Entwicklung einer Reportingplattform als entscheidungsunterstützendes System

- Ein komponentenbasierter Ansatz -

Von der Fakultät für Mathematik, Naturwissenschaften und Informatik
der Brandenburgischen Technischen Universität Cottbus

zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
(Dr.-Ing.)

genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Jörg Rieckmann
geboren am 26. Februar 1963 in Hamburg

Gutachter: Prof. Dr. rer. nat. Bernhard Thalheim

Gutachter: Prof. Dr. rer. nat. Joachim W. Schmidt

Gutachter: Prof. Dr.-Ing. Bernd Scholz-Reiter

Tag der mündlichen Prüfung: 12. Februar 2001

Inhaltsverzeichnis

ZUSAMMENFASSUNG

Die zunehmende Globalisierung der Märkte und der damit zusammenhängende steigende Wettbewerbsdruck stellen immer höhere Anforderungen an die teilnehmenden Unternehmen und deren Entscheider. Durch die Einführung flacher Hierarchien wird sukzessive mehr Verantwortung auf den einzelnen Mitarbeiter delegiert. Dies führt insbesondere auf der operativen Ebene zu einem überproportionalen Anstieg der Komplexität einzelner Aufgabenbereiche.

Die vorliegende Dissertation beschreibt die Entwicklung eines Systems zur Unterstützung des Endanwenders in komplexen Entscheidungssituationen. Aus der jeweiligen Anwendungsdomain werden als stabil erachtete Varianten (Prozesse, SQL-Statements und Programme) als Komponenten im Sinne von Componentware modelliert und dem Anwender mit robusten Verknüpfungsmechanismen zur Verfügung gestellt. Hieraus kann sich der Anwender (neue) Lösungsstrategien zusammenstellen und mittels einer systemseitig gesteuerten Benutzerführung bearbeiten. Die flexible Ausnahmebehandlung (bei Exceptions) wird durch die Verwendung der Componentwarekonzepte und über die mögliche Verknüpfung / Adaption der Komponenten durch den Anwender zur Laufzeit gewährleistet.

ABSTRACTS

The increasing globalization of markets and the resulting competition raise high requirements for the participating companies and their executives. By the introduction of flat company hierarchies, responsibility increasingly moves towards the single employee. In particular in the operational level, specific areas suffer from a more than proportional rise of their complexity.

This thesis describes the development of a system for the support of end users in making complex decisions. After surveying the according application area, stable variants (as processes, SQL statements, and programmes) are modeled in terms of components in the spirit of componentware. In addition, robust mechanisms for combining components are offered to the end user. Thus, the user may compile (new) solution strategies while being guided by the system. The flexible exception handling is guaranteed by the concepts of the componentware and by combining and adapting given components by the user at run time.

1 EINFÜHRUNG

1.1 ÜBERSICHT

Die zunehmende Globalisierung der Märkte und der damit zusammenhängende steigende Wettbewerbsdruck stellen immer höhere Anforderungen an die daran teilnehmenden Unternehmen und deren Entscheider auf den unterschiedlichen Ebenen. Ständig neue Wettbewerbsfaktoren erfordern zeitnahes Handeln. Durch die Einführung flacher Hierarchien wird sukzessive mehr Verantwortung auf den einzelnen Mitarbeiter delegiert. Dies führt auf sämtlichen Hierarchieebenen zu einem überproportionalen Anstieg der Komplexität einzelner Aufgabenbereiche. Die Vielzahl an unterschiedlichen Problemstellungen ist deshalb nur durch langjährige Erfahrungen in adäquater Art und Weise lösbar. Aufgrund der zu bearbeitenden Informationsflut bzw. der meist unter Termindruck zu fällenden Entscheidungen ist eine korrekte Umsetzung der jeweiligen Geschäfts- bzw. Entscheidungsprozesse oft nicht mehr gewährleistet. Die seitens der kommerziellen Anbieter im Bereich der Informationstechnologie bereitgestellten Werkzeuge bieten insbesondere in Anwendungsbereichen (*Domains*), die eine starke Anpassung verschiedener Prozesse an konkrete Situationen benötigen, keine Hilfestellungen, so daß die Mitarbeiter bei der Lösung komplexer Arbeitssituationen mit vielfältigen Ausnahmebehandlungen keine ausreichende Unterstützung erhalten.

Die in der vorliegenden Arbeit entwickelte *Reportingplattform* unterstützt den Anwender in diesen Entscheidungssituationen durch die Komposition und Bereitstellung verschiedener Basistechnologien. Der Begriff Reportingplattform drückt die Vorgehensweise aus, Problemstellungen durch die Informationsbereitstellung in Form unterschiedlichster "Reports" am Bildschirm und in Papierform zu lösen. Die Bereitstellung der Informationen muß dazu auf das jeweilige zu lösende Problem ausgerichtet sein. Hierzu wird die Reihenfolge und der strukturelle Aufbau der Reports im Rahmen komplexer Entscheidungsprozesse systemseitig in Form von Lösungsvorschlägen angeboten, die daraufhin benutzerseitig angepaßt werden können. Die "Plattform" zur Generierung der Reports stellt Technologien zur Verfügung, die zur Unterstützung der Entscheidungsfindung dienen und aufgrund der Flexibilität der entwickelten Systemarchitektur beliebig um weitere Bausteine ergänzt werden können.

Der Prozeß der Entscheidungsfindung vermischt in der Regel Informationen und die Erfahrung, diese Informationen problembezogen zu deuten und in Beziehung zueinander zu setzen. Da Erfahrungen nur bedingt formalisiert und strukturiert werden können, ist eine Kombination beider Einflußfaktoren bei der Entscheidungsfindung aus Sicht der Informationstechnologie äußerst schwierig.

In verschiedenen Anwendungsbereichen müssen Lösungen an die jeweiligen Entscheidungssituationen angepaßt werden. Zur Unterstützung dieser Art von Prozessen müßte eine Vielzahl von Varianten in Form von Prozessen und Informationsabfragen vormodelliert werden, was durch eine entsprechend große Anzahl zu entwickelnder Programme erfolgen könnte, die den jeweiligen Workflow spezifizieren. Dies wäre jedoch zu unflexibel und scheidet aus wirtschaftlichen Gründen aus. Zudem unterliegen die Varianten ebenfalls einer ständigen Veränderung und würden damit zu einem permanenten Wartungsaufwand führen.

Eine weitere Möglichkeit wäre, von verschiedenen Varianten zu abstrahieren bzw. einzelne *Use Cases* und *Szenarien* [Booc96, S.89] zu generalisieren, um so einen Großteil der Varianten abzudecken. In diesem Fall ist jedoch mit einer hohen Anzahl nicht lösbarer Problemstellungen zu rechnen.

Wenn es jedoch gelänge, die verschiedenen Varianten auf Basiskomponenten zurückzuführen, deren Modellierung mit vertretbarem Aufwand möglich wäre und die zukünftig einer geringen Änderungswahrscheinlichkeit unterliegen, könnte dies eine Ausgangsbasis zur Unterstützung des Anwenders bilden.

Notwendig ist dazu die Entwicklung eines intelligenten Werkzeugs zur Erstellung kleiner Basiskomponenten und der Steuerung des Zusammenspiels dieser Varianten. Dabei ist denkbar, dieses

Werkzeug für den Modellierer des Systems in der jeweiligen Domain, für den Endanwender direkt oder für beide kombiniert einzusetzen.

Die *Dekomposition* komplexer Prozesse in eine Vielzahl von Prozeßbausteinen erlaubt unter geeigneten Bedingungen eine Wiederverwendbarkeit und Kombinationsmöglichkeit dieser Bausteine. Hierbei ergibt sich unmittelbar die Fragestellung, unter welchen Bedingungen ein Teilprozeß innerhalb eines schwach strukturierten Aufgabengebiets verwendet werden kann, sofern man den Anspruch einer kreativen Prozeßführung erhebt, also der Vermeidung statisch komplexer Ablaufmodelle. In Abbildung 1 wird dargestellt, wie einzelne Prozeßbausteine innerhalb schwach strukturierter Gesamtprozesse wiederverwendet werden können [Poh+95, S. 55].

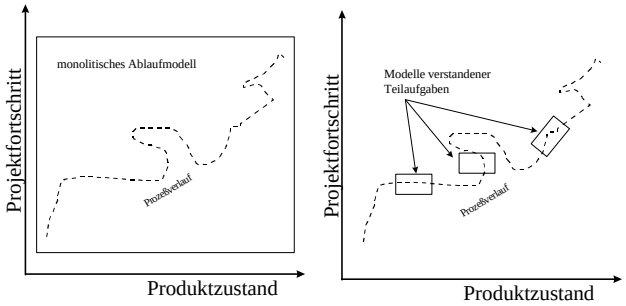


Abbildung 1: Modellierung von gut verstandenen und schwach strukturierten Geschäftsprozessen

Diese Art von Prozessen kann nun durch die Reportingplattform unterstützt werden. Allerdings sind für die Realisierung einige weitere Mechanismen zur Verfügung zu stellen. So müssen die Varianten (Teillösungen) in Form der Basiskomponenten an die jeweilige Problemstellung angepaßt werden können, um für unterschiedlichste Aufgabenstellungen spezielle Lösungen zu entwickeln.

Neben Anpassungsmechanismen der einzelnen Komponenten (z.B. durch Parametrisierung [GuPo98]) muß jedoch ebenfalls die Möglichkeit des Zusammenspiels verschiedener Teillösungen gegeben sein. Dazu sind Komponenten orthogonal miteinander zu kombinieren, zu neuen Komponenten zusammenzufassen bzw. nur Teile der Ursprungskomponenten zu verwenden. Dies kann durch unterschiedliche Abstraktionsmechanismen (z.B. durch Generalisierung / Spezialisierung und Aggregation / Dekomposition [MaSc90][Thal94]) umgesetzt werden.

Exkurs: Problemlösen

Probleme zu lösen bzw. die Lösung einer Aufgabe zu entdecken heißt, Verbindungen zwischen vorher getrennten Dingen oder Ideen zu finden [Póly67, S.16]. Je weiter die Verbindungen der Dinge auseinander liegen, desto größer ist der Verdienst, die Verbindungen zu entdecken. Die Verbindungen können als Metapher in Form einer Brücke gesehen werden. Im Bereich der Mathematik kann ein Beweis auch eine Kette von Argumenten und Schlußfolgerungen sein, wobei die Kette nicht stärker ist, als ihr schwächstes Glied.

Den Werdegang einer Lösung kann man wie folgt nachvollziehen [Póly67, S. 17 ff.]:

- Zunächst die Frage nach der Aufgabe. Wie ist das Ziel der Aufgabe zu definieren? Was ist an Daten vorhanden bzw. worauf kann man aufbauen?
- Wie ist die Lösungsidee, welcher Kurs soll eingeschlagen werden? Wenn man die Aufgabe nicht lösen kann, sollte man eine geeignete verwandte Aufgabe suchen.
- Entwicklung der Idee von verschiedenen Standpunkten aus. Was ist unbekannt, was kann wie beschafft werden etc.?
- Durchführung der Idee, sofern sie schlüssig ist und entsprechend ihrer Machbarkeit geprüft wurde.

Es lassen sich hierbei zwei Phasen erkennen [Póly67, S.27]. Zunächst beschäftigt sich der Aufgabenlöser mit dem Verstehen der Aufgabe. Anschließend entwickelt er das System der logischen

Beziehungen und legt den Lösungsplan fest. Sofern dieser Plan in allen Einzelheiten bestimmt ist, spricht Pólya von einem *Lösungsprogramm*.

Der Werdegang der Lösung ist dadurch gekennzeichnet, daß der Problemlöser bei jedem Schritt neue einschlägige Erkenntnisse heranzieht und z.B. einen neuen ihm bekannten Lehrsatz anwendet. "Somit erscheint die geistige Arbeit des Aufgabenlösers als das Ins-Gedächtnis-Rufen einschlägiger Elemente aus seiner früheren Erfahrung und das In-Beziehung-Setzen dieser Elemente mit der ihm vorliegenden Aufgabe als eine Arbeit des *Mobilisierens* und *Organisierens*" [Póly67, S.29]. Die zur Lösung heranziehbaren Elemente haben bereits in *ähnlichen Situationen* geholfen.

Eine Vorgehensweise zur *erfahrungsbasierten Problemlösung* läßt sich wie folgt beschreiben:

- Suche nach *Analogien*. Analogie ist eine Art Ähnlichkeit auf einem bestimmten, ins begriffliche gehobenen Niveau [Póly62, S.35]. Ähnliche Dinge stimmen miteinander in irgendeiner Beziehung überein. Werden diese Beziehungen, in denen sie übereinstimmen auf bestimmte Begriffe gebracht, so werden diese ähnlichen Dinge als analog betrachtet. "Zwei Systeme sind analog, wenn sie miteinander in bezug auf klar definierbare Beziehungen zwischen ihren sich jeweils entsprechenden Teilen übereinstimmen" [Póly62, S.35].
- *Verallgemeinerung* der Analogien: Verallgemeinerung ist der Übergang der Betrachtung eines Aggregats von Objekten zu der eines größeren Aggregats, das das gegebene enthält.
- *Spezialisieren*, als der Übergang von der Betrachtung eines gegebenen Aggregats von Objekten auf ein kleineres Aggregat, das in dem gegebenen enthalten ist. Oftmals wird von einer Klasse von Objekten auf ein einzelnes Objekt übergegangen, das in der Klasse enthalten ist.

Verallgemeinerung, Spezialisierung und Analogie bilden oft die Basis bei der Lösung mathematischer Aufgaben.

Exkursende

In der vorliegenden Arbeit wurde eine *Reportingplattform* als ein *entscheidungsunterstützendes System* für Prozesse entwickelt, die ermöglicht, zunächst Basiskomponenten in Form von *Prozeßbausteinen* und *Informationsabfragen* zu modellieren und anschließend diese durch den Anwender zur Laufzeit so flexibel zusammenstellen zu lassen, daß ad hoc komplexe Entscheidungsprozesse zur erfahrungsbasierten Problemlösung unterstützt werden können. Die Entscheidungsunterstützung wird dabei für Prozesse geleistet, die sich aus Basislösungen oder ähnlichen Lösungen zusammensetzen lassen.

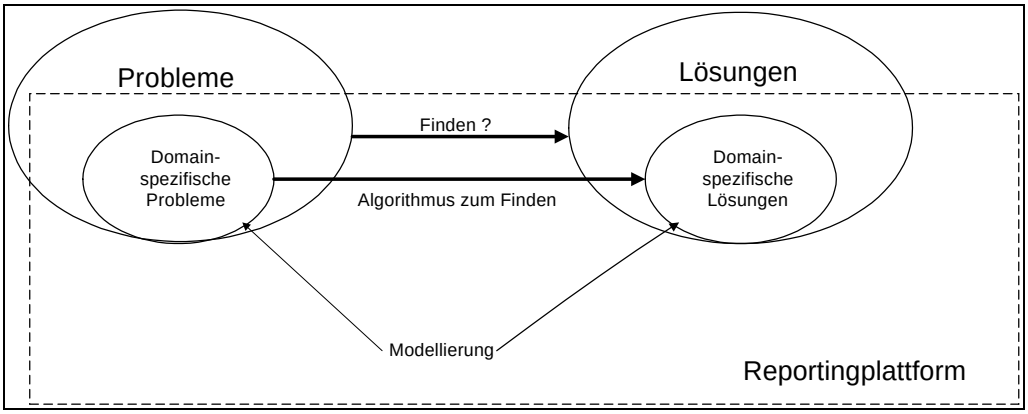


Abbildung 2: Entscheidungsunterstützung durch die Reportingplattform

Abbildung 2 stellt dar, wie das Auffinden von Lösungen für einzelne Problemstellungen mittels der Reportingplattform unterstützt wird. Da es nicht möglich ist, für sämtliche Problemstellungen entsprechende Lösungen vorzuhalten, muß sich auf einzelne Anwendungsbereiche beschränkt werden, für die zu diversen Problemstellungen im Rahmen einer Modellierungsphase Lösungen zugewiesen werden können. Selbst bei dieser ersten Einschränkung wäre es unrealistisch anzunehmen, daß im laufenden Betrieb jedesmal genau eine Problemstellung eintritt, für die eine im Vorwege zugeordnete Lösung existiert. Vielmehr werden Problemstellungen vorkommen, die zu bereits bearbeiteten

Problemstellungen *ähnlich* sind oder aber aus einer Vielzahl von Problemstellungen bestehen, für die bereits ähnliche Lösungen gefunden wurden und damit im System vorhanden sind. Für das Finden dieser ähnlichen Lösungen wird durch die Reportingplattform ein Algorithmus zur Verfügung gestellt.

Das Einsatzgebiet der Reportingplattform bilden *datenintensive Anwendungen* von *dynamischen Workflows*, wobei unter datenintensiven Informationssystemen komplexe Strukturen persistenter Daten, die leicht wartbar und weiterentwickelbar sind, verstanden werden [Tha+95]. Die Reportingplattform unterstützt Anwender in Bereichen, in denen sie zur Lösung ihrer Problemstellungen mit dem notwendig ständigen Wechsel von Informationsabfragen unterschiedlichster Komplexität konfrontiert werden und bislang intuitiv die Reihenfolge selbst bestimmen mußten bzw. durch die genutzten Applikationen eine starre Vorgabe erhielten. Die Workflows lassen sich in diesem Bereich nicht ganzheitlich strukturieren, sondern ändern sich vielmehr von Aufgabenstellung zu Aufgabenstellung. Allerdings wiederholen sie sich oftmals in Teilen oder unterscheiden sich in ihren neuen Ausprägungen nicht vollständig, so daß sich einzelne Basisstrukturen lokalisieren lassen, die sich in ähnlicher Form wiederholen (*teilstrukturierte Arbeitsvorgänge* [Jab+97, S.90], siehe *Kapitel 2.4 Workflowmanagement*). Zur Unterstützung bietet die Reportingplattform den Anwendern die Möglichkeit, diese Prozeßabläufe nun auch zur Laufzeit zu verändern. Insbesondere die Kombination eines flexiblen Reportingtools mit der systemgestützten Benutzerführung zur Entscheidungsunterstützung bilden Alleinstellungsmerkmale der Reportingplattform.

Zur Entwicklung der Reportingplattform sind zunächst Standardisierungen für Input und Output der Anwendung zu entwerfen und anschließend Kombinationsmöglichkeiten der Komponenten sowie deren korrekte *Zusammensteckmechanismen* zu entwickeln. Hierdurch lassen sich komplexe Entscheidungsszenarien, wie sie z.B. im Bereich der LKW-Disposition auftreten (siehe *Kapitel 1.4 Beispieldomain Transportlogistik*) gut unterstützen. Anhand der Beispieldomain LKW-Disposition im Bereich der Transportlogistik wird die Funktionalität der Reportingplattform evaluiert.

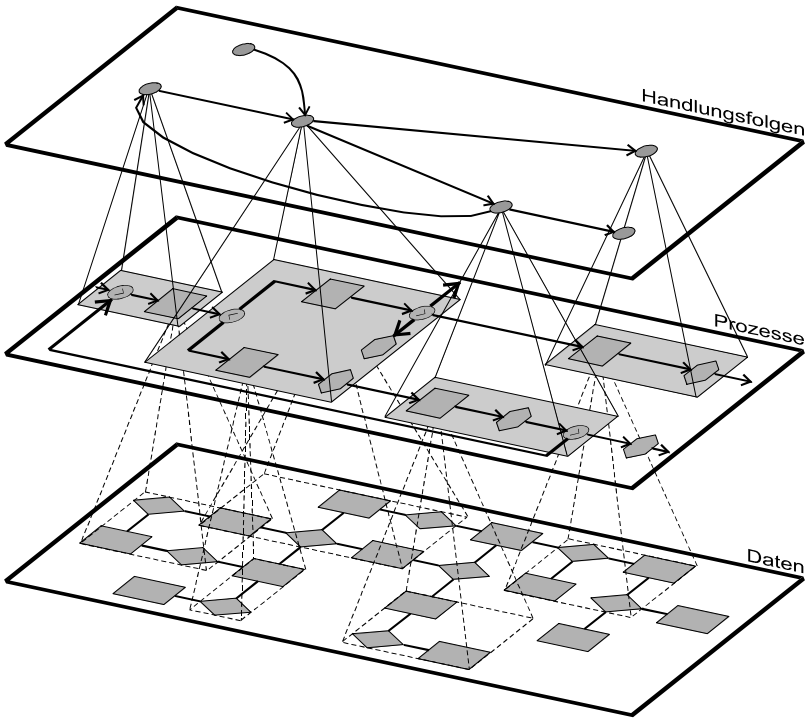


Abbildung 3: Schematische Darstellung des Systemkonzepts

Schematisch stellt sich das entwickelte Gesamtsystem wie in Abbildung 3 dar. Der Anwender kommuniziert mit der Reportingplattform auf der Ebene des Fachkonzepts (Ebene Handlungsfolgen). Die Verwendung von *Query by Example* (*QBE*) [Zloo75] wäre bei komplexeren Abfragen für den Anwender zu kompliziert, da analog zur Verwendung von SQL eine genaue Kenntnis des zugrundeliegenden Datenbankschemas notwendig ist. Natürlichsprachliche Zugänge zu RDBMS (z.B.

NATHAN – Natürlichsprachliches Heuristisches Anfragesystem [Noac89] oder der Ansatz von [Jana82]) scheiden aufgrund der hohen Komplexität und der verschiedenen Restriktionen aus.

Zur erfahrungsbasierten Problemlösung führt der Benutzer unterschiedliche Handlungen aus (Zuordnung von Gütern zu Transportressourcen im Bereich Transportlogistik, Informationsabfragen etc.), die er durch einfaches Point & Click initiiert und spezifiziert. Dabei wird SQL vor dem Benutzer durch die Bereitstellung von DataSets (s.u.) gekapselt. Die Abbildung der Handlungsfolgen auf Prozesse gewährleistet die Systemarchitektur. Einzelne Handlungen determinieren ggf. eine Anzahl an Prozeßschritten und darin enthaltenen Funktionen (Rechtecke auf der Prozeßebene, analog der Syntax für *ereignisgesteuerte Prozeßketten* (*EPK*) [Sche94]). Neben Funktionen können die Prozesse auch weitere Subprozesse, SQL-Statements oder externe Programme enthalten. Durch Subprozesse würden weitere Prozeßebenen zwischen der mittleren Ebene und der Datenebene einzufügen sein.

Der Anwender erhält über die Prozesse und die darin enthaltenen SQL-Statements unterschiedliche Sichten auf die Datenstrukturen (Ebene Daten). Der Datenzugriff erfolgt über entsprechend adaptierbare Komponenten auf Prozeßebene. Änderungen auf der Ebene der Handlungsfolgen bewirken die transparente Adaption auf der Prozeßebene mit der zugehörigen neuen Abbildung auf eine Datensicht.

1.2 ANFORDERUNGEN AN DIE REPORTINGPLATTFORM (BENUTZERSICHT)

Die Reportingplattform wurde zur (Entscheidungs-) Unterstützung sowohl für das Management als auch im operativen Bereich entwickelt. Als Voraussetzung für dieses Informationsangebot erstellt ein IT-Spezialist im Rahmen einer *Modellierungsphase* Szenarien für Problemstellungen mit dazu passenden *Lösungsstrategien* in Form o.g. Komponenten.

Die Reportingplattform bietet dem Benutzer über die modellierten Komponenten eine systemseitige Benutzerführung in Form von *Entscheidungs- oder Lösungsprozessen,* die der Anwender als Reaktion auf aktuelle Ereignisse zur Laufzeit um beliebige Prozesse ergänzen oder einschränken kann. Zusätzlich bietet die Reportingplattform Möglichkeiten zur flexiblen Adaption von *SQL-Statements* unterschiedlicher Komplexität, ohne daß der Anwender direkt mit SQL in Berührung kommt. Die Verknüpfung der Komponenten muß der Anwender über eine benutzerfreundliche Oberfläche durchführen können. Die so spezifizierte große Anzahl unterschiedlicher Komponenten ist jedoch zu vielfältig bzw. zu wenig strukturiert und damit als Werkzeug nicht praktikabel. Aus diesem Grunde sollte die Reportingplattform die durch den Anwender in seiner Terminologie eingegebene Problemspezifikation als Grundlage für Lösungsvorschläge nutzen. Sämtliche Lösungsvorschläge basieren auf den in der Modellierungsphase entworfenen Komponenten.

Die Einbindbarkeit externer Programme ermöglicht die Integration domainspezifischer Anwendungssysteme zu einem homogenen Gesamtsystem. Durch die Möglichkeit der benutzerseitigen Modifikation von Prozessen und SQL-Statements auch zur Laufzeit wird das flexible Handling von Exceptions ermöglicht.

Sollte der Modellierer unterstützt werden, so wird hierdurch die Entwurfsproblematik bei der Vielzahl zu realisierender Varianten eingegrenzt. Ein Großteil von modellierten Varianten kann so in anderen Domains ebenfalls eingesetzt werden oder ist nur minimal zu modifizieren. Der Prozeß zum Anlegen eines Kunden stimmt in vielen Bereichen überein, so daß lediglich eine Anpassung des SQL-Statements erfolgen muß. Meist ist ausreichend, die Tabellennamen im Repository zu ändern.

