Lehrstuhl für Forstliche Arbeitswissenschaft
und Angewandte Informatik

Die KOMET-Architektur: Eine Integrationsplattform für forstliche Entscheidungsunterstützungskomponenten

Martin Döllerer

Vollständiger Abdruck der von der Fakultät Wissenschaftszentrum Weihenstephan für Ernährung, Landnutzung und Umwelt der Technischen Universität München zur Erlangung des

akademischen Grades eines

Doktors der Forstwissenschaft (Dr. rer. silv.)

genehmigten Dissertation.

Prüfer der Dissertation:
1. Univ.-Prof. Dr. H.-D. Quednau, i. R.
2. Univ.-Prof. Dr. H. Pretzsch
3. Univ.-Prof. Dr. Th. Knoke

Die Dissertation wurde am 22.10.2007 bei der Studienfakultät für Forstwissenschaft eingereicht und durch die Fakultät Wissenschaftszentrum Weihenstephan für Ernährung, Landnutzung und Umwelt am 21.12.2007 angenommen.

Danksagung

Für Rosemarie, Sepp und Seppi

Ein einziger Mensch verbürgt sich, der alleinige Urheber eines Werkes wie dieses hier zu sein. Das ist richtig, stimmt aber nicht so ganz. Es gibt nämlich eine ganze Menge guter Geister, die ihr Unwesen ganz dezent im Hintergrund treiben. Und wenn sich so eine Arbeit langsam ihrem Ende zuneigt, ist es an der Zeit, all diesen guten Geistern zu danken. Das ist gar nicht so einfach, weil man sehr leicht jemanden vergisst. Und deshalb möchte ich an erster Stelle mit einem herzlichen »Vergelt's Gott« bei all jenen bedanken, die ich hier vergessen habe.

Herrn Prof. Dr. Quednau, meinem Doktorvater, möchte ich ein recht herzliches »Vergelt's Gott« sagen. Er hatte immer ein offenes Ohr für meine kleinen Anliegen, gab mir stets die Freiheit, mich in die verschiedensten Themen zu vertiefen, und schließlich konnte ich in unseren gemeinsamen Lehrveranstaltungen Dozentenluft schnuppern.

Auch Herrn Prof. Dr. Warkotsch, dem Leiter des Lehrstuhls für Forstliche Arbeitswissenschaft und Angewandte Informatik, an den es mich jetzt endgültig wieder zurückgespült hat, möchte ich an dieser Stelle von ganzem Herzen »Vergelt's Gott« sagen für die angenehme und produktive Atmosphäre am Lehrstuhl und dafür, dass er mich eine ganze Zeit lang an das Fachgebiet von Herrn Prof. Dr. Quednau ausgeliehen hat.

Martin Hemm gab mir die Möglichkeit, meine Ideen am lebenden Objekt auszuprobieren. Bis ich die Dinge bis ins Letzte verstand, musste er mir das eine oder andere Mal auf die Sprünge helfen. Dafür möchte ich mich bei ihm an dieser Stelle mit einem herzlichen »Vergelt's Gott« bedanken.

Peter Biber und Ralf Moshammer vom Lehrstuhl für Waldwachstumskunde möchte ich dieser Stelle auch herzlich danken. Um den Waldwachstumssimulator SILVA kommt so leicht niemand herum - ich auch nicht. Erstens hat mir der Lehrstuhl für Waldwachstumskunde das SILVA-Programm freimütig überlassen, zweitens bekam ich immer dann, wenn zu haken schien, prompte und professionelle Hilfe von einem der beiden. Dafür möchte ich ihnen recht herzlich »Vergelt's Gott« sagen.

Meinen lieben Kolleginnen und Kollegen am Lehrstuhl gebührt natürlich ein besonderer Dank für ihr Verständnis - vor allem während der berühmt-berüchtigten heißen Phase kurz vor Schluss. Da musste ich sie ab und an ein wenig darben lassen. Auch für die vielen Gespräche und Ideen möchte ich bei allen mit einem »Vergelt's Gott« bedanken.

Auch Matthias Müller möchte ich an dieser Stelle von ganzem Herzen »Vergelt's Gott« sagen, obwohl er mich meistens vom promovieren abgehalten hat, wegen vordergründig nebensächlicher Administrations-Arbeiten an unseren heiligen Servern, die uns beide so manches Mal fast an den Rand des Wahnsinns getrieben hätten.

Als mich Andi Mohr damals (das war 1996...) fragte, ob ich seine Diplomarbeit betreuen wolle, und ich sein Anliegen bejahte, hat mich das endgültig auf die wissenschaftliche Bahn geworfen. Für seine liebenswerte, aber bestimmte Art wie er mich immer wieder auf den Boden der wissenschaftlichen Tatsachen zurückholte und die vielen interessanten nicht enden wollenden Gespräche möchte ich ihm und vor allem seiner Familie von ganzem Herzen »Vergelt's Gott« sagen.

Dr. Michael Makas hat dieses Machwerk mehrmals kritisch beäugt und mich so manches Mal mit seiner sagenumwobenen Brotzeit vor dem sicher geglaubten Hungertod bewahrt. Ihm und seiner lieben Familie sei an dieser Stelle mit einem »Vergelt's Gott« von ganzem Herzen gedankt.

Und last but not least - wie der Lateiner so schön sagt - möchte ich noch Michael Sichler an dieser Stelle ganz herzlich »Vergelt's Gott« sagen fürs Ratschen beim Kaffee - Sonntags nach der Kirche bei der Franca in Unterwössen.

Inhaltsverzeichnis

Verzeichnis der Abbildungen

Verzeichnis der Listings

Verzeichnis der Abkürzungen

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CASE	Computer Aided Software Engineering
CORBA	Common Object Request Broker Architecture
CRM	Customer Relationship Management
DBMS	Datenbank Management System
DGM	Digitales Geländemodell
DGMS	Dialog Generierungs und Management System
DSS	Decision Support System
DTD	Document Type Definition
EAI	Enterprise Application Integration
ERP	Enterprise Resource Planning
ESRI	Environmental Systems Research Institute
EUS	Entscheidungs-Unterstützungs-System
GIS	Geografisches Informationssystem
GNU	GNU's Not Unix
GPL	GNU Public License
GUI	Graphical User Interface
IIOP	Internet Inter Orb Protocol
IT	Informationstechnologie
JDBC	Java Data Base Connectivity
JDK	Java Development Kit
JRE	Java Runtime Environment
JVM	Java Virtual Machine
KOMET	Komponentenorientierte Methode
KometML	KOMET Markup Language
MBMS	Modellbank Mangement System
MDE	Mobiles Dateneingabe Gerät
MSIL	Microsoft Intermediate Language
ODBC	Open Data Base Connectivity
OMG	Object Management Group
OMT	Object Modelling Technique

OOSE	Object Oriented Software Engineering
OWL	Web Ontology Language
PDA	Personal Digital Assistant
RDBMS	Relationales Datenbank Mangement System
RDF	Resource Description Framework
RDFS	RDF Schema
REUS	Räumliches Entscheidungs-Unterstützungs-System
RMI	Remote Method Invocation
RUP	Rational Unified Process
SAX	Simple API for XML Processing
SDSS	Specific Decision Support System
	Spatial Decision Support System
SOA	Serviceorientierte Architektur
SOAP	Simple Object Access Protocol
UML	Unified Modelling Language
XML	Extensible Markup Language
XP	Extreme Programming

1 Einführung und Problemstellung

Zunehmende Kundenorientierung, steigender Kostendruck und der Trend zu einer nachhaltigen und multifunktionalen Waldbewirtschaftung sorgen für eine ständig zunehmende Komplexität forstlicher Managemententscheidungen. Im Bereich der strategischen, taktischen und operativen Planung der Holzernte können beispielsweise Teilprobleme unter anderem aus folgenden forstlichen Forschungsfeldern auftreten:

- Forstliche Verfahrenstechnik (Holzerntetechnik)
- Forstliche Arbeitswissenschaft (Einsatzplanung)
- Waldbau und Forsteinrichtung (Forstliche Planung)
- Waldernährung und Wasserhaushalt (Nährstoffversorgung, Nährstoffentzug)
- Bodenmechanik (Befahrung)
- Waldwachstumskunde (Wuchsmodell, Biodiversität, Strukturparameter)
- Forstliche Wirtschaftslehre (Holzmarkt)
- Forstbotanik, Forstzoologie (Biodiversität, Habitatveränderungen).

Zur effizienten Lösung von Fragestellungen mit derart vielfältigen Randbedingungen ist der Einsatz moderner Informationstechnologie notwendig (Becker, Jaeger und Koch [1998], Shao und Reynolds [2006]). Dabei werden oft mehrere Applikationen, die jeweils einen Teilbereich der Fragestellung abdecken, nacheinander verwendet. Beispiele für solche Teillösungen sind der Waldwachstumssimulator SILVA (Pretzsch [2001], Pretzsch, Biber und Dursky [2002]), das Bodeninformationssystem ProFor (Ziesak [1999], Ziesak [2004]) bzw. ein System zur Simulation von Holzerntemaßnahmen (Hemm und Oros [2005], Hemm [2006]). Der Datentransfer zwischen diesen Anwendungen sowie die Zusammenführung der verschiedenen Ergebnisse erfolgen meist manuell (Hemm [2006]), wodurch Medienbrüche und Fehlerquellen entstehen.

Eine Zusammenführung solcher Teillösungen zu einem Gesamtsystem ist wünschenswert, damit die Entscheidungsfindung in Zukunft effizienter und zudem auch ganzheitlicher als heute erfolgen kann. Zeitraubende und fehleranfällige manuelle Arbeitsschritte, wie zum Beispiel die zuvor angeführte Datenweitergabe von einer Applikation zur nächsten, erfolgen automatisiert, und durch Medienbrüche bedingte Fehlerquellen - im aufwändigsten Fall durch manuelle Datenübertragung - werden dadurch von vornherein eliminiert.

Bei derartigen Anwendungs-Szenarien kommen vor allem so genannte ***E**ntscheidungs-**U**nterstützungs-**S**ysteme* (*EUS*) (engl. ***D**ecision **S**upport **S**ystems* (*DSS*)) zum Einsatz. Nach SPRAGUE [1980] werden diese als Computersysteme charakterisert, die:

- es ermöglichen, schwach bis unstrukturierte Probleme zu bearbeiten, denen typischerweise Manager in höheren Führungsebenen gegenüberstehen,
- in der Lage sind, analytische Modelle sowie Datenzugriffs- und Selektionsmethoden zu kombinieren,
- so benutzerfreundlich sind, dass sie auch von Nicht-EDV-Fachleuten bedient werden können, und
- so flexibel und anpassbar sind, dass sie auf Veränderungen der Systemumgebung oder des Entscheidungsfindungsprozesses reagieren können.

Um ein reibungsloses Zusammenwirken der einzelnen Komponenten zu gewährleisten, ist es notwendig, Basisdienste zur Verfügung zu stellen und Rahmenbedingungen zu definieren. Innerhalb eines solchen Systems muss beispielsweise bekannt sein, welche Daten weitergegeben werden und wie darauf zugegriffen werden kann, damit diese modulübergreifend zur Verfügung stehen können. Darüber hinaus ist es notwendig, zusätzliche Informationen über die Bedeutung der Daten (*Semantik*) im System zu hinterlegen, um unterschiedliche Interpretationsmöglichkeiten der Daten (z. B.: Volumen *mit* Rinde bzw. Volumen *ohne* Rinde) ausschließen zu können (KUROPKA UND WESKE [2006]).

Ferner sollte die Möglichkeit bestehen, weitere Anwendungen einzubinden, zum Beispiel forstliche Standardsoftware oder Geografische Informationssysteme (*GIS*).

Mit Hilfe solch eines EUS, welches alle wesentlichen Teillösungen integriert, können (beispielsweise durch mehrmalige Ausführung der Anwendung) die Untersuchung und der Vergleich der Auswirkungen verschiedener Handlungsalternativen sehr effizient gestaltet werden.

Um diesen Anforderungen Rechnung zu tragen, haben sich zur Realisierung von Entscheidungs-Unterstützungs-Systemen Vorgehensweisen etabliert, die auf *Software-Engineering* sowie auf *Software-Komponenten* beruhen. Diese werden auch im Forstbereich eingesetzt (MOHR [1997], LEMM, ERNI UND THEES [2002], NUTE ET AL. [2000]).

2 Zielsetzung der Arbeit

Ziel der vorliegenden Arbeit ist die Konzeption und Realisierung einer Integrationsplattform für forstliche Entscheidungs-Unterstützungs-Komponenten.

Die in Abschnitt 1 grob umrissenen Grundfunktionen und Rahmenbedingungen werden exakt spezifiziert, mit Hilfe einer **k**omponenten**o**rientierten **Met**hode eine Software-Architektur (*KOMET-Architektur*) entwickelt und deren konzeptionelles Modell beschrieben. Die *KOMET-Architektur* ist die Grundlage einer Integrationsplattform für Entscheidungs-Unterstützungs-Komponenten, mit deren Hilfe die Entwicklung und Anwendung spezialisierter Entscheidungs-Unterstützungs-Systeme sowie die Integration externer Anwendungen erleichtert wird. Dabei sollen sowohl neu zu entwickelnde Module als auch bereits bestehende Softwarelösungen in die Integrationsplattform eingebunden werden können.

Um der Komplexität des Systems Rechnung zu tragen und die erforderlichen Rahmenbedingungen nachvollziehbar und transparent zu halten, werden bei der Entwicklung der im Rahmen dieser Arbeit vorgestellten Integrationsplattform moderne Methoden der Anwendungsentwicklung in Form eines Software-Engineering-Prozesses angewandt.

Ziel der programmtechnischen Umsetzung ist eine Referenzimplementierung der *KOMET-Architektur*. Ferner wird eine Demonstrationsanwendung realisiert, welche die in Hemm [2006] beschriebenen Komponenten in Form eines **r**äumlichen **E**ntscheidungs-**U**nterstützungs-**S**ystems (REUS) zur Holzernte integriert. Dabei soll die grundsätzliche Machbarkeit der Implementierung von räumlichen Entscheidungs-Unterstützungs-Systemen auf Basis der *KOMET-Architektur* untersucht werden.

Die Verwendung dieser Demonstrationsanwendung wird exemplarisch anhand der Planung einer Durchforstungsmaßnahme in einem Musterbestand dargestellt. Dabei sollen zwei verschiedene Behandlungsvarianten, welche sich durch eine unterschiedliche Eingriffsstärke unterscheiden, zur Auswahl stehen. Die Durchführung beider Durchforstungsmaßnahmen soll mit Hilfe eines Holzerntesystems, bestehend aus Harvester und Forwarder, erfolgen.

Durch den Einsatz einer Holzernte-Simulation sollen verschiedene Merkmale quantifiziert und visualisiert werden, wie zum Beispiel:

- Anzahl entnommener Bäume
- Anzahl gerückter Abschnitte
- Geerntete Holzmenge
- Zurückgelegte Wegstrecke von Harvester und Forwarder
- Benötigte Arbeitszeit von Harvester und Forwarder.

3 Stand des Wissens

Zu Beginn der 60er Jahre, als Softwareentwickler nicht mehr in der Lage waren, die Erstellung großer Programme mit den damals bekannten Methoden und Werkzeugen zu beherrschen, wurde deutlich, dass die Entwicklung anspruchsvoller Software-Systeme ohne Planung im Vorfeld nicht mehr durchführbar war. Diese Situation führte schließlich zu einem Szenario, welches 1965 als *Softwarekrise* bezeichnet wurde (PAGEL UND SIX [1994]). Seitdem hat sich die Entwicklung von Software von einem Kunsthandwerk zu einer Ingenieursdisziplin (*Software Engineering*) gewandelt. Als Folge dieser Entwicklung stehen heute viele Methoden zur Verfügung, die den Software-Entwicklungsprozess unterstützen.

Sowohl der grundlegende Aufbau von Entscheidungs-Unterstützungs-Systemen als auch die dort eingesetzten Technologien spiegeln sich in Konzepten aus dem Software-Engineering wider.

3.1 Entscheidungs-Unterstützungs-Systeme

Der Komplex ***E****ntscheidungs-****U****nterstützungs-****S****ystem* (*EUS*) wird in SPRAGUE [1980] aus drei verschiedenen Sichten auf das System beleuchtet:

- Technik-orientierte Sicht
- Anwender-orientierte Sicht
- Inormationstechnologie-orientierte Sicht.

Charakteristika, die häufig bei Entscheidungs-Unterstützungs-Systemen beobachtet werden, dienen dabei als Ausgangspunkt.

Auf der grundlegenden Architektur von SPRAGUE [1980] basierend wurden Entscheidungs-Unterstützungs-Systeme im Laufe der Zeit weiterentwickelt und mit Mechanismen versehen, um Informationen aus immer vielfältigeren Quellen (Datenbanken, Geografische Informationssysteme, Simulationsmodelle, etc.) integrieren zu können (z. B. MAYER [1998], SEFFINO ET AL. [1999], SENGUPTA UND BENNETT [2003]).

3.1.1 Charakteristika

Folgende Charakteristika eines EUS werden in SPRAGUE [1980] genannt:

- die Tendenz, überwiegend schwach strukturierte bis unstrukturierte Probleme zu bearbeiten, denen typischerweise Manager in höheren Führungsebenen gegenüberstehen,
- der Versuch der Kombination von Modellen oder analytischen Techniken mit klassischen Datenzugriffs- und Selektionsmethoden,
- die Fokussierung auf Funktionen, welche diese Systeme auch durch Nicht-EDV-Fachleute einfach bedienbar machen, und
- die Betonung von Flexibilität und Anpassbarkeit, wodurch ein System auf Veränderungen der Systemumgebung sowie des Entscheidungsfindungsprozesses reagieren kann.

3.1.2 Technik-orientierte Sicht

Für den Betrachter ergeben sich aus technik-orientierter Sicht drei Ebenen:

- *spezielle Entscheidungs-Unterstützungs-Syteme* (***S**pecific **D**ecision **S**upport **S**ystems, SDSS)* auf oberster Ebene. Sie stellen die eigentlichen Systeme dar, die von Entscheidern zur Lösung ihrer speziellen Probleme eingesetzt werden.
- *Entscheidungs-Unterstützungs-System-Generatoren* (*DSS Generators*). Sie stellen eine Sammlung von Hard- und Software dar, mit deren Hilfe spezifische Entscheidungs-Unterstützungs-Systeme einfach und mit wenig Aufwand erzeugt werden können.
- *Entscheidungs-Unterstützungs-System-Werkzeuge* (*DSS Tools*). Auf dieser Ebene werden Hardware- oder Software-Elemente für EUS-Generatoren bereitgestellt. Dabei kann es sich beispielsweise um Programmierbibliotheken handeln oder um neueste Entwicklungen auf dem Hardware-Sektor, wie etwa ***m**obile **D**aten-**e**ingabegeräte* (*MDE*), Sprach- bzw. Handschriften-Erkennungssysteme oder ähnliches.

Zusätzlich zu den Ebenen werden in SPRAGUE [1980] Rollen definiert:

- *Der Anwender* entspricht dem Manager, welcher der Entscheidungssituation gegenübersteht und für die Konsequenzen verantwortlich ist.

- *Der Vermittler* ist dem Anwender in der Regel bei der Bedienung des EUS behilflich. Er kann auch beratende Funktion übernehmen.
- Die Aufgabe des *EUS-Konstrukteurs* ist die Erstellung eines speziellen EUS mit Hilfe der Möglichkeiten des EUS-Generators. Er muss mit dem Problemraum des Managers vertraut sein.
- *Der technische Entwickler* integriert, wenn dies erforderlich ist, neue Funktionalitäten oder Komponenten in den EUS-Generator. Neue Datenbanken, neue Analysemodelle und zusätzliche Berichtsformate werden bei Bedarf erstellt. An dieser Stelle werden fundiertes technisches Wissen und ein Grundverständnis der entsprechenden Entscheidungssituation benötigt.
- Zu den Aufgaben des *Tool-Programmierers* gehören unter anderem die Entwicklung von neuen Technologien, neuer Hard- und Software sowie die Verbesserung der Verbindungen der Subsysteme untereinander.

In Abbildung 3.1 sind die technik-orientierten Ebenen und ihre Zusammenhänge sowie die zugeordneten Rollen schematisch dargestellt. Spezielle Entscheidungs-Unterstützungs-Systeme können auch direkt aus EUS-Werkzeugen erstellt werden.

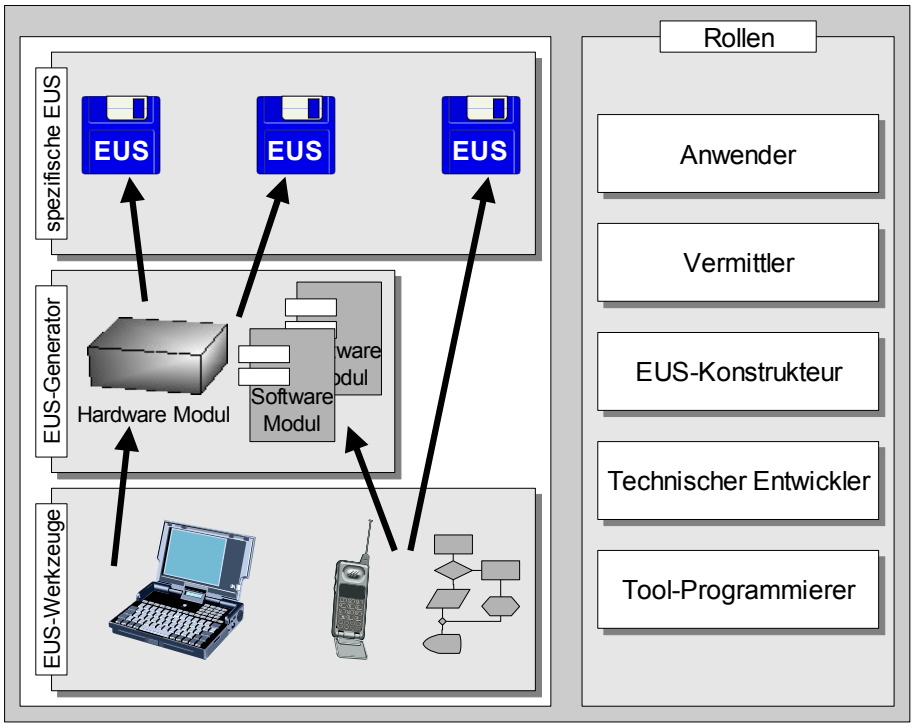


Abbildung 3.1: Technik-orientierte Ebenen eines EUS mit zugeordneten Rollen nach Sprague [1980]; verändert

3.1.3 Anwender-orientierte Sicht

Aus anwender-orientierter Sicht ergeben sich für die Entscheidungsebene Grundanforderungen an ein EUS. Sechs Grundanforderungen finden sich bei SPRAGUE [1980]. Es wird darauf hingewiesen, dass von einem speziellen EUS nicht zwingend alle Anforderungen erfüllt sein müssen.

Die Grundanforderungen sind:

1. Ein EUS soll das Management in Entscheidungssituationen unterstützen, der Schwerpunkt sollte auf schwach strukturierten bis unstrukturierten Problemen liegen.
2. Ein EUS soll das Management auf allen Ebenen unterstützen und wenn nötig für eine Integration zwischen den Führungsebenen sorgen.
3. Ein EUS soll sowohl in unabhängigen als auch in abhängigen Entscheidungssituationen eingesetzt werden können. Bei abhängigen Entscheidungssituationen müssen Entscheidungen entweder von mehreren Personen nacheinander oder in einer Gruppe gemeinsam getroffen werden.
4. Ein EUS soll alle Phasen eines Entscheidungsprozesses unterstützen. So besitzt etwa das Entscheidungsmodell von SIMON [1977] die Phasen:
   - *Intelligence*: In dieser Phase werden die Daten gesammelt, die zur Entscheidungsfindung benötigt werden.
   - *Design*: Mögliche Lösungen des Problems werden identifiziert und auf Machbarkeit untersucht.
   - *Choice*: Aus den möglichen Alternativen wird eine ausgewählt und umgesetzt.

   Alle dieser drei Phasen sollten mit einem EUS bearbeitet werden können.
5. Ein EUS soll mehrere Entscheidungsprozesse unterstützen. Neben dem Modell von SIMON [1977] existieren noch andere Entscheidungsprozesse (siehe z. B. VACIK UND LEXER [2007]). Das EUS soll den Anwender nicht auf ein bestimmtes Modell festlegen.
6. Ein EUS soll eine einfach zu bedienende Benutzeroberfläche besitzen.

3.1.4 Informationstechnologie-orientierte Sicht

Dem Anwendungsentwickler präsentiert sich ein Entscheidungs-Unterstützungs-System aus Informationstechnologie-orientierter Sicht, die aus drei Subsystemen besteht (SPRAGUE [1980]):

- das Datensubsystem,
- das Modellsubsystem und
- das Dialogsubsystem.

Mit Hilfe des Datensubsystems wird die Speicherung und Verwaltung der Daten realisiert, die zur Entscheidungsfindung benötigt oder während des Entscheidungsfindungsprozesses berechnet werden. Diese Berechnungen werden innerhalb des Modellsubsystems realisiert, da sich dort die Anwendungslogik befindet. Die Interaktion mit dem Benutzer erfolgt innerhalb des Dialogsubsystems. Abbildung 3.2 zeigt schematisch den Aufbau eines EUS aus seinen Subsystemen.

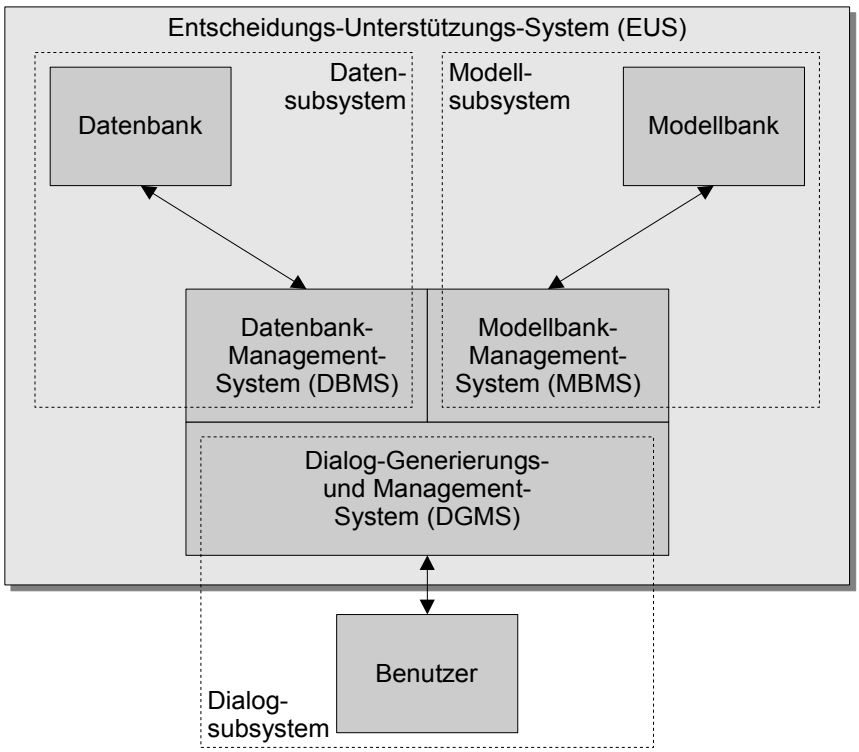


Abbildung 3.2: Bestandteile eines EUS nach Sprague [1980]; verändert

3.1.5 Unterstützung räumlicher Entscheidungen

Fragestellungen aus dem Bereich der Forstwirtschaft sind vielfach flächengebunden. Ein Entscheidungs-Unterstützungs-System, das für dieses Umfeld geeignet sein soll, muss

in der Lage sein, räumliche Fragestellungen zu bewältigen. Ein solches System wird als ***r**äumliches **E**ntscheidungs-**U**nterstützungs-**S**ystem* (*REUS*) bzw. ***S**patial **D**ecision **S**upport **S**ystem* (*SDSS*, siehe z. B. LEUNG [1997]) bezeichnet. Um diese Anforderung zu erfüllen, können beispielsweise einzelne Module eines EUS durch Integration eines ***G**eografischen **I**nformations**s**ystems* (*GIS,* siehe z. B. BARTELME [1995]) mit Funktionen zur räumlichen Analyse bzw. Visualisierung erweitert werden. Bei MOHR [1997] ist ein solcher Ansatz zur Erweiterung der Funktionalität einer forstlichen Standardsoftware näher beschrieben. Eine weitere Möglichkeit ist die Erweiterung eines GIS mit Funktionen aus der Entscheidungsunterstützung. Diese unter anderem von KEENAN [1997] beschriebene Möglichkeit wurde z. B. von LÜTHY [1998] sowie VACIK, LEXER UND PALMETZHOFER [2004] gewählt.

3.1.6 Modell-Integration

In umfangreichen Entscheidungs-Unterstützungs-Systemen können viele unterschiedliche Modelle zum Einsatz kommen, die dem Entscheidungsträger Ergebnisse zu Teilaspekten der jeweiligen Entscheidungssituation zur Verfügung stellen. Dabei können Szenarien entstehen, in denen bereits bestehende und neu entwickelte Modelle zusammenarbeiten müssen. Dabei kommen oftmals so genannte *Wrapper* zum Einsatz, die Modelle, welche als eigenständige externe Programme vorliegen - beispielsweise ältere Anwendungen, die bereits während der Entwicklung eines EUS verfügbar sind - über ein gemeinsames Datenmodell und eine gemeinsame Kommando-Sprache an einen Systemkern anbinden und so in das Gesamtsystem integrieren. Ein Wrapper ist ein kleines Programm, das sich zwischen Systemkern und Modell befindet, und mit beiden kommuniziert. Beim Datenaustausch zwischen Systemkern und Modell übernimmt es entsprechende Übersetzungen und Anpassungen in beide Richtungen, so dass die ausgetauschten Daten von dem jeweiligen Systemteil (Kern oder Modell) verstanden werden können. Auf diese Weise kann der Aufwand zur Anpassung bestehender Anwendungen an ein bestimmtes EUS sehr gering gehalten werden. Die Wrapper-Architektur von TAYLOR, WALKER UND ABEL, WALKER, ABEL [1999] ist in Abbildung 3.3 dargestellt.

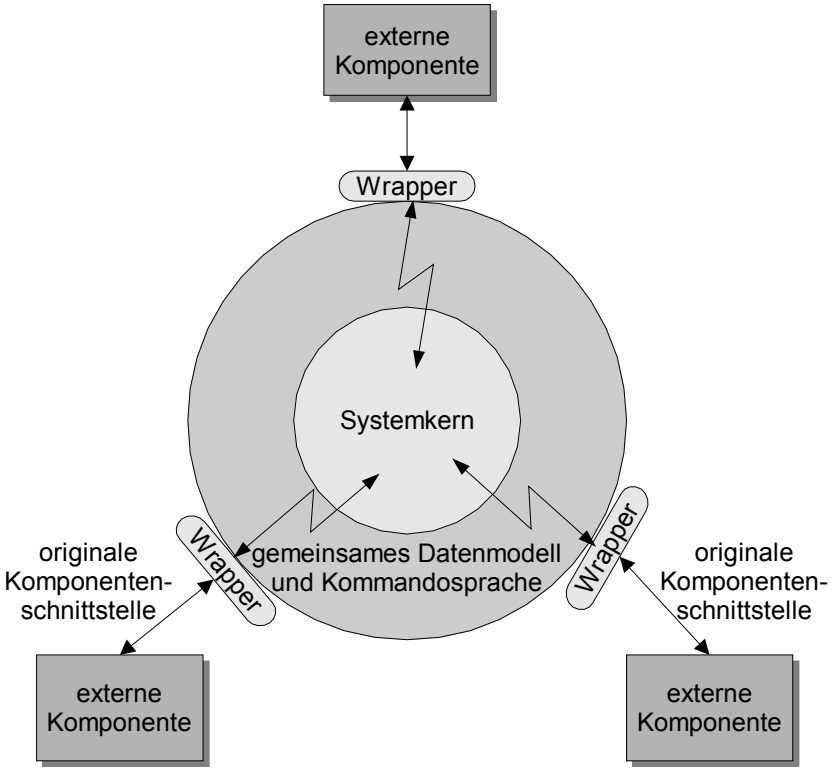


Abbildung 3.3: Wrapper-Architektur nach Taylor, Walker und Abel [1999]; verändert

Einige Konzepte zur Modell-Integration werden nachfolgend erläutert. Darüber hinaus sind weitere Beispiele unter anderem in BENNETT [1997], JANKOWSKI ET AL. [1997], CHURCH ET AL. [2000] bzw. BLAKE UND GOMAA [2005] zu finden.

Im von TAYLOR, WALKER UND ABEL [1999] beschriebenen Ansatz werden bestehende Programme in ein räumliches Entscheidungs-Unterstützungs-System integriert. Diese Programme werden als *Komponenten*, die TAYLOR, WALKER UND ABEL [1999] als *Solver* bezeichnen, in das Gesamtsystem eingebunden. Das geschieht mit Hilfe von so genannten Modelltreibern, die den zuvor beschriebenen Wrappern entsprechen, welche mit einem Systemkern kommunizieren. Die verteilte heterogene Datenhaltung der einzelnen eingebundenen Programme wird beibehalten und mit Hilfe der Modelltreiber zu einem nicht erweiterbaren objektorientierten Datenmodell vereinheitlicht. Die Modelltreiber stellen die jeweiligen Daten auf Anfrage zur Verfügung.

Bei WALLNAU UND PLAKOSH [1999] wird ein Ansatz zur Modellintegration beschrieben, bei dem *Komponenten* als dynamische Windows-Bibliotheken (*DLLs*) realisiert sind, die rudimentäre Funktionen zur Selbstdokumentation besitzen. Die Modelldaten werden mit Hilfe eines *Pipe*-Mechanismus von einer Komponente in die nächste transportiert.

Dabei müssen die transportierten Datentypen a priori festgelegt werden. Das Konzept wurde sowohl mit neu entwickelten Modellen als auch mit bereits bestehenden Modellen getestet, die mit Hilfe von Wrappern ins System eingebunden wurden.

NUTE, POTTER UND MAIER [2000] beschreiben mit NED-2 einen Ansatz, der für den Forstbereich entwickelt wurde. Zur Ansteuerung der Programme, welche in das forstliche EUS integriert sind, dienen in der Programmiersprache *Prolog* erstellte Agenten. Diesen Agenten stehen ebenfalls in Prolog vorliegende Regelsätze zur Verfügung mit deren Hilfe die dem System zur Verfügung stehenden Daten in die erforderlichen Eingabeformate der integrierten Programme umgewandelt werden können. Die verteilten Daten der Modelle werden in eine zentrale Datenbank zur späteren Auswertung übertragen. Die Kommunikation zwischen den Agenten und den Benutzeroberflächen-Modulen erfolgt indirekt über einen *Blackboard* genannten Mechanismus. Dabei werden Anfragen wie in einer Druckerwarteschlange nacheinander eingereiht und die jeweils erste Anfrage in der Liste wird abgearbeitet. Die Architektur des NED-2-Systems ist in Abbildung 3.4 schematisch dargestellt.

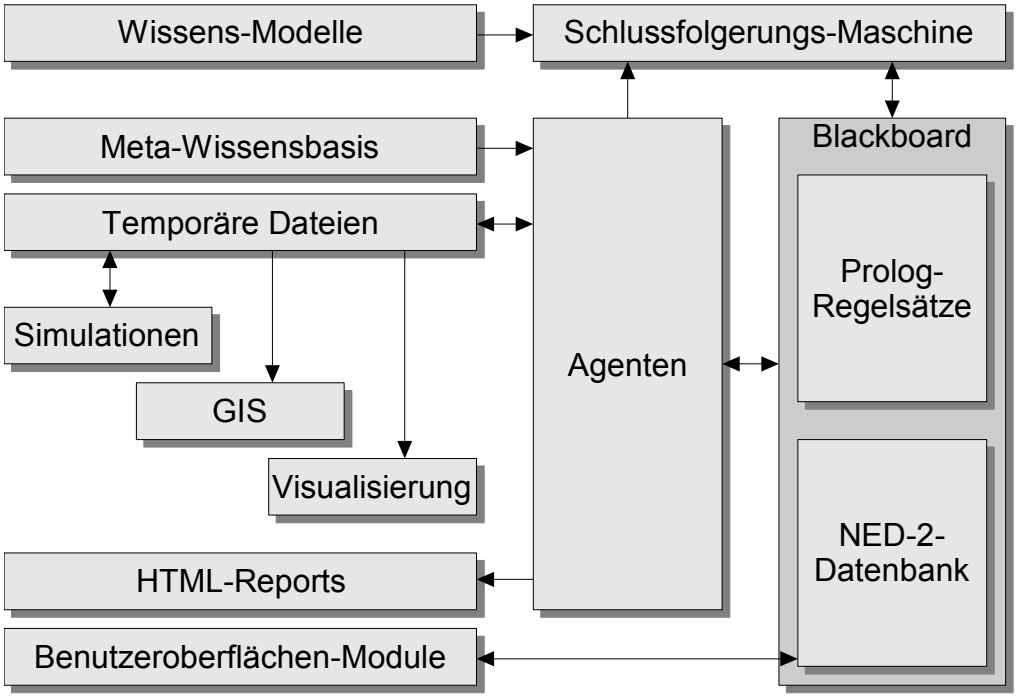


Abbildung 3.4: Schematische Darstellung der Architektur von NED-2 nach Nute, Potter und Maier [2000]; verändert

DENZER, GÜTTLER UND HELL [2002] koppeln die Komponenten über Daten- bzw. Applikationsserver, welche die Daten der zum Teil in FORTRAN programmierten

Modelle zur Verfügung stellen. Der Zugriff erfolgt indirekt über so genannte *Call-Server*, die über eine Registrierung, die als Namensserver dient, den jeweiligen Datenserver kontaktieren. Sowohl Client-Anwendungen als auch Datenserver kommunizieren mit einem zentralen Meta-Informations-System. Dabei handelt es sich um einen Mechanismus, der Daten über Daten (so genannte *Metadaten*, siehe Abschnitt 3.1.7) zur Verfügung stellt. Die Datenhaltung erfolgt dezentral. Dieses bedeutet, dass die einzelnen Komponenten über Mechanismen verfügen müssen, welche die zum jeweiligen Zeitpunkt benötigten Daten zur Verfügung stellen. Abbildung 3.5 zeigt die schematische Darstellung der Architektur dieses Ansatzes.

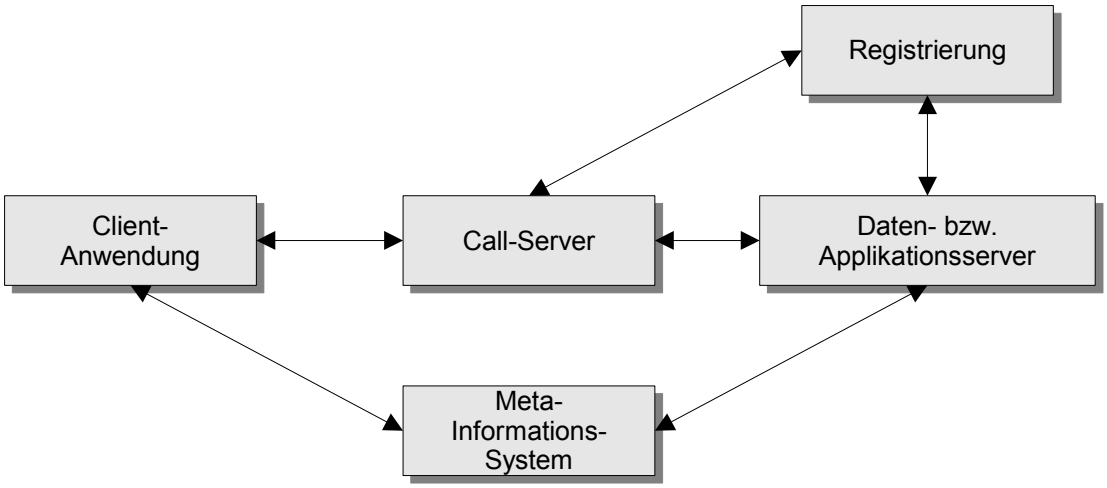


Abbildung 3.5: Die Architektur des Ansatzes nach Denzer, Güttler und Hell [2002]; verändert

Ein ähnlicher Ansatz wird auch von ENDEJAN [2002] beschrieben. Hier werden einer verteilten Datenbasis, die aus den eigentlichen Modelldaten sowie Metadaten (siehe Abschnitt 3.1.7) besteht, Komponenten, wie etwa der *Data-Object Provider* oder der *Metadata Harvester* sowie der *Metadata Manager* vorgeschaltet. Diese übernehmen die Datenverwaltung und kapseln das Datenmodell derart, dass bereits bestehende Modelle nicht umprogrammiert werden müssen.

VACIK, LEXER UND PALMETZHOFER [2004] integrieren im Rahmen einer ***C**omputergestützten **O**ptimierung von **N**utzungs**e**ingriffen im **S**eilgelände* (*CONES*) Produktivitätsmodelle für seilgestützte Holzerntemaßnnahmen, einen Waldwachstumssimulator und Modelle zur Abschätzung der Eintrittwahrscheinlichkeit von Schadereignissen. Die Module von CONES besitzt eine in C++ implementierte objektorientierte Architektur und ist in das

Geografische Informationssystem ArcGIS eingebunden. Sowohl räumliche als auch nicht-räumliche Daten werden in einer zentralen Datenbank gespeichert.

Ein EUS zur Optimierung des Waldumbaus ist in TORRES ROJO UND SANCHEZ OROIS [2005] beschrieben. Dabei werden fünf Module über ein in Visual Basic geschriebenes Interface miteinander verbunden. Der Datenaustausch zwischen den Modulen erfolgt mit Hilfe von Dateien, welche ASCII-Text enthalten. Mit Hilfe eines *Editors* werden die Eingangsgrößen vom Anwender eingegeben, welche mit Hilfe eines *Matrix-Generators* in Parameter eines nichtlinearen Gleichungssystems umgewandelt werden. Ein *Optimierer* berechnet anschließend eine optimale Lösung. Die Ergebnisse werden von einem *Reportgenerator* erzeugt.

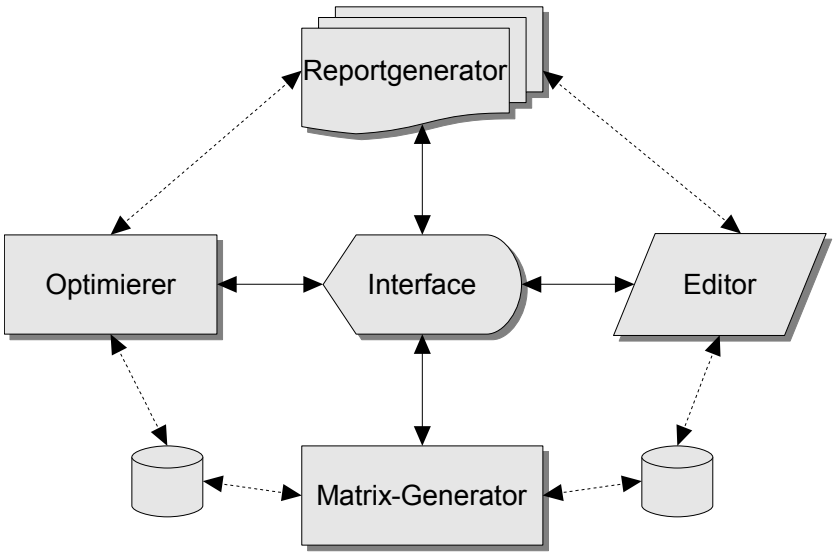


Abbildung 3.6: Die Architektur des EUS nach Torres Rojo und Sanchez Orois [2005]; verändert

Eine Umgebung für so genannte *Web-Services* aus dem Bereich der Standardsoftware für Unternehmen, wie beispielsweise ***C**ustomer **R**elationship **M**anagement* (*CRM*) oder ***E**nterprise **R**esource **P**lanning* (*ERP*), ist bei KUROPKA UND WESKE [2006] beschrieben. Neben Modulen zur automatischen Auswahl geeigneter Web-Services als Dienstlieferanten dient eine Beschreibung der Bedeutung (*Semantik*) aller im System vorhandenen Daten in Form einer *Ontologie* (siehe Abschnitt 3.1.7.2) als maßgebliches Integrationswerkzeug. Auch in diesem Ansatz wird die heterogene verteilte Datenhaltung beibehalten.

