benutzersicht, aus der sich Fragen der Art ergeben, welche Integritätsbedingungen als Unterstützung bei der Komposition sinnvoll sind, und die Komponentensicht, die die technischen Möglichkeiten für die Formulierung von Integritätsbedingungen untersucht.

Aus Perspektive der Benutzer sollte ein Integritätsmechanismus zum Ziel haben, Endanwendern das Anpassen ihrer Applikationen zu erleichtern und Verständnis für die Funktionalität von Einzelkomponenten und ihrem Zusammenspiel innerhalb einer Komposition zu verbessern. Aus dieser Herangehensweise ergeben sich verschiedene Forderungen an das Integritätskonzept. Sie leiten sich her aus den in Kapitel 2 diskutierten Eigenschaften an Komponentenarchitekturen, die in Kapitel 3 beschriebenen Anforderungen an eine Anpassungsschnittstelle und Unterstützungstechniken während der Anpassung und den in Kapitel 5 aufgezeigten Schwächen der aktuellen Anpassungsumgebung:

- **Die Betrachtung der Anpassungsoperationen und Unterstützung steht im Vordergrund**: Aus Sicht der Endanwender ist eine Anpassungstransaktion definiert als eine Sequenz von ausgeführten Anpassungsoperationen. Da diese sich, wie schon in Kapitel 2 gezeigt, dicht an den Eigenschaften des FLEXIBEANS-Komponentenmodells orientieren, können auch diese betrachtet werden. Durch die Operationen können die Zustände einer Komposition geändert werden. Damit lässt sich folgern, dass ausschließlich die Ausführung einer Operation einen Einfluss auf den Integritätszustand einer Komposition haben kann. Zu betrachten sind also
  - die Existenz von Komponenten innerhalb einer Komposition,
  - die Parameterbelegung der einzelnen Komponenten und
  - die Verbindungen der Komponenten untereinander.
- **Definierte Integritätsbedingungen sollten für Endanwender verständlich sein und das Verständnis für die Applikation erweitern**: Da es im Rahmen dieses Vorhabens primär darum geht, Benutzern das Anpassen ihrer Anwendungen dadurch zu erleichtern, dass sie selbst ein Verständnis für existierende Kompositionen und die Auswirkungen von Änderungen an ihnen verstehen, sollten die Integritätsbedingungen darauf abgestimmt werden. Sie dienen nicht vordringlich dazu, automatisch Korrekturen durchzuführen, sondern den Benutzern Probleme darzulegen und auf diese Weise wiederum Verständnis für die Komponenten, die Komposition und die Anpassungsoperationen zu erzeugen [Alda et al., 2002].
  Hierbei sind zwei Sichten zu betrachten: Zum einen müssen den Integritätsbedingungen zugrunde liegende Konzepte leicht zugänglich sein. Die Art der Regelbildung, ihr

Test und mögliche Korrekturen sollten für die Zielgruppe verständlich sein. Zum anderen sollten die Regeln selbst in ihrer Bedeutung klar werden.

- **Integritätsbedingungen sollten domänenabhängig formuliert werden, um den Anwendungsbezug zu erhöhen**: Die Beschreibungssprache für Integritätsregeln sollte es zulassen, flexibel Regeln im Anwendungskontext zu erweitern oder zu verändern. Auf diese Weise lässt sich eine bessere Integration in den Arbeitskontext der Anwender gewährleisten. Optimalerweise lassen sich Integrationsregeln speziell für bestimmte Anwendungstypen formulieren, so dass für eine komponentenbasierte Anwendung vom Typ A andere Integritätsbedingungen gelten als für eine weitere Applikation vom Typ B. Zum einen lassen sich die Integritätsbedingungen auf diese Weise näher auf den Anwendungskontext hin formulieren. Dies erleichtert den Zugang zur Anpassung, da ein stärkerer Anwendungsbezug speziell für weniger Programmiererfahrene den notwendigen Abgleich zwischen realer Welt und dem Computersystem erleichtert. Von dieser Vorstellung gehen beispielsweise auch Repenning und Summer [Repenning und Summer, 1995] aus, die eine visuelle Programmiersprache entwickelten, um den Bezug zur realen Welt möglichst unmittelbar zu schaffen. Zum anderen weisen Lewis und Olsen [Lewis und Olson, 1987]) darauf hin, dass es für Endbenutzer häufig schwierig ist, mit Low Level-Primitiven, die nicht an einen Anwendungskontext gebunden sind, zu programmieren. Mit Hilfe von domänenabhängigen Integritätsbedingungen lassen sich solche Low-Level-Primitive in den Kontext einer Anwendung stellen und so leichter verstehen.
- **Natürlichsprachliche Formulierung der Integritätsregeln**: Wie schon oben angesprochen dienen sowohl die Integritätsbedingungen als auch eine darauf basierende Prüfung der Komposition primär dem Erlernen der Anpassungssprache. Insofern ist es sinnvoll, Integrationsbedingungen so zu formulieren, dass sie für die anpassenden Benutzer als Hilfestellung dienen können. Die Regeln müssen dazu nicht notwendigerweise in natürlicher Sprache abgelegt sein. Ein Interpreter, der die interne Darstellung in eine für die Anwender verständlichere Form aufbereitet, bietet wahrscheinlich deutlich flexiblere Möglichkeiten.

Die zweite Perspektive, aus der sich ein Integritätskonzept erschließen lässt, betrachtet die Zusammenhänge aus der Sicht des Software-Engineering. Hier lässt sich die Frage nach einem semantischen Integritätskonzept wie folgt präzisieren:

- **Welche Integritätskontrollen bieten Laufzeitumgebung, Programmiersprache oder ein Entwicklungs-Framework von sich aus an?** –Schon bei der Integration neuer Komponenten in den Komponentenpool kann geprüft werden, ob diese der gewünschten Spezifikation entspricht. Auf diese Weise können schon durch erste Vorprüfungen grundsätzliche Fehler ausgeschlossen werden. Weiterhin lassen sich in komponentenbasierten Architekturen über die Definition der Schnittstellen erste Prüfungen vornehmen. So sollte eine Anpassungsumgebung die Typgleichheit von Schnittstellen prüfen, bevor zwei Komponenten miteinander verbunden werden. Diese Prüfung ist in den meisten Entwicklungsumgebungen integriert. Sie entspricht der Integrität im Bereich der Datenbanken, die sich automatisch durch die Definition des Datenbankschemas ergibt (siehe hierzu Kapitel 4).
- **Welche Informationen kann eine Komponente bereitstellen?** – Es sollte geklärt werden, welche Informationen mit wie hohem Extraaufwand bereitgestellt werden können. Während die Schnittstellentypisierung von den Entwicklern normalerweise in jedem Fall vorgenommen werden muss, sind weitere Informationen – wie die Abhängigkeit von Parametern innerhalb einer Komponente – nur mit Mehraufwand bereitzustellen. Hier müssen einfache Beschreibungsmöglichkeiten gefunden werden, um die

notwendigen Informationen ohne zu großen Aufwand erzeugen zu können. Solche Abhängigkeiten sind vergleichbar mit Constraints oder Triggers, die für eine Datenbank definiert werden. In ähnlicher Weise können auch Aktionen mit der Nicht-Einhaltung solcher Bedingungen verknüpft werden. Sie können von der Anzeige von Hilfstexten bis hin zu automatischen Korrekturfunktionen reichen (vgl. [Hutchins et al., 1986]).

- **Was kann eine Applikationsvorlage (Applikationsgruppe) beschreiben (Parameter)**? – Einige Integritätsbedingungen lassen sich nicht innerhalb einer Komponente definieren. Dazu zählen beispielsweise Abhängigkeiten zwischen Parametern unterschiedlicher Komponenten. Solche würden – wenn sie innerhalb einer Komponente formuliert würden – die Grundsätze der komponentenbasierten Software verletzen, die eine unabhängige Entwicklung vorsieht. Schablonen (Patterns, Templates), die eine Gruppe von Applikationen beschreiben, können solche Informationen bereitstellen. Auch hier ist die Nähe zu Constraints (s.o.) zu sehen. Sie werden aber nicht für eine Komponente sondern eine generische Komponentenmenge formuliert.
- **Was kann eine Applikationsvorlage (Applikationsgruppe) beschreiben (Verbindungen)**? – Weiterhin können Schablonen auch ein generisches Komponentennetz beschreiben, wie es ähnlich in abgewandelter Weise auch in CoPL [Birngruber, 2000a] definiert wird. Dieses Netz kann durch die benötigten Funktionalitäten – repräsentiert durch benötigte Schnittstellen einzelner Komponenten und Verbindungen zwischen ihnen – beschrieben werden. So lassen sich auch direkte Abhängigkeiten zwischen eingesetzten Komponenten formulieren und es können domänen-spezifische (oder auch applikations-spezifische) generische Komponentennetze mit bestimmten Beschränkungen formuliert werden. Die allgemeine Anpassungssprache wird so in einen Domänenkontext [Zdun, 2002] gesetzt und erleichtert so das Komponieren eigener Applikationen.

Wie schon oben ausgeführt, ist es sinnvoll, von den Anpassungsoperationen als auch von den „natürlichen“ und von den Benutzern durch Anpassungsoperationen veränderbaren Merkmalen einer Komponente bzw. Komposition auszugehen. Es zeigt sich, dass sich diese Perspektiven stark ähneln und ineinander überführen lassen:

- **Einfügen/Entfernen von Komponenten**: Das allgemeinste Unterscheidungsmerkmal zwischen zwei Komponenten ist die Semantik der Komponente selbst. Wird also eine Komponente ausgewählt und in eine Komposition eingefügt oder eine schon vorhandene entfernt, so ändert sich damit die Funktionalität der Komposition. Globale Bedingungen – in Applikationsvorlagen beschrieben – lassen sich hier dadurch festmachen, dass innerhalb einer Applikationsgruppe bestimmte Komponenten/Elemente von Komponentengruppen unbedingt benötigt werden, oder nicht enthalten sein dürfen.
- **Änderungen von Parametern innerhalb einer Komponente**: Ein weiteres Merkmal einer Komponente sind ihre zugänglichen und änderbaren Parameter. Durch sie lassen sich sowohl Funktionalität als auch Aussehen der Komponente anpassen. Integritätsbedingungen können hier für erlaubte Wertebereiche formuliert werden. Dies könnte – wie oben angedeutet – durch Festsetzen des Wertebereichs eines Parameters auf Source-Code-Ebene erfolgen. Auf diese Weise lassen sich die zulässigen Wertebereiche jedoch nicht mehr anwendungsfeld-spezifisch regulieren. Hier müsste dann schon zur Entwicklungszeit der Komponente eine Entscheidung getroffen werden, die bestimmt ist durch möglichst große Generizität einerseits und möglichst optimale Integration in spätere Anwendungsfelder andererseits. Dies lässt sich vermeiden, indem sich der Wertebereich eines Parameters einer Komponente durch veränderbare oder später zu formulierende Integritätsbedingungen einstellen lässt. Weiterhin sollte es möglich sein, Abhängigkeiten zwischen Parametern zu definieren. Solche Integritätsbedingungen

sollten zwingend explizit beschrieben werden, um die Unabhängigkeit einer Komponente von einer anderen zu erhalten. Solche globalen Betrachtungen über die Grenzen einer Komponente hinweg können in Applikationsvorlagen definiert werden. Globale Abhängigkeiten ergeben sich durch sinnvolle Verknüpfung von Parametern unterschiedlicher Komponenten.
Ein Beispiel für solche Abhängigkeiten ist der Wunsch, dass sich visuelle Komponenten innerhalb einer Bildschirmmaske unter Ergonomiegesichtspunkten nicht überlagern sollten. Dies kann gewährleitstet werden, indem man für alle visuellen Komponenten ein Regel-Set definiert, dass Größe und Position aller Komponenten prüft. Solche Bedingungen lassen sich mit den aus den Datenbanken bekannten Constraints definieren.

- **Bearbeiten von Verbindungen**: Die Ports einer Komponente ermöglichen ihr die Interaktion mit anderen. Schon in Kapitel 2.2 wurde das aus verschiedenen Ansätzen bestehende Interaktionskonzept der FLEXIBEANS vorgestellt. Neben der Formulierung von Bedingungen, dass ein Port (typgerecht[17]) verbunden sein muss, kann man darüber hinaus feinere Beschreibungen für Integritätsbedingungen formulieren. Eine genaue Analyse der Informationsflüsse innerhalb eines Komponentennetzes kann feststellen, welche Nachrichten von einer Komponente erzeugt verwertet werden. An einem einfachen Beispiel kann dies illustriert werden:

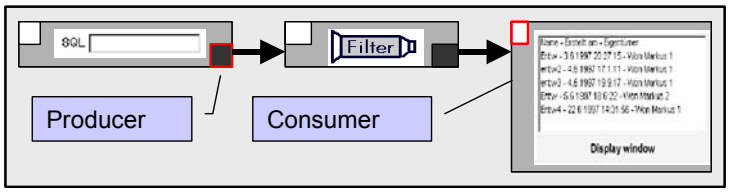


Abbildung 16: Komponenten als Konsumenten und Produzenten von Ereignissen

Abbildung 16 zeigt einen Ausschnitt eines Suchwerkzeugs. Eine Suchmaschine ist verbunden mit einem Filter (zur Aufbereitung der Suchergebnisse), der wiederum mit einer Ausgabekomponente verknüpft ist (von links nach rechts). Um als Integritätsbedingung zu formulieren, dass die von der Suchmaschine erzeugten Items angezeigt werden müssen, reicht es nicht festzulegen, dass die Schnittstelle der Suchmaschine mit einer weiteren Komponente verbunden ist. Dafür würde dann eine direkt verbundene Ausgabekomponente oder aber auch der Filter ausreichen, der nichts ausgibt. Ein Erweiterung der Integritätsbedingung darum, dass eine Filterkomponente immer sowohl am Eingang als auch am Ausgang mit anderen Komponenten verbunden sein sollte, trifft den Kern der semantischen Aussage nur unzureichend und ist dementsprechend schwierig für Benutzer nachvollziehbar.

[17] Dies kann durch Typisierung der Schnittstellen zur Entwicklungszeit gewährleistet werden.

Ein besserer Weg ist hier, den Ausgabeport der Suchmaschine als einen zu beschreiben, der unbedingt zu konsumierende Nachrichten produziert. Weiterhin kann die Ausgabekomponente als ein möglicher Konsument solcher Nachrichten definiert werden, wohingegen die Filterkomponente Nachrichten nur weiterleitet (Forwarder) und weder Konsument noch Produzent ist. Dieser Ansatz wird im Folgenden (siehe Abschnitt 8) als Event Flow Integrity (EFI) bezeichnet und detaillierter diskutiert. Zusammenfassend kann man die oben genannten Aspekte als Bewertungskriterien für die Güte einer Integritätsprüfung für komponentenbasierte Anpassbarkeit betrachten.

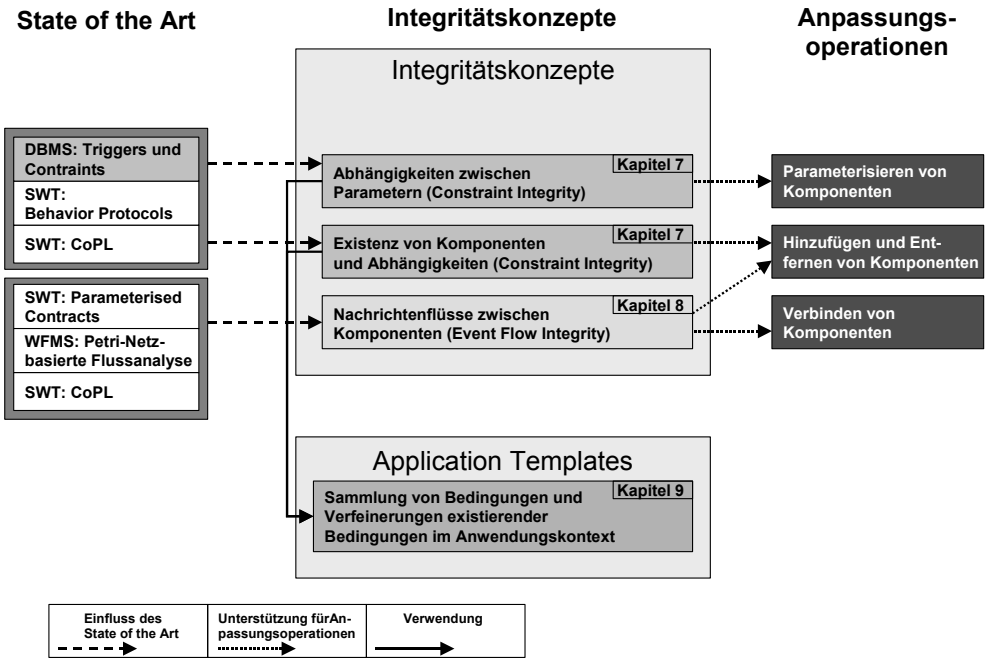


Abbildung 17: Erste Konzeptualisierung

In den folgenden Kapiteln sollen nun eine Constraint-basierte Integritätsprüfung und eine Überprüfung der Nachrichtenflüsse (Event Flow Integrity) (Kapitel 7 und 8) eingeführt und detailliert diskutiert werden. Beide werden dabei sowohl aus Sicht der Anwendung als auch aus Sicht der Architektur (Modell) diskutiert. Sie stellen dann die beiden Säulen für eine applikations-spezifische Integritätsprüfung dar. Dabei fließen die in Kapitel 4 diskutierten Integritätskonzepte wie in Abbildung 17 abgebildet ein. Auf der rechten Seite zeigt sich, dass mit Hilfe dieser beiden Konzepte alle möglichen Anpassungsoperationen, wie sie in Komponentenarchitekturen zur Verfügung stehen, unterstützt werden. Die Motivation aus Benutzungssicht wurde schon gegeben. Darauf aufbauend werden in Kapitel 9 beide Integritätskonzepte näher in den Anwendungskontext gesetzt. Dazu dienen die sogenannten Application Templates. Eine Evaluierung der Konzepte findet aus technischer Sicht in den einzelnen Abschnitten statt. Aus Benutzungssicht wird das gesamte Integritätskonzept in Kapitel 10 evaluiert.

7 Bedingungen an Kompositionen – Constraints und Restricted Solutions

Um Benutzern Hilfestellungen beim Anpassen zu geben, lassen sich die Möglichkeiten bei der Komposition durch Nebenbedingungen (Constraints) einschränken. Bedingungen sind hier Beschreibungen eines Zustands einer Komposition. Gilt für eine Komposition eine Bedingung, so deutet dies auf einen Fehler hin. Allgemeiner formuliert veranlasst das Eintreten einer Bedingung (Fehler) die Integritätsprüfung zu einer Aktion. Ein Beispiel für eine solche Bedingung ist „Wenn eine Komponente vom Typ A existiert, wird der Hilfstext ‚Komponente A muss immer parameterisiert werden' ausgegeben werden". In diesem Fall wacht die Integritätsprüfung bei jeder Operation „Komponente Einfügen", ob es sich dabei um eine entsprechende handelt. Tritt die Bedingung ein, so wird automatisch der oben beschriebene Hilfstext ausgegeben. Für die Überprüfung gibt es in diesem Fall zwei Möglichkeiten: Entweder es wird nach jeder durchgeführten Operation der Gesamtzustand der Komposition auf die Einhaltung von Bedingungen (in unserem Fall das Nicht-Zutreffen) geprüft und gegebenenfalls sich daran anschließende Aktionen durchgeführt oder die Prüfung erfolgt direkt auf den durchgeführten Operationen.

Die Bedingungen werden jeder einzelnen Komponente, für die sie gelten sollen, beigefügt. In der ersten Instanz werden sie von den Entwicklern der Komponente angelegt, um ihren Nutzungskontext feiner zu spezifizieren. Da sie in einer für Experten leicht verständlichen Sprache (XSemL, siehe unten) vorliegen, lassen sie sich von Administratoren anpassen. Auf diese Weise können Nutzern innerhalb einer Organisation besser auf den Anwendungskontext zugeschnittene Bedingungen zur Verfügung gestellt werden. Die Benutzer schließlich verwenden die Integritätsbeschreibungen implizit, wenn sie ihre Kompositionen auf Fehler überprüfen lassen.

Das Kapitel beschreibt zunächst die Funktionsweise von Constraints als Prüfmöglichkeit auf Kompositionen intuitiv. Eine Formalisierung wird im Folgenden gegeben. Weiterhin wird das Konzept der Solutions bzw. Restricted Solutions eingeführt. Sie erlauben automatische Korrekturen auf Basis der Integritätsbedingungen. Auch hier ist es notwenig, die Semantik formal zu definieren. Abschließend werden verschiedene Verfeinerungen, die zur leichteren Benutzbarkeit dieses Ansatzes führen, vorgestellt. Sie erweitern das Basiskonzept und wurden in der beschriebenen Form prototypisch implementiert.

7.1 Beschreibung von Bedingungen in XSemL (Constraints)

Die Beschreibung der einzelnen Constraints erfolgt in einer eigens dafür definierten Sprache. XSemL („Extensible Semantic Language") dient hier als eine Beschreibungssprache für semantische Informationen, die basierend auf den Anforderungen für die Integritätsprüfungen entwickelt wurde (für Details siehe [Krüger, 2002]). Eine Grammatik (DTD) dieser auf XML [W3C, 2000] basierenden Sprache findet sich ebenfalls dort. Abbildung 18 zeigt die mögliche Struktur einer semantischen Bedingung in XSemL. Auf die Bedeutung der einzelnen Tags wird in den folgenden Abschnitten genauer eingegangen.

```
<semantics>
<semantic component="Button" type="component">
<descriptions>[..] </descriptions>
<constraints> [..] </constraints>
<solutions> [..] </solutions>
<behaviors> [..] </behaviors>
</semantic>
</semantics>
```

Abbildung 18: XSemL-Code

Die Constraint-based Integrity stellt das Basiskonzept für die Kontrolle einer Komposition durch Bedingungsbeschreibungen dar. Zugrundegelegt wird dabei das aus den Datenbanken bekannte Konzept der Nebenbedingungen, das in den Bereich der komponentenbasierten Architekturen transferiert wird.[18]

Bedingungen können sich auf alle Eigenschaften einer Komponente beziehen. Nebenbedingungen zu Komponenten beschreiben die Komponentenmenge und deren mögliche hierarchische Ordnung. Beispielsweise muss eine Nebenbedingung alle benötigten Komponenten einer Anwendung beschreiben können. Nebenbedingungen zu Ports beschreiben Bedingungen, die Ports einer Komponente erfüllen sollten, beispielsweise mit welcher Komponente ein Port verbunden sein sollte. Weiterhin sind auch Nebenbedingungen für Parameter einer Komponente zu betrachten. Durch solche Bedingungen lassen sich die Wertebereiche verfeinern oder Abhängigkeiten zwischen Parametern unterschiedlicher Komponenten definieren.

Wird eine der Bedingungen verletzt, so wird eine Fehlermeldung erzeugt und durch die Anpassungsschnittstelle angezeigt. Abbildung 19 zeigt eine mögliche Nebenbedingung in XSemL-Format. Hier wird beschrieben, dass der Parameter „X-Pos“ der Komponente zwischen 1 und 800 liegen sollte. Weiterhin wird durch den „Errorlevel“ der Schweregrad des Fehlers in fünf Stufen festgelegt. Schließlich wird die Fehlermeldung definiert.

```
<constraint type=”param” name="X-Pos"
   operation="between" value="1 800"
   errorlevel="minor">
        Die X-Position dieser Komponente sollte zwischen 1 und 800 liegen.
</constraint>
```

Abbildung 19: Codebeispiel für eine Nebenbedingung

Würde also während des Anpassungsvorgangs der Parameter „X-Pos“ dieser Komponente auf einen Wert außerhalb des Bereichs gesetzt werden (z. B. 900), so hätte dies zur Folge, dass die Integritätsprüfung dies als Fehler erkennen würde. In Abhängigkeit von Schweregrad und der Konfiguration des TailorClient (siehe Kapitel 6) würde die Fehlermeldung angezeigt werden. Der Benutzer wird aufgefordert, diesen Fehler zu korrigieren.

7.2 Formale Beschreibung der Constraints

Für die formale Beschreibung von Kompositionen wird ein Modell benötigt, mit dem es möglich wird, sowohl die Berechenbarkeit von Funktionen darzustellen als auch alle möglichen Operationen, die ein Computer ausführen kann, zu modellieren. Den ersten Grundstein in diese Richtung legte Turing [Turing, 1936] mit der Definition der Turing-Maschinen. Diese formale Beschreibung von Maschinen, die bei gegebenem Imput einen Output berechnen, galt lange als theoretische Basis. Sie lässt jedoch verschiedene Aspekte

[18] Diese Transfermöglichkeit wurde in 4 diskutiert.

moderner Computer-Systeme unberücksichtigt. Vor allem die Interaktion von mehreren Maschinen oder einer Maschine mit ihrer Umwelt lassen sich nicht modellieren [Leeuwen und Wiedermann, 2000], [Wegner, 1996], [Wegner und Goldin, 2003]. Speziell diesen Aspekt stellen die sogenannten Interaction Machines [Leeuwen und Wiedermann, 2000] in den Vordergrund der Betrachtung. Für die formale Beschreibung von Kompositionen sind Kalküle (z. B. CCS [Milner, 1980], CSP [Hoare, 1985], π-Kalkül [Milner et al., 1992]) besonders geeignet. Stiemerling [Stiemerling, 2000] greift in seiner Beschreibung des FLEXIBEANS-Komponenten-Modell auf das Data Spaces Kalkül [Cremers und Hibbard, 1978] zurück. Eine Komponente und auch die Komposition mehrerer Komponenten lässt sich formal als Data Space betrachten. Für die weitere Diskussion ist diese Betrachtung nicht notwendig und dient hier nur dem grundlegenden Verständnis. Stattdessen werden erweiterte Definitionen für Komponenten verwendet (für Details siehe [Stiemerling, 2000]).

> **Definition Data Space**: *(X,F,p)* ist ein Data Space, genau dann wenn *(X,p)* ein Transitionssystem ist. Dabei ist X eine (möglicherweise unendliche) Menge von Zuständen und $p \subseteq X \times X$. F ist eine Informationsstruktur auf X, z. B. eine (möglicherweise unendliche) Menge von totalen Funktionen über X und beliebigen Wertebereichen. Weiterhin gilt:
>
> (1) $\forall_{x,y \in X}(\forall_{f \in F} : f(x) = f(y)) \Rightarrow x = y$ (Vollständigkeit)
>
> (2) $\forall_{e \in F \to X}\exists_{x \in X}\forall_{f \in F} : f(x) = f(e(f)) \Rightarrow x = y$ (Orthogonalität)

Zur Notation: $F \to X$ beschreibt die Menge aller totalen Funktionen, die jeweils eine Funktion f auf einen Zustand in X abbilden. Damit besagt Bedingung (2), dass die Funktionen in F unabhängig voneinander sind.

Auf Basis dieser Definition lässt sich eine Komponente als Data Space auffassen. Dabei wird der Zustand der Komponente (ausgedrückt durch den Zustand, den Variablen beschreiben) durch die Menge der Funktionen in F in Abhängigkeit vom Informationsraum X definiert. Transitionen (Zustandübergänge) werden beschrieben durch P.

Weiterhin wird in Stiemerling [Stiemerling, 2000] ausgeführt, dass die Zusammenführung zweier Data Spaces (Komposition) erneut einen Data Space darstellt. Ebenso lassen sich Teile herauslesen, ohne dass die definierenden Eigenschaften verletzt werden. Für die hier geführte Diskussion ist diese Definition und die damit verbundenen Eigenschaften ausreichend. Da Komponenten im Zusammenhang dieser Arbeit als atomar betrachtet werden, ist die Betrachtung dergleichen lediglich von außen sinnvoll. Veränderungen an den (atomaren) Komponenten selbst werden nicht vorgenommen.

Definition Einfache Komponente: Eine einfache Komponente C ist ein Tupel (n,D,P,A) mit

$n \in N_{components}$.	Name der Komponente
D ist ein Data Space (X,F,p).	Informationsstruktur der Komponente
$P \subseteq N_{ports} \times N_{porttypes} \times \{required,provided\} \times \wp(F)$	Hierbei ist $\wp(F)$ die Potenzmenge. Eine Komponente verfügt über eine Menge von Ports, die eindeutige Namen und Typen haben. Außerdem ist festgelegt, ob es sich um Eingangs- oder Ausgangsports handelt. Als Informationsstruktur für einen Port dient eine Funktion $f \in F$.
$A \subseteq N_{parameters} \times F$	A ist die Menge aller Parameternamen und deren korrespondierenden Informationsstrukturen innerhalb des Data Space.

Weiterhin gelten folgende Bedingungen:

a. Portnamen sind eindeutig und ihre Informationsstrukturen überlappen sich nicht

b. Parameternamen sind eindeutig und verweisen auf einen Teil der Informationsstruktur der Komponente und überlappen sich nicht mit denen von anderen Parametern oder Ports

c. Die Informationsstruktur eines Ports ist nicht leer

$N_{components}$, $N_{instances}$, N_{ports}, $N_{porttypes}$ *und* $N_{parameters}$ sind in diesen Zusammenhang Mengen eindeutiger Namen. Sie können als Zeichenketten betrachtet werden. Mit Hilfe dieser Definition lässt sich eine Komponente als ein Data Space mit zusätzlichen Eigenschaften beschreiben. Diese zusätzlichen Eigenschaften stellen keine funktionale Erweiterung dar. Sie dienen vielmehr der eindeutigen Beschreibung einer Komponente durch Vergabe von Namen.

Definition Komponenten-Set: Ein Komponenten-Set CS ist eine nicht-leere Menge von einfachen Komponenten C_i mit $C_i = (n_i,D_i,P_i,A_i)$, für die folgende Bedingungen gelten:

a. $n_i = n_j \Rightarrow i=j$. Die Komponentennamen sind eindeutig innerhalb von CS.

b. Zwei Ports des selben Typs haben die gleichen Informationsstrukturen. Es lässt sich eine isomorphe Abbildung f zwischen beiden Informationsstrukturen definieren.

Ein Komponenten-Set ist also eine Menge von Komponenten, die sich eindeutig unterscheiden lassen. Weiterhin gilt innerhalb dieses Komponenten-Set, dass Ports mit gleichem Namen auch gleiche Informationsstrukturen haben, so dass sie miteinander verbunden werden können.

Definition Komplexe Komponente: Eine komplexe Komponente C ist ein Tupel (n,P,A,S,B_1,B_2,B_3) mit

$n \in N_{components}$	Eine komplexe Komponente hat einen eindeutigen Namen.
$P \subseteq N_{ports} \times N_{porttypes} \times \{required,provided\}$.	Eine komplexe Komponente verfügt über Ports, die typisiert und gerichtet sind.
$A \subseteq N_{parameters} \times R$	Alle Parameter der komplexen Komponente haben einen Informationsraum, der durch R (die Menge aller Wertebereiche der Funktionen F der einzelnen Data Spaces der enthaltenen einfachen Komponenten) definiert ist..
$S \subseteq N_{components} \times N_{instances} \times \wp(N_{parameters} \times V)$	S definiert den gemeinsamen Informationsraum in C. Dabei ist V die Vereinigung aller Elemente von R.
$B_1 \subseteq N_{instance} \times N_{ports} \times N_{instance} \times N_{ports}$	B_1 beschreibt die Bindungen zwischen enthaltenen Komponenten innerhalb der komplexen Komponente.
$B_2 \subseteq N_{ports} \times N_{instances} \times N_{ports}$	B_2 beschreibt Bindungen zwischen Ports der komplexen Komponente und den in ihr enthaltenen.
$B_3 \subseteq N_{parameters} \times N_{instances} \times N_{parameters}$	B_3 beschreibt die gebundenen Parameter.

Weiterhin gelten folgende Bedingungen:

a. Komponenteninstanzen haben eindeutige Namen.

b. Die Portnamen sind eindeutig.

c. Die Parameternamen sind eindeutig.

d. Die referentielle Integrität aller Subkomponenten ist gesichert.

e. Die referentielle Integrität der Portverbindungen ist gesichert unter Berücksichtigung der Instanzen und Portnamen.

f. Die referentielle Integrität der Parameter ist gesichert unter Berücksichtigung der Instanzen und Parameternamen.

g. Es gibt keine Konflikte zwischen den gebundenen Parametern und den gegebenen Parametern der Subkomponente.

h. Ein Port einer Subkomponente ist entweder an eine andere Subkomponente gebunden oder an einen Port der komplexen Komponente.

i. Parameter einer Subkomponente können maximal einmal gebunden werden.

j. Ports einer Subkomponente können maximal einmal an einen Port der komplexen Komponente gebunden sein.

Mit Hilfe der Definition für eine komplexe Komponente lässt sich sichern, dass komplexe (auch zusammengefasste oder abstrakte) Komponenten innerhalb des gegebenen Namensraums eindeutig bleiben. Auch die in ihr enthaltenen Komponenten respektieren diese Eigenschaft. Weiterhin werden in einer komplexen Komponente Verbindungen zwischen Komponenten innerhalb der komplexen Komponente beschrieben wie auch Bindungen zwischen Ports der komplexen Komponente und den in ihr enthaltenen. Gleiches gilt für Parameter, die sich auf diese Weise binden lassen. Entscheidend für die Erhaltung der Funktionalität der enthaltenen Komponenten ist, dass ihre Informationsräume (beschrieben durch Data Spaces) nicht verändert werden.

Definition Komponentensystem: Ein Komponentensystem ist ein Tupel *(r,H,C)* mit

$r \in N_{components}$

H	ist eine endliche, nicht-leere Menge komplexer Komponenten
C	ist eine Komponenten-Set

Des weiteren gelten folgende Bedingungen:

a. Alle zusammengesetzten und einfachen Komponenten haben eindeutige Namen.

b. Die referentielle Integrität der Subkomponenten ist gesichert.

c. Die referentielle Integrität aller Ports ist gesichert.

d. Die referentielle Integrität aller Parameter ist gesichert.

e. Die Typkompatibilität und Polarität aller Subkomponenten ist gesichert.

f. Die Typkompatibilität aller Parameter ist gesichert.

g. Die Typkompatibilität und Polarität aller Bindungen ist gesichert.

h. Die Typkompatibilität aller Parameterisierungen der Subkomponenten ist gesichert.

i. Es gibt keine Zyklen oder Selbstreferenzierungen.

j. Es existiert eine Wurzelkomponente (*s_component*), die nicht Subkomponente einer anderen Komponente ist.

k. Referentielle Integrität aller Subkomponenten ist gesichert.

Ein Komponentensystem beschreibt somit eine funktionsfähige Anwendung. Sie hat einen Namen r und eine Menge an abstrakten Komponenten, die andere Komponenten enthalten. Da Kreisschlüsse (Komponenten, die sich selbst enthalten) nach der Definition für komplexe Komponenten unzulässig sind, lässt sich ein Komponenten-Set nach noch zu definierenden Regeln instanziieren. Die Funktionalität des Komponentensystems ergibt sich zum einen durch C, in dem die atomaren Komponenten beschrieben werden. Dabei ist die Semantik jeder einzelnen Komponente durch ihren Data Space definiert. Zum anderen beschreibt die Menge H die erforderlichen Verbindungen zwischen den Komponenten.

In Stiemerling [Stiemerling, 2000] wird weiterhin der Instanziierungsprozess genau untersucht. Dabei wird der Instanziierungsvorgang durch einen Operator, der auf ein Komponentensystem angesetzt wird, eingeführt. Dies führt zu einem instanziierten Komponentensystem, dass schließlich die lauffähige Anwendung (im Gegensatz zu einem Anwendungsplan) darstellt.

Für die weitere Betrachtung ist jedoch nicht notwendig, ein Komponentensystem zu instanziieren, da sich alle Integritätsprüfungen auf den Applikationsplänen (CAT) oder den korrespondierenden Komponentensystemen durchführen lassen.

Eine Bedingung, wie sie weiter oben eingeführt wurde, bezieht sich also auf ein Komponentensystem und lässt sich als eine Funktion darstellen. Dabei sind unterschiedliche Funktionen für verschiedene Arten von Bedingungen zuständig.

> **Definition Regelfunktion**: Eine Regelfunktion ist eine Funktion $r: CAS \times P \rightarrow \{wahr,falsch\}$, mit P Parameterliste. Dabei wird auf einem Komponentensystem $C \in CAS$ eine Prüfung entsprechend der Funktion und der dazugehörigen Parameter durchgeführt. CAS ist die Vereinigung der Mengen aller möglicher Komponentensysteme.

Im Folgenden werden verschiedene Regelfunktionen beschrieben. Sie beziehen sich auf die Untersuchung jeweils bestimmter Aspekte eines Komponentensystems. Durch die eindeutige Namensgebung innerhalb eines Komponentensystems ist dabei der Zugriff auf beliebige seiner Teilaspekte möglich.

Betrachtung der Existenz von Komponenten

Die Betrachtung der Verhältnisse zwischen Super- und Subkomponenten ist grundlegend für das Verhalten einer Komposition. Insofern sind Anfragen zu formulieren, die eine Prüfung auf solche Eigenschaften zulassen:

$$hasComponent: CAS \times (N_{\text{components}} \times N_{\text{components}}) \rightarrow \{wahr, falsch\}$$

überprüft, ob eine komplexe Komponente innerhalb eines Komponentensystems eine weitere als Subkomponente enthält. Dabei ist *hasComponent(...)* definiert wie folgt:

$$hasComponent(C,Super,Sub) := \begin{cases} wahr \Leftrightarrow \exists_{(Super,P,A,S,B1,B2,B3) \in H} \exists_{(n1,n2,Param) \in S}: n_1 = Sub \\ falsch\ sonst \end{cases}$$

mit $C=(r,H,C) \in CAS$ und CAS die Menge aller möglichen Komponentensysteme. Da der Namensraum innerhalb eines Komponentensystems eindeutig ist und seine Integrität syntaktisch durch die Definition gegeben ist, reicht eine Überprüfung auf Basis der Namenszuordnungen aus.