Beim Einsatz des Werkzeugs für den Endanwender wäre die Voraussetzung für diesen, daß er aufgrund seiner Erfahrungen in dem jeweiligen Fachgebiet entscheiden kann, welche der einzelnen Prozeß- oder Informationsabfragevarianten er nutzen möchte oder welche der bereits modellierten oder neu verknüpften Varianten durch eine neue Kombination zur Lösung des aktuellen Problems beitragen würden. Passen diese nicht zusammen, so ist der Anwender durch Dialogboxen mit in die Konfliktlösung einzubeziehen oder es wird eine obligatorische Fortsetzung realisiert, so daß in jedem Fall die weitere Bearbeitung gewährleistet ist. In diesem Fall muß das System hochgradig robust sein, damit der Anwender bei "inkompatiblen Komponenten" nicht mit systembedingten Problemen konfrontiert

wird. Neu zusammengesetzte / adaptierte Komponenten sollten gespeichert werden können und dienen so zukünftig bei analogen Problemkonstellationen ebenfalls als mögliche Lösungen.

Sind ad hoc neue Problemstellungen zu bearbeiten, so müssen zur Laufzeit die aktuell verwendeten Komponenten "eingefroren" werden können, damit z.B. eine im Gesamtablauf nach der eingefrorenen Komponente zu bearbeitende Komponente sofort genutzt werden kann.

Nachfolgend werden aus der Sicht des Benutzers Anforderungen an die Reportingplattform für dieses "Wunschszenario" zusammengestellt:

1. Einfache Spezifikation von Problemstellungen in der Terminologie der Domain.
2. Kommunikation des Anwenders mit dem System auf Fachkonzeptebene über Listboxen und mittels Drag & Drop, damit die Benutzung auch für weniger versierte Anwender möglich ist (ergonomische Benutzeroberfläche).
3. Aufbereitung der Daten mit Werkzeugen eines EIS-Reportingtools.
4. Systemseitige Bewertung von Lösungsstrategien hinsichtlich der Relevanz zur Problemstellung.
5. Benutzergesteuertes Zusammenfügen verschiedenster Komponenten zu neuen Prozessen / SQL-Statements. Die Konsistenz stellt das System sicher.
6. Benutzergesteuerte Adaption / Anpassung auch komplexer SQL-Statements an den aktuellen Informationsbedarf, ohne daß SQL-Kenntnisse erforderlich sind. Keine Beschränkung auf Snowflake- oder Star-Schemes als Datenstrukturen.
7. Möglichkeit des Durchhangelns durch die verschiedenen Datenbestände mittels Point & Click.
8. Einbindbarkeit externer Programme.
9. Systemseitige Steuerung der Aktivitäten. Dabei soll das "Zementieren von Abläufen" vermieden werden. Auf Exceptions kann der Anwender jederzeit durch Veränderung der Prozesse reagieren.
10. Speichern / "Einfrieren" aktueller Bearbeitungsstände für die spätere Weiterbearbeitung.
11. Gewährleistung der "Lernfähigkeit" des Systems durch Speicherung neuer Lösungen. Diese Funktionalität dient ebenfalls der Wiederverwendung neu generierter Lösungen bei analogen Problemstellungen.
12. Datensicherheit.
13. Einsetzbarkeit in verschiedenen Domains.

1.3 NACHTEILE EXISTIERENDER ANSÄTZE

Für die Erfüllung der genannten Anforderungen existieren bereits Ansätze aus unterschiedlichen Bereichen, die bisher jedoch jeweils für sich genutzt wurden und deshalb mit ihren Nachteilen nicht die notwendige Unterstützung für die Anwender erzielten. In Verbindung mit den in *Kapitel 2 Basistechnologien der Reportingplattform* beschriebenen Konzepten lassen sich einzelne Funktionalitäten hieraus jedoch zu einem Gesamtsystem zusammenfügen, das den gestellten Ansprüchen genügt.

EIS-Generatoren

EIS-Generatoren [Humm98] zur Realisierung von *Führungsinformationssystemen* (siehe *Kapitel 2.3 EIS - Executive Information Systems*) ermöglichen für IT-Spezialisten die Entwicklung und Modellierung von Anwendungen, die anschließend dem Anwender zur Verfügung gestellt werden. Dazu werden einzelne Prozesse definiert und Informationsabfragen in Form von parametrisierbaren SQL-Statements vorgehalten, die der Anwender abrufen kann. Allerdings können keine komplexen Entscheidungsprozesse modelliert werden. Es stehen lediglich Standardberichte (*Briefing Book*) und festvordefinierte Abfragemasken bereit.

Die Verwendung der vordefinierten SQL-Statements ist zu unflexibel und gibt den weniger versierten Anwendern ohne spezielle Kenntnisse der zugrundeliegenden Datenbankschemata keine Anpassungsmöglichkeiten. Meist setzen die SQL-Statements auf *Star-* oder *Snowflake-Schemata* auf und determinieren daher den Einsatz von *Data Warehouses* [Inmo92].
Durch die starke Fixierung der Entscheidungsprozesse und der SQL-Abfragen ergibt sich ein hoher Pflegeaufwand bei neuen Anforderungen. Des weiteren existieren keine Schnittstellen zur Integration externer Programme, so daß mit unterschiedlichsten Programmsystemen gearbeitet werden muß. Positiv hervorzuheben ist die oftmals sehr einfach und intuitiv zu bedienende Benutzerschnittstelle, da diese Systeme gerade für nicht versierte Anwender entwickelt wurden.

Workflowmanagementsysteme

Workflowmanagementsysteme dienen der Steuerung des Ablaufs einer Gesamtheit von Aktivitäten, die sich auf Teile eines Geschäftsprozesses oder andere Vorgänge beziehen. Dazu können externe Programme verschiedener Anwendungsdomains integriert werden. Der Datenfluß zwischen den einzelnen Funktionen ist durch das Workflowmanagementsystem ebenfalls gewährleistet.

Workflowmanagementsysteme sind für die Unterstützung im Einsatzbereich der Reportingplattform ebenfalls nicht geeignet, da zum einen die vollständige Modellierung sämtlicher Geschäftsprozesse für die Vorgangssteuerung nicht durchführbar ist [Poh+95] und zum anderen leider noch nicht die gewünschte Flexibilität vorhanden ist, so daß sie oftmals nur für stark strukturierte Arbeitsabläufe sinnvoll einsetzbar sind [KiUn94]. Insbesondere fehlt die mögliche Reaktionsfähigkeit des Anwenders auf Ausnahmen (*Exceptions*), die nicht im Vorwege modellierbar waren.

1.4 BEISPIELDOMAIN TRANSPORTLOGISTIK

Steigende Anforderungen im Tagesgeschäft sind insbesondere im Bereich der *Disposition* in Transportunternehmen anzutreffen, die zur Anhäufung zu verarbeitender Informationen führen. Es muß eine Vielzahl an Informationen dynamisch miteinander verknüpft werden, wobei dies nicht mittels der individuell entwickelten Anwendungsprogramme möglich ist.

Insbesondere im operativen Bereich einer *LKW-Disposition* nimmt die Komplexität der einzelnen Aufgabenbereiche stetig zu. Auch wenn zunehmend eine IT-Unterstützung in Form von Tourenoptimierungen und verbesserten Auftragsabwicklungssystemen gegeben ist, so sind trotzdem immer noch spezielle Ausnahmesituationen zu bewältigen, die nur von Spezialisten mit langjährigen Erfahrungen gelöst werden können. Sofern diese jedoch zu bestimmten Zeiten nicht verfügbar sind oder das Unternehmen verlassen, können die anfallenden Probleme nur teilweise oder mit erheblichem Aufwand bzw. Zeitverzögerung gelöst werden.
Beispielaktivitäten, die in einer LKW-Disposition vorkommen, sind u.a. die nachfolgenden Prozesse mit darin enthaltenen, beliebig geschachtelten Subprozessen:

Prozeß: Disposition	**Prozeß: Kapazitätsabgleich**	**Prozeß: Ersatzbeschaffung für ausgefallenen Zug**	**Prozeß: Ersatzbeschaffung für ausgefallenen Fahrer**	**Prozeß: Ersatzbeschaffung für im Stau aufgehaltenen Fahrer**
Überblick des Auftragsbestands zusammenstellen	Prüfung, ob Kapazitätsengpaß besteht	Reparaturabwicklung	Ermittlung Ausfallgrund Fahrer	Prüfen, ob Ersatzfahrer benötigt wird
Überblick der Ressourcen erstellen	Prüfung, ob Kapazitätsverlagerung möglich ist	Schadenauswertung	Feststellen, ob Transport verschoben werden kann	Ermittlung eines Ersatzfahrers
Zuordnung von Ressourcen zu Gütern	Ermittlung von Subunternehmern	Feststellen, ob Transport verschoben werden kann	Ermittlung eines Ersatzfahrers	Feststellen, ob ein Ersatzzug in der Nähe ist
Nutzung von Tourenplanungssystemen		Ermittlung eines Ersatzzugs	Feststellen, ob Ersatzzug in der Nähe ist	Ermittlung eines Subunternehmers
Ausnahmebehandlungen (siehe Prozesse rechts)		Feststellen, ob Ersatzzug in der Nähe ist	Ermittlung eines Subunternehmers	
		Ermittlung eines Subunternehmers		

Tabelle 1: Hauptprozesse mit Subprozessen

Diese Prozesse können nun in den verschiedensten Kombinationen auftreten, wobei beliebig viele Ausnahmesituationen denkbar sind, die nicht in den Standardabläufen vormodelliert wurden.
Zusätzlich können Abhängigkeiten zwischen den einzelnen Prozessen entstehen, neue Kombinationen von Prozeßteilen notwendig werden oder einzelne Subprozesse zu überspringen sein. Hinzu kommen weitere Ausnahmesituationen, die aufgrund von externen Ereignissen eintreten können:

1. Das Eintreffen neuer Aufträge, für die noch keine Ablauforganisation existiert.
2. Ein Disponent kann ausfallen, so daß plötzlich zusätzliche Aufgaben übernommen werden müssen (z.B. Import- und Exportabwicklung).
3. Die Geschäftsleitung benötigt kurzfristig Kennzahlen, die in dieser Form noch nicht vorlagen.

Einige dieser Exceptions werden in Tabelle 2 dargestellt.

Exception: Export	**Exception: Kennzahlen ergänzen**	**Exception: Neue Partnerschaften**	**Exception: LKW-Informationen bereitstellen**	**Exception: ...**
Exportpapiere zusammenstellen	Prüfung der vorhandenen Daten	Erfassung neuer Abwicklungsmodalitäten	Zugriff auf Werkstattsysteme	
Spezielle Anforderungen prüfen	Vorgehensweise festlegen	Erfassung neuer Abrechnungsmodalitäten	Zugriff auf Fuhrparkmanagement	
.....	Verknüpfung der Daten zu neuen Kennzahlen	Erfassung neuer Partner-Ressourcen	Zugriff auf Fahrerdaten	
	Alternativen testen	Abstimmung neuer Routen	Zugriff auf Transportdaten	

Tabelle 2: Mögliche Exceptions

Die Anwender arbeiten meist mit den speziell für ihren Anwendungsbereich entwickelten Programmen (z.B. Auftragsabwicklungs- und Dispositionsprogramme). Diese bieten kaum Flexibilität hinsichtlich sich ändernder Informationsbedürfnisse und umfangreicher Geschäftsprozesse mit der Notwendigkeit zur Ausnahmebehandlung. Die Unterstützung der Ladungszusammenstellung mittels Touren- und Routenoptimierungsprogrammen bietet weitere Vorteile, wobei aufgrund der Komplexität der Problemstellungen und der aktuellen Situationen (Staumeldungen, Baustellen, Brückensperrungen etc.) manuelles Nachbessern bzw. Verfeinern die Regel ist.

Sinnvoll ist deshalb, die anfallenden, auch arbeitsplatzübergreifenden, Geschäftsprozesse in einer flexiblen Art und Weise abzubilden und den Anwendern ein hohes Maß an Freiheit bei der Zusammenstellung der jeweils notwendigen Informationen zuzugestehen.

1.5 AUSBLICK AUF DIE ARBEIT

In den nachfolgenden Kapiteln wird die Entwicklung der Reportingplattform zur Unterstützung für das erfahrungsbasierte Problemlösen beschrieben. Aufgrund der hohen Anforderungen an das System (siehe *Kapitel 1.2 Anforderungen an die Reportingplattform (Benutzersicht)*) müssen unterschiedlichste Technologien der Informatik zu einem konsistenten Gesamtsystem zusammengefügt werden, was in dieser Konstellation so noch nicht realisiert wurde. Insbesondere die Kombination von EIS-Funktionalitäten in Form einer ergonomischen Benutzerschnittstelle mit einer auf Workflowmanagementfunktionalitäten basierenden Benutzerführung sowie der Verwendung von Konzepten aus dem Bereich Componentware zur dynamischen Änderung der Prozesse zur Laufzeit und dem fallbasierten Schließen für die Qualifizierung unterschiedlichster Informationsbausteine lassen die Komplexität bei der Systementwicklung erahnen. Durch die Zusammenführung der unterschiedlichen Konzepte werden die Nachteile einzelner Ansätze (EIS-Generatoren und Workflowmanagementsysteme) aufgehoben, da ausschließlich deren Stärken einfließen.

Das folgende *Kapitel 2 Basistechnologien der Reportingplattform* gibt eine Übersicht zu den einzelnen Basiskonzepten, deren Zuordnung zu den Verwendungsbereichen nachfolgend dargestellt wird:

Benutzeroberfläche
Die Benutzeroberfläche der Reportingplattform orientiert sich an den Konzepten der Gestaltung ergonomischer Benutzerschnittstellen (siehe *Kapitel 2.1 Konzepte zum Entwurf ergonomischer Benutzeroberflächen*) und Executive Information Systems (EIS) [Hann96] [Humm98] (siehe *Kapitel 2.3 EIS - Executive Information Systems*). Insbesondere die einfache Bedienbarkeit des Systems sowohl für den ungeübten User als auch den Poweruser ist eine grundlegende Anforderung. Die Spezifikation der Problemstellungen erfolgt in der Terminologie des Anwenders.

Basiskomponenten
Die Modellierung der einzelnen Komponenten für Informationsabfragen (*SQL*) und Prozeßbausteine (*Workflows*) erfolgt auf Basis des Design Patterns *Template* [Gam+95] (siehe *Kapitel 2.5.2 Das Design Pattern Template*). Hierdurch wird ein hoher Grad an Wiederverwendbarkeit erreicht und es können Komponenten unterschiedlichster Granularität, optimiert auf den jeweiligen Anwendungskontext, verwendet werden.

Verknüpfung / Komposition der Komponenten
Zur flexiblen Verknüpfung der *SQL*- und *Prozeßtemplates* werden Mechanismen aus dem Bereich Componentware [NiTs95] [Griff98] genutzt (siehe *Kapitel 2.5.3 Mechanismen komponentenbasierter Frameworks*). Die Entwicklung einer skriptbasierten Kopplung der Komponenten ermöglicht dem Anwender jederzeit, auch zur Laufzeit, auf Exceptions zu reagieren. Bei der benutzerinitiierten Kombination nicht zusammenpassender Komponenten wird der Anwender über Dialogboxen [Sche95] mit einbezogen, so daß ein robustes System entsteht. Die Ablaufsteuerung übernimmt ein komponentenbasiertes Framework (siehe *Kapitel 2.5.4 Konstruktion eines Komponentenframeworks mittels Templates*).

Spezifikation der Workflows
Die Modellierung der Entscheidungsprozesse erfolgt mittels Methoden der Geschäftsprozeß-modellierung [Sche94] (siehe *Kapitel 2.4 Workflowmanagement*). Die Kopplungsmöglichkeiten der Prozeßkomponenten werden mittels Spezifikationen aus dem Bereich Workflowmodellierung [Jab+97] gesteuert und überwacht (siehe ebenfalls *Kapitel 2.4 Workflowmanagement*). Für die Übergänge von einem Prozeß auf den nächsten werden Integritätsbedingungen zur Konsistenzerhaltung verwendet.

Einbeziehung vorhandener Lösungen
Die Beherrschbarkeit der Ähnlichkeit zur Verwendung von Analogien für die erfahrungsbasierte Problemlösung wird durch Mechanismen des *fallbasierten Schließens* [Kolo93] gewährleistet (siehe *Kapitel 2.6 Fallbasiertes Schliessen*). Das System unterstützt den Modellierer und den späteren Anwender beim Finden bereits existierender Lösungsbausteine. Dabei werden die gefundenen Lösungen bzgl. ihrer Relevanz zum spezifizierten Problem bewertet. Die Abspeicherung einer neuen Lösung ermöglicht das Anwachsen der für zukünftige Problemstellungen zur Verfügung stehenden Lösungsmenge.

Auf Basis dieser Technologien wird die Entwicklung der Reportingplattform realisiert (siehe *Kapitel 3 Entwicklung der Reportingplattform*). Zum besseren Verständnis werden zunächst Grundlagen der Reportingplattform dargestellt (siehe *Kapitel 3.2 Basismechanismen der Reportingplattform*). Eine Übersicht zu den verwendeten Mechanismen aus dem Bereich Componentware wird in *Kapitel 3.3 Verwendung von Componentwarekonzepten* gegeben. Eine kurze Einführung zur Entwicklung des Templatekonzepts wird in *Kapitel 3.4 Templates* gegeben. Hier schließen sich die detaillierten Beschreibungen der Entwicklung von SQL- und Prozeßtemplates an (siehe *Kapitel 3.5 SQL-Templates* und *Kapitel 3.6 Prozesstemplates*). Mechanismen aus dem Bereich des fallbasierten Schließens zur systemseitigen Unterstützung beim Wiederauffinden von bereits im System vorhandenen Problem-stellungen (SQL-Templates) werden in *Kapitel 3.7 Konzept zum Wiederauffinden ähnlicher Problem-beschreibungen* beschrieben.

Die Erläuterung verschiedener Implementierungsdetails schließt sich in *Kapitel 4 Implementierung* an. Nach einer kurzen Übersicht (siehe *Kapitel 4.1 Übersicht*) wird der Schwerpunkt auf die Beschreibung des Werkzeugs zur Entscheidungsunterstützung mit der Spezifikation einzelner Dialogabläufe und dem Wiederauffinden von Prozessen gelegt (siehe *Kapitel 4.2 Das DSS-Werkzeug*).

In *Kapitel 4.3 Entwicklung der Benutzerschnittstelle* wird anhand der Modellierungstools für SQL- und Prozeßtemplates die Umsetzung der in *Kapitel 3 Entwicklung der Reportingplattform* entwickelten Konzepte beispielhaft erläutert. Hieran schließt sich ein Sitzungsszenario mit dem DSS-Tool und der darin abgebildeten CBR-Funktionalitäten sowie eine Übersicht zu den einzelnen Werkzeugen der Analysekomponente an.

Eine Beschreibung der Architektur der Reportingplattform erfolgt in *Kapitel 4.4 Architektursichten der Reportingplattform.* Die Zusammenfassung der Erfahrungen bei der Nutzung der Reportingplattform bzw. der Implementierung wird in *Kapitel 4.5 Anwendungserfahrungen* gegeben.

Abschließend werden die Ergebnisse sowie ein Ausblick auf zukünftige Entwicklungen in *Kapitel 5 Weiterentwicklung* zusammengefaßt.

Das so entstehende System erfüllt die in *Kapitel 1.2 Anforderungen an die Reportingplattform (Benutzersicht)* gestellten Anforderungen. Die Einsetzbarkeit der Reportingplattform zur Entscheidungsunterstützung mit deren EIS- / und DSS-Werkzeugen wird anhand einer Beispieldomain aus dem Bereich der Transportlogistik gezeigt (siehe *Kapitel 4 Implementierung*). Die Verwendbarkeit in anderen Domains ist ebenfalls gewährleistet, da lediglich die Informationsmodellierung den Anwendungsbereich determiniert. Im Rahmen der Entwicklung der Reportingplattform wird ebenfalls gezeigt, daß einzelne Mechanismen des fallbasierten Schließens verläßliche Methoden bereitstellen, bei ähnlichen Problemstellungen bereits verwendete und im System verfügbare Lösungsbausteine zu finden und ggf. mittels Adaption so anzupassen, daß die neue Problemstellung gelöst wird. Eine hohe Flexibilität des Exception Handlings wird durch die Verwendung des Design Patterns Template und Mechanismen aus dem Bereich Componentware erzielt, so daß diese Methodologien als richtungsweisend für die Entwicklung zukünftiger Workflowmanagementsysteme angesehen werden können. Des weiteren wird gezeigt, daß ein hoher Grad an Wiederverwendbarkeit von bereits implementierten Sourcecodes durch die Verwendung des Design Patterns Template realisierbar ist.

2 BASISTECHNOLOGIEN DER REPORTINGPLATTFORM

In diesem Abschnitt erfolgt eine Beschreibung der verschiedenen Basiskonzepte, die bei der Entwicklung der Reportingplattform Verwendung finden.

2.1 KONZEPTE ZUM ENTWURF ERGONOMISCHER BENUTZEROBERFLÄCHEN

2.1.1 MUSE - Method for User Interface Engineering

Die Entwicklung der Benutzerschnittstelle eines interaktiven Systems erfordert bei Aufgaben- und Benutzerorientierung verschiedene generische Aktivitäten zur Dekomposition komplexer Gestaltungsfragen, die z.B. durch das Konzept *MUSE* unterstützt werden [Gorn94], [Quin95]. Die grundlegende Vorgehensweise wird im folgenden skizziert. MUSE reduziert die Entscheidungskomplexität durch Hervorhebung einzelner relativ umfassender Betrachtungsfelder, die eine jeweilige Sicht auf das zu entwickelnde Gesamtsystem bilden. Insgesamt besteht das MUSE-Konzept aus den vier folgenden Betrachtungsweisen:

Konzeptuelle Sicht
Die *Konzeptuelle Sicht* fokussiert im Wesentlichen die Funktionalität des interaktiven Systems. Dazu werden die sich aus der Aufgabenanalyse ergebenden Werkzeugfunktionen in die vier folgenden Subkategorien eingeteilt. Das Ergebnis der konzeptuellen Sicht ist eine Liste von Werkzeugfunktionen.

- Anwendungsfunktionen (zur Erledigung der eigentlichen Aufgabe).
- Steuerfunktionen (zur Handhabung des Werkzeugs).
- Adaptierfunktionen (zur Regelung des Dialogverhaltens und der Bildschirmdarstellung).
- Metafunktionen (zur Unterstützung des Benutzers bei der Anwendung des Systems).

Strukturelle Sicht
Die *Strukturelle Sicht* legt Dialogformen für die Verwendung der Werkzeugfunktionen auf die Objekte der Aufgabe fest. Hierzu werden die Grundformen Dateneingabe, Kommandodialog, Auswahldialog und Direkte Manipulation sowie Kombinationsformen (Maskendialoge) verwendet.
Dialogabläufe werden durch die *Dialogstruktur* der Benutzerschnittstelle in Form von Vorbedingungen beschrieben, die alle zulässigen Dialogabläufe im Sinne der Erreichbarkeit und Anwendbarkeit von Werkzeugfunktionen auf Objekte spezifiziert. Temporale Bedingungen bestimmen die Festlegung des logischen Ablaufs von Arbeitsschritten.

Konkretisierungssicht
Die konkreten Erscheinungsformen der ausgewählten Dialogformen werden durch die *Konkretisierungssicht* festgelegt. Für die Dialogform "Auswahldialog" können so z.B. die Interaktionsarten Menüs, Listboxen, Radiobuttons, Checkboxen und Funktionstasten genutzt werden. Bei den möglichen Ausprägungen der Interaktionsarten sind ergonomische Kenntnisse erforderlich, um die sinnvolle Gestaltung von Menüstrukturen (logische Gruppierungen, Reduktion des Menüs auf maximal 10 Einträge, Ausblenden oder Grauschalten nicht verfügbarer Funktionen usw.) und das Menüdesign (Piktogrammenü oder Textmenü) zu entwickeln. Die *Präsentationsstruktur* umfaßt sämtliche Designentscheidungen.

Realisierungssicht
Die Festlegung der ausgewählten Dialogboxen, Listboxen, Comboboxen usw. in Aussehen, Größe und Plazierung erfolgt in der Realisierungssicht. Hierzu wird die Wortwahl für die Beschriftungen der Menüs und Schaltflächen, sowie die Auswahl und Anordnung geeigneter Piktogramme vorgenommen. Diese Erscheinungsformen sollten nach dem "WYSIWYG"-Prinzip ("What You See Is What You Get") mittels Werkzeugunterstützung in prototypischer Vorgehensweise realisiert werden.

2.1.2 Die Werkzeug-Material-Metapher

Die *Werkzeug-Material-Metapher* ist eine objektorientierte Methode zur Analyse, dem Design und der Implementierung interaktiver Softwaresysteme [BuZü90]. Insbesondere der Gestaltungsaspekt für Benutzeroberflächen von Softwaresystemen steht hierbei im Vordergrund. Arbeitsgegenstände werden

hierzu in zwei Kategorien eingeteilt: Objekte auf denen gearbeitet wird (*Materialien*) und Objekte, die als Arbeitsmittel fungieren (*Werkzeuge*) [Bud+92, S.136]. Werkzeuge dienen der Manipulation und Umformung von Arbeitsgegenständen, den Materialien. Die *Aspekte* eines Materials bestimmen sämtliche Möglichkeiten, die sich ein Werkzeug zur Bearbeitung des Materials nutzbar machen kann. Durch die Kombinierbarkeit von Werkzeugen bei Vorhandensein eines verbindenden Aspekts können auch komplexere Arbeitssituationen entsprechend unterstützt werden, ohne daß eine Spezialisierung der Werkzeuge auf einzelne Materialien notwendig ist. Werkzeuge bilden aufgrund verschiedener Arbeitssituationen Materialien anderer Werkzeuge.