3.1.7 Metadaten

Wie in den Abschnitten zuvor beschrieben, können durch Integration verschiedener Programme und Datenquellen komplexe Umgebungen zur Entscheidungsunterstützung entstehen, wobei die Ergebnisse eines Modells als Eingabedaten für andere Modelle dienen können. Es müssen deshalb Mechanismen vorgesehen werden, die einen modellübergreifenden Datenzugriff ermöglichen, um die gespeicherten Daten anderen Komponenten zugänglich zu machen. Daher müssen zusätzlich zu den eigentlichen Modelldaten auch Daten verwaltet und gepflegt werden, welche sowohl die Eingabedaten als auch die bereitgestellten Ergebnisse jeder einzelnen Komponente beschreiben. Mit Hilfe dieser so genannten *Metadaten* (Daten über Daten, siehe z. B. GÜNTHER [1998], WEST UND HESS [2002]) ist es einer Komponente möglich, die Daten anderer Komponenten als solche zu erkennen, zu interpretieren und weiterzuverarbeiten.

3.1.7.1 Metadaten-Standards

Als Austauschformat für Daten aller Art hat sich die ***Ex**tensible **M**arkup **L**anguage* (*XML*) bereits weitgehend etabliert. XML ist eine Sprachfamilie, innerhalb derer man eigene XML-Dialekte kreieren kann. Die Definition eines XML-Dialektes, der als *XML-Instanz* bezeichnet wird, kann im wesentlichen auf zwei verschiedene Arten vorgenommen werden. Die einfachere der beiden Möglichkeiten ist die Definition einer XML mit Hilfe einer ***D**ocument **T**ype **D**efinition* (*DTD*). Die Syntax von DTDs ist sehr einfach und Dateien, die eine DTD enthalten sind relativ gut lesbar. Die Möglichkeiten einer DTD sind jedoch im Vergleich zur zweiten Möglichkeit, der Definition mit Hilfe von *XML Schema*, eingeschränkt. XML Schema bietet weitreichende Möglichkeiten der Definition einer XML. XML-Schema-Dateien enthalten sehr viel verschachtelten Text, was die Lesbarkeit ohne ein spezielles Visualisierungsprogramm erschwert. Ein großer Vorteil von XML-Schema ist, dass zur Definition einer XML wiederum XML zur Anwendung kommt.

Verschiedene Autoren (z. B. ENDEJAN [2002]) weisen auf Standards hin, die XML-Dialekte zum geregelten Austausch von Meta-Information definieren. Dazu gehören das ***R**esource **D**escription **F**ramework* (*RDF*) (MANOLA UND MILLER [2004]) sowie der Dublin Core Standard (DUBLIN CORE METADATA INITIATIVE [2007]).

3.1.7.2 Ontologien

Im *Semantic Web* Umfeld, das von Tim Berners-Lee ins Leben gerufen wurde, werden Verfahren entwickelt, mit deren Hilfe Sinn und Bedeutung (*Semantik*) des Inhalts von Internetseiten beschrieben werden kann (BERNERS-LEE ET AL. [2001]). Im Rahmen dieser Weiterentwicklung des heutigen Internet sollen mit Hilfe dieser Verfahren Möglichkeiten geschaffen werden, den Inhalt von im World-Wide-Web publizierten Dokumenten maschinell auszuwerten. Dabei gewinnen so genannte *Ontologien* (siehe z. B. ZIEGLER [2003], FLÜGGE UND SCHMIDT [2005]) zunehmend an Bedeutung. Ontologien dienen zur Definition und Darstellung von konzeptionellen Modellen, welche einen bestimmten und exakt abgegrenzten Anwendungsbereich widerspiegeln. Dabei werden die semantischen Beziehungen sämtlicher Konzepte eines Teilaspektes der Realität erfasst und formalisiert (ZIEGLER [2003]). Dies wird realisiert, indem Objekte mit bestimmten Eigenschaften definiert werden. Mit Hilfe von Regeln kann festgelegt werden, welche Werte diese Eigenschaften aufweisen müssen, damit verschiedene Objekte in einer bestimmten Beziehung zueinander stehen.

Bei der Integration von Komponenten in einem EUS oder anderen Softwaresystemen spielt die Semantik eine große Rolle (TAYLOR, WALKER UND ABEL [1999], NUTE, POTTER UND MAIER [2000], KUROPKA UND WESKE [2006]).

Es existieren formale Sprachen, mit deren Hilfe Ontologien formuliert werden können, wie die ***W**eb **O**ntology **L**anguage* (*OWL*) (MCGUINNES UND VAN HARMELEN [2004], SMITH ET AL. [2004], DEAN UND SCHREIBER [2004]). OWL ist eine XML-Instanz, sehr eng verwandt mit *RDF/XML* und als solche eine Erweiterung des ***R**esource **D**escription **F**rameworks* (*RDF*). Viele Sprachelemente von OWL bedienen sich des *RDF* bzw. des ***RDF S**chema* (*RDFS*).

3.2 Software Engineering

Ähnlich wie Bauwerke besitzen auch größere Softwaresysteme, wie beispielsweise Entscheidungs-Unterstützungs-Systeme, eine innere Struktur, welche als Architektur bezeichnet wird. Im Rahmen des Software Engineerings werden Methoden angewandt, mit deren Hilfe diese Architektur formal beschrieben und allgemein verständlich visualisiert werden kann.

Software Engineering ermöglicht eine plan- und kalkulierbare Gestaltung des Software-Entwicklungsprozesses mit dem Ziel, Software mit folgenden Merkmalen zu produzieren (PAGEL UND SIX [1994]):

- hohe Qualität
- kostengünstig, einen gewissen Budgetrahmen nicht überschreitend
- Fertigstellung zum geplanten Zeitpunkt.

Diese Vorgaben sind auch für Software im forstlichen Umfeld interessant, weshalb Methoden des Software Engineering auch zur Erstellung forstlicher Software angewandt werden (MOHR [1997]).

3.2.1 Vorgehensmodelle

Im Rahmen des Software Engineering stehen Methoden zur Verfügung, um den Entwicklungsprozess zu definieren. Einige der wichtigsten dieser so genannten Vorgehensmodelle werden im Folgenden dargestellt.

3.2.1.1 Das Phasenmodell

Dem Phasen- oder Wasserfallmodell liegt ein systematisches, sequentielles Modell zu Grunde, das den Software-Entwicklungsprozess in vier Phasen gegliedert. In PAGEL UND SIX [1994] werden sie mit *Analyse und Definition*, *Entwurf*, *Implementation*, und *Test* bezeichnet, SOMMERVILLE [1992] nennt sie *Requirements analysis and definition*, *System and software design*, *Implementation and unit testing* und *Integration and system testing*. Nicht mehr zum eigentlichen Entwicklungsprozess gehört die oft mit angeführte Phase *Einsatz und Wartung* (PAGEL UND SIX [1994]) bzw. *operation and maintainance* (SOMMVERVILLE [1992]). Abbildung 3.7 zeigt eine grafische Darstellung des Phasenmodells.

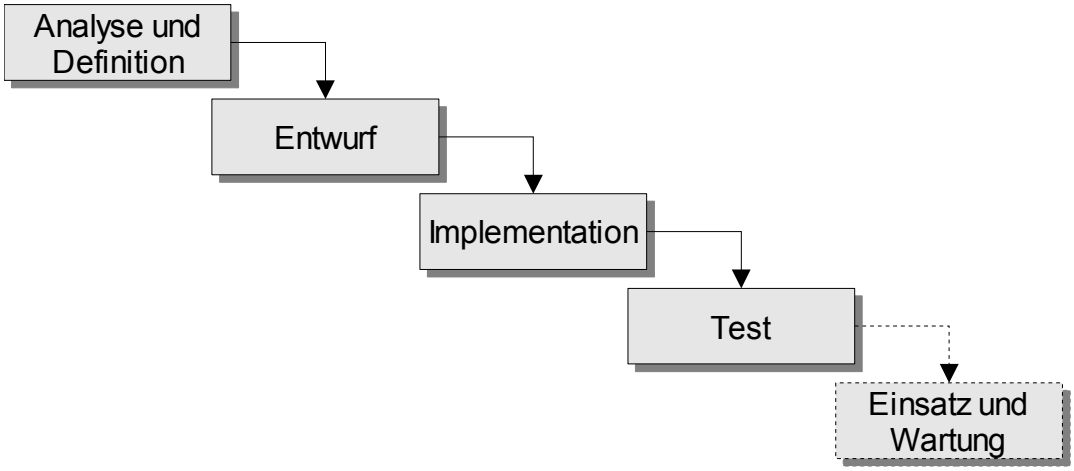


Abbildung 3.7: Das Phasenmodell nach Pagel und Six [1994]; verändert

Neben dieser klassischen Form des Phasenmodells gibt es eine Reihe von Abwandlungen dieser Vorgehensweise. Im iterierten Phasenmodell ermöglichen Rückkopplungen die Rückkehr in frühere Phasen. Dieses wird dann notwendig, wenn in einer späteren Phase Fehler einer vorangegangenen aufgedeckt werden. Zusätzlich kann das Phasenmodell um eine Prototypingphase erweitert werden, die sich zwischen Analyse und Entwurf befindet. In einem iterativen Prozess wird der Prototyp so lange revidiert, bis die Anforderungen von Auftraggeber und Benutzern feststehen. Dieser Prototyp wird in der Regel nicht weiter verwendet und verworfen. Abbildung 3.8 zeigt die grafische Darstellung des iterierten Phasenodells mit Prototypingphase.

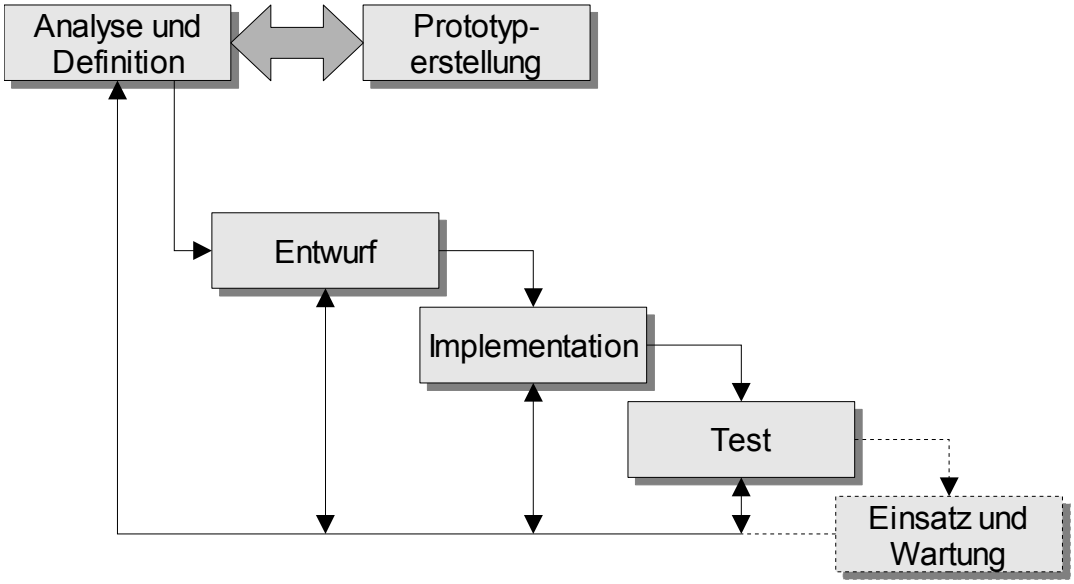


Abbildung 3.8: Das iterierte Phasenmodell mit Prototypingphase nach Pagel und Six [1994]; verändert

Die Vorteile des Phasenmodells bestehen darin, dass sich durch die Aufteilung des Entwicklungsprozesses in aufeinanderfolgende Phasen etablierte Managementtechniken zur Projektplanung und -überwachung auf die Software-Entwicklung übertragen lassen. Die Phasen Test und Wartung unterstreichen die Bedeutung von Qualität und Wartbarkeit der Software und stellen diese Kriterien nach außen dar. Durch die in sich geschlossenen Phasen lässt sich die Komplexität der Softwaresysteme bewältigen, da sich Möglichkeiten ergeben, die Ergebnisse weiter aufzuteilen.

Das Phasenmodell weist jedoch folgende Kritikpunkte auf:

- Software-Entwicklung stellt keinen linearen Prozess dar, sondern jede Phase löst Rückwirkungen auf vorangegangene Phasen aus.
- Das vorliegende Modell berücksichtigt keine unpräzisen Vorgaben oder fehlerhaften Produktdefinitionen.
- Änderungswünsche seitens des Auftraggebers können bei dieser Vorgehensweise nicht berücksichtigt werden, da das Phasenmodell keine Beteiligung des Auftraggebers über die Analyse- und Definitionsphase hinaus vorsieht.

Einige dieser Kritikpunkte werden durch die beschriebenen Abwandlungen des Phasenmodells kompensiert. Dennoch wurden mehrere andere Ansätze konzipiert, deren Ziel es vordergründig ist, die Nachteile des Phasenmodells zu vermeiden.

Das bei VACIK UND LEXER [2007] beschriebene Vorgehen zur Entwicklung eines EUS entspricht weitgehend dem iterierten Phasenmodell.

3.2.1.2 Objektorientierte Software-Entwicklung

Bei objektorientierten Vorgehensweisen werden Daten (*Attribute*) und Funktionen zu deren Manipulation (*Methoden*) zu *Klassen* zusammengefasst, die als Vorlagen für *Objekte* dienen. Objekte und Klassen stehen miteinander in Beziehungen und besitzen Verantwortlichkeiten. Als neue Abstraktionsebene mit Bezeichnungen aus dem Sprachraum des Anwenders dienen diese der besseren Veranschaulichung des Softwaremodells. Objekte besitzen daneben noch drei weitere grundlegende Eigenschaften:

- Kapselung
- Vererbung
- Polymorphismus.

Unter Kapselung wird die Eigenschaft verstanden, dass Attribute eines Objektes von außen grundsätzlich nicht zugänglich sind. Sie können nur durch Operationen, die das Objekt zur Verfügung stellt (*Methoden*), verändert werden. Mit Hilfe der Vererbung können Eigenschaften eines Objektes auf ein anderes übertragen werden, und der Polymorphismus stellt sicher, dass bei Objekten die richtigen vererbten Operationen aufgerufen werden. Der Programmcode, welcher auf Klassen und Objekten basiert, kann auf Grund dieser Eigenschaften mit wenig Aufwand wiederverwendet werden.

BOOCH [1996] teilt in der nach ihm benannten *Booch-Methode* den objektorientierten Entwicklungsprozess in einen Mikro- und einen Makroprozess auf.

Der Mikroprozess besteht aus den zyklisch aufeinanderfolgenden (*iterierenden*) Aktivitäten:

- Identifizierung von Klassen und Objekten
- Identifizieren der Klassen- und Objektsemantik
- Identifizieren der Beziehungen zwischen Klassen und Objekten
- Spezifizieren der Schnittstellen von Klassen und Objekten.

Abbildung 3.9 Stellt den Mikroprozess grafisch dar.

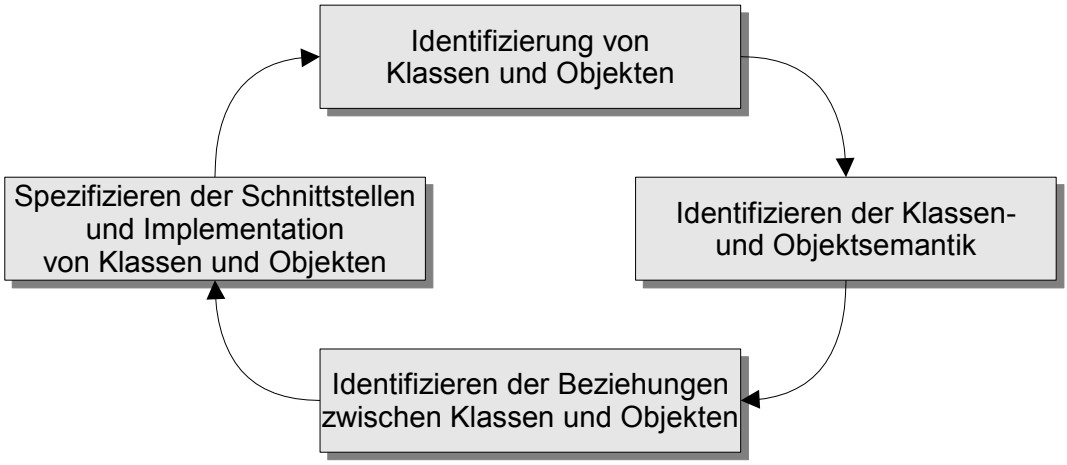


Abbildung 3.9: Der objektorientierte Mikro-Entwicklungsprozess nach Booch [1996]; verändert

Der Makroprozess stellt das organisatorische Gerüst für den Mikroprozess dar und besteht aus folgenden Aktivitäten:

- Festlegen der Kernanforderungen (Konzeptualisierung)
- Einrichten eines Modells des gewünschten Verhaltens (Analyse)
- Erzeugen einer Architektur (Design)

- Entwickeln der Implementierung (Evolution)
- Verwaltung der Entwicklung nach der Auslieferung (Wartung).

Abbildung 3.10 zeigt die grafische Darstellung des Makroprozesses. Der objektorientierte Makroprozess ist dem Phasenmodell sehr ähnlich und weist deshalb die gleichen Vor- und Nachteile auf.

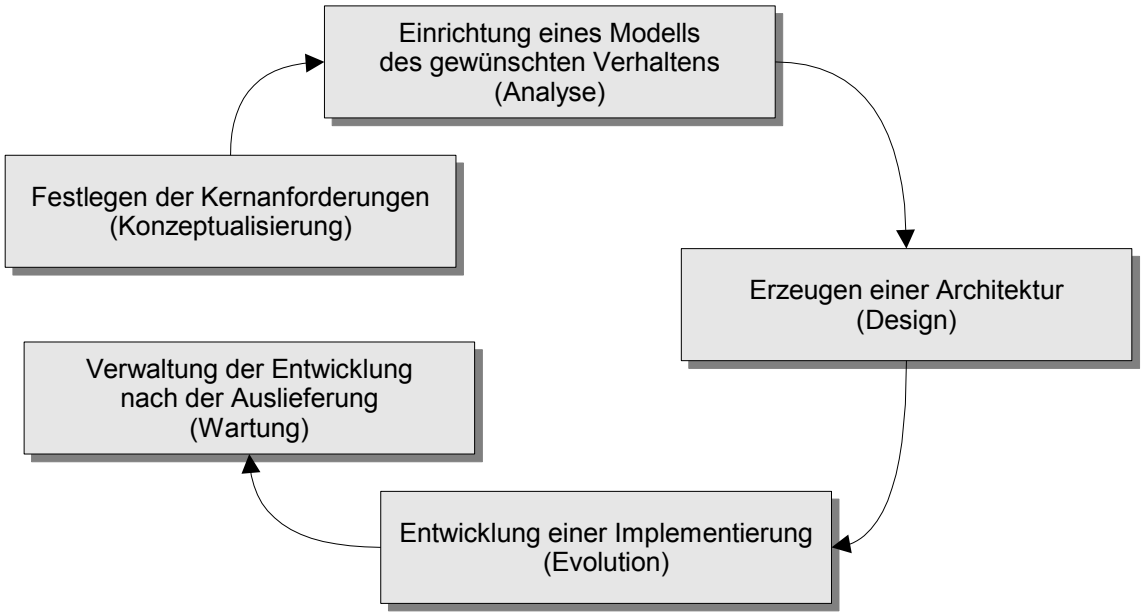


Abbildung 3.10: Der objektorientierte Makro-Entwicklungsprozess nach Booch [1996]; verändert

3.2.1.3 Evolutionäre Software-Entwicklung

Im Rahmen der evolutionären Software-Entwicklung werden in einer Folge von Entwicklungszyklen Prototypen erstellt und schrittweise weiterentwickelt, so dass am Ende eines jeden Zyklus eine verbesserte Version des Produktes steht. Eine explizite Wartungsphase existiert bei diesem Ansatz nicht.

Die Nähe zwischen Anwendungsentwickler, Auftraggeber und Benutzer ist der größte Vorteil der evolutionären Software-Entwicklung. Dadurch werden Missverständnisse und Fehler frühzeitig entdeckt und Wünsche können fortlaufend berücksichtigt werden. Andererseits bestehen wenig Möglichkeiten der fortlaufenden Dokumentation. Die ständig weiterentwickelten Prototypen müssen so programmiert sein, dass sie übersichtlich und leicht veränderbar sind. Die evolutionäre Software-Entwicklung stellt das Projektmanagement vor größere Probleme als das Phasenmodell, da sich durch die Dynamik des Entwicklungsprozesses Schwierigkeiten bei der Planung der Ressourcen

und des Projektfortschritts ergeben. Eine schematische Darstellung der evolotionären Software-Entwicklung findet sich in Abbildung 3.11.

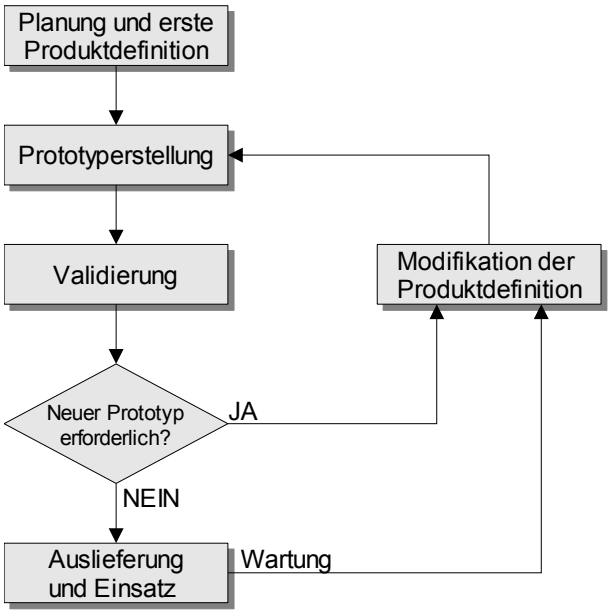


Abbildung 3.11: Evolutionäre Software-Entwicklung nach Pagel und Six [1994]; verändert

3.2.1.4 Das Spiralmodell

Das Spiralmodell versucht die Vorteile von Phasenmodell und evolutionärer Software-Entwicklung zu vereinigen, indem es den Einsatz bereits existierender Ansätze unter ständiger Kontrolle des Managements erlaubt. Jede Windung enthält die Aktivitäten:

- Festlegung von Zielen, Alternativen und Rahmenbedingungen
- Evaluierung der Alternativen, Erkennen und Reduzieren der Risiken
- Realisierung und Überprüfung des Zwischenprodukts
- Planung der Projektfortsetzung.

Am Ende jeder Windung wird der aktuelle Projektfortschritt bewertet, anschließend wird die nächste Windung geplant. Im Zuge der dritten Aktivität - *Realisierung und Überprüfung des Zwischenprodukts* - wird die eigentliche Entwicklungsmethode angewandt. Es handelt sich bei dem hier beschriebenen Modell also eigentlich um ein Metamodell, da in ihm andere Vorgehensmodelle eingebettet werden können. Eine schematische Darstellung des Spiralmodells zeigt Abbildung 3.12.

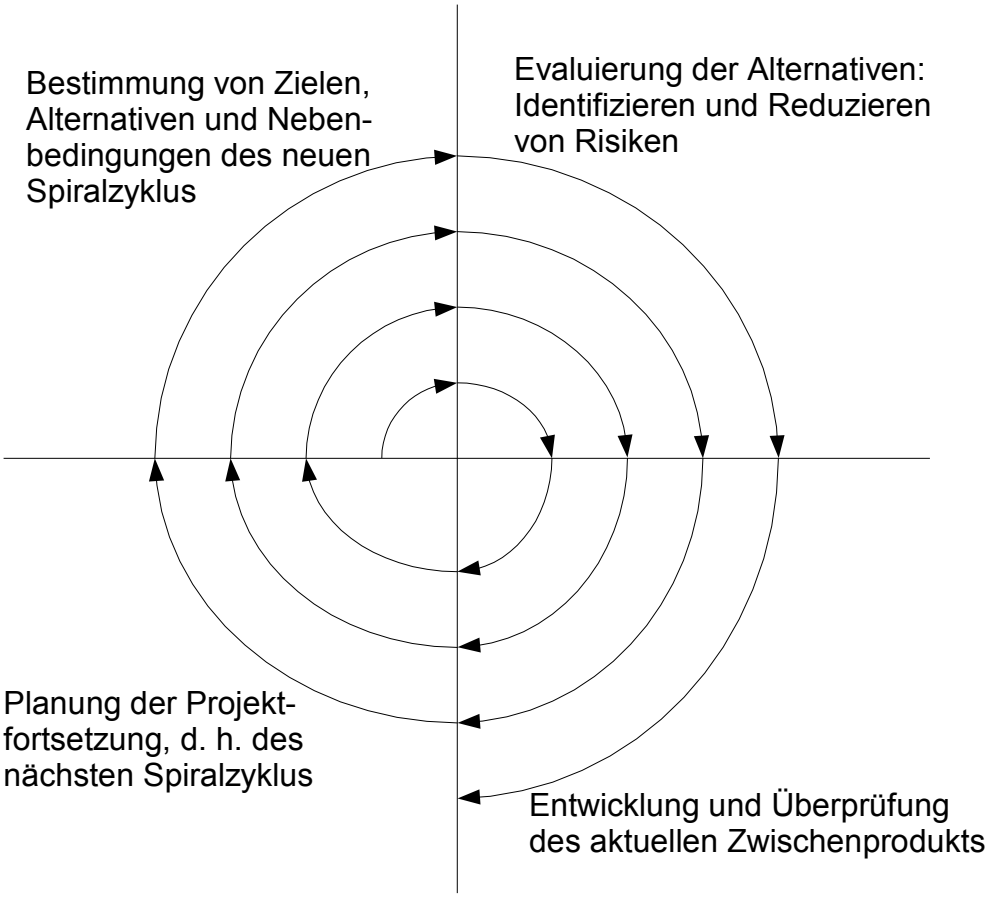


Abbildung 3.12: Das Spiralmodell nach Mohr [1997]; verändert

3.2.1.5 Der Rational Unified Process

Im Zuge der Definition einer einheitlichen Beschreibungssprache für Softwaremodelle (***U**nified **M**odelling **L**anguage*, *UML*), auf die in Abschnitt 3.2.3 näher eingegangen werden soll, wurde bei der Firma *Rational Software* der ***R**ational **U**nified **P**rocess* (*RUP*) entworfen. Der Rational Unified Process (KRUCHTEN [1996], REINHOLD UND VESTEEGEN [2002]) ähnelt in weiten Teilen der evolutionären Software-Entwicklung. Es werden ebenso mehrere *Zyklen* (*Iterations*) durchlaufen und jeweils eine neue Software-Version erstellt. Der iterative Prozess besitzt ein organisatorisches Gerüst, mit dem das Projektmanagement den Fortschritt überwachen kann. Die Tätigkeiten innerhalb der iterativen Zyklen werden im Rational Unified Process *Disziplinen* (*Disciplines*) genannt, das organisatorische Gerüst wird in vier *Phasen* (*Phases*) aufgeteilt. In KRUCHTEN [1996] werden die Disziplinen *Planung* (*Planning*), *Analyse* (*Analysis*), *Architektur* (*Architecture*), *Entwurf* (*Design*), *Implementation*, *Integration* und *Test* (*Assessment*) sowie die Phasen *Anfang* (*Inception*), *Ausarbeitung* (*Elaboration*), *Konstruktion* (*Construction*) und *Übergang* (*Transition*) unterschieden. In der fünften Phase

Evolution kann der gesamte Prozess für eine neue Softwaregeneration wiederholt werden. Die Gewichtung jeder einzelnen Disziplin kann für jeden einzelnen Zyklus neu bestimmt werden. In einer Matrix angeordnet ergibt sich ein Diagramm, wie es in Abbildung 3.13 dargestellt ist.

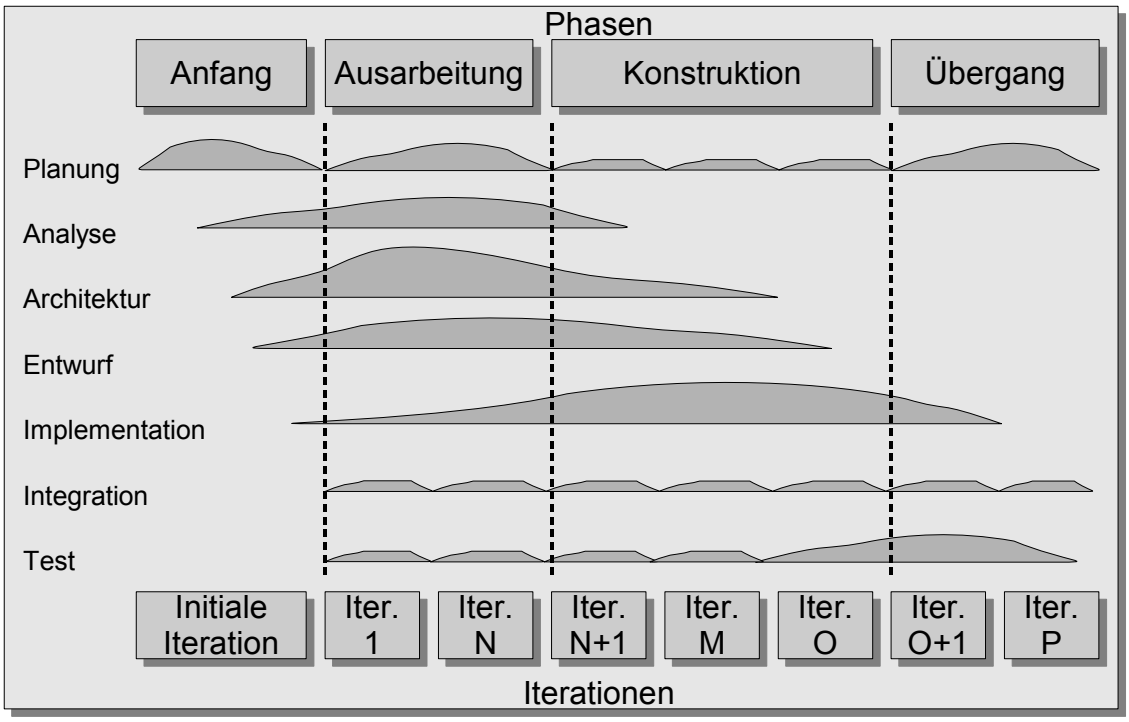


Abbildung 3.13: Phasen und Disziplinen des Rational Unified Process nach Kruchten [1996]; verändert

Ein großer Nachteil dieses Verfahrens ist die enge Bindung an die Firma *IBM Corporation*, die nach der Übernahme von *Rational Software* den Rational Unified Process als kommerzielles Produkt vermarktet.

3.2.1.6 Extreme Programming

Die wesentlichen Arbeitsschritte eines Projekts bei dem von Kent Beck entworfenen *Extreme Programming* (*XP*) beschränken sich auf *Versionsplanung* (*Release Planning)*, *Iterationen* mit Programmierarbeiten (*Iteration*) sowie Abnahmetests für die einzelnen *Versionen* (*Releases*). Dabei konzentrieren sich die Entwickler ganz auf die eigentliche Programmierarbeit. Analysearbeiten werden nicht sehr intensiv betrieben oder es wird ganz darauf verzichtet. Dieses Vorgehen folgt aus der Zielsetzung, die Software-Versionen möglichst schnell zu erzeugen. So bekommt der Kunde sehr früh ein Programm, das seine Anforderungen bereits zum Teil erfüllt, und er kann ebenso früh

Einfluss auf dessen weitere Entwicklung nehmen. Bei Unklarheiten werden *Schnellschüsse* (so genannte *Spikes*) erzeugt. Das sind ganz einfache Programme, die keinerlei Randbedingungen berücksichtigen, nur ganz spezielle Aufgaben behandeln und wieder verworfen werden. Sie sollen helfen, die zu Grunde liegenden Probleme auszuloten und Lösungsmöglichkeiten dafür zu erproben.

Die Versionsplanung erfolgt als so genanntes *Planspiel* (*Planning Game*), bei dem der Auftraggeber Probleme schildert, die das Programm lösen soll. Diese Problembeschreibungen werden in XP *User Stories* genannt. Für jedes Teilproblem wird eine User Story formuliert und auf eine Karteikarte geschrieben. Jede dieser Problembeschreibungen sollte innerhalb von ein bis drei Wochen realisiert werden können. Der komplette Programmablauf ergibt sich aus der Vereinigung aller User Stories. Für den weiteren Projektfortschritt werden gemeinsam von Projektteam und Auftraggeber Prioritäten für die User Stories festgelegt. Die Prioritäten bestimmen die Reihenfolge der Implementierung.

Die Problembeschreibungen werden in mehreren Iterationen implementiert. Das geschieht, indem vor dem eigentlichen Programm automatische Tests realisiert werden und erst nach der Erstellung der Tests in Paaren zu zweit an einem Rechner programmiert wird (*Pair Programming*). Nach jedem Zyklus werden die Programmteile Stück für Stück integriert. Dabei werden die zuvor erstellten Tests ausgeführt, und das entstandene Teilprogramm muss 100% der Tests bestehen. Erfolgen während des Projektes Änderungswünsche, so werden neue User Stories geschrieben. Die Programmierer bilden bei jeder Iteration neue Paare, so dass jeder Entwickler mit der Zeit den gesamten Programmcode kennt. Neben dem Programmquelltext werden während der Programmierung auch vereinzelt so genannte *Klassen-Verantwortlichkeits-Kollaborations-Karten* (*Class-Responsibility-Collaboration-Cards, CRC-Karten*) erstellt, beispielweise nach Programmierung eines Schnellschusses. Die CRC-Karten dienen in der objektorientierten Modellierung dazu, einzelne Objektklassen anhand von Name, Verantwortlichkeit und anderen benötigten (kollaborierenden) Klassen zu beschreiben.

Hauptziel ist, die Programme so einfach wie möglich zu halten (»*do the simplest thing that could possibly work*« (Wegener [1999])). Die Entwickler sind deshalb angehalten, den Programmcode zu *refaktorieren* (*Refactoring*). Dies bedeutet, dass ein Modul

umgeschrieben und einfacher implementiert wird, sobald es zu kompliziert erscheint. Diese Refaktorierung muss schonungslos (»*mercilessly*« (WEGENER [1999])) erfolgen. Der Programmcode stellt ein kollektives Eigentum des gesamten Entwicklerteams dar, so dass es erlaubt ist, denjenigen Code zu ändern, den ein anderer Entwickler erstellt hat, was bei der Refaktorierung auch notwendig ist. Der Projektfortschritt kann anhand der implementierten User Stories überwacht werden.

Die wesentlichen Elemente des Extremen Programmierens, wie zum Beispiel die Versionsplanung, das Programmieren in Paaren oder die kontinuierliche Integration von Code, wirken zusammen, ergänzen sich gegenseitig und werden als *Praktiken* (*Practices*) bezeichnet.

Extreme Programming wird kontrovers diskutiert. Die Vorteile von XP liegen in der Betonung des einfachen Designs und des Testens. Die Programmierung in Paaren ist ein Instrument zur Qualitätssicherung. Andererseits kann die paarweise Programmierung auch Probleme zwischenmenschlicher Art hervorrufen. Der Architekturbegriff ist in XP schwach definiert (es existiert lediglich eine System-Metapher). Dies wird weiter verstärkt durch das Refaktorieren, wodurch es passieren kann, dass zwei Projekte, die ursprünglich die selbe Komponente benutzten, nach einiger Zeit mit deutlich unterschiedlichen Komponenten arbeiten. Diesem Problem kann man nur mit einer rigorosen Wiederverwendungs-Strategie entgegenwirken. Auch die Dokumentation wird beim Extremen Programmieren sehr vernachlässigt, die Anforderungen an die Software befinden sich beispielsweise nur auf den zuvor erwähnten Karteikarten in Form einer Niederschrift der Problembeschreibungen. Extremes Programmieren ist in den Abbildungen 3.14 bis 3.17 schematisch dargestellt.

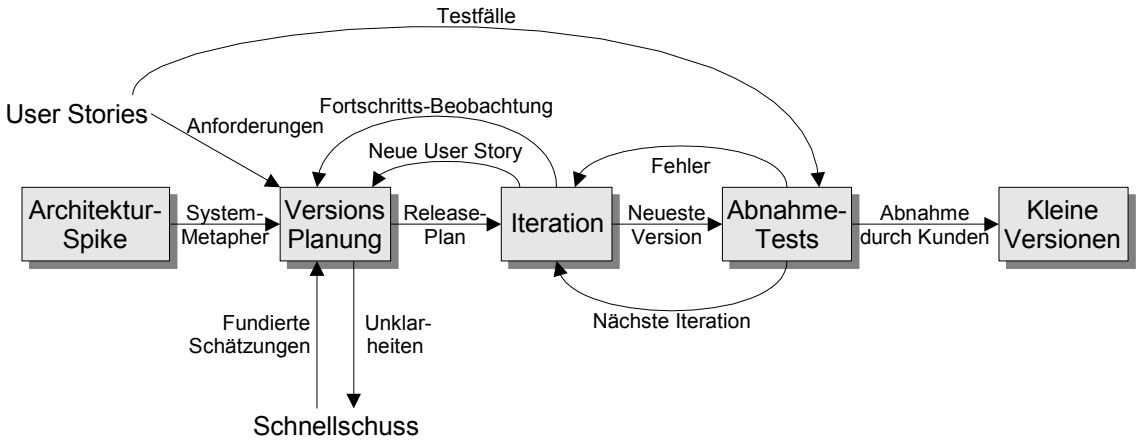


Abbildung 3.14: Arbeitsschritte des Extreme Programming nach Wells [2006]; verändert

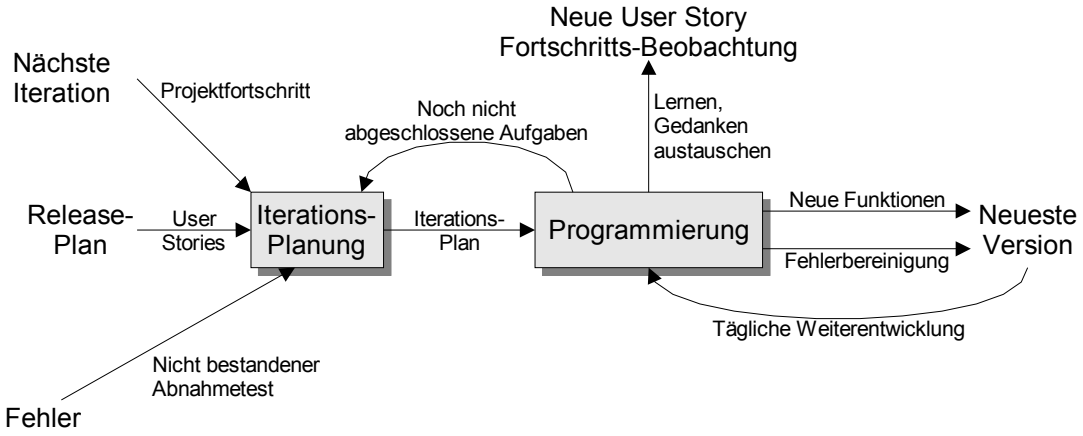


Abbildung 3.15: Extreme Programming - Iteration nach [Wells 2006]; verändert

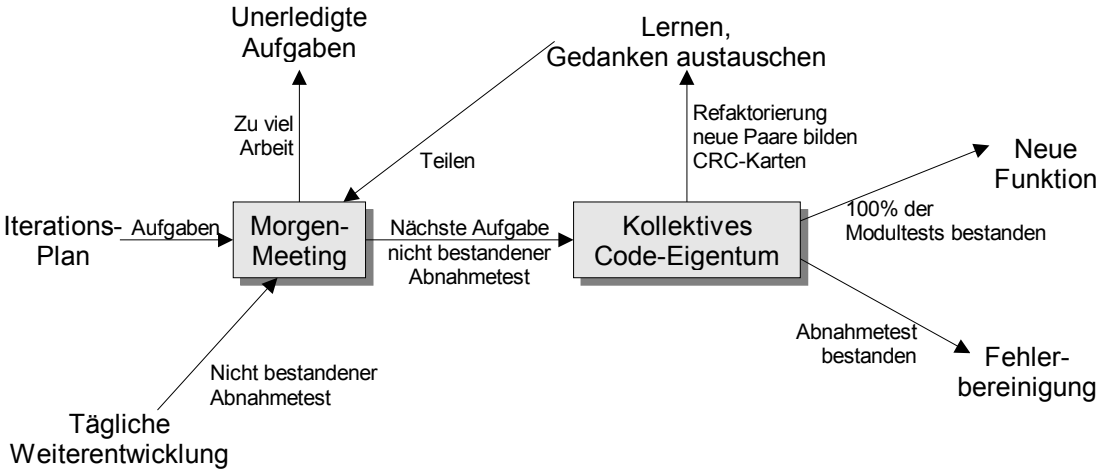


Abbildung 3.16: Extreme Programming - Programmierung nach Wells [2006]; verändert

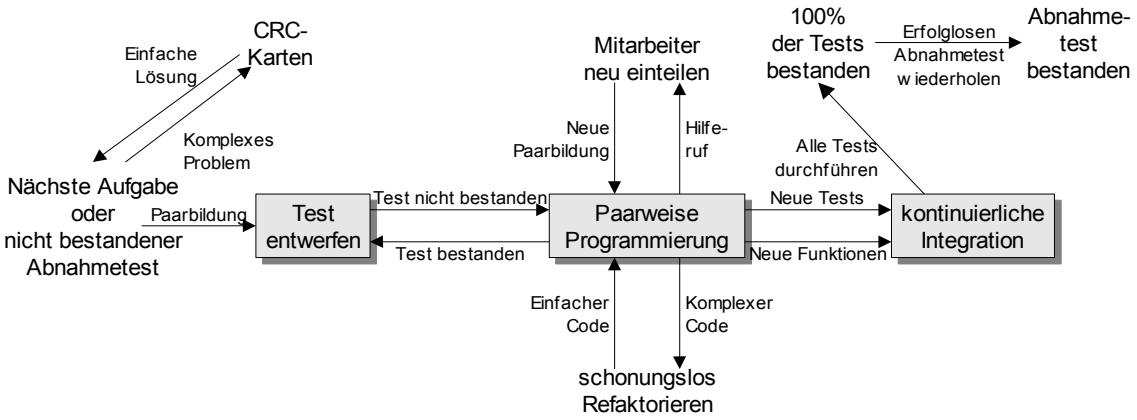


Abbildung 3.17: Extreme Programming - Kollektives Code-Eigentum nach Wells [2006]; verändert

In WELLS [2006] bzw. ELTING UND HUBER [2001] ist extremes Programmieren detailliert beschreiben.

