C.v.O. Universität Oldenburg

Studiengang Diplom-Informatik

Diplomarbeit

VEGES:
Framework zur Verknüpfung von Geodaten mit beliebigen Sachdaten am Beispiel von InterGIS

vorgelegt von: Thorben Kundinger

Gutachter: Prof. Dr. H.-J. Appelrath
Zweitgutachter: Dipl. Inform. Jörg Friebe

Oldenburg, November 2000

Danksagung

Für die ausgezeichnete Betreuung meiner Diplomarbeit möchte ich mich bei Holger Jaekel bedanken. Weiterer Dank gilt meinen Eltern, welche mir mein Studium ermöglichten. Ferner bedanke ich mich bei Dagmar Wendt, Martin Sparenberg und Jörg Baldzer für ihre Anregungen und das geduldige Korrekturlesen dieser Arbeit.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **5**
2.1 Objektorientierte Programmierung 5
2.2 Entwurfsmuster 7
2.3 Frameworks 9
2.3.1 Ziele 11
2.3.2 Wiederverwendung 11
2.3.3 Vor- und Nachteile 14
2.3.4 Dokumentation 15
2.4 Komponenten 17
2.4.1 Schnittstellen und Dokumentation 19
2.4.2 Komponenten gleich Software-ICs? 20
2.4.3 Komponentenmarkt 21
2.4.4 Sicherheitsaspekte 22
2.4.5 Generative Programmierung 25
2.4.6 Vor- und Nachteile 26

3 Entwurf von Veges **31**
3.1 Zielsetzung 31
3.2 Veges-Framework 32
3.2.1 Zugangsserver 33
3.2.2 Framework-Kern 35
3.2.3 Konfigurationsdaten 39
3.2.4 Sicherheitsaspekte 41
3.3 Schnittstellen 41
3.3.1 DBMS und Sachdatenkomponenten 42
3.3.2 Geodatenkomponente und GIS 44
3.3.3 Zugangsserver-Protokoll 44
3.3.4 Komponenten-Protokoll 48
3.4 Komponentengenerator 53

3.4.1 Spezifikation 55
3.4.2 Assistent 56

4 Implementierung von Veges 57
4.1 Wahl der Implementierungssprache 57
4.2 Framework-Kern 60
4.2.1 Konfigurieren von Komponenten 61
4.2.2 Laden von Komponenten 62
4.2.3 Beobachter 64
4.3 Zugangsserver 66
4.4 Komponentengenerator 69
4.4.1 Spezifiktionsdatei 70
4.5 Sachdatenkomponente 73
4.6 Assistent 75
4.6.1 Sonderfälle 76
4.6.2 Sekundärschlüsselformat 77
4.7 InterGIS 77
4.7.1 InterGISGeodatenKomponente 77
4.7.2 SSOZugangsServer 79
4.8 Beispiele 81
4.8.1 SplitPaneDemo 81
4.8.2 TabbedPaneDemo 82
4.8.3 InternalFrameDemo 82

5 Zusammenfassung und Ausblick 85

A Unified Modeling Language 87

B Abkürzungsverzeichnis 89

C Glossar 93

Literaturverzeichnis 97

Index 103

Abbildungsverzeichnis

2.1 Entwurfsmuster Beobachter 8

3.1 Proxy für mehrere Zugangsserver 34
3.2 Zentraler Entwurf des Frameworks 36
3.3 Dezentraler Entwurf des Frameworks 36
3.4 Adapter 37
3.5 Konfigurationsbaum einer Komponentenklasse 40
3.6 Wiederverwendung der Protokolle nach der Umstellung 43
3.7 Transparenz durch Trennung der Protokolle 43
3.8 Auswirkungen des Aktion-Feldes 51
3.9 Prozess zur Erzeugung von Komponenten 53

4.1 Hierarchie der VEGES-Interfaces 61
4.2 Klasse `Eigenschaften` 62
4.3 Beobachter-Klassenhierarchie 65
4.4 Zugangsserver-Klassenhierarchie 66
4.5 Klasse `ZugangsServerInterface` 67
4.6 Bildschirmausschnitt einer generierten Sachdatenkomponente . 74
4.7 Klassenübersicht einer erzeugten Sachdatenkomponente 74
4.8 Bildschirmausschnitt des Assistenten 76
4.9 Klassenhierarchie der `InterGISGeodatenKomponente` 78
4.10 Klasse `InterGISGeodatenKomponente` 79
4.11 Klassenhierarchie des `SSOZugangsServers` 81
4.12 Bildschirmausschnitt des `SplitPaneDemos` 82
4.13 Bildschirmausschnitt des `TabbedPaneDemos` 82
4.14 Bildschirmausschnitt des `InternalFrameDemos` 83

A.1 Klassen und Schnittstellen 87
A.2 Sichtbarkeit von Attributen und Methoden 87
A.3 Vererbung und Schnittstellenimplementation 88
A.4 Aggregation 88
A.5 Pakete und Notizen 88

Kapitel 1

Einleitung

Bei konventionellen Geografischen Informationssystemen (GIS) (etwa SICAD/SD oder InterGIS) werden ergänzende Sachdaten zu den raumbezogenen Daten als Attribute direkt an die geografischen Objekte „angehängt“. Werden die Sachdaten nicht ebenfalls bei der Vermessung miterfaßt, so stammen sie aus anderen Informationssystemen und müssen nachträglich eingespielt und konsistent gehalten werden. Bei einem Wechsel des GIS müssen die Sachdaten in das neue System übertragen werden.

In vielen Anwendungszenarien wird die vom GIS erzeugte Karte nur zur Vereinfachung der Orientierung bei der Bearbeitung der Sachdaten verwendet. Als Beispiel sei eine Adressdatenbank genannt, welche den Wohnort mit Hilfe eines GIS anzeigt. Anstatt nun die Sachdaten in ein spezielles GIS zu integrieren, könnte man die Anbindung beliebiger GIS ermöglichen, welche über eine wohldefinierte Schnittstelle mit dem „Sachdatensystem“ kommunizieren.

Die rasante technologische Entwicklung der Rechner ermöglichte GIS neue Einsatzgebiete. Mit der stetigen Zunahme der Speicherkapazität sank die relative Größe der Datenmengen, welche im GIS verwaltet werden. Sie erfordern zwar absolut gesehen immer noch große oder sogar steigende Kapazitäten, aber diese sind mittlerweile zu geringeren Kosten zu erhalten. Diese Entwicklung ermöglicht einen breiteren Einsatz dieser Systeme. In vielen dieser neu entstandenen Einsatzgebiete dienen die Geodaten nur noch zur Visualisierung der Sachdaten. In solchen Fällen wird neben dem schon vorhandenen Informationssystem für klassische, tabellenorientierte Daten ein GIS für geografische Daten hinzugefügt.

Noch müssen beide Informationssysteme – ein GIS für die Raumdaten und ein üblicherweise relationales Datenbanksystem für die Sachdaten – zum größeren Teil getrennt erworben und aufeinander abgestimmt werden, aber mit steigender Verbreitung dieser Nutzungsform werden Komplettsysteme

der Regelfall. Diese werden am Anfang dann beide Informationssysteme als jeweils optimierte Teilsysteme enthalten. Eventuell wird auch die jeweilige Optimierung mit steigender Rechenleistung für eine einfachere Struktur fallengelassen, ähnlich wie bei dem relationale Datenmodell, das auch bei Abfragen weniger effizient als das Netzwerkdatenmodell ist, dafür aber eine einfachere Datenspeicherung erlaubt.

Eine Brücke zwischen beiden „Systemwelten“ zu schlagen, ist das Ziel dieser Diplomarbeit. Sie soll die Entwicklung von Anwendungen zur Bearbeitung von verknüpften Geo- und Sachdaten vereinfachen. Dieses Ziel soll mit Hilfe eines komponentenbasierten Frameworks erreicht werden, einer Variante der komponentenbasierten Softwareentwicklung (Griffel 1998). Bei der komponentenbasierten Softwareentwicklung werden vorgefertigte Bausteine nur mit dem Wissen über ihre Schnittstellen, nicht über ihren inneren Aufbau, zu einem Produkt integriert. Ein Framework ist eine abstrakte Lösung für eine bestimmte Art von Problemen (etwa Benutzungsoberflächen oder Buchhaltung), welches für den Einsatz noch konkretisiert werden muß (Mattsson 1999; Johnson 1992). Bei einem komponentenbasierten Framework wird die Konkretisierung durch die Auswahl und Konfiguration der benötigten Komponenten erreicht. Dieses Vorgehen ist vergleichbar mit der Produktion von elektronischen Geräten mit Hilfe von integrierten Schaltkreisen (Integrated Circuit (IC)). Bei einem Netzteil beispielsweise hat der IC zur Spannungskontrolle eine feste Schnittstelle durch seine Pins am Gehäuse (Komponente), und wird mit Hilfe von Kondensatoren und Widerständen konfiguriert. Die Schaltung hat trotz der unterschiedlichen erzeugten Spannungen und Stromstärken immer den gleichen grundlegenden Aufbau (Framework).

Der Anwendungsentwickler soll zusätzlich durch die Entwicklung eines Generators für Komponenten zur Bearbeitung von Sachdaten unterstützt werden. Dieser Generator soll die Komponenten gemäß einer Spezifikation automatisch erzeugen können, aber auch auf Wunsch entsprechende Zwischenprodukte generieren, welche dann nachbearbeitet werden können. Durch die einzelnen Schritte bei der Erstellung einer Spezifikation soll der Entwickler mit Hilfe eines Assistenten geleitet werden.

Zur Visualisierung der Geodaten wird ein vorhandenes Modul der Inter-GIS-Arbeitsgruppe des Oldenburger Forschungs- und Entwicklungsinstitut für Informatik-Werkzeuge und -Systeme (OFFIS) zu einer zu diesem Framework kompatiblen Komponente erweitert werden. Die dafür zu entwickelnde Schnittstelle soll offen genug für andere Anzeigemodule von anderen GIS sein.

Die Entwicklung eines Anwendungssystems würde dann wie folgt ablaufen: Nachdem ein Bedarf für eine neue Anwendung festgestellt und ihre Anforderungen definiert worden sind, werden die Abfragen an die Sachdatenbank für die Dialoge mit dem Benutzer erstellt und die Verknüpfung zwischen den

beiden Datenquellen festgelegt. Mit diesen Informationen kann wahlweise der Assistent zur Erstellung der Spezifikation des Komponenten-Generators aufgerufen werden oder die Spezifikation manuell erstellt werden. Werden komplexere Dialoge bei der Sachdatenkomponente benötigt, wird die Benutzungsoberfläche der vom Assistenten erzeugten Komponente nachbearbeitet werden müssen. Nicht alle Randbedingungen zwischen den Datenfeldern sind in der Struktur der Datenbank enthalten, sondern sie ergeben sich erst aus dem Anwendungskontext. Danach kann diese Komponente zusammen mit dem Framework und der passend ausgewählten Geodatenkomponente ausgeliefert werden. Für den Einsatz beim Kunden müssen dann noch letzte Anpassungen vorgenommen werden, etwa die Zugriffspfade für die Datenquellen konfigurieren. Danach ist die Anwendung einsatzbereit.

Zu beachten ist, daß bei dieser Lösung aufgrund der nicht so großen Komplexität des Frameworks nicht die von Ritter (2000) vorgeschlagene Trennung zwischen dem Entwickler der Komponente(n) und dem Konfigurator der Anwendung vorgenommen wurde. Der Entwickler der Sachdatenkomponente vereint diese beiden Rollen in einer Person. Die Entwicklung einer Sachdatenkomponente kann als Konfiguration im weiteren Sinne gesehen werden. Die Gründe dafür werden insbesondere in Kapitel 3 (Entwurf) erläutert.

Im zweiten Kapitel werden die für den Entwurf und die darauf folgende Implementierung des VEGES-Frameworks zugrundeliegende Konzepte erläutert (VEGES steht für „*V*erknüpfung von *Ge*odaten mit *S*achdaten“). Nach einer kurzen Einleitung über die objektorientierte Programmierung und Entwurfsmuster wird das Framework-Konzept als Möglichkeit der sogenannten White-Box-Wiederverwendung ebenso vorgestellt, wie die Komponenten, eine Variante der sogenannten Black-Box-Wiederverwendung. Komponenten sind das Thema des letzten Abschnittes des zweiten Kapitels.

Diese Grundlagen werden bei der Erstellung des Entwurfes berücksichtigt, welcher im dritten Kapitel beschrieben wird. Zuerst werden die Anforderungen an das zu erstellende VEGES-Framework mit Hilfe eines typischen Anwendungsszenarios ermittelt. Danach wird die grundlegende Aufteilung in die verschiedenen Komponenten vorgenommen, welche sich zum Teil schon aus den externen Anforderungen ergibt. Abgeschlossen wird das dritte Kapitel mit einer Beschreibung des Komponentengenerators und seines Assistenten.

Im vierten Kapitel werden ausgewählte Probleme bei der Implementierung des Frameworks und ihre Lösung beschrieben.

Abgeschlossen wird diese Arbeit in Kapitel 5 durch eine Zusammenfassung der gewonnenen Erkenntnisse und einen Ausblick auf die Möglichkeiten eines weiteren Ausbaus von VEGES.

Kapitel 2

Grundlagen

Eines der Hauptanliegen der Softwaretechnik ist es, die Wiederverwendung von Konzepten, Entwürfen, Testdaten und vor allem der Quelltexte bei der Entwicklung von Software zu erhöhen. Im Vergleich zur Halbleiterindustrie, wo z.B. beim Standardzellenentwurf zum großem Teil auf vorgefertigte Schaltungselemente zurückgegriffen wird, werden noch relativ viele Programme „von Grund“ auf neu entwickelt (Griffel 1998). Mit der Einführung der objektorientierten Programmierung wurde die Hoffnung verbunden, den Grad der Wiederverwendung zu erhöhen. Durch die dort eingeführten OO-Konzepte wurde auch die Entwicklung von Entwurfsmustern und Frameworks vorangetrieben. Ergänzend wird mit dem Ansatz der Komponentenbildung versucht eine höhere Entkoppelung bei der Softwareentwicklung zu erreichen. Diese Konzepte und Techniken sollen im folgenden näher erläutert werden.

2.1 Objektorientierte Programmierung

Objektorientierte Verfahren werden immer häufiger beim Entwurf und bei der Implementierung von Softwareprogrammen eingesetzt. Die objektorientierte Programmierung besteht im wesentlichen aus drei Konzepten: Kapselung, Vererbung und Polymorphismus. Im Mittelpunkt steht das Objekt. Ein Objekt soll eine modellhafte Abbildung von realen Gegenständen oder Personen sein. Die abgebildeten Eigenschaften werden Attribute genannt. Neben den Attributen gibt es noch Methoden, welche Operationen auf den Attributen ermöglichen. Eine Klasse ist die Vorlage für ein Objekt, ähnlich wie ein Variablentyp in einer typisierten Programmiersprache die Vorlage für eine konkrete Variable ist. Die von einer Klasse erzeugten konkreten Objekte werden Instanzen genannt.

Bei der Kapselung werden die Attribute eines Objektes vor anderen Ob-

jekten „versteckt“. Auf die gekapselten Attribute kann nur noch mittels der Methoden des Objektes zugegriffen werden, dem die Attribute gehören. In einem Objekt werden die Attribute mit den zu ihrer Bearbeitung notwendigen Methoden zusammengefasst.

Eine Klasse ist nicht nur eine Vorlage zum Erzeugen von Instanzen, sie kann auch als Basis für weitere Klassen dienen. Im diesem Fall spricht man von Vererbung. Bei der Vererbung gibt es zwei Varianten, die Einfach- und die Mehrfachvererbung. Bei der Einfachvererbung bekommt die neue Klasse alle Attribute und Methoden ihrer einen Vorlage, bei der Mehrfachvererbung bekommt sie diese von allen ihren Vorlagen. Die Einfachvererbung ist in der Software-Entwicklung der Regelfall, da es bei der Mehrfachverbung zu Konflikten und Mehrdeutigkeiten kommen kann. Aus diesen Gründen lassen einige Programmiersprachen auch nur die Einfachvererbung zu (etwa Borland Pascal oder Java[1]). Die neu erzeugte Klasse kann danach mit neuen Attributen und Methoden erweitert werden. Je nach Deklaration können Attribute und Methoden von der Vorlage benutzt oder überschrieben werden. Die Verwendung von Attributen der Vorlage in der Klasse verletzt das Prinzip der Kapselung, wonach Attribute nur innerhalb einer Klasse bekannt sein sollten, und nicht in der Vorlage *und* der abgeleiteten Klasse. Durch die Vererbung entsteht eine Klassenhierarchie, welche grafisch dargestellt werden kann. Eine Übersicht verschiedener Notationen findet sich in Balzert (1996).

Polymorphismus ermöglicht die Ausführung von Methoden gleichen Namens auf ähnlichen Klassen. Das ausführende Objekt braucht also nicht genau die Klasse des Objektes zu kennen, dessen Methode es aufruft. Es ruft eine Methode einer Klasse auf, welche, direkt oder indirekt, die Vorlage aller verwendeten Klassen ist. Die konkreten Instanzen führen dann die gleichnamige, evtl. überschriebene, Methode aus. Der Vorteil dieses Verfahren ist, daß man sich bei der Implementierung nicht auf eine feste Anzahl von Klassen festlegt, sondern auch externe Entwickler eine beliebige Anzahl von Klassen programmieren können, da nur eine Schnittstelle definiert wurde. Allerdings muß der Aufrufer wissen, daß die gewünschte Methode existiert. Um die möglichen Klassen nicht unötig einzuschränken und um möglichst wenig über die später ausgeführten Methoden zu wissen, werden abstrakte Klassen eingeführt. Eine abstrakte Klasse enthält eine oder mehr abstrakte Methoden, d.h. Methoden, welche nicht ausgeführt werden können. Sie werden erst bei den abgeleiteten Klassen realisiert. Daher können von abstrakten Klassen üblicherweise keine Objekte instanziert werden. Abstrakte Klassen bilden eine Schnittstelle zu einer Menge von abgeleiteten Klassen.

[1]Das Interface-Konzept von Java ist keine Mehrfachvererbung, erlaubt es aber, eine solche nachzubilden.

Durch den Einsatz des Polymorphismusses steht zum Übersetzungszeitpunkt noch nicht genau fest, welche Methoden aufgerufen werden sollen. So sind die Methoden einer Klasse, die von der im Quelltext stehenden Klasse abgeleitet wurde, auch mögliche Kanditaten. Erst zur Laufzeit ist also bekannt, von welcher Klasse die Methoden benötigt werden. Dieses Verfahren, wo die Entscheidung, welche Methode aufgerufen wird, erst zur Laufzeit getroffen wird, wird spätes oder dynamisches Binden genannt. Eine Methode, welche das späte Binden zulässt, wird virtuelle Methode genannt.

2.2 Entwurfsmuster

Entwurfsmuster sind nach Gamma u. a. (1996) „Beschreibungen zusammenarbeitender Objekte und Klassen, die maßgeschneidert sind, um ein allgemeines Entwurfsproblem in einem bestimmten Kontext zu lösen.“ In einer Beschreibung wird zuerst das zu lösende Problem erläutert, gefolgt von den möglichen Lösungen. Abgerundet wird die Beschreibung durch eine Auflistung der Konsequenzen bei der Nutzung des Entwurfsmusters.

Entwurfsmuster wurden zuerst in der Architektur und in der Stadtplanung entwickelt: Eine Sammlung von Lösungsvorschlägen für häufig auftauchende Probleme, etwa den Aufbau eines Badezimmers. Gamma u. a. gehen noch weiter zurück, indem sie Entwurfsmuster auch in der Literatur ansiedeln, etwa das Muster „tragisch gefallener Held“ oder „Liebesroman“.[2]

Ivanov (1996) klassifiziert die Entwurfsmuster auf zwei Arten, einerseits nach der Art der benutzten Entitäten, andererseits nach dem beabsichtigen Zweck. Im ersten Fall kann man unterscheiden, ob sich das Entwurfsmuster auf Klassen oder Objekte bezieht. Bei Klassenentwurfsmustern sind die Beziehungen zwischen den Ober- und Unterklassen statisch, d.h. sie stehen zum Übersetzungszeitpunkt fest. Im zweiten Fall, den Objektentwurfsmustern, entstehen die Beziehungen zwischen den einzelnen Instanzen erst zur Laufzeit, sind also dynamisch.

Bei der Einteilung nach dem Zweck von Entwurfsmustern sind die folgenden drei Gruppen möglich (nach Ivanov (1996)):

- *Erzeugende Muster:* Vom konkreten Erzeugungsprozeß eines Objektes wird abstrahiert. Dies ermöglich mehr Flexibilität, weil gleichwertige Objekte eingesetzt werden können.

[2]Die Literatur hat sich mittlerweile von zu strikten Normen gelöst. Dramen werden heutzutage kaum noch in der klassischen Aufteilung in fünf Akten geschrieben, auch auf die Beschränkung auf „Charactere von Stand“, wie sie noch vor dem Sturm und Drang galt, ist mittlerweile gefallen.

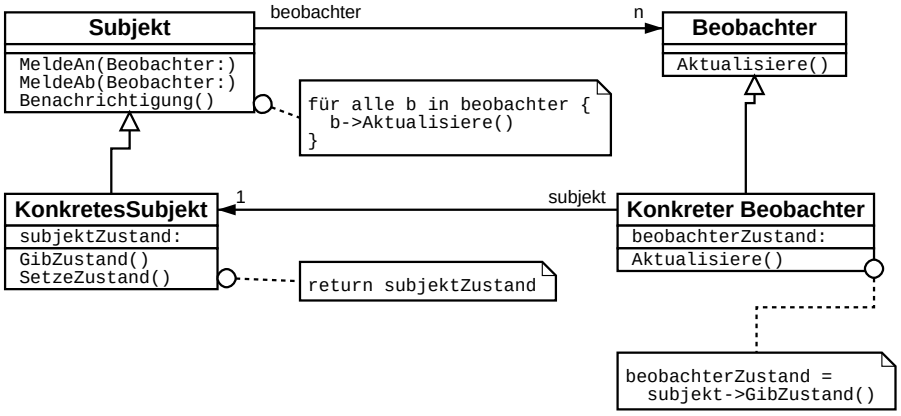

Abbildung 2.1: Entwurfsmuster Beobachter (nach Gamma u. a. (1996))

- *Strukturelle Muster:* Sie ermöglichen das Zusammenfügen von Klassen, Objekten oder Datentypen zu größeren Strukturen, welche wiederum die Basis für noch größere Strukturen sein können.

- *Verhaltensbezogene Muster:* Sie beschreiben die Kontrollflüsse zwischen einzelnen Objekten/Klassen bei der Abarbeitung von Algorithmen.

Als Beispiel für ein verhaltensbezogenes Muster soll hier das Muster „Beobachter“ kurz vorgestellt werden. Eine ausführliche Beschreibung findet sich in Gamma u. a. (1996).

Das Beobachtermuster definiert eine 1-zu-n Abhängigkeit zwischen Objekten, so daß die Änderungen des Zustandes eines Objektes dazu führt, daß alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden. Auf diese Weise ist etwa eine Trennung der Darstellung von den Datenobjekten möglich. Die Beobachter können Benutzungsschnittstellen für die Daten sein. Eine Änderung der dargestellten Daten durch den Benutzer im Beobachter führt nur zu einer Änderung der Daten im Subjekt. Diese Änderung wird dann vom Subjekt allen eingetragenen Beobachtern, mitgeteilt. Dies ermöglicht mehrere Benutzungsschnittstellen für ein Subjekt, ohne daß diese Schnittstellen von einander wissen. Ein weiterer Effekt dieser Entkoppelung von Subjekt und Beobachter ist, daß beide unabhängig von einander entwickelt werden können. Die Struktur dieses Musters ist in Abbildung 2.1 dargestellt.

Der Einsatz von Entwurfsmustern hat eine Reihe von Vorteilen:

- Das Nutzen von bewährten Lösungen für übliche Probleme vermindert das Risiko von Entwurfsfehlern. Es bleibt aber das Risiko, ein nicht

angemessenes Entwurfsmuster zu wählen oder eines falsch zu implementieren.

- Mit einem Entwurfsmuster wird das Zusammenspiel mehrerer Objekte oder Klassen standardisiert beschrieben und dokumentiert. Damit ist zwischen den Entwicklern eine Verständigung über den Entwurf auf einer höheren Abstraktionsebene möglich.

- Entwurfsmuster sind nicht direkt an eine spezielle objektorientierte Sprache gebunden. Eventuell lassen sich einige Entwurfsmuster in bestimmten Sprachen leichter realisieren. Dies sollte dann, falls bekannt, im Konsequenzenabschnitt erwähnt werden. Die Programmiersprachenunabhängigkeit ermöglicht eine Verständigung von Entwicklern über „Sprach“-Barrieren hinweg.

Eine Einordnung der Entwurfsmuster nach ihrer Komplexität, meistens nach der Anzahl der verwendeten Klassen aber auch nach Quelltextzeilen, ist ebenfalls möglich. Es wird üblicherweise davon ausgegangen, daß ein Entwurfsmuster zumindest mehr als eine Klasse beinhaltet. Ansonsten kann eine Einordnung je nach gewünschter Abstraktionsebene vorgenommen werden, wobei ab einer gewissen Komplexität besser Frameworks verwendet werden sollten.

2.3 Frameworks

Ein Framework ist eine abstrakte Lösung für eine bestimmte Art von Problemen (etwa Benutzungsoberflächen oder Buchhaltung), welches für den Einsatz noch konkretisiert werden muß (Mattsson 1999; Johnson 1992). Es enthält das Grundgerüst für eine Problemlösung in einer Anwendungsdomäne. Dieses Grundgerüst muß vom Anwendungsprogrammierer konkretisiert werden, um aus dem Framework ein einsatzfähiges Anwendungsprogramm zu erzeugen. Dazu besteht das Framework aus einem festen Anteil, welcher nicht verändert wird und sogenannten Variationspunkten oder *hot-spots*. Der konstante Anteil, welcher üblicherweise die Verarbeitungslogik der Anwendung enthält, wird im Kontrast dazu auch als *frozen-spot* bezeichnet. Für das Konkretisieren eines Frameworks kann der Entwickler des Frameworks prinzipiell zwei Möglichkeiten vorgesehen haben: Einerseits ist die Auswahl einer schon vorgesehenen Verhaltensweise, also das reine Parametrisieren, eine Variante. Andererseits ist die Erweiterung des Frameworks um die gewünschte Funktionalität durch den Anwender ebenfalls möglich. Der erste Fall dürfte in stabilen Anwendungsdomänen häufiger vorkommen, etwa

die Auswahl einer geeigneten „Mahn-Strategie“ bezüglich unbezahlter Rechnungen bei Firmen. Das zweite Extrem dürfte in Bereichen vorkommen, in denen die Anforderungen der einzelnen Nutzer zu verschiedenartig sind, so z.B. unterschiedliche Tarif- und Rabattsysteme.

Im Gegensatz zu klassischen Bibliotheken wird bei Frameworks der Kontrollfluß umgekehrt („Hollywoodprinzip[3]“). Der Anwendungsprogrammierer ruft nicht die Bibliotheksfunktionen aus seinem Programm auf, um es nach der Ausführung fortzuführen, sondern es ist umgekehrt. Das Framework ruft die Programmfragmente des Anwendungsprogrammierers auf, um nach deren Ausführung wieder die Kontrolle zu übernehmen. Hierbei handelt es sich um ein Konzept, welches auch bei der ereignisgesteuerten Programmierung eingesetzt wird.

Die meisten Frameworks sind in objektorientierten Sprachen geschrieben, da es die Prinzipien der Vererbung und des Polymorphismusses ermöglichen[4], außerhalb des Framework-Quelltextes die Funktionalität zu erweitern oder zu verändern. Es gibt aber auch nicht-objektorientierte Frameworks, welche dann mit Konventionen für die Entwickler das für Frameworks wichtige späte (oder dynamische) Binden emulieren. Das späte Binden wird bei objektorientierten Sprachen schon vom Laufzeitsystem realisiert, wobei der Compiler schon zur Übersetzungszeit automatisch die dafür benötigten Tabellen generiert, was die Möglichkeit von Programmierfehlern in diesem Zusammenhang stark verringert.

Eines der ersten Frameworks, das eine weite Verbreitung fand, war die Benutzungsschnittstelle von Smalltalk-80, Model-View-Controller (MVC) (Fayad 1999). Andere Frameworks für grafische Benutzungsschnittstellen folgten; zu den bekannteren zählen MacApp von Apple, die Microsoft Foundation Classes (MFC) von Microsoft oder das Abstract Window Toolkit (AWT) für Java von Sun. Mit Frameworks können aber nicht nur Benutzungsoberflächen erzeugt werden, sondern auch vollständige Anwendungen. Dieser Idee entspricht das „Genossenschaftliches Büro, Kommunikations- und Organisationsystem“-Framework (Gebos) der Rechenzentrale Würtembergischer Genossenschaften (RWG), mit dem für Banken Anwendungsprogramme für die verschiedenen Arbeitsumgebungen entwickelt werden können (Bäumer u. a. 1997). Etwas abstrakter ist wiederum das San-Francisco-Framework von IBM, womit allgemeine betriebliche Anwendungssysteme erzeugt werden können (Abinavam u. a. 1998).

Da ein objektorientiertes Framework ebenso wie eine Klassenbibliothek

[3] don’t call us – we call you

[4] Abstrakte Klassen oder Methoden sind hilfreich, aber nicht notwendig. Man kann die entsprechenden Methoden auch „leer“ implementieren, erhält dann bei der Instanziierung einer solchen Klasse aber keine entsprechenden Warnungen vom Compiler.

aus einer Menge von Klassen besteht, können die beiden verwechselt werden. Diese Konzepte unterscheiden sich wie folgt: Die Klassen einer Klassenbibliothek können einzeln verwendet werden, bei einem Framework hingegen haben sie neben der Vererbung Abhängigkeiten untereinander, welche eine Nutzung nur im Zusammenspiel mit anderen Frameworkklassen möglich macht. Die Übergänge sind aber fließend. Johnson (1997) schlägt, leicht ironisch, zur Unterscheidung vor: „Je mehr sich die Programmierer über die Komplexität der Klassenbibliothek beschweren, desto wahrscheinlicher handelt es sich um ein Framework“.