Modellierung mittels der Werkzeug-Material-Metapher

Es lassen sich die drei Klassen *Materialklasse*, *Werkzeugklasse* und *Aspektklasse* identifizieren.

Klasse Materialien:
Materialien sind die zu verändernden Objekte, wobei die Modifikationen nicht direkt auf den Materialien durchgeführt werden können, sondern immer nur mittels Objekten der Werkzeugklasse, die die Operationen der Materialobjekte nutzen können. Materialien enthalten keine Angaben über ihren Verwendungskontext. Ihre Operationen orientieren sich mittels Verwendung von Aspektklassen, an möglichen Formen des Umgangs mit ihnen, nicht an der Bereitstellung zu manipulierender Attribute.

Klasse Werkzeug:
Werkzeuge spiegeln die Arbeitsorganisation wider. In ihnen manifestieren sich die Erfahrungen, wie Materialien in effektiver Art und Weise durch Applikationen bearbeitet werden können. Werkzeuge halten Mechanismen bereit, um auf Materialien zu arbeiten. Des weiteren geben sie den Benutzern eine stetige Sicht auf den Bearbeitungsstand der Materialien. Gut entworfene Werkzeuge wirken transparent und geben dem Benutzer den Eindruck einer direkten Manipulation.
Es sollte die systemgestützte Führung vordefinierter Abläufe des Benutzers vermieden werden. Die Möglichkeit des schnellen Wechsels zwischen einzelnen Werkzeugen ist eine Zielsetzung, damit der kontinuierliche Arbeitsfluß nicht unterbrochen wird. Werkzeuge werden in eine *funktionale Komponente* und eine *Interaktionskomponente* unterteilt.

Klasse Aspekte:
Werkzeuge können Materialien für andere Werkzeuge sein, so daß sich der jeweilige Zustand nicht als Eigenschaft definieren läßt. Werkzeuge werden entworfen, um auf unterschiedlichen Materialien, allerdings nicht notwendigerweise auf allen, zu operieren. Auf Materialien können die unterschiedlichsten Werkzeuge arbeiten, also nicht nur ein materialspezifisches Werkzeug. Werkzeuge können die Aspekte unterschiedlicher Materialien hervorheben. Für jedes Werkzeug sind die hervorzuhebenden Aspekte der Materialien in *Aspektklassen* beschrieben. Eine Aspektklasse stellt eine Schnittstellenspezifikation zur Verfügung, die Materialien zu unterstützen haben, um Werkzeugen überhaupt die Arbeit zu ermöglichen.
Die Aspektklassen sind Schnittstellenklassen der Materialklassen. Dieses erfordert im Rahmen der Modellierung eine Mehrfachvererbung. Die Gesamtschnittstelle einer Materialklasse wird durch die Summe der einzelnen Aspektklassen gebildet, von denen sie erbt. Die Aspektklasse stellt einem Werkzeug sämtliche benötigten Operationen zur Verfügung, die zur Bearbeitung eines Materials notwendig sind. Eine Aspektklasse ist somit eine abstrakte Klasse, die den Kontext zur Interpretation einer Klasse als Werkzeug und anderer Klassen als passende Materialien für dieses Werkzeug festsetzt.
Auf der Benutzeroberfläche werden den Anwendern Werkzeuge und Materialien in einer der Arbeitssituation adäquaten Form (z.B. basierend auf der elektronischen Schreibtischmetapher) zur Verfügung gestellt. Die Werkzeug-Material-Metapher läßt die Manipulation nur über Werkzeuge zu. Da deren Verhalten jedoch für den Benutzer transparent sein soll, erhält dieser trotzdem den Eindruck einer direkten Manipulationsmöglichkeit.

Gestaltungsmerkmale / Umgang mit Werkzeugen

Zur Gestaltung von Werkzeugen werden folgende Kriterien verwendet [BuZü90, S.205 ff.]:
- Namen oder graphisches Symbol zur Aktivierung des Werkzeugs.
- Der Benutzer erhält während der Aktivierung des Werkzeugs die Sicht auf das zu bearbeitende Material.

- Angebot von Optionen zur zielgerichteten, zweckgerechten und angemessenen Handhabung des Werkzeugs. Hierdurch kann der Benutzer Einstellungen am Werkzeug vornehmen, die sich auf die Materialsicht und den Materialzustand auswirken.
- Anzeige des aktivierten Werkzeugs mit der aktuellen Werkzeugeinstellung und der Menge der gültigen Optionen.

Im Rahmen des Werkzeugeinsatzes kann es sinnvoll sein, die Arbeit mit einem Werkzeug zu unterbrechen und mit einem weiteren Werkzeug fortzusetzen. Der Wechsel zwischen Werkzeugen muß von dem Benutzer explizit vorgenommen bzw. zumindest bestätigt werden. Das "pausierende" Werkzeug kann später wieder verwendet werden. Als Orientierungshilfe für den Benutzer muß das Werkzeug Metainformationen ausgeben, die vor allem über die zum aktuellen Zeitpunkt zulässigen Aktionen informieren. Des weiteren sollten Werkzeuge Helpfunktionen zur Orientierungshilfe enthalten.

2.2 SQL - STRUCTURED QUERY LANGUAGE

SQL - Structured Query Language [Cham74, S. 249 ff.] [Codd71] bildet den de facto Standard für relationale Datenbanksysteme. SQL kombiniert die Vorteile der beiden theoretischen Anfragesprachen *Relationenalgebra* und *Relationenkalkül* und ist vor allem dazu geeignet, *conjunctive queries* auszudrücken [Abi+95, S.143 ff]. SQL deckt sämtliche Aspekte des Datenbankzugriffs, eingeschlossen die Datendefinition (*DDL*), die Datenmodifikation (*DML*) und die Viewdefinition (*Viewkonstrukt*) ab. Der Aufbau allgemeingültiger SQL-Anfragen wird durch die *SELECT-FROM-WHERE*-Klausel gebildet:

```
SELECT <zu selektierende Attribute>
FROM   <Liste der Relationennamen>
WHERE  <Bedingung>
```

Eine Beispielanfrage wäre: *Zeige alle zu transportierenden Fahrzeuge der Niederlassung "Hamburg".*

```
SELECT Niederlassung, Fabrikat, Modell, Fahrgestell-Nr.
FROM   Fahrzeuge, Transportaufträge
WHERE  Fahrzeuge.Fahrgestell-Nr. = Transportaufträge.Fahrgestell-Nr.  AND
       Transportaufträge.Niederlassung = 'Hamburg'   ;
```

In diesen Anfragen werden die Relationennamen selbst benutzt, um die Variablen zu bezeichnen, die die *Tupel* der korrespondierenden Relation durchwandern. Ein Relationenname zusammen mit dem, durch einen Punkt getrennten, Attributnamen referenzieren Tupelkomponenten. Das Schlüsselwort SELECT bildet den *Projektionsoperator* der Relationenalgebra. Das Schlüsselwort FROM entspricht dem *Kreuzprodukt* und WHERE beinhaltet die *Selektion*.

Sofern sämtliche Attribute der in der FROM-Klausel genannten Relationen ausgegeben werden sollen, kann * anstelle der Attributliste in der SELECT-Klausel benutzt werden. Die WHERE-Bedingung kann Konjunktionen, Disjunktionen, Negationen und eingebettete *SELECT-FROM-WHERE*-Klauseln enthalten. Durch die Verbindung von FROM- und WHERE-Klauseln erhält man den *equi-join*-Operator.
Falls mehr als eine Variable über dieselbe Relation variiert werden muß, so können Variablen in der FROM-Klausel eingeführt werden.

```
SELECT       R1.Attribut1, R1.Attribut2, R1.Attribut3
FROM         Relation1 R1, Relation1 R2
WHERE        R1.Attribut1 = R2.Attribut2      ;
```

Relationen werden in SQL durch den Relationennamen, die Attributnamen und die Skalartypen spezifiziert:

```
CREATE TABLE Fahrzeuge
( Fabrikat          CHARACTER [30]
  Modell            CHARACTER [30]
  Fahrgestell-Nr.   CHARACTER [17] ) ;
```

SELECT-FROM-WHERE-Blöcke können in verschiedenen Kombinationen variiert werden. So sind die Mengenoperationen *Vereinigung*, *Durchschnitt* und *Differenz* verfügbar.

Geschachtelte Relationen (*nested relations*) erlauben die Einbettung von SQL-Anfragen in WHERE-Klauseln anderer SQL-Anfragen: *Zeige alle Fahrzeuge, die nicht in Hamburg zum Transport stehen.*

```
SELECT Fabrikat, Modell, Fahrgestell-Nr.
FROM   Fahrzeuge
WHERE  Fahrgestell-Nr. NOT IN
       ( SELECT  Fahrgestell-Nr.
         FROM    Fahrzeuge, Transporte
         WHERE   Fahrzeuge.Fahrgestell-Nr. = Transportaufträge.Fahrgestell-Nr.
                  AND
                 Transportaufträge.Standort = 'Hamburg' )   ;
```

Durch die *Negation* kann die Mengendifferenz ausgedrückt werden. Die Verwendungsmöglichkeiten von Vergleichsoperatoren <, <=, >, >= gehen über den Umfang der theoretischen Sprachen hinaus. Das Vorhandensein der Vergleichsoperatoren und der Datentypen *Integer* und *Real* ermöglicht arithmetische Operationen. Erweiterungen werden ebenfalls durch die Einbeziehung von Aggregationsoperatoren gegeben.

Die Einschränkung einer auszugebenden Tupelmenge erfolgt durch die WHERE-Klausel. Eine weitere Einschränkung der durch die GROUP BY-Klausel spezifizierten Tupelgruppen erfolgt durch das Schlüsselwort HAVING, welches der GROUP BY-Klausel folgt [Ullm88, S.218]. Es können auf den Integer- und Real-Domänen die üblichen arithmetischen Operationen ausgeführt werden. Im Gegensatz dazu berechnen die Aggregationsfunktionen tupelübergreifend Summen (SUM), Durchschnitte (AVG) oder Anzahlen (COUNT). Durch die GROUP BY-Klausel können Aggregationsfunktionen auf disjunkte Teilmengen einer Tupelmenge angewendet werden.

2.3 EIS - EXECUTIVE INFORMATION SYSTEMS

Executive Information Systems (EIS) oder *Führungsinformationssysteme (FIS)* [Jahn94] [Rieg90], sollen das Topmanagement mit führungsrelevanten Informationen versorgen und es bei seiner Entscheidungsfindung unterstützen. Dazu werden entscheidungsrelevante Informationen aus unternehmensinternen und -externen Quellen zusammengeführt. Dies erfordert eine spezielle Präsentation bzw. Verarbeitung der Daten zum Lokalisieren von Abweichungsursachen und der Entwicklung zukunftsbezogener Aktionspläne [Moun96, S.63 ff.].

2.3.1 Komponenten von Executive Information Systems

EIS setzen sich aus unterschiedlichen Teilsystemen zusammen. In Abbildung 4 ist die Grobstruktur eines EIS dargestellt [Turb93, S.414].

- *Benutzerschnittstelle*: Über die Benutzerschnittstelle findet der interaktive Dialog mit dem Anwender statt.
- *Funktionen*: Funktionen beinhalten die Methoden für die benutzergerechte Darstellung und Analyse der Daten. Dieses sind die verschiedenen Werkzeuge zur Datenpräsentation (z.B. Tabellen- und Graphikwerkzeuge) und zur Datenauswertung (z.B. Prognose-, Planungs- und Diagnosewerkzeuge).
- *EIS-Datenbank*: Es werden Daten aus den verschiedensten Datenquellen extrahiert und in verdichteter Form in einer speziell aufbereiteten Datenbank, dem *Data Warehouse*, abgelegt.
- *Datentransfer*: Hierzu wird die Überführung und Transformation der Daten aus unterschiedlichsten Datenquellen spezifiziert.
- *Interne Daten*: Umfassen Daten aus den operativen Systemen und sonstigen Unternehmenseinrichtungen.
- *Externe Daten*: Daten aus dem Umfeld einer Unternehmung, die für den Geschäftsbetrieb von Interesse sind (z.B. Branchendaten von Marktforschungsinstituten oder aktuelle Wechselkurse).

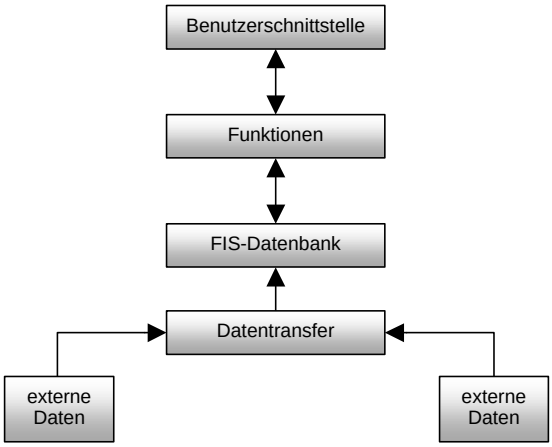


Abbildung 4: EIS-Struktur

2.3.2 Anforderungen an Executive Information Systems

EIS haben die generelle Aufgabe, dem Management als Navigationsinstrument zu dienen. Ein Ausschnitt des Informationsbedarfs ist in Abbildung 5 dargestellt [Korn94, S.21 ff.].

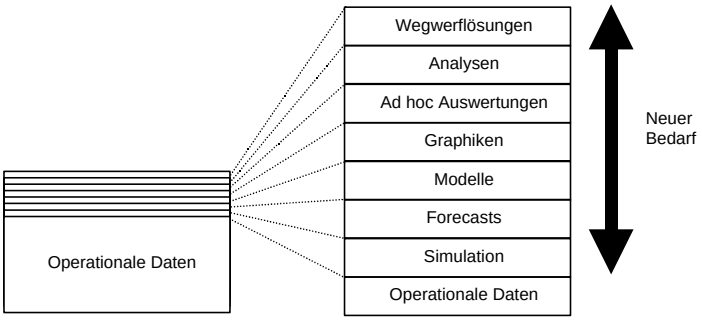


Abbildung 5: Profil des Informationsbedarfs

Nachfolgend sind Anforderungen an die Funktionalität eines EIS zusammengefaßt [Korn94, S.23].

- Zugriffsmöglichkeit auf Informationen sämtlicher Unternehmensbereiche sowie externer Daten in einer hochverdichteten und aussagestarken Form. Der gezielte Zugriff auf Detailinformationen muß ebenfalls durchführbar sein.
- Möglichkeit der Datenextraktion nach individuell formulierten Such- oder Selektionskriterien über *Drill-Down* oder Führung des Benutzers beim Navigieren. Integration in die Bürokommunikation.
- Trendanalyse und Disaggregation bereits in der Vergangenheit kumulierter Werte zur Dynamisierung des statischen Berichtswesens.
- Ein auf den kognitiven Stil des Management zugeschnittenes Exception Reporting mittels Traffic Lighting.
- Datenaktualität durch Anschluß an externe Datenbanken.
- Integration von Tabellen, Graphiken, Texten, Sprache und Bild als Ergänzung von Berichten.
- Verknüpfbarkeit von Informationen aus verschiedenen Quellen einschließlich externer Daten unter Nutzung komfortabler Benutzeroberflächen.
- Einfach zu bedienende Werkzeuge zur Datenanalyse und Simulation.
- Verwendung übersichtlicher, individueller Szenarien.

Die wichtigsten Leistungsmerkmale von EIS werden nachfolgend kurz dargestellt.
Datendarstellung und *-analyse*: Es werden zwei Arten der Informationsdarstellung unterschieden; mittels *Briefing Books* werden periodische Standardberichte generiert, deren fertige Berichtsformate sich bereits in einer Dokumentendatenbank befinden. Ebenfalls können einzelne Datenbankabfragen

fest hinterlegt sein und die zugehörigen Berichte werden bei Bedarf generiert.

Des weiteren existiert die Möglichkeit, dem Anwender die individuelle Zusammenstellung von Informationen zu überlassen (*ad hoc Abfragen* mit ständig variierenden Inhalten). Der Benutzer kann hierbei aus einem sog. Informationspool, in dem sich ein strukturiertes Angebot zahlreicher Informationen befindet, seine Anfragen auf einfache Weise selbst zusammenstellen. Das System übernimmt die Transformation des vom Anwender formulierten Informationsbedarfs in die Datenbankanfrage [Raab96, S.75 ff.].

Die Grundfunktion der Datenanalyse ist das *Filtern von Daten* nach den jeweiligen Bedürfnissen der Anwender*:*
Die *Extraktion* einer Datenmenge kann nach individuell formulierten Such- und Selektionskriterien erfolgen. Zusätzlich sind Informationen auf verschiedenen Aggregationsstufen anzubieten, durch die die Benutzer mittels sog. *Drill-Down-Techniken* von einer Aggregationsstufe bis hin zu den zugrunde liegenden Detaildaten navigieren können. Hierbei ist die Führungskraft auf Abweichungen von festgelegten Schwellenwerten hinzuweisen (*Exception Reporting*). Kritische Erfolgsfaktoren werden dazu mit individuellen Toleranzgrenzen versehen, die bei Über- oder Unterschreiten farblich gekennzeichnet werden (*Color Coding*). Bei der Verwendung spezieller Farben (negative Kennzahlen rot, positive Kennzahlen grün und in der Toleranzgrenze liegende Kennzahlen gelb) wird der Begriff *Traffic Lighting* verwendet [Serw96, S.35] (siehe Abbildung 6 [Leme97, S. 35]).

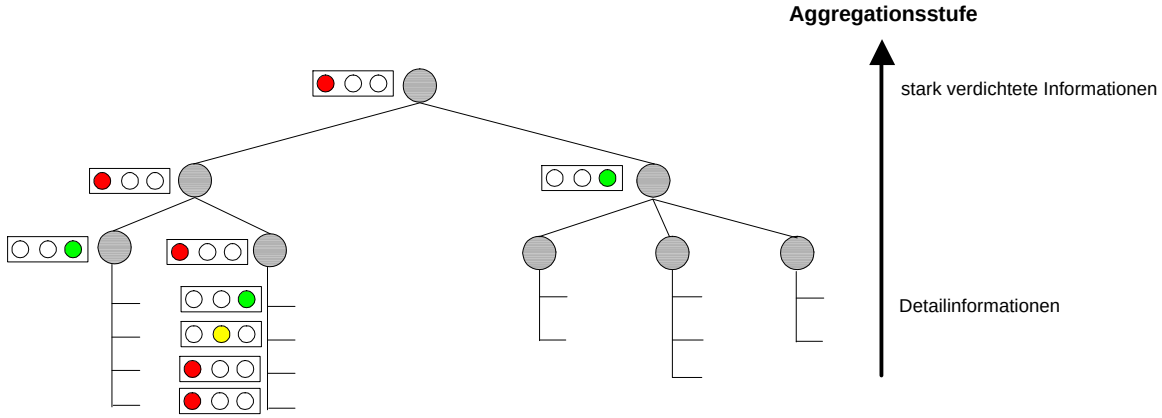


Abbildung 6: Exception Reporting mittels Traffic Lighting auf allen Aggregationsstufen

Durch eine *Methodendatenbank* können komplexere Datenanalysefunktionen integriert werden. Diese unterstützen den Anwender bei der Planung, Beurteilung und Bewertung von alternativen Vorgehensweisen (z.B. Trendrechnung, Simulation, Kennzahlensysteme, Sensitivitätsanalysen etc.). Für die Informationspräsentation bietet ein EIS neben den üblichen tabellarischen Übersichten vor allem die graphische Aufbereitung der Daten (z.B. die Darstellung zeitlicher Verläufe als Balken-, Säulen- oder Liniendiagramm, Kreuz- und Pivottabellen). Die Zielgruppe der nicht IT-versierten Anwender eines EIS impliziert eine hohe *Bedienerfreundlichkeit* der Benutzeroberfläche.

2.4 WORKFLOWMANAGEMENT

2.4.1 Übersicht

Ein *Prozeß* ist definiert als miteinander verbundene *Aktivitäten* oder *Teilprozesse* zur Bearbeitung einer *Aufgabe* oder einer einzelnen *Aktivität* [DIN96, S. 15 ff.]. Eine Aufgabe umfaßt die Spezifikation eines Ziels mit allen zur Erreichung des Ziels notwendigen Angaben. Die Bearbeitung der Aufgabe durch einen *Agenten* (Person oder Maschine) wird als Aktivität bezeichnet. *Geschäftsprozesse* in einem Unternehmen, als Untermenge der Prozesse, bestimmen die Kernfähigkeiten oder -kompetenzen, die im Hinblick auf den Kundennutzen als *betriebliche Abläufe* zu optimieren sind [Gai+94, S. 166] (siehe auch [Sche94] [FeSi95]).

Das *Ziel der Geschäftsprozeßmodellierung* ist die Prozeßanalyse zur Aufdeckung schlecht strukturierter Prozesse mit anschließender Optimierung unter Berücksichtigung der Unternehmensziele.
Geschäftsprozesse lassen sich in drei Kategorien einteilen [Ober96, S. 14 ff.]:

1. *Fallbasierte Geschäftsprozesse:* Fallbasierte Geschäftsprozesse haben eine konkrete sich wiederholende Struktur. Beispiele hierfür wären z.B. Bestellung oder Vertragserstellung.
2. *Ad hoc Geschäftsprozesse*: Ad hoc Geschäftsprozesse sind durch ihre einmaligen und spontanen Abläufe gekennzeichnet. Es handelt sich um Einzelvorgänge, wie z. B. die Aufwandsabschätzung für ein IT-Projekt.
3. *Generische Geschäftsprozesse:* Generische Geschäftsprozesse beschreiben fallbasierte Geschäftsprozesse auf einer abstrakten Ebene. Ein generischer Geschäftsprozeß wird von einer Reihe gleichartiger Prozesse abstrahiert.

Des weiteren können Geschäftsprozesse *strukturiert*, *unstrukturiert* oder *teilstrukturiert* sein. Ein Prozeß gilt als *komplex*, wenn er eine hohe Anzahl von Subprozessen aufweist und die logischen Beziehungen zwischen ihnen mehrheitlich parallel gegenüber sequentiell sind oder die einzelnen Arbeitsschritte starke Abhängigkeiten besitzen bzw. einen hohen Rückkopplungsaufwand aufweisen. Die Geschäftsprozesse werden nicht soweit spezifiziert, daß sie durch ein Workflowmanagementsystem unterstützt werden könnten.

Die Einführung einer weiteren Modellierungsebene zwischen Geschäftsprozessen und ausführbaren Workflows in Form eines *Arbeitsvorgangsmodells* ermöglicht eine strukturiertere Umsetzung. *Arbeitsvorgänge* sind strukturierte Abfolgen von Teilschritten. Hierbei steht der Arbeitsschritt, die Abfolge und Koordination der Arbeit, im Mittelpunkt. Bei der Modellierung ist zu untersuchen, wie die Dekomposition in Teilprozesse vorgenommen wird und wie die Prozesse miteinander zu verbinden sind. Das *Ziel der Arbeitsvorgangsmodellierung* ist die optimale Umsetzung und Anpassung der Geschäftsprozesse in einem konkreten Unternehmen. Die Klassifikation von Arbeitsvorgängen kann an ihrer Struktur vorgenommen werden.

- *Strukturierte Arbeitsvorgänge* können als gerichtete Graphen dargestellt werden. Die Aktivitäten entsprechen den Knoten und die Kanten spezifizieren die Reihenfolge der Teilaufgaben. Der Ablauf ist deterministisch mit festen Regeln. Alle Einflußfaktoren sind bekannt und werden festgelegt. Für die Beschreibung der Struktur werden einfache Kontrollstrukturen benötigt, die eine sequentielle, parallele, nebenläufige Durchführung ermöglichen. Die Formulierung von Schleifen ist ebenfalls möglich.
- *Teilstrukturierte Arbeitsvorgänge* haben keine eindeutige Struktur (Graph) wie die strukturierten Arbeitsvorgänge. Oft bilden sie Netze, in denen die Knoten strukturierte oder unstrukturierte Teilaufgaben abbilden, aber die Kanten nicht spezifiziert werden können. Das Ziel, das Ergebnis und der Weg sind teilweise bekannt. Die Steuerung kann durch Eingriffe des Anwenders erfolgen. Ein Workflowmanagementsystem verhält sich an diesen Stellen passiv. Wenn ein strukturierter Teilprozeß ausgewählt wurde, kann er wieder aktiv werden. Die Arten von Arbeitsvorgängen können weiterhin aufgeteilt werden in [Jab+97, S.90]:
  - *Arbeitsvorgänge mit bekannten Teilaufgaben, aber Freiheit in ihrer Ausführung*. Die Teilaufgaben sind bekannt, aber für ihre Abfolge gibt es mehrere zulässige Möglichkeiten. Hier sind Kontrollflußkonstrukte einsetzbar, die diese Flexibilität anbieten.
  - *Arbeitsvorgänge mit Teilaufgaben in situationsabhängiger Abfolge*. Hier sind auch die Teilaufgaben bekannt, aber ihre Notwendigkeit und ihre Abfolge hängt von der Situation ab. Die Abfolge von Aktivitäten kann nur implizit definiert werden.
  - *Unstrukturierte oder ad hoc Arbeitsvorgänge*. Sie entstehen eher spontan und lassen sich nur teilweise planen. Trotzdem kann ein Workflowmanagementsystem Vorschläge für das weitere Vorgehen unterbreiten.