3.2.2 Entwurfsmuster

Wiederverwendung spielt sowohl auf der Ebene der Implementation als auch auf der Ebene von Analyse und Entwurf eine Rolle. GAMMA ET AL. [1995] haben einen Katalog von wiederkehrenden Designelementen erstellt und dafür den Begriff *Entwurfsmuster (Design Patterns)* geprägt. In GAMMA ET AL. [1995] werden Entwurfsmuster folgendermaßen definiert: »*A Design Pattern names, abstracts, and identifies the key aspects of a common design structure that makes it useful for creating a reusable object-oriented design.*«

Entwurfsmuster im oben beschriebenen Sinne bestehen aus vier Teilen:

- dem *Musternamen*
- der Beschreibung des *Problems*, zu dessen Lösung es eingesetzt werden kann
- der Beschreibung der *Lösung*, d. h. der Entwurfselemente, deren Beziehungen, Verantwortlichkeiten und Interaktionen
- der Beschreibung der *Konsequenzen*, also der Vor- und Nachteile der Benutzung des Musters (an ihnen kann man sich bei der Abwägung zwischen Alternativentwürfen orientieren).

Die Design Patterns im Katalog von GAMMA ET AL. [1995] zeichnen sich durch weitgehende Allgemeingültigkeit und saubere Definition aus, weshalb diese oft als Referenzen herangezogen werden (SIEDERSLEBEN UND DENERT [2000]). Während der

Analyse- und Entwurfsphase sollte auf Ähnlichkeiten mit diesen Mustern geachtet und gegebenenfalls sollten Teile des konzeptionellen Modells durch diese ersetzt werden.

3.2.3 Notation

Da mit fast jedem Vorgehensmodell die Definition einer eigenen Notation einher ging, haben sich im Laufe der Zeit eine Reihe von Notationen etabliert. Grady Booch, Ivar Jacobson und Jim Rumbaugh haben die Notationen ihrer Vorgehensmodelle - *Booch-Methode*, *OOSE* (***O**bject **O**riented **S**oftware **E**ngineering*) und *OMT* (***O**bject **M**odelling **T**echnique*) - zusammengeführt und ***U**nified **M**odelling **L**anguage* (*UML*) genannt. Bereits 1997 wurde UML 1.0 zur Standardisierung bei der ***O**bject **M**anagement **G**roup* (*OMG*) eingereicht. Die Autoren von Gegenentwürfen erkannten die Tragweite von UML und arbeiteten als UML-Partner an einer Konsolidierung von UML 1.0, so dass kurz darauf UML 1.1 bei der OMG eingereicht wurde. Zur Zeit ist der OMG-Standard UML 2.0 (Object Management Group [2002]). Wegen der Schwerpunktsetzung der drei Methoden (Booch-Methode, OOSE sowie OMT), welche der UML zu Grunde liegen, eignet sich UML sowohl für die Modellierung von Objekten mit komplexen Beziehungen untereinander als auch für die Modellierung von Abläufen mit Nebenläufigkeits- und Echtzeitanforderungen. So wie Baupläne für elektronische Geräte und Bauwerke nach bestimmten gleichbleibenden Regeln aufgebaut sind, so dass sie von sehr vielen Menschen verstanden werden können, existiert mit UML auch eine *lingua franca* für die objektorientierte Software-Entwicklung (Hitz und Kappel [1999]).

Mit UML kann sowohl die statische Struktur als auch das dynamische Verhalten der zu entwickelnden Software in Form von Modellen formuliert werden. Jedes Modell stellt die Abstraktion eines Aspektes der zu entwickelnden Software dar. Ein Diagramm visualisiert eine Sicht auf ein Modell. In UML gibt es zwölf verschiedene Diagrammarten (Object Management Group [2002]).

Zur Repräsentation von statischen Modellen gibt es vier Strukturdiagramme:

- Klassendiagramm
- Objektdiagramm
- Komponentendiagramm
- Verteilungsdiagramm.

Fünf Verhaltensdiagramme repräsentieren dynamische Modelle:

- Anwendungsfalldiagramm
- Sequenzdiagramm
- Aktivitätsdiagramm
- Kollaborationsdiagramm
- Zustandsdiagramm.

Daneben gibt es drei Modellmanagementdiagramme:

- Paketdiagramm
- Subsystemdiagramm
- Modelldiagramm.

UML beinhaltet durch die Vielzahl an Diagrammen gewisse Redundanzen und bietet dadurch Modellierungsvariationen.

UML kann unabhängig von

- Methode,
- Entwicklungsprozess,
- Modellierungswerkzeugen,
- Modellierungsrichtlinien und
- Programmiersprache

eingesetzt werden.

Abschnitt 4.1.1 gibt einen Überblick über die im Rahmen der vorliegenden Arbeit verwendeten Diagramme. Eine umfassende Beschreibung der Unified Modelling Language ist in HITZ UND KAPPEL [1999] zu finden.

3.2.4 Das Komponentenparadigma

In vielen Ingenieurwissenschaften werden komplexe Strukturen aus einfachen Bausteinen zusammengesetzt. Diese Vorgehensweise wurde in Form des Komponentenparadigmas auf die Softwareindustrie übertragen. Dahinter verbirgt sich die Idee, Softwaresysteme aus wiederverwendbaren Bausteinen zusammenzusetzen. Solche Bausteine werden als Softwarekomponenten bezeichnet.

SZYPERSKI [1998] charakterisiert Softwarekomponenten folgendermaßen:

- Eine Komponente ist eine unabhängige Funktionseinheit
- Eine Komponente ist eine Einheit, die von Dritten zum Aufbau von Systemen verwendet werden kann
- Eine Komponente hat keinen dauerhaften Status.

Diese Eigenschaften implizieren noch weitere Merkmale von Softwarekomponenten. Auf Grund der Unabhängigkeit einer Komponente muss diese von ihrer Umgebung und anderen Komponenten abgeschottet werden und es dürfen keine Seiteneffekte auftreten. Eine Komponente kann nur als Ganzes eingesetzt werden, da sie eine Funktionseinheit darstellt, ferner kann sie von Dritten - also von Personen, die den internen Aufbau der Komponente nicht kennen - zum Aufbau von Softwaresystemen eingesetzt werden. Dem Rechnung tragend müssen Vor- sowie Nachbedingungen genau spezifiziert werden. Eine Komponente muss ihre Implementierung einkapseln und nach außen über eine definierte Schnittstelle kommunizieren. Da diese Schnittstelle nicht verändert werden darf, kann sie auch als Vertrag zwischen der Komponente und ihrem Benutzer verstanden werden. Weil eine Komponente keinen dauerhaften Status besitzt (z. B. in Form von Attributen, die bestimmte Werte haben), können mehrere Kopien der selben Komponente im Hauptspeicher zu keinem Zeitpunkt voneinander unterschieden werden. Deshalb kann der Zeitpunkt zu dem eine Komponente ins System geladen und aktiviert wird, beliebig gewählt werden. SZYPERSKI [1998] definiert eine Softwarekomponente folgendermaßen: »*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*«

Die Komponentenorientierung hat ebenfalls Einfluss auf die Software-Entwicklung. Einerseits werden Komponenten entwickelt, andererseits werden Komponenten zu Anwendungssystemen zusammengesetzt. Dadurch können in Zukunft eine Reihe neuer Programmierdisziplinen entstehen, zum Beispiel *component system architect* oder *component assembler* (SZYPERSKI [1998]) bzw. *Komponentenlieferant* sowie *Anwendungsentwickler* (GRIFFEL [1998]).

Durch Anwendung des komponentenorientierten Ansatzes besteht die Möglichkeit, in Zukunft Programme zu entwickeln, deren Algorithmen sich nicht mehr im Programmcode der Komponenten befinden, sondern sich in den Kommunikationsmustern der

zwischen den einzelnen Komponenten ausgetauschten Nachrichten manifestieren. Es wird vermutet, dass solche Programme prinzipiell Programmen, die nicht auf Komponenten basieren, überlegen sind (WEGENER [1997] in GRIFFEL [1998]). Wenn diese Vermutung zuträfe, wäre es möglich, mit komponentenbasierten Programmen Probleme zu lösen, von denen heute noch angenommen wird, dass sie mit Computern prinzipiell nicht lösbar sind.

In vielen Ansätzen zur Integration von Entscheidungs-Unterstützungs-Modulen, die in Abschnitt 3.1.6 beschrieben sind, werden diese als Komponenten bezeichnet und weisen vielfach Architekturmerkmale von *Software-Komponenten* auf. Beim Entwurf und der Realisierung eines räumlichen Entscheidungs-Unterstützungs-Systems für forstliche Fragestellungen sollte deshalb das *komponentenorientierte Paradigma* berücksichtigt werden. Zudem bewerten LEMM, ERNI UND THEES [2002] die Einsatzmöglichkeiten von komponentenorientierter Software bei Fragestellungen der Forstlogistik als vielversprechend.

3.2.5 Komponenten-Technologien

Für die Umsetzung des Komponentenparadigmas existieren Technologien, die eine Kommunikation zwischen Komponenten sicherstellen. Im wesentlichen sind dies die Technik des Entfernten Methodenaufrufs (***R**emote **M**ethod **I**nvocation*, *RMI*) aus dem JAVA-Umfeld, der OMG-Standard *CORBA* (***C**ommon **O**bject **R**equest **B**roker **A**rchitecture*, siehe z. B. STAL [1995], LAUKIEN UND RESENDES [1998]), sowie die COM-Technologie (*ActiveX* bzw. *.NET*, sprich: »Dot Net«) aus dem Microsoft-Umfeld. Die genannten Technologien realisieren die Kommunikation zwischen Komponenten unterschiedlich und sind deshalb zueinander inkompatibel. Es sind jedoch Schnittstellen verfügbar, die zwischen den Techniken vermitteln und die Interoperabilität sicherstellen.

Unter Zuhilfenahme des Komponentenparadigmas ist es möglich, Softwaresysteme zu entwerfen und zu realisieren, die einem hohen Qualitätsstandard genügen. Die Software-Entwicklung ist auf jede einzelne Komponente und damit auf einen kleinen Teil des Programmquelltextes begrenzt. Fehler können auf diesen Bereich eingegrenzt und schneller gefunden werden. Bei kleineren Programmeinheiten ist es für den Programmierer einfacher, den Überblick zu behalten, so dass Fehler vermieden werden können. Die Komponentenschnittstellen gewährleisten in Verbindung mit den

Komponententechnologien die Flexibilität, die bei der rasanten Weiterentwicklung der Hard- und Softwaretechnologien notwendig ist. Durch Austausch oder Hinzufügen neuer Komponenten können neue Technologien bzw. neue Funktionen in die Software integriert werden. Dabei ist es nicht notwendig, die gesamte Software neu zu erstellen.

SZYPERSKI [1998] und GRIFFEL [1998] beschreiben das Komponentenparadigma und die an dieser Stelle grob skizzierten Komponententechnologien detailliert.

3.2.6 Mehrschichtige und serviceorientierte Software-Architekturen

Komplexen Softwaresystemen ist in der Architektur die Trennung von Datenhaltung, Programmlogik und Benutzerschnittstelle gemeinsam. In dieser so genannten Drei-Schichten-Architektur (siehe EDWARDS UND DEVOE [1997]) bzw. Mehr-Schichten-Architektur (*multi-tiered-architecture*) liegt auf der untersten Ebene die Schicht der Datenhaltung, darauf setzt die Schicht der Programmlogik auf. Auf der obersten Ebene ist die Schicht der Benutzerschnittstelle angesiedelt. Die Zugriffe zwischen den einzelnen Schichten erfolgen über einheitliche Schnittstellen. Dieser Aufbau entspricht dem in Abschnitt 3.1.4 beschriebenen Aufbau eines EUS aus IT-orientierter Sicht. Dabei entspricht das Datensubsystem der Datenschicht, das Modellsubsystem der Programmebene und das Dialogsubsystem der Schicht der Benutzerschnittstelle.

In großen unternehmensweiten Softwaresystemen setzt sich die Programmlogik oft aus unterschiedlichen Anwendungen zusammen. Im Rahmen der unternehmensweiten Anwendungs-Integration (***E**nterprise **A**pplication **I**ntegration*, *EAI*) wird das Zusammenwirken dieser Applikationen verbessert, indem sie über eine gemeinsame Benutzeroberfläche bedient werden können. Gleichzeitig werden die Programme so umgestaltet, dass sie ihre Funktionen als verteilte *Dienste* über das Internet bzw. Intranet zur Verfügung stellen (*Web-Services*). Mit Hilfe einer Zwischenschicht (*Middleware*) werden diese Dienste verwaltet und Geschäftsprozesse als Aneinanderreihung von Diensten abgebildet. Diese so genannten ***s**ervice**o**rientierten **A**rchitekturen* (*SOA*) werden bei KUROPKA UND WESKE [2006] sowie BURBECK [2000] näher beschrieben.

International anerkannte Standards gewinnen bei den Schnittstellen zwischen den einzelnen Schichten zunehmend an Bedeutung. Beim Übergang zwischen Programmebene und Datenebene haben sich bereits Datenzugriffsmechanismen, wie

zum Beispiel ***O**pen **D**ata**b**ase **C**onnectivity* (*ODBC*) oder ***J**ava **D**ata**b**ase **C**onnectivity* (*JDBC*) etabliert, die standardisierte Zugriffsmöglichkeiten auf Daten ermöglichen, welche in Datenbanken gespeichert sind. Im Bereich des Übergangs zwischen Programmebene und Benutzeroberfläche zeichnen sich Standardisierungsbestrebungen ab. Als Beispiele seien an dieser Stelle das ***S**imple **O**bject **A**ccess **P**rotocol* (*SOAP*) sowie die *Web-Services* erwähnt.

3.3 Schlussfolgerungen

In vielen der in Abschnitt 3.1 vorgestellten EUS-Konzepte werden Metainformationen verwaltet, welche die im EUS vorliegende Datenbasis beschreiben (siehe Abschnitt 3.1.7). Einige Ansätze sehen deren standardisierte Speicherung in Form von XML-Dialekten vor. Wird anstatt dieser Metainformationen ein *anwendungsübergreifendes abstraktes Datenmodell* in Form einer in OWL formulierten Ontologie realisiert, ergeben sich neue Möglichkeiten in der Entwicklung von Entscheidungs-Unterstützungs-Systemen, denn erstmals stehen die *syntaktische* und die *semantische* Beschreibung der EUS-Daten gleichzeitig in standardisierter Form zur Verfügung. Neben den beschreibenden Daten, welche Informationen sowohl über die EUS-Daten selbst als auch über deren taxonomische Beziehungen abbilden können, besteht die Möglichkeit, Regelsysteme zu realsieren, welche zum Verständnis des im EUS verfügbaren Datenmaterials beitragen können. Ontologien können ferner als Grundlage intelligenter Softwaresysteme dienen. Computergestützte Ontologien stellen ein relativ junges Forschungsfeld dar und deren Einbindung in Entscheidungs-Unterstützungs-Systeme seht erst am Anfang.

Einige der in Abschnitt 3.1 beschriebenen EUS-Anwendungen setzen sich aus *Software-Komponenten* zusammen (siehe Abschnitt 3.2.4). Neben der Möglichkeit, diese Komponenten in einem Netzwerk zu verteilen, können einzelne Software-Komponenten dynamisch hinzugefügt, entfernt oder ausgetauscht werden, wodurch die Flexibilität des EUS erheblich steigt. Dies ist vor allem dann von Nutzen, wenn der Entwicklungsprozess des EUS wie bei Vacik und Lexer [2007] von vielen Rückkopplungen geprägt ist und das EUS ständig weiterentwickelt werden soll. Neue bzw. geänderte Funktionen können entweder als neue Komponenten oder als neue Versionen bereits vorhandener Komponenten in das EUS integriert werden. Wegen der einheitlichen Aktivierungs-

schnittstelle können die erstellten Komponenten auch in anderen EUS-Anwendungen eingesetzt werden. Entscheidungs-Unterstützungs-Systeme, welche aus Software-Komponenten aufgebaut sind, besitzen ein heterogenes und verteiltes Datenhaltungskonzept (siehe z. B. DENZER, GÜTTLER UND HELL [2002], ENDEJAN [2002], SENGUPTA UND BENNETT [2003], BLAKE UND GOMAA [2005]). Dabei muss jedes Modell über eigene Mechanismen verfügen, mit deren Hilfe die verteilt vorgehaltenen Daten von Außen erreichbar sind, Aspekte der Datensicherheit und des Zugriffs-Schutzes müssen unter Umständen ebenfalls berücksichtigt werden. Dies erhöht die Komplexität der Modelle erheblich. Ein zentrales Datenhaltungskonzept bietet daher den Vorteil, dass sich Modell-Entwickler nahezu vollständig auf die fachliche Anwendungslogik konzentrieren können. Zudem besteht die Möglichkeit, mächtige Datenbank Management Systeme (DBMS) einzusetzen, welche über standardisierte Zugriffsverfahren, eine robuste Benutzerverwaltung und Möglichkeiten zur automatisierten Datensicherung verfügen. Die Kombination von komponenten-orientierter Softwarearchitektur und zentralem Datenhaltungsmodell wird im EUS-Umfeld bislang noch nicht eingesetzt.

Um die an dieser Stelle beschriebenen Lücken zu schließen, können im Rahmen der konzeptionellen bzw. programmtechnischen Realisierung die in Abschnitt 3.1.7.2 vorgestellten Ontologien sowie die in Abschnitt 3.2 beschriebenen Erkenntnisse aus dem Bereich der Informatik verwendet werden.

4 Material und Methoden

Wie in Abschnitt 3.2.4 beschrieben, ist ein komponentenorientiertes Vorgehen sehr gut geeignet, um verschiedene Problemlösungsmodule zu integrieren. Module, die als Komponenten realisiert sind, können in eine Systemumgebung eingebunden werden, die deren Zusammenwirken steuert und als Integrationsplattform dient. Die Modellierung dieser Entscheidungs-Unterstützungs-Komponenten, im Folgenden als *Solver* bezeichnet (siehe TAYLOR, WALKER UND ABEL [1999]) als solche kann nach dem objekt-orienterten Paradigma erfolgen, so dass die Verwendung standardisierter Entwurfs-muster möglich ist.

Bei der zuvor beschriebenen Vorgehensweise entsprechen die Solver den EUS-Werkzeugen der in Abschnitt 3.1.2 erläuterten technik-orientierten Sicht auf ein EUS. Sie werden mit Hilfe der Integrationsplattform, die dem EUS-Generator entspricht, zu einem speziellen EUS zusammengesetzt.

4.1 Gewähltes Vorgehensmodell

Als Vorgehensmodell wird auf Grund seiner geringen Komplexität und seiner einfachen Umsetzbarkeit ein modifiziertes Wasserfallmodell mit Prototypingphase verwendet. Die einzelnen Phasen sind an eine komponentenorientierte Vorgehensweise angepasst.

Ausgehend von einer *Anforderungsanalyse* erfolgt die Definition und schrittweise Präzisierung der *Anwendungsarchitektur*, die mit Hilfe von Prototypen verifiziert wird. Auf die anschließende Spezifikation der *Komponentenarchitektur* folgen *Komponentenentwurf*, *Komponentenimplementierung* sowie *Komponententest*. Der *Integrationstest* am Schluss dient der Überprüfung des Zusammenwirkens der Komponenten. Abbildung 4.1 zeigt eine schematische Darstellung dieses Vorgehensmodells.

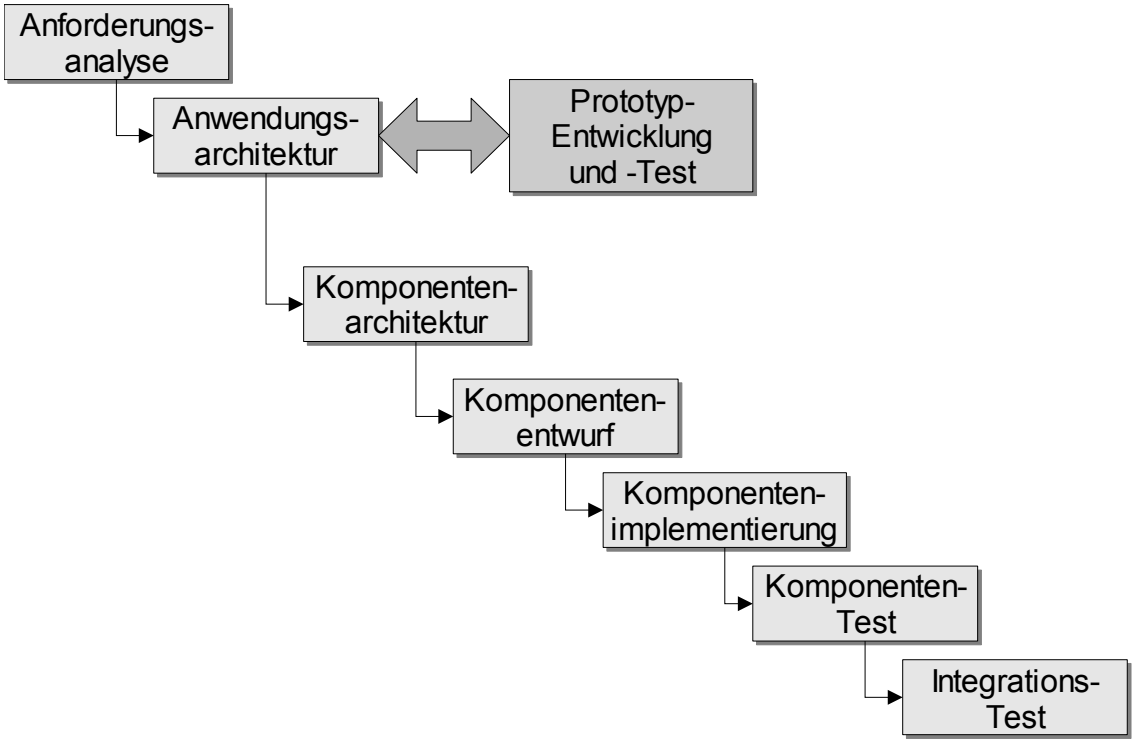


Abbildung 4.1: Im Rahmen der vorliegenden Arbeit gewähltes Vorgehensmodell

Definition sowie Visualisierung des konzeptionellen Modells erfolgen in Form von UML-Diagrammen, die im Laufe des Projektfortschritts durch Ergänzung der Ergebnisse späterer Projektphasen weiter verfeinert werden. Wie in Abschnitt 3.2.6 erläutert, handelt es sich bei UML um einen Quasi-Standard bei der Modellierung objektorientierter Software.

4.1.1 Unified Modelling Language (UML)

UML ist eine grafische Sprache, und es gibt keine exakte Beschreibung der Syntax von UML in Textform. Die Spezifikation von UML wird in dem so genannten *UML Meta Modell* ebenfalls in der Unified Modelling Language vorgenommen. Das bedeutet, dass man die Definition von UML erst dann versteht, wenn man UML bereits beherrscht. Deshalb sollen die im Rahmen der vorliegenden Arbeit verwendeten Diagramme sowie deren wichtigste Elemente an dieser Stelle kurz erklärt werden. Eine umfassende Beschreibung der Unified Modelling Language ist bei HITZ UND KAPPEL [1999] zu finden.

4.1.1.1 Anwendungsfalldiagramm

Im Anwendungsfalldiagramm wird ein Softwaresystem, das implementiert werden soll, durch die so genannten *Anwendungsfälle*, welche es zur Verfügung stellen soll,

dargestellt. Ein Anwendungsfall beschreibt ein bestimmtes Verhalten, das von der Software erwartet wird. Alle Anwendungsfälle zusammengenommen machen die Funktionalität des Systems aus. Ein Anwendungsfall wird in UML als Ellipse dargestellt. Zwischen Anwendungsfällen können Beziehungen festgelegt werden. Die beiden wichtigsten sind die *extend*-Beziehung und die *include*-Beziehung. In beiden Fällen wird ausgedrückt, dass ein Anwendungsfall die Funktionalität eines anderen nutzt. Bei der include-Beziehung ist der benutzte Anwendungsfall unbedingt notwendig, um die Funktionalität des benutzenden Anwendungsfalls sicherzustellen. Die extend-Beziehung besagt, dass die zusätzliche Funktionalität genutzt werden kann, aber nicht notwendigerweise genutzt werden muss.

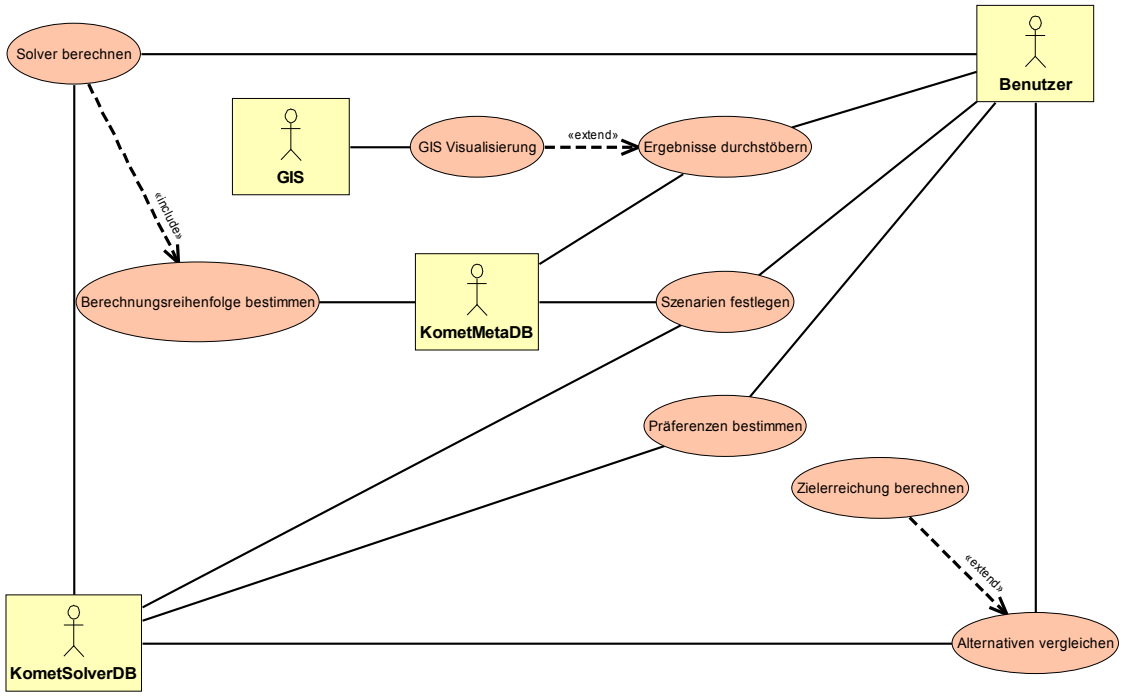


Abbildung 4.2: Anwendungsfalldiagramm

Wer mit der Software interagieren soll, wird durch *Akteure* festgelegt. Sie benutzen das System, indem sie Anwendungsfälle initiieren, oder werden vom System benutzt, indem sie Funktionalität zur Realisierung einzelner Anwendungsfälle zur Verfügung stellen. Akteure werden durch ein Rechteck mit einem Strichmännchen dargestellt. Abbildung 4.2 enthält das Anwendungsfalldiagramm einer generischen EUS-Anwendung. Der Akteur *Benutzer* benutzt das EUS, während der Akteur *GIS* vom EUS genutzt wird. Durch die include-Beziehung zwischen den Anwendungsfällen *Solver berechnen* und *Berechnungsreihenfolge bestimmen* wird verdeutlicht, dass vor dem Aufruf der

einzelnen Solver die richtige Reihenfolge festgelegt sein muss, damit sichergestellt ist, dass benötigte Eingangsdaten zur Verfügung stehen. Im Gegensatz dazu besteht zwischen den Anwendungsfällen *Ergebnisse durchstöbern* und *GIS Visualisierung* eine extend-Beziehung, weil die Darstellung der Ergebnisse in einem GIS optional erfolgen kann.

4.1.1.2 Klassendiagramm

Im Klassendiagramm werden Objektklassen sowie deren Beziehungen dargestellt. Es dient der Darstellung der statischen Grundstruktur einer Anwendung. Das Klassendiagramm enthält Klassen, die als Rechtecke dargestellt werden und Beziehungen zwischen den Klassen in Form von Linien oder Pfeilen. Je nach Detaillierungsgrad können die Klassenrechtecke ein bis drei Bereiche besitzen. Im obersten Bereich befindet sich der Klassenname, darunter die Attribute und im dritten Bereich am unteren Ende befinden sich die Methoden. Beziehungen verbinden in der Regel zwei Klassen miteinander. Sie können an beiden Enden Kardinalitäten (z. B.: *ein* Auto besitzt *einen* Motor, *ein* Auto ist für *bis zu 5* Insassen zugelassen) besitzen. Diese Beziehungen werden unterschieden in Assoziation, Aggregation, Komposition und Generalisierung. Die Assoziation beschreibt eine allgemeine Beziehung zwischen zwei Klassen. Assoziationen können eine Richtung besitzen, die entsprechenden Linien werden in diesem Fall zu Pfeilen. Bei der Aggregation handelt es sich um eine *Besteht-aus-Beziehung* (*part-of relationship*). Sie wird durch eine Raute am Ende der Aggregationskante gekennzeichnet, die sich bei der Aggregationsklasse befindet. Die Komposition ist eine strengere Form der Aggregation, für die folgende Einschränkungen gelten:

- Ein Kompositionsteil darf nur Teil eines Ganzen sein und
- kann nur so lange existieren wie die Kompositionsklasse.

Die Generalisierung beschreibt die taxonomische Beziehung zwischen einer Unterklasse und deren Oberklasse. Sie wird durch einen Pfeil mit einem nicht ausgefüllten gleichseitigen Dreieck als Spitze notiert, der von der Unterklasse zur Oberklasse führt. Abbildung 4.3 zeigt einen Ausschnitt des Klassendiagramms der *KOMET-Architektur*. In diesem Beispiel werden die Klassen *KometKernelClient*, *KometKernelImpl*, *XMLHelper*, *KometXML*, *KometTree*, *KometList* und *KometItem* sowie die Interfaces *KometKernel* und *KometRequest* definiert.

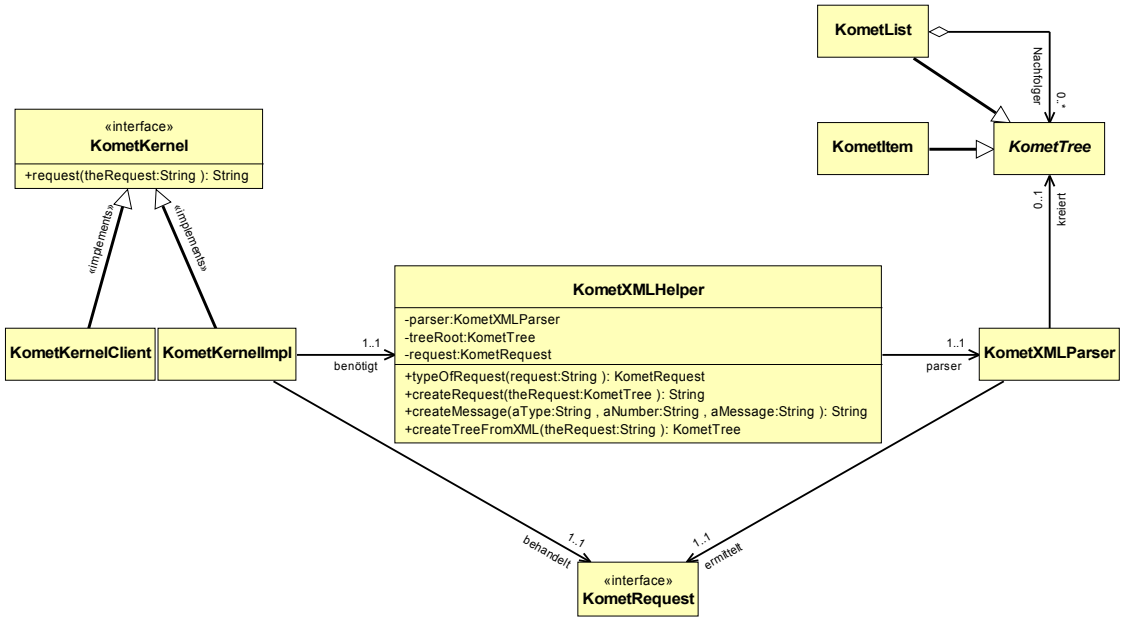


Abbildung 4.3: Klassendiagramm

4.1.1.3 Sequenzdiagramm

Das Sequenzdiagramm dient neben anderen der Veranschaulichung des dynamischen Systemverhaltens. Es wurde ursprünglich zur Modellierung von Telekommunikationssystemen entwickelt (*Message Sequence Charts*) (HITZ UND KAPPEL [1999]). Dabei wird die Art und Weise beschrieben, wie einzelne Objekte miteinander Nachrichten austauschen (*interagieren*), um gemeinsam eine bestimmte Aufgabe zu erfüllen. Bei objektorientierten Programmen entspricht der Aufruf von Objektmethoden dem Austausch von Nachrichten. Das Sequenzdiagramm eignet sich besonders gut, um einen einzelnen möglichen Ablauf exemplarisch darzustellen.

Sequenzdiagramme sind stets zweidimensional. Die vertikale Dimension entspricht der Zeitachse, horizontal werden die betrachteten Objekte aufgetragen. So kann die chronologische Reihenfolge von Nachrichten durch deren vertikale Anordnung im Diagramm spezifiziert werden. Dabei hat der Abstand zwischen den einzelnen Nachrichten auf der Zeitachse keinerlei Bedeutung, das heißt, die Zeitachse ist ordinal skaliert. Nachrichten zwischen Objekten werden als Pfeile dargestellt, wobei dem Methodenaufruf ein Pfeil mit einer durchgezogenen Linie entspricht und dem Methodenrücksprung ein Pfeil mit einer gestrichelten Linie. Wird ein Objekt erst während der Laufzeit des Sequenzdiagramms erzeugt, so mündet der Pfeil der erzeugenden Nachricht direkt im Symbol dieses Objekts. Die Interaktionen der

beteiligten Objekte beim Aufruf der Methode *createDB* der Klasse *KometDBFactory* sind im Sequenzdiagramm in Abbildung 4.4 dargestellt.

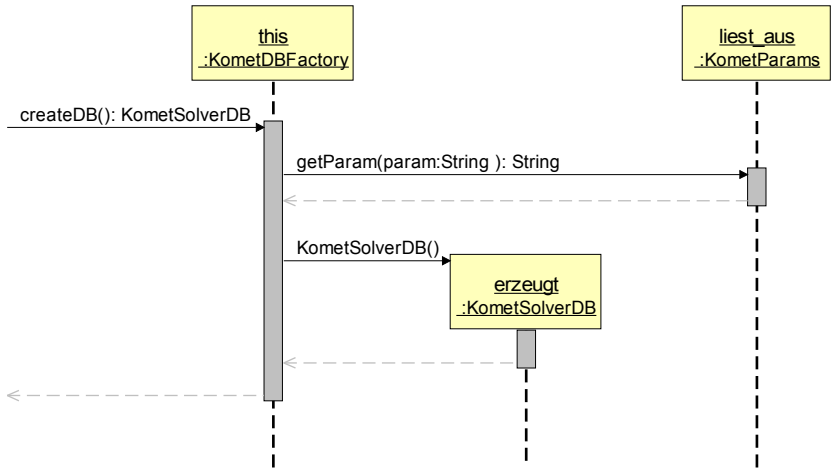


Abbildung 4.4: Sequenzdiagramm (`KometDBFactory.createDB`*)*

4.1.1.4 Aktivitätsdiagramm

Aktivitätsdiagramme ermöglichen die Beschreibung von Abläufen innerhalb der Software, wobei folgende Eigenschaften spezifiziert werden können:

- Tätigkeiten der einzelnen Schritte des Ablaufs (durch Bezeichnung des Schritts und Angabe der durch ihn manipulierten Objekte)
- Ausführungsreihenfolge
- Verantwortlichkeiten für die entsprechenden Ablaufschritte.

Die letzte Angabe ist optional und wird von dem im Rahmen dieser Arbeit eingesetzten Werkzeug, *Objekt Technologie Werkbank 2.4* (MÜLLER [2000]), nicht unterstützt. Die Modellierung erfolgt durch Angabe von Zuständen, in denen bestimmte Aktivitäten durchgeführt werden. Ein solcher *Aktivitätszustand* wird durch ein Rechteck mit halbkreisförmigen Vertikalen dargestellt. Beginn und Ende eines Ablaufes werden durch einen *Anfangszustand* und einen oder mehrere *Endzustände* gekennzeichnet. Der Startzustand wird als schwarzer Kreis dargestellt, der Endzustand als schwarzer Kreis mit einem Ring. Verbunden werden die einzelnen Aktivitäten durch gerichtete *Kontrollflusskanten* (als Pfeile dargestellt), die auch als *Transitionen* bezeichnet werden und die Ausführungsreihenfolge festlegen. Jede Transition kann mit einer Bedingung versehen werden, die bestimmt, unter welchen Umständen sie ausgeführt werden soll. Falls mehrere Möglichkeiten zur Weiterverarbeitung bestehen, kann ein

Entscheidungsknoten eingefügt werden, der als Raute dargestellt wird. Zusätzlich können in Aktivitätsdiagramme Objektzustandsknoten eingefügt und mit der entsprechenden Aktivität verbunden werden. Zur expliziten Beschreibung von parallel ablaufenden Schritten können Gabelungen und Vereinigungen verwendet werden. Eine Vereinigung impliziert eine Synchronisation, das heißt, die Aktivität nach einer Vereinigung wird erst dann ausgeführt, wenn alle parallel ablaufenden Stränge dort angekommen sind. Gabelung und Vereinigung werden als horizontale Linien gezeichnet. Das Aktivitätsdiagramm der Methode *createDB* der Klasse *KometDBFactory* ist in Abbildung 4.5 dargestellt.

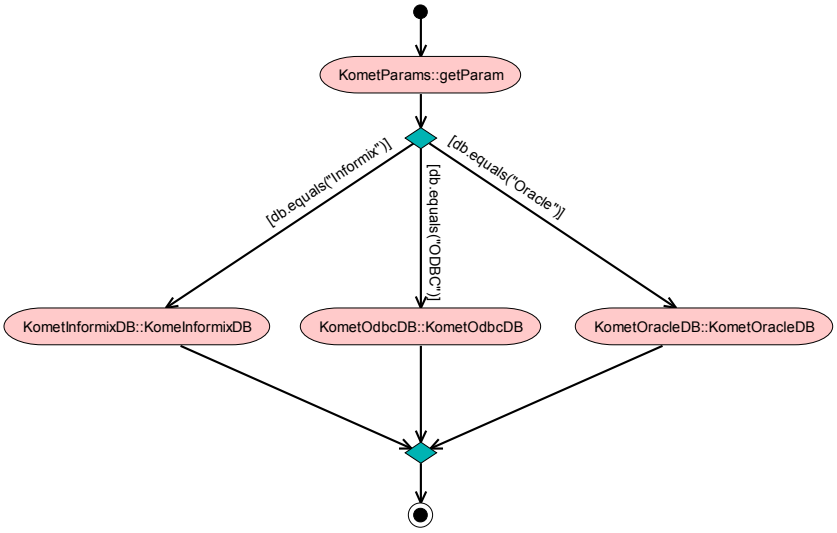


Abbildung 4.5: Aktivitätsdiagramm (`KometDBFactory.createDB`*)*

4.2 Anforderungsprofil

Während der Spezifikation der Anforderungen wird häufig unterschieden zwischen:

- funktionalen Anforderungen
- nichtfunktionalen Anforderungen.

Anforderungen des Benutzers an die Software, die erfüllt sein müssen, um die Funktionalität der Software sicherzustellen, werden als *funktionale Anforderungen* bezeichnet. Dabei bleibt unberücksichtigt, wie diese Funktionen realisiert werden sollen. Unter *nichtfunktionalen Anforderungen* werden solche verstanden, welche die Freiheit des Software-Architekten einschränken und Implementierungsdetails definieren. Es kann sich dabei beispielsweise um Rahmenbedingungen handeln, um die reibungslose Zusammenarbeit mit anderer Software zu erleichtern.

4.2.1 Funktionale Anforderungen

Während der Definition der funktionalen Anforderungen ist die Visualisierung in Form eines Anwendungsfalldiagramms gut geeignet. Darin werden die Hauptfunktionen der Anwendung als *Anwendungsfälle*, verschiedene Benutzergruppen sowie externe Systeme als *Akteure* dargestellt.

Abbildung 4.6 zeigt das Anwendungsfalldiagramm einer generischen REUS-Anwendung. Während der Definition der Anwendungsfälle wurde vor allem darauf geachtet, zu einer möglichst allgemeinen Lösung zu gelangen, die weitgehend übertragbar ist.

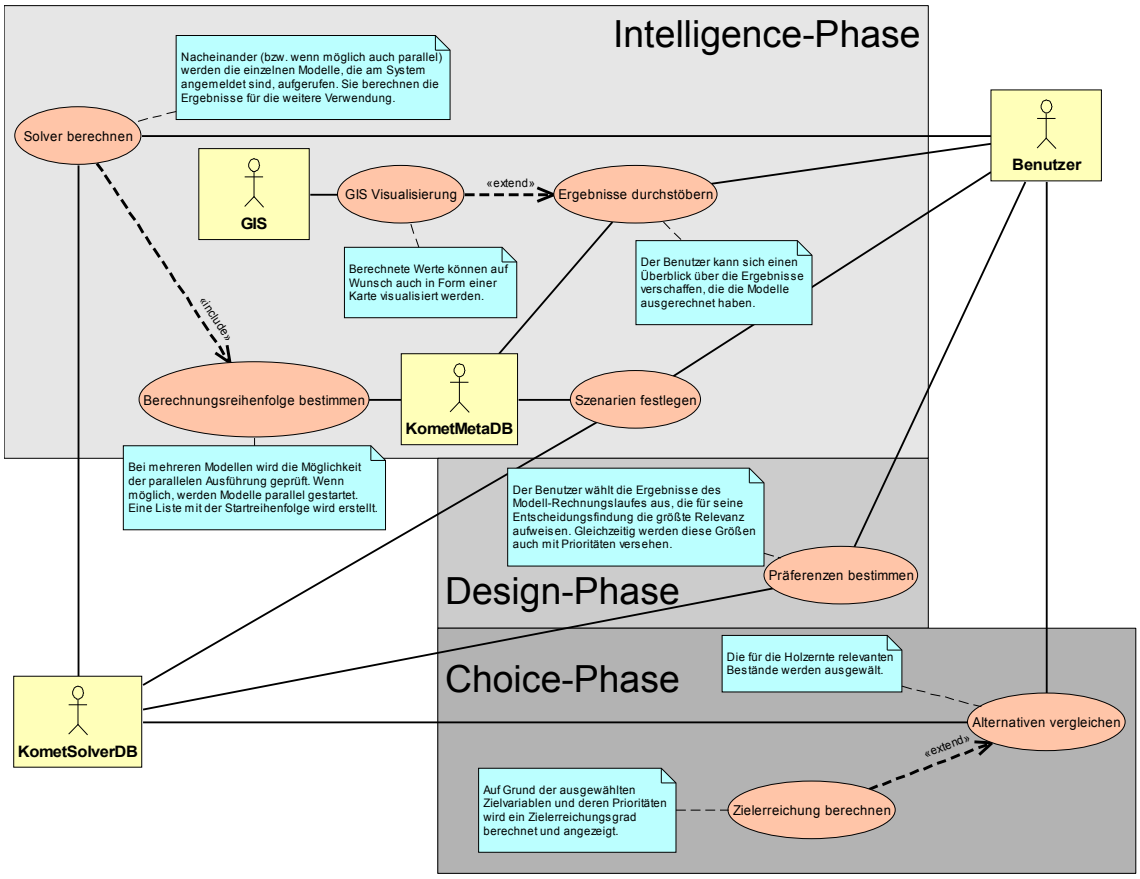


Abbildung 4.6: Anwendungsfalldiagramm einer generischen REUS-Anwendung

Für die REUS-Anwendung werden folgende Anwendungsfälle definiert:

- *Szenarien definieren:* Der Benutzer macht Voreinstellungen für die zu berechnenden Solver.
- *Solver berechnen:* Die Solver werden nacheinander gestartet und stellen deren berechnete Daten dem System zur Verfügung.
  - *Berechnungsreihenfolge bestimmen:* Vor dem Start der Solver muss die richtige Berechnungsreihenfolge bestimmt werden, damit sichergestellt ist, dass benötigte Eingangsdaten beim Start eines Solvers verfügbar sind.
- *Ergebnisse durchstöbern:* Der Benutzer kann sich einen Überblick über die Ergebnisse verschaffen, welche die Solver berechnet haben.
  - *GIS Visualisierung:* Es besteht die Möglichkeit, bestimmte Ergebnisse als Karte in einem GIS zu visualisieren. Dieser Anwendungsfall ist optional.
- *Präferenzen bestimmen:* Der Benutzer wählt die Ergebnisse aus, die für seine Entscheidung die größte Relevanz aufweisen. Gegebenenfalls können diese Ergebnisse mit Prioritäten versehen werden.
- *Alternativen auswählen:* Aus den angebotenen Alternativen kann sich der Anwender diejenige auswählen, die seinen Zielvorgaben am ehesten entspricht.
  - *Zielerreichung bestimmen:* Auf Grund der vorgegebenen Zielvariablen und deren Priorität wird ein Zielerreichungsgrad berechnet. Dieser Anwendungsfall ist optional.

