Visuelles Komponieren und Testen von Komponenten am Beispiel von Agenten im elektronischen Handel

Vom Fachbereich Informatik
der Technischen Universität Darmstadt genehmigte

Dissertation

zum Erlangen des akademischen Grades des
Doktor-Ingenieurs (Dr.–Ing.)

vorgelegt von
Dipl.–Inform. Ludger Martin
geboren in Mainz

Referent und Korreferentin der Arbeit:
Prof. Dr.–Ing. H.–J. Hoffmann
Prof. Dr.–Ing. M. Mezini

Tag des Einreichens: 24. März 2003
Tag der Prüfung: 16. Mai 2003

D17

Zusammenfassung

Die Software-Entwicklung mit Komponenten wird im Laufe der Zeit immer beliebter. Die Komponententechnologie verspricht eine effiziente Software-Entwicklung durch große Zuverlässigkeit und gute Wiederverwendbarkeit. Es fehlt aber noch an Werkzeugen, die die einzelnen Entwicklungsschritte der Komponententechnologie unterstützen.

In dieser Arbeit sollen deswegen Werkzeuge zur Unterstützung der komponentenbasierten Software-Entwicklung untersucht werden. Dazu werden Kriterien aufgestellt, die beim Entwickeln von Komponenten, beim Zusammenbauen von Anwendungen und beim Testen bzw. beim Verfolgen des Programmablaufs beachtet werden müssen. Diese werden mit Hilfe von anderen Arbeiten begründet. Der Aspekt einer visuellen Unterstützung und Programmierung wird dabei mit berücksichtigt.

Um zu zeigen, daß es möglich ist, anhand dieser Kriterien eine Entwicklungsumgebung zu konstruieren, wird HotAgent vorgestellt. HotAgent ist ein Komponentenentwurfsrahmen, um Agenten für den elektronischen Handel zu konstruieren. HotAgent bietet Werkzeuge zum Entwickeln von Komponenten, zum Zusammenbauen von Anwendungen, zum Testen von Komponenten und zum Verfolgen des Programmablaufs an.

HotAgent hält fast alle geforderten Kriterien ein und bietet so die Möglichkeit einer komfortablen und qualitativ hochwertigen Software-Entwicklung mit Komponenten. Die Werkzeuge stellen alle eine graphische Unterstützung neben der visuellen und textuellen Programmierung bereit. Außerdem wird auf eine gute Integration der einzelnen Werkzeuge geachtet.

Abstract

In recent years software development with components became more popular. Component technology promises efficient software development through reliability and re-usability. A drawback is that there are no development environments which support all development steps for component technology.

Therefore, tools to support component development will be investigated in this dissertation. Criteria to develop components, to compose applications, to test, and to analyze will be listed. These criteria are established using papers of others. The fact of visual support and programming is always considered.

To show the possibility of developing component based development environment according to these criteria HotAgent will be introduced. HotAgent is a

component framework to construct agents for electronic commerce. HotAgent offers tools to develop components, to compose applications, to test components and to analyze the runtime behavior of an application.

HotAgent complies with almost all discussed criteria and offers a comfortable and qualitative software development with components. The tools appear in a consistent user interface and allow visual as well as textual programming. All tools are well integrated with each other.

Vorwort

Doktorarbeiten entstehen über einen langen Zeitraum, in dem der Austausch mit anderen Leuten sehr wichtig ist. Die in der Zeit geführten Gespräche unterstützten meine Arbeit sehr fruchtbar. Es ist nicht möglich, alle Personen aufzuzählen, die beim Gelingen der Arbeit behilflich waren. Nur einige möchte ich hier erwähnen.

An erster Stelle möchte ich meinem Doktorvater, Herrn Prof. Dr.–Ing. H.–J. Hoffmann, danken, der jederzeit bereit war, mir richtungweisende Unterstützung zu bieten. Für die Betreuung und die Übernahme des Korreferats möchte ich mich auch herzlich bei Frau Prof. Dr.–Ing. M. Mezini bedanken. Das gleiche gilt für Herrn Prof. A. Buchmann, Ph.D., der im Auftrag der DFG für die Finanzierung meiner Arbeit sorgte.

Nicht zu vergessen sind natürlich meine Kollegen Dr. Elke Siemon, Ludger Fiege und Dr. Gero Mühl, die mit mir gerne über meine Arbeit diskutierten. Einen großen Beitrag leisteten die Studenten Anke Giesl, Thorsten Clausius, Torsten Scheidler, Michael Renker, Swen Haupt, Tekla Kiss und Alexander Grahl, die Teilthemen in ihrer Diplomarbeit, Studienarbeit oder einem Praktikum bearbeiteten.

Dank gilt natürlich auch meinen Eltern, Geschwistern und Freunden, die einerseits viel Zeit in das Korrekturlesen investierten, aber auch dafür sorgten, daß meine Freizeit nicht vernachlässigt wurde.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

In den letzten Jahren wurde die Software-Entwicklung mit Komponenten immer moderner. Sie versprechen das effiziente Konstruieren von Anwendungen aus Komponenten. Darüber hinaus wird auf die Wiederverwendung von Komponenten großer Wert gelegt. Die mit Komponenten erstellten Anwendungen sind nicht nur schneller konstruiert, sondern auch verläßlicher und wartbarer.

Durch Anwenden der Komponententechnik wird die visuelle Programmierung wieder interessant. Lange Zeit konnte diese nicht richtig Fuß fassen, da so erstellte Programme oft zu groß und unübersichtlich wurden. Die Komponententechnik ist ideal für die visuelle Programmierung. Komponenten haben eine abgegrenzte Funktionalität und eignen sich dadurch gut zur Programmierung im Großen, was wiederum die visuelle Programmierung begünstigt. Mit der Komponententechnik lassen sich auch Hierarchien aufbauen, so daß sich ein Programm in einzelne Teilprobleme gliedern läßt.

Die Komponententechnik verspricht hohe Qualität. Das bringt mit sich, daß gute Techniken vorhanden sein müssen, um die einwandfreie Funktionsweise sicherzustellen. Dazu gehört u.a., daß die Komponenten immer wieder getestet werden müssen. Fertige Programme müssen auf ihre richtige Funktionsweise untersucht werden können.

Die Frage ist, wie kann man die oben genannten Forderungen an Qualität und Effizienz erfüllen. Dies soll in dieser Arbeit geklärt werden. Dazu wird nacheinander ermittelt, welche Kriterien für das Verwenden von Komponenten, für das visuelle Programmieren mit Komponenten, für das Testen von Komponenten und für das Verfolgen von Programmabläufen existieren.

Anhand dieser Kriterien werden Werkzeuge erstellt, die für die einzelnen Entwicklungsabläufe notwendig sind, angefangen von dem Erstellen von Komponenten, über das Zusammensetzen von Anwendungen bis hin zu der Qualitätssicherung.

Diese werden als HotAgent Komponenten-Entwicklungsumgebung zusammengefaßt.

Als Anwendungsgebiet von HotAgent wird das Erstellen von Agenten für den elektronischen Handel in den Vordergrund gestellt. Diese Agenten sollen Routinearbeiten übernehmen, indem sie u.a. elektronische Briefe analysieren, bearbeiten und beantworten. Die Komponententechnik erlaubt, daß für diese Aufgabe speziell vorgefertigte Komponenten bereitgestellt werden, die bei der Entwicklung der Agenten wiederverwendet werden können. Die Komponententechnik ist aber dennoch so flexibel, daß jede Firma ihre Agenten nach ihren eigenen Bedürfnissen entwickeln kann. Es wird ein Baukastensystem bereitgestellt, aus dem verschiedene Arten von Agenten für den elektronischen Handel konstruiert werden können.

Diese Arbeit läßt sich wie folgt in einzelne Teilgebiete gliedern. In Kapitel 2 werden Grundlagen der Komponententechnologie erklärt. Hier werden Kriterien für Komponenten-Modelle gefunden, anhand deren einige der wichtigsten Komponentensysteme vorgestellt werden. HotAgent verwendet sein eigenes Komponentensystem, das daraufhin erläutert wird. Kapitel 3 beschäftigt sich mit der visuellen Programmierung. Auch hier werden wieder Kriterien aufgestellt, die in einer guten Entwicklungsumgebung erfüllt werden müssen. Kapitel 4 betrachtet die Qualitätssicherung. Es werden dabei die Teilgebiete des Komponententests und der Programmablaufverfolgung untersucht. Für diese werden wieder Kriterien aufgestellt. Kapitel 5 bespricht, wie Agenten für den elektronischen Handel mit Hilfe von Komponenten erstellt werden können. Dies wird anhand eines Beispiels gezeigt, das im Laufe dieses Kapitels genauer betrachtet wird. Kapitel 6 widmet sich den einzelnen Teilen der HotAgent Entwicklungsumgebung. Da Komponenten Qualität in der Funktionalität versprechen, ist es wichtig, daß die verwendete Entwicklungsumgebung auch qualitativ hochwertig ist. Dazu wird in Kapitel 7 die Gebrauchstauglichkeit von HotAgent untersucht. Kapitel 8 stellt andere Entwicklungsumgebungen vor, die zum Teil das Entwickeln mit Komponenten, visuelles Programmieren, Testen und Analysieren unterstützen. Die Arbeit schließt mit einer Zusammenfassung und einem Ausblick in Kapitel 9 ab.

In dieser Arbeit werden oft Quellen indirekt zitiert. Zur besseren Unterscheidung zum normalen Text werden diese Zitate im Konjunktiv wiedergegeben.

Kapitel 2

Komponentenmodelle

Die Komponententechnologie gewinnt immer mehr an Bedeutung bei der Softwareentwicklung in den verschiedensten Anwendungsbereichen. Johnson [Joh97] erklärt, daß die Programmierung mit Komponenten sehr effizient sei. Dies werde erreicht, da Wiederverwendung eine große Rolle spiele und die Erstellung und Wartung von Komponentenprogrammen einfach seien.

Völter [Völ02] fügt hinzu, daß die Komponententechnologie sehr erfolgreich sei, da sie in ähnlichen Umfeldern eingesetzt werde und dadurch eine große Wiederverwendung ermögliche. Es bestehe auch eine Trennung zwischen technischen und funktionalen Aspekten, d.h. das Komponentenmodell sei für die Kommunikation (Transaktionen, Sicherheit, usw.) zuständig und eine Komponente müsse nur den funktionalen, anwendungsbezogenen Teil bereitstellen.

Mezini und Haupt [MH01a] schreiben, in großen Projekten sei die Implementierung von Teilfunktionalität sehr zeitaufwendig. Vorteilhaft sei es, diese in Komponenten zu gliedern. Dadurch entständen Standard- und Spezialkomponenten, die leicht in Anwendungen integriert werden können.

Zu den oben genannten Punkten ist hinzuzufügen, daß ein Komponentenmodell relativ einfach sein muß, so daß leicht und preiswert neue Komponenten entwickelt werden können. Komponenten sind in der Regel unabhängig voneinander, so daß auch Komponenten von verschiedenen Anbietern in einer Anwendung verwendet werden können.

In dem folgenden Abschnitt wird erläutert, was eine Komponente genau ist. In Abschnitt 2.2 folgt eine Beschreibung, welche Kriterien ein Komponentenmodell bzw. deren Entwicklungsumgebung erfüllen muß. Danach werden in Abschnitt 2.3 bereits etablierte Komponentenmodelle vorgestellt. Abschnitt 2.4 beschreibt das von HotAgent benutzte Komponentenmodell. Dieses wird in Abschnitt 2.5 mit einem ereignisbasierten System verglichen.

2.1 Allgemein

Was eine Komponente ist, ist noch nicht ganz einheiltich definiert. Es gibt mehrere Definitionen, die in gewissen Punkten ähnlich sind, die aber auch Unterschiede aufweisen. In dieser Arbeit werden folgende Definitionen benutzt, um für HotAgent ein Komponentenmodell zu erstellen.

Szyperski [Szy98] definiert:

> A *component* is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Councill und Heineman [CH01] definieren eine Komponente als:

> A *software component* is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.

Zusätzlich zu einer Komponente wird ein Komponentenmodell benötigt. Councill und Heineman [CH01] definieren:

> A *component model* defines specific interaction and composition standards. A component model implementation is the dedicated set of executable software elements required to support the execution of components that conform to the model.

Eine offenere Definition von Meyer [Mey99b] ist:

> A *software component* is a program element. The element may be used by other program elements (clients). The clients and their authors do not need to be known to the element's authors.

Das Hauptmerkmal einer Komponente ist die einheitliche Schnittstelle. Diese garantiert, daß mehrere Komponenten miteinander kombiniert werden können, ohne, daß sie modifiziert werden müssen. Ebenfalls läßt sich eine Komponente auch problemlos durch eine andere, unabhängig von ihrer Funktion, austauschen. In einer Komponente wird eine spezielle Funktionalität gekapselt und kann dadurch auch unabhängig entwickelt werden. Mezini und Haupt [MH01a] bestätigen, eine

Komponente sei gekapselt und unabhängig voneinander entwickelt. Eine Komponente müsse eine Kommunikationsfähigkeit bieten.

Seiter *et al.* [SML99] erwähnen, die Unabhängigkeit von Komponenten fördere die Wiederverwendbarkeit. Wiederverwendung ist in der Komponententechnologie ein wichtiges Thema. Wird eine Stück Software mehr als zweieinhalbmal angewandt, so ist der Aufwand kleiner, um es wiederverwendbar zu machen, als wenn man das Programmstück jedesmal speziell neu schreibt. Die Wiederverwendbarkeit werde dadurch unterstützt, daß eine Komponente eine *schwarze Kiste* (engl. black box) sei, wie Weinreich und Sametinger [WS01] schreiben. Dadurch, daß nicht in eine Komponente hineingeschaut werden könne, stehen dennoch Mechanismen zur Verfügung, um sie für den jeweiligen Kontext zu konfigurieren.

Nach Cox und Song [CS01] definiere jedes Komponentenmodell unabhängige schwarze Kisten (*black-boxen*). Diese seien durch einheitlich definierte Schnittstellen ausgezeichnet, die zum Empfangen und Verschicken von Botschaften dienen.

Eine Komponente enthält alle Teile, die zur Ausführung notwendig sind. Zusätzlich müssen Beschreibungen der Funktionalität und der Schnittstelle mitgegeben werden. Die Komponententechnologie setzt keine Objektorientierung voraus. Es ist auch keine Möglichkeit gegeben, Vererbung bei Komponenten einzusetzen. Komponenten werden in der Regel konfiguriert und danach im Sinne von Parnas [Par76] aufgerufen (A hat seine Aufgabe erfüllt, wenn sie B richtig aufgerufen hat).

Nach Mezini et al. [ML98, SML99] seien Komponenten generisch und zusammensetzbar. Komponenten lägen meistens in binärer Form vor und könnten so zu Anwendungen zusammengesetzt werden. Unter Zuhilfenahme von Komponenten könnten Anwendungen erstellt werden, die ähnliche Aufgabenfelder abdecken.

2.2 Kriterien für Komponenten-Modelle und Entwicklungs-Umgebungen

Im vorangehenden Abschnitt wurde geklärt, was eine Komponente ist. Nun ist zu fragen, wie eine Komponenten-Entwicklungsumgebung aussehen muß. Dazu werden Kriterien aufgestellt, die für eine Komponenten-Entwicklungsumgebung hilfreich sind. Zu jedem Kriterium wird ein Beispiel angegeben, welche Folgen das Einhalten bzw. Nicht-Einhalten hat.

Zur Absicherung der Richtigkeit der Kriterien werden Literaturstellen zitiert, die die einzelnen Kriterien untermauern sollen. Die aufgeführten Kriterien können nicht als die einzig gültigen angesehen werden. Meines Erachtens sind das aber die wichtigsten.

Lüer und Rosenblum [LR00, LR01] stellen dazu sieben Anforderungen für komponentenbasierte Entwicklungsumgebungen auf:

Modulares Design (K1): Werde eine Komponente (nach [LR00, LR01]) aus mehreren Teilen zusammengesetzt, so müsse eindeutig zwischen privaten und öffentlichen Teilen unterschieden werden. Meyer [Mey90] definiert fünf Prinzipien für saubere Modularität. Eines ist das *Geheimnisprinzip*. Alle Teile seien modulintern, wenn sie nicht ausdrücklich als öffentlich definiert seien. So könne u.a. die Implementierung geändert werden, ohne daß es nach außen hin sichtbar werde. Er fordert auch *explizite Schnittstellen*. Jede Außenverbindung müsse deutlich gekennzeichnet werden, damit nachvollzogen werden könne, wie eine Komponente beeinflußt wird.

+ Durch ein *modulares Design* wird erreicht, daß für ähnliche Komponenten gleiche Schnittstellen geschaffen werden können und so z.B. leicht gegeneinander ausgetauscht werden können.

- Besteht ein Durcheinander von öffentlichen Teilen einer Komponente, leidet z.B. die Wartbarkeit. Es kann nicht nachvollzogen werden, welche Teile, z.B. nach dem Finden eines Mangels, gefahrenlos geändert werden dürfen oder nicht.

Komponenten-Selbstbeschreibung (K2): Eine Komponente (nach [LR00, LR01]) solle die Informationen, die für ihre Verwendung notwendig sind, selber zur Verfügung stellen. Weinreich und Sametinger [WS01] ergänzen, daß dazu auch Informationen über die Beziehungen zwischen Komponenten und über die Schnittstelle einer Komponente gehören. Knuth [Knu92] fordert das *literate programming*, bei der die Programmiersprache auch gleich Dokumentationssprache ist. Dies soll u.a. helfen, Abweichungen zwischen Quelltext und Dokumentation zu verhindern.

+ Durch die *Selbstbeschreibung* wird dem Programmierer die Suche nach der richtigen Dokumentation extrem erleichtert. Jede Komponente beinhaltet die zu ihr passende Dokumentation.

- Wird Quelltext ohne Dokumentation ausgeliefert, ist es oft schwer, die richtige Version einer Dokumentation zu finden. Glaubt man sie zu haben, besteht immer noch die Frage, ob sie auch aktuell ist, da oft das Nachtragen der Änderungen in der Dokumentation vergessen wird.

Globaler Namensraum (K3): Um eine Komponente (nach [LR00, LR01]) für eine Anwendung genau zu spezifizieren, sei es notwendig, daß es einen eindeutigen, globalen Namensraum gebe, damit keine Verwechslungen auftreten.

+ Existiert ein *globaler Namensraum*, so gibt es (nach [LR00, LR01]) für jede Komponente einen eindeutigen Namen. Somit kann sie eindeutig identifiziert werden.

- Ist keine Namensregelung vorhanden, so kann es vorkommen, daß zwei Komponenten den selben Namen tragen. Es ist unklar, welche jeweils vom Anwender gemeint ist.

Zweigeteilter Entwicklungsprozeß (K4): Der Entwicklungsprozeß lasse sich (nach [LR00, LR01]) in zwei Teile aufteilen: Das Entwickeln von Komponenten und das Erstellen der Anwendung aus Komponenten.

+ Durch das Zweiteilen läßt sich eine klare Trennung zwischen Komponenten und einer Anwendung machen. Dadurch kann leicht eine Komponente durch eine andere ersetzt werden, ohne die Anwendung zu ändern.

- Besteht keine klare Trennung, ist es schwer, einzelne Teile eines Programms wieder zu verwenden. Es fehlen die Grenzen einer Komponente. Es muß sicher gestellt werden, daß alle benötigten Teile extrahiert wurden.

Anwendung zusammenbauen (K5): Eine Anwendung werde (nach [LR00, LR01]) durch Anpassen und Verbinden von Komponenten erstellt.

+ Dieser Mechanismus stellt sicher, daß nur minimaler Aufwand notwendig ist, um eine Anwendung aus Komponenten zu erstellen.

- Ist zum Erstellen einer Anwendung mehr notwendig, ist es oft schwer herauszufinden, an welchen Stellen im Quelltext Änderungen bzw. Anpassungen vorgenommen werden müssen.

Verschiedene Ansichten (K6): Beim Entwickeln von Komponenten und Anwendungen werden (nach [LR00, LR01]) verschiedene Ansichten benötigt, dazu gehören z.B. Entwicklungs- und Kompositionsansichten.

+ Dieses Kriterium unterstützt das klare Trennen zwischen Komponenten- und Anwendungsentwicklung. Für jede Entwicklungsstufe kann es eine eigene Ansicht geben.

- Gibt es nur eine Ansicht, so kann der Entwickler verwirrt werden, da er nicht weiß, bei welcher Aufgabe er gerade ist.

Dieses Kriterium ist nicht auf Komponentenmodelle anwendbar, sondern bezieht sich nur auf deren Entwicklungsumgebungen.

Wiederverwendung durch Verweis (K7): Wird eine Komponente eingebunden, geschehe das (nach [LR00, LR01]) durch einen Verweis auf die Komponentenbeschreibung. Dadurch werde vermieden, daß mehrere unterschiedliche Kopien einer Komponente existieren.

+ Wird eine neue Komponente in ein System eingespielt, wird durch ein Verweis automatisch in jeder schon existierenden Anwendung die neue Komponente verwendet.
- Wird für jedes Programm eine Kopie einer Komponente angelegt, muß bei einem Versionswechsel einer oder mehrerer Komponenten die Anwendung aufwendig mit den neuen Komponenten aktualisiert werden.

Cox und Song [CS01] ergänzen diese Anforderungen durch den folgenden Punkt:

Test- und Verifikationsmethoden (K8): Für eine Komponente seien Test- und Verifikationsmethoden notwendig, um die Richtigkeit zu prüfen. Damit könne sie gut wiederverwandt werden. Es werde gefordert, daß Tests einer Komponente mitgegeben werden.

+ Werden Tests mit einer Komponente ausgeliefert, so kann jederzeit geprüft werden, ob sie richtig funktioniert. Die Testfälle können auch genutzt werden, um z.B. Anwendungsfälle zu erkennen.
- Funktioniert eine Komponente nicht, so wie man sich es erhofft, ist es schwer, ohne Testfälle herauszufinden, ob die Komponente fehlerhaft ist oder ob die Anwendung die Komponente falsch einbindet.

Die zuvor genannten Kriterien werden von mir durch die folgenden Punkte erweitert:

Klarer Aufgabenbereich (K9): In Abschnitt 2.1 wurden drei Definitionen für eine Komponente angegeben. In allen drei wird gefordert, daß eine Komponente leicht von Dritten benutzt werden könne. Um dies zu gewährleisten, benötigt eine Komponente einen abgegrenzten Aufgabenbereich. Dadurch läßt sich erkennen, wofür eine Komponente geeignet ist, und sie läßt sich somit leichter verwenden.

+ Ist der *Aufgabenbereich* klar definiert, ist es leicht, eine Komponente in einer Anwendung gezielt einzusetzen.
- Ist es nicht klar, welche Fähigkeit eine Komponente genau besitzt, ist es zweifelhaft, ob sie auch richtig und effizient eingesetzt werden kann.

Geringe Komplexität (K10): Mezini und Haupt [MH01a] fordern, daß eine Komponente unabhängig sei. Um dies zu fördern, ist es meiner Meinung nach notwendig, daß die Komponente eine geringe Komplexität aufweist. Dies ist leichter zu erreichen, wenn eine Komponente einen *klaren Aufgabenbereich (K9)* hat. Meyer [Mey90] sagt in dem Prinzip der *wenigen Schnittstellen*, daß mit möglichst wenig anderen kommuniziert werden solle, um möglichst unabhängig zu sein.

+ Besitzt eine Komponente eine *geringe Komplexität*, so läßt sich ihr Einsatzgebiet leicht erkennen und darin einsetzen.

- Ist eine Komponente von der Funktionalität her mächtig, ist es manchmal schwierig, das genaue Anwendungsgebiet festzustellen. Es schleichen sich auch eher Mängel ein, die später schwerer zu finden und zu beheben sind.

Sprachunabhängigkeit (K11): Komponenten versprechen Effizienz u.a. durch Wiederverwendung. Dies wird erleichtert, wenn möglichst viele Komponenten genutzt werden können. Wenn ein Komponentensystem sprachunabhängig ist, ist die Auswahl an Komponenten deutlich größer. Ein Komponentenmodell muß dafür Mechanismen bieten, daß sich in unterschiedlichen Sprachen implementierte Komponenten verständigen können.

+ Durch die Spachunabhängigkeit ist die Erstellung von Anwendungen durch eine größere Auswahl von Komponenten erleichtert.

- Ist ein System sprachabhängig, so können nur Komponenten, die für diese Sprache geschrieben wurden, genutzt werden. U.U. ist die Auswahl an Komponenten begrenzt.

In den folgenden Kapiteln werden Komponentenmodelle und Entwicklungsumgebungen vorgestellt. Dabei werden zum Vergleich der einzelnen Modelle bzw. Umgebungen die zuvor genannten Kriterien herangezogen. Eine tabellarische Bewertung anhand der Kriterien ist jeweils am Schluß der Beschreibung des Modells zu finden.

2.3 Überblick über bestehende Modelle

Es gibt eine ganze Reihe von unterschiedlichen Komponentenmodellen. Im folgenden werden einige vorgestellt.

2.3.1 Java Beans

Java Beans [Gie01, Eng97] ist eines der ersten Komponentenmodelle, die für Java entwickelt wurden. Änderungen an der Sprache Java selbst wurden nicht gemacht. Das Modell eignet sich zum Benutzen und Erzeugen von Komponenten. Diese sind speziell so entwickelt, daß sie in einer visuellen Entwicklungsumgebung genutzt werden können. Sun definiert [Eng97] *Java Beans* als: „A *Java Bean* is a reusable software component that can be manipulated visually in a builder tool.“

Jedes *Bean* wird durch eine normale Klasse ausgeprägt, die die Schnittstelle `java.io.Serializable` implementiert. Durch dieses Vorgehen ist automatisch ein Persistenz-Mechanismus aktiviert, der den Zustand eines *Beans* speichern kann. Der Zustand wird durch Eigenschaften definiert, die mit Hilfe von `set`- und `get`-Methoden verändert werden können. Zusätzlich besteht die Möglichkeit, Funktionalität mit weiteren Methoden oder anhand von Ereignissen zu definieren. Dabei wird das Interesse an einem Ereignis beim Sender vermerkt und dieser ruft beim Interessent eine spezielle Methode auf. Die Schnittstelle eines *Beans* bzw. einer Komponente wird durch die öffentlichen Methoden definiert. Dies entspricht dem Kriterium des *modularen Design* (*K1*), wobei dem Entwickler überlassen wird, wieviel von einem *Bean* öffentlich ist und ob es aus mehreren Klassen besteht.

Jedes *Bean* kann zur Laufzeit in eine Anwendung eingebunden werden. Dafür ist die Introspektion zuständig, die anhand von Entwurfsmustern versucht, Eigenschaften, Methoden und Ereignisse zu erkennen. Zum Unterstützen der Introspektion wird empfohlen, einfache Entwurfsmuster einzuhalten, wie z.B., daß Methoden für den Zugriff auf Eigenschaften jeweils mit `set` und `get` beginnen, gefolgt von dem Namen der Eigenschaft. Alternativ kann auch eine *InfoBean* Klasse, die die Komponente ausführlicher beschreibt (*K2*), implementiert werden. Welche Aufgabe (*K9*) ein *Bean* erfüllt und wie komplex (*K10*) sie ist, wird dem Entwickler überlassen.

Um *Beans* zu einer Anwendung zusammenzubauen, ist ein *Container* notwendig. Der *Container* übernimmt z.B. das Anordnen von *Beans* auf der Benutzungsoberfläche oder das Anpassen der Eigenschaften (*K5*) eines *Beans*. Durch das separate Entwickeln der *Beans* und das anschließende Erstellen der Anwendung ist auch ein zweigeteilter Entwicklungsprozesses (*K4*) zwischen Komponenten- und Anwendungsentwicklung gegeben.

Da ein *Bean* durch seinen Klassennamen referenziert wird, ist die *Wiederverwendung durch Verweis* (*K7*) gegeben. Über die Wahl eines Namens wird keine Aussage gemacht, deshalb ist kein eindeutiger Namensraum (*K3*) möglich.

Das Kriterium der *Test- und Verifikationsmethoden* (*K8*) erfordert, daß eine Komponente Testmethoden mitbringen soll. Dies ist bei *Java Beans* nicht vorgesehen. Ebenfalls nicht vorgesehen ist eine *Sprachunabhängigkeit* (*K11*).

K1	modulares Design	nicht erzwungen
K2	Selbstbeschreibung	in separater Klasse
K3	globaler Namensraum	nein
K4	zweigeteilter Entwicklungsprozeß	ja
K5	Anwendungen zusammenbauen	ja
K6	verschiedene Ansichten	nicht relevant
K7	Wiederverwendung durch Verweis	ja
K8	Testmethoden	nein
K9	klarer Aufgabenbereich	nicht erzwungen
K10	geringe Komplexität	nicht erzwungen
K11	Sprachunabhängigkeit	nein

Tabelle 2.1: Kriterien für *Java Beans*

2.3.2 InfoBus

InfoBus [SUN99] ist ein weiteres kommerzielles Komponentensystem speziell für das Erzeugen und Verwalten von kleinen wiederverwendbaren *Java Beans*.

Der *InfoBus* stellt ein Bussystem dar, über das alle Komponenten, die die Klasse `InfoBusMember` implementieren, miteinander kommunizieren. Die Zusammenarbeit der Komponenten geschieht über wenige Schnittstellen, wodurch für eine *geringe Komplexität* (*K10*) gesorgt wird. Die Kommunikation läuft asynchron und symmetrisch ab. Optional kann auch eine Komponente als *Master* dienen, die die Zusammenarbeit von anderen Komponenten regelt.

Im *InfoBus* gibt es drei verschiedene Arten von Komponenten. Sie können als Daten-Produzent und Daten-Konsument und Daten-Kontroller agieren. Der Daten-Kontroller hat die Aufgabe zwischen Produzent und Konsument zu vermitteln. Dabei geschieht die Kommunikation zwischen den Komponenten über ein Ereignissystem (*K1*). Der *InfoBus* stellt dazu wenige Ereignisse bereit. Die Semantik des Datenflusses wird durch die Kommunikationsdaten, die aus den eigentlichen Daten (Java Objekte) und den Navigationsinformationen bestehen, bestimmt. Ein Produzent kann auch mehrere Konsumenten bedienen. Da die Komponenten auf *Java Beans* basieren, kann eine Dokumentation (*K2*) wieder in der *InfoBean* Klasse angegeben werden.

Eine Trennung (*K4*) zwischen Komponentenentwicklung und Anwendungsentwicklung ist deutlich zu sehen. Das Zusammenbauen von Anwendungen geschieht, indem die Komponenten in den *InfoBus* eingehängt (*K5*) werden. Die Produzenten empfangen darauf Anfragen nach bestimmten Daten und die Konsumenten erhalten Benachrichtigungen, wenn entsprechende Daten bereitstehen. Die Daten werden dabei anhand von Namen identifiziert. Bei dem Entwickeln von Produzenten und Konsumenten muß auf ein einheitliches Datenformat geachtet werden.

Da eine Komponente durch ihren Klassennamen referenziert wird, ist die *Wiederverwendung durch Verweis* (*K7*) gegeben. Über die Wahl eines Namens ist keine Aussage getroffen, deshalb ist kein eindeutiger Namensraum (*K3*) möglich. Erstellt ein Entwickler eine Komponente, muß er selbst dafür sorgen, daß eine Komponente einen *klaren Aufgabenbereich* (*K9*) bietet. Das Beipacken von Testmethoden (*K8*) ist auch nicht vorgesehen. Der *InfoBus* ist nur in Java (*K11*) realisiert und beschränkt die Zusammenarbeit der Komponenten auf eine virtuelle Maschine.

K1	modulares Design	ja
K2	Selbstbeschreibung	in separater Klasse
K3	globaler Namensraum	nein
K4	zweigeteilter Entwicklungsprozeß	ja
K5	Anwendungen zusammenbauen	ja
K6	verschiedene Ansichten	nicht relevant
K7	Wiederverwendung durch Verweis	ja
K8	Testmethoden	nein
K9	klarer Aufgabenbereich	nicht erzwungen
K10	geringe Komplexität	ja
K11	Sprachunabhängigkeit	nein

Tabelle 2.2: Kriterien für *InfoBus*

2.3.3 ArchJava

Ein Komponentensystem aus dem akademischen Umfeld ist *ArchJava* [ACD02]. Das Hauptaugenmerk liegt darauf, die Konsistenz zwischen Quelltext und Architektur jederzeit zu erhalten. *ArchJava* vereinheitlicht dazu Architektur und Implementation von Software in einer Sprache. Die Sprache (*K11*) ist Java-ähnlich und auf eine virtuelle Java Maschine begrenzt.

Komponenten in *JavaArch* sind spezielle Arten von Objekten. Sie werden von Komponentenklassen ausgeprägt. Die zu wählenden Namen der Komponentenklassen sind gleich einer Java-Klasse und erlaubten dadurch keinen eindeutigen Namensraum (*K3*). Das Wiederverwenden (*K7*) wird durch einen Verweis erlaubt. Um eine Anwendung zu spezifizieren und eine Typprüfung durchzuführen, können Komponenten auch abstrakt (ohne Implementation) sein.

Die Kommunikation zwischen den Komponenten geschieht über definierte *Ports* und wird direkt ohne einen Dritten realisiert. Durch ein *provides Port* wird von einer Komponente die Funktionalität bereitgestellt. Ein *requires Port* einer Komponente zeigt an, daß weitere Komponenten notwendig sind, um der ersten zuzuarbeiten. Eine dritte Art ist das *broadcast Port*. Hier wird auch die Funktionalität

von anderen Komponenten benötigt, mit dem Unterschied, daß mehrere Komponenten angesprochen werden können. Diese dürfen aber kein Ergebnis zurückliefern.

Nach dem Erstellen der Komponenten (*K4*) kann mit dem Entwickeln der Anwendung (*K5*) begonnen werden. Dazu werden *requires Ports* mit genau einem *provides Port* verbunden. *Broadcast Ports* können mit mehreren *provides Ports* verbunden werden. Durch die strikte Kommunikation über die *Ports* wird ein *modulares Design* (*K1*) erreicht. Gleichfalls wird dadurch das Einhalten einer *geringen Komplexität* (*K10*) begünstigt. Beim Verbinden der Komponenten muß beachtet werden, daß die *Ports* die gleichen Objekte liefern und erwarten, sonst wird die Anwendung mit einer 'Ausnahme' abgebrochen.

Ein weiteres Mittel, um ein *modulares Design* (*K1*) zu erreichen, ist, daß Komponenten aus anderen zusammengesetzt werden können. Dadurch kann ein weiteres Aufteilen in private und öffentliche Teile geschehen. Dafür, daß eine Komponente einen *klaren Aufgabenbereich* (*K9*) hat, muß der Entwickler sorgen. *ArchJava* bietet keine Möglichkeit zum Beschreiben (*K2*) von Komponenten und zum Mitgeben von Testmethoden (*K8*).

K1	modulares Design	ja
K2	Selbstbeschreibung	nein
K3	globaler Namensraum	nein
K4	zweigeteilter Entwicklungsprozeß	ja
K5	Anwendungen zusammenbauen	ja
K6	verschiedene Ansichten	nicht relevant
K7	Wiederverwendung durch Verweis	ja
K8	Testmethoden	nein
K9	klarer Aufgabenbereich	nicht erzwungen
K10	geringe Komplexität	ja
K11	Sprachunabhängigkeit	nein

Tabelle 2.3: Kriterien für *ArchJava*

2.3.4 Enterprise Java Beans

Eines weiteres Komponentenmodell sind die *Enterprise Java Beans* (*EJB*) [Völ02, Ble01, MS00b, MH01b]. Es ist nicht ganz vergleichbar mit den zuvor beschriebenen Modellen, da es für verteilte komponentenbasierte Systeme gedacht ist. Es soll in dieser Arbeit trotzdem betrachtet werden, da in Kapitel 8 Systeme vorgestellt werden, die auf EJB basieren.

Das Augenmerk von *Enterprise Java Beans* liegt auf der Entwicklung von verteilten, geschäftskritischen Anwendungen. Dabei werden Gebiete wie Sicherheit oder

Transaktionsintegrität dem Programmierer abgenommen und vom Komponentenmodell bereitgestellt. Das Modell der EJB ist speziell auf Java zugeschnitten, dadurch ist keine Sprachunabhängigkeit (*K11*) gewährleistet.

Das Komponentenmodell gliedert die Aufgabe in zwei Teile. Die *Komponenten* sind für die funktionalen Belange zuständig, und die *Container* sorgen für die technischen Aufgaben.

Die *Container* bieten den Komponenten die Laufzeitumgebung. Sie erzeugen einzelne Ausprägungen von Komponenten und löschen diese, wenn sie nicht mehr benötigt werden. Sie bieten den Komponenten zusätzliche Dienstleistungen wie Sicherheit, Ressourcenverwaltung oder Transaktionsintegrität an, ohne daß der Entwickler viel Arbeit investieren muß.

Die eigentliche Entwicklungsarbeit liegt in den Komponenten, den EJBs. Es gibt drei verschiedene Arten von Komponenten:

Stateless Session Beans sind zustandslose Komponenten, die bei jedem Aufruf gleich sind. Sie führen eine abgeschlossene Aufgabe aus und beenden sich selbst sofort nach der Rückgabe des Ergebnisses. Eine Ausprägung dieser Komponenten kann von mehreren Anwendungen gleichzeitig benutzt werden.

Stateful Session Beans sind zustandsbehaftet, und für jede Anwendung wird eine neue Ausprägung erstellt. Diese Ausprägung kann inaktiviert werden, wenn sie zwischenzeitlich nicht mehr benötigt wird. Die Komponente kann viele Interaktionen ausführen, die miteinander im Zusammenhang stehen, und ist in der Lage, Informationen von Aufruf zu Aufruf zu speichern.

Entity Beans repräsentieren in der Regel Daten in einer Datenquelle (z.B. Datenbank). Es gibt immer nur eine Ausprägung für eine Datenquelle, die bei Bedarf auch inaktiviert werden kann. Dafür kann die Komponente von mehreren Benutzern gleichzeitig angesprochen werden. Eine wichtige Aufgabe ist, die Datenquelle immer mit der Komponente synchron zu halten. Diese Aufgabe kann durch die Persistenzverwaltung von einem *Container* übernommen werden.

Enterprise Java Beans sind Komponenten, die auf einem Anwendungs-Server ausgeführt werden. Dabei ist es egal, ob dieser lokal oder entfernt ist. Eine Anwendung merkt nicht, wo die Komponenten ausgeführt werden. Daß die Komponenten eindeutige Namen haben, um einen globalen Namensraum (*K3*) zu ermöglichen, wird von EJB nicht vorausgesetzt.

Jede Komponente ist aus mehreren Teilen aufgebaut:

Enterprise Bean Class implementiert die Funktionalität einer Komponente. Je nach Art der Komponente muß diese ein anderes *Enterprise Bean Interface* implementieren. Eine *Enterprise Bean Class* wird in ihrem *Container* ausgeführt.

EJB Remote Class definiert die Schnittstelle einer Komponente, die von einer Anwendung benutzt werden kann. Zusätzlich werden hier die Zugriffsrechte definiert.

EJB Home Class wird automatisch generiert und ist für die Laufzeitverwaltung zuständig. Anhand dieser Klasse kann ein *Container* eine Ausprägung der Komponente erstellen und auch wieder löschen.

XML Development Descriptor enthält Informationen über die Komponente und bietet dadurch eine Selbstbeschreibung (*K2*). Die Informationen beinhalten z.B., wie eine Komponente konfiguriert werden kann oder welche Sicherheitsregeln gelten.

Ist eine Komponente erstellt, so kann sie mittels einer *JAR*-Datei verpackt werden und durch einen Anwendungs-Server bereitgestellt werden. Diese Datei enthält alle oben beschriebenen Bestandteile einer Komponente. Durch die zentrale Lagerung der Komponenten ist sichergestellt, daß eine Anwendung eine Komponente immer nur durch Verweis (*K7*) anspricht. So wird garantiert, daß immer die aktuellste Version benutzt wird.

Sind die Komponenten auf dem Anwendungs-Server bereitgestellt (*K4*), können die Anwendungen aus diesen zusammengebaut werden. Dies geschieht indem die Methoden der *EJB Remote Class* in einer Anwendung verwendet werden, um so ein EJB zu konfigurieren und aufzurufen (*K5*).

K1	modulares Design	nicht erzwungen
K2	Selbstbeschreibung	in separater Datei
K3	globaler Namensraum	nein
K4	zweigeteilter Entwicklungsprozeß	ja
K5	Anwendungen zusammenbauen	ja
K6	verschiedene Ansichten	nicht relevant
K7	Wiederverwendung durch Verweis	ja
K8	Testmethoden	nein
K9	klarer Aufgabenbereich	nicht erzwungen
K10	geringe Komplexität	nein
K11	Sprachunabhängigkeit	nein

Tabelle 2.4: Kriterien für *Enterprise Java Beans*

Die Erstellung einer EJB ist durch die vielen verschiedenen Bestandteile einer Komponente recht komplex (*K10*). Dadurch, daß es verschiedene Arten von Komponenten gibt, wird die Auswahl der benötigten Klassen um einiges schwieriger. Ein sinnvolles *modulares Design* (*K1*) wird dem Entwickler genauso wie ein *klarer Aufgabenbereich* (*K9*) selbst überlassen. Eine Möglichkeit, ein EJB mit *Testmethoden* (*K8*) zu ergänzen, ist nicht vorgesehen.

2.4 HotAgent Komponentenmodell

Im vorausgehenden Teil wurden verschiedene Komponentenmodelle vorgestellt. Nun ist die Frage, welches ausgewählt wird, um HotAgent zu erstellen. Wird *Java Beans* (s. 2.3.1) betrachtet, fällt auf, daß die Schnittstelle sehr offen gehalten ist und dadurch ein *modulares Design* (*K1*) erschwert wird. Der *InfoBus* (s. 2.3.2) benötigt ein Bussystem für die Kommunikation von Komponenten und wird deshalb als nicht geeignet eingestuft. *ArchJava* (s. 2.3.3) wäre eine gute Wahl, bietet aber leider keine Möglichkeit der *Selbstbeschreibung* (*K2*). Das verteilte System *Enterprise Java Beans* (s. 2.3.4) ist zu komplex (*K10*). Alle vorgestellten Komponentenmodelle bieten auch keine Möglichkeit, *Testmethoden* (*K8*) einer Komponente beizulegen. Aus diesen Gründen wird in dieser Arbeit ein einfaches Komponentenmodell, wie im folgenden beschrieben, eingeführt.

Beim HotAgent Komponentenmodell wird auf eine geringe Komplexität (*K10*) geachtet, die keinen zusätzlichen Aufwand bei der Verwendung von Komponenten mit sich bringt. Die Kommunikation zwischen den Komponenten wird durch nur *eine* einheitliche Schnittstelle (*K10*) erreicht, die von jeder Komponente implementiert werden muß. Die Kommunikation geschieht direkt zwischen den betroffenen Komponenten ohne Beteiligung von Dritten. Es wird stets darauf geachtet, daß bei Bedarf das HotAgent Komponentenmodell auf ein anderes übertragen werden kann.

Das Komponentenmodell ist in Smalltalk implementiert. Die Definition ist aber so offen gehalten, daß es auch in anderen Sprachen umgesetzt werden kann. Für eine *Sprachunabhängigkeit* (*K11*) ist aber zusätzlich noch ein Mechanismus nötig, um die Kommunikation zwischen den verschiedenen Sprachen zu ermöglichen. Da eine Komponente nur eine einheitliche Schnittstelle (*K10*) hat, die für die Kommunikation mit allen anderen Komponenten benutzt wird, ist gewährleistet, daß alle Komponenten, unabhängig von ihrer Aufgabe, miteinander kommunizieren können. Jede Komponente stellt spezielle Ein– und Ausgänge zur Verfügung, um Verbindungen zu anderen herzustellen. Es ist nur möglich, über diese Ein– und Ausgänge die Funktionen der Komponente anzusprechen (*K1*). Es ist nicht vorgesehen, neue Schnittstellen für spezielle Komponenten zu schreiben. Eine Auflistung aller Kriterien für das HotAgent Komponentenmodell ist in Tabelle

2.5 zu sehen. Manche werden erst noch in den kommenden Abschnitten angesprochen.