Betrachtung der Existenz von Parametern und Verbindungen

Auch Parameter lassen sich bzgl. der ihnen zugewiesenen Werte überprüfen. Ein Beispiel gibt folgende Regelfunktion:

$$isSet: CAS \times (N_{ins\,\text{tan}\,ces} \times N_{parameters}) \rightarrow \{wahr,falsch\}$$

überprüft, ob ein Parameter definiert wurde. Falls dies nicht der Fall ist, verweist der Parameter auf Null.

$$isSet(C,CName,PName) := \begin{cases} wahr, \text{falls } isComponent(C,CName) \\ \quad \wedge\, getParameter(C,CName,PName) \neq NULL \\ falsch\ sonst \end{cases}$$

Die hier verwendete Funktion *isComponent(...)* liefert wahr zurück, wenn *CName* auf eine instanziierte Komponente verweist. *getParameter(...)* liefert dann den Wert des spezifizierten Parameters. Beide Hilfsfunktionen sind in Anhang D definiert. Weitere Funktionen, mit Hilfe derer sich alle Eigenschaften einer Komposition untersuchen lassen, sind in ähnlicher Weise ebenso in Anhang D definiert. Aus Gründen der Übersichtlichkeit wurde hier auf eine weitere Beschreibung verzichtet. Folgende Tabelle gibt einen Überblick über die definierten Funktionen und ihre Semantik.

Bedingungs-ebene	Teilbedingung (Operator)	Parameterlisten	Beschreibung
Komponente	hasComponent	Typname einer Unterkomponente	Komponentenmenge enthält Komponenten
Parameter	isSet	[CompName, ParamName]	Parameter wurde gesetzt
	<,<=,=,!=,>=,>	[CompName], [ParamName], [Wert] oder [CompName], [ParamName], [CompName], [ParamName]	Bedingungen an den Wertebereich (für Zahlwerte)
	equals, startsWith, endsWith, length, contains	[CompName], [ParamName], [Wert] oder [CompName], [ParamName], [CompName], [ParamName]	Bedingungen an den Wertebereich (für Zeichenketten)
Ports	hasPort	[CompName], [PortName]	Port ist vorhanden
	isBound	[CompName], [PortName]	Port ist verbunden
	isBoundAbstract	[CompName], [PortName]	Port ist an höhere Hierarchieebene gebunden
	isBoundTo	[CompName], PortName], [CompName], [PortName]	Zwei Ports sind miteinander verbunden (auf gleicher Hierarchiestufe)
Hochreichen von Ports und Parametern	IsBoundUpParam	[CompName], [PortName],[CompName]	Parameter ist mit Parameter in der Superkomponente verbunden
	IsBoundUpPort	[CompName], [PortName]	Port ist mit Port in der Superkomponente verbunden

Tabelle 9: Bedingungen an Kompositionen

Bisher wurde die Gültigkeit der Parameter lediglich informell eingeführt und betrachtet. Wichtig für die Ausführbarkeit einer Regelfunktion ist jedoch, dass die Parameterliste der Definition der Funktion entspricht. Damit ergibt sich folgende Definition.

Definition Gültigkeit der Parameter und ausführbare Regelfunktion: Eine Regelfunktion *r* und eine Parameterliste *P*, dann ist das *Tupel (r,p)* eine ausführbare Regelfunktion genau dann, wenn *paramTest(r,p) = wahr*. Dabei ist *paramTest* definiert als

$$paramTest : R \times PS \rightarrow \{wahr, falsch\}.$$

mit *R* Menge aller Regelfunktionen und *PS* Menge aller möglichen Parameterlisten, *PS* $\subseteq$ *{String, Integer, Float}**.

Eine ausführbare Regelfunktion ist demnach eine Regelfunktion inklusive dazugehöriger und zur Regelfunktion passender Parameter. Sie kann auf einem Komponentensystem angewendet werden, ist jedoch grundsätzlich unabhängig von ihm.

Definition Regelsystem und Ausführung: Ein Regelsystem *RS* ist eine Menge von ausführbaren Regelfunktionen. Weiterhin ist die Funktion *constraintTest: CAS* $\times$ $\wp$*(RS)* $\rightarrow$ *{wahr,falsch}* definiert mit

$$\text{contraintTest(C,RS)} := \begin{cases} wahr, \text{falls } \forall_{(r,P) \in RS} : r(C,P) = wahr \\ falsch \text{ sonst} \end{cases}.$$

Dabei ist $\wp$(RS) die Menge aller möglichen Regelsysteme (inkl. der leeren Menge), *CAS* ist die Menge aller Komponentensysteme.

Die Funktion *constraintTest(..)* definiert so eindeutig, wann ein Test positiv in Bezug zu einem gegebenen Komponentensystem und einem Regelsystem gewertet wird. Liefert die Funktion *falsch* zurück, so ist das Komponentensystem unter Berücksichtigung der Regelmenge nicht integer, hat also einen semantischen Fehler.

7.3 Solution Integrity

Eine Erweiterung der Constraints-based Integrity ist die Solution Integrity. Sie erweitert das Konzept der Fehleridentifizierung um eine Fehlerkorrektur. Grundlage dieses Ansatzes ist die Tatsache, dass sich eine Komposition K, wie sie in [Stiemerling, 2000] definiert wurde, durch verschiedene Transformationen in eine Komposition K' überführen lässt, ohne die an eine Komposition gestellten Bedingungen zu verletzen. Dabei sollen lediglich Operationen verwendet werden, die eben diese Transformations-Eigenschaft besitzen

Häufig ist über die Mitteilung, dass ein Fehler vorliegt, hinaus wünschenswert, Hilfestellungen zu bekommen, wie sich dieser Fehler beheben lässt. Diese Aktionen können dann automatisch durchgeführt werden oder durch weitere Benutzerinteraktion gesteuert werden.

Die Solution Integrity basiert auf bedingten Anweisungen (if-then-else), die in vielen Programmiersprachen als Kontrollstruktur verwendet werden. Für die if-Bedingung werden die Nebenbedingungen der Constraint-based Integrity verwendet. Im then- und else-Teil werden Aktionen beschrieben, die ausgeführt werden sollen. Diese Aktionen können Anpassungsoperationen, aber auch Operationen zur Nutzerinteraktion sein.

In Abbildung 20 zeigt ein Beispiel für eine Solution-Regel. Eine Nebenbedingung zu einem Parameter „X-Pos" wird in Zeile 3 definiert. Falls bei einer Prüfung diese Bedingung einen Fehler erzeugt, werden Aktionen des then-Teils als Lösung angeboten. Hier ist eine Aktion zum Setzen des richtigen Wertes des Parameters angegeben. Die Anpassungsaktionen, wie sie in Tabelle 10 dargestellt werden, verändern die Komposition. In Kapitel 10 wird detail-

liert beschrieben, wie hier eine Nutzerinteraktion erfolgt, um den gefundenen Fehler zu korrigieren.

```
<if>
   <constraints>
        <constraint type="param" operation="=" name="X-Pos" value="5"
             description="X-Position muss 5 sein! "/>
   </constraints>
   <then>
        <action command="setparam" reason="X-Position auf ‚5' setzen">
             <param name="name" value="X-Pos"/>
             <param name="value" value="5"/>
        </action>
   </then>
</if>
```

Abbildung 20: Beispiel einer Solution-Regel in XSemL

In vielen Fällen lassen sich Lösungen ermitteln, indem man den Bedingungsteil analysiert. Dies kann auch automatisch geschehen. Sinnvoll ist dies vor allem dann, wenn die Korrektur nicht automatisch erfolgt, sondern in einer Benutzerinteraktion mündet. Neben der technischen Integritätsprüfung stellt also weiterhin der anpassende Nutzer die letzte Prüfungsinstanz dar. Zu der in Abbildung 20 vorgestellten Regel ist beispielsweise ein automatisch generierter Korrekturvorschlag leicht zu realisieren (Der Parameter X-Pos wird auf 5 gesetzt). Abbildung 21 zeigt die Abhängigkeiten zwischen Bedingungen und daraus ableitbaren Lösungsoperationen.

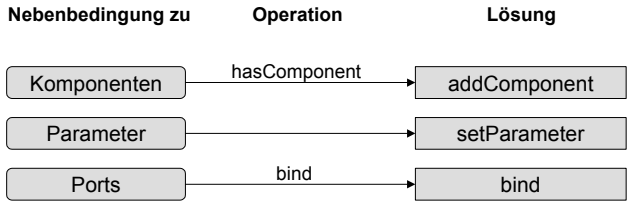


Abbildung 21: Abhängigkeiten zwischen Bedingungen und Lösungs-Aktionen

Aktionen und deren Ausführbarkeit

Bisher wurde anschaulich beschrieben, wie sich Solutions einsetzen lassen. Da Aktionen beschrieben werden können, die Änderungen an der Komposition bewirken, sind diese in Bezug auf ihre Zulässigkeit näher zu untersuchen. Hier stellen sich vor allen Dingen zwei Fragen:

1. Ist eine Komposition nach Ausführung einer Aktion weiterhin eine gültige Komposition?
2. Wie lässt sich der Zustand nach der Ausführung einer Aktion beschreiben?

Die erste Frage führt dazu, dass die möglichen (bereitgestellten Operationen) untersucht werden müssen. Sie lassen sich – wie Tabelle 10 zeigt – unterteilen in Operationen, die Veränderungen an der Komposition bewirken (oben), und anderen (unten). Für die weitere Diskussion sind die Operationen „Tip" und „Dialog" nicht weiter relevant. Sie geben lediglich Hinweise, die entweder der Operation beigefügt wurden oder automatisch generiert werden. Interessanter sind alle weiteren Operationen. Sie verändern die zu betrachtende Komposition. Allerdings korrespondieren sie mit den in [Stiemerling, 2000] definierten Anpassungsoperationen, auf die über das in FREEVOLVE bereitgestellte API zugegriffen werden kann. Dort wurden sie sprachunabhängig eingeführt und ihre Semantik in dem zu Beginn des Kapitels beschriebenen Modell definiert.

Aktion	Parameter	Beschreibung	Operation (nach Stiemerling)
addComponent	Name, Type	Fügt der Komponente (für die die Bedingung gilt) eine Sub-Komponente vom Typ "type" mit dem Namen "name" hinzu.	**addSubComponent**(type, name)
removeComponent	Name	Entfernt die Sub-Komponente „name" von dieser Komponente.	**removeSubComponent**(name)
renameComponent	OldName, NewName	Benennt die Sub-Komponente „oldname" in „newname" um.	Hier wird lediglich der Name der Komponente verändert, der Typ bleibt dabei ebenso unverändert wie auch die Bindungen und Parameter.
addParameter			**addParam**(name, type);
removeParameter			**deleteParam**(name)
addPort	Name, Interface, Polarity, Protocol	Fügt der Komponente einen Port hinzu.	**addPort**(name, type, polarity, protocol);
removePort	PortName	Entfernt einen Port.	**deletePort**(name);
bindPort	CompName1, Port1, CompName2, Port2	Verbindet zwei Sub-Komponenten miteinander.	**addPortBinding**(portname1, subcomponentname1, portname2, subcomponentname2) bzw. **addPortBinding**(portname1, subcomponentname1, portname2, subcomponentname2);
unbindPort	CompName1, Port1, CompName2, Port2	Entfernt eine Verbindung. zwischen zwei Sub-Komponenten.	**deletePortBinding**(portname1, subcomponentname1, portname2, subcomponentname2) bzw. **deletePortBinding**(portname1, subcomponentname1, portname2, subcomponentname2);
setParam	Name, Value	Setzt einen Parameter.	**setParam**(subcomponentname, parametername, value);
dialog	Title, Text	Erstellt einen Dialog.	
tip	-	Keine Aktion definiert. Hier wird automatisch auf Basis des Fehlers eine Fehlermeldung generiert.	

Tabelle 10: Aktionen in Solutions

Die zweite Frage bezieht sich auf den Zusammenhang zwischen dem Bedingungsteil und dem Anwendungsteil einer Solution. Wie Stiemerling zeigt, kann gesichert werden, dass eine Komposition nach der Anwendung einer Aktion (wie in Tabelle 10 beschrieben) in jedem Fall wieder eine gültige Komposition darstellt.

Darüber hinaus stellt sich die Frage, inwieweit eine Aktion (then-Teil) zur Behebung des durch eine Constraint (if-Teil) beitragen kann. Ein einfaches Beispiel für eine solche Regel ist in Abbildung 20 beschrieben. Hier hängt der Bedingungsteil direkt mit dem Aktionsteil zusammen. Wird die Aktion ausgeführt, weil die Bedingung verletzt wurde, kann anschließend sicher davon ausgegangen werden, dass die Bedingung nun erfüllt ist. Dies kann aber nicht verallgemeinert werden. Ein Gegenbeispiel kann gegeben werden, wenn der Aktionsteil nicht mehr die Methode *setParam(...)* aufruft, sondern lediglich einen Tip anzeigt. Wie auch im Bereich der Datenbanksysteme (Triggers oder ECA-Regeln, siehe z. B.

[Silberschatz et al., 2001]) ist nicht entscheidbar, ob ein System aus mehreren Regeln in sich schlüssig (sound) oder widerspruchsfrei ist.

Eine Prüfung einer einzelnen Regel auf ihre Konsistenz, lässt sich jedoch durchführen. In dem hier vorliegenden Fall, in dem Kompositionen auf Fehler geprüft werden sollen, und die möglichen Aktionen sehr eng mit den möglichen Regelfunktionen zusammenhängen, wird eine solche Überprüfung zudem erleichtert. Dies wird im folgenden Abschnitt, der die Restricted Solutions einführt, genauer beschrieben.

Grenzen der Solution Integrity

Eine Überprüfung der Regeln und sich evtl. daran anschließende Korrekturmaßnahmen sollten – wie schon weiter oben angedeutet – in Abhängigkeit der durchgeführten Anpassungsoperationen stattfinden. Neben der schon beschriebenen möglichen Inkonsistenz der Regelsysteme spielen dabei zwei weitere Probleme eine wichtige Rolle: Der Nicht-Determinismus bei der Anwendung von Regeln in unterschiedlicher Reihenfolge und das Nicht-Erkennen von Kreisschlüssen, die zu einer nicht terminierenden Ausführung von Solutions führen.

Der Nicht-Determinismus ist auch im Bereich der Datenbankmanagementsysteme bekannt. Das Problem entsteht dadurch, dass mehrere unterschiedliche Regeln zur gleichen Zeit anwendbar sind (siehe z. B. [Momjian, 2001]). Je nach Reihenfolge der automatischen Ausführung mehrerer anwendbarer ECA-Regeln kann es zu unterschiedlichen Endzuständen kommen. Die Implementierungen bekannter Datenbanksysteme greifen deswegen häufig auf eine eindeutige Priorisierung (z. B. durch manuell festgelegte Prioritäten oder eine Priorisierung in Abhängigkeit vom Erstellungsdatum) der ECA-Regeln zurück. Für das hier diskutierte Themenfeld ist dieses Problem jedoch nur bedingt relevant. Es geht hier nicht um eine vollautomatische Ausführung von Regelmengen auf Kompositionen, sondern um eine interaktive Benutzerunterstützung. Die Regeln dienen hier der Auffindung von Problemen und geben Hinweise auf mögliche Lösungen. Da eine sinnvolle Lösung nur in Abhängigkeit des mentalen Modells des Benutzers von seiner Applikation gefunden werden kann, ist eine vollständig automatische Korrektur in den wenigsten Fällen sinnvoll.

Zweiteres Problem entsteht beim Entwurf der Regelmenge. Hier kann es passieren, dass durch die im Aktionsteil beschriebenen Änderungsoperationen ein nicht endender Zyklus entsteht bzw. dass zwischen mehreren Solutions ungewollte Wechselwirkungen entstehen, die sich nicht auflösen lassen. Beispielsweise könnte eine Solution A einen Port P von einer Komponente hinzufügen, wenn er nicht existiert. Eine Solution B könnte diesen Port entfernen, sobald er existiert. Hier ist die Frage zu stellen, ob solche Kreisschlüsse generell erkannt werden können. Dies ist auch für unseren Ansatz ein relevantes Problem. Eine in sich fehlerhafte Regelmenge ist kaum dazu geeignet, Benutzer bei der Anpassung ihrer Applikation zu unterstützen und das Verständnis für die Semantik von Komponenten und Kompositionen zu verbessern. Dies führt zum Konzept der Restricted Solutions, das eben dieses Problem adressiert.

7.4 Restricted Solutions

Der Wunsch nach einer teil-automatisierten Fehlerkorrektur ist das Hauptanliegen bei der Einführung der Solution Integrity (siehe Abschnitt 7.3). Das Problem wurde im vorigen Abschnitt angesprochen. Falls der Aktionsteil einer Solution Anpassungsoperationen enthält, kann es vorkommen, dass diese Solution im Zusammenspiel mit anderen Solutions einen Kreislauf bildet, der sich nicht auflösen lässt. Nicht relevant sind hierbei Solutions, deren Aktion den Status einer Bedingung nicht verändern können. Dazu zählt beispielsweise

die Aktion „tip“, die lediglich einen textuellen Lösungsvorschlag ausgibt. Solche Solutions können unberücksichtigt bleiben, da sie, ohne den Status von Bedingungen zu ändern, auch keine Folgeprobleme auslösen können. Das Ziel ist nun, die Mächtigkeit der Solutions so einzuschränken, dass eine automatische Prüfung auf Erfüllbarkeit bzw. eine Prüfung auf die Nicht-Existenz oben angesprochener Kreisschlüsse durchgeführt werden kann.

Wie im folgenden Abschnitt gezeigt wird, ist es möglich, alle relevanten Solutions in logische Ausdrücke zu überführen. Ist der logische Ausdruck, der aus den relevanten Solutions und den Bedingungen gebildet worden ist, erfüllbar, so kann durch Ausführung geeigneter Aktionen erreicht werden, dass alle Bedingungen erfüllt und alle Solutions gelöst werden. Ist er nicht erfüllbar, so liegt ein Fehler in der Beschreibung der Solutions vor, der in der bisherigen Argumentation nicht berücksichtigt worden ist.

Wie Papadimitriou in [Papadimitriou, 1994] ausführt, ist die Auswertung allgemeiner logischer Ausdrücke NP-vollständig.[19] Es gibt hingegen viele Untermengen der Menge aller logischer Ausdrücke, deren Ausdrücke in polynomieller Zeit ausgewertet werden können. Für das Problem der Lösung von Solutions eignen sich dazu insbesondere Hornklauseln [Papadimitriou, 1994], wie weiter unten erläutert wird. Hornklauseln bilden eine echte Untermenge der Menge aller Ausdrücke die mit Solutions erzeugt werden können. Dennoch besitzt diese Untermenge eine interessante Mächtigkeit. Eine Solution, die so aufgebaut ist, dass der von ihr ableitbare logische Ausdruck nur aus Hornklauseln besteht, wird im folgenden Restricted Solution genannt.

Den Einschränkungen bei der Gestaltung der Restricted Solutions steht die Möglichkeit der automatischen Prüfung von Fehlern innerhalb eines Systems von Integritätsregeln gegenüber. Insofern sind Restricted Solutions für die hier vorgestellte Thematik äußerst wertvoll.

Die oben eingeführten Solutions lassen sich grundsätzlich als logische Ausdrücke darstellen und auch so auswerten. Beispielsweise kann die Regel

if X AND Y then A else B als Ausdruck der Form

$$(x \wedge y \Rightarrow A) \wedge (\neg(x \wedge y) \Rightarrow B)$$ formuliert werden.

Eine Hornklausel ist eine Disjunktion, in der maximal ein Literal nicht negiert ist. Ein Beispiel dafür ist

$$\neg x \vee \neg y \vee A.$$

Dieser Ausdruck lässt sich auch schreiben als Implikation der Form

$$x \wedge y \Rightarrow A.$$

Für Hornklauseln lässt sich zeigen, dass sich ihre Erfüllbarkeit in polynomieller Zeit prüfen lässt [Papadimitriou, 1994]. Die Restricted Solutions, wie sie in dieser Arbeit nun verwandt werden, sind eingeschränkt auf die oben angesprochene Form und haben keinen Else-Teil mehr.

Die Frage, inwieweit Restricted Solutions eine notwendige Beschränkung der allgemeineren Solutions darstellen und ob Kreisschlüsse ein häufig auftretendes Phänomen bei der Definition einer wohldurchdachten Menge von Integritätsbedingungen sind, lässt sich nicht leicht beantworten. Exemplarisch soll im Folgenden gezeigt werden, dass es Anwendungsfälle geben kann, in denen die Verwendung von Restricted Solutions von Vorteil ist.

[19] Dieses Problem wird SAT (Satisfiability) genannt.

Ein sehr einfaches Regelsystem mit Widerspruch stellt sich wie folgt dar (Notation in einer vereinfachten Pseudosprache):

```
(1) if hasComponent(A) then addcomponent B
(2) if hasComponent(A) then removecomponent B
```

Abbildung 22: Widersprüchliche Solutions

Sobald die Komponente A instanziiert wird, beinhalten die auszuführenden Solutions einen nicht-lösbaren Widerspruch. In diesem Falle sind jedoch die semantischen Informationen der die Komponente A und B enthaltenen Komponente fehlerhaft und lösen so den Widerspruch aus. Es ist offensichtlich, dass solche Regeln nicht gebildet werden sollten. Trotzdem kann dies manchmal geschehen. Eine Prüfung der Regelmenge wäre also bei der Definition auch hier sinnvoll.

Ein weiteres Beispiel zieht in Betracht, dass auch in komponentenbasierten Architekturen häufig bekannte Patterns verwendet werden. In diesem Fall wird das Decorator Pattern verwendet, um aus mehreren Einzelkomponenten ein Textausgabefenster zusammenzustellen. Die Komponenten sind in Tabelle 11 zusammengefasst dargestellt.

Komponente	Parameter	Beschreibung	Regel auf dieser Komponente
Hintergrund	BGColor	Erzeugt einen Hintergrund für den Text in der im Parameter „BGColor festgelegten Farbe.	If BGColor = $TextColor_i$ THEN deleteparam $TextColor_i$ (Die Regel legt fest, dass es keine Textfarben geben kann, die der Hintergrundfarbe entsprechen.)
Text	TextColor1... TextColorN	In dieser Komponente wird der Text in mehreren Farben, die fest definiert sind, dargestellt. Der Text selbst enthält Markierungen für die jeweils aktuelle Textfarbe.	
Link-Management		Komponente zur Verwaltung und Darstellung von Links	If yes setParam Text.TextColorN := blau (Links sind immer blau)

Tabelle 11: Beispiel für Restricted Solutions

Mit diesen Komponenten soll ein Ausschnitt eines Chat-Systems dargestellt werden. Die Hintergrundfarbe für normale Benutzer ist dabei rosé, für Übungsleiter blau. Betrachtet man nun das Regelsystem dieses Komponenten-Sets so zeigt sich ein unlösbarer Widerspruch bei der Komposition für die Übungsleiter. Die Hintergrundfarbe ist auf „blau“ gesetzt. Eben jene Farbe darf deswegen nicht in der Menge der Textfarben enthalten sein. Weiterhin soll aber die Linkfarbe immer blau sein, weswegen sie den Textfarben hinzugefügt werden muss. Aus der Sicht jeder einzelnen Komponente betrachtet ist jede Regel sinnvoll. Erst das Zusammenspiel und die Komposition erzeugen den Widerspruch, der bei der Entwicklung der Einzelkomponenten nicht vorhersehbar war. Hier zeigt sich, dass Widersprüche (Kreisschlüsse) in Regelsystemen auftreten können, ohne dass dies beabsichtigt war oder provoziert wurde. Weiterhin sieht man, dass das Auffinden solcher Widersprüche zur Designzeit (des Regelsystems) häufig schwierig ist. Insofern ist das Konzept der Restricted Solutions, die eine automatisierte Prüfung des Regelsystems erlauben, eine sinnvolle Hilfe.

7.5 Adaptionen des Constraints-Ansatzes

Der Constraint-basierte Ansatz ist sehr universell gehalten und lässt sich an verschiedenen Punkten vor allem hinsichtlich der Nutzerunterstützung verfeinern. Im Folgenden sollen deswegen vor allem zwei Spezialisierungen dieses Ansatzes diskutiert werden. Sie bringen aus technischer Sicht keine Erweiterung, sondern sind vor allem in Bezug auf die Anpassungsschnittstelle hilfreich.

- **Intuitive Integrity** (Beschreibungen): Diese Form der Anpassungsunterstützung wird nicht durch eine aktive Integritätsprüfung unterstützt. Stattdessen geht es hier darum, Komponenten Hilfstexte beizufügen, die auf Wunsch von den Benutzern abgerufen werden können. Auf diese Weise können den Benutzern auch nicht strukturierbare oder in Regeln formulierbare Hinweise zur Verfügung gestellt werden.

```
<descriptions>
   <component>Client-Komponente</component>
        <subcomponent name=”geochat.ui.ChatInput”>
             Sichtbare Komponente zur Eingabe
        </subcomponent>
   <port name= „login“> Dieser Port sollte lokal mit einer Komponente
        ‘Client’ verbunden sein
   </port>

   <parameter name=“AnzahlWiederholungen“> Gibt an, wie oft ein falsches
        Password eingegeben werden kann
   </parameter>
</descriptions>
```

Abbildung 23: Beispiel für Intuitive Integrity

  Die hier aufgeführten Beschreibungen werden den Benutzern im Anpassungsmodus auf Anforderung hin angezeigt. Beispielsweise lassen sie sich darstellen, wenn eine Komponente mit der rechten Maustaste ausgewählt wurde und im dann erscheinenden Kontextmenü der Menüpunkt „Beschreibung anzeigen“ gewählt wurde. Intuitive Integrity-Beschreibungen ähneln also Solutions, die als Bedingung ein „false“ bzw. ein „MarkedAndClicked“ haben und als Aktion ein „Dialog“. Sie sind hier genannt, da sie bei der Implementierung auf den Constraints aufsetzen und ein gemeinsames Sprachkonzept verwendet wird.

- **Visual Position Integrity**: Als separates Konzept soll die „Visual Position Integrity“ betrachtet werden. Sichtbare Komponenten einer Anwendung werden mit der Desktop-Metapher dargestellt. Dabei werden diese Komponenten wie auf einem Schreibtisch angeordnet. Es kann vorkommen, dass sich sichtbare Komponenten überlappen. Die „Visual Position Integrity“ nimmt für sichtbare Komponenten an, dass eine Überlappung der Komponenten zu vermeiden ist. Dazu testet diese Integritätsprüfung für alle visuellen Komponenten die Parameter für Position und Größe und vergleicht diese mit denen aller anderen sichtbaren Komponenten, um herauszufinden, ob sich Komponenten überlagern. Wenn dem so ist, wird eine Fehlermeldung erzeugt.

  Prinzipiell stellt dieses Konzept eine Operation für Nebenbedingungen dar, und man könnte diese Integritätsprüfung auch durch Nebenbedingungen beschreiben. Jedoch gilt die Unterstellung für alle sichtbaren Komponenten, die durch die Desktop-Metapher dargestellt werden.

7.6 Analyse der Integritätsbedingungen

Die implementierten Prüfverfahren für die Solution Integrity verwendet basieren auf einem naiven Ansatz. Es wird bei jedem Prüfvorgang die gesamte Komposition hinsichtlich der vorhandenen Integritätsbedingungen überprüft.

Im Bereich der Datenbanken finden sich jedoch eine Reihe deutlich optimierter Verfahren. Sie gehen davon aus, dass eine Datenmenge sich in einem konsistenten Zustand befindet. Bei einer Änderung muss dann nicht die gesamte Datenmenge, sondern eine spezielle Untermenge untersucht werden. Zu diesem Zweck werden die allgemein formulierten Integritätsbedingungen umformuliert. Sie reagieren dann auf Veränderungen der Datenbasis.

Die Verwendung eines naiven Prüfverfahrens reicht jedoch in dem hier beschriebenen Kontext aus verschiedenen Gründen aus. Während in Datenbanken üblicherweise große Datenbestände verwaltet werden und eine Prüfung dabei sehr aufwendig wird, ist die hier zu überprüfende Datenmenge (Anzahl der Komponenten) sehr klein. Dadurch spricht aus Effizienzgründen wenig für eine Optimierung dieses Ansatzes. Weiterhin liegt der Schwerpunkt der Arbeit liegt nicht auf der Implementierung möglichst effizienter Prüfverfahren, sondern auf der Untersuchung, inwieweit Benutzern durch ein Unterstützungskonzept das Anpassen von Applikationen erleichtert werden kann. Dies kann prototypisch auch mit den naiven Prüfungsverfahren evaluiert werden.

Die Reihenfolge, in der die Regeln überprüft werden, ist zufällig. Das gleiche gilt auch für die Reihenfolge die Überprüfung der Komponenten. Hierdurch kann es vorkommen, dass Benutzer, die Integritätsverletzungen in der Reihenfolge abarbeiten, in der sie vom System gemeldet werden, bei der gleichen Integritätsverletzung unterschiedliche Lösungswege wählen. Das hier vorgestellte Verfahren ist so also nicht deterministisch. Allerdings lässt sich zusichern, dass die ordnungsgemäße Korrektur aller Integritätsverletzungen zu einem integeren Zustand der Komposition bezüglich der vorhandenen Regelmenge führt.

7.7 Fazit

Mit Hilfe der hier vorgestellten Constraints, die anschließend zum Konzept der Restricted Solutions verfeinert wurden, wurde ein sehr mächtiges Integritätsprüfungskonzept dargestellt. Durch die Prüfung nahezu aller Eigenschaften von Kompositionen und aller in Komponentenarchitekturen möglichen Anpassungsoperationen kann jeder Anpassungsvorgang optimal unterstützt werden. Dabei stellt das hier vorgestellte Konzept lediglich einen technischen Rahmen dar, der – wie gezeigt wurde – überprüfbar korrekt arbeitet. Dies gilt insbesondere für die Restricted Solutions, mit denen es möglich wird, maschinell die Widerspruchsfreiheit von Regeln zu überprüfen. Es wurde gezeigt, dass mit Hilfe der hier vorgestellten Regelsprache grundsätzlich alle Eigenschaften einer Komposition prüfbar sind. Im Extremfall ließe sich also auf Basis von Constraints bzw. Restricted Solutions genau eine Komposition als gültig beschreiben. Der Grad der Unterstützung beim Anpassungsvorgang selbst ist weiterhin von weiteren Faktoren abhängig. Vor allen Dingen sind dies die Definition der in einem Anwendungskontext relevanten Regelmenge und die Unterstützung des Integritätskonzepts an der Benutzungsschnittstelle. Beides wird in Kapitel 10, das die vorgestellten Konzepte im Zusammenhang mit einer Anpassungsschnittstelle evaluiert, diskutiert.

8 Event Flow Integrity

Im vorigen Kapitel wurden die Restricted Solutions vorgestellt. Sie erlauben auf Basis von Regeln eine Überprüfung aller Eigenschaften einer Komposition. Dahingehend ist demzufolge keine weitere Integritätsprüfungsstrategie mehr notwendig. Nichtsdestotrotz gibt es Aspekte, die bei einer Überprüfung der Komposition durch die Restricted Solutions unberücksichtigt bleiben. Sie beziehen sich auf die Semantik einer Komposition, die vor allem durch die Existenz und die Interaktion zwischen Komponenten beschrieben wird. Die Event Flow Integrity (EFI) [Won, 2000] beschreibt lokale Integritätsbedingungen, die sich auf die Verwendung von Schnittstellen einer Komponente beziehen. Die Prüfung solcher Integritätsbedingungen geht dabei über die typkorrekte Verwendung von Schnittstellen hinaus.[20] Vielmehr steht bei dieser Integritätsprüfung die Untersuchung des Nachrichtenflusses zwischen den Komponenten einer Komposition im Vordergrund. Ein sehr einfaches Beispiel, das lediglich der ersten Motivation des Themas dienen soll, zeigt Abbildung 24.

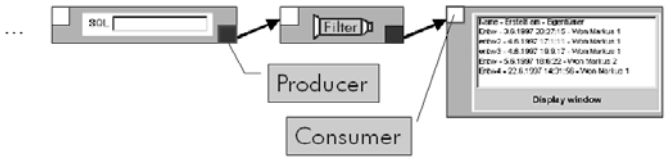


Abbildung 24: Event Flow Integrity am Beispiel eines Suchtools

In diesem Ausschnitt eines Suchwerkzeugs sind drei Komponenten miteinander verbunden: Die Suchmaschine, die eine Zugriffsmöglichkeit auf darunter liegende Datenbanken hat, eine Filterkomponente und ein Ausgabefenster.

In Integritätsbedingungen lässt sich definieren, dass die Suchmaschine eine Nachricht (Suchergebnis) erzeugt, die innerhalb der Komposition zu konsumieren ist. Für das Kompositionswerkzeug bedeutet das, dass überprüft werden muss, ob dieser Ausgangsport (Producer) an einen „sinnvollen" Eingangsport angeschlossen ist, der in der Lage ist, die übermittelten Nachrichten zu konsumieren. Im ersten Schritt ist also zu überprüfen, ob es innerhalb der Komposition eine Verbindung gibt, die den Ausgangsport der Suchmaschine mit einer anderen Komponente verbindet. Dies ist im oben skizzierten Beispiel der Fall. Allerdings ist die in der Mitte liegende Weiche kein regulärer Konsument (Consumer) der über die Schnittstelle übermittelten Nachrichten. Sie ist lediglich in der Lage, ein Suchergebnis zu modifizieren. Ein Prüfalgorithmus muss also hier die Anforderung, einen regulären Konsumenten zu finden, weiterpropagieren. Das Ausgabefenster ist ein solcher Konsument. Die oben beschriebene Konfiguration ist also gültig in dem Sinne, dass Suchmaschine und Ausgabefenster transitiv miteinander verbunden sind.

[20] Eine solche Überprüfung leistet FREEVOLVE selbstverständlich ebenso. Sie ist aber nicht Thema der folgenden Diskussion.

Die Betrachtung von Input- und Output-Streams in Data Spaces wird in [Cremers und Hibbard, 1978] diskutiert. Hier steht die Betrachtung von Kompositionen und sich daraus ergebenden Funktionen im Vordergrund. Eine Analyse der hier beschriebenen Kompositionen und den durch sie definierten Nachrichtenflüssen ist in dieser Form auf theoretischer Basis durchführbar. Praktischer und in unserem Fall, in dem es um Benutzerunterstützung geht, anschaulicher ist eine Modellierung auf Basis von Petri-Netzen. Weiterhin eignet sich diese Modellierung auch als Basis einer effizienten Prüfung. Eine ausführliche Beschreibung von Petri-Netzen findet sich beispielsweise in [Reisig, 1990], eine kurze Einführung gibt Anhang E. In den folgenden Abschnitten soll die Idee und die tieferliegenden Konzepte evolutionär entwickelt und verfeinert werden.

8.1 Petri-Netz-basierte Analyse – intuitiver Zugang

Zu Beginn ist es notwendig, eine gegebene Komposition in ein Petri-Netz zu übertragen. Diese Abbildung soll zu Beginn dieses Kapitels intuitiv erfolgen. Eine formale Beschreibung wird zu einem späteren Zeitpunkt nachgeliefert.

8.1.1 Abbildung eines Komponentennetzes auf ein gefärbtes Petri-Netz

Allgemein lässt sich ein Petri-Netz, wie es in Anhang E eingeführt wird, als ein gerichteter Graph auffassen. Dabei wird zwischen zwei Knotentypen, den Stellen und den Transitionen, unterschieden.

Wie auch in Produktionssystemen ist es sinnvoll, die Prozesselemente als Transitionen zu modellieren. Die grundlegende Idee des Mappings besteht also darin, alle Komponenten durch Transitionen darzustellen. Verbindungen zwischen solchen Transitionen erfolgen ebenso wie die Verbindungen zwischen Komponenten. Eine Verbindung wird dann repräsentiert durch eine Kante, eine Stelle und einer weitere Kante.

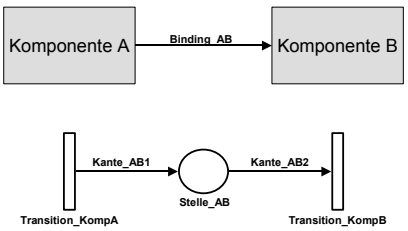


Abbildung 25: Mapping zwischen Komposition und Petri-Netz

Eine Transition kann in diesem Modell dann schalten, wenn die in ihrem Vorbereich liegenden Stellen markiert sind. Dies entspricht dem Gedanken, dass eine Komponente durch eingehende Events getriggert wird. Ebenso liefert sie Ergebnisse (Marken) über ausgehende Ports weiter. Im folgenden wird diese Regel (Schaltregel) weiter verfeinert. Für das grundlegende Verständnis ist sie hier jedoch ausreichend.