Im Anwendungsfalldiagramm werden die drei in Abschnitt 3.1.3 erwähnten Entscheidungs-Phasen *Intelligence*, *Design* und *Choice* berücksichtigt. Die Akteure KometSolverDB und KometMetaDB im Anwendungsfalldiagramm entsprechen der Datenbank bzw. der Modellbank in der IT-orientierten EUS-Sicht (vgl. Abschnitt 3.1.4).

4.2.2 Nichtfunktionale Anforderungen

Zur korrekten Interpretation der Daten sind *Metadaten* (siehe Abschnitt 3.1.7) unbedingt erforderlich. Ohne entsprechende Zusatzinformationen kann maschinell nicht festgestellt werden, ob es sich bei einem Zahlenwert beispielsweise um ein Volumen handelt und wenn ja, in welcher Einheit dieses Volumen gemessen wurde (Festmeter *ohne* Rinde, um Festmeter *mit* Rinde, Schüttraummeter, Kubikzentimeter, Milliliter etc.). Wird jedem Ergebnis ein bestimmter Objekttyp (*Klasse*) und eine Maßeinheit als

Attribut zugewiesen, so sind diese Informationen auch für maschinelle Auswertungen zugänglich. Um die Berechnungsreihenfolge der Solver bestimmen zu können, muss ebenfalls gespeichert werden, welche Daten von welchem Solver benötigt werden und welcher Solver welche Ergebnisse liefert. Diese Informationen sind auch notwendig, um dem Anwender eine Liste der Ergebnisse zu präsentieren und gegebenenfalls in einem GIS zu visualisieren. Zur GIS-Visualisierung müssen Schlüsselattribute identifiziert werden können, um die Solver-Ergebnisse den entsprechenden Geometrien zuzuordnen.

Aus dem Anwendungsfalldiagramm geht hervor, dass zwei zentrale Datenquellen nötig sind:

- Metadatenbank (KometMetaDB)
- Solverdatenbank (KometSolverDB).

Es besteht die Möglichkeit für diese Datenquellen entweder ein verteiltes Datenhaltungskonzept zu entwerfen (vgl. ENDEJAN [2002], DENZER, GÜTTLER UND HELL [2002]) oder eine zentrale Datenhaltung vorzusehen (vgl. NUTE, POTTER UND MAIER [2000]). Um die Akzeptanz der Integrationsplattform zu erhöhen, sollen die Solver so einfach wie möglich zu realisieren sein. Deshalb wird im Rahmen der vorliegenden Arbeit eine zentrale Datenspeicherung umgesetzt, da so die Erstellung von zusätzlichen Programmteilen, die für den externen Zugriff auf Solverdaten verantwortlich sind, entfällt.

Die Anwendungsarchitektur soll, wie in Abschnitt 3.2.6 beschrieben, in mehreren Schichten aufgebaut sein. Dies führt zu besserer Übersichtlichkeit sowie Wartbarkeit des Gesamtsystems.

Für neu zu erstellende Programme soll Java verwendet werden. Java ist eine moderne objektorientierte Programmiersprache. Übersetzte Programme werden interpretiert und von einer so genannten ***J**ava **V**irtual **M**achine* (*JVM*) ausgeführt. Dieses Vorgehen erlaubt den Einsatz von fertigen Java-Programmen auf vielen verschiedenen Plattformen und Rechner-Architekturen, ohne dass eine neue Übersetzung notwendig ist. Einzige Voraussetzung für die Ausführung von Java-Programmen ist die Verfügbarkeit einer JVM für die Zielplattform (siehe Abschnitt 4.2.2.1.1).

Die Anbindung der Solver soll so erfolgen, dass die Programmiersprache, in der sie geschrieben sind, keine Rolle spielt. Deshalb muss eine sprachunabhängige Schnittstelle

definiert werden. Die in Abschnitt 3.2.4 beschriebenen Komponententechnologien sind hierfür in Verbindung mit der in Abschnitt 3.1.7.1 erwähnten Extensible Markup Language (XML) gut geeignet. Wegen der Verwendung von Java als Programmiersprache wird die ***R**emote **M**ethod **I**nvocation* (*RMI*) eingesetzt (siehe Abschnitt 4.2.2.1.2). Um den Zugriff durch Solver in anderen Programmiersprachen zu erleichtern, wird die RMI-Schnittstelle kompatibel zu CORBA realisiert. Dazu wird eine Sonderform von RMI verwendet, die auf dem in CORBA verwendeten ***I**nternet **I**nter **O**RB **P**rotocol* (*IIOP*) basiert (*RMI over IIOP*).

4.2.2.1 Java und Remote Method Invocation (RMI)

4.2.2.1.1 Java

Java ist eine objektorientierte Programmiersprache und wurde von *SUN Microsystems, Inc.* entwickelt. Im Rahmen des ***J**ava **C**ommunity **P**rocess* (*JCP*) (SUN MICROSYSTEMS [1998]) wird Java unter der Aufsicht von *SUN Microsystems, Inc.* gepflegt und weiterentwickelt. Ende 2006 wurde Java quelloffen unter der ***G**NU **P**ublic **L**icense* (*GPL*) veröffentlicht (SUN MICROSYSTEMS [2006]). Zur Zeit ist Java in der Version 1.6 (auch als *Java 6* bezeichnet) aktuell. Wie in C++ oder C# ist der Sprachumfang von Java sehr klein (es gibt nur etwa 20 reservierte Wörter) und es existiert eine umfangreiche Klassenbibliothek. Deren Funktionen reichen von Dateimanipulationen über entfernten Netzwerkzugriff bis zur Gestaltung einer grafischen Benutzeroberfläche. Wie bereits in Abschnitt 4.2.2 erwähnt, werden Java-Programme nicht direkt von der CPU des Rechners ausgeführt, sondern in einen Zwischencode, den so genannten *Java-Bytecode*, übersetzt und innerhalb einer JVM ausgeführt. Dies gewährleistet eine Plattformunabhängigkeit des *übersetzten Programmcodes* im Gegensatz etwa zur Plattformunabhängigkeit nur des *Quellcodes* bei C++.

Im Microsoft .Net-Umfeld wird ganz ähnlich vorgegangen. .Net-Programme werden in einen Zwischencode übersetzt, der ***M**icro**s**oft **I**ntermediate **L**anguage* (*MSIL*) genannt wird. Mit Hilfe eines so genannten *Just-In-Time-Compilers* werden die Programme auf der Zielplattform während der Laufzeit in ausführbaren Maschinencode übersetzt. Solche Just-In-Time-Compiler existieren auch im Java-Umfeld, um die Ablaufgeschwindigkeit der Java-Programme zu erhöhen.

Klassen bilden das Grundgerüst von Java-Programmen. Mit Hilfe so genannter *Klassenvariablen* werden *Instanzen* der Klassen angelegt und darauf zugegriffen. Klassen werden automatisch aus dem Arbeitsspeicher entfernt, wenn keine Klassenvariable mehr auf sie verweist (*Garbage Collection*). Innerhalb von Klassen können Klassen, Attribute und Methoden definiert werden. Es gibt dabei die drei Sichtbarkeitsstufen *public*, *protected* und *private*. Elemente, die als private deklariert sind, sind außerhalb der Klasse nicht sichtbar, die Sichtbarkeitsstufe public erlaubt einen uneingeschränkten Zugriff. Elemente, die als protected definiert sind, sind nur innerhalb der umgebenden Klasse und innerhalb von Klassen sichtbar, die durch Vererbung davon abgeleitet wurden. Mit Hilfe dieser Sichtbarkeiten kann die in Abschnitt 3.2.5 erwähnte Kapselung umgesetzt werden. In Java gibt es eine besondere Methode, die bei der Instantiierung automatisch aufgerufen wird, den *Konstruktor*. Man erkennt den Konstruktor an seinem Namen - denn er heißt wie die Klasse - und an dem fehlenden Rückgabewert.

Java unterstützt Vererbung in der Form, dass nur eine Elternklasse angegeben werden kann (C++ unterstützt im Gegensatz dazu mehrere Elternklassen). Damit dies nicht zu einem Nachteil wird, kann eine Klasse beliebig viele so genannte *Interfaces* implementieren. Ein Interface kann als Datentyp verwendet werden und ist eine Sammlung von Methoden, die von der implementierenden Klasse zur Verfügung gestellt werden müssen. Bei Vererbung werden stets alle Eigenschaften der Elternklasse übernommen und es darf nichts weggelassen werden. Der Umfang einer Klasse kann also nur zunehmen, erweitert werden oder gleich bleiben aber niemals abnehmen. Deshalb wird im Java-Umfeld auch vielfach davon gesprochen, dass eine Kindklasse die Elternklasse *erweitert*. Dies wird durch die Verwendung des Schlüsselwortes *extends*, gefolgt vom Namen der Elternklasse, unterstrichen.

Mehrere Klassen können zu *Packages* zusammengefasst werden, die ihrerseits neben Klassen auch Packages enthalten können. Innerhalb von Packages können Klassen als public, protected oder private deklariert werden. Soll eine Klasse außerhalb des Packages sichtbar sein, muss sie public definiert werden. Klassen, die als protected definiert sind, sind nur innerhalb des Packages und in dessen Unterpackages sichtbar. Auch die Klassenbibliothek, die zu Java gehört, befindet sich in mehreren Packages.

4.2.2.1.2 Remote Method Invocation (RMI)

In Java gibt es verschiedene Möglichkeiten, über ein Netzwerk Methoden von Klassen aufzurufen, die sich auf entfernten Rechnern befinden. Eine dieser Möglichkeiten ist die Technik des entfernten Methodenaufrufs (***R**emote **M**ethod **I**nvocation*, *RMI*). Alles was der Programmierer dazu benötigt, ist bereits in der Java Standard Edition (J5SE) mitgeliefert und befindet sich zum größten Teil in den Packages *java.rmi* bzw. *javax.rmi*. Es wird unterschieden zwischen dem Server, dessen Objektmethode aufgerufen wird, und dem Client, der diese Methode aufruft. Es genügt, dass die Serverklasse die Klasse *javax.rmi.PortableRemoteObject* erweitert. Im Konstruktor sind einige Voreinstellungen vorzunehmen, zum Beispiel muss eine Instanz der Serverklasse beim Namens-Dienst registriert werden. Hat der Client die Serverklassen-Instanz mit Hilfe des Namens-Dienstes gefunden, wird durch den Aufruf der Methode `PortableRemoteObject.narrow` eine lokale Platzhalter-Instanz erzeugt, welche die Methodenaufrufe an die Server-Instanz weiterleitet. Dabei wird sie von einigen Hilfsklassen unterstützt, die mit Hilfe eines mitgelieferten Programms, des ***RMI-C**ompilers*, (*rmic*) erzeugt werden. Vor dem Start des Server- und des Client-Programms muss der ***O**bject **R**equest **B**roker **D**aemon* (*orbd*) gestartet werden, der unter anderem zwischen dem Klienten und dem Server vermittelt und den bereits erwähnten Namensdienst implementiert. Dieses Programm befindet sich ebenfalls im Lieferumfang der Java-Laufzeitumgebung.

4.3 Implementierungsstrategie

Auf Grund der funktionalen und nichtfunktionalen Anforderungen müssen Design-Entscheidungen getroffen werden, welche die Definition des konzeptionellen Modells beeinflussen.

4.3.1 Architekturmodell

Der grundlegende Aufbau der EUS-Anwendung besteht aus drei Schichten, welche sich wie folgt zusammensetzen:

- Solver und Planungskomponente
- EUS-Kern
- Datenbank für Solverdaten sowie Metadaten.

Der EUS-Kern bildet die Schnittstelle zwischen Daten- und Solverebene. Ebenso können Solver sowie Planungskomponenten über diesen Systemkern mit anderen Komponenten kommunizieren.

Die Solver sollen in einem so genannten Batch-Betrieb arbeiten. Das heißt, die benötigten Eingangsdaten werden aus der Solverdatenbank gelesen und die Ergebnisse werden in der Datenbank gespeichert. Eine Interaktion mit dem Anwender ist nicht notwendig. Eine Sonderstellung nimmt die Planungskomponente ein, die direkt mit dem Anwender kommuniziert. Die in Abbildung 4.7 gezeigte schematische Darstellung des Schichtenmodells entspricht der in Abschnitt 3.2.6 beschriebenen Drei-Schicht-Architektur bzw. dem Aufbau eines EUS aus IT-orientierter Sicht (siehe Abschnitt 3.1.4). Die Solver-Datenbank entspricht dem Datensubsystem, die Solver entsprechen dem Modellsubsystem und die Planungskomponente entspricht dem Dialogsubsystem. Der EUS-Kern übernimmt dabei Aufgaben von DBMS, MBMS sowie DGMS (siehe Abbildung 3.2) und vermittelt zwischen den Subsystemen. Diese Vermittlung wird über entsprechende Dienste realisiert, die der EUS-Kern zur Verfügung stellt. Mit Hilfe dieser Dienste bildet der EUS-Kern die in Abschnitt 4 einleitend beschriebene Integrationsplattform.

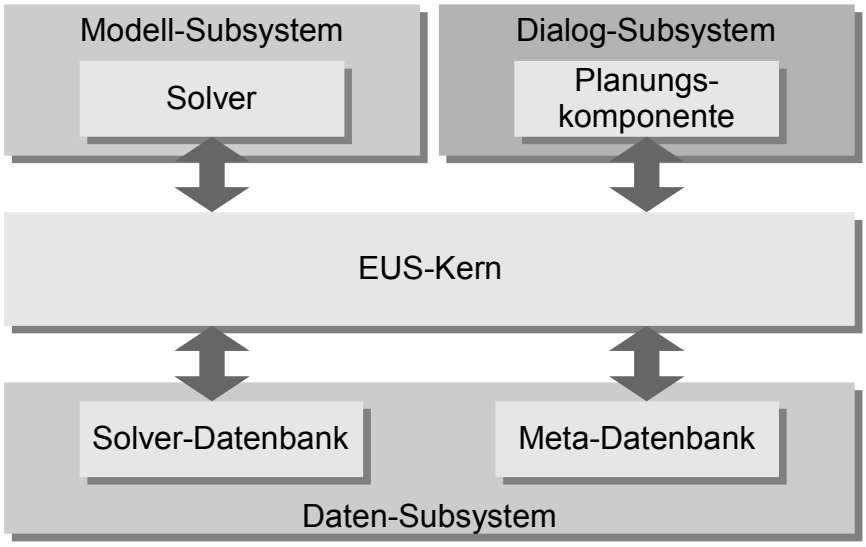


Abbildung 4.7: Schichtenmodell einer EUS-Anwendung

4.3.2 Datenhaltungsmodell

Die Speicherung der Daten soll in einem gemeinsamen ***r****elationalen* ***D****aten****b****ank****m****anagement****s****ystem* (*RDBMS*) erfolgen. In dem RDBMS werden sowohl Solverdaten als auch Metadaten gespeichert.

Zugriffe auf das RDBMS können sowohl über ODBC als auch über JDBC realisiert werden, so dass die Solver ihre eigenen Daten effizient direkt verwalten können. In einem relationalen DBMS werden die Daten in Form von Tabellen und Spalten mit Beziehungen (*Relationen*) zwischen Tabellen gespeichert. Dabei müssen Namensgleichheiten vermieden werden. Um sicherzustellen, dass zwei Tabellen zweier Solver verschiedene Namen erhalten, muss eine Namenskonvention eingeführt werden. Im Rahmen der vorliegenden Arbeit muss jeder Tabellenname mit einem Kürzel des Solvers gefolgt von einem Unterstrich beginnen, weitere Unterteilungen mit Unterstrichen sind möglich (z. B.: *slv_input_durchforstung*).

Der Datenzugriff soll auch indirekt über den EUS-Kern erfolgen können. Im Systemkern ist ein entsprechender Dienst vorzusehen.

4.3.3 Kommunikationsmodell

Die Kommunikation zwischen den Solvern bzw. der Planungskomponente und dem EUS-Kern erfolgt durch den Austausch von XML-Botschaften. Um einen bestimmten Dienst des Systemkerns zu nutzen, schicken die Solver eine entsprechende XML-Anfrage an den EUS-Kern. Nach Bearbeitung der Dienst-Anfrage sendet der Systemkern die entsprechende XML-Antwort an die anfragende Komponente zurück. Dies erlaubt völlige Unabhängigkeit von den eingesetzten Programmiersprachen, denn die Übergabe von komplexen Datentypen, die sprachspezifisch sind, wird vermieden. Es muss nur eine einzige Operation mit einem Rückgabewert vom Typ *String* und einem Parameter vom Typ *String* definiert werden. Diese kann ohne großen Aufwand in verschiedenen Programmiersprachen implementiert werden (siehe z. B. WIRTH [1993]).

4.3.3.1 Extensible Markup Language (XML)

XML ist eine Auszeichnungssprache, deren grundlegende Struktur durch Kennzeichnungselemente (so genannte *Tags*) festgelegt wird. Ein Tag ist ein

Schlüsselwort, das durch ein »<«- und ein »>«-Zeichen eingerahmt wird (z. B.: *<komet>*). Tags können Attribute, weitere Tags sowie freien Text enthalten und bestehen immer aus zwei Teilen, einem öffnenden Tag und einem schließenden. Beim schließenden Tag wird dem Schlüsselwort ein »/«-Zeichen vorangestellt. (z. B.: *</komet>*). Enthält ein Tag weder weitere Tags noch freien Text, kann statt des öffnenden Tags direkt gefolgt vom schließenden eine Kurzschreibweise verwendet werden (z. B.: *<komet />*). Wie bereits in Abschnitt 3.1.7 erwähnt, ist es möglich, individuelle Instanzen einer XML zu entwerfen. Dies geschieht durch die Definition von Tags und deren Inhalt.

Im Rahmen der vorliegenden Arbeit wird der Informationsaustausch zwischen den einzelnen Komponenten in Form von Botschaften realisiert, die in einer Extensible Markup Language kodiert sind. Dienstanfragen an den EUS-Kern bzw. Dienstantworten des Systemkerns werden durch die Definition entsprechender Tags realisiert. Dazu wird die ***Komet M**arkup **L**anguage* (*KometML*) mit Hilfe einer ***D**ocument **T**ype **D**efinition* (*DTD*) entworfen.

4.3.4 Metadatenmodell

Mit Hilfe der Metadaten, welche ein solverübergreifendes Datenmodell definieren, wird eine maschinelle Interpretation der Solverdaten ermöglicht. Auf diese Weise können Solver auf Daten anderer Komponenten zugreifen, ohne über Kenntnisse bezüglich deren interner Datenhaltungskonzepte verfügen zu müssen.

Diese Metadaten werden sowohl in der zentralen relationalen Datenbank als auch in OWL-Dateien (vgl. Abschnitt 3.1.7.2) abgelegt. Vordefinierte Regeln erlauben erweiterte Abfragemöglichkeiten, die durch die alleinige Speicherung in der Datenbank nicht möglich wären. Diese können z. B. zur GIS-Visualisierung verwendet werden.

Die Metadaten werden ausschließlich über den EUS-Kern verwaltet. Wird ein Solver an die Integrationsplattform angebunden, so müssen dessen Metadaten im System gespeichert werden. Wird ein Solver aus dem System entfernt, so müssen diese Metadaten wieder gelöscht werden. Über die Metadaten kann somit auch ermittelt werden, welche Solver sich im System befinden. Durch geeignete Abfragen (Welche Ergebnisse werden geliefert? - Welche Eingabedaten werden benötigt?) können

Abhängigkeiten zwischen einzelnen Solvern erkannt werden, die bei der Ermittlung der Startreihenfolge benötigt werden. Im EUS-Kern müssen entsprechende Registrierungs- und Abfragedienste vorgesehen werden.

4.3.4.1 Web Ontology Language (OWL)

Wie in Abschnitt 3.1.7.2 dargestellt, ist die ***W**eb **O**ntology **L**anguage* (*OWL*) als Bestandteil des *Semantic Web*, einer Weiterentwicklung des heutigen Internet, entstanden. Bei mit Hilfe von OWL erstellten Ontologien handelt es sich um XML-Dokumente, die in unterschiedlichen Anwendungen eingesetzt werden können. Es ist also möglich, die Metadaten eines EUS als Ontologie in OWL zu speichern und darauf zuzugreifen.

OWL existiert in drei Ausprägungen:

- OWL Lite
- OWL DL
- OWL Full.

Zur ausschließlichen Definition von taxonomischen Beziehungen zwischen Objekten kann *OWL Lite* angewendet werden. Neben Klassenhierarchien können mit OWL Lite nur einfache Beziehungen definiert werden. *OWL Full* ist so ausdrucksstark, dass nicht garantiert werden kann, dass ein Computerprogramm immer in der Lage ist, OWL Full Ausdrücke zu berechnen. OWL Full wird deshalb den unentscheidbaren Sprachen zugeordnet. Die Ausdrucksstärke von *OWL DL* liegt zwischen der von OWL Full und OWL Lite. DL steht dabei für *Description Logics*, welche in OWL DL zur Definition von Regeln verwendet werden können. Description Logics sind eine besondere Form der *Prädikatenlogik*. Neben gebräuchlichen logischen Operatoren, wie z. B.: »∧« (*und)*, »∨« (*oder*) bzw. »¬« (*nicht*) gibt es in der Prädikatenlogik die Operatoren »∀« (*Für alle Elemente gilt:*) und »∃« (*Es existiert ein Element, für das gilt:*).

Der folgende prädikatenlogische Ausdruck beschreibt die Aussage *Es gibt unendlich viele Primzahlen* (Formel 1):

$$\forall a: \exists b: b > a \wedge (\neg\exists c \wedge \neg\exists d): (c > 1) \wedge (d > 1) \wedge (b = c*d) \qquad (1)$$

In Textform ausgedrückt, bedeutet Formel 1:

Für jede beliebige Zahl *a* gilt: Es existiert eine Zahl *b* für die gilt: *b* ist größer als *a* und *b* besitzt die Eigenschaft, dass es keine Zahlen *c* und *d*, beide größer als 1 gibt, für die gilt: *c* mal *d* gleich *b*. Formel 1 beschreibt eine wahre Behauptung, was aber nur mit Hilfe eines zusätzlichen Beweises belegt werden kann (vgl. HOFSTADTER [1989]). Ein logischer Ausdruck, der eine Behauptung beschreibt, kann auch dann vorliegen, wenn eine Aussage von OWL Full nicht von einem Computer berechnet werden kann.

Mit Hilfe der Description Logics können Aussagen wie Formel 1 nicht erzeugt werden. Sie sind dennoch mächtige Werkzeuge, mit denen sich exakte Abbildungen von klar abgegrenzten Anwendungsdomänen modellieren lassen (ZIEGLER [2003]). Im Sprachumfang von OWL stehen XML-Konstrukte zur Verfügung, mit deren Hilfe logische Ausdrücke erzeugt werden können. Die Dokumente der Empfehlung des **W**orld **W**ide **W**eb **C**onsortium (W3C) MCGUINNESS UND VAN HARMELEN [2004], SMITH ET AL. [2004] und DEAN UND SCHREIBER [2004] enthalten eine detaillierte Beschreibung von OWL.

Im Rahmen der vorliegenden Arbeit wird eine Ontologie verwendet, um die in einem EUS benötigten forstlichen Messwerte und deren Beziehungen zu beschreiben. Dazu steht dem EUS-Kern ein mit Hilfe von OWL definiertes Klassengerüst zur Verfügung. Die Daten jedes Solvers werden als Objekte definiert, welche die Kern-Klassen instantiieren und deren Attribute mit Werten initialisieren. Per KometML werden diese Informationen mit der Registrierungsanfrage an den EUS-Kern übertragen, dort in OWL übersetzt und in die EUS-Ontologie eingebunden.

4.3.4.2 ESRI Forestry Data Model

Die Grundstruktur des im Rahmen der vorliegenden Arbeit konzipierten Klassengerüsts basiert auf dem *ESRI Forestry Datamodel* (STEFFENSON [2001]). Seitdem es mit der GIS-Software *ArcGIS* möglich ist, objektrelationale Datenmodelle zu implementieren, werden vom Hersteller Referenzdatenmodelle für die Forstwirtschaft und andere Anwendungsbereiche angeboten. In Abbildung 4.8 ist das *ESRI Forestry Datamodel* schematisch dargestellt.

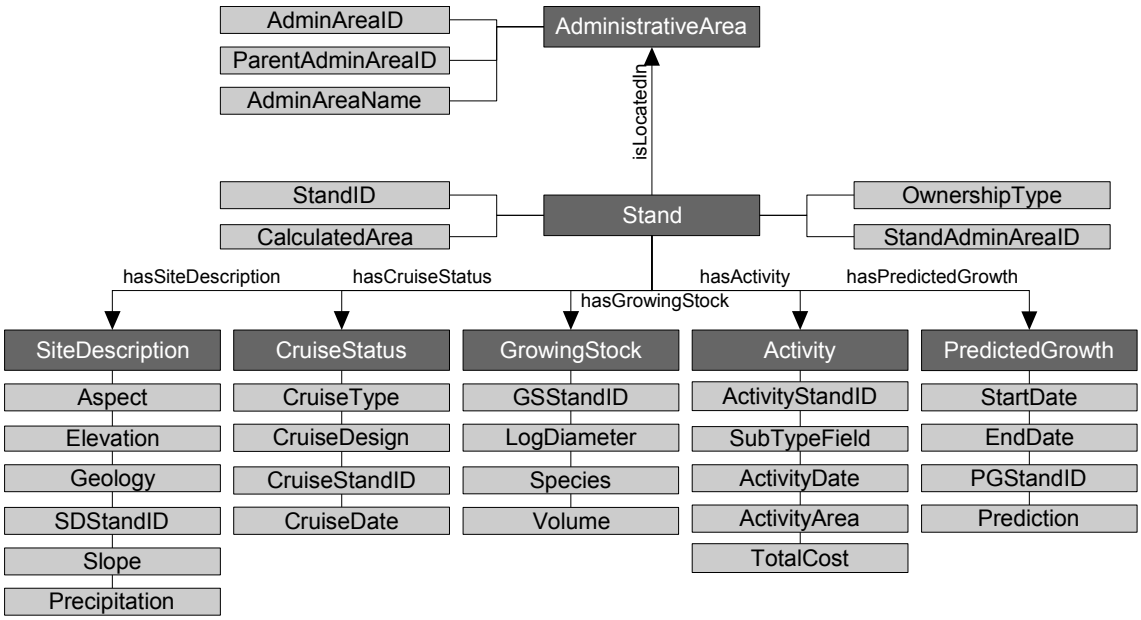


Abbildung 4.8: Schematische Darstellung der Ontologie des ESRI Forestry Datamodel

Ausgehend vom Bestand (*Stand*) werden Verwaltungseinheit (*AdministrativeArea*), Standortsbeschreibung (*SiteDescription*), Befahrungssituation (*CruiseStatus*), Bestockung (*GrowingStock*), Aktivität (*Activity*) und Wachstumsprognose (*Predicted-Growth*) in der Ontologie als Klassen definiert und mit der Klasse *Stand* in Beziehung gesetzt. Diesen Klassen, welche die Struktur festlegen (in der Abbildung als dunkle Rechecke dargestellt), werden im *ESRI Forestry Datamodel* Attribute zugeordnet. Diese werden der Ontologie ebenfalls als Klassen (helle Rechtecke) hinzugefügt und mit Hilfe von Beziehungen der Klassen, deren Attribute sie sind, zugewiesen.

4.4 Demonstrationsanwendung

Im Rahmen der vorliegenden Arbeit werden die bei HEMM [2006] beschriebenen Komponenten zu einem räumlichen Entscheidungs-Unterstützungs-System zur Holzernte integriert, um die grundsätzliche Machbarkeit der Realisierung eines REUS auf Basis der *KOMET-Architektur* zu überprüfen.

4.4.1 Integrierte Anwendungen

Folgende von HEMM [2006] verwendete Anwendungen werden mit Hilfe der *KOMET-Architektur* integriert:

- *Silva 2.2* des Lehrstuhls für Waldwachstumskunde der Technischen Universität München (PRETZSCH, BIBER UND DURSKY [2002])
- *Holzernte 7.0* der Forstlichen Versuchs- und Forschungsanstalt Baden-Württemberg (SCHÖPFER, KÄNDLER UND STÖHR [2003])
- *AutoMod 11.0* der Firma Applied Materials, Inc. (AUTOMOD [2007]).

4.4.2 Geodaten

Einige der zu integrierenden Anwendungen benötigen Geodaten, welche verwaltet und vorverarbeitet werden müssen. Dies sind:

- Bestandesdaten
- Einzelbaumdaten
- Rückegassennetz.

Die Geodaten werden mit Hilfe des Geografischen Informationssystems *ArcGIS* der Firma ESRI verwaltet. Deren Verarbeitung erfolgt mit Hilfe der von *ArcGIS Engine for Java 9.2* zur Verfügung gestellten Bibliothek *ArcObjects* (ZEILER [2002], ZEILER [2002a]). Auf diese Weise ist es möglich, ArcGIS-Funktionen mit Java zu nutzen.

Für die im Rahmen der vorliegenden Arbeit durchgeführten Simulationen werden Daten aus der Abteilung 4A des Forstamtes Schmallenberg im Forstbetrieb Mark Medelon der Landesforstverwaltung Nordrhein-Westfalen verwendet. Die Einzelbaumdaten mit georeferenzierten Stammfußpunktkoordinaten wurden durch Auswertung einer Geländebefliegung gewonnen. Sie wurden ebenso wie das Bestandespolygon, die Rückegassenlinien und das digitale Geländemodell (DGM) von der Landesforstverwaltung Nordrhein-Westfalen zur Verfügung gestellt. Eine Karte der Abteilung 4A ist in Abbildung 4.9 dargestellt.

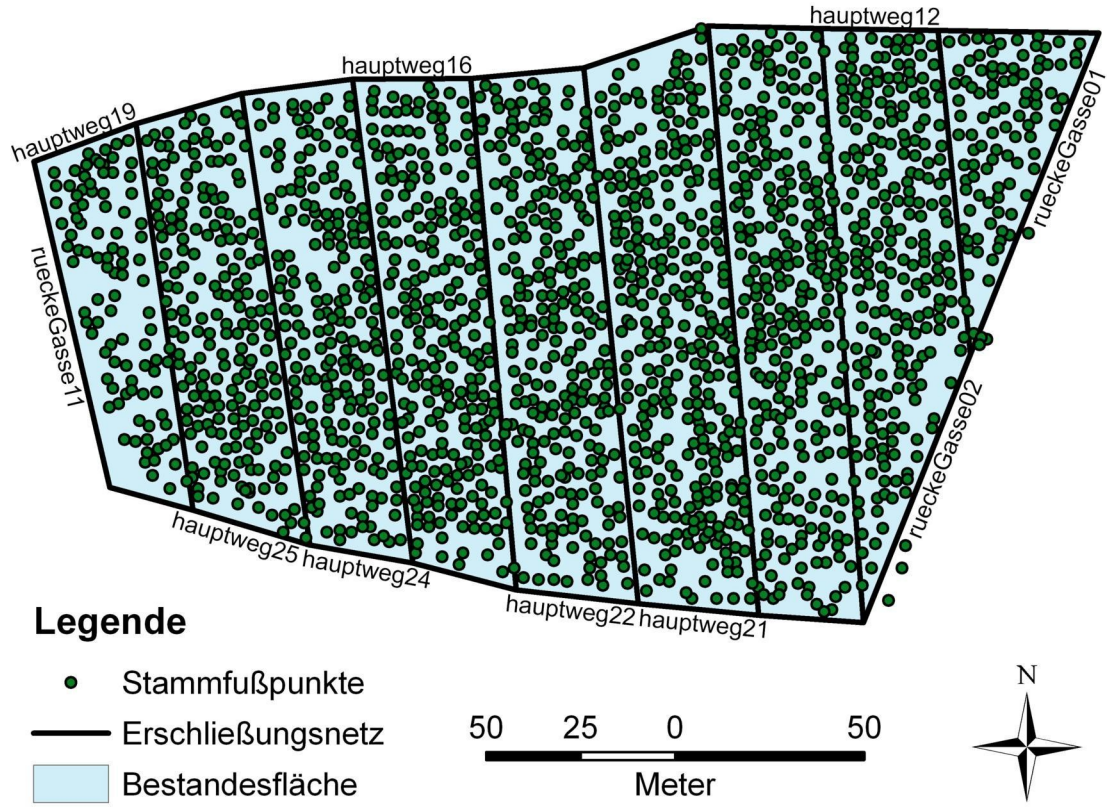


Abbildung 4.9: Kartografische Darstellung der Abteilung 4A

5 Ergebnisse

Nachfolgend werden das konzeptionelle Modell der *KOMET-Architektur* als Grundlage einer Integrationsplattform für forstliche Entscheidungsunterstützungskomponenten und deren Implementierung erläutert. Mit Hilfe der *KOMET-Architektur* werden die in HEMM [2006] beschriebenen Komponenten zu einem räumlichen Entscheidungs-Unterstützungs-System zur Holzernte integriert und dabei die grundsätzliche Machbarkeit der Implementierung eines REUS auf Basis der *KOMET-Architektur* belegt. Auf eine detaillierte Beschreibung der Ergebnisse jeder Phase des Vorgehensmodells wird dabei verzichtet, um den Umfang der vorliegenden Arbeit nicht zu sprengen. Abschnitt 5.1 enthält die Ergebnisse aus den Phasen *Anwendungsarchitektur*, *Komponentenarchitektur* und *Komponentenentwurf*, Abschnitt 5.2 die Ergebnisse aus den Phasen *Komponentenimplementierung*, *Komponententest* und *Integrationstest*.

5.1 Konzeptionelles Modell der KOMET-Architektur

5.1.1 Anwendungsarchitektur

Die Beschreibung der Anwendungsarchitektur entsteht während eines zyklischen Prozesses. Nach deren Definition wird sie in Form eines Prototypen implementiert, der verifiziert wird. Auf Grund dieser Ergebnisse wird eine neue Architektur durch Erweiterung, Verfeinerung bzw. Änderung der alten erstellt und wieder ein Prototyp erzeugt. Dies wird so lange wiederholt, bis die Grundstruktur der Anwendung einen zuvor definierten Qualitätsstandard erreicht hat (vgl. Abbildung 4.1). Im Fall der vorliegenden Arbeit ist dieser Qualitätsstandard erreicht, wenn die nichtfunktionalen Anforderungen erfüllt sind und sichergestellt ist, dass die funktionalen Anforderungen realisiert werden können.

Im Laufe der Anwendungsarchitektur-Phase wurde das Schichtenmodell aus Abschnitt 4.3.1 Schritt für Schritt zur *KOMET-Architektur* verfeinert, deren schematische Darstellung Abbildung 5.1 zeigt.

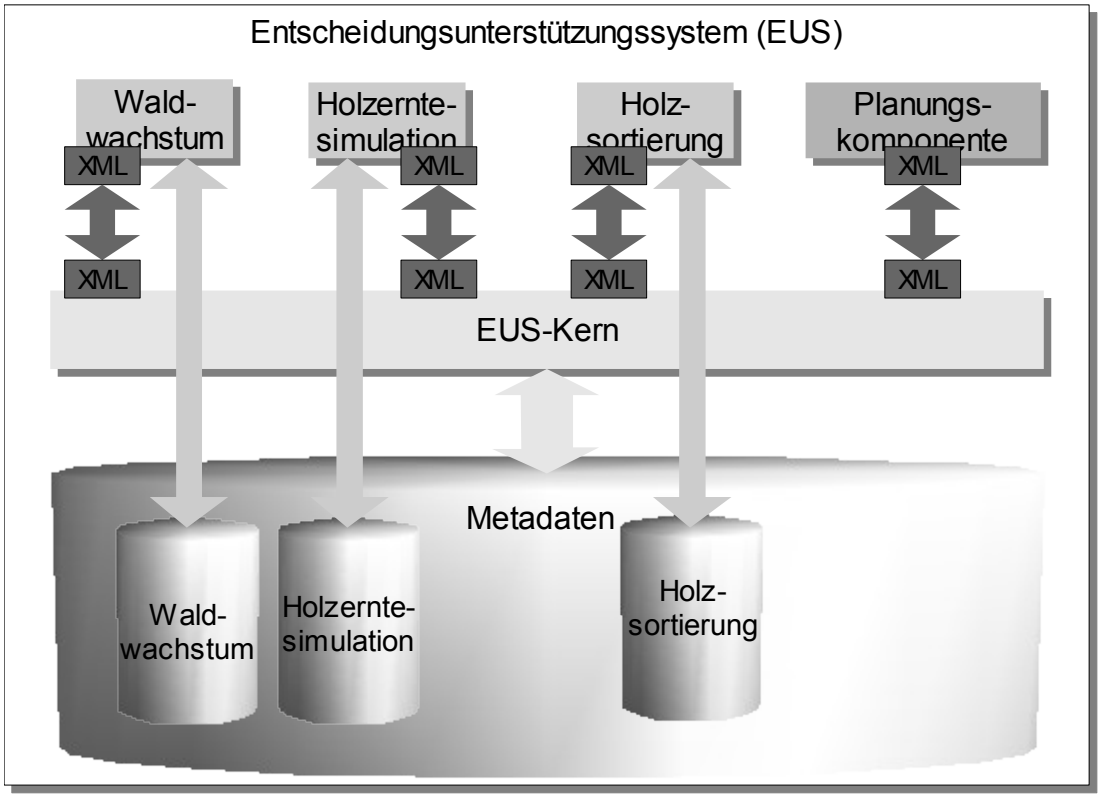


Abbildung 5.1: Schematische Darstellung der KOMET-Architektur

5.1.1.1 Dienste

Wichtigste Eigenschaft der *KOMET-Architektur* ist die Trennung von Dialog-Subsystem, Modell-Subsystem und Daten-Subsystem (vgl. Abbildung 3.2). Wie im Schichtenmodell beschrieben (siehe Abschnitt 4.3.1 bzw. Abbildung 4.7), spielt dabei der EUS-Kern eine besondere Rolle. Er vermittelt zwischen den Subsystemen. Dies wird durch Dienste realisiert, die der Kern zur Verfügung stellt.

Der EUS-Kern implementiert folgende Dienste:

- *Szenario-Informations-Dienst*

  Beim Szenario-Informations-Dienst werden grundlegende Solver-Einstellungen ausgelesen. Dabei kann es sich beispielsweise um die Durchforstungsstrategie oder die Art der Holzernte (motormanuell, mechanisiert) handeln.
- *Szenario-Schreib-Dienst*

  Der Szenario-Schreib-Dienst schreibt die geänderten Daten zurück in die Datenbank.

- *Solverdaten-Schreib-Dienst*

  Die Solver können einzeln aktiviert und deaktiviert werden. Diese Informationen spreichert der Solverdaten-Schreib-Dienst in der Datenbank.
- *Solver-Informations-Dienst*

  Der Solver-Informations-Dienst liest die im REUS vorhandenen Solver aus und ordnet sie auf Grund ihrer Abhängigkeiten in Ebenen. Solver einer Ebene sind von Informationen abhängig, die Solver einer niedrigeren Ebene produzieren.
- *Daten-Informations-Dienst*

  Der Daten-Informations-Dienst liefert Informationen über Solver-Daten. Dabei werden Ontologie-Angaben wie Objektklasse und Zugehörigkeit zu einer Behälterklasse (siehe Abschnitt 5.1.1.3 bzw. Listing 5.2 und Listing 5.3) an den REUS-Kern gesendet, welcher Tabellennamen, Spaltennamen etc. zurückgibt.
- *Daten-Abfrage-Dienst*

  Mit Hilfe des Daten-Abfrage-Dienstes kann ein SQL-Befehl an den EUS-Kern gesendet werden. Das Ergebnis dieses Befehls wird zurückgegeben.
- *Ergebnis-Informations-Dienst*

  Der Ergebnis-Infomations-Dienst liefert Angaben zu allen Solverdaten, die im REUS registriert sind.
- *Ontologie-Abfrage-Dienst*

  Mit Hilfe des Ontologie-Abfrage-Dienstes kann ein Befehl in der Ontologie-Abfragesprache RDQL zur Abfrage von Informationen, die in der Ontologie gespeicher sind, an den EUS-Kern geschickt werden. Das Ergebnis dieses Befehls wird zurückgesendet.
- *Registrierungs-Dienst*

  Solver werden am REUS mit Hilfe des Registrierungs-Dienstes angemeldet.
- *Un-Registrierungs-Dienst*

  Der Un-Registrierungs-Dienst kann dazu verwendet werden, Solver vom REUS abzumelden.

Mit Ausnahme der ersten drei Dienste, die während der Definition der Anwendungsarchitektur ergänzt wurden, gehen die aufgeführten Dienste aus den nichtfunktionalen Anforderungen hervor. Mit Hilfe dieser Dienste übernimmt der EUS-Kern Funktionen des Datenbank Management Systems und des Model Management Systems (siehe Abbildung 3.2).

5.1.1.2 Kommunikations-Schnittstelle

Die Kommunikation mit dem EUS-Kern erfolgt ausschließlich durch den Austausch von XML-Botschaften. Die Vorteile dieser Vorgehensweise sind in Abschnitt 4.3.3 beschrieben. Im Rahmen der vorliegenden Arbeit wird ein XML-Dialekt (*KometML*) mit Hilfe einer ***D**ocument **T**ype **D**efinition (DTD)* erstellt. Ein Auszug aus dieser DTD ist in Listing 5.1 dargestellt, dessen Inhalt im folgenden kurz erklärt wird.

Mit Hilfe des Schlüsselwortes *!ELEMENT* werden Tags innerhalb einer DTD beschrieben. *<!ELEMENT komet (request | response)>* in Zeile 5 bestimmt, dass der Tag *komet* entweder den Tag *request* oder den Tag *response* enthalten kann. Ebenso wird in den Zeilen 6 bis 11 für die Tags request und response bestimmt, welche Tags sie enthalten können. In Zeile 16 wird mit Hilfe des Schlüsselwortes *EMPTY* festgelegt, dass der Tag *error* keine weiteren Tags enthalten darf. Mit Hilfe der Sonderzeichen »*« und »+« kann festgelegt werden, dass sich Tags wiederholen dürfen. Im Fall von »+« muss ein Tag mindestens einmal vorkommen. So kann laut Zeile 24 der Tag *resultTree* beliebig viele *solver*-Tags enthalten. Auf Grund des Sterns ist es erlaubt, dass *resultTree* keinen Tag vom Typ *solver* enthält. Im Gegensatz dazu muss der Tag *solver* (Zeile 28) mindestens einen Tag vom Typ *subSolver* oder *resultSet* enthalten. Attribute werden mit Hilfe des Schlüsselwortes *!ATTLIST* definiert. Sie erhalten einen Namen, einen Typ sowie einen Vermerk ob das Attribut immer im Tag mit angegeben werden muss (*#REQUIRED*) oder ob es auch weggelassen werden darf (*#IMPLIED*). In den Zeilen 42 bis 52 werden für den Tag *result* die verpflichtenden Attribute *id* und *name* definiert sowie eine Reihe von fakultativen Attributen (z. B. *dbName*, *key*, *class*, etc.). Die Zeilen 3 und 4 enthalten Kommentare. Sie beginnen mit »<!--« und enden mit »-->«. Kommentare dürfen sich auch über mehrere Zeilen erstrecken, im Rahmen der vorliegenden Arbeit wurden sie wegen der besseren Übersichtlichkeit am Ende jeder Zeile geschlossen.