K1	modulares Design	ja	Kap. 2.4
K2	Selbstbeschreibung	ja	Kap. 2.4.1
K3	globaler Namensraum	nicht erzwungen	Kap. 2.4.1
K4	zweigeteilter Entwicklungsprozeß	ja	Kap. 6
K5	Anwendungen zusammenbauen	ja	Kap. 2.4.4
			Kap. 6.2
K6	verschiedene Ansichten	ja	Kap. 6.1
			Kap. 6.5
K7	Wiederverwendung durch Verweis	ja	Kap. 2.4.2
K8	Testmethoden	ja	Kap. 6.3
K9	klarer Aufgabenbereich	nicht erzwungen	Kap. 6.2
K10	geringe Komplexität	ja	Kap. 2.4.3
K11	Sprachunabhängigkeit	nein	Kap. 2.4

Tabelle 2.5: Kriterien für das HOTAGENT Komponentenmodell

Das wichtigste Element für die Realisierung von Komponenten ist die Klasse `HAComponent`. Diese Klasse dient als gemeinsame Klasse, mit der neue *Komponentenschablonen* erstellt werden können. Komponentenschablonen beschreiben, wie eine Komponente zur Laufzeit aussieht und welche Funktionalität sie bietet. Zum Erstellen einer Schablone erbt eine neue Klasse (z.B. `AComponent`) von der Klasse `HAComponent`. Die Klasse `HAComponent` bildet die einzige Schnittstelle einer Komponente. Ein modulares Design (*K1*) ermöglicht, daß die Komponentenschablone `AComponent` mehrere Klassen, die die Funktionalität für eine Komponente bereitstellen, kapseln kann. Diese gekapselten Klassen können somit nur über die von `HAComponent` bereitgestellte Schnittstelle angesprochen werden. Im folgenden werden diese Klassen, da es sich auch um eine einzelne handeln kann, zum Vereinfachen mit `AClass` bezeichnet.

Anhand der Komponentenschablonen kann ein *Komponentenexemplar* erstellt werden. Ein Komponentenexemplar ist eine Ausprägung einer Komponente und kann durch ein Komponentenprogramm genutzt werden. Dieses stellt die eigentliche Funktionalität einer Komponente bereit.

In den folgenden Abschnitten werden Methoden beschrieben, die für eine Komponente implementiert werden müssen und die die Erstellung einer Komponente unterstützen.

2.4.1 Beschreibungen

Um das Kriterium der *Selbstbeschreibung* (*K2*) zu erfüllen, bietet das HotAgent Komponentenmodell (s. Abb. 2.1) drei Methoden im Klassenprotokoll der Komponentenschablone `HAComponent` an. Mit diesen Methoden können Komponenten untereinander abgegrenzt werden.

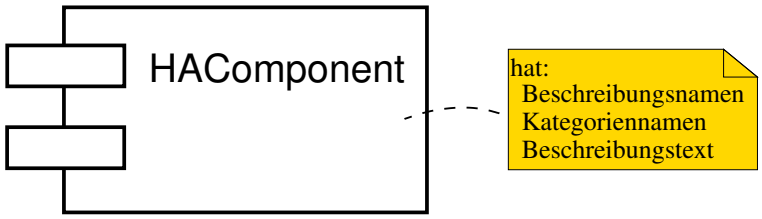

Abbildung 2.1: Komponenten Beschreibung

Die Methode `descriptionName` dient dazu, einen eindeutigen, globalen *Beschreibungsnamen* (*K3*) für eine Komponentenschablone zu definieren. Ein global eindeutiger Name wird aber nicht vom Komponentenmodell erzwungen. Als Konvention sollte eingehalten werden, daß der Anfangsbuchstabe des Beschreibungsnamens groß ist.

```
descriptionName
   "every component has a unique name"

   ^'Name'
```

Um Komponenten besser nach ihrer Funktion sortieren zu können, bzw. sie wiederzufinden, wird die Klassenmethode `categoryName` bereitgestellt. In ihr kann der *Kategoriename* angegeben werden, in die die Komponente einsortiert werden soll.

```
categoryName
   "every component has a category name"

   ^'Category'
```

Wird nun eine Komponente benötigt, kann sie anhand der Kategorie und ihres Namens leicht gefunden werden. Um sich zu vergewissern, daß die Komponente auch das tut, was man erwartet, existiert ein Text, der die Komponente kurz beschreibt. Dieser *Beschreibungstext* wird in der Methode `descriptionText` angegeben.

```
descriptionText
   "returns a text describing the component"

   ^'any text'
```

2.4.2 Lebensdauer-Verwaltung

Ist eine *Komponentenschablone* fertiggestellt, so kann ein *Komponentenexemplar* aus der Schablone erstellt werden (*K7*). Dazu muß die Klassenmethode `new` der Komponentenschablone `AComponent` aufgerufen werden.

```
| aComponent |
aComponent := AComponent new.
```

Ein Löschen, der nicht mehr benötigten Komponente, ist nicht notwendig, da Smalltalk eine automatische Speicherbereinigung [Lut87] anbietet. Durch die Komponentenschablone werden Exemplarmethoden bereitgestellt, mit denen das Laufzeitverhalten und die Speicherbelegung einer Komponente gesteuert werden können. Zum Initialisieren, Aktivieren, Deaktivieren und Entfernen einer Komponente stehen jeweils Methoden zur Verfügung, in denen benötigte Aktionen definiert werden können. Diese werden beim Starten bzw. Beenden einer Anwendung automatisch für jede verwendete Komponente von der Anwendung aufgerufen.

Initialisieren: Die erste zu implementierende Exemplarmethode ist `initialize`. Hier werden Anweisungen festgehalten, die zum Initialisieren benötigt werden. Es werden Aufrufe zum Ausprägen der zu kapselnden Klassen (`AClass`) und zum Erstellen von Ein– und Ausgängen (s. Abschnitt 2.4.3) benötigt.

```
initialize
   "set up component"
   super initialize.

   "create instances of classes"
   ...
   "create entrances and exits"
   ...
```

Wichtig ist die erste Zeile der Methode. Dort muß unbedingt die Exemplarmethode `initialize` der Elternklasse `HAComponent` aufgerufen werden.

Aktivieren: Das Aktivieren dient dazu, z.B. Prozesse anzustoßen, die für die Bearbeitung in der Komponente wichtig sind. Außerdem sollte bei einer sichtbaren Komponente (Erklärung folgt in Abschnitt 2.4.5) hier das Kontrollobjekt gesetzt werden. Die entsprechende Exemplarmethode hierfür heißt `activate`.

```
activate
   "Activate component"
   super activate.

   "Create controller"
   view controller: view defaultControllerClass new.
```

Zu beachten ist wieder der Aufruf von `super activate`.

Deaktivieren: Deaktiviert wird eine Komponente, wenn ihre Funktionalität im Moment nicht mehr gebraucht wird. Es sollten z.B. Prozesse angehalten werden und Kontrollobjekte entfernt werden. Die entsprechende Methode hierfür heißt `deactivate`.

```
deactivate
   "Create controller"
   view controller: NoController new.

   "Deactivate component"
   super deactivate.
```

Zu beachten ist wieder der Aufruf von `super deactivate`, der am Ende der Methode zu stehen hat.

Entfernen: Wird eine Komponente nicht mehr benötigt, ist es notwendig, eventuelle Prozesse oder andere Reste zu entfernen, damit eine effektive automatische Speicherbereinigung durchgeführt werden kann. Dies kann in der Methode `release` geschehen.

```
release
   "clean up"

   "Release component"
   super release
```

Auch hier darf der letzte Aufruf nicht in Vergessenheit geraten.

Damit ein Komponentenexemplar identifiziert werden kann, besitzt jedes einen *Exemplarnamen*, der in der Anwendung eindeutig sein muß, was aber ebenfalls auch nicht vom Komponentenmodell erzwungen wird. Dieser kann mit den Exemplarmethoden `name` abgefragt und mit `name:` gesetzt werden. Als Konvention soll der Exemplarname klein geschrieben werden. Ein möglicher Aufruf für unsere Beispielkomponente `aComponent` ist:

```
aComponent name: 'a unique name'.
```

2.4.3 Erstellen von Ein– und Ausgängen

Wie zuvor erwähnt, stellt die Schnittstellenklasse `HAComponent` Exemplarmethoden zur Konstruktion der *Ein–* und *Ausgänge* bereit. Zum Gewährleisten der Funktionalität helfen noch zusätzlich die Klassen `HAComponentEntrance` und `HAComponentExit`. Abbildung 2.2 stellt den für die Ein– und Ausgänge relevanten Teil der oben genannten Klassen dar. Die Klasse `AClass` dient als Platzhalter für die Klassen, die die Funktionalität für die Komponente bereitstellen. Die Ein– und Ausgänge greifen auf diese Klassen zu, um Operationen anzustoßen oder Informationen zu setzen bzw. zu erfragen.

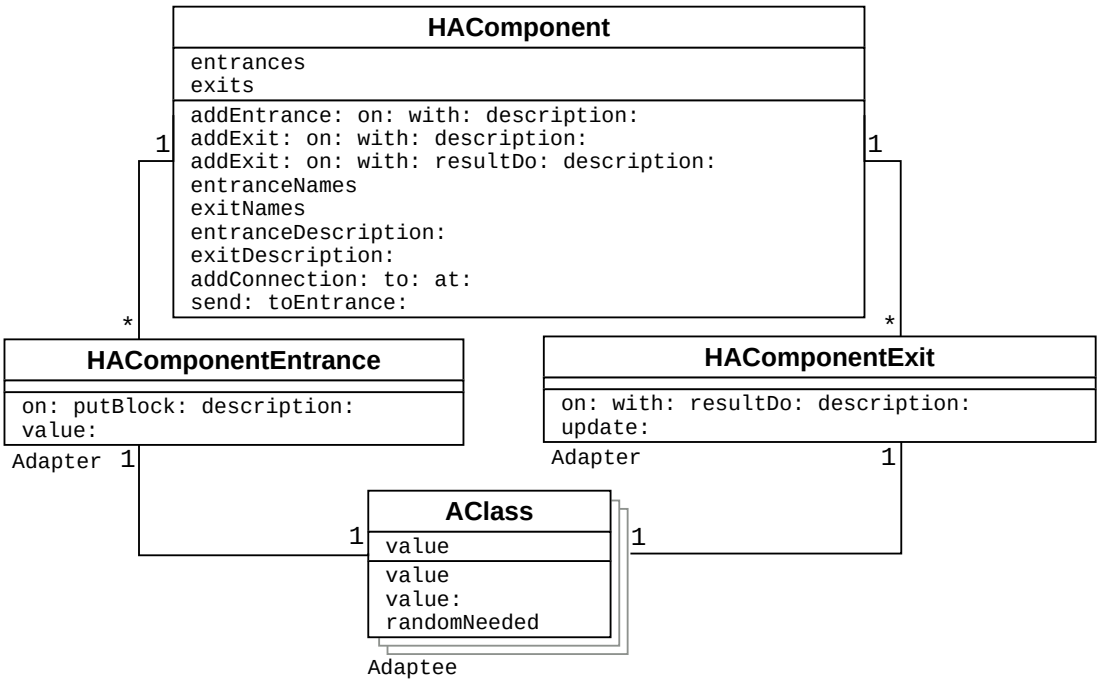

Abbildung 2.2: Ein– und Ausgänge einer Komponente (Ausschnitt)

Die im folgenden gezeigten Quelltextausschnitte sind jeweils in der Methode `initialize` anzugeben. Als erstes ist eine Ausprägung der zu kapselnden Klas-

se, die die Funktionalität für eine Komponente bereitstellt, zu erzeugen. In dem folgenden Beispiel ist es ein Behälter (`vh`) für eine Zahl:

```
| vh |
vh := ValueHolder newFraction.
```

Mit der Methode `addEntrance: on: for: description:` kann für eine Komponente ein *Eingang* definiert werden. Folgender Quelltext zeigt, wie ein Eingang set value erzeugt wird.

```
self
   addEntrance: 'set value'
   on: vh
   for: [:obj :param |
      obj value: param]
   description: 'stores a fraction value'.
```

Als Parameter werden ein Name für den Eingang (`'set value'`), das gekapselte Objekt (`vh`) und nach dem Schlüsselwort `for:` ein Anweisungsblock erwartet. Zusätzlich ist nach dem Schlüsselwort `description:` noch ein Text zur Beschreibung (*K2*) des Eingangs anzugeben. Der Anweisungsblock hat die Aufgabe, die Aktionen auszuführen, die bei einer Aktivierung des Eingangs ausgeführt werden sollen. Ihm wird die gekapselte Klasse (`vh`) als Parameter mit dem Namen `obj` übergeben. Jeder Eingang kann auch einen Parameter erhalten, der in diesem Block als Variable `param` bereit gestellt wird. Wird ein Eingang angesprochen, werden alle Anweisungen in diesem Block ausgeführt und durch die Variablen `obj` und `param` besteht die Möglichkeit, dabei auf die gekapselten Objekte und den übergebenen Parameter zuzugreifen. Beim Aufruf der Methode wird eine Instanz der Klasse `HAComponentEntrance` erzeugt, in der die Daten für den Zugriff auf `vh` abgelegt werden. Eine Referenz auf diese Ausprägung wird mit dem Eingangsnamen in der Komponente gespeichert. Die Namen aller Eingänge können mit `entranceNames` erfragt werden.

Sollen nicht nur Nachrichten z.B. über eine Zustandsänderung an eine Komponente geschickt werden, sondern auch Anfragen, so ist es notwendig, einen *Rückgabewert* zu spezifizieren. Beim Aufruf der Methode `addEntrance: on: for: description:` wird dazu, wie zuvor beschrieben, ein Anweisungsblock festgelegt, dessen letztes ausgewertetes Objekt als Rückgabewert betrachtet wird.

Die Methode `addExit: on: with: description:` definiert einen *Ausgang*. Das Beispiel zeigt, wie der Ausgang value changed erzeugt wird.

```
self
   addExit: 'value changed'
   on: vh
   with: #value
   description: 'fraction value changed'.
```

Diese Methode erhält den Ausgangsnamen, das gekapselte Objekt `vh` und ein Symbol für den Zugriff. Darauf wird eine Instanz der Klasse `HAComponentExit` erstellt und als Beobachter von `vh` eingetragen, d.h. die Ausprägung der Klasse `HAComponentExit` wird benachrichtigt, wenn sich der Zustand von `vh` ändert. Beim Aktivieren des Ausgangs wird die durch das Symbol (`#value`) repräsentierte Methode von dem gekapselten Objekt (`vh`) aufgerufen. Das dadurch erlangte Objekt wird als Parameter am Ausgang bereitgestellt. Eine Referenz auf diesen Ausgang wird wieder von der Komponente mit dem Ausgangsnamen abgespeichert. Die Namen aller Ausgänge können mit `exitNames` erfragt werden. Als letzter Parameter für diese Methode wird wieder eine Beschreibung erwartet.

Damit ein Komponentenausgang auch einen Rückgabewert verarbeiten kann, wird die Methode `addExit: on: with: resultDo: description:` angeboten. Diese Methode verlangt noch zusätzlich einen Anweisungsblock (s. Methode `addEntrance: ...`), der festlegt, wie das Ergebnis weiterverarbeitet werden soll. Der Quelltext zeigt, wie der Ausgang `random needed` das Ergebnis an die Methode `value:` übergibt.

```
self
   addExit: 'random needed'
   on: vh
   with: #randomNeeded
   resultDo: [ :obj :result |
      obj value: result]
   description: 'requests a random and stores the result'.
```

Betrachtet man nun eine Komponente (z.B. `AComponent`) genauer, fällt auf, daß sie nur eine Schnittstelle (*K10*) hat. Damit kann jede Komponente mit jeder andren verbunden werden. Smalltalk ist eine untypisierte Sprache, d.h. beim Verbinden von Komponenten muß keine Typprüfung oder Typumwandlung vorgenommen werden. Beim Verbinden von Komponenten muß der Entwickler darauf achten, daß die Daten, die von einer Komponente bereitgestellt werden und die von einer Komponente erwartet werden, zueinander passen. Bei unabhängig entwickelten Komponenten ist das allerdings keine triviale Aufgabe. Eine automatische Prüfung wird in Kapitel 9.2 kurz besprochen. Die so weit erstellte Komponente kann wie in Abb. 2.3 dargestellt werden.

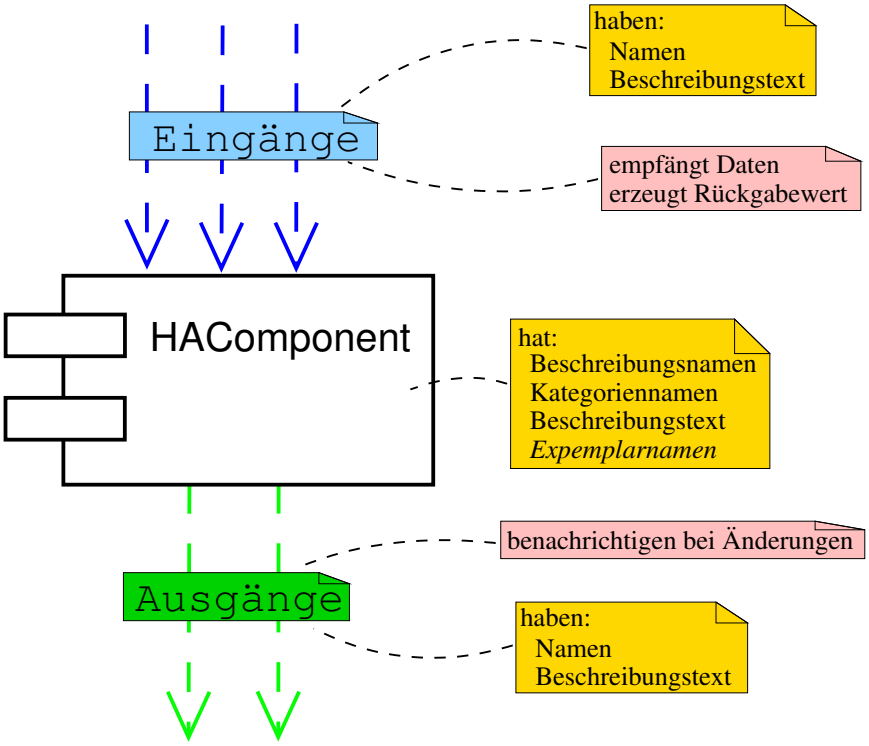

Abbildung 2.3: Komponente mit Ein- und Ausgängen

Beim Benennen der Ein– und Ausgänge einer Komponente ist darauf zu achten, daß diese innerhalb der Komponente einen eindeutigen Namen besitzen. Das Komponentenmodell erzwingt keine eindeutige Namensgebung.

Um die Beschreibungen (*K2*) eines Eingangs bzw. eines Ausgang zu erfragen, stellt eine Komponente zwei Methoden bereit. `entranceDescription:` gefolgt von einem Eingangsnamen liefert den Beschreibungstext des Eingangs und `exitDescription:` gefolgt von einem Ausgangsnamen entsprechend die Beschreibung zu dem Ausgang.

2.4.4 Verbinden

Sind nun die Komponentenschablonen fertig, geht es darum, aus ihnen Komponentenprogramme zu erstellen. Dazu müssen als erstes alle notwendigen Komponentenexemplare erzeugt werden. Ist dies geschehen, können sie miteinander verbunden (*K5*) werden, indem die Exemplarmethode `addConnection: to: at: label:` von der jeweiligen Quellkomponente verwendet wird.

Das folgende Beispiel benutzt zwei Komponenten. Die erste Komponente `AComponent` wurde in den vorangegangenen Abschnitten mit ihren Ein– und Ausgängen

beschrieben. Die zweite Komponente `RandomComponent` stellt den Eingang get random zur Verfügung, um eine Zufallszahl zu erfragen. Die erzeugte Zahl wird als Ergebnis zurückgegeben.

```
| aComponent randomComponent |
aComponent := AComponent new.
randomComponent := RandomComponent new.

aComponent
   addConnection: 'random needed'
   to: randomComponent
   at: 'get random'
   label: 'query a random'.
```

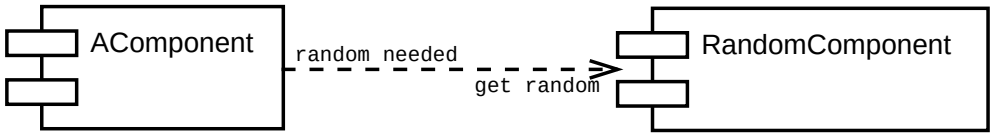

Abbildung 2.4: Verbinden von Komponenten

Der Quelltext zeigt, wie die Komponente `AComponent` den Ausgang random needed an den Eingang get random der Komponente `RandomComponent` legt (s. Abb. 2.4). Als Parameter werden hier der betreffende Ausgangsname, die Zielkomponente, deren Eingangsname und ein Beschreibungstext (*K2*) erwartet. Anders als bei den Ein– und Ausgängen einer Komponente, werden die Verbindungen durch die Komponentenausgänge in einer Liste gespeichert. Existieren mehrere Verbindungen von einem Ausgang aus, so werden diese in alphabetischer Reihenfolge anhand des Beschreibungstextes ausgeführt.

In Abb. 2.5 wird gezeigt, welche Methoden aufgerufen werden, wenn ein Ausgang aktiviert ist. Als Beispiel wird die zuvor erstellte Verbindung dargestellt. Die Komponente `AComponent` wird durch die Klassen `AComponent` (Komponentenschablone), `ValueHolder` (gekapselte Klasse) und `HAComponentExit` (Ausgang random needed) zusammengesetzt. Die Komponente `RandomComponent` besteht aus den zwei Klassen `RandomComponent` (Komponentenschablone) und der Klasse `HAComponentEntrance` (Eingang get random). Zuerst wird der Ausgang random needed (repräsentiert durch `HAComponentExit`) der Komponente `AComponent` mit der Methode `update:` benachrichtigt, daß sich der Zustand einer Variablen geändert hat. Darauf wird die Methode `send: toEntrance:` mit dem geänderten Objekt der Komponente `RandomComponent` aufgerufen. Diese leitet die Nachricht an den gefragten Eingang random needed (repräsentiert durch `HAComponentEntrance`) weiter. Dieser ermittelt nach Bedarf ein Ergebnis, das bis zum Ausgang random needed (`HAComponentExit`) durchgereicht wird, der mit der Methode `value:` die Weiterverarbeitung veranlaßt.

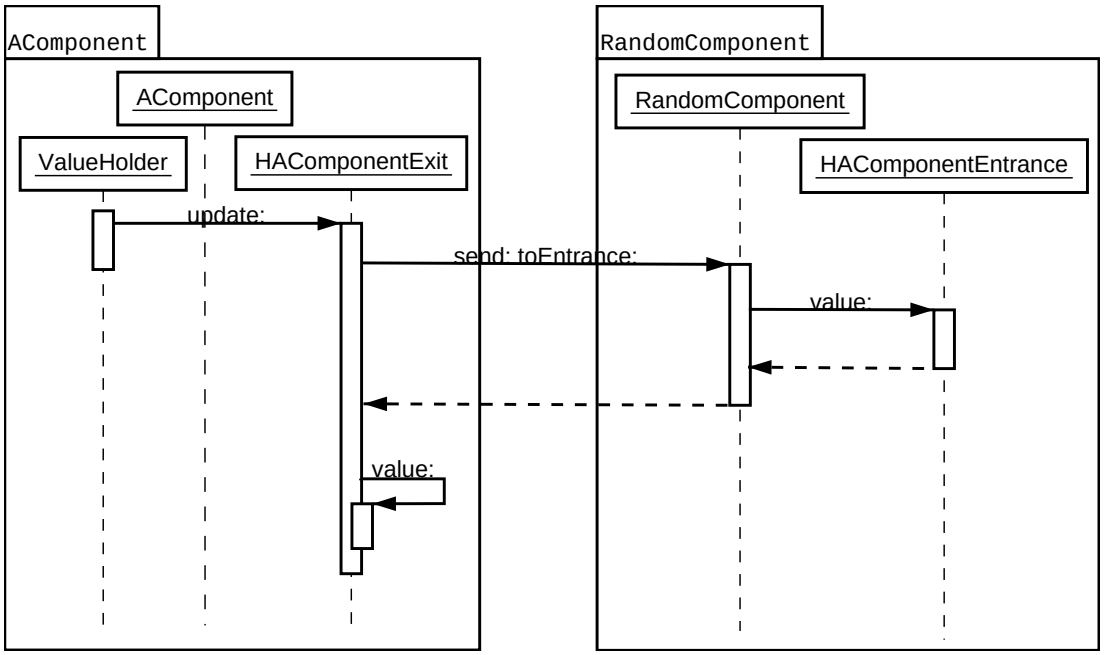

Abbildung 2.5: Verbinden von Komponenten

2.4.5 Modell–Ansicht–Kontroller

Mit Komponenten sollen alle möglichen Arten von Programmen erstellt werden können. Dazu gehören z.B. Programme mit einer graphischen Benutzungsoberfläche und solche, die hauptsächlich Berechnungen durchführen. Aus diesen Anforderungen lassen sich zwei Arten von Komponenten definieren:

Sichtbare Komponenten repräsentieren ein Element auf der graphischen Benutzungsoberfläche. Zu ihnen lassen sich z.B. Druckknöpfe oder Eingabefelder rechnen.

Nicht-sichtbare Komponenten sind nicht auf der Benutzungsoberfläche zu sehen. Sie dienen hauptsächlich dazu, Berechnungen durchzuführen oder Mechanismen für diverse verschiedene Kommunikationsmöglichkeiten anzubieten.

Für eine sichtbare Komponente ist es in VisualWorks Smalltalk üblich, das *Modell–Ansicht–Kontroll* Konzept anzuwenden. Die Modell–Klasse (hier `AClass` genannt) dient dazu, Funktionalität für eine Komponente bereitzustellen. Die Repräsentation (z.B. ein Druckknopf) auf der Benutzungsoberfläche wird durch die *Ansicht–Klasse* sichergestellt. Eine Ausprägung dieser Ansicht–Klasse wird in der Exemplarmethode `initialize` der Komponentenschablone der Variablen

`view` zugewiesen. Wird zusätzlich noch die Verarbeitung von Benutzeraktionen benötigt, so ist es notwendig, noch eine *Kontroll–Klasse* hinzuzufügen. Diese darf aber erst in der Exemplarmethode `activate` der Komponentenschablone gesetzt werden. Eine Deaktivierung dieser Klasse muß in der entsprechenden Exemplarmethode `deactivate` vorgenommen werden.

Das Klassendiagramm einer sichtbaren Komponente kann wie in Abb. 2.6 dargestellt werden. Die Komponentenschablone und das Modell haben jeweils gegenseitig einen Verweis, damit von der Komponente Änderungen im Modell vorgenommen werden können und das Modell die Schablone über Änderungen benachrichtigen kann. Das Modell und die Komponentenschablone haben jeweils einen Verweis auf die Ansicht. Dadurch kann das Modell der Ansicht mitteilen, daß sie aktualisiert werden muß, und die Komponentenschablone weiß durch welche Klasse sie dargestellt wird. Die Ansicht hat einen Verweis auf ihre Kontrollklasse und diese besitzt wiederum einen Verweis auf das Modell, um dort Änderungen vorzunehmen.

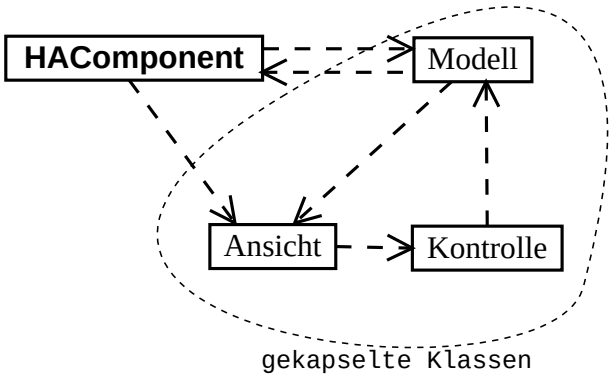

Abbildung 2.6: Klassendiagramm einer sichtbaren Komponente

Ist die Komponente nicht-sichtbar (z.B. eine Datenbankkomponente), erscheint sie nicht auf der Benutzungsoberfläche. Um beim sichtbaren Zusammenbauen der Komponenten auch die nicht-sichtbaren Komponenten besser unterscheiden zu können, ist es empfehlenswert, ein Bild für diese zu definieren. Dies kann mit der Klassenmethode `icon` der Komponentenschablone geschehen.

Damit festgestellt werden kann, ob eine Ausprägung einer Komponente sichtbar oder nicht-sichtbar ist, steht die Exemplarmethode `isVisual` der Komponentenschablone bereit.

Wird eine Komponente auf der Benutzungsoberfläche oder im Kompositionseditor plaziert, so hat sie einen Anfangspunkt. Dieser kann mit den Methoden `origin` und `origin:` angesprochen werden. Um die Abmessungen einer Komponente zu bearbeiten, stehen die Exemplarmethoden `dimension` und `dimension:` bereit. Die Koordinaten werden jeweils nach einem System berechnet, das links oben in der Ecke den Koordinatenursprung hat.

2.4.6 Komponentenspezifische Einstellungen

Jede Komponente hat spezielle Einstellungen wie z.B. den *Exemplarnamen*. Diese Einstellungen müssen beim Speichern und Laden einer Komponente bzw. eines Komponentenprogramms mit berücksichtigt werden, wie in Abschnitt 2.4.7 beschrieben. Vom Entwickler erstellte Komponenten benötigen oft zusätzliche Einstellungsmöglichkeiten, die ebenfalls mit gespeichert und geladen werden müssen.

Das HotAgent Komponentenmodell möchte das Speichern und Laden von zusätzlichen Einstellungen möglichst erleichtern. Dazu wird von der Komponentenschnittstelle `HAComponent` ein Verzeichnis (Wörterbuch) `optionsDictionary` bereitgestellt, in dem zusätzliche Einstellungen eingetragen werden können. Dafür ist es notwendig, die zu speichernden Daten mit einem Schlüsselwort zu versehen, um beim Laden wieder die richtige Zuordnung herstellen zu können.

2.4.7 XML Komponentenbeschreibung

Beim Zusammenfügen von Komponenten und Komponentenanwendungen ist es notwendig, die Konfigurationsdaten zu speichern, um sie später wieder laden zu können. Dies wird in XML [BM00] realisiert. Die verwendeten XML-Elemente ähneln dem von Birngruber [Bir01] vorgestellten Ansatz, der speziell für die Verwendung mit *Enterprise Java Beans* (s. Kapitel 2.3.4) vorgesehen ist. Die hier verwendete XML-Struktur ist auf die Bedürfnisse der HotAgent Komponenten abgestimmt.

Zur Speicherung einer Komponenten-Anwendung beginnt die XML-Beschreibung mit dem Wurzelelement `PROGRAM`. Unter diesem Element können beliebig viele Elemente für Komponenten und deren Verbindungen abgelegt werden.

Eine Komponente wird mit dem Element `COMPONENT` definiert. Die Einstellungen einer Komponente werden durch Attribute festgehalten. Der Name des Attributes entspricht dem Namen der Einstellung. Der Wert der Einstellung wird als Zeichenreihe abgelegt. In dem unten abgebildeten Beispiel ist die zweite Komponente ein `Push Button`. Als Attribute folgen die Abmessungen, der Expemplarname und der Anfangspunkt. Die Reihenfolge der Attribute ist implementierungsabhängig und ist durch die Namen der Attribute bestimmt.

```
<PROGRAM>
   <COMPONENT descriptionName="Window"
              dimensionx="425" dimensiony="451"
              name="Medical Advisory Service"
              originx="21" originy="18"></COMPONENT>
   <COMPONENT descriptionName="Push Button"
```

```
            dimensionx="80" dimensiony="32"
            name="Submit"
            originx="181" originy="420"></COMPONENT>
   ...
   <COMPONENT descriptionName="Data Base"
            dimensionx="32" dimensiony="32"
            name="MIDB"
            originx="674" originy="44"></COMPONENT>
   <CONNECTION color="#brown" entrance="text" exit="taText"
            fromComponent="control" label="set text"
            points="#(554.137@103.395 607@154 674.922@201.25)"
            toComponent="text analysis"></CONNECTION>
   <CONNECTION color="#green" entrance="query" exit="queryMIDB"
            fromComponent="control" label="query"
            points="#(554.031@82.8359 673.535@59.2497)"
            toComponent="MIDB"></CONNECTION>
   <CONNECTION color="#brown" entrance="keywords" exit="taSymptoms"
            fromComponent="control" label="setkeywords"
            points="#(554.137@103.395 597@169 674.391@210.672)"
            toComponent="text analysis"></CONNECTION>
   ...
</PROGRAM>
```

Die Verbindungen werden ähnlich notiert. Das Element `CONNECTION` und seine Attribute beschreiben jeweils eine Verbindung. Die zuerst erwähnte Verbindung im Beispiel hat die Farbe Braun und geht von der control Komponente und deren Ausgang taText zur text analysis Komponente und deren Eingang text. Das Attribut `label` gibt die Bezeichnung für die Verbindung an und `points` definiert die Koordinaten, über die die Verbindung auf der Arbeitsfläche geht.

Die XML-Beschreibung für zusammengesetzte Komponenten (s. Kapitel 6.2) und Testfälle (s. Kapitel 6.3) sieht ähnlich aus. Der Unterschied liegt darin, daß ein anderes Wurzelelement benutzt wird. Für Komponenten ist es das Element `COMPOSED_COMPONENT` und für Testfälle heißt es `TEST_CASE`.

2.5 HotAgent Komponentenmodell in verteilten ereignisbasierten Systemen

Fiege *et al.* [FMMB02] beschäftigen sich mit ereignisbasierten Systemen in Verbindung mit Komponentensystemen. Sie schränken die Sichtbarkeit von Ereignissen ein, um so Komponenten zu erstellen. In diesem Abschnitt soll die vorgestellte Methode mit dem HotAgent Komponentensystem verglichen werden,

um eine Verbindung zu anderen Arbeiten im Graduiertenkolleg „Infrastruktur für den elektronischen Markt“ darzustellen. Da das HotAgent Komponentensystem nur für die Kommunikation auf einem Computer gedacht ist, wird der Aspekt des Verteilens nicht betrachtet.

Nach Fiege *et al.* trete ein Ereignis nach einem interessanten Geschehen auf. Es könne z.B. nach der Änderung des Zustands eines Objektes oder einer Komponente auftreten. Zum Erzeugen eines Ereignisses werde ein Produzent benötigt. Ein Konsument könne ein Ereignis verarbeiten, wenn er dieses Ereignis abonniert habe. Eine Komponente werde definiert als ein Produzent und gleichzeitig als ein Konsument von Ereignissen. Durch das Verhalten von Komponenten als Produzent und Konsument könnten die Komponenten untereinander kommunizieren. Dafür spezifiziere jeder Produzent die Art des Ereignisses, und jeder Konsument gebe an, an welchen Ereignissen er Interesse habe. Die Ereignisse würden dann gefiltert und den richtigen Konsumenten zugestellt. Ein Produzent kenne dabei nicht die Konsumenten.

Beim HotAgent Komponentensystem kann auch von Ereignissen, die zwischen Komponenten verschickt werden, geredet werden. Eine Komponente dient, abhängig von ihren Ein- und Ausgängen, als Produzent und Konsument. Der Unterschied ist, daß die HotAgent Komponenten nicht die Art eines Ereignisses spezifizieren. Die Art eines Ereignisses wird durch den aktivierten Ausgang bestimmt. Anders ist auch, daß jede Komponente, die als Produzent agiert, zu jedem Ausgang ihre Konsumentenkomponenten mit den entsprechenden Eingängen kennt. In ereignisbasierten Systemen ist der Konsument unbekannt. Identisch ist, daß jeweils der Produzent ein Ereignis veranlaßt. Ebenfalls gleich ist, daß es im HotAgent Komponentenmodell und in ereignisbasierten Systemen nur eine Schnittstelle gibt, um Ereignisse zu verschicken.

Fiege *et al.* zeigen auch, wie Komponenten durch eine lose Koppelung zu neuen Komponenten zusammengesetzt werden könnten. Die so entstandenen Komponenten seien in ihrer Funktion leistungsfähiger. Dazu wird der Begriff *Sichtbarkeitsbereich* (engl. *scope*) als „*an abstraction that bundles a set of producers and consumers*“ eingeführt. Der Sichtbarkeitsbereich ermögliche hierarchische, ereignisbasierte Systeme. Dieser bündele Komponenten entweder nach ihrer Anwendungsstruktur oder nach ihren Aufgaben. Ereignisse werden nur in diesem Sichtbarkeitsbereich verschickt. Wenn sich Produzent und Konsument sehen, und wenn der Konsument sich für die Art der Ereignisse interessiere, werden Ereignisse ausgetauscht. Damit die so zusammengesetzten Komponenten weiterverwendet werden können, müssen diese eine einheitliche Schnittstelle bieten, die wieder eine Verwendung als Produzent und Konsument gewährleiste.

Im HotAgent Komponentensystem lassen sich auch Komponenten zu neuen Komponenten zusammenbauen. Die neu entstandenen Komponenten kapseln die enthaltenen Komponenten komplett ab, d.h. die enthaltenen Komponenten haben

keine Möglichkeit, außerhalb liegende Komponenten zu sehen. Die Kommunikation geschieht auch wieder durch die zuvor definierten Ein- und Ausgänge.

Die vorangegangenen Erläuterungen machen deutlich, daß das von Fiege *et al.* vorgestellte ereignisbasierte System durchaus als Grundlage des HotAgent Komponentenmodells genutzt werden kann. Die Ein- und Ausgänge lassen sich problemlos auf die Produzenten und Konsumenten abbilden. Nur lassen sich im HotAgent Komponentensystem die Art der Ereignisse nicht spezifizieren. Dies ist aber möglich, wenn der Name der Produzentenkomponenten mit dem Ausgangsnamen als Art des Ereignisses angegeben wird. Die Konsumenten werden dann als Interesse diese beiden Namen angeben. Im HotAgent Komponentenmodell werden normalerweise im Produzenten Verweise auf den Konsumenten gespeichert. Da in ereignisbasierten Systemen der Konsument unbekannt ist, spezifiziert der Konsument, wie zuvor beschrieben, sein Interesse. Bei den zusammengesetzten Komponenten gibt es keine großen Unterschiede, und sie lassen sich analog übernehmen.

Kapitel 3

Visuell–unterstützter Software–Entwurf

Nahezu jede Entwicklungsumgebung bietet heute eine visuelle Unterstützung für den Entwickler an. Egal, ob es sich um Werkzeuge handelt, die dem Entwickler das Schreiben von Quelltext erleichtern, oder ob eine visuelle Programmiersprache angeboten wird. In diesem Kapitel soll der Schwerpunkt hauptsächlich auf visuelle Programmiersprachen gelegt werden.

In Abschnitt 3.1 wird definiert, was unter visuell unterstütztem Software–Entwurf zu verstehen ist, und in Abschnitt 3.2 wird erläutert, welche Kriterien eine visuelle Programmier-Umgebung erfüllen muß.

3.1 Allgemein

Von Burnett *et al.* [BGL95] wird erklärt, was ein visuell unterstützter Software–Entwurf ist. Danach gibt es zwei Möglichkeiten, die sich aber nicht gegenseitig ausschließen, sie werden sogar oft in einer Entwicklungsumgebung integriert:

Visuelle Programmiersprachen: Sprachen mit *visueller Syntax*.

Visuelle Werkzeuge: Werkzeuge, die die Möglichkeit bieten, bildlich dargestellte Programmelemente zu manipulieren. Dabei könne es sich aber auch um textuelle Sprachen handeln, die dargestellt werden.

Shu [Shu88] definiert den Begriff der *visuellen Programmierung*:

> The use of meaningful graphic representations in the process of programming.

Im Programmierprozeß spielen graphische Elemente und Bilder eine Hauptrolle. Sie werden in einer speziellen Art und Weise angeordnet, um eine Beziehung zu dem jeweiligen Kontext zu zeigen.

Um visuell zu programmieren, ist eine *visuelle Sprache* notwendig. Schiffer [Sch98] definiert, daß eine visuelle Sprache eine *visuelle Syntax* und eine *visuelle Semantik* habe. Ein Symbol der Sprache sei ein Text, ein Bild oder eine Kombination aus beiden. Ein visuelles, syntaktisches Konstrukt zwischen zwei Symbolen sei z.B. ein Pfeil oder vor einem Symbol ein anderes graphisches Objekt. Ein semantisches Konstrukt sei z.B. die Farbe eines Symbols oder eines anderen syntaktischen Elements. Dadurch könne z.B. ein Zustand widergespiegelt werden.

Bilder bzw. Piktogramme können nach Lodding [Lod83] und Arnheim [Arn69] in drei Arten aufgeteilt werden:

Representationelle Piktogramme seien Bilder, die eine vereinfachte Darstellung von realen Objekten zeigen (z.B. eine Tanksäule).

Abstrakte Piktogramme, Sinnbilder, repräsentierten ein Konzept oder einen Begriff mit konkreten Objekten (z.B. für zerbrechlich).

Arbiträre Piktogramme seien speziell eingeführte Schilder, denen eine Bedeutung zugewiesen sei (z.B. radioaktiv).

Bei der Erstellung von Piktogrammen ist es nach Lodding [Lod83] wichtig, daß sie zum Programmieren geeignet konstruiert sind. Solche Piktogramme könnten die Arbeit sehr erleichtern. Sie müßten eine einfache Gestalt haben. Werden mehrere Objekte angeordnet, müßten sie ordentlich gruppiert sein und klar Überschneidungen zeigen. Außerdem sei es wichtig, daß die Objekte deutlich vom Hintergrund abgesetzt seien. Zu beachten sei auch, daß die Interpretation von Bildern kontextabhängig ist.

Nach der Klärung visuell unterstützter Programmierung kann nun der Frage nachgegangen werden, ob visuelle Programmierung besser als traditionelle textuelle Programmierung ist. Sie wird von Shu [Shu88] durch folgende Aussagen positiv beantwortet:

- Bilder seien aussagekräftiger als Text
- Bilder helfen beim Verstehen und Behalten
- Bilder geben einen Anreiz für das Lernen des Programmierens
- Bilder verstehen alle, unabhängig von ihrer Muttersprache

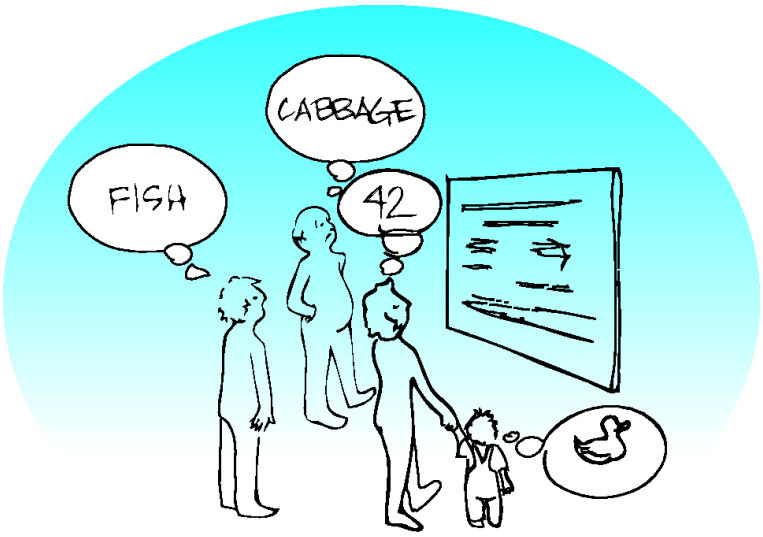

Abbildung 3.1: Does a given picture convey the same thousand words to all viewers? (aus Petre [Pet95] Seite 34)

Die Aussagen von Shu werden aber auch kritisch betrachtet, z.B. argumentiert Petre [Pet95], daß die Frage nicht sei 'Is a picture worth a thousand words?', sondern 'Does a given picture convey the same thousand words to all viewers?'. Er sagt, ein Bild sei nicht eindeutig und verdeutlicht dies an Abbildung 3.1. Jeder sieht etwas anderes. Speziell wenn verschiedene Nationalitäten einbezogen würden, könne ein Bild unterschiedlich interpretiert werden. Ein gutes Bild basiere auf *sekundärer Notation*. Sekundäre Notation stellt zusätzliche Information bereit, die nicht zu der Syntax eines Programms gehören, wie z.B. Farbe oder Positionen. Dabei ist aber zu beachten, daß dies unerfahrene Benutzer verwirren kann, erfahrene profitieren aber davon.