Eine etwas ausführlichere Beschreibung des Mappings zwischen Komposition und Petri-Netz lässt sich gut an folgendem Beispiel illustrieren (siehe Abbildung 26):

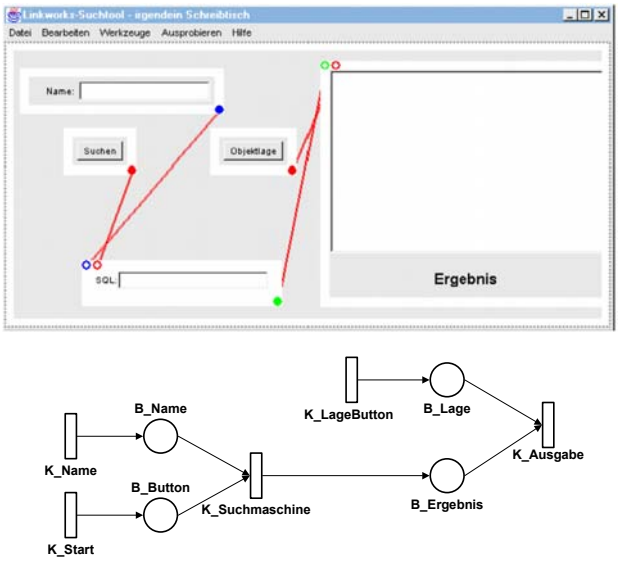


Abbildung 26: Eine Komposition (a) und das dazugehörige Petri-Netz (b)

Die Abbildung zeigt den Screenshot einer konkreten Komposition (hier ein Suchwerkzeug aus [Won, 1998]). Im Feld Name kann ein Begriff eingegeben werden. Sobald die Schaltfläche „Suchen“ gedrückt wird, erfolgt eine entsprechende Anfrage an eine SQL Datenbank. Deswegen sind die beiden Ausgänge von Name und Suchen mit der Datenbank verknüpft. Die Ausgabe der Datenbank wird ihrerseits wieder an ein Ergebnisfenster weitergeleitet. Die Ausgaben in diesem Fenster sind selektierbar, mit der Schaltfläche „Objektlage“ kann die Anzeige weiterer Details eines selektierten Eintrags erfolgen. Das entsprechende Petri-Netz ist darunter abgebildet. Jede Komponente entspricht einer Stelle des Petri-Netzes. Vereinfacht betrachtet werden zwischen den Komponenten Nachrichten ausgetauscht.

Aus der Abbildung lassen sich die Transitionen weiter in Eventquellen, Eventsenken und Eventträger untergliedern:

- **Eventquellen** (Produzenten) sind Komponenten, die neue Events erzeugen. Die Event-Erzeugung wird dabei in der Regel vom Betriebssystem angestoßen, beispielsweise durch ein Eingabegerät oder einen ausgelösten Interrupt. Eine Eventquelle ist im vorliegenden Beispiel der Such-Button. Die Transitionen des Petri-Netzes erzeugen keine Marken. Dies lässt sich simulieren, indem man den (leeren) Vorbereich jeder Eventquelle um eine markierte Stelle erweitert.
- **Eventsenken** (Konsumenten) sind Komponenten, die Events absorbieren. Die Information, die ihnen durch eingegangene Events übermittelt worden ist, werden dort verarbeitet. Die Verarbeitung kann in Form von Ausgaben oder externer Bearbeitung (Druckroutine des Betriebssystems) bestehen . In jedem Fall wird das Event nicht mehr weiter innerhalb der Komposition verwendet. Im beschriebenen Beispiel ist das Ergebnisfenster eine solche Eventsenke. Auch die Eventsenken werden erweitert. Sie erhalten im Nachbereich eine weitere Stelle, die das Ergebnis auffangen kann.

- **Eventträger** (Forwarder) sind Komponenten, die einerseits Events von anderen Komponenten aufnehmen und andererseits neue Events weiterleiten. Die Inhalte werden dabei typischerweise verändert. Auch können verschiedene eingehende Events gebündelt werden oder ein eingehendes Event kann in verschiedene aufgesplittet werden. Die Suchmaschine (SQL) ist ein Eventträger.

Durch die hier beschriebene Erweiterung der Eventquellen und Eventsenken ergibt sich nun folgendes Petri-Netz:

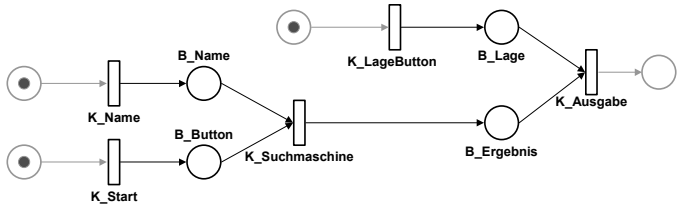


Abbildung 27: Um Eventquellen und -senken erweitertes Petri-Netz

Für die Analyse des Nachrichtenflusses innerhalb einer Komposition ist es nicht notwendig, komplexe Komponenten gesondert zu betrachten. Da sie aus Sicht der Architektur ein „Ordnungsmittel“ darstellen, können die in ihr enthaltenen Komponenten auch direkt auf das Petri-Netz übertragen werden. Verbindungen zwischen Komponenten, die in unterschiedlichen komplexen Komponenten enthalten sind, werden über diese umgeleitet. Da komplexe Komponenten die Nachrichten jedoch nicht verändern, kann auch die Abbildung der Verbindung vereinfacht werden, so dass solche einfachen Komponenten dann im Petri-Netz ebenso als direkt (über eine dazwischenliegende Stelle) miteinander verbundene Transitionen behandelt werden können.

8.1.2 Eventflüsse

Für unsere Analyse ist entscheidend, ob alle vorhandenen Markierungen (in der Anfangsmarkierung liegen diese in Eventquellen) zu Eventsenken gelangen können. Aus dieser allgemeinen Formulierung lassen sich Bedingungen formulieren:

- Alle Eventquellen sind mit mindestens einer Eventsenke verbunden.
- Alle Eventsenken sind umgekehrt mit mindestens einer Eventquelle verbunden.
- Alle Eventträger sind sowohl mit einer Eventquelle als auch mit einer Eventsenke verbunden.
- Es gibt eine Folgemarkierung der Anfangsmarkierung, in der alle Marken in Senken liegen.

Die hier ausgeführten Bedingungen sind als erster Schritt zur Prüfung der Integrität der Nachrichtenflüsse zu verstehen. Im Folgenden werden sie sukzessive verfeinert.

8.1.3 Optionale und essentielle Ports

Weiter oben wurde dargelegt, dass zu untersuchen ist, wie eine sinnvolle Anfangsmarkierung aussehen muss bzw. welche Bedingungen an eine Endmarkierung gestellt werden müssen. Betrachtet man die Eventflüsse des Petri-Netzes und deren Ursprung in der Komposition, so lassen sich zwei Arten von Nachrichtentypen unterscheiden: Optionale und für die Funktionalität einer Komposition essentielle. Während erstere innerhalb einer Komposition verwendet werden können, um bestimmte Effekte zu erzeugen, die nicht unbedingt notwendig sind, sind die essentiellen unbedingt notwendig. Gleiches gilt selbstverständlich auch für Eventsenken. Als Beispiel sei hier ein erweitertes Ergebnisfenster zur Anzeige von Suchergebnissen genannt (vgl. Abbildung 26). Es hat zwei Eingangsports. Ersterer dient dazu, ein Suchergebnis zu empfangen. Zweiterer ermöglicht es, eine Befehlsfläche anzuschließen, mit der sich zu den Suchergebnissen weitere Information zu den gefundenen Dokumenten (wie Datum, Größe, Eigentümer etc.) abfragen lassen. Der Eingang für die Suchergebnisse ist hier eine essentielle Eventsenke. Ohne ihn ist es sinnlos, das Ergebnisfenster in die Komposition zu integrieren. Andererseits ist es nicht unbedingt notwendig, Zusatzinformationen abrufen zu können (optionale Eventsenke).

Aus Sicht des Petri-Netzes werden entsprechend die Stellen, die den Eventquellen vorgeschaltet bzw. den Eventsenken nachgeschaltet sind, danach unterschieden, ob sie essentiell oder optional sind. In Abbildung 28 sind die beiden essentiellen Stellen dick umrahmt.

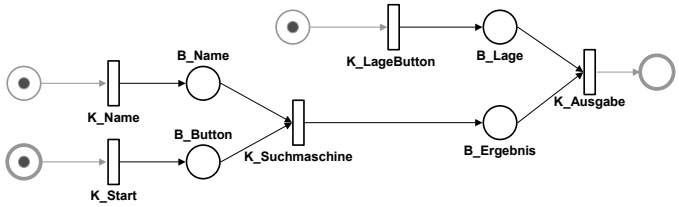


Abbildung 28: Um Eventquellen und -senken erweitertes Petri-Netz

Damit lassen sich die oben eingeführten Bedingungen für eine im Sinne der EFI gültige Komposition verfeinern:

- Alle essentiellen Eventsenken sind (transitiv) mit (beliebigen) Eventquellen verbunden.
- Die Anfangsmarkierung lässt sich bilden, indem alle Eventquellen Marken erhalten.
- Eine Komposition ist im Sinne von EFI integer genau dann, wenn es zur gegebenen Anfangsmarkierung und für jede essentielle Senke s mindestens eine Folgemarkierung gibt, so dass s eine Marke enthält.

Weiterhin muss auch der Eventrückfluss geprüft werden. Er besagt, dass alle in Eventquellen erzeugten Nachrichten auch verbraucht werden. In diesem Falle lässt sich das Petri-Netz umkehren (Drehung der Pfeile), aus Quellen werden Senken. Nun gelten folgende Bedingungen.

- Alle essentiellen Eventquellen sind (transitiv) mit (beliebigen) Eventsenken verbunden.
- Die Anfangsmarkierung lässt sich bilden, indem alle Eventsenken Marken erhalten.
- Eine Komposition ist im Sinne von EFI integer genau dann, wenn es zur gegebenen Anfangsmarkierung und für jede essentielle Quelle q mindestens eine Folgemarkierung gibt, so dass q eine Marke enthält.

8.1.4 Typisierungen/Färbungen von Events

Nachdem nun grundsätzlich dargelegt worden ist, wie sich Kompositionen mit Hilfe von Petri-Netzen darstellen lassen, geht es nun darum diesen Ansatz entsprechend den Notwendigkeiten, die sich aus den Komponentenarchitekturen ergeben, zu verfeinern. Bisher werden unterschiedliche Typen von Events nicht unterschieden. Dies ist jedoch für eine sinnvolle Analyse in jedem Fall eine Notwendigkeit. Hierzu bieten sich gefärbte Petri-Netze (siehe z. B. [Jensen, 1994]) an. Allerdings ist die Färbung des Petri-Netzes bzw. der in ihm enthaltenen Marken entsprechend der Typen der Events nur unzureichend, wie folgendes Beispiel zeigt:

Gegeben sei eine Textverarbeitung mit Rechtschreibkontrolle. Die Rechtschreibkontrolle hat eine Schnittstelle, um Texte zu importieren, die Textverarbeitung kann Texte exportieren. Beide Schnittstellen lassen sich also syntaktisch verbinden (sowohl der Eingangs- als auch der Ausgangsport sind vom Typ „Texte“). Trotzdem führt das nicht zum gewünschten Ergebnis, wenn die Rechtschreibkontrolle nur deutschen Text kontrollieren kann, die Textverarbeitung aber englischen exportiert. Um dieses Problem zu lösen, werden die Marken, auf die die Events abgebildet werden, gefärbt (Deutsch, Englisch). Diese Färbung existiert orthogonal zur syntaktischen Typisierung der Komponentendefinition. Teilweise kann sie sich decken, teilweise wird sie die syntaktische Typisierung verfeinern oder auch verschiedene syntaktische Typen zusammenfassen. Angenommen, die Rechtschreibkontrolle unterscheidet nicht zwischen Groß-/Kleinschreibung. Um effizienter zu arbeiten, kann sie nur Wörter testen, die vollständig in Großbuchstaben geschrieben sind. Darum wird der Kontrolle eine Komponente vorgelagert, die alle Buchstaben von Texten in Großbuchstaben umwandelt. Diese Komponente kann unabhängig von der Sprache arbeiten, in der der Text verfasst ist. Um diese Typisierung abzubilden, wird die Färbung der Marken hierarchisiert.

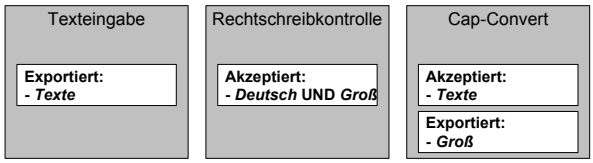


Abbildung 29: Beispiel Textverarbeitung

Im Beispiel gibt es dann eine Farbe Text, die weitere Farben, nämlich verschiedene Sprachen, als Söhne haben kann (siehe Abbildung 30). Eine Farbe einer niederen Hierarchiestufe kann an eine Komponente mit einer Schnittstelle einer Farbe höherer Hierarchiestufe weitergeleitet werden und verliert dadurch, auch beim Export aus dieser Komponente, seine Färbung nicht, sofern die Komponente die Färbung nicht explizit ändert. In unserem Fall muss die Komponente also wissen, dass sie spracherhaltend ist.

Sobald die Marke die Rechtschreibprüfung erreicht, ist es klar, dass es sich dabei um einen Text einer bestimmten Sprache handelt. Dass dieser Text aber auch gänzlich in Großbuchstaben geschrieben ist, geht aus der Marke alleine nicht hervor. Die Komponente zum Umwandeln der Buchstaben hat also eine neue Qualität erzeugt, die mit dieser Marke nicht ausgedrückt werden kann. Deswegen gibt sie der Marke eine weitere Farbe „Groß“ mit. Dies ist notwendig für die Rechtschreibprüfung, die ausschließlich Texte mit den Farben „Groß“ **und** „Deutsch“ verarbeiten kann. Diese Komponente erwartet also auch eine Marke, die beide Eigenschaften erfüllt.

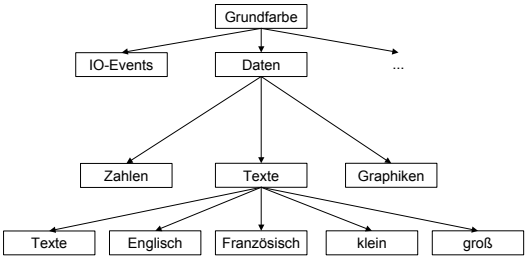


Abbildung 30: Färbungsbaum

Demnach muss die Integritätsprüfung, die Informationen über alle Komponenten innerhalb der Komposition hat, nicht nur deren zulässige Markenfarben kennen, sondern Zugriff auf den Vererbungsbaum der Färbungen haben. Nur so kann eine Textkomponente „wissen", dass sie spracherhaltend ist, auch wenn sie selbst nur auf der Texteigenschaft des Objektes operiert. Um dies zu gewährleisten, muss das Komponenten-Set, für das die Komponenten entwickelt werden, den Baum vordefinieren. Dies kann prinzipiell auf zwei verschiedene Arten geschehen:

- **Fest definierter Färbungsbaum**: Der Analysealgorithmus greift auf einen fest definierten Färbungsbaum zu. Jede Komponente kann dann nur Farben (Typen) dieses Baums verwenden. Dies führt jedoch zu einem sehr allgemeinen Färbungsbaum. Vor allem neue Komponenten, die nach dem Entwurf des Färbungsbaums hinzugefügt werden sollen, müssen die vorhandenen Typen verwenden.
- **Freier Entwurf von Färbungsbäumen**: Entwickler bzw. Administratoren der FREEVOLVE-Plattform haben Zugriff auf den Färbungsbaum und können diesen ändern, erweitern oder ersetzen. Diese Möglichkeit erhöht die Flexibilität und erlaubt es, schärfere Integritätsprüfungen vorzunehmen. Allerdings erhöht sich damit der Administrationsaufwand. Eine Möglichkeit besteht hier darin, den Entwicklern der Komponenten-Sets diese Aufgabe zu überlassen. Sie haben genaue Kenntnis der von ihnen verwendeten Java-Typen und kennen (in großen Teilen) die Anwendungsmöglichkeiten sowohl ihrer Einzelkomponenten als auch der möglichen Kompositionen aus ihnen. In Abschnitt 9 wird gezeigt, wie sich dieses Konzept nutzen lässt, um für eine Gruppe von gleichartigen Anwendungen einen gemeinsamen Färbungsbaum zu entwerfen.

Die hier eingeführte Färbung von Marken bzw. Petri-Netzen stellt so eine mächtige und zugleich sehr sinnvolle Erweiterung dar.

8.1.5 Schaltregeln

Bisher wurde davon ausgegangen, dass eine Transition immer dann schaltet, wenn in allen Stellen in ihrem Vorbereich Marken liegen. Dies gilt für gefärbte Petri-Netze grundsätzlich auch. Allerdings muss die Markenfarbe ebenso beachtet werden wie auch die Möglichkeit, dass eine Komponente in der Lage ist, eingehende Events zu verarbeiten und somit in andere Eventtypen zu transformieren. In die Sprache der Petri-Netze übertragen bedeutet dies, dass in einer Transformation mit Hilfe logischer Ausdrücke formuliert werden muss, wie sich die Transformation bei gegebenen Zuständen (bezogen auf ihren Vorbereich) verhält. Dies wird mit Hilfe von Schaltregeln festgelegt.

Eine Schaltregel beschreibt also, die nach der Schaltung im Nachbereich liegenden Marken in Abhängigkeit vom Vorbereich. Die Operanden haben dabei die Form [Stelle].[Farbe], als Operatoren sind ein AND und ein OR zugelassen. Selbstverständlich werden die Marken bei der Schaltung nach wie vor aus dem Vorbereich entfernt. Ein Beispiel für eine Schaltregel lässt sich für die Transition *K_Suchmaschine* formulieren:

$$B\_Name.Text \text{ AND } B\_Button.Action \rightarrow B\_Ergebnis.SearchResult$$

Für das Beispiel bedeutet dies, dass die Transition *K_Suchmaschine* nur dann schalten kann, wenn in den Stellen des Vorbereichs Marken mit den Farben *Text* und *Action* liegen. Nach der Schaltung werden die Marken entfernt, in der Stelle *B_Ergebnis* wird eine Marke mit der Farbe *SearchResult* hinzugefügt. Als Alternative steht auch ein OR als Operator zur Verfügung. Wird in obiger Schaltregel das AND durch das OR ersetzt, reicht eine der beiden Marken zum Schalten aus. Liegen in beiden Stellen passende Marken, so wird eine für den Schaltvorgang ausgewählt. Dieses Verhalten entspricht in der Sprache der Petri-Netze einer Alternative oder einer hinreichenden Bedingung.

Die Beachtung von „falschen“ Marken im Vorbereich einer Transition kann entfallen. Sie werden bei der Schaltung nicht berücksichtigt. Da ein Netz bei Nichterfüllung der oben beschriebenen Bedingungen als nicht integer eingestuft wird, haben solche Marken folglich auch keinen Einfluss auf das Ergebnis.

8.1.6 Behandlung von Fan-In und Fan-Out

In einem Komponentennetz können weiterhin mehrere Empfänger an genau einen Ausgangsport eines Sender angeschlossen werden (Fan-Out) oder auch mehrere Sender an einen Eingangsport eines Empfängers (Fan-In). Die zugrundeliegende Komponentenarchitektur lässt dies zu, die einzelnen Empfänger (oder Sender beim Fan-In) sind nicht unterscheidbar.

Entsprechend der Semantik (der zugrundeliegenden Komponentenarchitektur) werden im Falle des Fan-Out alle Empfänger gleichsam bedient. Das heisst, die ausgehende Marke wird dupliziert und an alle Empfänger weitergereicht.

Bei einem Fan-In wird wie bei einer in einer Schaltungsregel beschreibbaren Disjunktion verfahren. Für die Empfänger-Komponente ist nicht erkennbar, aus welcher Komponente die Marke, die sie verwerten kann, stammt. Sie wird dem Sender jedoch entfernt und dem Empfänger hinzugefügt.

8.1.7 Fazit

In den vorangegangenen Abschnitten wurde anschaulich beschrieben, wie sich ein zu einer Komposition gehöriges Petri-Netz definieren lässt. Dabei wurden die Komponenten mit ihren Bindungen übertragen. Zur Darstellung der unterschiedlichen Eventtypen wurde das Petri-Netz um farbige Marken erweitert. Sie führen zu einer Erweiterung der Regel, wann eine Transition in der Lage ist zu schalten. Im Folgenden wird diese informelle Beschreibung formalisiert. Anschließend werden die zur Analyse des Netzes notwendigen Techniken erläutert.

8.2 Abbildung einer Komposition auf ein Petri-Netz

Nachdem bisher ein intuitives Verständnis für den Ansatz gebildet wurde, geht es jetzt darum eine formal saubere Abbildung einer Komposition, wie sie in Kapitel 7 eingeführt wurde, auf ein Petri-Netz zu geben. Wie schon diskutiert, sind dazu einige Erweiterungen des Petri-Netzes notwendig.

> **Definition Farbe, Färbungsrelation, Farblinien und Farbbaum**: Sei *F* die endliche Menge aller in einer Komposition vorkommenden Farben. Dann ist eine Färbungsrelation *FR* definiert durch $FR \subseteq F \times F$. Dabei sei für $(f,g) \in FR$ *g* der Vater von *f*.
>
> Die Menge der Farblinien *FL* einer Färbungshierarchie ist die transitive Hülle über die Elemente der Färbungsrelation, mit anderen Worten
>
> $$(f,g) \in FL(FR) \Leftrightarrow \exists_n : \overset{n}{\underset{i=1}{\forall}} (a_i, a_{i+1}) \in FR \wedge a_1 = \wedge a_n = g .$$
>
> Eine Färbungsrelation *FR* ist ein Färbungsbaum genau dann, wenn $\forall f \in F : (f,f) \notin FL(FR)$. Damit ist ein Färbungsbaum eine zyklenfreie Färbungshierarchie.

Auf diese Weise lässt sich der in Abschnitt 8.1.4 eingeführte Färbungsbaum als eine Ordnungsrelation darstellen. Wie schon beschrieben wird ein solcher Baum nicht innerhalb einer Komposition definiert, sondern extern in XSemL beschrieben. Der in Abbildung 30 dargestellte Färbungsbaum hat dann in XSemL formuliert folgende Gestalt:

```
<efitree>
   <IO-Events/>
   <Daten>
        <Zahlen/>
        <Texte>
             <Deutsch/>
             <Englisch/>
             <Französisch/>
             <klein/>
             <gross/>
        </Texte>
        <Grafiken/>
   </Daten>
   <...../>
</efitree>
```

Abbildung 31: Färbungsbaum in XSemL

Die Farben werden nicht explizit beschrieben, sondern ergeben sich direkt aus dem Färbungsbaum. Weiterhin ist es notwendig, Marken als Nachrichtenträger zu definieren. Sie können mehrere Farben gleichzeitig enthalten.

> **Definition Marke**: Eine Marke *m* ist eine Menge von Farben $m \subseteq F$. *M(s)* ist die Markenfunktion, die alle Marken der Stelle s zurückliefert.[21]

Schließlich muss noch festgelegt werden unter welchen Bedingungen eine Transition innerhalb eines Netzes aktiviert werden kann. Die einfache Betrachtung ihres Vorbereichs ist hier nicht mehr ausreichend.

---

[21] Diese Definition entspricht der Markenfunktion der originären Petri-Netze und wird aus diesem Grund nicht weiter ausgeführt.

Für die weitere Beschreibung eines Petri-Netzes zur Überprüfung der Event Flow Integrität sind einige Hilfsfunktionen notwendig. So liefert die Funktion MF (*Markenfarbe*) alle Farben, die eine Marke in Abhängigkeit von einem gegebenen Färbungsbaum annehmen kann, zurück. Dabei geht es darum, nicht nur die in der Marke enthaltenen, sondern auch alle ihre Vorgänger im Färbungsbaum zu erfassen. Damit ist die Funktion Markenfarbe definiert als

$$MF(m, FH) = \{f \mid f \in m \vee \exists_{(g,f) \in FL(FH)} : g \in m\}.$$

> **Definition Erhaltende Marke**: Eine Marke *m* ist eine erhaltende Marke, wenn sie nach dem Schaltungsvorgang in der Stelle erhalten bleibt und so einer folgenden Schaltung wieder zur Verfügung steht.

In unserem Fall wird die erhaltende Marke in den Eventquellen[22] verwendet. Dem liegt die (positive) Annahme zugrunde, dass diese Quelle nicht nur ein, sondern auch mehrere Events erzeugen kann. Eine erhaltende Marke ist vergleichbar mit einer auf unendlich gesetzten Markenzahl.

Wie auch bei einem normalen Petri-Netz sind Transitionen in der Lage zu schalten oder aktiviert zu werden, wenn ihr Vorbereich bestimmte Bedingungen erfüllt. Dies führt zur Definition der Schaltungs-Vorbedingung:

> **Definition Schaltungs-Vorbedingung**: Eine Schaltungs-Vorbedingung ist eine Funktion $sb: \wp(S \times F) \times \{AND, OR\} \rightarrow \{wahr, falsch\}$ mit

$$sb(msm, op) := \begin{cases} wahr, \text{ falls } op = AND \wedge \forall_{(s,f) \in msm} \exists_{m \in m(s)} : f \in MF(m) \\ falsch \text{ sonst} \end{cases}.$$

Hierbei sind die Kreuzprodukte aus Stellen und Farben Einzelbedingungen, die an die Marken im Vorbereich gestellt werden. Die Verknüpfung der Bedingungen wird mit der Wahl *{AND,OR}* festgelegt. *m(s)* sei dabei die aktuelle Markierung an der Stelle *s* (siehe Anhang E). Weiterhin wird für eine Schaltung eine Beschreibung benötigt, wie sich der Nachbereich nach Aktivierung der Transition verhält.

> **Definition Schaltungs-Aktion**: Eine Schaltungs-Aktion $sa \in S \times F$ definiert, welche Marke (mit einer bestimmten Farbe) welcher Stelle des Nachbereichs zugewiesen wird.

Nun lässt sich detailliert bestimmen, wann eine Schaltung aktivierbar ist.

> **Definition Schaltregel**: Eine Schaltregel sr ist ein Tupel der Form *(t,sb,sa)* mit
>
> *t* Transition
> *sb* Schaltungs-Vorbedingung
> *sa* Schaltungs-Aktion.

Schließlich wirken sich Schaltungen auf die Markenbelegung innerhalb des Netzes aus. Die Anwendung einer ausgewählten aktivierbaren Transition *t* und der dazugehörigen Schaltregel *sr =(t,sb,sa)* führt also zu einem Übergang von *m(s)* nach *m'(s)* insbesondere in den Stellen des Vor- und Nachbereichs der Transition t. Es gilt für *sb=(msm,AND)*:

[22] Beim Eventrückfluss sind dies die Eventsenken.

$$m'(s) := \begin{cases} m(s), \text{falls } s \notin t\bullet \wedge s \notin \bullet t \\ m(s) \cup \{F\}, \text{falls } (s,F) \in sa \\ m(s) - m^*, \text{mit } m^* = getMark(s,f), \text{falls } (s,f) \in getOps(sb) \end{cases}$$

und entsprechend für *sb=(msm,OR)*

$$m'(s) := \begin{cases} m(s)\ \forall s : s \notin t\bullet \wedge s \notin \bullet t \\ m(s) \cup \{F\}, \text{falls } (s,F) \in sa \\ m(s) - m^*, \text{mit } m^* = getMark(s,f), \text{für genau ein } (s,f) \in getOps(sb) \end{cases}$$

Dabei ist $getMark: S \times F \rightarrow \wp(F)$ eine Funktion, die zu einer gegebenen Stelle s und einer Farbe f eine (beliebige) Marke liefert, die diese Farbe enthält.

$$getMark(s,f) := \begin{cases} m^*, \text{falls } \exists_{m^* \in m(s)} : f \in MF(m^*) \\ NULL \text{ sonst} \end{cases}$$

Außerdem wird die Methode $getOps: \wp(S \times F) \times \{AND, OR\} \rightarrow \wp(S \times F)$ benötigt. Sie liefert lediglich die Operanden der Schaltungs-Bedingung zurück.

$$getOps(sb) := mfm, \text{ mit } sb = (msm, operator)$$

In Anlehnung an [Baumgarten, 1990] sei ein Integrity Checking System definiert durch ein erweitertes gefärbtes Petri-Netz.

Definition Integrity Checking System (ICS): Ein 9-Tupel (S,T,F,R,C,CT,MF,I,M_0) sei ein Integrity Checking System, wobei

$S \neq \emptyset$	Menge der Stellen
T	Menge der Transitionen
$F: (S \times T) \cup (T \times S) \rightarrow \{0,1\}$	Flussrelation
R	Menge von Schaltregeln
C	Menge der Farben
CT	Färbungsbaum
$MF: S \rightarrow \wp(F)$	Markierungsfunktion
$I: S \rightarrow \{essentiell, optional\}$	Wichtigkeit der Stellen
$m: S \rightarrow \wp(F)$	Aktuelle Markierung mit F Menge der Farben
m_0	Startbelegung,

wobei zusätzlich gilt: $S \cap T = \emptyset$ und für alle Stellen $s^* \in S$ gibt es mindestens eine Schaltregel mit (s^*,sa,sb).

Das Petri-Netz ist bezüglich der Kantengewichte und der Kapazitäten vereinfacht. Diese Beschränkungen sind nicht mehr notwendig, da die simulierten Komponenten aufgrund der Spezifikation der FLEXIBEANS potentiell unendlich große Speicher zur Akkumulation eingehender Nachrichten besitzen. Die Kantengewichte, die den Kommunikationsfluss beschränken werden nicht benötigt, da auch dies durch die Spezifikation (Java-Methodenaufrufe) geregelt wird.

8.2.1 Mapping der Komposition auf das Integrity Checking System

Nachdem nun das Integrity Checking System (ICS) inkl. der Färbungen und der Zustandsübergänge während des Schaltens einer Transition definiert wurde, geht es nun darum, eine Komposition, wie sie in Kapitel 7 definiert wurde, darauf abzubilden. Gegeben sei in diesem Fall ein Komponentensystem (r,H,C).

Die Transitionen eines ICS lassen sich dann sofort aus der Menge aller instanziierten, einfachen Komponenten ableiten. Damit ist

$$T = \{s \mid \exists_{(n,P,A,Sp,B1,B2,B3)\in H}\exists_{(Type,Inst,Par)\in Sp}\exists_{(m,D,P,B)\in r} : Inst = t \wedge m = type\}\ .$$

Weiterhin müssen die Bindungen simuliert werden. Zu diesem Zweck wird für jede zwischen zwei Komponenten existierende Verbindung im ICS eine Stelle und zwei Kanten eingefügt. Dabei liefern folgende Ausdrücke Bindungen zwischen allen direkt miteinander verbundenen Instanzen:

$$S = \left\{[IA.PA.IB.PB] \mid (IA,PA,IB,PB) \in B1 \wedge IA \in T \wedge IB \in T\right\} \text{ und}$$

$$F = \left\{\{(IA,[IA.PA.IB.IB]),([IA.PA.IB.IB],IB)\} \mid (IA,PA,IB,PB) \in B1 \wedge IA \in T \wedge IB \in T\right\}\ .$$

Weiterhin ist es auch notwendig, die transitiv (über komplexe Komponenten) miteinander verbundenen Instanzen miteinander zu verbinden. Hier ist die Stellen- und Verbindungssuche innerhalb der Komposition etwas aufwendiger. Es ergibt sich:

$$S = S \cup \left\{ [IA.PA.IB.PB] \middle| \begin{array}{l} \exists_{(na,P,A,Sp1,B1,B2,B3)\in H}\exists_{(A1Port,InstA,PortA)\in B2} : IA = InstA \wedge PA = PortA \wedge \\ \exists_{(m,Q,B,Sp2,B4,B5,B6)\in H}\exists_{(A2Port,InstB,PortB)\in B5} : IB = InstB \wedge PB = PortB \wedge \\ \exists_{L\subseteq H}\forall_i\exists_{(o,R,C,Sp3,B7,B8,B9)\in L} : (si, pi, si+1, pi+1) \in B7 \\ \wedge\ s_1 = na \wedge p_1 = A1Port \wedge s_n = m \wedge p_n = A2Port \end{array} \right\}.$$

Entsprechend wird auch F erweitert

$$F = F \cup \left\{ \{(IA,[IA.PA.IB.IB]),([IA.PA.IB.IB],IB)\} \middle| \begin{array}{l} \exists_{(na,P,A,Sp1,B1,B2,B3)\in H}\exists_{(A1Port,InstA,PortA)\in B2} : IA = InstA \wedge PA = PortA \wedge \\ \exists_{(m,Q,B,Sp2,B4,B5,B6)\in H}\exists_{(A2Port,InstB,PortB)\in B5} : IB = InstB \wedge PB = PortB \wedge \\ \exists_{L\subseteq H}\forall_i\exists_{(o,R,C,Sp3,B7,B8,B9)\in L} : (si, pi, si+1, pi+1) \in B7 \\ \wedge\ s_1 = na \wedge p_1 = A1Port \wedge s_n = m \wedge p_n = A2Port \end{array} \right\}$$

Analyse der EFI-Informationen in XSemL

Nun sind die für EFI relevanten Elemente des XSemL-Files zu untersuchen. Auf Basis dieser Analyse ergibt sich, welche der Instanzen (entsprechend ihres Typs) Quellen bzw. Senken für Nachrichten sind. Solchen Transitionen werden dann Stellen im Vor- bzw. Nachbereich hinzugefügt. Weiterhin werden die entsprechenden Quellen-Stellen auch mit Marken versehen. Die Vorgehensweise lässt sich folgendermaßen beschreiben (vgl. Abbildung 32):

- Analysiere alle im XSemL-File als Quellen aufgeführten Komponententypen[23] (type)
- Finde alle Instanz-Transitionen t des entsprechenden Typs unter Zuhilfenahme der Instanziierungsrelation S der komplexen Komponenten und ihres auf die Transitionen übertragenen Namens. Für alle diese t gilt:
  - $\bullet t = \bullet t \cup [t.farbe]$ Es wird eine Stelle mit dem Namen der Transition und der Farbe *farbe* hinzugefügt.
  - $m_0([t.farbe]) = \{\{farbe\}\}$ Der Stelle wird eine erhaltende Marke mit der Farbmenge *farbe* hinzugefügt.

```
<!ELEMENT efi (type+)>
   <!ELEMENT type (quelle|senke|traeger)>
        <!ELEMENT quelle (farbe,modus)+>
        <!ELEMENT senke (farbe,modus)+>
        <!ELEMENT traeger (transitionen*, (farbe,modus)+>

        <!ELEMENT farbe (#PCDATA)>
        <!ELEMENT modus (#PCDATA)>

        <!ELEMENT transitionen (#PCDATA)>
```

Abbildung 32: EFI in XSemL

Ebenso werden auch die Senken markiert. Sie ergeben sich aus dem XSemL-File in ähnlicher Weise. Für alle Transitionen, die nach der XSemL-Definition eine Senke darstellen, wird eine weitere Stelle im Nachbereich dieser Transition t eingefügt. Damit ergibt sich:

- $t\bullet = t\bullet \cup [t.farbe]$Es wird eine Stelle mit dem Namen der Transition und der Farbe *farbe* hinzugefügt.
- Falls eine Senke als essentiell beschrieben ist, gilt I([t.farbe]) = essentiell, ansonsten I([t.farbe]) = optional.

Für die Träger lassen sich aus dem XSemL -File die Schaltregeln extrahieren. Sie werden in der in Abschnitt 8.1.5 informell vorgestellten (vom Parser auswertbaren) Syntax beschrieben und direkt in die Transition übertragen.

Damit ist das ICS vollständig beschrieben. Der nächste Schritt ist nun zu untersuchen, wann ein ICS in der Lage ist zu schalten und wann ein gültiger Endzustand erreicht wird.

> **Definition Aktivierbarkeit einer Transition**: Eine Transition *t* ist aktivierbar, wenn es eine entsprechende Schaltregel gibt, deren Schaltungs-Vorbedingung erfüllt ist. Mit anderen Worten:
>
> Transition *t* ist aktivierbar genau dann, wenn $\exists_{(tr,sa,sb) \in R}: tr = t \wedge sb = true$

[23] Ein Komponententyp kann sowohl Quelle, Senke und auch Träger für unterschiedliche Farben gleichzeitig sein. Dementsprechend kann einer Quell-Transition auch mehr als eine Quell-Stelle zugeordnet werden. Gleiches gilt für Senken und Träger.

Definition Schaltfolge: Eine Schaltfolge $a=(a_1,...,a_n)$ ist gültig in Abhängigkeit von einer Ausgangsmarkierung, wenn die einzelnen in ihr enthaltenen Transitionen a_i nacheinander aktivierbar sind und schalten.

Definition Erreichbarkeit eines Zustands: Ein Zustand $m*$ gilt als erreichbar, wenn es ausgehend von einer Startmarkierung m_0 eine Schaltfolge $a=(a_1,....a_n)$ gibt, so dass gilt:

$$a_n(...(a_1\ (m_0)...) = m*$$

8.3 Analyse des Netzes

Ein ICS lässt sich zur Untersuchung der Erreichbarkeit von essentiellen Senken nutzen. Eine positive Analyse besagt dann, dass es in der zugrundeliegenden Applikation die Möglichkeit gibt, dass es einen Eventfluss gibt, der eine essentielle Senke beliefert. Der Analysealgorithmus überprüft also, ob sich die Markierung des Netzes durch Schalten der Transitionen so überführen lässt, dass eine Marke auf einer essentiellen Stelle liegt.