Im Rahmen der vorliegenden Arbeit wird in den Zeilen 9 bis 11 für jeden Dienst des EUS-Kerns ein Tag innerhalb des *request*-Tags definiert. Anhand dieses Tags kann ermittelt werden, welcher Dienst beim EUS-Kern angefragt wird. Entsprechend werden Tags für die Dienstantworten in den Zeilen 7 und 8 definiert. Einige Anfragen werden mit einer Erfolgs- oder Fehlermeldung beantwortet.

```
 1 <?xml version='1.0' encoding='utf-8' ?>
 2
 3 <!-- Document Type Definition for KOMET's XML conversation:    -->
 4 <!-- KometML                                                   -->
 5
 6 <!ELEMENT komet (request | response)>
 7 <!ELEMENT response (dataTree | editTree | error | queryOntology |
 8           resultTree | solverTree | success | valueList)>
 9 <!ELEMENT request (dataTree | editTree | queryOntology | register |
10           resultTree | solverTree | unRegister | valueList |
11          writeEditTree | writeSolverTree)>
12
13 <!-- For Error handling the Server can create an error message -->
14 <!-- embedded in komet's XML dialect                           -->
15
16 <!ELEMENT error EMPTY>
17 <!ATTLIST error id   CDATA #REQUIRED
18                name CDATA #REQUIRED>
19
20 <!-- The ResultTree Request consists of an empty resultTree    -->
21 <!-- tag, whereas the Response is populated with the tags:     -->
22 <!-- solver, subSolver, resultSet and result                   -->
23
24 <!ELEMENT resultTree (solver*)>
25 <!ATTLIST resultTree id        CDATA #IMPLIED
26                      name      CDATA #IMPLIED>
27
28 <!ELEMENT solver (subSolver | resultSet)+>
29 <!ATTLIST solver     id        CDATA #REQUIRED
30                      name      CDATA #REQUIRED
31                      active    CDATA #IMPLIED>
32
33 <!ELEMENT subSolver (subSolver | resultSet)+>
34 <!ATTLIST subSolver  id        CDATA #REQUIRED
35                      name      CDATA #REQUIRED>
36
37 <!ELEMENT resultSet (result)+>
38 <!ATTLIST resultSet  id        CDATA #REQUIRED
39                      name      CDATA #REQUIRED>
40
41 <!ELEMENT result EMPTY>
42 <!ATTLIST result     id        CDATA #REQUIRED
43                      name      CDATA #REQUIRED
44                      dbName    CDATA #IMPLIED
45                      key       CDATA #IMPLIED
46                      class     CDATA #IMPLIED
47                      ioTtype   CDATA #IMPLIED
48                      dataType  CDATA #IMPLIED
49                      scope     CDATA #IMPLIED
50                      unit      CDATA #IMPLIED
51                      belongsTo CDATA #IMPLIED
52                      value     CDATA #IMPLIED>
53 ...
```

Listing 5.1: Auszug aus der DTD zur Definition von KometML

Basierend auf diese DTD kann ein geregelter Informationsaustausch zwischen den einzelnen Komponenten und dem EUS-Kern erfolgen. Listing 5.2 zeigt eine Anfrage an den EUS-Kern, in der nach den Objekten *BKZ*, *Variante* und *Aktiv* gesucht wird, die zum Objekt *REUS* gehören. Listing 5.3 zeigt die entsprechende Antwort: Diese Objekte befinden sich in den Spalten *bkz*, *variante* und *aktiv* der Tabelle *reus_input_varianten*.

```
 1 <?xml version="1.0" encoding="utf-8" ?>
 2 <!DOCTYPE komet SYSTEM
 3 'file:K:/user/mascht/promotion/arbeit/xml/komet.dtd'>
 4 <komet>
 5   <request>
 6     <dataTree>
 7       <item ontClass="BKZ" belongsTo="REUS" />
 8       <item ontClass="Variante" belongsTo="REUS" />
 9       <item ontClass="Aktiv" belongsTo="REUS" />
10     </dataTree>
11   </request>
12 </komet>
```

Listing 5.2: XML-Anfrage an den EUS-Kern

```
 1 <?xml version="1.0" encoding="utf-8" ?>
 2 <!DOCTYPE komet SYSTEM
 3 'file:K:/user/mascht/promotion/arbeit/xml/komet.dtd'>
 4 <komet>
 5   <response>
 6     <dataTree>
 7       <table id="reus.variante" name="Varianten"
 8              dbName="reus_input_varianten">
 9         <column ontClass="BKZ" belongsTo="REUS"
10                 id="reus.variante.bkz" name="BKZ" dbName="bkz" />
11         <column ontClass="Variante" belongsTo="REUS"
12                 id="reus.variante.var" name="Variante"
13                 dbName="variante" />
14         <column ontClass="Aktiv" belongsTo="REUS"
15                 id="reus.variante.aktiv" name="Aktiv"
16                 dbName="aktiv" />
17       </table>
18     </dataTree>
19   </response>
20 </komet>
```

Listing 5.3: XML-Antwort des EUS-Kerns auf die Anfrage aus Listing 5.2

Der Antwort in Listing 5.3 ging eine Abfrage der vom EUS-Kern verwalteten Metadaten voraus.

5.1.1.3 Metadaten

Mit Hilfe von Metadaten wird die Organisation der Datenablage der Solver in der Datenbank beschrieben. Die Speicherung der Metadaten erfolgt zweigleisig. Sie werden zum einen in der selben Datenbank wie die Solver-Daten abgelegt und zum anderen als Ontologie gespeichert.

Die relationale Speicherung der Metadaten erfolgt in Form von drei Datenkatalogen:

- Solverkatalog,
- Tabellenkatalog,
- Spaltenkatalog.

Die Datenkataloge implementieren eine logische Sicht auf die Tabellen und Spalten der Solverdaten. Abbildung 5.2 zeigt ein Entity-Relationship-Modell der Metadaten.

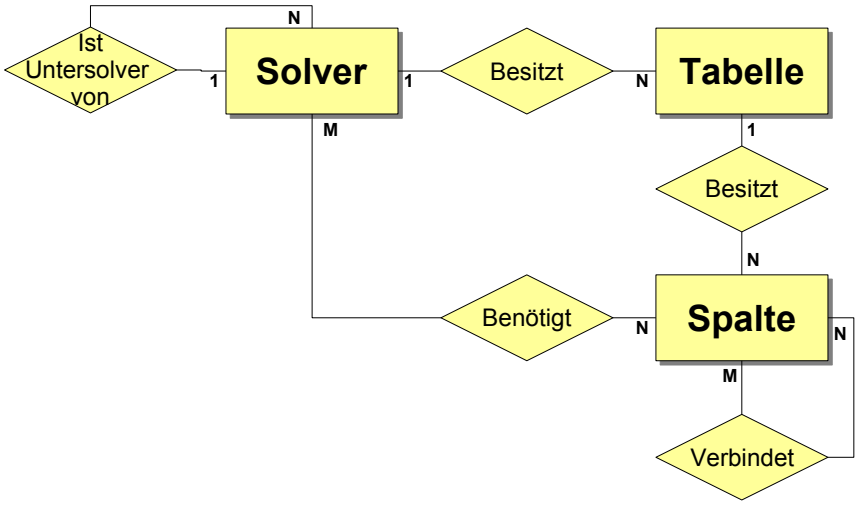


Abbildung 5.2: Entity-Relationship-Modell der Metadaten

Mit Hilfe der Relation *»Solver ist Untersolver von Solver«* kann jeder Solver hierarchisch strukturiert werden. Die erste Zugehörigkeitsrelation *»Solver besitzt Tabelle«* weist den Solvern ihre Tabellen zu. Spalten einzelner Tabellen können mit Hilfe der Relation *»Tabelle besitzt Spalte«* identifiziert werden. Zur Ermittlung von Abhängigkeiten zwischen Solvern kann die Relation *»Solver benötigt Spalte«* verwendet werden. Relationen innerhalb eines solvereigenen Datenmodelles können über die Relation *»Spalte verbindet Spalte«* in den Metadaten gespeichert werden.

Neben dieser relationalen Speicherung der Metadaten wird im Rahmen der vorliegenden Arbeit eine Ontologie verwendet, um forstliche Messwerte und deren Beziehungen untereinander zu beschreiben.

Dabei werden mit Hilfe der Ontologie zunächst Klassen definiert, welche die Struktur der Ontologie festlegen. Mit Hilfe dieser Klassen werden von den Solvern Objekte definiert, welche die Ontologie erweitern. Zwei Arten von Klassen können dabei unterschieden werden:

- Behälterklassen
- Messwertklassen.

Die Behälterklassen, welchen die Messwertklassen mit Hilfe von Beziehungen zugeordnet werden, beschreiben ein objektorientiertes Datenmodell, dessen

Grundstruktur aus dem *ESRI Forestry Datamodel* abgeleitet ist (siehe Abschnitt 4.3.4.2).

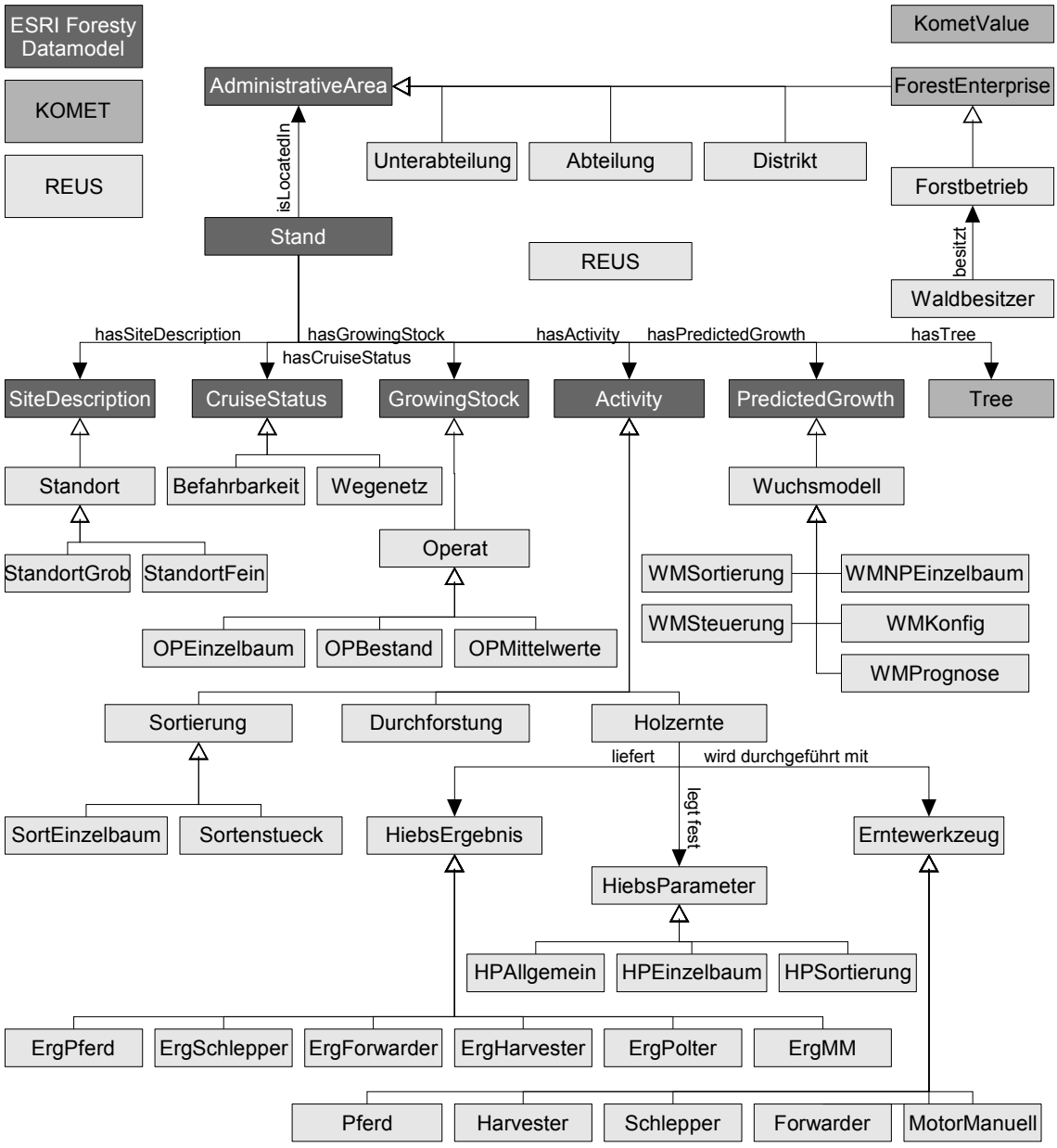


Abbildung 5.3: Schematische Darstellung der Behälterklassen des REUS

Ausgehend von dieser Klassendefinition werden mit Hilfe der Vererbung die Behälterklassen für die Messwertklassen des REUS definiert. Im Rahmen der vorliegenden Arbeit werden die in Abbildung 5.3 schematisch dargestellten Behälterklassen definiert, die das gesamte objektorientierte Datenmodell des REUS beschreiben. Die Notation orientiert sich an UML, das heißt, Pfeile mit einem weißen gleichseitigen Dreieck als Spitze stellen eine Vererbungsbeziehung dar.

Neben den Behälterklassen des REUS werden drei Klassen definiert, die von der *KOMET-Architektur* benötigt werden:

- ForestEnterprise
- Tree
- KometValue.

Jedem Messwertobjekt wird einer der drei Geltungsbereiche *Einzelbaum*, *Bestand* oder *Forstbetrieb* zugewiesen. Für den Geltungsbereich *Bestand* kann die im ESRI Forestry Datamodel vorhandene Klasse *Stand* bzw. dessen Schlüsselattribut *StandID* verwendet werden. Für die Geltungsbereiche *Forstbetrieb* und *Einzelbaum* werden die Behälterklassen *ForestEnterprise* und *Tree* definiert. Sie enthalten den Einzelbaumschlüssel (*TreeID*) sowie den Betriebsschlüssel (*EnterpriseID*) als Messwertklassen, die neben *StandID* als Elternklassen für diese Schlüsselwerte verwendet werden müssen, um sicherzustellen, dass diese als solche erkannt werden. Dabei ist *ForestEnterprise* eine Kindklasse von *AdministrativeArea*.

KometValue ist die Basisklasse aller Messwertklassen. Sie besitzt folgende Attribute:

- *hasScope*
  Geltungsbereich: Einzelbaum (*Tree*), Bestand (*Stand*) oder Forstbetrieb (*Enterprise*)
- *hasUnit*
  Maßeinheit
- *hasIOType*
  Eingabewert (*Input*), Ausgabewert (*Output*) oder Ein- und Ausgabewert (*InOut*)
- *hasDataType*
  Datentyp: *Integer*, *Float*, *String* oder *Date*
- *belongsTo*
  Zugehörigkeit zu einem Behälterobjekt.

Neben den Attributen sind Regeln definiert, die beispielsweise Eingabewerte und Ausgabewerte charakterisieren. Für Geltungsbereiche, Maßeinheiten Ein-/Ausgabetypen und Datentypen werden ebenfalls Klassen und Objekte definiert. Diese Definitionen sind beliebig erweiterbar. Die Definition der Klasse *KometValue* ist in Listing 5.4 in Auszügen dargestellt.

```
 1 <?xml version="1.0"?>
 2
 3 <!DOCTYPE rdf:RDF [
 4   <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
 5   <!ENTITY owl "http://www.w3.org/2002/07/owl#">
 6 ]>
 7
 8 <rdf:RDF
 9   xmlns     ="http://www.forst.wzw.tum.de/01-komet#"
10   xml:base  ="http://www.forst.wzw.tum.de/01-komet#"
11   xmlns:owl ="http://www.w3.org/2002/07/owl#"
12   xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
13   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
14   xmlns:xsd ="http://www.w3.org/2001/XMLSchema#">
15
16   <owl:Ontology rdf:about="">
17   </owl:Ontology>
18 ...
21   <owl:Class rdf:ID="KometValue">
22     <rdfs:subClassOf>
23       <owl:Restriction>
24         <owl:onProperty rdf:resource="#hasDataType" />
25         <owl:cardinality
26          rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
27       </owl:Restriction>
28     </rdfs:subClassOf>
29     <rdfs:subClassOf>
30       <owl:Restriction>
31         <owl:onProperty rdf:resource="#hasIOType" />
32         <owl:cardinality
33          rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
34       </owl:Restriction>
35     </rdfs:subClassOf>
36     <rdfs:subClassOf>
37       <owl:Restriction>
38         <owl:onProperty rdf:resource="#hasScope" />
39         <owl:cardinality
40          rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
41       </owl:Restriction>
42     </rdfs:subClassOf>
43     <rdfs:subClassOf>
44       <owl:Restriction>
45         <owl:onProperty rdf:resource="#hasUnit" />
46         <owl:cardinality
47          rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
48       </owl:Restriction>
49     </rdfs:subClassOf>
50     <rdfs:subClassOf>
51       <owl:Restriction>
52         <owl:onProperty rdf:resource="#belongsTo" />
53         <owl:cardinality
54          rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
55       </owl:Restriction>
56     </rdfs:subClassOf>
57   </owl:Class>
58 ...
59   <owl:Class rdf:ID="InputValue">
60     <owl:intersectionOf rdf:parseType="Collection">
61       <owl:Class rdf:about="#KometValue" />
62       <owl:Restriction>
63         <owl:onProperty rdf:resource="#hasIOType" />
64         <owl:hasValue rdf:resource="#Input" />
65       </owl:Restriction>
66     </owl:intersectionOf>
67   </owl:Class>
68 ...
69 </rdf:RDF>
```

Listing 5.4: komet-core.owl - Auszug aus der Definition der Klasse KometValue

Für jeden Messwert (z. B. *BHD*, *Vorrat*, *Kraftstoff-Verbrauch* etc.) muss eine Klasse definiert werden. Die im Rahmen der vorliegenden Arbeit erstellte Ontologie enthält Klassen, welche alle Messwerte beschreiben, die im REUS zur Holzernte benötigt werden. Diese Messwertklassen werden Behälterklassen zugeordnet und erhalten entweder *KometValue* als Elternklasse oder eine der Messwertklassen des *ESRI Forestry Datamodel*, die ihrerseits Kindklassen von *KometValue* sind.

Die Ontologie ist modular aufgebaut. In separaten Dateien erfolgt die Definition der Basisklasse *KometValue* (*komet-core.owl*), der Klassen des ESRI Forestry Datamodel (*forestry-datamodel.owl*), der Klassen des REUS (*komet-sdss.owl*) sowie der Objekte der Solver (*<name-des-solvers>.owl*). Diese Dateien werden von *komet.owl* importiert. Durch diese Import-Beziehungen dient *komet.owl* als zentrale Ontologie-Datei der *KOMET-Architektur* mit deren Hilfe alle Informationen abgefragt werden können. Listing 5.5 zeigt die Datei *komet.owl* für das im Rahmen der vorliegenden Arbeit implementierte REUS zur Holzernte.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE rdf:RDF [
  <!ENTITY kc "http://www.forst.wzw.tum.de/01-komet#">
  <!ENTITY ko "http://www.forst.wzw.tum.de/01-komet-sdss#">
]>

<rdf:RDF xmlns="http://www.forst.wzw.tum.de/01-komet-sdss#"
         xml:base="http://www.forst.wzw.tum.de/01-komet-sdss#"
         xmlns:owl="http://www.w3.org/2002/07/owl#"
         xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
  <owl:Ontology rdf:about="">
    <owl:imports rdf:resource="file:reus.owl" />
    <owl:imports rdf:resource="file:gis.owl" />
    <owl:imports rdf:resource="file:silva.owl" />
    <owl:imports rdf:resource="file:ernte.owl" />
    <owl:imports rdf:resource="file:amod.owl" />
  </owl:Ontology>
</rdf:RDF>
```

Listing 5.5: komet.owl - zentrale Ontologie-Datei des REUS zur Holzernte

Pro Solver wird für jeden Messwert ein Objekt definiert sowie Werte für die Attribute, die in *KometValue* beschrieben sind. Diese Informationen werden mit Hilfe von KometML mit der Registrierungsanfrage an den EUS-Kern übertragen, der eine Ontologiedatei erzeugt und in das REUS einbindet. Somit stehen dem REUS die Messwerte eines jeden Solvers als Objekte zur Verfügung, deren Attribute sowie deren Beziehungen untereinander abgefragt werden können.

Die Definition von drei Objekten des Solvers *silva* aus der Datei *silva.owl* ist in Listing 5.6 dargestellt.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE rdf:RDF [
  <!ENTITY kc "http://www.forst.wzw.tum.de/01-komet#">
  <!ENTITY ko "http://www.forst.wzw.tum.de/01-komet-sdss#">
]>

<rdf:RDF xmlns="http://www.forst.wzw.tum.de/01-komet-sdss#"
         xml:base="http://www.forst.wzw.tum.de/01-komet-sdss#"
         xmlns:kc="http://www.forst.wzw.tum.de/01-komet#"
         xmlns:ko="http://www.forst.wzw.tum.de/01-komet-sdss#"
         xmlns:owl="http://www.w3.org/2002/07/owl#"
         xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
  <owl:Ontology rdf:about="">
    <owl:imports rdf:resource="file:komet-sdss.owl" />
  </owl:Ontology>
  <ko:BKZ rdf:ID="silva.best.sort.bkz">
    <kc:hasIOType rdf:resource="&kc;InOut" />
    <kc:hasDataType rdf:resource="&kc;String" />
    <kc:hasScope rdf:resource="&kc;Stand" />
    <kc:hasUnit rdf:resource="&kc;NoUnit" />
    <kc:belongsTo rdf:resource="&ko;WMSortierung" />
  </ko:BKZ>
...
  <ko:SortTyp rdf:ID="silva.best.sort.sorttyp">
    <kc:hasIOType rdf:resource="&kc;Output" />
    <kc:hasDataType rdf:resource="&kc;Integer" />
    <kc:hasScope rdf:resource="&kc;Stand" />
    <kc:hasUnit rdf:resource="&kc;NoUnit" />
    <kc:belongsTo rdf:resource="&ko;WMSortierung" />
  </ko:SortTyp>
...
  <ko:Volumen rdf:ID="silva.eb.natural.vol">
    <kc:hasIOType rdf:resource="&kc;Output" />
    <kc:hasDataType rdf:resource="&kc;Float" />
    <kc:hasScope rdf:resource="&kc;Tree" />
    <kc:hasUnit rdf:resource="&kc;CubicMeter" />
    <kc:belongsTo rdf:resource="&ko;WMNPEinzelbaum" />
  </ko:Volumen>
...
</rdf:RDF>
```

Listing 5.6: *silva.owl - Definition der Objekte* silva.best.sort.bkz, silva.best.sort.sorttyp *sowie* silva.eb.natural.vol

5.1.1.4 EUS-Kern

5.1.1.4.1 Architektur

Der EUS-Kern der *KOMET-Architektur* besteht aus 19 Klassen und drei Interfaces. Zentrale Bausteine der Architektur sind die abstrakte Klasse *KometTree* sowie die konkreten Klassen *KometList* und *KometItem*. Ein *KometList*-Objekt kann ein oder mehrere Objekte des Typs *KometTree* (*KometList* oder *KometItem*) enthalten, die ihrerseits *KometTree*-Objekte enthalten können. Auf diese Weise kann eine verschachtelte Struktur, ein so genannter Baum generiert werden. *KometItem* kann keine

KometTree-Objekte enthalten und wird in einer Baumstruktur für so genannte Blattobjekte verwendet. *KometList* und *KometItem* können darüber hinaus beliebig viele Objekte des Typs *KometAttribute* beinhalten, welche neben dem Attributnamen einen beliebigen Wert speichern können. *KometList* und *KometItem* können somit als universeller hierarchischer Datenspeicher dienen und implementieren das in GAMMA ET AL. [2000] beschriebene *Composite-Muster*. Da XML-Dokumente hierarchisch aufgebaut sind, können diese sehr einfach in eine Baumstruktur umgewandelt werden, die aus *KometList*- und *KometItem*-Objekten besteht. Für den Registrierungs-Dienst ist eine weitere Klasse *KometRegList* vorgesehen, die zusätzlich ein Registrierungsobjekt besitzt, welche das Interface *KometRegHandler* implementiert. So kann jedem Objekt vom Typ *KometRegList* individuell die geeignete Registrierungsmethode zugewiesen werden. Auch *KometRegList*-Objekte können zu einer Baumstruktur zusammengesetzt werden.

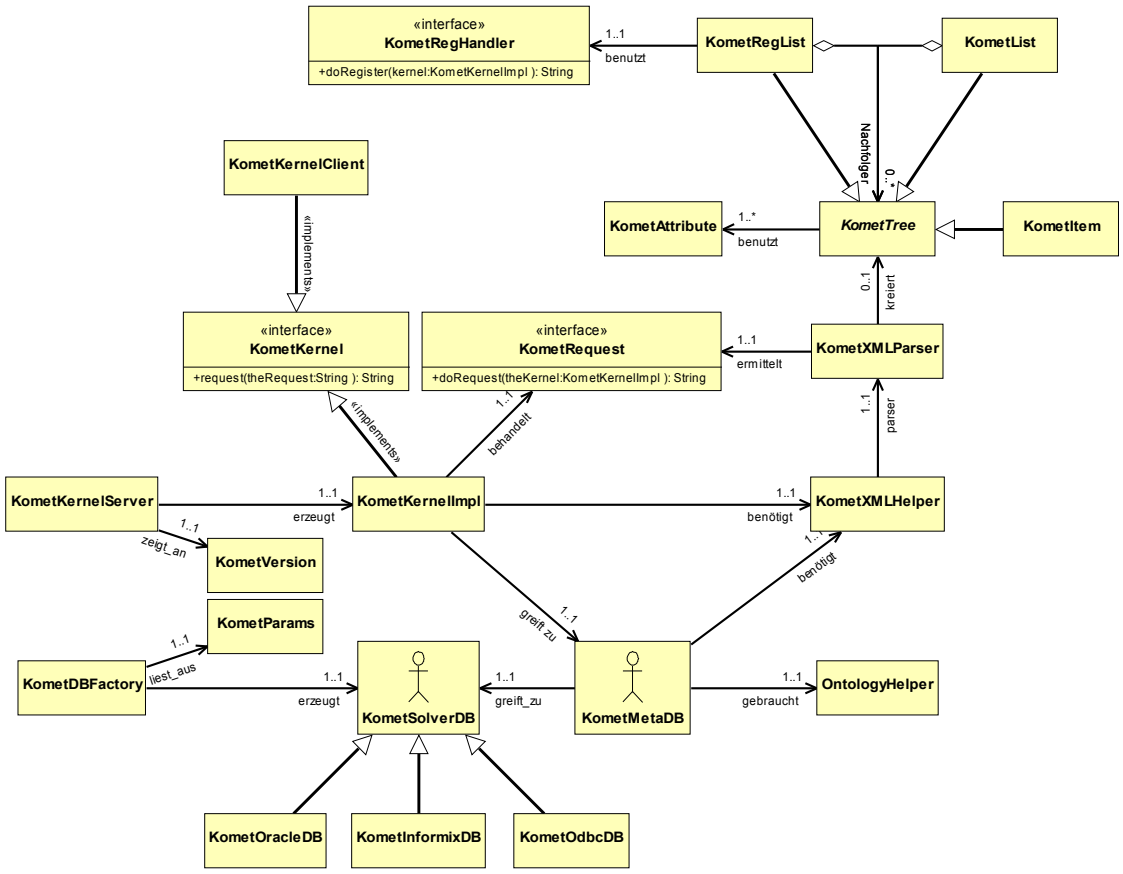


Abbildung 5.4: Klassendiagramm des EUS-Kerns

Für die Verarbeitung der Dienst-Anfragen ist die Klasse *KometMetaDB* verantwortlich. Für jeden Dienst existiert eine Methode, die nach der Auswertung der XML-Anfrage

entsprechend aufgerufen wird. Die Auswertung der XML-Anfragen wird von den Klassen *KometXMLHelper* bzw. *KometXMLParser* durchgeführt. Bei *KometXML-Parser* handelt es sich um eine Erweiterung des Syntax-Analysators (*Parser*) des SAX-Standards (***S**imple **A**PI for **X**ML processing*). Dieser so genannte *SAX-Parser* wird von der Java-Klassenbibliothek zur Verfügung gestellt. *KometXMLHelper* und *KometXML-Parser* sind auch für die Umwandlung eines XML-Textes in eine Baumstruktur bestehend aus *KometList*-, *KometItem*- bzw. *KometRegList*-Objekten und umgekehrt verantwortlich. Das Klassendiagramm des EUS-Kerns ist in Abbildung 5.4 dargestellt.

5.1.1.4.2 Schnittstellen

Als Aufruf-Schnittstelle des EUS-Kerns dient das Interface *KometKernel*. Dessen einzige Methode ist `KometKernel.request`. Als Methodenparameter wird eine Zeichenkette (*String*) mit der Dienstanfrage angegeben, die Dienstantwort wird von der Methode als Zeichenkette zurückgegeben.

Das Interface *KometKernel* wird von zwei Klassen implementiert:

- *KometKernelImpl*
  die eigentliche Serverklasse, die die Dienstanfragen auswertet und an die Klasse *KometMetaDB* weiterleitet. Eine Instanz von *KometKernelImpl* wird von der Klasse *KometKernelServer* erzeugt. Sie dient zum Start und zur Initialisierung des EUS-Kerns.
- *KometKernelClient*
  eine lokale Platzhalterklasse, die dazu dient, die Netzwerk-Kopplung an *KometKernelImpl* zu initialisieren und Anfragen weiterzuleiten. Dadurch werden netzwerkspezifische Implementationsdetails der *KOMET-Architektur* vom Programmierer ferngehalten.

Der Aufruf von `KometKernelClient.request` wird an *KometKernelImpl* weitergeleiet. Mit Hilfe von `KometXMLHelper.typeOfRequest` wird ermittelt, um welche Art von Dienst-Anfrage es sich handelt. Dabei wird ein Objekt vom Typ *KometRequest* übergeben, dessen Methode `KometRequest.doRequest` entsprechend der Art der Dienst-Anfrage initialisiert wurde. *KometKernelImpl* ruft indirekt über `Komet-Request.doRequest` die entsprechende Methode von *KometMetaDB* auf.

Das Sequenzdiagramm von `KometKernelClient.request` sowie das Aktivitätsdiagramm von `KometKernelImpl.request` sind in den Abbildungen 5.5 und 5.6 dargestellt.

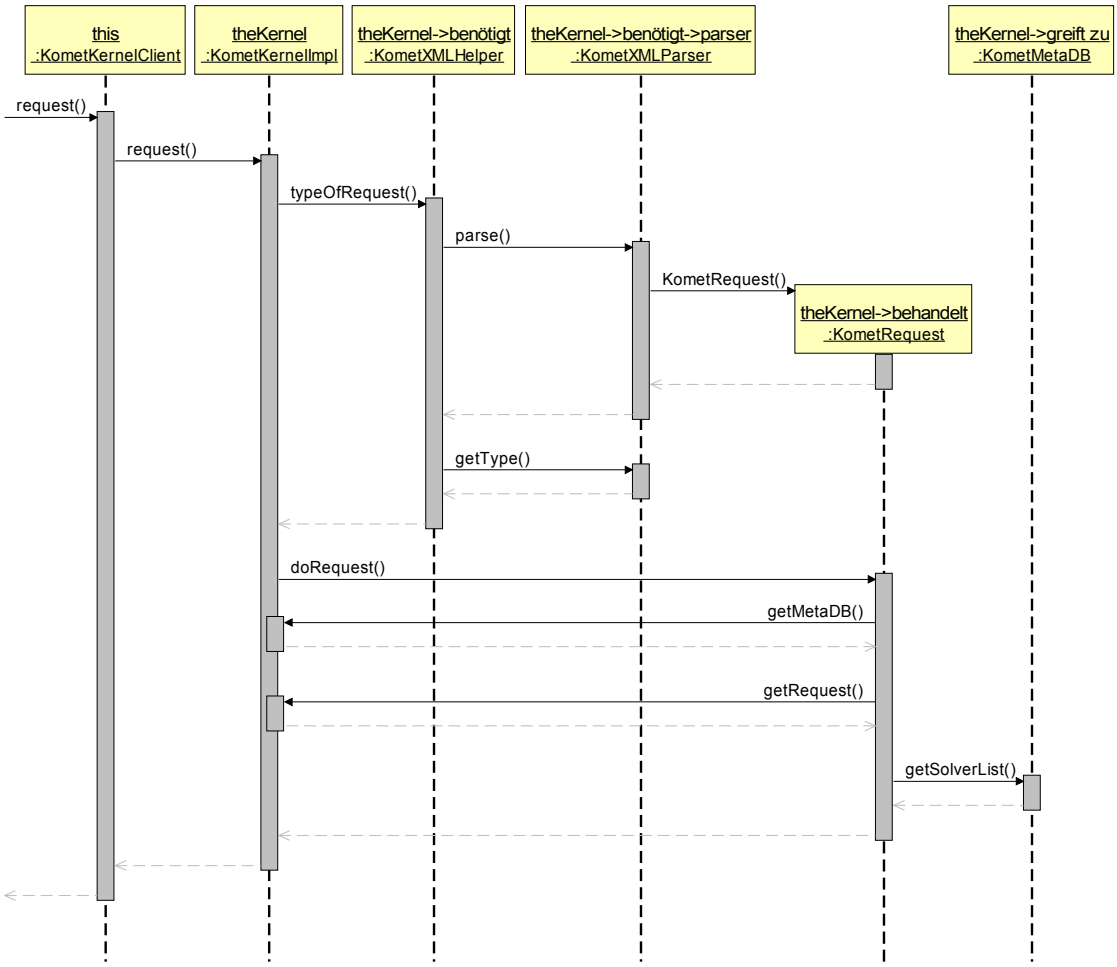


Abbildung 5.5: Sequenzdiagramm von `KometkernelClient.request`

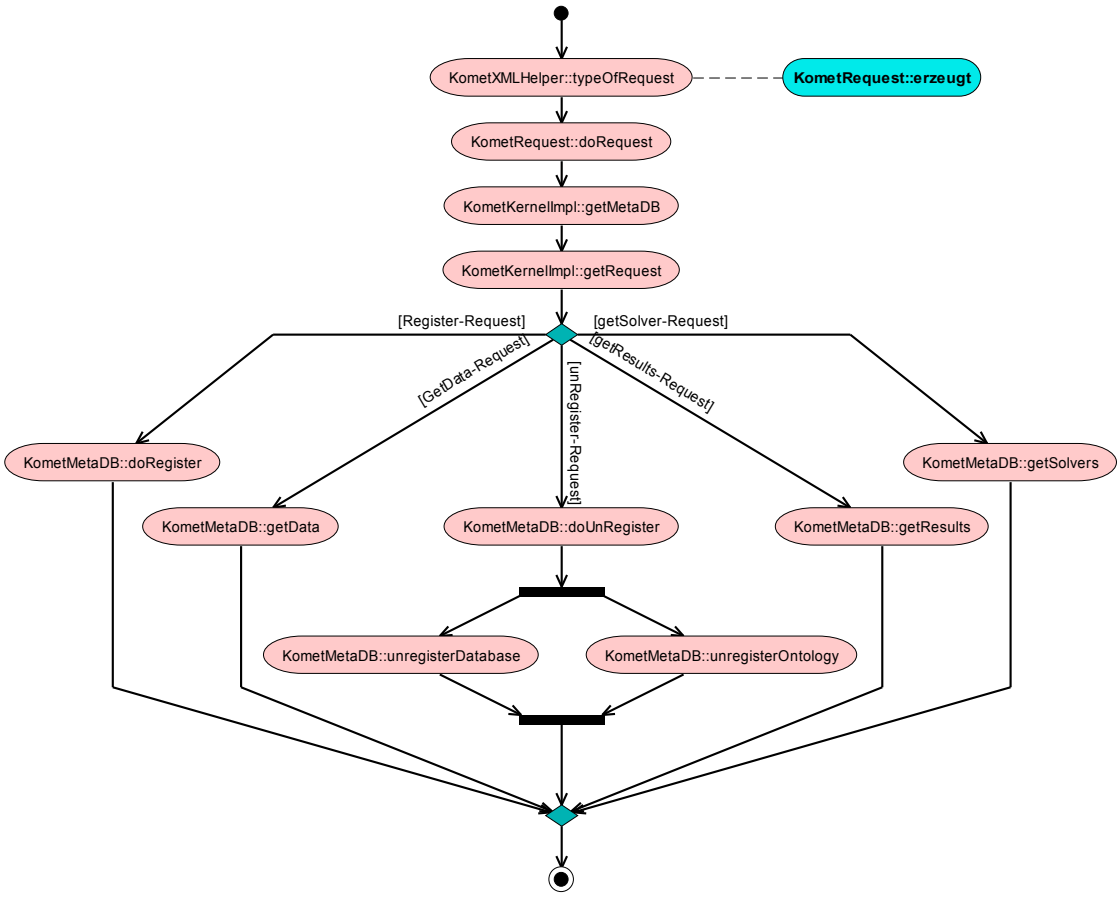


Abbildung 5.6: Aktivitätsdiagramm von `KometKernelImpl.request`

Die Initailisierung der Datenbank erfolgt im EUS-Kern und ist so ausgelegt, dass verschiedene Datenbanken verwendet werden können. Zur Zeit ist die Nutzung folgender Datenbanken möglich:

- Informix
- Oracle
- ODBC.

Welche Datenbank benutzt wird, ist im EUS-Kern konfiguriert. Um sicherzustellen, dass alle Programme dieselbe Datenbank benutzen, muss die Datenbank über die Klasse *KometDBFactory* initialisiert werden. Deren Klassenmethode `createDB` liest die Einstellungen aus, erzeugt das entsprechende Datenbankobjekt und gibt dieses an das aufrufende Objekt zurück.

Weitere Datenbanken können durch Definition einer Kindklasse von *KometSolverDB* und Anpassung der Methode `KometDBFactory.createDB` hinzugefügt werden.

Die Klasse Komet*XMLHelper* stellt Hilfsmethoden zur Verarbeitung von XML zur Verfügung. Insbesondere kann mit der Methode `createTreeFromXML` ein Text, der XML enthält, in eine Baumstruktur umgewandelt werden, die aus *KometList*- und *KometItem*-Objekten besteht. Analog erstellt die Methode `createXMLFromTree` einen XML-Text aus einer solchen Struktur. Neben `createResponse`, `createRequest`, `createMessage`, `createOntology` und `typeOfRequest` kann mit Hilfe der Methode `findError` geprüft werden, ob es sich bei einem XML-Text um eine Fehlermeldung handelt.

OntologyHelper stellt mit `createOntologyTree`, `readOntology`, `writeOntology` und `queryOntology` Methoden zur Verarbeitung von Ontologien und OWL-Dateien zur Verfügung.

5.1.1.5 Benutzeroberfläche

Obwohl nur die Planungskomponente über eine Benutzeroberfläche verfügt, werden auf Grund der Wiederverwendbarkeit einige zentrale Bestandteile in den EUS-Kern verlegt. Der Systemkern übernimmt so Aufgaben des Dialog Generierungs und Management Systems wahr (siehe Abbildung 3.2) Nachfolgend werden diese Elemente näher beschrieben.

Das Programmfenster der Demonstrationsanwendung zur Holzernte ist in Abbildung 5.7 dargestellt. Werden die vom EUS-Kern zur Verfügung gestellten Elemente der Benutzeroberfläche verwendet, sieht die damit erstellte REUS-Anwendung ähnlich aus wie die Demonstrationsanwendung. Voraussetzungen sind, dass zur Erstellung einer REUS-Anwendung *Java* als Programmiersprache und *Swing* als Benutzeroberflächen-Bibliothek eingesetzt werden.

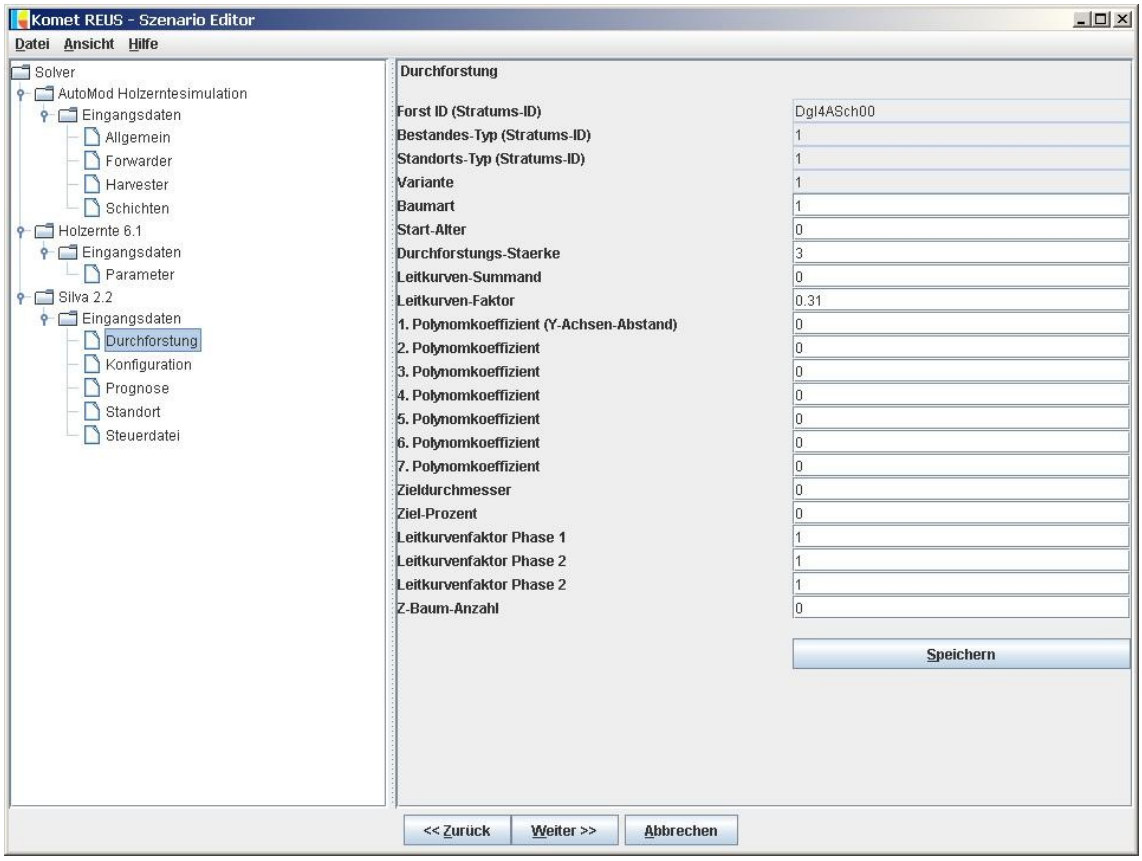


Abbildung 5.7: Programmfenster der REUS-Anwendung zur Holzernte

5.1.1.5.1 Architektur

Neben den zentralen Datenspeicherklassen *KometTree*, *KometList*, *KometItem* und *KometAttribute* stellt die *KOMET-Architektur* für den Aufbau einer Benutzeroberfläche elf Klassen und zwei Interfaces bereit. Das Anwendungsfenster wird durch die Klasse *KometAppWindow* definiert und dargestellt. Die visuellen Komponenten werden in Menüleiste (*KometMenu*), Fensterinhalt (*KometFrameContent*) und Statusleiste (*KometStatusBar*) unterteilt. Der Fensterinhalt wird durch einen verschiebbaren Balken, ähnlich wie im Windows Explorer, in einen linken Teil und einen rechten Teil getrennt. Im linken Teil befindet sich eine Baumansicht (*KometTreeView*), im rechten Teil ein Informations- oder Editierbereich (*KometInfoDialog*, bzw. *KometEditDialog*). Wenn der Benutzer einen Menüpunkt auswählt oder auf einen Knopf klickt, wird eine Ereignismethode der Fensterklasse *KometAppWindow* aufgerufen. Kann die Fensterklasse nicht selbständig auf das Ereignis reagieren, wird der Aufruf an eine entsprechende Methode des Interface *KometView* weitergeleitet, welches im Zuge der Erstellung einer REUS-Anwendung implementiert werden muss. Als Hauptprogramm

ist das Interface *KometPlan* vorgesehen, welches ebenfalls implementiert werden muss. Das entsprechende Klassendiagramm ist in Abbildung 5.8 dargestellt.