Lodding [Lod83] begründet den Vorteil von visueller Programmierung durch die Schnelligkeit von Bildverarbeitung und Bildwiedererkennung. Winkler [Win90] sagt, daß Hauptanliegen sei, den Entwurfsprozeß zu vereinfachen, so daß auch Nichtprogrammierer oder Anfänger leichteren Zugang finden. Ein graphisches Programm nehme aber manchmal mehr Platz ein als ein textuelles, da eine Graphik weniger formal sei als Text. Für das Programmieren im Großen eigne sich eher eine graphische Lösung, wobei Programmieren im Kleinen besser textuell geschehe. Gut sei es, also beide Möglichkeiten zu kombinieren.

Meiner Meinung nach, wird die visuelle Programmierung durch das Anwenden der Komponententechnik wieder interessant. Nach Winkler [Win90] eignet sich die visuelle Programmierung besonders für das Programmieren im Großen. Komponenten haben einen *klaren Aufgabenbereich* und bilden dadurch große Funktionseinheiten, die leicht zusammengesetzt werden können. Durch die Möglichkeit einer hierarchischen Strukturierung eines Komponentenprogramms kann ein vi-

suelles Programm bzw. Teilprogramm stets übersichtlich gehalten werden. Wird eine *sekundäre Notation* (s. Petre [Pet95]) ergänzt, könne zusätzlich Übersichtlichkeit erreicht werden.

3.2 Kriterien für visuelle Entw.-Umgebungen

Nun sollen Kriterien aufgestellt werden, die eine visuelle Entwicklungsumgebung unterstützen sollen. Diese sollen nur auf eine Entwicklungsumgebung bezogen werden und nicht auf das darunterliegende Komponentenmodell. Zur Absicherung der Richtigkeit der Kriterien werden Literaturstellen zitiert, die die einzelnen Kriterien untermauern sollen. Die aufgeführten Kriterien können nicht als die einzig gültigen angesehen werden, meines Erachtens aber die wichtigsten. Zu jedem Kriterium wird ein Beispiel angegeben, welche Folgen das Einhalten bzw. Nicht-Einhalten hat.

Um gute visuelle Entwicklungs-Umgebungen zu erstellen, definieren Green und Petre [GP96] sieben Merkmale:

Nähe der Abbildung (V1): Der Abstand (s. [GP96]) zwischen der abzubildenden Problemwelt und der Programmiersprache werde als Nähe der Abbildung bezeichnet. Je besser ein Problem in einer Programmiersprache abgebildet werden könne, desto leichter und übersichtlicher werden die resultierenden Programme. Green und Petre schlagen deshalb eine auf den Anwendungsbereich spezialisierte Programmiersprache vor.

+ Durch eine geeignete Abbildung wird dem Programmierer das Erstellen von Programmen deutlich einfacher gemacht. Er findet z.B. Piktogramme in der Sprache, die seinen Anwendungsbereich verdeutlichen.

- Wird versucht, eine textbasierte Programmiersprache durch Puzzle-Teile (s. [Mar00]) abzubilden, ist schnell festzustellen, daß die Teile zu unflexibel sind und so keine Programme konstruierbar sind oder daß sie zu groß und unübersichtlich werden.

Viskosität (V2): Die Viskosität (s. [GP96]) bezeichne den Aufwand, der notwendig ist, um kleine Programmänderungen durchzuführen. Bei visuellen Programmiersprachen sei damit auch das Umordnen von Elementen gemeint, damit neue eingefügt werden können oder um ein Programm übersichtlicher zu machen.

+ Durch die *Viskosität* ist es möglich, mit nur wenigen Handgriffen Änderungen an einem Programm durchführen.

- Wenn nur geringe Änderungen an einem Programm vorgenommen werden müssen, und das z.B. strukturelle Veränderungen mit sich bringt, ist ein großer Aufwand notwendig, dadurch besteht die Gefahr neuer Mängel.

Versteckte Abhängigkeiten (V3): Wenn implizite Verbindungen zwischen Programmteilen bestehen (s. [GP96]) und für den fehlerfreien Ablauf eines Programms notwendig sind, werde von versteckten Abhängigkeiten gesprochen. Werden diese Abhängigkeiten nicht klar dargestellt, können bei Änderungen unvorhersehbare Fehler auftreten. Meyer [Mey90] spricht dabei auch von *expliziten Schnittstellen*.

+ Sind alle Abhängigkeiten dargestellt, ist dem Entwickler klar, welche Folgen eine Änderung einzelner Teile des Programms hat.
- Sind einige Abhängigkeiten nicht dargestellt, kann der Entwickler nicht feststellen, was eine Änderung im Komponenten-Programm bewirken kann. Im schlimmsten Fall kann ein Programm nicht mehr ausgeführt werden.

Schwere mentale Operationen (V4): Sachverhalte, die komplex dargestellt werden müssen (s. [GP96]), seien zu vermeiden. Es muß versucht werden, Aufgaben einfach und verständlich darzustellen.

+ Werden Aufgaben einfach abgebildet, kann ein Programm leicht verstanden und verändert werde.
- Ist eine Aufgabe unübersichtlich abgebildet, ist es für Dritte schwer zu erkennen, wo eventuelle Änderungen gemacht werden können und welche Auswirkungen diese haben.

Erzwungenes Vorausdenken (V5): Die Problematik, inwieweit vorausgeplant und vorab Entscheidungen getroffen werden müssen, werde als erzwungenes Vorausdenken bezeichnet (s. [GP96]). Dies werde vor allem durch implizite Abhängigkeiten bewirkt. Bei visueller Programmierung sei ein gewisses Vorausdenken notwendig, um nicht ständig die Programme neu ordnen zu müssen.

+ Erlaubt ein Programmiersystem Änderungen leicht und schnell durchzuführen, ist es nicht nötig, daß ein Entwickler eine Aufgabe vollständig berücksichtigt. Eventuelle Änderungen kann er zu einem späteren Zeitpunkt vornehmen.
- Sind Änderungen schlecht möglich, hat ein Entwickler alle möglichen Anforderungen vor dem Erstellen einer Anwendung zu bedenken.

Sekundäre Notation (V6): Zusätzliche Information (s. [GP96]), die nicht zu der Syntax eines Programms gehören, zählen zur sekundären Notation. Sie sollen die Lesbarkeit von Programmen erhöhen. Beim visuellen Programmieren könne schon das Gruppieren von Elementen oder das Einhalten von Konventionen dazu zählen.

+ Eine Anwendung läßt sich übersichtlicher gestalten, wenn z.B. eine einheitliche Farbe für ähnliche Verbindungen gewählt wird. Gibt es Überschneidungen von verschiedenfarbigen Verbindungen, so können diese trotzdem auseinander gehalten werden.

- Sind z.B. alle Verbindungen in einem Programm mit der gleichen Farbe dargestellt, wird es bei Überschneidungen schnell unübersichtlich.

Sichtbarkeit (V7): Sie zeige (s. [GP96]), wie leicht ein Programm oder ein Teil eines Programms erfaßt werden kann. Da heute Anwendungen häufig großen Umfang haben, sei es besonders wichtig, daß Programme leicht untersucht und Teile gesondert betrachtet werden können. Dabei solle es auch möglich sein, Teile nebeneinander anzuzeigen.

+ Können von einer großen Anwendung kleine Teile betrachtet werden, so läßt sich ein Programm leichter analysieren.

- Ist es nur möglich, ein Programm in der vollen Größe zu betrachten, ist es oft schwer, einen Überblick, speziell auch über Einzelheiten, zu erlangen.

Von Giesl [Gie01] werden die von Green und Petre aufgestellten Merkmale speziell für visuelle komponentenorientierte Entwicklungsumgebungen durch die folgenden Punkte ergänzt:

Komponentenentwicklung visuell und textuell (V8): Da visuelle Programmierung manchmal viel Platz beansprucht (s. [Gie01]), solle es möglich sein, Komponenten auch textuell zu erstellen. Auf der anderen Seite sollen auch Komponenten aus anderen Komponenten visuell erstellt werden können, um eine gute Gliederung zu ermöglichen. Viele Entwicklungsumgebungen bieten diese Möglichkeit leider nur für visuelle Komponenten an. Winkler [Win90] sagt, daß für das Programmieren im Großen sich eher eine graphische Lösung eigne, wobei das Programmieren im Kleinen besser textuell geschehe.

+ Je nach Art eines Problems ist es manchmal leichter etwas textuell auszudrücken. Deswegen sollte eine visuelle Entwicklungsumgebung auch eine textuelle Programmierung erlauben.

- Wird versucht, komplizierte Schleifen oder Bedingungen durch visuelle Programmierung auszudrücken, wird ein Programm oft unübersichtlich.

Komponenten- und Anwendungsentwicklung (V9): Eine Entwicklungsumgebung müsse die beiden Teilgebiete Komponenten- und Anwendungsentwicklung unterstützen können (s. [Gie01]). Die Aufgabenbereiche lassen sich nicht vollständig getrennt betrachten, sondern der Entwickler solle für beides Entwicklungswerkzeuge vorfinden. Bei textueller Entwicklung von Komponenten solle der Programmierer auch unterstützt werden, um z.B. durch Schlüsselwörter, Ein- und Ausgänge zu erstellen.

+ Wird die Komponenten- und Anwendungsentwicklung in einer Umgebung unterstützt, kann ein Entwickler bei Bedarf neue Komponenten entwerfen, wenn er feststellt, daß ihm eine spezielle fehlt.
- Sind für Komponenten- und Anwendungsentwicklung zwei getrennte Programme notwendig, muß sich ein Entwickler mit zwei verschiedenen Programmen auskennen, wenn er fehlende Komponenten selbst erstellen möchte.

Dokumentationen direkt verfügbar (V10): Das Kriterium der Selbstbeschreibung (*K2*) von Komponenten nutze dem Entwickler wenig (s. [Gie01]), wenn die Entwicklungsumgebung nicht in der Lage sei, sie geeignet zu präsentieren. Dabei seien unterschiedliche Arten von Informationen an verschiedenen Stellen im Entwicklungsprozeß interessant.

+ Bietet eine Entwicklungsumgebung jederzeit eine passende Dokumentation, so muß der Entwickler nicht nach hilfreichen Erläuterungen suchen.
- Ist eine Dokumentation nur separat verfügbar, benötigt ein Entwickler mehr Zeit, um passende Hinweise zu finden.

Testfunktionalität (V11): Testmöglichkeiten seien wichtig (s. [Gie01]), da der Entwicklungsprozeß kontinuierlich geprüft werden muß. Gerade bei komponentenorientierter Programmierung sei es notwendig, daß eine Entwicklungsumgebung eine geeignete Testumgebung bereitstellt. Selbst erstellte Komponenten müssen ausreichend getestet werden, damit eine sichere Wiederverwendung gewährleistet werden kann (s. Kapitel 4).

+ Unterstützt eine Entwicklungsumgebung das Erstellen von Tests, hat ein Entwickler wenig Aufwand, Mängel zu finden und zu vermeiden.
- Wenn keine Tests möglich sind, kann auch nicht geprüft werden, ob ein Programmstück mangelfrei ist.

Programmablaufvisualisierung (V12): Um den tatsächlichen Ablauf einer Anwendung darzustellen, könne die Programmablaufvisualisierung genutzt werden (s. [Gie01]). Visuelles Programmieren mit Komponenten biete schon eine statische Visualisierung des Programms. Es empfehle sich, diese Darstellung so zu erweitern, daß eine dynamische Visualisierung entsteht (s. Kapitel 4).

+ Durch die Programmvisualisierung können leicht Mängel in einem Programm anhand eines Anwendungsfalls aufgespürt werden. Die Ausführung eines Programm kann übersichtlich dargestellt werden.
- Wird zum Finden eines Mangels ein Debugger benutzt, ist das Lokalisieren des Mangels oft erschwert.

Die zuvor genannten Kriterien werden von mir durch folgenden Punkt ergänzt:

Einheitliches Aussehen (V13): Besteht eine Entwicklungsumgebung aus mehreren unterschiedlichen Werkzeugen, ist es wichtig, daß alle ein einheitliches Aussehen und gleiche Bedienung bieten.

+ Eine einheitliche Benutzungsoberfläche erlaubt ein schnelles Arbeiten mit unterschiedlichen Werkzeugen, auch wenn diese nicht oft benutzt werden.
- Wenn jedes Werkzeug unterschiedlich zu bedienen ist, muß sich der Entwickler in immer neue Programme einarbeiten und benötigt somit mehr Zeit.

Im vorangegangenen Text wurden einige Kriterien für eine visuelle Entwicklungsumgebung und auch im speziellen für eine visuelle komponentenbasierte Entwicklungsumgebung aufgeführt. In dieser Arbeit werden verschiedene Entwicklungsumgebungen betrachtet, die an Hand der oben genannten Kriterien untersucht werden.

Kapitel 4

Testen und Ablaufverfolgung von Komponenten

Die Bedeutung von *Tests* in der Softwareentwicklung wird oft verkannt. Meyer *et al.* [MMS98] schreiben, daß die Industrie nicht viel Wert auf Tests lege, da es zu teuer sei. Dabei verließen sie sich aber auf wiederverwertbare Komponenten, die durch Qualität und Verläßlichkeit ausgezeichnet seien, aber dies ohne Testen. Beck und Gamma [BG98] meinen, je weniger Tests gemacht werden, um so weniger produktiv sei man, und um so weniger stabil sei der erstellte Quelltext. Ihr Motto ist: „codiere ein wenig, teste ein wenig“.

Beck [Bec00] sagt, daß nur *die* Software existiere, die sich auch messen lasse. Tests z.B. zeigten, daß auch wirklich das implementiert sei, was implementiert werden solle. Er geht soweit und sagt, erst wenn einem keine Tests mehr einfielen, dann könne man davon reden, daß etwas fertig implementiert sei. Dijkstra [Dij72] sieht das anders, daß Tests nur genutzt werden können, um die Anwesenheit von Fehlern zu zeigen, aber nicht, um deren Abwesenheit zu zeigen.

Im *Extreme Programming* [Bec00] ist das Testen ein fester Bestandteil des Software-Entwicklungszyklus. So darf es auch beim Entwickeln von Komponenten nicht fehlen. Wieviel getestet wird, bleibt dem Entwickler überlassen.

In vielen Fällen kann die *Ablaufverfolgung* auch beim Testen helfen. Sie wird aber selten bei der Entwicklung von Software eingesetzt. Ein Hauptbestandteil der Ablaufverfolgung ist die Programm-Visualisierung, die dazu dient, das Laufzeitverhalten eines Programms darzustellen.

Wiggins [Wig98] erwähnt, daß es schon lange diverse Techniken zur Visualisierung gebe. Dazu zählen u.a. Flowcharts, Nassi-Shneiderman Diagramme und Pretty-Printing. Zu diesen Techniken existieren auch jeweils Werkzeuge zur automatischen Erstellung solcher Diagramme, was auch Winkler [Win90] bestätigt.

In dem folgenden Abschnitt wird zuerst allgemein über das Testen gesprochen. Darauf folgt in Abschnitt 4.2 eine Betrachtung von Möglichkeiten der Ablaufverfolgung. Anschließend werden in Abschnitt 4.3 Kriterien, die aus diversen Arbeiten erarbeitet wurden, vorgestellt, die beim Testen und bei der Ablaufverfolgung beachtet werden müssen.

4.1 Testen allgemein

Wie zuvor erwähnt, ist das Testen ein wichtiger Bestandteil in der Softwareentwicklung. Beck [Bec00] definiert zwei verschiedene Arten von Tests:

Komponententests seien dazu da, die Schnittstelle und die Funktionalität von Programmteilen zu testen. (Der Begriff Komponente hat hier nichts mit der Definition aus Kapitel 2.1 zu tun.) Weyuker [Wey01] nennt diese Art von Tests auch *Integrationstests*. Andere gebräuchliche Begriffe sind auch *Modultest* oder *Systemtest*. Komponententests werden vorwiegend von den Entwicklern oder einem Testteam vorgenommen.

Funktionstests dienen dazu, die Anforderung an ein Softwareprogramm zu prüfen. Funktionstests werden in der Regel vom Kunden definiert, die aber dann oft ein spezielles Testteam umsetzt, da der Kunde meist nicht genug Wissen habe, solche Tests durchzuführen.

Beck [Bec94] definiert auch zwei Arten von Fehlern:

- Ein *Mangel* (engl. failure) sei ein vorhersehbares Problem. Wenn Tests geschrieben werden, werde getestet, ob ein erwartetes Ergebnis erscheint. Sei dies nicht der Fall, so gebe es einen Mangel.
- Ein *Fehler* (engl. error) sei gravierender. Diese Art von Fehlern könne nicht vorausgesehen werden, und sie sei auch nicht testbar.

Im folgenden werden nur noch Mängel betrachtet, die gesucht werden können. Der Tester definiert Eingabewerte, nach denen er ein erwartetes Ergebnis errechnet. Anhand dieser Werte erstellt der Tester einen Anweisungsblock, um mögliche Mängel zu finden. Dieser Block wird Testfall (engl. test fixture oder test case) genannt.

McCarthy [McC97] definiert den Begriff *Einheitstest* (engl. unit test). Dies sei das Testen einer Funktion, eines Moduls oder eines Objekts getrennt vom Rest des Programms. Normalerweise sei eine Einheit der kleinste Teil eines Programms.

In objekt-orientierter Programmierung sei die kleinste Einheit die Klasse. In der Komponententechnologie ist es eine Komponente, die als schwarze Kiste (engl. black box) betrachtet wird. In die Komponenten kann nicht hineingeschaut werden, um z.B. die einzelnen enthaltenen Klassen zu testen. Sie kann nur als ganzes an den dafür vorgesehenen Schnittstellen getestet werden.

Atkinson *et al.* [ABB+02] definieren drei Schritte beim Testen. Zuerst müssen Testfälle aufgestellt werden. Dies geschehe nach Beck und Gamma [BG98] in idealer Weise direkt neben der Implementierung eines Programmteils. Darauf müssen alle Tests ausgeführt werden. Die letzte Aufgabe bestehe darin, die Ergebnisse der Tests zu analysieren und dabei mögliche Mängel zu finden.

Beim Testen ist es wichtig, daß einmal definierte Tests immer wieder ausgeführt werden können. Das wird *Regressionstest* genannt. Der Sinn von Regressionstests ist, daß nach Änderungen an dem Quelltext sofort getestet werden kann, ob er noch mangelfrei funktioniert bzw., daß das Verhalten nicht geändert wurde. Pauli [Pau01] fügt hinzu: „Test everything you can, and test it often“.

4.2 Ablaufverfolgung allgemein

Wenn Softwaresysteme umfangreicher sind oder von unterschiedlichen Entwicklergruppen erstellt werden, kann es schwierig sein, einen Mangel zu finden. Dazu kann die Ablaufverfolgung von Programmen helfen. Sie läßt sich in zwei Aufgabenbereiche gliedern. In die Analyse (s. Abschnitt 4.2.1) von Programmen und die Visualisierung (s. Abschnitt 4.2.2) der zuvor gewonnenen Daten.

4.2.1 Analyse

Es gibt zwei Arten von Analysen. Die statische Analyse analysiert nur den Quelltext eines Programms. Die dynamische Analyse geschieht während der Ausführung eines Programms. Es wird also mindestens ein ausführbares Programm benötigt, welches aber nicht fehlerfrei sein muß. Die statische Analyse ist auch bei Programmen möglich, die nicht übersetzt werden können, was bei der dynamischen nicht möglich ist. In dieser Arbeit wird auf die dynamische Analyse der Schwerpunkt gelegt.

Da die dynamische Analyse während der Ausführung eines Programms geschieht, ist es wichtig, daß die Analyse das Programm nicht wesentlich in den Laufzeiteigenschaften beeinflußt. Sonst könnte das Verhalten des beobachteten Programms verändert werden. Reiss [Rei97] fordert deswegen eine schnelle, unkomplizierte Technik, um Daten während der Ausführung zu sammeln.

Nach Gao *et al.* [GZS01] setzt die Analyse voraus, daß die Objekte bzw. Komponenten eines Programms verfolgt (engl. track) werden können. Um das Verhalten festzustellen, sei es wichtig, die Eingabedaten und die Ergebnisse festzuhalten. Dadurch könne jederzeit gesagt werden, in welchem Zustand sich ein Objekt oder eine Komponente befindet oder befand. Anhand dieser Daten könne eine Spur (engl. trace) aufgezeichnet werden, die die Interaktion an den Schnittstellen einer Komponente oder zwischen Komponenten darstellt.

4.2.2 Visualisierung

Ist nun die Analyse eines Programms abgeschlossen, ist es wichtig, daß die festgehaltene Spur eines Programms geeignet visualisiert wird.

Myers [Mye90] stellt zwei Fragen zur Visualisierung:

- Was der *Visualisierungsbereich* sei, welche Art von Informationen visualisiert werden sollen, ob es z.B. Quelltexte, Daten oder Algorithmen seien.
- Welche *Art von Visualisierung* gefragt sei, eine statische oder dynamische. Entwicklungsumgebungen, in denen Programme visuell konstruiert werden, bieten von Grund auf bereits eine statische Visualisierung.

Visualisierung darf aber nicht mit visueller Programmierung vermischt werden. Bei visueller Programmierung wird ein Programm visuell erstellt, wobei die Programmvisualisierung die Darstellung eines Programms bedeutet, egal wie es erstellt wurde. Visualisierung ist nützlich, um Programme zu verstehen. Softwareentwickler, die ein Programm nicht kennen, können damit schneller und besser einen Überblick über die Funktionalität eines Programms erlangen. Speziell in objektorientierten oder komponentenbasierten Programmen ist es oft schwer, die Zusammenhänge zu verstehen, wenn nur der Quelltext zur Verfügung steht. UML wird oft zur Visualisierung von Programmen benutzt, wobei aber die Diagramme nicht mit dem wirklichen Quelltext oder Laufzeitverhalten des Programms übereinstimmen müssen. Das Ziel der Visualisierung ist es, das aktuelle Verhalten eines Programms graphisch zu beschreiben. Wichtig ist dabei, daß die Darstellung klar und einfach ist. Die Informationen müssen abstrakt dargestellt werden und dürfen nicht mit Details überladen sein. Souder *et al.* [SMS01] fügt hinzu, daß verschiedene Sichten nützlich seien. Dabei könne jede Sicht unterschiedliche Informationen klar darstellen, um dadurch einen guten Überblick von einem Programm zu erhalten.

Die größten Probleme der aktuellen Visualisierungswerkzeuge seien nach Reiss [Rei97], daß sie schwierig zu benutzen seien und daß sie nicht in der Lage seien,

große Datenmengen zu verarbeiten bzw. für den Menschen geeignet zu präsentieren. Um das zu lösen, müsse so ein Werkzeug Filtermechanismen anbieten, um die Datenmengen zu reduzieren. Für eine Visualisierung müsse ein geeigneter Mittelweg gefunden werden, um sie nicht zu kompliziert zu machen, aber dennoch genügend Informationen anzubieten.

Eine Visualisierung kann oft als Datenflußsystem angesehen werden. Komponenten oder andere Softwarekonstrukte, die Daten abgeben, können als Datenquellen betrachtet werden. Komponenten, die Daten erwarten, sind entsprechend Datensenken. Die Beziehungen, d.h. die Kommunikation zwischen den Quellen und Senken, können somit als gerichteter Graph abgebildet werden.

4.3 Kriterien für Testen und Ablaufverfolgung

Nachdem zuvor das Testen und die Ablaufverfolgung angesprochen wurde, werden nun Kriterien aufgeführt, die bei der Erstellung solcher Werkzeuge notwendig sind. Da ich dafür keine Zusammenstellung von Kriterien fand, werde ich im folgenden, unter Bezug auf Literaturstellen, wichtige zusammentragen. Die aufgeführten Kriterien können nicht als die einzig gültigen angesehen werden. Zu jedem Kriterium wird ein Beispiel angegeben, welche Folgen das Einhalten bzw. Nicht-Einhalten hat.

Schwarze Kiste Tests (T1): Diese Tests (engl. black box test) sind anwendbar für Komponentensoftware. Die kleinste Einheit zum Testen bei der Komponententechnologie ist die Komponente. Es ist nicht notwendig, die Bestandteile einer Komponente einzeln zu testen, sondern es ist besser, eine Komponente als ganze zu betrachten, wie sie auch später eingesetzt wird. Demeyer *et al.* [DDN03] sagen, es sei wichtig, die Schnittstelle zu Testen, nicht die Implementation.

Da Komponenten Blöcke mit Ein- und Ausgängen bilden, ist es praktikabel, sie mit einer Kombination aus Kontroll- und Datenflußtests zu verifizieren. Eine Nachricht oder Daten können in einen Eingang gegeben werden, und die an den Ausgängen bereitgestellten Daten können getestet werden.

+ Ein Testfall darf nur die Schnittstelle berücksichtigen. So kann sichergestellt werden, daß eine Komponente genau das erfüllt, wozu sie spezifiziert wurde.

- Wird anhand der Implementation eine Komponente getestet, muß u.U. der Testfall neu erstellt werden, wenn an der Implementation etwas geändert wird.

Normale Schnittstelle (T2): Die normale Komponentenschnittstelle muß ausreichen, um aussagekräftige Tests zu erstellen. Sobald spezielle Schnittstellen zum Setzen von Zuständen, wie bei dem Component+ [Com01] Projekt benutzt werden, sind die Tests nicht mehr repräsentativ.

+ Wird die *normale Schnittstelle* genutzt, kann nur mit der Schnittstelle getestet werden, mit der eine Komponente später eingesetzt wird.
- Durch speziell zum Testen vorgesehene Mechanismen der Komponente, können zusätzlich Mängel eingebaut werden bzw. andere verborgen werden.

Benötigte Funktionalität testen (T3): Ein Komponentensoftware-Entwickler solle nur die benötigte Funktionalität testen, beschreiben Harrold *et al.* [HLS99]. Zusätzliche Aufrufe, von sonst unbenutzter Funktionalität, können Variable setzen, die anderenfalls uninitialisiert sind und einen Mangel verursachen. Beck [Bec00] nennt diese Tests Komponententests.

Der Komponentenentwickler muß hingegen die gesamte Komponente testen. Er muß dadurch sicherstellen, daß sie in allen möglichen Einsatzgebieten einwandfrei funktioniert. Der Komponentenentwickler weiß nicht, für welchen genauen Zweck ein Anwendungsentwickler seine Komponenten einsetzen möchte. Durch seine Tests muß er zeigen, daß die Schnittstelle der Komponente der Spezifikation entspricht. Beck nennt diese Funktionstests.

+ Testet der Software-Entwickler nur die benötigte Funktionalität, kann er prüfen, ob eine Komponente genau die Aufgabe erfüllt, die er auch in seiner Anwendung benötigt.
- Nach Harrold *et al.* [HLS99] kann nicht sichergestellt werde, daß eine Komponente in ihrem speziellen Kontext einwandfrei funktioniert, wenn mehr als die benötigte Funktionalität getestet wird.

Verklemmungen/Zeitverhalten (T4): Das Finden von Verklemmungen (engl. Deadlock) und das Prüfen des Zeitverhaltens einer oder mehrerer Komponenten ist eine wichtige Aufgabe eines Testwerkzeugs. Im Component+ [Com01] Projekt wird erwähnt, daß Verklemmungen in einzelnen Komponenten unwahrscheinlich seien. Sie kommen eher im Zusammenspiel mit mehreren Komponenten vor.

Handelt es sich bei einer Komponentenanwendung um ein Echtzeitsystem, ist es notwendig, die benötigten Komponenten nicht nur auf die Funktionalität zu testen, sondern auch auf ihr Zeitverhalten zu untersuchen.

+ Das Zeitverhalten einer Komponente ist wichtig, wenn es darum geht, viele gleiche Operationen durchzuführen. Mitunter kann man so zwischen unterschiedlichen Implementierungen die beste herausfinden.

- Wird nicht auf Verklemmungen geprüft, kann es sein, daß eine Anwendung in manchen Fällen nicht weiterarbeiten kann. Diese Fälle sind dann oft schwer zu identifizieren.

Testspezifikation (T5): Die Testspezifikation soll nach Morris *et al.* [MLP+01] sprachunabhängig und leicht zu lesen sein. Vorteilhaft sei es, wenn nicht nur die Testsoftware die Spezifikation versteht, sondern daß sie auch für den Entwickler verständlich ist. Es solle möglich sein, mehrere Testfälle für eine Komponente zu spezifizieren. Cox und Song [CS01] fordern, daß Testfälle mit jeweils den entsprechenden Komponenten mitgeliefert werden sollen.

+ Wenn mehrere Testfälle angegeben werden können, ist es möglich, in kleinen, verständlichen Tests eine Komponente zu testen.
- Werden keine Testfälle mit einer Komponente ausgeliefert, ist es für einen Entwickler schwerer, eigene Testfälle zu entwickeln, um die benötigte Funktionalität zu testen.

Regressionstests (T6): Das wiederholte Testen der gleichen Testfälle, z.B. nach Änderungen an Quelltexten, wird Regressionstest genannt. Siepmann und Newton [SN94] legen auf Regressionstests großen Wert. Wenn der Quelltext einer Komponente geändert werde, sei es notwendig zu testen, ob sich das Verhalten nicht geändert habe, bzw. ob Mängel eliminiert seien und daß sich keine neuen eingeschlichen haben. Beck und Gamma [BG98] bestärken dies, in dem sie sagen, daß Tests am Laufen bleiben müßten. Morris *et al.* [MLP+01] fügen hinzu, daß Regressionstests leicht und effizient durchführbar sein müßten, damit der Entwickler nicht die Lust am Testen verliert. Beck [Bec00] fordert, daß Regressionstests automatisiert werden müßten und nur eine positive bzw. negative Rückmeldung liefern dürften, ob sie erfolgreich waren.

+ Durch Regressionstests kann jederzeit die Korrektheit einer Ansammlung von Komponenten sichergestellt werden.
- Werden keine Regressionstests durchgeführt, können sich leicht schon bei kleinen Änderungen Fehler einschleichen.

Filtermechanismen (T7): Ein Ablaufverfolgungswerkzeug müsse Filtermechanismen anbieten, fordern Souder *et al.* [SMS01]. Dadurch sei es möglich, die großen Datenmengen, die bei der Analyse anfielen, zu reduzieren. Ein Werkzeug solle Regeln anbieten, um Daten auszuschließen bzw. wieder einzubinden.

+ Durch Filtern von Daten, kann eine klarere Ansicht eines Programms erzielt werden.

- Werden keine Daten gefiltert, so kann die Erstellung von Visualisierungen bei großen Datenmengen zeitintensiv und u.U. problematisch werden.

Suchmöglichkeiten (T8): Nach Bassil und Keller [BK01] sind Suchmöglichkeiten für graphische und textuelle Elemente in einem Visualisierungswerkzeug notwendig. Sie machten in Unternehmen eine Untersuchung, die zeigte welche Kriterien für ein Visualisierungswerkzeug wichtig seien. Suchmöglichkeiten waren dabei die wichtigsten.

+ Durch Suchmöglichkeiten kann gezielt eine bestimmte Komponente in einer Visualisierung angesteuert werden.
- Existieren keine Suchmöglichkeiten, so ist es bei größeren Programmen schwer, einzelne Programmteile wiederzufinden.

Farben (T9): Eine große Rolle spiele nach Bassil und Keller [BK01] auch Farbe.

+ Nach Bassil und Keller [BK01] werde eine Darstellung durch das Verwenden von Farben übersichtlicher.
- Werden z.B. alle Verbindungen zwischen Komponenten gleichfarbig dargestellt, so ist es schwer, sie zu unterscheiden.

Hierarchie (T10): Die Präsentation der Hierarchie eines Programms sei ebenfalls wichtig (nach Bassil und Keller [BK01]). Gefordert werde neben der Darstellung aber auch die Navigationsmöglichkeit durch die verschiedenen Ebenen.

+ Navigationsmöglichkeiten durch verschiedene Ebenen vereinfachen die Untersuchung von hierarchischen Programmen.
- Besteht keine Möglichkeit verschiedene Hierarchien eines Programms anzuzeigen, wird eine komplette Darstellung der Programmausführung oft zu unübersichtlich.

Einstellen der Ansicht (T11): Für ein Visualisierungswerkzeug ist es wichtig, daß die Ansicht auf die Kommunikationsdaten individuell einstellbar und während der Visualisierung veränderbar ist.

+ Je nach Aufgabengebiet, hat der Entwickler die Möglichkeit, die beste Visualisierung einzustellen.
- Kann keine Visualisierung geändert werden, ist es für einen Entwickler nicht möglich, Details eines Programms herauszufinden.

Kapitel 5

Agenten im elektronischen Handel

Agenten gibt es schon seit vielen Jahrzehnten oder Jahrhunderten. Der sicher berühmteste Agent in unserer Zeit ist James Bond. Nein, James Bond hat nichts mit Software zu tun und ist auch nicht im elektronischen Handel tätig. Hong *et al.* [HGZ99] sagen, daß Software-Agenten sich ähnlich wie menschliche Agenten verhalten. Deshalb soll im nächsten Abschnitt James Bond zur Hilfe genommen werden, um Software-Agenten zu erklären. In Abschnitt 5.2 wird dann ein Software-Agent vorgestellt, der als Beispiel dienen soll, HotAgent (s. Kapitel 6) und andere Entwicklungs-Umgebungen (s. Kapitel 8) zu untersuchen. Abschnitt 5.3 beschäftigt sich damit, wie der zuvor eingeführte Agent in der Komponententechnik umgesetzt werden kann. Abschließend wird in Abschnitt 5.4 das Umfeld vorgestellt, in dem die mit *HotAgent* erstellten Agenten eingesetzt werden können.

5.1 Allgemein

Eine einheitliche Definition für Software-Agenten existiert noch nicht. Schaut man sich mehrere Definitionen an, merkt man aber, daß sie sich doch ähneln. Hong *et al.* [HGZ99] schreiben:

> Software agents are entities that function autonomously and perform laborious information processing tasks cooperatively.

Pichler *et al.* [PPW02] schreiben:

> Ein Software-Agent ist ein im Auftrag eines Benutzers weitgehend autonom handelndes Programm, das im Idealfall auch mit einer gewissen Entscheidungskompetenz ausgestattet ist.

Gemeinsam haben die Definitionen, daß ein Agent selbständig handelt. Bekommt James Bond einen Auftrag, so wird von ihm auch erwartet, daß er seine Aufgaben selbständig ausführt, ohne daß er ständig Rückfragen stellt. Dafür ist es auch notwendig, daß er eine gewisse Entscheidungskompetenz hat. Ohne die kann er nicht selbständig arbeiten. Dazu gehört aber auch, daß er mit anderen zusammenarbeiten kann.

Hilmer [Hil99] und Hong *et al.* [HGZ99] stellen Eigenschaften auf, die ein Agent meistens erfüllt. Sie fordern aber nicht, daß alle Kriterien erfüllt werden müssen, damit ein sogenannter ‚Agent' auch ein Agent laut obiger Definition ist.

Künstlich: Ein Agent ist künstlich, da er von Menschenhand geschaffen wurde.

Intelligent: So wie James Bond, muß ein Software-Agent die Situation bzw. die gebotenen Informationen richtig interpretieren, damit er eine Entscheidung treffen kann. Ist dies nicht der Fall, kann er nicht auf die gerade gegebene Situation eingehen und handelt u.U. falsch. Hoffmann [Hof01a] fordert eine Balance zwischen Intelligenz und Einfachheit, damit die Komplexität eines Agenten nicht zu hoch werde.

Autonom: Ist ein Agent in der Lage, selber Entscheidungen zu treffen, kann er unabhängig arbeiten. Wie oben erwähnt, ist es nicht gewollt, daß ein Agent immer bei der zuständigen Organisation nachfragt, ob er etwas so oder anders machen soll. Hat James Bond eine Aktion ausgeführt, so muß er, was auch für einen Software-Agenten gilt, sein Handeln kontrollieren und bewerten.

Sozial: Wichtig für James Bond ist es, daß er sich mit anderen Menschen austauschen kann. Es bringt wenig, wenn er vor sich hin arbeitet, ohne daß er jemandem mitteilen kann, wie groß seine Erfolge sind oder wie er neue Aufträge erhält. Genau so muß auch ein Software-Agent in der Lage sein, von Menschen seine Aufträge zu bekommen, eventuell mit anderen Agenten zusammenarbeiten zu können und seine Ergebnisse zu präsentieren.

Initiativ: James Bond arbeitet im Auftrag seiner Majestät und verfolgt somit deren Ziele. Genauso hat ein Software-Agent einen eindeutigen Auftraggeber und muß deshalb die Initiative ergreifen, um ihn zu unterstützen.

Lernfähig: Bei jedem Auftrag, egal ob er geglückt oder gescheitert ist, lernt James Bond etwas dazu. Das gleiche wird von einem Software-Agenten erwartet. Er muß sich merken, bei welchen Informationen er so oder so entschieden hat, um aus eventuellen früheren Fehlern zu lernen.

Einsatzbereit: James Bond muß Tag und Nacht auf der Hut sein, weil er nicht weiß, wann die Bösewichte ihn behelligen. Genauso muß ein Software-Agent

auch ständig einsatzbereit sein, um anfallende Informationen sofort bearbeiten zu können.

Mobil: Bekommt James Bond einen Auftrag, ist es notwendig, seinen Arbeitsort zu wechseln. Für viele Arbeiten muß er vor Ort sein und kann sie nicht bequem von seinem Schreibtisch aus erledigen. Das bringt es mit sich, daß er sich auch den jeweiligen Gegebenheiten anpassen muß. Ein Software-Agent ist zwar nicht ganz so flexibel wie ein Mensch, es besteht aber für ihn auch die Möglichkeit, seinen Einsatzort zu wechseln und sich u.U. anzupassen.

Mental definiert: Das Handeln von James Bond ist in der Regel mental beeinflußt. Er handelt so, wie es ihm sein Verstand vorgibt. Software-Agenten haben auch die Möglichkeit, nach ihrem Verstand bzw. den gegebenen Informationen zu handeln.

Emotional definiert: Das Handeln von James Bond ist in den verschiedenen Situationen oft sehr emotional betont. Ob das bei Software-Agenten auch der Fall sein kann, wird mit Recht skeptisch betrachtet. Hier soll darauf nicht weiter eingegangen werden.

Sandholm [San99] sagt, daß Software-Agenten erfolgreicher seien, wenn es darum gehe, Geschäfte abzuschließen. Ob das auch im Bezug auf James Bond stimmt, ist zweifelhaft. Er ist aber auch nur eine ausgedachte Filmfigur. Software-Agenten kommen auch bei unterschiedlichen Anforderungen schneller zu einer Lösung.

Handl und Hoffmann [HH99b] beschäftigen sich mit Arbeitsfluß-Leitsystemen (engl. Workflow Management Systems) bei der Dokumentenverarbeitung für den elektronischen Handel. Sie sagen, daß diese Systeme nicht leistungsfähig genug sind, um spontane Reaktionen zu ermöglichen. Agenten hingegen sind flexibel genug, aber nicht effizient genug, was das Unterstüten des Arbeitsflusses angeht. Eine Kombination aus beiden Systemen kann aber die Lösung bringen. Dabei gibt es drei Arten von Agenten:

Agierende Agenten: Sie erfüllen jeweils eine spezielle Aufgabe. Sie werden von Arbeitsfluß-Leitsystemen aufgerufen, um z.B. Dokumente zu bearbeiten. Die Agenten führen dabei vorwiegend Routinearbeiten aus.

Persönliche Agenten: Routinearbeiten werden von persönlichen Agenten ausgeführt und agieren auf Veranlassung von einzelnen Benutzern.

Betriebsinterne Agenten: Arbeitsfluß-Leitsysteme werden durch betriebsinterne Agenten unterstützt. Sie erstellen z.B. verschiedene Ansichten für unterschiedliche Benutzer und verwalten Zugriffsrechte für die einzelnen Dokumente.

5.2 Beispiel elektronische Apotheke

Im folgenden wird ein Software-Agent vorgestellt, der als Beispiel dienen soll, um HotAgent (s. Kapitel 6) und andere Entwicklungsumgebungen (s. Kapitel 8) zu untersuchen. Zugegebenermaßen hat dieser Agent nicht viel mit James Bond gemeinsam, außer daß er hilft, Viren zu bekämpfen, die einen Patienten plagen.

Der Medical Advisory Service Agent (MASA) [Cla01] ist ein Agent, der Patienten Ratschläge für ein mögliches Medikament gibt. Dazu erhält der Agent elektronische Post von Patienten, die er nach möglichen Krankheitssymptomen analysiert. Anhand dieser Symptome versucht der Agent eine Krankheit festzustellen. Ist dies eindeutig möglich, kann der Agent ein mögliches Medikament vorschlagen. Ist eine Zuordnung nicht eindeutig möglich, muß der Agent eine Rückfrage an den Patienten formulieren, um so mehr Informationen über eine mögliche Krankheit zu erfragen.

Ob ein solcher Agent in Deutschland rechtlich vertretbar ist oder nicht, soll außer acht gelassen werden. Andere Anwendungsgebiete ähnlicher Art liegen auf der Hand.

Der Patient soll mit dem Agenten über elektronische Post kommunizieren. Um den Prototyp des Agenten zu vereinfachen, wurde eine graphische Benutzungsoberfläche erstellt, die den Postaustausch vereinfachen soll. Der Benutzer tippt einfach seinen Brief in eine Eingabebox ein und drückt dann den Knopf *Submit*, um den elektronischen Brief abzuschicken. Die gleiche Eingabebox wird dann wieder benutzt, um die Antwort des MASA anzuzeigen.

Abbildung 5.1: Anfrage an den Agenten

Abbildung 5.2: Antwort des Agenten

Abbildung 5.1 und 5.2 zeigen die Benutzungsoberfläche des MASA. Abbildung 5.1 zeigt die Anfrage des Patienten und Abbildung 5.2 enthält die Antwort des

Agenten. Der Prototyp des Agenten kann immer nur Anfragen nacheinander beantworten.

Im vorangegangenen Abschnitt 5.1 wurden Eigenschaften für Agenten vorgestellt. Nun soll kurz betrachtet werden, welche auch für den MASA gelten. Der Agent ist *künstlich*, da er ja von Menschenhand geschaffen wurde. Es wird von ihm auch eine selbständige Arbeit verlangt, wodurch er als *autonom* bezeichnet werden kann. Er muß mit Patienten kommunizieren und wird deshalb als *sozial* eingestuft. *Initiativ* ist er, da er stets das Bestreben hat, gute Vorschläge für ein Medikament zu machen. *Lernfähig* ist er im derzeitigen Ausbaustadium nicht. Dazu müßte er sich merken, welche Empfehlungen er gegeben hat und diese zum geeigneten Zeitpunkt auswerten. *Einsatzbereit* muß er immer sein, denn er muß jederzeit mit einer Patientenanfrage rechnen. *Mobil* ist er nicht, er macht keine Krankenbesuche, sondern wartet nur auf elektronische Anfragen. *Mental* oder *emotional definiert* ist der MASA nicht.

5.3 Aufteilung von Agenten in Komponenten

Nun soll untersucht werden, wie sich der MASA in Komponenten mit möglichst minimaler Komplexität aufteilen läßt. Dazu haben sich Clausius [Cla01], Giesl [Gie01] und wir [HHM03] Gedanken gemacht. Im folgenden werden Komponenten in drei verschiedene Gruppen eingeteilt, um ein hierarchisches Zusammensetzen von Komponenten zu erlauben. Diese drei Gruppen sind aber nicht notwendigerweise die einzig möglichen.

Systemkomponente: (engl. top system components) Die Systemkomponente ist die umfassende Komponente eines fertiggestellten Hauptprogramms und ist der Einstiegspunkt bzw. Aktivierungspunkt für das ausführbare Programm.

Zwischenkomponenten: (engl. intermediate components) Diese helfen (optional), um ein System zu strukturieren, und besitzen eine definierbare Funktionalität. Sie sollen helfen, die Komplexität eines Agenten zu reduzieren.

Minimalkomponenten: (engl. minimal components) Diese Komponenten sind die kleinsten Teile in der Hierarchie und stellen elementare Funktionen bereit.

In den verschiedenen Geschäftsanwendungen gibt es immer ähnliche Aufgabengebiete. Dies erlaubt es, spezielle Komponenten immer wieder zu verwenden. Es können z.B. Komponenten sein, die bestimmte Kommunikationsprotokolle bereitstellen, damit mit speziellen Partnern kommuniziert werden kann.