2.3.1 Ziele

Bei der Entwicklung und Verwendung von Frameworks gelten auch die allgemeinen Ziele der Softwareentwicklung. Die Programme sollen fehlerfrei, portabel, effizient und kostengünstig sein. Diese Ziele sind aufgrund der steigenden Anzahl von Hard- und Softwarearchitekturen und den Unterschieden bei den Betriebssystemen schwer zu erreichen. Im Gegensatz zur Nutzung von Funktionen aus Bibliotheken sollen bei Frameworks der Quelltext und das Design wiederverwendet werden. Durch die schon für das Design erfolgte Analyse und die evtl. vorhandenen Standardimplementierungen der abstrakten Klassen, soll der Aufwand zur Entwicklung einer Anwendung basierend auf dem Framework geringer sein als eine Neuentwicklung von Grund auf (Mattsson 1996). Die Menge des neu zu schreibenden Quelltextes soll möglichst gering sein. Das Ziel ist die Erstellung einer Anwendung durch „direkte Manipulation“ (Johnson 1993).

Der Unterschied zu Applikationsgeneratoren besteht darin, daß Applikationsgeneratoren jeweils ihre eigene formale Sprache zur Beschreibung der jeweiligen Anwendungsdomäne haben und daraus automatisch den Quelltext für eine Applikation erzeugen. Bei Frameworks hingegen wird die Anwendungsdomäne direkt programmiersprachlich beschrieben.

2.3.2 Wiederverwendung

Frameworks sollen die Wiederverwendung erleichtern, da die mehrfache Nutzung bei der Softwareentwicklung nach Jaekel (1999) die folgenden Vorteile hat:

- *Höhere Zuverlässigkeit:* Durch den Einsatz der wiederverwendeten Softwarebausteine unter realen Bedingungen könnten schon Fehler erkannt und beseitigt worden sein.

- *Vergrößerung der Planungssicherheit:* Durch die Veringerung der Komplexität im wiederverwendeten Bereich sind die Kosten besser abzuschätzen.

- *Bessere Ausnutzung von Spezialisten:* Spezialisten implementieren einmal eine Lösung und stehen danach wieder für andere Aufgaben zur Verfügung, anstatt dasselbe Problem immer wieder zu lösen.

- *Leichtere Einhaltung von Standards:* Die Standards werden als Softwarebausteine implementiert und in verschiedenen Softwareprojekten wiederverwendet.

- *Verkürzung der Entwicklungszeit:* Wiederverwendung führt zu kürzeren Entwicklungszeiten, womit ein Produkt schneller fertig wird (*time-to-market*).

Daneben gibt es noch die allgemeinen Probleme der Wiederverwendung: Es ist unbekannt, ob die gewünschte Software schon entwickelt worden ist und wo sie gegebenenfalls angeboten wird. Vor der Entscheidung „Wiederverwendung oder Neuentwicklung“ sollte daher eine Abschätzung gemacht werden, wie wahrscheinlich es ist, daß die gewünschte Software schon existiert, da – während der Suche und Evaluierung – schon Neuentwickelt werden könnte.

Die Wiederverwendung läßt sich nach zwei Kriterien einteilen. Zum einen nach der Größe der wiederverwendeten Teile. Lajoie und Keller (1994) schlagen hierfür eine dreistufige Einteilung in klein, mittel und groß vor. Klein steht für eine Wiederverwendung von Klassen, Methoden oder Quelltext-Fragmenten. Die nächste größere Einteilung besteht dann aus mehreren Klassen und deren Verhalten untereinander. Das enge Zusammenspiel dieser Klassen nennen sie Mikroarchitekturen (*micro-architectures*). Die Mikroarchitekturen können Entwurfsmuster sein, müssen es aber nicht. In dieselbe Einteilung fällt dann auch die Wiederverwendung mittels Frameworks, da ein Framework aus dem Zusammenspiel mehrerer Mikroarchitekturen besteht. Werden ganze Anwendungen oder ähnlich unabhängige Systeme unverändert wiederverwendet, so ist dies eine Wiederverwendung im Großen (s. z.B. Schwonbeck und Roos (2000)).

Neben der Größe der wiederverwendeteten Softwareprodukte kann nach nach dem „Einblick“ klassifiziert werden, den der Wiederverwender hat:

- Bei der *White-Box*-Wiederverwendung stehen dem Entwickler die Quelltexte zur Verfügung. Ist keine Dokumentation beigefügt, erzeugt sich der Entwickler alleine aus diesen sein eigenes Bild, wie die Schnittstellen aufgebaut und zu verwenden sind.

Eine Gefahr bei der White-Box-Wiederverwendung ist, daß auf die Dokumenation weniger Sorgfalt verwendet wird, da alle Details aus den Quelltexten hervorgehen.

Auch wird der Anwendungsentwickler in einer grundlegenden Entwurfsentscheidung, der Wahl der zu verwendenen Programmiersprache, eingeschränkt. Auch eine evtl. Portierung zur Wiederverwendungs des Entwurfs ist nicht immer möglich, da es Unterschiede in den Fähigkeiten der einzelnen Sprachen gibt, so hat z.B. Java keine Templates und C++ keine Garbage Collection.

- Anschaulich bekommt der Entwickler bei der *Black-Box*-Wiederverwendung etwas, in das er nicht hineinschauen kann, um zu verstehen, wie es funktioniert. Verwenden kann er es nur über die dokumentierten Schnittstellen. Ihm bleiben also die Details der Implementierung verborgen. Es ist ihm daher nicht möglich, Annahmen über ihren inneren Aufbau oder ihre Implementierung in sein eigenes Produkt einfließen zu lassen. So wird das Problem verhindert, daß diese Annahmen bei einer neuen Version des wiederverwendeten Softwareproduktes nicht mehr gültig sein müssen, ohne daß sich an der Schnittstelle etwas geändert hat.

  Bei einer Black-Box-Wiederverwendung ist die Dokumentation sehr wichtig, da fehlende Details von engagierten Programmierern nicht aus den Quelltexten entnommen werden können.

Zwischen den beiden Extremen, offene Quellen oder keine Quellen gibt es fließende Übergänge. Bei der sogenannten *Grey-Box-* oder *Glass-Box-*Wiederverwendung werden bestimmte Teile der Quelltexte offengelegt. So können einige mitgelieferte Werkzeuge bei einem Softwarepaket mit den Quellen geliefert werden, da sie als Beispiel für eine Nutzung der Schnittstelles des Hauptprogramms dienen.

Wooridge (2000) unterscheidet nach der Art der Wiederverwendung. Da gibt es neben der opportunistischen auch eine systematische Wiederverwendung. Im opportunistischen Fall findet eine Wiederverwendung nur statt, weil, mehr oder minder, zufällig die passenden Teile gerade vorhanden sind. Dabei handelt es sich meistens um eine White-Box-Wiederverwendung. Diese Art ist beim systematischen Fall seltener der Fall, da hier gezielt der Entwurf, die Entwicklung sowie die eigentliche Wiederverwendung geplant wird.

2.3.3 Vor- und Nachteile

Neben den oben genannten allgemeinen Vorteilen einer Wiederverwendung ergeben sich nach Fayad (1999) für Frameworks weiter die folgenden Vorteile: Modularität, Erweiterbarkeit und Umkehrung des Kontrollflusses. Als entscheidener Vorteil gilt die Wiederverwendung des Designs, da man dadurch im Idealfall keine Analyse der Anwendungsdomäne durchführen muß. Dieses Konzept sollte den Grad der Wiederverwendung in der Softwareentwicklung erhöhen (Mattsson 1996).

Leider gibt es neben den Vorteilen auch Nachteile. Diese betreffen die Entwickler der Frameworks, deren Nutzer oder gar beide Seiten.

Nachteile für die Frameworkentwickler

Frameworkentwickler müssen mit einem höheren Entwicklungsaufwand rechnen, da sie nicht nur eine Anwendung, sondern mehrere mögliche Anwendungen berücksichtigen müssen. Dafür muß eine wesentlich gründlichere Analyse erstellt werden. Da die Qualität des Entwurfes aufgrund der geplanten Wiederverwendung entscheidend ist, sollte ein Framework auch nur von Experten der jeweiligen Domäne geschrieben werden. Als Experte nach Mattsson (1999) hat sich qualifiziert, wer zumindest mehrere Anwendungen entwickelt hat, welche später auf Basis des Frameworks realisiert werden sollen.

Die Wartung des Frameworks ist aufwendiger, da mehrere Anwender mit unterschiedlichen Interessen gehört werden müssen. Eine Schnittstellenänderung scheidet aus, aber auch die Behebung eines Fehlers kann blockiert werden, da einige Nutzer schon Workarounds eingerichtet haben, welche danach nicht mehr funktionieren (Laitinen 1999). Eine Überprüfung des abstrakten Entwurfes und dessen Implementierung ist schwieriger als bei einer konkreten Anwendung. Für Johnson (1993) bedeutet das Testen eines Frameworks, das Entwickeln einer darauf basierenden Anwendung.

Nachteile für die Frameworknutzer

Aber auch die Nutzer von Frameworks sollten mit Nachteilen rechnen, etwa bedarf es eines gewissen Einarbeitungsaufwandes, um ein Framework verwenden zu können, oder um zu erkennen, daß es für die eigenen Ziele nicht geeignet ist.

Auch ist es nicht immer möglich, ein Framework mit einem anderem Framework zu einer Anwendung zu integrieren (Garlan u. a. 1995). Das Hauptproblem ist die Umkehrung des Kontrollflusses, die dazu führt, das evtl. beide Frameworks glauben, die gesamte Kontrolle über die Anwendung zu haben.

Die Umkehrung des Kontrollflusses führt auch zu einer Erschwerung der Fehlersuche bei der Anwendungsentwicklung, weil übliche Debugger nicht dafür ausgelegt worden sind. Um diesen Mehraufwand zu verringern schlägt Laitinen (1999) vor, neben den anwendungsspezifischen Anpassungsmöglichkeiten auch noch die Möglichkeit von Ausgaben zur Fehlersuche in ein Framework einzubauen.

Eine gute Erweiterbarkeit geht meist zu Lasten den Effizienz, da Zwischenschichten und Schnittstellen eingebaut worden sind. Allerdings wird bei der Entwicklung eines Frameworks im allgemeinen mehr Wert auf Effizienz gelegt, so daß sich diese beiden Effekte ausgleichen können.

Auch kann es sein, daß trotz Frameworks mehr Quelltext von den Anwendungsentwicklern geschrieben werden muß. Dieser Fall tritt z.B. dann ein, wenn schon vorhandene Anwendungen oder Bibliotheken (*legacy-code*) eingebunden werden sollen, weil dann zusätzliche Klassen geschrieben werden müssen, um die vorhandenen Produkte mittels sogenannter Wrapper-Klassen in die Klassenhierarchie des Frameworks einzubinden. Verschiedenen Lösungsvorschläge werden in Lundberg und Mattsson (1996) vorgestellt. Diese Wrapper-Klassen stoßen an ihre Grenzen, wenn es darum geht, mehrere Frameworks miteinander zu verbinden. Dann kann man zwar das Verhalten so anpassen, daß die Frameworks integriert werden können. Aber unter Umständen können interne Nachrichten des einen Frameworks nicht an das andere Framework weitergeleitet werden, da entsprechende Erweiterungsmöglichkeiten nicht vorgesehen waren (Garlan u. a. 1995)[5].

Auch gibt es noch keine Methoden zur Aufwandabschätzung für eine Entwicklung einer Anwendung auf Basis von Frameworks. Die konventionellen Verfahren berücksichtigen weder die Wiederverwendung noch den Lernaufwand.

Allgemeine Nachteile

Neben den Problemen mit denen jeweils die Entwickler oder die Nutzer zu kämpfen haben, gibt es noch Probleme, welche beide betreffen. Es gibt noch keine Standards für den Entwurf, die Implementierung, die Dokumentation oder die Anpassung von Frameworks. Hier bleibt abzuwarten, ob sich in diesem Bereich ein Standard entwickelt und durchsetzen kann.

2.3.4 Dokumentation

Für den Anwender eines Frameworks ist eine gute Dokumentation wichtig. Da ein Framework als eine komplexe Klassenbibliothek angesehen werden

[5] zitiert Holzle aus dem OOPSLAA 93

kann, sollte es mindestens ebensogut wie eine Klassenbibliothek dokumentiert werden. Diese Dokumentation reicht aber nicht aus, da zum Verständnis des Frameworks auch Wissen über die jeweilige Anwendungsdomäne erforderlich ist. Die Nutzer eines Frameworks benötigen also nicht nur Wissen aus der Softwaretechnik, sondern auch über die Anwendungsdomäne. Da dieses Wissen am Anfang eines Softwareprojektes meistens noch nicht gefestigt ist, könnte es aufgrund der Möglichkeit, gleiche Sachverhalte der Anwendungsdomäne verschieden darzustellen, zu Problemen kommen.

Nach Mattsson (1996) gibt es für die Frameworkdokumentation mindestens drei Zielgruppen:

- Entwickler, welche auf der Suche nach dem passenden Framework sind, brauchen eine kurze Beschreibung der Fähigkeiten, welche möglichst am Anfang der Dokumentation stehen sollte.
- Entwickler, welche das Framework einsetzen, benötigen Informationen über den vom Frameworkentwickler beabsichtigten Gebrauch des Frameworks.
- Entwickler, welche in die Tiefen des Framework hinabsteigen möchten, sollten eine Beschreibung der abstrakten Algorithmen und der einzelnen Klassen erhalten.

Ansätze

Je nach Zielgruppe der Dokumentation kann ein unterschiedlicher Ansatz verwendet werden. Einige der bekannteren Ansätze sollen hier kurz vorgestellt werden:

- *Kochbücher* und *Tutorials* erläutern nicht in erster Linie den Aufbau und die Struktur eines Frameworks, sondern wie es genutzt wird. Während Kochbücher die Nutzung anhand von Beispielen verdeutlichen, erklären Tutorials neben der schrittweisen Abarbeitung eines Beispiels die gerade benötigten Konzepte.
- *Entwurfsmuster* (s. Abschnitt 2.2) helfen beim Finden von Mikroarchitekturen in der Vielzahl von Frameworkklassen. In einem Entwurfsmuster wird das Zusammenspiel mehrer Klassen standardisiert beschrieben. Dies hat den Vorteil, das die Frameworkstruktur auf einer höheren Abstraktionsebene betrachtet werden können.
- *motifs* (Johnson 1992; Lajoie und Keller 1994) sind eine Art „Dokumentationsmuster“. Sie beschreiben zuerst das Problem, welches der

Nutzer lösen möchte, wägen danach eine Reihe von Lösungmöglichkeiten ab, um dann die Schritte zu einer Lösung zusammenzufassen. Ein Erweiterung von motifs sind *hooks* (Froehlich u. a. 1999). Sie verzichten auf die Abwägung, erwähnen dafür, ähnlich wie bei der Dokumentation zu einer Klassenbibliothek, die Seiteneffekte und die Vorbedingungen. Auch klassifizieren sie die Art der Änderung, ob es sich um das De-/Aktivieren, das Ersetzen, das Verbessern oder das Hinzufügen einer Funktionalität handelt.

- *Framework Definition Languages (FDL)* sind Sprachen, welche die in den obigen Ansätzen in natürlicher Sprache gehaltene Dokumentation formal beschreiben können. Mit diesem Ansatz könnte die Korrektheit eines Entwurfes automatisch überprüft werden. Die FDL von Wilson und Wilson wird in Mattsson (1996) kurz vorgestellt.

2.4 Komponenten

Der Begriff Komponente ist aufgrund seiner allgemeinen Grundbedeutung (ein Teil eines Ganzen) in der Informatik mehrfach belegt. Hier soll im folgenden mit einer Komponente die Softwarekomponente gemeint sein, welche nach Ritter (2000) an den folgenden Eigenschaften zu erkennen ist:

- Sie wird als Black-Box ausgeliefert.
- Die Nutzung ihrer Fähigkeiten erfolgt ausschließlich über die definierten Schnittstellen.

Aufgrund der oben genannten Eigenschaften haben Komponenten weitere (vorteilhafte) Erkennungsmerkmale:

- Komponenten für den gleichen Aufgabenbereich können von unabhängigen Softwareproduzenten entwickelt worden sein.
- Sie können unabhängig voneinander eingesetzt werden, d.h. sie wurden nicht speziell für einen Einsatz entwickelt, so daß auch andere neben dem ersten Kunden sie einsetzen können.
- Mehrere Komponenten können zusammen ein Anwendungsprogramm ergeben.

Griffel erweitert den Komponentenbegriff um die Granularität einer Komponente. Eine Komponente sollte einerseits klein genug sein, um in einem

Stück erzeugt und gepflegt werden zu können, andererseits sollte sie wiederum groß genug sein, um eine sinnvoll einsetzbare Funktionalität zu enthalten.

Auch wenn Komponenten üblicherweise kleiner als Frameworks sein werden, so sollte die Kohäsion größer sein. Mit der Kohäsion eines Systems bezeichnet Ritter (2000) den Grad der wechselseitigen Verbundenheit seiner Elemente. Diese sollte in einer Komponente hoch sein, aber zwischen Komponenten klein, um die Austauschbarkeit nicht zu gefährden. Dies ist der Unterschied zu Frameworks: Da ein Framework eine gesamte Anwendung abdeckt, wird es Teilbereiche geben, welche nur lose an den Rest gekoppelt wurden. Dies sollte man bei einer Komponente vermeiden, und stattdessen eine Trennung in zwei oder mehr Komponenten vornehmen.

Komponenten müssen ebenso wie Frameworks nicht in einer objektorientierten Sprache entwickelt worden sein. Die verwendete Programmiersprache sollte nach Möglichkeit keinen Einfluß auf die Komponentenschnittstelle haben. Aus diesem Grunde scheiden direkte Schnittstellen aus. Bei ihnen erfolgen die Aufrufe direkt auf der Ebene der Programmiersprache, etwa durch Methodenaufrufe bei objektorientierten Sprachen. Statt dessen erfolgt der Aufruf anhand von Nachrichten, welche sich die Komponenten über Kommunikationskanäle schicken. Intern werden diese Nachrichten nach dem Empfang in die entsprechenden Aufrufe in der Programmiersprache umgesetzt.

Eine Kombination von Komponenten und Frameworks sind komponentenbasierte Frameworks. Die Konkretisierung erfolgt dort durch die Auswahl der verwendeten Komponenten und nicht durch die Erweiterung der Klassenhierarchie.

Beispiele

Beispiele für Komponenten sind etwa die Systemprogramme unter Unix, um ein paar Beispiele aus der reichhaltigen Auswahl zu nennen, etwa `uniq`, `test` oder `sort`. Diese Programme kommunizieren über eine Pipe, eine der einfachsten möglichen Schnittstellen. Bei einer Pipe wird die Ausgabe des einen Programmes die Eingabe des nächsten Programmes. Trotz dieser Einfachheit können daraus mächtige Programme, Skripte genannt, erzeugt werden, welcher wiederum Komponenten eines weiteren Skriptes sein können.

Plug-Ins für die Internet-Browser (etwa den Microsoft Internet Explorer oder den Netscape Navigator) stellen auch Komponenten dar. Hier erzeugen sie zwar in der Summe kein Anwendungssystem, sondern erweitern es nur, aber man kann, trotz seiner Größe, auch den Internet-Browser als Komponente ansehen.

ActiveX ist ein Komponentenmodell, welches nach Griffel (1998) aus der normalen Produktpflege von Microsoft-Produkten (Windows und Office) her-

vorgegangen ist. Leider hat dieses Modell noch keine strikte Trennung zwischen Komponenten und Betriebssystem.

JavaBeans von Sun sind visuell manipulierbare Komponenten, welche in Java geschrieben sind. Um als Komponenten erkannt zu werden, muß sich der Entwickler an bestimmte Konventionen halten.[6] Während JavaBeans sich eher für die Entwicklung von grafischen Benutzungoberflächen eignen, sind Enterprise JavaBeans ein Komponentenmodell für verteilte Anwendungen.

2.4.1 Schnittstellen und Dokumentation

Da Komponenten üblicherweise eine Black-Box-Wiederverwendung darstellen,[7] sind die Schnittstellen um die Funktionalität der Komponenten nutzen zu können, von entscheidener Bedeutung. Üblicherweise wird eine Schnittstelle nach der Analyse des Anwendungsgebietes und der Komponentenzielgruppe entworfen. Dabei findet eine Abwägung zwischen Einfachheit und Mächtigkeit statt. Nach Johnson (1997) bevorzugen Entwickler im allgemeinen die Mächtigkeit, also im Falle einer Komponente eine möglichst große Konfigurierbarkeit, um eine große Anzahl von Einsatzgebieten abdecken zu können. Die Nutzer hingegen, bei denen es sich in den meisten Fällen nicht um Experten, sondern um Gelegenheitsnutzer handelt, möchten eine einfache Schnittstelle, also eher überschaubare Konfigurationsmöglichkeiten. Diese Abwägung ist mit Sorgfalt zu treffen. Ist die Komponentenschnittstelle zu einfach, wird sie unter Umständen nicht häufig genug wiederverwendet, ist sie hingegen zu mächtig, ist der Lernaufwand für die Nutzer so groß, das sich die Komponente durch die Zusatzkosten für die nötige Schulungen ebenfalls nicht rentiert.

Bei Komponenten werden die Schnittstellenbeschreibungen auch Verträge genannt, um den bindenen Charakter zu verdeutlichen. Die Stärke der Bindung hängt von der Sprache des Vertragstextes ab. Allgemein gehaltene Verträge in der natürlichen Sprache haben weniger Wert als ein detaillierter Vertrag, welcher eine formale Sprache als Grundlage hat. Büchi und Weck (1997) führen zur Verdeutlichung den Unterschied zwischen einem Geschäftsvertrag und einem Hochzeitsvertrag an.

Verträge können mehr als die reinen Schnittstellen der Komponenten enthalten, etwa auch nicht-funktionale Aspekte, wie Perfomanz oder Ressourcenverbrauch (Ritter 2000).

Der bei funktionalen Bibliotheken ausreichende Ansatz von Vor- und Nachbedingungen (*pre-* und *post-conditions*) reicht bei Komponenten auf-

[6] `www.javasoft.com/beans/spec.html`

[7] Open-Source-Projekte können Komponenten naturgemäß nicht ohne Quelltext ausliefern

grund der möglichen asynchronen Kommunikation nicht mehr aus. So kann es etwa von entscheidener Bedeutung sein, ob garantiert werden kann, das Nachrichten immer in der angegebenen Reihenfolge eintreffen. Wird dieses Verhalten nicht spezifiziert, so kann es sein, daß andere Komponentenentwickler es als (unbewußte) Annahme über das Verhalten mit in ihre Komponenten aufnehmen. Wird dieses Verhalten geändert, so hat sich nicht die Schnittstelle verändert, trotzdem funktioniert die Zusammenarbeit nicht mehr. So plädieren Büchi und Weck (1997) für eine teilweise Aufhellung der Black-Box, so daß nicht nur der Zusammenhang zwischen Eingabe und Ausgabe dokumentiert wird, sondern auch interne Zusammenhänge, soweit sie benötigt werden.

Bei der Dokumentation der Schnittstelle gibt es im Gegensatz zu Frameworks nur zwei Zielgruppen: Die Nutzer auf der Suche nach der passenden Komponente und die Nutzer, welche die Komponente einsetzen wollen.

Die bei den Frameworks vorgestellten Ansätze können im Prinzip so für Komponenten übernommen werden. Änderungen sind nur in der Hinsicht notwendig, daß Kochbücher oder Tutorials nicht die inneren Konzepte der Komponente erläutern. Diese sollen ja gerade verborgen bleiben.[8]

2.4.2 Komponenten gleich Software-ICs?

Im Vergleich zur Softwareentwicklung ist der Grad der Wiederverwendung bei der Herstellung von integrierten Schaltkreisen höher. Griffel (1998) erklärt diesen Umstand daraus, daß die Hardwareindustrie mehr Zeit zur Ausbildung von Standards zur Verfügung hatte. Auch wenn der Zeitraum von der Entwicklung des ersten IC fast genauso lang ist, wie die Geschichte der Softwareentwicklung, so hat sich die Industrie nach einer Phase von Einzellösungen doch zur Standardisierung bereitgefunden. Dies wurde dadurch begünstig, daß zum anderen die Anzahl der Parameter (etwa Spannung oder Widerstand) ebenso wie die Anzahl der Grundfunktionen geringer ist als bei der Softwaretechnik.

Griffel sieht auch eine unterschiedliche Entwicklungphilosophie: Hardware-Ingenieure versuchen grundsätzlich ein Problem mit bereits bestehenden (Hardware-) Komponenten zu lösen, während Software-Ingenieure eher „mal eben“ etwas neu dazuentwickeln. Dieser Umstand könnte durch die unterschiedlichen langen Entwicklungszyklen entstanden sein. Bei ICs muß man evtl. Wochen auf eine neue Version warten, während ein Übersetzungvorgang einige Minuten dauert. Ob diese Vorgehensweise mit der zunehmenden Verbreitung von Hardwareprogrammiersprachen, etwa VHDL, und der

[8] Außer für die Weiterentwickler an Open-Source-Projekten, welche diese Interna dann in einer gesonderten Dokumentation finden sollten.

Software-Emulation von ICs bestehen bleibt, ist abzuwarten.

Die größere Verbreitung von Software-ICs, wie Komponenten auch genannt werden (Griffel 1998; Johnson 1997), bleibt das Ziel. Ähnlichkeiten sind vorhanden. So haben beide Ansätze einen Verdrahtungsstandard, bei ICs das Rastermaß für die Platine, bei Komponenten Distributed COM (DCOM) oder Common Object Request Broker Architecture (CORBA). Bei beiden, ICs und Komponenten, läßt die Schnittstellenbeschreibung wenig Rückschlüsse auf den inneren Aufbau zu.

Da sich die Komplexität von Hardwarekomponenten auf einer anderen Abstraktionsebene befindet, spielen Sicherheitsaspekte eine geringere Rolle. Zugriffe auf andere Komponente sind nur durch Leitungen möglich, damit hat man schon eine sehr feingranulare Rechteverteilung erreicht. Dadurch wird eine mißbräuchliche Nutzung erschwert, aber nicht unmöglich gemacht. So gibt es geheime Möglichkeiten, die Codeabfrage eines Anrufbeantworters zu umgehen (sogenannte „Servicecodes“ der Werkstätten).

Trotz aller vorhandenen Schwierigkeiten besteht die Hoffnung, daß irgendwann einmal Software-ICs kein Ziel mehr sind, sondern eine einsetzbare Technik. Die Entwicklung geht zumindet in diese Richtung.

2.4.3 Komponentenmarkt

Durch die Eigenschaft von Komponenten, unabhängig voneinander entwickelt werden zu können, entsteht die Möglichkeit für kleinere Softwarefirmen, Zusatzprodukte oder Erweiterungen für größere Softwaresysteme anzubieten. Auch häufig benötigte Teile eines Anwendungssystems können von verschiedenen Herstellern angeboten werden.

Ein Problem mit dem Verkauf von Komponenten sind die Eigenschaften von Softwareprodukten (Gorny 1999): Da Software ein immaterielles Produkt ist, altert und verschleißt sie nicht, aber sie veraltet. Daraus resultiert die folgende Konsequenz für den Komponentenentwickler: Ist eine Komponente verkauft, kann sie der Käufer ohne Mehrkosten immer wieder in gleicher Qualität in sein Endprodukt integrieren. Dies unterscheidet die Softwarekomponeten von üblichen Industriekomponenten, etwa einem Elektromotor oder einem IC, welcher nur einmal eingebaut werden kann. In dieser Hinsicht ähnelt eine Softwarekomponente eher einer Lizenz als ihrem Vorbild aus der Industrie. Eine mögliche Lösung dieses Problems wäre die Einführung von von Techniken, welche eine Bezahlung pro Benutzung erlauben. Diese Techniken werden nach Griffel (1998) maßgeblichen Einfluß auf den kommerziellen Erfolg des Komponentenparadigmas haben. Ein weitere Möglichkeit ist, daß durch das schnelle Veralten der Softwarekomponenten der Kunde doch häufig genug „neue“ Komponenten erwerben muß, so daß

sich die Entwicklungskosten für die Hersteller rechnen.

Distribution

Während der immaterielle Charakter der Softwareprodukte zu Schwierigkeiten bei der Bezahlung führt, so hat er doch entscheidene Vorteile beim Anbieten und der Auslieferung der Komponenten. Diese können direkt durch Computernetze zum Käufer geliefert werden. Nur bei einen Wunsch nach gedruckter Dokumentation müssen konventionelle Wege verwendet werden. Die elektronischen Vertriebswege erlauben dazu auch eine automatische Aktualisierung der Komponenten beim Kunden.

Die beiden derzeit aktuellen Komponentenarchitekuren, ActiveX[9] und JavaBeans[10] bieten unabhängigen Softwareherstellern eine Plattform zum Anbieten ihrer Produkte in der jeweiligen Architektur. Aber auch neutrale Anbieter helfen bei der Vermittlung von Komponenten. So kann man etwa als Hersteller bei ComponentSource[11] eine Liste der Kundenwünsche einsehen, um die Marktchancen einer evtl. Entwicklung abzuschätzen.

Bei den oben genannten Angeboten erfolgt die Beschreibung und die Suche mit Hilfe der natürlichen Sprache mit all ihren möglichen Mißverständnissen. Erste Ansätze zur Beschreibung von Komponenten mittels formaler Sprachen existieren bereits (Ritter 2000).

2.4.4 Sicherheitsaspekte

Der Erwerb einer Komponente auf dem freien Markt ist mit Risiko verbunden. Bei der Verwendung einer Komponenten sollte sichergestellt werden, daß der Entwickler keine böswilligen Absichten hegte oder sich eine Hintertür zum eigenen System einrichtet. Um dieses Risiko auszuschließen, sollte der Erwerb der Komponente über ein sicheres Kommunikationssystem erfolgen, welches nach Baldzer (1999) die folgenden sechs Eigenschaften erfüllen sollte:

1. *Authentizität*: (Schärfere Form der Autorisierung) Es wird sichergestellt, daß die Person auch diejenige ist, für die sie sich ausgibt, und nicht nur, ob sie auch berechtigt ist.

2. *Integrität*: Die übermittelten Nachrichten sind unverändert geblieben.