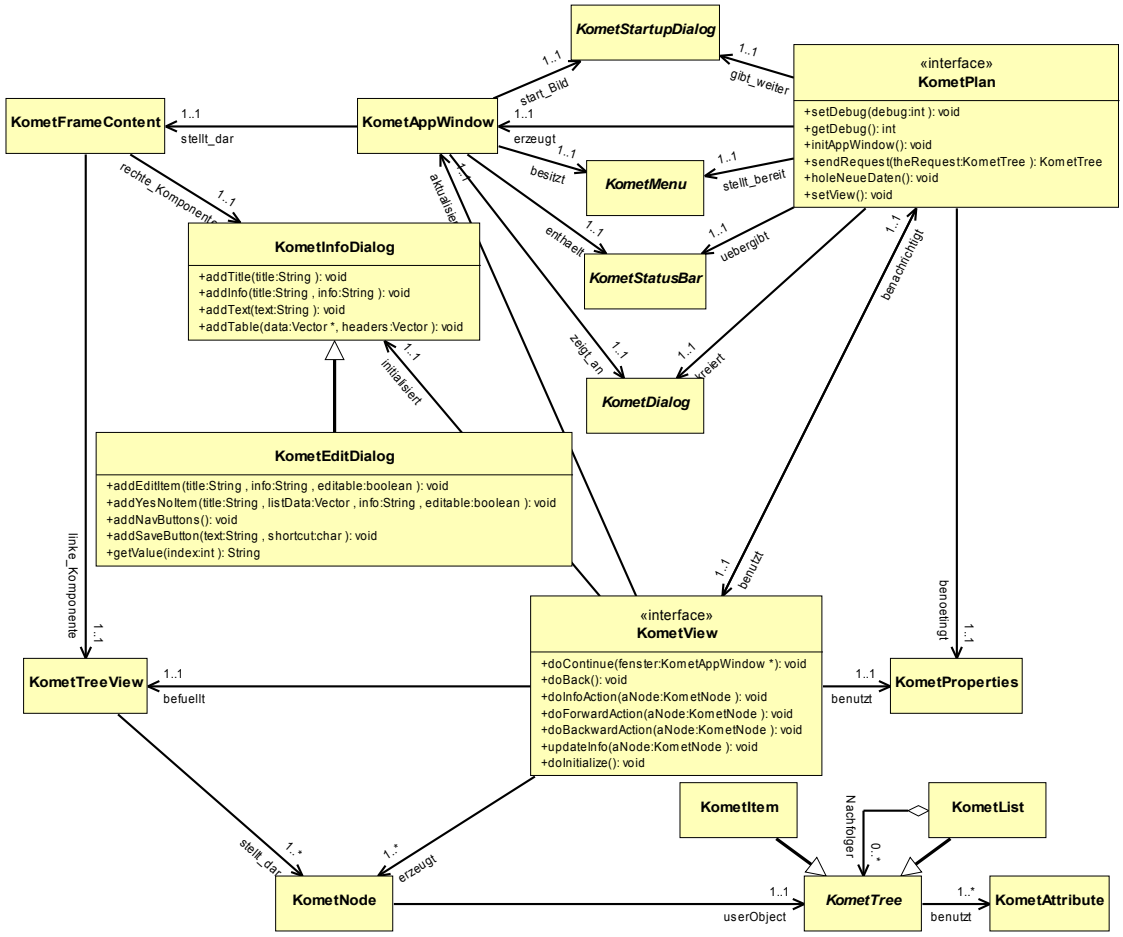


Abbildung 5.8: Klassendiagramm der Benutzeroberfläche

5.1.1.5.2 Schnittstellen

Direkt nach dem Start des Programms soll die Initialisierung des Anwendungsfensters stattfinden. Dies kann mit Hilfe der Methode `KometPlan.InitAppWindow` erfolgen. Während der Initialisierung müssen Startbildschirm, Menüleiste, Statusleiste und ein Informationsfenster an das Fensterobjekt übergeben werden. Dabei ist das Startbildschirm-Objekt im Konstruktor als Parameter anzugeben, die restlichen Objekte werden dem Fenster-Objekt mit Hilfe der Methoden `setMenu`, `setStatusBar` und `setAboutDialog` übergeben. Dazu müssen die abstrakten Klassen *KometStartupDialog*, *KometMenu*, *KometStatusBar* und *KometDialog* erweitert werden.

Zur Bedienung der REUS-Anwendung, stehen in der Fensterklasse Methoden bereit, die es erlauben, schrittweise vorwärts (`KometAppWindow.programContinue`) und

rückwärts (`KometAppWindow.programBack`) zu navigieren. Diese Art der Bedienung ist sehr ähnlich zu jener von so genannten Assistenten, die im Windows-Umfeld häufig anzutreffen sind. Im Interface *KometView* sind die Methoden `doContinue` und `doBack` definiert, die von `programContinue` bzw. `programBack` aufgerufen werden. Ein Mechanismus, durch den `programBack` bzw. `programContinue` aufgerufen werden (z. B. Knöpfe in der Statusleiste oder Menüpunkte) und eine geeignete Implementierung von `KometView.doContinue` bzw. `KometView.doBack` genügen, um das Grundgerüst einer REUS-Anwendung zu programmieren.

Zur Anzeige von Informationen steht der Inhaltsbereich zur Verfügung, der standardmäßig zweigeteilt und dem Windows Explorer sehr ähnlich ist. Im linken Bereich befindet sich eine Baumansicht, die mit Hilfe der Klassen *KometTreeView* und *KometNode* mit Informationen gefüllt werden kann. Einem *KometNode* kann direkt ein Datenobjekt (z. B. *KometList* oder *KometItem*) zugewiesen werden. Wenn der Anwender auf ein Blatt im Baum klickt, wird ein Ereignis ausgelöst und automatisch die Methode `KometView.updateInfo` aufgerufen. Abhängig von der jeweiligen Implementierung von `updateInfo` können nahezu beliebige Aktionen durchgeführt werden. Zur Anzeige von Informationen im rechten Teil des Inhaltsbereichs steht die Klasse *KometInfoDialog* zur Verfügung. Soll der Anwender die Gelegenheit haben Daten zu verändern, kann *KometEditDialog* verwendet werden. Benutzereingaben können in der Methode `KometView.doInfoAction` verarbeitet werden, wenn dem Dialog mit Hilfe von `KometEditDialog.addSaveButton` ein Knopf hinzugefügt wurde. Es besteht die Möglichkeit, beliebige Funktionalität durch Erweiterung der Klassen *KometInfoDialog* bzw. *KometEditDialog* hinzuzufügen. Ebenso kann *KometAppWindow* erweitert werden, um die Implementation weiterer Funktionen zu ermöglichen.

5.1.2 Komponentenarchitektur

5.1.2.1 Solver

Die Solver sind die Problemlösungskomponenten der *KOMET-Architektur*. Sie entsprechen weitgehend dem Modellsubsystem eines EUS aus IT-orientierter Sicht (siehe Abschnitte 4.3.1 bzw. 3.1.4). Ihr grundlegender Aufbau wird im Folgenden genauer beschrieben.

5.1.2.1.1 Architektur

Für die Definition der Solver stellt die *KOMET-Architektur* die drei Klassen *KometSolverClient*, *KometSolverImpl* und *KometKernelHelper* sowie das Interface *KometSolver* bereit. Hinzu kommen eine Reihe von modulübergreifenden Hilfsklassen. Dazu gehören neben den Datenspeicherklassen *KometTree*, *KometList*, *KometItem* und *KometAttribute* die Klassen *KometXMLHelper*, *KometDBFactory* und *KometSolverDB* mit ihren drei Kindklassen *KometOdbcDB*, *KometInformixDB* und *KometOracleDB*. Sie können unter anderem zur Kommunikation mit dem EUS-Kern verwendet werden.

Um einen Solver zu erstellen, muss das Interface *KometSolver* implementiert werden. Nach dem Start warten die Solver auf ihre Aktivierung. Dies geschieht über die Methode `KometSolver.run`, die jeder Solver implementieren muss. Ein Solver wird über den mit Java mitgelieferten Namens-Dienst gefunden. Die Klasse *KometSolverImpl*, die während der Initialisierung des Solvers eingebunden wird, übernimmt die Registrierung beim Namens-Dienst.

Die Klasse *KometSolverClient* ist wie *KometKernelClient* eine lokale Platzhalterklasse, die netzwerkspezifische Implementierungsdetails vom Programmierer fernhält. Bei der Initialisierung werden die notwendigen Netzwerkeinstellungen vorgenommen und der Aufruf von `KometServerClient.run` wird an den Solver weitergeleitet. Das zugehörige Klassendiagramm ist in Abbildung 5.9 dargestellt.

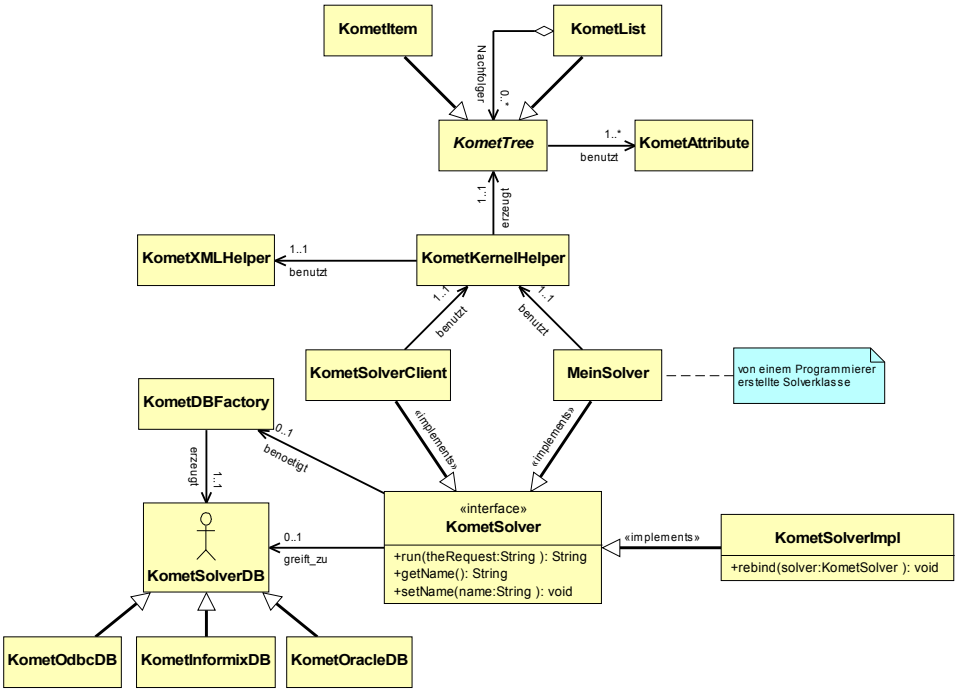


Abbildung 5.9: Klassendiagramm der Solver

5.1.2.1.2 Schnittstellen

Analog zum EUS-Kern besitzen die Solver das Interface *KometSolver*. Mit Hilfe dessen einziger Methode `KometSolver.run` werden sie aktiviert. Auch diese Methode besitzt einen String als Parameter und gibt einen String zurück. Im Rahmen der vorliegenden Arbeit werden diese Parameter nicht ausgewertet. Sie können z. B. genutzt werden, um an die Solver ebenfalls XML-Botschaften zu senden. `KometSolver.run` wird im Rahmen der vorliegenden Arbeit genutzt, um weitere private Methoden der einzelnen Solver aufzurufen.

Die Klasse *KometSolverHelper* stellt den Solvern mit `getOptionInfo`, `sendRequest`, `createDataTreeRequest`, `createDataTreeItem`, `createValueListRequest` und `createValueListItem` Hilfsmethoden zur Generierung häufig benötigter Dienstanfragen an den EUS-Kern zur Verfügung.

5.1.2.2 Planungskomponente

Wie bereits in Abschnitt 5.1.1.4 dargestellt, besteht die Planungskomponente weitgehend aus Implementationen der Interfaces *KometView* und *KometPlan*. Die Funktionalität der Planungskomponente wird durch ihr Zusammenwirken maßgeblich bestimmt. Nachfolgend wird dieser Mechanismus genauer beschrieben.

5.1.2.2.1 Architektur

Die Planungskomponente könnte aus den Klassen bestehen, die in Abbildung 5.10 dargestellt sind. Die Implementierung von *KometPlan* (*MeinPlanungsModul*) steuert dabei das Verhalten der Anwendung. Es werden mehrere Implementationen von *KometView* erzeugt (*MeinEinstellungsView*, *MeinSolverStartView*, *MeinAuswertungs-View*), welche von *MeinPlanungsModul* in einer bestimmten Reihenfolge mit *KometAppWindow* verknüpft und dadurch angezeigt werden. Die Daten für die *KometView* Implementationen stammen vom EUS-Kern, werden von *MeinPlanungs-Modul* abgefragt, mit Hilfe von *XMLHelper* aufbereitet und an *MeinEinstellungsView*, *MeinSolverStartView* sowie *MeinAuswertungsView* weitergeleitet. Erfolgt die Aktivierung der Solver mit Hilfe von *KometSolverClient*, werden die notwendigen Netzwerkeinstellungen automatisch vorgenommen.

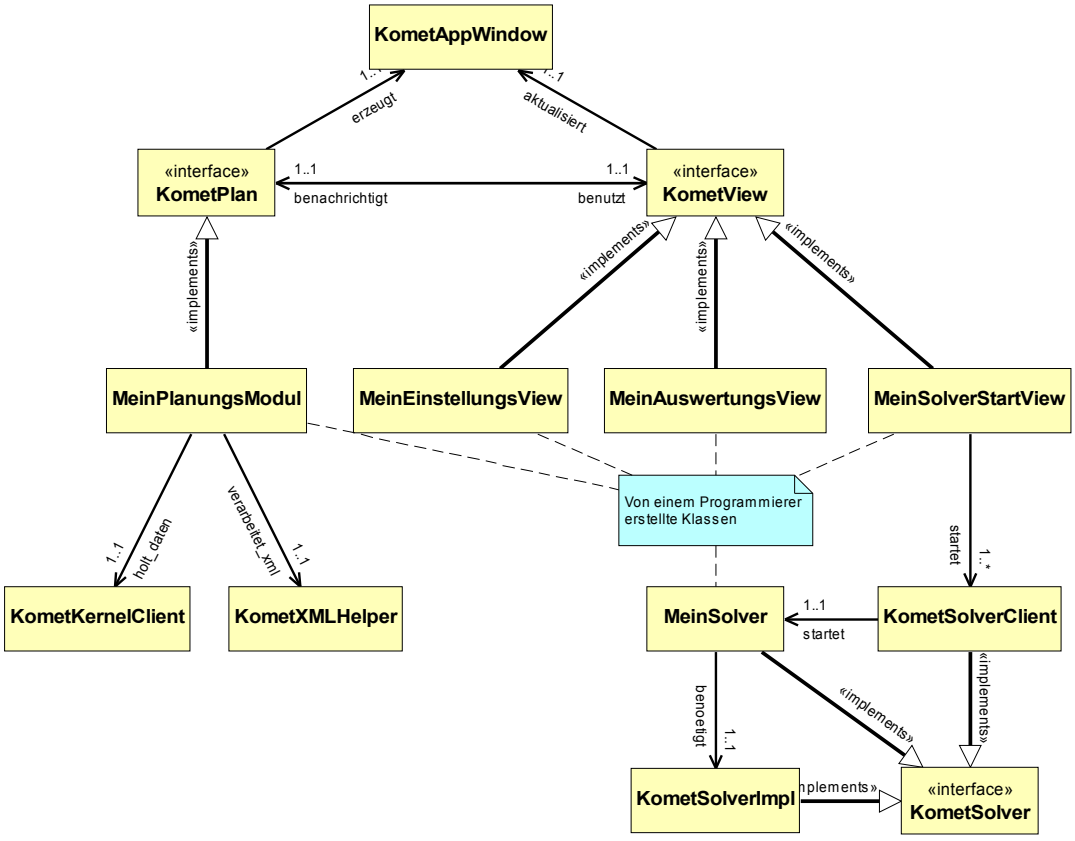


Abbildung 5.10: Klassendiagramm einer einfachen Planungsanwendung

5.1.2.2.2 Schnittstellen

Wie in Abschnitt 5.1.1.4 dargestellt, kann die Initialisierung des Anwendungsfensters nach dem Start des Programms mit Hilfe der Methode `KometPlan.initAppWindow` erfolgen. Das entsprechende Sequenzdiagramm ist in Abbildung Abbildung 5.11 dargestellt.

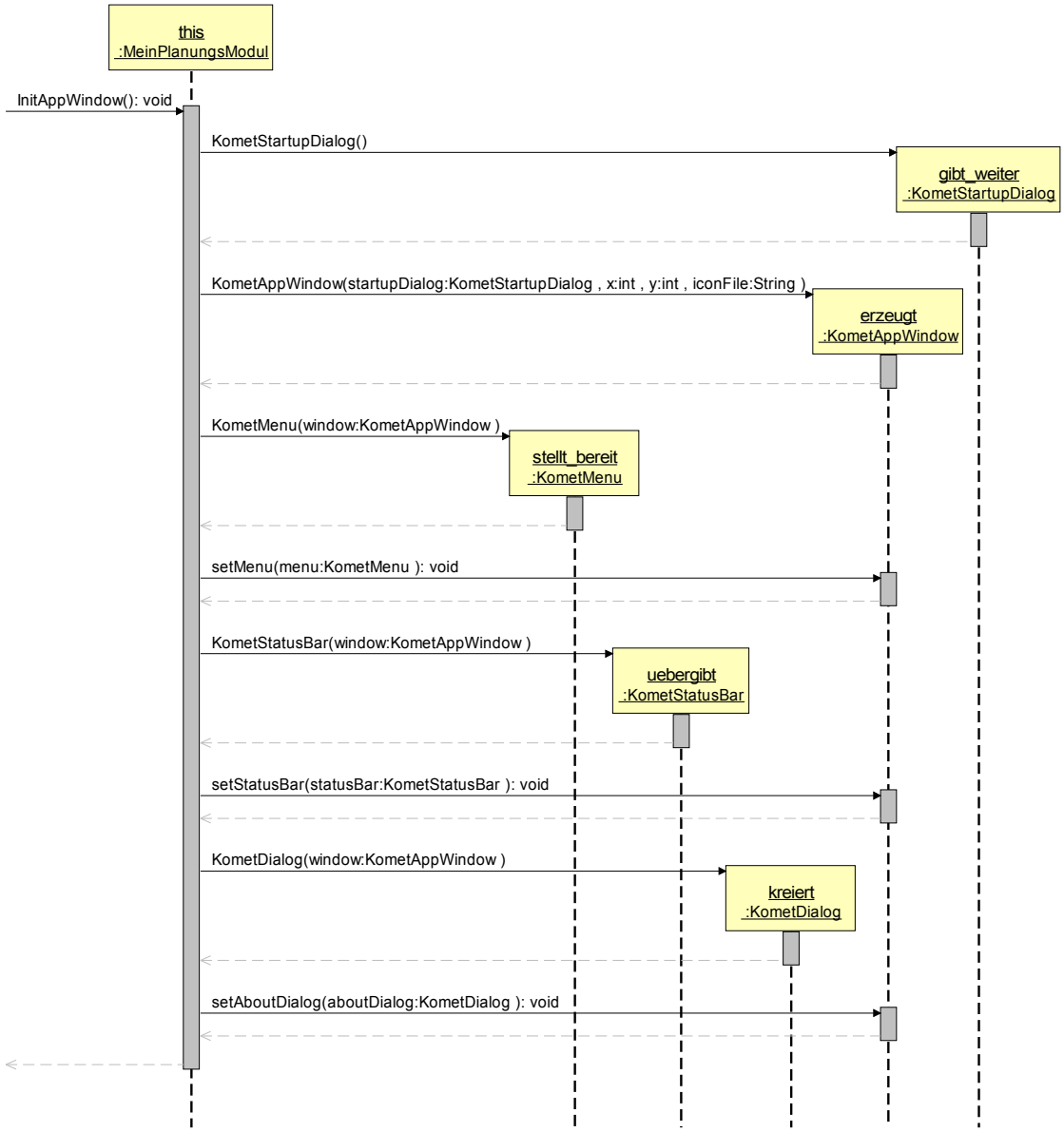


Abbildung 5.11: Sequenzdiagramm von `KometPlan.initAppWindow`

Um dem Fenster-Objekt ein *KometView*-Objekt zuzuweisen, kann kann die Methode `KometAppWindow.setView` verwendet werden. Sie ruft `KometView.doInitialize`

auf, die während der Initialisierung des *KometView*-Objektes Aktualisierungen des Fensterobjektes vornimmt.

Mit Hilfe von visuellen Komponenten kann man erreichen, dass die Methode `KometAppWindow.programContinue` gestartet wird, die `KometView.doContinue` aufruft. Daraufhin kann das *KometView*-Objekt das *MeinPlanungsModul*-Objekt dazu veranlassen, das nächste *KometView*-Objekt zu initialisieren und dem Fenster-Objekt zuzuweisen.

5.2 Programmtechnische Umsetzung (Implementierung)

5.2.1 EUS-Kern

Im Rahmen der vorliegenden Arbeit werden die Prototypen, die während der *Anwendungsarchitektur*-Phase erzeugt werden, zunächst nicht verworfen, sondern dienen als Grundlage für den nächsten Prototypen. Auf Grund der zyklischen Vorgehensweise während dieser Phase wird der EUS-Kern schrittweise verfeinert und verallgemeinert. Dieses Vorgehen entspricht der Praktik des *Refaktorierens* (*Refactoring*) im *Extreme Programming* (*XP*, siehe Abschnitt 3.2.1.6).

Der EUS-Kern ist in fünf Java-Packages strukturiert:

- *de.tum.wzw.forst.komet.gui* enthält die Klassen für die Benutzeroberfläche
- *de.tum.wzw.forst.komet.kernel* enthält die Klassen des EUS-Kerns
- *de.tum.wzw.forst.komet.solver* enthält die Basis-Klassen der Solver
- *de.tum.wzw.forst.komet.plan* enthält die Basisklassen der Planungskomponente
- *de.tum.wzw.forst.komet.util* enthält modulübergreifende Hilfsklassen

Zentraler Bestandteil des EUS-Kerns ist die Klasse *KometKernelImpl* im Package *de.tum.forst.komet.kernel*, welche Dienstanfragen entgegennimmt, zur Verarbeitung weiterleitet und Dienstantworten zurücksendet. Dabei spielt die Interpretation der XML-Anfragen eine große Rolle. In Listing 5.7 ist ein Teil des Quelltextes, der für die XML-Interpretation verantwortlich ist, dargestellt. Listing 5.8 enthält den Quelltext der Klasse *KometKernelImpl*.

Der EUS-Kern ist als Java-Archiv verfügbar. Andere Komponenten können auf Klassen des EUS-Kerns zugreifen, indem sie die Archivdatei *kometcore.jar* einbinden.

```
1  package de.tum.wzw.forst.komet.util;
2  ...
3  public class KometXMLParser extends DefaultHandler {
4
5    private KometTree theRoot, theElement;
6    private KometRequest theType;
7    private boolean error;
8
9    public void startDocument()
10   throws SAXException
11   {
12     theElement = theRoot;
13     theType = null;
14   }
15
16   public void startElement(String namespaceURI,
17                            String sName, // simple name (localName)
18                            String qName, // qualified name
19                            Attributes attrs)
20   throws SAXException
21   {
22     String eName = sName; // element name
23     if ("".equals(eName))
24       eName = qName; // namespaceAware = false
25
26     // Detect the type of request
27     if ("resultTree".equals(eName))
28       theType = new KometRequest() {
29         public String doRequest(KometKernelImpl kernel) throws SQLException {
30           return kernel.getMetaDB().getResults();
31         }
32       };
33 ...
34     // Is there an error?
35     else if ("error".equals(eName))
36       error = true;
37
38     // These keywords require a KometList to be created and inserted into
39     // the object tree. In contrast to the KometItem a KometList can have
40     // children
41     else if ("solver".equals(eName) | "subSolver".equals(eName) |
42            "resultSet".equals(eName) | "solverSet".equals(eName) |
43            "solverPgm".equals(eName) | "table".equals(eName) |
44            "values".equals(eName) | "row".equals(eName) |
45            "editSolver".equals(eName) | "editSubSolver".equals(eName)|
46            "editTable".equals(eName)){
47       KometList aList = new KometList();
48       aList.setValue("xmlText",eName);
49       if (theElement != null)
50         theElement.append(aList);
51       theElement = aList;
52     }
53
54     // These keywords indicate leaf objects and threrefore KometItem Objects
55     // are created
56     else if ("result".equals(eName) | "item".equals(eName) |
57              "column".equals(eName) | "keyColumn".equals(eName)|
58              "valueItem".equals(eName) | "editItem".equals(eName)) {
59       KometItem anItem = new KometItem();
60       anItem.setValue("xmlText",eName);
61       if (theElement != null)
62         theElement.append(anItem);
63       theElement = anItem;
64     }
65
66     // Here the registration objects are created.
67     // We can use one class for all registration types as they are supplied
68     // their particular registration handler.
69     else if ("regSolver".equals(eName)) {
70       KometRegList theRegList = new KometRegList();
71       theRegList.setValue("xmlText",eName);
72       if (theElement != null)
73         theElement.append(theRegList);
74       theElement = theRegList;
75       theRegList.setRegistrationHandler(new KometRegHandler() {
76         public void doRegister(KometRegList reglist) throws SQLException {
77           reglist.getMetaDB().registerSolver(reglist); }
78         });
79     }
80 ...
81   }
82 ...
83 }
```

Listing 5.7: Teile des Quelltextes zur XML-Interpretation

```
1  package de.tum.wzw.forst.komet.kernel;
2
3  import java.io.PrintWriter;
4  import java.io.StringWriter;
5  import java.sql.SQLException;
6  import javax.rmi.PortableRemoteObject;
7  import de.tum.wzw.forst.komet.util.KometXMLHelper;
8
9
10 public class KometKernelImpl extends PortableRemoteObject
11                              implements KometKernel {
12
13   private KometMetaDB theMetaDB;
14   private KometXMLHelper xmlHelper;
15   private KometRequest requestHandler;
16   private String theRequest;
17
18   public KometMetaDB getMetaDB() { return theMetaDB; }
19
20   public String getRequest() { return theRequest; }
21
22   public void addRequestHandler(KometRequest handler) {
23     requestHandler = handler;
24   }
25
26   public KometKernelImpl() throws java.rmi.RemoteException {
27     super();
28     xmlHelper = new KometXMLHelper();
29     theMetaDB = new KometMetaDB(xmlHelper);
30     requestHandler = null;
31   }
32
33   public String request(String aRequest) throws java.rmi.RemoteException {
34
35     String theResponse;
36
37     // store the Request so that it is accessible by the request handler...
38     theRequest = aRequest;
39     requestHandler = xmlHelper.typeOfRequest(aRequest);
40     try {
41       if (requestHandler == null)
42         throw new SQLException("malformed Request");
43       theResponse = requestHandler.doRequest(this);
44       return theResponse;
45     }
46     catch (SQLException e) {
47       StringWriter sw = new StringWriter();
48       PrintWriter pw = new PrintWriter(sw);
49       e.printStackTrace(pw);
50       theResponse = xmlHelper.createMessage("error","3",
51           "Database Error: " + e.getMessage(), sw.toString(),true);
52       return theResponse;
53     }
54   }
55 }
```

Listing 5.8: Quelltext der Klasse KometKernelImpl

5.2.2 Demonstrationsanwendung

Im Rahmen der vorliegenden Arbeit wird auf Basis der in HEMM [2006] beschriebenen Komponenten ein **r**äumliches **E**ntscheidungs**U**nterstützungs**S**ystem (REUS) zur Holzernte implementiert um die korrekte Funktion der *KOMET-Architektur* zu demonstrieren. Die Anwendungen *SILVA 2.2*, *Holzernte 7.0* und *AutoMod 11.0* (vgl. Abschnitt 4.4.1) werden mit Hilfe von Wrappern (siehe Abschnitt 3.1.6.1) in das REUS integriert. Zusätzlich wird ein Solver zur Einbindung der Bestandes-Geodaten auf der Basis von *ArcGIS Engine for Java 9.2* neu implementiert. Mit Hilfe einer

Planungsanwendung werden die notwendigen Einstellungen vorgenommen, die Solver aktiviert und die Ergebnisse angezeigt, die vom Benutzer verglichen werden können.

5.2.2.1 Planungsanwendung

Wie in Abschnitt 5.1.2.2 beschrieben besteht die Planungskomponente im Wesentlichen aus Implementationen der Interfaces *KometView* und *KometPlan*. In der Planungskomponente des REUS zur Holzernte sind dies die Klassen *JEPlan*, *JEPSolverView*, *JEPOptionView*, *JEPEditorView*, *JEPLauncherView*, *JEPResultView* und *JEPTargetView*, welche im Package *de.tum.wzw.forst.komet.jep* definiert sind. Durch Klick auf den *Zurück*- bzw. *Weiter*-Knopf kann der Benutzer zwischen den Anzeigen wechseln. Die selben Funktionen befinden sich auch in der Menüleiste. Die Ereignismethoden der *KometView*-Objekte (`KometView.doBack` bzw. `KometView.doContinue`) werden vom Fensterobjekt aktiviert und rufen die entsprechende Methode in *JEPlan* zur Anzeige des nächsten oder des vorherigen *KometView*-Objektes auf. Dazu werden die Methoden `JEPlan.showLauncherView`, `JEPlan.showSolverView`, `JEPlan.showOptionView`, `JEPlan.showEditorView`, `JEPlan.showResultView` und `JEPlan.showTargetView` zur Verfügung gestellt.

JEPSolverView stellt die am REUS registrierten Solver in einer Baumansicht dar. Dabei sind die Solver in verschiedenen Ebenen organisiert, welche sich aus den Abhängigkeiten ergeben. Solver einer Ebene sind voneinander unabhängig und besitzen Abhängigkeiten zu Solvern niedrigerer Ebenen. Der Benutzer kann die Solver einzeln aktivieren und deaktivieren. Der Aktivierungsstatus wird in aus den Metadaten gelesen und bei Veränderung wieder gespeichert.

Mit Hilfe von *JEPOptionView* wird die Möglichkeit implementiert, verschiedene Varianten mit unterschiedlich eingestellten Optionen für mehrere Bestände berechnen zu können. Im REUS vorhandene Kombinationen aus Bestand und Variante werden angezeigt und die gewünschte kann vom Benutzer aktiviert werden.

JEPEditorView dient zur Einstellung der Parameter einer vorher ausgewählten Variante. Dazu werden mit Hilfe von Metadaten jene Datenfelder aus der Solver-Datenbank selektiert, in denen die Parameter gespeichert sind. Bei der Registrierung eines Solvers kann mit Hilfe des XML-Parameters *edit = true* bzw. *edit = false* eingestellt werden, ob ein Datum in *JEPEditorView* angezeigt werden soll oder nicht.

Die Solver werden gestartet, wenn während der Anzeige von *JEPLauncherView* auf die Schaltfläche *Solver jetzt starten* geklickt wird. Der Start der Solver erfolgt in einem eigenen Prozess, der *KometLauncherView* über den Fortschritt der Solver-Aktivierung informiert. Dies wird dem Benutzer in Form eines Fortschrittsbalkens angezeigt. Die Solver Geo-Informationssystem, Wuchsmodell, Sortierung und Holzernesimulation werden der Reihe nach aktiviert. Das Sequenzdiagramm in Abbildung 5.12 zeigt deren Aktivierungsreihenfolge.

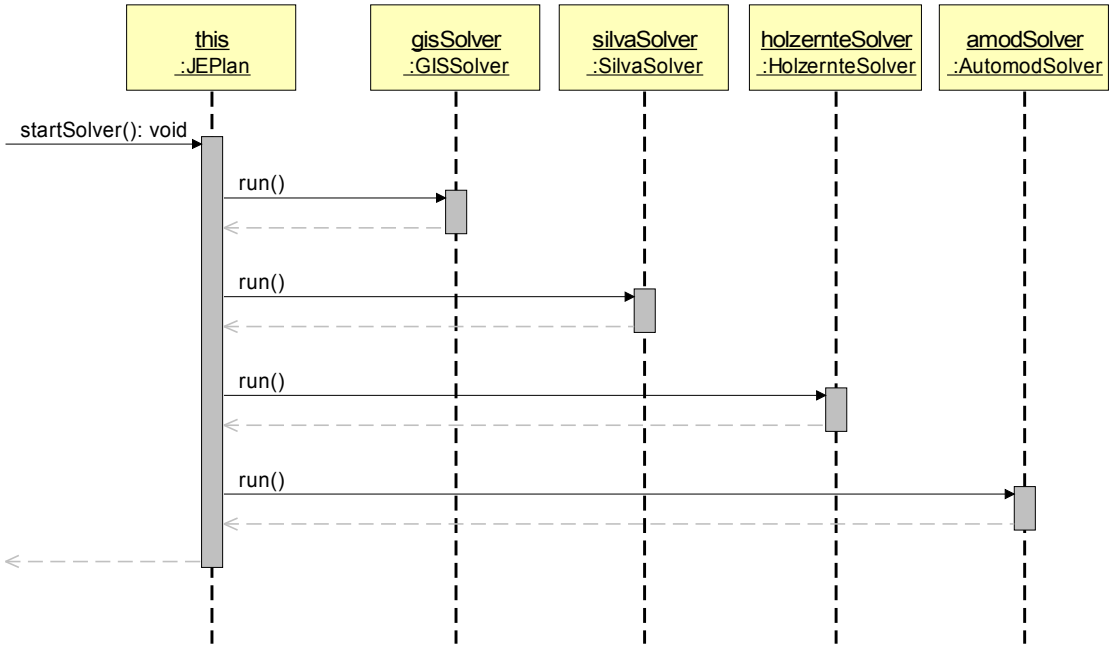


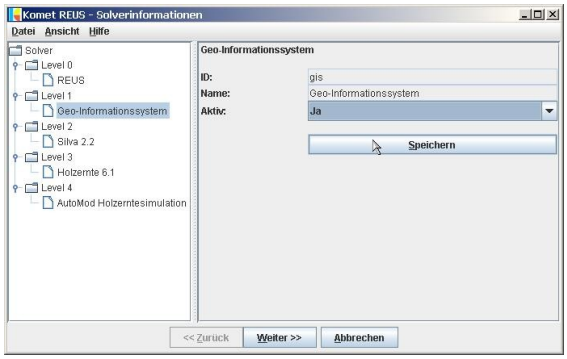
Abbildung 5.12: Aktivierung der Solveranwendungen

Nach Ausführung der Solver werden die Ergebnisse von *JEPResultView* in einer Baumansicht präsentiert. Ein Klick auf ein Ergebnis zeigt weitere Details und ein Auswahlfeld mit dessen Hilfe der Anwender einstellen kann, ob das Ergebnis in *JEPTargetView* angezeigt werden soll.

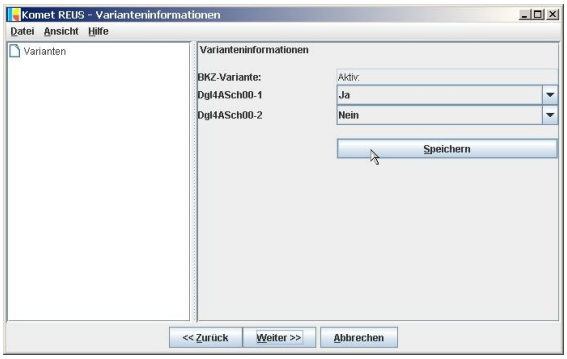
In *JEPTargetView* werden die vom Benutzer ausgewählten Ergebnisse für jede Variante angezeigt. Im Rahmen der vorliegenden Arbeit ist nur die Anzeige dieser Ergebnisse implementiert. Es können auch Optimierungsfunktionen eingesetzt werden, die auf Grund dieser Ergebnisse und der gewählten Ziele die beste Variante identifizieren.

Visuelle Elemente, die von den *KometView*-Objekten angezeigt werden, sind in den Klassen *JEPStartupDialog*, *JEPMenuBar*, *JEPStatusBar*, *JEPAboutDialog* und *JEPSolverRunDialog* im Package *de.tum.wzw.forst.komet.jep.gui* definiert.

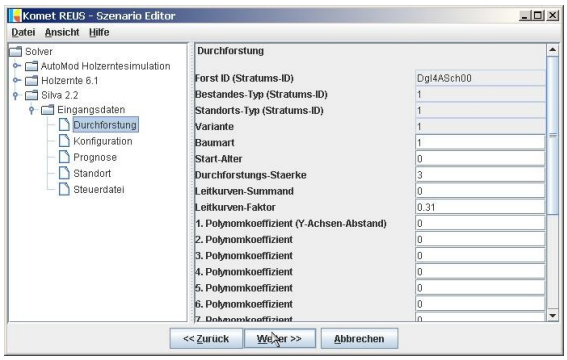
Die Bildschirmanzeigen der *KometView*-Objekte sind in Abbildung 5.13 a) bis f) dargestellt.



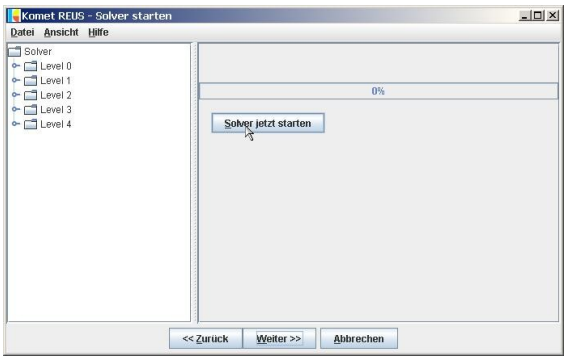
a) JEPSolverView



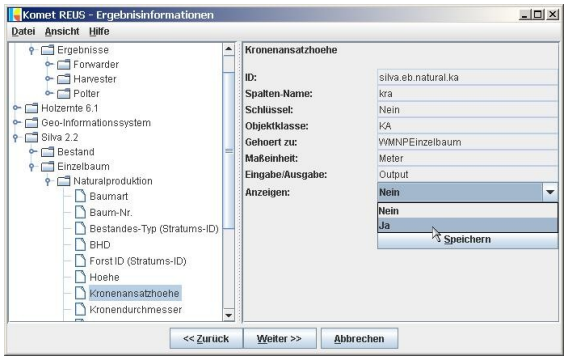
b) JEPOptionView



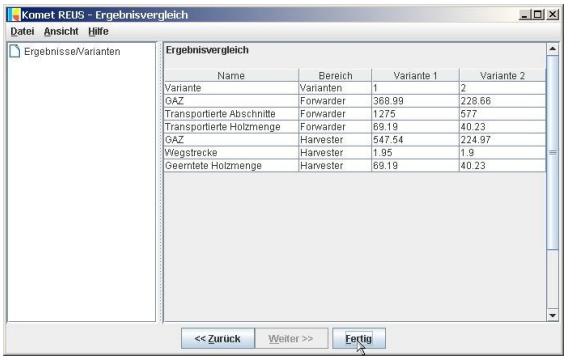
c) JEPEditorview



d) JEPLauncherView



e) JEPResultView



f) JEPTargetView

Abbildung 5.13 a) – f): Bildschirmanzeigen der Planungsanwendung

5.2.2.2 Solver REUS

Das REUS ist im Rahmen de vorliegenden Arbeit so ausgelegt, das für verschiedene Bestände eines Forstbetriebs verschiedene Varianten der Waldbehandlung miteinander verglichen werden können. Hierzu müssen im REUS Daten zum aktuellen Bestand und zur aktuell angewendeten Variante bekannt sein und vorliegen. Dies geschieht mit Hilfe des Solvers REUS. Dieser Solver besitzt eine Tabelle in der Solver-Datenbank, in der

der Bestandeskürzel, Nummern der Behandlungsvarianten sowie ein boolescher Wert (Ja/Nein-Wert) zur Erkennung der aktiven Kombination aus Bestand und Varianten-Nummer gespeichert sind. Alle anderen Solver greifen auf diese Tabelle zu und lesen die Werte aus, um auf die entsprechenden Daten in ihrem eigenen Datenbestand zuzugreifen.

Im Rahmen der Programmierung eines Solvers kann auf die Methode `KometSolverHelper.getOptionInfo` zugegriffen werden, welche die Datenbankabfrage erledigt und die beiden Werte für aktuelles Bestandeskürzel sowie aktuelle Variante zurückgibt.