Wie die MASA Systemkomponente in Zwischenkomponenten und Minimalkomponenten aufgeteilt werden kann, soll im folgenden untersucht werden. Dazu wurden mehrere Möglichkeiten gefunden.

5.3.1 Lineare Verarbeitung

Der erste Ansatz geht davon aus, daß alle Komponenten für einen Agent miteinander linear verknüpft werden (s. Abb. 5.3). Dabei erhält die erste Komponente die initialen Daten, bearbeitet diese und reicht sie an die nachfolgende Komponente weiter. Dies geschieht so lange nach dem vorgegebenen Muster, bis die letzte Komponente die Daten bekommen und verarbeitet hat.

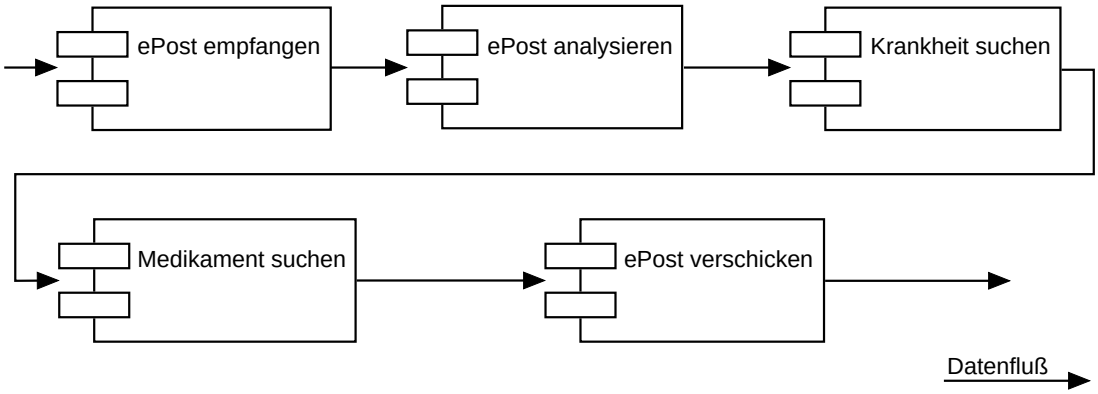

Abbildung 5.3: Lineare Verarbeitung

Bei dieser Aufteilung sind alle verwendeten Komponenten Zwischenkomponenten. Die Komponenten sind stark auf die benötigte Funktionalität zugeschnitten und können dadurch nicht gut wiederverwandt werden. Ein anderer Nachteil ist, daß alle Daten von der ersten bis zur letzten Komponente in der Kette durchgereicht werden müssen, z.B. wird die ePostadresse von der ePost empfangen Komponente bis zu der ePost verschicken Komponente weitergereicht.

Der MASA läßt sich dabei in folgende fünf Komponenten aufteilen (s. Abb. 5.3):

ePost empfangen: Diese Komponente wartet auf elektronische Briefe, die eine Beschreibung der Krankheitssymptome enthält. Sie stößt nach Erhalt das Verarbeiten an und gibt die Absenderadresse mit dem Brieftext weiter.

ePost analysieren: Der Agent muß den Brief auf Symptome analysieren. Um die Komponente zu vereinfachen, sucht sie nur nach Schlüsselworten und benutzt keinen Algorithmus aus dem Gebiet der künstlichen Intelligenz. Sie stellt die gefundenen Krankheitssymptome bereit und gibt die Absenderadresse unverändert weiter.

Krankheit suchen: An Hand der Symptome kann nun die Krankheit gesucht werden, die der Patient haben könnte. Die Komponente gibt wieder die Adresse weiter und gibt dazu noch die gefundene Krankheit an.

Medikament suchen: Für das Antwortschreiben fehlt dem Agenten noch eine Medikamentenempfehlung. Dafür ist diese Komponente zuständig. Sie stellt wieder die Adresse mit der Krankheit bereit und fügt noch ein passendes Medikament hinzu.

ePost verschicken: Die letzte Aufgabe ist, aus der Krankheit und dem Medikament eine Antwort für den Patienten zu erstellen. Das erledigt diese Komponente und schickt die Antwort an den Patienten zurück.

5.3.2 Wiederaufnahme und Transaktionsmanagement

Lineare Verarbeitung ist nicht ausreichend, wenn es um wiederkehrende Aufgaben im elektronischen Handel geht. Dazu wird ein geschlossener Kreislauf benötigt. Zusätzlich wird eine Komponente warte eingefügt (s. Abb. 5.4):

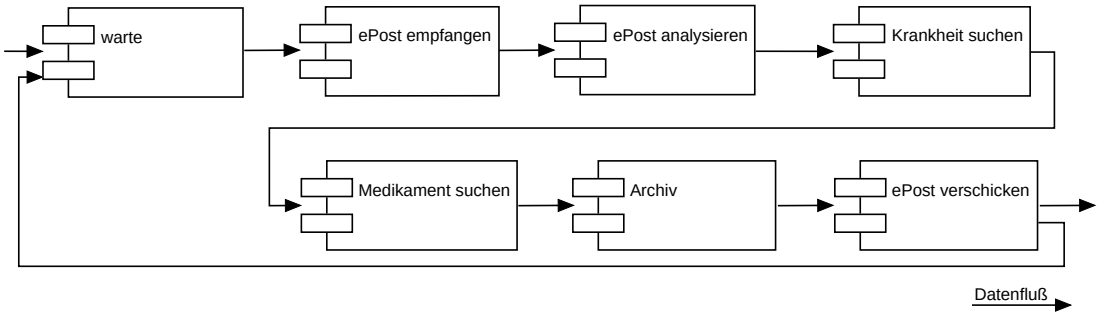

Abbildung 5.4: Lineare Verarbeitung mit Wiederaufnahme und Transaktionsmanagement

warte: Diese Komponente hält die Verarbeitung an, bis eine neue Anfrage eingeht.

Der in Abschnitt 5.3.1 vorgestellte Agent bietet keine Möglichkeit, zuvor gemachte Empfehlungen zu speichern. Er besitzt dafür kein Patientenarchiv. Sinnvoll ist es aber bei einer erneuten Untersuchung des gleichen Patienten zuvor gemachte Empfehlungen zu berücksichtigen. Die in diesem Abschnitt verwendete Variante des MASA bietet dazu noch die folgenden Möglichkeiten, auf frühere Transaktionen zurückzugreifen:

- Eine zusätzliche zentrale Archiv-Komponente, in der alle Patientendaten und Krankeitssymptome gespeichert werden (s. Abb 5.4).
- Spezifische Archivfunktionen, die in jeder Komponente des Agenten angesiedelt sind. Es wird davon ausgegangen, daß in dem zu bearbeitenden Dokument jeweils Informationen über frühere Transaktionen enthalten sind, wie in den Arbeiten Handl [Han01] und Hoffmann [HH99a] erklärt wird.

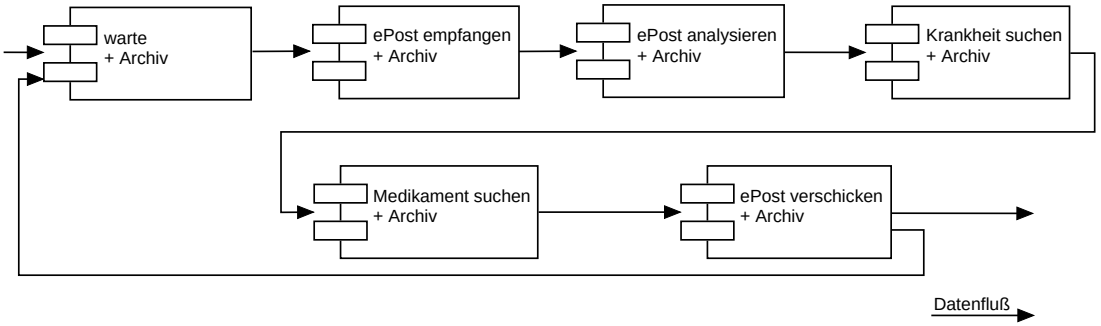

Abbildung 5.5: Lineare Verarbeitung mit Wiederaufnahme und Transaktionsmanagement im Dokument

Archiv: Diese Komponente realisiert ein Archiv für alle ePost-Anfragen. Sie speichert alle Transaktionen mit den Patienten.

... + Archiv: Diese Komponenten ersetzen die entspechenden Komponenten aus Abbildung 5.3 und 5.4 durch solche mit Archivfunktionen.

5.3.3 Dienst- und Steuerungskomponenten

Eine andere Möglichkeit ist, die Systemkomponente des MASA einzuführen und diese dann in Dienst- und Steuerungskomponenten aufzuteilen, um Funktionalität und Verhalten zu trennen. Jeder Agent hat in der Regel eine *Steuerungskomponente*, die eine Zwischenkomponente darstellt. Diese beinhaltet das wesentliche Verhalten des MASA und wird verwendet, um die Aufgaben der Dienstkomponenten zu steuern. Die *Dienstkomponenten* (Zwischen- oder Minimalkomponenten) hingegen realisieren Standardfunktionalität und zeichnen sich durch eine hohe Wiederverwendbarkeit aus (s. Abb. 5.6).

Die Steuerungskomponente des MASA hat folgende Aufgabe:

Steuerung: Sie repräsentiert den Zustand des Agenten und bildet das Herzstück. Sie hat die Aufgabe, die einzelnen Bearbeitungsschritte zu koordinieren. Sie

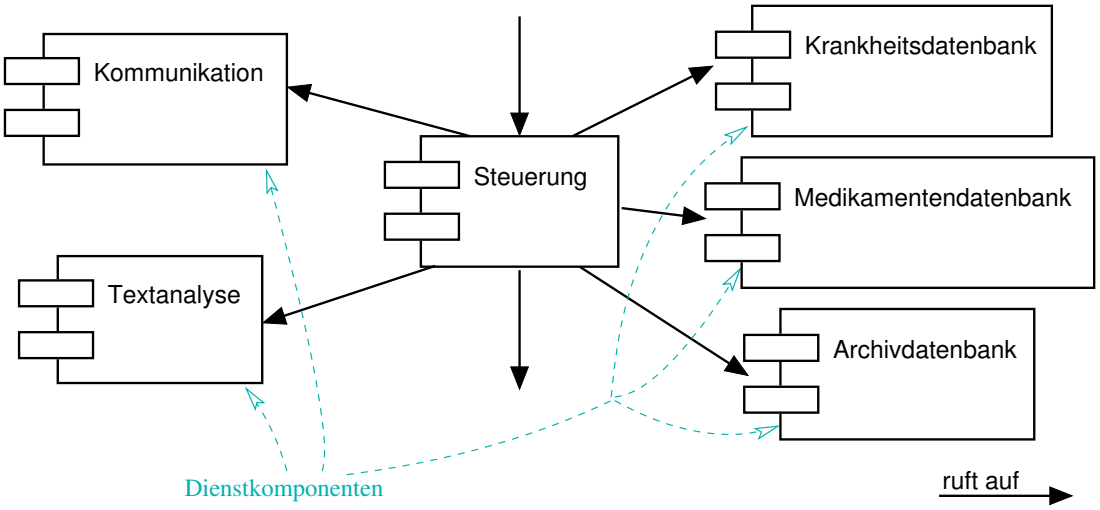

Abbildung 5.6: Dienst- und Steuerungskomponenten

ruft (im Sinne von Parnas [Par76]) die Dienstkomponenten auf und teilt diesen kleine Aufgabenbereiche zu. Dazu besitzt sie entsprechende Ein- und Ausgänge, um mit den Dienstanbieterkomponenten zu kommunizieren. Die Steuerungskomponente ist sehr auf die Bedürfnisse eines Agenten zugeschnitten und muß deshalb meist von Hand mittels einer textuellen Programmiersprache programmiert werden.

Neben der Steuerungskomponente benötigt der MASA noch folgende Dienstkomponenten:

...datenbank: In dieser Komponente können beliebige Datensätze gespeichert werden, die an Hand eines Schlüssels identifiziert werden können. Die Datenbank unterstützt neben dem Einfügen und Löschen von Datensätzen auch die Abfrage von Datensätzen nach definierbaren Kriterien.

Die Datenbankkomponente kann an drei Stellen in dem MASA eingesetzt werden. In der *Krankheitsdatenbank* werden die möglichen Symptome mit den dazugehörigen Krankheiten verknüpft. Um Empfehlungen für Medikamente auszusprechen, ist die *Medikamentendatenbank* notwendig. Die *Archivdatenbank* dient dazu, um Informationen über die Patienten abzulegen.

Textanalyse: Diese Komponente hat die Aufgabe, einen elektronischen Brief zu analysieren. Auch hier ist die Analyse von Texten auf die Erkennung von Schlüsselworten reduziert, da die Analyse von natürlicher Sprache nicht Thema dieser Arbeit ist. Wird der MASA in einem echten System eingesetzt, ist es kein Problem, diese Komponente durch eine leistungsfähigere auszutauschen.

Kommunikation: Die Kommunikationskomponente ist für das Abwickeln der Kommunikation zwischen Patienten (Geschäftspartnern) und dem MASA zuständig.

5.3.4 Zusammengesetzte Steuerungskomponenten

Wie zuvor erwähnt, sind Steuerungskomponenten speziell auf einen Agenten zugeschnitten. Diese werden meist textuell in einer normalen Programmiersprache entwickelt. Besser und einfacher ist es aber, Steuerungskomponenten aus bereits vorgegebenen Komponenten zu erstellen. So lassen sich auch leicht generische Steuerkomponenten anfertigen, die durch nur geringe Anpassungen auf eine spezielle Aufgabe zugeschnitten werden können, was Wagner und Horling [WH01] fordern. Mit Hilfe von Zwischen- und Minimalkomponenten können Steuerkomponenten erstellt werden, die wieder System- oder Zwischenkomponenten sind.

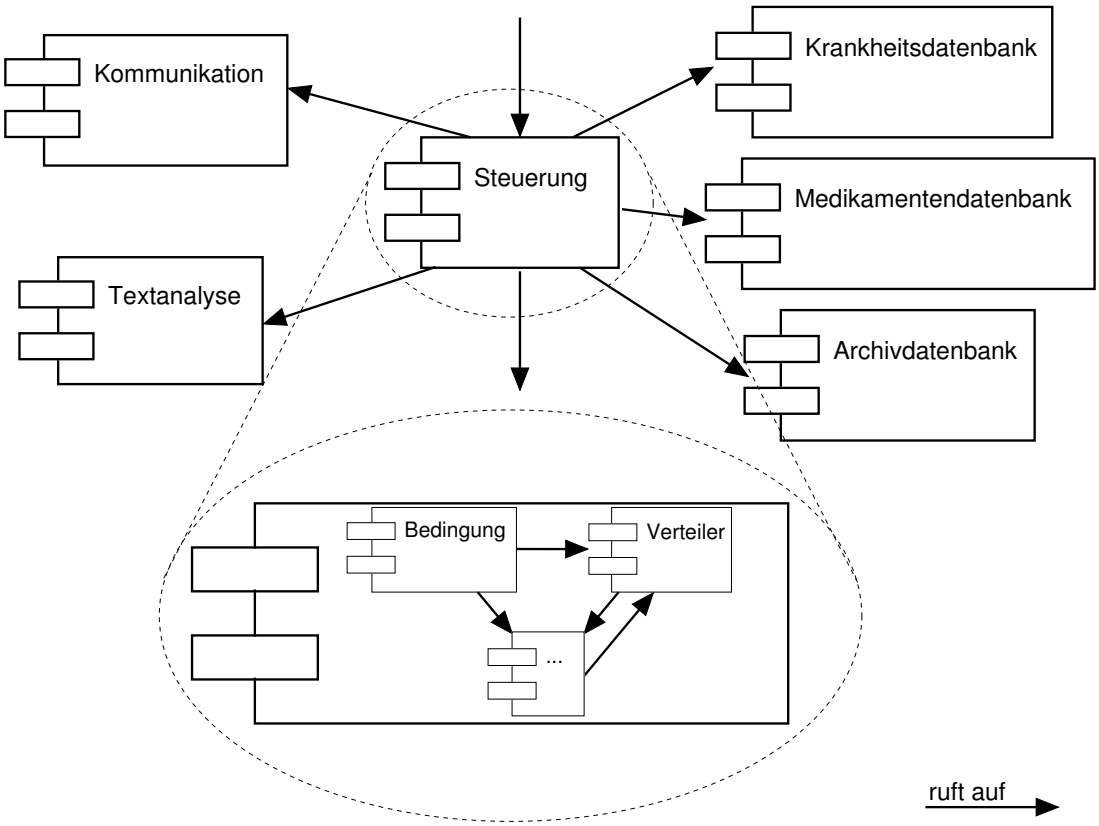

Abbildung 5.7: Zusammengesetzte Steuerungskomponenten

Im MASA (s. Abb. 5.7) wird die Aufteilung von Dienst- und Steuerkomponenten, wie in Abschnitt 5.3.3, beibehalten. Die Steuerkomponente (Zwischenkom-

ponente) wird aber durch mehrere Minimalkomponenten zusammengesetzt. Diese Minimalkomponenten sind z.B. Verteiler, Auswähler, Bedingungen und Schleifen.

Durch die Definition von Mustern, wie die zuvor genannten Minimalkomponenten zusammengesetzt werden können, ist es möglich, generische Steuerkomponenten zu erstellen. Diese brauchen dann beim Verwenden in einem Agenten nur noch angepaßt und konfiguriert werden.

5.4 HOTxxx Projekt

Das HOTxxx Projekt [Hof01b], zu dem auch HotAgent gehört, hat seine Wurzeln im Jahr 1995. Das Projekt begann mit dem Entwickeln eines flexiblen Dokumentverwaltungssystems HOTDoc [Buc98, Buc00]. Die Idee war, in ein elektronisches Dokument verschiedene Dokumentteile einzufügen. Diese Teile können z.B. Text, Mediendaten, Tabellen (HOTCalc) usw. sein. Es enthält eine Skriptsprache (HOTScrip), mit der bestimmte Mechanismen, wie z.B. das Erstellen einer Graphik (HOTDraw), automatisiert werden können. Ebenfalls erlaubt sie auch Interaktion mit dem Anwender. Das HOTDoc Dokumentverwaltungssystems wurde darauf in weiteren Arbeiten ergänzt. HOTSimple [Kun02] ist z.B. ein Simulations- und Planungswerkzeug, mit dem komplexe Tabellenkalkulationen ausgeführt werden können.

Auf diesen Arbeiten basiert das internet-basierte Arbeitsfluß-Leitsystem HOTFlow [Han01, HH99a]. Es regelt den Dokumentenaustausch zwischen Geschäftspartnern im Internet. Es wird u.a. kontrolliert, wer welche Dokumentteile betrachten und ändern darf. Zusätzlich wird für das Archivieren der Kommunikation gesorgt.

In diese Arbeit läßt sich HotAgent eingliedern. Die mit HotAgent erstellten Agenten sollen Routinearbeiten für den elektronischen Handel übernehmen und damit einem Geschäftspartner Arbeit abnehmen. Sie können z.B. elektronische Dokumente analysieren, bearbeiten und beantworten. Eine andere Aufgabe ist, eine Sortierung der Post anhand des Inhalts vorzunehmen und den zuständigen Sachbearbeitern zuzustellen. Um dies zu ermöglichen, lassen sich die Agenten in den Arbeitsfluß von HOTFlow eingliedern.

Kapitel 6

HotAgent Entwicklungsumgebung

HotAgent [Mar, Mar01] ist eine visuelle Entwicklungsumgebung für Agenten im elektronischen Handel. Sie beinhaltet einen Komponentenentwurfsrahmen, der spezielle Komponenten für die leichte Erstellung von Agenten bietet.

HotAgent bietet verschiedene Werkzeuge für die komponentenbasierte Entwicklung von Agenten an. Darunter fallen Werkzeuge zum Erstellen von Agenten (s. Abschnitt 6.1), zum Entwickeln von Komponenten (s. Abschnitt 6.2), zum Testen von Komponenten (s. Abschnitt 6.3 und 6.4) und zum Visualisieren von Komponentenprogrammen (s. Abschnitt 6.5). In Kapitel 3.2 wird gefordert, daß *Komponenten- und Anwendungsentwicklung (K4, V9)* zwei getrennte Teilbereiche sein sollen, was durch die verschiedenen Werkzeuge von HotAgent gewährleistet wird. In Abschnitt 6.6 wird HotAgent Center vorgestellt, das eine zentrale Steuerung der einzelnen Werkzeuge ermöglicht.

Beim Entwickeln von HotAgent wurde auf ein einheitliches Konzept geachtet. HotAgent erlaubt nur das Entwickeln mit Komponenten. Dadurch wird verhindert, daß der Entwickler das Konzept der Komponententechnologie mit der objektorientierten Entwicklung mischen kann, das andere Entwicklungsumgebungen zulassen. Damit werden von vornherein Probleme ausgeschlossen, die das Entwickeln von Anwendungen beeinträchtigen könnten.

Alle Werkzeuge präsentieren sich in einer *einheitlichen Benutzungsoberfläche (V13)*. Dadurch muß der Entwickler nur den Umgang mit einem Werkzeug lernen und kann seine Vorgehensweise direkt auf die anderen Werkzeuge übertragen. Alle Werkzeuge bieten eine visuelle Unterstützung beim Entwickeln und darüber hinaus eine Möglichkeit des visuellen Programmierens (s. Kapitel 3.1).

Die Werkzeuge können auch gleichzeitig benutzt werden, z.B. kann eine neue

Komponente entwickelt werden, während zur gleichen Zeit deren Testfall spezifiziert wird. Es ist auch möglich, ein Werkzeug aus einem anderen aufzurufen.

In den folgenden Abschnitten werden die einzelnen Werkzeuge von HotAgent vorgestellt. Zuerst wird auf konzeptionelle Bestandteile eingegangen. Dann wird das Vorgehen zum Erstellen einer Anwendung bzw. Komponente beschrieben. Zur besseren Übersichtlichkeit wird die Vorgehensweise eingerückt gedruckt.

6.1 HotAgent Assembly

Das erste Werkzeug HotAgent Assembly [Mar02b] dient dazu, aus vorgefertigten Komponenten Agenten zusammenzusetzen. In unseren Arbeiten [Mar00, MS00a] haben wir gezeigt, daß sich die Graphenmetapher am besten für die visuelle Komposition von Komponenten eignet. Die Komponenten werden dabei als Piktogramme (Symbole) und die Verbindungen (syntaktische Elemente) als Pfeile dargestellt. Die Komponenten können auf der Arbeitsfläche beliebig angeordnet werden, wodurch ein Gruppieren nach Aufgabenbereichen erlaubt wird. Außerdem können die Piktogramme der Komponenten je nach Anwendungsgebiet der Komponenten frei gewählt werden, um ein besseres Wiedererkennen zu ermöglichen. Dies entspricht dem Kriterium der *Nähe zur Abbildung (V1)* und bietet eine *Ansicht (K6)* auf die Komponentenprogramme.

Komponenten können zusammengesetzt werden, indem Ausgänge einer Komponente mit Eingängen einer anderen verbunden werden. Es können sowohl mehrere Ausgänge mit einem Eingang als auch ein Ausgang mit mehreren Eingängen verbunden werden. Da ausnahmslos alle Verbindungen zwischen den Komponenten angezeigt werden, ist dafür gesorgt, daß keine *versteckten Abhängigkeiten (V3)* existieren.

HotAgent Assembly erlaubt das Zusammensetzen von Anwendungen zur Entwicklungszeit. Dabei werden alle Komponenten und ihre Verbindungen visualisiert. Es gibt zwei verschiedene Arten von Komponenten:

Sichtbare Komponenten repräsentieren Elemente auf der graphischen Benutzungsoberfläche.

Nicht-sichtbare Komponenten sind Komponenten, wie z.B. Datenbanken, und sind nicht graphisch auf der Benutzungsoberfläche abgebildet.

Die Arbeitsfläche des HotAgent Assembly Editors (s. Abb. 6.1) besteht aus zwei Teilen. Alle nicht-sichtbaren Komponenten können auf dem *Hauptteil* positioniert werden. Der *Rahmen* hingegen kann sichtbare und nicht-sichtbare Komponenten aufnehmen.

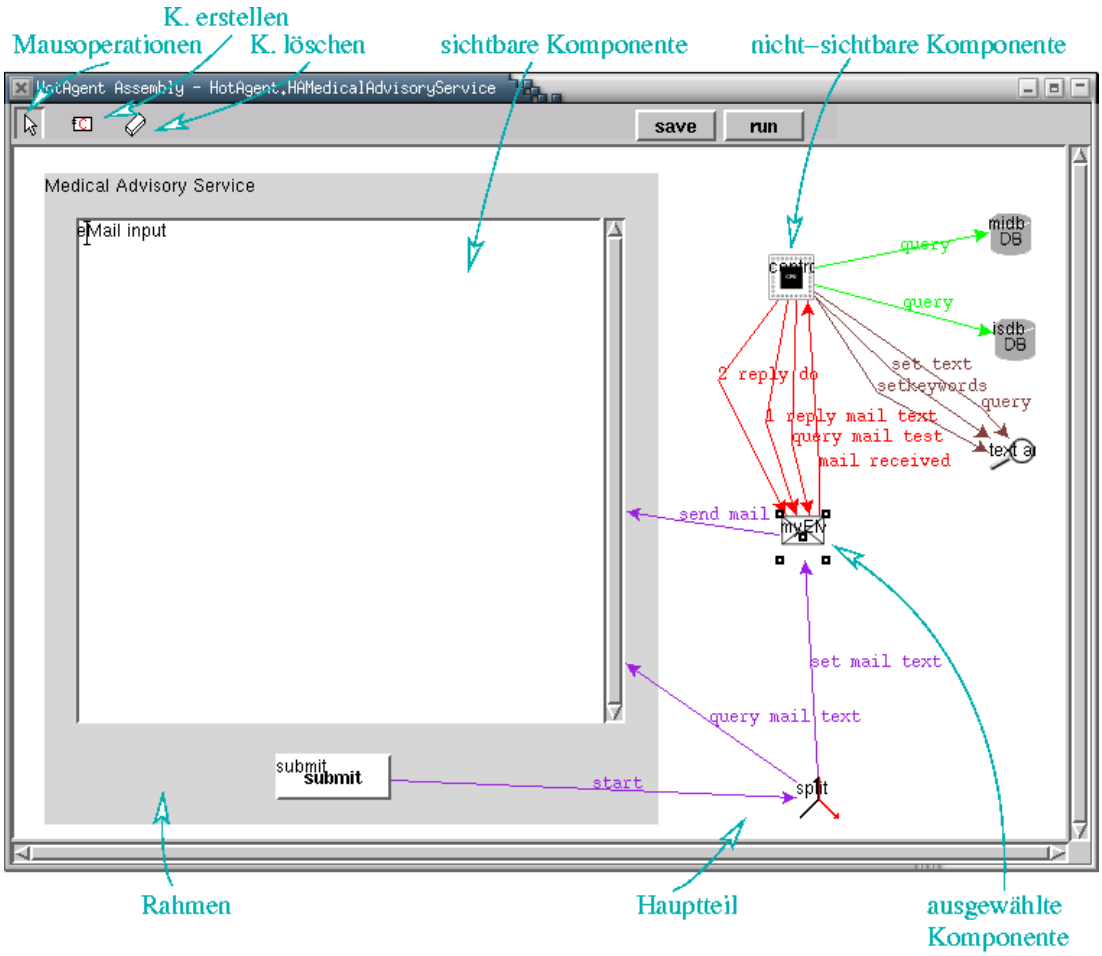

Abbildung 6.1: HOTAGENT ASSEMBLY Editor

Um herauszufinden, ob eine Komponente die erwartete Funktionalität bereitstellt, kann jederzeit eine Beschreibung (*V10*) einer Komponente angezeigt werden. In dieser Beschreibung werden auch alle vorhandenen Ein- und Ausgänge der Komponenten erwähnt. Zusätzlich zu den Namen der Ein- und Ausgänge existiert auch hier wieder eine kurze Beschreibung.

Jedes Komponentenexemplar benötigt einen Namen, der in dem erstellten Agenten eindeutig sein muß. Im Gegensatz zu dem Komponenten-Beschreibungsnamen, der im globalen Namensraum gespeichert wird, wird der Exemplarname nur in dem Namensraum des jeweiligen Agenten definiert.

Soll eine neue Anwendung entwickelt werden, muß als erstes die Position für eine neue Komponente auf der Arbeitsfläche gewählt werden. Danach erscheint ein Eingabefenster, (s. Abb. 6.2) um eine Komponente auszuwählen. Wie zuvor erwähnt, werden Komponenten in

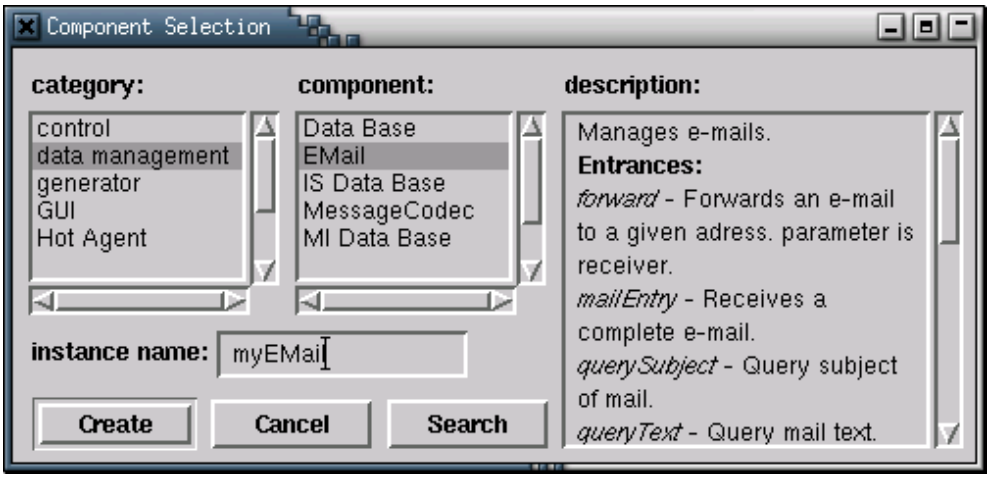

Abbildung 6.2: Erstellen von Komponenten

mehrere Kategorien eingeteilt. Das Fenster zeigt eine Liste, um eine Kategorie zu wählen, und nachdem das geschehen ist, kann eine Komponente gewählt werden. Hier gibt es ein Feld, in dem eine Beschreibung der ausgewählten Komponenten gezeigt wird. Jedes Komponentenexemplar benötigt einen Namen, der ebenfalls in dem Eingabefenster definiert wird. Nach dem Auswählen des Knopfes *Create*, wird die Komponente an die zuvor gewählte Position auf die Arbeitsfläche gesetzt. In dem Beispiel wird eine EMail Komponente mit dem Exemplarnamen (*instance name*) myEMail erstellt.

Wenn eine große Anzahl von Komponenten vorhanden ist, kann es schwierig sein, eine passende Komponente mit der oben beschriebenen Methode zu finden. Eine andere Möglichkeit, eine Komponente zu finden, ist die Angabe eines Suchausdrucks. Nachdem der Knopf *Search* gedrückt wird, kann ein Suchausdruck definiert werden. Die Suche ist eine Volltextsuche, d.h. alle Komponentennamen und deren Beschreibungen, inklusive der Ein- und Ausgänge, werden durchsucht. Das Suchergebnis wird in einer Liste dargestellt. Zusätzlich wird für eine ausgewählte Komponente wieder die Beschreibung angezeigt.

Sobald alle benötigten Komponenten auf die Arbeitsfläche plaziert wurden, ist es notwendig, diese zu verbinden. Um eine Verbindung herzustellen, ist die Quellkomponente zu markieren. Darauf kann der mittlere Markierungspunkt auf die Zielkomponente gezogen werden (s. Abb. 6.3). Nachdem die Verbindung hergestellt wurde, prüft HotAgent, ob mindestens ein Quellkomponentenausgang und ein Zielkomponenteneingang existiert. Ist dies der Fall, erscheint ein Eingabefenster (s. Abb. 6.4), in dem die entsprechenden Aus- und Eingänge aus einer Liste gewählt werden können. Abhängig von der Auswahl

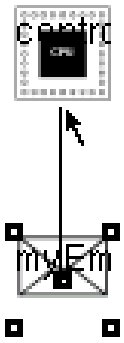

Abbildung 6.3: Verbinden von Komponenten

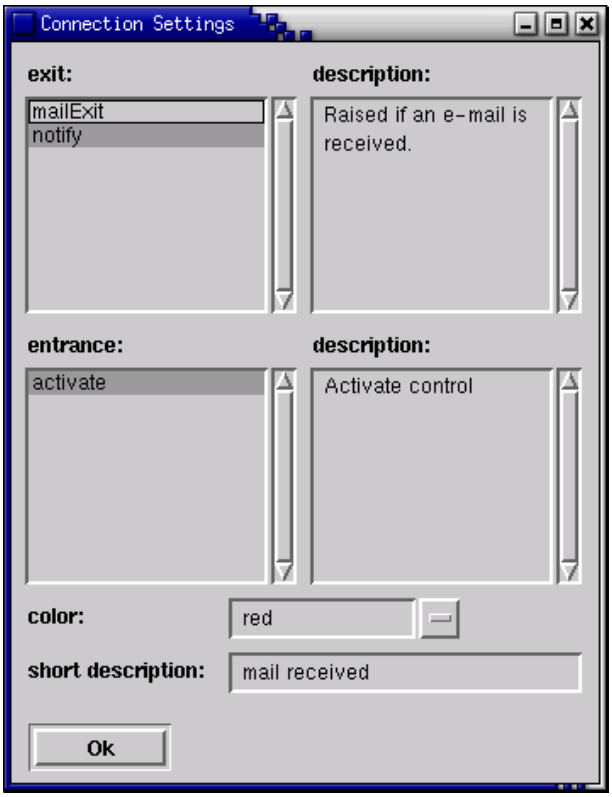

Abbildung 6.4: Erstellen von Verbindungen

wird eine Beschreibung des Ausgangs bzw. des Eingangs angezeigt. Zusätzlich kann ein Beschreibungstext und eine Farbe für die Verbindung spezifiziert werden. Die Beschreibung kann ein beliebiger Text sein, um eine kurze Erläuterung der Verbindung zu geben. Abbildungen 6.3 und 6.4 zeigen, wie eine Verbindung vom dem Ausgang notify der Komponente myEMail zu dem Eingang activate der Komponente control erstellt wird.

Um eine *sekundäre Notation (V6)* (s. Kapitel 3.2) und semantische Ausdrücke zu unterstützen, kann jede Verbindung benannt und eingefärbt werden. Petre [Pet95] warnt, daß eine *sekundäre Notation* unerfahrene Benutzer eher verwirre, erfahrene Benutzer könnten aber großen Nutzen daraus ziehen. Deshalb hat jeder Benutzer die Wahl, ob und wie er eine *sekundäre Notation* einsetzt.

In dem Beispiel (s. Abb. 6.1) haben die Verbindungen verschiedene Farben, je nach deren Aufgabe. Zusätzlich hat auch jede Verbindung eine kurze Beschreibung um deren Aufgabe zu verdeutlichen. Auch durch die Anordnung der Komponenten kann der Entwickler eine *sekundäre Notation* anwenden. Durch diese Möglichkeiten kann ein übersichtlich erstellter Agent bereitgestellt werden.

Liegen die Komponenten an einigen Stellen dicht aneinander, ist es möglich, daß die Arbeitsfläche doch unübersichtlich wird. Um dieses Problem zu lösen, besteht die Möglichkeit, die Ansicht zu vergrößern, zu verkleinern und/oder neu zu positionieren. Nach dem Ändern der Größe werden die Positionen der Komponenten geändert, aber nicht deren Abmessungen. Dies sogt dafür, daß zwischen den Komponenten mehr Platz entsteht. Wird diese Technik angewandt, ist es z.B. möglich, weitere nicht-sichtbare Komponenten zwischen sichtbare Komponenten zu plazieren. Es ist auch möglich, die Darstellung von nicht-sichtbaren Komponenten, Verbindungen oder Beschreibungen auszuschalten.

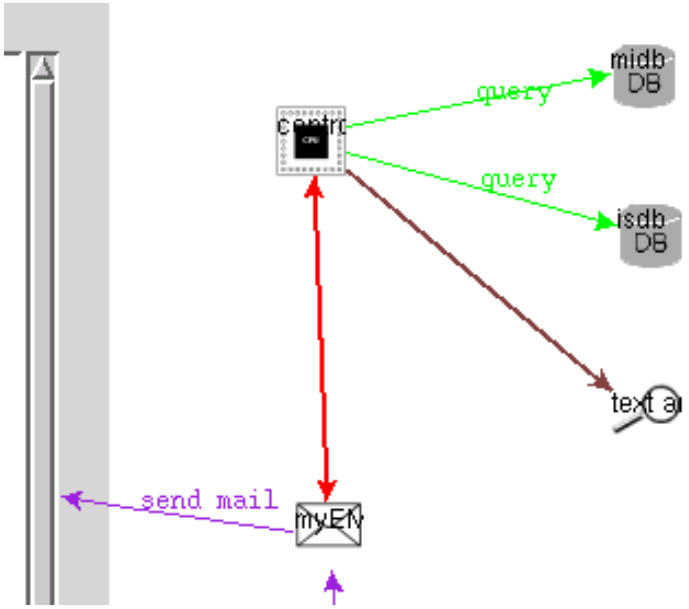

Abbildung 6.5: Darstellung mit Grad des Details

Als Alternative zum Verändern der Darstellungsgröße ist es auch möglich, den *Grad des Details* (engl. level of detail) zu ändern. Dazu untersucht HOTAGENT alle Verbindungen und vereinigt alle Verbindungen zwischen jeweils zwei Komponenten. Die neue Verbindung wird mit einer dickeren Linie dargestellt. Sie hat keinen Beschreibungstext und ihre Farbe ist eine Mischung aus den Farben aller vereinigten Verbindungen. Die vereinigte Verbindung hat einen oder zwei Pfeilspitzen, abhängig von den Richtungen der vereinigten Quellverbindungen. Wird Abbildung 6.5 mit Abbildung 6.1 verglichen, fällt auf, daß die vier roten Verbindungen zu einer Verbindung mit zwei Pfeilspitzen und die drei braunen zu einer mit nur einer Spitze verbunden sind. Um über die vereinigten Verbindungen Informationen zu erhalten, ist es möglich, diese zu markieren und die Vereinigung aufzuheben. Dadurch werden die ursprünglichen Verbindungen dargestellt. Es ist auch möglich, die Option *Trennung des Bezugs* (engl. separation of concern) zu aktivieren. Dabei werden nur Verbindungen mit ähnlichen Aufgaben vereinigt. Dabei wird die Angelegenheit nach der Farbe der Verbindung bestimmt. Es

werden nur Verbindungen mit derselben Farbe verbunden. Die hier vorgestellten Möglichkeiten zum Verändern der Darstellung der Arbeitsfläche erfüllen das Kriterium der *Sichtbarkeit (V7)*.

HotAgent unterstützt den Entwickler, wenn er Änderungen vornehmen möchte. Wenn eine Verbindung zwischen zwei Komponenten nicht richtig im Bezug auf die Aus- und Eingänge verbunden ist, ist es möglich, diese ohne Löschen der Verbindung zu ändern.

Um eine Verbindung zu ändern, wird diese und der Punkt *settings* im Kontextmenu ausgewählt. Darauf öffnet sich das Eingabefenster, wie in Abbildung 6.4 dargestellt. In diesem Fenster können der Quellausgang, der Zieleingang, die Farbe und der Beschreibungstext geändert werden.

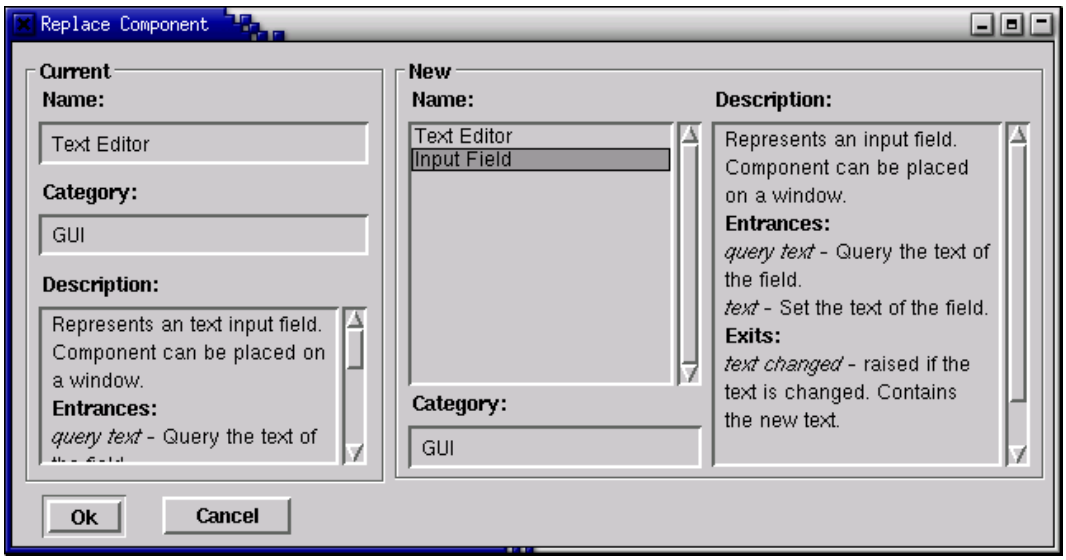

Abbildung 6.6: Ersetzen einer Komponente

Eine andere Möglichkeit, Änderungen vorzunehmen, ist, eine ganze Komponente zu ersetzen, ohne Verbindungen zu und von der Komponente zu löschen. In einen Eingabefenster (s. Abb. 6.6) erscheinen mögliche Alternativen für diese Komponente. Abhängig von den benutzten Aus- und Eingängen werden die möglichen Alternativen angeboten. Diese Änderungsmöglichkeiten erfüllen die Kriterien der *Viskosität (V2)* und der Reduktion des *Vorausdenkens (V5)* (s. Kapitel 3.2).

6.2 HotAgent Component

Der HotAgent Component [Mar02b] Editor ist ein Werkzeug, um nichtsichtbare Komponenten visuell zu erstellen. Er unterstützt den Entwickler bei der Erzeugung von neuen Komponenten unter Verwendung bereits definierter Komponenten. Das Kriterium, daß jede Komponente einen *klaren Aufgabenbereich (K9)* haben soll, kann nicht erzwungen werden. Dafür ist der Entwickler selbst zuständig. HotAgent Component basiert auf dem HotAgent Assembly Editor (s. Abschnitt 6.1), wodurch er alle Eigenschaften von HotAgent Assembly erbt.

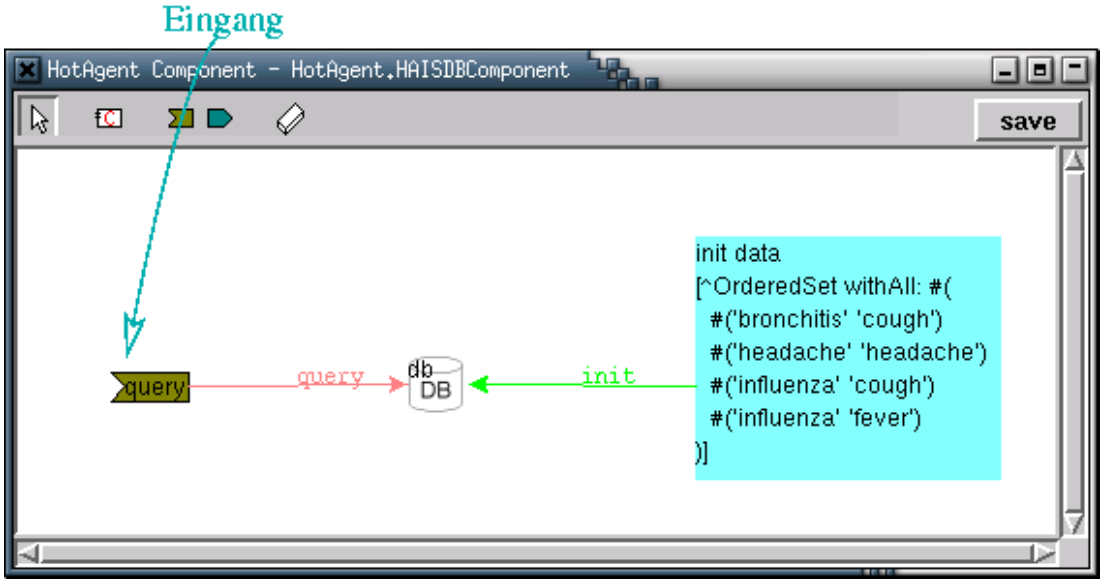

Abbildung 6.7: HotAgent Component Werkzeug

Abbildung 6.7 zeigt den HotAgent Component Editor. Komponenten können auf der Arbeitsfläche plaziert und miteinander verbunden werden. Die Werkzeugleiste am oberen Rand des Fensters bietet zwei zusätzliche Knöpfe. Der linke kann genutzt werden, um neue Eingänge zu erzeugen und der rechte dient zum Erzeugen von neuen Ausgängen für die gerade bearbeitete Komponente. Werden neue Ein- oder Ausgänge erzeugt, erscheint ein Eingabefenster, in dem ein Name und eine Beschreibung für den Ein- bzw. Ausgang angegeben werden können. Diese Ein- und Ausgänge können wie normale Komponenten mit anderen Komponenten verbunden werden. Sie bieten einen Aus- bzw. Eingang an, damit von ihnen bzw. zu ihnen eine Verbindung gezogen werden kann. Durch die Nutzung dieses Werkzeugs kann ein Entwickler Komponenten erstellen, ohne daß er viel über das zugrundeliegende Komponentenmodell wissen muß.