[9] ActiveX Gallery: `http://www.microsoft.com/msdn/thirdparty`

[10] Bean Directory: `http://industry.java.sun.com/solutions/0,2346,beans,00.html`

[11] `http://www.componentsource.com`

3. *Vertraulichkeit*: Ein Unbefugter kann die Nachrichten nicht entschlüsseln und damit nicht lesen.

4. *Anerkennung*: Der Sender einer Nachricht kann nicht nachträglich leugnen, diese jemals verschickt zu haben.

5. *Zugriffskontrolle*: Nur berechtigte Nutzer haben Zugriff auf das Kommunikationsmedium.

6. *Verfügbarkeit*: Es können immer Nachrichten ausgetauscht werden.

Auch wenn die Vertraulichkeit auf dem ersten Blick als nicht unbedingt notwendig erscheint, so sollte sie dennoch gewährleistet sein, um zu verhindern, daß Konkurrenten sich frühzeitig ein Bild aus den Aktivitäten auf dem Komponentenmarkt machen können.[12] Um das Erzeugen eines Gesamtbildes aus Puzzleteilen zu verhindern, haben Systeme mit höheren Sicherheitsanforderungen zusätzlich noch die Anforderung, das evtl. Lauscher nicht erfahren können, wer mit wem kommuniziert hat (so etwa der Informationsverbund Bonn-Berlin (IVBB), der die Datenübertragung zwischen den aufgeteilten Bundesministerien in Bonn und in Berlin seit dem Umzug im Jahre 1991 realisiert).

Zertifikate

Bekanntes Beispiel sind die Zertifikate für Microsoft ActiveX-Controls. Im Normalfall ist es dem Laien nicht möglich, zu erkennen, ob der zertifizierte Entwickler der Komponente wirklich vertrauenswürdig ist. So gibt es etwa das Programm „Exploder“[13], welches auf die konzeptionellen Sicherheitslücken bei ActiveX hinweisen wollte. Es startet einen Countdown von zehn Sekunden und beendet danach Microsoft Windows. Dieses Programm war eine Zeit lang auch zertifiziert (Schallenberg 1999) zu erhalten. Mittlerweile ist es auf Druck der Zertifizierungsstelle VeriSign[14] nicht mehr vom Autor zertifiziert zu erhalten. Es liegt aber eine Anleitung bei, wie es jeder, der im Besitz einer Kreditkartennummer[15] ist, zertifizieren kann.

[12] Ein Beispiel für die Problematik, das mehrerer Informationen, welche an sich noch nicht schützenswert sind, in ihrer Summe eine schützenwerte Information ergeben, wird in Stoll (1989) zitiert. Dort war es gelungen, aus den nicht geheimen Bestellisten einer Flugzeugfabrik in den Vereinigten Staaten die geheime Metalllegierung eines Militärflugzeuges zu erhalten.

[13] `http://www.halcyon.com/mclain/ActiveX/`

[14] `http://www.verisign.com`

[15] Diese muß nicht seine eigene sein, es existieren auf einschlägigen Seiten Generatoren für Kreditkartennummern.

Diese Problematik tritt in den Hintergrund, wenn man nur Komponenten von renommierten Herstellern nach eingehender Prüfung einsetzt. Mit dieser Vorgehensweise kann man sich jedoch um die Vorteile und Dynamik eines Marktes für Komponenten bringen, auf dem auch unbekannte Hersteller ihre Komponenten anbieten. Ein Großteil der Dynamik geht schon verloren, wenn man verlangt, daß zumindest die Zertifizierungsstelle Einblick in die Komponente haben soll, und so eine Überprüfung anhand der Quelltexte und den Entwurfsdokumenten vornehmen kann. Daraus ergeben sich interessante Fragen bezüglich der Haftung. Die meisten Programme werden heutzutage leider mit einem mehr oder minder kompletten Haftungsauschluss ausgeliefert („... is provided as is ...“). Würden die Zertifizierungsstellen dieses Vorgehen übernehmen, so hätte man als Kunde im Schadensfalle nichts gewonnen. Zusammenfassend läßt sich sagen, daß eine Zertifizierung die Vertrauensfrage nur verschiebt. War es vorher die Frage, ob man dem Programmierer vertraut, so lautet sie jetzt: „kann man der Zertifizierungsstelle trauen“.

Eine weitere Möglichkeit sicherzustellen, daß eine Komponente keine unerwünschten Aktionen durchführt, sich also an ihre Verträge hält, ist die Begrenzung der Rechte der Komponente auf die jeweils benötigten Systemressourcen. Denn selbst wenn der Entwickler der Komponente keine böswilligen Absichten hegte, so kann doch ein Programmierfehler zu einer Sicherheitslücke führen. Ein Beispiel für ein solches Konzept sind die verschiedenen Benutzer(-gruppen) auf Unixsystemen. Dort lassen sich neben dem Administrator, der uneingeschränkten Zugriff auf alle Systemressourcen hat, weitere Benutzer(-gruppen) mit weniger Rechten einrichten. Zentrale Dienste, etwa der Mailserver, benötigen für Teilaufgaben die Rechte des Administrators. Als diese Dienste noch monolithisch aufgebaut waren, reichte eine Sicherheitslücke, um den gesamten Server zu übernehmen. Mittlerweile ist man bei vielen Systemen zu einem modulareren Aufbau mit einer feingranulareren Rechteverteilung übergegangen. Jetzt erhalten nur die Teilsysteme die Rechte, welche sie auch im Betrieb benötigen.

Diese Möglichkeit hat ein normaler Nutzer nicht, da er keine Gruppen und keine weiteren Nutzer anlegen kann. Damit haben alle von ihm installierten Programme die gleichen Zugriffsrechte wie er selbst. Ein von ihm installiertes Spiel könnte also seine gesamten persöhnlichen Daten ausspähen, unter anderem z.B. seine Kreditkartennummer.

Sandkastenspiele

Um dieses Risiko zu vermeiden, wurde in Java das Konzept des sogenannten Sandkasten (*sandbox*) eingeführt. Programme, welche in der Sandkasten-Umgebung ablaufen, erhalten einen sehr stark eingeschränkten Zugriff auf

die Systemressourcen, so dürfen sie z.B. keine Dateien auf dem ausführenden Rechner lesen oder gar verändern. Der Sandkasten ist mit der Gefängnisszelle von Büchi und Weck (1997) vergleichbar. In den ersten Java-Versionen war dieses Konzept noch relativ starr umgesetzt: Benötigte das Java-Programm eine der Systemressourcen zur produktiven Arbeit, so mußte es aus seinem Sandkasten entlassen werden, und erhielt damit den gleichen Zugriff auf die Systemressourcen wie der Nutzer, der es startete. Ab Java Version 1.3 gibt es ein feingranulares System zur Rechtevergabe, welches diese „ganz oder gar nicht“-Problematik vermeidet.

Aber auch bei der Granularität der Rechtevergabe sollte eine Abwägung zwischen Einfachheit und Mächtigkeit stattfinden. Ist das System zu einfach, so muß man unter Umständen einem Programm zuviele Rechte geben, was ein Sicherheitsrisiko darstellen kann. Ein mächtiges feingranulares System erfordert hingegen einen höheren Lernaufwand um es richtig zu nutzen. Während der Lernphase sind Bedienfehler nicht ausgeschlossen, welche ebenfalls ein Sicherheitsrisiko darstellen können. Viele Sicherheitspannen bei Internetservern haben diese Ursache (ein aktuelles Beispiel findet sich in Bleich (2000)).

Um die Lernkurve bei der komplexen Rechtevergabe abzuflachen, werden z.B. bei Computer Supported Cooperative Work (CSCW)-Systemen unter anderem Regelmengen anstelle von Rechtematrixen verwendet (so etwa im POLITEAM-Projekt (Cremers u. a. 1998)).

2.4.5 Generative Programmierung

Bei der generativen Programmierung gibt es eine strikte Trennung zwischen dem Problem- und dem Lösungsraum. Ebenso wie bei den Applikationsgeneratoren wird aus einer Problemspezifikation eine Lösung erzeugt. Der Unterschied zu Applikationsgeneratoren besteht darin, daß es neben der domänenspezifischen Sprache (Domain Specific Language (DSL)) auch eine implementationsspezifische Sprache gibt (Implementation Components Connection Language (ICCL)). Die ICCL beschreibt die verschiedenen Verknüpfungsmöglichkeiten der verwendeten Bausteine auf der untersten Abstraktionsebene. Die generative Programmierung setzt eine Spezifikation nicht direkt in eine Programmiersprache um, sondern erst in die Zwischensprache ICCL[16] Außerdem muß der Entwickler mit der generativen Programmierung nicht unbedingt ein gesamtes Anwendungssystem erzeugen, sondern kann auch Komponenten erzeugen.

Die Erzeugung einer Komponenten läuft üblicherweise in der folgenden Reihenfolge ab:

[16]Diese Aufteilung in ein Front- und ein Backend ist üblich bei Compilern.

1. Zuerst wird eine, evtl. unvollständige, Spezifikation erstellt.

2. Die fehlenden Angaben in der Spezifikation werden vom Generator durch domänenabhängige Standardvorgaben ergänzt.

3. Nachdem die Spezifikation vollständig geworden ist, können noch Optimierungen vorgenommen werden.

4. Der Generator wählt die passenden Bausteine aus, und fügt sie zusammen. Dabei kann die Spezifikation verändert werden. Dieser Fall kann beispielsweise eintreten, wenn der verwendete Baustein eine bestimmte Eigenschaft hat, welche im Wahlbereich der ursprünglichen Spezifikation lag. Nach der Konkretisierung hat die Komponentenspezifikation die Eigenschaft des eingesetzten Bausteins.

5. Die fertige Komponente wird mit der endgültigen Anforderungs- und Implementationsspezifikation ausgeliefert.

Diese Vorgehen hat nach Eisenecker und Czarnecki (1999) den Vorteil, daß der Generator, die Bausteine, das Konfigurationswissen und die erzeugte Komponente nicht in derselben Sprache realisiert sein müssen.

2.4.6 Vor- und Nachteile

Komponenten können im Gegensatz zu Frameworks direkt an den Nutzer eines Anwendungssystems veräußert werden (etwa bei den Browser Plug-Ins), und sind somit nicht nur für Softwareentwickler interessant. Dazwischen sind eine Vielzahl von Zwischenstufen möglich, aber insgesamt lassen sich die folgenden vier verschiedenen Kategorien ausmachen:

1. Die Hersteller,

2. die Berater,

3. die Kunden einer Komponente und

4. die Kunden eines zusammengesetzten Anwendungssystems.

Deren Vor- und Nachteile durch den Einsatz von Komponenten sollen im folgenden erläutertert werden.

Hersteller

Einer der Hauptvorteile von Komponenten ist die Möglichkeit. daß mehrere Softwarehersteller unabhängig voneinander Komponenten entwickeln können. Dies wird ermöglicht durch eine weitere Zwischenschicht. Aufrufe von Funktionen aus einer Komponente rufen nicht direkt Funktionen einer anderen Komponente auf, sondern senden eine Nachricht über die vorher definierte Kommunikationsschnittstelle. Bei diesen Kommunikationsschnittstellen wird es sich aufgrund der Dauer eines Normierungsverfahrens seltener um einen internationalen Standard handeln, sondern eher um de-facto Standards einzelner Anbieter. Zwar gibt es mittlerweile de-facto Standards über die Kommunikationskanäle, etwa CORBA oder DCOM, welche über TCP/IP kommunizieren, aber noch nicht für die darüber ausgetauschten Inhalte. Dies führt zu mehr Konfigurationsaufwand, da alle Verträge noch erstellt werden müssen.

Leichte Unterschiede in der Schnittstelle kann man versuchen mit Hilfe weiterer Komponenten auszugleichen. Diese Komponenten, deren einzige Ausgabe meist die Anpassung an Schnittstellen darstellt, werden auch Glue (engl. für Kleber) genannt. Allerdings besteht auch bei Komponenten die Gefahr einer Monopolbildung. Sollte es einem Hersteller(-konsortium) gelingen, ihr Komponentenmodell als das, mehr oder minder, allein gültige zu etablieren, so entsteht trotz der prinzipiellen Offenheit des Systems die gleiche Abhängigkeit von der Komponentensammlung des Modells, wie von den Schnittstellen eines klassischen Systems (Griffel 1998).

Die Indirektion durch die Kommunikation über Nachrichten führt zur Sprachunabhängigkeit. Komponenten eines Anwendungssystem müssen also nicht in derselben Sprache geschrieben worden sein, was Vorteile bei Einbindung von schon vorhandenen Systemen bietet. Auch müssen nicht alle Komponenten weiterentwickelt werden, wenn eine neue Sprache eingeführt wird. Das Problem der Einbindung in eine Klassenhierarchie, wie es z.B. bei Frameworks besteht, kann es daher nicht geben. Allerding besteht die Abhängigkeit von den Kommunikationsprotokollen. TCP/IP ist zwar zur Zeit der de-facto Standard, aber es sollte nicht vergessen werden, daß noch Ende der achtziger Jahre erwartet wurde, daß TCP/IP in naher Zukunft durch Protokolle auf Basis des ISO/OSI-Basisreferenzmodells abgelöst wird (McGregor 1995).

Aufgrund der Black-Box-Eigenschaft von Komponenten können sie an andere Softwarehersteller ausgeliefert werden, ohne eigenes Wissen über die Anwendungsdomäne preiszugeben. Dies unterscheidet sie von Frameworks, welche z.T. nur Firmenintern verwendet werden, da sie während der ersten Entwicklungszyklen meist noch White-Box-Charakter haben. Allerdings ist diese Sicherheit manchmal trügerisch, da Komponenten, wie normale Pro-

gramme oder Bibliotheken auch, analsysiert werden können (*reverse engineering*). Dies war beispielsweise bei den VisualBasic Controls (VBX) von Microsoft Mitte der neunziger Jahre der Fall (Griffel 1998).

Bei einer Komponente muß die gewünschte Funktionalität von Anfang an in der Komponent vorhanden sein. Ein Hinzufügen von Funktionalität ist aufgrund des Black-Box-Charakters nicht möglich. Das Verhalten kann nur durch Konfigurationsparameter verändert werden. Eine Komponente hat eine höhere Einsatzwahrscheinlichkeit, je besser sie parametrisiert werden kann. Allerdings steigt mit der Anzahl der Parameter auch der Lernaufwand für den Nutzer. Dieses Problem ist als Skalierungsproblem von Bibliotheken bekannt (Biggerstaff 1994). Ein Lösungsansatz wäre die generative Programmierung.

Berater

Vorteile ergeben sich auch für die Berater von Kunden. Parametrisieren (neudeutsch: *customizing*) sie z.Z. größtenteils monolithische Anwendungssysteme, können sie bei einer weiteren Verbreitung der Komponententechnologie den Kunden bei der Auswahl der geeigneten Komponenten unterstützen. Dadurch daß Komponenten in Idealfall von mehreren Herstellern angeboten werden, reduziert sich auch die Abhängigkeit der Berater von den Softwareherstellern.

Kunde der Komponente

Da Komponenten aufgrund ihrer Unabhängigkeit von mehreren Firmen parallel entwickelt werden können, könnte dies die Qualität der erstellten Produkte steigern, getreu dem Motto: Konkurenz belebt das Geschäft. Kein kleineres Softwarehaus hat genügend Ressourcen, um große, monolithische Anwendungssysteme, wie etwa SAP R/3, von Grund auf neu zu entwickeln.

Kunde des Anwendungssystems

Da Komponenten möglichst eigenständig sein sollen, also möglichst wenig Annahmen über die Umgebung beinhalten sollen, in der sie eingesetzt werden, wird evtl. die Redundanz im fertigen Produkt erhöht. So kann es z.B. sein, das mehrere Komponeten ihre eigenen Routinen zur Ein-/Ausgabe mitbringen. Auch die Verwendung von dynamischen Bibliotheken ist nicht immer eine Lösung, da im schlimmsten Fall alle Beteiligten eine andere dynamische Bibliothek verwenden.

Das Verwenden verschiedener Ein-/Ausgabe-Bibliotheken birgt die Gefahr von unergonomischen Programmen mit sich, da die Eigenschaft der Konsistenz bei der Gestaltung der Dialoge nicht unbedingt erfüllt wird. Dies

führt im allgemeinen zu einer höheren Fehlerrate bei den Endnutzern der Komponenten (Brodbeck und Rupietta 1994).

Komponentenbasierte Systeme haben einen erhöhten Kommunikationsbedarf, da sie mit anderen Teilen der Anwendung nicht über direkte Schnittstellen kommunizieren. Diese Indirektion sollte die Effizienz des Systems beeinträchtigen.[17]

Komponenten haben im Vergleich zu monolithischen Programmen Vorteile im Bereich Sicherheit. Durch die bessere Trennung untereinander ist es einem potentiellen Angreifer nur möglich, die Kontrolle über die entsprechende Komponente zu erhalten. Von dort aus kann er seinen Angriff auf weitere Komponenten des Systems fortsetzen, aber bei einem konventionellen monolithischen Programm wäre er jetzt schon am Ziel. Auch lassen sich die zum Betrieb benötigten Rechte über Systemressourcen bei Komponenten feingranularer einstellen.

[17] obwohl es aus der Praxis auch Gegenbeispiele gibt, etwa der Vergleich zwischen `sendmail` und `postfix`, zwei Mailservern mit vergleichbarer Mächtigkeit (Schmachtel und Schreck 2000)

Kapitel 3

Entwurf von Veges

Nach der Vorstellung der Grundlagen zur komponentenbasierten Softwareentwicklung, sollen in diesem Kapitel die Ziele und der Entwurf des komponentenbasierten Frameworks VEGES vorgestellt werden. VEGES steht dabei für „*Ve*rknüpfung von *Ge*odaten mit *S*achdaten“. Als erstes wird die allgemeine Kommunikationsstruktur zwischen den Komponenten festgelegt, danach werden die Schnittstellen zwischen ihnen entworfen. Abschließend werden die daraus folgenden Anforderungen an den Komponentengenerator aufgeführt.

3.1 Zielsetzung

Bei dem im Rahmen dieser Diplomarbeit zu entwickelnden komponentenbasierten Framework sollen Daten aus einem Geografischen Informationssystem (GIS) mit beliebigen Sachdaten verknüpft werden. Dabei sollen auf Anforderung die geografischen Daten zu einem Sachdatensatz und der Sachdatensatz zu den geografischen Daten angezeigt werden.

Im typischen Anwendungsfall werden an einem Bildschirmarbeitsplatz Sachdaten bearbeitet. Um solche Anwendungen zu erleichtern oder überhaupt erst zu ermöglichen, werden die Geodaten entsprechend ihrer Verknüpfung mit den Sachdaten angezeigt. Neben dem Navigieren auf den Sachdatensätzen ist auch ein Navigieren auf der Karte möglich, wie etwa das Verschieben des angezeigten Kartenausschnittes oder das Ändern der Vergrößerung. Die Verknüpfung zwischen Geo- und Sachdaten ist beidseitig, so daß Objekte auf der Karte ausgewählt und daraufhin die entsprechenden Sachdaten angezeigt werden.

Das Anwendungsprogramm läuft auf dem Arbeitsplatzrechner, da die heutigen Rechner für diese Aufgabe leistungsfähig genug sind. Die Sach- und

Geodaten werden aufgrund ihrer Größe[1] und der vereinfachten Verwaltung auf externen Rechnern gespeichert. Dies ermöglicht es mehreren Benutzern diese Ressourcen zu teilen.

Das Anwendungsprogramm soll sich trotz seines komponentenbasierten Aufbaus dem Benutzer mit einer konsistenten Benutzungsoberfläche präsentieren. Bei den Benutzern handelt es sich um in ihrem Aufgabenbereich geschulte Anwender, welche Dialogerfahrung besitzen.

Das Framework zur Erstellung dieser Anwendungsprogramme soll komponentenbasiert sein, damit unabhängige Entwickler Bausteine dafür erstellen können. Die Entwickler sollten ausgebildete Software-Entwickler mit Kenntnissen in objektorientierter Programmierung sein und sich in das VEGES-Framework eingearbeitet haben. Der für die Bearbeitung der Sachdaten zuständige Teil des Frameworks soll mit Hilfe eines Generators erzeugt werden. Der Generator ermöglicht es den Entwicklern des Anwendungsprogramms, in kürzerer Zeit das System an Kundenwünsche anzupassen oder für neue Aufgabenbereiche zu entwickeln, indem er den stark variablen Sachdatenaspekt (halb-)automatisch erzeugen kann.

3.2 Veges-Framework

Einige grundlegende Entwurfsentscheidungen waren durch den äußeren Rahmen und die Einbettung der Diplomarbeit schon vorgegeben. Zum einen ist eine Komponente zur Visualisierung von geografischen Daten (im folgenden Geodatenkomponente genannt) schon im OFFIS für das InterGIS entwickelt worden, zum anderen soll ebenfalls im Rahmen dieser Diplomarbeit ein Generator zur Erzeugung von Sachdatenkomponenten entwickelt werden. Daraus ergab sich die Einteilung in die beiden Klassen von Komponenten, welche mittels des Frameworks verknüpft werden. Um für zukünftige Anforderungen offen zu sein, sollen jeweils beliebig viele Komponenten eingesetzt werden können. Auch wenn bisherige Anwendungen des InterGIS diese Funktionalität nicht nutzen, so ist sie zumindest denkbar. Auf diese Weise können beispielsweise zwei Geodatenkomponenten eingesetzt werden, von denen eine die Nahansicht der gewünschten Sachdaten anzeigt, während die andere eine Übersichtskarte mit geringerer Vergrößerung darstellt. Diese Übersichtskarte könnte die Orientierung des Benutzers bei Sachdaten erleichern, welche nicht benachbart auf der Karte liegen, sondern auf ihr verteilt sind. Bei dieser Konstellation kann der folgende Effekt besonders leicht auftreten:

[1] Bei der Entwicklung bei den Festplattengröße in den letzten Jahren ist auch eine kostengünstige Vorhaltung der Daten auf dem Arbeitsplatz denkbar. Bei dieser Lösung ist allerdings bei sich ändernen Daten ein größerer Aufwand für den Abgleich zu betreiben.

Ändert sich bei einem Datensatzwechsel der gesamte Bildinhalt der Geodatenkomponente, so hat der Benutzer keine Merkmale mehr, an denen er sich orientieren könnte. Dieser Effekt kann als „springen“ auf einer Karte bezeichnet werden. Der daraus resultierende Orientierungsverlust tritt vor allem dann auf, wenn die Sachdaten nicht nach räumlicher Nähe, sondern nach anderen Kriterien sortiert sind, und der Benutzer diese der Reihe nach bearbeiten muß. Dann ist evtl. bei jedem Datensatzwechsel eine Neuorientierung auf der Karte nötig. Die zweite Karte könnte in diesem Falle hilfreich sein, da sie zur gröberen Orientierung dient. Beispielsweise in dem sie Stadtteile anzeigt, während die Hauptkarte Straßenzüge anzeigt.

Durch die Möglichkeit, zwei Geodatenkomponenten auf die Anforderungen einer Sachdatenkomponente reagieren zu lassen, ist auch der Vergleich von Karten aus verschiedenen Geodatenquellen möglich. Ebenso lassen sich so auch zwei Geodatenkomponenten verschiedener Hersteller vergleichen.

3.2.1 Zugangsserver

Werden mehrere Komponenten eingesetzt, so muß sich der Benutzer unter Umständen bei jeder einzelnen Komponente bei der jeweiligen Datenquelle anmelden. Die Anmeldung erfolgt bei einer Sachdatenkomponente üblicherlicherweise bei einer Datenbank und bei einer Geodatenkomponente bei einem GIS. Dies wäre bei zwei Komponenten, und den daraus resultierenden zwei Anmeldungen, noch gerade zumutbar. Sobald die Anzahl der Anmeldungen darüber hinausgeht, evtl. noch unterschiedliche Benutzernamen und Paßwörter verwendet werden, sollte dem Benutzer eine Unterstützung geliefert werden, damit dieser nicht unnötig Zeit für das Anmelden verbraucht. Idealerweise sollte sich der Benutzer nur einmal anmelden und danach direkt arbeiten können. Diese Funktionalität kann durch eine Schnittstelle zu einem Zugangsserver erreicht werden. Der Zugangsserver speichert für den Benutzer seine jeweiligen Benutzernamen und Paßwörter zu den dazugehörigen Datenquellen und liefert sie auf Anfrage zurück.

Allerdings kann dieses Vorgehen zum Teil unerwünscht sein, da alle Zugangsdaten zentral gespeichert werden, was den Zugangsserver zu einem potentiellen Sicherheitsrisiko gegenüber Angreifern macht. So soll es trotz Verwaltung der Zugangsdaten für die meisten Datenquellen auch möglich sein, daß sich der Benutzer aus Sicherheitsgründen bei bestimmten Datenquellen persönlich anmeldet.

Sollte der Bedarf nach mehreren Zugangsservern bestehen, etwa weil schon bestehende Systeme jeweils für die Sach- und Geodatenseite vorhanden sind, so kann dies mit Hilfe eines Zugangsserverproxies realisiert werden (s. Abbildung 3.1). Der Proxy, auch vorgelagerter Stellvertreter genannt,

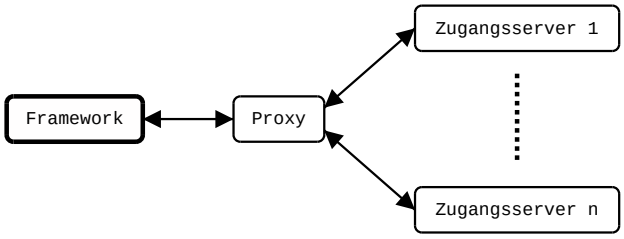

Abbildung 3.1: Proxy für mehrere Zugangsserver

fragt bei einer Anfrage bei allen registrierten Zugangsservern nach, ob sie die gewünschten Anmeldungsdaten haben, und reicht das Ergebnis an das Framework zurück. Diese Funktionalität wurde aus dem Framework entfernt, weil sie nicht immer benötigt wird und so die Entwicklung des Frameworks vereinfacht wurde. Die üblichen Erweiterungen für Proxies, etwa das Zwischenspeichern von Daten (*caching*) oder unterschiedliche Abfragestrategien (z.B. sequenzieller oder paralleler Zugriff) können so implementiert werden, ohne das Framework selbst zu ändern.

Abrechnung der Kosten

Viele Zugangsserver enthalten Schnittstellen zur Abrechnung von angefallenen Kosten während der Benutzung der entsprechenden Dienste. Um diese Funktionalität den Komponenten ebenfalls zur Verfügung stellen zu können, sollte der Zugangsserver daher auch die Kosten für die angemeldeten Benutzer verwalten können. Die Wahl des Weges, den die Rechnungen nehmen, ist nicht unproblematisch. Aus Sicherheitsgründen sollten die Rechnungen von den Datenquellen direkt an den Zugangsserver gesandt werden, da die Sach- und Geodatenkomponenten sowie der Framework-Kern im typischen Anwendungsszenario auf dem System des Benutzers laufen und daher dort manipuliert werden können. Eine Alternative ist die Übertragung der Rechnungen durch das potentiell unsichere System mit Hilfe von verschlüsselten Nachrichten. Falls nicht so hohe Sicherheitsanforderungen erreicht werden können oder gegeben sind (weil z.B. auch die Datenquellen auf dem System des Benutzers laufen oder die Abrechnung nur für eine Nutzungsstatistik benötigt wird) so können auch unverschlüsselte Rechnungen verwandt werden. Diese Anforderungen ermöglichen es auch den Komponenten ihrerseits, Rechnungen an den Zugangsserver schicken.

3.2.2 Framework-Kern

Neben den einzelnen Komponenten wird noch ein Rahmen benötigt, um ein Anwendungssystem zu erhalten. Dieser Rahmen, im folgenden Framework-Kern genannt, übernimmt alle nicht von den Komponenten selbst durchgeführten Aufgaben. Eine wichtige Aufgabe ist die Kommunikation im Framework selbst. So müssen die Sachdaten- und die Geodatenkomponenten auf Anfragen der „Gegenseite“ achten. Reagieren beide Komponentenklassen bei jeder Änderung, so würde dies dem Beobachtermuster (vergleiche das Beispiel im Abschnitt 2.2) mit dem Unterschied entsprechen, daß hier das Subjekt des Einen der Beobachter des Anderen ist. Die beiden Komponenten würden sich also gegenseitig beobachten, um auf Änderungen zu reagieren. Dieses Entwurfsmuster läßt sich auch verwenden, wenn nicht bei jeder Änderung, sondern nur auf Wunsch aktualisiert wird.

Im folgenden sollen zwei Möglichkeiten präsentiert werden, wie die Kommunikation zwischen den Komponenten durchgeführt werden kann: Zentral oder dezentral. Die Komponenten verständigen sich in beiden Fällen über Kommunikationskanäle untereinander. Bei einem zentralen Aufbau laufen alle Nachrichten über die Zentrale und werden dort weiter verteilt. Dieser Aufbau ist mit der klassischen Post zu vergleichen. Beim dezentralen Entwurf kommunizieren alle Komponenten direkt miteinander, was mit relativ unabhängigen Boten zu vergleichen wäre.

Die Komponenten kommunizieren im zentralen Entwurf (s. Abbildung 3.2) direkt mit ihren jeweiligen Datenquellen, aber die Verknüpfung der Sachdaten- und der Geodatenkomponente erfolgt über das Framework, durch das alle Nachrichten laufen. Dem gegenüber tauschen die Komponenten im dezentralen Entwurf (s. Abbildung 3.3) Nachrichten nicht nur direkt mit ihren Datenquellen, sondern auch mit allen ihnen zugeordneten Komponenten aus. Das Framework hat zwar auch hier Verbindungen zu allen Komponenten, aber im Gegensatz zum ersten Entwurf werden diese nur zum Verwalten der Komponenten verwendet.

Die jeweiligen Vor- und Nachteile der beiden Entwürfe sollen nun gegenübergestellt werden.