Diese Erreichbarkeitsanalyse muss für alle essentiellen Stellen unabhängig durchgeführt werden. Damit ist die Analyse des Eventflusses erst erfolgreich abgeschlossen, wenn es positive Einzelanalysen für alle essentiellen Eventsenken gibt.

Definition Integrität des Petri-Netzes: Die Integrität eines Petri-Netzes zur Analyse einer Komposition kann als gegeben bezeichnet werden, wenn es für jede essentielle Eventsenke eine Schaltfolge gibt, mit der ausgehend von einer Anfangsmarkierung ein Zustand erreicht wird, in der sie (diese Senke) ihre essentiell benötigten Marken enthält.

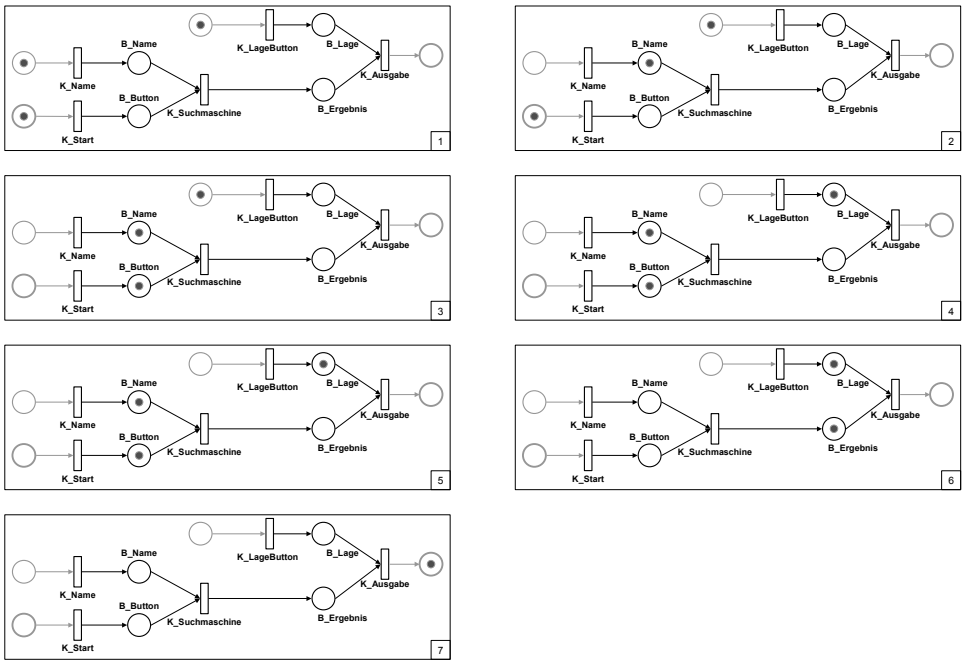


Abbildung 33: Fluss im ICS

Speziell ist die Anfangsmarkierung wie oben beschrieben so zu wählen, dass alle möglichen Marken enthalten sind (sowohl essentielle als auch optionale). Weiterhin wird die Menge der zu erreichenden Senken (essentiell) definiert durch die Funktion I, die ebenfalls Teil des ICS ist.

Eine Überprüfung des Netzes kann mit den bekannten Algorithmen (siehe z. B. [Reisig, 1990]) erfolgen. Sie können deterministisch ermitteln, ob eine Markierung bei einer gegebenen Startmarkierung erreicht werden kann.[24] Wie dies am in Abbildung 26 eingeführten Beispiel aussieht, wird in Abbildung 33 vereinfacht dargelegt. Diese Vereinfachung bezieht sich hier auf die nicht interpretierten Schaltregeln. In [Baumgarten, 1990] wird diskutiert, dass die Erweiterung eines Petri-Netzes zu einem beliebigen IM-System den Flussalgorithmus nicht grundsätzlich verändert, sondern die Analyse nur aufwendiger macht.

8.3.1 Effiziente Erreichbarkeitsanalyse durch Erkennung von Zyklen

Problematisch wird eine Analyse wie sie weiter oben beschrieben wurde vor allem dann, wenn die Komposition so gestaltet sein kann, dass das dazugehörige Petri-Netz Zyklen enthält [Krings, 2003]. Während das Nachweisen der Existenz eines Zyklus recht leicht zu zeigen ist, ist es ein nicht-triviales Problem zu ermitteln, ob ein gültiger Endzustand erreicht werden kann. Ein einfaches Durchschalten der Transitionen führt möglicherweise nicht zum gewünschten Ziel.

Es kann passieren, dass durch einen Kreis eine endlose Schaltfolge möglich ist. Wie schon weiter oben erwähnt, kann die Erreichbarkeit von bestimmten Markierungen in Petri-Netzen immer deterministisch festgestellt werden. Die Verfahren dazu sind jedoch recht komplex und damit zeitaufwendig. Baumgarten [Baumgarten, 1990] zeigt, dass für das allgemeine Erreichbarkeitsproblem in Petri-Netzen eine Komplexität von $O(2^{2^n})$ gilt. Die Erreichbarkeit einer bestimmten Stelle ist jedoch vergleichsweise einfach zu überprüfen. Für die zu implementierende Integritätsprüfung reicht hierzu eine partielle Erreichbarkeitsprüfung. Die Grundlage bildet dabei ein Erreichbarkeitsbaum. Die Knoten dieses Baumes werden aus den Markierungen gebildet, die Kanten sind Schaltungen. Als Beispiel zur Verdeutlichung der Funktionsweise des Algorithmus dient das Petri-Netz in Abbildung 34. Die Wurzel des Erreichbarkeitsbaums bildet die Anfangsmarkierung. Die Markierungen im Erreichbarkeitsbaum werden durch die Anzahl der Marken an den Stellen s1, s2 und s3 kodiert.

Die Söhne dieser Anfangsmarkierung sind alle Markierungen, die direkt durch Schalten einer Transition von der Anfangsmarkierung aus erreichbar sind. Die Kanten zu diesen Söhnen werden mit der Transition beschriftet, über die die Schaltung läuft. Die Folgemarkierungen dieser Söhne werden entsprechend eingetragen. Jede erreichbare Markierung ist durch einen Knoten im vollen Erreichbarkeitsbaum vertreten. Hat das Petri-Netz jedoch eine endlose Schaltfolge, so ist auch der Baum unendlich groß, wie im Beispiel. Eine Möglichkeit, den vollen Erreichbarkeitsbaum zu kürzen, stellen wiederkehrende Markierungen dar. Wird eine Markierung wiederholt erreicht, so können von dort aus nur Markierungen erreicht werden, die bereits von der entsprechenden früheren Markierung aus erreichbar gewesen sind. Da unser Ziel die Feststellung der Erreichbarkeit von Stellen ist, kann der weitere Baum abgeschnitten werden. Ein solcher Baum heißt beschnittener Erreichbarkeitsbaum (siehe Abbildung 34, Mitte).

[24] Diese Verfahren werden hier nicht näher beschrieben, da im folgenden Abschnitt ein sehr viel effizienteres Verfahren eingeführt wird.

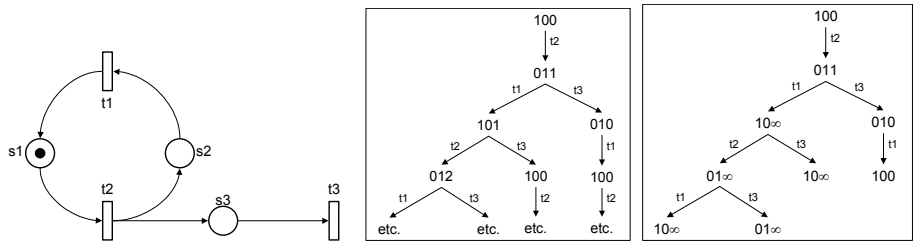


Abbildung 34: Partielle Erreichbarkeitsanalyse

Auch ein beschnittener Erreichbarkeitsbaum kann noch unendlich groß sein, wenn an einer Stelle im Petri-Netz unbegrenzt viele Marken akkumuliert werden können. Zur weiteren Verkürzung des Baumes kann die Eigenschaft helfen, dass genau dann an einer Stelle unendlich viele Marken akkumuliert werden können, wenn es zwei Markierungen gibt, die völlig identisch sind, bis auf die Tatsache, dass sich an eben jener Stelle in der Markierung tiefer unten im Baum mehr Marken befinden, und ein Pfad von der ersten zu der letzten Markierung existiert. Durch abermaliges Ablaufen des Weges zwischen diesen beiden Stellen kann die Markenanzahl wieder entsprechend erhöht werden. Bei einer solchen wiederkehrenden Markierung wird nun anstelle der erhöhten Markenzahl der betreffenden Stelle dort der Wert „unendlich“ eingetragen. Ansonsten wird der Baum genau so gebildet wie der beschnittene Erreichbarkeitsbaum. Insbesondere wird jedoch bei einem wiederholten Vorkommen der gleichen Markierung der Rest des Baumes abgeschnitten. Ein solcher Baum heißt dann Überdeckungsbaum (Abbildung 34, rechts).

Dort schaltet zunächst t2, die einzige Transition, die aus der Anfangsmarkierung heraus schalten kann. Die Stelle s1 verliert darauf hin ihre Markierung, die Stellen s2 und s3 erhalten jeweils eine. Nun kann sowohl t3 als auch t1 schalten. In ersterem Falle verliert lediglich s3 eine Marke. In letzterem Fall gewinnt s1 eine Marke hinzu und s2 verliert eine. Nun ist der oben beschriebene Fall eingetreten, dass alle Stellen genau gleich viele Marken enthalten, wie eine Vormarkierung bis auf eine Stelle, nämlich s3. Die Markierung ist demnach 101, wie auch im Erreichbarkeitsbaum zu sehen. Im Überdeckungsbaum wird jedoch die oben beschriebene Regel angewandt, weshalb der Knoten in diesem Baum den Namen 10∞ erhält. Baumgarten [Baumgarten, 1990] hat gezeigt, dass ein Überdeckungsbaum immer endlich ist. Das oben skizzierte primitive Analyseverfahren lässt sich also nun ersetzen durch eines, das auf der Berechnung von Überdeckungsbäumen basiert. Eine Erweiterung auf ein gefärbtes Petri-Netz ergibt sich analog.

8.3.2 Beschreibung des Eventrückflusses

Bisher wurde getestet, ob alle essentiellen Eventsenken durch eine Marke mit passender Farbe erreicht werden. Im Folgenden geht es nun darum, den Eventrückfluss zu definieren und zu prüfen. Der Eventrückfluss testet, ob alle essentiell erzeugten Marken auch verbraucht werden können. Dazu wird das Netz invertiert, die Startmarkierung geändert und die Funktion I, die die Wichtigkeit der einzelnen Stellen angibt, re-definiert. Für die Schaltregeln hat diese Invertierung insofern Einfluss, als dass sich der Übergang für die Funktion m (aktuelle Markierung) ändert.

Die Inversion ist leicht einzusehen, wenn man die Symmetrie der zugrundeliegenden Fragestellung betrachtet. Statt zu betrachten, welches Event weitergeleitet wird von einer Stelle, geht es nun darum zu fragen, welche Stelle Tokens von anderen fordert. Die Anfangsmarkierung sieht dann vor, dass Markierungen in allen Eventsenken liegen. Ein gülti-

ger Endzustand wird erreicht, wenn alle essentiellen Eventquellen eine Marke enthalten (vgl. [Krings, 2003]).

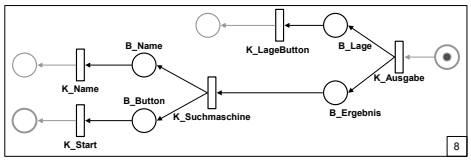
Definition Invertiertes ICS: Ein invertiertes ICS *C'* wird aus einem gegebenen ICS *C* gebildet, durch

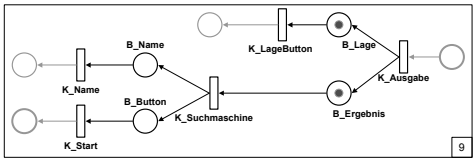
- **Änderung der Flussrichtung**: $(a,b) \in F' \Leftrightarrow (b,a) \in F$. Damit ersetzt *F'* die alte Flussrelation *F*.
- **Startmarkierung**: Weiterhin wird m_0 re-definiert, so dass nun für Instanzen *t* deren Komponententypen, die anhand des XSemL-Files als essentiell definiert werden können, gilt: $M_0([t.farbe]) = \{\{farbe\}\}$. Der Stelle wird eine erhaltende Marke mit der Farbmenge *farbe* hinzugefügt. Die Stelle mit dem Namen *[t.farbe]* wurde dabei schon in Abschnitt 8.3.1 definiert.
- **Wichtigkeit**: Die Wichtigkeitsfunktion *I* wird definiert wie folgt. Falls eine Quelle t als essentiell beschrieben ist, gilt $I([t.farbe]) = essentiell$.
- **Beschreibung der Übergänge von m nach m' nach Schaltung**:

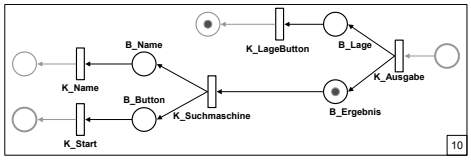
$$m'(s) = \begin{cases} m(s), \text{falls } s \notin t\bullet \wedge s \notin \bullet t \\ m(s) \cup \{F\}, \text{falls } (s,F) \in getOps(sb) \\ m(s) - m^*, \text{mit } m^* = getMark(s,f), \text{falls } (s,f) \in sa \end{cases}.$$

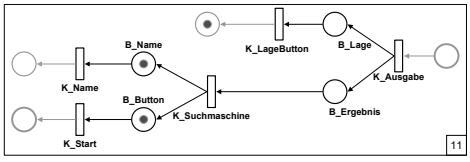
Die Definition des Schaltvorgangs zeigt, dass bei der Rückrichtung nicht mehr zwischen dem AND und dem OR als Operator zwischen den Schaltbedingungs-Operanden unterschieden wird. Stattdessen wird auf der Rückrichtung implizit ein AND angenommen. Dies entspricht der Idee, dass nicht der tatsächliche Eventfluss simuliert wird, sondern lediglich geprüft wird, ob eine Möglichkeit besteht, alle essentiellen Events zu verbrauchen. Da dies bei einem AND ebenso der Fall ist wie bei einem OR, ist diese Unterscheidung beim Rückfluss nicht notwendig.

Auf diese Weise können sowohl für den Eventfluss als auch für den Eventrückfluss die gleichen Prüfverfahren angewendet werden. Damit ergibt sich für das in Abbildung 33 eingeführte Beispiel:









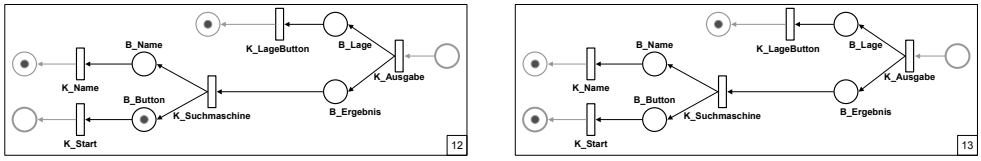


Abbildung 35: Invertiertes ICS und Rückfluss

Zusammenfassend wurde in den vorigen Abschnitten gezeigt, wie sich eine Eventfluss-Analyse auf einer Komposition durchführen lässt. Zu Beginn wurde dieser Ansatz intuitiv eingeführt. Anschließend wurde eine formale Beschreibung sowohl eines Integrity Check Systems – einer Erweiterung des gefärbten Petri-Netzes –, einer Abbildung von einer gegebenen Komposition auf ein ICS als auch die durch Schaltregeln gegebene Dynamik geliefert.

8.4 Diskussion von EFI in Bezug auf Nutzbarkeit in Komponentenarchitekturen

Die formale Korrektheit des Analyseverfahrens lässt sich unter Nutzung schon vorhandener Ergebnisse leicht zeigen (siehe hierzu [Baumgarten, 1990]), die entsprechenden Ansätze wurden in den vorigen Abschnitten dargestellt. Die Korrektheit der Abbildung ergibt sich direkt aus der Konstruktion der Abbildungsregeln.

Weiterhin ist jedoch für die hier geführte Diskussion notwendig zu prüfen, inwieweit ein Konzept wie EFI geeignet ist, um komponentenbasierte Anwendungen zu analysieren bzw. die Verwendung von Komponenten in bestimmter Art zu ermöglichen, zu steuern oder gar zu garantieren.

Häufig werden Komponenten zum Zweck der Entwicklung einer Applikation oder einer Applikationsfamilie entwickelt. In diesem Fall existiert zu Beginn das mentale Modell eines Gesamtsystems. Dieses wird dann in einzelne funktionale oder semantische Einheiten (Komponenten) zerlegt. Daraus resultiert dann die Flexibilität der Komponentenarchitekturen, denn diese möglichst unabhängigen Komponenten lassen sich miteinander kombinieren, neue Komponenten können dem bestehenden Komponenten-Set hinzugefügt werden und so die mögliche Funktionalität erweitern (siehe Kapitel 2.2). Um also die Nutzbarkeit von EFI allgemein evaluieren zu können, muss die Zerlegung einer Anwendung in Komponenten und damit die mögliche Komposition betrachtet werden. Wiederkehrende Kompositionstechniken können dann daraufhin analysiert werden, ob EFI bei geeignet beschriebenen Integritätsbedingungen helfen kann, Fehler bei solchen Kompositionen zu finden.

Ein anderer Weg liegt darin, von Entwicklern häufig verwendete Entwurfsmuster (Patterns) zu untersuchen. Folgt man der Diskussion in [Szyperski, 1998] dann sind es verschiedene Techniken, die bei der Software-Entwicklung für eine Wiederverwendbarkeit in Frage kommen. Dazu gehören unter anderen die Wiederverwendung einer Programmiersprache. Aber auch Entwurfsmuster, Schnittstellen und Frameworks sind zu betrachten. Die hier geführte Diskussion um anpassbare Software setzt im wesentlichen auf diese Wiederverwendung einzelner Komponenten in unterschiedlichen Kontexten auf (siehe Kapitel 2). Der Zusammenhang zwischen Frameworks und Patterns wird ebenfalls in [Szyperski, 1998] beschrieben. Komponenten-Frameworks zeichnen sich demnach u. a. dadurch aus, dass unterschiedliche Komponenten innerhalb eines Framework gemeinsame Schnittstellen verwenden. In vielen Fällen[25] liegen der Entwicklung eines Frameworks schon konkrete

[25] für die hier diskutierten trifft dies zu

Anwendungsfelder oder Anwendungsfälle zugrunde. Weiterhin kommen beim Design eines Frameworks verschiedene Patterns zum Einsatz. Dabei wird implizit angenommen, dass die einem Framework zugehörigen Komponenten in der vorgesehenen Weise verwendet werden. Abweichungen von dieser Norm liegen dann in der Verantwortung des Komponisten (Entwicklers).

Der Schwerpunkt dieser Arbeit liegt jedoch auf der Unterstützung von Endanwendern beim Anpassen. Unter diesem Gesichtspunkt ist zu prüfen, inwieweit die EFI dazu geeignet ist die durch die während der Entwicklung verwendeten Patterns vorgegebene Interaktion der Komponenten zu sichern. Eine solche Untersuchung kann dann auf allgemeine Weise die Untersuchung einer Vielzahl von Applikationen ersetzen.

Dabei ist zu beachten, dass Patterns im Rahmen der komponentenbasierten Software-Entwicklung [Brown et al., 1999] nicht gleichzusetzen sind mit denen in der objekt-orientierten Entwicklung (OOD). Nichtsdestotrotz sind im Bereich der Interaktion oder der Erzeugung zwischen Komponenten Parallelen zur objekt-orientierten Welt zu sehen.

Da die Diskussion hier jedoch noch in der Entstehung begriffen ist [Eskelin, 2003], kann hier auf bekannte Patterns aus dem Bereich der OOD zurückgegriffen werden. Dabei wird die Verwendung der jeweiligen Patterns dediziert motiviert.

Eine ausführliche Einführung in Design Patterns (bzgl. OOD) geben Gamma et al. [Gamma et al., 1995]. Patterns beschreiben die Abhängigkeiten unterschiedlicher Klassen zueinander und die Interaktion von Objekten dieser Klassen. Dieser Ansatz lässt sich auf Komponenten übertragen, wobei abstrakte Klassen hier eine weniger große Rolle spielen und deswegen in erster Linie Klassen, die später instanziiert werden, zu untersuchen sind. Exemplarisch für die Analyse wird hier das Strategy Pattern vorgestellt, da es die Möglichkeiten von EFI in sehr generischer Weise zeigt. Die Motivation des Strategy Patterns besteht darin, verschiedene Lösungsstrategien für ein Problem durch eine gemeinsame Schnittstelle darzustellen. Dadurch können die einzelnen Strategien je nach Anwendungskontext dynamisch ausgetauscht werden. Als Beispiel nennt Gamma einen Algorithmus zum Zeilenumbruch. Dieser kann verschiedene Strategien verfolgen und damit verschiedenen Anforderungen genügen. Er kann sehr schnell und effizient umbrechen (SimpleCompositor), dafür aber kein perfektes Schriftbild erzeugen. Umgekehrt kann er ein sehr ausgeglichenes Schriftbild erzeugen (TEXCompositor), dafür aber in einer höheren Komplexitätsklasse liegen. Ein weiterer Algorithmus (ArrayCompositor) könnte so umbrechen, dass jede Zeile eine feste Anzahl von Objekten erhält, wie dies beispielsweise bei der gleichmäßigen Anordnung graphischer Icons nötig ist.

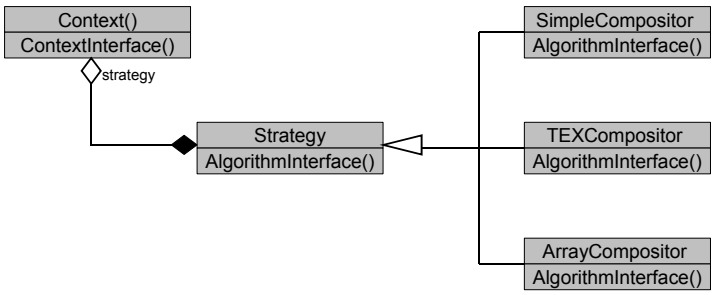


Abbildung 36: Strategy Pattern

In Abbildung 36 ist das Klassendiagramm zu obigem Beispiel aufgezeichnet. Der Context hält das Objekt, in diesem Beispiel den umzubrechenden Text. Die Strategy ist eine abstrakte Klasse, die ein allgemeines Interface zum Textumbruch zur Verfügung stellt. Die ConcreteStrategies sind Söhne dieser Klasse, welche ihrerseits jeweils eine Umbruchstrategie implementieren.

Der Färbungsbaum zu diesem Pattern wird eine Farbe für die Kontextinformation, also den Text enthalten. Eine Marke dieser Farbe wird von den konkreten Strategien erwartet. Diese Marke ist für die Strategien essentiell. Nach der Verknüpfung eines Textobjektes mit einem Objekt einer Umbruchstrategie, kann eine Marke eines richtigen Typs von ersterem zu letzterem fließen. Das Objekt der Umbruchstrategie kann auch die notwendige Marke erhalten. Es können wahlweise weitere Umbruchstrategien instanziiert werden, die bei Erstellung des Kontextobjekts nicht bekannt sein mussten. Sie müssen jeweils essentiell eine Marke des Typs Text erwarten.

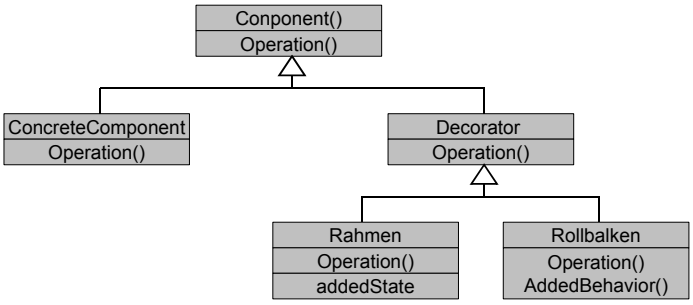


Abbildung 37: Decorator Pattern

Ein weiteres Beispiel stellt das Decorator Pattern dar. Mit Hilfe dieses Patterns können dynamisch zusätzliche Verantwortlichkeiten an Objekte geknüpft werden. Beispielsweise kann damit die Darstellung eines Fensterinhalts in einer graphischen Benutzeroberfläche um Funktionalität erweitert werden (einem Rahmen oder einem Rollbalken). Die Abhängigkeiten zwischen den beteiligten Objekten sind in Abbildung 37 zu sehen. Die abstrakte Komponente Component stellt eine Funktionalität zur Verfügung, in diesem Beispiel eine visuelle Darstellung. Die konkrete Komponente implementiert die Grundfunktionalität dieser Darstellung, also das Darstellen von Text. Der Decorator selbst ist wieder abstrakt und stellt eine Schnittstelle zu den weiteren Funktionen zur Verfügung, also dem Rollbalken oder dem Rahmen. Diese werden deshalb von ihr abgeleitet.

Mit diesem Pattern kann eine weitere Stärke von EFI demonstriert werden, die im vorigen Beispiel nicht zum Ausdruck kam: Die Möglichkeit der Vererbung von Farben im Färbungsbaum. Beim Decorator kann das dynamische Nachladen der Objekte in der Komponentenwelt auf Anpassungsoperationen durch den Benutzer abgebildet werden. In diesem Falle kann der Benutzer durch Hinzufügen von weiteren Komponenten zur Decorator-Komponente dieser neue Eigenschaften geben. Diese hinzugefügten Komponenten stellen dann „Dienstleistungen“ zur Verfügung, weshalb sie als Marken-Quellen semantisch erfasst werden. Die Hauptkomponente ist die Senke. Die Hauptkomponente erwartet einen vergleichsweise allgemeinen Markentyp, da sie recht unterschiedliche Dienstleistungen verarbeiten kann. Der Typ könnte beispielsweise den Namen „Visuelle Darstellung“ tragen. Die Dienstleistungskomponenten werden alle einen Subtyp zur Verfügung stellen, weil sie nur für spezielle visuelle Ausgaben zuständig sind, beispielsweise „Rollbalken-Darstellung“. Trotzdem ist eine Verknüpfung beider Komponenten semantisch korrekt und würde auch entsprechend bewertet, da es erlaubt ist, Subtypen an Senken höheren Typs zu versenden.

Weitere Beispiel-Analysen von Design Patterns finden sich in [Krings, 2003], sie bestätigen jedoch lediglich die hier getroffenen Aussagen. Weiterhin können auch weitere Patterns betrachtet werden. Dabei sind für diese Analyse in erster Linie Verhaltens-Patterns (Behavioral Patterns) interessant, da sie das Zusammenspiel der Komponenten untereinander darstellen.

8.5 Fazit

Das Konzept der Event Flow Integrity dient der Analyse von Nachrichtenflüssen zwischen Komponenten. Dabei wird – anders als in bestehenden Ansätzen – nicht nur die Kommunikationsbeziehung zwischen zwei Komponenten, sondern der Nachrichtenfluss in der Gesamtkomposition betrachtet. Für die Analyse wird dabei aus der Komposition selbst und den dazugehörigen Integritätsbedingungen ein erweitertes Petri-Netz konstruiert.

Zusammenfassend kann ein Durchlauf der EFI-Analyse wie folgt beschrieben werden:

1. **Bildung des ICS**: Auf Basis der von FREEVOLVE zur Verfügung gestellten Informationen über die Komposition und die in XSemL beschriebenen Integritätsbedingungen lässt sich ein ICS erzeugen, der die Komposition in abstrakter Form beschreibt.
2. **Festlegung der Anfangsmarkierung**: Dabei wird festgelegt, welche Stellen als Eventquellen mit Anfangsmarkierung dienen. Die Information darüber finden sich in den Integritätsinformationen der einzelnen in der Komposition enthaltenen Komponenten. Dort ist hinterlegt, welcher Port als „essentiell“ (required) oder „optional“ (optional) verwendet werden soll. Die Marken werden entsprechend ihrer Typisierung eingefärbt. Ebenso sind die Eventsenken zur Analyse des Rückflusses festzulegen.
3. **Prüfung auf Erreichbarkeit**: Die eigentliche Analyse erfolgt mit Hilfe des vorgestellten Algorithmus „Partielle Erreichbarkeitsanalyse“. Mit ihr wird festgestellt, ob es möglich ist, mit der vorgegebenen Anfangsmarkierung eine gültige Endmarkierung im Sinne von EFI zu erreichen. Diese Analyse bedient sich dem extern abgelegten Färbungsbaum zur Prüfung der Gültigkeit von Marken innerhalb der einzelnen Stellen.
4. **Prüfung des Rückflusses**: Hierzu werden die Transitionen invertiert und eine entsprechend andere Anfangsmarkierung (Marken in den Eventsenken) gesetzt. Die Prüfung erfolgt dann mit den unter 3. beschriebenen Techniken.

Anhand zweier Beispiele aus dem Bereich der Design Patterns wurde gezeigt, dass sich das hier beschriebene Analyseverfahren auf unterschiedlichen, typischen Interaktionsszenarien (durch Patterns beschrieben) anwenden lässt. Nicht nur die schon weiter oben motivierte transitive Analyse bei der Verarbeitung von Events innerhalb eines Komponentennetzes ist notwendig. Auch die Betrachtung von erweiterten Typenbeziehungen, die im angesprochenen Färbungsbaum strukturiert sind, ist wichtig, um bekannte Design Patterns vollständig analysieren zu können.

9 Integritätsprüfungen im Anwendungskontext - Application Templates

Bisher beziehen sich die Integritätsbedingungen vornehmlich auf Einzelkomponenten. Das gilt für die Formulierung von Constraints oder Restricted Solutions ebenso wie die Einstufung der Ports in essentielle und optionale zur Analyse der Event Flow Integrity. Selbstverständlich erzeugt dies Abhängigkeiten und Beziehungen zwischen den Komponenten innerhalb einer Komposition, wenn die Integrität geprüft wird. Die Beschreibung der Bedingungen ist jedoch ausschließlich lokal anzusiedeln. Dies schränkt die Möglichkeiten der Beschreibung von Integritätsbedingungen stark ein. Sie müssen unabhängig vom Wissen über andere Komponenten definiert werden. Es kann sein, dass Komponentenentwickler wechselseitig nichts von ihrer Arbeit wissen und so nicht die Möglichkeiten haben, auf mögliche Abhängigkeiten einzugehen.

9.1 Sammlungen von Integritätsbedingungen als Anwendungsvorlagen

Schon weiter oben wurde erwähnt, dass die Integritätsbedingungen in XSemL beschrieben werden. Dies ermöglicht den Administratoren einer FREEVOLVE-Installation, Anpassungen an den Integritätsregeln vorzunehmen. So können die Integritätsbedingungen besser auf das Anwendungsfeld abgestimmt werden. Darüber hinaus ist es möglich, Integritätsbedingungen für unterschiedliche in das System eingebundene Komponenten-Sets zu integrieren und so aufeinander abzustimmen.

Eine weitere Möglichkeit stellen die Application Templates dar. Sie definieren Integritätsbedingungen, die für eine ausgewählte Klasse von Applikationen (Kompositionen) gelten sollen. In einer FREEVOLVE-Installation können demnach mehrere Application Templates verfügbar sein. In diesen Vorlagen werden Mengen von Bedingungen ebenso beschrieben wie Bedingungen an Einzelkomponenten. Die Integritätsbedingungen werden dann entweder als eine Verfeinerung der schon vorhandenen allgemeinen aufgefasst oder können diese überschreiben (zur Vermeidung von Widersprüchen). Auf diese Weise ähnelt der Ansatz den in Kapitel 4.3 diskutierten BeanPlans [Birngruber und Hof, 2000].

Des weiteren kann jedes Application Template den Färbungsbaum erweitern. Auf diese Weise wird die oben geforderte Flexibilität und Präzision bei der Definition der unterschiedlichen Typen von Events (Abschnitt 8.1) erreicht.

Grundsätzlich können diese Application Templates auch lediglich dazu genutzt werden, um die in den allgemeinen Regeln vorhandenen Fehlermeldungen zu redefinieren. Hierdurch wird der Anwendungsbezug erhöht und damit der Zugang zum Anpassen erleichtert (siehe hierzu [Repenning und Summer, 1995] oder [Lewis und Olson, 1987]).

Eine Möglichkeit, Application Templates in FREEVOLVE zu integrieren, ist die hierarchische Komposition (siehe Kapitel 3). Mit Hilfe von abstrakten bzw. zusammengesetzten Klassen lassen sich Integritätsbedingungen für alle in ihr enthaltenen Komponenten formulieren. Auf diese Weise können abstrakte Klassen verwendet werden, um domänenspezifisch oder applikationsspezifisch Integritätsregeln zu verwalten. In einem solchen Fall ist eine abstrakte Komponente von sich aus leer. Wird sie geladen, dann gelten für sie alle in ihr enthaltenen Integritätsbedingungen.

Bei Start des FREEVOLVE-Systems gelten alle Regeln für die Komponenten, die in einer aktuellen Applikation enthalten sind. Da die Integritätsbedingungen unabhängig von der Komposition gespeichert werden, ist es möglich, Application Templates beliebig nachzuladen (jeweils nur eine[26]) oder auszutauschen. Die im Application Template enthaltenen Regeln gelten dann für die Applikation. Durch dieses Vorgehen wird darauf Rücksicht genommen, dass sich das mentale Modell der Applikation während des Anpassungsvorgangs ändern kann. Dann ist es sinnvoll, dass eine passende Applikationsvorlage verwendet wird, um den Anpassungsprozess möglichst optimal durch eine Integritätsprüfung zu unterstützen.

9.2 Fazit

In den Kapiteln 6 bis 9 wurden die Erweiterungen zu FREEVOLVE vorgestellt, wie sie in dieser Arbeit entwickelt worden sind. Die Constraint-based Integrity erlaubt es logische Bedingungen für eine Komposition zu formulieren. Sie werden lokal (bezogen auf eine Komponente) überprüft. Ein erweitertes Konzept stellen hier Restricted Solutions dar, die die Constraints um semi-automatische Lösungstechniken erweitern und gleichzeitig Inkonsistenzen im Regelwerk verhindern. Mit den Restricted Solutions ist die Definition der Solutions aber nicht nur widerspruchsfrei geworden, sondern erfasst auch globale semantische Fehler. Dieser Übergang von einer rein lokalen Prüfung (Constraints) hin zu einer globalen Fehlererkennung ist ein wichtiger Schritt.

Mit der Event Flow Integrity wurde der Semantikprüfung des Komponentenframeworks ein weiteres Konzept hinzugefügt, das ein neues und mächtiges Ausdrucksmittel zur Darstellung und Auswertung semantischer Informationen in Bezug auf Nachrichtenflüsse innerhalb einer Komposition schafft. Diese Analyse reicht über die Betrachtung einer Einzelkomponente und ihre Verbindungen zu den Nachbarkomponenten hinaus. Hierbei steht im Vordergrund des Interesses, die Informationen, die Komponenten untereinander austauschen zu betrachten und zu sichern, dass diese Nachrichten verwendet werden können. Dies geschieht in der Annahme, dass eingeschätzt werden kann, welche der zur Verfügung gestellten Ports einer Komponente essentiell für deren Funktionalität ist.

Zu Beginn wurden verschiedene Forderungen an ein Integritätskonzept zur Unterstützung von Anpassungsmechanismen für Endanwender gestellt. Dabei ging es vordringlich darum, dass möglichst alle Anpassungsoperationen unterstützt werden bzw. sich durch die Formulierung geeigneter Integritätsbedingungen unterstützen lassen. Der hier vorgestellte Ansatz geht aus von den Möglichkeiten, die Komponentenarchitekturen bieten und untersucht, welche Eigenschaften wie zu unterstützen sind. Die direkte Verbindung zwischen Eigenschaften von Komponentenarchitekturen und möglichen Anpassungsoperationen schafft hier die Verbindung zur Benutzerunterstützung, sie konnte auch formal gezeigt werden.

[26] Dies stellt keine Einschränkung des Konzepts dar. Konzeptuell ist auch ein mengenorientierter Ansatz zur Verwaltung mehrerer Application Templates oder eine hierarchische Strukturierung der Application Templates möglich. Jedoch wurde in der Implementierung (siehe Kapitel 10) bisher darauf verzichtet.

Im Bereich der Softwaretechnik schafft der komponentenbasierte Ansatz eine deutliche Beschleunigung des Entwicklungsprozesses und damit einhergehend eine qualitative Verbesserung der resultierenden Produkte. Durch die hier vorgestellten Integritätskonzepte ergeben sich nochmals Optimierungspotentiale für den Software-Entwicklungsprozess. Zum einen lassen sich die hier vorgestellten Techniken, die vor dem Hintergrund der Unterstützung von Endbenutzer-Anpassbarkeit entworfen wurden, auch in der Softwareentwicklung einsetzen. Hier können sie entscheidend dazu beitragen, Tests automatisiert durchführen zu können. Solche Tests beziehen sich dann nicht mehr nur auf syntaktische Korrektheit, sondern schaffen es Teile der den Komponenten innewohnenden Semantik formal zu beschreiben und so Tests auch auf dieser (semantischen) Ebene durchführen zu können. Zum anderen kann auf diese Weise der Bereich des End-User Development weiter ausgebaut werden. Hier ist die Idee, über einfache Anpassungen hinaus Software von Endanwendern selbst entwickeln zu können, um eine bessere Integration in die Anwendungsfelder zu erreichen. Dafür wurden vor allen Dingen zwei Forderungen an das Integritätskonzept gestellt: Einfachheit und damit Verständlichkeit und Flexibilität.