Um die neu erstellte Komponente mit festen Daten zu konfigurieren (*K5*), kann eine Data Komponente genutzt werden. Diese Datenkomponente interpretiert einen

Anweisungsblock während der Aktivierung. Dieser Anweisungsblock kann beliebige Smalltalk Anweisungen enthalten. Nach dem Auswerten dieses Blocks wird das Ergebnis an dem Ausgang Data bereitgestellt.

> Um das Einstellen einer Komponente zu demonstrieren, wird eine Data Komponente (init data) genutzt, um eine Data Base Komponente (db) zu initialisieren. Abbildung 6.7 zeigt einen möglichen Block (der init data Komponente) mit Krankheiten und deren Symptomen zu Initialisierung der Datenbank (db Komponente). Dazu ist der Ausgang data der init data Komponente mit dem init Eingang der db Datenbankkomponente verbunden. Sind mehrere Data Komponenten notwendig, definiert der Name der Komponente die Aufrufreihenfolge. Die im Alphabet zuerst vorkommende Data Komponente wird als erstes ausgewertet. Gleiches gilt auch, falls von einem Ausgang mehrere Verbindungen ausgehen. Auch hier wird die im Alphabet zuerst vorkommende Verbindung als erste aktiviert. Die neue Komponente hat einen Eingang query, der mit dem Eingang query der db Komponente verbunden ist. Dadurch wird eine Anfrage an die Datenbank, an die eigentliche Datenbankkomponente, weitergeleitet.

Um die neu erstellte Komponente zu identifizieren, ist es möglich, ein spezielles Piktogramm zu definieren, das in den Editoren dargestellt wird. Zusätzlich wird ein Eingabefenster angeboten, um den Namen, die Kategorie und die Beschreibung der neuen Komponente zu spezifizieren.

Wird eine Komponente aus anderen erstellt, wird das Kriterium der *Sichtbarkeit (V7)* (s. Kapitel 3.2) erfüllt. Wenn in einer Komponente mehrere andere Komponenten enthalten sind, wird die dadurch entstehende Anwendung strukturierter und dadurch auch übersichtlicher.

In Kapitel 5.3.4 wurde vorgeschlagen, Steuerungskomponenten aus mehrere Komponenten zusammenzubauen (Programmieren im Großen). Dies kann mit dem HotAgent Component Editor gemacht werden. In Kapitel 3.2 wird verlangt, daß die Möglichkeit bestehe, *Komponenten sowohl visuell als auch textuell zu entwickeln (V8)*. Wie dies visuell möglich ist, wurde in den vorangegangenen Abschnitten beschrieben. Zum textuellen Entwickeln (Programmieren im Kleinen) steht eine spezielle nicht-sichtbare Komponente Instruction Block bereit, in der ein Smalltalk Anweisungsblock definiert werden kann. Die Komponente bietet, fünf Eingänge und fünf Ausgänge, die vom Entwickler nach seinen Bedürfnissen genutzt werden können. Mit den Eingängen können Variable gesetzt werden, die bei der Auswertung des Blocks zur Verfügung stehen (`get`-Methoden) und mit `set`-Methoden können Ausgänge während der Interpretation aktiviert werden. Ein zusätzlicher Eingang do ist vorhanden, um die Interpretation des Anweisungsblocks anzustoßen.

Die Steuerungskomponente für den MASA ist mit zwei Instruction Block Komponenten erstellt worden. Der erste, der eine Anfrage für eine Datenbank Komponente erzeugt, sieht wie folgt aus:

```
[ :holder |
holder b: (Array with: #rating: with: holder a).
]
```

In der Variablen a werden die gefundenen Symptome zu einer Anfrage zusammengebaut und der Variablen b zugewiesen, die darauf den Ausgang b aktiviert.

Dadurch, daß die Wahl besteht, eine neue Komponenten aus mehreren vorhandenen zusammenzusetzen oder sie textuell zu programmieren, kann für die umzusetzende Aufgabe die beste Variante gewählt werden. Dies unterstützt die Forderung, daß *schwere mentale Operationen (V4)* zu vermeiden sind. Ob *schwere mentale Operationen* ganz vermieden werden können, hängt auch sehr von der Aufgabenstellung ab. Die Entwicklungs-Umgebung sorgt auf jeden Fall dafür, daß der Entwickler möglichst gut unterstützt wird.

6.3 HotAgent Test

Ein Kriterium für Entwicklungsumgebungen ist, daß eine *Testfunktionalität (V11)* (s. Kapitel 3.2) vorhanden ist. Dies wird durch das Programm HotAgent Test [Mar02a] sichergestellt. Auch dieses Werkzeug arbeitet ähnlich wie HotAgent Assembly (s. Abschnitt 6.1). Der Unterschied ist, daß die Arbeitsfläche nur aus dem Hauptteil besteht. Es können aber sowohl sichtbare als auch nicht-sichtbare Komponenten auf die Arbeitsfläche gesetzt werden. Die Werkzeugleiste am oberen Rand des Fensters hat auch zwei zusätzliche Knöpfe, die später beschrieben werden.

Ein Kriterium zum Testen (s. Kapitel 4.3) lautet, daß die Komponenten als *schwarze Kiste (T1)* getestet werden. Ein weiteres ist, daß nur die *normale Schnittstelle (T2)* genutzt werden darf. Der HotAgent Test spricht dafür jede Komponente als ganz normale Komponente über die definierten Ein- und Ausgänge an. Als weiteres muß zwischen dem Umfang der zu testenden Funktionalität unterschieden werden. Hier soll ein *Funktionstest (T3)* ausgeführt werden, wie ihn ein Komponentenentwickler durchführt.

Als Beispiel zur Demonstration von HotAgent Test wird die im vorigen Abschnitt erzeugte Datenbankkomponente getestet. Sie erhält

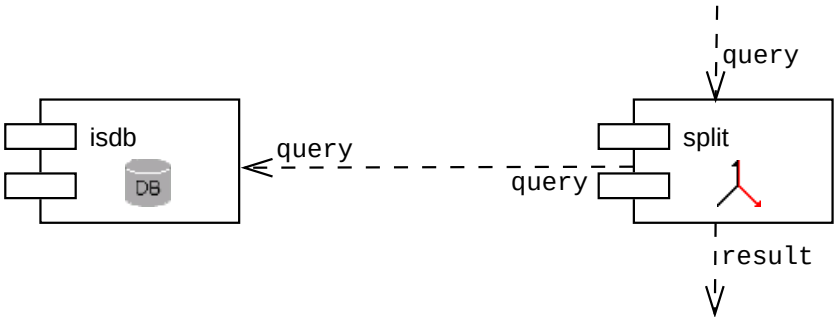

Abbildung 6.8: Komponentenzusammensetzung für den Testfall

den Exemplarnamen isdb. Sie ist bereits mit Datensätzen aus Symptomen und Krankheiten initialisiert, so daß nun ein geeigneter Testfall definiert werden muß. Dazu wird die Komponente Split mit Exemplarnamen split, wie in Abbildung 6.8 angegeben, benutzt. Um den Eingang query der Datenbank zu testen, wird eine Abfrage benötigt. Ein Anweisungsblock zur Erzeugung dieser Abfrage kann wie folgt aussehen:

```
[ #(#onlyObjects: #('cough')) ]
```

Diese Daten werden zu dem Eingang query der Komponente split geschickt. Diese leitet die Anfrage an die Komponente data base weiter, die die Anfrage bearbeitet. Das Ergebnis der Anfrage wird wieder an die Komponente split zurückgegeben, die das Ergebnis an dem Ausgang result bereitstellt. Um das Ergebnis zu testen, ist ein Testblock notwendig, der einen Wahrheitswert liefert. Wird der Block als wahr ausgewertet, gilt der Test als erfolgreich. Ein möglicher Block kann wie folgt aussehen:

```
[ :testData |
(testData includes: 'influenza') and:
[testData includes: 'bronchitis'] ]
```

In dem Beispiel werden Krankheiten mit dem Symptom `'cough'` erfragt. Als Antwort werden die Krankheiten `'influenza'` und `'bronchitis'` erwartet.

In diesem Beispiel ist es nur notwendig, eine einfache Liste zu testen. Es ist auch möglich, kompliziertere Werte, wie z.B. Ausprägung von Bäumen, zu testen. In einem Testblock ist jede Anweisung möglich. Dadurch kann flexibel jedes beliebige Objekt getestet werden.

Wie zuvor gezeigt, ist es nicht schwer, einen Testfall zu erzeugen. Jeder Testfall kann nach dem gleichen Schema erstellt werden:

1. Erzeugen aller Komponenten
2. Verbinden der Komponenten
3. Testdaten an einen oder mehrere Eingänge schicken
4. Testfall für jeweils einen Ausgang definieren

Soll ein Testfall mit HotAgent Test erstellt werden, müssen als erstes alle notwendigen Komponenten auf der Arbeitsoberfläche angeordnet und verbunden werden. In unserem Fall werden die Komponenten isdb und split, wie zuvor beschrieben verbunden.

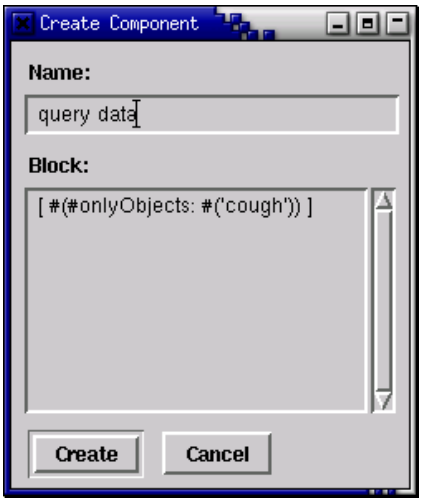

Abbildung 6.9:
Datenkomponente erzeugen

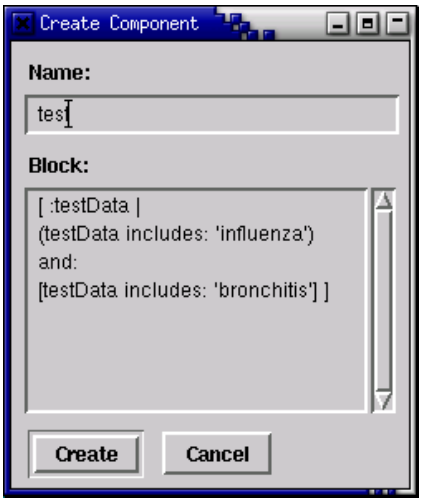

Abbildung 6.10:
Testkomponente erzeugen

Ist dies geschehen, müssen die Daten für den zu testenden Eingang erzeugt werden. Das HotAgent Test Werkzeug bietet dazu einen speziellen Knopf. Ähnlich zu einer normalen Komponente muß die Position ausgewählt werden. Danach erscheint ein Eingabefenster (s. Abb. 6.9), in dem der Exemplarname und ein Anweisungsblock zum Errechnen der Daten angegeben werden kann. Werden mehrere Datenkomponenten für einen Testfall benötigt, gibt der jeweilige Exemplarname die Reihenfolge der Aufrufe an. Der alphabetisch erste wird als erstes ausgeführt.

Ähnlich dazu existiert auch ein Knopf, um einen Testblock zu erzeugen. Nach Wahl einer Position erscheint wieder ein Eingabefenster (s. Abb. 6.10) in dem der Exemplarname und der Testblock eingetragen werden können.

Als letztes muß nur noch der Komponentenausgang result der Komponente split mit der Testkomponente verbunden werden. Dazu bietet die Testkomponente den Eingang test an.

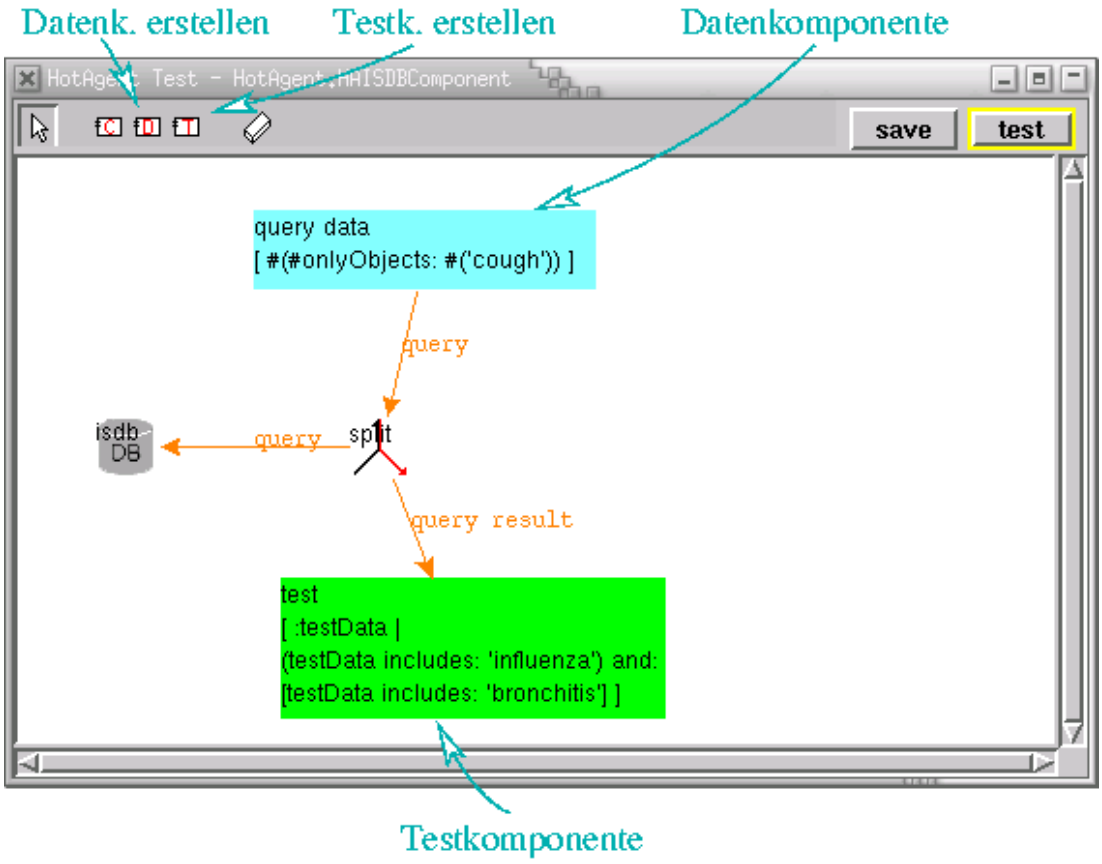

Abbildung 6.11: HotAgent Test Werkzeug

In Abbildung 6.11 ist der vollständige Testfall abgebildet. Wird der Test ausgeführt, ändert sich die Farbe der Testkomponente. Burnett *et al.* [BRCR01] erwähnen, daß Farben nützlich seien, um die verschiedenen Zustände während des Testens anzuzeigen. HotAgent Test benutzt einen dunklen Zyanhintergrund für den Testblock, wenn noch kein Test ausgeführt wurde. Dieser ändert sich zu Gelb, wenn der Test gerade aktiv ist. Ist der Test erfolgreich, wird der Hintergrund grün, ansonsten wird er rot.

Ein Kriterium für Komponentenmodelle ist, daß *Test- und Verifikationsmethoden (K8)* (s. Kapitel 2.2) mit einer Komponente gespeichert werden müssen. Dies wird auch durch HotAgent Test unterstützt, indem, wie in Kapitel 2.4.7 beschrieben, eine XML-Beschreibung mit der Komponente gespeichert wird. Dadurch wird auch erfüllt, daß eine Testspezifikation (*T5*) sprachunabhängig und leicht zu

lesen sein muß. Hinzu kommt, daß auf der Arbeitsfläche mehrere Testfälle (*T5*) für eine Komponente spezifiziert werden können.

Ein weiteres Kriterium ist, daß ein Testwerkzeug mit *Verklemmungen (T4)* umgehen und auch *Zeitverhalten (T4)* (s. Kapitel 4.3) behandeln kann. Auch diese Anforderungen werden durch HotAgent Test erfüllt, indem optional ein Zeitlimit für jeden Testfall angegeben wird. Wird dieses überschritten, so gilt der Test als fehlerhaft.

6.4 HotAgent Regression

Regressionstests (T6) sind ein weiteres Kriterium, das in Kapitel 4.3 gefordert wird. Regressionstests sind das wiederholte Testen nach Änderungen an dem Quelltext. Ist der Quelltext einer Komponente geändert worden, ist es notwendig zu zeigen, daß ihr Verhalten unverändert ist bzw. daß ein eventueller Fehler eliminiert wurde.

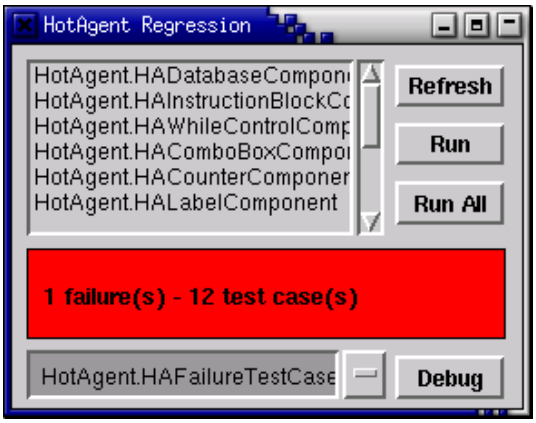

Abbildung 6.12: HotAgent Regression Werkzeug

HotAgent Regression [Mar02a] ist ein Werkzeug, um Regressionstests auszuführen. Es ist ähnlich zu dem *SUint test runner* [Cam] Werkzeug. Abbildung 6.12 zeigt das HotAgent Regression Werkzeug. Die Liste rechts oben in dem Fenster legt alle definierten Testfälle vor. Wird ein Testfall ausgewählt und der Knopf *Run* gedrückt, wird dieser ausgeführt. Wählt man den Knopf *Run All*, werden alle aufgelisteten Testfälle ausgeführt. Der farbige Bereich in dem Fenster symbolisiert den Status der Testfälle. Auch dieser ist am Anfang dunkelzyan. Er wird gelb, wenn ein oder mehrere Tests ausgeführt werden. Sind alle Testfälle erfolgreich, so wird es grün. Schlägt aber

einer fehl, so wird das Feld rot eingefärbt. In diesem Fall werden alle fehlerhaften Testfälle in einer Liste gesammelt, mit deren Hilfe das Werkzeug HotAgent Test aufgerufen werden kann, um den fehlerhaften Testfall zu untersuchen.

6.5 HotAgent Visualize

Ein andres Kriterium für Entwicklungsumgebungen ist die Unterstützung der *Programmvisualisierung (V12)* (s. Kapitel 3.2). Dies wird von HotAgent Visualize [MGM02b, MGM02a] unterstützt. Es visualisiert das gemessene Laufzeitverhalten eines Komponentenprogramms. Ehe es ein Programm visualisiert, führt es eine dynamische Analyse des Komponentenprogramms durch.

Die dynamische Analyse beginnt mit der Sammlung der Daten. Der *Visualisierungsbereich* (s. Kapitel 4.2.2) ist die Kommunikation zwischen den Komponenten, also muß die Kommunikation zwischen allen Komponenten aufgenommen werden.

Um Daten über eine erfolgte Kommunikation zu sammeln, müssen Änderungen an Teilen des Komponentenprogramms gemacht werden. Zu beachten ist, daß Änderungen an dem Quelltext möglichst gering gehalten werden. Es ist schwer sicherzustellen, daß Änderungen an den richtigen Stellen und fehlerfrei vorgenommen wurden. Ebenfalls müssen diese Änderungen wieder entfernt werden, wenn ein Programm ausgeliefert werden soll. Der Vorteil des HotAgent Komponentenmodells ist, daß alle Komponenten von einer Klasse abgeleitet sind und so nur Änderungen an genau dieser Klasse gemacht werden müssen. Zusätzlich kommt hinzu, daß für die Kommunikation genau eine Methode zuständig ist, so daß nur an dieser Methode Änderungen notwendig sind. Diese Tatsache reduziert die Gefahr von zusätzlichen Fehlern und verhindert auch, daß die Anwendung sehr im Laufzeitverhalten beeinträchtigt wird. Einige Programmiersprachen, wie z.B. auch Smalltalk, erlauben es, einzelne Klassen oder Methoden separat zu übersetzen. Vor Beginn der dynamischen Analyse wird genau dies zu Hilfe genommen, um die Kommunikationsmethode durch eine andere, die neben der Durchführung der Kommunikation diese auch protokolliert, zu ersetzen. Zum Protokollieren werden die Kommunikationsdaten in eine fortlaufende Liste gespeichert. Dies gewährleistet, daß das Laufzeitverhalten des Programms so wenig wie möglich beeinträchtigt wird. Ist die Analyse beendet, wird diese Methode einfach wieder durch die Originalmethode ersetzt.

Die während der Analyse gesammelten Daten bestehen aus dem Quellkomponentennamen, ihrem Ausgangsnamen, dem Zielkomponentennamen, ihrem Eingangsnamen und den transportierten Daten. Diese werden in eine Liste gespeichert, die

nach der Programmausführung analysiert werden kann. Ein Kriterium aus Kapitel 4.3 ist, daß ein Visualisierungswerkzeug *Filtermechanismen (T7)* anbieten soll. HotAgent Visualize bietet Filter an, Komponenten mit ihren Ein- und Ausgängen auszuschließen. Wird ein Filter definiert, ist es möglich, die Menge an Daten für die Visualisierung deutlich zu reduzieren. Es ist nicht sinnvoll, schon während der Ausführung des Programms zu Filtern, denn das Filtern ist zu zeitintensiv und würde die Laufzeit des Komponentenprogramms zu sehr verlängern.

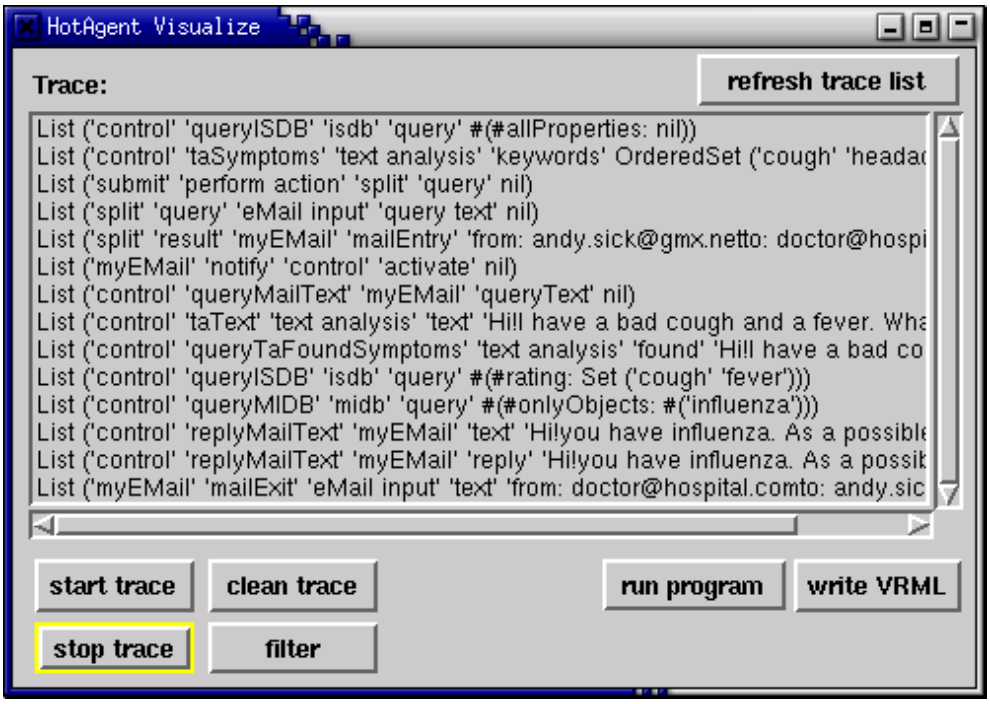

Abbildung 6.13: Analysesteuerung von HotAgent Visualize

Abbildung 6.13 zeigt die Analysesteuerung von HotAgent Visualize. In einer Liste werden alle aufgenommenen Kummunikationsdaten dargesellt. Durch Wahl der Knöpfe *start trace* und *stop trace* ist es jederzeit möglich, die Analyse während der Ausführung eines Komponentenprogramms zu starten und anzuhalten. Durch Wahl des Knopfes *filter* kann eine Liste der zu filternden Komponenten angegeben werden. Diese Liste setzt sich aus den Namen der zu filternden Komponenten zusammen. Soll z.B. die submit Komponente herausgefiltert werden, kann folgende Angabe gemacht werden `#( 'submit' )`. Umfangreichere Filtermöglichkeiten sind in dieser Version nicht enthalten, können aber leicht ergänzt werden. Im gleichen Fenster besteht noch die Möglichkeit, das zu analysierende Programm zu starten und nach der Analyse eine Visualisierung (VRML Quelltext [HW96]) zu erstellen.

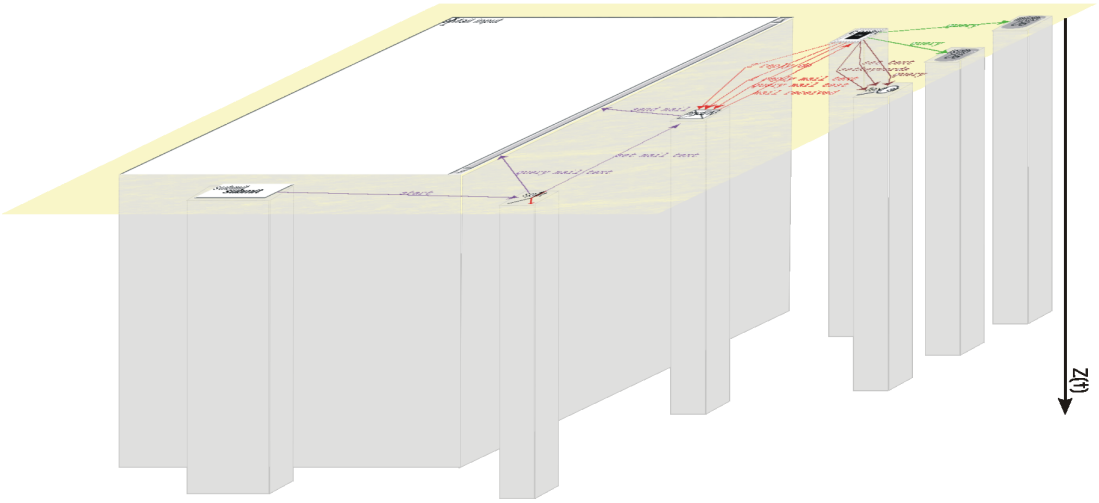

Abbildung 6.14: Visualisierung vom HotAgent Assembly Editor abgeleitet

Alle Komponentenprogramme, die visualisiert werden, wurden in der Regel mit HotAgent Assembly (s. Abschnitt 6.1) erstellt. Dort wird das Programm in zwei Dimensionen konstruiert, wobei die Komponenten als Piktogramme und die Kommunikation zwischen diesen als farbige Pfeile mit einer kurzen Beschreibung dargestellt werden. HotAgent Assembly (s. Abschnitt 6.1) bietet somit gleichzeitig eine statische Programmvisualisierung. HotAgent Visualize, das eine dynamische Visualisierung bietet, nutzt diese statische Darstellung und erweitert sie in der dritten Dimension (z-Achse nach unten), um die Laufzeit abzubilden, wie in Abbildung 6.14. Komponenten werden dabei als Quader dargestellt, deren Höhe (z-Ausdehnung) die Lebensdauer symbolisiert. Die Position in der x/y-Ebene ist dieselbe wie in der statischen Visualisierung. Da die Darstellung ähnlich der visuellen Programmierung (s. Abb. 6.15) ist, muß der Programmierer sich nicht erneut mit der Struktur des Programms auseinandersetzen, sondern kann die Kommunikation leicht erkennen (s. auch Giesl [Gie01]).

Wird die schematische Darstellung (s. Abbildung 6.15) der Visualisierung betrachtet, fällt auf, daß auf der Oberfläche jedes Quaders ein Piktogramm hinzugefügt ist, das auch bei der visuellen Programmierung benutzt wurde. Dadurch wird die Visualisierung der Darstellung der visuellen Programmierung noch ähnlicher.

Das Laufzeitverhalten von einem Komponentenprogramm wird gezeigt, indem Pfeile in der z-Achse von oben nach unten relativ zu der gemessenen Ausführungszeit dargestellt werden (s. Abb. 6.15). Zur besseren Gestaltung wird in Kapitel 4.3 vorgeschlagen, die Visualisierung farbig *(T9)* zu machen. Deshalb haben die Pfeile die gleiche Farbe, wie die Pfeile, die während der visuellen Programmierung definiert wurden. Die Position der Pfeile auf der z-Achse, d.h. den Höhen, symbolisiert den relativen Zeitpunkt der Kommunikation während der gemes-

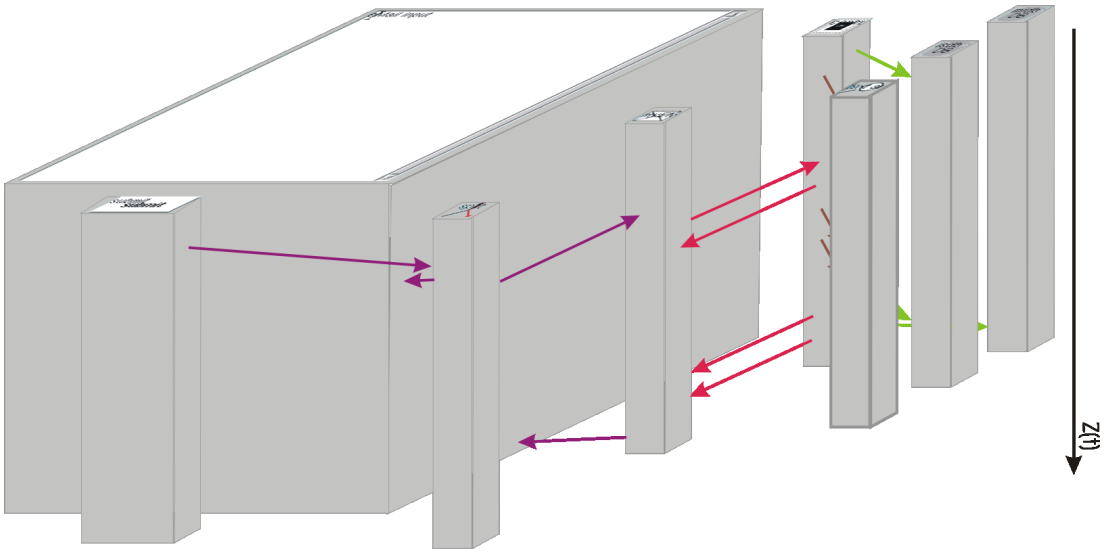

Abbildung 6.15: Schematische Visualisierung

senen Ausführung. Erscheint ein Pfeil unter einem anderen, bedeutet das, daß die Kommunikation später stattgefunden hat. Der oberste Pfeil repräsentiert die erste Kommunikation.

Abbildung 6.16 zeigt die eigentliche Visualisierung. Sie ist mit dem VRML-Betrachter *COSMO-player* erstellt.

In dem Beispiel (s. Abb 6.16) hat der Entwickler purpurfarbene Pfeile für die Kommunikation zwischen der Benutzungsoberfläche, rote Pfeile für die Kommunikation zwischen myEMail und control, grün zwischen control und Datenbank und braun zwischen der control Komponente und der text analysis Komponente gewählt.

Die Initialisierung beginnt mit dem obersten grünen Pfeil zwischen der control und der isdb, um alle Symptome zu erfragen, um sie dann mit dem braunen Pfeil an die text analysis Komponente zu schicken. Mit Betätigung des Knopfes *Submit* beginnt die eigentliche Verarbeitung. Dies wird durch den ersten purpurfarbenen Pfeil verdeutlicht. Der zweite purpurfarbene Pfeil zwischen der split und der eMail input Komponente erfragt die elektronische Post und schickt diese mit dem nächsten Pfeil an die myEMail Komponente. Die nächsten beiden roten Pfeile zwischen der myEMail und der control Komponente tauschen den eigentlichen Brieftext aus. Der darauf folgende braune Pfeil zwischen der control und der text analysis beinhaltet den Brieftext, den die text analysis Komponente untersuchen soll. Ist dies erledigt, werden die gefundenen Schlüsselworte übermittelt und mit den grünen

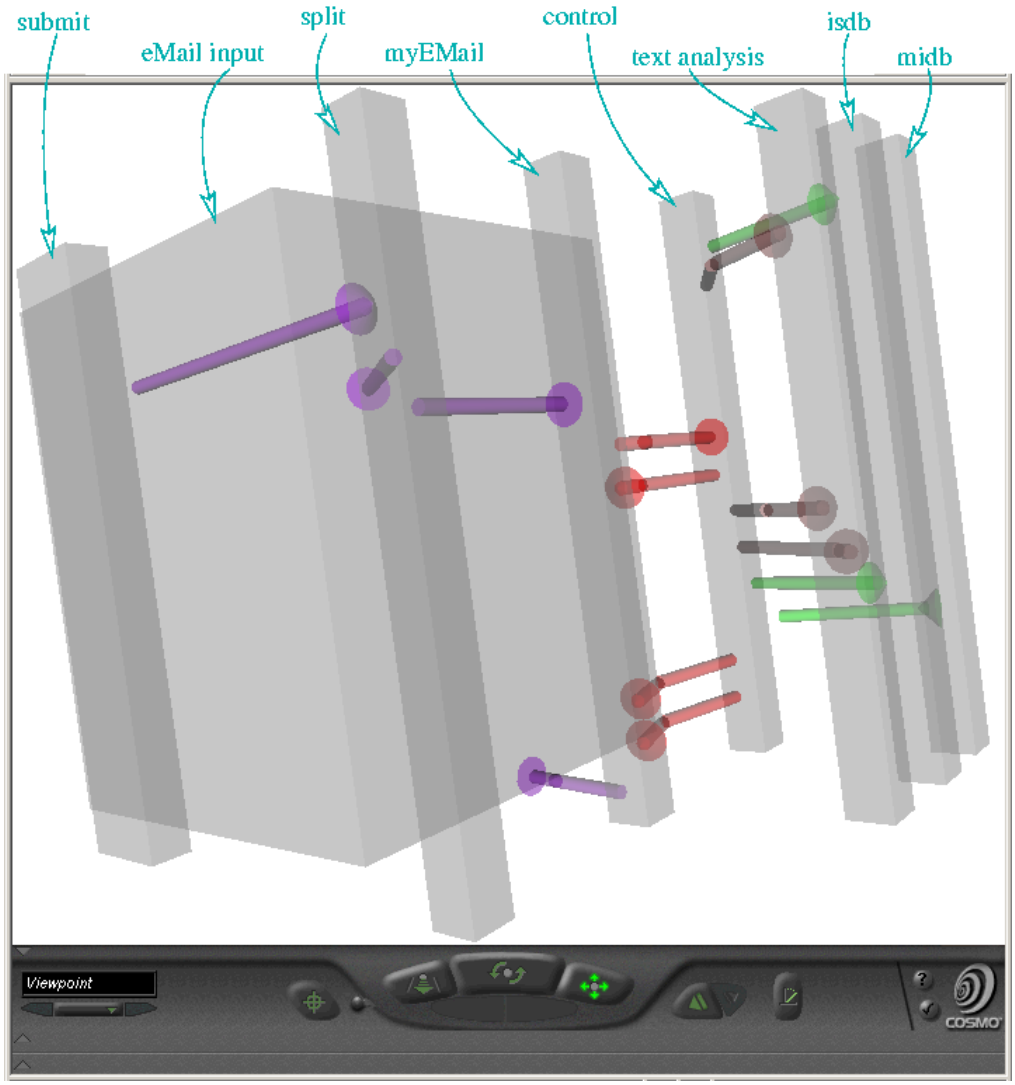

Abbildung 6.16: Programmvisualisierung

Pfeilen die isdb und midb nach einer passenden Krankheit und einem möglichen Medikament befragt. Die letzten drei Pfeile dienen zum Erstellen einer entsprechenden Antwort.

Zusätzlich zu der Information der Ausführungsreihenfolge, ist es möglich, weitere Informationen über eine Kommunikation zu erhalten. Wird ein Pfeil ausgewählt, so erscheinen Informationen über den Ausgangsnamen, den Eingangsnamen und über die transportierten Daten, wie in Abbildung 6.17 gezeigt. Es ist vorgesehen diese Daten nur auf Nachfrage zu präsentieren, denn andernfalls könnte die Darstellung überladen und unübersichtlich wirken.

Ein Kriterium ist, daß sich die Ansicht (*T11*, *K6*) individuell einstellen lassen

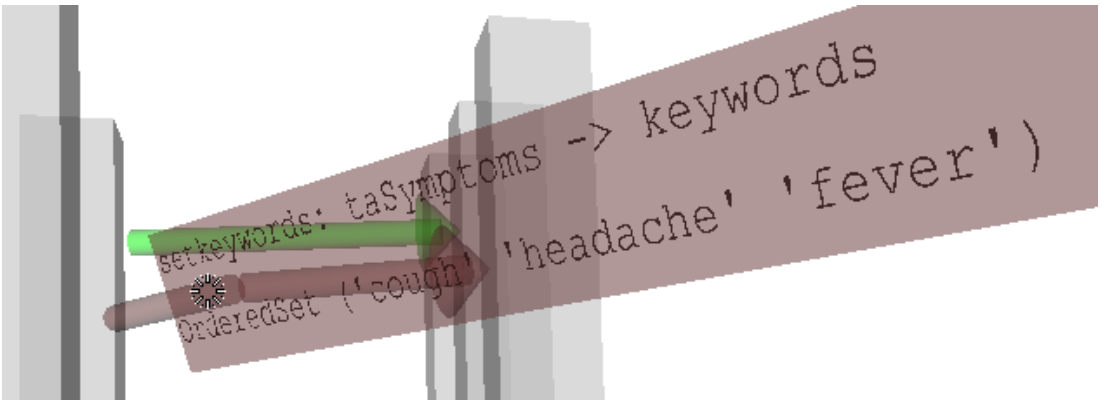

Abbildung 6.17: Informationen über eine Kommunikation

muß. Deswegen bietet HotAgent Visualize die Möglichkeit, die Analysedaten zu filtern, um so mehrere Ansichten zu erzeugen. Es ist auch möglich, die dreidimensionale Ansicht zu drehen und so einzelne Teile eines größeren Programms klar zu untersuchen. HotAgent Visualize ist sowohl in der Lage, den vollständigen Programmablauf als auch nur Teile davon vergrößert darzustellen.

Zusätzlich wird angeboten, verschiedene *Hierarchien (T10)* darzustellen. Dies ist möglich, indem entweder von dem gesamten Programm oder von einer beliebigen, zusammengesetzten Komponente eine Visualisierung erstellt wird.

Suchmöglichkeiten (T8) sind auch wichtig. Dadurch, daß HotAgent Visualize eine VRML Darstellung erzeugt, hängt es von dem VRML-Betrachter ab, wie gut dort die Suchmöglichkeit nach einzelnen Objekten und Texten ist.

6.6 HotAgent Center

Die zentrale Steuerung der einzelnen Werkzeuge übernimmt das Programm HotAgent Center. Es bietet eine Benutzungsoberfläche, um alle einzelnen Werkzeuge zusammenzufassen. Neben der Möglichkeit, die einzelnen Werkzeuge aufzurufen, werden auch alle zur Verfügung stehenden und erstellten Komponenten, die Komponentenprogramme und die Testfälle aufgelistet (s. Abb. 6.18).

Um zu zeigen, daß die HotAgent Entwicklungsumgebung auch für größere Programme geeignet ist, wurde der HotAgent Center vollständig aus Komponenten mit Hilfe von HotAgent erstellt. Die einzelnen Werkzeuge bilden jeweils eine Komponente, die mit eingebunden wurde.

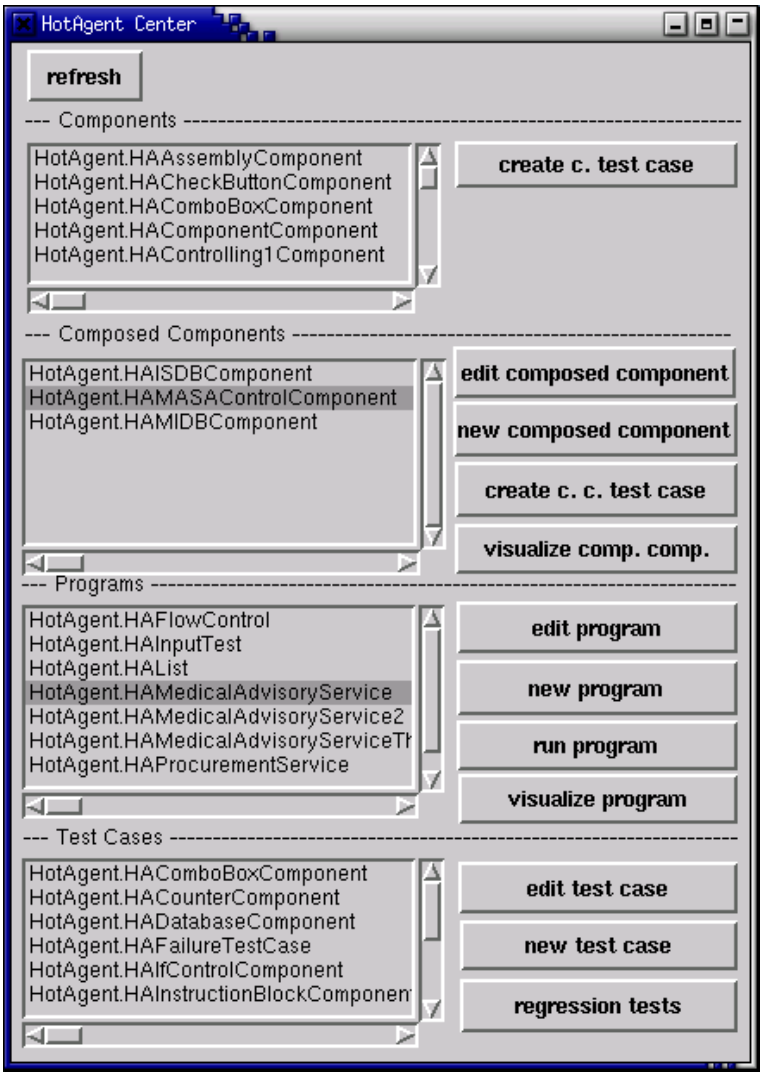

Abbildung 6.18: HotAgent Center

6.7 Zusammenstellung der Kriterien für HotAgent

In diesem Kapitel werden nochmals alle Kriterien, die in den Kapiteln 2.2, 3.2 und 4.3 aufgestellt wurden, aufgelistet und die Bedeutung zu HotAgent dargestellt. Dazu wird jeweils auch auf das Kapitel verwiesen, in dem ein Kriterium näher besprochen wird.