Zentraler Entwurf

Die Nachteile des zentralen Entwurfs lassen sich wie folgt zusammenfassen:

- Wie bei allen zentralisierten Systemen „steht und fällt“ das System mit der Funktionstüchtigkeit der Zentrale. Ist diese nicht erreichbar, sind keine Aktualisierungen möglich. Im Gegensatz ist bei der dezentralen

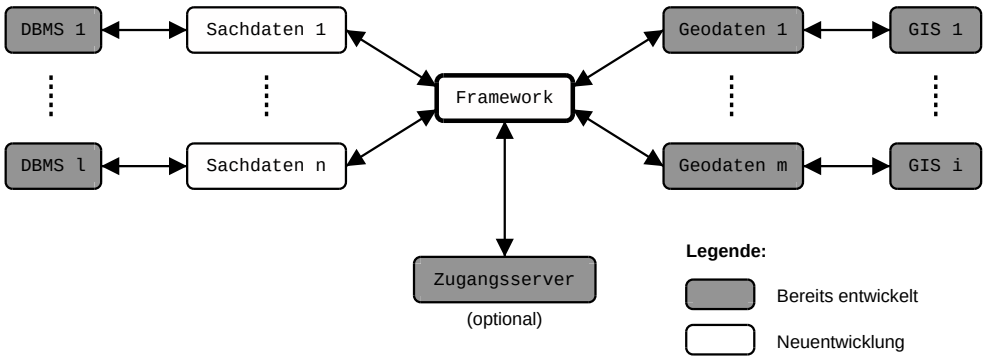

Abbildung 3.2: Zentraler Entwurf des Frameworks

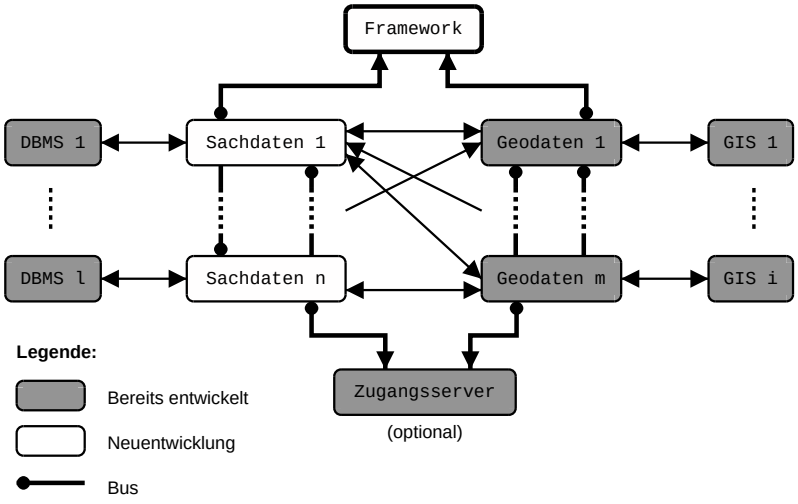

Abbildung 3.3: Dezentraler Entwurf des Frameworks

Lösung bei einem Ausfall des Framework-Kerns nur das An- oder Abmelden der Komponenten nicht mehr möglich ist.

- Im ungünstigen Fall wird der Kommunikationsaufwand erhöht. Würden etwa die Sach- und die Geodatenkomponente auf demselben und das Framework auf einem anderen Rechner aktiviert, so müßten alle Nachrichten den Umweg über das Framework nehmen, anstatt direkt lokal miteinander zu kommunizieren.

- Da die Zentrale auf dem Rechner des Benutzers läuft, besteht die Gefahr, daß Rechnungen bei ihrem Weg durch die Zentrale manipuliert werden.

Aber es sind auch Vorteile vorhanden:

- Hat eine Komponente mehrere Zuhörer, so müssen weniger Nachrichten verschickt werden, da der Sender nur eine Nachricht an das Framework senden muß, welches die Nachrichten dann entsprechend weiterleitet und dabei vervielfältigt.

- Werden Adapter, etwa zur Protokollumsetzung bei verschiedenen Versionen, eingesetzt, so braucht der Adapter nur einmal erzeugt zu werden, und nicht bei jedem Zuhörer (s. Abbildung 3.4). Dadurch können Ressourcen eingespart werden.

Abbildung 3.4: Adapter

Dezentraler Entwurf

Der dezentrale Entwurf hat die folgenden Nachteile:

- Bei einer Änderung der beteiligten Komponenten müssen alle registrierten Zuhörer benachrichtig werden und selbst entscheiden, ob sie betroffen sind.

- Werden die Nachrichten über ein ungesichertes Medium (etwa eine Netzverbindung) versandt, so muß Sorge getragen werden, daß keine Schwierigkeiten mit der Nebenläufigkeit entstehen. Dies ist beispielsweise der Fall, wenn eine sendende Komponente Nachrichten an einen

Zuhörer sendet, der sich schon beim Framework abgemeldet hat. Dies ist möglich, weil aufgrund von Verzögerungen bei der Netzwerkkommunikation der Sender eventuell noch keine Nachricht über die Abmeldung erhalten hatte.

- Die Funktionalität zur Verwaltung aller Kommunikationskanäle muß von den einzelnen Komponenten bereitgestellt werden, wodurch ihre Komplexität steigt.

Seine Vorteile sind:

- Die Komplexität des Framework-Kerns ist gering, da bis auf die Verwaltung der angemeldeten Komponenten die Funktionalität der Kommunikation in den Komponenten angesiedelt ist.
- Die Vorteile des verteilten Rechnens können genutzt werden. Sind die einzelnen Komponenten auf mehrere Rechner verteilt worden, so kann die Kommunikation zwischen ihnen direkt erfolgen. Die Überlastung einer Zentrale wird dadurch unwahrscheinlicher.
- Die Gefahr der Manipulation von Rechnungen wird verringert, da die einzelnen Teilnehmer ihre Rechnungen direkt an den Zugangsserver schicken.

Nach der Vorstellung der einzelnen Vor- und Nachteile der beiden Varianten, welche in der Tabelle 3.1 noch einmal zusammengefasst dargestellt werden, sollen diese nun gegeneinander abgewägt werden.

	Dezentral	**Zentral**
Komplexität des Framework-Kerns	niedrig	hoch
Komplexität der Komponenten	hoch	niedrig

Tabelle 3.1: Gegenüberstellung der beiden Entwürfe

Die mit diesem Framework erzeugten Anwendungen werden wohl im Bereich von Anwendungen für einen Arbeitsplatzrechner liegen. Die Datenquellen werden in diesem Szenario auf externen Rechnern vorgehalten. Damit können die Unterschiede in der Kommunikation zwischen den Komponenten vernachlässigt werden. Beide Varianten würden zur Kommunikation die schnellen internen Verbindungen im Rechner verwenden, wo eine Überlastung der Zentrale unwahrscheinlich ist. Ihre eventuell vorhandenen Datenquellen werden weiterhin über Schnittstellen zu anderen Rechnern angesprochen. Der Vorteil des verteilten Rechnens kann in dieser Konstellation beim dezentralen

Entwurf nicht ausgenutzt werden. Anwendungen für einen Arbeitsplatzrechner haben aber in der Regel keinen solchen Leistungbedarf.

Aus der unterschiedlichen Kommunikationsstruktur folgt die unterschiedliche Verlagerung der administrativen Aufgaben. Ein Großteil dieser Funktionalität wird beim zentralen Entwurf in den Framework-Kern, beim dezentralen Entwurf in die einzelnen Komponenten integriert. Aus der Aufgabenstellung folgt, daß der Framework-Kern einmal entwickelt werden soll, die Komponenten hingegen mehrfach. Um den Entwicklungsaufwand der Komponenten möglichst gering zu halten, sollte ihre Komplexität nicht unnötig erhöht werden. Zwar sollen die Sachdatenkomponenten mit Hilfe eines Generators erzeugt werden, so daß auch diese Funktionalität ebenfalls automatisch in die erzeugten Komponenten integriert werden könnte. Sollte allerdings eine Sachdatenkomponente einmal von Grund auf ohne den Generator erstellt werden müssen, so müßte auch dieser Teil neu implementiert werden. Diese Funktionalität in eine Basisklasse zu integrieren, von der sowohl der Generator als auch der Entwickler seine Komponentenklasse ableitet, wäre nur ein Teilerfolg, da die in Abschnitt 2.3.3 beschriebenen Probleme mit der Einbindung in eine Klassenhierarchie bestehen bleiben.

Die Vorteile einer Auslagerung der admistrativen Aufgaben verstärken sich bei den Geodatenkomponenten, da diese zumeist Teil eines GIS-Paketes sein werden, und deswegen in der Regel mit Hilfe von Adapter an das Frameworkprotokoll angepaßt werden müssen. Durch die Auslagerung werden die Anforderungen an die Adapter verringert und dadurch ihre Entwicklung vereinfacht.

Fazit

Durch die Integration auf einem Arbeitsplatzrechner kann keiner der beiden Entwürfe seine jeweiligen Vorteile bei der Kommunikation ausspielen, die Kommunikationstruktur kann also vernachlässigt werden. Ein entscheidener Vorteil für den zentralen Entwurf ist die geringere Komplexität der Komponenten im Vergleich zum dezentralen Entwurf. Dieses vereinfacht die Entwicklung von Komponenten durch externe Anbieter. Aus diesen Gründen wird der zentrale Entwurf realisiert.

3.2.3 Konfigurationsdaten

Die Bausteine eines komponentenbasierten Frameworks (der Kern und die Komponenten) benötigen ein Verfahren, um Informationen über den Zeitraum ihrer Laufzeit hinaus zu speichern. Mit diesen Konfigurationsdaten wird die Flexibilität der Komponenten erhöht, indem einige Parameter erst

zur Laufzeit abgefragt, und nicht schon zur Übersetzungzeit fest vorgegeben werden.

Von einer implementierten Komponente können mehrere Instanzen angelegt werden. Diese Instanzen haben das Verhalten ihrer Vorlage übernommen, können aber zur Laufzeit völlig unabhängig voneinander agieren. Die Vorlagen werden Komponentenklassen genannt. Dieses Konzept ähnelt dem der objektorientierten Programmierung, nur daß hier ganze Programme anstelle von Variablen verwendet werden. Jede Instanz hat einen eigenen Speicher zur Laufzeit für ihren Zustand den sie nicht mit anderen Instanzen teilt. Ein unterschiedliches Verhalten läßt sich durch Konfiguration erreichen. Dadurch wird der Laufzeitspeicher mit Werten initialisiert.

Da heute die meisten modernen Betriebssysteme mehrbenutzerfähig sind, sollte bei den Konfigurationsdaten zwischen den Daten einzelner Instanzen der Komponente und der Konfiguration der einzelnen Benutzer unterschieden werden. Eine Konfiguration für eine Klasse von Komponenten wird für alle Benutzer wie folgt erzeugt: Ein Benutzer, in der Regel der Systemadministrator, erzeugt eine allgemeine Konfiguration für die gewünschte Komponentenklasse. Danach wird diese Konfiguration für die einzelnen Komponenteninstanzen angepaßt, indem die Basiskonfiguration erweitert oder überschrieben wird. Komponenteninstanzen sind hierbei nicht laufende Programme, sondern auf den jeweiligen Anwendungsfall konfigurierte Komponenten.

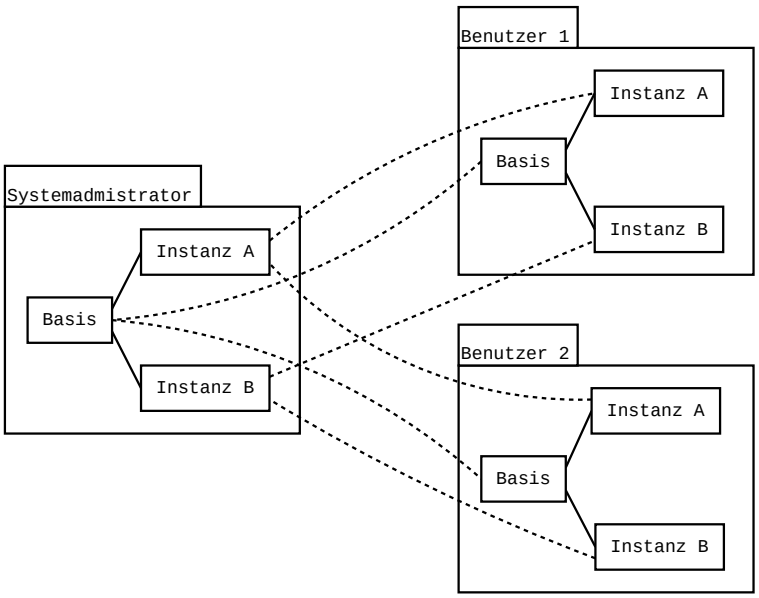

Abbildung 3.5: Konfigurationsbaum einer Komponentenklasse

Diese Menge von Konfigurationsdateien enthält die Standardkonfiguration für alle normalen Benutzer[2]. Diese können aber ihrerseits die Basiskonfiguration *sowie* die speziellen Konfigurationen der Instanzen nach ihren Wünschen anpassen. Der daraus resultierende baumartige Aufbau wird in Abbildung 3.5 dargestellt.

Diese Vorgehensweise hat den Vorteil, das Eigenschaften zentral geändert werden können, der administrative Aufwand somit sinkt und dennoch benutzerspezifische Anpassungen möglich sind.

3.2.4 Sicherheitsaspekte

Die Datenquellen der einzelnen Komponenten benötigen Schutz vor unberechtigtem Zugriff, spätestens wenn sie personenbezogene Daten enthalten. Dieser Schutz soll durch einen Kennwortschutz gewährleistet werden. Diese Kennwörter können aber ausspioniert werden, wenn die Komponenten auf dem Rechner des Benutzers laufen oder die Paßwörter über ungesicherte Leitungen übertragen werden. Der zentrale Entwurf erhöht das Risiko, da die gesamte Kommunikation der Komponenten mit dem Zugangsserver über den Framework-Kern läuft. Die Komponenten sowie der Framework-Kern können analysiert, oder die Kommunikation zwischen den Partnern belauscht werden. Gegen eine solche Analyse würde eine Ausführung der Komponenten auf einem gesicherten Rechner schützen, wo etwa wie beim X-Windowsystem unter Unix im Prinzip nur der Bildinhalt an den Rechner des Benutzers übermittelt wird. Gegen ein Belauschen der Kommunikationskanäle hilft der Einsatz von Verschlüsselungsverfahren. Das Verfahren könnte von evtl. Standards im Anwendungsgebiet vorgegeben sein. Ist dies nicht der Fall, so wird eine Abwägung zwischen dem Sicherheitsbedürfniss und den daraus resultierenden Kosten erfolgen, da sichere Verfahren einen höhren Ressourcenverbrauch haben.

Die meisten Verschlüsselungsverfahren arbeiten transparent, so daß sie mit Hilfe des Adaptermusters vor beide Schnittstellen der Kommunikationspartner vorgelagert werden können. Dies ermöglicht die Wahl des Verfahrens je nach den Sicherheitsanforderungen im Anwendungsfall.

3.3 Schnittstellen

Das Protokoll zwischen den Sach- und Geodatenkomponenten besteht aus zwei Unterprotokollen, eines für die Kommunikation mit einer anderen Komponente, das anderes für die Interaktion mit dem Zugangsserver. Diese Tren-

[2]Ein Arbeiten mit der Anwendung als Systemadmistrator sollte vermieden werden

nung hat zum einen den Vorteil, daß das Protokoll zum Zugangsserver für die Schnittstelle zwischen Framework-Kern und Zugangsserver wiederverwendet werden kann. Der weitaus größere Vorteil besteht aber darin, daß die Kommunikationstruktur einfacher umgestellt werden kann. Sollte sich die Abwägung zugunsten des zentralen Entwurfes als falsch erweisen, so könnten die Unterprotokolle als Schnittstellen wiederverwendet werden. Die Kommunikation würde nicht mehr durch den Framework-Kern erfolgen, sondern direkt (s. Abbildung 3.6). Bei einer Umstellung müßte dann nur ein Protokoll zur An- und Abmeldung bei dem Framework-Kern entwickelt werden. Eine Neuentwicklung der gesamten Protokolle zwischen den Komponenten untereinander, sowie zwischen den Komponenten und dem Zugangsserver von Grund auf entfiele.

Die einzelnen Komponenten des Frameworks kommunizieren miteinander durch Nachrichten. Jede Schnittstelle stellt ein Protokoll mit einem bestimmten Satz möglicher Nachrichten zur Verfügung, welche zum Teil nur in einer festgelegten Reihenfolge aufgerufen werden können.

Beim Aufbau der Verbindung wird von beiden Kommunikationspartnern die gewünschte Protokollversion ausgehandelt. Dadurch wird sichergestellt, daß inkompatible Versionen erkannt werden. Fehler im Protokoll werden durch eine Protokollfehlernachricht dem Verursacher mitgeteilt. Diese Nachricht ist in jedem der im folgenden vorgestellten Protokolle enthalten.

Der Framework-Kern wird bei den Schnittstellen als transparent angesehen, d.h. es werden Schnittstellen zwischen den beiden Kommunikationspartnern vorgestellt. Der Framework-Kern hat jeweils die gleiche Schnittstelle, wie sie auch der eigentliche Kommunikationspartner haben würde. Dieser Aufbau ist dem des Stellvertreters mit dem Unterschied ähnlich, daß der Framework-Kern die Nachrichten nicht nur durchreicht, sondern auch noch für die Verteilung verantwortlich ist. In der Abbildung 3.7 ist die Struktur beispielhaft mit zwei Komponenten dargestellt.

3.3.1 DBMS und Sachdatenkomponenten

Diese Schnittstelle ist je nach den Anforderungen an die Sachdatenkomponente frei wählbar. So ist unter Umständen gar kein Datenbank Management System (DBMS) notwendig, wenn eine Sachdatenkomponente aus anderer Quelle erfährt, welchen Ort die Geodatenkomponenten auf der Karte anzeigen soll. Dies wäre z.B. bei einem Taxi-Leitstand der Fall, wenn die Taxis mit Geräten zur Positionsermittlung versehen sind und die Sachdatenkomponente von dort die Koordinaten erhält.

Üblicherweise wird aber eine Sachdatenkomponenten auf einen Datenbestand zugreifen wollen, und dieser wird wohl, ab einer gewissen Größe

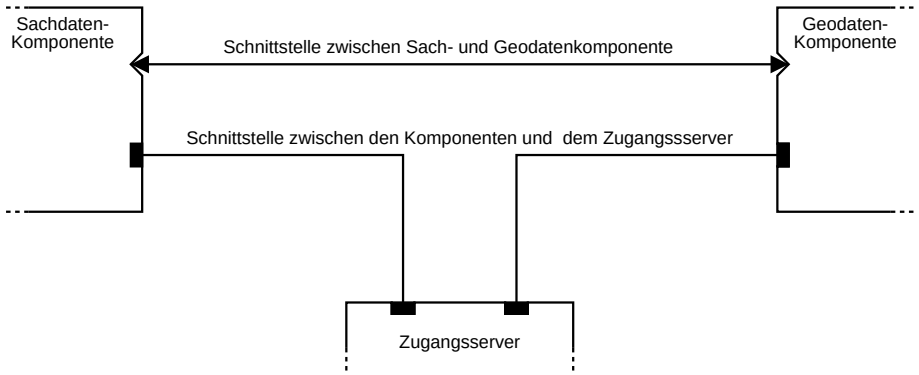

Abbildung 3.6: Wiederverwendung der Protokolle nach der Umstellung

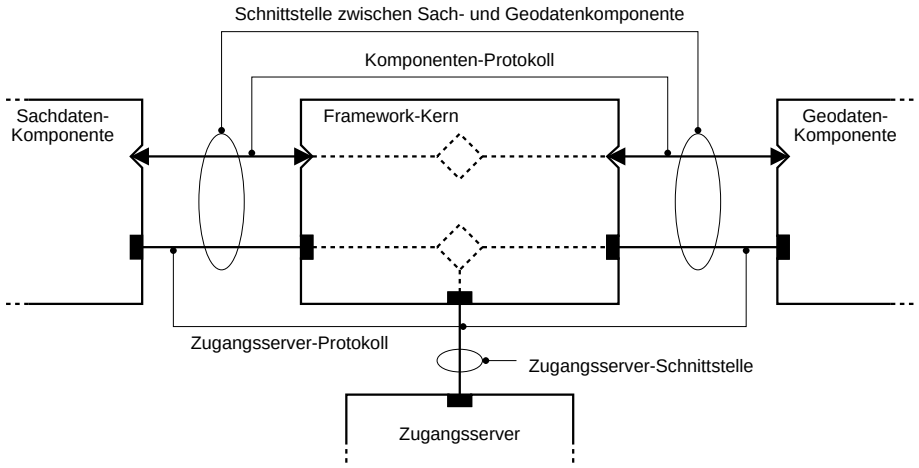

Abbildung 3.7: Transparenz durch Trennung der Protokolle

oder Komplexität, in einer Datenbank gespeichert. In diesem Fall sollte die Kommunikationsschnittstelle passend zur Programmiersprache der Sachdatenkomponente und dem DBMS gewählt werden. Das Verwenden einer Datenbankschnittstelle in der gewünschten Programmiersprache vereinfacht einerseits die Entwicklung der Komponente, anderseits wird so meist ein effizienterer Zugriff ermöglicht. Soll eine größere Unabhängigkeit von der verwendeten Datenbank erreicht werden, so kann das Entwurfsmuster Schablone eingesetzt werden. Sollte die vorhandende Schnittstelle nur ein Protokoll können, so kann evtl. ein weiteres über das Entwurfsmuster Adapter angebunden werden.

3.3.2 Geodatenkomponente und GIS

Für die Schnittstelle zwischen der Geodatenkomponenten und dem GIS gilt ebenfalls das schon für die Schnittstelle zwischen DBMS und Sachdatenkomponente oben Gesagte: Sie ist frei wählbar, und kann daher nach den Entwurfszielen der Geodatenkomponente (Effizienz, Unabhängigkeit) entworfen werden.

3.3.3 Zugangsserver-Protokoll

Mit dem Zugangsserver kommunizieren die Sach- und die Geodatenkomponente über den Framework-Kern. Um zu verhindern, daß sich der Benutzer immer wieder anmelden muß, sollten die Komponenten zu ihrer Datenquelle das jeweilige Login und Paßwort haben. Um das Verlegen von Datenquellen zu ermöglichen, gibt der Zugangsserver auch den Ort der Datenquelle zurück. Dies ermöglicht es, den Zugangsserver auch als Namens- und Verzeichnisdienst zu nutzen, indem die Datenquellen mit symbolischen Namen (auch Dienste genannt) abgefragt werden. Da jede Datenbank oder jedes GIS-Paket eine eigene Konvention hat, welches Format die Adresse eines ihrer Server zu haben hat, kann von der Komponente ein Format gewünscht werden. In einem betrieblichen Anwendungsfall wird eventuell neben der Auskunftsfunktionalität auch eine Erfassung der Nutzung der Dienste von einzelnen Benutzern oder Benutzergruppen benötigt, um interne oder externe Rechnungen ausstellen zu können. Stellen externe Dienstleister einige Datenquellen zur Verfügung, ist eine Maklerfunktion (engl.: *broker*) im Zugangsserver hilfreich.

Ein Benutzer kann dann beispielsweise die der Aufgabe angemessene Datenquelle auswählen, wie etwa eine günstigere Übersichtskarte um den gewünschten Ausschnitt zu finden. Dieser Ausschnitt kann danach von einer detailreicheren, und damit in der Regel teureren, Datenquelle angefordert werden.

Auch bei der Konfiguration von Anwendungen bringt ein Makler Vorteile. Eine Auswahl aus der Auflistung ist angenehmer und weniger fehlerträchtig als das Eintragen der Dienste aus den Übersichten der Anbieter, welche eventuell nur in gedruckter Form vorliegen. Aus diesen Anforderungen ergeben sich die folgenden Nachrichten. Eine Übersicht über die möglichen Nachrichten findet sich in der Tabelle 3.2.

Nachricht	Funktionalität
Anmelden Abmelden	An- und Abmelden
Anfrage Anfrage – Antwort	Auskunft zu einem bestimmten Angebot
Rechnung	Abrechnung
Diensteanfrage Diensteanfrage – Anwort	Auskunft über das Angebot

Tabelle 3.2: Nachrichtenübersicht: Zugangsserver-Protokoll

Anmelden

Mit dieser Nachricht meldet sich der Benutzer an dem Zugangsserver an. Erst nach der Anmeldung sind Anfragen möglich.

Login Das ist der Benutzername (Login) auf dem Zugangsserver.

Paßwort Mit diesem Paßwort zu dem obigem Login autorisiert sich der Benutzer beim Zugangsserver. Das Feld könnte das Paßwort auch verschlüsselt enthalten, um ein „Abhören“ im Framework-Kern oder in den Komponenten zu verhindern.

Abmelden

Mit dieser Nachricht meldet sich der Benutzer vom Zugangsserver ab. Die Abmeldung ist im allgemeinen nicht erforderlich. Sollte ein Zugangsserver eine Art Lizenzverwaltung betreiben, d.h. eine Anzahl von Zugängen zu einer Datenquelle für eine größere Anzahl von Benutzern bereithalten, könnte er mit Hilfe dieser Nachricht eine Lizenz freischalten und sie dann später neu vergeben. Daher sollte man sich vom Zugangsserver erst abmelden, wenn man alle Zugänge geschlossen hat, welche man von ihm erhalten hat.

keine Parameter

Anfrage

Nach einer erfolgreichen Anmeldung kann man mit dieser Nachricht Anfragen an den Zugangsserver stellen.

Dienst Hier wird der Name des gewünschten Dienstes übermittelt. Der symbolische Namen wird vom Zugangsserver in die Adresse einer Datenquelle umgewandelt. Bei Zugangsservern, welche keine Umsetzung von Diensten in Adressen durchführen, sondern nur zu einer Datenquelle die Zugangsdaten liefern, sollte die Datenquelle eingetragen werden.

Format Mit Hilfe dieses Feldes wird das Format der zurückzuliefernden Adresse der Datenquelle bestimmt. Jeder Zugangsserver muß das Format „kein Format“ unterstützen, da nicht jeder konkrete Zugangsserver alle anderen spezielleren Formate unterstützen muß. Auf diese Weise ist es möglich, zu einer bekannten Datenquelle die Zugangsdaten zu erhalten, ohne das der Zugangsserver eine Konvertierung versucht.

Anfrage – Antwort

Nach der Anfrage werden die Zugangsdaten mit dieser Antwortnachricht zurückgeliefert.

gültig In diesem Feld wird zurückgeliefert, ob zu der vorherigen Anfrage ein gültiger Zugang in den folgenden Datenfeldern enthalten ist. In diesem Feld wird auch der Grund eines eventuellen Fehlschlages kodiert. Mögliche Ursachen wären etwa:

- Die Datenquelle existiert nicht,
- Der Benutzer hat keinen Zugriff auf die Datenquelle oder
- Die Datenquelle ist zur Zeit nicht erreichbar, etwa durch fehlende freie Lizenzen oder durch Netzwerkschwierigkeiten.

Auch eine teilweise Gültigkeit der obigen Datenfelder ist möglich, etwa wenn der Benutzer sich selbst bei der entsprechenden Komponente anmelden muß. In diesem Falle wird kein gültiges Paßwort mitgeliefert.

Login Dieses Feld enthält das Login zur gewünschten Datenquelle.

Paßwort Hier ist das Paßwort zur gewünschten Datenquelle enthalten, außer der Zugangsserver meldet, daß eine persönliche Anmeldung an der Datenquelle erforderlich ist.

Datenquelle-Adresse Dieser Eintrag entält die Adresse der Datenquelle.

Rechnung

Mit dieser Nachricht können Komponenten und der Framework-Kern Rechnungen an den Zugangsserver versenden. Hat der Zugangsserver keine Abrechnungsfunktionalität, werden diese Nachrichten ignoriert. Eine Rechnung darf nur nach einer erfolgreichen Anmeldung verschickt werden.

Betrag Der Betrag, über den diese Rechnung ausgestellt wurde.

Währung Die Währung zum obigen Betrag. Falls Statistiken erstellt werden sollen, könnte anstelle einer Währung auch eine andere Einheit verwendet werden, etwa die Anzahl der erfolgten Abfragen.

Quelle Bei einer Komponente handelt es sich entweder um den Dienst oder um die Datenquelle-Adresse, welche der Komponente mit der Anfrage – Antwort Nachricht mitgeteilt wurde. Beim Framework-Kern sollte sie sich aus dem Rechnernamen und dem Benutzer zusammensetzen.

Posten Hier ist eine freie Beschreibung der in Rechnung gestellten Dienstleitung möglich. Sie sollte so gewählt werden, daß sie als Eintrag in einer übliche Rechnung verwandt werden kann.

Diensteanfrage

Nach einer erfolgreichen Anmeldung kann der Benutzer mit dieser Nachricht die für ihn erhältlichen Dienste abfragen.

Damit ist es dem Benutzer möglich, im Einzelfall zu entscheiden, welcher Dienst für das Erreichen seines Zieles am besten geeignet ist.

keine Parameter

Diensteanfrage – Antwort

Mit dieser Nachricht werden die für einen Benutzer verfügbaren Dienste als Anwort auf die Diensteanfrage zurückgeliefert.

Anzahl Die Anzahl der verfügbaren Dienste. Durch die Übermittlung der Anzahl am Anfang der Nachricht ist es möglich, dem Benutzer bei langsamen Verbindungen mit Hilfe von Fortschrittsbalken eine Rückmeldung zu liefern.

Die Anzahl der Dienste kann null sein, etwa wenn keine für die Dienste benötigten Lizenzen zur Zeit zur Verfügung stehen.

Dienste Hier werden die verfügbaren Dienste aufgelistet. Dieses Feld kann leer sein.

3.3.4 Komponenten-Protokoll

Die Verbindung zwischen der Sachdatenkomponente und der Komponente zur Anzeige der Geodaten bildet das „Herzstück“ dieses Frameworks. Die Sachdatenkomponente ist zwar mit dem Framework-Kern als Kommunikationszentrale verbunden, aber die Nachrichten werden von diesem an die Geodatenkomponenten weitergereicht. Daneben wird über das Framework noch eine Verbindung zu dem Zugangsserver aufgebaut, um das Anmelden an die eigene Datenquelle zu ermöglichen. Dieses Protokoll wurde im Unterabschnitt 3.3.3 erläutert.