Arbeitsvorgänge sind selten homogen, so daß sie nur zu einer Klasse gehören. Oft sind sie vermischt und enthalten Elemente aus unterschiedlichen Arten. Zur Modellierung von Arbeitsvorgängen existiert eine Vielzahl von Ablaufbeschreibungssprachen [Ober96, S.34]. Ein Beispiel für eine flußorientierte Beschreibung bilden die *ereignisgesteuerten Prozeßketten* (*EPK*) nach [Sche94] (siehe Anhang *Kapitel*

11.2 EPK - Ereignisgesteuerte Prozessketten - Notation). Zur Unterstützung der komplexen Modellierungsaufgabe vom Geschäftsprozeßmodell zum Workflowmodell existieren verschiedene Konzepte (Architektur integrierter Informationssysteme-ARIS [Sche94], Kommunikationsstrukturanalyse-KSA [Kral96] [Scho90], Semantisches Objektmodell-SOM [FeSi95])

Ein *Workflow* ist eine komplette oder teilweise Automation eines Geschäftsprozesses, wobei innerhalb der Ausführung gemäß eines Regelwerks Dokumente, Informationen oder Aufgaben zwischen Teilnehmern übermittelt werden [WfMC94]. Das *Ziel der Workflowmodellierung* (*WFM*) ist es, die Arbeitsabläufe in eine formale Beschreibung für ein Workflowmanagementsystem umzusetzen. Das bedeutet, daß die modellierten Modelle alle benötigten Informationen, die eine Ausführung eines Ablaufs erlauben, enthalten. Deshalb werden zusätzliche Informationen benötigt, die nicht im Arbeitsvorgangsmodell definiert werden. Einen detaillierten Überblick zu verschiedenen Modellierungsarten bietet [Stud98].

2.4.2 Das Grundverständnis des Workflows

Abbildung 7 präsentiert das Grundverständnis für fest strukturierte, vorgeplante Workflows als auch für ad hoc Workflows sowie Groupwaresysteme [Ambe96].

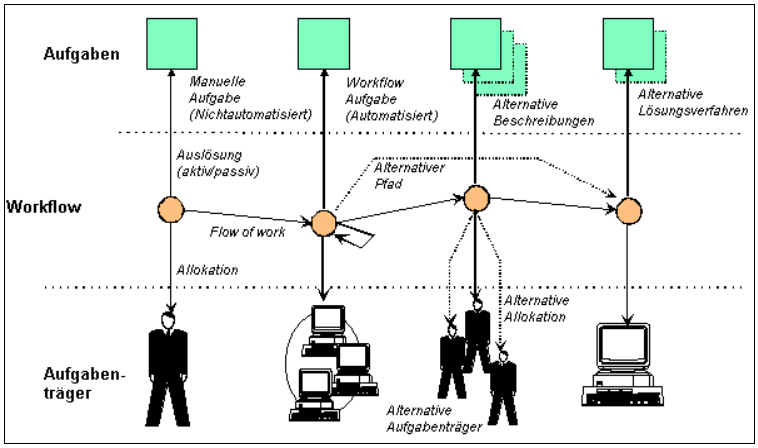


Abbildung 7: Das Grundverständnis eines Workflows

Laut der Definition eines Workflows kann er in mehrere Subworkflows zerlegt werden, um eine komplexe Aufgabe zu bewältigen. Es können so beliebig komplexe Netze modelliert werden. Elementare Workflows können von einem Aufgabenträger durchgeführt werden. An sie können Aufgaben angehängt werden, die wiederum von einem Aufgabenträger ausgeführt werden.

2.4.2.1 Arten von Workflows

Die Arten der Workflows leiten sich von den Arbeitsvorgangsarten ab, weil sie durch Workflows umgesetzt und abgewickelt werden sollen. Die Arbeitsvorgänge haben einen entscheidenden Einfluß darauf, was eine *Workflowsprache* und ein *Workflowmanagementsystem* zu leisten haben. Deshalb werden ad hoc Workflows (unstrukturierte), administrative Workflows (teilstrukturierte) und Transaktionsworkflows (strukturierte) unterschieden (siehe Abbildung 8 [Weiß96]).

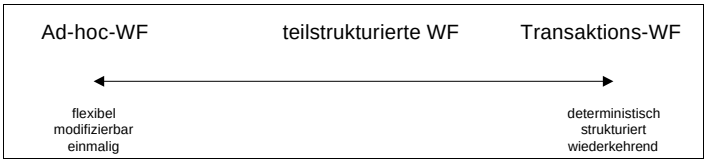


Abbildung 8: Klassen von Workflows

Weiterhin können Workflows anhand der Art der Strukturierung in *elementare* und *komposite* Workflows unterschieden werden.

Der elementare Typ liegt vor, wenn er nicht weiter in mehrere Workflows zerlegt ist. Dies ist die kleinste Einheit einer Tätigkeit und referenziert die Applikationen. Der komposite Workflow besteht

im Gegensatz dazu aus Workflows, die ihm zugeordnet sind. Der komposite Workflow kann aus elementaren oder kompositen Workflows bestehen. Dadurch entsteht ein Workflowbaum. Die Arten von Workflows werden bei der Workflowspezifikation festgelegt.

Ein Workflow kann an verschiedenen Stellen in einem Workflowbaum eingesetzt werden. Dann übernimmt er Rollen. Die Rollen sind *Topworkflow*, *Superworkflow* und *Subworkflow*. Ein Subworkflow existiert dann, wenn er in einem anderen Workflow benutzt wird. Diese Subworkflows sind einem Superworkflow zugeordnet. Ein Workflow, dem keine weiteren Superworkflows zugeordnet sind, heißt Topworkflow.

2.4.2.2 Zustände von Workflows

Ein Workflow befindet sich im Laufe der Zeit in verschiedenen Zuständen. So ein dynamisches Verhalten wird in Form von *Zustandsübergangsdiagrammen* (state transition diagrams) abgebildet. Dieses Diagramm besteht aus Kreisen, die einen Zustand eines Objekts widerspiegeln und aus Pfeilen, die erlaubte Zustandsübergänge definieren. An den Pfeilen kann ein Auslöser oder ein Guard oder eine Aktion [Rumb91] untergebracht werden. Die Zustandsdiagramme gehören z.B. bei der *Object Modeling Technique* (OMT) zum dynamischen Modell.

Die Modellierung der Zustände (aktiv, bereit, ...) von Workflows wird bei der Fehlerbehandlung (siehe *Kapitel 2.4.2.3 Fehler und Ausnahmen bei Workflows*) und bei der Verwendung von Kontrollflußkonstrukten [Jabl95b] genutzt und spezifiziert die gültigen Zustandsübergänge. Aufgrund des Terminierungszustands eines Workflows kann ein Workflowmanagementsystem ermitteln, ob ein Fehler aufgetreten ist oder nicht. Ein Fehler bewirkt, daß ein Workflow abnormal terminiert und damit die aufgerufene Aktivität mit einem negativen Ergebnis endet. Wenn bei der Workflowspezifikation die Zustände explizit deklariert werden, wird eine automatische Fehlerbehandlung möglich. Das System ist in der Lage, selbst zu entscheiden, ob und wie die Fehlerbehandlung durchgeführt wird [Jab+97, S.118]. Sonst muß der Benutzer die Fehlerbehandlung selbst durchführen, indem er z.B. eine Aktivität startet.

Die Semantik von Kontrollflußkonstrukten kann durch Zustandsübergangsdiagramme erläutert werden. Das Diagramm legt fest, welche Zustandsübergänge für Subworkflows erlaubt sind, die durch ein bestimmtes Kontrollkonstrukt miteinander in Beziehung gesetzt werden. Zum Beispiel bei einer seriellen Ausführung zweier Subworkflows *B* (bereit) und *C* (blockiert): wenn *B* in den Zustand *ausgeführt* übergeht, wird der Workflow *C* durch ein Workflowmanagementsystem in den Zustand *bereit* gestellt, nachdem er die Funktion *freigeben()* für *C* gestartet hat.

Das Zustandsübergangsdiagramm zeigt auch indirekt, welche Operationen einem Benutzer bzw. einem Workflowmanagementsystem bei der Verarbeitung eines Workflows zur Verfügung stehen. Solche Informationen müssen bei der Spezifikation eines Zustandsübergangsgraphen bei Workflows berücksichtigt werden.

Ein Schema der elementaren Zustände eines Prozesses ist in der Abbildung 9 zu sehen [WfMC94, S.24]. Die Übergänge zwischen den Zuständen werden durch verschiedene Operationen ausgelöst, die in einem Workflow festgelegt werden.

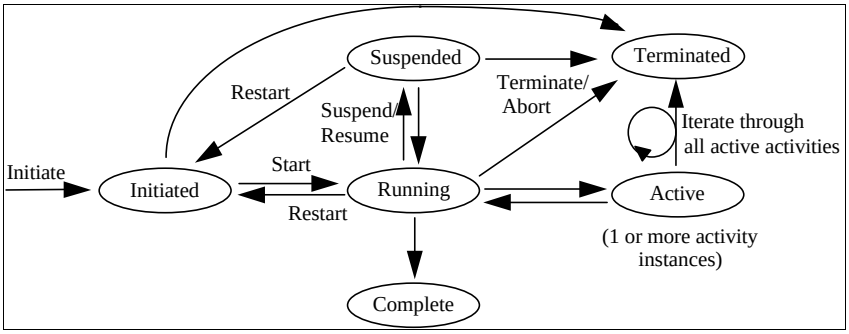


Abbildung 9: Zustandsdiagramm für einen kompositen Workflowprozeß

2.4.2.3 Fehler und Ausnahmen bei Workflows

Zu den *Ausnahmen bei Workflows* gehören z.B. spontane Änderung eines Arbeitsvorgangs, Nicht-Verfügbarkeit von Ressourcen, semantische Fehler beim Modellieren, Systemfehler und Transaktionsfehler. Hinzu kommt, daß die Realität nicht vollständig in einem Modell abbildbar ist.

Weil die korrekte Handhabung der Arbeitsvorgänge für die Workflowmodellierung wichtig ist, müssen bereits bei der Modellierung zusätzliche Informationen zur Fehler- und Ausnahmebehandlung in die Spezifikation von Workflows aufgenommen werden.
Im Workflowbereich treten verschiedene Fehlerarten auf. In [Jab+97, S.115] wird unterschieden zwischen:

- *Workflowmanagementsystemfehler*. Hier handelt sich um Systemfehler, die innerhalb eines Workflowmanagementsystems (z.B. in der Engine, Hard- und Softwarefehler) vorkommen.
- *Aktivitätsfehler* sind Fehler, die im Laufe der Abarbeitung eines Arbeitsvorgangs auftreten. Dazu zählen z.B. Fehler in einer Workflowapplikation (externes Programm), Zeitüberschreitung, Nichterfüllung der Pre- oder Postcondition einer Aktivität oder ein Benutzerabbruch.
- *Kommunikationsfehler*. Diese Klasse von Fehlern tritt bei einer verteilten Abarbeitung von Workflows (z.B. bei Client / Serverarchitekturen) auf.
- *Ablauffehler*. Bei diesen Fehlern werden die Differenzen zwischen dem modellierten Ablauf eines Workflows und konkreten Ausprägungen (z.B. Änderungen in der Abarbeitungsreihenfolge) festgestellt.

Workflowmanagementsysteme sollten in der Lage sein, alle diese Fehlerarten abfangen zu können. Die Workflowmanagementsystemfehler, Aktivitätsfehler und Kommunikationsfehler können überwiegend vom System selbst erkannt werden und eine automatische Ausführung auslösen. Ablauffehler verlangen eine manuelle Behandlung.

2.4.2.4 Aspekte eines Workflows

2.4.2.4.1 Überblick

Workflows können durch Aktivitäten, Ziele, Aktoren und Abhängigkeiten erfaßt werden. Deshalb besteht ein Workflowmodell aus einem *aktivitätenbezogenen*, *aktorenbezogenen*, *abhängigkeits-bezogenen* und *kausalen Aspekt*. Diese Aspekte dienen als Entwicklungsrichtung für das Workflow-modell, damit das Ziel der Orthogonalität und Modularität [Jabl95a, S.14]) erreicht wird. Deshalb läßt sich der Inhalt eines Workflows in verschiedene Aspekte (siehe Abbildung 10) aufteilen [JaBu96]. Die Liste der Aspekte ist im Prinzip offen und kann für spezielle Anwendungsgebiete erweitert werden [Jabl95a, S.18]. Auf diese Weise wird die Workflowmodellierung überschaubar und damit besser handhabbar.

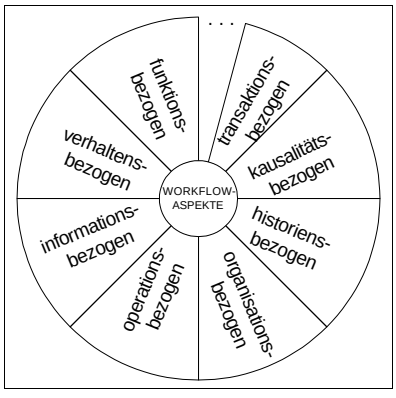


Abbildung 10: Aspekte von Workflows

Im folgenden werden die Inhalte des Funktions-, Informations- und Verhaltensaspekts eines Workflows näher beschrieben. Für die Beschreibung der anderen Aspekte sei auf [Jab95a] verwiesen.

2.4.2.4.2 Funktionsaspekt

Die Aufbaustruktur eines Workflows soll unterschiedlichen Qualitätskriterien genügen wie *Übersichtlichkeit*, *Wiederverwendbarkeit* und *Änderungsfreundlichkeit*. Diese Kriterien gelten ebenfalls für Arbeitsvorgänge und Geschäftsprozesse.
Übersichtlichkeit bedeutet, daß sich die Aktivitäten gruppieren und auf unterschiedlichen Abstraktionsstufen darstellen lassen. Wiederverwendbarkeit ermöglicht, die einzelnen Bausteine zusammenzufassen und erneut an anderer Stelle einzusetzen. Änderungsfreundlichkeit ist notwendig, da die Struktur eines Workflows ständigen Veränderungen unterliegt. Die Änderungen bei Arbeitsvorgängen fordern auch Anpassungen bei den Workflows, so daß neue Aktivitäten hinzugefügt, entfernt oder verfeinert werden müssen.

2.4.2.4.3 Informationsaspekt

Aktivitäten benötigen Daten die ein Workflowmanagementsystem zur Verfügung stellen muß. Es werden drei Datenarten unterschieden [WfMC94]:

- *Applikationsdaten* (*Workflow Application Data*). Dies sind Daten, die für eine Applikation notwendig und für ein Workflowmanagementsystem nicht zugänglich sind.
- *Workflowrelevante Daten* (*Workflow relevant Data*). Sie dienen einem Workflowmanagementsystem, um Zustandsübergänge wie Pre-, Post- oder Übergangsbedingungen sowie einen Workflowteilnehmer zu bestimmen.
- *Workflowkontrolldaten* (*Workflow Control Data*). Dies sind Daten, die nur dem Workflowmanagementsystem bekannt sind. Die gesteuerten Applikationen haben keinen Zugriff darauf. Workflowkontrolldaten sind z.B. interne Variablen, Prozeßspezifikationen (Workflow, Subworkflow usw.), Ablaufbeschreibungen (z.B. Ereignisse, Kontrollstrukturen) und Zustandsvariablen, die intern von einem Workflowmanagementsystem verwaltet werden. Sie sind die lokalen Variablen eines Workflows, die den Informationsfluß zwischen Subworkflows sichern.

2.4.2.4.4 Verhaltensaspekt

Dieser Aspekt spezifiziert, wann ein Subworkflow gestartet und ausgeführt wird. Er beschreibt die zeitliche und logische Reihenfolge von Prozessen. Durch diesen Aspekt werden die Aktivitäten miteinander in Beziehung gebracht und die *Kontrollflußlogik* eines Vorgangs vereinbart. Die vier am häufigsten vorkommenden Arten, das Verhalten eines Ablaufs auszudrücken, sind [Jab+97, S.193]:

- Darstellung durch Ereignisse.
- Darstellung durch Informationsfluß.
- Darstellung durch Petri-Netze [DeOb96].
- Darstellung durch Kontrollausdrücke [Jab+97].

Weiterhin wird unterschieden zwischen verschiedenen Typen der Modellierung der Kontrollflußlogik:

- *Explizit*. Bei diesen Netzen wird ein Kontrollflußkonstrukt als ein eigener Knoten präsentiert. So stellen z.B. ereignisgesteuerte Prozeßketten (EPK) die Operatoren (z.B. AND, OR, XOR) als eigene Knoten dar.
- *Implizit*. Die Ablauflogik wird durch die Übergangsbedingungen (*Transition Conditions*) definiert. Ein Beispiel für diese Modellierung sind Aktivitätenmodelle von FlowMark [Ley+95] oder der WfMC (Workflow Management Coalition).

2.4.3 Das Referenzmodell MOBILE für Workflowmanagementsysteme

Um Workflows zu spezifizieren bzw. um Workflowschemata intern darzustellen sind programmiersprachliche Konstrukte notwendig. Zur Spezifikation von Workflows wird in *MOBILE* die Sprache *MSL* (*Mobile Script Language*) verwendet [JaBu96]. Eine Zusammenfassung verschiedener Workflowsprachen wurde in [Riff98] erstellt.

Das Referenzmodell MOBILE stellt ein allgemeines Modell für Workflows dar und bietet verschiedene Aspekte, von denen nachfolgend der *abhängigkeitsbezogene Aspekt* dargestellt wird [Jabl95a]:

Abhängigkeitsbezogener Aspekt: Das Ablaufverhalten von Workflows ist durch die Ablaufkontrolle spezifiziert. Sie bestimmt, wann ein Workflow ausgeführt werden kann. Prinzipiell stehen zwei Möglichkeiten zur Spezifikation der *Ablaufkontrolle* zur Verfügung:

Präskriptive Ablaufkontrolle

serielle → (b, c)
Es fordert die strikte Einhaltung des Ablaufschemas [bc].

alternative if (cond(), b;~ cond(), c)
Hier entscheidet eine Ablaufbedingung *cond()*, welcher der alternativen Workflows [b] und [c] durchgeführt wird. Die Vorbedingung für diese Form der Ablaufkontrolle besteht darin, daß die *cond()* entscheidbar ist.

parallele || (b, c)
Die Workflows, die parallel ausgeführt werden, sind unabhängig voneinander. Der mögliche Ablauf lautet [bc] [cb].

iterative Ψ(b) == || (b, b, b, ...)
Eine Iteration liegt vor, wenn ein Workflow b mehrfach ausgeführt wird.

Die Abbruchbedingung kann die maximale Anzahl an Instanzen oder der maximale Zeitraum sein.

Deskriptive Ablaufkontrolle

Zeitliche Bedingung < (b, c) „Limitierung“

Die Ausführung eines Workflows wird durch die Ausführung eines anderen Workflows eingeschränkt. Für die Limitierung gelten folgende Vorschriften:
b kann ausgeführt werden, solange *c* noch nicht begonnen hat.
c kann ausgeführt werden, solange *b* noch nicht gestartet wurde.
c kann ausgeführt werden, sobald *b* abgeschlossen ist.
Der mögliche Ablauf lautet [], [b], [c] oder [bc].

Zeitliche Bedingung "Verzögerung" > (c, b)
Sie spezifiziert, daß der Workflow *c* nur dann gestartet werden kann, wenn *b* nie ausgeführt wird oder seine Ausführung abgeschlossen ist. Der Workflow *b* kann jederzeit ausgeführt werden.

Der mögliche Ablauf lautet [], [b], [c] oder [bc]. Bei der Verzögerung muß *c* warten, bis entweder *b* fertig ist oder Informationen darüber vorliegen, daß *b* nie ausgeführt wird. Bei einer Limitierung kann der Workflow *c* immer ausgeführt werden; bei einer Verzögerung muß *c* warten, bis entweder *b* fertig ist oder *b* nie ausgeführt wird.

Existenzabhängigkeit ⇒ (b , c)
Das Konstrukt beschreibt die existentiellen Zusammenhänge zwischen Workflows. Wenn b ausgeführt worden ist, muß c ausgeführt werden. Die möglichen Abläufe sind [], [c], [bc] oder [cb].

2.5 DESIGN PATTERNS / COMPONENTWARE / FRAMEWORKS

2.5.1 Übersicht

Entwurfsmuster (*Design Patterns*) sind im Gegensatz zu den in den nachfolgenden Kapiteln beschriebenen Komponenten keine Softwarebausteine, sondern Lösungsvorschriften, um typische, sich wiederholende komplexe Entwurfsprobleme mit Mitteln der Objektorientierung zu lösen. Entwurfsmuster erlauben es, statische und dynamische Strukturen sowie das Zusammenwirken verschiedener Komponenten in erfolgreich realisierten Lösungen auf neue Probleme zu übertragen, die im Rahmen der Softwareentwicklung auf verschiedensten Gebieten auftreten. Sie unterstützen die Entwicklung wiederverwendbarer Komponenten und Frameworks durch die Modellierung der Strukturen und Interaktionen der verschiedenen Teile einer Softwarearchitektur auf einem höheren Level als Sourcecodes oder objektorientierte Designmodelle, die sich auf individuelle Objekte und Klassen fokussieren.

Zur Beschreibung von Design Patterns sind unterschiedliche Formate verwendet worden (siehe [Bus+96], [CoSc95] und [Gam+95]) wobei nachfolgend die Spezifikation von [Gam+95] genutzt wird. Die nächste Generation von Frameworks wird eine große Anzahl von Mustern enthalten, wobei die Muster genutzt werden, um die Formen und Inhalte der Frameworks zu beschreiben. Idealerweise werden die Mustersysteme und Frameworks durch *Online-Pattern-Browser* integriert, die mittels Hypertextlinks ein schnelles Navigieren durch verschiedene Abstraktionsniveaus ermöglichen [Schm95, S.74]. So könnten mehrere Muster zur Lösung eines umfangreicheren Problems zusammengesetzt werden [Bus+96, S.87], indem sie z.B. hintereinandergeschaltet werden. Aus diesem Grunde ist die umfangreiche Beschreibung der Einsetzbarkeit bzw. der Verbindungsmöglichgkeiten zu anderen Mustern notwendig.

Komponenten einer *Componentware* sind speziell für die Wiederverwendung entwickelte Software-Bausteine [Lind96, S.12], die meist auf eine bestimmte Domain ausgerichtet sind und von unterschiedlicher Granularität sein können [NiMe95].

Jede *Komponente* stellt eine Gruppe öffentlicher Dienste zur weiteren Nutzung bereit. Um ein Anwendungsprogramm aus den verschiedenen Komponenten zusammensetzen zu können, muß der Entwickler das Rahmenprogramm selbst bereitstellen, welches die einzelnen Komponenten nutzt und zu einem kooperierenden System integriert. Oftmals übernimmt dies jedoch ein Framework, das die notwendigen Steuer- und Kooperationsmechanismen zur Verfügung stellt.

Sofern es sich um Komponenten handelt, die in einer objektorientierten Sprache implementiert wurden, bestehen sie aus Klassen, von denen eine spezielle Gruppe ihre Methoden zur Verfügung stellt, während andere Klassen nur zur Realisierung der Dienste dienen. Mit Hilfe von Componentwaresystemen können Komponenten hergestellt werden, die hierfür die entsprechenden Sprachen und Werkzeuge anbieten. CORBA [OMG96] ist z.B. eine sprachneutrale Spezifikation für objektorientierte Componentwaresysteme, wobei die *Interface Definition Language* (*IDL*) dazu dient, die Dienste der in den unterschiedlichsten Programmiersprachen implementierten Komponenten in der Form von Klassen zu spezifizieren.

Der Nutzer von Componentware soll mittels Plug & Play seine Anwendung zusammenstecken können, wodurch sein Anwendungsprogramm auf ein Minimum reduziert würde. Die Anwendungsentwicklung wird durch die Kombination der verschiedenen Komponenten bestimmt und umfaßt im Prinzip lediglich die Kooperation zwischen den Komponenten. Der Nachrichtenaustausch zwischen den einzelnen Komponenten kann nur funktionieren, wenn die Objekte identische Objektspezifikationen aufweisen und das gleiche Nachrichtenformat akzeptieren.

Im Gegensatz zu Klassenbibliotheken, deren Verwendung durch die Bildung anwendungsspezifischer Subklassen gekennzeichnet ist, sind die Komponenten der Componentware nur zur Endmontage geeignet. Das Verhalten kann vom Benutzer nicht verändert werden, außer durch Kompositionsänderung in Form von Hinzufügen, Austauschen oder Entfernen (s.u.).

Objektorientierte Klassenbibliotheken müssen stets mit den Sourcecodes ausgeliefert werden, damit sie dem Anwender höchstmögliche Flexibilität erlauben. Nur so können dann Veränderungen in der Vererbungshierarchie vorgenommen werden. Bei einem möglichen Versionswechsel ist der Benutzer der Bibliothek dann zum Kompilieren sämtlicher Anwendungen gezwungen. Componentware soll dagegen in binärer Form vollständig nutzbar und versionierbar sein.