K1	modulares Design	ja	Kap. 2.4
K2	Selbstbeschreibung	ja	Kap. 2.4.1

K3	globaler Namensraum	nicht erzwungen	Kap. 2.4.1
K4	zweigeteilter Entwicklungsprozeß	ja	Kap. 6
K5	Anwendungen zusammenbauen	ja	Kap. 2.4.4
			Kap. 6.2
K6	verschiedene Ansichten	ja	Kap. 6.1
			Kap. 6.5
K7	Wiederverwendung durch Verweis	ja	Kap. 2.4.2
K8	Testmethoden	ja	Kap. 6.3
K9	klarer Aufgabenbereich	nicht erzwungen	Kap. 6.2
K10	geringe Komplexität	ja	Kap. 2.4.3
K11	Sprachunabhängigkeit	nein	Kap. 2.4
V1	Nähe der Abbildung	ja	Kap. 6.1
V2	Viskosität	ja	Kap. 6.1
V3	versteckte Abhängigkeiten	ja	Kap. 6.1
V4	schwere mentale Operationen	ja	Kap. 6.2
V5	erzwungenes Vorausdenken	ja	Kap. 6.1
V6	sekundäre Notation	ja	Kap. 6.1
V7	Sichtbarkeit	ja	Kap. 6.1
		ja	Kap. 6.2
V8	Komp. visuell und textuell	ja	Kap. 6.2
V9	Komponenten- und Anwendungsentw.	ja.	Kap. 6
V10	Dokumentation direkt verfügbar	ja	Kap. 6.1
V11	Testfunktionalität	ja	Kap. 6.3
V12	Programmablaufvisualisierung	ja	Kap. 6.5
V13	einheitliches Aussehen	ja	Kap. 6
T1	schwarze Kiste Tests	ja	Kap. 6.3
T2	normale Schnittstelle	ja	Kap. 6.3
T3	benötigte Funktionalität testen	nicht erzwungen	Kap. 6.3
T4	Verklemmungen/Zeitverhalten	ja	Kap. 6.3
T5	Testspezifikation	ja	Kap. 6.3
T6	Regressionstests	ja	Kap. 6.4
T7	Filtermechanismen	ja	Kap. 6.5
T8	Suchmöglichkeiten	externe Apl.	Kap. 6.5
T9	Farbe	ja	Kap. 6.5
T10	Hierarchie	ja	Kap. 6.5
T11	Einstellen der Ansicht	ja	Kap. 6.5

Kapitel 7

Gebrauchstauglichkeit von HotAgent

Nachdem in den vorigen Abschnitten die HotAgent Entwicklungsumgebung vorgestellt wurde, stellt sich nun die Frage, ob sie auch wirklich geeignet ist, um Anwendungen zu entwickeln. Dazu soll in diesem Kapitel die Gebrauchstauglichkeit anhand einer experimentellen Untersuchung erörtert werden.

Zuerst ist zu klären, was unter Gebrauchstauglichkeit in Bezug auf ein Computerprogramm verstanden wird. Ivory und Hearst [IH01] definieren:

> Usability is the extent to which a computer system enables users, in a given context of use, to achieve specified goals effectively while promoting feeling of satisfaction.

Nach Oppermann *et al.* [OMRK92] sollten Untersuchungen zur Gebrauchstauglichkeit die folgenden drei Fragestellungen berücksichtigen:

Aufgabenbewältigung: Die erste Frage sei, inwieweit ein Programm den Benutzer bei der Erledigung einer Aufgabe unterstützt. Ziel sei eine *menschengerechte* Arbeitsweise mit dem System.

Benutzung: Wichtig bei einem Programm sei, daß es leicht erlernbar ist. Die Arbeit könne aber auch deutlich erleichtert werden, wenn ein Programm individuell an einen Benutzer bzw. dessen Tätigkeiten anpaßbar ist.

Funktionalität: Wie weit ein System den Benutzer aufgabenrelevant unterstütze, wird als Funktionalität bezeichnet. Dabei wird untersucht, inwieweit die Arbeitsabläufe hinreichend in der Software abgebildet sind.

Es sei nach Oppermann *et al.* [OMRK92] angebracht, bei folgenden drei Gelegenheiten eine Untersuchung zur Gebrauchstauglichkeit durchzuführen.

- Während der Systemgestaltung ließen sich Schlüsse ziehen, was noch getan werden muß, um ein gutes Endprodukt zu erlangen.
- Sollten Fortschritte in verschiedenen Versionen eines Programms sichtbar gemacht werden, müssen die einzelnen Systemausprägungen bewertet werden.
- Stünden unterschiedliche Programme für eine Aufgabe zur Auswahl, so kann eine Bewertung zum Zwecke eines Systemvergleichs oder einer Marktforschung durchgeführt werden.

In diesem Kapitel geht es darum, festzustellen, wie gut HotAgent arbeitet und was eventuell noch verbessert werden muß, also eine Untersuchung während der Systemgestaltung.

7.1 Vorgehensweise

Zur Untersuchung der Gebrauchstauglichkeit gibt es viele Methoden. Ivory und Hearst [IH01] haben 75 von diesen für graphische Benutzungsoberflächen und 57 für Web-Oberflächen untersucht. Die wichtigsten sollen kurz hier vorgestellt werden.

Wandmacher [Wan93] gibt zwei generelle Unterteilungen dieser Methoden an. Eine quantitative Erfassung messe die Arbeit, die in einer gewissen Zeit von einem Benutzer mit einen System geleistet wird. Diese Art von Erfassungen sei aber unzuverlässig, da zu viele Randbedingungen einfließen, wie z.B. die Geübtheit eines Benutzers. Die qualitativen bzw. empirischen Methoden ziehen Beurteilungen von mehreren Benutzern oder Experten zu Rate, die ein System getestet haben. Diese sind nach Nielsen und Mack [NM94] die am häufigsten genutzten Methoden.

Oppermann *et al.* [OMRK92] gliedern die empirischen Methoden wieder in drei Gruppen:

Befragung: Diese Methode basiere auf einer mündlichen oder schriftlichen Befragung von Benutzern und sei die am wenigsten aufwendige. Die Benutzer gehen entweder mit eigener Initiative an ein Programm heran, was eine größere Erfahrungsgrundlage benötigt, oder bekommen eine Aufgabe gestellt, die sie mit diesem Programm lösen sollen. Die zweite Herangehensweise sei dabei besser zu kontrollieren, da sie durch die Aufgabenstellung

beeinflußt werden kann. Ist die Phase des Programmtestens abgeschlossen, werde die zuvor erwähnte Befragung durchgeführt. Die Fragen müssen aber gut geprüft werden, damit ein brauchbares Ergebnis entsteht.

Nielsen und Mack [NM94] untersuchten, daß eine Person durchschnittlich 35% der Probleme eines Programms finde. Drei Personen hingegen fänden schon 71% der schwerwiegenden und 59% der leichten Probleme. Ideal sei es, wenn eine Gruppe aus drei bis fünf Personen zustande käme, da eine größere Gruppe nicht viel mehr Nutzen bringe, was Abbildung 7.1 verdeutlicht.

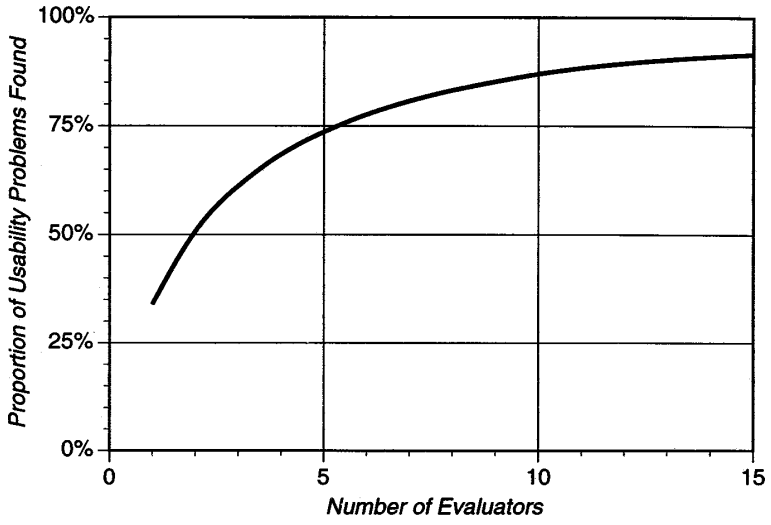

Abbildung 7.1: Gefundene Probleme und Tester aus [NM94] Seite 33

Nach Wandmacher [Wan93] sei diese Art von Untersuchung abhängig von den Kenntnissen und Erfahrungen der Beurteiler. Deshalb solle auf eine ausgewogene Zusammenstellung einer Gruppe geachtet werden.

experimentelle Evaluation: Werde ein Benutzer bei seiner Arbeit an einem Programm beobachtet, wird von experimenteller Evaluation gesprochen. Dabei sei interessant, was der Benutzer am Bildschirm mit der Maus macht und welche Tastatureingaben er vornimmt. Diese Beobachtungen könnten sowohl offen als auch verdeckt vorgenommen werden. Sie seien aber sehr abhängig von Faktoren wie der Aufgabe, der Motivation des Benutzers und der Tageszeit.

Leitfadenorientierte Evaluation: Hier wird ein Programm durch Experten geprüft. Dabei werden software-ergonomische Fragenkataloge herangezogen, die sich in dem Grad der Detailliertheit stark unterscheiden können. Nach Wandmacher [Wan93] seien die Grundlagen für diese Kataloge entweder Standards für Benutzungsoberflächen, die sehr allgemein gehalten sind,

oder Entwurfsrichtlinien, die auch Gestaltungsempfehlungen und Beispiele enthalten.

Zur Untersuchung von HotAgent wird die Methode der Befragung eingesetzt. Diese ist diejenige, die am einfachsten mit denen an der Universität vorhandenen Mitteln umgesetzt werden kann, wie im nächsten Abschnitt erläutert wird.

7.2 Durchführung

Die Befragung der Gebrauchstauglichkeit von HotAgent fand im Rahmen eines Grundstudiumseminars zum Thema *visuell-unterstütztes Programmieren* im Sommersemester 2002 statt. Durch die Vorträge der Studenten wurde ein Grundwissen in diesem Thema geschaffen, so daß sich die letzte Aufgabe mit der Befragung zur Gebrauchstauglichkeit beschäftigen konnte.

Zwei Studenten waren für die Befragung zuständig. Sie werden im folgenden Text zur besseren Unterscheidung Auswerter genannt. Die Pflichten der Auswerter waren das Formulieren der Aufgabe, das Erstellen eines Fragebogens für HotAgent und dann im Anschluß das Auswerten.

Die Aufgabe wurde vorgegeben, so daß die Auswerter nur noch ein geeignetes Aufgabenblatt erstellen mußten. Es sollte ein Programm zum Verschicken von verschlüsselter oder unverschlüsselter elektronischer Post aus mehreren Komponenten zusammengesetzt werden. Die Aufgabe gliederte sich in drei Teile.

1. Zuerst sollte eine zusammengesetzte Komponente gebaut werden, die mit dem Schlüssel 1 einen Text verschlüsseln kann. Diese Komponente sollte jeweils nur einen Ein- und Ausgang anbieten, wie Abbildung 7.2 zeigt. Dazu sind die Komponenten Data Block und MessageCodec, wie deren Piktogramme verdeutlichen, verwendet worden.

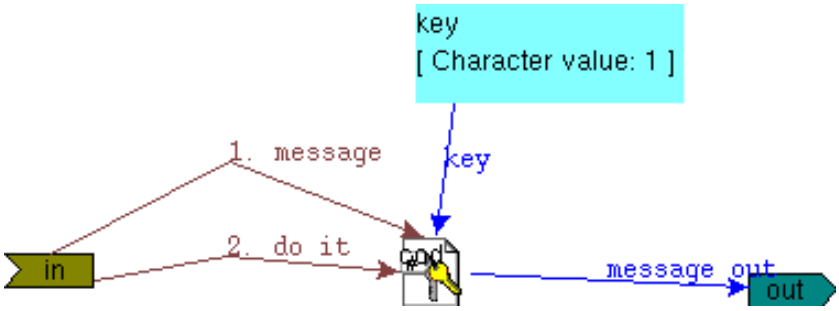

Abbildung 7.2: Zusammengesetzte Komponente zum Verschlüsseln

2. Die zweite Aufgabe bestand darin, einen Testfall für diese Komponente zu erstellen, mit dem das richtige Verhalten getestet werden sollte.

3. Der dritte Teil galt der Erstellung des eigentlichen Programms. Es sollte ein Eingabefeld für den Text, einen Auswahlknopf zum Verschlüsseln und einen Knopf zum Verschicken von Post anbieten. Durch Benutzung der zuvor erstellten Komponente zur Verschlüsselung ist das in Abbildung 7.3 dargestellte Programm eine mögliche Lösung. Die verwendeten Komponenten lassen sich wieder an deren Namen und deren Piktogramme identifizieren.

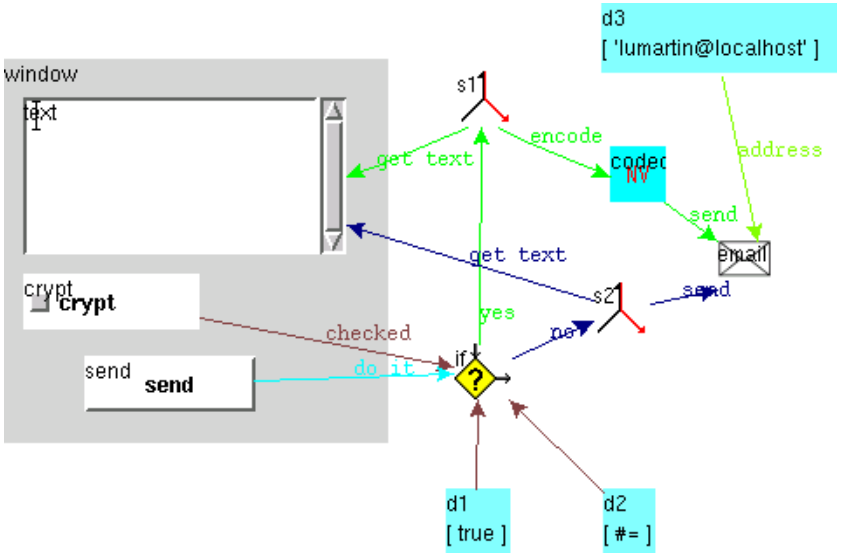

Abbildung 7.3: Programm elektronische Post

Das Aufgabenblatt bestand aus der Beschreibung der drei Teilgebiete und einer Zusammenfassung der Komponenten, die für die Lösung der Aufgabe nützlich sind (s. Anhang B).

Die Studenten Maruhn und Kofink [MK02] übernahmen als Auswerter die Erstellung des Fragebogens (s. Anhang C). Dieser umfaßte acht verschiedene Themengebiete aus denen Fragen gestellt wurden. Die Fragen konnten anhand einer Skala mit sieben Unterteilungen beantwortet werden. Am Ende jedes Themengebietes wurde die Möglichkeit geboten, einen Kommentar zu geben.

Vor der Durchführung der Untersuchung wurde von mir eine kleine Einführung in die einzelnen Werkzeuge von HotAgent gegeben. Dabei wurde u.a. angesprochen, wie man Komponenten erstellt, diese verbindet und Testfälle definieren kann.

An der Untersuchung nahmen 16 Studenten, die im Folgenden als Tester bezeichnet werden, des Faches (Wirtschafts-)Informatik aus dem zweiten und vierten

Semester teil. Keiner von ihnen hatte Erfahrung in der Bewertung von Software, und nur wenige hatten schon vorher mit visueller Programmierung zu tun. Zur Bewältigung der Aufgabe standen den Testern die studentischen Unix-Rechner zur Verfügung, an denen sie beliebig lange arbeiten durften. Von den 16 Testern, die selbständig die Bewertung erstellen sollten, löste einer die Aufgabe komplett und zwei teilweise. Wir stellten den Studenten frei, den Fragebogen anonym oder mit Namen abzugeben. Daraus ergab sich leider auch, daß wir drei Fragebögen von Testern bekamen, die sich HOTAGENT nicht angeschaut hatten. Abbildung 7.4 zeigt, wieviel Zeit für die Untersuchung von HOTAGENT in Anspruch genommen wurde. Der Tester, der die Aufgabe löste, benötigte 164 Minuten.

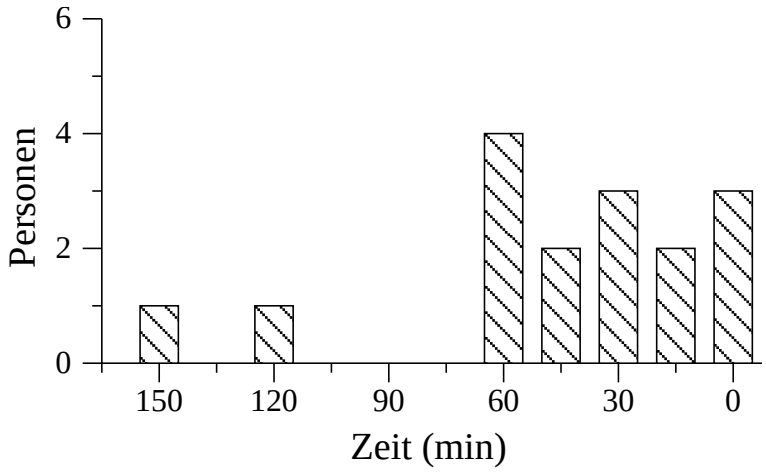

Abbildung 7.4: Benötigte Zeit bei der Untersuchung

7.3 Fazit

Die Auswertung der Fragebögen ergab nach Maruhn und Kofink [MK02], daß HOTAGENT übersichtlich und flexibel ist. Hinzu kommt aber, daß es nicht intuitiv bedienbar und frustrierend ist.

Im folgenden werden einige der interessantesten Ergebnisse vorgestellt, die bei der Befragung zustande kamen (s. auch Maruhn und Kofink [MK02]). Die Antworten des Testers, der die Aufgabe vollständig gelöst hat, soll zum Vergleich besonders (mit gekreuzter Schraffur) hervorgehoben werden, da davon ausgegangen werden kann, daß er die größte Erfahrung mit HOTAGENT hat:

- Läßt sich HOTAGENT einfach bedienen?

  Die Bedienbarkeit (s. Abb. 7.5) von HOTAGENT wird als schwierig eingestuft. Dies läßt sich sicher daraus erklären, daß es keine Hilfe zu HOTAGENT

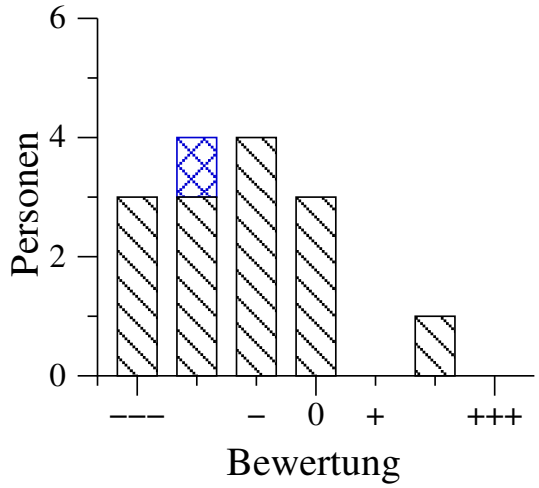

Abbildung 7.5: Läßt sich HotAgent einfach bedienen?

selbst gab. Die Tester waren vollständig auf sich selbst angewiesen. Bei einer produktreifen Version mit Hilfen sähe dieser Punkt sicher ganz anders aus.

- Verwendet HotAgent einheitliche Bezeichnungen?

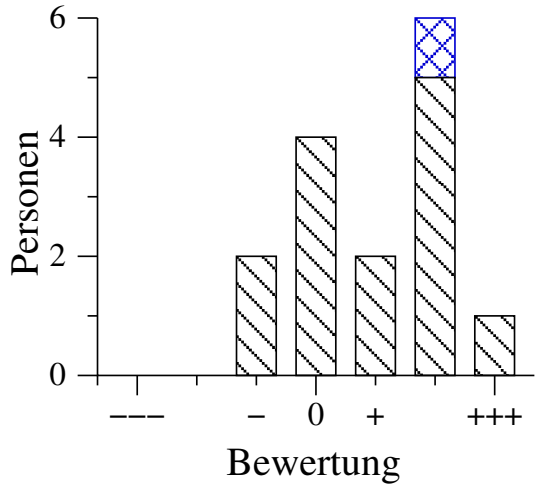

Abbildung 7.6: Verwendet HotAgent einheitliche Bezeichnungen?

Diese Fragestellung brachte ein sehr gutes Ergebnis (s. Abb. 7.6). Es läßt sich daraus erklären, daß in allen Werkzeugen, die HotAgent bereitstellt, einheitliche Bezeichnungen verwendet werden.

- Besitzt HotAgent eine einheitliche Gestaltung?

  Positiv ist auch die einheitliche Gestaltung (s. Abb. 7.7) der einzelnen Werkzeuge aufgefallen. Alle Werkzeuge von HotAgent verwenden die gleichen

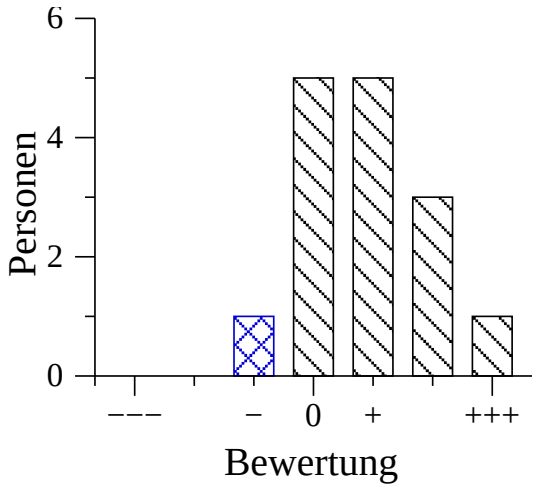

Abbildung 7.7: Besitzt HOTAGENT eine einheitliche Gestaltung?

Symbole und auch weitgehend die gleichen Dialoge.

- Läßt sich die Bildschirmdarstellung von HOTAGENT individuell einstellen?

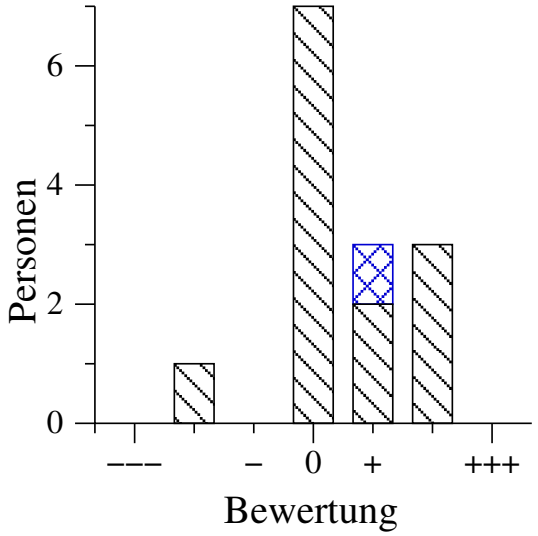

Abbildung 7.8: Läßt sich die Bildschirmdarstellung von HOTAGENT individuell einstellen?

Die allgemein positive Antwort (s. Abb. 7.8) auf diese Frage läßt sich auf die Anpassungsfähigkeit der Arbeitsfläche von HOTAGENT zurückführen. Sie bietet u.a. an, den Vergrößerungsfaktor einzustellen und die anzuzeigenden Elemente zu bestimmen. Daneben lassen sich auch die Details in mehreren Stufen festlegen.

- Läßt sich die Arbeit mit HOTAGENT in kurzer Zeit erlernen?

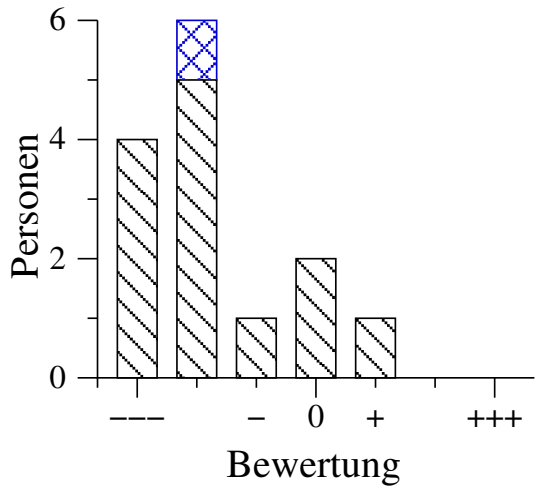

Abbildung 7.9: Läßt sich die Arbeit mit HotAgent in kurzer Zeit erlernen?

> Die Benutzer stuften HotAgent als nicht intuitiv bedienbar ein (s. Abb. 7.9). Das läßt sich wieder dadurch erklären, daß es kein Handbuch und auch keine Kontexthife gab. Um das auszugleichen, hätte die Einführung ausführlicher sein müssen.

Wie erwähnt, hat die Untersuchung ergeben, daß HotAgent übersichtlich, flexibel, nicht intuitiv bedienbar und frustrierend ist. Im ganzen ein nicht so gutes Ergebnis. Woran liegt das?

Bei der Durchführung wurde erkannt, daß die Aufgabenstellung zu kompliziert war, so daß sich viele der Tester überfordert fühlten. Bei einer Wiederholung dieser Befragung müßte daher die Aufgabenstellung überarbeitet werden, und es müßte eine ausführlichere Einführung in HotAgent gegeben werden. Ursache war, daß die Auswerter und ich noch unerfahren in diesem Thema waren.

Daß HotAgent noch ein Prototyp ist, hat sich auch an den Ergebnissen erkennen lassen. An vielen Stellen wurde das Fehlen einer Hilfefunktion von HotAgent sichtbar. Bei einem produktreifen Programm wäre dieses Manko behoben.

Ein weiterer Gesichtspunkt ist, daß die Untersuchung mit geringem Aufwand durchgeführt werden mußte. Mit den Mitteln an einer Universität ist es nicht möglich, ein Testlaboratorium aufzubauen und so eine geeignetere Testumgebung zu schaffen. Die Untersuchung brachte dennoch wertvolle Hinweise und Ergebnisse mit denen HotAgent noch verbessert werden kann.

Kapitel 8

Andere Entwicklungsumgebungen

In diesem Kapitel werden Entwicklungsumgebungen vorgestellt, die ähnliche Aufgaben wie HotAgent haben. Dazu werden die Kriterien aus den Abschnitten 2.2, 3.2 und 4.3 zum Vergleich herangezogen. Java Studio (s. Abschnitt 8.1), Visual Age for Java (s. Abschnitt 8.2) und JBuilder (s. Abschnitt 8.3) werden vorgestellt, da sie kommerzielle Entwicklungsumgebungen aus dem Java-Umfeld sind. AgentSheets (s. Abschnitt 8.4) hat seine Ursprünge im akademischen Bereich. SAM (Solid Agents in Motion) (s. Abschnitt 8.5) präsentiert eine dreidimensionale Programmierung. Prograph ist eine Datenflußsprache und wird in Abschnitt 8.6 vorgestellt. Abschließend wird wieder ein kommerzielles System Aonix SELECT Component Factory (s. Abschnitt 8.7) betrachtet, daß speziell für die komponentenbasierte Entwicklung gedacht ist.

In der Beschreibung der Programme wird jeweils auf einige der vorgestellten Kriterien Bezug genommen. Eine vollständige Auflistung aller Kriterien ist im Anschluß zu finden. Dabei werden folgende Abkürzungen verwendet: SUN Java Studio – JS, IBM Visual Age – VA, Borland JBuilder – JB, SAM (Solid Agents in Motion) – SAM, AgentSheets – AS, Prograph – PG, Aonix SELECT Component Factory – CF, ja – j, nein – n, nicht erzwungen – ne, separate Datei – sD, separate Klasse – sK, nicht anwendbar – -, nicht bekannt – ?.

Kriterium		JS	VA	JB	AS	SAM	PG	CF
K1	modulares Design	ne	ne	ne	j	?	j	ne
K2	Selbstbeschreibung	sK	sK	sD	?	?	?	j
K3	globaler Namensraum	n	n	n	?	?	?	n
K4	zweigeteilter Entwicklungsprozeß	j	j	j	j	j	j	j
K5	Anwendungen zusammenbauen	j	j	j	j	j	j	j

Kriterium		JS	VA	JB	AS	SAM	PG	CF
K6	verschiedene Ansichten	j	j	j	j	j	j	j
K7	Wiederverwendung durch Verweis	j	j	j	?	?	?	j
K8	Testmethoden	n	n	n	?	n	?	n
K9	klarer Aufgabenbereich	ne	ne	ne	j	j	ne	ne
K10	geringe Komplexität	ne	ne	n	?	j	j	n
K11	Sprachunabhängigkeit	n	n	n	n	n	n	n
V1	Nähe der Abbildung	j	j	-	j	j	j	j
V2	Viskosität	n	n	-	?	?	?	n
V3	versteckte Abhängigkeiten	j	j	-	j	j	j	j
V4	schwere mentale Operationen	n	n	-	?	?	?	n
V5	erzwungenes Vorausdenken	n	n	-	?	?	?	n
V6	sekundäre Notation	n	n	-	n	?	n	n
V7	Sichtbarkeit	n	n	-	?	?	n	n
V8	Komp. visuell und textuell	j	j	n	n	n	n	j
V9	Komp.- und Anwendungsentw.	j	j	j	n	?	j	j
V10	Dokumentation direkt verfügbar	j	n	n	?	?	?	j
V11	Testfunktionalität	j	n	j	?	n	j	n
V12	Programmablaufvisualisierung	n	n	n	?	j	n	n
V13	Einheitliches Aussehen	j	j	j	j	j	j	j
T1	schwarze Kiste Tests	j	-	n	-	-	j	-
T2	normale Schnittstelle	j	-	j	-	-	j	-
T3	benötigte Funktionalität testen	ne	-	ne	-	-	ne	-
T4	Verklemmungen/Zeitverhalten	n	-	n	-	-	?	-
T5	Testspezifikation	n	-	j	-	-	?	-
T6	Regressionstests	n	-	n	-	-	n	-
T7	Filtermechanismen	-	-	-	-	n	-	-
T8	Suchmöglichkeiten	-	-	-	-	?	-	-
T9	Farbe	-	-	-	-	?	-	-
T10	Hierarchie	-	-	-	-	?	-	-
T11	Einstellen der Ansicht	-	-	-	-	j	-	-

8.1 SUN JavaStudio

SUN Java Studio ist eine Entwicklungsumgebung, die sich vor allem an Leute ohne Programmiererfahrung wendet. Der Schwerpunkt liegt dabei auf der Programmierung von Benutzungsoberflächen mit einfacher Funktionalität. Von Giesl [Gie01] und mir [Mar00] wurde SUN Java Studio näher betrachtet. SUN Java Studio bietet eine rein visuelle Programmierung, wobei es allerdings keine Möglichkeit gibt, zur gleichen Zeit noch selbst Quelltext zu schreiben. Die erstellten Programme basieren auf dem Datenflußmodell, wobei Kontrollbefehle auch als Daten interpretiert werden.

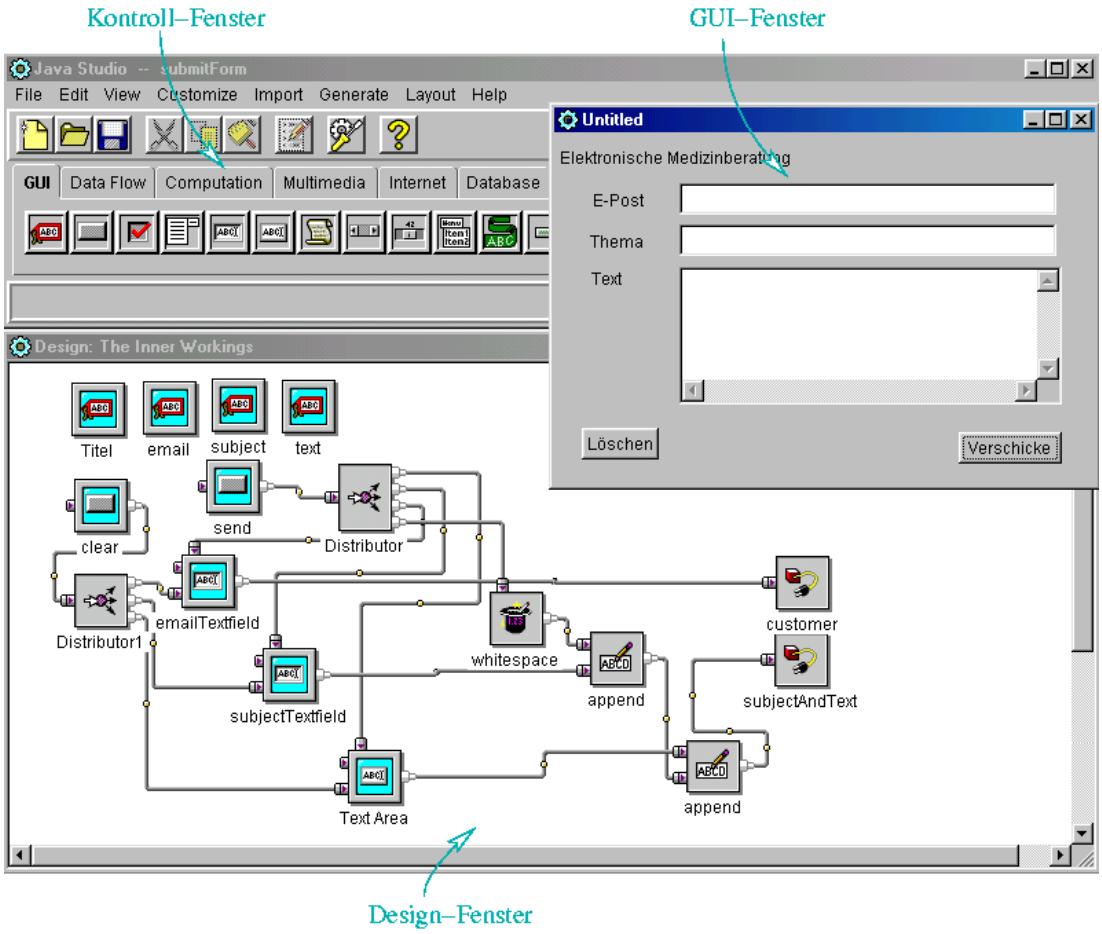

Abbildung 8.1: ePost Eingabemaske *Bean* (aus Giesl [Gie01])

SUN Java Studio ist aus drei Fenstern aufgebaut. Das Kontroll-Fenster dient zur Verwaltung der Programme und stellt alle vorhandenen *Beans* (Komponenten) zur Auswahl bereit. Das GUI-Fenster dient dazu, die Benutzungsoberfläche zu gestalten. In ihm werden nur die sichtbaren *Beans* angezeigt. Im Design-Fenster werden alle Komponenten, die sichtbaren und die nicht-sichtbaren, des erstellten Programms abgebildet (s. Abb. 8.1). Die *Beans* werden dabei durch Piktogramme symbolisiert, an denen jeweils die möglichen Ein- und Ausgänge durch kleine Kästchen dargestellt werden. Jeder Ausgang läßt sich mit genau einem Eingang verbinden. Soll ein Ausgang auf mehrere Eingänge oder mehrere Ausgänge auf genau einen Eingang gelegt werden, so muß speziell eine Verteiler- bzw. Zusammenführ-*Bean* benutzt werden. Die Verbindungen zwischen den *Beans* werden alle, unabhängig von ihren Aufgaben, einheitlich dargestellt. Sind viele Verbindungen für ein Programm notwendig, kann es dadurch leicht unübersichtlich werden. Sichtbare *Beans* werden sowohl im Design als auch im GUI-Fenster abgebildet. Bei vielen sichtbaren *Beans* kann es sein, daß der Programmierer Probleme bei

der Zuordnung zwischen den einzelnen Ansichten der *Beans* hat.

Der *Bean*vorrat von SUN Java Studio kann auf zwei Arten erweitert werden (*V8*, *V9*). Die erste Möglichkeit ist, aus vorhandenen *Beans* neue *Beans* zusammenzubauen. Das wird als *packaged design* bezeichnet. Um diese *packaged designs* mit anderen *Beans* kommunizieren lassen zu können, müssen Ein- und Ausgänge definiert werden. Die zweite Möglichkeit ist, selbst mit Java programmierte *Beans* in SUN Java Studio einzubinden. Die Methoden der *Beans* stellen dabei Eingänge dar, vorausgesetzt, sie benötigen nur einen Parameter, andere werden ignoriert. Ausgänge werden mit Nachrichten, die ein *Bean* schicken kann, realisiert. So in SUN Java Studio eingebundene *Beans* können nicht eingesehen werden, d.h. ein *packaged design* kann auch nicht während der Benutzung geändert werden.

Ein SUN Java Studio Programm ist schon während der Ausführung testbar (*V11*). Dies bringt es aber mit sich, daß ein erstelltes Programm keinen definierten Ausgangszustand hat. Als Hilfe für das Testen werden spezielle *Debug*-Komponenten angeboten, die aber nur beim manuellen Test behilflich sein können. Möglichkeiten, Testfälle zu erstellen, gibt es nicht.

Wird ein Programm umfangreicher, so wird es schnell durch die feingranularen *Beans* unübersichtlich. Die einheitlich dargestellten Verbindungen tragen zusätzlich noch ihren Teil dazu bei. Die einzige Möglichkeit, Abhilfe zu schaffen, ist die Nutzung des *packaged designs*. Der MASA wurde auch mit SUN Java Studio erstellt. Ein Teil ist in Abbildung 8.1 dargestellt. Sie zeigt ein *packaged design* der ePost Eingabemaske. Zu sehen ist, daß schon dieser kleine Teil, sehr viel Platz in Anspruch nimmt.

8.2 IBM Visual Age

IBM Visual Age für Java ist eine Entwicklungsumgebung, die unter anderem auch die visuelle Programmierung ermöglicht (*V8*). Möchte man mit IBM Visual Age programmieren, sind aber Java Kenntnisse zwingend erforderlich. Von Giesl [Gie01] und mir [Mar00] wurde IBM Visual Age näher betrachtet.

Das Hauptfenster zeigt alle geladenen Pakete, Klassen und Schnittstellen. Zur textuellen Bearbeitung kann jeweils ein Fenster geöffnet werden, in dem eine Klasse im ganzen oder einzelne Methoden bearbeitet werden können. Es steht auch ein Assistent bereit, der hilft, den Quelltext automatisch zu vervollständigen.

Die visuelle Programmierung wird durch ein Kompositionseditor ermöglicht, der es erlaubt, Java *Beans* miteinander zu verknüpfen. Er basiert dabei auf der *Parts* Technik von ObjectShare. Wird ein Programm visuell erstellt, erzeugt der Editor automatisch einen Quelltext, der die Funktionalität der visuell programmierten Teile beinhaltet. Möchte der Programmierer noch selbst Funktionalität hin-

zufügen, darf er dies nur in speziell gekennzeichneten Bereichen tun, damit seine Änderungen nicht bei der nächsten Quelltexterzeugung verloren gehen.

IBM Visual Age realisiert die Kommunikation auf der Basis von Java *Beans*. Zur Kommunikation stehen sechs verschiedene Arten von Verbindungen zwischen Eigenschaften, Methoden und Ereignissen von *Beans* bereit. Jede Art der Verbindung wird durch eine andersfarbige Linie mit speziellen Endpunkten dargestellt (*V1*). Außer den Endpunkten läßt sich die Position der Linien beliebig anordnen.

IBM Visual Age bietet eine große Auswahl an Komponenten. Es gewährleistet aber keine eindeutige Abgrenzung zwischen Komponenten und Klassen, wodurch der Programmierer leicht irritiert werden kann, wenn er von eingebundenen Klassen die gleiche Funktionalität wie bei Komponenten erwartet.

Die Benutzungsoberfläche läßt sich rein visuell gut programmieren. Dabei werden sichtbare Komponenten in einem Fenster plaziert und nicht-sichtbare um das Fenster herum. Sie müssen jeweils einen eindeutigen Exemplarnamen haben. Die sichtbaren Komponenten werden genauso abgebildet, wie sie auf der Benutzungsoberfläche zu sehen sind. Die nicht-sichtbaren werden durch ein Puzzlestück dargestellt, wobei für jede Komponente dasselbe Piktogramm benutzt wird.

Während der Programmierung werden spezielle Kenntnisse über die Komponenten erwartet. IBM Visual Age bietet während der Auswahl einer Komponente keine Möglichkeit, eine Dokumentation (*V10*) über eine Komponente zu erhalten. Wird eine Komponente mit einer anderen verbunden, werden nur die Verbindungsmöglichkeiten angezeigt, die in dem speziellen Kontext möglich sind. Somit ist es zu keiner Zeit möglich, einen Überblick über die ganze Komponente zu erhalten.

Die Programmierung von nicht visuellen Komponenten ist sehr eingeschränkt. Es gibt z.B. keine Möglichkeit, Methoden oder Nachrichten für ein neu erstelltes *Bean* zu definieren, die als Ein- oder Ausgang dienen können. Dies kann nur durch eine Hilfskonstruktion realisiert werden, indem spezielle *Beans* generiert werden, die einen Ein- bzw. Ausgang realisieren.

Ist ein Programm größer, wird die visuelle Darstellung schnell recht unübersichtlich. Die Farbe von Verbindungen wird durch die Art der Verbindung bestimmt. Es läßt sich auch oft nicht vermeiden, daß sich viele Verbindungen überschneiden, denn nicht-sichtbare Komponenten können nur außerhalb der sichtbaren plaziert werden. Informationen über Verbindungen lassen sich auch nicht direkt erfragen. Sie lassen sich nur durch ein Fenster erhalten, in dem eine Verbindung definiert werden kann.

Zum Testen von Programmen bietet IBM Visual Age nur einen *Debugger* an. Es besteht keine Möglichkeit, einen Testfall (*V11*) zu spezifizieren.

In Abbildung 8.2 wird gezeigt, wie der MASA mit IBM Visual Age umgesetzt wurde. Die Hauptanwendung mit den Benutzungsoberflächenelementen wurde

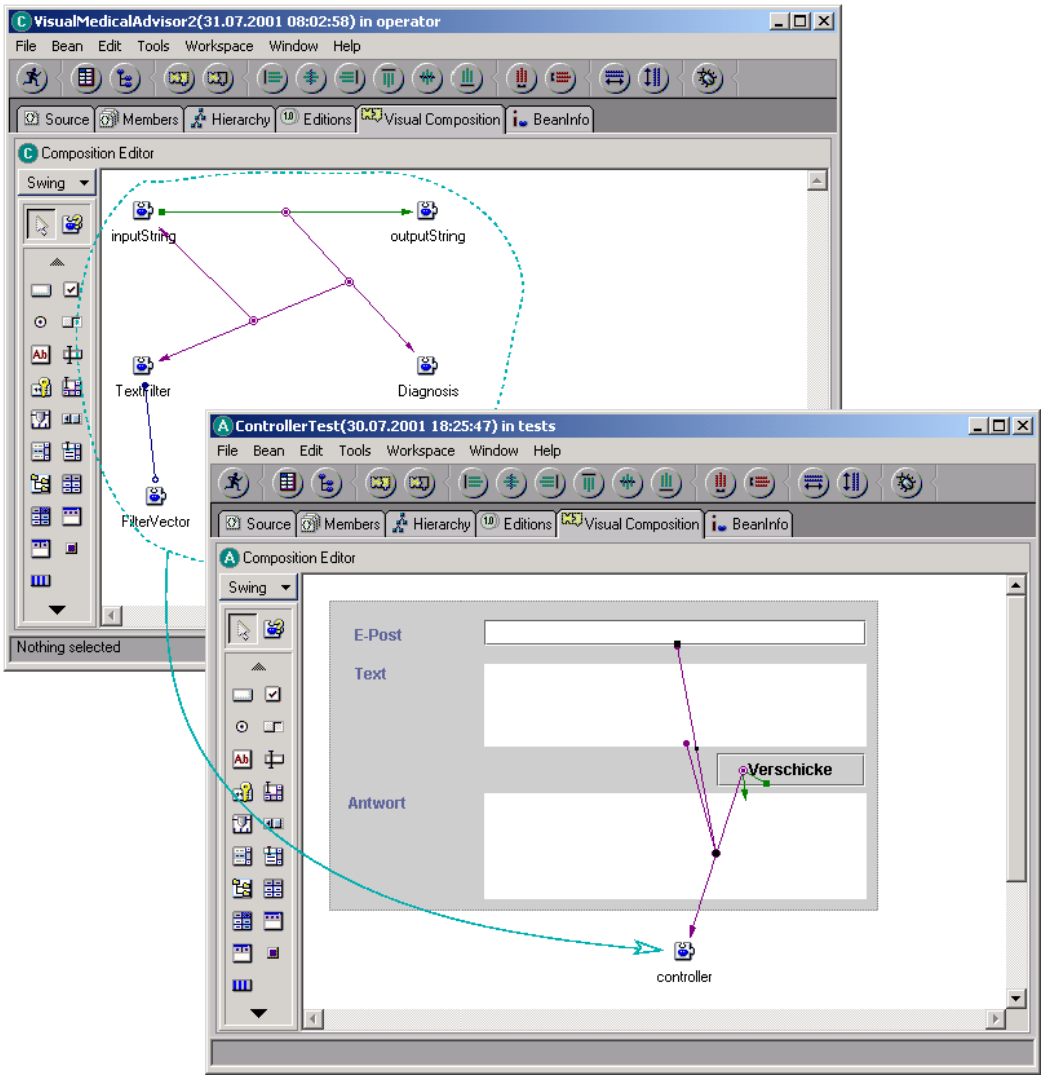

Abbildung 8.2: Agent erstellt mit IBM Visual Age (aus Giesl [Gie01])

rein visuell programmiert. Es wurde auch versucht, die nicht-sichtbare controller Komponente visuell zu programmieren. Dabei kamen Komponenten zum Einsatz, die textuell programmiert wurden.