Ein wichtiger Punkt in Bezug auf die Lernförderlichkeit ist die Verständlichkeit der Regeln. Die beiden Basiskonzepte „Constraints-based Integrity“ und „Event Flow Integrity“ sind einfach zu verstehen, jedoch sehr mächtig. Die Verständlichkeit der Regeln ist maßgeblich abhängig von guten Fehlerbeschreibungen und einer sinnvollen Integration in die Anpassungsumgebung. Eine Beschreibung dieser Arbeiten und eine darauf bezogene Nutzer-Evaluierung ist Inhalt des folgenden Kapitels.

Eine weitere Forderung bezog sich auf die Flexibilität des Integritätsmechanismus. Hier sollte bei der Nutzung einer Integritätsprüfung als Anpassungsunterstützung die Anwendungsdomäne berücksichtigt werden. Durch die Beschreibung der Integritätsbedingungen in einer für Administratoren gut verständlichen Sprache (XSemL) ist es möglich, diese entsprechend den Bedürfnissen in einem Anwendungsfeld anzupassen. Zudem lassen sich außerdem mit Hilfe von Application Templates für Anwendungen maßgeschneiderte Integritätsbedingungen entwerfen.

10 Integration der Integritätsprüfung in die Anpassungsumgebung und Evaluation

Die Integrationsprüfung, die in den Kapiteln 6 bis 9 vorgestellt wurde, dient der Unterstützung der Anwender beim Anpassen ihrer komponentenbasierten Applikationen. Zu diesem Zweck wurde in Kapitel 5 eine Anpassungsumgebung vorgestellt. Sie erlaubt es, die unterschiedlichen Aspekte einer Komposition zu untersuchen und Änderungen daran vorzunehmen. Dieses Kapitel dient nun in drei Schritten der Integration beider Vorarbeiten ineinander. Es wird dargelegt, wie sich eine Integritätsprüfung in den TailorClient integrieren lässt, um den Anpassungsvorgang einfacher und fehlerärmer zu gestalten.

Dazu gliedert sich dieses Kapitel in vier Abschnitte: Nachdem in Abschnitt 10.1 einige grundlegende Überlegungen zur Integritätsprüfung als Anpassungsunterstützung ausgeführt werden, geht Abschnitt 10.2 drauf ein, wie sich die Erweiterung des TailorClient um eine Integritätsprüfung für die Benutzung auswirkt. Es geht hier vordringlich um die Darstellung von Fehlern, wie sie von der Integritätsprüfung erkannt werden, bei der Komposition im GUI. Abschnitt 10.3 stellt dann die Implementierung in den Vordergrund. Hier sind verschiedene Änderungen in der Architektur von FREEVOLVE nötig, damit die in den Kapiteln 7 und 8 beschriebenen Prüfalgorithmen auf alle benötigten Informationen zugreifen können. Umgekehrt ist hier natürlich auch die Interaktion mit der Benutzungsschnittstelle zu untersuchen. Abschnitt 10.4 schließlich stellt die Ergebnisse einer zweiten Benutzerstudie dar. Nachdem in Kapitel 5 die allgemeine Funktionalität des TailorClient schon untersucht worden ist, geht es hier schwerpunktmäßig darum herauszufinden, inwieweit eine Integritätsprüfung dazu geeignet ist, Benutzer bei ihren Anpassungsaufgaben zu unterstützen und Fehler zu vermeiden.

10.1 Grundsätzliche Überlegungen

In Kapitel 5 wurde die Funktionalität des TailorClient bereits weitgehend beschrieben und evaluiert. Im Folgenden steht deswegen die Integration der Integritätsprüfung im Vordergrund. Aus Sicht der Benutzer hat die Integritätsprüfung zum Ziel, den Anpassungsvorgang zu unterstützen. Daraus lassen sich verschiedene Unteraspekte ableiten:

- **Hinweis auf Fehler**: Häufig hindert Benutzer die Angst, eine Applikation zu „zerstören“, am Anpassen. Dies und die Tatsache, dass der Anpassungsvorgang an sich eine nicht alltägliche Arbeit ist, sollte das System Fehler weitest möglich erkennen und anzeigen. Obwohl eine solche Fehlerprüfung wie schon in Kapitel 3 diskutiert niemals die semantische Intention eines Benutzers beim Anpassen und damit die Korrektheit der Komposition vollständig erfassen kann, ist auch eine partielle Fehlerprüfung hilfreich.
- **Korrekturvorschläge bei Fehlern/Optimierungsvorschläge/Nutzungshinweise**: Auf Basis der Fehlererkennung sollte das System Korrekturvorschläge anbieten. Sie können die Benutzer anleiten, Fehler zu entfernen. Im gleichen Maße ist es hilfreich, den einzelnen Komponenten Nutzungshinweise beizufügen, wie sie auf Basis der Intuitive Integrity generiert werden. Eine vollständig automatische Korrektur ist hier nicht angestrebt.

- **Verbesserung des Verständnisses für die aktuelle Komposition**: Die Darstellung der Zusammenhänge zwischen den Einzelkomponenten und der Nachrichtenflüsse zwischen ihnen soll das Verständnis für die aktuelle Komposition fördern. Die Integritätsprüfung gibt zudem Hinweise für die Nutzung von Einzelkomponenten und analysiert die Nachrichtenflüsse. Fehler werden angemerkt. Auf Basis solcher Fehlerbeschreibungen wird die Funktionalität sowohl der Einzelkomponenten als auch der Gesamtkomposition klarer.
- **Verbesserung des Verständnisses für die Grundsätze der komponentenbasierten Anpassbarkeit**: Ein wichtiges Ziel der Integritätsprüfung ist, das Verständnis der Benutzer für die komponentenbasierte Anpassbarkeit zu erhöhen. Durch Fehlerbeschreibungen und Visualisierungen der Kommunikationsflüsse innerhalb einer Komposition kann dies geleistet werden. Die Benutzung des TailorClient soll helfen, Verständnis für die Idee komponentenbasierter Software zu entwickeln. Auf diese Weise werden Benutzer nicht nur bei der Bedienung des TailorClient sicherer. Sie werden zudem im Umgang mit ihrem System im allgemeinen geschult und sind in der Lage, es effektiver einzusetzen.

Die genannten Aspekte stehen bei der Gestaltung der Erweiterung des Anpassungswerkzeugs im Vordergrund. Eine weitere wichtige Entscheidung ergibt sich aus der Frage, wann die Integritätsprüfung aktiviert wird und durch wen. Hierfür gibt es zwei Ansätze, die in verschiedenen Umgebungen unterschiedlich zum Einsatz kommen:

- **Explizite Prüfung**: Die explizite Prüfung wird vom Anpassenden gestartet. Dementsprechend gibt es nur beschränkte Zeiträume, in denen sie aktiv ist. Der Zustand der Komposition zum Startzeitpunkt der Prüfung ist dann ausschlaggebend für das Ergebnis. Grundlage der Prüfung ist dabei entweder die gesamte Komposition oder die seit der letzten Prüfung geänderten Aspekte. Ein bekanntes Beispiel hierfür ist die Prüfung der Steuerungselektronik eines Flugzeugs vor dem Start, die von Piloten und Kopiloten durchgeführt wird.
- **Implizite Prüfung**: Die implizite Prüfung ist während des gesamten Anpassungsvorgangs aktiv und unterstützt die Anpassenden aktiv. Dabei muss jede Veränderung an der Komposition registriert und auf ihre Korrektheit im Sinne der Integritätsbedingungen geprüft werden. Die meisten Kontrollinstrumente in Autos (Ölstand, Batterieladung etc.) funktionieren auf diese Weise.

Für die in den TailorClient zu integrierende Integritätsprüfung wurde der Ansatz der expliziten Prüfung gewählt. Sie hat verschiedene Vorteile. Zum einen ist die Intention der Anpassenden während eines Anpassungsvorgangs, der als Transaktion betrachtet werden kann, nur sehr schwierig zu erfassen. Ein einzelner Anpassungsschritt ist aber in den meisten Fällen kontextfrei betrachtet wenig sinnvoll. Allerdings erzeugt die Prüfung dann sehr viele Fehlermeldungen, die unnötig sind und eher verwirren und behindern als unterstützend wirken. Fügt ein Benutzer beispielsweise eine Komponente, die (durch Integritätsbedingungen festgelegt) in bestimmter Weise verbunden oder parameterisiert werden muss in eine Applikation ein, so würde diese Aktion eine Reihe von Fehlermeldungen nach sich ziehen („Komponente muss verbunden werden etc.“). Hier ist es nicht sinnvoll, schon während des noch nicht abgeschlossenen Änderungsvorgangs eine Prüfung durchzuführen. Auf der anderen Seite ist für das System aber nicht erkennbar, wann die Transaktion abgeschlossen ist. Eine explizite Aufforderung zur Prüfung der Komposition ist hier hilfreicher. Sie deutet gleichzeitig das Ende einer Anpassungs-Transaktion aus Sicht des Benutzers an.

Weiterhin spielt bei der Nutzung einer Integritätsprüfung als Kontrollmechanismus neben dem Wunsch, fehlerfreie Applikationen zu erzeugen, der Lerneffekt durch Hinweise auf Fehler eine wichtige Rolle. Der Wechsel zwischen dem Anpassungsvorgang und der Analyse des Systems sollte deswegen bewusst erfolgen und wird durch eine explizite Aufforderung zur Prüfung markiert.

10.2 Beschreibung der Funktionalität

Der TailorClient bleibt von seiner Funktionalität her unverändert. Die Integritätsprüfung ist an verschiedenen Stellen in das System integriert. Da sie vom Anpassenden gestartet werden muss, ist dafür ein weiterer Menüpunkt im Hauptmenü eingerichtet worden. Die Fehler selbst werden in allen unterschiedlichen Sichten angezeigt und sind – so nahe dies möglich ist – an die Stelle innerhalb der Komposition gebunden, an der der Fehler aufgetreten ist. Um einen Information Overload zu vermeiden, werden die Fehlermeldungen nicht komplett angezeigt, sondern lediglich Fehler-Icons an den entsprechenden Stellen eingeblendet. Abbildung 38 zeigt verschiedene Sichten des TailorClient. Durch Kreise sind dort die Fehler-Icons markiert. Die Icons zeigen lediglich an, dass an den entsprechenden Stellen Fehler aufgetreten sind. Die Fehlerart wird erst durch erneute Interaktion mit dem System angezeigt. Auf diese Weise wird als erste Information dargelegt, ob und wo ein Fehler aufgetreten ist. Auch die Abwesenheit von Fehlern lässt sich so schnell überblicken. Wichtig ist zu bemerken, dass Fehler entsprechend der hierarchischen Struktur einer Komposition erkennbar sind. In Abbildung 38.2 wird dies deutlich: In fünf Komponenten, die zur Chat-Server-Komponente gehören, sind Fehler mit unterschiedlichen Schweregraden aufgetreten. Daraus abgeleitet wird auch die Chat-Server-Komponente mit einem Fehler-Icon markiert.

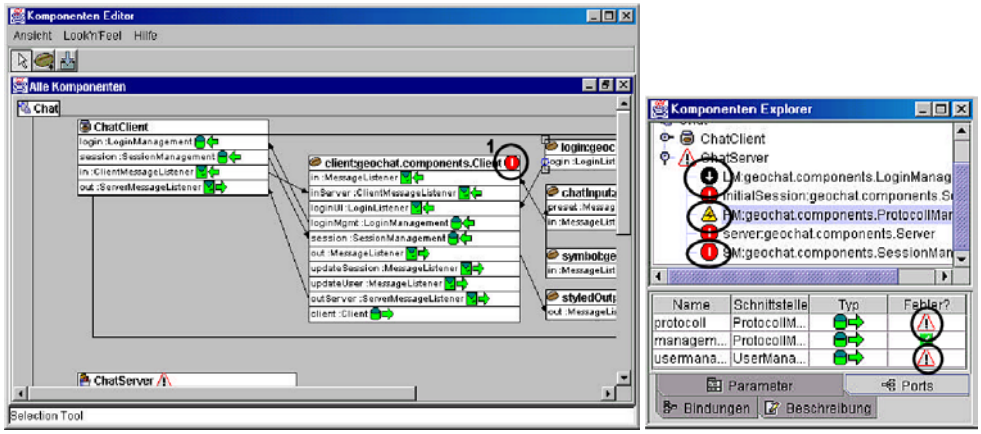


Abbildung 38: Fehler in den unterschiedlichen Sichten

Es werden unterschiedliche Fehler-Icons je nach Wichtigkeit eines Fehlers verwendet. Tabelle 12 gibt eine Übersicht über die verschiedenen Symbole.[27] Der Fehlerlevel wird in den Integritätsbedingungen definiert (vgl. Kapitel 7) und ist insofern gleichzeitig eine Wichtigkeits-Einstufung der Regel selbst.

[27] Andere Symbole – wie z. B. rote Kreise mit dem Schweregrad des Fehlers als Zahl in ihnen – können leicht integriert werden. In den Benutzertests (s.u.) zeigte sich, dass die gewählten Symbole nach einer kurzen Einführung gut verstanden und wiedererkannt wurden.

Level	Icon	Bedeutung
0		Es existiert eine Integritätsbedingung, aber kein Fehler ist aufgetreten.
1		Warnung.
2		Geringer Fehler.
3		Fehler.
4		Größerer Fehler.
5		Schwerwiegender Fehler.
6		System-Fehler.

Tabelle 12: Fehlerlevel

Mit einem Mausklick auf die Fehler-Icons erhält man detailliertere Informationen (siehe Abbildung 39). Die dort gezeigten Texte sind in den Integritätsbedingungen hinterlegt. Falls dort keine Fehlermeldungen vorliegen und auch keine Korrekturvorschläge definiert sind, werden diese Hilfstexte wie in Kapitel 7 beschrieben automatisch generiert.

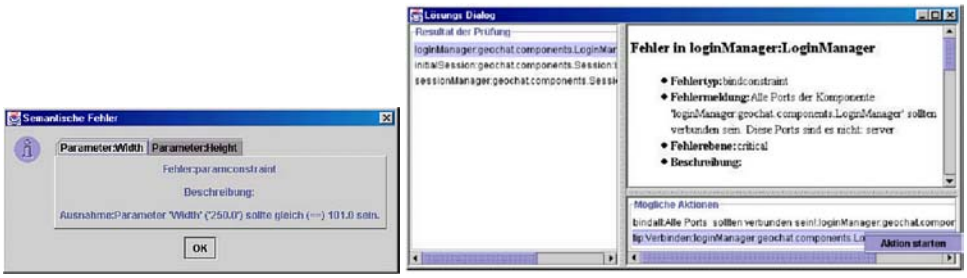


Abbildung 39: Möglicher Fehler- und Lösungsdialog

Zusätzlich zu den kontextgebundenen Fehlermeldungen ist es möglich, sich das Ergebnis der Integritätsprüfung im Überblick anzuschauen. Diese Darstellung wird aus dem Hauptmenü heraus aktiviert und stellt alle von den einzelnen Prüfverfahren erzeugten Meldungen übersichtlich dar. Um einen möglichst guten Überblick zu erhalten, ist die Formatierung in HTML realisiert. Die Prüfalgorithmen erzeugen zu diesem Zweck XML-Dokumente, die in HTML gewandelt dargestellt werden.

Die Hinzunahme dieser Sicht auf die Fehler entspricht auch der schon oben (Abschnitt 10.1) angeführten Überlegung, dass die Integritätsprüfung dazu dienen soll, die Funktionalität der Komponenten und des Anpassungswerkzeugs besser zu verstehen. Aus diesem Grund ist diese Übersichtsdarstellung sehr wertvoll. Sie dient nicht in erster Linie der Korrektur von Fehlern, sondern erhöht das Verständnis für das Gesamtsystem.

10.3 Implementierung der Integritätsbedingungen

Die folgende Beschreibung dient dazu, einen Überblick über die notwendigen Implementierungsarbeiten zu gewinnen. Diese Beschreibung ist nicht vollständig, es werden lediglich die Schlüssel-Klassen und ihre Bedeutung für die Integritätsprüfung aufgeführt. Sehr viel detailliertere Beschreibungen sind in [Krüger, 2002] und [Krings, 2003] nachzulesen. Aus software-technischer Sicht muss die gesamte Prüflogik in den bestehenden TailorClient integriert werden. Diese Arbeiten lassen sich in fünf Bereiche aufspalten:

- **Generierung von Integritätsbedingungen**: Benötigt werden Werkzeuge zur Definition von Integritätsbedingungen.
- **Speicherung der für die Integritätsprüfung notwendigen Bedingungen und Informationen**: Die Integritätsbedingungen beziehen sich jeweils auf einzelne Komponenten einer Komposition. Aus diesem Grund sollten sie auch lokal dort zugeordnet werden. Gleichzeitig muss es den Prüfverfahren möglich sein, auf diese Informationen zuzugreifen.
- **Integration der Prüfverfahren**: Für die Prüfverfahren müssen eigene Klassen definiert werden. Sie werden gestartet, lesen dann – entweder komplett oder iterativ – die Integritätsbedingungen der aktuellen Komposition aus und werten diese aus. Anschließend müssen die Ergebnisse der Prüfung am GUI dargestellt werden.
- **Erweiterung des TailorClient um Befehlselemente**: Die Integritätsprüfung muss über das vorhandene GUI des TailorClient aktiviert werden können. Eine Erweiterung um entsprechende Befehle muss also implementiert werden. Ebenso ist es notwendig, Application Templates auswählen zu können.
- **Bereitstellung eines API zur Anzeige von Fehlern und Hilfen**: Wie schon angesprochen, muss es möglich sein, die Ergebnisse der Integritätsprüfung am GUI zu visualisieren. Zu diesem Zweck ist ein API zu definieren und zu implementieren, dass von den Integritätsprüfungsklassen angesprochen werden kann.

Diese fünf Bereiche sind für eine geeignete Integration essentiell und sollen deswegen im Folgenden eingehender beschrieben werden. Dabei ist es für das Verständnis nur an wenigen Stellen notwendig, auf Implementierungsdetails einzugehen. Sie werden um der Übersichtlichkeit willen nur skizziert.

10.3.1 Generierung von Integritätsbedingungen

Integritätsbedingungen werden – in der Regel von den Komponenten-Entwicklern – in XML beschrieben. Zu diesem Zweck wurde eine DTD (siehe Anhang B) entworfen. Sie kann von XML-Editoren (z. B. Epic [Arbortext, 2003]) verwendet werden, um leicht Bedingungen zu definieren, die syntaktisch korrekt von der Integritätsumgebung ausgewertet werden können.

10.3.2 Speicherung der Informationen (Regeln, EFI)

Der Kompositionsbaum einer FREEVOLVE-Anwendung wird gebildet durch Objekte der Klasse ComponentTreeNode. Die gesamte Applikation ist damit zugreifbar über die entsprechende ComponentTreeNode-Instanz der Superkomponente. Jede ComponentTreeNode-Instanz kann wiederum beliebig viele Kinder haben. Hat sie keine Kinder, so ist die dazugehörige Komponente eine instanziierte.

Die ComponentTreeNodes verwalten lediglich die Struktur der Anwendung. Die Komponenten selbst sind jedoch darüber zugreifbar. Insofern können die ComponentTreeNodes als Proxies für die Verwaltung der Komposition betrachtet werden. Eine vereinfachte Darstellung der Zusammenhänge zwischen den einzelnen Klassen zeigt Abbildung 40.

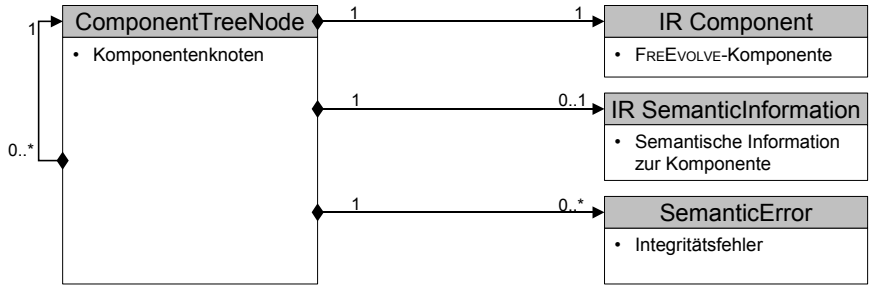


Abbildung 40: ComponentTreeNode

Die Speicherung (s.u.) der Integritätsinformationen erfolgt in zwei über die ComponentTreeNodes zugreifbare Objekte: Dies sind zum einen Objekte, die das Interface SemanticInformation implementieren. Sie enthalten die Informationen, die extern in XML-Dateien vorliegen. Zum anderen dienen Objekte der Klasse SemanticError dazu, die möglicherweise in diesem Knoten der Komposition aufgetretenen Fehler zu speichern.

Das Interface SemanticInformation wird von der Klasse SemanticInformation implementiert (siehe Abbildung 41). Objekte dieser Klasse haben Zugriff auf Objekte, die eigentlich die den Strategien entsprechenden Informationen enthalten. Zudem hat die Klasse SemanticInformation eine Referenz auf den EFITree. Der als static definierte Färbungsbaum wird von der EFI-Prüfung benötigt. Er muss im Gesamtsystem nur einmal vorhanden sein, ist also nicht direkt an eine Komponente gebunden.

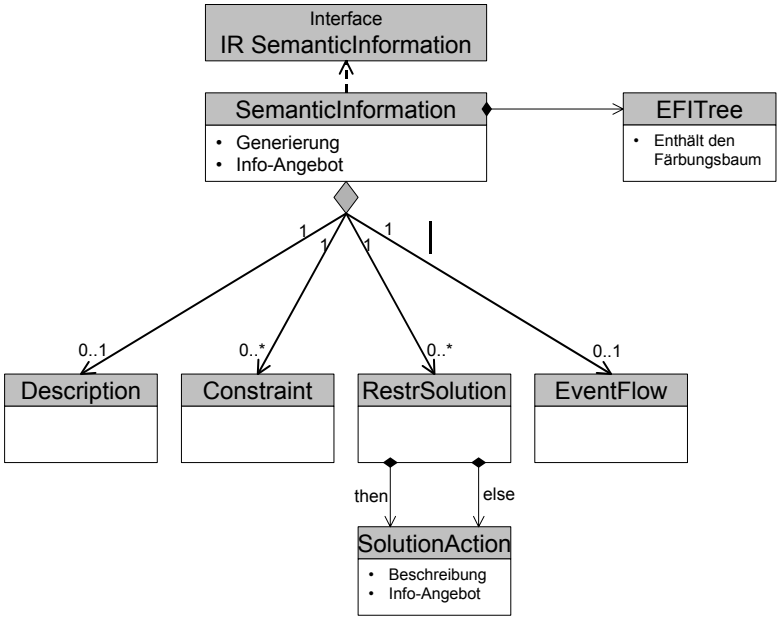


Abbildung 41: SemanticInformation mit speziellen Integritätsinformationen

Ein weiterer wichtiger Schritt ist festzulegen, auf welche Weise die Integritätsbedingungen in das System geladen werden. Getriggert durch die Auswahl eines neuen Application Templates oder durch den erstmaligen Start der Integritätsprüfung startet das Integrity-Environment – die Hauptklasse der Integritätsprüfung (siehe Abbildung 42) – den Ladevorgang.

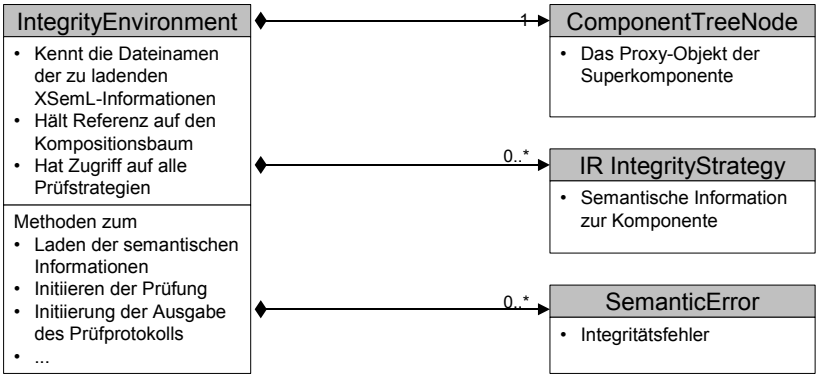


Abbildung 42: Das IntegrityEnvironment

In diesem Fall werden die einzelnen XSemL-Dateien geöffnet und analysiert. Entsprechend ihren Einträgen werden sie dann den einzelnen den ComponentTreeNodes zugehörigen SemanticInformation-Objekten als neue Objekte wie in Abbildung 41 zu sehen hinzugefügt. Außerdem wird der Färbungsbaum, der ebenfalls in XSemL vorliegt, geparst und ein entsprechendes neues Objekt erzeugt.

Um auf die einzelnen ComponentTreeNode-Objekte zugreifen zu können, hat das IntegrityEnvironment eine Referenz auf den ComponentTreeNode der Superkomponente, die abstrakte Komponente, die die gesamte Applikation repräsentiert.

10.3.3 Implementierung der Prüfverfahren

Das IntegrityEnvironment ist ebenso zuständig für die Durchführung einer Prüfung. Zu diesem Zweck hat das IntegrityEnvironment Referenzen auf alle Integritätsprüfungsstrategien. Eine Übersicht darüber zeigt Abbildung 43:

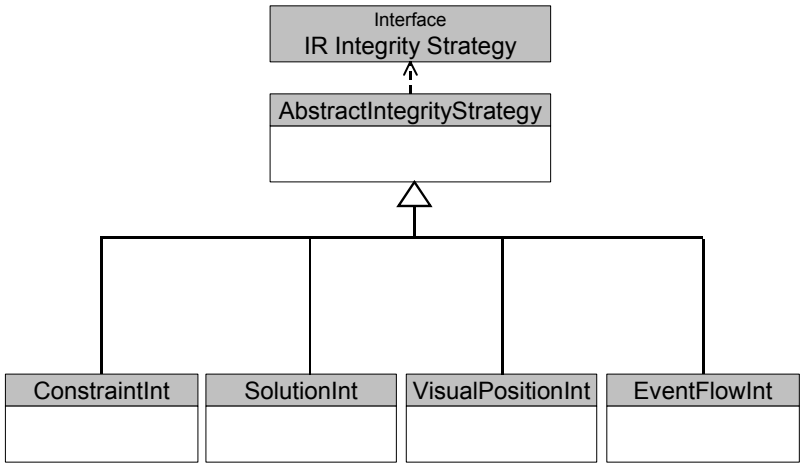


Abbildung 43: Die Integritätsprüfungsklassen

Die Prüfverfahren werden vom IntegrityEnvironment der Reihe nach gestartet. Dabei werden ihnen die jeweils zu prüfenden ComponentTreeNode mit übergeben. Standardmäßig wird hier der ComponentTreeNode der Superkomponente übergeben. Die Prüfung selbst erfolgt dann rekursiv, so dass nur ein Aufruf nötig ist und die gesamte Komposition geprüft wird. Die Ergebnisse der Prüfung werden den entsprechenden ComponentTreeNodes als SemanticErrors beigefügt. Sie lassen sich dann sowohl von IntegrityEnvironment (Fehlerübersicht) als auch vom TailorClient (kontextualisierte Fehlerbeschreibung) selbst abrufen und ausgeben.

10.3.4 Integration notwendiger Befehle

Da alle Menüpunkte von einer MenuFactory erzeugt werden, ist eine Integration recht einfach. Es werden hier lediglich zwei neue Einträge eingebracht: Der Menüpunkt „Integritätsprüfung starten“ dient lediglich dazu, die Integrationsprüfung zu starten. Dazu wird in der vorgelagerten Integritätsprüfung (IntegrityEnvironment) die Methode „checkIntegrity“ aufgerufen.

Ein weiterer Menüpunkt wird benötigt für die Auswahl des ApplicationTemplate. Bei Auswahl dieses Menüpunkts öffnet sich ein erweiterter Auswahldialog. Das Ergebnis dieser Auswahl wird der IntegrityEnvironment übergeben.

10.3.5 API zur Anzeige von Fehlern und Hilfen

Wie schon oben beschrieben ist im FREEVOLVE-System jede Komponente zu Steuerungs- und Visualisierungszwecken von einem Proxy-Objekt, dem ComponentTreeNode-Objekt, umgeben. Diese Objekte stellen nun auch Funktionalität zur Visualisierung der Fehler zur Verfügung. Des weiteren wurde in die jeweiligen Kontext-Menüs ein Befehl hinzugefügt, der dazu dient, erweiterte Informationen der Komponente anzuzeigen.

Die weitere Darstellung von Fehlern, wie sie in Abschnitt 10.2 beschrieben wurde, konnte davon unabhängig implementiert werden. Lediglich die entsprechenden Anker wurden benötigt.

Schließlich gibt es einen Menüeintrag um sich eine Übersicht der letzten durchgeführten Integritätsprüfung anzeigen zu lassen. Dieser ruft lediglich im IntegrityEnvironment die entsprechende Methode auf. Die Informationen liegen dann schon vor und werden lediglich formatiert und angezeigt.

10.4 Evaluierung der Integritätsprüfung aus Nutzersicht

In Kapitel 5 wurde der TailorClient in seiner grundsätzlichen Funktionalität evaluiert. In einer abschließenden Evaluation sollen nun die um eine Integritätsprüfung erweiterte Anpassungsumgebung einem Benutzungstest unterzogen werden. Um die Ergebnisse möglichst gut vergleichen zu können, wurde der Versuchsablauf unverändert belassen. Zehn Personen mit unterschiedlichen Kenntnissen im Bereich IT wurden wiederum gebeten nach der Thinking-Aloud-Methode[28] verschiedene Anpassungsaufgaben zu lösen. Diesmal waren zu allen Komponenten Integritätsbedingungen definiert. Die Benutzungsschnittstelle war um die Integritätsprüfung wie in den Abschnitten 10.1 bis 10.3 beschrieben erweitert. Es wurden wiederum die Beispielaufgaben, wie sie auch schon in der ersten Studie vorgestellt wurden, bearbeitet.

Die Kernaufgabe der Integritätsprüfung liegt in der Unterstützung der Benutzer beim Anpassen ihrer Applikationen. Nichtsdestotrotz verändert sich durch die Erweiterung des TailorClient um eben diesen Mechanismus die Art, wie Benutzer ihre Kompositionen wahrnehmen und ihren Bedürfnissen entsprechend verändern. Aus diesem Grund reicht es nicht, ausschließlich die in Nielsens Kategorisierung aufgeführten Punkte „Fehlervermeidung“ und „Unterstützung der Benutzer bei der Erkennung und Korrektur von Fehlern“ (siehe Kapitel 3) zu thematisieren. Stattdessen hat die Integritätsprüfung Auswirkungen auf die gesamte Anpassungsschnittstelle.

- **Sichtbarkeit des Systemstatus**: Die Anzeige des Systemstatus bleibt auch nach der Erweiterung um die Integritätsprüfung unverändert. Die Probanden empfanden die zusätzlichen Hinweis-Icons, die noch vorhandene Fehler signalisieren, als sehr hilfreich. Für sie waren diese Hinweise eine wertvolle Unterstützung bei der Einschätzung, inwieweit die angepasste Applikation korrekt ablaufen würden.

[28] Die Beschreibung der Methodik wurde bereits in Kapitel 5 gegeben.

- **Erkennbarer Abgleich zwischen realer Welt und dem Computersystem**: Die in den Testapplikationen zum Einsatz kommenden Komponenten sind durch die Namensgebung nur unzureichend beschrieben. Dies hat sich schon in der ersten Evaluation gezeigt. Jedoch wird durch die zusätzliche Integritätsprüfung und die Beschreibung der Fehler[29] das Abstraktionsniveau durch weitere Beschreibungen verringert. Es besteht hier die Möglichkeit, die Integritätsprüfung im Sinne einer kontext-sensitiven Hilfe zu verstehen, die den aktuellen Zustand (wie andere kontext-sensitiven Hilfen auch) und den Soll-Zustand, der einer korrekten Applikation im Sinne der Integritätsbedingungen entspricht, mit einbezieht.
- **Benutzerkontrolle und Freiheit**: Durch die hinzugefügte Integritätsprüfung wird die Freiheit der Benutzer nicht eingeschränkt. Die Fehler müssen nicht beseitigt werden, sondern sind eher als Hinweise zu verstehen. Dennoch hat sich in der Testphase gezeigt, dass die Benutzer bestrebt sind, alle Fehler zu korrigieren und umgekehrt stark verunsichert sind, wenn sie die Applikation mit vermeintlichen Fehlern starten. Hier ist bei der Gestaltung des Front-Ends und auch bei Dokumentationen darauf zu achten, dieses Missverständnis auszuräumen. Die Integritätsprüfung wird notwendigerweise bei der semantischen Bewertung einer Applikation immer unvollständig sein. Insofern sind die angezeigten Hinweise als prüfenswerte Teilkonfigurationen innerhalb der Gesamtkomposition oder als Empfehlungen (Verbesserungsvorschlag) zu verstehen. Hier ist es sehr wichtig, auf eben diese Umstände hinzuweisen.
- **Konsistenz**: In der Benutzerstudie ergaben sich kaum Probleme mit der Verständlichkeit der verwendeten Symbole und der Beschreibung der Fehler. Auch bei der Erweiterung des TailorClient um die Integritätsprüfung zeigte sich wieder, wie nützlich Kontextmenüs bei der Gestaltung und Nutzung sind.
- **Fehlervermeidung**: Dies ist eines der Hauptziele, das mit der Integritätsprüfung erreicht werden sollte. Allerdings steht nicht die direkte Fehlervermeidung im Sinne des Unterbindens von Fehlern im Vordergrund, sondern die Erkennung selbiger. Erst mit dem Ende einer Anpassungstransaktion sollte die Anwendung aus Sicht der Anpassenden fehlerfrei sein.
  Wie schon oben angesprochen scheint hier noch das Design der Integritätsprüfung an der GUI geändert oder die Erläuterungen, wofür die Integritätsprüfung dient und was sie leisten kann, erweitert werden zu müssen. Zwar waren in den Tests alle Applikationen am Ende einer Transaktion fehlerfrei im Sinne der Integritätsprüfung, jedoch hatten die Probanden Mühe, nicht-integere Applikationen zu akzeptieren und den Anpassungsmodus zu verlassen. Diese Art der Nutzung der Integritätsprüfung führt zu einer zu starken Einschränkung des Anpassungsvorgangs.
- **Bestätigung statt Erinnerung**: Wie auch im TailorClient ohne Integritätsprüfung sind alle Änderungen sofort sichtbar. Zur Kontrolle können diese Änderungen bei Bedarf in die laufende Anwendung übernommen werden. Da der Status des Systems also immer vollständig dargestellt wird, ist es allen Benutzern möglich, jederzeit durchgeführte Änderungen zu erkennen.

[29] Auch hier kommt es maßgeblich auf die gewählte Sprache und die verwendeten Metaphern an, wie verständlich Fehlermeldungen für Benutzer sind.

- **Flexibilität und Effizienz**: Mit Hilfe der Integritätsprüfung gelangen vor allem Anfänger schneller zu den bei ihren Anpassungsaktionen beabsichtigten Zielen. Sie lassen sich vom System führen und reagieren, so zeigte sich in der Studie, strukturiert auf die angezeigten Fehlermeldungen. Fortgeschrittene Nutzer verwandten die Prüfung lediglich am Ende einer Anpassungsaktion zur Endkontrolle. Durch die Einbettung der Integritätsprüfung als expliziten Mechanismus kann die Integritätsprüfung auch vollständig ignoriert werden. Sie verschlechtert in einem solchen Fall die Flexibilität nicht. Allerdings sollte hier nochmals erwähnt werden, dass die Nutzer ihre Freiheit durch die übergenaue Beachtung der Integritätsprüfung selbst einschränken. Trotzdem führt die Nutzung der Integritätsprüfung in Zweifelsfällen zu mehr Sicherheit und damit zu höherer Effizienz.
- **Ästhetisches und minimalistisches Design**: Das Design des Prototypen wurde gegenüber der Vorversion nicht verändert. Lediglich die Integritätsprüfung wurde integriert, wie in den vorigen Abschnitten beschrieben wurde. Die Komplexität der Gesamtapplikation hat sich nur minimal erhöht. Durch die Nutzung von Kontextmenüs bleiben viele Funktionen verborgen. Die zusätzliche Anzeige von Integritätsinformationen geschieht (bis auf die Icons) nur auf Wunsch und ist so eher als zusätzliche Hilfe zu betrachten.
- **Unterstützung der Benutzer bei der Erkennung und Korrektur von Fehlern**: In der originären Version des TailorClient gibt es keine Unterstützung zur Identifizierung und Korrektur von Fehlern. Die Erkennung wurde auf Basis der Integritätsprüfung hinzugefügt. Sie bietet im Falle der Solution Integrity auch eine Unterstützung bei der Fehlerkorrektur. In der Benutzerstudie wurde dieser Mechanismus vor allem von den eher unerfahreneren Benutzern gern benutzt. Ein sehr charakteristischer Kommentar in diesem Zusammenhang war die Frage, wie denn vorher komponentenbasierte Anpassbarkeit für Endanwender überhaupt möglich gewesen sei. Dabei wurde der Mechanismus sowohl als Arbeitsunterstützung empfunden als auch durch sie ein „sicheres Gefühl“ im Umgang mit der Anpassungsumgebung erzeugt. Die Anzeige der Fehler am GUI wurde durchweg lobend erwähnt. Auch die Hinweise, die erst auf zusätzliche Interaktion hin dargestellt werden, wurden genutzt. Die Hilfstexte waren für die Korrektur sehr hilfreich.
- **Hilfe und Dokumentation**: Der mit einer Integritätsprüfung ausgestattete TailorClient verfügt ebenso wie die Vorversion nicht über eine Hilfefunktion. Auch auf eine Dokumentation wurde im Rahmen der Studie verzichtet, da während der gesamten Studie ein Experte anwesend war, der sowohl in das System und die Aufgabenstellung einführte als auch anschließend für Fragen zur Verfügung stand.
  Speziell die unerfahreneren Probanden nutzten jedoch die Integritätsprüfung mit ihren Hinweisen und Korrekturmöglichkeiten als interaktive und kontext-sensitive Hilfe. Auf diese Weise erhielten sie sukzessive darüber Auskunft, wie eine Komponente sich benutzen lässt. Schwierig war in diesem Zusammenhang lediglich die Auswahl der Komponenten, die nicht durch ein Hilfesystem unterstützt wird. Die an einigen Stellen wenig aussagekräftigen Namen halfen den Benutzern nicht immer, die Funktionalität der Komponente zweifelsfrei zu erkennen. Hier half wieder der eher explorative Ansatz bei der komponentenbasierten Anpassbarkeit weiter. Komponenten lassen sich auswählen, in die Komposition integrieren, und auf diese Weise kann ihre Funktionalität erforscht werden.