Unter *Frameworks* versteht man grundsätzlich eine Menge kooperierender Klassen, die eine Wiederverwendung einer bestimmten Klasse von Software ermöglicht (z.B. graphische Benutzerschnittstellen) [Lew+95]. Ein Framework unterstützt eine Softwarearchitektur durch die Aufteilung des Designs in verschiedene abstrakte Klassen, wobei die Verantwortlichkeiten und das Zusammenwirken der Klassen geregelt ist. Der Entwickler paßt das Framework seinen Bedürfnissen an, indem er Subklassen bildet und Instanzen der Frameworkklassen miteinander verknüpft [Gam+95, S. 360]. Diese so angepaßten Komponenten sind in den steuernden Rahmen des Frameworks eingebettet. Diese steuernde Komponente des Frameworks wird auch die zentrale Steuerung im späteren Anwendungssystem behalten.

Frameworks können so auch als konkrete Realisierungen von Mustern gesehen werden, die die direkte Wiederverwendung von Entwurf und Code erleichtern. Der Unterschied zwischen Mustern und Frameworks besteht darin, daß die Muster in einer sprachunabhängigen Form beschrieben werden, während Frameworks i.a. für eine spezielle Programmiersprache bestimmt sind. Ebenso sind Muster kleinere Architekturelemente als Frameworks, da ein typisches Framework immer mehrere Muster enthält, aber nicht umgekehrt [Gam+95, S.28]. Des weiteren sind Muster weniger spezialisiert als Frameworks, die meist ein besonderes Anwendungsgebiet umfassen, während Muster die Lösung für ein verallgemeinertes Problem beschreiben.

Neben den klassenbasierten Frameworks werden derzeit insbesondere komponentenbasierte Frameworks entwickelt, da diese einige entscheidende Vorteile in bezug auf die Wiederverwendbarkeit und Entwicklungszeiten aufweisen. Nachfolgend wird die Definition dieses Begriffs abgeleitet:

Softwarekomposition ist die systematische Konstruktion von Softwareapplikationen aus Komponenten, die Abstraktionen einer besonderen Problemdomain implementieren [Nier95, S.4]. Eine *Softwarearchitektur* ist eine Beschreibung in der Art, wie ein spezifisches System durch seine Komponenten zusammengesetzt wird. Eine *generische Softwarearchitektur* ist eine Beschreibung einer Klasse von Softwarearchitekturen mittels Termen von Komponentenschnittstellen, Kompositionsmechanismen und den Regeln zur Steuerung der Softwarekomposition.

Unter einem *Komponentenframework* versteht man eine generische Softwarearchitektur zusammen mit einer korrespondierenden Sammlung von generischen Softwarekomponenten, die verwendet werden können, um flexible Applikationen zu realisieren, deren Architektur mit der generischen Familie konform gehen [Nier95, S.5].

In den folgenden Kapiteln wird ein Überblick zu den verschiedenen Implementierungskonzepten komponentenbasierter Frameworks gegeben.

2.5.2 Das Design Pattern *Template*

In [Gam+95] wird eine Vielzahl unterschiedlichster Entwurfsmuster in einer Art Katalog vorgestellt. Eine fundamentale Methode im Bereich der Wiederverwendbarkeit von Code bildet das Entwurfsmuster *Template* [Gam+95, S.329]. Ein Template (siehe Abbildung 11) definiert einen Algorithmus in Ausdrücken abstrakter Methoden, die von Subklassen überschrieben werden, um das konkrete Verhalten der Anwendung zu bestimmen. Durch die Spezifikation einiger Ausführungsschritte eines Algorithmus als abstrakte Methoden, wird eine Reihenfolge festgelegt, jedoch können die einzelnen Subklassen die einzelnen Schritte nach ihren Bedürfnissen variieren. Die invarianten Teile eines Algorithmus werden so genau einmal festgelegt und in den Unterklassen wird das variierende Verhalten implementiert. Um die Verdopplung von Code zu vermeiden, wird gemeinsames Verhalten von Unterklassen herausfaktorisiert und in einer allgemeinen Klasse plaziert.

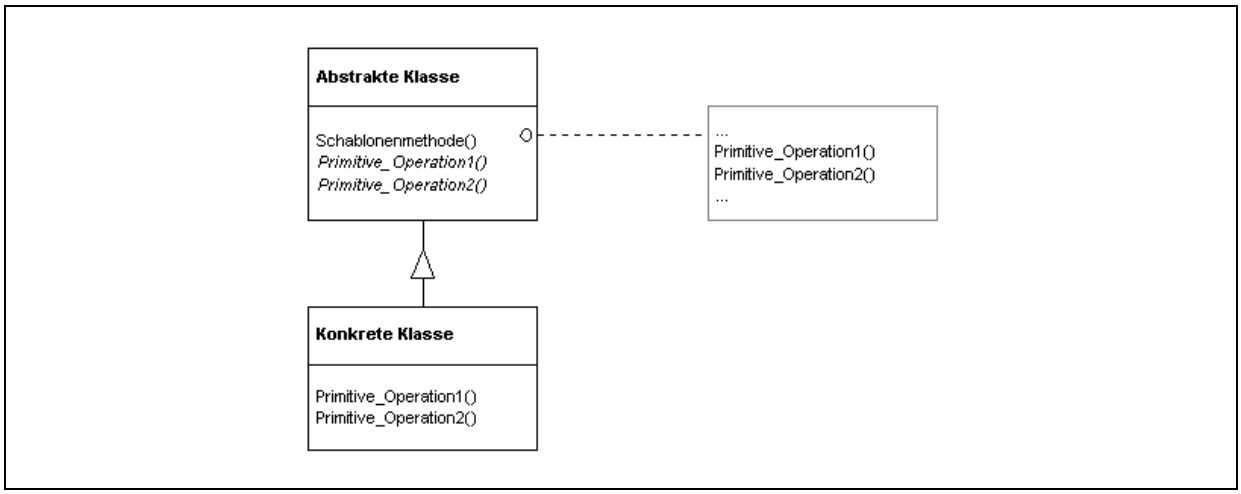


Abbildung 11: Design Pattern Template

Nachfolgend soll das Entwurfsmuster *Template* mittels verschiedener Kriterien genauer spezifiziert werden [Gam+95, S. 325 ff.].

Name und Klassifizierung: Template, Klassenverhaltensmuster.

Absicht: Definiert das Skelett eines Algorithmus in einer Operation und verschiebt einige Schritte in Subklassen. Subklassen können bestimmte Ausführungsschritte des Algorithmus redefinieren, ohne daß sich die Struktur des gesamten Algorithmus ändert.

Motivation: Das Template soll im Rahmen der Reportingplattform zum einen zur Generierung von SQL-Statements genutzt werden, so daß eine große Anzahl an wiederverwendbaren Basisstatements modelliert werden kann, die durch entsprechende Verfeinerung den zu modellierenden Informationsbedarf abdeckt. Hierdurch erhält man eine höhere Flexibilität, als wenn jedes Statement individuell spezifiziert werden muß.
Dasselbe gilt für Prozesse, die in Form von Prozeßtemplates definiert werden. Diese generischen Prozeßbausteine werden durch weitere Prozeßtemplates oder Elementarprozesse verfeinert.

Anwendbarkeit: Das Entwurfsmuster *Template* sollte in den folgenden Fällen genutzt werden:
Zur einmaligen Implementation der invarianten Teile eines Algorithmus, so daß die Subklassen das spezielle Verhalten variieren können.
Falls allgemeingültiges Verhalten zwischen Subklassen aufgeteilt und in einer allgemeinen Klasse zusammengefaßt werden soll, um das Duplizieren von Code zu vermeiden. So könnten erst die Unterschiede des existierenden Codes identifiziert werden, um diese dann in neue Methoden[1] aufzuteilen. Abschließend wird der herausfaktorisierte Code durch eine *Templatemethode* (oder *Schablonenmethode*) ersetzt, die die jeweiligen neuen Methoden aufruft.
Zur Kontrolle von Subklassen Erweiterungen: So kann eine Templatemethode definiert werden, die an speziellen Punkten den Aufruf von *Einschubmethoden* (*hook methods)* erlaubt.

Teilnehmer:
Abstrakte Klasse:
Definiert abstrakte primitive Methoden, die von konkreten Subklassen zur Implementierung der Ausführungsschritte des Algorithmus definiert werden.
Implementiert eine Templatemethode zur Spezifikation eines Algorithmusskeletts. Die Templatemethode ruft neben primitiven Methoden auch Methoden abstrakter Klassen oder Methoden anderer Klassen auf.

[1] Gamma et. al. [Gam+95] sprechen anstelle von Methoden von Operationen zur Manipulation von Objekten. Im folgenden wird jedoch in Anlehnung an die i.a. benutzte Terminologie im Bereich der Objektorientierung der Begriff *Methode* verwendet.

Konkrete Klasse:
Implementiert die primitiven Methoden zur Ausführung subklassenspezifischer Ausführungsschritte des Algorithmus.

Zusammenarbeit:
Konkrete Klassen bauen auf abstrakten Klassen auf, um die invarianten Schritte eines Algorithmus zu implementieren.

Konsequenzen:
Templates bilden eine fundamentale Technik für die Wiederverwendung von Code. Aus diesem Grunde werden sie insbesondere bei der Realisierung von Klassenbibliotheken verwendet, da sie allgemeingültiges Verhalten in die Klassen der Bibliotheken aufteilen.
Templates rufen die folgenden Methodenarten auf:

Konkrete Methoden (aus einer konkreten Klasse oder einer Clientklasse).

Konkrete Methoden einer abstrakten Klasse (z.B. Methoden, die i.a. für Subklassen nützlich sind).

Primitive Methoden (z.B. abstrakte Methoden).

Factory Methoden (siehe [Gam+95, S. 107]).

Einschubmethoden (hook methods), die ein Defaultverhalten unterstützen, welches von Subklassen bei Bedarf erweitert werden kann. Einschubmethoden tun oftmals nichts defaultmäßiges. Für Templates ist es wichtig genau zu spezifizieren, welche Methoden *hooks* sind, die überschrieben werden *können* und welche Methoden abstrakte Methoden sind, die überschrieben werden *müssen*. Damit abstrakte Klassen wiederverwendet werden können, müssen die Entwickler von Subklassen verstehen, welche Methoden entworfen wurden, damit sie überschrieben werden.

Bekannte Verwendungen:
Templates können aufgrund ihrer fundamentalen Konzepte in fast jeder abstrakten Klasse gefunden werden.

Verbundene Muster:
Factory Methode [Gam+95, S.107] und Strategy [Gam+95, S.305].

2.5.3 Mechanismen komponentenbasierter Frameworks

2.5.3.1 Begriffe

Nachfolgend wird anhand der verwendeten Terminologie ein kurzer Überblick zu dem Umfeld der *Komponentenframeworks* gegeben.

Ressourcen bilden Daten, die von Anwendungen interpretiert werden und meist in Dateien vorliegen [Pree97, S.14]. Die Komplexität und Struktur diese Daten kann stark variieren. So können einfache Wertelisten oder komplexere Daten zur Beschreibung einer Nutzerschnittstelle in Ressourcendateien abgelegt werden. Ebenso schränken sie weder die Art der Anwendung noch ihren Entwurf oder ihre Implementierung (konventionell oder objektorientiert) ein. *Skripte* und *Makros* bilden konzeptuell eine spezielle Ausprägung von Ressourcen [Pree97, S.15 ff.]. Sie unterscheiden sich konkret darin, daß die Ressourcen von Skripten mittels eines expliziten Interpreters verarbeitet werden, der getrennt von der Anwendung vorliegt. Die Ressourcendaten bilden bereits eine Art kleines Programm. Der Interpreter verarbeitet die Skript- oder Makroressourcen und initiiert Aktionen, die gerade auf den Kommandos dieser Ressourcen basieren. Man kann eine Skriptsprache als einfache Programmiersprache sehen, die z.B. durch eine *Attributgrammatik* [PiPe92] beschrieben werden kann.

Ein Framework umfaßt eine Sammlung verschiedener, individueller Komponenten mit einem definierten Kooperationsverhalten zur Erfüllung bestimmter Aufgaben [Pree97, S.7]. Einige der individuellen Komponenten werden so entworfen, daß sie austauschbar sind. Diese werden als *Hot Spots*

bezeichnet und ermöglichen einen flexiblen Einsatz des Frameworks, da sie sich anpassen lassen. Der Großteil des gewünschten anwendungsspezifischen Verhaltens sowie die Steuerung der Kooperation der Komponenten wird durch das Framework mittels sog. *Frozen Spots* determiniert. Mit diesem Begriff werden Komponenten bezeichnet, die nicht adaptiert bzw. ausgetauscht werden können.

White-Box-Frameworks bestehen aus einer Reihe unvollständig spezifizierter Klassen mit abstrakten Methoden. Für die Verhaltensänderung in White-Box-Frameworks wird Vererbung verwendet, und die Unterklassen der Frameworkklassen werden überschrieben [Pree97, S.19].

Die Anpassung des Frameworks erfolgt durch das Überschreiben von *Einschubmethoden* (hook methods des Template Patterns), die dann von anderen Modulen des Frameworks genutzt werden.

Die Konzepte zur Anpassung von *Black-Box-Frameworks* basieren nicht auf unfertigen Klassen, sondern auf einer Anzahl verwendbarer Komponenten, deren Komposition zu der gewünschten Änderung führt.

Die Verbindung zwischen den Komponenten kann über Skripte erfolgen, die die verschiedenen Komponenten zusammenbinden [Nie+91, S.1]. Skripte dienen als *Leim* (*Glue*), der zusammensteckkompatible Softwarekomponenten verbindet.

Ports sind Schnittstellen, über die das Binden der Komponenten erfolgt [Nie+91, S.3] ([Lew+95] nennt diese *Konnektoren*, s.u.). *Inputports* repräsentieren eine Menge von verfügbaren Diensten / Eigenschaften der Komponente. *Outputports* umfassen Dienste, die durch andere Komponenten bereitgestellt werden. Inputports passen i.d.R. zu mehren Outputports und damit zu mehreren Clients. Diese Ports mit mehreren Verbindungsmöglichkeiten werden *Multiports* genannt. Genauso kann es sinnvoll sein, daß Dienste nur für einen einzigen Client erbracht werden. In diesem Fall wird der Inputport s*ingular* genannt.
Ports haben einen Namen, einen Typ und eine Polarität [NiMey93, S.4]. Die Typen fallen in eine der drei folgende Kategorien [Nie+90, S.13]:

- *Value Links*: übergeben einen festen Wert an einen Inputport.
- *Reference Links*: stellen eine Client / Serverbeziehung zwischen zwei Komponenten her.
- *Export Links*: spezifizieren, welche Ports innerhalb eines Skripts für externe Clients zugreifbar sind.

Im Falle von Value Links kann der Porttyp entweder ein primitiver Datentyp oder ein komplexes Objekt sein. Reference Links unterscheiden sich von den Value Links, da die Objekte des Value Links *private* in bezug auf das empfangene Objekt werden, während ein Reference Link nur einen Handle auf ein *Shareable Object* übergibt. Der Typ eines Reference Links ist einfach die Referenz auf ein Objekt eines bestimmten Typs. Export Links kopieren den Typ eines internen Ports auf die Schnittstelle eines Skripts.

Ein Skript spezifiziert nun die Bindung zwischen einer Menge von Komponenten. Die Bindung von Ports kann entweder *optional* oder *obligatorisch* sein. Outputports müssen i.d.R. gebunden werden, um das Verhalten der Komponente zu komplettieren, während Inputports i.a. die Verfügbarkeit von Diensten widerspiegeln und deshalb ungebunden bleiben können. Des weiteren sollten *Defaultbindungen* unterstützt werden.

Die Schichtenstruktur einer Applikation mit *Scripted Components* ist in Abbildung 12 dargestellt [Pryc96]. Durch die Verwendung von Einschubmethoden können flexible Frameworks entwickelt werden [Pree97, S.57]. Sie spezifizieren die Stellen, an denen die Spezialisierung der Frameworks durchgeführt werden kann (Hot Spots). Elementare Klassen verwenden die gleichen Konstruktionsmuster, wie die abstrakteren und komplexeren Klassen, um eine entsprechende Flexibilität zu erhalten. Es unterscheidet sich nur die Granularität und Semantik der Einschubmethoden, was durch den jeweiligen Bezeichner für die Methoden ausgedrückt wird.

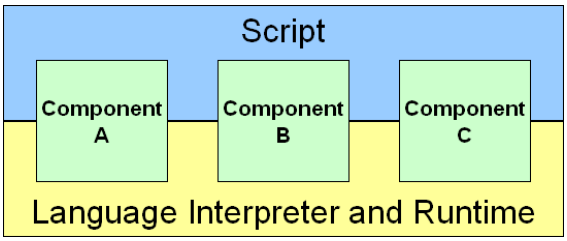


Abbildung 12: Layerstruktur einer Skriptapplikation

2.5.3.2 Skripte

Skripte können ebenfalls Komponenten sein, um sie wiederum in anderen Skripten als Komponenten zu verwenden. Eine Skriptkomponente kapselt eine Menge partiell gebundener Komponenten und determiniert die für weitere Skripte sichtbaren Ports (wobei diese nur renamed, nicht aber gebunden werden). Diese Ports *parametrisieren* sozusagen das Verhalten der Skripte / Komponenten [Nie+90].

Ebenso kann eine Skriptkomponente "Löcher" enthalten, die Platz für später einzubindende Komponenten freihalten. In diesem Fall repräsentiert der Port des "Lochs" sämtliche Dienste der fehlenden Komponenten.

Die Integration der verschiedenen Komponenten erfolgt mittels einer Datenmodellarchitektur, die als unabhängige Verbindung zwischen den Anwendungsdaten und sämtlichen darauf zugreifenden Komponenten dient [Lew+95, S.46]. Dazu werden *Viewkomponenten* (*view components*) definiert. Jede Viewkomponente ist mit einem Customizingwerkzeug ausgestattet, das dem Anwender die Spezialisierung entsprechend ihrer Anforderung erlaubt. Damit können visuelle Aspekte und das Verhalten der Daten spezifiziert werden. Die Datenmodellarchitektur dient als Application Programming Interface (API) zur Spezifikation der Kommunikationsprotokolle zwischen den Views und den zu bearbeitenden Daten.

Data Wrappers [Lew+95, S.48] werden benutzt, um Daten von einer Komponente zur nächsten zu transportieren, wobei dies in formatunabhängiger Art und Weise geschieht. Data Wrappers können bzgl. der enthaltenen Daten abgefragt werden und sie können die Daten konvertieren. Komponenten können eine oder mehrere Zustandsvariablen haben [Lew+95, S.252]. Diese können an Konnektoren gebunden werden, wie aktuelle Parameter an formale Parameter in einem Prozeduraufruf. Hierdurch kann der Programmierer definieren, wie Informationen durch die Verbindungen fließen. Konnektoren definieren die Semantik der Parameterübergabe für jede gebundene Zustandsvariable (eingehende-, ausgehende und ein-und-ausgehende-Konnektoren). So erhält z.B. eine Zustandsvariable eines eingehenden Konnektors den Wert eines ausgehenden Konnektors. Werteübergabe zwischen nicht passenden Konnektoren führt zu einem Fehler und muß beim Aufruf überprüft werden. Transferfunktionen runden das Datenflußmodell ab, indem sie definieren, wie sich die Werte der Zustandsvariablen ändern, während sie "durch die Komponente fließen".

Die Essenz des Scripting ist das *Verleimen* von Elementen zu einem Großen, z.B. durch die Komposition von Softwarekomponenten zu ausführbaren Applikationen. Für das Scripting sind neben den beiden Konzepten Vererbung und Nachrichtenversand flexiblere Kompositionsprinzipien notwendig, wie Ereignisse und Trigger [Nie+90, S.14]. Objekte können Ereignisse produzieren und andere Objekte (*sensors*) warten auf Ereignisse und reagieren durch Triggern einer Sequenz von Aktionen und weiteren Ereignissen.

Für den Entwurf von komponentenbasierten Frameworks sind deshalb Abstraktionen auf einem höheren Level notwendig, die in der Form einer *Composition Language* zu entwickeln sind [NiMe95, S.148].

2.5.3.3 Composition Language

Komponenten fungieren auf der Ebene der Komposition (Frameworklevel), während Objekte auf dem funktionalen Level (Applikationslevel) wirken [Nier95, S.7]. Die Schnittstellen sind deshalb unter-

schiedlich, da unterschiedliche Ziele verfolgt werden. Objektschnittstellen müssen den berechenbaren Notwendigkeiten genügen und Komponentenschnittstellen werden entworfen, um den Entwurf flexibler Softwaresysteme zu ermöglichen.

Zur Unterstützung der Entwicklung *offener Systeme* muß eine Composition Language in einer ersten Näherung folgende Spezifikationen unterstützen:
- Systeme als Komposition unterschiedlicher Komponenten.
- Komponentenframeworks mit der Spezifikation von Standardschnittstellen, Protokollen und Verhalten.
- Abbilden von Schnittstellen auf bestehende Komponenten, die ggf. mit einer anderen Programmiersprache implementiert wurden.

Hieraus lassen sich die drei folgenden Anforderungen an eine Composition Language ableiten [NiMe95, S.151]:
- *Offene Topologie*: offene Anwendungen sind in der Regel *nebenläufig* und *verteilt*.
- *Heterogenität*: Anwendungen können auf unterschiedlichen Hard- und Softwareplattformen ablaufen
- *Evolutionär*: Anforderungen sind im voraus nicht vollständig zu fixieren, so daß eine flexible Architektur notwendig ist, um sie an Veränderungen leicht anzupassen.

Diese Charakteristiken motivieren nun die folgenden technischen Anforderungen an eine Composition Language:
- *Kapselung*: damit die hohe Komplexität handhabbar ist und eine flexible Architektur unterstützt wird, sind Objekte und Komponenten notwendig.
- *Objekte als Prozesse*: Objekte können aktiv oder passiv, lokal oder remote, einfach oder zusammengesetzt sein. Aber alle Objekte können als Prozesse angesehen werden.
- *Komponenten als Abstraktionen*: Komponenten sind (möglicherweise higher order) Softwareabstraktionen, die auf verschiedenste Arten zu Applikationen zusammengestellt werden.
- *Zusammensteckkompatibilität*: ein Typmodell für Objekte und Komponenten muß entworfen werden, damit Bindungen und Kompositionen unterstützt werden.
- *Formales Objektmodell*: es wird ein Standardobjektmodell benötigt, um den Gap zwischen Komposition, Sprache und Implementation der Komponenten auf der einen Seite und higher level Kompositionswerkzeugen auf der anderen Seite zu überbrücken.
- *Skalierbarkeit*: die Nutzung der Composition Language sollte von kleinen bis zu großen Systemen skalierbar sein und von hoch dynamischer Verwendbarkeit mit unvollständigen Typinformationen bis zu einer kompilierten und optimierten Nutzung einsetzbar sein.

Technologisch gesehen umfaßt ein formales Modell eine Anzahl unterschiedlicher Funktionen:
- *Kommunikation und Bindung*: das Objektverhalten besteht aus dem Austausch von Nachrichten; die Funktionalität von Komponenten wird durch das Binden formaler Parameter an Werte instanziiert. Objekte und Komponenten können ebenfalls Werte sein.
- *Nebenläufigkeit*: eine Applikation ist eine zusammenhängende Komposition von Objekten.
- *Auswahl*: ein Objekt unterstützt typischerweise eine Schnittstelle, die aus einer Auswahl verschiedener Dienste besteht. Eine Komponente kann aus einer Vielzahl unterschiedlicher Arten zusammengesetzt sein.
- *Abstraktion*: Objekte und Komponenten sind abstrakte Entitäten deren Verhalten und Funktionalität nur über ihre Schnittstelle zugreifbar ist.
- *Instanziierung*: Objekte können dynamisch instanziiert werden, so daß es möglich sein muß, neue Namen für Objekte und ihre Kommunikationswege zu generieren und diese Namen an existierende Objekte weiterzugeben.

Im Rahmen eines Ansatzes zur wissensbasierten parallelen Programmierung [Dec+94] unterstützt die Entwicklungsumgebung einen Anwendungsprogrammierer dabei, ein Algorithmusskelett zu finden, das gut zur Lösung eines bestimmten Problems geeignet ist und dieses dann entsprechend zu ergänzen.

Ein ähnlicher Ansatz kann mittels Design Patterns realisiert werden [Nie+96, S.9]:

- Die Composition Language kann verwendet werden, um *Design Pattern Component Templates* zu spezifizieren, die lediglich an applikationsspezifische Klassen und Komponenten gebunden werden müssen.
- Mittels der Composition Language als Glue sollte es möglich sein, Teile von Applikationen oder vollständige Applikationen durch die Komposition von Softwarekomponenten zu kreieren. Applikationen bleiben flexibel, da sie es erlauben, Komponenten zu modifizieren oder zur Laufzeit auszutauschen [Nie+97, S.2]. Spezielle Skriptsprachen und 4GLs unterstützen unterschiedliche Arten von Glue. Glue kann entweder hoch strukturiert sein und eine bestimmte Art der Architektur für die Applikationskomposition widerspiegeln, oder er ist unstrukturiert.