In diesem Abschnitt wird beschrieben, mit welchen Nachrichten die Verknüpfung der Daten zwischen den Komponenten erfolgt. Als erstes soll der Fall behandelt werden, in dem eine Geodatenkomponente um die Anzeige eines bestimmten Objektes gebeten wird. Dafür benötigt die Geodatenkomponente die Information, welchen Kartenausschnitt sie in welcher Vergrößerung anzeigen soll. Die Sachdatenkomponenten kann diese Informationen der Geodatenkomponente auf zwei verschiedene Arten mitteilen:

1. den Kartenausschnitt und die Vergrößerung oder
2. einen Sekundärschlüssel und die Vergrößerung des anzuzeigenden Objektes.

Bei der ersten Variante ergeben sich daraus die folgenden Nachteile:

- Werden Koordinaten übermittelt, so muß das Koordinatensystem eventuell angepaßt werden, d.h. beide Komponenten müssen sich auf das gleiche Koordinatensystem beziehen.
- Die gewünschte Vergrößerung bezieht sich häufig auf die Größe des anzuzeigenden Objektes im Verhältniss zum zur Verfügung stehenden Bildschirmplatz, etwa wenn es den Bildschirm voll ausfüllen soll. Dann muß eine der beiden Komponenten die passende Vergrößerung berechnen.
- Die Sachdatenquellen kennen gewöhnlich keine Koordinaten. Beispielsweise liefern die wenigsten Adressdatenbanken zu einer Adresse die Koordinaten.

Aber es sind auch Vorteile bei der Übertragung eines Kartenausschnittes mit Koordinaten vorhanden:

- Koordinaten bezogene Anfragen sind in der Regel auf der Geodatenkomponentenseite leistungsfähiger abarbeitbar, da ein neuer Ausschnitt evtl. nur ein Verschieben nötig macht, welches die Komponente aus ihrem Cache durchführen kann. Damit können Anfragen an den GIS-Server eingespart werden, welche bei einer Sekundärschlüsselanfrage zuerst die Position des Objektes bestimmen müßte.

- Mit Hilfe einer Koordinatenabfrage können auch die Positionen von Objekten angezeigt werden, welche der GIS-Server gar nicht kennt.

- Das Koordinatensystem ermöglicht auch das Anzeigen von Objekten, deren Position sich nur auf Sachdatenseite ändert, da ein schreibender Zugriff auf die (meistens auf lesenden Zugriff optimierten) GIS-Server zu ineffizient ist. Dieses Vorgehen wäre eine Lösungsmöglichkeit, um den häufigen Schreibzugriff bei sich bewegenden Objekten zu vermeiden. Mögliche Anwendungen wären Informationssysteme für Fuhrparks. Ein weiterer Vorteil dieser Lösung ist, daß durch den fehlenden Schreibzugriff auf die Geodaten die Nutzung eines externen Dienstleisters möglich ist.

Bei der Verwendung der zweiten Variante, Anfragen über den Sekundärschlüssel, tritt die folgende Problematik auf:

- Die von der Sachdatenkomponente verwendete Datenbank und das GIS sind sehr eng miteinander verknüpft. Werden in die eigentlichen Sachdaten die Sekundärschlüssel der Geo-Objekte eingetragen, so sind bei einem Wechsel eines der beiden Systeme unter Umständen Änderungen am anderen System nötig. So müßten bei einem Wechsel des GIS-Paketes die Tabellen im DBMS angepaßt werden. Um eine zu enge Verknüpfung der Systeme oder eine Veränderung der Kerndaten zu verhindern, müßte eine Umsetzung, etwa durch Hilfstabellen, stattfinden, welches eine gewisse Leistungseinbuße darstellt.

Da man nicht von allen GIS eine Sekundärschlüsselschnittstelle erwarten kann, sollte die Schnittstelle die Auswahl des anzuzeigenden Ausschnittes mittels der Koordinaten ermöglichen. Damit wäre eine Einbindung von Kartenservern möglich, welche im Vergleich zu GIS nur eingerasterte Karten anbieten und keine Objekte mit Sekundärschlüsseln kennen. Da diese Entscheidung keine entweder-oder Entscheidung ist, sollte es daneben auch noch

die Sekundärschlüsselschnittstelle geben, um die Kommunikation eventuell zu vereinfachen und um das Anlegen von Hilfsdaten auf einer von beiden Seiten im besten Falle zu verhindern.

Das Protokoll umfaßt also zwei Klassen von Nachrichten, einmal Nachrichten auf Basis des Koordinatensystems zum anderen Nachrichten auf Basis des Sekundärschlüssels. Eine Übersicht findet sich in Tabelle 3.3.

Nachricht	**Funktionalität**
Zeige Ausschnitt (Quader)	Koordinaten
Zeige Ausschnitt (Sekundärschlüssel) Zeige Objekt	Sekundärschlüssel

Tabelle 3.3: Nachrichtenübersicht: Komponenten-Protokoll

Zeige Ausschnitt (Quader)

Mit Hilfe dieser Nachricht teilt die Sachdatenkomponente der Geodatenkomponente den anzuzeigenden Ausschnitt in Form eines Quaders mit.

Quader In diesem Quader befinden sich die darzustellenden Objekte. Der Quader wird aufgespannt durch zwei Punkte mit jeweils den drei Koordinaten x, y, z im dreidimensionalen Raum und einer Zeitachse. Die dritte Dimension wurde hinzugefügt, um auch Informationssysteme bei mehrstöckigen Häusern zu ermöglichen. Die Zeitachse könnte entweder als echte Zeitachse, bei z.B. Wetterkarten dienen oder als symbolische Zeitachse für eine Versionskontrolle, etwa bei Bebauungsplänen.

Es wurde als geometrische Form der Quader gewählt, da dieser sich einfach in eine zweidimensionale, rechteckige Darstellung konvertieren läßt. Diese Form haben die meisten Anzeigen. Sollten sich diese Anforderungen durch neue Entwicklungen bei den Benutzungsoberflächen (Stichwort: *skins*) ändern, so könnte eine Protokollerweiterung anstelle eines Rechteckes auch andere geometrische Objekte vorsehen, etwa einen Kreis für den Radarschirm eines Fluglotsen.

Vergrößerung In diesem Feld wird die gewünschte Vergößerung des Ausschnittes mitgeteilt. Dieser Wert kann von der Geodatenkomponente bei einigen Aktionen vernachlässigt werden, etwa bei der Aktion „Vollbild“.

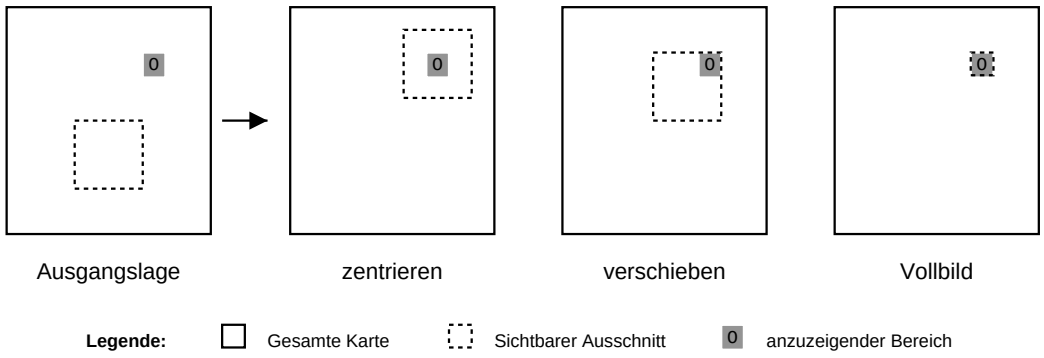

Abbildung 3.8: Auswirkungen des Aktion-Feldes

Aktion Es gibt verschiedenene Möglichkeiten wie die Positionsänderung zu einem neuen Quader auf dem Bildschirm dargestellt wird (s. Abbildung 3.8):

1. Zentrieren: Der Quader wird in der übergebenen Vergrößerung in der Mitte des Darstellungsbereichs dargestellt. Dies führt unter Umständen zu einem starken „springen“ auf der Karte bei der Navigation, welche vom Benutzer eine Neuorientierung verlangt. Abhilfe könnte die folgende Aktion bringen.
2. Verschieben: Der Quader wird mit der übergegebenen Auflösung in den Sichtbereich der Darstellung gebracht. Befand sich der Quader schon darin, wird keine Aktion durchgeführt, andernfalls wird der sichtbare Bereich soweit verschoben, daß der Quader sichtbar ist.
3. Vollbild: Die übergebene Vergrößerung wird ignoriert und der Quader füllt den gesamten sichtbaren Bereich der Darstellung aus.

Zeige Ausschnitt (Sekundärschlüssel)

Die Geodatenkomponente zeigt das von der Sachdatenkomponente gewünschte Objekt an.

Sekundärschlüssel Dieser Sekundärschlüssel bestimmt das anzuzeigenene Objekt. Die dazugehörigen Koordinaten ermittelt das GIS aus seinem eigenen Datenbestand und verfährt danach wie bei der Koordinatennachricht.

Vergrößerung In diesem Feld wird die gewünschte Vergößerung des Ausschnittes mitgeteilt. Dieser Wert kann von der Geodatenkomponente bei einigen Aktion vernachlässigt werden, etwa bei der Aktion „Vollbild“.

Aktion Hier wird die Vorgehensweise bestimmt, wie zu einem Objekt auf dem Bildschirm gegangen wird. Es gibt die gleichen Möglichkeiten wie bei der Koordinatennachricht, nur werden sich die Aktionen auf das Objekt und nicht auf den Quader beziehen.

In der Rückrichtung sollten keine Koordinaten von der Geodatenkomponente an die Sachdatenkomponente geschickt werden. Es sind eine Vielzahl von Koordinaten möglich und die Sachdatenkomponente hat nur zu verhältnismäßig wenigen von diesen Informationen. Auch fehlen den meisten DBMS die geometrischenen Operationen, etwa ob ein Punkt sich innerhalb einer Fläche befindet. Die Sachdatenkomponente sollte also nur Schlüssel aus ihrer Datenquelle erhalten. Dies setzt auf der Geodatenseite ein GIS-Paket voraus, welches die Geodaten mit Attributen anreichern kann sowie eine Darstellungskomponente, welche die Benutzeraktionen (etwa einen Mausklick) bestimmten Objekten zuordnen kann. Das Anzeigemodul eines reinen Kartenservers wäre ungeeignet, oder müßte um diese Funktionalität erweitert werden. Es sind auch Geodatenkomponenten denkbar, welche keine Objektauswahlnachrichten verschicken, weil die Karte nur eine rein passive Anzeige ist.

Zeige Objekt

Mit dieser Nachricht bittet die Geodatenkomponente die Sachdatenkomponente die dazugehörigen Daten zu diesem Objekt anzuzeigen.

Sekundärschlüssel Der Sekundärschlüssel des anzuzeigenden Objektes.

In diesem Abschnitt wurde bei den Beschreibungen der Nachrichten davon ausgegangen, daß Sach- mit Geodatenkomponenten miteinander verknüpft werden. Ebenso wäre auch die Verknüpfung von gleichartigen Komponentenklassen untereinander denkbar. Eine solche Verknüpfung kann durch die Anmeldung der gleichartigen Komponenten als Beobachter erreicht werden. Wenn beispielsweise eine Sachdatenkomponente ihrem Beobachter die

Änderung des Sekundärschlüssels meldet, so wird dieser allen angemeldeten Zuhörern dies mit einer entsprechenden Nachricht mitteilen. Die Nachricht wäre im Falle einer Geodatenkomponente „ZeigeAusschnitt (Sekundärschlüssel)“, bei einer SachdatenKomponente „ZeigeObjekt“. Auf diese Weise könnte dann im Anwendungsbeispiel Personalverwaltung ein Dialog erzeugt werden, wo neben den Stammdaten des Mitarbeiters (Sachdatenkomponente A) auch die dazugehörigen Projekte aufgelistet werden (Sachdatenkomponente B). Daneben wird in einer Geodatenkomponente sein Büro angezeigt. Verknüpft werden diese drei Komponenten über den Sekundärschlüssel Personalnummer.

3.4 Komponentengenerator

Aufgrund der Vielzahl von möglichen Kombinationen von DBMS und ihren konkreten Schemata der enthaltenen Daten, sollen die Sachdatenkomponenten mit Hilfe eines Generators erzeugt werden. Die Entwicklung aber auch die spätere Konfiguration einer Komponente, welche alle Möglichkeiten nur mit Hilfe der Eigenschaften abzudecken versucht, würde einen unverhältnismäßig hohen Aufwand bedeutet. Auch wäre eine solche Komponente durch die Aufnahme von Programmteilen für jeden auch nur denkbaren Spezialfall allein durch ihre daraus resultierende Größe zu ineffizient. Stattdessen sollen hier mit Hilfe eines Generator Komponenten erzeugt werden, bei denen alle wesentlichen Entscheidungen schon bei der Erzeugung getroffen werden. Können Entscheidungen auch zur Laufzeit zu einem vertretbaren Aufwand getroffen werden, so sollten sie auf diesen Zeitpunkt verschoben werden, um die Flexibilität der Komponente zu erhöhen.

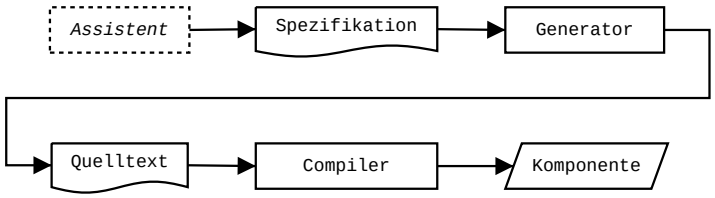

Abbildung 3.9: Prozess zur Erzeugung von Komponenten

Die Erzeugung einer Komponente, wie sie in Abbildung 3.9 dargestellt wird, beginnt mit der Erzeugung der Spezifikation einer Komponente. Diese Spezifikation kann „von Hand“ oder mit Hilfe eines Assistenten erzeugt werden. Ein Assistent, auch neudeutsch *wizard* genannt, erfragt mit einer

ansprechenden Benutzungsoberfläche[3] die Entwurfsentscheidungen. Anhand der in der Spezifikation getroffenen Entwurfsentscheidungen erzeugt daraufhin der Generator einen Quelltext für eine Komponente. Die Erzeugung von Komponentenquelltexten steht eigentlich im Widerspruch zum Black-Box-Komponentengedanken, aber sie bietet die folgenden Vorteile:

- Die Spezifikation muß selber nicht jeden Spezialfall abdecken. Sind unvorhersehbare Änderungen nötig, können sie noch nach der Generierung durchgeführt werden. Wäre dies nicht möglich, könnte gar keine Komponente erzeugt werden, oder eine Komponentenerzeugung wäre erst nach der Anpassung des Komponentengenerators möglich.

- Bei grafischen Komponenten würde zur Erstellung der Spezifikation auch das Design des Benutzungsdialogs gehören. Die Erstellung eines solchen ausreichend mächtigen Dialogeditors würde den Rahmen dieser Diplomarbeit sprengen. Liegt der Quelltext der Komponente vor, kann der Entwickler seinen gewohnten Dialogeditor verwenden, was eventuell Reibungsverluste durch den Wechsel wegfallen läßt.

- Formal hält sich dieser Entwicklungsprozess an die im Abschnitt 2.4 vorgestellen Komponentenanforderungen von Ritter (2000), da der Quelltext von einem Entwickler nach einer eventuell Nachbearbeitung am Ende doch als Black-Box ausgeliefert wird.

Es treten allerdings die folgenden Nachteile auf:

- Durch die persistente Erzeugung von Zwischenschritten tritt ein erhöhter Ressourcenbedarf auf.

- Für die Verwendung des Generator ist ein erhöhter Lernaufwand nötig, da der Entwickler die verschiedenen Eingriffsmöglichkeiten und ihre Auswirkungen kennen muß.

Das Generieren von Komponenten mit Zwischenschritten ermöglicht somit einerseits die schnelle Erzeugung von Prototypen, wenn auf die Bearbeitung der Zwischenschritte verzichtet wird. Mit diesen Prototypen kann dann daß prinzipielle Funktionieren einer Anwendung getestet werden, ohne den Aufwand für das Design einer Oberfläche schon erbracht zu haben. Andererseits können die Komponenten noch verfeinert werden, ohne Änderungen am Generator selbst vornehmen zu müssen. So kann der Entwickler für jede Aufgabe das passende Werkzeug verwenden. Diese beiden Vorteile

[3] Im Vergleich zu dem formalen Spezifikationsformat.

sollten den erhöhten Lernaufwand ausgleichen, zumal der Entwickler die Formate der erzeugten Zwischenschritte aus seiner täglichen Arbeit kennt. Der erhöhte Ressourcenbedarf sollte sich bei Rechnern für die Softwareentwicklung nicht weiter bemerkbar machen.

3.4.1 Spezifikation

Der zu erstellende Generator wird Komponenten für das typische Anwendungsszenario erzeugen. Die dortigen Sachdatenkomponenten haben eine externe Datenquelle in Form einer Datenbank und sind mit Geodatenkomponenten verknüpft. Auf die Möglichkeit zur Erzeugung von anderen Sachdatenkomponenten, welche beispielsweise ihre Koordinaten aus einer anderen Quelle beziehen, wurde verzichtet. Der zusätzliche Aufwand steht nicht im Verhältnis zu der erwarteten Nutzungshäufigkeit. Solche „Spezialfälle“ werden dann manuell zu entwickeln sein. Eventuell liefert der Generator dem Entwickler einen hilfreichen Rahmen, in dem dann die Datenbankfunktionalität entfernt werden muß.

In einer Spezifikation für den Generator zur Erzeugung einer Sachdatenkomponente mit Datenbank-Anbindung werden zumindest die folgenden Informationen benötigt:

Name Um die erzeugte Komponente von anderen zu unterscheiden, braucht sie einen Namen. Dieser sollte eindeutig sein, zumindest in einem Anwendungssystem. Es handelt sich dabei um den Namen der Komponentenklasse. Die jeweiligen Namen der davon angelegten Instanzen legt der Anwendungsentwickler fest.

Datenbank Aus dieser Datenbank bezieht die Sachdatenkomponente zur Laufzeit ihre Daten. Der Generator braucht diese Information, um zu der Datenbank eine Schnittstelle zu erzeugen.

Tabelle Aus dieser Tabelle werden die Daten angezeigt. Hierbei muß es sich nicht um eine reine Tabelle im Sinne einer Datenbank handeln, die Tabelle kann auch das Ergebnis einer Abfrage sein, welche von der Datenbank zur Laufzeit erzeugt wird.

Koordinate bzw. Sekundärschlüssel Aus dieser Spalte der Tabelle sollen die Koordinaten des Quaders bzw. Sekundärschlüssel ermittelt werden. Mit Hilfe dieses Eintrages kann die Komponente mit anderen Komponenten verknüpft werden.

Diese Informationen sollten in einem plattformunabängigen Format abgespeichert werden, um die Portabilität sicherzustellen. Eine mögliche Lösung

wäre eine Textdatei. Auf diese Weise könnten Spezifikationen einfach von Hand eingegeben oder verändert werden.

3.4.2 Assistent

Um dem Entwickler bei der Aufstellung der Spezifikationsdaten zu unterstützen, wurde der Einsatz eines Assistenten vorgeschlagen. Der Assistent sammelt schrittweise, durch gezieltes Nachfragen beim Entwickler, die zur Erstellung nötigen Informationen. Bei diesen Nachfragen bietet er ihm, wann immer möglich, eine Liste mit den möglichen Antworten zur Auswahl an. In einigen Fällen ist dies nicht sinnvoll, etwa bei der Frage nach dem Paßwort. Der Assistent kann eventuell schon während der Erstellung Randbedingungen prüfen, und Rückmeldungen an den Entwickler geben. Das Überprüfen von Randbedingen kann aber auch ein Nachteil sein, etwa wenn die Randbedingungen während der Erstellung der Spezifikation sich von den Randbedingungen während der Ausführung unterscheiden. Dieser Fall tritt auf, wenn auf Datenquellen aus Sicherheitsgründen nur am späteren Arbeitsplatz des Benutzers, und nicht am Arbeitsplatz des Entwicklers zugegriffen werden kann. Abhilfe könnte der Einsatz von Datenquellen bringen, welche anonymisierte Daten enthalten, und deswegen gefahrlos verwendet werden können. Vorteilhaft hingegen ist, daß der Assistent, sofern korrekt implementiert, immer eine syntaktisch korrekte Spezifikation zurückliefert.

Der Assistent könnte dem Entwickler von weiteren Routinetätigkeiten entlasten, indem er als Endprodukt sofort die fertige Komponenten in Form eines Prototypen erzeugt. Dabei sollten ebenfalls die Zwischenschritte (Spezifikation und Quelltext) dem Entwickler zugänglich gemacht werden, so daß dieser selbst entscheiden kann, ob und wenn ja, an welcher Stelle noch Veränderungen zur Fertigstellung der Komponente durchzuführen sind.

Kapitel 4

Implementierung von Veges

Nach dem noch allgemein gehaltenen Entwurf sollen nun ausgewählte Aspekte der Implementierung des Frameworks vorgestellt werden. Begonnen wird mit der Begründung für die Wahl der Implementierungssprache Java, darauf folgen die Beschreibungen ausgewählter Klassen einzelnener Bausteine.

4.1 Wahl der Implementierungssprache

Die Implementierungssprache sollte die im folgenden Eigenschaften haben, um die Entwicklung und die Nutzung von komponentenbasierten Frameworks zu erleichtern.

- *Objektorientiert:* Die Sprache sollte objektorientiert sein, um die im Entwurf erwähnten Entwurfsmuster leichter realisieren zu können. Die Objektorientierung bietet zudem mit der Kapselung die Möglichkeit, Schnittstellen zu definieren. Diese können dann durch Vererbung erweitert werden.

- *Portabel:* Die Wahl einer portablen Sprache ermöglicht es, entwickelte Komponenten auf möglichst vielen Plattformen einzusetzen, ohne sie zuerst in einer für das jeweilige Zielsystem passenden Sprache neu zu implementieren. Eine portable Sprache erleichtert auch die Wartung, da sie wenig Annahmen über die Zielplattform enthält. Dadurch sind etwa auch nach einer Aktualisierung des Betriebssystems nur geringe bis gar keine Änderungen nötig.

- *Günstig:* Der Preis für den Kauf der verwendeten Sprachumgebung ist für das spätere Einsatzgebiet im betrieblichen Umfeld von nicht so

starkem Interesse. Allerdings können zu hohe Kosten eine weite Verbreitung behindern. Vermieden werden sollten nach Möglichkeit Laufzeitlizenzmodelle, bei denen für jede Kopie eines entwickelten Anwendungssystems Gebühren fällig werden. Dieses Lizenzmodell verhindert die Abgabe von kostenlosen Testmustern. Die Kosten für eine Entwicklerlizenz sind durch die einmalige Anschaffung eher zu vernachlässigen. Bei den Kosten sind aber auch die Hardware-Anforderungen der Sprache zu beachten.

- *Verbreitung:* Die jeweilige Erfahrung der Entwickler mit bestimmten Sprachen sollte berücksichtigt werden, da es bei jedem Erlernen einer Sprache Reibungsverluste gibt und mit der Erfahrung auch die Qualität des erzeugten Codes steigt. Insofern sollte gerade bei der Entwicklung von Frameworks auf die Verwendung einer stark verbreiteten Sprache geachtet werden.

Die beiden derzeit üblichen objektorientierten Sprachen für eine portable Programmierung sind C++ und Java. C++ ist eine objektorientierte Erweiterung der Sprache C. Die Sprache C hat eine weite Verbreitung gefunden, da in ihr der Kern des Unix-Betriebssystems geschrieben ist. Leider gibt es für C++ im Gegensatz zu C noch keinen ANSI-Standard, so daß bei einer Übersetzung der Quelltexte mit verschiedenen Compilern Probleme auftreten können. Die Übersetzung der Quelltexte ist für jede Plattform nötig, auf der das Programm eingesetzt werden soll.

Die Alternative Java ist eine von Sun entwickelte Sprache. Die Quelltexte von Java werden im Gegensatz zu denen von C++ nicht in direkt von der Maschine ausführbaren Anweisungen übersetzt, sondern in eine Zwischensprache, welche dann von einer virtuellen Maschine interpretiert wird. Der Vorteil dieses Vorgehens ist, daß nicht wie bei C++ ein maschinenabhängiges Programm ausgeliefert wird, sondern ein Programm, welches auf jedem Rechner ausgeführt werden kann, auf denen eine entsprechende virtuelle Maschine vorhanden ist. Es entfallen daher die Neuübersetzungen bei der Portierung für eine neue Plattform. Der Preis für diese Portabilität ist die Indirektion durch einen Umweg über die virtuelle Maschine. Diese Zwischenschicht erfordert weitere Ressourcen, wodurch die damit realisierten Programme eigentlich weniger effizient sein sollten. Aber Prechelt (1999) zeigt in seiner Studie, daß die Qualität der Programmierer für die Effizienz wichtiger sein kann als die Struktur der verwendeten Sprache. Er stellte Programmierern mit mehrjähriger Erfahrung eine Programmieraufgabe. Die Programmierer durften das Programm in der von ihnen bevorzugten Sprache abgeben. Der Zeitbedarf und der Ressourcenverbrauch wurde danach verglichen. Dabei zeigte

sich sowohl, daß in der Regel Java-Programme langsamer sind als C/C++-Programme, aber auch, daß die drei schnellsten Java-Programme doppelt so schnell waren, wie die C/C++-Programme im Median. Ein guter Programmierer kann demnach in gewissen Grenzen strukturbedingte Nachteile ausgleichen, und Programme in Java sind demnach nicht zwingend langsamer als in C/C++ entwickelte Programme.

Beide Sprachen sind mit ihren benötigten Werkzeugen kostenlos zu erhalten. Laufzeitlizenzen fallen nicht an. Neben den kostenlosen werden auch kostenpflichtige Werkzeuge und Bibliotheken angeboten, welche bestimmte zusätzliche Leistungen enthalten. Bei diesen können Laufzeitlizenzen anfallen. Tendenziell dürte der Ressourcenverbrauch bei Java höher sein, da zumindest eine virtuelle Maschine gestartet werden muß.

Obwohl ich mehr Erfahrung mit der Programmiersprache C++ habe, schätze ich den Wert der Portabilität von Java höher ein. Vor allem die Probleme bei den nötigen Neuübersetzungen bei Plattformwechseln oder bei Aktualisierungen des Betriebssystems sprechen gegen C++. Die Erzeugung von wirklich portablem Code erfordert Erfahrung mit einer Vielzahl von verschiedenen Compilern, ein Problem, welches bei Java in dieser Form nicht existiert. Die Sprache wurde von Sun entwickelt und standardisiert. Die virtuelle Maschine gibt es im Vergleich zu C++ zwar für weniger Systeme, aber für die wichtigsten Arbeitsplatzsysteme (Windows ab 95, Linux und Macintosh). Im Serverbereich könnte es von Vorteil sein, daß Sun einer der Marktführer ist.

Fazit

Zusammenfassend läßt sich sagen, daß beide Sprachen objektorientiert und der Einsatz entsprechender Entwicklungsumgebungen kostenlos sind. Für beide Sprachen sind Schnittstellen zum InterGIS vorhanden. C++ hat leichte Vorteile bei der Effizienz, dafür ist Java portabler. Portabilität ist für dieses Framework eine wichtige Eigenschaft, da die Zielplattform nicht spezifiziert wurde. Deshalb Veges-Framework wird in seiner prototypischen Form in Java implementiert.

Wahl der Sprachversion

Verwendet wird die Java-Version 1.2 (auch Java 2 genannt), da ab dieser Version unter anderem auch Swing standardmäßig enthalten ist. Swing erleichtert die Entwicklung von ansprechenden Benutzungsoberflächen und ersetzt den alten Standard AWT. Im Gegensatz zu anderen Sprachen werden bei Java neuere Versionen nicht zwangsläufig von der Mehrzahl der Entwickler

übernommen. Dies hat folgende Ursache: In Java 1.2 geschriebene Programme können leider nicht in einem aktuellen Browser ohne ein entsprechendes Java 1.2 PlugIn ausgeführt werden. Sollen Programme für einen Browser, so genannte „Applets“, entwickelt werden, soll der Benutzer üblicherweise diese Programme ohne Nachinstallieren verwenden können. Diese Applets werden nicht auf dem Rechner des Benutzers installiert, sondern bei jedem Start aus dem Netzwerk geladen. Mit dem VEGES-Framework sollen aber Anwendungen erstellt werden, die – im Gegensatz zu Applets – (noch) installiert werden. Bei diesem Installationsprozess kann dann unter anderem auch die Laufzeitumgebung der gewünschten Java-Version installiert werden.

Eine Rückportierung auf Version 1.1 wäre möglich, wenn auf Swing und das `Collection`-Framework verzichtet werden würde. Das Java Tutorial von Sun (Sun Microsystems 2000) enthält dafür jeweils eine Anleitung.

4.2 Framework-Kern

Der VEGES-Framework-Kern hat die beiden folgenden Hauptaufgaben: Er ist für das Laden der einzelnen Komponenten und für die Verwaltung einer Kommunikationsstruktur zwischen den Komponenten verantwortlich. Die Kommunikationsstruktur ist in der Regel statisch. Sie wird üblicherweise beim Starten der Anwendung initialisiert und bleibt danach in dieser Form bestehen. Neben den Kommunikationsverbindungen verwaltet der Framework-Kern auch die Ein- und Ausgabe-Ressourcen der Anwendung.

Die Kommunikation der einzelnen Komponenten erfolgt über eine direkte Schnittstelle, d.h. es werden Methoden ohne Umweg über eine Netzwerkverbindung in einem anderen Objekt aufgerufen. Eine indirekte Schnittstelle, also ein Methodenaufruf über das Netzwerk, kann bei Bedarf später noch realisiert werden. In diesem Falle würden Stellvertreter-Objekte die direkten Methodenaufrufe in Aufrufe zu ihren entfernten Objekten umsetzen. Wenn die Netzwerkverbindung über einen Socket erfolgt, können beide Objekte in unterschiedlichen Programmiersprachen entwickelt worden sein. Auf diese Option wurde bei der prototypischen Erstellung des Frameworks verzichtet, da alle Komponenten in Java entweder zur Verfügung stehen oder darin entwickelt werden.