5.2.2.3 Solver Geo-Informationssystem

Die Basisdaten von Beständen und Einzelbäumen werden in dem Geografischen Informationssystem *ArcGIS* abgelegt (vgl. Abschnitt 4.4.2) Aufgabe des Solvers *Geo-Informationssystem* ist es, diese Daten im REUS bekannt zu machen und in die Solver-Datenbank zu importieren. Der Solver liest mit Hilfe von *ArcGIS Engine for Java 9.2* die relevanten Informationen, wie z. B. Bestandesausdehnung und Einzelbaumdaten (BHD, Höhe, etc.) aus den entsprechenden Geo-Objekt-Dateien, die im *Shape-File-Format* vorliegen, und speichert deren Inhalt in der Solver-Datenbank. Auf diese Weise werden die Daten anderen Solvern zugänglich gemacht. Listing 5.9 Enthält einen Auszug des Quelltextes des Solvers *Geo-Informationssystem*.

```
 1 package de.tum.wzw.forst.gis;
 2 ...
 3 public class GISSolver implements KometSolver {
 4
 5   private String theName;
 6   private String theBKZ;
 7 ...
 8   public String run(String theRequest) throws RemoteException {
 9 ...
10     storeOutputData();
11 ...
12     return "GIS erfolgreich abgearbeitet";
13   }
14
15   public void storeOutputData() throws SQLException, IOException {
16
17     IWorkspaceFactory wsf;
18     IFeatureWorkspace fws;
19     IFeatureClass rgFC = null;
20     IFeatureCursor rgFeatures;
21     IFeature rgFeature;
22
23     int iRGName,iRGstartX,iRGstartY,iRGendX,iRGendY,iRGLaenge;
24
25     String rgName;
26     double rgStartX,rgStartY,rgEndX,rgEndY,rgLaenge;
27
28     String bkz, baumArt;
29
30     String             cmd;
31     KometSolverDB      db;
32     Connection         conn;
33     PreparedStatement  pstmt;
34     Statement          stmt;
35 ...
36     wsf = new ShapefileWorkspaceFactory();
37     fws = (IFeatureWorkspace) wsf.openFromFile(gisWorkSpace, 0);
38     rgFC = fws.openFeatureClass(theBKZ+"_rg");
39     baumFC = fws.openFeatureClass(theBKZ+"_baum");
40
41     db = KometDBFactory.createDB();
42     conn = db.connect();
43 ...
44     rgFeatures = new FeatureCursor(rgFC.search(null, false));
45     rgFeature = rgFeatures.nextFeature();
46
47     if (debugOn())
48       System.out.println("Writing skid-road table");
49     while (rgFeature != null) {
50
51       rgName = (String) rgFeature.getValue(iRGName);
52       rgStartX = ((Double)
53                   rgFeature.getValue(iRGstartX)).doubleValue();
54       rgStartY = ((Double)
55                   rgFeature.getValue(iRGstartY)).doubleValue();
56       rgEndX = ((Double) rgFeature.getValue(iRGendX)).doubleValue();
57       rgEndY = ((Double) rgFeature.getValue(iRGendY)).doubleValue();
58       rgLaenge = ((Double)
59                   rgFeature.getValue(iRGLaenge)).doubleValue();
60
61       cmd = "Insert INTO gis_ruckweg VALUES ( ?,?,?,?,?,?,?,? )";
62       pstmt = conn.prepareStatement(cmd);
63       pstmt.setString(1,bkz);
64       pstmt.setString(2,rgName);
65       pstmt.setDouble(3,rgStartX);
66       pstmt.setDouble(4,rgStartY);
67       pstmt.setDouble(5,rgEndX);
68       pstmt.setDouble(6,rgEndY);
69       pstmt.setDouble(7,rgLaenge);
70       pstmt.setInt(8,theOption);
71       pstmt.execute();
72       pstmt.close();
73
74       rgFeature = rgFeatures.nextFeature();
75
76     }
77 ...
78   }
79 ...
80 }
```

Listing 5.9: *Teile des Quelltextes des Solvers* Geo-Informationssystem

5.2.2.4 Solver Wuchsmodell

Wie in Abschnitt 4.4.1 dargestellt, wird als Wuchsmodell in der Demonstrationsanwendung *SILVA 2.2* verwendet. Alle Eingabedaten sind in der Solver-Datenbank als Tabellen gespeichert und können vom Anwender verändert werden. Die Baumdaten können als Einzelbaumdaten oder Bestandesmittelwerte vorliegen. Der Solver liest die Tabellen aus, erzeugt die Text-Dateien, die *SILVA 2.2* benötigt, und startet *SILVA 2.2* im so genannten Batch-Modus, das heißt der Anwender muss während der Wuchs-Simulation nicht eingreifen. Es wird nur eine Perionde simuliert, da lediglich die Bäume von Interesse sind, die vom Durchforstungsmodul von *SILVA 2.2* als ausscheidend markiert werden. Die während des Simulationslaufes erzeugten Dateien, werden in die Solver-Datenbank zurückgespielt und stehen anderen Solvern zur Verfügung.

5.2.2.5 Solver Sortierung

Die Holzsortierung wird mit Hilfe der Anwendung *Holzernte 7.0* berechnet (vgl. Abschnitt 4.4.1). *Holzernte 7.0* verfügt im Gegensatz zu *SILVA 2.2* nicht über einen Batch-Modus und muss deshalb vom Anwender bedient werden. Während der REUS-Anwendung können die Baumdaten mit voreingestellten vorhandenen Hiebs-Informationen verknüpft und anschließend eine Sortenberechnung durchgeführt werden. Eingangsdaten sind die Baumdaten, die *SILVA 2.2* als ausscheidenden Bestand ausgibt. Diese Liste wird in die MS-Access-Datenbank von *Holzernte 7.0* übertragen. Die Sorteninformationen, welche *Holzernte 7.0* berechnet, werden anschließend in die Solver-Datenbank kopiert.

5.2.2.6 Solver Holzerntesimulation

Der Holzerntevorgang wird mit Hilfe der Software *AutoMod 11.0* simuliert (vgl. Abschnitt 4.4.1). Neben den Sorten- und Einzelbaum-Informationen, welche automatisch an *AutoMod 11.0* übertragen werden, kann der Anwender Einstellungen zu den Holzernteverfahren vornehmen. Diese Daten werden zusammen mit den Sorten- und Einzelbaum-Informationenen in Text-Dateien gespeichert, welche *AutoMod 11.0* einliest und auswertet. Zusätzlich müssen an *AutoMod 11.0* Geometriedaten übergeben werden. Dazu gehören das Feinerschließungsnetz sowie die Schnittpunkte der Lotlinien jedes Baums mit der nächstgelegenen Rückegasse bzw. Forststraße. Diese

Informationen werden mit Hilfe von *ArcGIS Engine 9.2* aus den Geo-Objekt-Daten berechnet und in den entsprechenden Eingabedateien abgelegt. *AutoMod 11.0* verfügt über einen interaktiven Modus mit Grafikausgabe und über einen Batch-Modus ohne grafische Ausgabe. Der Anwender kann wählen, welcher Modus verwendet werden soll. Die Ergebnisse der Holzerntesimulation werden in Dateien gespeichert, die der Wrapper in die Solver-Datenbank kopiert, damit sie zur Auswertung zur Verfügung stehen.

5.2.3 Durchführung von Simulationen mit Hilfe der Demonstrationsanwendung

Mit Hilfe der Demonstrationsanwendung wird eine Durchforstungsmaßnahme in der Abteilung 4A des Forstamtes Schmallenberg in Nordrhein-Westfahlen simuliert. Dabei werden zwei Simulationsläufe mit unterschiedlicher Eingriffsstärke durchgeführt und deren Ergebnisse gegenübergestellt. In den folgenden Abschnitten wird das schrittweise Vorgehen im Detail beschrieben.

5.2.3.1 Definition von Behandlungsvarianten

Nach der Integration der Geodaten des zu betrachtenden Bestandes in das Geografische Informationssystem (GIS) muss in der zentralen Datenbank in einigen Tabellen manuell für jede Kombination aus Bestand und Behandlungsvariante je eine Zeile mit Voreinstellungen hinzugefügt werden. Auszüge des Inhalts einer dieser Tabellen sind in Abbildung 5.14 dargestellt.

Abbildung 5.14: Inhalt der Tabelle SLV_INPUT_DFO *in Auszügen*

Sämtliche der Demonstrationsanwendung bekannten Kombinationen aus Bestand und Behandlungsvariante werden dem Anwender angezeigt und er kann eine davon aktivieren. Alle vom Benutzer definierten Einstellungen und Berechnungen beziehen sich auf diese aktive Kombination. Abbildung 5.15 zeigt den Varianten-Auswahl-Dialog zur Aktivierung einer Kombination aus Bestand und Behandlungsvariante.

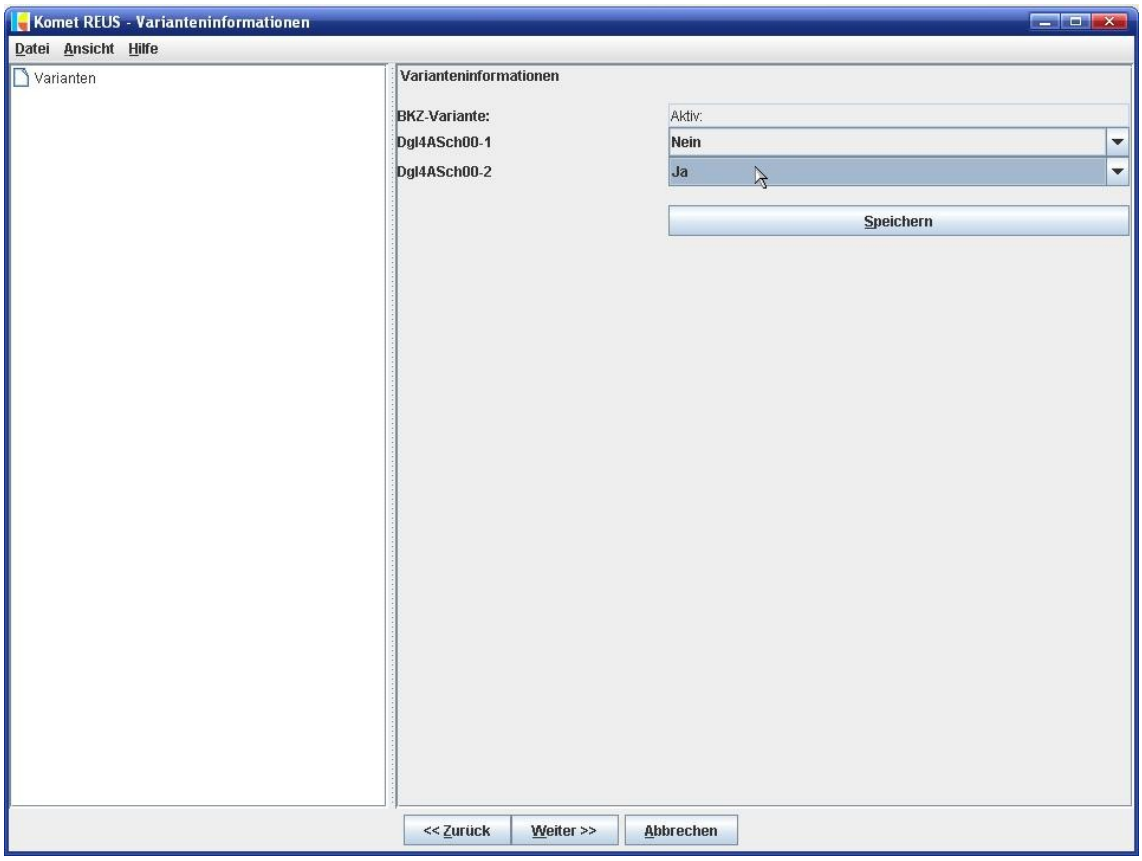


Abbildung 5.15: Dialog zur Aktivierung einer der bereitgestellten Varianten

Werden mehrere Varianten nacheinander aktiviert und anschließend Simulationsrechnungen ausgeführt, so besteht die Möglichkeit deren Ergebnisse später visuell zu vergleichen. Die in den Abschnitten 5.2.3.2 und 5.2.3.3 beschriebenen Schritte müssen deshalb nach vorangegangener Aktivierung der jeweiligen Behandlungsvariante je nach Anzahl der zu vergleichenden Varianten mehrmals durchgeführt werden. Der Benutzer kann nach Ausführung der Solver wieder zum Varianten-Auswahl-Dialog zurücknavigieren, eine andere Variante aktivieren, die Schritte wiederholen und dabei andere Eingangsparameter festlegen.

5.2.3.2 Definition der Eingangsparameter

Mit Hilfe eines auf der *KOMET-Architektur* basierenden Szenario-Editors werden im Rahmen der vorliegenden Arbeit für die Simulation von zwei Durchforstungsvarianten unterschiedliche Eingriffs-Stärken eingestellt.

In beiden Varianten soll eine Hochdurchforstung durchgeführt werden. Der entsprechende Einstellungsbildschirm ist in Abbildung 5.16 dargestellt. Für jede der drei

Altersphasen eines Bestandes wird daher für die Durchforstungsart ein Wert von 5 konfiguriert, mit dessen Hilfe in *Silva 2.2* eine Hochdurchforstung kodiert wird (BIBER ET AL. [2000]).

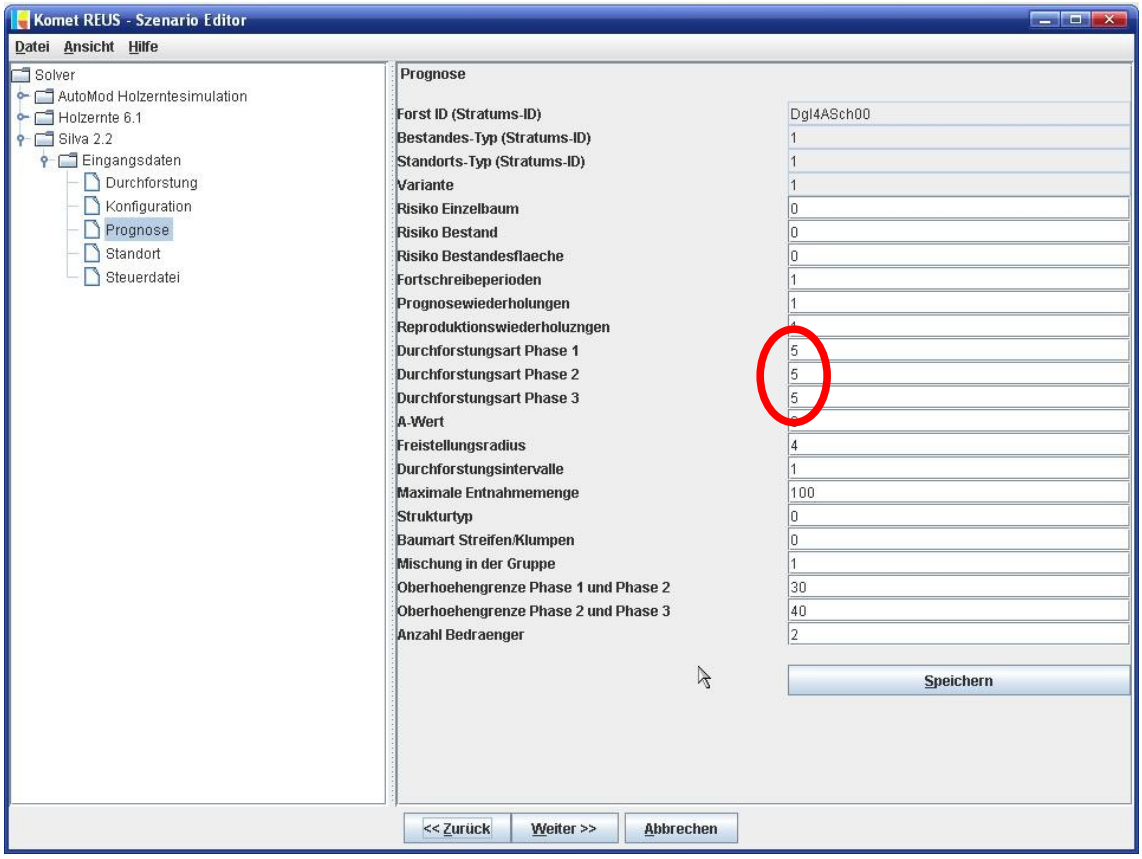


Abbildung 5.16: Einstellung der Durchforstungsart

Die grobe Eingriffsstärke wird über einen Durchforstungs-Stärken-Kode (»*Durchforstungs-Staerke*«) geregelt. Der Wert 3 bedeutet, dass eine stammzahlorientierte, starke Durchforstung durchgeführt werden soll (BIBER ET AL. [2000]). Mit Hilfe eines Faktors, der die Leitkurve skaliert, welche die Stammzahl eines Bestandes in Abhängigkeit seines Alters beschreibt, kann die Eingriffs-Stärke feinreguliert werden. In Variante 1 beträgt der Faktor 0,31 (stärkerer Eingriff, weniger Bäume verbleiben im Bestand), in Variante 2 0,4 (schwächerer Eingriff, mehr Bäume verbleiben im Bestand). Abbildung 5.17 zeigt die Konfiguration der Eingriffsstärke für Variante 1, der gleiche Vorgang für Variante 2 ist in Abbildung 5.18 dargestellt.

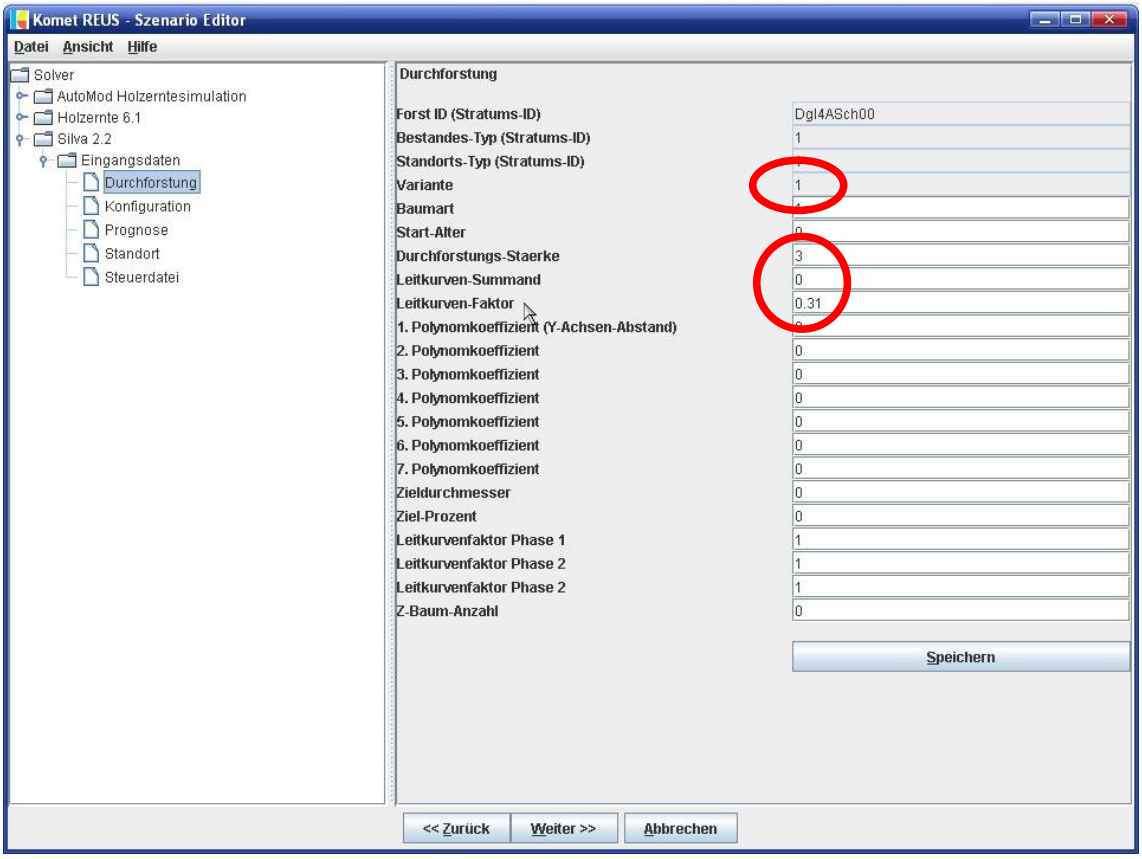


Abbildung 5.17: Einstellung der Durchforstungsstärke für Behandlungsvariante 1

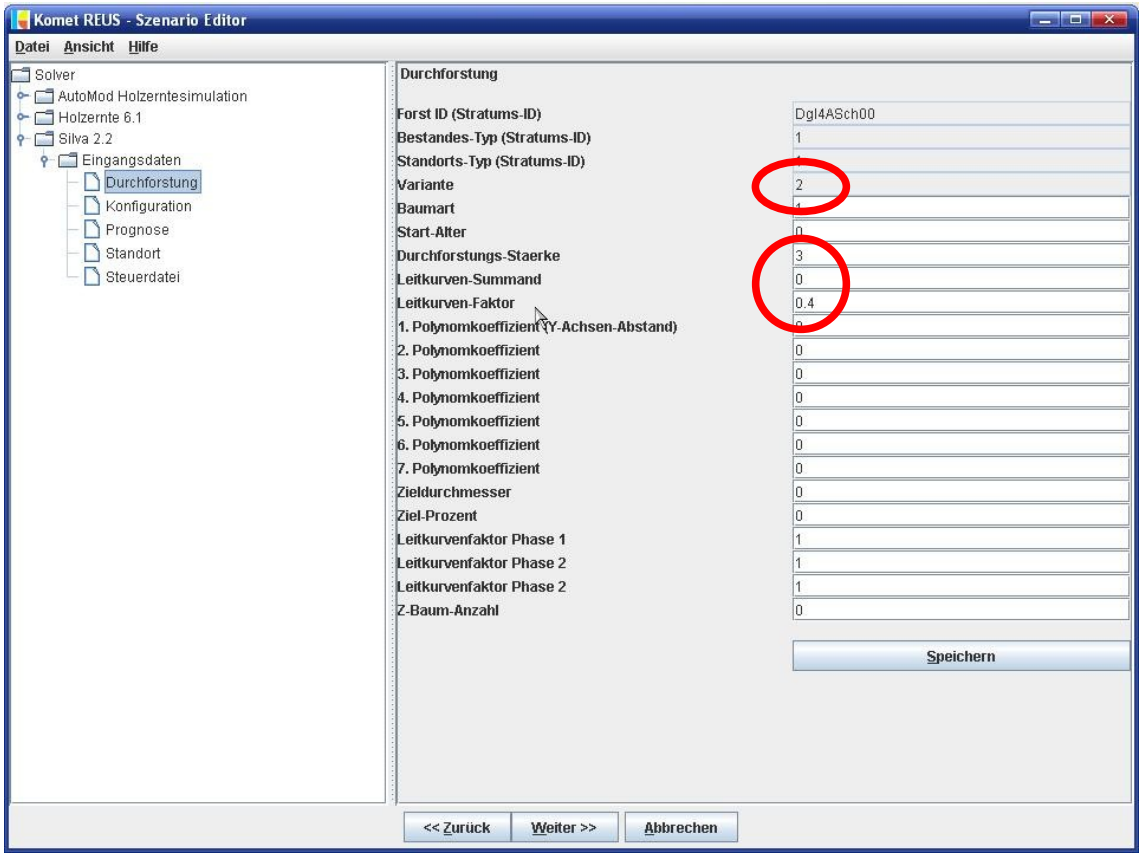


Abbildung 5.18: Einstellung der Durchforstungsstärke für Behandlungsvariante 2

Da im Programm *Holzernte 7.0* nicht vorgesehen ist, dass Berechnungen ohne Interaktion mit dem Benutzer durchgeführt werden können, müssen die Einstellungen vor dem Start der Demonstrationsanwendung direkt in *Holzernte 7.0* vorgenommen werden. Für die Berechnung der Sortimente wird dabei ein Hieb angelegt, der eine Aushaltungsstrategie enthält, welche in beiden Varianten angewendet wird. Durch die Zuordnung der Baumliste, welche die Einzelbauminformationen der jeweiligen Variante enthält, werden die geernteten Bäume entsprechend der im Hieb definierten Aushaltungsstrategie sortiert.

Vor der Simulation einer Holzerntemaßnahme können im Simulationsmodell von *AutoMod 11.0* umfangreiche Parameter eingestellt werden. Im allgemeinen Teil kann das Arbeitsverfahren definiert werden, wobei auch Kombinationen verschiedener Holzerntesysteme möglich sind. Für beide Varianten wird ein Verfahren gewählt, das aus der Kombination von Harvester und Forwarder besteht. Wie in Abbildung 5.19 dargestellt, kann der Einsatz eines jeden Einzelsystems im Szenario-Editor mit Hilfe eines Zahlenkodes definiert werden. Der Wert *Eins* bedeutet, dass das System verwendet wird und *Null*, dass das Erntesystem nicht verwendet wird.

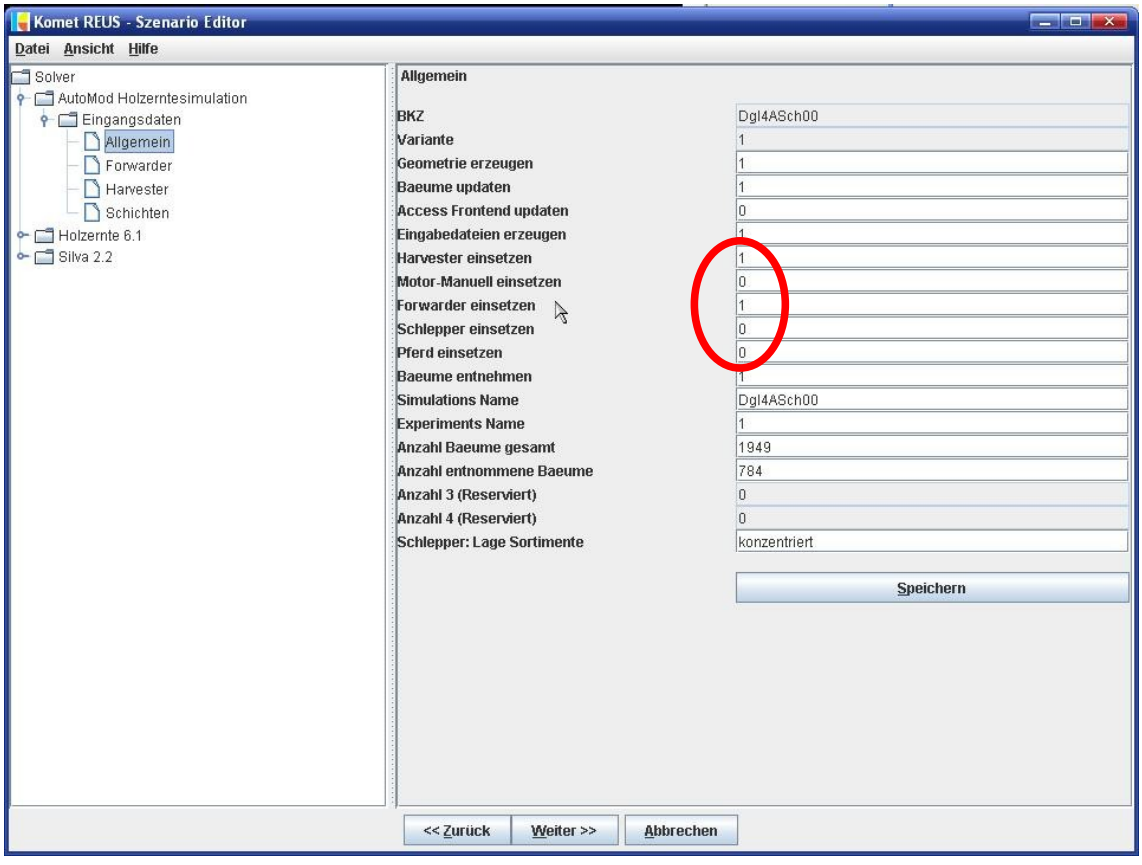


Abbildung 5.19: Allgemeine Einstellungen von AutoMod 11.0

5.2.3.3 Ausführung der Solver

Nach der Festlegung der Voreinstellungen können die Solver der Reihe nach aktiviert werden. Dies geschieht automatisch durch die Demonstrationsanwendung, welche auch die richtige Reihenfolge mit Hilfe der Metadaten ermittelt. Ein Mausklick auf den Knopf *»Solver jetzt starten«* setzt die Aktivierung in Gang. Über einen Fortschrittsbalken wird der Anwender über die einzelnen Ausführungsschritte informiert. Der entsprechende Dialog ist in Abbildung 5.20 dargestellt.

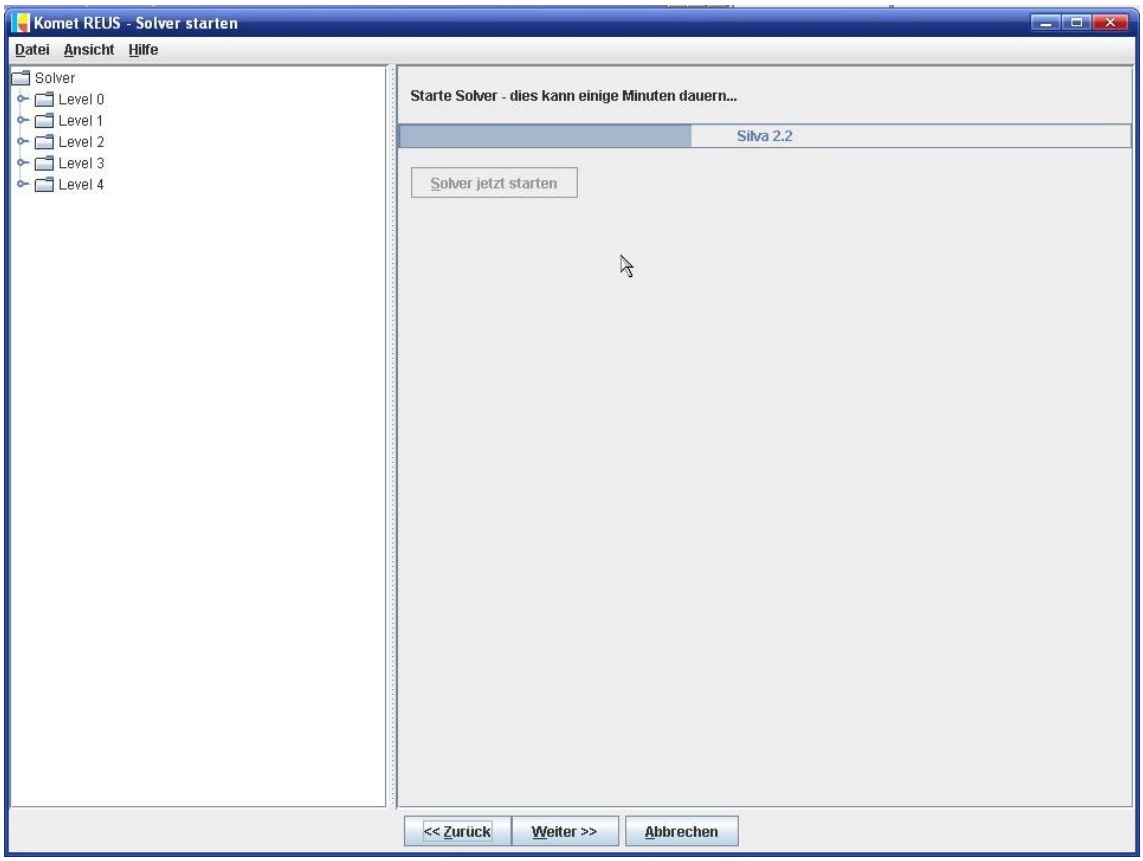


Abbildung 5.20: Dialog zum Start der Solver

Nach der Aufbereitung der Geo-Daten durch den Solver *GIS* und deren Einspielung in die zentrale Datenbank wird mit Hilfe von *Silva 2.2* eine Periode simuliert. Dabei wird eine Durchforstung mit den im Szenario-Editor eingestellten Parmetern vorgenommen. Die Ausgabedaten von *Silva 2.2* werden anschließend in die zentrale Datenbank kopiert. Ein Bildschirmausschnitt mit *Silva 2.2* während der Simulation ist in Abbildung 5.21 dargestellt.

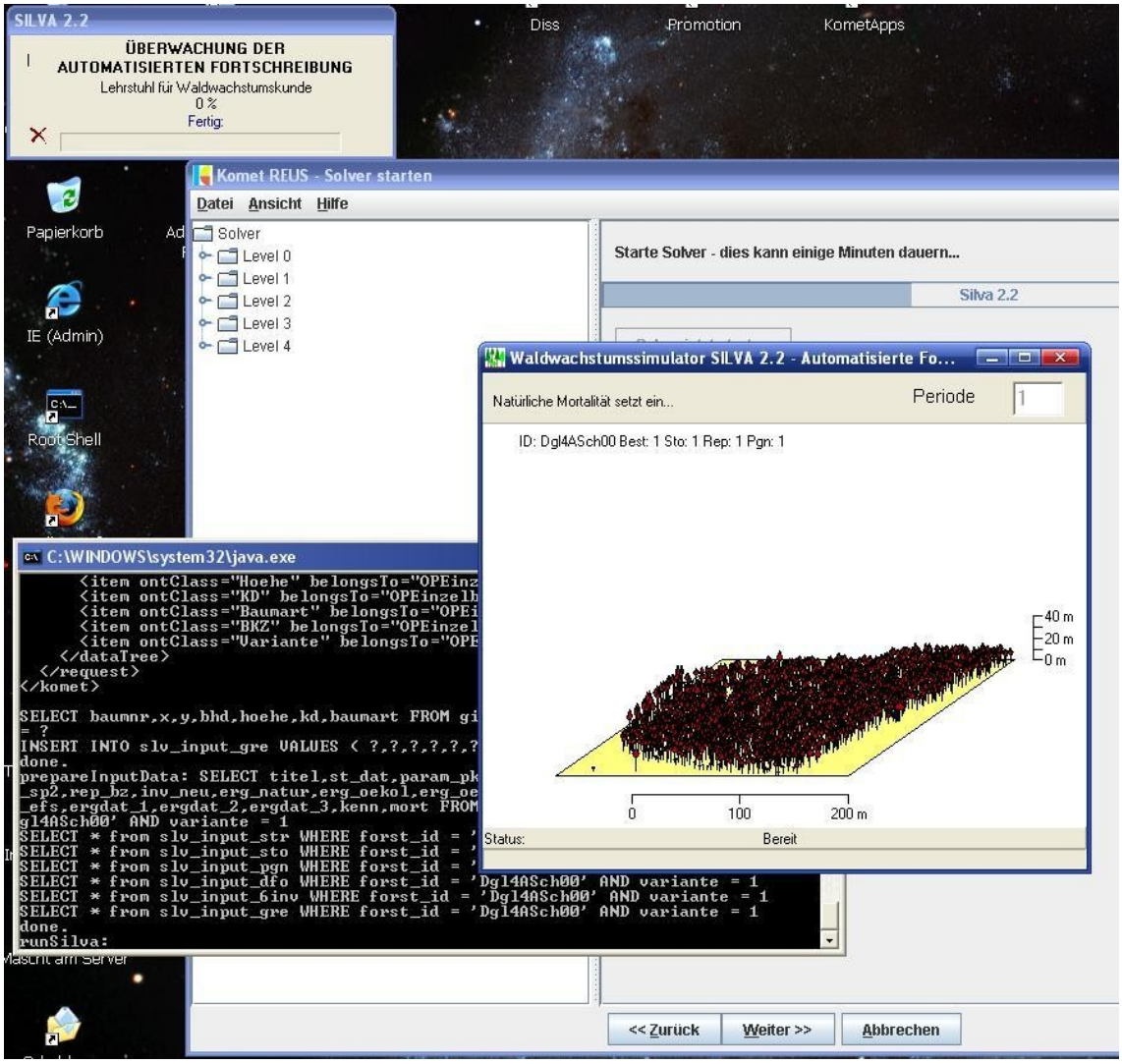


Abbildung 5.21: Silva 2.2 während der Wuchssimulation

Wie bereits erwähnt, können in *Holzernte 7.0* Berechnungen ohne Interaktion mit dem Benutzer nicht durchgeführt werden. Im Rahmen der Aktivierung des Solvers Holzernte wird *Holzernte 7.0* automatisch gestartet. Der Anwender muss die Einzelbaumdaten des ausscheidenden Bestandes mit dem entsprechenden Hieb manuell mit Hilfe eines Dialogs innerhalb von *Holzernte 7.0* verbinden. Dieser Dialog ist in Abbildung 5.22 dargestellt. Zuvor werden die Einzelbaumdaten automatisch in die Access-Datenbank von *Holzernte 7.0* kopiert. Nachdem der Anwender *Holzernte 7.0* beendet hat, werden die Sortimentsdaten automatisch in die zentrale Datenbank eingespielt.

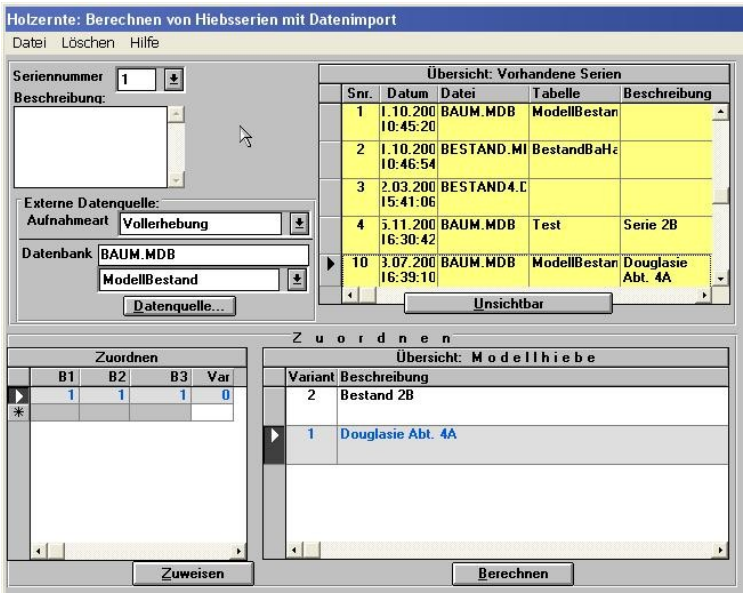


Abbildung 5.22: Dialog zur Zuweisung von Hiebs- und Einzelbaumdaten

Nach der Berechnung der Sortimente durch *Holzernte 7.0* sind alle Informationen, welche von *AutoMod 11.0* zur Ausführung des Holzerntemodells benötigt werden, vorhanden. Die Demonstrationsanwendung kann das Holzerntemodell automatisch starten. Das Programmfenster von *AutoMod 11.0* während der Simulation ist in Abbildung 5.23 dargestellt.

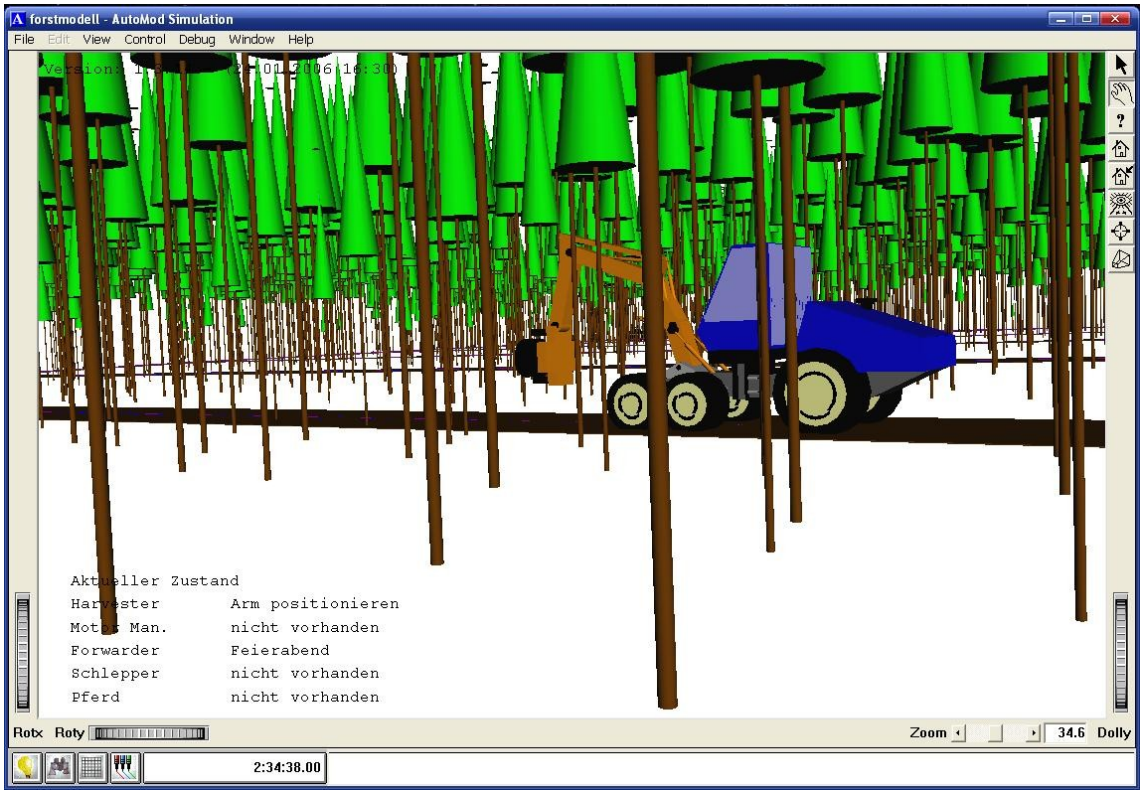


Abbildung 5.23: AutoMod 11.0 während der Simulation des Forstmodells

5.2.3.4 Visuelle Darstellung der Ergebnisse

Nach der zweimaligen Durchführung der Holzerntesimulation für beide Eingriffsstärken können die Ergebnisse mit Hilfe der grafischen Oberfläche visuell miteinander verglichen werden. Im Ergebnis-Auswahl-Dialog hat der Anwender die Möglichkeit, diejenigen Ergebnisse auszuwählen, die später angezeigt werden sollen. Abbildung 5.24 zeigt den entsprechenden Dialog.

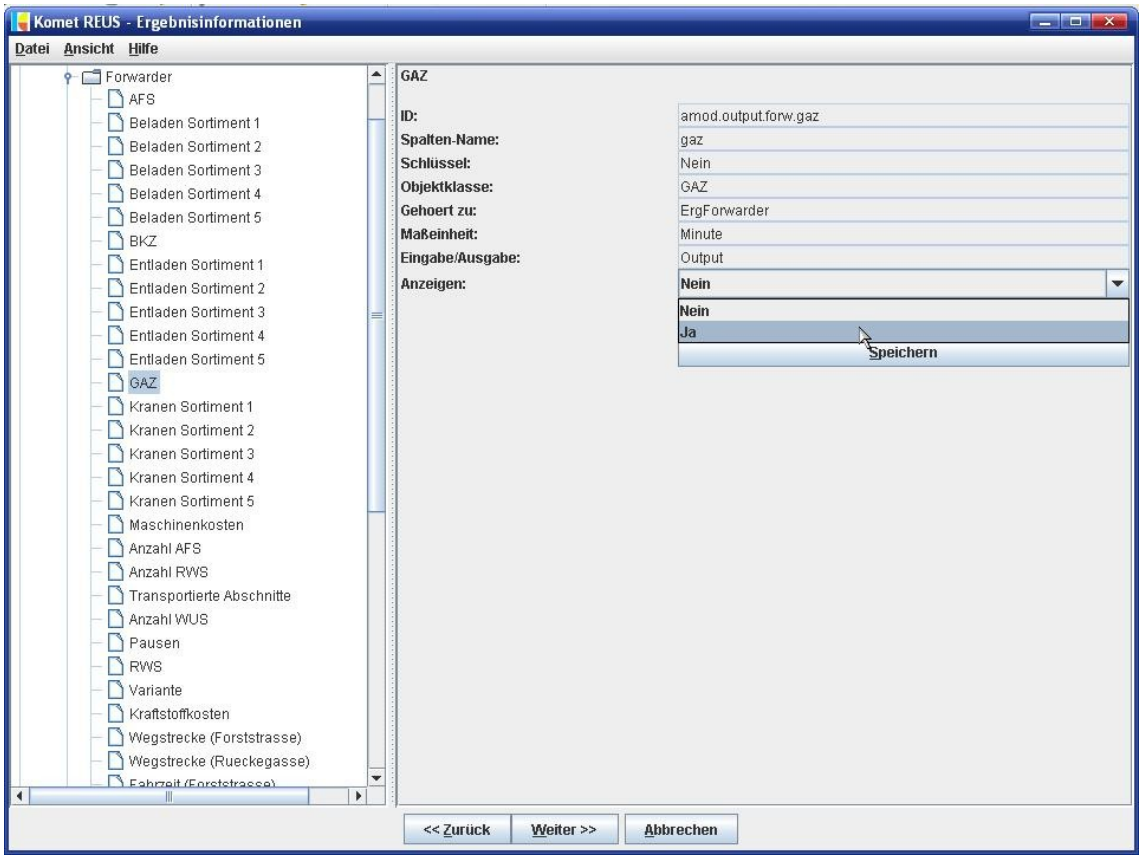
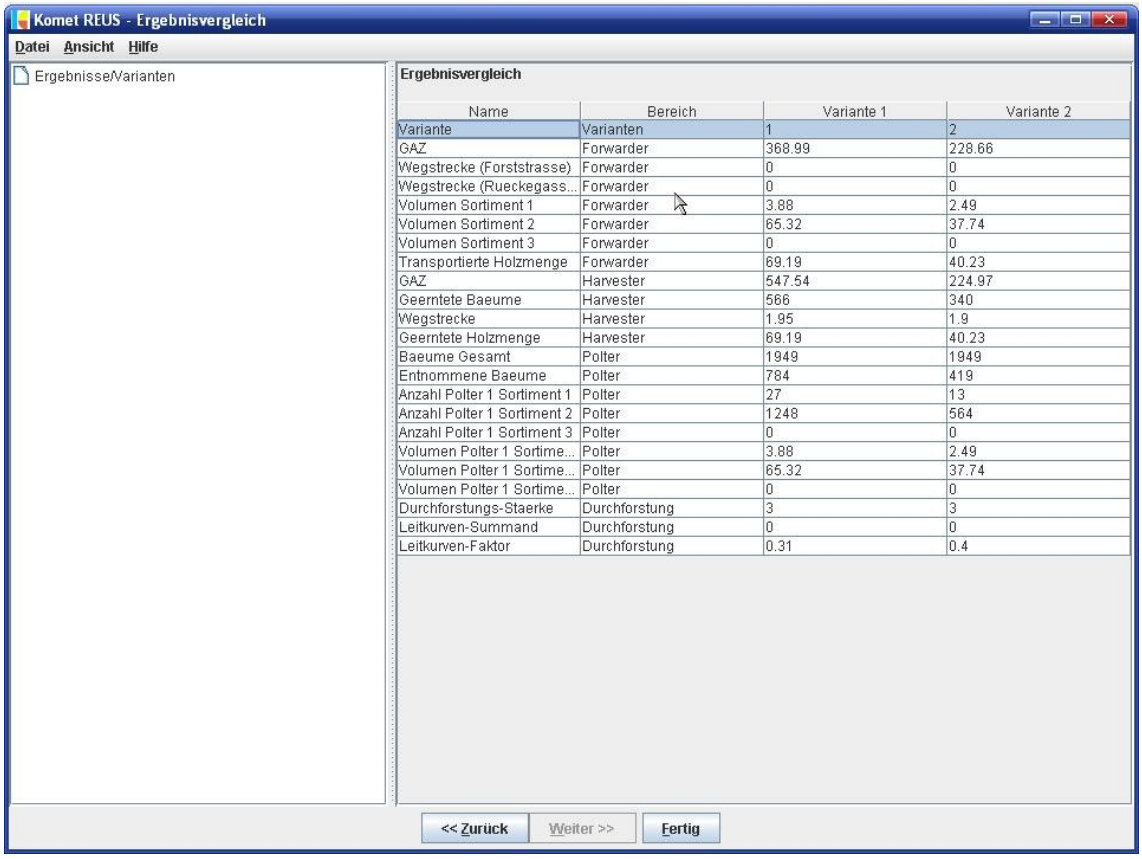


Abbildung 5.24: Ergebnis-Auswahl-Dialog

Nach Auswahl der anzuzeigenden Ergebnisse werden diese für jede Variante in einer übersichtlichen Tabelle formatiert, mit deren Hilfe der Benutzer die Ergebnisse visuell analysieren kann. Dieser abschließende Dialog ist in Abbildung 5.25 dargestellt.