8.3 Borland JBuilder

Die Entwicklungsumgebung Borland JBuilder dient zur Entwicklung von jeglichen Arten von Java-Programmen. Die Zielgruppe ist der professionelle Java-Entwickler. Bordan JBuilder wurde von Giesl [Gie01] näher betrachtet.

Wird JBuilder gestartet, erscheint ein Hauptfenster (s. Abb. 8.3), das sich in

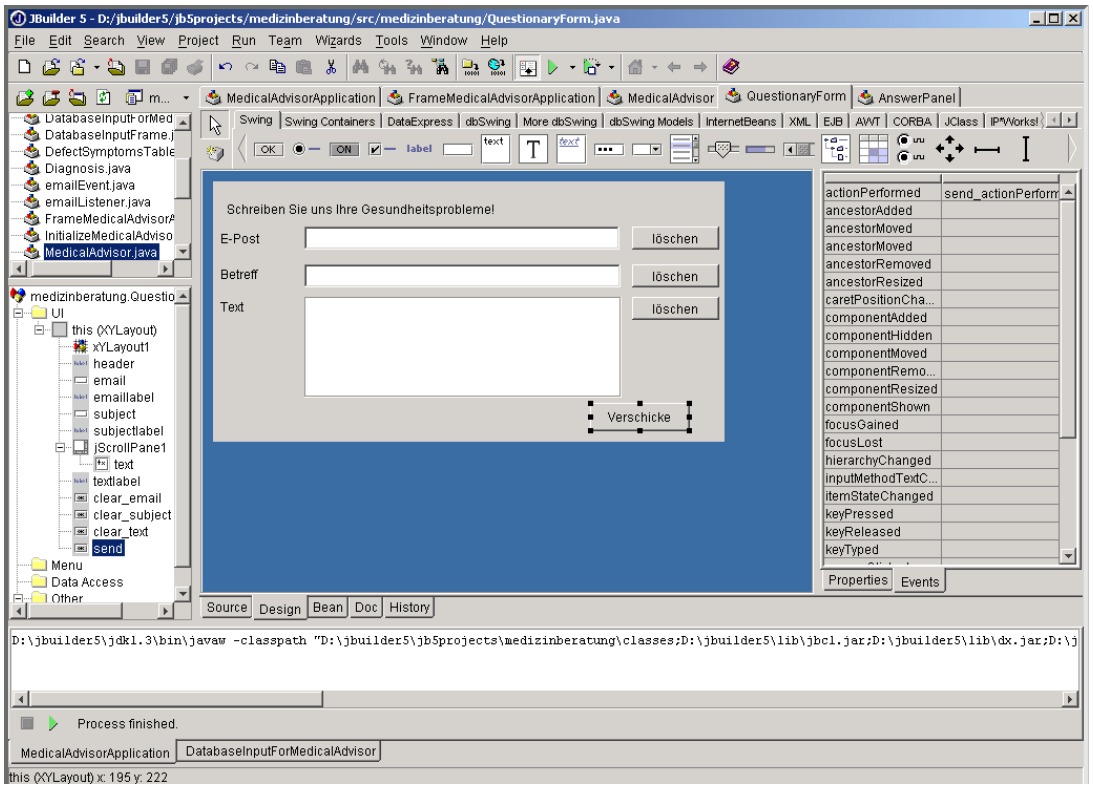

Abbildung 8.3: Agenten-Benutzungsoberfläche mit Borland JBuilder erstellt (aus Giesl [Gie01])

mehrere kleine Fenster aufteilen läßt. In JBuilder lassen sich mehrere Projekte gleichzeitig laden, aber es kann immer nur eines bearbeitet werden. Die geladenen Projekte werden im Projektfenster mit den dazugehörigen Dateien, Klassen und Schnittstellen dargestellt. Das Inhaltsfenster zeigt die geöffneten Dateien an. Jede Datei wird in einem Register abgelegt, so daß ein schnelles Wechseln zwischen den einzelnen Dateien möglich ist. Das Strukturfenster zeigt die Methoden an, die in der gerade bearbeiteten Datei vorhanden sind. Wird eine Methode ausgewählt, springt der Eingabezeiger automatisch an den Anfang der entsprechenden Methode, wodurch ein schnelles Navigieren innerhalb einer Datei möglich ist. Zusätzlich werden dort auch eventuelle Fehlermeldungen angezeigt.

Zum Bearbeiten einer Datei stehen im Inhaltsfenster fünf verschiedene Ansichten (*K6*) bereit. Die einfachste Ansicht ist die Quelltextansicht. Darauf folgt eine Ansicht, um die Benutzungsoberfläche visuell zu erzeugen; weiter stehen Ansichten zur Versionsverwaltung, Dokumentation und zur Konfiguration von *Java Beans* bzw. *Enterprise Java Beans* bereit.

In der Bentzungsoberflächenansicht ist es nur möglich, Benutzungsoberflächen visuell zusammenzusetzen. Nicht möglich ist eine visuelle Programmierung der

unterliegenden Funktionalität (*V8*). Unterstüzt wird nur, speziellen Ereignissen von den Benutzungsoberflächenelementen Methoden zuzuweisen, die im Falle des Eintretens aufgerufen werden. Die Oberfläche wird direkt in Quelltext übersetzt. Werden an dem Quelltext Änderungen durchgeführt, sind diese sofort in der Design-Ansicht der Oberfläche zu sehen.

Zur Unterstüztung der textuellen Entwicklung werden diverse Werkzeuge angeboten, z.B. ist es möglich, sich das Quelltextgerüst für ein *Enterprise Java Bean* erzeugen zu lassen, in dem nachher nur noch die Methoden für die Funktionalität eingetragen werden müssen. Beim Editieren wird der Entwickler auch zusätzlich durch ein Quelltext-Ergänzungswerkzeug unterstützt, das ihm viel Schreibarbeit ersparen kann. Darüber hinaus gibt es auch ein Werkzeug, das einen Testfall (*V11*) für ein *Bean* erzeugen kann. Dieses untersucht ein *Bean* und erstellt automatisch Mustertestfälle für jede vorhandene Beanmethode (*T2*).

Das Beispiel in Abbildung 8.3 zeigt, wie die Benutzungsoberfläche für den MASA mit JBuilder erstellt wurde.

Der hier betrachtete JBuilder hat die Version 5. Die im Moment aktuelle Version 7 stand leider nicht zur Verfügung. Sie böte zusätzlich die Möglichkeit, *Regressionstest* (*T6*) zu erstellen. Ebenfalls lasse sich eine statische Visualisierung des Quelltextes mittels UML zu erzeugen. Ein ergänzendes Produkt Borland Optimizeit Suite unterstützt den Entwickler bei der Optimierung eines Programms.

8.4 AgentSheets

AgentSheets [Gie01, Rep91, RC93, Rep93, RS95, Rep95, Rau98, Rep00, RIP+01] ist ein Programm, um branchenspezifische visuelle Programmiersprachen und Simulationswerkzeuge zu erstellen. Es ist ein kommerzielles Werkzeug (stammt ursprünglich aus dem akademische Bereich), mit dem auch Nicht-Informatiker visuelle Programmiersprachen erstellen können. Das gleiche System bietet auch die Möglichkeit, die mit der neuen Sprache erstellten Programme, meistens Simulationen, auszuführen.

Das Gerüst bietet ein zweidimensionales Gitternetz (*V1*), in denen Agenten plaziert werden können. Agenten, die nebeneinander in einem Gitter sitzen, können miteinander kommunizieren (*V3*). Die Kommunikation besteht aus dem Schicken von Botschaften, wodurch ein Agent veranlaßt werden kann, seinen Zustand zu ändern oder auch eine andere Position im Gitter einzunehmen. Die Agenten werden auch durch Bilder dargestellt. Je nach Zustand eines Agenten kann sich deren Aussehen ändern.

Das spezielle Verhalten der Agenten kann mittels einer visuellen Programmiersprache AgentTalk spezifiziert (*V8*) werden. Einfache Regeln können darin mit

Hilfe eines Menüs zusammengestellt werden. Die Regeln bestehen aus konkreten Bildern und sind dadurch gut verständlich.

Ein Hauptanwendungsgebiet von AgentSheets sind Simulationen. Diese können sowohl passiv, ohne fremdes Zutun, als auch interaktiv, mit Eingreifen eines Benutzers ablaufen. Ein mögliches Szenario eines Ecosystems wird in Abbildung 8.4 dargestellt.

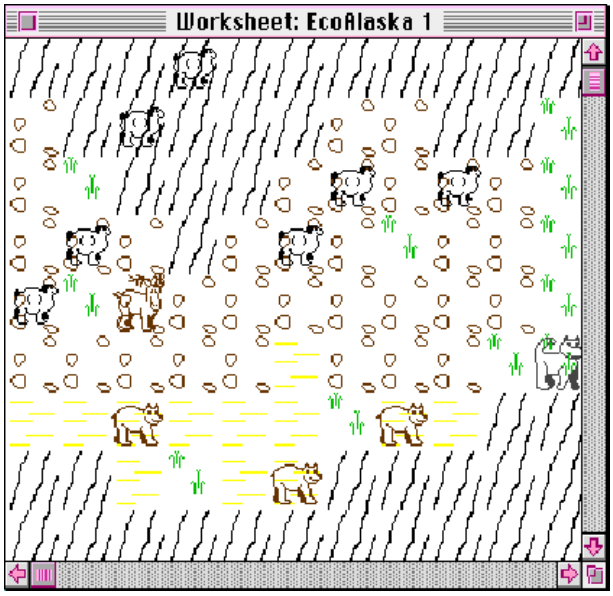

Abbildung 8.4: Ecosystem Simulation (aus Reppening und Citrin [RC93])

Die Agenten können auch im übertragenen Sinne als Komponenten angesehen werden. Es ist aber nicht möglich, aus mehreren Agenten neue Agenten zusammenzubauen (*V9*), wie es in der Komponententechnik möglich ist. Ein neuer Agent muß immer von Grund auf neu programmiert werden.

8.5 SAM

Ein Programmiersystem für dreidimensionale animierte Programme ist SAM (Solid Agents in Motion) [GMR98, GLM98, Gie01]. Dort bestehen die Programme aus Agenten, die miteinander korrespondieren. Die Kommunikation untereinander geschieht durch Botschaften, die von einem Agent zum anderen geschickt werden. Das Verhalten der Agenten wird durch Produktionsregeln bestimmt. Beim Ablauf eines Programms prüfen die Agenten zuerst alle eigenen Regeln, ehe sie im nächsten Schritt die Botschaften verschicken.

Für ein Programm gibt es zwei verschiedene Darstellungen (*K6*). Die abstrakte Form dient zum Programmieren und zum Fehlersuchen. Die Agenten werden

als halb durchsichtige Kugeln und deren Ein- und Ausgänge als Trichter an der Oberfläche dargestellt. Die Regeln sind dreidimensionale Objekte. Die konkrete Repräsentation zeige das fertige Programm. Mit ihr wird das animierte dreidimensionale Szenario gezeigt. Die Agenten werden durch beliebige Bilder repräsentiert, und die Botschaften bewegen sich animiert durch den Raum. Abbildung 8.5 zeigt eine abstrakte und eine konkrete Darstellung eines *Wildlife* Szenarios.

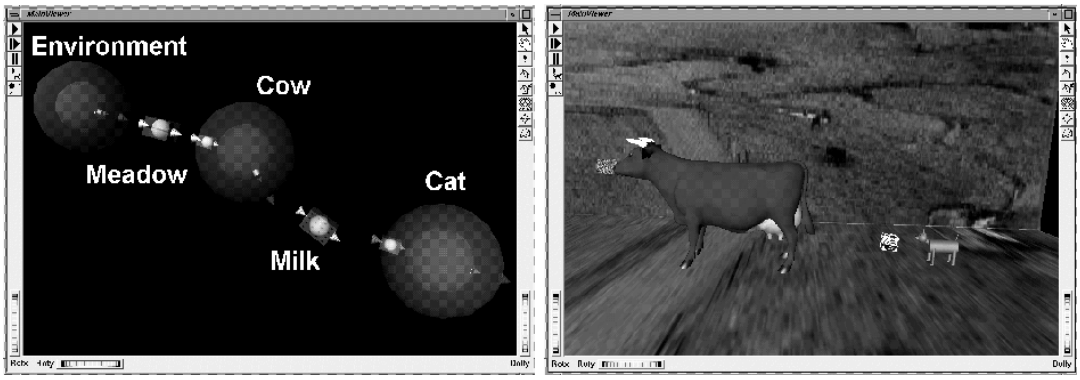

Abbildung 8.5: Wildlife aus Geiger *et al.* [GLM98]

Dadurch, daß die Botschaften sich langsam durch den Raum (*V12*) bewegen und eine automatische Größenanpassung vorgenommen wird, soll es dem Programmierer bzw. dem Tester erleichtert werden, den Programmablauf nachzuvollziehen. Dies wird ihm aber dadurch erschwert, daß der Ablauf nicht festgehalten werden kann, sondern flüchtig ist, wodurch er das komplette frühere Geschehen im Gedächtnis halten muß.

8.6 Prograph

Die Entwicklungsumgebung Prograph [Sch96, CS94, KS01] ist rein visuell (*V8*) und kann auch von Nicht-Programmierern benutzt werden.

Die Programmierung basiert auf einer graphischen Datenflußsprache. Prograph-Programme werden aus Blöcken zusammengebaut, die ähnlich wie Komponenten sind. Sie seien aber nicht so leistungsstark wie Komponenten, wodurch ein Programm größer (*V7*) werden kann, wenn die Aufgaben nicht geeignet in neue Blöcke gekapselt werden. An den Blöcken werden die Ein- und Ausgänge mit Kreisen gekennzeichnet. Verbunden werden die Kreise mit Linien, wobei der Datenfluß von oben nach unten angenommen wird. Es besteht keine Möglichkeit, eine *sekundäre Notation (V6)* zu verwenden. Die Ausführung eines Blocks ist datengesteuert. An jedem Eingang müssen Daten anliegen, damit ein Block mit der Ausführung beginnt. Wie zuvor erwähnt, ist es auch möglich, neue Blöcke (*V9*)

zu definieren. Die Ein- und Ausgänge werden dabei durch Balken repräsentiert, die sich am oberen bzw. unterem Arbeitsfenster-Rand befinden.

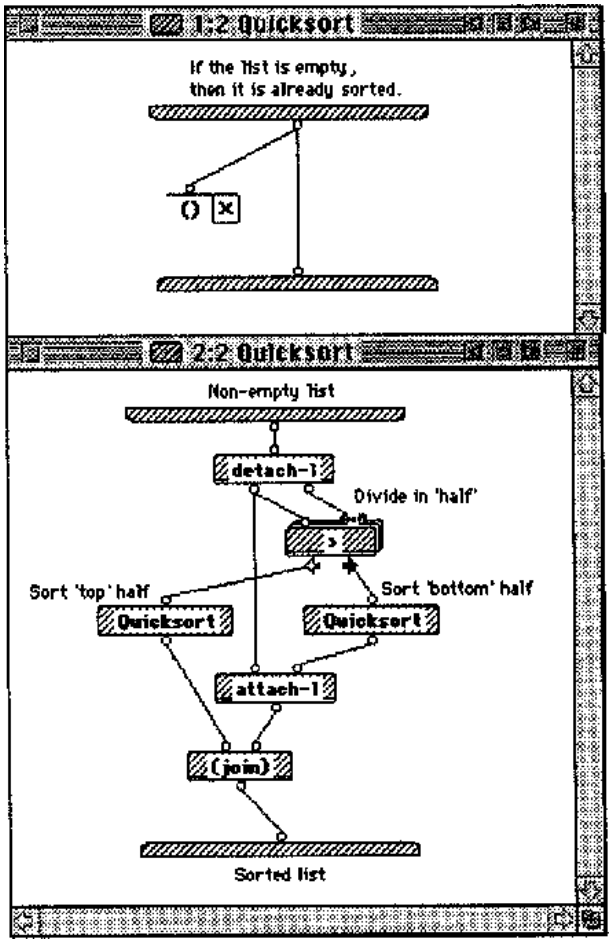

Abbildung 8.6: *quicksort* Sortieralgorithmus aus Schmucker [Sch96]

Abbildung 8.6 zeigt einen *quicksort* Sortieralgorithmus, der mit Hilfe von Prograph implementiert wurde.

Zur Unterstützung des Softwaretests von Prograph-Programmen bietet die Entwicklungsumgebung an, Tests (*V11*) zu definieren, und erlaubt, visuell Datenflußtests zu erstellen.

8.7 Aonix SELECT Component Factory

SELECT Component Factory [Aon] ist eine Entwicklungsumgebung zur Erstellung von Anwendungen aus Komponenten. Sie bietet verschiedene visuelle Werkzeuge zur Erzeugung und Verwaltung von Komponenten und zur Konstrukti-

on von Anwendungen (*K4, V9*). *SELECT* stellt eine Anwendung mit Hilfe von UML Diagrammen dar. Zur Erstellung einer Anwendung müssen diverse UML-Diagramme wie *Process Hierachy, Use Case, Class* und *Object Sequence* gebildet werden (s. Abb. 8.7).

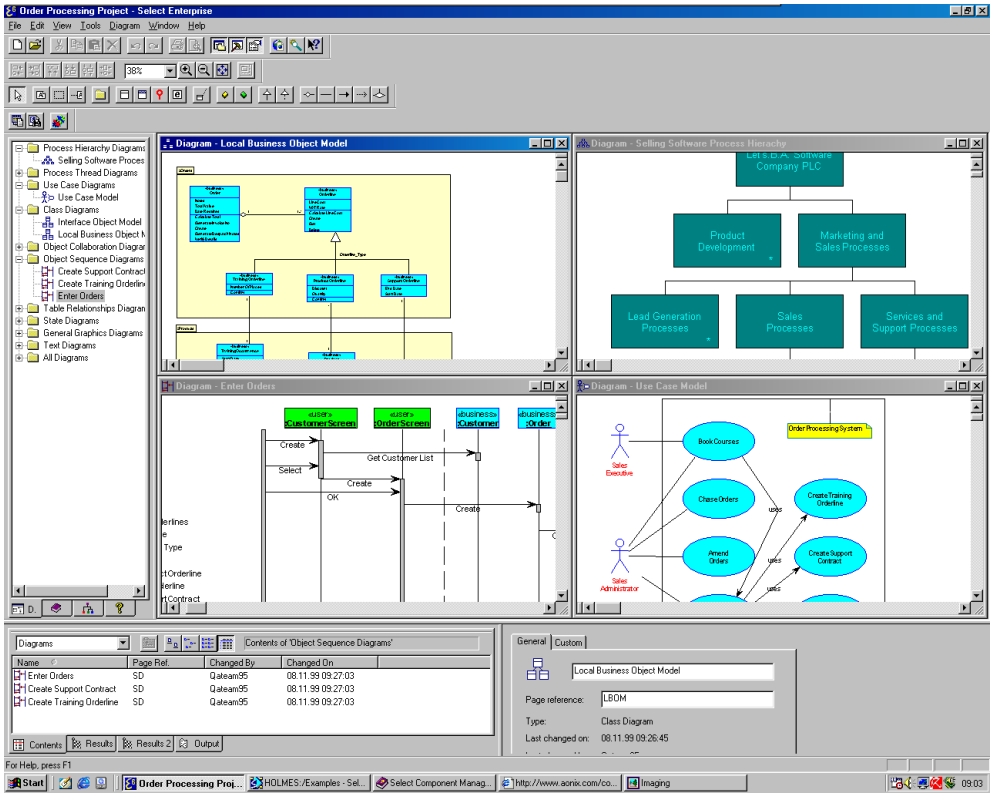

Abbildung 8.7: Aonix SELECT Component Factory

Durch die Vielzahl von Diagrammen, die für eine Anwendung nötig sind, kann leicht der Überblick (*V7*) verloren gehen. UML Diagramme können auch schnell umfangreich und unübersichtlich werden. Als Abhilfe wird angeboten, z.B. mehrere *Sequence Diagramme* zu erstellen, was aber nicht wirklich hilft, da dadurch die Anzahl der einzelnen Diagramme weiter erhöht wird.

SELECT stellt die Komponenten mit Hilfe ihrer Schnittstellenklassen dar. Dadurch wird der Unterschied zwischen Komponenten und normalen Objekten allerdings nicht deutlich.

Die Komponenten werden mit dem Werkzeug *Component Manager* verwaltet. Dort sind die Komponenten hierarchisch gegliedert. Wählt man eine Schnittstelle einer Komponente aus, wird das Protokoll derselben angezeigt. Wird dort eine Methode ausgewählt, erscheint eine kurze Beschreibung (*K2, V10*) und ein Kommentar zur Erklärung. Leider ist oft weder eine Beschreibung noch ein Kommentar vorhanden.

Kapitel 9

Zusammenfassung und Ausblick

In den vorhergehenden Kapiteln wurden Merkmale vorgestellt, die für Komponentensysteme und deren Entwicklungsumgebungen wichtig sind. Anhand dieser wurde die Entwicklung von HotAgent gezeigt. Die wichtigsten Eigenschaften von HotAgent werden im Abschnitt 9.1 noch einmal aufgegriffen. In dieser Arbeit wurden auch die Vor- und Nachteile von HotAgent in einer Gebrauchstauglichkeits-Studie untersucht. Mögliche Erweiterungen werden in Abschnitt 9.2 angesprochen.

9.1 Zusammenfassung

HotAgent ist ein Komponentenentwurfsrahmen, um Agenten für den elektronischen Handel zu konstruieren. Es werden spezielle Komponenten bereitgestellt, um Agenten zu konstruieren, die z.B. Routinearbeiten übernehmen. HotAgent bietet verschiedene Werkzeuge an, um Komponenten zu entwickeln und zu testen und um Agenten aus Komponenten zusammenzubauen und deren Laufzeitverhalten zu analysieren. Durch diese Werkzeuge deckt HotAgent jeden Schritt in der komponentenorientierten Entwicklung ab. Die Werkzeuge bieten alle eine graphische Unterstützung neben der visuellen und textuellen Programmierung. Während der Entwicklung wurde auf eine gute Integration der einzelnen Werkzeuge geachtet und daß alle eine ähnliche Bedienung aufweisen. Der Entwickler wird dabei unterstützt, indem er ein durchgängiges Entwicklungsprinzip wiederfindet. Es ist nur möglich, Agenten aus Komponenten (nicht etwa auch aus Objekten) zusammenzubauen.

HotAgent Component dient dazu, um neue Komponenten aus vorhandenen zu erstellen. Dazu können diese auf eine Arbeitsfläche plaziert werden, um sie mit anderen Komponenten und neuen Ein- und Ausgängen zu verbinden. Mit diesem

Werkzeug kann ein Entwickler neue Komponenten erzeugen, ohne viel über das benutzte Komponentenmodell wissen zu müssen.

Mit diesen erstellten und anderen vordefinierten Komponenten können unter Verwendung von HotAgent Assembly Agenten erstellt werden. Dazu wird eine ähnliche Methode wie die in HotAgent Component verwendet. HotAgent Assembly erlaubt dadurch das leichte und übersichtliche Erstellen von Agenten. Zusätzlich werden Funktionen angeboten, um Agenten komfortabel zu warten.

Um qualitativ hochwertige Agenten zu erstellen, ist es wichtig, daß die verwendeten Komponenten getestet sind. HotAgent Test erlaubt das Testen von Komponenten durch ihre Ein- und Ausgänge. Dabei können mehrere Testfälle für eine Komponente angegeben werden, die durch HotAgent Regression als Regressionstests aufgerufen werden. Dadurch kann sichergestellt werden, daß alle Testfälle noch einwandfrei funktionieren. Sollte ein Testfall Mängel zeigen, kann er von dort schnell behoben werden.

HotAgent Visualize erstellt eine dreidimensionale Visualisierung des Laufzeitverhalten eines Agenten. Dazu wird das Verhalten durch eine dynamische Analyse untersucht. Durch Filter können die ermittelten Daten weiter aufbereitet werden, so daß eine übersichtliche Visualisierung entsteht. Es kann sowohl der gesamte Agent als auch Teile von diesem betrachtet werden.

In den Kapiteln 2.2, 3.2 und 4.3 wurde Kriterien erarbeitet, die beim Entwickeln von Komponenten, beim Zusammenbauen von Anwendungen und beim Testen bzw. beim Verfolgen des Programmablaufs beachtet werden müssen. HotAgent berücksichtigt fast alle dieser Kriterien. In Kapitel 8 wurden andere Entwicklungsumgebungen mit Hilfe dieser Kriterien vorgestellt. Ein Vergleich zwischen HotAgent und anderen Entwicklungsumgebungen ist in der untenstehenden Tabelle verdeutlicht. Dabei werden folgende Abkürzungen verwendet: SUN Java Studio – JS, IBM Visual Age – VA, Borland JBuilder – JB, AgentSheets – AS, SAM (Solid Agents in Motion) – SAM, Prograph – PG, Aonix SELECT Component Factory – CF, HotAgent – HA, ja – j, nein – n, nicht erzwungen – ne, separate Datei – sD, separate Klasse – sK, externe Applikation – eA, nicht anwendbar – -, nicht bekannt – ?.

Kriterium		JS	VA	JB	AS	SAM	PG	CF	HA
K1	modulares Design	ne	ne	ne	j	?	j	ne	j
K2	Selbstbeschreibung	sK	sK	sD	?	?	?	j	j
K3	globaler Namensraum	n	n	n	?	?	?	n	ne
K4	zweigeteilter Entwicklungsprozeß	j	j	j	j	j	j	j	j
K5	Anwendungen zusammenbauen	j	j	j	j	j	j	j	j
K6	verschiedene Ansichten	j	j	j	j	j	j	j	j
K7	Wiederverwendung durch Verweis	j	j	j	?	?	?	j	j
K8	Testmethoden	n	n	n	?	n	?	n	j

Kriterium		JS	VA	JB	AS	SAM	PG	CF	HA
K9	klarer Aufgabenbereich	ne	ne	ne	j	j	ne	ne	ne
K10	geringe Komplexität	ne	ne	n	?	j	j	n	j
K11	Sprachunabhängigkeit	n	n	n	n	n	n	n	n
V1	Nähe der Abbildung	j	j	-	j	j	j	j	j
V2	Viskosität	n	n	-	?	?	?	n	j
V3	versteckte Abhängigkeiten	j	j	-	j	j	j	j	j
V4	schwere mentale Operationen	n	n	-	?	?	?	n	j
V5	erzwungenes Vorausdenken	n	n	-	?	?	?	n	j
V6	sekundäre Notation	n	n	-	n	?	n	n	j
V7	Sichtbarkeit	n	n	-	?	?	n	n	j
V8	Komp. visuell und textuell	j	j	n	n	n	n	j	j
V9	Komp.- und Anwendungsentw.	j	j	j	n	?	j	j	j
V10	Dokumentation direkt verfügbar	j	n	n	?	?	?	j	j
V11	Testfunktionalität	j	n	j	?	n	j	n	j
V12	Programmablaufvisualisierung	n	n	n	?	j	n	n	j
V13	Einheitliches Aussehen	j	j	j	j	j	j	j	j
T1	schwarze Kiste Tests	j	-	n	-	-	j	-	j
T2	normale Schnittstelle	j	-	j	-	-	j	-	j
T3	benötigte Funktionalität testen	ne	-	ne	-	-	ne	-	ne
T4	Verklemmungen/Zeitverhalten	n	-	n	-	-	?	-	j
T5	Testspezifikation	n	-	j	-	-	?	-	j
T6	Regressionstests	n	-	n	-	-	n	-	j
T7	Filtermechanismen	-	-	-	-	n	-	-	j
T8	Suchmöglichkeiten	-	-	-	-	?	-	-	eA
T9	Farbe	-	-	-	-	?	-	-	j
T10	Hierarchie	-	-	-	-	?	-	-	j
T11	Einstellen der Ansicht	-	-	-	-	j	-	-	j

9.2 Ausblick

Im vorigen Abschnitt wurden die einzelnen Werkzeuge von HotAgent vorgestellt. Als Ergänzung zu diesen ist ein Werkzeug (s. Abb. 9.1) denkbar, mit dem neue Komponenten für die Benutzungsoberfläche erstellt werden können. HotAgent Visual Component Drawer wird z.Z. im Rahmen eines Software-Praktikums erstellt. Der erste Schritt ist, ein vordefiniertes Entwurfsmuster, wie z.B. Druckknöpfe oder Schiebebalken, auszuwählen. Darauf kann mit dem Zeichnen eines neuen Elementes für die Benutzungsoberfläche angefangen werden. Es werden Elemente wie Linien, Rechtecke und Ellipsen in verschiedenen Farben angeboten. Je nach Entwurfsmuster ist auch das Definieren mehrerer Zustände, wie z.B. für „gedrückt“, oder der Bewegungsrichtung des Schiebebalkens möglich. Ist auch diese Aufgabe abgeschlossen, erzeugt das Programm selbständig den notwendigen Quelltext für die Komponente. Ein ähnlicher Ansatz wurde schon von Siemon [Sie01] vorgestellt, der sich aber nicht mit Komponenten beschäftigte.

Meyer [Mey87] hat den Begriff *Design by Contract* eingeführt. Dabei geht er

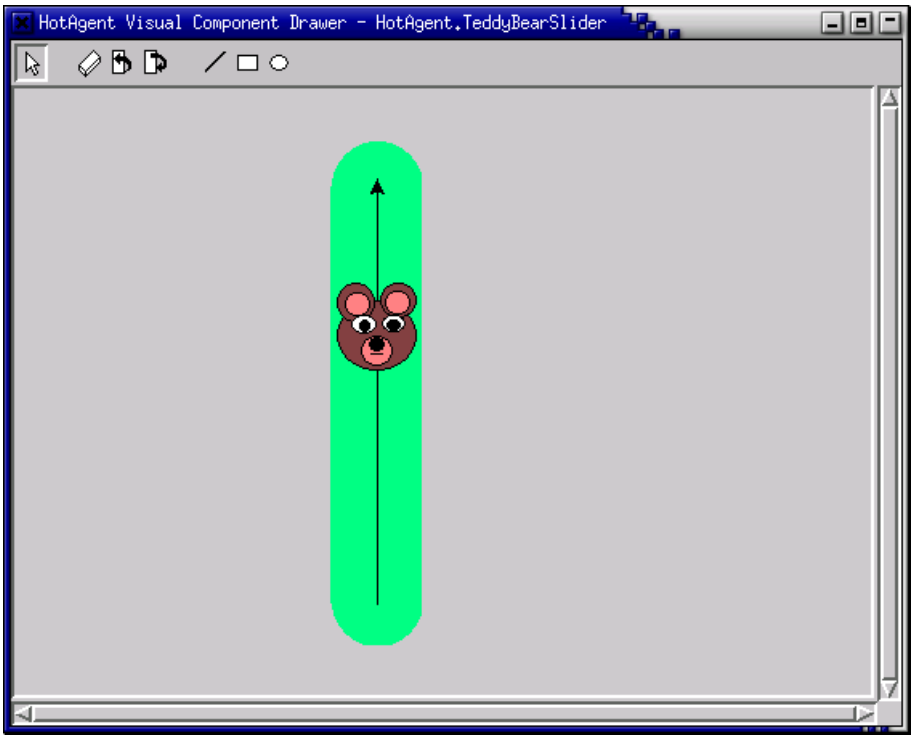

Abbildung 9.1: HotAgent Visual Component Drawer

davon aus, daß zu jeder Methode Vorbedingungen, Nachbedingungen und Invarianten definiert werden. Diese dienten dazu Mängel zu vermeiden bzw. mögliche Mängel zu finden. Meyer [Mey99a] schreibt, daß diese Techniken auch für Komponenten angewandt werden können. Ich beabsichtige, diese Technik in das HotAgent Komponentenmodell einzubauen. Dadurch läßt sich eine automatische Prüfung realisieren, ob Aus- und Eingänge von Komponenten aneinander passen, indem ihre Nach- und Vorbedingungen gegeneinander geprüft werden. Da das Komponentenmodell von HotAgent in Smalltalk implementiert ist, und diese untypisiert ist, liegt die Herausforderung darin, ein Regelwerk zu erstellen, um Vor- und Nachbedingungen zu spezifizieren. Ist dies geschehen, kann während der Konstruktion von Agenten oder Komponenten mit HotAgent Assembly oder Component sofort geprüft werden, ob eine Verbindung sinnvoll ist. Das kommt auch der Forderung von Mezini und Lieberherr [ML98, Lie96] nach, daß nur „direkte Freunde“ von Komponenten miteinander kommunizieren können dürfen.

Eine andere Erweiterungsmöglichkeit ist, die Kommunikation zwischen den HotAgent Komponenten an andere Systeme anzupassen und dadurch eine bessere Kompatibilität und Kommunikationsmöglichkeit zu erlangen. Die kann z.B. durch die Integration des ebXML Standards (Electronic Business using eXtensible Markup Language) [ebX] geschehen. ebXML ist eine modulare Sammlung von Spezifikationen, die es Unternehmen erlaubt, über das Internet zu kommunizieren.

Wie der Name schon sagt, ist es eine XML-Erweiterung, die es gestattet, Nachrichten verschiedener Art und verschiedenen Inhalts auf standardisierte Weise auszutauschen. Die Integration in das HotAgent Komponentensystem ist nicht aufwendig. Ich plane, dazu eine Komponente zu entwickeln, die auf Basis des ebXML Standards eine Kommunikation mit anderen Systemen gewährleistet.

Betrachtet man sich HotAgent Assembly, fällt auf, daß es manchmal schwierig sein kann, die Komponenten und deren Verbindungen klar anzuordnen. Besonders bei größeren Programmen mit vielen Verbindungen zwischen den Komponenten kann es kompliziert sein, diese übersichtlich anzuordnen. Ich sehe die Möglichkeit in einer Erweiterung, indem ein Algorithmus zu entwickeln ist, der die Arbeit erleichtert. Dazu gibt es bereits schon einige Ansätze aus der Forschung, die mit zu berücksichtigen sind. Dieser Algorithmus ordnet wahlweise die Verbindungen oder auch die Komponenten so neu an, daß möglichst wenig Überschneidungen auftreten. Dadurch wird ein klarerer Arbeitsbereich geschaffen. Denkbar ist ebenso, daß der Algorithmus die sekundäre Notation mit berücksichtigt. Zwei sich schneidende Verbindungen mit unterschiedlichen Farben lassen sich eher identifizieren als solche mit den gleichen Farben.

HotAgent Visualize bietet zur Zeit eine dynamische Visualisierung des Laufzeitverhaltens eines Agenten. Eine mögliche Erweiterung ist die Entwicklung einer animierten dynamischen Visualisierung. Dabei kann das Laufzeitverhalten eines Agenten Schritt für Schritt dargestellt werden, indem die gleiche Visualisierung benutzt wird, nur daß je nach Ausführungspunkt die entsprechende Verbindung hervorgehoben wird und eventuell die Ansicht auf diese eingestellt wird. Dabei wird der Entwickler Schritt für Schritt durch den Agenten geführt. Optional können zur gleichen Zeit die Kommunikationsdaten für die jeweils hervorgehobene Verbindung angezeigt werden. Zusätzlich ist es auch möglich, die Präsentation anzuhalten oder andere Bereiche noch einmal anzuschauen. Diese Art der Visualisierung ist vor allem bei größeren Programmen hilfreich, deren Laufzeitvisualisierung umfangreicher ist, so daß die Visualisierung immer auf die entsprechende Position eingestellt wird.

HotAgent ist z.Z. als Prototyp implementiert. Es ist in der Praxis zu prüfen, inwieweit die Produktivität durch HotAgent in der komponentenorientierten Software-Entwicklung gesteigert wird. Ebenso ist zu prüfen, wie skalierbar HotAgent ist. Lassen sich genügend Komponenten einbinden? Ist die Visualisierung bei großen Projekten noch brauchbar. Wie Kapitel 7 zeigt, sind solche extensiven Studien an einer Universität nicht ohne weiteres durchführbar.

Anhang A

Alle Kriterien

In diesem Kapitel werden für einen besseren Überblick alle Kriterien zusammengestellt, die in den Kapiteln 2.2, 3.2 und 4.3 erarbeitet wurden.

Modulares Design (K1): Werde eine Komponente (nach [LR00, LR01]) aus mehreren Teilen zusammengesetzt, so müsse eindeutig zwischen privaten und öffentlichen Teilen unterschieden werden. Meyer [Mey90] definiert fünf Prinzipien für saubere Modularität. Eines ist das *Geheimnisprinzip*. Alle Teile seien modulintern, wenn sie nicht ausdrücklich als öffentlich definiert seien. So könne u.a. die Implementierung geändert werden, ohne daß es nach außen hin sichtbar werde. Er fordert auch *explizite Schnittstellen*. Jede Außenverbindung müsse deutlich gekennzeichnet werden, damit nachvollzogen werden könne, wie eine Komponente beeinflußt wird.

Komponenten-Selbstbeschreibung (K2): Eine Komponente (nach [LR00, LR01]) solle die Informationen, die für ihre Verwendung notwendig sind, selber zur Verfügung stellen. Weinreich und Sametinger [WS01] ergänzen, daß dazu auch Informationen über die Beziehungen zwischen Komponenten und über die Schnittstelle einer Komponente gehören. Knuth [Knu92] fordert das *literate programming*, bei der die Programmiersprache auch gleich Dokumentationssprache ist. Dies soll u.a. helfen, Abweichungen zwischen Quelltext und Dokumentation zu verhindern.

Globaler Namensraum (K3): Um eine Komponente (nach [LR00, LR01]) für eine Anwendung genau zu spezifizieren, sei es notwendig, daß es einen eindeutigen, globalen Namensraum gebe, damit keine Verwechslungen auftreten.

Zweigeteilter Entwicklungsprozeß (K4): Der Entwicklungsprozeß lasse sich (nach [LR00, LR01]) in zwei Teile aufteilen: Das Entwickeln von Komponenten und das Erstellen der Anwendung aus Komponenten.

Anwendung zusammenbauen (K5): Eine Anwendung werde (nach [LR00, LR01]) durch Anpassen und Verbinden von Komponenten erstellt.

Verschiedene Ansichten (K6): Beim Entwickeln von Komponenten und Anwendungen werden (nach [LR00, LR01]) verschiedene Ansichten benötigt, dazu gehören z.B. Entwicklungs- und Kompositionsansichten. Dieses Kriterium ist nicht auf Komponentenmodelle anwendbar, sondern bezieht sich nur auf deren Entwicklungsumgebungen.

Wiederverwendung durch Verweis (K7): Wird eine Komponente eingebunden, geschehe das (nach [LR00, LR01]) durch einen Verweis auf die Komponentenbeschreibung. Dadurch werde vermieden, daß mehrere unterschiedliche Kopien einer Komponente existieren.

Test- und Verifikationsmethoden (K8): Für eine Komponente seien Test- und Verifikationsmethoden notwendig, um die Richtigkeit zu prüfen. Damit könne sie gut wiederverwandt werden. Es werde gefordert, daß Tests einer Komponente mitgegeben werden.

Klarer Aufgabenbereich (K9): In Abschnitt 2.1 wurden drei Definitionen für eine Komponente angegeben. In allen drei wird gefordert, daß eine Komponente leicht von Dritten benutzt werden könne. Um dies zu gewährleisten, benötigt eine Komponente einen abgegrenzten Aufgabenbereich. Dadurch läßt sich erkennen, wofür eine Komponente geeignet ist, und sie läßt sich somit leichter verwenden.

Geringe Komplexität (K10): Mezini und Haupt [MH01a] fordern, daß eine Komponente unabhängig sei. Um dies zu fördern, ist es meiner Meinung nach notwendig, daß die Komponente eine geringe Komplexität aufweist. Dies ist leichter zu erreichen, wenn eine Komponente einen *klaren Aufgabenbereich (K9)* hat. Meyer [Mey90] sagt in dem Prinzip der *wenigen Schnittstellen*, daß mit möglichst wenig anderen kommuniziert werden solle, um möglichst unabhängig zu sein.

Sprachunabhängigkeit (K11): Komponenten versprechen Effizienz u.a. durch Wiederverwendung. Dies wird erleichtert, wenn möglichst viele Komponenten genutzt werden können. Wenn ein Komponentensystem sprachunabhängig ist, ist die Auswahl an Komponenten deutlich größer. Ein Komponentenmodell muß dafür Mechanismen bieten, daß sich in unterschiedlichen Sprachen implementierte Komponenten verständigen können.

Nähe der Abbildung (V1): Der Abstand (s. [GP96]) zwischen der abzubildenden Problemwelt und der Programmiersprache werde als Nähe der Abbildung bezeichnet. Je besser ein Problem in einer Programmiersprache

abgebildet werden könne, desto leichter und übersichtlicher werden die resultierenden Programme. Green und Petre schlagen deshalb eine auf den Anwendungsbereich spezialisierte Programmiersprache vor.

Viskosität (V2): Die Viskosität (s. [GP96]) bezeichne den Aufwand, der notwendig ist, um kleine Programmänderungen durchzuführen. Bei visuellen Programmiersprachen sei damit auch das Umordnen von Elementen gemeint, damit neue eingefügt werden können oder um ein Programm übersichtlicher zu machen.

Versteckte Abhängigkeiten (V3): Wenn implizite Verbindungen zwischen Programmteilen bestehen (s. [GP96]) und für den fehlerfreien Ablauf eines Programms notwendig sind, werde von versteckten Abhängigkeiten gesprochen. Werden diese Abhängigkeiten nicht klar dargestellt, können bei Änderungen unvorhersehbare Fehler auftreten. Meyer [Mey90] spricht dabei auch von *expliziten Schnittstellen*.

Schwere mentale Operationen (V4): Sachverhalte, die komplex dargestellt werden müssen (s. [GP96]), seien zu vermeiden. Es muß versucht werden, Aufgaben einfach und verständlich darzustellen.

Erzwungenes Vorausdenken (V5): Die Problematik, inwieweit vorausgeplant und vorab Entscheidungen getroffen werden müssen, werde als erzwungenes Vorausdenken bezeichnet (s. [GP96]). Dies werde vor allem durch implizite Abhängigkeiten bewirkt. Bei visueller Programmierung sei ein gewisses Vorausdenken notwendig, um nicht ständig die Programme neu ordnen zu müssen.

Sekundäre Notation (V6): Zusätzliche Information (s. [GP96]), die nicht zu der Syntax eines Programms gehören, zählen zur sekundären Notation. Sie sollen die Lesbarkeit von Programmen erhöhen. Beim visuellen Programmieren könne schon das Gruppieren von Elementen oder das Einhalten von Konventionen dazu zählen.

Sichtbarkeit (V7): Sie zeige (s. [GP96]), wie leicht ein Programm oder ein Teil eines Programms erfaßt werden kann. Da heute Anwendungen häufig großen Umfang haben, sei es besonders wichtig, daß Programme leicht untersucht und Teile gesondert betrachtet werden können. Dabei solle es auch möglich sein, Teile nebeneinander anzuzeigen.