Strukturierter Glue beschränkt die Flexibilität durch Limitierung der Möglichkeiten, wie Komponenten konfiguriert werden. Allerdings unterstützt er durch die explizite Darstellung der Architektur die Wartbarkeit und das Verständnis des Systems. *Unstrukturierter Glue* ist sehr flexibel, jedoch schwer wartbar. Eine Composition Language wird deshalb eine Kombination einer Beschreibungssprache für eine Architektur und einer Skriptsprache sein.

Der Glue ist verantwortlich für eine korrekte Nachrichtenübermittlung zwischen den Komponenten [Nie+97, S.5]. Sollte das Datenformat einer Komponente nicht dem erwarteten Datenformat einer anderen Komponente entsprechen, so ist eine Transformation durch den Glue zu gewährleisten. Dies bedeutet, daß der Glue eine Reihe von Datenformaten unterstützen muß und in der Lage ist, eine Transformation vorzunehmen. Neue Datenformate müssen hinzugefügt werden können und der Benutzer muß in der Lage sein, spezielle Komponenten einzufügen, die die Transformation durchführen.

Connectors [DuRi97] sind Runtime-Entitäten, die eine Menge von Regeln definieren, wie Objekte reagieren und miteinander über externe Nachrichten interagieren [Nie+97, S. 5]. Sie verändern das überwachbare Verhalten ohne daß die Objekte selbst verändert werden müssen. Deshalb können Connectors als eine Art higher level Glue zur Synchronisation und Komposition von Objekten angesehen werden. Hierdurch sind sie eine der Hauptabstraktionsmechanismen zur Spezifikation eines Architekturglues für umfangreiche Applikationen. Sie sind einfach zu integrieren, sofern die Composition Language *message interceptors* unterstützt.
Ein weiterer wichtiger Schritt für eine standardisierte Kommunikation zwischen Komponenten ist die Nutzung von DCOM oder CORBA-Funktionalitäten [Nie+96, S.11].

2.5.4 Konstruktion eines Komponentenframeworks mittels Templates

In diesem Abschnitt soll in Anlehnung an [Pree97] eine Vorgehensweise zur Nutzung von Design Patterns beim Entwurf von komponentenbasierten Frameworks dargestellt werden.

Abstrakte Klassen ermöglichen, daß andere auf ihnen basierende Klassen bereits implementiert werden können [Pree97, S.26], ohne daß die abstrakte Klasse vollständig spezifiziert wurde. Die entstehenden Softwarekomponenten hängen so von dem Protokoll der abstrakten Klasse ab. Es besteht eine *abstrakte Kopplung*. Diese Komponenten können problemlos mit Instanzen von allen zukünftigen Nachfolgern der abstrakten Klasse zusammenarbeiten, ohne daß eine Änderung oder gar ein Neukompilieren notwendig ist.

Zur Implementierung dieser Komponenten werden Referenzvariablen genutzt, die als statischen Typ den Typ der Klasse, auf die sich die Komponente bezieht, besitzen. Grundlage der Zusammenarbeit von Komponenten und Instanzen der Nachfolgeklassen ist *Polymorphismus*. Durch dynamisches Binden können die Instanzen jeweils ihr eigenes Verhalten bestimmen.

Die Problematik besteht nun darin, geeignete Abstraktionen zu finden, damit Softwarekomponenten ohne Kenntnis der Einzelheiten konkreter Objekte miteinander interagieren können.

Zur Wiederverwendung von komponentenbasierten Frameworks sollten diese generisch angelegt sein, damit sie leicht an neue Anforderungen angepaßt werden können. Ein Ansatz zum Erreichen dieser

Flexibilität ist die Verwendung von Einschubmethoden [Pree97, S.39]. Einschubmethoden (hook methods) können als Platzhalter oder flexible Hot Spots betrachtet werden, die durch komplexere Methoden aufgerufen werden. Templates definieren abstraktes Verhalten oder einen generischen Kontrollfluß sowie die Interaktion zwischen Objekten. Sie sind Frozen Spots, da sie in ihrer Struktur nicht veränderbar sind. Mittels Einschubmethoden kann durch Überschreiben an vorgegebenen Einschubstellen das Verhalten des Objekts geändert werden, ohne daß der Sourcecode der zugehörigen Klasse geändert werden muß. Hierfür ist das dynamische Binden der Einschubmethoden notwendig.

Sollten sich Template- und Einschubmethode in einer Klasse befinden, so kann Verhalten nur verändert werden, wenn zusätzliche Unterklassen definiert werden [Pree97, S.42]. Dies behindert jedoch Anpassungen zur Laufzeit, da die Subklassen mit der speziellen Klasse nicht abstrakt gekoppelt sind. Aus diesem Grund sollen die Klassen der Einschubmethoden mit der Klasse der Templatemethode abstrakt gekoppelt sein. Für eine Anpassung werden dann Komponenten mit konkreten Implementierungen für die Einschubmethoden eingefügt.

Im Vergleich von Einschubmethode und der sie aufrufenden Templatemethode ist die Einschubmethode der elementarere Teil. Jedoch kann die Templatemethode in einem anderen Kontext genauso als eine Einschubmethode einer anderen Templatemethode auftreten.
Idealerweise müssen nur die elementarsten Einschubmethoden überschrieben werden, damit das Verhalten der umfassenden Templatemethode beeinflußt wird. Dies wird als Prinzip der *Vererbung enger Schnittstellen* (*principle of narrow inheritance interface*) bezeichnet. Hierbei ergeben sich jedoch einige Zielkonflikte.

Durch einen guten Entwurf von Klassenschnittstellen für Frameworks sollte ein ausgeglichenes Verhältnis zwischen der gewünschten Flexibilität und dem Anpassungsaufwand der Klassen erhalten werden. Extrem wenig Anpassungsaufwand fordert eine Klasse mit mächtigen Templatemethoden und wenigen Einschubmethoden. Die Anpassung des Verhaltens erfolgt durch Überschreiben der Einschubmethoden oder Teilen davon in Unterklassen. Allerdings kann die Mächtigkeit der Templatemethoden die Flexibilität der ganzen Oberklassen einschränken.

Die Klasse, die die Einschubmethoden enthält wird mit **H** (hook) bezeichnet, die Klasse, die die Templatemethoden enthält wird **T** genannt. Hierdurch parametrisiert die Einschubklasse quasi ihre Templateklasse. Dies ist eine kontextabhängige Unterscheidung, unabhängig von der Komplexität beider Klassen. Durch die Kombination der beiden Klassenarten können nun die entscheidenden Konstruktionsprinzipien zur Gestaltung und Flexibilität abgeleitet werden [Pree97, S.46]. Beide Klassen können von beliebiger Komplexität sein, so daß auch die Konstruktionsprinzipien beliebig skalierbar sind.

Vereinigungsmuster und Trennmuster sind grundlegende Kombinationsmöglichkeiten beider Klassen. Bei einer Vereinigung beider Klassen zu **TH** können Anpassungen nur durch Vererbung erfolgen, was einen Neustart der Anwendung determiniert. Die Trennung der beiden Klassen **T** und **H** entspricht der abstrakten Kopplung der Klassen und das Verhalten eines **T**-Objekts kann durch Komposition, also das Einfügen von **H**-Objekten, auch zur Laufzeit und ohne Neustart, verändert werden.

Bei kompilierten Sprachen gilt, daß dynamisches Binden durch die Betriebssystemumgebung unterstützt werden muß. Zusätzlich müssen zur Laufzeit Metadaten über Objekte und Klassen gehalten werden, um weitere **H**-Objekte in **T**-Objekte einzusetzen [Pree97, S.48].

Eine Templateklasse kann auch Nachfolgerin einer Einschubklasse sein. Als Extremfall sind beide Klassen vereinigt. Nicht sinnvoll ist es, Einschubklassen als Nachfolger ihrer Templateklassen zu definieren, da Templatemethoden i.a. konkreter sind als Einschubmethoden, denn diese werden erst in Unterklassen überschrieben.

Den rekursiven Kompositionen ist gemein, daß mit ihnen Graphen aus miteinander verbundenen Objekten erstellt werden können. Nachrichten werden innerhalb dieses Objektgraphen weitergeleitet. Durch die rekursive Trennung von Template- und Einschubklassen wird gegenüber der einfachen

Trennung mehr Raum für die Anpassung durch Komposition erhalten, da gerichtete Graphen über Objekte gebildet werden können.

Die **TH**-Kette - a) in Abbildung 13 (in Anlehnung an [Pree97, S.49]) könnte ebenso eine Kette von **T**-Objekten und einem abschließenden **H**-Objekt sein. Ebenso ist es möglich, daß die Knoten der komplexeren Kette **TH**-Objekte sein könnten – b). Sofern **T**- oder **TH**-Objekte beliebige Anzahlen von **H**- oder **TH**-Objekten verwalten können (z.B. über eine Liste), ist der Aufbau von Bäumen möglich – c). Die verschiedenen Template- und Einschubmethoden können innerhalb komplexerer Basiskomponenten zusammengefaßt sein. Die für einen Workflow zusammenzufügenden Kettenglieder in Form von T-, H- und TH-Objekten existieren demnach als eigenständige Komponenten oder sind Teile komplexer Komponenten, so daß sie aus diesen herausgelöst und eingebunden werden können (gestrichelte Pfeile). Die Verwendung der komplexen Komponenten als ein einzubindendes Objekt ist aufgrund der Orthogonalität dieses Baukastensystems ebenfalls möglich (s.u.).

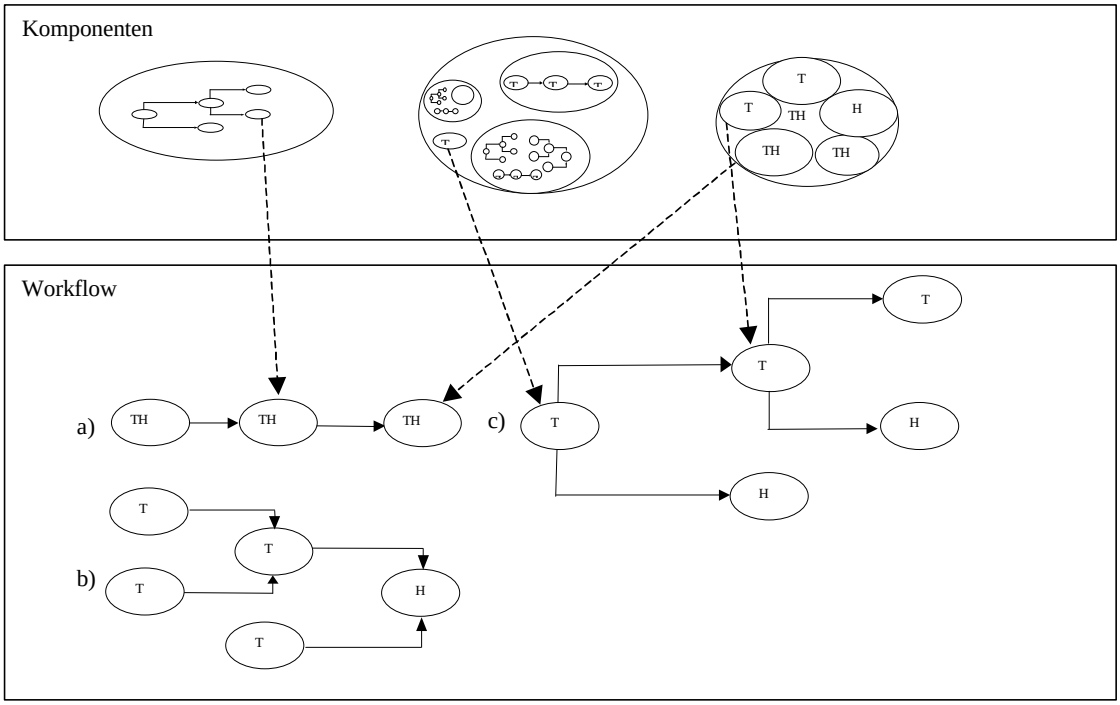


Abbildung 13: Beispiele für die Komposition von Objektgraphen

Nachfolgend wird die Kernstruktur der Templates anhand eines Codeausschnitts (Java) dargestellt [Pree97, S.50].

```
public class T extends H {
      ....
      H hRef;
      ....
      public void TH() {
            ....
            if (hRef)
                  hRef.TH()
            ....
      }
      ....
}
```

Damit Nachrichten im Graphen weitergeleitet werden, wird die gleiche Bezeichnung für Template- und Einschubmethoden verwendet, z.B. *TH()* . Eine Templatemethode ruft die Einschubmethoden aller im Graph nachfolgenden Objekte auf. Im Codebeispiel besitzt die Templateklasse eine Referenzvariable *hRef*, die die Beziehungen zu ihren H-Objekten verwaltet. T kann als Platzhalter für alle Objekte gesehen werden, die im gerichteten Graphen nachfolgen. Es ist hierdurch ausreichend, wenn dem T-Objekt eine Nachricht gesendet wird und nicht jedem nachfolgenden Objekt. Die Weiterleitung an die

anderen Objekte erfolgt automatisch. Zur Vermeidung von Endlosschleifen darf der gerichtete Graph keine Zyklen enthalten.

Durch die Eigenschaft der Templatemethoden in rekursiven Kompositionen Nachrichten weiterzuleiten, kann auch eine Baumhierarchie von Objekten wiederum als einzelnes Objekt behandelt werden.

2.6 FALLBASIERTES SCHLIESSEN

2.6.1 Übersicht

Das *fallbasierte Schließen* (*Case-based Reasoning - CBR*) umfaßt eine Kombination aus den Ideen und Methoden der Statistik, des Information Retrieval, des maschinellen Lernens und der Entwicklung von Expertensystemen [BaWe96]. Die Methodologie lehnt sich an der Vorgehensweise eines menschlichen Experten bei der Lösung eines Problems an. Wie bereits in *Kapitel 1.1 Übersicht* mit dem Ansatz von [Póly67] zur erfahrungsbasierten Problemlösung dargestellt, löst ein Experte Probleme oft auf der Basis von bekannten Fällen, die ihrer Problemstruktur nach vergleichbar mit der aktuellen Problemsituation sind, um die bereits bekannte Lösung effizient zur Lösung der neuen Aufgabe einzusetzen [Alt+92a]. Das fallbasierte Schließen stellt nun genau hierfür Mechanismen zur Verfügung, die diese auf Erfahrungen in der Vergangenheit basierende Vorgehensweise zur Problemlösung unterstützt.

Nachfolgend werden die Zielsetzungen des fallbasierten Schließens dargestellt, die als eine Verbesserung zu den bekannten Defiziten klassischer Systeme zu sehen sind [AlAa96]:

- Verbesserung der Performance wissensbasierter Systeme.
- Erweiterung der Einsatzbereiche.
- Vereinfachte Einsatzmöglichkeiten.

Die *Steigerung der Performance* ist durch eine Einschränkung des Suchraums zu erwarten, da durch das problemspezifische Wissen eine systematische Beschreibung des Suchraums erfolgt, während die Navigation über den Zugriff auf geeignete *Fallbeispiele* ausgeführt wird. Einem Fallbeispiel kommt bei dieser Systematik die Funktion einer Suchraumbeschränkung zu. So werden bei der Suche nur Fallbeispiele untersucht, die im Kontext zur aktuellen Problemstellung (-beschreibung) stehen. Des weiteren wird durch die Aufnahme eines neuen Fallbeispiels das erneute, vollständige Durchsuchen des Suchraumes vermieden, falls die identische Problemstellung nochmals als Anfrage an das System abgesetzt wird.

Eine *Erweiterung der Einsatzbereiche* ist deshalb zu erwarten, weil das Konzept des fallbasierten Schließens die Problemlösungsverfahren zusammen mit einer Lernkomponente integriert, wodurch bei jeder neu gemachten Erfahrung das zusätzlich gewonnene Wissen in einer Art Problemlösungsgedächtnis abgelegt wird. Dieser Ansatz der Kombination der Lernfähigkeit und das durch eine Anwendung neu erzielte Wissen resultierte aus der Erfahrung, die bei dem Einsatz von klassischen, wissensbasierten Systemen gemacht wurde, da hierbei eine strikte Trennung zwischen aktueller und früherer Problemlösung erfolgte [Wess95].

Die *vereinfachte Einsatzmöglichkeit* fallbasierter Systeme ergibt sich aus der Neuorientierung des Einsatzziels, da diese Methode die Entscheidungsunterstützung und nicht Entscheidungsfindung in den Vordergrund stellt. Bei den fallbasierten Systemen wird der menschliche Experte nicht ersetzt, sondern in den Entscheidungsprozeß mit einbezogen [AlWe91]. Durch die Möglichkeit der aktiven Einflußnahme liegt die Entscheidungskompetenz immer beim Anwender.

2.6.2 Grundlagen des fallbasierten Schließens

2.6.2.1 Prozeßmodell des fallbasierten Schließens

Ein bereits früher gelöstes und abgespeichertes Problem ist ein *Fallbeispiel*, das eine Problemsituation und die bei der Bearbeitung gemachte Erfahrung (Lösung) beschreibt. Die neu gewonnenen Erfahrungen in Form eines Fallbeispiels werden zur späteren Wiederverwendung in einer Fallbasis gespeichert. Eine *Fallbasis* repräsentiert eine Menge von gesammelten Fallbeispielen, die in sinnvoller

Weise in dieser organisiert werden. Diese Menge von gesammelten und gespeicherten Fallbeispielen kann jedoch ein aktuell auftretendes Fallbeispiel nicht mit den bereits gespeicherten in Beziehung setzen. Deshalb sind Methoden erforderlich, die einen systematischen Zugriff auf die gespeicherten Erfahrungen ermöglichen. Über diese Zugriffsmethoden soll eine Wiederverwendung bekannter Fallbeispiele zur Lösungsfindung eines aktuellen Fallbeispiels, für das noch keine Lösung vorliegt, erzielt werden. Deshalb nutzt das *fallbasierte Schließen* die Wiederverwendung (Reuse) von spezifischen Problemlösungswissen im Kontext zu einem aktuell zu lösenden Problem [Wess95].

2.6.2.2 Allgemeines Prozeßmodell

Nachfolgend wird das *Prozeßmodell des fallbasierten Schließens* nach Aamondt und Plaza [AaPl94] dargestellt. Das Prozeßmodell wird in die vier Subprozesse *Retrieve*, *Reuse*, *Revise* und *Retain* untergliedert. Ein *fallbasiertes System* implementiert zur Realisierung der erfahrungsbasierten Problemlösung die folgenden vier Prozeßphasen (siehe Abbildung 14):

1. Retrieve: Bereitstellung eines geeigneten Fallbeispiels.
2. Reuse: Wiederverwendung von gespeicherten Fallbeispielen.
3. Revise: Testen und modifizieren der gefundenen Lösung.
4. Retain: Speichern der neu gefundenen Lösung.

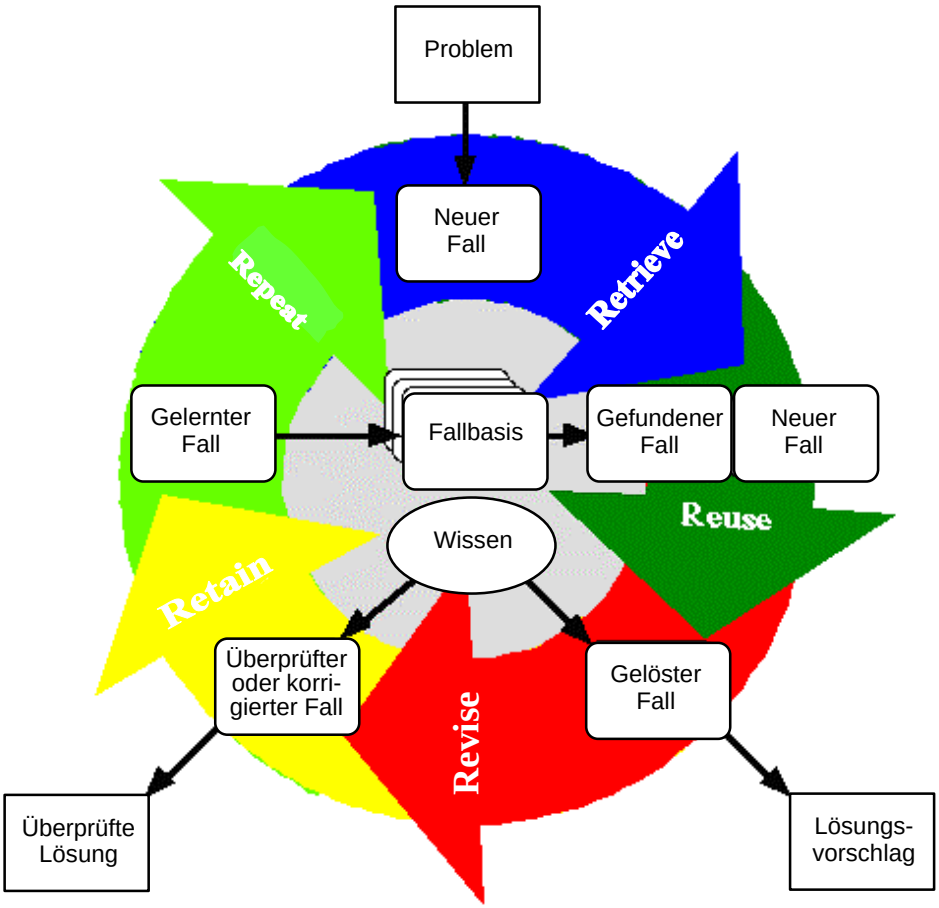


Abbildung 14: CBR-Phasenmodell

Die Retrievephase

In der *Retrievephase* wird ein Fallbeispiel oder eine Menge von Fallbeispielen aus der Fallbasis ausgewählt, die geeignet sind, um ein aktuelles Problem zu ösen [Berg96]. Dabei sind folgende Fragen von zentraler Bedeutung:

- Wann ist ein Fallbeispiel geeignet, um zur Lösung des aktuellen Problems herangezogen zu werden?
- Wie können diese geeigneten Fälle effizient aus der Fallbasis ausgewählt werden?

Die erste Fragestellung wird über Ähnlichkeitsbeurteilungen beantwortet. Dabei ist die Ähnlichkeit zwischen einem aktuell vorliegenden Problem und der Problembeschreibung eines Fallbeispiels aus der Fallbasis zu ermitteln. Dieser Ansatz geht davon aus, daß ähnliche Problemstellungen auch ähnliche Lösungen besitzen. Die Ermittlung der Ähnlichkeitsbeurteilungen erfolgt dabei über die Berechnung eines implementierten Ähnlichkeitsmaßes, welches häufig auf der Basis einer geometrischen Interpretation berechnet wird [Berg96]. Bei der geometrischen Interpretation werden die Fallbeispiele in einem n-dimensionalen Raum betrachtet, wobei jede Dimension einem Attribut aus der Problembeschreibung entspricht.

Da die Anzahl der gespeicherten Fallbeispiele in einer Fallbasis sehr groß sein kann, sind effiziente Zugriffs- bzw. Suchverfahren notwendig. Dazu ist eine entsprechende Strukturierung der verwendeten Fallbasis erforderlich. So müssen Verfahren aus dem Datenbankbereich, wie Bäume, Hash-Tabellen und Indexierungstechniken bereitgestellt werden (siehe hierzu [Egge97]).

Die Reusephase
In der *Reusephase* werden die in der Retrievephase ermittelten Fallbeispiele zur Gewinnung einer Lösung zu dem aktuellen Problem verwendet. Dazu werden die konkreten Erfahrungen, die durch die bereits gelösten Fallbeispiele gemacht worden sind, zur Generierung einer Lösung für das aktuelle Problem wiederverwendet. Die Komplexität der Reusephase ist stark von dem Aufgabenbereich abhängig, für den das fallbasierte System eingesetzt wird. Generell lassen sich die Aufgabenbereiche in *analytische* und *synthetische Problemstellungen* unterscheiden [Berg96].

Bei analytischen Problemstellungen kann weitgehend davon ausgegangen werden, daß die bei der Wiederverwendung behandelten Lösungen direkt als Lösung auf die aktuelle Problemstellung übertragen werden können und eine Anpassung der Lösung nicht erforderlich ist. Im Bereich der synthetischen Problemstellungen (z.B. Konfiguration oder Aktionsplanung) ist die Menge möglicher Lösungen wesentlich umfangreicher. Bei analytischen Problemen kann davon ausgegangen werden, daß für jede mögliche Lösung ein Fall vorliegt, was bei synthetischen Problemstellungen nicht sein muß.
Um aus dem bereits bekannten Wissen von vergleichbaren Fallbeispielen eine neue Lösung zu erzeugen, muß zuvor ein Transfer des relevanten Wissens zu der aktuell zu erstellenden Lösung erfolgen. Dieser Transfer läßt sich in vier Arten unterscheiden [Wess95]:

Vollständiger Transfer: Übernahme der kompletten Lösung.

Partieller Transfer: Teilweise Übernahme der Lösung.

Singulärer Transfer: Verwendung von nur einem Fallbeispiel.

Multipler Transfer: Verwendung von mehreren kombinierten Teillösungen verschiedener Fallbeispiele.