Name	Bereich	Variante 1	Variante 2
Variante	Varianten	1	2
GAZ	Forwarder	368.99	228.66
Wegstrecke (Forststrasse)	Forwarder	0	0
Wegstrecke (Rueckegass...	Forwarder	0	0
Volumen Sortiment 1	Forwarder	3.88	2.49
Volumen Sortiment 2	Forwarder	65.32	37.74
Volumen Sortiment 3	Forwarder	0	0
Transportierte Holzmenge	Forwarder	69.19	40.23
GAZ	Harvester	547.54	224.97
Geerntete Baeume	Harvester	566	340
Wegstrecke	Harvester	1.95	1.9
Geerntete Holzmenge	Harvester	69.19	40.23
Baeume Gesamt	Polter	1949	1949
Entnommene Baeume	Polter	784	419
Anzahl Polter 1 Sortiment 1	Polter	27	13
Anzahl Polter 1 Sortiment 2	Polter	1248	564
Anzahl Polter 1 Sortiment 3	Polter	0	0
Volumen Polter 1 Sortime...	Polter	3.88	2.49
Volumen Polter 1 Sortime...	Polter	65.32	37.74
Volumen Polter 1 Sortime...	Polter	0	0
Durchforstungs-Staerke	Durchforstung	3	3
Leitkurven-Summand	Durchforstung	0	0
Leitkurven-Faktor	Durchforstung	0.31	0.4

Abbildung 5.25: Tabellarische Anzeige der Ergebnisse zum Varianten-Vergleich

6 Diskussion und Ausblick

6.1 KOMET-Architektur

6.1.1 Vorgehensmodell

Der Einsatz eines Software-Engineering-Prozesses im Rahmen der Definition und Realisierung der *KOMET-Architektur* hat sich als richtig und zielführend erwiesen, denn erst dadurch wurde ein zielgerichteter und strukturierter Entwicklungsverlauf ermöglicht. Der Mehraufwand der Planung im Vorfeld wurde durch eine lückenlose Vorgehens- und Programmdokumentation sowie einen geringeren Codierungsaufwand aufgewogen. Das im Rahmen der vorliegenden Arbeit verwendete modifizierte Wasserfallmodell hat sich bewährt. Grundsätzlich erscheinen vom Wasserfallmodell abgewandelte Vorgehensmodelle für den Einsatz in kleinen Arbeitsgruppen gut geeignet, da sie wegen ihrer geringen Komplexität sehr einfach eingeführt und umgesetzt werden können. Das strukturierte Vorgehen und die laufende Verifikation der Anwendungsarchitektur während der Prototypingphase führten zu einer schrittweisen Verfeinerung und Verallgemeinerung der Software-Architektur. Als Ergebnis entstand die *KOMET-Architektur* als Grundlage einer Plattform zur Integration von Entscheidungs-Unterstützungs-Komponenten.

6.1.2 Architekturmodell

Im Rahmen der Konzeption und Realisierung der *KOMET-Architektur* wurden folgende Konzepte realisiert:

- Realisierung eines EUS aus *kooperierenden Komponenten*
- Möglichkeit der *Verteilung* der Komponenten in einem *Netzwerk*
- Definition eines anwendungsübergreifenden abstrakten Datenmodells sowohl *syntaktisch* als auch *semantisch*
- *Ausschließliche Verwendung internationaler Normen und Standards*
- Möglichkeit der anwendungsübergreifenden *zentralen Datenspeicherung.*

Die *KOMET-Architektur* schließt somit die in Abschnitt 3.3 identifizierten Lücken des derzeitigen Kenntnisstandes.

Im EUS-Kern stehen Klassen und Methoden bereit, welche die Erstellung von EUS-Komponenten erleichtern und beschleunigen, indem sie Implementierungsdetails der *KOMET-Architektur* vor dem Entwickler verbergen und über definierte Schnittstellen zur Verfügung stellen. Bei konsequenter Benutzung dieser Mechanismen können diese Implementierungen unter Beibehaltung der bisherigen Schnittstellen verändert oder ausgewechselt werden. Auf diese Weise könnte z. B. *KometML* durch eine andere Technologie zur Kommunikation zwischen dem EUS-Kern und den Solvern ersetzt werden. Dies erhöht die Flexibilität der *KOMET-Architektur* und ermöglicht die Einbindung neuer Technologien, ohne dass bereits bestehende EUS-Komponenten geändert werden müssen.

Solver, die mit Hilfe der *KOMET-Architektur* in ein REUS eingebunden werden, zeichnen sich durch die in Abschnitt 3.2.4 beschriebenen Eigenschaften von Softwarekomponenten aus:

- Sie sind *unabhängige Funktionseinheiten*, weil sie sich nicht gegenseitig beeinflussen und keine Seiteneffekte auftreten.
- Sie können *von Dritten zum Aufbau eines REUS* verwendet werden, ihre interne Funktionsweise muss einem REUS-Entwickler nicht bekannt sein. Die Schnittstelle eines Solvers ist in den Metadaten manifestiert.
- Sie besitzen *keinen dauerhaften Status*.

Auf Grund dieser Eigenschaften sind Solver *Komponenten* im Sinne von SZYPERSKI [1998]. Die *KOMET-Architektur* ist somit *komponentenorientiert*.

Die *Komet-Architektur* beschreibt Dienste, deren Kommunikationsmuster auf XML-Botschaften beruhen. Auch die Solver stellen ihre Funktionalität als Dienst zur Verfügung. Die *KOMET-Architektur* entspricht daher in weiten Teilen einer ***s**ervice**o**rientierten **A**rchitektur* (*SOA*) (siehe Abschnitt 3.2.6).

Bei der *KOMET-Architektur* handelt es sich also um eine *komponentenorientierte SOA*. Da diese beiden Konzepte bei der Integration von Anwendungen aus dem Umfeld der Unternehmenssoftware ebenfalls eine große Rolle spielen, kann davon ausgegangen werden, dass die *KOMET-Architektur* als Integrationsplattform sowohl für Entscheidungs-Unterstützungs-Komponenten als auch für andere Standard-Anwendungen aus dem forstlichen Umfeld grundsätzlich geeignet ist.

Die *KOMET-Architektur* als Grundlage einer Integrationsplattform für Entscheidungs-Unterstützungs-Komponenten liefert in zweifacher Hinsicht ein Beitrag zur räumlichen Entscheidungs-Unterstützung für Fragestellungen aus dem forstlichen Bereich und weit darüber hinaus:

- *Während der Entwicklung eines REUS* wird die Integration von Solvern wesentlich erleichtert.
- *Während des Ablaufs des REUS* werden durch automatische Datenweitergabe sowie Aktivierung der Module mögliche Fehlerquellen eliminiert. Zudem kann die Dateneingabe mit Hilfe einer einheitlichen Benutzeroberfläche erfolgen.

6.1.3 Metadatenmodell

Die *KOMET-Architektur* sieht die Verwaltung von Metadaten und den Einsatz einer Ontologie zur taxonomischen Beschreibung forstlicher Messgrößen vor, um eine maschinelle Interpretation der Solverdaten zu ermöglichen. Dadurch wird ein solverübergreifendes objektorientiertes Datenmodell realisiert. Somit werden erstmals *Syntax* und *Semantik* der EUS-Daten gleichzeitig in standardisierter Form abgelegt. Dabei sind die beschreibenden Daten in der Lage, Informationen sowohl über die EUS-Daten selbst als auch über deren Beziehungen untereinander abbilden, während die Regelsysteme zum *Verständnis* der EUS-Daten beitragen können. Ontologien liefern ein mächtiges Instrumentarium zur maschinellen Interpretation des gesamten im EUS verfügbaren Datenmaterials, gleichzeitig wird die Auswertung der EUS-Daten durch standardisierte Zugriffsmechanismen erleichtert. Der automatische Transfer der Informationen zwischen den Solvern sowie die geeignete Weiterverarbeitung der Daten werden wesentlich erleichtert und vielfach erst ermöglicht.

Aus den relationalen Metadaten können auf einfache Weise (JDBC, siehe Abschnitt 3.2.6) Informationen gewonnen werden, die für den reibungslosen Ablauf des REUS notwendig sind. Beispielsweise kann die Planungskomponente die korrekte Reihenfolge der Komponentenaktivierung automatisch ermitteln und sicherstellen, dass ein Solver erst dann aktiviert wird, wenn alle notwendigen Eingangsgrößen vorhanden sind. Ferner können Solver fremde Informationen ausschließlich mit Hilfe des solverübergreifenden Datenmodells identifizieren und darauf zugreifen. Auf diese Weise kann die Verwendung von voreingestellten Tabellen- und Spaltennamen im Quelltext komplett

vermieden werden und es ist dadurch möglich, die Datenweitergabe vollständig zu automatisieren. Eine explizite Kenntnis der relationalen Speichermodelle anderer Solver ist nicht nötig.

Durch den Einsatz einer Ontologie stehen dem REUS die Messwerte eines jeden Solvers als Objekte zur Verfügung, deren Attribute sowie deren gegenseitige Beziehungen abgefragt werden können. So besteht beispielsweise die Möglichkeit, Baumschlüssel zu ermitteln, indem Objekte gesucht werden, deren Klasse sich innerhalb der Vererbungshierarchie der Baumschlüsselklasse *TreeID* befindet, die der Klasse *Tree* zugeordnet ist. Eine derart gestaltete Abfrage könnte in einem rein relationalen Metadatenmodell nur mit erheblichem Aufwand realisiert werden.

Die im Rahmen der vorliegenden Arbeit konzipierte und realisierte zweigleisige Speicherung der Metadaten vereint folgende Vorteile der relationalen Datenhaltung sowie der Datenhaltung mit Hilfe einer Ontologie auf Basis von OWL-DL:

- Relationale Datenhaltung
  - Hohe Performance und Flexibilität durch Einsatz eines austauschbaren RDBMS
  - Einfach zu realisierender Datenzugriff mit Hilfe von Standards (z. B. JDBC bzw SQL)
- OWL-DL
  - Flexible Abfragemöglichkeiten
  - Möglichkeit der Realisierung sehr komplexer Abfragen durch Definition eines mächtigen Regelsystems.

Das gesamte Metadatenmodell der *KOMET-Architektur* wurde zudem so konzipiert, dass das es *dynamisch verändert* werden kann. Werden weitere Solver mit Hilfe des Registrierungs-Dienstes in ein EUS eingebunden und neue Datenobjekte als Instanzen der bereitgestellten Klassen definiert sowie neue Einträge in der relationalen Metadatenbank generiert, wird dadurch das Datenmodell *erweitert*. Analog werden durch entfernen von Solvern mit Hilfe des Un-Registrierungs-Dienstes die entsprechenden Datenobjekte und Datenbankeinträge gelöscht und so das Datenmodell *verkleinert*. Dies setzt *neue Maßstäbe* in Bezug auf *Flexibilität* und *Konfigurierbarkeit* von Entscheidungs-Unterstützungs-Systemen.

6.1.4 Datenhaltungsmodell

Die *KOMET-Architektur* sieht eine zentrale Datenhaltung vor. Diese wurde gewählt, weil für Solver die Notwendigkeit der Bereitstellung von Methoden für den externen Zugriff auf deren Daten entfällt. Aspekte der Datensicherheit und des Datenschutzes können mit Hilfe von Funktionen eines zentralen Datenbank Management Systems berücksichtigt werden. Mechanismen bezüglich Zugriffskontrolle, Verschlüsselung oder Datensicherung innerhalb der Solver sind deshalb ebenfalls nicht nötig. Dadurch steht ein standardisierter Zugriffspfad für sämtliche im EUS verfügbaren Daten bereit und die Komplexität der Solver wird im Vergleich zu einer verteilten Datenhaltung erheblich verringert. Da sich in der Regel Experten in einem bestimmten Fachgebiet mit der Entwicklung von Solvern beschäftigen, welche keine Informatiker sind, ist dies ein großer Vorteil, denn dieser Personenkreis kann sich nahezu vollständig auf die fachliche Funktionalität der Solver konzentieren.

6.2 Demonstrationsanwendung

Im Rahmen der Realisierung des REUS zur Holzernte wurden verschiedene externe Anwendungen aus dem forstlichen Umfeld mit Hilfe von Wrappern auf Basis der *KOMET-Architektur* integriert und eine Komponente neu implementiert. Die Existenz dieser Demonstrationsanwendung ist Beleg für die grundsätzliche Machbarkeit der Realisierung eines REUS mit Hilfe der *KOMET-Architektur*.

Im Rahmen der Realisierung der Demonstrationsanwendung war die Implementierung vieler Funktionen für Komponenten-Integration und Generierung der grafischen Oberfläche nicht notwendig, da sie bereits vom des EUS-Kerns zur Verfügung gestellt werden. Somit kann der Einsatz der *KOMET-Architektur* die Realisierung eines REUS erleichtern und beschleunigen.

Durch den Einsatz der *KOMET-Architektur* konnten alle REUS-Daten in einer ORACLE-Datenbank zentral gespeichert und der Datentransfer von einer Anwendung zur nächsten vollständig automatisiert werden. Dies belegt die potenziell geringere Fehleranfälligkeit eines REUS auf Basis der *KOMET-Architektur* auf Grund der nicht mehr vorhandenen Medienbrüche.

Die Einstellungen der Solver-Anwendungen wurden mit Hilfe der *KOMET-Architektur* ebenfalls in dieselbe ORACLE-Datenbank geschrieben stehen somit und für eine nachträgliche Auswertung bzw. zur Dokumentation des Entscheidungsfindungsprozesses zur Verfügung. Die Möglichkeit, die Entscheidungsfindung durch den Einsatz der *KOMET-Architektur* transparent und nachvollziehbar zu gestalten, ist somit gegeben.

Sämtliche der bei HEMM [2006] angeführten Verbesserungsmöglichkeiten hinsichtlich einer applikationsübergreifenden Datengrundlage konnten mit Hilfe der *KOMET-Architektur* umgesetzt werden, die von HEMM [2006] geforderte Einbindung digitaler Karten konnte ebenfalls mit Hilfe des neu entwickelten Solvers *Geo-Informationssystem* realisiert werden. Dies unterstreicht die Mächtigkeit und Flexibilität der *KOMET-Architektur*.

Der Waldwachstussimulator *SILVA 2.2* wird im Rahmen der Demonstrationsanwendung zur Simulation von Durchforstungseingriffen verwendet. Die vollständige Integration von *SILVA 2.2* ermöglicht die Nutzung aller Funktionen des Programms.

Das in HEMM [2006] beschriebene Modell zur Holzerntesimulation ist zur Zeit auf einen Bestand beschränkt und benötigt detaillierte einzelbaumbezogene Eingabedaten (Stammfußpunktkoordinaten, BHD, Kronenansatzhöhe, etc.). Mit Hilfe der *KOMET-Architektur* können diese Einschränkungen durch Integration weiterer Komponenten umgangen werden. Bestandesübergreifende Simulationen sind beispielsweise möglich, indem eine zusätzliche Komponente Simulationen in verschiedenen Beständen durchführt und deren Ergebnisse zusammenfasst. Detaillierte Einzelbauminformationen können zum Beispiel mit Hilfe des in *SILVA 2.2* enthaltenen Struktur-Generators aus bestandesweiten Mittelwerten berechnet werden. Dies ist in Demonstrationsanwendung bereits implementiert.

6.3 Ausblick

Beim derzeitigen Entwicklungsstand besteht im REUS zur Holzernte die Möglichkeit, Ergebnisse mehrerer Varianten, die auf unterschiedlichen Einstellungen basieren, visuell direkt miteinander zu vergleichen. Statt der Anzeige der Ergebnisse können in Zukunft Optimierungsverfahren zur automatischen Auswahl einer bestimmten Variante realisiert werden. Dabei kann die Optimierungsmethode *beliebig* gewählt werden, da die Möglichkeit besteht, Optimierungsmodule als Solver zu implementieren. Sowohl eine einfache Nutzwertanalyse als auch ein komplexer Optimierungsalgorithmus, wie beispielsweise *Simulated Annealing* (siehe CHEN UND V. GADOW [2002]), können integriert werden. Die Bereitstellung mehrerer verschiedener Verfahren zur Optimierung ist möglich, wobei der Benutzer wählen kann, welche Methode verwendet werden soll.

Mit Hilfe der GUI-Elemente, welche die *KOMET-Architektur* zur Verfügung stellt, kann eine einfache grafische Oberfläche realisiert werden, die für alle integrierten Anwendungen gleich ist. Unter Zuhilfenahme der Metadaten kann diese Oberfläche in weiten Teilen automatisch generiert werden, so dass der Programmieraufwand sehr gering gehalten werden kann. Durch die Erstellung weiterer Oberflächen-Elemente, wie zum Beispiel Schieberegler oder Auswahllisten kann der Bedienungskomfort weiter erhöht werden. Die in den Metadaten gespeicherten Informationen erlauben die automatisierte Auswahl des geeigneten Oberflächen-Elements für die Dateneingabe durch den Benutzer.

Die Aktivierungsschnittstelle der Solver ist in der *KOMET-Architektur* so definiert, dass auch mit den Solvern eine bidirektionale Kommunikation möglich ist. In zukünftigen Entscheidungs-Unterstützungs-Systemen kann mit Hilfe dieser Schnittstelle ein Informationsaustausch nach dem Vorbild der Kommunikation der Solver mit dem EUS-Kern implementiert werden, um die Funktionalität des REUS zu erweitern. Auf diese Weise könnten zum Beispiel formalisierte Problemstellungen an intelligente Solver übermittelt werden, welche in der Lage sind, diese selbständig zu lösen.

Beim Zugriff auf Objekte in der Ontologie kann nicht vollständig auf explizites Wissen verzichtet werden. Ein Solver kann noch nicht selbst entscheiden, welcher Messwertklasse und welchem Behälterobjekt ein bestimmter Messwert zugeordnet ist. Diese Informationen müssen von einem Programmierer an geeigneter Stelle abgelegt

werden. In diesem Zusammenhang kann untersucht werden, inwieweit dieser Nachteil durch die Entwicklung von geeigneten Regeln, welche in OWL-DL darstellbar sind, eliminiert werden kann.

Mit Hilfe der *KOMET-Architektur* war es möglich, ein REUS zu realisieren, welches die Bereiche Waldwachstumssimulation, Sortierung und Holzerntesimulation abdeckt. Eine Erweiterung dieses REUS auf die Bereiche Holzabfuhr und Holzlogistik ist aus heutiger Sicht möglich und machbar. Somit besteht die Möglichkeit, den kompletten Prozess der Holzbereitstellung vom Wald bis ins Sägewerk im Computer darzustellen. Die *KOMET-Architektur* kann hierfür als wertvolles Integrationswerkzeug dienen.

Das Anwendungsgebiet eines REUS auf Basis der *KOMET-Architektur* wird durch die eingesetzte Ontologie maßgeblich bestimmt, da durch deren Klassenstruktur festgelegt wird, welche Informationen sich im REUS befinden können. Es ist möglich mit Hilfe der *KOMET-Architektur* ein REUS zu realisieren, das *Fragestellungen aus einem beliebigen Bereich* abdecken kann, wenn die im Rahmen der vorliegenden Arbeit erstellte Ontologie ersetzt bzw. erweitert wird. Je universeller und umfangreicher die eingesetzte Ontologie ist, desto mehr Klassen stehen den Solvern für die Definition ihrer Objekte zur Verfügung und desto breiter ist das mögliche Anwendungsfeld des REUS. In diesem Zusammenhang können Ontologien in OWL-DL entwickelt werden, die den Einsatzbereich der *KOMET-Architektur* auf weitere forstliche Fragestellungen ausweiten oder neue Einsatzbereiche aus anderen Bereichen erschließen.

Die *KOMET-Architektur* könnte sich im Rahmen eines zuvor beschrieben Weiterentwicklungsprozesses auf Grund ihrer weitgehenden Allgemeingültigkeit und Flexibilität als *Standard-Software-Architektur* etablieren.

Ontologien sind schon jetzt sehr mächtige Werkzeuge und deren Erforschung steht erst am Anfang. Mit ihrer Hilfe könnte es möglich sein, intelligente Softwaresysteme zu konstruieren, welche in Ansätzen *»wissen was sie tun«*.

7 Literatur

Applied Materials, Inc. (2007): *Manufacturing Simulation and Modeling Software. Applied Materials.* http://www.automod.com, letzter Aufruf: 20.06.2007.

Bartelme, Norbert (1995): *Geoinformatik. Modelle, Strukturen, Funktionen.* Springer, Berlin, Heidelberg.

Becker, Gero; Jaeger, Dirk; Koch, Barbara (1998): *Innovative Techniken zur Unterstützung der operativen Planung im Forstbetrieb. AFZ/Der Wald*, 26, S. 1577-1579.

Bennett, David A. (1997): *A framework for the integration of geographical information systems and modelbase management. International Journal of Geographical Information Science*, 11, H. 4, S. 337-357.

Berners-Lee, Tim; Hendler, James; Lassila, Ora (2001): *The Semantic Web. Scientific American*, 284, H. 5, S. 34-43.

Biber, Peter; Dursky, Jan; Pommerening, Arne (2000): *Silva 2.2. Benutzerhandbuch.* Freising.

Blake, Brian M.; Gomaa, Hassan (2005): *Agent-oriented compositional approaches to services-based cross-organizational workflow. Decision Support Systems*, 40, S. 31-50.

Bleul, Andreas (2001): *Porogrammier-Extremisten: Softwareprojekte auf den Kern reduziert. c't - Magazin für Computer Technik*, 3, S. 182-185.

Booch, Grady (1994): *Objektorientierte Analyse und Design.* Addison-Wesley, Bonn, Paris, Reading u.a.

Burbeck, Steve (2000): *The Tao of e-business services.* http://www.ibm.com/software/developer/library/ws-tao/index.html, letzter Aufruf: 14.03.2007.

Chen, BoWang; v. Gadow, Klaus (2002): *Timber harvest planning with spatial objectives, using the method of simulated annealing. Forstwissenschaftliches Centralblatt*, 121, S. 25-34.

Church, Richard L.; Murray, Alan T.; Figueroa, Michael A. (2000): *Support System development for forest ecosystem management. European Journal of Operational Research*, 121, H. 2, S. 247-258.

Dublin Core Metadata Initiative (2007): *DublinCore Metadata Initiative (DCMI).* http://www.dublincore.org, letzter Aufruf: 28.03.2007.

DEAN, Mike; SCHREIBER, Guus (Hrsg.) (2004): *OWL Web Ontology Language Reference. W3C Recommendation 10 February 2004.* http://www.w3.org/TR/owl-ref/, letzter Aufruf: 20.06.2007.

DENZER, Ralf; GÜTTLER, Reiner; HELL, Thorsten (2002): *An Architecture for Integrated Spatial Decision Support Systems.* Pillmann, Werner; Tochtermann, Klaus (Hrsg.): *Environmental Communication in the Information Society. Proceedings of the 16th Conference "Informatics for Environmental Protection". Part 2.* Wien, S. 182-189.

DÖLLERER, Martin (2003): *Implementierung von Entscheidungsunterstützungssystemen auf Komponentenbasis.* Köhl, M.; Quednau H.-D. (Hrsg.): *Tagungsband der 14. Tagung der Sektion Forstliche Biometrie und Informatik der DVFFA in Tharandt, 3.-5.April 2002.* Ljubljana, S. 27-33.

EDWARDS, Jeri; DEVOE, Deborah (1997): *3-Tier Client/Server At Work.* Wiley, New York.

ELTING, Andreas; HUBER, Walter (2001): *Immer im Plan? Programmieren zwischen Chaos und Planwirtschaft. c't - Magazin für Computer Technik*, 2, S. 184-191.

ELTING, Andreas; HUBER, Walter (2001): *Schnellverfahren: Mit Extreme Programming immer im Plan? c't - Magazin für Computer Technik*, 3, S. 186-191.

ENDEJAN, Marcel (2002): *A Software Architecture for Integrated Assessment Models: Design and Implementation.* Pillmann, Werner; Tochtermann, Klaus (Hrsg.): *Environmental Communication in the Information Society. Proceedings of the 16th Conference "Informatics for Environmental Protection". Part 2.* Wien, S. 190-197.

ERNI, Vinzenz; LEMM, Renato; FRUTIG, Fritz (2002): *Abbildung von Prozessketten mit Componentware, dargestellt am Beispiel der Holzerntekette. Schweizerische Zeitschrift für Forstwesen*, 153, H. 12, S. 462-470.

FLÜGGE, Matthias; SCHMIDT, Kay Uwe (2005): *Verständnisvolle Maschinen. Web Services mit der OWL. iX - Magazin für professionelle Informationstechnik*, 5, S. 136-143.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph (1995): *Design Patterns: Elements of Reusable Object-Oriented Software.* Boston, San Franzisco, Addison-Wesley, New York u.a.

GRIFFEL, Frank (1998): *Componentware: Konzepte und Techniken eines Softwareparadigmas.* Dpunkt-Verlag, Heidelberg.

GÜNTHER, Oliver (1998): *Environmental Information Systems.* Springer, Berlin, Heidelberg.

HEMM, Martin (2006): *Simulation forsttechnischer Arbeitsprozesse unter Einsatz der Software AutoMod durch Modellierung eines Testbestandes aus dem Staatlichen*

Forstamt Paderborn. Technische Universität München, Lehrstuhl für Forstliche Arbeitswissenschaft und Angewandte Informatik, Dissertation, Freising.

HEMM, Martin; OROS, Sven (2005): *Modelldokumentation zur Erstellung einer Simulationssoftware für die Entscheidungsunterstützung in der forsttechnischen Produktion.* Freising.

HITZ, Martin; KAPPEL, Gerti (1999): *UML @ Work: Von der Analyse zur Realisierung.* Dpunkt-Verlag, Heidelberg.

HOFSTATTER, Douglas (1989): *Gödel, Escher, Bach: Ein endloses geflochtenes Band. 12. Auflage.* Klett-Cotta, Stuttgart.

HUMM, Bernhard; WIETEK, Frank (2005): *Architektur von Data Warehouses und Business Intelligence Systemen. Informatik Spektrum*, 28, H. 1, S. 3-14.

JANKOWSKI, Piotr; NYERGES, Timothy L.; SMITH, Alan (1997): *Spatial group choice: a SDSS tool for collaborative spatial decision-making. International Journal of Geographical Information Science*, 11, H. 6, S. 577-602.

KEENAN, Peter (1997): *Using a GIS as a DSS Genarator.* http://mis.ucd.ie/staff/pkeenan/gis_as_a_dss.html, letzter Aufruf: 17.01.2007.

KÖHL, M.; QUEDNAU H.-D. (Hrsg.) (2003): *Tagungsband der 14. Tagung der Sektion Forstliche Biometrie und Informatik der DVFFA in Tharandt, 3.-5.April 2002.* Ljubljana.

KRUCHTEN, Philippe (1996): *A Rational Development Process. Crosstalk*, 9, H. 7, S. 11-16.

KUROPKA, Dominik; WESKE, Mathias (2006): *Die Adaptive Services Grid Plattform: Motivation, Potential, Fnktionsweise und Anwendungsszenarien. EMISA Forum*, 27, H. 1, S. 13-25.

LAUKIEN, Mark; RESENDES, Robert (1998): *Versteckte Details: Einführung in die CORBA-Programmierung. iX - Magazin für professionelle Informationstechnik*, 12, S. 158-165.

LEMM, Renato; ERNI, Vinzenz; THEES, Oliver (2002): *Komponentenbasierte Softwareentwicklung: Neue Perspektiven für forstliche Modellierung und Informationsverarbeitung. Schweizerische Zeitschrift für Forstwesen*, 153, H. 1, S. 3-9.

LEUNG, Yee (1997): *Intelligent Spatial Decision Support Systems.* Springer, Berlin, Heidelberg.

LÜTHY, Denise (1998): *Unterstützung der Holzernteplanung im Alpengebiet: Entwicklung eines Prototyps mit ARC/INFO. Arc Aktuell*, 1, S. E10.

LÜTHY, Denise (1998a): *Entwicklung eines "Spatial Decision Support"-Systems (SDSS) für die Holzernteplanung in steilen Geländeverhältnissen.* vdf Hochschulverlag AG an der ETH Zürich, Zürich.

MANOLA, Frank; MILLER, Eric (Hrsg.) (2004): *RDF Primer. W3C Recommendation 10 February 2004.* http://www.w3.org/TR/rdf-primer/, letzter Aufruf: 10.06.2007.

MAYER, Margaret K. (1998): *Future trends in model management systems: parallel and distributed extensions. Decision Support Systems*, 22, S. 325-255.

MCGUINNESS, Deborah L.; VAN HARMELEN, Frank (Hrsg.) (2004): *OWL Web Ontology Language Overview. W3C Recommendation 10 February 2004.* http://www.w3.org/TR/owl-features/, letzter Aufruf: 20.06.2007.

MOHR, Andreas (1997): *Integration eines Desktop-Mapping-Systemes in eine bestehende forstliche Anwendung unter besonderer Berücksichtigung professioneller Methoden der Programmentwicklung.* Ludwigs-Maximilians-Universität München, Fachgebiet für Biometrie und Angewandte Informatik, Diplomarbeit, Freising.

MÜLLER, Frank (2000): *Was der Fall ist: Entwicklungswerkzeuge im Vergleich: OEW und OTW. iX - Magazin für professionelle Informationstechnik*, 2, S. 68-72.

NUTE, Donald; POTTER, Walter D.; MAIER, Frederik (2002): *Intelligent Model Management in a Forest Ecosystem Management Decision Support System.* Rizzoli, Andrea E.; Jakeman, Anthony J. (Hrsg.): *Integrated Assessmend and Decision Support. Prodeedings of the First Biennal Meeting of the International Environmendal Modelling and Software Society. Volume 3.* Manno, Schweiz, S. 396-401.

OBJECT MANAGEMENT GROUP (2007): *Unified Modelling Language. UML Resource Page.* http://www.uml.org, letzter Aufruf: 06.03.2007.

PAGEL, Bernd U.; SIX, Hans W. (1994): *Software Engineering: Die Phasen der Softwareentwicklung.* Addison-Wesley, Bonn, Paris, Reading u.a..

PILLMANN, Werner; TOCHTERMANN, Klaus (Hrsg.) (2002): *Environmental Communication in the Information Society. Proceedings of the 16th Conference "Informatics for Environmental Protection". Part 1.* Wien.

PILLMANN, Werner; TOCHTERMANN, Klaus (Hrsg.) (2002): *Environmental Communication in the Information Society. Proceedings of the 16th Conference "Informatics for Environmental Protection". Part 2.* Wien.

PRETZSCH, Hans (2001): *Modellierung des Waldwachstums.* Blackwell, Berlin.

PRETZSCH, Hans; BIBER, Peter; DURSKY, Jan (2002): *The single tree-based stand simulator SILVA: construction, application and evaluation. Forest Ecology and Management*, 162, S. 3-21.

REINHOLD, Markus; VERSTEEGEN, Gerhard (2002): *Geordnetes Chaos: V-Modell versus Rational Unified Process. iX - Magazin für professionelle Informationstechnik*, 5, S. 135-137.

RIZZOLI, Andrea E.; JAKEMAN, Anthony J. (Hrsg.) (2002): *Integrated Assessmend and Decision Support. Prodeedings of the First Biennal Meeting of the International Environmendal Modelling and Software Society. Volume 3.* Manno, Schweiz.

SCHÖPFER, Walter; KÄNDLER, Gerald; STÖHR, Daniele (2003): *Entscheidungshilfen für Forst- und Holzwirtschaft - zur Abschlussversion von HOLZERNTE. Forst und Holz*, 58, H. 18, S. 545-550.

SEFFINO, Laura A.; BAUZER MEDEIROS, Claudia; ROCHA JANSLE V. (1999): *WOODS - a spatial decision support system based on workflows. Decision Support Systems*, 27, S. 105-123.

SOFTWARE ENGINEERING INSTITUTE, Carnegie Mellon University (Hrsg.) (2000): *3rd International Workshop on Component-Based Software Engineering. Reflection in Practice.* Limerick, Irland.

SENGUPTA, Raja R.; BENNETT, David A. (2003): *Agent-based modelling environment for spatial decision support. International Journal of Geographical Information Science*, 17, H. 2, S. 157-180.

SHAO, Guofan; REYNOLDS, Keith M. (Hrsg.) (2006): *Computer Applications in Sustainable Forest Management. Including Perspectives on Collaboration and Integration.* Springer, Dordrecht, Niederlande.

SIEDERSLEBEN, Johannes; DENERT, Ernst (2000): *Wie baut man Informationssysteme? Überlegungen zur Standardarchitektur. Informatik Spektrum*, 23, H. 4, S. 247-257.

SIMON, Herbert A. (1977): *The New Science of Management Decision.* Prentice Hall.

SMITH, Michael K.; WELTY, Chris; MCGUINNESS, Deborah L. (Hrsg.) (2004): *OWL Web Ontology Language Guide. W3C Recommendation 10 February 2004.* http://www.w3.org/TR/owl-guide/, letzter Aufruf: 20.06.2007.

SOMMERVILLE, Ian (1992): *Software-Engineering. Fourth Edition.* Addison-Wesley, Workingham, Reading, Menlo Park u.a.

SPRAGUE, Ralph H. (1980): *A Framework for the Development of Decision Support Systems. MIS Quarterly*, 4, H. 4, S. 1-26.

STAL, Michael (1995): *Der Zug rollt weiter: CORBA 2.0 und weitere OMG-Standards. iX - Magazin für professionelle Informationstechnik*, 5, S. 160-168.

STEFFENSON, J. (2001): *Forestry Data Model.* http://support.esri.com/index.cfm?fa=downloads.dataModels.filteredGateway&dmid=10, letzter Aufruf: 12.02.2007.

STROBL, Josef; BLASCHKE, Thomas; GRIESEBNER, Gerald (Hrsg.) (2004): *Angewandte Geographische Informationsverarbeitung XVI: Beiträge zum AGIT-Symposium Salzburg '04.* Wichmann, Heidelberg.

STROBL, Josef; DOLLINGER, Franz (Hrsg.) (1998): *Angewandte Geographische Informationsverarbeitung X: Beiträge zum AGIT-Symposium Salzburg '98.* Wichmann, Heidelberg.

SUN MICROSYSTEMS (1998): *The Java Community Process(SM) Program.* http://jcp.org/en/home/index, letzter Aufruf: 17.01.2007.

SUN MICROSYSTEMS (2006): *Free and Open Source Java. Overview.* http://www.sun.com/software/opensource/java/index.jsp, letzter Aufruf: 17.01.2007.

SZYPERSKI, Clemens (1999): *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, New York.

TAYLOR, Kerry; WALKER, Gavin; ABEL, Dave (1999): *A framework for model integration in spatial decision support systems. International Journal of Geographical Information Science*, 13, H. 6, S. 533-555.

TORRES ROJO, Juan M.; SANCHEZ OROIS, Sofia (2005): *A decision support system for optimizing the conversion of rotation forest stands to continuous cover forest stands Forest Ecology and Management*, 207, S. 109-120.

VACIK, Harald; LEXER, Manfred J. (2007): *Decision Support Systems zum Wissens-transfer. Forst und Holz*, 62, H. 9, S. 28-33.

VACIK, Harald; LEXER, Manfred J.; PALMETZHOFER, Dietmar (2004): *Anwendung der Entscheidungshilfe CONES zur Planung von Verjüngungseingriffen im Gebirgs-wald.* Strobl, Josef; Blaschke, Thomas; Griesebner, Gerald (Hrsg.): *Angewandte Geographische Informationsverarbeitung XVI: Beiträge zum AGIT-Symposium Salzburg '04.* Wichmann, Heidelberg, S. 715-723.

WALLNAU, Kurt C.; PLAKOSH, Daniel (2000): *WaterBeans: A Custom Component Model and Framework.* Software Engineering Institute, Carnegie Mellon University (Hrsg.): *3rd International Workshop on Component-Based Software Engineering. Reflection in Practice.* Limerick, Irland.

WEGENER, Hans (1999): *Extreme Ansichten: Für und Wider des Extreme Programming. iX - Magazin für professionelle Informationstechnik,* 12, S. 126-130.

WELLS, Don (2006): *Extreme Programming: A gentle introduction.* http://www.extremeprogramming.org, letzter Aufruf: 17.01.2007.

WEST, Lawrence A. (2002); HESS, Traci J.: *Metadata as a knowledge management tool: supporting intelligent agent and end user access to spatial data. Decision Support Systems*, 32, S. 247-264.

WIRTH, Niklaus (1983): *Algorithmen und Datenstrukturen. 3., überarbeitete Auflage.* Teubner, Stuttgart.

ZEILER, Michael (Hrsg.) (2002): *Exploring ArcObjects. Vol. 1 - Applications and Cartography.* ESRI Press, Redlands.

ZEILER, Michael (Hrsg.) (2002a): *Exploring ArcObjects. Vol. 2 - Geographic Data Management.* ESRI Press, Redlands.

ZIEGLER, Cai (2003): *Surfende Maschinen. Web Ontology Language (OWL): Vokabulare fürs Web. iX - Magazin für professionelle Informationstechnik*, 12, S. 108-113.

ZIESAK, Martin (1999): *Ein Informationssystem für den bodenschonenden Einsatz von Forstmaschinen. Forsttechnische Informationen des KWF*, 5+6, S. 55-57.

ZIESAK, Martin (2004): *Entwicklung eines Informationssystems zum bodenschonenden Forstmaschineneinsatz.* Technische Universität München, Lehrstuhl für Forstliche Arbeitswissenschaft und Angewandte Informatik, Dissertation, Freising.

ZIESAK, Martin; BRUCHNER, Anne-Katrin; HEMM, Martin (2004): *Simulation technique for modelling the production chain in forestry. European Journal of Forest Research*, 123, S. 239-244.

8 Zusammenfassung

Anhaltende Trends im Forstbereich zu einer nachhaltigen multifuktionalen Waldbewirtschaftung, schonendem Maschineneinsatz und zunehmende Kundenorientierung sorgen für steigende Komplexität forstlicher Managemententscheidungen. Deshalb gewinnt die computerunterstützte Entscheidungsfindung zunehmend an Bedeutung.

Die im Rahmen dieser Arbeit vorgestellte *KOMET-Architektur* beschreibt eine offene Integrationsplattform für forstliche Entscheidungsunterstützungskomponenten. Sie ermöglicht einen hohen Automatisierungsgrad, wodurch Medienbrüche und daraus resultierende Fehler vermieden werden können. Die Architektur ist in drei Schichten aufgebaut. Der *EUS-Kern* stellt den Komponenten alle notwendigen Dienste zur Verfügung und verwaltet die Metadaten sowohl in einer zentralen relationalen Datenbank als auch in einer Ontologie, die auf der *Web Ontology Language* (*OWL*) basiert. Diese Metadaten beschreiben ein komponentenübergreifendes objektorientiertes Datenmodell und ermöglichen eine maschinelle Interpretation der Solverdaten. Der Informationsaustausch zwischen den Kompenten und dem Systemkern ist mit Hilfe der *Extenxible Markup Language* (*XML*) realisiert. Hierfür wurde die *KometML* entworfen. Dies gilt auch für das Planungsmodul, das ebenfalls als Komponente realisiert ist. Im Zuge der Implementation einer Demonstrationsanwendung zur Simulation von Holzerntemaßnahmen wurden *ArcGIS Engine 9.2*, *Silva 2.2*, *Holzernte 7.0* und *AutoMod 11.0* in die Integrationsplattform eingebunden. Dabei hat sich gezeigt, dass die *KOMET-Architektur* zum Aufbau solcher Entscheidungsunterstützungssysteme geeignet ist.

Die Integration von Entscheidungs-Unterstützungs-Komponenten ist nicht auf das forstliche Umfeld beschränkt. Die *KOMET-Architektur* ist für die Einbindung von Komponenten aus anderen Anwendungsbereichen ausgelegt.

9 Abstract

Ongoing trends in forestry such as sustainable multifunctional forest management, low impact mechanized forest operations and consumer oriented marketing increase complexity of forest decisions. Thus, there is an emerging trend towards computer aided decision-making.

The *KOMET-architecture* discussed in this paper describes an open integration platform for forestry decision support components providing a high grade of automation rather than manual data transfer between applications causing media breaks and errors. The core architecture is three tiered. The system kernel (*EUS-Kern*) provides all necessary services to the components and maintains the meta data stored in a central relational data base management system as well as an ontology that is based upon the *Web Ontology Language* (*OWL*). These meta data describe an inter-component object oriented data model that enables computational interpretation of a solver's data. The conversation between the components and the kernel is realised using the *Extensible Markup Language* (*XML*). Therefore *KometML* was designed. This includes the planning module, which is implemented as a special component. During the implementation of a demonstration SDSS for simulation of forest harvesting operations *ArcGIS Engine 9.2*, *Silva 2.2*, *Holzernte 7.0* and *AutoMod 11.0* were linked into the integration platform. It has shown that for building such decision support systems the *KOMET-architecture* is feasible.

The *KOMET-architecture* is not restricted to maintain only forestry decision support components. Integration of components from other fields of application is possible by design.