10.5 Fazit

Dieses Kapitel hat gezeigt, wie die Integritätsprüfung, die auf theoretischer Basis in Kapitel 6 bis 9 entwickelt wurde, in den bestehenden TailorClient, die Anpassungsumgebung für FREEVOLVE, integriert wurde. Hier stand schwerpunktmäßig die Unterstützung von Endanwendern ohne Programmiererfahrung im Vordergrund. Insofern ist bei der Integration darauf zu achten, dass die Bedien- und Anzeigeelemente leicht verständlich sind und die Komplexität der Anwendung nicht weiter erhöhen, als dies unbedingt notwendig ist. Zu diesem Zweck wurde eine sehr einfache Bedienung der Prüfung gewählt, die lediglich das explizite Starten der Prüfung auf der aktuellen Komposition und die Anzeige des Prüfungsprotokolls erlaubt. Des weiteren werden erkannte Verletzungen der Integritätsbedingungen durch Icons angezeigt, die gleichzeitig als Schaltfläche zur Anzeige detaillierter Informationen dienen.

In einer weiteren Benutzerstudie wurde der Prototyp erneut evaluiert. Dabei wurden aus Gründen der Vergleichbarkeit der Ergebnisse die Komponentensammlung ebenso beibehalten wie die zu lösenden Aufgaben. Jedoch hatten die Testpersonen nun die Möglichkeit, die Integritätsprüfung als Unterstützungsmechanismus zu Hilfe zu nehmen. Die Studie hat ergeben, dass speziell unerfahrenere Benutzer eine Prüfung ihrer Kompositionen auf Integrität nutzen und sie eine sinnvolle Erweiterung der schon vorhandenen Unterstützungstechniken darstellt. Die Möglichkeit, Fehler soweit sie durch Integritätsbedingungen beschrieben werden zu vermeiden, erhöht die Sicherheit beim Anpassen und erleichtert den Zugang zu dieser nicht alltäglichen Arbeitsaufgabe. Weiterhin wird das Verständnis für die Komponenten, ihre Funktionalität und die Zusammenarbeit zwischen ihnen gefördert. Für erfahrene Nutzer stellt die Integritätsprüfung weniger eine interaktive Hilfe und Führung als eine Kontrollmöglichkeit dar. Auch sie profitieren von der Integritätsprüfung, da sich Flüchtigkeitsfehler bei der Anpassung verringern.

11 Zusammenfassung und Ausblick

Dieses letzte Kapitel fasst noch einmal die wichtigsten Ergebnisse der Arbeit zusammen. Abschließend wird ein Ausblick über Fragen, die im Rahmen der Arbeit offengeblieben sind, und Anschlussmöglichkeiten gegeben. Ziel der Arbeit war die Entwicklung und Erprobung einer Integritätsprüfung zur Unterstützung bei Anpassungen an komponentenbasierter Software. Dabei stand die Unterstützung von unerfahreneren Anwendern ohne Programmierkenntnisse im Vordergrund der Betrachtung. Durch eine Integritätsprüfung, die semantische Fehler innerhalb einer Komposition aufdecken soll, wird weiterhin das Verständnis sowohl für den Anpassungsprozess als auch für die anzupassenden Artefakte gefördert.

Eine Anpassungssprache bietet verschiedene Operationen, die sich an den Eigenschaften (Applikationen werden gebildet durch Parameterisieren und Verbinden einzelner ausgewählter Komponenten) der Komponenten-Architekturen orientieren. Darauf setzen auch die Integritätskonzepte auf, die eben diese grundlegenden Eigenschaften und die damit zusammenhängenden Operationen betrachten.

Im Rahmen dieser Arbeit wurden zwei grundlegend unterschiedliche Integritätskonzepte vorgestellt: Die Constraint Integrity einerseits und die darauf aufbauende Restricted Solution Integrity und die Event Flow Integrity andererseits. Constraints beziehen sich jeweils lokal auf die Eigenschaften einer Komponente. Werden Constraints nicht eingehalten, so wird dies als Verletzung der Integrität gewertet. Solutions bieten in diesem Zusammenhang dann schon Vorschläge zur Korrektur an. Verschiedene Spezialisierungen dieses Ansatzes wurden eingeführt und diskutiert.

Die Event Flow Integrity beschäftigt sich mit der Analyse der Nachrichtenflüsse innerhalb einer Komposition. Ziel dieser Prüfung ist sicherzustellen, dass innerhalb der Komposition erzeugte Nachrichten auch konsumiert werden. Dem liegt die Annahme zugrunde, dass wichtige Nachrichten nicht einfach ignoriert werden dürfen. Ansonsten gilt eine solche Komposition als nicht integer. Integritätsbedingungen werden einzelnen Komponenten oder Gruppen von ihnen in Form von externen Beschreibungen mitgeliefert.

Die Korrektheit der einzelnen Integritätskonzepte in Bezug auf Berechenbarkeit ebenso wie auf Sinnhaftigkeit in Bezug zur Aufgabenstellung wurde ausführlich diskutiert. Dabei diente sowohl das Grundlagenwissen aus der theoretischen Informatik wie auch Erfahrungswissen aus dem Bereich der Software-Technik zur Überprüfung der Konzepte.

Eine Anpassungsumgebung für die FREEVOLVE-Plattform wurde zur Prüfung dieser Integritätskonzepte in Bezug auf praktische Umsetzbarkeit und zur Evaluierung des Nutzens für Anpassende implementiert. Bei FREEVOLVE handelt es sich um eine Client-Server-Umgebung, in der komponentenbasierte Applikationen gestartet und genutzt werden können. Der hier entwickelte TailorClient dient dazu, Applikationen zur Laufzeit anpassen zu können.

Die oben genannten Integritätskonzepte wurden umgesetzt in einer Erweiterung des TailorClients. Sie erlaubt es, eine explizite Prüfung der aktuellen Komposition zu initiieren. Bei der Prüfung gefundene Integritätsverletzungen werden in verschiedenen Kompositionsansichten des TailorClient dargestellt. Bei Bedarf lassen sich dazu erweiterte Informationen und im Falle der Solution Integrity auch Lösungsvorschläge anzeigen.

In einer dreistufigen Benutzer-Evaluation wurde zu Beginn überprüft, inwieweit Endanwender grundsätzlich in der Lage sind, vorhandene Kompositionen in Bezug auf ihre Funktionalität zu interpretieren, zu verändern oder eigene Kompositionen zu entwickeln. Als Testapplikation diente hier ein anpassbares, komponentenbasiertes Email-Filter-System, das zur Unterstützung von Arbeitsgruppen entwickelt wurde. Die Studie ergab grundsätzlich, dass der komponentenbasierte Ansatz auch aufgrund seiner Pendants in der realen Welt geeignet ist, um Endanwendern das Anpassen ihrer Applikationen zu ermöglichen. Die Detailergebnisse dieser Vorstudie flossen in die Entwicklung des TailorClient mit ein. Sie gaben erste Aufschlüsse darüber, wie die Bedienung gestaltet sein sollte, welche Interaktionsmechanismen benötigt werden und wie eine sinnvolle Darstellung von Komponenten erreicht werden kann.

Die beiden folgenden Testphasen dienten der Evaluierung des TailorClient selbst. Dabei wurden Testpersonen Anpassungsaufgaben gestellt, die sie mit Hilfe des Anpassungswerkzeugs an vorhandenen Applikationen lösen sollten. Im ersten Durchgang wurde dabei auf die Integritätsprüfung verzichtet, im zweiten Durchgang war sie vorhanden und konnte genutzt werden. Dieser Versuchsaufbau diente zur Überprüfung, inwieweit nicht andere Einflüsse für das Gelingen oder Misslingen der Aufgabenbearbeitung verantwortlich waren. Die Studie zeigte ein grundsätzlich positives Ergebnis. Alle Beteiligten kamen mit der Integritätsprüfung, den vorhandenen Integritätsprüfungstechniken und auch mit ihrer Darstellung und den dazugehörigen Erläuterungen auf Anhieb gut zurecht. Während unerfahrenere Benutzer jedoch während des gesamten Anpassungsvorgangs die Integritätsprüfung als Unterstützung nutzten, reichte den erfahrenen Nutzern (Administratoren oder Programmierer) eine Überprüfung ihrer Komposition nach der Aufgabenbearbeitung. Die Evaluierung hat so zeigen können, dass eine Integritätsprüfung ein sinnvolle Erweiterung einer Anpassungsumgebung für komponentenbasierte Architekturen darstellt.

Die Aufgabenstellung kann als abgeschlossen betrachtet werden. Sie wurde ausführlich diskutiert und zu verschiedenen Arbeiten in Bezug gesetzt. Darauf basierend wurden der Integritätsbegriff für komponentenbasierte Architekturen eingeführt und verschiedene Prüfverfahren entwickelt. Die abschließende Studie zeigt, dass dieses Konzept sich in eine Anpassungsumgebung integrieren lässt und den Anpassungsprozess speziell für weniger erfahrene Benutzer deutlich erleichtert. Nichtsdestotrotz sind während der Bearbeitung der Aufgabenstellung verschiedene neue und weiterreichende Fragen aufgetreten, die im Rahmen dieses Vorhabens nicht mehr beantwortet wurden. Auf sie soll im Folgenden genauer eingegangen werden.

Optimierungspotential

Die Entwicklung der Integritätskonzepte wurde bisher von den Anpassungsoperationen, die in komponentenbasierten Architekturen zur Verfügung stehen, getrieben. Dieses Vorgehen ist naheliegend, die Sinnhaftigkeit wird auch durch die Evaluation bestätigt. Auf theoretischer Ebene orientiert sich diese Evaluation an der Korrektheit der auf Basis der Integritätskonzepte entwickelten Prüfalgorithmen. Weiterhin wird untersucht, inwieweit sich Programmierpatterns leicht durch die Nutzung von Integritätsbedingungen unterstützen lassen.

Hier stellt sich die Frage, inwieweit diese Patterns bei der Entwicklung von Komponenten, die primär zur Konstruktion von Applikationen von den Anwendern selbst zum Einsatz kommen, auch im Bereich der komponentenbasierten Anpassbarkeit Verwendung finden können und aus Sicht der Verständlichkeit sollten. Eine breitere Studie, in der solche Komponenten und Applikationen untersucht werden, kann darüber Aufschluss geben. Die zentrale Fragestellung ist hierbei, inwieweit objekt-orientierte Programmier-Patterns bei der Entwicklung von anpassbaren komponentenbasierten Applikationen zum Einsatz kommen. Weiter lässt sich hinterfragen, ob neue Patterns identifiziert werden

können, die speziell in diesem Anwendungskontext Gültigkeit haben. Die Ergebnisse einer solchen Studie und die Beschreibung von Integritätsbedingungen für solche Komponenten-Sets können dann der nächste Schritt sein, um die in dieser Arbeit entwickelten Konzepte breiter zu evaluieren.

Eines der Ziele dieser Arbeit war, den Lernprozess und das Verständnis für komponentenbasierte Anpassbarkeit zu fördern. Die Evaluation im Rahmen dieses Vorhabens wurde in zwei (bzw. drei) Stufen durchgeführt. Dabei stand im Vordergrund des Interesses, inwieweit Endanwender mit der Unterstützung durch eine Integritätsprüfung bestehende Applikationen leichter anpassen können. Über eine Entwicklung bezüglich der Kompetenzen beim Anpassen von Software lassen sich mit diesen punktuellen Studien nur schwache Aussagen formulieren. Um über den Kompetenzzuwachs und damit die Lernförderlichkeit des Ansatzes valide befinden zu können, ist eine über einen längeren Zeitraum durchgeführte Benutzerstudie notwendig. Allerdings zeigt sich schon anhand der hier durchgeführten Befragungen, dass Anpassungen für Endanwender mit Hilfe der Integritätsprüfung leichter durchzuführen sind und dass auch inhaltliche Fragen zur Funktionalität der Komponenten und der Applikation als ganzes besser beantwortet werden. Dies lässt immerhin den Schluss zu, dass diese noch ausstehende Langzeitstudie die hier postulierten Ergebnisse bestätigen wird.

Erweiterungsmöglichkeiten

Die im Rahmen dieses Vorhabens entwickelten Konzepte lassen sich recht gut in andere Arbeiten integrieren. Beispielsweise können die Integritätsbedingungen verwendet werden, um für Explorationsumgebungen, wie sie in Kapitel 3 vorgestellt wurden, automatisch Daten zu erzeugen.

Derzeit werden die meisten Explorationsumgebungen manuell mit Daten befüllt. Die Auswahl geeigneter Testdaten stellt dabei, wie in den meisten Simulations- oder Testumgebungen, keine triviale Aufgabe dar. Durch den komponentenbasierten Ansatz lassen sich Testdaten dadurch leichter entwerfen, dass die Einzelkomponenten für sich betrachtet werden können. Die verhältnismäßig schwache Abhängigkeit zwischen den Komponenten wirkt sich hier insofern günstig auf diesen Entwurf aus, als dass kritische Fälle basierend auf speziellen Eigenschaften oder Funktionsweisen der Komponenten für sich betrachtet werden können. Nichtsdestotrotz ist für die Entwicklung sinnvoller Testdaten die genaue Kenntnis über die Funktionsweise eine Komponente unbedingte Voraussetzung.

Integritätsbedingungen werden den Komponenten ebenso wie detaillierte Beschreibungen über ihre Funktionalität von den Entwicklern mitgeliefert. Sie weisen implizit auf kritische Fälle bei der Verwendung von Komponenten hin. Dies können ebenso Abhängigkeiten zwischen den einzelnen Parametern einer Komponenten wie zu anderen Komponenten sein. Eine automatische Auswertung der Integritätsbedingungen findet in dem hier beschriebenen Ansatz schon statt. Auf Basis einer weitergehenden Analyse können so Daten erzeugt werden, die im Falle einer Simulation genau die kritischen Punkte einer Komponente aufzeigen können.

Ebenso ist zu überlegen, inwieweit nicht auch die Generierung von Integritätsbedingungen unterstützt werden kann. Bisher werden die Bedingungen mit Hilfe eines XML-Editors erzeugt. Dieser ist in der Lage, die Syntax in Abhängigkeit von der definierten DTD zu prüfen. Eine weitere Erleichterung ergäbe sich daraus, dass Eigenschaften der Komponente analysiert und extrahiert würden und im XML-Editor – viele dieser Editoren sind aufgrund der Komplexität von XML programmierbar – direkt angeboten würden, um für sie Integritätsbedingungen zu formulieren. Ein ähnliches Verfahren wird derzeit schon eingesetzt, um die für die Komponentenbeschreibungen benötigten CAT-Dateien zu erzeugen. Hier werden die Komponenten mit Hilfe der Java Introspection API analysiert. Die extrahierten Eigenschaften (Parameter und Ports) werden

anschließend in einem kleinen Zusatzwerkzeug dargestellt. Die Entwickler können entscheiden, welche der Eigenschaften im Anpassungswerkzeug sichtbar und änderbar sein sollen. Auf Basis dieser Informationen wird dann die CAT-Datei erzeugt.

Mängel an der Komponentenplattform FREEVOLVE bzw. Schwächen an den Konzepten der komponentenbasierten Anpassbarkeit im Allgemeinen

Während der Arbeit mit der FREEVOLVE-Plattform haben sich einige Schwächen offenbart, die im folgenden diskutiert werden sollen. Notwendige Erweiterungen, um FREEVOLVE außerhalb der Forschung in Anwendungsfeldern einsetzen zu können, werden ebenso in diesem Zusammenhang beschrieben.

Im Kontext dieses Vorhabens waren keine Beschränkungen im Anpassungsmodus vorgesehen. Alle Benutzer hatten das Recht, Anpassungen an den Applikationen vorzunehmen. Für die Fragestellung der Arbeit war eine Einschränkung weder notwendig noch sinnvoll, sollte den Probanden in den Benutzertests doch größtmögliche Freiheit eingeräumt werden.

Für die Nutzung von FREEVOLVE als Anwendungsplattform in der Praxis sind Zugriffsrechte für den Anpassungsmodus zu entwerfen. Ähnlich wie allgemeine Zugriffskontrollsysteme muss das System unberechtigte Änderungen an der Applikation verhindern. Dabei ist zu diskutieren, wie Rechte zur Anpassung gestaltet sein müssen. Die zu schützenden Artefakte sind hier sowohl die vorhandenen Applikationen wie auch die Komponenten, die für die Nutzung innerhalb einer Applikation zur Verfügung stehen. Weiterhin sind die Anpassungsoperationen zu betrachten. Da hier viele Wechselwirkungen zwischen der Nutzung der Artefakte und den Operationen bestehen, sind hier genaue Untersuchungen nötig. Auch die Teilnutzung einer Komponente kann eventuell sinnvoll sein, so dass Zugriffsregeln ebenso auch unterhalb des Komponentenniveaus zu beschreiben sind. Speziell im Anwendungskontext Groupware, für den FREEVOLVE konzipiert wurde, ist hier eine dies berücksichtigende, konzeptuelle Erweiterung unabdingbar.

In diesem Kontext ist auch zu prüfen, wie Anpassungen am Server vorgenommen werden können. Solche Veränderungen unterliegen der besonderen Problematik, dass sie nicht nur für den anpassenden Nutzer, sondern für alle Nutzer des Systems Auswirkungen haben. Hier stellt sich nicht nur die Frage, wie unerwünschte Anpassungen mit Hilfe von Zugriffsregeln im Anpassungsmodus zu unterbinden sind.

Vielmehr muss diskutiert werden, ob nicht auch Integritätsbedingungen helfen können, Anpassungen am Server sicherer zu gestalten. Dies zielt auf die Definition temporärer Integritätsbedingungen ab, die sich aus den aktuellen Konfigurationen des Servers und allen angeschlossenen Clients ableiten. Mit Hilfe solcher Integritätsbedingungen könnte definiert werden, welche Komponenten minimal zum Betrieb der Clients benötigt werden. Erste Arbeiten in diesem Bereich finden sich in [Alda et al., 2002].

Während des Starts einer Applikation in die FREEVOLVE-Plattform werden entsprechend vorhandener CAT-Dateien einzelne Komponenten instanziiert, parameterisiert und miteinander verbunden. Während dieses Zeitraums ist die Applikation in einem undefinierten Zustand. Es ist nicht klar, wann der konsistente Zustand (vereinfacht definiert durch die vollständige Bearbeitung aller benötigten CAT-Dateien) erreicht ist. Änderungen am Zustand (sowohl zur Laufzeit als auch im Anpassungsmodus) der Applikation sind während dieses Startvorgangs nicht deterministisch. Dieser unerwünschte Effekt lässt sich dadurch beheben, dass die Applikation erst dann aktiviert wird und damit zur Nutzung und Anpassung freigegeben wird, wenn ein stabiler Zustand erreicht ist.

Aufgrund obiger Diskussion, die sich mit der Änderungen am Server und damit einhergehenden Problemen auf den Clients beschäftigte, ist dies jedoch nur oberflächlich betrachtet eine geeignete Lösung. Eine Erweiterung des FLEXIBEANS-Komponentenmodells, die mögliche Ausfälle in der Kommunikation zwischen Komponenten (seien sie nicht vorhan-

den oder nur temporär nicht erreichbar) berücksichtigt, könnte hier Abhilfe schaffen. In jedem Fall ist auch hier sowohl Forschungs- als auch Entwicklungsbedarf, um Komponentenarchitekturen als geeignete Plattform für durch Endanwender veränderbare Software bereitzustellen.

Schließlich zeigte sich während der Konzeption sowohl der Erweiterungen an der Plattform selbst wie auch bei der Implementierung der Beispiel-Komponenten-Sets, dass die bewusst strikte Trennung der Plattform von den in ihr ablaufenden Kompositionen zu starken Einschränkungen führt. Beispielsweise musste für die GeoChat-Applikation ein eigenes User-Management entwickelt werden. Die FREEVOLVE-Plattform verfügt über eine Benutzerverwaltung. Sie ist jedoch – wie auch andere Funktionalitäten – für die Kompositionen nicht nutzbar. Gleiches gilt für die Basis-Operationen des Anpassungs-API (Einfügen und Löschen von Komponenten, Verändern von Parametern oder Bindungen etc.). Solche Zugriffe können dann sinnvoll sein, wenn Applikationen selbst-adaptiv konzipiert werden sollen. In der in Anhang C beschriebenen GeoChat-Applikation ist es beispielsweise sinnvoll, mehrere Chat-Sessions getrennt voneinander verwalten zu können. Üblicherweise würde man einen Session-Manager damit beauftragen, für jede neue Session eine eigene Komponente zu instanziieren und der Komposition hinzuzufügen. Dies ist jedoch unter FREEVOLVE nicht vorgesehen. Für weiterführende Konzepte wie Selbst-Adaptivität, mit Hilfe derer Anwendungen sich vollständig in Abhängigkeit von veränderten Außenbedingungen rekonfigurieren, sind solche Möglichkeiten jedoch unabdingbar.

Hier sollte über ein weiteres – vom Anpassungs-API getrenntes – Interface nachgedacht werden, dass Zugriffe der Applikation auf die sie umgebende Plattform zulässt. Diese Trennung der beiden Interfaces ist sinnvoll, da die Möglichkeiten und Rechte für Applikationen und Benutzer vollständig unterschiedlich sind. Ein anderer Ansatz ist, dies in Kombination mit oben angedeuteter Diskussion über Anpassungsrechte zu verbinden.

Mit dieser Arbeit wurde gezeigt, dass Integritätsbedingungen einen Weg darstellen, um komponentenbasierte Anpassbarkeit für Benutzer zugänglich zu machen und ihnen so zu ermöglichen, auch komplexe Anforderungen selbstständig durchführen zu können. Der grundsätzliche Ansatz, Software komponentenbasiert zu entwickeln, wurde hierbei wiederum erfolgreich bestätigt. Nichtsdestotrotz sind verschiedene Fragen, die über das Thema dieser Arbeit hinausgehen, unbeantwortet geblieben, neue haben sich zusätzlich während den hier beschriebenen Arbeiten ergeben. Teilweise werden diese Fragestellungen schon jetzt in anderen Arbeiten eingehender untersucht.

12 Literatur

[Alda et al., 2002] Alda, S., Won, M. und Cremers, A. B.: “Managing Dependencies in Component-Based Distributed Applications"”, in: Proceedings of FIDJI 2002 (International Workshop on scientific engineering of Distributed Java Applications), Luxembourg, LNCS, Springer Verlag, S. 143-154, 2002.

[Alda et al., 2002a] Alda, Sascha; Radetzki, Uwe; Won, Markus; Cremers, Armin B.: “Eine anpassbare, komponentenbasierte Plattform für vernetzte Kooperationen im Bauwesen“. Ergebnisband zur Tagung “Bauen mit Computern“ der VDI, S. 351-370, April 2002, Bonn, Deutschland.

[Allen, 1997] Allen, R. J.: “A Formal Approach to Software Architecture”, Carnegie Mellon University, Pittsburg, ph.D. Thesis, 1997.

[ANSI, 1992] ANSI: “Database Language SQL. Dokument: ANSI X3.135-1992”, 1992.

[ANSI, 1996] ANSI: “Database Language SQL. Dokument: ANSI X3.135-1996”, 1996.

[Apple, 1987] Apple: *Human Interface Guidelines: The Apple Desktop Interface.* Reading: Addison Wesley, 1987.

[Arbortext, 2003] Arbortext: “Arbortext Epic”. Ann Arbor, MI, USA, 2003, (http://www.arbortext.com/index.html).

[AspectJ, 2002] AspectJ: “The AspectJTM Programming Guide”, 2002, (http://aspectj.org/doc/dist/progguide/index.html).

[Baster et al., 2001] Baster, G., Konana, P. und Scott, J. E.: “Business Components - A Case Study of Bankers Trust Australia Limited”, in: *Communications of the ACM*, vol. 44 (3), S. 92-98, 2001.

[Baumgarten, 1990] Baumgarten, B.: *Petri-Netze - Grundlagen und Anwendungen*. Mannheim/Wien/Zürich: Wissenschaftsverlag, 1990.

[Bentley und Dourish, 1995] Bentley, R. und Dourish, P.: “Medium versus Mechanism: Supporting Collaboration through Customisation”, in: Proceedings of ECSCW 95, Stockholm, Sweden, Kluwer, S. 133-148, 1995.

[Bernstein et al., 1980] Bernstein, P. A., Blaustein, T. und Clarke, E. M.: “Fast maintenance of semantic integrity assertions using redundant aggregate data”, in: Proceedings of 6th VLDB, S. 126-136, 1980.

[Birngruber, 2001] Birngruber, D.: “A Software Composition Language and Its Implementation”, in: *Perspectives of System Informatics (PSI 2001)*, vol. LNCS 2244, D. Bjorner, M. Broy und A. V. Zamulin, Eds. Novosibirsk, Russland: Springer, S. 519-529, 2001.

[Birngruber und Hof, 2000] Birngruber, D. und Hof, M.: “Using Plans for Specifiying Preconfigured Bean Sets”, in: Proceedings of TOOLS USA'00, Santa Barbara, USA, 2000.

[Birngruber, 2000a] Birngruber, D. H., M.: “Interactive Decision Spot for Java Beans Composition”, Universität Linz, Linz, Working Paper 2000a.

[Boehm, 1988] Boehm, B. W.: “A Spiral Model of Software Development and Enhancement”, in: *IEEE-CS Computer*, vol. 21, May 88, S. 61-72, 1988.

[Bonar und Soloway, 1985] Bonar, J. und Soloway, E.: “Preprogramming knowledge: A major source of misconceptions in novice programmers”, in: *Studying the novice programmer*, E. Soloway und J. Spohrer, Eds. Hillsdale: Laurence Earlbaum Associates, S. 133-161, 1985.

[Brown et al., 1999] Brown, K., Eskelin, P. und Pryce, N.: “A Component Distribution Pattern Language”, vol. 2003, Working Paper ed, 1999, (http://www.geocities.com/ResearchTriangle/Campus/1574/distribut.pdf).

[Bruckman und Edwards, 1999] Bruckman, A. und Edwards, E.: “Should we leverage natural language knowledge? An analysis of user errors in natural-language-style programming languages”, in: Proceedings of Confereence on Human-Computer Interaction, S. 207-214, 1999.

[Carroll und Carrithers, 1984] Carroll, J. M. und Carrithers, C.: “Training Wheels in a User Interface”, in: *Communications of the ACM*, vol. 27, S. 800-806, 1984.

[Conklin und Begeman, 1988] Conklin, J. und Begeman, M. L.: “gIBIS: a hypertext tool for exploratory policy discussion”, in: Proceedings of Conference on Computer-supported cooperative work 1988, Portland, Oregon, ACM Press, S. 140-152, 1988.

[Cook et al., 2001] Cook, M. L., Gary Ericson, M. G., Hendrix, J. M., Robertson, S. J. W., Snapp, S. und McKenzie, S. K.: *Microsoft Office XP. Die technische Referenz*: Microsoft Press Deutschland, 2001.

[Cremers und Hibbard, 1978] Cremers, A. B. und Hibbard, T. N.: “Formal Modeling of Virtual Machines”, in: *IEEE Transactions on Software Engineering*, vol. 4, 5, S. 426-436, 1978.

[Cremers und Hibbard, 1978] Cremers, A. b. und Hibbard, T. N.: “Functional Behavior in Data Spaces”, in: *Acta Informatica*, vol. 9, S. 293-307, 1978.

[Curtis et al., 1988] Curtis, B., Sheppard, S., Kruesi-Bailey, E., Bailey, J. und Boehm-Davis, D.: “Experimental Evaluation of Software Documentation Formats”, in: *Journal of Systems and Software*, vol. 9, S. 1-41, 1988.

[Denning et al., 1990] Denning, S., Hoiem, D., Simpson, M. und Sullivan, K.: “The Value of Thinking-Aloud Protocols in Industry: A Case Study at

Microsoft Corporation", in: Proceedings of Human Factors Society, 34th Annual Meeting, 1990.

[DIN/ISO, 1996] DIN/ISO, -. "Ergonomische Anforderungen für Bürotätigkeiten mit Bildschirmgeräten, Teil 10: Grundsätze der Dialoggestaltung", in, 1996.

[Dix et al., 1998] Dix, A., Finlay, J., Abowd, G. und Beale, R.: *Human-Computer Interaction*, zweite Auflage ed: Prentice Hall Europe, 1998.

[Dourish, 1992] Dourish, P.: "Designing for Change: Reflective Metalevel Architecture for Deep Customization in CSCW", in: Proceedings of Toronto, Canada, ACM Press, 1992.

[du Boulay, 1989] du Boulay, B.: "Some Difficulties of Learning to Program.", in: *Studying the Novice Programmer.*, E. Soloway und J. C. Spohrer, Eds. Hillsdale, New Jersey: Lawrence Erlbaum Associates, S. 283-299, 1989.

[Eiffel, 2002] Eiffel: "Eiffel Tutorial", 2002, (http://www.eiffel.com/doc/online/eiffel50/intro/language/tutorial-00.html).

[Elrad et al., 2001] Elrad, T., Filman, R. E. und Bader, A.: "Aspect Oriented Programming (Editor's introduction)", in: *Communications of the ACM*, vol. 44, No. 10, S. 29-32, 2001.

[Engelskirchen, 2000] Engelskirchen, T.: "Explorationsunterstützung in Groupware am Beispiel eines anpassbaren Suchwerkzeugs", Universität Bonn, Bonn, Diplomarbeit, 2000.

[Eskelin, 2003] Eskelin, P.: "Component Design Patterns", vol. 2003: Wiki Web, 2003, (http://c2.com/cgi/wiki?ComponentDesignPatterns).

[Floyd et al., 1989] Floyd, C., Reisin, M. und Schmidt, G.: "STEPS to Software Development with Users", in: Proceedings of ESEC ´89 - 2nd European Software Engineering Conference, Coventry, England, S. 48-64, 1989.

[Förster, 2002] Förster, S.: "Kommunikations- und Kollaborationsunterstützung auf Basis serverseitiger E-Mail-Filter", Universität Bonn, Bonn, Diplomarbeit, 2002.

[Fossa und Sloman, 1996] Fossa, H. und Sloman, M.: "Implementing Interactive Configuration Management for Distributed Systems", in: Proceedings of ICCDS´96 (Conference on Configurable Distributed Systems, Annapolis, Maryland, IEEE Press, 1996.

[Fowler und Scott, 1999] Fowler, M. und Scott, K.: *UML Distilled. A Brief Guide to the Standard Object Modeling Languange.*, zweite Auflage ed: Addison-Wesley Professional, 1999.

[Frese et al., 1991] Frese, M., Irmer, C. und Prümper, J.: "Das Konzept Fehlermanagement: Eine Strategie des Umgangs mit Handlungsfehlern in der Mensch-Computer Interaktion", in: *Software für die Arbeit von morgen*, M. Frese, C. Kasten und C. Scarpelis, Eds. Berlin: Springer Verlag, S. 241-252, 1991.

[Friedrich, 1990] Friedrich, J.: “Adaptivität und Adaptierbarkeit informationstechnischer Systeme in der Arbeitswelt - Zur Sozialverträglichkeit zweier Paradigm”, in: Proceedings of 20. GI-Jahrestagung, Heidelberg, Springer, S. 178-191, 1990.

[Gamma et al., 1995] Gamma, E., Helm, R. und Johnson, R.: *Design Patterns. Elements of Reusable Object-Oriented Software.*: Addison Wesley Longman, 1995.

[Gellenbeck und Cook, 1991] Gellenbeck, E. M. und Cook, C. R.: “Does Signalling Help Professional Programmers Read and Understand Computer Programs?”, in: *Empirical Studies of Programming: Fourth Workshop*, J. Koenemann-Belliveau, T. G. Moher und S. P. Robertson, Eds. New Brunswick, New Jersey: Ablex Publishing Corporation, S. 82-98, 1991.

[Gertz und Lipeck, 1996] Gertz, M. und Lipeck, U. W.: “Deriving optimized integrity monitoring triggers from dynamic Integrity Constraints”, in: *Data & Knowledge Engeneering*, vol. 20, S. 163-193, 1996.

[Goldenson und Wang, 1991] Goldenson, D. R. und Wang, B. J.: “Use of Structure Editing Tools by Novice Programmers”, in: *Empirical Studies of Programming: Fourth Workshop*, J. Koenemann-Belliveau, T. G. Moher und S. P. Robertson, Eds. New Brunswick, New Jersey: Ablex Publishing Corporation, S. 99-120, 1991.

[Gosling und McGilton, 1996] Gosling, J. und McGilton, H.: “The Java Language Environment White Paper”, vol. 2003: SUN, 1996, (http://java.sun.com/docs/white/langenv/).

[Green, 1990] Green, T. R. G.: “Programming Languages as Information Structures”, in: *Psychology of Programming*, J.-M. Hoc, T. R. G. Green, R. Samurçay und D. J. Gilmore, Eds. London: Academic Press, S. 118-137, 1990.

[Green und Petre, 1996] Green, T. R. G. und Petre, M.: “Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework”, in: *Visual Languages and Computing*, vol. 7, S. 131-174, 1996.

[Green et al., 1991] Green, T. R. G., Petre, M. und Bellamy, R. K. E.: “Comprehensibility of Visual and Textual Programs: A Test of Superlativism Against the 'Match-Mismatch' Conjecture.”, in: *Empirical Studies of Programming: Fourth Workshop*, J. Koenemann-Belliveau, T. G. Moher und S. P. Robertson, Eds. New Brunswick, New Jersey: Ablex Publishing Corporation, S. 121-146, 1991.

[Griefahn, 1997] Griefahn, U.: “Reactive Model Computation: A Uniform Approach to the Implementation of Deductive Databases”, Bonn, PhD thesis, 1997.

[Griffiths, 2001] *Griffiths*, *A.*: “Introducing JUnit”, 2001, (www.octopull.demon.co.uk/java/Introducing_JUnit.html).

[Haaks, 1999] Haaks, D.: “Anpaßbare Informationssysteme - Auf dem Weg zu aufgaben- und benutzerorientierter Systemgestaltung und Funktionalität”, Universität Hamburg, Hamburg, Promotion, 1999.

[Hallenberger, 2000] Hallenberger, M.: “Entwicklung einer interaktiven 3D-Benutzerschnittstelle am Beispiel einer Anpassungsschnittstelle für komponentenbasierte Anpassbarkeit”, Universität Bonn, Bonn, Diplomarbeit, 2000.

[Hau und Mertens, 2002] Hau, M. und Mertens, P.: “Computergestützte Auswahl komponentenbasierter Anwendungssysteme”, in: *Informatik Spektrum*, vol. Band 25, Heft 5, S. 331-340, 2002.

[Hawlitzek, 1999] Hawlitzek, F.: *Java-Programmierung mit IBM VisualAge*: Addison-Wesley, 1999.

[Heinlein, 2000] Heinlein, C.: “Workflow und Prozeßsynchronisation mit Interaktionsausdrücken und -graphen. Konzeption und Realisierung eines Formalismus zur Spezifikation und Implementierung von Synchronisationsbedingungen”, Universität Ulm, Ulm, Promotion, 2000.

[Henderson und Kyng, 1991] Henderson, A. und Kyng, M.: “There's No Place Like Home: Continuing Design in Use”, in: *Design At Work - Cooperative Design of Computer Artefacts*, J. Greenbaum und M. Kyng, Eds. Hillsdale, New Jersey: Lawrence Erlbaum Associates, Publishers, S. 219-240, 1991.

[Heuer, 1992] Heuer, A.: *Objektorientierte Datenbanken*: Addison-Wesley, 1992.

[Hils, 1992] Hils, D. D.: “Visual Languages and Computer Survey: Data Flow Visual Programming Languages”, in: *Journal of Visual Languages and Computing*, vol. 3, S. 69-101, 1992.

[Hinken, 1999] Hinken, R.: “Verteilte Anpassbarkeit für Groupware - Eine Laufzeit und Anpassungsplattform.”, Universität Bonn, Bonn, Diplomarbeit, 1999.

[Hoare, 1972] Hoare, C. A. R.: “Proof of Correctness of Data Representation”, in: *Acta Informatica*, vol. 1, S. 271-281, 1972.

[Hoare, 1985] Hoare, C. A. R.: *Communicating Sequential Processes*. Englewood Cliffs, New Jersey: Prentice Hall International, 1985.