Die direkten Schnittstellen werden anstelle von Objekten mit Java-Interfaces realisiert. Ein Java-Interface ist mit einer abstrakten Klasse zu vergleichen (Flanagan 1996). Allerdings sind bei einem Interface immer alle Methoden abstrakt und es können keine Attribute definiert werden. Gemeinsam ist beiden, daß von ihnen keine Instanzen angelegt und sie durch Ableiten erweitert werden können. Um ein Interface zu nutzen (zu *implementieren*),

muß eine Klasse definiert werden, welche die abstrakten Methoden des Interfaces konkretisiert. Die Methoden des Interfaces kommen zu den eventuell ererbten Methoden der Vorlage hinzu, womit eine Mehrfachvererbung nachgebildet werden kann. Klassen können mehrere Interfaces implementieren.

Interfaces haben einen entscheidenen Vorteil gegenüber der Nutzung von (abstrakten) Klassen: Um sie zu implementieren, braucht man seine eigene Klassenhierarchie nicht an eine andere Klassenhierarchie anzupassen. Interfaces ermöglichen zusätzlich eine Art Mehrfachvererbung.

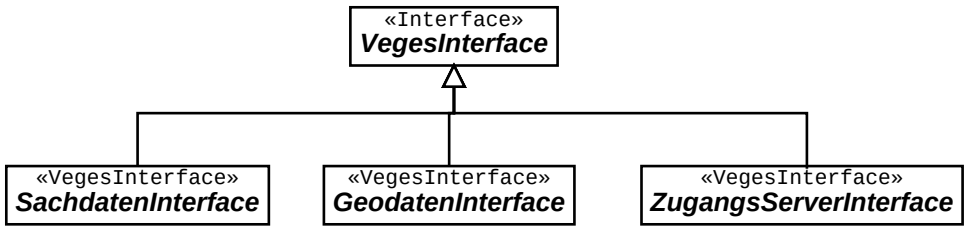

Abbildung 4.1: Hierarchie der VEGES-Interfaces

In Abbildung 4.1 ist die Hierarchie der erstellten Interfaces in Unified Modeling Language (UML)-Notation abgebildet. Eine kurze Übersicht über UML findet sich im Anhang A. Das `VegesInterface` übernimmt als Vorlage nur die Aufgabe der Versionskontrolle. Die anderen Interfaces werden im folgenden an geeigneter Stelle beschrieben.

4.2.1 Konfigurieren von Komponenten

Die Konfigurationen von Anwendungen werden in Java in der Klasse `Properties` (engl. für Eigenschaften) gespeichert. Die Daten dieser Klasse können dauerhaft in Form einer Textdatei gespeichert werden. Um das im 3. Kapitel auf Seite 39 beschriebene Verfahren eines Konfigurationsbaumes (s. Abbildung 3.5) zu realisieren, braucht eine Komponente vier `Properties`-Objekte für die vier Konfigurationen. Um die Konstruktor-Schnittstelle einer Komponente nicht unnötig zu vergößern und um die Auswertungslogik schon vom Framework-Kern aus zur Verfügung zu stellen, wurde die Klasse `Eigenschaften` implementiert (s. Abbildung 4.2).

Die Klasse `Eigenschaften` erhält bei der Erschaffung die vier Eigenschaftsdateien entweder als Dateinamen oder als schon geladene `Properties`.[1] Um Anfragen auszuwerten werden die Konfigurationen in der folgenden Reihenfolge durchsucht:

[1] Die Klasse `Eigenschaften` selbst kann eine beliebige Anzahl von `Properties`-Objekten oder Dateinamen aufnehmen.

1. Konfiguration der Komponenteninstanz des Benutzers
2. Konfiguration der Komponentenklasse des Benutzers
3. Standard-Konfiguration der Komponenteninstanz
4. Standard-Konfiguration der Komponentenklasse.

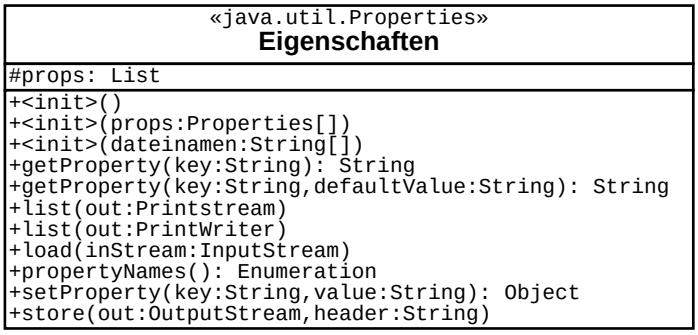

Abbildung 4.2: Klasse `Eigenschaften`

Sobald in einer Konfiguration die gewünschte Eigenschaft spezifiziert ist, wird diese verwendet. Ist in keiner der Konfigurationen diese Eigenschaft vorhanden, so kann ein vom Komponentenentwickler vorgegebener Wert verwendet werden. In den meisten Fällen dürfte es allerdings besser sein, die Komponente beim Fehlen jeglicher Konfiguration nicht zu starten.

4.2.2 Laden von Komponenten

Der Framework-Kern ist für das Laden der Komponenten zuständig. Eine Komponente im Sinne des VEGES-Frameworks ist eine Klasse, welche einen folgenden Konstruktor hat:

```
public void Klasse(Properties             properties,
                   ZugangsServerInterface zugangsserver,
                   AbstrakterBeobachter   beobachter);
```

Ein Konstruktor ist diejenige Methode einer Klasse, welche beim Anlegen einer Instanz aufgerufen wird. In Java kann es mehrere Konstruktoren einer Klasse geben. Mit Hilfe der Java Reflection API (Sun Microsystems 2000) kann zur Laufzeit die Menge der Konstruktoren einer ebenfalls zur Laufzeit eingebundenen Klasse ermittelt werden. Aufgrund dieser Funktionalität kann

auf die Verwendung einer abstrakten Basisklasse, von der konkrete Komponentenklassen abgeleitet werden müssen, verzichtet werden. Die Ersteller von VEGES-Komponenten brauchen ihre Komponenten daher nicht fest in die VEGES-Klassenhierarchie einbinden. Es entsteht nur eine „lose“ Verbindung, da die erstellten Komponenten VEGES-Interfaces implementieren und Objekte von VEGES-Klassen übergeben bekommen.

Nachdem die Komponente anhand ihres Namens zur Laufzeit geladen wurde (sie muß dazu im Klassenpfad zu finden sein), wird ebenfalls mit der Reflection API die Menge der implementierten Interfaces ermittelt. Je nach unterstützten Interface(s) können die geladene Klassen danach in die entsprechenden Benachrichtigungslisten des übergebenden Beobachters eingetragen werden.

Diese hohe Flexibilität hat aber auch ihren Preis. Das geringere Problem ist die höhere Ladezeit einer Komponente auf diesem Wege. Da alle Komponenten in der Regel beim Start der Anwendung geladen werden, tritt diese Verzögerung nur einmal auf. Zudem ist die Verzögerung im Vergleich zu den üblicherweise beim Start einer Komponente durchgeführten Aktionen, etwa dem Aufbau einer Verbindung zu einer externen Datenquelle, gering.

Ein weitaus größeres Problem besteht in der Stabilität der Anwendung. Die Reflection API führt zwar eine Typüberprüfung durch, aber es kann nicht sichergestellt werden, daß alle vom konkreten Konstruktor einer zu ladenen Klasse geworfenen Ausnahmen auch gefangen und bearbeitet werden.[2] Eine Ausnahme wird in Java `Exception` genannt. Bei dem Aufruf eines Konstruktors ohne die Reflection API wird das Fehlen einer Ausnahmebehandlungsroutine vom Java Compiler entdeckt, das Programm kann dann nicht erfolgreich übersetzt werden.

Das Problem mit unbekannten Ausnahmen besteht bei den Interfaces nicht, da deren abstrakte Methoden bei einer Erweiterung nicht um neue Ausnahmen ergänzt werden dürfen.

Konkret bedeutet dies für den Anwender, daß er mit Fehlermeldungen konfrontiert werden kann, welche bei der Erstellung des Frameworks unbekannt waren. Von diesen Fehlermeldungen wird lediglich der Text in einem Nachrichtenfenster dargestellt. Eine weitere Hilfestellung findet nicht statt.

Unterstützung erhält der Benutzer hingegen bei einer `ZugangsServer-Exception`, in der jeder Rückgabewert einer `ZugangsServerInterface`-Konstante als Text eingetragen werden kann. Die Umsetzung wird mit Hilfe der Bibliotheksklasse `ZugangsServerNachrichtenTexte` durchgeführt. Eine Ausnahme ist in Java auch die einzige Möglichkeit den Aufrufer eines Konstruktors detailliert über Erfolg oder Mißerfolg zu benachrichtigen, da

[2] Unbearbeitete Ausnahmen werden vom Laufzeitsystem auf der Konsole ausgegeben

Konstruktoren – im Gegensatz zu Methoden – keinen Rückgabewert liefern können. Da Komponenten einen Zugangsserver als Parameter bei ihrem Konstruktor übergeben bekommen und ihn üblicherweise zur Ermittlung ihrer Zugangsdaten verwenden, ist es wahrscheinlich, daß eine solche Ausnahme bei gescheiterten Anfragen geworfen wird. Wird der Benutzer gebeten, sich persönlich anzumelden, so wird dies mit Hilfe eines Login-Dialogs durchgeführt. Alle anderen Schwierigkeiten dem Benutzer werden in einem Nachrichtenfenster mitgeteilt.

Um Komponenten zu laden, sollte die Klasse `KomponentenLader` verwendet werden. Sie lädt und konfiguriert die gewünschte Komponente. Ähnlich wie beim erzeugenden Entwurfsmuster Fabrik bleibt dem Entwickler die Komplexität der Erzeugung verborgen. Der Entwickler übergibt dem `KomponentenLader` den Namen der Komponente zur Laufzeit. Dieser lädt dann die dazugehörige Klassen aus dem Klassenpfad und startet sie mit einer passenden Konfiguration, einem Zugangsserver und einem Beobachter. Je nach unterstützten Interface(s) der Komponente wird sie in die entsprechenden Beobachterlisten eingetragen. Abschließend wird die Komponente als Rückgabewert dem Aufrufer der `lade`-Methode zurückgeliefert.

Wird Swing als Benutzungsoberfläche verwendet, sollte dem `SwingKomponentenLader` der Vorzug gegeben werden. Dieser schreibt die eventuell auftretenden Fehlermeldungen nicht auf die Konsole, sondern teilt sie dem Benutzer mit Hilfe eines Nachrichtendialoges mit.

4.2.3 Beobachter

Der Beobachter stellt die Kommunikationszentrale zwischen den Komponenten im VEGES-Framework dar. Die einzelnenen Komponenten unterrichten ihn von einer Änderung ihres Zustandes, woraufhin der Beobachter die Benachrichtung aller bei ihm angemeldeten Komponenten vornimmt. Dabei unterscheidet er zwischen Geo- und Sachdatenkomponenten und verwendet die jeweils entsprechende Schnittstelle (`GeodatenInterface` bzw. `SachdatenInterface`). Die Klassenhierarchie der Beobachter-Klassen ist in Abbildung 4.3 dargestellt. Die Aufgabe der Kommunikationszentrale wird nur von der Klasse `VerteilenderBeobachter` übernommen. Die anderen Klassen sind Hilfsklassen für Testzwecke oder zur Fehlersuche.

AbstrakterBeobachter

Diese Klasse ist die Vorlage für alle anderen Beobachter im VEGES-Framework.

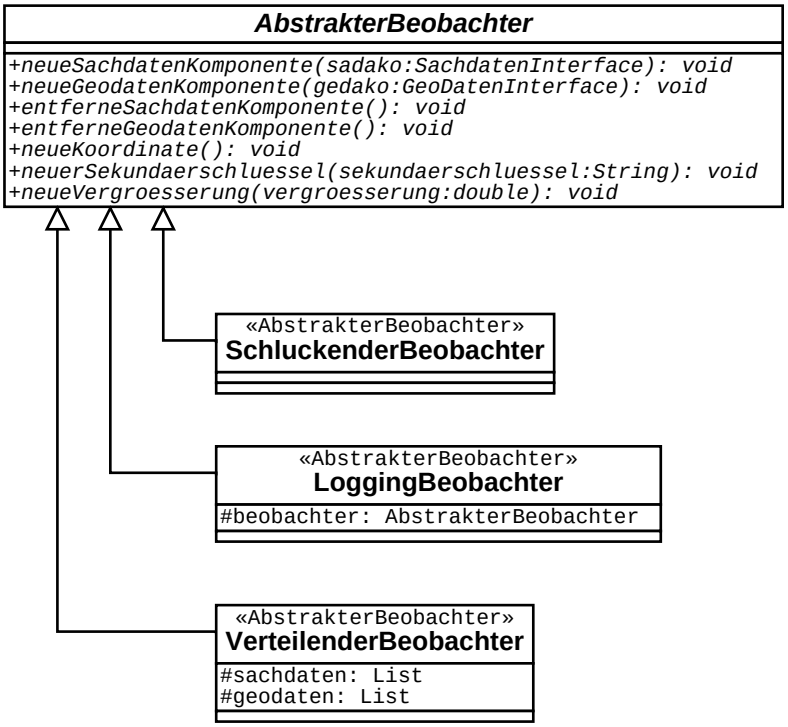

Abbildung 4.3: Beobachter-Klassenhierarchie

SchluckenderBeobachter

Die Klasse `SchluckenderBeobachter` „verschluckt“ alle eingehende Änderungen, d.h. keine Änderung führt zu einer Reaktion. Er kann dort eingesetzt werden, wo kein Beobachter benötigt wird, aber ein Beobachter als Parameter übergeben werden muß.

LoggingBeobachter

Der `LoggingBeobachter` ist ein Proxy oder Stellvertreter für einen anderen Beobachter. Jeder Aufruf wird protokolliert und danach an den anderen Beobachter weitergeleitet. Er erleichtert die Fehlersuche.

VerteilenderBeobachter

Diese Klasse stellt die gewünschte Kommunikationszentrale zur Verfügung, welche alle Zustandsänderungen der einzelnen Komponenten allen registrierten Komponenten mitteilt. Ihre Funktionalität entspricht damit im Prinzip dem Entwurfsmuster Beobachter (s. Abbildung 2.1 auf Seite 8), nur daß in

diesem Falle eine weitere Zwischeninstanz eingefügt wurde. In diesem speziellen Fall sind alle beteiligten Kompenenten zugleich Empfänger (Objekt) und Sender (Subjekt) der Zustandsänderungen. Beim Beobachtermuster gibt es nur einen Sender bei dem sich alle Empfänger anmelden. Eine Übernahme dieser Struktur würde zu einem dezentralen Aufbau führen (vgl. die Diskussion im Unterabschnitt 3.2.2). Vor der Verwendung von Entwurfsmustern bei Systemen mit verteilten Objekten sollte beachtet werden, daß Entwurfsmuster ihren Ursprung als Mikroarchitekturen für die Kommunikation von Objekten über direkte Schnittstellen und nicht in der Nutzung von verteilten Objekten über unsichere Verbindungen haben.

4.3 Zugangsserver

Die Komponenten ermitteln mit Hilfe eines Zugangsservers die Zugangsdaten zu ihren Datenquellen. Der Framework-Kern verbindet sich mit einem Zugangsserver und leitet alle Anfragen an diesen weiter. Das VEGES-Framework stellt einige Zugangsserver zur Verfügung. Die Klassenhierarchie dieser Zugangsserver ist in Abbildung 4.4 dargestellt.

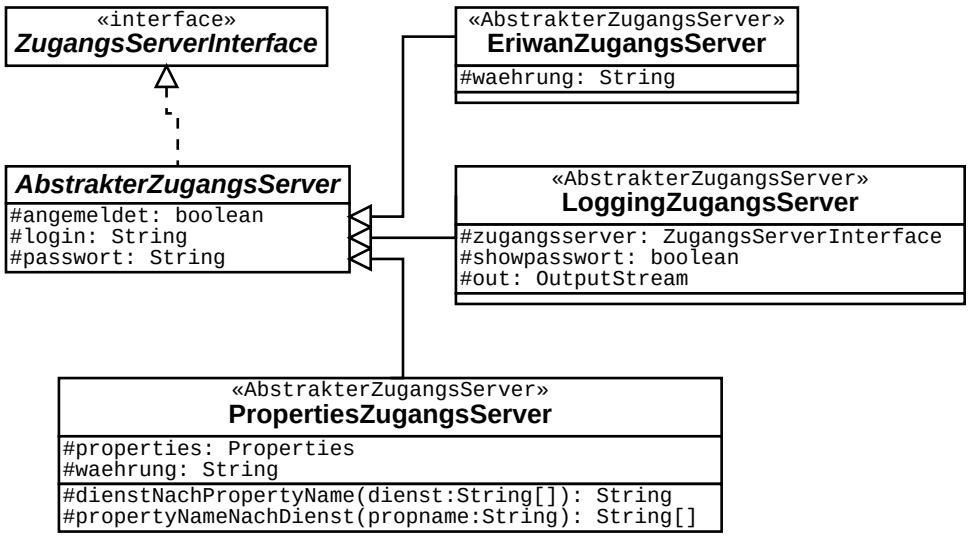

Abbildung 4.4: Zugangsserver-Klassenhierarchie

In Abbildung 4.5 ist eine Detailansicht der Schnittstelle zu finden. Die Gründe für eine Erweiterung gegenüber der Entwurfsfassung werden im Unterabschnitt 4.7.2 erläutert.

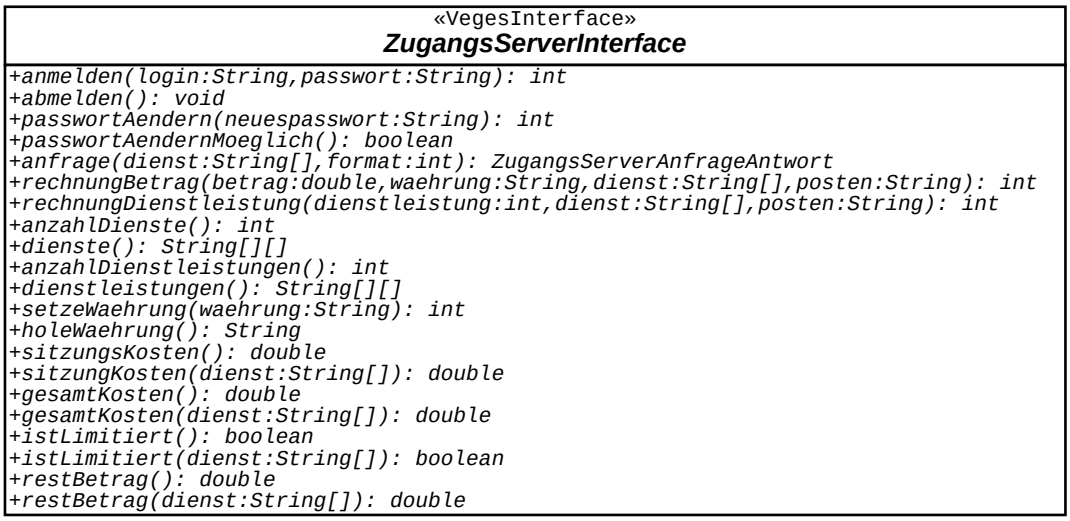

Abbildung 4.5: Klasse `ZugangsServerInterface`

AbstrakterZugangsServer

Diese Klasse ist die Vorlage für alle anderen Zugangsserver im VEGES-Framework.

LoggingZugangsServer

Der `LoggingZugangsServer` ist für Testzwecke gedacht. Er protokolliert jeden Aufruf im `ZugangsServerInterface` (s. Abbildung 4.5 auf Seite 67) und reicht ihn dann an einen anderen Zugangsserver weiter. Es handelt sich um einen Stellvertreter oder Proxy.

Außer zu Testzwecken oder zur Fehlersuche läßt sich der `LoggingZugangsServer` auch zur Abrechnung verwenden, da jede Rechnung im Protokoll vermerkt wird. Allerdings sind bei dieser Lösung keine Abfragen zum Kontostand möglich, da die Protokolldateien extern ausgewertet werden müssen.

EriwanZugangsServer

Der `EriwanZugangsServer` ist ein Zugangsserver für Testzwecke und kleine Anwendungen. Seinen Namen hat er vom sprichwörtlichen Sender Eriwan, der jede Antwort auf eine Höreranfrage mit den Worten begann: „Im Prinzip ja, aber ...“. Analog zu seinem Namensgeber gestattet dieser Zugangsserver den Zugang zu jeder gewünschten Datenquelle („Im Prinzip ja“), allerdings

muß sich der Benutzer bei jeder Datenquelle persönlich anmelden („aber“). Aufgrund der prinzipiellen Offenheit für jeden Benutzer gibt es keine Kontoverwaltung, d.h. eine Abrechung findet nicht statt.

Da er nicht installiert werden muß, eignet er sich besonders zum Testen von einzelnen Komponenten. Kleinere Anwendungen mit zwei, höchstens drei Komponenten lassen sich ebenfalls realisieren, solange keine Abrechnung benötigt wird. Bei einer größeren Komponentenanzahl sollte aber entweder auf einen netzwerkbasierten Zugangsserver (etwa den später vorgestellten SSO-Server vom InterGIS) oder auf den `PropertiesZugangsServer` ausgewichen werden.

PropertiesZugangsServer

Für kleinere Anwendungen oder zum Testen ohne Verbindung zu einem netzwerkbasierten Zugangsserver eignet sich der `PropertiesZugangsServer`. Dieser bezieht seine Zugangsdaten aus der Java-Klasse `Properties`. Durch die Speicherung der Zugangsdaten in einer Datei können diese von mehreren Benutzern verwendet werden. Auf diese Weise kann die Einrichtung eines netzwerkbasierten Zugangsservers eingespart werden. Allerdings entsteht ein erhöhter Administrationsaufwand, da Änderungen manuell eingetragen werden müssen, während bei netzwerkbasierten Zugangsservern in der Regel Werkzeuge die Administration erleichtern. Da der `PropertiesZugangsServer` die Zugangsdaten jedem zur Verfügung stellt, der die Properties-Datei lesen kann[3], findet keine Abrechnung statt.

Können in dem Einsatzgebiet Dateien über das Netzwerk verteilt werden (etwa über NFS) und ist die Zahl der Komponenten und der verschiedenen Benutzer nicht zu groß, so kann dieser Zugangsserver einen netzwerkbasierten Zugangsserver ersetzen. Bei Einzelarbeitsplätzen ohne Netzwerkzugang und einer höheren Komponentenanzahl kann er die einzige Lösung sein, um langwierige Anmeldungen bei den Datenquellen jeder einzelnen Komponente zu vermeiden.

[3] Auf UNIX-Systemen kann verhindert werden, daß jeder Benutzer die Zugangsdaten lesen kann, der auch die Anwendung verwenden darf. Dazu wird die Java-Anwendung mit Hilfe eines Skriptes gestartet, dessen `setuid`-Flag gesetzt ist. Auf diese Weise erhält die Anwendung die Rechte eines bestimmten Benutzers (beispielsweise `veges`) und nicht die des Benutzers, der die Anwendung gestartet hat. In diesem Beispiel sollte dann der Benutzer `veges` die Datei lesen können, alle anderen Benutzer aber nicht.

4.4 Komponentengenerator

Der Komponentengenerator erzeugt Sachdatenkomponenten gemäß einer Spezifiation. Dazu generiert er zuerst den Quelltext der Komponente und kopiert weitere Quelltexte (Interfaces und Hilfsklassen) des VEGES-Frameworks in das Unterverzeichnis `veges` hinzu. Danach werden diesen Quellen kompiliert und die Kompilate in einem Java Archiv (JAR) zusammengefaßt. Dieses JAR-Archiv ist ein Binärformat und kann damit als Komponente ausgeliefert werden. Um die Komponente zu testen, muß das JAR-Archiv noch in den Klassenpfad eingefügt werden.[4] Zur Erleichterung des Testens von Komponenten erzeugt der Generator eine kleine Testumgebung. Die Testumgebung wird mit folgenden Befehl gestartet:

```
java <Komponentenname>
```

So kann das korrekte Verhalten der Komponente ohne Anbindung an den Framework-Kern getestet werden. Auf diese Weise lassen sich einige Fehlerklassen ausschliessen, etwa Probleme beim Aufbau der Datenbankverbindung oder der Benutzungsoberfläche.

Um dem Entwickler die eventuell nötigen Anpassungen der erzeugten Komponente nach seinen Wünschen zu erleichtern, wird eine Dokumentation mit dem Java-Werkzeug `javadoc` im Unterverzeichnis `doc` automatisch aus den Quellen erstellt.

Ein Nachteil der vollständigen Einbindung der Interfaces und Hilfsklassen in das JAR-Archiv der Sachdatenkomponente ist die Redundanz zwischen verschiedenen erzeugten Sachdatenkomponenten. Alle enthalten die gleichen Kompilate der Klassen aus dem `veges`-Unterverzeichnis. Zur Vermeidung der Redundanz könnte der konstante Anteil einer Sachdatenkomponente entfernt und in einem weiteren JAR-Archiv ausgeliefert werden. Dann wären zum Starten einer Sachdatenkomponente zwei Archive im Klassenpfad notwendig.

JDBC

Ihre Daten bezieht die Sachdatenkomponente über das JDBCJDBC[5]-API von Java. Mit Hilfe dieser Schnittstelle können Datenbankoperationen oh-

[4] Laut Java-Dokumentation sollte auch die Nutzung des Parameters `classpath` möglich sein. Dieser ist aber zur Zeit fehlerhaft implementiert. Bei einer Verwendung wird die Umgebungsvariable `CLASSPATH` ignoriert, dort sind aber üblicherweise die verfügbaren JDBC-Treiber enthalten. Ein Fehlen der JDBC-Treiber verhindert dann eine Datenbankverbindung.

[5] JDBC ist keine Abkürzung, sondern ein Markenzeichen von Sun. In Anlehnung an die ODBC-Application Program Interface (API) von Microsoft wird JDBC auch mit „Java Database Connectivity“ übersetzt.

ne Kenntniss der konkreten Datenbank implementiert werden. Näheres zu dieser Schnittstelle findet sich in Horstmann und Cornell (2000). Die Anbindung an eine Datenbank erfolgt in der Regel durch einem vom Hersteller gelieferten Treiber, welcher im Klassenpfad zu finden sein muß. Leider ist diese Abstraktion von konkreten Datenbanken nicht vollständig gelungen. Dies liegt zum einem daran, daß Anfragen an die Datenbank in der Structured Query Language (SQL) gestellt werden. Es gibt zwar einen Standard, wie die Syntax von SQL-Anfragen sein muß, aber eine von JDBC unterstützte Datenbank braucht diesen Standard nicht einzuhalten. Da die SQL-Anfrage nicht vom Treiber in Datenbankoperationen umgesetzt wird, sondern direkt zur Datenbank durchgereicht wird, können Kleinigkeiten, wie etwa die Frage, ob ein abschließendes Semikolon am Ende der Abfrage vorhanden sein muß, die Portablilität in Frage stellen.

Ein weiteres Problem ist, daß ein Treiber nicht alle in der API definierten Methoden auch implementieren muß. Zwar gibt es eine Teilmenge von Methoden, welche für einen JDBC-Compliant-Treiber mindestens implementiert werden müssen, aber nicht jeder JDBC-Treiber muß JDBC-Compliant sein.[6] Zumindest läßt sich diese Eigenschaft vom Treiber abfragen, im Gegensatz zur Erweiterung der API, JDBC 2. Bei JDBC 2 ist der Entwickler auf „Versuch und Irrtum“ angewiesen.

Um diese Probleme für den Endanwender oder Einrichter einer Anwendung zu entschärfen, sollte eine Liste mit getesteten Kombinationen von Treibern und Datenbanken und den dabei auftretenen Problemen geführt werden. Eine Evaluation mit einer nennenswerten Anzahl von Datenbanken und Treiber zur Ermittlung der Informationen würde allerdings den Rahmen dieser Diplomarbeit sprengen.

4.4.1 Spezifiktionsdatei

Die Spezifikation der Sachdatenkomponenten wird in einer Datei mit der Endung `.generator.properties` gespeichert. Da Dateisysteme auf den gängigsten Betriebssystemen vorhanden sind, ermöglicht dieses Vorgehen den Austausch des Assistenten, da auf diese Weise eine Schnittstelle zu dem Komponentengenerator definiert wird. Folgende Parameter können spezifiziert werden (Verweise auf andere Parameter sind mit einen → gekennzeichnet):

Name Unter diesem Namen wird die Komponente erzeugt. Der Generator erzeugt ein Verzeichnis mit diesem Namen, generiert dort den Quelltext und kopiert dorthin alle weiteren benötigten Dateien.

[6] Beispielsweise implementiert die zweite Version des `mm.mysql`-Treibers große Teile der JDBC 2-API, ohne JDBC-Compliant zu sein.

DBMS Der Name des DBMS, also der Hersteller und der Produktname inklusive der Versionnummer.

Dienst Der gewünschte Dienst. Ein Dienst kann mit Hilfe von beliebig vielen Unterdiensten definiert werden. Wenn der Dienst aus dem reinen Uniform Resource Locator (URL) der Datenquelle besteht, dann muß der Dienst an oberster Stelle stehen. Der Generator schreibt die URL der Datenquelle als Dienst in die Spezifikation, eine Anpassung an eine evtl. vorhandene Dienstnamenkonvention muß dann noch bei der Installation der Komponente gemacht werden. Dieser Eintrag hat keinen Einfluß auf den →JDBCEnvironmentCheck. Der Eintrag braucht nicht der JDBC-Syntax zu entsprechen, solange der verwendete Zugangsserver eine Namensumsetzung durchführt.

Dieser Eintrag wird in die Eigenschaftsdatei der erzeugten Komponente kopiert und damit zur Laufzeit ausgewertet.

SQLQuery Mit dieser SQL-Abfrage wird die Ergebnismenge von der Datenbank abgefragt.

Hinweis: Die SQL-Anfragen werden von der JDBC-API durch den JDBC-Treiber als Text direkt an die darunterliegende Datenbank durchgereicht. Dieses Vorgehen hat einen entscheidenden Nachteil: Die Syntaxüberprüfung wird nicht von der JDBC-API vorgenommen, sondern von der Datenbank, was die Plattformunabhängigkeit gefährdet. So benötigen einige Datenbanken das Semikolon am Ende der Abfrage, während andere dann einen Syntaxfehler melden.

Sekundaerschluessel In dieser Spalte stehen die Daten, welche entweder als Koordinanten oder als Sekundärschlüssel interpretiert werden.

Diese Spalte muß Teil der Ergebnismenge der von →SQLQuery definierten Abfrage sein.