Zur Wiederverwendung des Wissens von bekannten Fallbeispielen existieren verschiedene *Adaptionsstrategien*, die eine Anpassung der aktuellen Lösung an bereits vorhandenes Wissen ermöglichen. In [Wess95] werden zwei verschiedene Strategien aufgezeigt:

- *Transformationsorientierte Strategie*: Dabei wird die in einem Fallbeispiel gefundene Problemlösung durch die Verwendung von problemspezifischen Wissen direkt verändert und auf die aktuelle Problemstellung angepaßt.
- *Prozeßorientierte Strategien*: Bei dieser Strategie ist nicht die eigentliche Lösung der zentrale Punkt, sondern der zur Lösung führende Prozeß. Der Lösungsprozeß wird im Zusammenhang mit der aktuellen Problemstellung wiederholt, wobei die eventuell erforderlichen Modifikationen vorgenommen werden.

In der Praxis wird die transformationsorientierte der prozeßorientierten Strategie vorgezogen, obwohl dabei nicht korrekte Lösungen entstehen können. Dieses resultiert aus der Schwierigkeit, die mit den

gestellten Anforderungen an prozeßorientierte Strategien verbunden sind bzgl. der Notwendigkeit zur Protokollierung und der Bereitstellung eines Mechanismus zur Abarbeitung der Lösungsprozesse.

Die Revisephase
In der *Revisephase* wird die in der Reusephase ermittelte Lösung überprüft und ggf. modifiziert. Die dabei durchzuführende Überprüfung kann über eine Simulation oder über eine Validierung in der realen Welt erfolgen [Berg96]. In der Praxis lassen sich allerdings selten Ansätze zur Simulation von erreichten Lösungen finden, da häufig ein umfangreiches Simulationswissen erforderlich ist.

Die Retainphase
In der *Retainphase* werden die durch die aktuelle Problemlösung gewonnenen Erfahrungen in der Fallbasis abgespeichert. Bevor ein neues Fallbeispiel in die Fallbasis übernommen wird, muß noch getestet werden, ob das neue Fallbeispiel der *reinen Wissensakquisition* oder auch dem *Lernprozeß* dient. Dabei stehen sich grundsätzlich der Aspekt der Bereitstellung allgemein neuen Wissens, der zum Aufbau der Fallbasis führt, und der Aspekt, daß die Fallbasis wirklich nur neues Wissen lernen soll, gegenüber. Es sollten nur Fallbeispiele in die Fallbasis aufgenommen werden, die Erfahrungen in das System einbringen, die stark von ihrer Basislösung abweichen. Sofern nur sehr geringe Informationsabweichungen bestehen, kann auch das Basisfallbeispiel um diese neuen Informationen erweitert werden [ReWi95].

2.6.2.3 Typen fallbasierter Systeme

Es werden die beiden Klassen (Typen) fallbasierter Systeme unterschieden [Alt+92b]:

Fallvergleichende Systeme: Ein fallvergleichendes System wird in den Bereichen eingesetzt, in denen die Ermittlung eines Fallbeispiels in Form eines Präzedenzfalls von Interesse ist, der bereits die Lösung des aktuellen Problems darstellt. Dabei wird ein neues Problem darauf untersucht, ob dieses wie eine bekannte Situationsbeschreibung interpretiert oder klassifiziert werden soll, oder ob Unterschiede erkennbar sind. Das Ergebnis des Fallvergleichs stellen Hypothesen und Begründungen dafür dar, warum eine Situationsbeschreibung in einer spezifischen Vorgehensweise interpretiert werden soll. Fallvergleichende Systeme werden häufig in den Bereichen der Rechtswissenschaft, der Beratung oder der Entscheidungsunterstützung eingesetzt.

Falladaptierende Systeme: Falladaptierende Systeme finden in den Bereichen ihre Verwendung, in denen zu einer gegebenen Problemstellung eine neue spezifisch angepaßte Lösung konstruiert werden soll. Der Schwerpunkt liegt auf der prozeßorientierten Generierung einer modifizierten Lösung für ein neues Fallbeispiel, welches in der Fallbasis hinterlegt werden soll. Die Modifikation der geeigneten Lösungsbeschreibungen ist notwendig, da die meist synthetischen Problemstellungen (z.B. Arbeitsplanung oder Konfigurationsprobleme) eine Vielfalt von Lösungen erfordern, die nicht alle in der Fallbasis gespeichert sind. Daher kann bei den falladaptierenden im Verhältnis zu den fallvergleichenden Systemen auch eine neue Lösung (neues Fallbeispiel) generiert werden. In der praktischen Anwendung werden jedoch meistens Mischformen der beiden Klassen benötigt, da keine genaue Abgrenzung zwischen den Schwerpunkten der beiden Systemklassen vorgenommen werden kann.

2.6.3 Bestimmung der Ähnlichkeit

Das generelle Problem bei der Ermittlung lösungsrelevanter Fallbeispiele ist, daß vor der eigentlichen Lösungsgenerierung ein Fallbeispiel ausgewählt werden muß, um im Anschluß die Aussage zu treffen, inwieweit es für die Gewinnung einzelner Lösungsschritte dienlich ist. Es muß also ein Kriterium bestimmt werden, was im Prinzip vorhersagt, wie nützlich ein Fallbeispiel für die Lösungsfindung sein wird (*a posteriori Kriteriums der Nützlichkeit*, das mangels anderer Wissensquellen, im fallbasierten Schließen auf das *a priori Kriterium der Ähnlichkeit* reduziert wird) [Wess93]. Es muß also die Nützlichkeit einer Problemlösung über die Ähnlichkeit formuliert bzw. prognostiziert werden.

2.6.3.1 Grundlagen zur Ähnlichkeitsbestimmung

Die *Nützlichkeit* eines gewählten Fallbeispiels wird anhand folgender Kriterien ermittelt [Wess95]:

- Ermöglichung einer Lösung des aktuellen Problems.
- Vermeidung früherer Fehler.
- Effiziente bzw. schnelle Lösungsfindung.
- Nachvollziehbarkeit der Lösung für den Benutzer.

Der Grad der Nützlichkeit steigt mit der Abnahme des Modifikationsaufwands. Je geringer der Aufwand für die Anpassung der gefundenen an die aktuelle Lösung ist, desto nützlicher erscheint das gefundene Fallbeispiel. Das Problem bei der Auswahl von relevanten Fallbeispielen besteht darin, mittels *unsicherer Informationen* vorherzusagen, ob die noch unbekannte und zu findende Lösung *nützlich* ist. Diese Problematik veranschaulicht Abbildung 15 [Wess95, S.43].

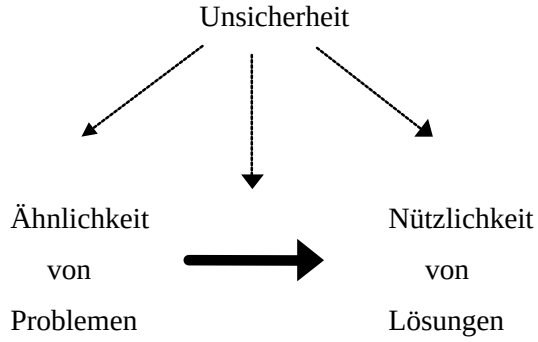


Abbildung 15: Ähnlichkeit, Nützlichkeit und Unsicherheit

2.6.3.2 Formale Bestimmung der Ähnlichkeit

Nachfolgend werden die drei verschiedenen Formalisierungen des Ähnlichkeitsbegriffs im fallbasierten Schließen betrachtet [WeGl94]:

- *Ähnlichkeit als Prädikat*: Betrachtung der Ähnlichkeit als eine Beziehung zwischen Objekten oder Ergebnissen, die entweder besteht oder nicht besteht.
- *Ähnlichkeit als Präferenzrelation*: Betrachtung der Ähnlichkeit als eine Beziehung zwischen Objekten oder Ergebnissen, die mehr oder weniger besteht.
- *Ähnlichkeit als Maß*: Betrachtung der Ähnlichkeit als eine Beziehung zwischen Objekten oder Ergebnissen, wobei die Ähnlichkeit quantifiziert wird.

2.6.3.2.1 Ähnlichkeit als Prädikat

Bei dieser Formalisierungsform wird die Ähnlichkeit in Form eines Prädikats $SIM \subseteq U^2$, wobei U alle Objekte des Universums bezeichnet, mit $SIM(x,y) = \begin{cases} 1; & y \text{ ist ähnlich zu } x \\ 0; & \text{sonst} \end{cases}$ dargestellt. Als Resultat dieser binären Ähnlichkeitsbeurteilung kann entweder die Aussage erfolgen, daß eine Ähnlichkeit zwischen zwei Objekten besteht oder nicht besteht.

Anforderungen: Für $SIM(x,y)$ wird die Erfüllung folgender *Eigenschaften* gefordert:

Reflexivität: $\forall x \in U \quad SIM(x,x)$

Symmetrie: $\forall x, y \in U \quad SIM(x,y) \Leftrightarrow SIM(y,x)$

Auswahl von Fallbeispielen: Bei einer beliebigen Realisierung von *SIM* kann dann für eine Problemstellung $x \in U$ eine Lösung gefunden werden, wenn gilt:

$$\exists x \in CB \quad SIM(x, y)$$

Die durch das Prädikat *SIM* bestimmte Menge an Fallbeispielen $L_x = \{y \in CB | SIM(x, y)\}$ sollte sinnvollerweise nur ein Element (Fallbeispiel) enthalten, da sonst noch eine separate Konfliktlösungsstrategie definiert werden muß.

2.6.3.2.2 Ähnlichkeit als Präferenzrelation

Eine weitere Möglichkeit zur Formalisierung des Ähnlichkeitsbegriffs stellt die Präferenzrelation dar, bei der die Ähnlichkeit im Sinne von mehr oder weniger in eine Ordnung gebracht wird. Folgende Basisrelation $\Pr\ddot{a} \subseteq U^4$ gilt für die Definition der Präferenzrelation:

$$\Pr\ddot{a}(x, y, u, v) \Leftrightarrow x \text{ ist mindestens so ähnlich zu } y \text{ wie } u \text{ zu } v$$

Anforderungen: Für Prä wird die Erfüllung folgender Eigenschaften gefordert:

Maximales Element: $\Pr\ddot{a}(x, x, u, v)$

Symmetrie in den Argumenten: $\Pr\ddot{a}(x, y, u, v) \Leftrightarrow \Pr\ddot{a}(y, x, u, v) \Leftrightarrow \Pr\ddot{a}(x, y, v, u)$

Reflexivität: $\Pr\ddot{a}(x, y, x, y)$

Transitivität: $\Pr\ddot{a}(x, y, u, v) \wedge \Pr\ddot{a}(u, v, s, t) \Rightarrow \Pr\ddot{a}(x, y, s, t)$

Antisymmetrie: $\Pr\ddot{a}(x, y, u, v) \wedge \Pr\ddot{a}(u, v, x, y) \Rightarrow (x, y) \sim (u, v)$

Die Relation *Prä* induziert damit eine Partialordnung $\geq$ über den Tupeln aus U, d.h.

$$\Pr\ddot{a}(x, y, u, v) \Leftrightarrow (x, y) \geq (u, v)$$

Die induzierte Ordnung $\geq$ kann dabei aufgeteilt werden in einen strikten Teil $\succ$ mit:

$$(x, y) \succ (u, v) \Leftrightarrow (x, y) \geq (u, v) \wedge \neg[(u, v) \geq (x, y)],$$

wobei $\succ$ antisymmetrisch und transitiv ist, und in einen indifferenten Teil mit:

$$(x, y) \sim (u, v) \Leftrightarrow (x, y) \geq (u, v) \wedge (u, v) \geq (x, y),$$

wobei $\sim$ reflexiv, symmetrisch und transitiv ist, d.h. ist eine Äquivalenzrelation. Die Tupel $(u, v) \sim (x, y)$ sind dann ununterscheidbar in *Prä* und können nicht bzgl. $\geq$ geordnet werden.

Die Relation *Prä* bzw. die von *Prä* induzierte Präferenzordnung $\geq$ über U kann nun zur Auswahl von Fallbeispielen verwendet werden. Zur Vereinfachung wird zunächst eine dreistellige Ordnungsrelation $S \subseteq U^3$ definiert:

$$S(x, y, z) \Leftrightarrow \Pr\ddot{a}(x, y, x, z)$$

Die entsprechenden Axiome von *Prä* behalten für S ihre Gültigkeit. Die durch S induzierte Präferenzordnung kann auch als

$$S(x, y, z) \Leftrightarrow y \geq_x z$$

dargestellt werden.

$\exists y \forall x \in CB$

$S(x, y, z) \Leftrightarrow \exists y \forall z \in CB$

$y \geq_x z$

2.6.3.2.3 Ähnlichkeit als Maß

Die Formalisierung der Ähnlichkeit über die Angabe eines Distanz- oder Ähnlichkeitsmaßes wird außer der Ordnungsinformation auch noch das Ausmaß der Ähnlichkeit ermittelt [Wess95].
Das Ähnlichkeitsmaß wird dabei als eine Funktion $sim(x, y) : U^2 \rightarrow [0,1]$ definiert, an die folgende *Anforderungen* gestellt werden [AlWe91]:

Reflexivität: $\forall x \in U \quad sim(x, x) = 1$

Symmetrie: $\forall x, y \in U \quad sim(x, y) = sim(y, x)$

Auswahl von Fallbeispielen: Bei der Auswahl wird das Fallbeispiel gesucht, welches die maximale Ähnlichkeit zu $x \in U$ aufweist:

$$\exists y \in CB \forall z \in CB \quad sim(x, y) \geq sim(x, z)$$

Durch den konkreten Wert der Ähnlichkeit $sim(x, y)$ kann eine Aussage über das Ausmaß der Nützlichkeit des Fallbeispiels *y* zur Lösungsfindung des Problems *x* gemacht werden. Eine weitere Möglichkeit der Formalisierung der Ähnlichkeit ist der geometrische Ansatz, bei dem die Ähnlichkeit über ein *Distanzmaß* bestimmt wird.
Das Distanzmaß wird als eine Funktion $d(x, y) : U^2 \rightarrow R_0^+$ definiert, die den Tupeln (x, y) aus $U \times U$ einen numerischen Wert aus R_0^+ zugeordnet wird. Die Funktion muß dabei folgenden Anforderungen genügen:

Reflexivität: $d(x, x) = 0$

Symmetrie: $d(x, y) = d(y, x)$

Falls ein Distanzmaß $d(x, y)$ die Eigenschaften einer Metrik aufweist, kann zur Gruppierung von Objekten die räumliche Anschauung herangezogen werden, welche als metrischer Raum definiert wird, für den *d* folgende zusätzliche Anforderungen erfüllen muß:
Minimum: $d(x, y) = 0 \Rightarrow x = y$

Dreiecksgleichung: $d(x, y) + d(y, z) \geq d(x, z)$

Hierbei stellt d eine Metrik und (U, d) einen metrischen Raum dar.

2.6.3.2.4 Arten der Ähnlichkeitsinformationen

Binäre Ähnlichkeitsinformation

Zur Bestimmung des ähnlichsten Fallbeispiels aus der Fallbasis muß jeder Formalisierungsansatz auf eine binäre Entscheidung in Form von $SIM(x, y)$ reduziert werden. Das über diese Entscheidung ermittelte Fallbeispiel ist danach die Grundlage für die nachfolgenden Prozeßschritte des Zyklus des fallbasierten Schließens. Die Funktion bzw. die Bedeutung, die SIM zugeordnet wird, hängt von der Bedeutung des entsprechenden Problemlösungsprozesses ab, für den SIM verwendet wird. Dabei könnte SIM eine Klassifikationsaufgabe von Fallbeispielen aus der Fallbasis beigemessen werden [Wess95].

Ordinale Ähnlichkeitsinformation

Die ordinale Ähnlichkeitsskalierung dient der Ordnung bzw. der Bestimmung der Ränge von Fallbeispielen aus einer Fallbasis. Im Mittelpunkt dieser Betrachtung steht die ordinale Bewertung der Ähnlichkeit, die über die Verwendung von Präferenzrelationen realisiert wird. Die Funktion bzw. Bedeutung, die der verwendeten Präferenzrelation beigemessen wird, hängt von der mit der Auswahl verfolgten Zielstellung ab. Soll über die Präferenzrelation $y \geq_x z$ beispielsweise ein Fallbeispiel *y* gefunden werden, welches für die Lösungsgewinnung von *x* leichter modifizierbar ist als für *z*, oder führt der in *y* enthaltene Arbeitsplan zu einer kostengünstigeren Alternative als der Arbeitsplan in *z* hinsichtlich der Problemstellung *x* ?

Die generelle Notwendigkeit des Einsatzes von ordinal skalierten Fallbeispielen ergibt sich aus dem Umstand, daß Systeme, die sich noch in der Lernphase befinden, nicht genügend Informationen zur eindeutigen Bestimmung des ähnlichsten Fallbeispiels über ein spezifisches Prädikat SIM (binäre Entscheidung) besitzen. Es kann also der Fall auftreten, daß mehr als ein Fallbeispiel in der durch *SIM* erzeugten Menge des Lösungskandidaten enthalten ist. Um dennoch einen geeigneten Lösungskandidaten aus dieser Menge auszuwählen, wird zusätzlich kodiertes Wissen in Form einer Präferenzrelation auf die Menge der Lösungskandidaten angewendet, wodurch die Elemente dieser Menge nach ihrer Lösungsrelevanz geordnet werden.

Kardinale Ähnlichkeitsinformation

Eine kardinale Ähnlichkeitsskalierung kann als Erweiterung der Informationen der ordinalen Ähnlichkeitsskalierung aufgefaßt werden, da zusätzlich zu der Ordnung der einzelnen Fallbeispiele auch noch die Differenz der Ähnlichkeit zweier Fallbeispiele oder der Betrag der absoluten Ähnlichkeit bekannt ist. Der Verwendung einer kardinalen Ähnlichkeitsskala kommt die Bedeutung einer quantitativen Bewertung von Ähnlichkeiten zu, wodurch die Möglichkeit entsteht, das eventuell aus der über ordinal skalierte Ähnlichkeiten vorgeschlagene ähnlichste Fallbeispiel als völlig ungeeignet zu erklären. Des weiteren kann durch die Angabe eines Schwellwerts $sim(x, y) \geq d$ auch festgestellt werden, daß vielleicht alle Fallbeispiele der Fallbasis zur Lösung des aktuellen Problems *x* ungeeignet sind [RiWe91].

2.6.3.3 Ebenen der Ähnlichkeitsbewertung

Nachfolgend wird ein formaler Vergleich von Ähnlichkeitsbewertungen vorgenommen, wobei der Zusammenhang zwischen dem im System repräsentierten und dem in der Anwendung genutzten Ähnlichkeitsbegriff betrachtet wird. Die beiden Ebenen der Ähnlichkeit lassen sich wie folgt definieren [Wess95]:

Ähnlichkeit auf der Repräsentationsebene $Sim_{\mathrm{Re}p}$: Der Ähnlichkeitsbegriff $Sim_{\mathrm{Re}p}$ repräsentiert den systemseitig formulierten Ähnlichkeitsbegriff, der als Grundlage für jede Entscheidung in der Auswahlphase dient.

Ähnlichkeit auf der Anwendungsebene Sim_{App}: Der Ähnlichkeitsbegriff Sim_{App} stellt die in der Anwendung tatsächlich ermittelte Ähnlichkeit dar, die eine Aussage darüber liefert, inwieweit sich ein gefundenes Fallbeispiel zur Lösung des aktuellen Problems als nützlich erweist. In der Test- und Lernphase des fallbasierten Systems findet ein Abgleich zwischen dem vom System repräsentierten und dem in der Anwendung tatsächlich vorhandenen Ähnlichkeitsbegriff statt.

Hieraus kann eine grundsätzliche Zielsetzung des fallbasierten Schließens formuliert werden, die Übereinstimmung bzw. eine große Annäherung des durch das System repräsentierten und des in der Anwendung tatsächlich vorhandenen Ähnlichkeitsbegriffs zu erreichen. Denn gerade durch eine annähernde Übereinstimmung drückt sich der Erfolg oder die Qualität eines Systems aus. Die Kriterien, die den Erfolg der Lernphase eines Systems beurteilen, sollen nachfolgend aufgeführt werden [Wess95]:

Übereinstimmung von Ähnlichkeitsbegriffen: Zur Bestimmung der Übereinstimmung zweier Ähnlichkeitsbegriffe Sim_1, Sim_2 werden die von den entsprechenden Relationen $\Pr\ddot{a}_1$ und $\Pr\ddot{a}_2$ induzierten Ordnungen $\succ_1, \succ_2$ auf *U* verglichen:

Sim_1 und Sim_2 *stimmen partiell überein,* falls $\forall a,b,c \in U \quad a \succ_c^1 b \Rightarrow a \succ_c^2 b$

Sim_1 und Sim_2 *stimmen überein,* falls $\forall a,b,c \in U \quad a \succ_c^1 b \Leftrightarrow a \succ_c^2 b$

Korrektheit von Ähnlichkeitsbegriffen: Der repräsentierte Ähnlichkeitsbegriff $Sim_{\mathrm{Re}\,p}$ heißt korrekt bzgl. des Ähnlichkeitsbegriffs Sim_{App}, falls Sim_{App} und $Sim_{\mathrm{Re}\,p}$ partiell übereinstimmen: $\forall a,b,c \in U \quad a \succ_c^{\mathrm{Re}\,p} b \Rightarrow a \succ_c^{App} b$

Vollständigkeit von Ähnlichkeitsbegriffen: Der repräsentierte Ähnlichkeitsbegriff $Sim_{\mathrm{Re}\,p}$ heißt vollständig bzgl. des Ähnlichkeitsbegriffs der Anwendung Sim_{App}, falls $Sim_{\mathrm{Re}\,p}$ und Sim_{App} *partiell übereinstimmen*: $\forall a,b,c \in U \quad a \succ_c^{App} b \Rightarrow a \succ_c^{\mathrm{Re}\,p} b$

Korrektheit und Vollständigkeit von Ähnlichkeitsbegriffen: Der repräsentierte Ähnlichkeitsbegriff $Sim_{\mathrm{Re}\,p}$ heißt korrekt und vollständig bzgl. des Ähnlichkeitsbegriffs Sim_{App}, falls $Sim_{\mathrm{Re}\,p}$ und Sim_{App} übereinstimmen: $\forall a,b,c \in U \quad a \succ_c^{\mathrm{Re}\,p} b \Leftrightarrow a \succ_c^{App} b$

Es gilt das Ziel der *Korrektheit* und *Vollständigkeit* von den Ähnlichkeitsbegriffen $Sim_{\mathrm{Re}\,p}$ und Sim_{App} zu erreichen.

2.7 ZUSAMMENFASSUNG DER BASISTECHNOLOGIEN / ÜBERBLICK ZUR UMSETZUNG

In *Kapitel 2 Basistechnologien der Reportingplattform* wurde ein Überblick zu den Konzepten gegeben, die notwendigerweise in die Entwicklung der Reportingplattform eingeflossen sind, um die gestellten Anforderungen zu erfüllen. Im folgenden *Kapitel 3 Entwicklung der Reportingplattform* schließt sich die Beschreibung an, wie die Integration dieser Technologien zu einem konsistenten Gesamtsystem erfolgt. Zunächst wird nochmals der jeweilige Verwendungszweck der beschriebenen Basistechnologien erläutert.

Benutzerschnittstelle
Die in *Kapitel 2.1 Konzepte zum Entwurf ergonomischer Benutzeroberflächen* beschriebenen Verfahren für den sinnvollen Entwurf von Benutzerschnittstellen fließen in das Systemdesign der Reportingplattform ein (siehe *Kapitel 4.2.2.2 Dialog zur Problemspezifikation*). Aus dem Bereich EIS (*Kapitel 2.3 EIS - Executive Information Systems*) finden Methoden zur einfachen, intuitiven Benutzerführung sowie der Gestaltung der Benutzeroberflächen Verwendung. Hierzu werden Darstellungswerkzeuge, wie Tabellen, Kreuztabellen, Graphiken etc. implementiert und mit Drill-Down, Traffic Lighting Funktionen zur Abweichungsanalyse versehen (siehe *Kapitel 4.3.3 Werkzeuge der Analysekomponente*). Der Zugriff auf Unternehmensdaten sowohl in aggregierter Form als auch auf operative Datenbestände zum intuitiven Durchhangeln wird in *Kapitel 3.2 Basismechanismen der Reportingplattform* und *Kapitel 3.5 SQL-Templates* beschrieben. Der gesamte Ablauf einer Sitzung kann mittels Point & Click erfolgen. Insbesondere die hierdurch sehr einfache Anpassung gefundener Lösungen sowie das Einfrieren aktueller Bearbeitungsstände bieten größtmögliche Unterstützung für den Anwender.

Informationsabfragen
SQL als de facto Standardanfragesprache für relationale Datenbankmanagementsysteme (RDBMS), findet als Querysystem Verwendung. Aufgrund der für den Anwender nicht ohne weiteres zu verwendenden Notation von SQL werden Konzepte für die Kapselung vor dem Anwender entwickelt, die gleichzeitig ein hohes Maß an Adaptionsmöglichkeiten bieten.

Modellierung von Entscheidungsprozessen
Aufgrund der Einschränkung der Reportingplattform für den Einsatz in Domains mit dynamischen Workflows und teilstrukturierten Arbeitsvorgängen können Konzepte aus dem Bereich Geschäftsprozeß- / Arbeitsvorgangs- und Workflowmodellierung (siehe *Kapitel 2.4 Workflowmanagement*) genutzt werden, um Prozeßbausteine zu modellieren und zur Benutzerführung im Rahmen der Entscheidungsunterstützung zu verwenden. Zur Beschreibung der Prozesse werden EPK verwendet (siehe Anhang *Kapitel 11.2 EPK - Ereignisgesteuerte Prozessketten* - Notation). Die Umsetzung wird in *Kapitel 3.6.4 Abbildung der Prozesse* gezeigt.