Komponentenentwicklung visuell und textuell (V8): Da visuelle Programmierung manchmal viel Platz beansprucht (s. [Gie01]), solle es möglich sein, Komponenten auch textuell zu erstellen. Auf der anderen Seite sollen auch Komponenten aus anderen Komponenten visuell erstellt werden

können, um eine gute Gliederung zu ermöglichen. Viele Entwicklungsumgebungen bieten diese Möglichkeit leider nur für visuelle Komponenten an. Winkler [Win90] sagt, daß für das Programmieren im Großen sich eher eine graphische Lösung eigne, wobei das Programmieren im Kleinen besser textuell geschehe.

Komponenten- und Anwendungsentwicklung (V9): Eine Entwicklungsumgebung müsse die beiden Teilgebiete Komponenten- und Anwendungsentwicklung unterstützen können (s. [Gie01]). Die Aufgabenbereiche lassen sich nicht vollständig getrennt betrachten, sondern der Entwickler solle für beides Entwicklungswerkzeuge vorfinden. Bei textueller Entwicklung von Komponenten solle der Programmierer auch unterstützt werden, um z.B. durch Schlüsselwörter, Ein- und Ausgänge zu erstellen.

Dokumentationen direkt verfügbar (V10): Das Kriterium der Selbstbeschreibung (*K2*) von Komponenten nutze dem Entwickler wenig (s. [Gie01]), wenn die Entwicklungsumgebung nicht in der Lage sei, sie geeignet zu präsentieren. Dabei seien unterschiedliche Arten von Informationen an verschiedenen Stellen im Entwicklungsprozeß interessant.

Testfunktionalität (V11): Testmöglichkeiten seien wichtig (s. [Gie01]), da der Entwicklungsprozeß kontinuierlich geprüft werden muß. Gerade bei komponentenorientierter Programmierung sei es notwendig, daß eine Entwicklungsumgebung eine geeignete Testumgebung bereitstellt. Selbst erstellte Komponenten müssen ausreichend getestet werden, damit eine sichere Wiederverwendung gewährleistet werden kann (s. Kapitel 4).

Programmablaufvisualisierung (V12): Um den tatsächlichen Ablauf einer Anwendung darzustellen, könne die Programmablaufvisualisierung genutzt werden (s. [Gie01]). Visuelles Programmieren mit Komponenten biete schon eine statische Visualisierung des Programms. Es empfehle sich, diese Darstellung so zu erweitern, daß eine dynamische Visualisierung entsteht (s. Kapitel 4).

Einheitliches Aussehen (V13): Besteht eine Entwicklungsumgebung aus mehreren unterschiedlichen Werkzeugen, ist es wichtig, daß alle ein einheitliches Aussehen und gleiche Bedienung bieten.

Schwarze Kiste Tests (T1): Diese Tests (engl. black box test) sind anwendbar für Komponentensoftware. Die kleinste Einheit zum Testen bei der Komponententechnologie ist die Komponente. Es ist nicht notwendig, die Bestandteile einer Komponente einzeln zu testen, sondern es ist besser, eine Komponente als ganze zu betrachten, wie sie auch später eingesetzt wird.

Demeyer *et al.* [DDN03] sagen, es sei wichtig, die Schnittstelle zu Testen, nicht die Implementation.

Da Komponenten Blöcke mit Ein- und Ausgängen bilden, ist es praktikabel, sie mit einer Kombination aus Kontroll- und Datenflußtests zu verifizieren. Eine Nachricht oder Daten können in einen Eingang gegeben werden, und die an den Ausgängen bereitgestellten Daten können getestet werden.

Normale Schnittstelle (T2): Die normale Komponentenschnittstelle muß ausreichen, um aussagekräftige Tests zu erstellen. Sobald spezielle Schnittstellen zum Setzen von Zuständen, wie bei dem Component+ [Com01] Projekt benutzt werden, sind die Tests nicht mehr repräsentativ.

Benötigte Funktionalität testen (T3): Ein Komponentensoftware-Entwickler solle nur die benötigte Funktionalität testen, beschreiben Harrold *et al.* [HLS99]. Zusätzliche Aufrufe, von sonst unbenutzter Funktionalität, können Variable setzen, die anderenfalls uninitialisiert sind und einen Mangel verursachen. Beck [Bec00] nennt diese Tests Komponententests.

Der Komponentenentwickler muß hingegen die gesamte Komponente testen. Er muß dadurch sicherstellen, daß sie in allen möglichen Einsatzgebieten einwandfrei funktioniert. Der Komponentenentwickler weiß nicht, für welchen genauen Zweck ein Anwendungsentwickler seine Komponenten einsetzen möchte. Durch seine Tests muß er zeigen, daß die Schnittstelle der Komponente der Spezifikation entspricht. Beck nennt diese Funktionstests.

Verklemmungen/Zeitverhalten (T4): Das Finden von Verklemmungen (engl. Deadlock) und das Prüfen des Zeitverhaltens einer oder mehrerer Komponenten ist eine wichtige Aufgabe eines Testwerkzeugs. Im Component+ [Com01] Projekt wird erwähnt, daß Verklemmungen in einzelnen Komponenten unwahrscheinlich seien. Sie kommen eher im Zusammenspiel mit mehreren Komponenten vor.

Handelt es sich bei einer Komponentenanwendung um ein Echtzeitsystem, ist es notwendig, die benötigten Komponenten nicht nur auf die Funktionalität zu testen, sondern auch auf ihr Zeitverhalten zu untersuchen.

Testspezifikation (T5): Die Testspezifikation soll nach Morris *et al.* [MLP+01] sprachunabhängig und leicht zu lesen sein. Vorteilhaft sei es, wenn nicht nur die Testsoftware die Spezifikation versteht, sondern daß sie auch für den Entwickler verständlich ist. Es solle möglich sein, mehrere Testfälle für eine Komponente zu spezifizieren. Cox und Song [CS01] fordern, daß Testfälle mit jeweils den entsprechenden Komponenten mitgeliefert werden sollen.

Regressionstests (T6): Das wiederholte Testen der gleichen Testfälle, z.B. nach Änderungen an Quelltexten, wird Regressionstest genannt. Siepmann und

Newton [SN94] legen auf Regressionstests großen Wert. Wenn der Quelltext einer Komponente geändert werde, sei es notwendig zu testen, ob sich das Verhalten nicht geändert habe, bzw. ob Mängel eliminiert seien und daß sich keine neuen eingeschlichen haben. Beck und Gamma [BG98] bestärken dies, in dem sie sagen, daß Tests am Laufen bleiben müßten. Morris *et al.* [MLP+01] fügen hinzu, daß Regressionstests leicht und effizient durchführbar sein müßten, damit der Entwickler nicht die Lust am Testen verliert. Beck [Bec00] fordert, daß Regressionstests automatisiert werden müßten und nur eine positive bzw. negative Rückmeldung liefern dürften, ob sie erfolgreich waren.

Filtermechanismen (T7): Ein Ablaufverfolgungswerkzeug müsse Filtermechanismen anbieten, fordern Souder *et al.* [SMS01]. Dadurch sei es möglich, die großen Datenmengen, die bei der Analyse anfielen, zu reduzieren. Ein Werkzeug solle Regeln anbieten, um Daten auszuschließen bzw. wieder einzubinden.

Suchmöglichkeiten (T8): Nach Bassil und Keller [BK01] sind Suchmöglichkeiten für graphische und textuelle Elemente in einem Visualisierungswerkzeug notwendig. Sie machten in Unternehmen eine Untersuchung, die zeigte welche Kriterien für ein Visualisierungswerkzeug wichtig seien. Suchmöglichkeiten waren dabei die wichtigsten.

Farben (T9): Eine große Rolle spiele nach Bassil und Keller [BK01] auch Farbe.

Hierarchie (T10): Die Präsentation der Hierarchie eines Programms sei ebenfalls wichtig (nach Bassil und Keller [BK01]). Gefordert werde neben der Darstellung aber auch die Navigationsmöglichkeit durch die verschiedenen Ebenen.

Einstellen der Ansicht (T11): Für ein Visualisierungswerkzeug ist es wichtig, daß die Ansicht auf die Kommunikationsdaten individuell einstellbar und während der Visualisierung veränderbar ist.

Anhang B

Aufgabenstellung: Befragung zur Gebrauchstauglichkeit

Die Aufgabenstellung zu HotAgent steht auf den folgenden beiden Seiten.

Programm zum Verschicken von verschlüsselter oder unverschlüsselter ePost

Es soll ein Programm geschrieben werden, das ein Eingabefeld für Text, einen Auswahlknopf zum Verschlüsseln und einen Knopf zum Verschicken enthält.

1. Erstellt zuerst soll eine neue Komponente namens *„SemCod“* (‚New Composed Component'), die einen Text verschlüsselt. Diese besteht aus der gegebenen Verschlüsselungseinheit, einem Eingang und einem Ausgang.
   Der Verschlüsselungseinheit wird der Schlüssel durch einen ‚DataBlock' übergeben. Dazu wird der Wert „[Character value: 1]“ unter ‚Component Settings'/‚block'/‚value' eingetragen.
   Hinweis: Der zu verschlüsselnde Text muß sowohl an ‚MessageEntry' als auch ‚Do' von ‚MessageCodec' anliegen.

2. Schreibt einen Testfall für *„SemCod“*, mit dem geprüft wird, ob die Komponente richtig arbeitet (‚create c. c. test case').
   Der Testcase besteht hierbei aus einem ‚DataBlock' zur Testeingabe und einem ‚TestBlock', in dem der erwartete Ausgabewert eingegeben wird (unter ‚Component Settings'/‚block'/‚value' z.B. „[:testData | testData = *'erwartete Ausgabe'*]“).

3. Schreibt das Programm unter Zuhilfenahme von *SemCod*.
   Die hierzu benötigten Komponenten werden im folgenden erläutert.

Benötigte Komponenten:

- ***Data Block***
  Aufgabe: Interpretiert den eingegebenen Smalltalk Quelltext
  Ausgänge:
  - *data* interpretierte Daten
- ***MessageCodec***
  Aufgabe: Die Komponente MessageCodec ist eine nicht-visuelle Komponente. Sie soll die Nachricht an ihrem Eingang ver- bzw. entschlüsseln. Die Kodierung in der ersten Ausbaustufe ist eine reine Byte Verschlüsselung. Jedes Zeichen der Eingangsnachricht bekommt zu seinem ASCII-Wert den ASCII-Wert des Schlüssels aufaddiert. Bei der Entschlüsselung wird der Vorgang umgekehrt, dabei wird der Schlüssel von den einzelnen Zeichen der Nachricht wieder abgezogen.
  Eingänge:
  - *messageEntry* Dieser Eingang legt die Nachricht fest, die in der Komponente ver- bzw. entschlüsselt wird. Der Parameter ist eine Zeichenkette.
  - *decode* Dieser Eingang legt die Richtung der Kodierung fest. Der Parameter ist ein Wahrheitswert. Soll die Nachricht am Eingang messageEntry entschlüsselt werden, so liegt hier der Wert true an bzw. zum Verschlüsseln false.
  - *key* Dieser Eingang legt den Schlüssel für die Ver- bzw. Entschlüsselung fest. Der Parameter ist ein einzelnes Zeichen, später in höheren Ausbaustufen eine Zeichenkette.
  - *do* Dieser Eingang initiiert die Ver- bzw. Entschlüsselung der Nachricht. Es wird kein Parameter an diesem Eingang erwartet.

  Ausgänge:
  - *messageExit* An diesem Ausgang liegt die ver- bzw. entschlüsselte Nachricht an. Der Wert ist eine Zeichenkette.

- ***Split Result***
  Aufgabe: Leitet eine Anfrage an eine andere Komponente weiter und stellt den Rückgabewert an einem Ausgang zur Verfügung
  Eingänge:
  - *query* Eingehende Anfrage

Ausgänge:
- *query* Weitergeleitete Anfrage
- *result* Ergebnis der Anfrage

- ***SemEMail***
  Aufgabe: Sendet einen Text als ePost. Benutzt dazu localhost als SMTP-Server
  Eingänge:
  - *to* Empfängeradresse (als Zeichenkette)
  - *send with text* Text, der als ePost gesendet werden soll. Nach Setzen des Textes wird eine ePost gesendet (als Zeichenkette).

  Ausgänge:
  - *done* Gibt true zurück, wenn das Verschicken erfolgreich war.
- ***IfControl***
  Aufgabe: Die Komponente IfControl ist eine nicht-sichtbare Komponente. Sie soll als Entscheidungsstruktur dienen und bei positiven Vergleich an ihren Ausgängen einen Wahrheitswert anlegen.
  Eingänge:
  - *leftArg* An diesem Eingang wird das Argument links vom Vergleichsoperator gesetzt. Der Parameter ist eine Zahl/Wahrheitswert.
  - *rightArg* An diesem Eingang wird das Argument rechts vom Vergleichsoperator gesetzt. Der Parameter ist eine Zahl/Wahrheitswert .
  - *comSym* An diesem Eingang wird der Vergleichsoperator spezifiziert. Der Parameter ist ein Symbol und kann eines der folgenden Symbolen sein: #=, #<, #<=, #> oder #>=
  - *compare* Dieser Eingang aktiviert den Vergleich. Es wird kein Parameter an diesem Eingang erwartet.

  Ausgänge:
  - *ifTrue* An diesem Ausgang liegt ein Wahrheitswert true an, wenn der Vergleich positiv war.
  - *ifFalse* An diesem Ausgang liegt ein Wahrheitswert true an, wenn der Vergleich negativ war.
- ***Check Button***
  Aufgabe: Realisiert einen Auswahlknopf
  Eingänge:
  - *entrance* Eingang für Daten, die weitergeleitet werden, wenn Knopf ausgewählt ist

  Ausgänge:
  - *changed* Aktiviert, wenn der Status geändert wurde. Gibt true bzw. false zurück, je nachdem ob ausgewählt oder nicht ausgewählt wurde
  - *exit* Enthält Daten, die an *entrance* anliegen, wenn Knopf gewählt wurde
- ***Push Button***
  Aufgabe: Realisiert einen Druckknopf
  Ausgänge:
  - *perform action* Knopf wurde gedrückt
- ***Text Editor***
  Aufgabe: Eingabefeld für mehrzeilige Texte. (Achtung! Text muß per „Cut & Paste“ eingefügt werden - rechte Maustaste)
  Eingänge:
  - *query text* Erfragen des enthaltenen Textes (Rückgabewert ist eine Zeichenkette)
  - *text* Setzen eines Textes (Zeichenkette erwartet)

  Ausgänge:
  - *text changed Zeigt an, daß sich der Text geändert hat und liefert den neuen Text zurück* (Zeichenkette)

Anhang C

Fragebogen: Befragung zur Gebrauchstauglichkeit

Der Fragebogen zu HotAgent besteht aus den folgenden fünf Seiten.

Fragebogen zu HOTAGENT

Aufgabenangemessenheit:

HOTAGENT ...	--- -- - 0 + ++ +++	
ist kompliziert zu bedienen.	○ ○ ○ ○ ○ ○ ○	ist unkompliziert zu bedienen.
bietet schlechte Möglichkeiten, sich häufig wiederholende Bearbeitungsvorgänge zu automatisieren.	○ ○ ○ ○ ○ ○ ○	bietet gute Möglichkeiten, sich häufig wiederholende Bearbeitungsvorgänge zu automatisieren.
erfordert überflüssige Eingaben.	○ ○ ○ ○ ○ ○ ○	erfordert keine überflüssigen Eingaben.
verwendet uneinheitliche Bezeichnungen/Ausdrücke	○ ○ ○ ○ ○ ○ ○	verwendet durchweg einheitliche Begriffe/Ausdrücke.

Begründungen, Beispiele und Anmerkungen zu den oben gegebenen Antworten:

.....

.....

.....

Selbstbeschreibungsfähigkeit:

HOTAGENT ...	--- -- - 0 + ++ +++	
bietet einen schlechten Überblick über sein Funktionsangebot	○ ○ ○ ○ ○ ○ ○	bietet einen guten Überblick über sein Funktionsangebot.
verwendet schlecht verständliche Begriffe, Bezeichnungen, Abkürzungen oder Symbole in Masken und Menüs.	○ ○ ○ ○ ○ ○ ○	verwendet gut verständliche Begriffe, Bezeichnungen, Abkürzungen oder Symbole in Masken und Menüs.
liefert in unzureichendem Maße Informationen darüber, welche Eingaben zulässig oder notwendig sind.	○ ○ ○ ○ ○ ○ ○	liefert in zureichendem Maße Informationen darüber, welche Eingaben zulässig oder notwendig sind.
bietet auf Verlangen keine situationsspezifischen Erklärungen, die konkret weiterhelfen.	○ ○ ○ ○ ○ ○ ○	bietet auf Verlangen situationsspezifische Erklärungen, die konkret weiterhelfen.

bietet von sich aus keine situationsspezifischen Erklärungen, die konkret weiterhelfen.	○ ○ ○ ○ ○ ○ ○	bietet von sich aus situationsspezifische Erklärungen, die konkret weiterhelfen.

Begründungen, Beispiele und Anmerkungen zu den oben gegebenen Antworten:

.....

.....

.....

Steuerbarkeit:

HOTAGENT ...	--- -- - 0 + ++ +++	
bietet keine Möglichkeit, die Arbeit an jedem Punkt zu unterbrechen und dort später ohne Verluste wieder weiterzumachen.	○ ○ ○ ○ ○ ○ ○	bietet die Möglichkeit, die Arbeit an jedem Punkt zu unterbrechen und dort später ohne Verluste wieder weiterzumachen.
erzwingt eine unnötig starre Einhaltung von Bearbeitungsschritten.	○ ○ ○ ○ ○ ○ ○	erzwingt keine unnötig starre Einhaltung von Bearbeitungsschritten.
ermöglicht keinen leichten Wechsel zwischen einzelnen Menüs oder Masken.	○ ○ ○ ○ ○ ○ ○	ermöglicht einen leichten Wechsel zwischen einzelnen Menüs oder Masken.
ist so gestaltet, dass der Benutzer nicht beeinflussen kann, wie und welche Informationen am Bildschirm dargeboten werden.	○ ○ ○ ○ ○ ○ ○	ist so gestaltet, dass der Benutzer beeinflussen kann, wie und welche Informationen am Bildschirm dargeboten werden.

Begründungen, Beispiele und Anmerkungen zu den oben gegebenen Antworten:

.....

.....

.....

Erwartungskonformität:

HOTAGENT ...	--- -- - 0 + ++ +++	
erschwert die Orientierung, durch eine uneinheitliche Gestaltung.	○ ○ ○ ○ ○ ○ ○	erleichtert die Orientierung, durch eine einheitliche Gestaltung.

verwendet verwirrende Dialoge zur Eingabe von Daten.	○ ○ ○ ○ ○ ○ ○	verwendet einfache Dialoge zur Eingabe von Daten.
läßt einen im unklaren darüber, ob eine Eingabe erfolgreich war oder nicht.	○ ○ ○ ○ ○ ○ ○	läßt einen nicht im unklaren darüber, ob eine Eingabe erfolgreich war oder nicht.
reagiert mit schwer vorhersehbaren Bearbeitungszeiten.	○ ○ ○ ○ ○ ○ ○	reagiert mit gut vorhersehbaren Bearbeitungszeiten.
läßt sich nicht durchgehend nach einem einheitlichen Prinzip bedienen.	○ ○ ○ ○ ○ ○ ○	läßt sich durchgehend nach einem einheitlichen Prinzip bedienen.

Begründungen, Beispiele und Anmerkungen zu den oben gegebenen Antworten:

.....

.....

.....

Fehlertoleranz:

HOTAGENT ...	--- -- - 0 + ++ +++	
ist so gestaltet, daß kleine Fehler schwerwiegende Folgen haben können.	○ ○ ○ ○ ○ ○ ○	ist so gestaltet, daß kleine Fehler keine schwerwiegenden Folgen haben können.
informiert zu spät über fehlerhafte Eingaben.	○ ○ ○ ○ ○ ○ ○	informiert sofort über fehlerhafte Eingaben.
liefert schlecht verständliche Fehlermeldungen.	○ ○ ○ ○ ○ ○ ○	liefert gut verständliche Fehlermeldungen.
erfordert bei Fehlern im grossen und ganzen einen hohen Korrekturaufwand.	○ ○ ○ ○ ○ ○ ○	erfordert bei Fehlern im grossen und ganzen einen geringen Korrekturaufwand.
gibt keine konkreten Hinweise zur Fehlerbehebung.	○ ○ ○ ○ ○ ○ ○	gibt konkrete Hinweise zur Fehlerbehebung.
hilft nicht, Fehler zu vermeiden.	○ ○ ○ ○ ○ ○ ○	hilft Fehler, zu vermeiden.

Begründungen, Beispiele und Anmerkungen zu den oben gegebenen Antworten:

.....

.....

.....

Individualisierbarkeit:

HOTAGENT ...	--- -- - 0 + ++ +++	
ist so gestaltet, daß der Benutzer die Bildschirmdarstellung schlecht an seine individuellen Bedürfnisse anpassen kann.	○ ○ ○ ○ ○ ○ ○	ist so gestaltet, daß der Benutzer die Bildschirmdarstellung gut an seine individuellen Bedürfnisse anpassen kann.

Begründungen, Beispiele und Anmerkungen zu den oben gegebenen Antworten:

.....

.....

.....

Lernförderlichkeit:

HOTAGENT ...	--- -- - 0 + ++ +++	
erfordert viel Zeit zum Erlernen.	○ ○ ○ ○ ○ ○ ○	erfordert wenig Zeit zum Erlernen.
ermutigt nicht dazu, auch neue Funktionen auszuprobieren.	○ ○ ○ ○ ○ ○ ○	ermutigt dazu, auch neue Funktionen auszuprobieren.
erfordert, daß man sich viele Details merken muß.	○ ○ ○ ○ ○ ○ ○	erfordert nicht, daß man sich viele Details merken muß.
ist so gestaltet, daß sich einmal Gelerntes schlecht einprägt.	○ ○ ○ ○ ○ ○ ○	ist so gestaltet, daß sich einmal Gelerntes gut einprägt.
ist schlecht ohne fremde Hilfe oder Handbuch erlernbar.	○ ○ ○ ○ ○ ○ ○	ist gut ohne fremde Hilfe oder Handbuch erlernbar.
ist nicht für Benutzer mit verschiedenen Kenntnisständen geeignet.	○ ○ ○ ○ ○ ○ ○	ist für Benutzer mit verschiedenen Kenntnisständen geeignet.

Begründungen, Beispiele und Anmerkungen zu den oben gegebenen Antworten:

.....

.....

.....

Allgemeine Einschätzungen:

HOTAGENT ist ...	--- -- - 0 + ++ +++	
nicht intuitiv bedienbar	◯ ◯ ◯ ◯ ◯ ◯ ◯	intuitiv bedienbar
unflexibel	◯ ◯ ◯ ◯ ◯ ◯ ◯	flexibel
frustrierend	◯ ◯ ◯ ◯ ◯ ◯ ◯	nicht frustrierend
unübersichtlich	◯ ◯ ◯ ◯ ◯ ◯ ◯	übersichtlich

Hier ist noch Platz, um bisher nicht angesprochene Probleme zu nennen und Hinweise zu geben, wie HOTAGENT verbessert werden kann:

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Literaturverzeichnis

[ABB+02] Atkinson, Colin, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqa, Dirk Muthig, Barbara Raech, Jürgen Wüst und Jörg Zettel: *Component-based Product Line Engineering with UML*. Addison-Wesley, 2002.

[ACD02] Aldrich, Jonathan, Craig Chamber und Notkinm David: *ArchJava: Connecting Software Architecture to Implementation*. In *Proceedings of International Conference on Software Engineering 2002*, Mai 2002.

[Aon] Aonix: *SELECT Component Factory*. <http://www.aonix.com/content/products/select/compfact.html> (Mai 2002).

[Arn69] Arnheim, Rudolf: *Visual Thinking*. University of California Press, 1969.

[Bec94] Beck, Kent: *Simple Smalltalk Testing: With Patterns*. Smalltalk Report, Oktober 1994. <http://www.xprogramming.com/testfram.htm> (Februar 2002).

[Bec00] Beck, Kent: *Extreme Programming - Das Manifest*. Addison-Wesley, 2000.

[BG98] Beck, Kent und Erich Gamma: *Wie Programmierer das Test-Schreiben lieben lernen*. Java Spektrum, 5(15):22 – 32, September/Oktober 1998.

[BGL95] Burnett, Margaret, Adele Goldberg und Ted Lewis (Herausgeber): *Visual object-oriented programming: concepts and environments*. Manning Publications Co., 1995.

[Bir01] Birngruber, Dietrich: *CoML: Yet Another, But Simple Component Composition Language*. In Schneider, Jean-Guy und Markus Lumpe (Herausgeber): *Workshop on Composition Languages*, Seite 1–13, September 2001.

[BK01] Bassil, Sarita und Rudolf K. Keller: *Software Visualization Tools: Survey and Analysis*. Ninth International Workshop on Program Comprehension, Mai 2001.

[Ble01] Blevins, David: *Overview of the Enterprise Java Beans Component Model*. In Heineman, George T. und William T. Councill (Herausgeber): *Component-Based Software Engineering*, Seite 589 – 606. Addison Wesley, 2001.

[BM00] Behme, Henning und Stefan Mintert: *XML in der Praxis*. Addison-Wesley, München, 2000.

[BRCR01] Burnett, Margaret, Bing Ren, Curtis Cook und Gregg Rothermel: *Visually Testing Recursive Programms in Spreadsheet Languages*. In *IEEE Symposia on Human-Centric Computing Languages and Environments*, Seite 288 – 295, September 2001.

[Buc98] Buchner, Jürgen: *HotDoc – Ein flexibles System für den kooperativen Aufbau zusammengesetzter Dokumentstrukturen*. Dissertation, Technische Universität Darmstadt, 1998.

[Buc00] Buchner, Jürgen: *HotDoc, a framework for compound documents*. ACM Computing Surveys, 32(1), März 2000.

[Cam] Camp Smalltalk: *ANSI-ST-tests Project*. <http://ANSI-ST-tests.sourceforge.net/> (Februar 2002).

[CH01] Councill, Bill und George T. Heineman: *Definition of a Software Component and Its Elements*. In Heineman, George T. und William T. Councill (Herausgeber): *Component-Based Software Engineering*, Seite 5 – 19. Addison Wesley, 2001.

[Cla01] Clausius, Thorsten: *Komponenten zur Konstruktion von Agenten*. Diplomarbeit, Fachbereich Informatik, FG Programmiersprachen und Übersetzer, Technische Universität Darmstadt, 2001.

[Com01] Componente+ Partners: *Built-In Testing for Component Based Development*. Fachbericht, EC IST 5th Framework Project IST-1999-20162 Component+, November 2001. <http://www.component-plus.org> (Juli 2002).

[CS94] Cox, Philip T. und Trevor J. Smedley: *Using Visual Programming to Extend the Power of Spreadsheet Computation*. Proceedings of the Workshop on Advanced Visual Interfaces, Seite 153 – 161, 1994.

[CS01] Cox, Philip T. und Baoming Song: *A Formal Model for Component-Based Software*. In *IEEE Symposia on Human-Centric Computing Languages and Environments*, Seite 304 – 311, September 2001.

[DDN03] Demeyer, Serge, Stéphane Ducasse und Oscar Nierstrasz: *Object-Oriented Reengineering Patterns*. Morgan Kaufmann Publishers, 2003.

[Dij72] Dijkstra, Edsger Wybe: *Notes on structured programming*. In Dahl, Dljkstra und Horne (Herausgeber): *Structured Programming*. Academic Press, 1972.

[ebX] *ebXML - Enabling A Global Electronic Market*. <http://www.ebxml.org/> (November 2002).

[Eng97] Englander, Robert: *Developing Java Beans*. O'Reilly, 1997.

[FMMB02] Fiege, Ludger, Mira Mezini, Gero Mühl und Alexandro P. Buchmann: *Engineering Event-Based Systems with Scopes*. 16th European Conference on Object-Oriented Programming, 2002.

[Gie01] Giesl, Anke: *Evaluierung von Komponenten-Entwicklungsumgebungen*. Diplomarbeit, Fachbereich Informatik, FG Programmiersprachen und Übersetzer, Technische Universität Darmstadt, November 2001.

[GLM98] Geiger, Christian, Georg Lehrenfeld und Wolfgang Müller: *Authoring Communicating Agents in Visual Environments*. In *Proceedings of the Australasian Computer Human Interaction Conference*, 1998.

[GMR98] Geiger, Christian, Wolfgang Müller und Waldemar Rosenbach: *SAM – An Animated 3D Programming Language*. In *IEEE Symposium on Visual Languages*, Seite 228 – 235, August 1998.

[GP96] Green, T. R. G. und Marian Petre: *Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework*. Journal of Visual Languages and Computing, 7(2):131–174, 1996.

[GZS01] Gao, Jerry, Eugene Y. Zhu und Shim Simon: *Tracking Software Components*. Journal of Object-Oriented Programming, Seite 13 – 22, Oktober/November 2001.

[Han01] Handl, Daniela: *HotFlow: E-Commerce processes from a language/action perspective*. In Dilip Ratel et al. (Herausgeber): *OOIS*

2000. Proceedings of the 6th International Conference on Object Oriented Information Systems, Seite 95 – 101, 2001.

[HGZ99] Hong, Liu, Zeng Guangzhou und Lin Zongkai: *A Construction Approach for Software Agents Using Components.* ACM SIGSOFT Software Engineering Notes, 24(3):76 – 79, Mai 1999.

[HH99a] Handl, Daniela und Hans-Jürgen Hoffmann: *Document exchange as a basis for business-to-business co-operation.* In Roger, J.-Y. und et al. (Herausgeber): *Business and Work in the Information Society*, Band 2, Seite 325 – 331, Amsterdam, 1999. IOS Press.

[HH99b] Handl, Daniela und Hans-Jürgen Hoffmann: *Workflow agents in the document–centred communication in MALL2000 systems.* 1st Intl. Workshop on Agent-oriented information systems, 1999. http://www.aois.org/99/handl.html.

[HHM03] Handl, Daniela, Hans-Jürgen Hoffmann und Ludger Martin: *Enhancing the E-business community by software component technology.* International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, e-Medicine, and Mobile Technologies on the Internet, Januar 2003. CD-ROM.

[Hil99] Hilmer, Stefan: *Objektorientierte Agenten - Synthese und Anwendung zweier Technologien.* 5. Fachkonferenz Smalltalk und Java, September 1999.

[HLS99] Harrold, Mary Jean, Donglin Liang und Saurabh Sinha: *An Approach to Analyzing and Testing Component-Based Systems.* Proceedings of the First International ICSE Workshop on Testing Distributed Component-Based Systems, Mai 1999.

[Hof01a] Hoffmann, Hans-Jürgen: *"Less is more" in B2B.* Proceedings SSGRR 2001, International Conference Advances in Infrastructure for Elektronic Business, Science, and Education on the Internet, August 2001. CDROM.

[Hof01b] Hoffmann, Hans-Jürgen: *Unterstützen elektronischer Geschäftsprozesse: Das HOTxxx-Projekt.* In Horster, P. (Herausgeber): *Elektronische Geschäftsprozesse*, Seite 227–241. it-Verlag, 2001.

[HW96] Hartman, Jed und Josie Wernecke: *The VRML 2.0 Handbook.* Addison Wesley, 1996.

[IH01] Ivory, Melody Y. und Marti A. Hearst: *The State of the Art in Automating Usability Evaluation of User Interfaces*. ACM Computer Surveys, 33(4):470 – 516, Dezember 2001.

[Joh97] Johnson, Ralph E.: *Frameworks = (Components + Patterns)*. Communications of the ACM, Seite 39 – 42, Oktober 1997.

[Knu92] Knuth, Donald E.: *Literate Programming*. University of Chicago Press, 1992.

[KS01] Karan, Marcel R. und Trevor J. Smedley: *A Testing Methodology for Dataflow Based Visuel Programming Language*. In *Symposia on Human-Centric Computing Languages and Environments*, Seite 280 – 287, September 2001.

[Kun02] Kunstmann, Thomas: *Rechnergestützte Simulation und Planung auf der Grundlage von Tabellenkalkulation*. Dissertation, Technische Universität Darmstadt, 2002.

[Lie96] Lieberherr, Karl J.: *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.

[Lod83] Lodding, Kenneth N.: *Iconic Interfacing*. IEEE Computer Graphics and applications, Seite 11 – 20, März/April 1983.

[LR00] Lüer, Chris und David S. Rosenblum: *Wren – An Environment for Component-Based Development*. Fachbericht, Department of Information and Computer Science, University of California, Irvine, September 2000.

[LR01] Lüer, Chris und David S. Rosenblum: *Wren – An Environment for Component-Based Development*. In *Proceedings of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Seite 207 – 217, September 2001.

[Lut87] Lutze, Rainer: *Die Implementierung von Smalltalk*. In Hoffmann, Hans-Jürgen (Herausgeber): *Smalltalk verstehen und anwenden*, Seite 129 – 188. Hanser, 1987.

[Mar] Martin, Ludger: *HotAgent homepage*. <http://www.gkec.informatik.tu-darmstadt.de/HotAgent/> (Juli 2002).

[Mar00] Martin, Ludger: *Visualisierung von Komponenten für Benutzungsoberflächen*. Diplomarbeit, Fachbereich Informatik, FG Programmiersprachen und Übersetzer, Technische Universität Darmstadt, 2000.

[Mar01] Martin, Ludger: *Visual Development Environment Based on Component Technique*. In *Proceedings IEEE Symposia on Human-Centric Computing Languages and Environments*, Seite 346 – 347, September 2001.

[Mar02a] Martin, Ludger: *Visual Component Integration and Regression Test*. In *ICSR7 2002 Workshop on Component-based Software Development Processes*, April 2002.

[Mar02b] Martin, Ludger: *Visual Composition of Components*. In *The 6th IASTED International Conference Software Engineering and Applications*, Seite 501 – 508, November 2002.

[McC97] McCarthy, Adrian: *Unit and Regression Testing*. Dr. Dobbs Journal, Februar 1997.

[Mey87] Meyer, Bertrand: *Design by Contract, Technical Report TR-EI-12/CO*. Fachbericht, ISE Inc., 1987.

[Mey90] Meyer, Bertrand: *Objektorientierte Softwareentwicklung*. Hanser, 1990.

[Mey99a] Meyer, Bertrand: *Design by Contract, Components and Debugging*. The Journal of Object-Oriented Programming, 11(8):75 – 79, Januar 1999.

[Mey99b] Meyer, Bertrand: *On To Components*. IEEE Computer, 32(1):139 – 140, Januar 1999.

[MGM02a] Martin, Ludger, Anke Giesl und Johannes Martin: *Dynamic Component Program Visualization*. Working Conference on Reverse Engineering, Seite 298 – 298, Oktober 2002.

[MGM02b] Martin, Ludger, Anke Giesl und Johannes Martin: *Dynamische Komponenten-Programm-Visualisierung*. 4. Workshop Software-Reengineering, April 2002.

[MH01a] Mezini, Mira und Michael Haupt: *Neue Programmierparadigmen: Integrationsorientierte Programmierung*. OBJEKTspektrum, Seite 48 – 54, März/April 2001.

[MH01b] Monson-Haefel, Richard: *Enterprise Java Beans.* O'Reilly, 2001.

[MK02] Maruhn, Steven und Till Kofink: *Experimentelle Untersuchung zu HotAgent (Seminarausarbeitung).* Fachbericht, Technische Universität Darmstadt, SS 2002.

[ML98] Mezini, Mira und Karl Lieberherr: *Adaptive Plug-and-Play Components for Evolutionary Software Development.* In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, Seite 07 – 116, 1998.

[MLP+01] Morris, John, Gareth Lee, Kris Parker, Gary A. Bundell und Chiou Peng Lam: *Software Component Certification.* IEEE Computer, Seite 30 – 36, September 2001.

[MMS98] Meyer, Bertrand, Christine Mingins und Heinz Schmidt: *Providing Trusted Components to the Industry.* IEEE Computer, 31(5):104 – 105, Mai 1998.

[MS00a] Martin, Ludger und Elke Siemon: *Component Visualization Based on Programmer's Conceptual Models.* In *OOPSLA ´00 Companion*, Seite 73 – 74, 2000.

[MS00b] Matena, Vlada und Beth Stearns: *Applying Enterprise Java Beans.* Addison Wesley, 2000.

[Mye90] Myers, B.: *Taxonomies of Visual Programming and Program Visualization.* Journal of Visual Languages and Computing, 1(1):97, 12 1990.

[NM94] Nielsen, Jakob und Robert L. Mack: *Usability Inspection Methods.* John Wiley & Sons, Inc, New York, 1994.

[OMRK92] Oppermann, Reinhard, Bernd Murchner, Harald Reiterer und Manfred Koch: *Software-ergonomische Evaluation - Der Leitfaden EVADIS II.* de Gruyter, Berlin; New York, 1992.

[Par76] Parnas, David L.: *Some Hypotheses About the "Uses" Hierarchy for Operating Systems, Forschungsbericht BS I 76/1.* Fachbericht, Technische Hochschule Darmstadt, März 1976.

[Pau01] Pauli, Kevin: *Pattern your way to automated regression testing.* JavaWorld, September 2001. <http://www.javaworld.com/javaworld/jw-09-2001/jw-0921-test.html> (Februar 2002).

[Pet95] Petre, Marian: *Why looking isn't always seeing: readership skills and graphical programming.* Communications of the ACM, 38(6):33 – 44, Juni 1995.

[PPW02] Pichler, J., R. Plösch und R. Weinreich: *MASIF und FIPA: Standards für Agenten - Übersicht und Anwendung.* Informatik Spektrum, Seite 91 – 100, April 2002.

[Rau98] Rausch, Martin: *AgentSheets - Programming above C-Level.* Computer Graphic Topics, 10:10 – 12, 1998.

[RC93] Repenning, Alexander und Wayne Citrin: *Agentsheets: Applying Grid-Based Spatial Reasoning to Human-Computer Interaction.* In *IEEE Workshop on Visual Languages*, Seite 77–82, Bergen, Norway, 1993.

[Rei97] Reiss, Steven P.: *Cacti: A Front End for Program Visualization.* In *Proceedings of the 1997 IEEE Symposium on Information Visualization*, Seite 46 – 49, 1997.

[Rep91] Repenning, Alexander: *Creating User Interfaces with Agentsheets.* In *Proceedings of the 1991 IEEE Symposium on Applied Computing*, Seite 190–196, Kansas City, Missouri, April 1991.

[Rep93] Repenning, Alexander: *Agentsheets: A Tool for Building Domain–Oriented Visual Programming Environments.* In *Proceedings of the Conference on Human Factors in Computing Systems*, Seite 142 – 143, 1993.

[Rep95] Repenning, Alexander: *Bending the Rules: Steps Toward Semantically Enriched Graphical Rewrite Rules.* In *Proceedings of Visual Languages*, Seite 226–233, Darmstadt, Germany, 1995.

[Rep00] Repenning, Alexander: *AgentSheets: an Interactive Simulation Environment with End-User Prorammable Agents.* In *Interaction 2000*, Tokyo, Japan, 2000.

[RIP+01] Repenning, Alexander, Andri Ioannidou, Michele Payton, Wenming Ye und Roschelle Jeremy: *Using Components for Rapid Distributed Software Development.* IEEE Software, Seite 38 – 45, März/April 2001.

[RS95] Repenning, Alexander und T. Sumner: *Agentsheets: A Medium for Creating Domain-Oriented Visual Languages.* IEEE Computer, 28(3):17–25, März 1995.

[San99] Sandholm, Tuomas: *Automated Negotiation.* Communications of the ACM, Seite 84 – 85, März 1999.

[Sch96] Schmucker, Kurt J.: *Rapid Prototyping using Visual Programming Tools.* Proceedings of the CHI'96 Conference on Human Factors in Computing Systems, Seite 359 – 360, April 1996.

[Sch98] Schiffer, Stefan: *Visuelle Programmierung.* Addison Wesley, 1998.

[Shu88] Shu, Nan C.: *Visual Programming.* Van Nostrand Rheinhold Company, New York, 1988.

[Sie01] Siemon, Elke: *Über den Entwurf von Benutzungsschnittstellen technischer Anwendungen mit visuellen Spezifikationsmethoden und Werkzeugen.* Dissertation, Technische Universität Darmstadt, 2001.

[SML99] Seiter, Linda, Mira Mezini und Karl Lieberherr: *Dynamic Component Gluing.* In Eisenecker, Ulrich und Krzysztof Czarnecki (Herausgeber): *First International Symposium on Generative and Component-Based Software Engineering*, 1999.

[SMS01] Souder, Tim, Spiros Mancoridis und Maher Salah: *Form: A Framework for Creating Views of Programm Executions.* In *Proceedings IEEE International Conference on Software Maintenance*, Seite 612 – 620, November 2001.

[SN94] Siepmann, Ernst und A. Richard Newton: *TOBAC: a test case browser for testing object-oriented software.* In *Proceedings of the 1994 international symposium on software testing and analysis*, Seite 154 – 168, 1994.

[SUN99] SUN microsystems: *InfoBus 1.2 Specification*, Februar 1999. <http://java.sun.com/products/javabeans/infobus/spec12/IB12Spec.htm> (März 2003).

[Szy98] Szyperski, Clemens: *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

[Völ02] Völker, Markus: *Grundlagen von Komponenteninfrastrukturen.* Java Spektrum, Seite 11 – 17, März/April 2002.

[Wan93] Wandmacher, Jens: *Software-Ergonomie.* de Gruyter, Berlin; New York, 1993.

[Wey01] Weyuker, Elaine J.: *The Trouble with Testing Components*. In Heineman, George T. und William T. Councill (Herausgeber): *Component-Based Software Engineering*, Seite 499 – 512. Addison Wesley, 2001.

[WH01] Wagner, Thomas und Bryan Horling: *The Struggle for Reuse and Domain Independence: Research with TAEMS, DTC, and JAF*. Proceedings of the 2nd Workshop on Infrastructure for Agents, MAS, and Scalable MAS (Agents 2001), Juni 2001.

[Wig98] Wiggins, Melissa: *An Overview of Program Visualization Tools and Systems*. In *Proceedings of the 36th annual Southeast regional conference*, Seite 194 – 200, 1998.

[Win90] Winkler, Jürgen F. H.: *Visualisierung in der Software-Entwicklung*. Informatik-Fachberichte, GI 20. Jahrestagung 1, Seite 40 – 72, Oktober 1990.

[WS01] Weinreich, Rainer und Johannes Sametinger: *Component Models and Component Services: Concepts and Principles*. In Heineman, George T. und William T. Councill (Herausgeber): *Component-Based Software Engineering*, Seite 33 – 48. Addison Wesley, 2001.

Index

Lebenslauf

Name:	Ludger MARTIN
Geburtsdatum:	9. Juli 1973
Geburtsort:	Mainz

Schulbildung:

1980 – 1984	Grundschule in Mainz
1984 – 1991	Gymnasium in Mainz und Wiesbaden
1991 – 1994	Technisches Gymnasium in Mainz Abschluß: Allgemeine Hochschulreife

Studium:

10/1994 – 9/1995	Studium der Elektrotechnik an der Technischen Universität Darmstadt
10/1995 – 6/2000	Studium der Informatik an der Technischen Universität Darmstadt Nebenfach: Flugverkehrstechnik, Berufspädagogik Thema der Diplomarbeit: „Visualisierung von Komponenten für Benutzungsoberflächen“ Abschluß: Diplom
1/1999 – 4/1999	Auslandsstudium an der „University of Victoria“ in Victoria BC, Kanada
seit 7/2000	DFG-Promotionsstipendium am Graduiertenkolleg „Infrastruktur für den elektronischen Markt“ im Fachbereich Informatik der Technischen Universität Darmstadt

Erklärung

Hiermit erkläre ich, die vorgelegte Arbeit zur Erlangung des akademischen Grades „Dr.-Ing.“ mit dem Titel „Visuelles Komponieren und Testen von Komponenten am Beispiel von Agenten im elektronischen Handel“ selbstständig und ausschließlich unter Verwendung der angegebenen Hilfsmittel erstellt zu haben. Ich habe bisher noch keinen Promotionsversuch unternommen.

Darmstadt, den 24. März 2003 Ludger Martin