Die im Entwurf vorgestellte Fassung wurde noch um einige Einträge erweitert. Die zusätzlichen Parameter ergaben sich zum einen aus Zwängen der Entwicklungsumgebung (etwa die JDBC-spezifischen Parameter), zum anderen können so Erweiterungen zum Testen je nach Bedarf aktiviert werden.

Columns Die Sachdatenkomponente zeigt alle hier aufgelisteten Spalten in der Reihenfolge ihres Auftretens an. Der Standardwert ist '∗', also alle Felder der Ergebnismenge. Mit dieser Option wird die Mehrfachausgabe von Spalten ermöglicht. Die einzelnen Einträge werden durch Leerzeichen getrennt.

Alle Spalten müssen Teil der Ergebnismenge der von →SQLQuery definierten Abfrage sein.

Toolbar Mit dieser kann festgelegt werden, ob die Komponente mit einer Werkzeugleiste erzeugt werden soll, welche Navigationsmöglichkeiten auf der Ergebnismenge anbietet. Die meisten dieser Navigationsmöglichkeiten sind allerdings JDBC 2 spezifisch, und benötigen eine bildlauffähige Ergebnismenge der Abfrage. Sind diese Eigenschaften nicht gegeben, so kann die Navigation auf einer beim Benutzer zwischengespeicherten Ergebnismenge erfolgen.

AutoShowOnMove Wenn diese Option aktiviert wird, dann sendet die Sachdatenkomponente bei jedem Datensatzwechsel den neuen Sekundärschlüssel an den Beobachter.

AutoUpdateOnMove Wird diese Option aktiviert, werden Änderungen am angezeigten Datensatz beim Wechsel zu einem anderen Datensatz automatisch in den Datenbestand der Datenbank übernommen. Andernfalls werden die Änderungen verworfen. Diese Funktion benötigt einen JDBC 2 Treiber.

DefaultUser Standardmäßig meldet sich die Komponenten mit dem Login des Benutzers an, welcher sie gestartet hat. Auf Systemen ohne Benutzerverwaltung wird das Login '`anonymous`' verwendet. Sollte das Login aber für alle Benutzer gleich sein, so kann dieses mittels DefaultUser eingestellt werden.

Dieser Eintrag wird in die Eigenschaftsdatei der erzeugten Komponente kopiert und damit zur Laufzeit ausgewertet.

JDBCDriver Mit diesem JDBC-Treiber hat der Entwickler die Verbindung zur Datenbank aufgebaut. Diese Entwicklungsumgebung wird als Standard in die Eigenschaftsdatei der erzeugten Komponente kopiert, sowie bei aktivierten →JDBCEnvironmentCheck zum Vergleich mit der Einsatzumgebung verwendet.

JDBCEnvironmentCheck Um auf die eventuellen Plattformunterschiede zwischen den verschiedenen JDBC-kompatiblen Datenbanken reagieren zu können, kann mit diesem „Schalter“ eine Überprüfung aktiviert werden. Bei einem Unterschied zwischen dem bei der Entwicklung und dem beim Endbenutzer eingesetzten Datenbank/Treiber-Paar wird eine Warnung auf der Konsole ausgegeben.

DriverManagerLogging Ist dieser „Schalter“ aktiviert, so wird die JDBC-Verfolgungs-Funktionalität der JDBC-API in der Testumgebung verwendet.

Dieser Eintrag wird in die Eigenschaftsdatei der erzeugten Komponente kopiert und damit zur Laufzeit ausgewertet.

LoggingZugangsServer Diese Option bestimmt, ob die Testumgebung neben dem `EriwanZugangsServer` auch den `LoggingZugangsServer` enthält. Dieser erleichtert die Fehlersuche bei Kommunikationsproblemen mit einem Zugangsserver.

Author Hiermit läßt sich der Autor der Komponente ändern, bei dem es sich standardmässig um den Benutzer handelt, der den Komponentengenerator startete.

Used jdbc.drivers In den meisten Fällen wird der gleiche JDBC-Treiber des Entwicklers auch beim Benutzer eingesetzt, da es sich in der Regel um die gleiche Datenbank handelt. Dieser Treiber wird als Standard für die erzeugte Komponente verwendet.

Dieser Eintrag wird in die Eigenschaftsdatei der erzeugten Komponente unter dem Namen jdbc.drivers kopiert und zur Laufzeit ausgewertet.

Version Hier wird die Version des Formats der Spezifikationsdatei gesetzt. Der Generator kann auf diese Weise den Entwickler bei Inkompatibilitäten warnen.

4.5 Sachdatenkomponente

Im Anschluss an die Spezifikationsdatei soll nun der Aufbau der erzeugten Komponente vorgestellt werden. Die Hauptklasse der Komponente ist in die Java-Swing-Klassenhierarchie eingebunden. Ihre Vorlage ist das `JPanel`, ein sehr allgemeiner Container, welcher weitere Java-Swing-Komponenten aufnehmen kann. Als VEGES-Sachdatenkomponente implementiert sie das `SachdatenInterface`. In Abbildung 4.6 ist eine mit Toolbar erzeugte Komponente abgebildet.

Die in Abbildung 4.7 vorgestellte Klasse ist auf Erweiterung ausgelegt. Einige Methoden (etwa `onLeave` oder `doUpdate`) lagern Operationen aus und können vom Entwickler noch für seine Zwecke angepaßt werden. Diese Anpassung kann auch durch Ableiten einer neuen Klasse von der erzeugten Klasse geschehen, d.h. Änderungen müssen nicht in jedem Fall direkt

in dem erzeugten Quelltext vorgenommen werden. Die Klasse implementiert das Entwurfsmuster Schablone.

Durch den Einsatz von Java-`Action`s wird der Aufwand zur Anpassung der Benutzungoberfläche verringert. `Action`s ermöglichen es, Aktionen an verschiedene Auslöser zu binden. Bei der erzeugten Komponente sind die Aktionen an die Knöpfe der Werkzeugleiste gebunden, sie könnten aber auch in ein Menü integriert werden. Da `Action`s an einer Stelle definiert werden, sinkt bei mehreren Auslösern die Fehlerträchtigkeit. Ein weiterer Vorteil ist, daß alle Auslöser dadurch mit einem zentralen Aufruf deaktiviert werden können.

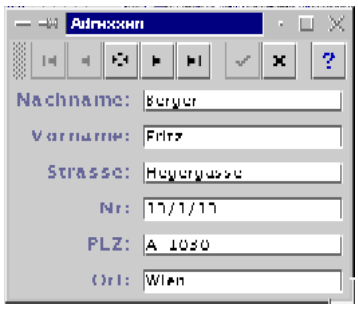

Abbildung 4.6: Bildschirmausschnitt einer generierten Sachdatenkomponente

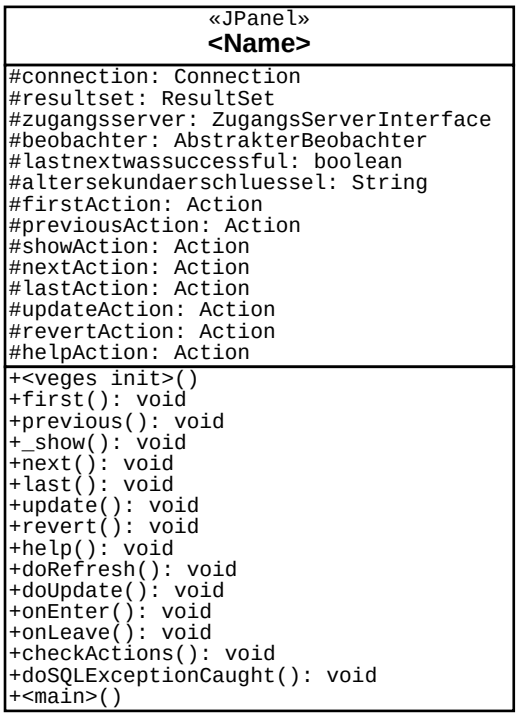

Abbildung 4.7: Klassenübersicht einer erzeugten Sachdatenkomponente

4.6 Assistent

Der Assistent soll dem Entwickler bei der Erstellung einer Spezifikationsdatei einer Sachdatenkomponente für den Komponentengenerator unterstützen. Dazu sammelt er mit Hilfe einer grafischen Benutzungsoberfläche schrittweise die Informationen für eine Spezifikation ein. Der Vorgang hat die folgenden Schritte:

1. Anmeldung bei der gewünschten Datenbank.

2. Anzeige der Produktinformationen der Datenbank und des verwendeten JDBC-Treibers.[7]

3. Hier wird die Tabelle gewählt, aus der die Sachdatenkomponente ihre Daten bezieht. Dies kann einerseits eine Datenbanktabelle oder eine vom Entwickler per SQL-Anweisung definierte Tabelle sein.

4. Der in einer Tabellenspalte enthaltene, zur Verknüpfung benötigte Sekundärschlüssel wird im nächsten Schritt ausgewählt.

5. Da eventuell nicht alle Spalten der Tabelle in der Sachdatenkomponente angezeigt werden sollen (etwa nicht benötigte weitere Sekundärschlüssel), kann die Menge der anzuzeigenden Spalten bestimmt werden.

6. Im letzten Schritt können noch Optionen zum äußeren Erscheinungbild (etwa Werkzeugleiste oder automatische Synchronisation der Daten) und die Integration von Hilfen zur Fehlersuche (beispielsweise eine erweiterte Ausgabe beim Laden der JDBC-Treiber) ausgewählt werden.

Danach werden diese Informationen in einer `.generator.properties`-Datei gespeichert und an den `KomponentenGenerator` übergeben, welcher dann die Erzeugung der Komponente übernimmt.

[7] Hier könnten Informationen über die „Verträglichkeit“ der Datenbank/Treiber-Kombination angezeigt werden. Da der Bildschirmplatz begrenzt ist, sollte diese Information auf Anfrage in einem weiteren Fenster erscheinen. Bei bekannten Unverträglichkeiten sollte das Dialogelement zum Aufruf der weiteren Informationen deutlich hervorgehoben sein. Möglich wäre auch, sich den den Hinweis mit einem Nachrichtenfenster bestätigen zu lassen.

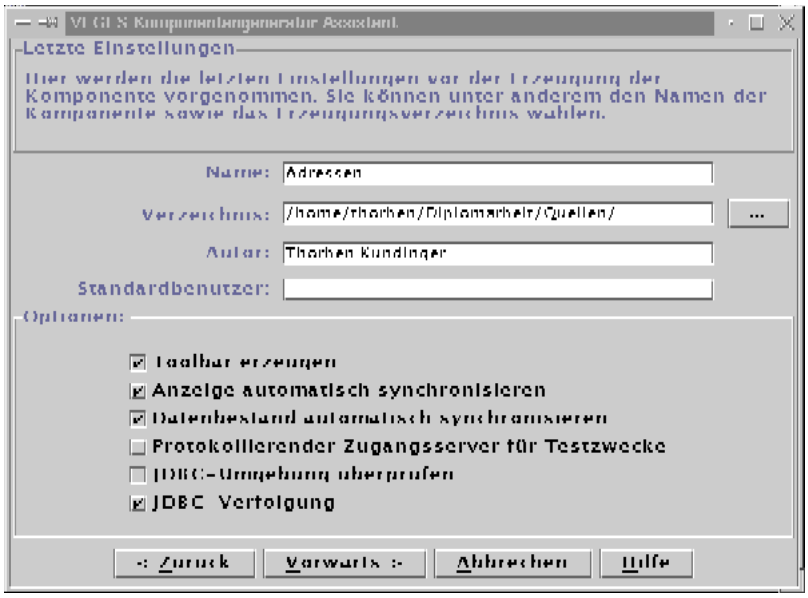

Abbildung 4.8: Bildschirmausschnitt des Assistenten

4.6.1 Sonderfälle

Mit dem Assistenten können nicht alle Sonderfälle bei der Erzeugung von Sachdatenkomponenten abgedeckt werden. Die Berücksichtigung von Sonderfällen hätte die Komplexität des Assistenten ohne konzeptionellen Gewinn vergrößert.

Quader

Der Assistent hat keine direkte Unterstützung, falls eine Geodatenkomponente keinen Sekundärschlüssel zur Anzeige von Objekten benötigt, sondern einen Quader. Eine solche Verknüpfung kann auf Umwegen auch mit Assisten erzeugt werden. Bei einer SQL-Abfrage können mehrere Tabellenspalten mit Hilfe der `concat`-Anweisung zu einer zusammengefaßt werden. Diese Spalte kann dann als Sekundärschlüssel verwendet werden. Als Spalten werden also die Koordinaten und ein Sekundärschlüssel zusammengefaßt. Dieser zusammengesetzte Sekundärschlüssel wird von der Geodatenkomponente an die Sachdatenkomponente bei einer Auswahl versandt. Die einzelnen Spalten werden durch ein Trennzeichen getrennt. Die Struktur der SQL-Abfrage würde dann wie folgt aussehen (MySQL 2000):

```
select concat(<Koordinate>, "#", <Sekundaerschluessel>)
      AS <neuer Name>  From <Tabellen>;
```

Nach der Erzeugung der Komponente muß noch der Quelltext angepaßt werden. Eine erzeugte Sachdatenkomponente teilt eine Zustandsänderung in der Methode `_show` mit. Hier kann mit Hilfe der Java-Klasse `StreamTokenizer` der zusammengesetzte Sekundärschlüssel wieder in seine Bestandteile zerlegt und danach die entsprechenden Methode des Beobachters aufgerufen werden.

4.6.2 Sekundärschlüsselformat

Unterscheidet sich das Format des gewünschten Sekundärschlüssels in der Datenbank von dem Format der restlichen Komponenten, kann dies eventuell mit Hilfe einer SQL-Abfrage angepaßt werden. Ist die notwendige Änderung des Formats zu komplex, bleibt noch die Anpassung in der `_show`-Methode.

4.7 InterGIS

Das InterGIS ist eine offene Geo-Server-Architektur zur Verwaltung und Bereitstellung raumbezogener Daten. Es soll beispielhaft für andere GIS mit dem VEGES-Framework verbunden werden. InterGIS enthält neben einer Darstellungskomponente für Geodaten auch einen Zugangsserver. Sie werden an die VEGES-Schnittstellen in Klassen im `veges.intergis`-Paket angepaßt. Die Sammlung in einem eigenen Java-Paket ermöglicht es später, die Anwendung auch ohne InterGIS-Fähigkeiten auszuliefern. Um auf InterGIS-Dienste zugreifen zu können, muß sich neben dem `veges.intergis`-Paket auch ein InterGIS-Client-Paket im Klassenpfad befinden.

4.7.1 InterGISGeodatenKomponente

Zur Darstellung der Geodaten wird im Rahmen dieser Diplomarbeit eine Komponente aus dem InterGIS-Paket verwendet (s. Abbildung 4.9).

Um eine Swing-fähige Komponente für die Beispielanwendungen zu erstellen, mußte auch diese Komponente in die Java-Swing Klassenhierarchie eingebunden werden. Um als VEGES-Geodatenkomponente erkannt zu werden, implementiert die `InterGISGeodatenKomponente` das `GeodatenInterface` und hat den VEGES-Komponenten-Konstruktor. Die InterGIS-Komponente enthält bisher keine Methoden, um die verschiedenen Möglichkeiten zur Ausschnittbewegung zu realisieren, welche in Abbildung 3.8 vorgestellt wurden. Zur Zeit ist nur die Aktion „Vollbild“ möglich.

Die Konfiguration dieser Komponente wird in einer `Properties`-Datei dauerhaft gespeichert. Folgende Parameter können konfiguriert werden:

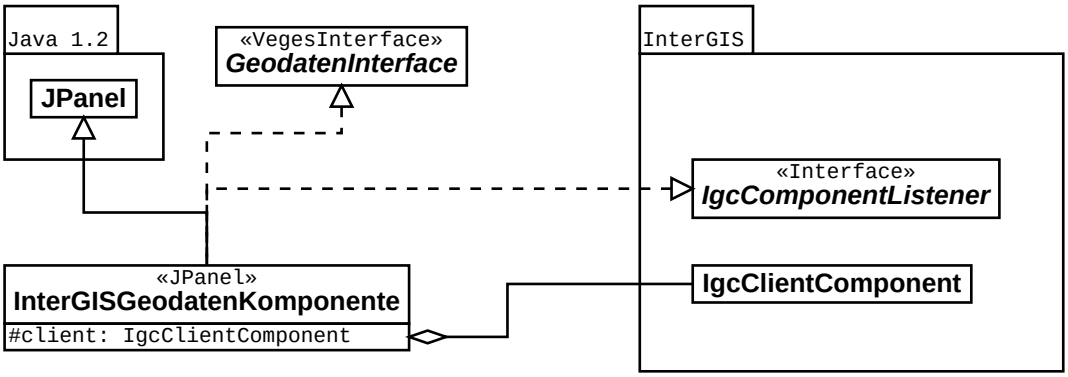

Abbildung 4.9: Klassenhierarchie der `InterGISGeodatenKomponente`

Dienst Die Zugangsdaten zu diesem Dienst werden vom Zugangsserver abgefragt.

Schema Die Geoinformationen können mit verschiedenen Farbschemata dargestellt werden. Dieser Parameter bestimmt das zur Darstellung verwendete Farbschema. Das Schema muß auf dem Server für den Benutzer zur Verfügung stehen.

BoundingBoxSize Dieser Wert bestimmt die maximale Größe eines darzustellenden Objektes. Der Standardwert sollte für die meisten Anwendungen ausreichend sein und nur mit Vorsicht vergrößert werden, da mit einem größeren Ausschnitt auch mehr Daten zum Benutzer übertragen werden.

Abfrage Diese Anfrage wird dem InterGIS-Server zur Ermittlung der Position eines Objektes anhand seines Sekundärschlüssels übermittelt. Eine Beschreibung dieser für GIS angepaßten SQL-Variante findet sich bei Friebe (2000).

AttributSchluessel InterGIS unterstützt attributierte Geo-Objekte. Wird auf der Darstellungskomponente ein Objekt ausgewählt (üblicherweise mit einem Doppelklick), so wird anhand dieses Schlüssels der Sekundärschlüssel ermittelt.

Die geografischen Daten werden beim InterGIS zweidimensional abgespeichert. Bei Quader-Anfragen bleiben daher die dritte Dimension z und die Zeitachse t unberücksichtigt.

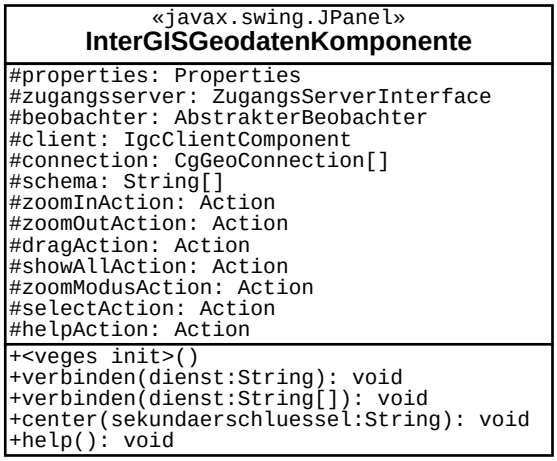

Abbildung 4.10: Klasse `InterGISGeodatenKomponente`

4.7.2 SSOZugangsServer

Der Single Sign On (SSO)-Server erfüllt die Aufgabe des Zugangsservers beim InterGIS. Das SSO-Protokoll liefert zu gewünschten Diensten die Zugangsdaten, es kennt bei den Anfragen keine Datenquellen. Es enthält die Funktionalität zur Abrechnung von Dienstleistungen (hier Produkte genannt).

Bei der Erstellung einer Wrapper-Klasse, welche Anfragen der Zugangsserver-Schnittstelle in Anfragen an einen SSO-Server umsetzt, traten die folgende Probleme auf:

- Beim SSO-Protokoll besteht der Name eines Dienstes aus zwei Ebenen: Dem Dienst und dem Unterdienst. Bei der Zugangsserver-Schnittstelle ist im Entwurf nur eine Ebene vorgesehen.
- Die Abrechnung erfolgt über Produkte anstelle von Beträgen.
- Neben dem Versenden von Rechnungen ist beim SSO-Protokoll auch die Ermittlung der angefallenen Kosten möglich. Auf Wunsch ist dies auch aufgeschlüsselt nach Diensten, aber nicht nach Unterdiensten möglich.
- Ein formales Abmelden ist beim SSO-Server nicht möglich.
- Das Ändern des Paßwortes für das Anmelden am SSO-Server ist möglich.

Bis auf die fehlende Möglichkeit sich abzumelden, haben alle Probleme ihre Ursache in der zu geringen Mächtigkeit der Zugangsserver-Schnittstelle.

Sie wird um die folgenden Punkte erweitert, um die obigen Probleme zu lösen:

- Die Datenquelle wird in Dienst umbenannt, da der Begriff Dienst allgemeiner ist als der Begriff Datenquelle. Ein Dienst wird nicht mehr durch eine Zeichenkette spezifiziert, sondern durch eine beliebig lange Liste von Zeichenketten. Damit sind dann „beliebige“ Hierarchieebene für Dienste möglich.

- Die Abrechnung anhand von Dienstleistungen oder Produkten wird eingefügt. Man kann eine Liste der möglichen Dienstleistungen erhalten, und Rechnungen können auch anhand von Dienstleistungen erstellt werden.

- Es werden Abfragen zur Ermittlung der angefallenen Kosten eingefügt. Dazu kommen Anfragen nach dem verbleibenden Kontostand, wenn Dienstleistungen im voraus bezahlt werden müssen oder über einen Etat abgerechnet werden.

- Das Ändern des Paßwortes wird generell möglich sein. Ob ein konkreter Zugangsserver diese Funktionalität zur Verfügung stellt, kann anhand einer Anfrage ermittelt werden.

Die Schnittstelle aus dem Entwurf wurde daraufhin erweitert.

Um den InterGIS-`NiceSsoServerClient` im VEGES-Framework nutzen zu können, muß eine Umsetzung von der InterGIS-Schnittstelle an das `ZugangsServerInterface` erfolgen. Dafür gibt es mindestens diese beiden Möglichkeiten:

1. Das `NiceSsoServerClient`-Objekt wird Teil einer Klasse, welche von der Klasse `AbstrakterZugangsServer` abgeleitet wurde.

2. Eine Unterklasse des `NiceSsoServerClient`s implementiert das `ZugangsServerInterface`.

Die erste Lösung hat eine Reihe von Vorteilen gegenüber der Zweiten:

- Es findet keine Einbindung von VEGES in die Klassenhierarchie des InterGIS statt. Bei jeder Erweiterung oder Änderung in den Klassen von denen der `NiceSsoServerClient` abgeleitet worden ist, müßte der `SSOZugangsServer` ebenfalls geändert werden. Dies würde zu einer zu starken Abhängigkeit führen.

- Die entstehende Klasse wird übersichtlicher. Wenn eine abgeleitete Klasse ein Interface implementiert, erhält es alle Methoden des Interfaces zusätzlich zu den Methoden der Oberklasse. Bei dieser Lösung würden in den Interface-Methoden dann die abgeleiteten Methoden aufgerufen.

In Abbildung 4.11 ist die realisierte Klassenhierarchie dargestellt.

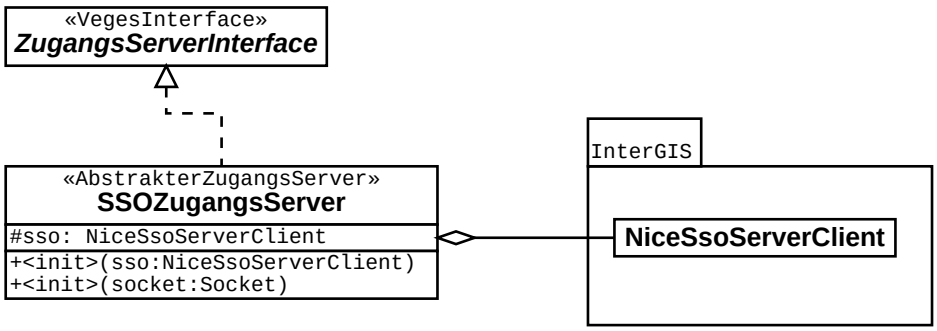

Abbildung 4.11: Klassenhierarchie des `SSOZugangsServers`

Hinweis: Die bisherige Implementierung des `NiceSsoServerClients` beherrscht noch nicht die verschlüsselte Kommunikation mit dem InterGIS-SSO-Server. Zur Nutzung des `SSOZugangsServers` ist daher ein SSO-Server mit deaktivierter Verschlüsselung notwendig.

4.8 Beispiele

Abschließend sollen noch drei Beispiele präsentiert werden, welche als Vorlage für reale Anwendungen dienen können. In den Beispielen werden vom Generator erzeugte Sachdatenkomponenten und die InterGIS-Geodatenkomponente verwendet.

4.8.1 SplitPaneDemo

Im `SplitPaneDemo` (s. Abbildung 4.12) teilen sich zwei Komponenten den Bildschirmplatz der Anwendung. Die Platzverteilung zwischen ihnen kann interaktiv mit Hilfe des sie trennenden Balkens verändert werden. Diese Anordnung eignet sich besonders für Anwendungen, in denen eine Sachdaten- mit einer Geodatenkomponente verknüpft werden sollen.

Abbildung 4.12: Bildschirmausschnitt des `SplitPaneDemo`s

4.8.2 TabbedPaneDemo

Beim `TabbedPaneDemo` (s. Abbildung 4.13) sind wieder zwei Komponenten sichtbar, allerdings stehen auf jeder Hälfte[8] mehrere Komponenten zur Auswahl. Die gewünschte Komponente wird durch einen der „Reiter“ ausgewählt. Diese Anordnung kann verwendet werden, wenn mehr als zwei Komponenten eingesetzt werden sollen, es aber ausreichend ist, daß zwei Komponenten sichtbar sind.

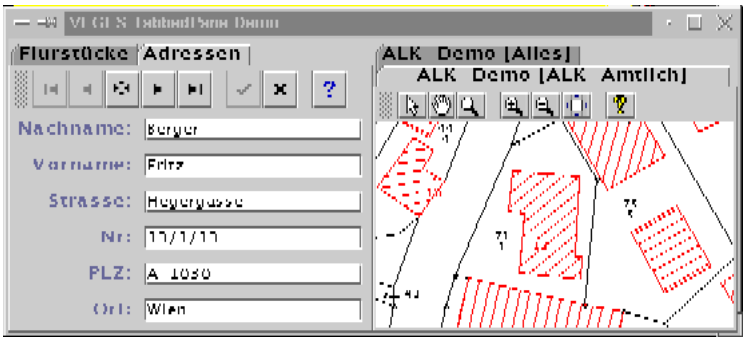

Abbildung 4.13: Bildschirmausschnitt des `TabbedPaneDemo`s

4.8.3 InternalFrameDemo

Das `InternalFrameDemo` ist ein Beispiel für eine Anwendung, bei der dem Benutzer eine große Freiheit über die Aufteilung seiner Arbeitsfläche überlassen wird. Die Anwendung ähnelt einer grafischen Benutzungoberfläche ei-

[8] Im Gegensatz zum `SplitPaneDemo` bestimmt hier der Entwickler die Verteilung zwischen den Komponenten, nicht der Benutzer.

nes „Betriebssystems im Kleinen“. Der Benutzer kann eine beliebige Anzahl Komponenten starten sowie ihre Position und Größe ändern. Komponenten können sich überlappen oder auf Symbolgröße verkleinert werden (s. Abbildung 4.14).

Diese Möglichkeiten eignen sich für variable Aufgaben, bei denen sich die einzusetzenden Komponenten und deren Platzbedarf zu schnell ändern, um wie in den anderen Beispielen fixiert zu werden.

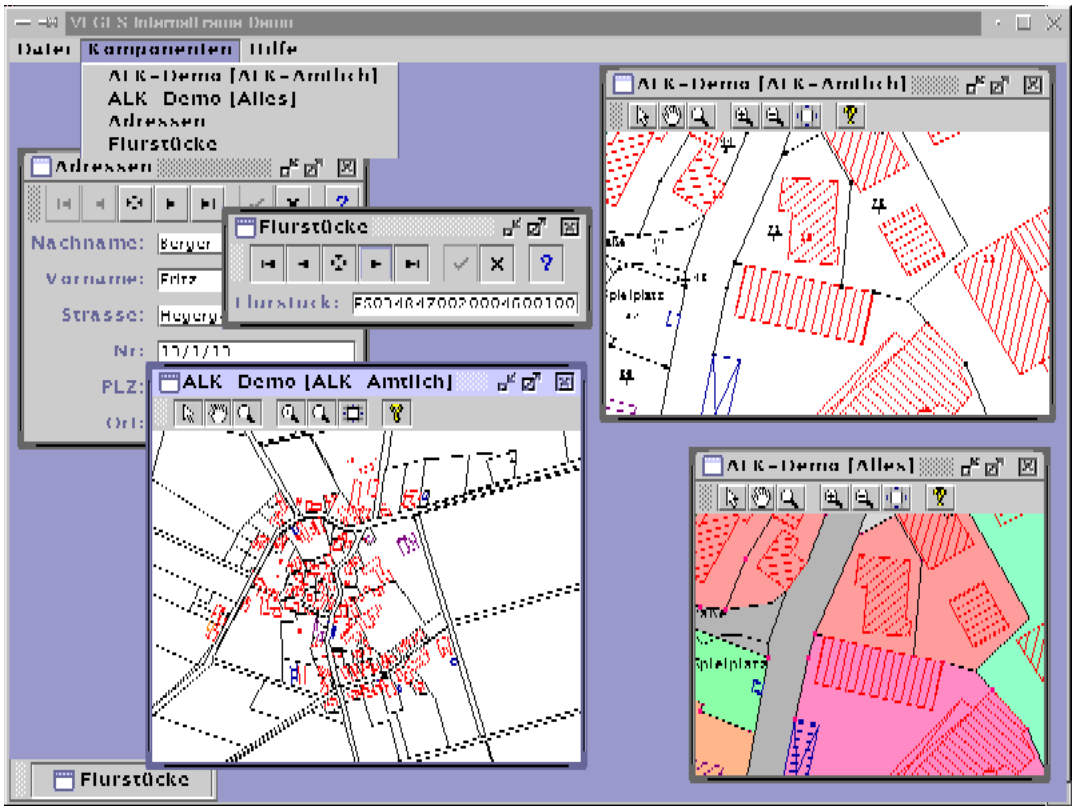

Abbildung 4.14: Bildschirmausschnitt des `InternalFrameDemos`

Kapitel 5

Zusammenfassung und Ausblick

Die vorliegenden Diplomarbeit stellt das komponentenbasierte Framework VEGES vor. Mit diesem Framework lassen sich Anwendungen zur Bearbeitung von verknüpften Geo- und Sachdaten erstellen. Mit Hilfe der Verknüpfung können die zusammengehörigen Daten auf Anforderung dargestellt werden.