[Hutchins et al., 1986] Hutchins, E. L., Hollan, J. D. und Norman, D. A.: “Direct Manipulation Interface”, in: *User centered system design - New Perspectives on Human-Computer Interaction*, S. W. Draper, Ed. Hillsdale, New Jersey: Lawrence Erlbaum Associates, S. 87-124, 1986.

[IBM, 2002] IBM: “Visual age for Java product brochure.”, vol. 2002, 2002, (http://www-3.ibm.com/software/ad/vajava/pdf/g3255352.pdf).

[ISO-9126, 2001] ISO-9126: “ISO/IEC 9126-1, Part 1 - Qualitity Model”. Genf: International Organization for Standardisation, 2001.

[Jablonski et al., 1997] Jablonski, S., Böhm, M. und Schulze, W.: “Workflow- Management. Entwicklung von Anwendungen und Systemen”: Dpunkt Verlag, 1997.

[JavaSoft, 1997] JavaSoft: “JavaBeans 1.0 API Specification”. Mountain View, California: SUN Microsystems, 1997, (http://java.sun.com/products/javabeans/docs/spec.html).

[Jensen, 1994] Jensen, K.: “An Introduction to the Theoretical Aspects of Coloured Petri Nets”, in: *A Decade of Concurrency, Lecture Notes in Computer Science*, vol. Vol. 803, J. W. d. Bakker, W.-P. d. Roever und G. Rozenberg, Eds.: Springer-Verlag, S. 230-272, 1994.

[Jørgensen, 1989] Jørgensen, A. H.: “Using the Thinking-Aloud Method in System Development”, in: *Designing and Using Human-Computer Interfaces and Knowledge Based Systems*, G. Salvendy und M. J. Smith, Eds. Amsterdam: Elsevier Science Publishers B.V., S. 743-750, 1989.

[Kahler, 1995] Kahler, H.: “From Taylorism to Tailorability: Supporting Organizations with Tailorable Software and Object-orientation”, in: Proceedings of HCI '95 - Symbiosis of Human and Artefact - Future Computing and Design for Human-Computer Interaction, Amsterdam, Elsevier, S. 995-1000, 1995.

[Kahler et al., 2000] Kahler, H., Mørch, A., Stiemerling, O. und Wulf, V.: “Tailorable Systems and Cooperative Work, Special Issue des Journal Computer Supported Cooperative Work (JCSCW)”, vol. 9, No. 1, 2000.

[Kahler et al., 1999] Kahler, H., Stiemerling, O., Wulf, V. und Hoepfner, J.-G.: “Gemeinsame Anpassung von Einzelplatzanwendungen”, in: Proceedings of Fachtagung Software-Ergonomie 1999, Heidelberg, S. 183-194, 1999.

[Kemper und Eickler, 1997] Kemper, A. und Eickler, A.: *Datenbanksysteme. Eine Einführung, 2. Auflage*. München: Oldenbourg, 1997.

[Knäpper und Perc, 2000] Knäpper, M. und Perc, P.: *Anwendungsentwicklung unter Lotus Notes/Domino 5 . Konzepte, Technologien*: Addison-Wesley, 2000.

[Kramer, 1998] Kramer, R.: “iContract - The Java - Design By Contract - Tool”, in: Proceedings of Tools '98, ACM-Press, 1998.

[Krings, 2003] Krings, M.: “Erkennung semantischer Fehler in komponentenbasierten Architekturen”, Universität Bonn, Bonn, Diplomarbeit, 2003.

[Krüger, 2002] Krüger, M.: “Semantische Integritätsprüfung für die Anpassung von Komponenten-Kompositionen”, Universität Bonn, Bonn, Diplomarbeit, 2002.

[Kühme et al., 1992] Kühme, T., H. Dieterich, Malinkowski, U. und Schneider-Hufschmidt, M.: “Approaches to Adaptivity in User Interface

Technology: Survey and Taxomony", in: Proceedings of IFIP '92, 1992.

[Leeuwen und Wiedermann, 2000] Leeuwen, J. v. und Wiedermann, J.: "The Turing Machine Paradigm in Contemporary Computing", in: Proceedings of Mathematics Unlimited -- 2001 and Beyond, Springer, 2000.

[Lewis und Olson, 1987] Lewis, C. und Olson, G. M.: "Can Principles of Cognition Lower the Barriers to Programming?", in: *Empirical Studies of Programmers: Second Workshop*, G. M. Olson, S. Sheppard und E. Soloway, Eds. Norwood, New Jersey: Aplex, S. 248-263, 1987.

[Lieberman, 2001] Lieberman, H.: *Your Wish is my Command. Programming by Example*. New York: Morgan Kaufmann Publishers, 2001.

[Lieberman et al., 2002] Lieberman, H., Wulf, V., Paterno, F. und Repenning, A.: "Workshop on End-User Development". Pisa, Italien, 2002.

[Mackay, 1990] Mackay, W. E.: "Users and customizable Software: A Co-Adaptive Phenomenon", MIT, Boston (MA), PhD Thesis, 1990.

[MacLean et al., 1990] MacLean, A., Carter, K., Lövstrand, L. und Moran, T.: "User-tailorable Systems: Pressing the Issue with Buttons", in: Proceedings of Computer Human Interaction (CHI '90), Seattle (Washington), ACM-Press, New York, S. 175-182, 1990.

[Magee et al., 1995] Magee, J., Dulay, N., Eisenbach, S. und Kramer, J.: "Specifying Distributed Software Architectures", in: Proceedings of 5th European Software Engineering Conference, Barcelona, 1995.

[Malone et al., 1994] Malone, T. W., Lai, K.-Y. und Fry, C.: "Experiments with Oval: A Radically Tailorable Tool for Cooperative Work", in: Proceedings of CSCW ´92, Toronto, Canada, ACM Press, S. 289-297, 1994.

[McIlroy, 1968] McIlroy, D.: "Mass-produced software components", in: Proceedings of Conference on Software Engineering, Garmisch-Partenkirchen (D), North Atlantic Treaty Organization (NATO), 1968.

[Meyer, 1988] Meyer, B.: "Eifel: A Language and Environment for Software Engeneering", in: *Journal of Systems and Software*, 1988.

[Meyer, 1992] Meyer, B.: "Applying 'Design by contract'", in: *IEEE Computer*, S. 40-51, 1992.

[Microsoft, 1995] Microsoft: *The Windows Interface Guidelines for Software Design*. Redmond: Microsoft Press, 1995.

[Microsoft, 1996] Microsoft: "The Component Object Model Technical Overview", Microsoft Corp., Microsoft Developer Network, Redmond, USA 1996, (http://premium.microsoft.com/msdn/library/specs/tech1/d1/s1cfa0.htm).

[Milner, 1980] Milner, R.: *A Calculus of Communicating Systems*, vol. 92 (LNCS). Berlin, Heidelberg, New York: Springer-Verlag, 1980.

[Milner et al., 1992] Milner, R., Parrow, J. und Walker, D.: “A calculus of mobile processes, Parts I and II”, in: *Journal of Information and Computing*, vol. 100, S. 1-77, 1992.

[Momjian, 2001] Momjian, B.: *PostgreSQL - Introduction and Concepts*: Addison Wesley, 2001.

[Myers, 1990] Myers, B. A.: “Taxonomies of Visual Programming and Program Visualization”, in: *Journal of Visual Languages and Computing*, vol. 1(1), S. 97-123, 1990.

[Nardi, 1993] Nardi, B. A.: *A Small Matter of Programming - Perspectives on end-user computing*. Cambridge et al: MIT-Press, 1993.

[Newham und Rosenblatt, 1998] Newham, C. und Rosenblatt, B.: *Learning the bash Shell.*, 2. Auflage ed: O'Reilly & Associates, 1998.

[Ng und Kramer, 1995] Ng, K. und Kramer, J.: “Automated Support for Distributed Software Design”, in: Proceedings of CASE ´95 (International Workshop on Computer-Aided Software Engineering), Toronto, Canada, 1995.

[Nielsen, 1992] Nielsen, J.: “Evaluating the Thinking-Aloud Technique for Use by Computer Scientists”, in: *Advances in Human Computer Interaction*, vol. 3, S. 69-82, 1992.

[Nielsen, 1993] Nielsen, J.: *Usability Engeneering*: Morgan Kaufmann, 1993.

[Nielsen, 1994] Nielsen, J.: “Heuristic Evaluation”, in: *Usability Inspection Methods*, J. Nielsen und R. L. Mack, Eds. New York: John Wiley & Sons, S. 25-62, 1994.

[Oberquelle, 1994] Oberquelle, H.: “Situationsbedingte und benutzerorientierte Anpaßbarkeit von Groupware”, in: *Menschengerechte Groupware - Softwareergonomische Gestaltung und partizipative Umsetzung*, A. Hartmann, T. Herrmann, M. Rohde und V. Wulf, Eds. Stuttgart: Teubner, S. 32-50, 1994.

[OMG, 1999] OMG: “Unified Modeling Language Specification, Version 1.3”, Object Management Group 1999.

[Pane und Myers, 1996] Pane, J. F. und Myers, B. A.: “Usability Issues in the Design of Novice Programming Systems”, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, Technical Report, Nr: CMU-CS-96-132, August 1996, 1996.

[Pane et al., 2001] Pane, J. F., Ratanamahatana, C. A. und Myers, B. A.: “Studying the language and structure in non-programmers' solutions to programming problems”, in: *International Journal of Human Computer Studies*, vol. 54, No. 2, S. 237-264, 2001.

[Papadimitriou, 1994] Papadimitriou, C. H.: *Computational Complexity*. Reading, Massachusetts: Addison-Wesley Publishing, 1994.

[Paul, 1994] Paul, H.: *Exploratives Agieren*. Frankfurt/M: Peter Lang, 1994.

[Petri, 1979] Petri, C. A.: “Introduction to General Net Theory”, in: Proceedings of Proceedings of the Advanced Course on General Net

	Theory of Processes and Systems 1979, Hamburg, Springer-Verlag, S. 1-19, 1979.
[Picot et al., 1998]	Picot, A., Reichwald, R. und Wigant, R.: *Die grenzenlose Unternehmung - Information, Organisation und Management*, 3. Auflage ed. Wiesbaden: Gabler, 1998.
[Plásil et al., 2000]	Plásil, F., Visnovsky, S. und Besta, M.: "Behavior Protocols", Dep. of SW Engineering, Charles University, Prague, Technical Report, Nr: 2000/7, August 2000, 2000.
[Pospísil, 1999]	Pospísil, R. P., F.: "Describing the functionality of EJB using the behavior protocols", Presented at the week for doctoral studies 1999.
[Raj, 1998]	Raj, G. S.: "A Detailed Comparison of CORBA, DCOM and Java/RMI", Object Management Group (OMG) whitepaper, September '98. 1998, (http://my.execpc.com/~gopalan/misc/compare.html).
[Raskin, 2000]	Raskin, J.: *The Humane Interface - New directions for Designing Interactive Systems*: Addison Wessley, 2000.
[Reisig, 1990]	Reisig, W.: *Petrinetze, Eine Einführung*, 2.Auflage ed: Springer-Verlag, 1990.
[Repenning und Summer, 1995]	Repenning, A. und Summer, T.: "Agentsheets: A Medium for Creating domain-Oriented Visual Languages", in: *IEEE Computer*, vol. März 1995, S. 17-25, 1995.
[Reussner, 2001]	Reussner, R. H.: "The use of parameterised contracts for architecting systems with software components", in: Proceedings of Sixth International Workshop on Component-Oriented Programming (WCOP'01), 2001.
[Reussner, 2001a]	Reussner, R. H.: "Parameterisierte Verträge zur Protokolladaption bei Software-Komponenten", Universität Karlsruhe, Karlsruhe, Dissertation, 2001a.
[Sane und Harris, 2003]	Sane, A. und Harris, D.: "Design By Contract Assertions Vs Unit Tests Vs Types", vol. 2003: Wiki, 2003, (http://c2.com/cgi/wiki?DesignByContractAssertionsVsUnitTestsVsTypes).
[Schneider-Hufschmidt et al., 1993]	Schneider-Hufschmidt, M., Kühme, T. und Malinowski, U.: *Adaptive User Interfaces: Principles and Practice*. Amsterdam: North Holland Elsevier, 1993.
[Shapiro, 1991]	Shapiro, R. M.: "Validation of a VLSI Chip Using Hierarchical Colored Petri Nets", in: *Microelectronics and Reliability*, vol. Vol. 31, S. 607-625, 1991.
[Silberschatz et al., 2001]	Silberschatz, A., Korth, H. F. und Sudarshan, S.: *Database System Concepts*: Osborne McGraw-Hill, 2001.
[Smith et al., 1994]	Smith, D. C., Cypher, A. und Spohrer, J.: " KidSim: Programming Agents Without a Programming Language", in: *Communications of the ACM*, vol. 37(7), S. 54-67, 1994.

[Soloway et al., 1983] Soloway, E., Bonar, J. und Ehrlich, K.: “Cognitive Strategies and Looping Constructs: An Empirical Study”, in: *Communications of the ACM*, vol. 26, No. 11, S. 853-860, 1983.

[Stiemerling, 1997] Stiemerling, O.: “CAT: Component Architecture for Tailorability”, Universität Bonn, Bonn, Arbeitspapier, 1997.

[Stiemerling, 1998] Stiemerling, O.: “FlexiBeans Specification”, University of Bonn, Department for Computer Science III, Bonn, Working Paper 1998.

[Stiemerling, 2000] Stiemerling, O.: “The Evolve Project”, Universität Bonn, Bonn, Dissertation, 2000.

[Stiemerling und Cremers, 1998] Stiemerling, O. und Cremers, A. B.: “Tailorable Component Architectures for CSCW-Systems”, in: Proceedings of 6. Euromicro Workshop on Parallel and Distributed Programming, Madrid, Spanien, IEEE-Press, S. 302-308, 1998.

[Stiemerling et al., 2000] Stiemerling, O., Won, M. und Wulf, V.: “Zugriffskontrolle in Groupware - Ein nutzerorientierter Ansatz”, in: *Wirtschaftsinformatik*, vol. Heft 4, 2000.

[Stonebraker, 1975] Stonebraker, M.: “Implementing Integrity Constraints and Views by Query modification”, in: Proceedings of 1975 ACM SIGMOD Intl. Conf. On Management of Data, San Jose, USA, S. 65-78, 1975.

[SUN, 1998] SUN: “Getting Started Using RMI”, SUN Microsystems, Mountain View, California 1998, (http://java.sun.com/products/javaspaces/tutorials/api/html /getstartdoc.html).

[SUN, 2001] SUN: *Java Look and Feel Design Guidelines*, zweite Auflage ed: Addison Wesley, 2001.

[Szyperski, 1998] Szyperski, C.: *Component Software - Beyond object-orientated programming*. New York: ACM Press, 1998.

[Szyperski und Pfister, 1996] Szyperski, C. und Pfister, C.: “Component-Oriented Programming: WCOP '96 Workshop Report”, in: Proceedings of ECOOP 1996, Linz (A), Springer Verlag, S. 127-130, 1996.

[TogetherSoft, 2002] TogetherSoft: “Together Soft: Practical Guides”, vol. 2002, 2002, (http://www.togethersoft.com/services/practical_guides /index.jsp).

[Turing, 1936] Turing, A. M.: “On computable numbers, with an application to the Entscheidungsproblem”, in: *Proc. London Math. Soc.*, vol. 42-2, S. 230-265, 1936.

[van der Aalst et al., 1997] van der Aalst, W., Hauschildt., D. und Verbeek, H. M. W.: “A Petri-Net-based Tool to analyze workflows”, in: Proceedings of Petri Nets in System Engineering (PNSE'97), Hamburg, Universität Hamburg, S. 78-90, 1997.

[van der Aalst und van Hee, 2002] van der Aalst, W. und van Hee, K.: *Workflow Management - Models, Methods, and Systems*. Cambridge, Massachusetts: MIT Press, 2002.

[W3C, 2000] W3C: “Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation”, World Wide Web Consortium, W3C 2000, (http://www.w3.org/TR/REC-xml.).

[Wagner und Mehner, 2000] Wagner, A. und Mehner, K.: “Aspektorientierte Entwicklung nebenläufiger Systeme”, Universität Paderborn, Paderborn, Forschungsbericht 2000.

[Waskiewicz, 1995] Waskiewicz, F.: “An object-oriented Framework for Manufacturing”, OMG BOMSI, Ottawa, Research Report 1995.

[Wegner, 1996] Wegner, P.: “Interactive Software Technology”, in: *The Computer Science and Engineering Handbook*, A. B. J. Tucker, Ed.: CRC Press, in cooperation with ACM, 1996.

[Wegner und Goldin, 2003] Wegner, P. und Goldin, D.: “Computation beyond turing machines”, in: *Communications of the ACM*, vol. 46, Issue 4, S. 100 - 102, 2003.

[Won, 1998] Won, M.: “Komponentenbasierte Anpassbarkeit - Anwendung auf ein Suchtool für Groupware”, Universität Bonn, Bonn, Diplomarbeit, 1998.

[Won, 2000] Won, M.: “Checking integrity of component-based architectures”, in: Proceedings of CSCW 2000, Workshop on Component-Based Groupware, Philadelphia, USA, 2000.

[Won und Cremers, 2002] Won, Markus, Cremers, Armin B.: “Supporting End-User Tailoring of Component-Based Software - Checking Integrity of Composition“, in: Proceedings of Colognet 2002 (Conjuction with LOPSTR 2002), Madrid, Spain, 19.-20.09.2002.

[Won, 2003] Won, Markus: “Supporting End-User Development of Component-Based Applications by Checking Semantic Integrity“, in: Workshop in Perspectives on End-User Development, in Conjunction with CHI 2003, Fort Lauderdale, Florida, August 2003.

[Won, 2003a] Won, Markus: “Supporting End-User Development of Component-Based Software by Checking Semantic Integrity“, in: ASERC Workshop on Software Testing, 19.2.2003, Banff, Canada, 2003.

[Won und Wulf, 2003] Won, Markus; Wulf, Volker: “Anpassungsumgebung für komponentenbasierte Software: Kooperativ und lernförderlich“, in: Oberquelle, Horst; Prinz, Wolfgang; Ziegler, Jürgen (Hrsg.): i-com - Zeitschrift für interaktive und kooperative Medien, 1/2003, 2003, S. 28-34.

[Wulf, 1999] Wulf, V.: “Let's see your Search-Tool! - Collaborative use of Tailored Artifacts in Groupware“, in: Proceedings of GROUP '99, New York, ACM-Press, S. 50-60, 1999.

[Wulf, 2000] Wulf, V.: “Exploration Environments: Supporting Users to Learn Groupware Functions”, in: *Interacting with Computers*, vol. 13, No. 2, S. 265-299, 2000.

[Wulf und Golombek, 2001] Wulf, V. und Golombek, B.: “Direct Activation: A Concept to Encourage Tailoring Activities”, in: *Behavior and Information Technology*, vol. 20, No. 4, S. 249-263, 2001.

[Yakovlev et al., 1996] Yakovlev, A., A.M.Koelmans, Semenov, A. und D.J.Kinniment: “Modelling, Analysis and Synthesis of Asynchronous Control Circuits Using Petri Nets”, in: *VLSI Journal*, vol. Vol. 21, S. 143-170., 1996.

[Zdun, 2002] Zdun, U.: “Language support for dynamic and evolving software architectures”, Universität Essen, Essen, Dissertation, 2002.

Anhang A FREEVOLVE – Eine Beschreibung der Architektur aus Anwendersicht

Die FREEVOLVE-Laufzeitumgebung ist modular aufgebaut. Für das Verständnis der Arbeit und der Zusammenhänge soll hier kurz ihre Funktionsweise dargestellt werden. Dabei geht es nicht so sehr darum, die Architektur im Detail zu diskutieren, sondern die groben Abläufe beim Start einer Applikation und während des Anpassungsvorgangs darzustellen. Für das Verständnis ist ebenso nicht von vordringlichem Interesse, dass es sich bei FREEVOLVE um eine verteilte Plattform handelt. Aus diesem Grund wird zu Beginn auf diese Unterscheidung (Client vs. Server) verzichtet und später nachgereicht.

Abbildung 44 zeigt schematisch den Aufbau der FREEVOLVE-Plattform. Die beiden Module in der Mitte (Kontrolle, Anpassungsoperationen) stellen den Kern des Systems dar. Beim Start durchsucht das Kontrollmodul die FREEVOLVE-Datenbasen nach Beschreibungen von Applikationen im CAT-Format. Solche Applikationsbeschreibungen enthalten Informationen darüber, welche Komponenten zu verwenden sind und wie diese parameterisiert und verbunden werden müssen. Jede Applikation hat einen eindeutigen Namen.

Die verfügbaren Applikationen werden in der Laufzeitumgebung angezeigt. Sie können vom Benutzer ausgewählt werden. Geschieht dies, lädt das Kontrollmodul die Applikation in folgenden Schritten:

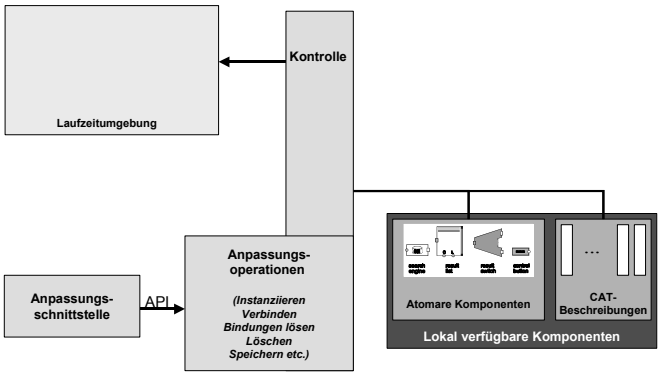


Abbildung 44: FREEVOLVE im Überblick

Die zur Applikation gehörige CAT-Datei wird gelesen und analysiert (siehe Abbildung 45, rechts oben). Daraus ergibt sich, welche Komponenten benötigt werden. Sie werden aus dem Komponenten-Repository (Abbildung 45, rechts unten) geladen. Jede Komponente wird dann zusammen mit einem sie umgebenden Proxy-Objekt in der Laufzeitumgebung instanziiert. Anschließend werden die einzelnen Komponenten parameterisiert und entsprechend dem Kompositionsplan (CAT-Datei) miteinander verbunden. Die Applikation ist nun lauffähig.

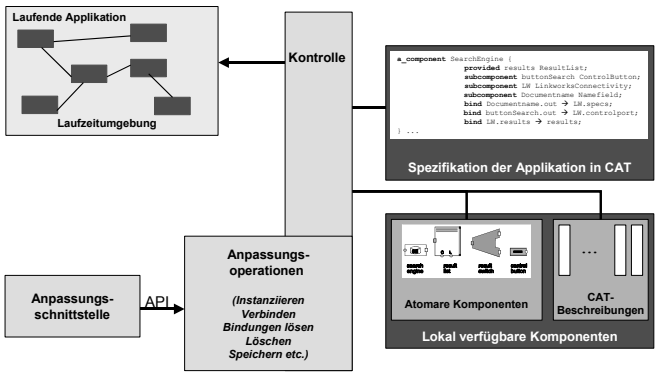


Abbildung 45: FREEVOLVE mit geladener Applikation

Die Proxy-Objekte dienen zur Kontrolle der ansonsten völlig vom System unabhängigen Komponenten. Sie stellen Funktionen zum Zugriff auf die Komponenten bereit und erlauben so, das Erzeugen oder Lösen von Verbindungen, die Änderung der Parameter und das Löschen einer nicht mehr benötigten Komponente. Zu diesem Zweck stellt die FREEVOLVE-Plattform ein API mit den benötigten Anpassungsoperationen zur Verfügung.

Auf diese Schnittstelle wird von der Anpassungsumgebung aus zugegriffen. Sie erlaubt es, alle vorhandenen Proxy-Objekte und die enthaltenen Komponenten zu identifizieren. Über den Umweg der Proxy-Objekte lassen sich der Komponenten auch auf verschiedene Weisen visualisieren. Zu diesem Zweck stellen die Proxy-Objekte verschiedene Routinen bereit. Änderungen, die in der Anpassungsumgebung vorgenommen werden, können – je nach Design der Anpassungsumgebung – direkt umgesetzt werden, da es möglich ist, die Proxy-Objekte direkt anzusteuern und so Änderungen unmittelbar auf die Komposition zu übertragen.

In der in dieser Arbeit konzipierten Anpassungsumgebung ist dies anders gelöst worden. Hier werden Änderungen akkumuliert und sind vorerst nur in der Anpassungsumgebung sichtbar. Sie werden auf der abstrakten Sicht auf die Komposition durchgeführt und erst dann übertragen, wenn der Anpassende dies explizit veranlasst. Der Vorteil liegt hier darin, dass die Komposition nicht so leicht in einen inkonsistenten Zustand gerät. Der Austausch einer Komponente ist beispielsweise ein problematischer Anpassungsvorgang. Zuerst muss eine alte Komponente entfernt werden. Erst dann kann die neue hinzugefügt werden. In dem Moment, in dem die Komponente entfernt ist, die neue aber noch nicht eingefügt wurde, kann die Applikation nicht korrekt im Sinne des Anwenders arbeiten. Hier können also Fehler auftreten. Durch die Sammlung verschiedener Anpassungsoperationen und die gemeinsame Durchführung aller Änderungsschritte wird dieser Fehler vermieden.

FREEVOLVE als Client-Server-Plattform

Bisher wurde die FREEVOLVE-Plattform nicht in Bezug auf verteilte Applikationen untersucht. Dies ist für das grundlegende Verständnis auch nicht notwendig. Eine Erweiterung in Bezug auf Verteiltheit sowohl der Plattform als auch der auf ihr lauffähigen Applikationen sollte dennoch diskutiert werden. Abbildung 46 zeigt die Erweiterung schematisch:

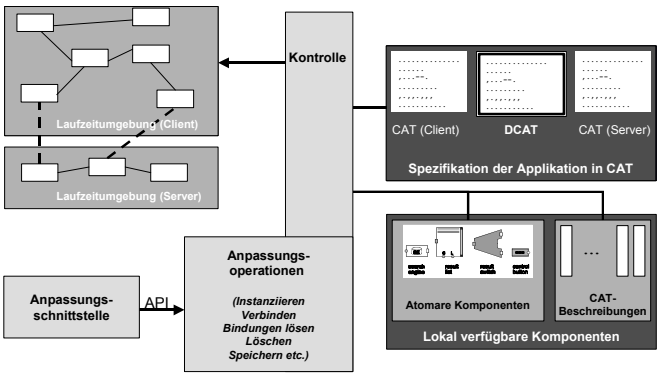


Abbildung 46: Client-Server-Applikationen

Das Kontrollmodul liegt jetzt sowohl auf dem Client (Zugriff auf die Client-seitige Laufzeitumgebung und Bereitstellung des Anpassungs-API) als auch auf dem Server (Verwaltung der CAT- und Komponenten-Repositories). Wichtig ist, dass alle Komponenten und auch die Kompositionspläne auf dem Server verbleiben. Sie werden bei Bedarf auf den Client transferiert und dort instanziiert. Für eine Applikation werden nun drei zusammengehörige CAT-Dateien benötigt. Ein Server-CAT-Plan beschreibt die Server-Applikation. Entsprechendes gilt für das Client-CAT-File. Ein zusätzliches DCAT-File (= Distributed CAT) beschreibt Verbindungen zwischen Client und Server.

Über das Anpassungs-API kann von der Client-Seite aus auf die Gesamtapplikation zugegriffen werden. Dazu werden für die entfernten Komponenten auf der Client-Seite sogenannte Schattenobjekte verwaltet. Dies sind leere Proxy-Objekte, die die entfernten Komponenten symbolisieren und Zugriffe auf sie verwalten. Für eine Komponente bleibt so transparent, ob sie mit einer lokalen Komponente oder einer entfernten kommuniziert. Zugriffe erfolgen in beiden Fällen über die entsprechenden Proxy-Objekte.

Aus Sicht der FREEVOLVE-Plattform gibt es keine Einschränkungen, wer Änderungen an einer Komposition vornehmen darf. Die Rechteverwaltung muss also in der Anpassungsumgebung erfolgen. Für die hier bearbeitete Fragestellung ist die Beschreibung und Durchsetzung von Anpassungsrechten nicht relevant. Es wurde hier auf eine Einschränkung des Anpassungsmodus verzichtet, so dass jeder Benutzer beliebige Anpassungen vornehmen kann. Selbstverständlich kann dies zu unbrauchbaren Kompositionen führen. Auch Abstürze durch Veränderungen speziell an der Server-Applikation und daraus resultierenden Inkonsistenzen zwischen dem geänderten Server-Verhalten und anderen Clients können auftreten.

Anhang B XSemL im Überblick

Die XSemL definierende DTD ist im Folgenden dargestellt und kommentiert. Sie bildet die Grundlage zur Definition eigener Integritätsbedingungen. Die Dokumentwurzel wird definiert durch *semantics*. Das Element kann dann entweder einen *EFITree* (Färbungsbaum) oder Integritätsbedingungen bezogen auf einzelne Komponenten (einfache sowie abstrakte) enthalten.

```
<?xml version="1.0" encoding="US-ASCII"?>
   <!ELEMENT semantics (efitree|semantic+)>
        <!ELEMENT semantic efi?|constraints?|descriptions?|solutions?)*>
             <!ATTLIST semantic
                  component CDATA #IMPLIED
                  type CDATA #IMPLIED
             >
```

Abbildung 47: Dokumentwurzel für XSemL-Dateien

Für den *EFITree* (Färbungsbaum) sind keine weiteren syntaktischen Vorgaben definiert. In einer Baumstruktur werden die einzelnen Typen und Subtypen definiert. Dieser Baum wird verwendet, um zu prüfen, inwieweit ein Event gültig für einen Eingangs- bzw. Ausgangsport ist.

```
<!ELEMENT efitree ANY>
```

Abbildung 48: Definition "EFITree"

Ein mögliches Beispiel für einen Färbungsbaum ist im Folgenden dargestellt:

```
<efitree>
   <IO-Events/>
        <Daten>
        <Zahlen/>
        <Texte>
             <Deutsch/>
             <Englisch/>
             <Französisch/>
             <klein/>
             <gross/>
        </Texte>
        <Grafiken/>
        </Daten>
        <...../>
</efitree>
```

Abbildung 49: Beispiel für einen Färbungsbaum

Die unterschiedlichen Schnittstellen werden pro Komponente definiert. Hier werden Informationen über den Typ (*Quelle*, *Senke* oder *Traeger*) benötigt. Außerdem muss definiert werden, welche Farbe(n) der Port erzeugt (*Quelle*) oder akzeptiert (*Senke*, *Traeger*). In Kapitel 8 wurde diskutiert, dass ein Eventtyp mehrere Farben gleichzeitig haben kann. Der Modus legt schließlich fest, ob der Port *optional* oder *essentiell* für die sinnvolle Nutzung der Komponente ist. Bei den Trägern (Ports, die Nachrichten bearbeiten und – evtl. umgefärbt – weiterleiten) kommen noch optional *Transitionen* hinzu. Diese bestimmen, welche anliegenden weiteren Ports zu einer *Transition* zusammenzufassen sind.

```
<!ELEMENT efi (type+)>
   <!ELEMENT type (quelle|senke|traeger)>
        <!ELEMENT quelle (farbe,modus)+>
        <!ELEMENT senke (farbe,modus)+>
        <!ELEMENT trager (transitionen*, (farbe,modus)+>

        <!ELEMENT farbe (#PCDATA)>
        <!ELEMENT modus (#PCDATA)>

        <!ELEMENT transitionen (#PCDATA)>
```

Abbildung 50: EFI-Informationen

Weiterhin lassen sich Beschreibungen zu den Komponenten, deren Ports oder ausgewählten Parametern definieren. Sie werden bei Bedarf als Informationen beim Anpassen angezeigt. Sie dienen auch als Hinweise, wenn die Integritätsprüfung Fehler in einer der beteiligten Komponenten ausmachen kann.

```
<!ELEMENT descriptions (component?,port*,parameter*,subcomponent*)*>

        <!ELEMENT component (#PCDATA)*>
             <!ATTLIST component
                  image CDATA #IMPLIED
             >

        <!ELEMENT port (#PCDATA)*>
             <!ATTLIST port
                  name CDATA #REQUIRED
             >

        <!ELEMENT parameter (#PCDATA)*>
             <!ATTLIST parameter
                  name CDATA #REQUIRED
             >

        <!ELEMENT subcomponent (#PCDATA)*>
             <!ATTLIST subcomponent
                  name CDATA #REQUIRED
             >
```

Abbildung 51: Descriptions

Constraints legen Bedingungen für die Nutzung von Komponenten fest. Eine Beschreibung, welche Bedingungen zulässig sind, findet sich in Kapitel 7. Sie sind nicht Bestandteil der DTD und werden direkt von der Integritätsprüfung auf Korrektheit überprüft.

```
<!ELEMENT constraints (constraint)*>

   <!ELEMENT constraint (param*)>
        <!ATTLIST constraint
             type CDATA #REQUIRED
             operation CDATA #REQUIRED
             errorlevel CDATA #REQUIRED
             description CDATA #REQUIRED
             name CDATA #IMPLIED
             value CDATA #IMPLIED
        >

   <!ELEMENT param EMPTY>
        <!ATTLIST param
             type CDATA #IMPLIED
             name CDATA #REQUIRED
             value CDATA #REQUIRED
        >
```

Abbildung 52: Constraints

Eine Erweiterung der Constraints stellen die Restricted Solutions dar. Hier werden im Falle eines Fehlers Korrekturvorschläge in Form von Aktionen (*action*) beschrieben. Mögliche Aktionen sind hier die Änderung eines Parameters auf einen neuen Wert o.ä. Mögliche Aktionen sind ebenso wie die Bedingungen nicht Teil der DTD. In Kapitel 7 finden sich detaillierte Tabellen, die mögliche Aktionen beschreiben.

```
<!ELEMENT solutions (if)*>
   <!ELEMENT if (constraints,then)>
   <!ELEMENT then (action*)>
   <!ELEMENT action (param*)>

        <!ATTLIST action
             command CDATA #REQUIRED
             reason CDATA #REQUIRED
        >
```

Abbildung 53: Restricted Solutions

Anhang C Zwei Beispielapplikationen

Zur Evaluation der im Rahmen dieser Arbeit entwickelten Konzepte und deren Umsetzung in einer Anpassungsumgebung für die FREEVOLVE-Plattform wurden Benutzerstudien durchgeführt. In den in den Kapiteln 5 und 10 vorgestellten Testszenarien hatten die Probanden verschiedene Aufgaben zu bearbeiten. Sie alle basierten darauf, existierende Kompositionen zu verändern und Fehler in ihnen zu erkennen.

Zu diesem Zweck wurden zwei unterschiedliche Komponenten-Sets entworfen. Mit dem ersten, recht einfachen Komponenten-Set lässt sich ein imaginärer Computer entwerfen. Es dient zur Einführung in die komponentenbasierte Anpassbarkeit. Das zweite Set enthält Komponenten, mit denen sich ein Kommunikationswerkzeug konstruieren lässt. Es ist deutlich aufwendiger und komplexer. Im Folgenden sollen nun beide Komponenten-Sets und die sich daraus konstruierbaren Applikationenfamilien kurz vorgestellt werden.

Einführendes Beispiel - Modularer Computer

Für viele der Probanden, die an der Benutzerstudie teilnahmen, war das Konzept der komponentenbasierten Anpassbarkeit neu. Aus diesem Grund wurde zur Einführung ein sehr einfaches Beispiel-Komponenten-Set entworfen. Es besteht aus Komponenten, die gemeinsam einen Computer bilden (Tastatur, Maus, Monitor, Lautsprecher, Gehäuse etc.). Dieses Beispiel wurde vor dem Hintergrund gewählt, dass durch eine direkte Entsprechung der Komponenten zu bekannten Elementen in der realen Welt der Zugang zum komponentenbasierten Anpassen erleichtert wird.

Keine der Komponenten verfügt über Funktionalität, insofern lässt sich aus ihnen auch keine lauffähige Applikation konstruieren. Stattdessen erläutern Beschreibungen (Descriptions) der einzelnen Komponenten ihre Funktionsweise und ihre Anforderungen an ihre Umwelt. Weiterhin wurde Wert darauf gelegt, dass Schnittstellen sauber definiert wurden und verschiedene Parameter veränderbar sind. Detaillierte Integritätsbedingungen erfordern eine exakte Komposition, an deren Ende die Applikation von der Integritätsprüfung als fehlerfrei erkannt werden kann.

Das Komponenten-Set wurde ausschließlich in den Benutzerstudien eingesetzt, um den Zugang sowohl zum grundlegenden Konzept zu erleichtern als auch den Umgang mit der Anpassungsumgebung, ihren unterschiedlichen und synchronisierten Sichten und den verschiedenen möglichen Anpassungsoperationen zu schulen.

GeoChat – Ein komponentenbasiertes Kommunikationswerkzeug

Ein weiteres Komponenten-Set wurde im Rahmen des vom BMBF geförderten Projekt „Geoinformation – Neue Medien für die Einführung eines Querschnittfachs“ entworfen. Es enthält Komponenten zur Konstruktion eines Kommunikationswerkzeugs „GeoChat“.

Ziel des Projekts ist es, die Lehre im Querschnittsstudienfach „Geo-Information“ multimedial zu unterstützen. Zu diesem Zweck werden verschiedene Lernmodule entwickelt, die sich sowohl für die Präsenzlehre (animierte Präsentationsmaterialien) als auch für das Selbststudium (multimediale Lerneinheiten mit Tests) eignen. Weiterhin soll die Kommunikation zwischen den Studierenden beim Selbststudium unterstützt werden.