Die Geodaten werden mit Komponenten des InterGIS dargestellt. Es erfolgt eine Umsetzung des VEGES-Protokolls auf die InterGIS-Schnittstelle.

Zur Bearbeitung von Sachdaten können von einem Generator erzeugte Komponenten eingesetzt werden. Dieser Generator wurde ebenfalls im Rahmen dieser Diplomarbeit entwickelt, um die Erzeugung von Sachdatenkomponenten zu vereinfachen. Zum Generator gehört ein Assistent, welcher den Entwickler bei der Erstellung der Spezifikation unterstützt.

Abschließend wurden drei Beispielanwendungen vorgestellt, welche als Vorlagen für reale Anwendungen dienen können.

Folgende Erweiterungen des VEGES-Frameworks bieten sich an:

- Im Komponentengenerator könnte eine Option hinzugefügt werden, um den Quelltext in einer Struktur zu erzeugen, welche von Dialogeditoren erkannt wird. Dies würde die Nachbearbeitung der Oberfläche der generierten Komponente erleichtern.

- Neben dem InterGIS könnten auch andere GIS in das Framework integriert werden. Auf diese Weise wäre das Framework in einer größeren Zahl von Umgebungen einsetzbar.

Um den Entwurf und die Implementierung des Frameworks zu testen,

sollten reale Anwendungen damit entwickelt werden. Die folgenden Punkte sollten bei diesem Praxistest beachtet werden:

- Wurde mit der Verwendung der JDBC-Schnittstelle ein ausreichend plattformunabhängiger Ansatz zur Datenbankverbindung gewählt wurde.
- Welche Schwierigkeiten treten bei der Kombination des VEGES-Frameworks mit anderen Frameworks auf.

Die dort gewonnenen Erkenntnisse würden dann in die nächste Version des VEGES-Frameworks einfließen.

Anhang A

Unified Modeling Language

Die Unified Modeling Language ist eine Sprache zur objektorientierten Analyse und Design (OMG 1999; Oestereich 1997). Es werden hier die in dieser Diplomarbeit verwendeten Notationen kurz vorgestellt, welche nur einen Ausschnitt der gesamten Möglichkeiten darstellen.

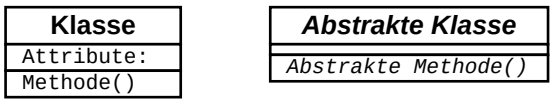

Abbildung A.1: Klassen und Schnittstellen

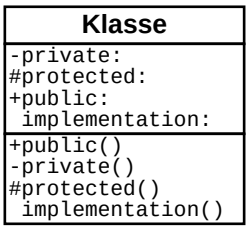

Abbildung A.2: Sichtbarkeit von Attributen und Methoden

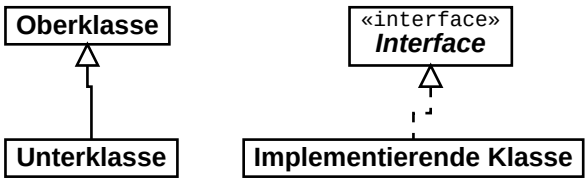

Abbildung A.3: Vererbung und Schnittstellenimplementation

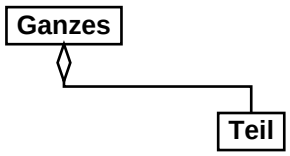

Abbildung A.4: Aggregation

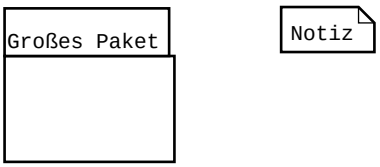

Abbildung A.5: Pakete und Notizen

Anhang B

Abkürzungsverzeichnis

Hilfreich bei der Erstellung dieses Abkürzungsverzeichnis war das Verzeichnis EDV-Relevanter Akronyme Heidelbach (2000).

ANSI American National Standard Institute

API Application Program Interface

AWT Abstract Window Toolkit (Sun, Java)

COM Component Object Model

CORBA Common Object Request Broker Architecture

CSCW Computer Supported Cooperative Work

DCOM Distributed COM

DBMS Datenbank Management System

DSL Domain Specific Language

FDL Framework Definition Languages

Gebos Genossenschaftliches Büro, Kommunikations- und Organisationsystem

GIS Geografisches Informationsystem

IBM International Bussiness Machines `http://www.ibm.com`

IC Integrated Circuit

ICCL Implementation Components Connection Language

IP Internet Protocol (RFC 793)

IS Informationssysteme

ISO International Standards Organisation

ISV Independent Software Vendors, unabhängige Softwarehersteller

IVBB Informationsverbund Bonn-Berlin

JAR Java Archiv (Sun, Java)

JDBC keine Abkürzung, sondern ein Markenzeichen (Sun, Java)

JDK Java Development Kit (Sun Java)

MFC Microsoft Foundation Classes

MVC Model-View-Controller

NFS Network File System (Sun, Unix)

ODBC Open DataBase Connectivity

OFFIS Oldenburger Forschungs- und Entwicklungsinstitut für Informatik-Werkzeuge und -Systeme

OLE Object Linking and Embedding

OMA Object Management Architecture

OMG Object Management Group

OO objektorientiert

OSI Open Systems Interconnection

RDBMS Relationales DBMS

RWG Rechenzentrale Würtembergischer Genossenschaften

SQL Structured Query Language (ISO 9075)

SSO Single Sign On (InterGIS)

TCP Transmission Control Protocol (RFC 793)

UML Unified Modeling Language

URL Uniform Resource Locator (RFC 1738)

VBX VisualBasic Controls

VEGES Verknüpfung von Geodaten mit Sachdaten

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

Anhang C

Glossar

Dieser Abschnitt beschreibt kurz die in der Arbeit verwendeten Fachbegriffe. Die Seitenangaben beziehen sich auf das erste relevante Auftreten im Text. Verweise auf andere im Glossar verwendete Begriffe sind →*kursiv* gesetzt. Als Grundlage für die Erstellung dieses Glossars dienten die Glossare aus Balzert (1996); Griffel (1998); Ritter (2000).

abstrakte Klasse
: Spielt eine wichtige Rolle in Vererbungsstrukturen (→*Vererbung*), wo sie die Gemeinsamkeiten einer Gruppe von ihr abgeleiteten →*Klassen* definiert. Im Gegensatz zu einer Klasse können von einer abstrakten Klasse keine Objekte erzeugt werden.

⟶*Seite 6*

Applet
: In Anlehnung an „Applikation“ kleines Programm, das meist eine visuelle Repräsentation besitzt. Ursprünglich im Rahmen von Java verbreitetes Konzept von Anwendungskomponenten für Internet-Browser, mittlerweile aber auch häufig allgemein für „kleine“ (Dienstleistungs-) Programme zu finden.

⟶*Seite 60*

Assistent
: Ein Werkzeug welches mit moderner grafischer Benutzungsoberfläche den Benutzer bei der Erstellung von Produkten hilft, indem es ihn durch den Erstellungsprozess leitet und Routineaufgaben abnimmt. Bei der Programmentwicklung ist das Produkt ein Codegerüst. Dieses muß dann in der Regel vom Programmierer ergänzt oder spezialisiert wer-

den. Geschieht auch dieses automatisch, liegt eine →*generative Programmierung* in Reinform vor.

⟶*Seite 53*

Attribut

Bei der objektorientierten Programmierung beschreibt es, welche Daten die Objekte der →*Klasse* enthalten. Bei GIS können geografische Objekte mit nicht geografischen Daten in Form von Attributen erweitert werden.

⟶*Seite 6*

Ausnahme

Ausnahmen sind ein programmiersprachliches Konzept. Eine Folge von Anweisungen wird solange abgearbeitet bis eine Anweisung eine Ausnahme ausgelöst – geworfen (engl. *throw*) – wird oder die Folge beendet ist. Wurde eine Ausnahme geworfen, wird eine spezielle entwicklerdefinierte Anweisung (die sogenannte *catch*-Routine) zur Fehlerbehandlung ausgeführt. Sprachen mit Ausnahmen (engl. *Exception*) sind beispielsweise C++ und Java.

⟶*Seite 63*

Einfachvererbung

Jede →*Klasse* besitzt maximal eine direkte Vorlage. Daraus ergibt sich eine Baumhierarchie (→*Mehrfachvererbung*).

⟶*Seite 6*

Entwurfsmuster

Entwurfsmuster (engl: *design patterns*) sind in der objektorientierten Softwareentwicklung etablierte, musterhafte Beschreibungen von typischen Strukturen und Verhaltensweisen mehrerer Objekte. Sie sind im Vergleich zu →*Frameworks* kleinere architektonische Einheiten (engl. *microarchitectures*), die als Designrichtlinien für den Aufbau von Frameworks und framework-basierten Anwendungssystemen genutzt werden können.

⟶*Seite 7*

Farbschema

Ein Farbschema eines GIS ist mit der Legende einer konventionellen Karte zu vergleichen. Es wird die Darstellungsart (etwa Farbe oder

Muster) einer Fläche eines bestimmten Typs bestimmt. Die Darstellungsart ist von dem Zweck der Karte abhängig. So sind etwa auf einer politischen Karte im Gegensatz zu einer Wanderkarte keine Wälder eingezeichnet.

⟶*Seite 75*

Framework

Ein Framework stellt eine Sammlung wiederverwendbarer Entwurfsentscheidungen eines Anwendungsgebietes dar. Sie werden bei einem objektorientierten Framework in Form von (abstrakten) →*Klassen* und einer Beschreibung der Zusammenarbeit der aus ihren Klassen erzeugten Instanzen ausgeliefert.

⟶*Seite 9*

Generative Programmierung

Methode der Softwareentwicklung, die auf dem Einsatz eines Generators beruht, der aus bestehenden Grundgerüsten einer Softwarekomponente automatisch neue, einsatzfähige →*Komponenten* erzeugt, wobei der Vorgang unter Zuhilfename von Schablonen (siehe auch →*Entwurfsmuster*, Regeln und einem Wissen über den geplanten Anwendungsbereich abläuft – möglicherweise unter dem steuerenden Einfluß eines Entwicklers. Die entstehende Software gehört zu einer Produktfamilie, die sich die durch den Generator repräsentierten Konzepte und Eigenschaften teilt.

⟶*Seite 25*

Kapselung

Auf die →*Attribute* eines Objektes kann nur über die →*Methoden* eines Objektes zugegriffen werden. Für andere →*Klassen* und Objekte sind die Attribute einer Klasse oder eines Objektes unsichtbar.

⟶*Seite 5*

Klasse

Beschreibt in Form einer Schablone eine Kategorie von Objekten, die gleiche oder ähnliche Verhaltensmuster aufweisen. Von einer Klasse können Instanzen erzeugt werden.

⟶*Seite 5*

Klassenpfad
In diesen Verzeichnissen sucht der Javainterpreter nach den Systemklassen und den benutzerdefinierten Klassen. Der Klassenpfad kann üblicherweise beim Start des Interpreters mit dem Parameter `-classpath` angeben werden. Alternativ kann eine Umgebungsvariable `CLASSPATH` gesetzt werden, welche der Interpreter beim Start auswertet.

⟶*Seite 62*

Komponente
Eine Komponente im Sinne dieser Arbeit ist ein in sich geschlossenes, also nicht veränderbares oder erweiterbares Element der Softwareentwicklung mit vertraglich festgelegten Schnittstellen und Verhalten sowie angegebenen Anforderungen an die Umgebung wie beispielsweise die benötigte Laufzeitumgebung. Eine Komponente wird als Black-Box ausgeliefert.

⟶*Seite 17*

Mehrfachvererbung
Jede →*Klasse* kann mehr als eine direkte Vorlage besitzen. Werden gleichnamige →*Attribute* oder →*Methoden* von verschiedenen Vorlagen geerbt, dann muß der Namenskonflikt aufgelöst werden (siehe auch →*einfache Vererbung*).

⟶*Seite 6*

Methode
Ausführbare Tätigkeit im Sinne einer Funktion oder Prozedur bzw. eines Algorithmus. Eine Methode beschreibt das Verhalten eines Objektes bzw. einer →*Klasse*.

⟶*Seite 6*

Polymorphismus
Polymorphismus beschreibt die Möglichkeit eines Objekts, auf denselben Methodenaufruf in verschiedenen Situationen verschiedenartig zu reagieren. Polymorphismus ermöglicht die Behandlung von Kontexten.

⟶*Seite 6*

Socket
Ein Socket bezeichnet eine mögliche (virtuelle) Verbindung zwischen zwei Systemen unter Verwendung des TCP. Ein Socket gibt seinem

Benutzer eine Punkt-zu-Punkt-Verbindung, welche die Komplexität des verwendeten Netzwerkes vor ihm verbirgt.

⟶*Seite 60*

Variationspunkt
Bereits im Entwurf einer Software vorgesehene Orte, die für eine spätere konzeptuelle oder technische Modifikation oder Ergänzung geeignet oder sinnvoll erscheinen, ohne die ursprünglich geplante Funktionalität zu gefährden.

⟶*Seite 9*

Vererbung
→*Attribute* und →*Methoden* einer Vorlage werden an die neue Klasse vererbt. Man unterscheidet die →*einfache Vererbung* und die →*Mehrfachvererbung*.

⟶*Seite 6*

Literaturverzeichnis

Abinavam u. a. 1998 Abinavam, Srinath ; Buch, Amit ; Cattoir, Eric ; Chilanti, Michele ; El-Rafei, Sherif ; Krause, Alan ; Müller, Werner ; Zuliani, Fernando: *San Francisco Concepts & Facilities*. Rochester : IBM Corporation, Februar 1998. – URL `http://www.ibm.com/Java/Sanfrancisco`

Baldzer 1999 Baldzer, Jörg: Sicherheit und Zahlung im Internet / Carl von Ossietzky Universität Oldenburg. Oldenburg, April 1999 (IS 42). – Interne Berichte. Zwischenbericht der Projektgruppe: Digitale Stadt Oldenburg – Teil B –

Balzert 1996 Balzert, Helmut: *Lehrbuch der Software-Technik*. Heidelberg : Spektrum Akademischer Verlag, 1996

Bäumer u. a. 1997 Bäumer, Dirk ; Gryczan, Guido ; Knoll, Rolf ; Lilienthal, Carola ; Riehle, Dirk ; Züllighoven, Heinz: Framework development for large systems. In: *Communications of the ACM* 40 (1997), Oktober, Nr. 10, S. 52–59

Biggerstaff 1994 Biggerstaff, Ted J.: The Library Scaling Problem and the Limits of Concrete Component Reuse / Microsoft Research – Advanced Technology Division. Redmond, November 1994 (MSR-TR-94-19). – Technical Report. Paper presented at Third International Conference on Reuse, Nov. 1994

Bleich 2000 Bleich, Holger: Hochverfügbare Ausfälle – Web-Hosting bei Strato: Jetzt soll alles besser werden. In: *c't* 15 (2000), S. 52f.

Brodbeck und Rupietta 1994 Brodbeck, Felix C. ; Rupietta, Walter: Fehlermanagement und Hilfesysteme. In: Eberleh, Edmund (Hrsg.) ; Oberquelle, Horst (Hrsg.) ; Oppermann, Reinhard (Hrsg.): *Einführung in die Software-Ergonomie*. 2. völlig neu bearbeitete Auflage. Berlin : Walter de Gruyter & Co., 1994 (Mensch Computer Kommunikation – Grundwissen 1), Kap. 5, S. 197–234

Büchi und Weck 1997 Büchi, Martin ; Weck, Wolfgang: A Plea for Grey-Box Components / Turku Centre for Computer Science. Turku, August 1997 (122). – TUCS Technical Report. – URL `http://www.tucs.abo.fi`

Cremers u. a. 1998 Cremers, Armin B. ; Kahler, Helge ; Pfeifer, Andreas ; Stiemerling, Oliver ; Wulf, Volker: PoliTeam – Kokonstruktive und evolutionäre Entwicklung einer Groupware. In: *Informatik-Spektrum* 21 (1998), S. 194–202

Eisenecker und Czarnecki 1999 Eisenecker, Ulrich W. ; Czarnecki, Krzysztof: In Einzelteilen – Generative Programmierung: wie [sic] man Komponenten baut und nutzt. In: *iX* 2 (1999), Februar, S. 126–132

Fayad 1999 Fayad, Mohamed E.: Application Frameworks. In: Fayad, Mohamed E. (Hrsg.) ; Schmidt, Douglas C. (Hrsg.) ; Johnson, Ralph E. (Hrsg.): *Building Application Frameworks: Object-Oriented Foundation of Framework Design.* New York : John Wiley & Son, 1999, Kap. 1, S. 3–28

Flanagan 1996 Flanagan, David: *Java in a Nutshell.* 1. Auflage. O'Reilly/International Thomson, 1996

Friebe 2000 Friebe, Jörg: *Eine Architektur für GIS im Internet.* 2000. – Vorabauszug aus einer Dissertation

Froehlich u. a. 1999 Froehlich, Garry ; Hoover, H. J. ; Liu, Ling ; Sorenson, Paul: Reusing Hooks 219. In: Fayad, Mohamed E. (Hrsg.) ; Schmidt, Douglas C. (Hrsg.) ; Johnson, Ralph E. (Hrsg.): *Building Application Frameworks: Object-Oriented Foundation of Framework Design.* New York : John Wiley & Son, 1999, Kap. 9, S. 219–236

Gamma u. a. 1996 Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software.* Bonn : Addison-Wesley, 1996

Garlan u. a. 1995 Garlan, David ; Allen, Robert ; Ockerbloom, John: Architectural Mismatch, or, Why it's hard to build systems out of existing parts. In: *Proceedings of the 17th International Conference on Software Engineering.* Seattle, Washington, April 1995, S. 179–185

Gorny 1999 Gorny, Peter: *Einführung in die Software-Ergonomie.* 1999. – Skript zur Vorlesung zum Wintersemester 1999/2000

Griffel 1998 Griffel, Frank: *Componentware – Konzepte und Techniken eines Softwareparadigmas*. Heidelberg : dpunkt-Verlag, 1998

Heidelbach 2000 Heidelbach, Oliver. *V.E.R.A. – Verzeichnis EDV-Relevanter Akronyme*. 2000. – URL `http://fub46.zedat.fu-berlin.de:8080/oheiabbd-cgi-bin/veramain.pl`

Horstmann und Cornell 2000 Horstmann, Cay S. ; Cornell, Gary: *Core Java 2 – Expertenwissen*. Markt+Technik, 2000

Ivanov 1996 Ivanov, Evgeni: Entwicklung und Anwendung von Frameworks – Probleme und Lösungsansätze / TU Ilmenau. Dezember 1996. – Studie

Jaekel 1999 Jaekel, Holger: *Konfiguration von frameworkbasierten Anwendungen*, Carl von Ossietzky Universität Oldenburg, Diplomarbeit, 1999

Johnson 1992 Johnson, Ralph E.: *Documenting Frameworks using Patterns*. 1992. – To be presented at OOPSLA'92

Johnson 1993 Johnson, Ralph E. *How to Design Frameworks*. 1993

Johnson 1997 Johnson, Ralph E.: Frameworks = (components + patterns). In: *Communications of the ACM* 40 (1997), Oktober, Nr. 10, S. 39–42

Laitinen 1999 Laitinen, Mauri: Framework Maintenance: Vendor Viewpoint. In: Fayad, Mohamed E. (Hrsg.) ; Schmidt, Douglas C. (Hrsg.) ; Johnson, Ralph E. (Hrsg.): *Building Application Frameworks: Object-Oriented Foundation of Framework Design*. New York : John Wiley & Son, 1999, Kap. Sidebar 9, S. 620ff.

Lajoie und Keller 1994 Lajoie, Richard ; Keller, Rudolf K.: Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert. In: *to appear in Proceedings of the 62nd Congress of the Association Canadienne Française pour l'Avancement des Sciences (ACFAS)*. Montreal, Canada, Mai 1994. – Colloquium on Object Orientation in Databases and Software Engineering.

Lundberg und Mattsson 1996 Lundberg, Christer ; Mattsson, Michael: On Using Legacy Software Components with Object-Oriented Frameworks. In: *Proceedings of Systemarkitekturer '96,*. Borås, 1996

Mattsson 1996 MATTSSON, Michael: *Object-Oriented Frameworks – A survey of methodological issues*, University College of Karlskrona/Ronneby, Diplomarbeit, 1996

Mattsson 1999 MATTSSON, Michael. *Frameworks FAQ's*. 1999. – URL `http://www.ipd.hk-r.se/michaelm/fwpages/fwfaqs.html`

McGregor 1995 MCGREGOR, Tony. *ASN.1*. Juli 1995. – URL `http://byerley.cs.waikato.ac.nz/tonym/articles/asn/asn.1.html`

MySQL 2000 MYSQL: *MySQL Reference Manual*. 2000. – URL `http://www.mysql.com`. – Version 3.23.10-alpha

Oestereich 1997 OESTEREICH, Bernd: *Objektorientierte Softwareentwicklung mit der UML*. 3. aktualisierte Auflage. R. Oldenbourg, 1997

OMG 1999 OMG: *Unified Modeling Language Specification*. 1999. – Version 1.3

Prechelt 1999 PRECHELT, Lutz: Comparing Java vs. C/C++ Efficiency Differences to Interpersonal Differences. In: *Communications of the ACM* 42 (1999), Oktober, Nr. 10, S. 109–112

Ritter 2000 RITTER, Jörg: *Prozessorientierte Konfiguration komponentenbasierter Anwendungssysteme*, Carl von Ossietzky Universität Oldenburg, Dissertation, 2000

Schallenberg 1999 SCHALLENBERG, Andreas: Komponentenmodelle / Carl von Ossietzky Universität Oldenburg. Oldenburg, Oktober 1999 (IS 44). – Interne Berichte. Endbericht der Projektgruppe: Digitale Stadt Oldenburg – Teil B –

Schmachtel und Schreck 2000 SCHMACHTEL, Martin ; SCHRECK, Olaf: Verteilter Verteiler – Postfix als Sendmail-Alternative. In: *iX* 5 (2000), Mai, S. 92–97

Schwonbeck und Roos 2000 SCHWONBECK, Susanne ; ROOS, Ute: Zeit statt Geld – taz: Redaktionssystem aus freien Komponenten. In: *iX* 5 (2000), Mai, S. 100–103

Sommerville 1996 SOMMERVILLE, Ian: *Software engineering*. 5. überarbeitete Auflage. Addison-Wesley, 1996

Stoll 1989 STOLL, Clifford: *Kuckucksei*. Fischer Taschenbuch, 1989

Sun Microsystems 2000 Sun Microsystems. *The JavaTM Tutorial*. 2000. – URL `http://java.sun.com/docs/books/tutorial/index.html`

Wooridge 2000 Wooridge, Richard. *Components and Reuse*. 2000

Index

Fettgedruckte Seitennummern verweisen auf Seiten, wo Begriffe aus dem Glossar zum ersten Mal auftreten. Normal gesetzte Seitennummern beziehen sich auf Erwähnungen im Text.
Bei in `Schreibmaschinenschrift` gesetzten Begriffen handelt es sich im Klassen oder Bezeichner aus Java-Implementierung.

Symbole

`_show`, 74

A

Abrechnung, 34
abstrakte Klasse, **6**, 61
`AbstrakterZugangsServer`, 77
`Action`, 72
ActiveX, 18, 22, 23
Adapter, 37, 43
Administrator, 24
Anerkennung, 22
ANSI, 58
API, 62, 63, 68, 69, 71
Applet, **60**
Applikationsgenerator, 11, 25
Assistent, 2, **53**
Attribut, 5, **6**
Attribute, 60
Ausnahme, **63**
Authentizität, 22
Autorisierung, 22
AWT, 10, 59

B

Beobachter, 8, 35, 65
Binden
    dynamisches, 7, 10
    spätes, 7, 10
Black-Box, 3, 13, 17, 19, 54
broker, 44

C

C, 58
C++, 58
`Collection`, 60
Compiler, 63
`concat`, 74
CORBA, 20, 26
CSCW, 25
customizing, 28

D

DBMS, 42–44, 49, 52, 53, 69
DCOM, 20, 26
dezentraler Entwurf, 35
Dialogeditor, 54
Dienst, 44
Dokumentationsmuster, 16
`doUpdate`, 72
DSL, 25
Dynamische Bindung, 10

E

`Eigenschaften`, 61
Einfachvererbung, **6**
Enterprise JavaBeans, 18
Entwurf
    dezentraler, 35
    zentraler, 35
Entwurfsmuster, **7**, 16, 57, 65
    Adapter, 41, 43
    Beobachter, 8, 35, 65
    Fabrik, 63
    Schablone, 43, 72
ereignissgesteuerte Programmierung, 10
`EriwanZugangsServer`, 66, 71
`Exception`, 63
Exploder, 23

F

Farbschema, **75**
FDL, 17
formale Sprache, 11, 22
Framework, 2, **9**, 17, 27
    komponentenbasiert, 18
frozen-spot, 9

G

Gebos, 10
Generative Programmierung, **25**, 27
Generator, 2, 32, 39, 53, 55
`GeodatenInterface`, 75
`GeodatenInterface`, 64
GIS, 1, 2, 31, 33, 39, 44, 49, 52, 74, 75
Glass-Box, 13
Glue, 27
Granularität, 17
Grey-Box, 13

H

Hollywoodprinzip, 10
hook, 16
hot-spot, 9

I

IBM, 10
IC, 2, 20, 21
ICCL, 25
Indirektion, 27
Instanz, 5, 40
    Komponente, 40
Integrität, 22
`InterGISGeodatenKomponente`, 75
InterGIS, 2, 32, 74–76
`InternalFrameDemo`, 79
Internet-Browser, 18
IP, 26, 27
ISO, 27
IVBB, 23

J

JAR, 67, 68
Java, 24, 57, 58
JavaBeans, 18, 22
    Enterprise, 18
`javadoc`, 68
JDBC, 68–71, 73, 82
JDBC-Verfolgung, 71
`JPanel`, 72

K

Kapselung, **5**, 57
Kartenserver, 49
Klasse, **5**, 61
    abstrakte, 6, 61
    Wrapper-, 15

Klassenbibliothek, 15
Klassenhierarchie, 6, 27, 39
Klassenpfad, **62**, 67, 68
Kochbuch, 16
Kohäsion, 17
Komponente, 2, **17**
`KomponentenGenerator`, 73
Komponentenklasse, 40
`KomponentenLader`, 63
Konfiguration, 40
Konstruktor, 62
Kontrollfluß, 10
Koordinatensystem, 48
Kosten, 34

L

legacy-code, 15
Lizenz, 45
Lizenzverwaltung, 45
`LoggingZugangsServer`, 66
`LoggingBeobachter`, 64
`LoggingZugangsServer`, 66, 71

M

Makler, 44
Mehrfachvererbung, **6**, 61
Methode, 5, **6**
    abstrakte, 6
    virtuelle, 7
MFC, 10
Mikroarchitektur, 12, 16, 65
motif, 16
MVC, 10

N

Nachbedingung, 19
NFS, 67
`NiceSsoServerClient`, 77
Nutzungsstatistik, 34

O

Objektorientiert, 57
objektorientierte Programmierung, 5
ODBC, 68
OFFIS, 2, 32
`onLeave`, 72
OO, 5
Open-Source, 19, 20
OSI, 27

P

Pin, 2
Pipe, 18
Plug-In, 18, 26
PlugIn, 60
Polymorphismus, **6**
portabel, 57
Portabilität, 55
post-condition, 19
pre-condition, 19
`Properties`, 61
Programmierung
    ereignissgesteuerte, 10
    generative, 25
    objektorientierte, 5
`Properties`, 61, 66, 75
`PropertiesZugangsServer`, 66, 67
Protokoll, 42
Protokollumsetzung, 37
Prototyp, 54
Proxy, 33, 64, 66

R

Rechnung, 34, 44, 47
Rechtematrix, 25

Reflection, 62, 63
Regelmenge, 25
reverse engineering, 27
RWG, 10

S

`SachdatenInterface`, 64, 72
sandbox, 24
Sandkasten, 24
Schablone, 43, 72
`SchluckenderBeobachter`, 64
Schnittstelle
    direkte, 60
Schnittstellen
    direkte, 65
show, *siehe* `_show`
skins, 50
Skripte, 18
Socket, **60**
Späte Bindung, 10
`SplitPaneDemo`, 78
Sprache
    formale, 11
SQL, 68, 69, 74, 75
SSO, 76
`SSOZugangsServer`, 77
Stellvertreter, 33, 42, 64, 66
`StreamTokenizer`, 74
Sun, 59
Swing, 59, 64
`SwingKomponentenLader`, 64

T

`TabbedPaneDemo`, 78
TCP, 26, 27
Tutorial, 16

U

UML, 61
Unix, 58
URL, 69

V

Variationspunkt, **9**
VBX, 27
`VegesInterface`, 61
Verbreitung, 58
Vererbung, **6**, 57
Verfügbarkeit, 22
Verschlüsselung, 41
`VerteilenderBeobachter`, 64
Vertraulichkeit, 22
VHDL, 20
virtuelle Maschine, 58
Vorbedingung, 19

W

Wartung, 57
White-Box, 3, 12
Wiederverwendung
    Black-Box, 13
    White-Box, 12
wizard, 53
Wrapper-Klasse, 15

Z

zentraler Entwurf, 35
Zertifikat, 23
`ZugangsServerException`, 63
`ZugangsServerInterface`, 63
`ZugangsServerNachrichten-Texte`, 63
`ZugangsServerInterface`, 77
Zugangsserver, 33, 34, 41, 44, 45, 47, 48, 65
`ZugangsServerInterface`, 66, 77
Zugangsserverproxy, 33
Zugriffskontrolle, 22

Erklärung zur Urheberschaft

Hiermit versichere ich, daß ich diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe.

Oldenburg, den 14.11.2000

Thorben Kundinger