Die Anforderungen an den GeoChat gehen weit über die allgemein bekannten Chat-Systeme hinaus. Eine kurze Beschreibung der Hauptanforderungen soll dies unterstreichen:

- **Aufzeichnung von Sessions und erweiterte Protokollierung**: Eine Chat-Session sollte aufgezeichnet werden können. Dabei ist wichtig, alle Beiträge so aufzuzeichnen, dass sich der komplette Ablauf einer Sitzung nachvollziehen lässt. Die Speicherung sollte in einem lesbaren Format erfolgen, so dass Nachbearbeitungen oder das Einfügen von Kommentaren oder Zusammenfassungen leicht möglich sind. Alternativ dazu ist auch möglich, die Aufzeichnung mit einem zusätzlichen (externen) Werkzeug nachzubearbeiten. Sessions sollten sich weiterhin zu Lernzwecken laden und anzeigen lassen.
- **Verwaltung unterschiedlicher Sessions**: Da der Übungsbetrieb ebenso unterstützt werden soll wie die Zusammenarbeit in kleinen Lerngruppen, ist es notwendig, parallele Sessions zuzulassen. Diese sollten unabhängig voneinander ablaufen. Auch Wechsel zwischen schon begonnenen Sessions sollten z. B. für Übungsleiter möglich sein. Hier ist wiederum die Protokollierung einer Session notwendig, denn nur so kann der Diskussionsstand schnell nachvollzogen werden.
- **Integration verschiedener Kommunikationswerkzeuge**: Ein Chat sieht grundsätzlich die textuelle Kommunikation im Vordergrund. Für verschiedene Fragestellungen im Bereich der Geo-Information bietet es sich an, über Skizzen zu diskutieren. Die Integration eines verteilten Whiteboard scheint deswegen notwendig. Dabei werden die Zeichenbefehle in Textformate übertragen und so über das Basis-Chat-System zwischen den beteiligten Rechnern ausgetauscht. Ebenso sollte das System so offen gestaltet sein, dass auch andere gemeinsam benutzte Werkzeuge leicht integriert werden können[30]. Hierzu ist die Festlegung eines standardisierten Kommunikationsprotokolls notwendig.
- **Verwaltung von Verweisen auf Chat-Beiträge**: In Diskussionen ist es häufig notwendig, auf vorher eingebrachte Beiträge verweisen zu können. Zu diesem Zweck wird ein Verweis-Konzept benötigt. Bezüge lassen sich dann je nach Darstellungsform visualisieren. Sie sollen für alle Kommunikationswerkzeuge (beispielsweise also auch das Whiteboard) zur Verfügung stehen.
- **Flexibilität in Bezug auf wechselnde Anforderungen an Darstellung und Funktionalität**: Je nach Rolle in einer Diskussion werden verschiedene Funktionalitäten benötigt. Da der Platz auf dem Bildschirm begrenzt ist und auch da mit der Bereitstellung einer Funktionalität ein Recht, sie zu benutzen, einhergeht, sollten verschiedene Chat-Clients rollen- und aufgabenabhängig zur Verfügung stehen. Beispielsweise sollten nicht alle Übungsgruppenteilnehmer beliebig Sessions wechseln können, dies ist den Gruppenleitern vorbehalten. Weiterhin ist in einigen Diskussionen ein Whiteboard unnötig und kann deswegen ausgeblendet werden.
  Solche Wünsche erfordern ein hohes Maß an Flexibilität und Anpassbarkeit der Anwendung. Es zeigt sich, dass der komponentenbasierte Ansatz geeignet ist, um ein solches Werkzeug oder eine entsprechende Familie von Werkzeugen zu entwickeln.

[30] Ein Beispiel dafür ist eine Programmier-Umgebung *GeoJava* das in einem weiteren Teilprojekt entwickelt wird.

Nach einer eingehenden Analyse der genauen Anforderungen wurden diverse Komponenten entworfen [Krüger, 2002]. Sie lassen sich so komponieren, dass alle bisher aufgeführten Anforderungen erfüllt werden können. Die Komponenten lassen sich funktional in die Aufgaben des Servers (Server, SessionManager, LoginManager, ProtocolManager), der Sitzung (Session) und des Clients aufteilen. Zur Darstellung der Sitzung, des Empfangs und der Ausgabe von Mitteilungen dienen verschiedene visuelle Komponenten. Sie sollen im Folgenden detaillierter beschrieben werden:

- **Server**: Die (abstrakte) Server-Komponente stellt alle Basisfunktionalitäten des Chat-Servers zur Verfügung. Sie enthält die weiter unten beschriebenen Management-Komponenten, die wiederum über die Server-Komponente mit den Clients kommunizieren.
- **SessionManager:** Der SessionManager ist eine Komponente des Servers, der die einzelnen Sitzungen auf den Clients registriert und bei Bedarf zur Verfügung stellt. Es stellt dazu Methoden zur Verfügung, die es erlauben Informationen über alle im System befindlichen Sessions zu erhalten. Das Management kann auch Protokolle von Sitzungen an einen ProtocolManager weiterleiten.
- **ProtocolManager:** Der ProtocolManager sorgt für das Erzeugen und Abspeichern der Protokolle. Weiterhin wird Funktionalität bereitgestellt, um schnell auf gespeicherte Sessions zugreifen zu können.
  Der ProtocolManager bietet darüber hinaus auch Funktionen zur Unterstützung des LoginManagers bei der Überprüfung von Login-Kennungen. Diese Kombination wurde vor allem gewählt, da für beide Funktionalitäten auf Serverseite eine Datenbankanbindung notwendig ist.
- **LoginManager:** Der LoginManager realisiert eine Nutzerverwaltung. Hauptaufgabe ist die Überprüfung von Benutzerkennungen, die durch den ProtocolManager überprüft werden.
- **Session:** Session-Komponenten stellen die Knotenpunkte bei der Kommunikation im Chat-System dar. Sie dienen dazu, Nachrichten, die von unterschiedlichen Chat-Komponenten (Text, Whiteboard, GeoCafe etc.) erzeugt werden, zu empfangen und an die entsprechenden Komponenten auf anderen Clients weiterzuleiten. Um die Flexibilität des Systems zu erhöhen und den Server zu entlasten, ist es möglich Session-Komponenten sowohl auf dem Server als auch auf einem Client zu instanziieren.
  Die Session-Komponente lässt sich weiterhin mit dem ProtocolManager verbinden, so dass die aktuelle Session dauerhaft aufgezeichnet wird.
- **Client:** Die Client-Komponente ist eine abstrakte Komponente. Sie dient als Adapter zwischen der Server-Komponente und den verschiedenen Interface-Komponenten. Da u. U. recht viele Komponenten mit der Client-Komponente zu verbinden sind und dazu unterschiedliche Ports genutzt werden, sind Erläuterungen und Integritätsbedingungen notwendig, um Fehler bei der Komposition zu vermeiden.

- **Interface-Komponenten:** Verschiedene Interface-Komponenten ermöglichen die Interaktion mit dem Chat-System. Dazu gehören vor allem unterschiedliche Komponenten zur Anzeige der im System registrierten Benutzer, zur Verwaltung der verschiedenen Sessions und zum Öffnen und Bearbeiten früherer Sessions.
  Weiterhin gehören auch die eigentlichen Chat-Komponenten zu den Interface-Komponenten. Hier finden sich verschiedene Chat-Anzeige-Fenster, die jeweils unterschiedliche Aspekte der Ergonomie betonen und für unterschiedliche Bildschirmgrößen geeignet sind. Auch diverse Eingabekomponenten (einfacher Text, Unterstützung von HTML-Text-Elementen, Emoticons etc.) werden angeboten. Das Whiteboard wurde bisher als monolithische Komponente realisiert. Auch hier ist zu überlegen, inwieweit eine Zerlegung sinnvoll ist.

Diese deutlich komplexere Anwendung wurde in den Benutzerstudien vorgestellt. Auch hier waren Aufgaben zu lösen, die zeigen sollten, inwieweit die Anpassungsumgebung geeignet ist, Anpassungen von Endanwendern selbst durchführen lassen zu können. Weiterhin sollte überprüft werden, ob durch die Zuhilfenahme einer Integritätsprüfung der Anpassungsprozess erleichtert werden kann und ob sich Fehler in den resultierenden Applikationen vermeiden lassen. Beide Fragen geben indirekt einen Aufschluss darüber, inwieweit eine Integritätsprüfung sich als Unterstützungsmechanismus beim Anpassen lernförderlich bzgl. der Verwendung einer Anpassungsschnittstelle ebenso wie eines konkreten Komponenten-Sets auswirken kann.

Zur Überprüfung des Nutzens einer Integritätsprüfung wurden zu allen Komponenten semantische Informationen bereitgestellt. Dazu gehören auch Descriptions, die die Funktionalität der Komponenten und die Verwendung ihrer Ports und Parameter detailliert illustrieren. Zusätzliche Constraints und Restricted Solutions dienen der weiteren Erleichterung des Anpassungsvorgangs. Bei der Implementierung dieser prototypischen Anwendung zeigte sich, dass vor allem die automatisch generierten Constraints der Visual Position Integrity sehr hilfreich bei Anpassen waren. Weiterhin konnten in vielen Fällen aus Constraints automatisiert sinnvolle Lösungsvorschläge abgeleitet werden.

Die Komponenten werden zur Zeit weiterentwickelt und sollen ebenso wie die FREEVOLVE-Plattform selbst im Rahmen des oben erwähnten BMBF-Projekts zum Einsatz kommen.

Anhang D Definition der Constraints-Funktionen

Betrachtung der Existenz von Komponenten

hasComponent(...) überprüft, ob eine Komponente innerhalb einer Komposition enthalten ist:

$$hasComponent:\ CAS\ \times (N_{\text{components}} \times N_{\text{components}}) \rightarrow \{wahr, falsch\}$$

mit

$$hasComponent(C,Super,Sub) := \begin{cases} wahr, \text{falls } \exists_{(Super,P,A,S,B1,B2,B3)\ \in\ H}\ \exists_{(n1,n2,Param)\ \in\ S}:\ n_1 = Sub \\ falsch\ sonst \end{cases}$$

mit $C=(r,H,C) \in CAS$ und CAS die Menge aller möglichen Komponentensysteme. Da der Namensraum innerhalb eines Komponentensystems eindeutig ist und seine Integrität syntaktisch durch die Definition gegeben ist, reicht eine Überprüfung auf Basis der Namenszuordnungen aus.

Weiterhin wird im folgenden eine Hilfsfunktion *isComponent(...)* benötigt, die wahr zurückliefert, wenn eine Komponenteninstanz in der Komposition enthalten ist:

$$isComponent:\ CAS\ \times N_{Ins\ \tan\ ces} \rightarrow \{wahr, falsch\}$$

definiert für einfache Komponenten(namen)

$$isComponent(C,CompName) := \begin{cases} wahr, \text{falls } \exists_{(CompName,\ P,\ A,\ S,\ B1,B2,\ B3)\ \in\ AComps} \exists_{(Comp,\ Inst,\ I)\ \in\ S} : CompName = Inst \\ falsch\ sonst \end{cases}$$

mit C=(r,AComps, Comps)[31] und analog für abstrakte Komponenten

$$isComponent(C,CompName) := \begin{cases} wahr, \text{falls } \exists_{(CompName,\ P,\ A,\ S,\ B1,B2,\ B3)\ \in\ AComps} \\ falsch\ sonst \end{cases}.$$

Weiterhin wird für die Analyse die Beschreibung einer Komponente benötigt. Als Hilfsfunktion sei deswegen

$$getComponentDescr:\ CAS\ \times N_{Ins\ \tan\ ces} \rightarrow SimpleComponents$$

definiert mit *SimpleComponents* als die Menge aller einfachen Komponenten. Damit liefert *getComponents(...)* ein Tupel *(n,D,P,A)* zurück, das die Komponenteninformationen zum übergebenen Instanznamen enthält.

[31] Diese Definition von AComps bzw. Comps gilt auch in folgenden Definitionen.

Betrachtung der Existenz von Parametern und Verbindungen

Die Betrachtung von Parametern und ihren zugewiesenen W spielen bei der Untersuchung der Eigenschaften einer Komposition eine eine wichtige Rolle. Um einfach auf die Parameter einer Komponente zugreifen zu können wird eine weitere Hilfsfunktion *getParameter(...)* benötigt.

$$getParameter: N_{Components} \times N_{parameters} \rightarrow Values \text{, mit}$$

Values = int ∪ float ∪ String. Values die Menge aller gültigen Parameterwerte entsprechend der Java-Spezifikation [Gosling und McGilton, 1996]. *getParameter(...)* wird für einfache und abstrakte Komponenten getrennt definiert:

$$getParameter(C, CompName, ParamName) := \begin{cases} b, \text{falls } isComponent(C, CompName) \\ \quad \wedge\ getComponentDescr(C, CompName) = (n,D,P,A) \\ \quad \wedge \exists_{(a,b) \in A}: a = ParamName \\ \\ Null\ sonst \end{cases}$$

$$getParameter(C, CompName, ParamName) := \begin{cases} b, \text{falls } isComponent(C, CompName) \\ \quad \wedge \exists_{(CompName,A,S,B1,B2,B3) \in AComps} \\ \quad \wedge\ \exists_{(a,b) \in A}: a = ParamName \\ \\ Null\ sonst \end{cases}$$

isSet(...) überprüft, ob ein Parameter definiert wurde.

$$isSet: CAS \times (N_{instances} \times N_{parameters}) \rightarrow \{wahr, falsch\}$$

isSet(...) wird für einfache und abstrakte Komponenten getrennt definiert:

$$isSet(C,CName,PName) := \begin{cases} wahr, \text{falls } isComponent(C, CName) \\ \quad \wedge\ getParameter(C, CName, PName) \neq NULL \\ falsch\ sonst \end{cases}$$

Weiterhin lassen sich Vergleiche zwischen oder mit festen Werten als Constraints formulieren. Die folgenden Vergleichsoperationen lassen sich in XSemL in der Inorder-Notation schreiben (z.B. „<“ statt „isSmaller“). Weiterhin ist zu unterscheiden, ob ein Parameter mit einem festen Wert oder mit einem weiteren Parameter verglichen werden soll.

Vergleiche zwischen numerischen Parametern und Werten

Parameter-Parameter-Vergleich:

$$isSmaller:\ CAS \times N_{Components} \times N_{parameters} \times N_{Components} \times N_{parameters} \rightarrow \{wahr,falsch\}$$

$$isSmaller(C, CName_1, PName_1, CName_2, PName_2) := \begin{cases} wahr, \text{falls } isComponent(C, CName_1) \\ \quad \wedge isComponent(C, CName_2) \\ \quad \wedge\ isSet(C, CName_1, PName_1) \\ \quad \wedge isSet(C, CName_2, PName_2) \\ \quad \wedge comparable(\\ \quad\quad getParameter(C, CName_1, PName_1), \\ \quad\quad getParameter(C, CName_2, PName_2)) \\ \quad \wedge\ getParameter(C, CName1, PName_1) \\ \quad < getParameter(C, CName2, PName_2) \\ falsch\ sonst \end{cases}$$

Parameter-Wert-Vergleich:

$$isSmaller:\ CAS \times N_{Components} \times N_{parameters} \times Values \rightarrow \{wahr,falsch\}$$

$$isSmaller(C, CName, PName, Value) := \begin{cases} wahr, \text{falls } isComponent(C, CName) \\ \quad \wedge\ isSet(C, CName, PName) \\ \quad \wedge comparable(getParameter(C, CName, \\ \quad\quad PName), Value) \\ \quad \wedge\ getParameter(C, CName, PName) \\ \quad < Value \\ falsch\ sonst \end{cases}$$

In gleicher Weise lassen sich auch die Vergleichsoperatoren ≤, >, ≥, = und ≠ mit den Methoden *isSmallerEquals(...), isGreater(...), isGreaterEquals(...), isEqual(...)* und *isNotEqual(...)* definieren.

Für Vergleiche zwischen Zeichenketten werden Methoden benötigt, die eine Prüfung auf Gleichheit (*equals*), Gleichheit der Anfänge (*startsWith*), Enden (endsWith) oder enthaltene Zeichenketten (*contains*) und die Länge von Zeichenketten (*length*) durchführen können.[32]

$$equals:\ CAS \times N_{Components} \times N_{parameters} \times Values \rightarrow \{wahr,falsch\}$$

[32] Im Folgenden sind lediglich die Methoden-Definitionen der Parameter-Wert-Vergleiche dargestellt. Die Vergleiche zwischen Parametern unterschiedlicher Komponenten ergeben sich analog wie mit *isSmaller(...)* gezeigt.

$$equals(C,CName,PName,Value) := \begin{cases} wahr, \text{falls } isComponent(C,CName) \\ \quad \wedge\ isSet(C,CName,PName) \\ \quad \wedge comparable(getParameter(C,CName, \\ \quad\quad PName),Value) \\ \quad \wedge\ getParameter(C,CName, \\ \quad\quad PName).equals(Value) \\ falsch\ sonst \end{cases}$$

$$startsWith:\ CAS \times N_{Components} \times N_{parameters} \times Values \rightarrow \{wahr,falsch\}$$

$$startsWith(C,CName,PName,Value) := \begin{cases} wahr, \text{falls } isComponent(C,CName) \\ \quad \wedge\ isSet(C,CName,PName) \\ \quad \wedge comparable(getParameter(C,CName, \\ \quad\quad PName),Value) \\ \quad \wedge\ getParameter(C,CName, \\ \quad\quad PName).startsWith(Value) \\ falsch\ sonst \end{cases}$$

$$endsWith:\ CAS \times N_{Components} \times N_{parameters} \times Values \rightarrow \{wahr,falsch\}$$

$$endsWith(C,CName,PName,Value) := \begin{cases} wahr, \text{falls } isComponent(C,CName) \\ \quad \wedge\ isSet(C,CName,PName) \\ \quad \wedge comparable(getParameter(C,CName, \\ \quad\quad PName),Value) \\ \quad \wedge\ getParameter(C,CName, \\ \quad\quad PName).endsWith(Value) \\ falsch\ sonst \end{cases}$$

$$contains:\ CAS \times N_{Components} \times N_{parameters} \times Values \rightarrow \{wahr,falsch\}$$

$$contains(C,CName,PName,Value) := \begin{cases} wahr, \text{falls } isComponent(C,CName) \\ \quad \wedge\ isSet(C,CName,PName) \\ \quad \wedge comparable(getParameter(C,CName, \\ \quad\quad PName),Value) \\ \quad \wedge\ getParameter(C,CName, \\ \quad\quad PName).contains(Value) \\ falsch\ sonst \end{cases}$$

$$length:\ CAS \times N_{Components} \times N_{parameters} \times Values \rightarrow \{wahr,falsch\}$$

$$length(C, CName, PName, Value) := \begin{cases} wahr, \text{falls } isComponent(C, CName) \\ \quad \wedge\ isSet(C, CName, PName) \\ \quad \wedge\ comparable(getParameter(C, CName, \\ \qquad PName), Value) \\ \quad \wedge\ getParameter(C, CName, \\ \qquad PName).length(Value) \\ falsch\ sonst \end{cases}$$

Die Methode *comparable:Values × Values → {wahr, falsch}* liefert wahr zurück, wenn die beiden Werte (*Values*) typgleich sind oder sich sinnvoll ineinander überführen lassen. Dieser Überführung liegt zum einen die Tatsache zugrunde, dass es sich bei den Value-Typen ausschließlich um Ganzzahlen, Gleitkommazahlen und Zeichenketten gemäß der Java-Spezifikation handelt. Da in den hier aufgeführten Definitionen auf Methoden der Sprache Java zurückgegriffen wird, kann die Vergleichbarkeit auf die Java-Spezifikation [Gosling und McGilton, 1996] zurückgeführt werden. Für einfache Typen (int, float) ergibt sie sich unmittelbar aus der Spezifikation der Vergleichsoperatoren (<, > etc.). Für die Zeichenkettenoperationen lässt sie sich durch die Betrachtung der Methodensignaturen der jeweils verwendeten Objektmethoden auslesen.

Verbindungen zwischen Komponenten

Neben den Parametern werden Komposition in erster Linie durch die Verbindungen zwischen den enthaltenen Komponenten beschrieben. Zum Testen der Verbindungen stehen verschiedene Methoden zur Verfügung. *hasPort(...)* prüft auf die Existenz eines Ports in einer Komponente:

$$hasPort:\ CAS \times N_{Components} \times N_{ports} \rightarrow \{wahr, falsch\}$$

ist für einfache Komponenten wie folgt definiert:

$$hasPort(C, CName, P_0Name) := \begin{cases} wahr, \text{falls } isComponent(C, CName)\ mit \\ \quad getComponentDesrc(C, CName) = (n, D, P, A) \\ \quad \wedge \exists (PName, PType, P_{Dir}) \in P : P_0Name = PName, \\ falsch\ sonst \end{cases}$$

für komplexe Komponenten entsprechend:

$$hasPort(C, CName, P_0Name) := \begin{cases} wahr, \text{falls } isComponent(C, CName) \\ \quad \exists (CName, P, A, S, B_1, B_2, B_3) \in AComps \\ \quad \exists (PName, PType, P_{Dir}) \in P : P_0Name = PName, \\ falsch\ sonst \end{cases}$$

Weiterhin lässt sich mit *isBound(...)* überprüfen, ob ein Port verbunden ist. Entsprechend den in Kapitel 7.2 eingeführten Definitionen für Komponenten, kann dies nur innerhalb von anderen (komplexen) Komponenten beschrieben werden. Die Methode wird unterschiedlich

definiert je nachdem, ob es sich bei der Komponente, die untersucht wird, um eine einfache oder komplexe handelt. Zur Prüfung, ob eine Komponente mit der sie umgebenden komplexen Komponente verbunden ist, sei

$$isBoundAbstract:\ CAS \times N_{Components} \times N_{ports} \rightarrow \{wahr,falsch\}$$

definiert wie folgt:

$$isBoundAbstract(C,CName,P_0Name) := \begin{cases} wahr, \text{falls } \exists(n,P,A,S,B_1,B_2,B_3) \in AComps \\ \quad mit\ C = (r, AComps, Comps) \\ \quad \exists(CompType, CName, Inf) \in S \\ \quad \exists(Port, CN, PN) \in B_2 : \\ \quad CN = CName \wedge PN = P_0Name \\ falsch\ sonst \end{cases}$$

Soll geprüft werden, ob eine Komponente auf der gleichen hierarchischen Ebene mit einer anderen Komponente verbunden ist, sei

$$isBound:\ CAS \times N_{Components} \times N_{ports} \rightarrow \{wahr,falsch\}$$

mit

$$isBound(C,CName,PName) := \begin{cases} wahr, \text{falls } \exists(n,P,A,S,B_1,B_2,B_3) \in AComps \\ \quad mit\ C = (r, AComps, Comps) \\ \quad \exists(CompType, CName, Inf) \in S \\ \quad \exists(N_1, PN_2, CN_2, PN_2) \in B_1 : \\ \quad CN_1 = CName \wedge PN_1 = PName \\ \quad \wedge CN_2 = CName \wedge PN_2 = PName \\ falsch\ sonst \end{cases}$$

Weiterhin kann untersucht werden, ob zwei Komponenten über bestimmte Ports miteinander verbunden sind. Dazu dient folgende Methode:

$$isBondTo:\ CAS \times (N_{Component} \times N_{ports}) \times (N_{Components} \times N_{ports}) \rightarrow \{wahr,falsch\}$$

mit

$$isBoundTo(C,CN_1,P_0N,CN_2,P_0N_2) := \begin{cases} wahr, \text{falls } \exists(n,P,A,S,B_1,B_2,B_3) \in AComps \\ \quad mit\ C = (r, AComps, Comps) \\ \quad \exists(CompType_1, CN_1, Inf_1) \in S \\ \quad \exists(CompTyp_2, CN_2, Inf_2) \in S \\ \quad \exists(N, P, N_2, P_2) \in B_1 : \\ \quad N_1 = CN_1 \wedge N_2 = PN_2 \wedge \\ \quad P_1 = PN_1 \wedge P_2 = PN_2 \\ falsch\ sonst \end{cases}$$

Struktur innerhalb der Komposition (gebundene Parameter und Ports)

Schließlich muss auch die Struktur einer hierarchischen Komposition analysierbar sein. Sie wird definiert durch ineinander enthaltenen Komponenten (*hasComponents(...)*, s.o.) und Ports und Parametern, die an die nächsthöhere Hierarchieebene gebunden werden. Diese Regelfunktionen werden im Folgenden definiert:

$$isBoundUpParam: CAS \times N_{Components} \times N_{parameters} \times N_{Components} \rightarrow \{wahr,falsch\}$$

$$isBoundUpParam(C,CName,PName,SuperName) := \begin{cases} wahr, \text{falls } isComponent(C,CName) \\ \quad \wedge isComponent(C,SuperName) \\ \quad \wedge getParameter(C,CName,PName) \neq Null \\ \quad \wedge \exists(SuperName,P,A,S,B_1,B_2,B_3) \in AComps \\ \quad mit\ C = (r, AComps, Comps) \\ \quad \exists(PAName, I_CName, PcName) \in B_3 : \\ \quad\quad CName = I_CName \\ \quad \wedge PName = P_CName \\ falsch\ sonst \end{cases}$$

Für zwischen den hierarchischen Ebenen verbundenen Ports gilt dann entsprechend:

$$isBoundUpPort: CAS \times N_{Components} \times N_{ports} \times N_{Components} \rightarrow \{wahr,falsch\}$$

$$isBoundUpPort(C,CName,P_0Name,SuperName) := \begin{cases} wahr, \text{falls } isComponent(C,CName) \\ \quad \wedge isComponent(C,SuperName) \\ \quad \wedge hasPort(C,CName,P_0Name) \\ \quad \wedge \exists(SuperName,P,A,S,B_1,B_2,B_3) \in AComps \\ \quad (C = (r, AComps, C)) \\ \quad \wedge \exists(P_0AName, I_CName, P_0cName) \in B_2 : \\ \quad\quad CName = I_CName \\ \quad \wedge PName = P_{0C}Name \\ falsch\ sonst \end{cases}$$

Anhang E Grundlagen von Petri-Netzen

Petri-Netze wurden 1962 von Petri [Petri, 1979] eingeführt. Sie dienen als Modell zur Beschreibung und Analyse von (nebenläufigen) und unter Umständen nicht-deterministischen Prozessen.

Definition Petri-Netz:

Ein Tupel $PN = \{S, T, F, V, K, M, m_0\}$ sei ein *Petri-Netz* mit

$S \neq \emptyset$	Menge der Stellen
T	Menge der Transitionen
$F: (S \times T) \cup (T \times S) \rightarrow \{0,1\}$	Flussrelation
$W: F \rightarrow N \setminus \{0\}$	Kantenbewertung
$K: F \rightarrow N \setminus \{0\}$	Kapazitäten
$M: P \rightarrow N$	Aktuelle Markierung
$m_0 \in M$	Anfangsmarkierung,

wobei zusätzlich gilt: $S \cap T = \emptyset$ und $0 \leq M(s) \leq K(s)$.

Damit ist ein Petri-Netz ein gerichteter Graph, der aus zwei disjunkten Knotenmengen besteht. Die eine Knotenmenge setzt sich aus der Menge der Transitionen zusammen. Transitionen beschreiben Ereignisse, Aktionen und Operationen. Die zweite Knotenmenge – die Stellen – dient als Puffer oder als Zwischenablage von Daten. Transitionen und Stellen sind – wie in der Flussrelation beschrieben – durch Kanten miteinander verbunden. Weiterhin gilt:

- Die Stelle s ist *Eingangsstelle* der Transition *t*, und die Transition *t* ist die A*usgangstransition* der Stelle *s*, genau dann wenn $(s,t) \in F$ gilt.
- Die Transition *t* ist die *Eingangstransition* der Stelle s, und die Stelle s ist die *Ausgangsstelle* der Transition *t*, genau dann wenn $(t,s) \in F$ gilt.
- Der *Vorbereich* einer Stelle oder Transition $x \in S \cup T$ sei definiert als $\bullet x = \{y \mid (y,x) \in F\}$. Falls $x \in S$, dann *ist* $y \in T$ und umgekehrt. Dies geht direkt aus der Definition für F hervor.
- der *Nachbereich* einer Stelle oder Transition $x \in S \cup T$ entsprechend als $x \bullet = \{y \mid (x,y) \in F\}$.

Um einen Prozess festzulegen, werden die Plätze markiert. Die Markierung *m* eines Platzes entspricht der Abbildung $m: P \rightarrow \mathbf{N}$, für jedes $m \in M$. Die Anzahl der Marken in einer Stelle spiegelt den Erfüllungsgrad einer Bedingung wieder, und hat somit Einfluss auf die Arbeitsfähigkeit der angrenzenden (abhängigen) Transitionen. Die Funktion der Platzkapazität beschreibt die maximale Anzahl an Marken, die ein Platz aufnehmen kann.

Schaltregeln für Transitionen

Für das Schalten der Transitionen gelten folgende Regeln:

- Eine Transition *t* kann nur schalten, wenn jede Eingangsstelle s von *t* mindestens *W(s,t)* Marke enthält.
- Beim Schalten einer Transition wird jeder Eingangsstelle *s W(s,t)* Marken entfernt, und jeder Ausgangsstelle *r W(t,r)* Marke hinzugefügt.

Dabei darf keine Stelle eine nach einem Schaltvorgang negative Anzahl von Markierungen haben. Ebenso darf die durch K vorgegebene maximale Kapazität in den Ausgangsstellen nicht überschritten werden. Das Schalten einer Transition *t* ändert die Markierung *m* in eine neue Markierung m' gemäß folgender Beziehung:

$$m'(s) = \begin{cases} m(s) - W(s,t), wenn (s,t) \in S \times T \\ m(s) + W(t,s), wenn (t,s) \in T \times S \\ m(s), sonst \end{cases}$$

Erreichbarkeitsproblem

Beim Hintereinanderschalten der Transitionen entsteht eine *Markierungsfolge*. Die Schaltvorgänge der Transitionen $t_1 \rightarrow ... \rightarrow t_n$ heißen dann eine *Schaltfolge*, wenn eine Markierung von m_1 nach m_{n+1} überführt wird. Durch diese Schaltfolge entsteht folgendes Bild:

$$m_1 \xrightarrow{t_1} m_2 \xrightarrow{t_2} ... \xrightarrow{t_{n-1}} m_n \xrightarrow{t_n} m_{n+1}$$

Aus der Schaltfolge der Transitionen impliziert sich das *Erreichbarkeitsproblem*. Es lässt sich als Frage formulieren, ob sich ein gegebenes Petri-Netz PN mit einer Anfangsmarkierung m_0 in eine Markierung m* überführen lässt. Die Menge der Markierungen, die von einer Ausgangsmarkierung m_i erreichbar ist, wird als *Erreichbarkeitsmenge* bezeichnet..

Unterschiedliche Klassen von Petri-Netzen

Gefärbte Petri-Netze gehören zur Gruppe der IM-Netze (Systeme mit individuellen Marken, auch höhere Netze). Die Erweiterung besteht in erster Linie darin, dass die Marken nicht mehr als gleichartig angesehen werden. Stattdessen sind sie jeweils als wohldefinierte formale Objekte (z. B. Zahlen oder Farben) zu sehen. Weiterhin können zwei Marken auch den gleichen Wert haben. Damit ist die Markierung einer Stelle beschrieben durch eine Multimenge (z.B. {rot, grün, blau, blau}).

Aus diesem grundlegenden Ansatz ergibt sich eine Erweiterung der Schaltbedingungen. Während bisher nur quantitative Aspekte die Schaltvorgänge beeinflussten, kommen nun qualitative hinzu. Gefärbte Petri-Netze eignen sich zur Modellierung der operativen Betriebsführung. In diesem Falle verkörpern die Transitionen die Prozesselemente.[33] Die Arbeitsfortschrittbedingungen, die Auslöser für Prozesse sind, werden durch Stellen repräsentiert. Ein gefärbtes Petri-Netz ist die Abbildung mehrerer allgemeingültiger Petri-Netze mit unterschiedlicher Struktur in einem Netz. Mit Hilfe der unterschiedlichen Farben der Marken werden die lokalen Strukturen verdeutlicht.

[33] Petri-Netze werden häufig zur Modellierung von Abläufen in der Fertigung eingesetzt. Dabei werden Maschinen oder Teilaggregate als Prozesselemente behandelt. Hier werden Komponenten betrachtet, die aus der Sicht der Petri-Netz-Modellierung ähnliche Eigenschaften aufweisen.

Zusammenfassung

Anpassbarkeit ist für moderne Applikationen eine wichtige Eigenschaft. Üblicherweise werden Anpassungen von den Benutzern als zusätzliche Aufgabe im Rahmen ihrer normalen Arbeit durchgeführt. Um diese Aktivitäten zu fördern, müssen Anpassungsumgebungen leicht zu erlernen sein und sollten den Anpassungsakt selbst technisch unterstützen. Speziell auf Basis komponentenbasierter Architekturen lassen sich hochgradig anpassbare Anwendungen zu entwickeln, der Anpassungsprozess ist dabei auch aufgrund der Nähe dieses Ansatzes zu anderen Konstruktionstecniken für Endbenutzer leicht zugänglich. Komponentenbasierte Architekturen lassen sich mit einer Sprache anpassen, die verschiedene Operationen zur Verfügung stellt. Sie orientieren sich an den grundlegenden Eigenschaften (Applikationen werden gebildet durch Parameterisieren und Verbinden einzelner ausgewählter Komponenten) der Komponenten-Architekturen.

Nichtsdestotrotz zeigen verschiedene Studien, dass durch Integration verschiedener Unterstützungstechniken sowohl der erste Zugang zur Software als auch die weitere Nutzung deutlich erleichtert werden kann. Die meisten solcher Konzepte zielen dabei darauf ab, durch eigenes Erfahren, Explorieren oder durch kooperative Prozesse das Verständnis für die Software zu erhöhen und damit Anpassungen zu erleichtern.

Im Rahmen dieser Arbeit wurde eine technische Unterstützung der Anpassungsaktivitäten vorgestellt, die den explorativen Zugang zur Anpassung unterstützen soll. Das Ziel ist es, technisch zu überprüfen, inwieweit eine Anpassung sinnvoll im Sinne der Erwartungskonformität ist oder auch ob die resultierende Applikation lauffähig ist. Dabei stand die Unterstützung von unerfahreneren Anwendern ohne Programmierkenntnisse im Vordergrund der Betrachtung. Eine Integritätsprüfung, die die Komposition auswertet, dient dazu, semantische Fehler aufzudecken und auf diese Weise sowohl den Anpassungsprozess zu unterstützen als auch das Verständnis für die Komposition ebenso wie auch für die Einzelkomponenten und die Anpassungsoperationen zu fördern.

Im Rahmen dieser Arbeit werden zwei grundlegend unterschiedliche Integritätskonzepte vorgestellt: Die Constraint Integrity einerseits und die darauf aufbauende Restricted Solution Integrity und die Event Flow Integrity andererseits. Constraints beziehen sich jeweils lokal auf die Eigenschaften einer Komponente. Werden Constraints nicht eingehalten, so wird dies als Verletzung der Integrität gewertet. Solutions bieten in diesem Zusammenhang dann weiterführend Vorschläge zur Korrektur an.

Die Event Flow Integrity beschäftigt sich mit der Analyse der Nachrichtenflüsse innerhalb einer Komposition. Ziel dieser Prüfung ist sicherzustellen, dass innerhalb der Komposition erzeugte Nachrichten auch konsumiert werden. Dem liegt die Annahme zugrunde, dass wichtige Nachrichten nicht einfach ignoriert werden dürfen. Ansonsten gilt eine solche Komposition als nicht integer. Integritätsbedingungen werden einzelnen Komponenten oder Gruppen von ihnen in Form von externen Beschreibungen mitgeliefert.

Die Korrektheit der beiden Integritätskonzepte in Bezug auf Berechenbarkeit ebenso wie auf Sinnhaftigkeit in Bezug zur Aufgabenstellung werden ausführlich diskutiert. In ersterem Fall diente das Grundlagenwissen aus der theoretischen Informatik zur Überprüfung der Konzepte. Die Implementierung der Integritätsprüfung setzt auf die schon vorhandene FREEVOLVE-Plattform auf, in der komponentenbasierte Applikationen genutzt und angepasst werden können. Eine spezielle Anpassungsumgebung (TailorClient) stellt Benutzern sowohl die Anpassungsoperationen als auch die Integritätsprüfung zur Verfügung. Eine mehrstufige qualitative Evaluation zeigt, dass die Integritätsprüfung dazu beiträgt, Fehler beim Anpassen zu verringern. Die Evaluierung hat so zeigen können, dass eine Integritätsprüfung ein sinnvolle Erweiterung einer Anpassungsumgebung für komponentenbasierte Architekturen darstellt.

Erklärung

An Eides Statt versichere ich, dass ich die Arbeit

„Interaktive Integritätsprüfung für komponentenbasierte Architekturen – Technische Unterstützung für Endanwender beim Anpassen komponentenbasierter Software“

unter der Leitung von Herrn Professor Cremers und Herrn Professor Wulf als Korreferent selbst und ohne jede unerlaubte Hilfe angefertigt habe, dass diese oder eine ähnliche Arbeit noch keiner anderen Stelle zur Prüfung vorgelegen hat und dass sie weder vollständig noch auszugsweise veröffentlicht worden ist.