Dynamische Steuerung der Benutzerführung durch Prozesse
Die in *Kapitel 2.4 Workflowmanagement* beschriebenen Technologien werden genutzt, um den Anwender durch eine Prozeßsteuerung bei der Lösung seiner Problemstellungen zu unterstützen. Die Spezifikationen der verschiedenen Aspekte eines Workflows sowie die Möglichkeit für den Anwender durch Point & Click Anpassungen an den Prozeßabläufen vorzunehmen gewährleisten die benötigte hohe Flexibilität des Systems (siehe *Kapitel 3.6.4.6 Steuerung von Prozessen*).

Flexible Anpassung und Komposition der einzelnen Bausteine
Durch die Verwendung des in *Kapitel 2.5.2 Das Design Pattern Template* dargestellten Entwurfsmusters Template für die Modellierung der SQL-Abfragen und Prozesse sind die Voraussetzungen für eine Adaptierbarkeit z.B. durch Parametrisierung sowie für den Aufbau eines komponentenbasierten Frameworks gegeben (siehe *Kapitel 3.5 SQL-Templates* und *3.6 Prozesstemplates*).
Die Verwendung der in *Kapitel 2.5.3 Mechanismen komponentenbasierter Frameworks* beschriebenen Konzepte hinsichtlich Schnittstellen und Kopplung ermöglichen die Umsetzung der geforderten Flexibilität bei der Anpassung und Komposition der in Form von Komponenten modellierten SQL- und Prozeßtemplates (siehe *Kapitel 3.5.3 Modellierung von SQL-Templates* und *Kapitel 3.6.3 Verknüpfen von Prozeßkomponenten*).
Erst durch die Mechanismen der Componentware läßt sich das Handling der Exceptions auch zur Laufzeit umsetzen. Die Steuerung des gesamten Systems erfolgt durch ein Framework (siehe *Kapitel 4.4.1 Systemarchitektur des Frameworks*).

Unterstützung des Anwenders bei der Lösungssuche
Durch die in *Kapitel 2.6 Fallbasiertes Schliessen* beschriebenen Mechanismen des fallbasierten Schließens wird die Möglichkeit gegeben, den Anwender bei der Suche nach Lösungen in Form von SQL-Templates und Prozeßtemplates zu unterstützen (siehe *Kapitel 3.6.5 Zusammenfassung* und *4.2.2.1 Die Prozeßfallbasis*). Dieselben Mechanismen kann auch der Modellierer der SQL-Templates und der Prozeßbausteine nutzen (siehe *Kapitel 3.5.4 Wiederauffinden modellierter Templates*), wobei diese über das auch dem späteren Endanwender zur Verfügung gestellten DSS-Tool nutzbar sind (siehe *Kapitel 4.2 Das DSS-Werkzeug*). Die systemseitige Bewertung der Ähnlichkeit der gefundenen und angebotenen Lösungen zur spezifizierten Problemstellung ermöglichen es dem Anwender, sich auf die wesentlichen Lösungen zu fokussieren (siehe *Kapitel 3.7.3.3 Beschreibung der implementierten Retrieveverfahren*).

Die Speicherung neuer Lösungen als letzte Phase im CBR-Prozeßmodell läßt die verfügbare Lösungsmenge der Reportingplattform kontinuierlich anwachsen und führt damit zu einem *lernenden System* (siehe *Kapitel 4.2.2.4 Dialog zur Bearbeitung, Beurteilung und Übernahme einer Lösung*).

Tabelle 3 faßt die Basistechnologien und deren Nutzen bei der Entwicklung der Reportingplattform nochmals zusammen.

Basistechnologie	Nutzen für die Reportingplattform
Executive Information Systems	Benutzeroberfläche, Anwendungsbereich, Funktionalitäten
Decision Support Systems	Anwendungsbereich
Geschäftsprozeß-, Arbeitsvorgangs-, Workflowmodellierung	Spezifikation der Entscheidungsprozesse
Workflowmanagementsysteme	Benutzerführung
Exception Handling	Ad hoc Benutzerführung, Adaption von Prozessen
Design Pattern Template	Entwurf von Komponentenstrukturen Wiederverwendbarkeit
Componentware	Flexibilität bei der Kooperation von Bausteinen (SQL- und Prozeßtemplates) sowie deren Komposition
Parametrisierung	Adaption der Komponenten
Frameworks	Systemarchitektur der Reportingplattform
Fallbasiertes Schließen	Wiederauffinden bereits gespeicherter Lösungen, Adaption und Speicherung neuer Lösungen, Ähnlichkeitsbewertung
SQL	Anfragesprache an DBMS
Softwareergonomie, Dialognetze	Oberflächengestaltung der Benutzerschnittstelle

Tabelle 3: Übersicht der Basistechnologien und deren Verwendung

3 ENTWICKLUNG DER REPORTINGPLATTFORM

Nachfolgend wird sukzessive die Entwicklung des Reportingplattformkonzepts dargestellt. Die Beschreibung ausgewählter Algorithmen und Datenstrukturen erfolgt in *VDM-Notation* (*Vienna Development Method*) [Jone90] [WoHe93] (siehe Anhang *Kapitel 11.1 VDM - Vienna Development Method - Notation*).

3.1 ÜBERSICHT

Eine Zielsetzung der Reportingplattform ist die Entwicklung einer Templatearchitektur für eine bestimmte Art von SQL-Anfragen. Hierdurch wird ein hoher Grad an Wiederverwendbarkeit erreicht, da einmal spezifizierte SQL-Templates (siehe *Kapitel 3.5 SQL-Templates)* in verschiedenen Kontexten durch leichte Modifikationen / Adaptionen an den jeweiligen Informationsbedarf anpaßbar sind. Dazu sind entsprechende Metadaten zu modellieren und in einem Repository abzulegen.

Neben der Nutzung des Templatekonzepts für die Wiederverwendung von SQL-Statements werden ebenfalls Prozesse in Form von Templates spezifiziert (siehe *Kapitel 3.6 Prozesstemplates)*. Dies bildet die Basis für eine flexible Kombinationsmöglichkeit und Adaption der Prozesse. Die Adaption der Prozeßtemplates zur Laufzeit wird durch Konzepte aus dem Bereich Componentware ermöglicht. Diese gewährleisten, daß z.B. bei Veränderung der Prozeßstrukturen kein Neukompilieren der gesamten Prozeßinstanz erforderlich ist (siehe *Kapitel 3.6.3 Verknüpfen von Prozeßkomponenten*).

Die Unterstützung des Anwenders / Modellierers hinsichtlich des Wiederauffindens bereits abgespeicherter Lösungen und deren Bewertung in bezug auf die Relevanz zu einer neuen Problembeschreibung wird durch Mechanismen des fallbasierten Schließens umgesetzt (siehe *Kapitel 3.7 Konzept zum Wiederauffinden ähnlicher Problembeschreibungen*).

Zunächst werden jedoch einige Basiskonzepte der Reportingplattform erläutert, die die Grundlage für die o. g. weitergehenden Mechanismen bilden.

3.2 BASISMECHANISMEN DER REPORTINGPLATTFORM

Vor der späteren Beschreibung des SQL-Templatekonzepts wird nachfolgend zunächst ein SQL-Generator entwickelt, der durch die Verwendung modellierter Metadaten SQL-Statements generiert. Anschließend wird dieses Konzept durch die Integration *festdefinierter SQL-Statements* erweitert. Hierdurch gewährleistet das System die Verfügbarkeit des gesamten SQL-Sprachumfangs bei der Notwendigkeit zur Verwendung komplexer Statements. Nachfolgend wird ein Überblick zum Repository, den DataSets als modellierte Informationsabfragen, die als Schnittstelle zum Anwender fungieren und festdefinierten SQL-Statements gegeben.

3.2.1 Generelle Struktur des Repositorys zur Erzeugung von SQL-Statements

Das Repository der Reportingplattform enthält u.a. alle erforderlichen Informationen über die Datenbankstruktur, die nötig sind, um systemgestützt SQL-Statements zu generieren. Die Daten gelangen über den *Repositorymanager* in eine eigens für das Repository notwendige Tabellenstruktur, in der die Daten in einem Format abgespeichert sind, das für den zu entwickelnden SQL-Generator interpretierbar ist [Kuge96]. Um den Aufbau des Repositorys zu veranschaulichen, soll folgendes Beispiel dienen, in dem die erforderlichen Tabelleneinträge zur Realisierung einer exemplarischen Datenbankanfrage im Repository erläutert werden.

```
SELECT  KPartner.Name, KRegelauftrag.Beschreibung, COUNT(*)
FROM    Verplanen, KPartner, KRegelauftrag
WHERE   Verplane.AU_VerantwortlichenID = KPartner.ObjektID      AND
        Verplanen.AU_OberauftragsID    = KRegelauftrag.ObjektID
GROUP BY KPartner.Name, KRegelauftrag.Beschreibung ;
```

In der Tabelle *RP_Links* werden die Schlüssel- und Fremdschlüsselbeziehungen zwischen den Datenbanktabellen beschrieben, indem eine Zuordnung zwischen den entsprechenden Tabellen und den sie verknüpfenden Spalten vorgenommen wird.

RP_Links	Link	Basistabelle	Feld	Fremdtabelle	Fremdfeld
	Verplanen_Partner	Verplanen	AU_VerantwortlichenID	Kpartner	ObjektID
	Verplanen_KRegelauftrag	Verplanen	AU_OberauftragsID	KRegelauftrag	ObjektID

Tabelle 4: Repositorytabelle RP_Links

Außer den Links sind *Attribute* zu beschreiben, durch deren Auswahl der Benutzer der Reportingplattform zur Laufzeit seinen Informationsbedarf formulieren kann (s.u.). Diese Attribute müssen nicht identisch sein mit den einzelnen Spalten der Datenbanktabellen die nachfolgend als Spalten oder auch als Felder bezeichnet werden. Die Namen aller Attribute, die dem Anwender zur Informationsgewinnung zur Verfügung stehen, werden über eine View auf die Tabelle RP_Partattribute zusammengestellt.

In der Tabelle *RP_Partattribute* ist für jedes Attribut definiert, wie es sich aus anderen Bestandteilen zusammensetzt. Diese Bestandteile können z.B. einfache Tabellenspalten, Spaltenkombinationen einer Tabelle, Gruppenfunktionen auf Spalten einer Tabelle, Attribute aus einer anderen Tabelle, Kombinationen von Attributen zweier Tabellen und Gruppenfunktionen auf Attribute sein. Ist ein Attribut mit einer Tabellenspalte identisch, so wird es als *atomar* bezeichnet. Über das Feld *Link* werden Attribute aus Fremdschlüsseln anderer Tabellen - sogenannte *Fremdattribute* - festgelegt. *Atomare Attribute* stellen im Beispiel *Name, Beschreibung* und *Anzahl Fahrzeuge* dar.

Soll eine Spalte auch durch ein Attribut erreichbar sein, das nicht in der Basistabelle dieser Spalte zugeordnet ist, sondern einer Tabelle, in der die Spalte einen Fremdschlüssel darstellt, so ist ein *Fremdattribut* zu definieren, welches das Attribut der Basistabelle referenziert und den *Link*, über den die beiden Tabellen miteinander verknüpft sind, angibt.

Im Beispiel beinhaltet das Attribut *Name* mit der Basistabelle *KPartner* direkt die gleichnamige Tabellenspalte. Das Attribut *Auftraggeber* der Basistabelle Verplanen, in der die Spalte *Name* unter der Bezeichnung *AU_VerantwortlichenID* als Fremdschlüssel enthalten ist, verweist auf das Attribut *Name* und auf *Verplanen_Partner* als zugehörigen Link zwischen den Tabellen. Derartige Verkettungen können sich auch über mehrere Stufen vollziehen, so daß zunächst Fremdattribute auf weitere Fremdattribute verweisen, bis am Ende der Kette ein atomares Attribut referenziert wird.

RP_Partattribute	Attribut	Tabelle	Quelle	Link
	Name	KPartner	[[Name]]	
	Beschreibung	KRegelauftrag	[[Beschreibung]]	
	Auftraggeber	Verplanen	Name	Verplanen_Partner
	Auftrag	Verplanen	Beschreibung	Verplanen_Regelauftrag
	Anzahl Fahrzeuge	Verplanen	COUNT([[TP_ObjektID]])	
	Anzahl Avise	Avise	COUNT([[TP_ObjektID]])	
	Gesamtanzahl	Verplanen	COUNT([[TP_ObjektID]]) + <<Anzahl Avise>>	

Tabelle 5: Repositorytabelle RP_Partattribute

Attributbezeichnungen müssen nicht eindeutig sein, sondern es können mehrere gleichnamige *Partialattribute* bestehen, welche erst zusammengefaßt das gesamte Attribut bilden. Es kann sich bei diesen Partialattributen sowohl um über Links verbundene Atomar- oder Fremdattribute handeln als auch um völlig unabhängige, deren Basistabellen überhaupt nicht miteinander verknüpft sind.
Ergänzend können Attributen bestimmte Beschreibungen zugeordnet werden, etwa ob sie für den Anwender sichtbar sind, ob nach ihnen gesucht werden darf oder ob bei Auswahl bestimmter Attribute eine Gruppierungsklausel in das SQL-Statement integriert werden soll, wie es im Beispiel für das Attribut *Anzahl Fahrzeuge* der Fall ist, hinter welchem sich eine Aggregationsfunktion verbirgt.

RP_Eigenschaften	Attribut	Tabelle	Eigenschaft	Wert
	Anzahl Fahrzeuge	Verplanen	BGoup	TRUE
	Anzahl Avise	Avise	BGroup	TRUE
	Gesamtanzahl	Verplanen	BCalc	TRUE

Tabelle 6: Repositorytabelle RP_Eigenschaften

Die Tabelleninhalte der zur Attributdefinition wichtigen Tabellen RP_Partattribute, RP_Links und RP_Eigenschaften haben folgende Bedeutung:

Bezeichnung	Beschreibung
Attribut	Name des Attributs wie im Informationssystem angezeigt.
Tabelle	Tabelle, aus der die Attributbestandteile stammen.
Quelle	Definition eines neuen Attributs aus Tabellenspalten, anderen Attributen oder Gruppenfunktionen. Namen von bereits vorhandenen Attributen werden hier in << >> - Zeichen und die von Tabellenspalten in [[ ]] – Zeichen eingeschlossen.
Link	Name einer Verknüpfung zweier Tabellen.
Basistabelle, Feld, Fremdtabelle, Fremdfeld	Diese vier Felder geben die Schlüssel- Fremdschlüssel-Beziehungen zwischen zwei Tabellen an.
ItemID	Besteht aus dem Attributnamen und der Tabelle und identifiziert jedes Attribut eindeutig.
Objekt	Typ des Attributs (Attribut oder Partialattribut).
Eigenschaft	Hier werden die Eigenschaften der Attribute angegeben. Diese werden im Informationssystem verarbeitet. Derzeit sind die Eigenschaften BGroup, BSearch, BCalc, BView, BUnique und BOuter vorhanden.
Wert	Die Werte TRUE oder FALSE geben zu einer Eigenschaft an, ob diese gültig ist oder nicht.

Tabelle 7: Inhalt der Repositorytabellen

3.2.2 DataSet und Mengenwerkzeug

Nachfolgend werden grundlegende Modellierungsaspekte und Mechanismen der Reportingplattform erläutert.

DataSet

Damit der spätere Anwender seine gewohnte Terminologie verwenden kann, wird seine Begriffswelt in einfach zu erfassenden Objekten, den *DataSets* spezifiziert. Ein DataSet stellt innerhalb der Reportingplattform den zentralen Datentyp für Informationsabfragen dar, der zunächst einmal vollständig unabhängig von der zugrundeliegenden Datenbank ist. Ein DataSet selbst hat keine Zugriffsmöglichkeit auf die vorhandene Datenbank, da in diesem nur Attribute und Filter gespeichert sind, die in einem selbstsprechenden und für den Anwender verständlichen Klartext formuliert sind. Die Attribute beschreiben über Einträge im Repository abstrakt einzelne Datenfelder. Innerhalb eines DataSets bilden sich aus den Attributen zwei Mengen, die Definitions- und die Selektionsmenge.

In der *Definitionsmenge* eines DataSet sind alle Attribute enthalten, die zueinander kompatibel bzw. über Links aus gemeinsamen Basistabellen zu gewinnen sind. Da aber unter Umständen die Anzahl zueinander kompatibler Attribute in der Definitionsmenge sehr groß sein kann, besteht für den Anwender die Möglichkeit, nur die ihn interessierenden Attribute in die *Selektionsmenge* des DataSets auszuwählen.
Die in einem DataSet potentiell integrierbaren Filter schränken die tatsächlich selektierten Attribute über beliebig komplexe Operatorverknüpfungen ein. Für die Filter gilt, wie auch für die Attribute, die vollständige Datenbankunabhängigkeit.

Jeder Filter kann einer Filterklasse zugeordnet werden, wobei drei Filterklassen zu unterscheiden sind:

- *Statische Filter*: Es existiert für jeden DataSet genau ein statischer Filter, dessen Lebensdauer identisch zu der der Menge ist. Innerhalb des statischen Filters ist wiederum ein beliebig komplexer dynamischer Filter enthalten, der durch den statischen Filter verwaltet wird.
- *Atomare dynamische Filter*: Ein atomarer dynamischer Filter besteht aus einem Filterattribut, welches kein Teilattribut ist, aus einem Operator und einem Vergleichswert. Jeder Vergleichswert besteht entweder aus einer Konstante oder einem Attribut und wird über den Operator mit dem Filterattribut verknüpft.
- *Zusammengesetzte dynamische Filter*: Ein zusammengesetzter dynamischer Filter besteht aus einer Menge von atomaren Filtern, die beliebig verschachtelt über die Operatoren UND, ODER sowie NICHT verknüpft sind. Dadurch ist die Möglichkeit der Formulierung von komplexen Filterprädikaten gewährleistet.

Die Spezifikation eines DataSets lautet wie folgt:

DataSet = *Attribute* × *Filter*
Attribute = *Attribut-set*
Attribut = *Char**

Die eigentliche Ergebnismenge, die über die Anfragesprache SQL gewonnen werden soll, wird mittels des *SQL-Managers* erzeugt, der die Verbindung eines DataSets, den darin enthaltenen Attributen und Filtern, zu den SQL-Funktionen der Datenbank herstellt. Der SQL-Manager generiert also aus den in den DataSets abstrakt formulierten Attributen und Filtern in Verbindung mit den Repositorydaten ein entsprechendes SQL-Statement, welches direkt an das RDBMS weitergegeben werden kann.
Ein DataSet kann daher als ein zentraler Datentyp verstanden werden, der die Attribute und die darauf angewendeten Filter (-einschränkungen) einer interessierenden Ergebnismenge in einer abstrakten Zwischendarstellung hält, welche wiederum über den *SQL-Manager* in ein konkretes SQL-Statement transformiert werden können. Abbildung 16 veranschaulicht nochmals den Sachverhalt:

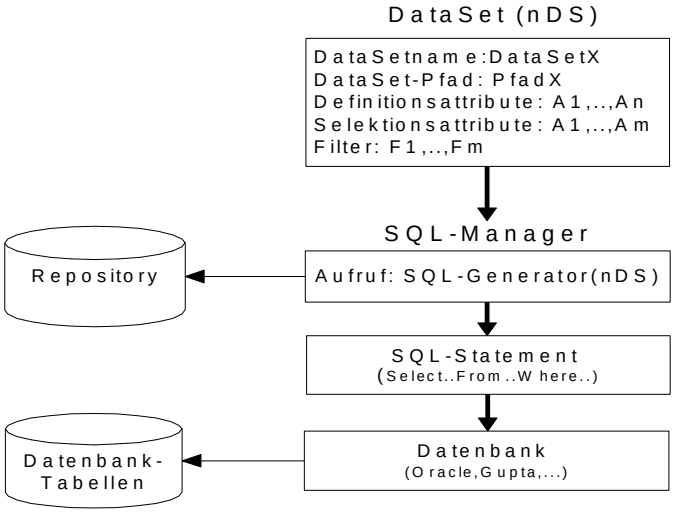


Abbildung 16: Transformationsweg eines DataSets zu einem SQL-Statement

Beispiel DataSet:

Attribute sind Begriffe, die in dem modellierten Anwendungsbereich die kleinste Einheit bilden, wie z.B. "Ladestellencode" oder "Ladestellenname". Der DataSet "Ladestelle", besteht aus den Attributen "Ladestellencode", "Ladestellenname" und "Ladestellen-PLZ / -ort / -straße". Der Filter "Ladestellen-PLZ = 20*" schränkt die Ergebnismenge des DataSet "Ladestellen" auf Ladestellen im PLZ-Gebiet "20" ein. Abbildung 17 stellt den beispielhaften Aufbau des DataSets graphisch dar.

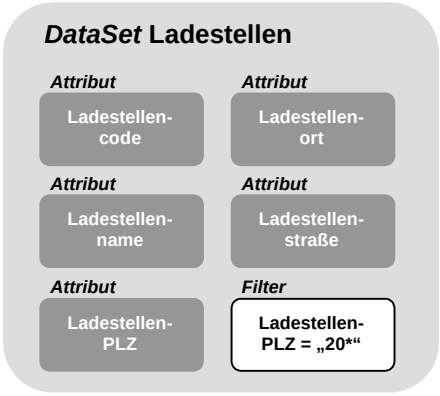


Abbildung 17: Zusammensetzung des Beispiel-DataSets "Ladestellen"

Durch den Mechanismus der Übersetzung eines DataSets in SQL mittels SQL-Generator können sämtliche SQL-Statements (inkl. Outer Joins, UNION) mit Ausnahme der IN, EXISTS und anderen Subqueries generiert werden.

```
SELECT *, Spaltenname, sämtliche Aggregationsfunktionen wie COUNT, SUM, AVG
FROM   Tabellen, Views
WHERE  Feldname <> Feldname AND/OR
       Feldname <OPERATOR>  ‘String’/Integer  AND/OR
       Feldname = Feldname (+)
GROUP BY 1, 2, 3
HAVING Feldname > 200
ORDER BY 2, 1  ;
```

Mengenwerkzeug

Das *Mengenwerkzeug* (*Mengentool*) dient der Erstellung eines DataSets. Über dieses Werkzeug wird eine Selektionsmenge für einen DataSet aus den Attributen des Repositorys gewählt, worauf automatisch die Definitionsmenge ermittelt und dem DataSet zugewiesen wird. Weiterhin besteht die Möglichkeit, dynamische Filter für die Menge zu formulieren, die z.Zt. allerdings noch in einer für den SQL-Generator interpretierbaren Form eingegeben werden müssen. Die Filter können nur als Wertebereichseinschränkungen für die Selektionsattribute formuliert werden. Zusätzlich kann die Datenmenge mit einem Namen versehen werden.

Über verschiedene Zuweisungmechanismen kann der erstellte DataSet anschließend an verschiedene Auswertungstools (Graphik-, Tabellen- und Kreuztabellentool) übergeben werden, wobei zuvor mittels des SQL-Generators die Transformation in ein SQL-Statement erfolgt, welches anschließend an die Datenbank abgesetzt wird.

Dynamische Einschränkung von DataSets

Damit der geforderte *Durchhangelmechanismus* innerhalb der Reporting-Sitzungen realisiert wird, können einzelne Elemente von Datenmengen (in Form von Zeilen) z.B. innerhalb des Tabellentools mittels Mausklick markiert werden. Diese so durch Markierung auf eine Teilmenge eingeschränkte Datenmenge impliziert eine Ergänzung des DataSets um einen weiteren dynamischen Filter und kann anschließend z.B. per Drag & Drop an ein anderes Tool übergeben werden. Das durch den SQL-Generator zu erzeugende SQL-Statement wird dann gemäß des neuen Filters adaptiert.

3.2.3 Realisierung der Datensicherheit

Zur Realisierung der notwendigen Sicherheitsanforderungen wird jeder Benutzergruppe eine Basis-bibliothek mit DataSets zugewiesen, die aufgrund der jeweiligen Benutzerrechte eingeschränkt wurde. Dabei handelt es sich analog zum Viewkonzept bei RDBMS um eine wertabhängige Zugriffskontrolle. Allerdings werden die Einschränkungen nicht auf Tabellenebene, sondern auf der Ebene der DataSets realisiert. Dazu werden bestimmte Filter so gesetzt, daß die zu generierende SQL-Abfrage eingeschränkt wird und nur der zugewiesene Ausschnitt an Daten abgefragt werden kann.
Innerhalb der Basisbibliothek eines Niederlassungsleiters werden z.B. sämtliche DataSets so modelliert, daß jeweils der Filter *Niederlassung = eigene* gesetzt ist. Dieser Filter ist für den Benutzer

nicht sichtbar und kann daher auch nicht verändert werden. Damit dieses gewährleistet ist, wird für das Attribut *Niederlassung* kein Eintrag in der Tabelle RP_Eigenschaften vorgenommen, der den Wert 'anzeigbar' als Eigenschaft enthält.

Ebenfalls ist es für die Benutzer nur möglich, die einzelnen DataSets mit Hilfe der Filter weiter einzuschränken, nicht aber zu erweitern. Selbst bei der Verknüpfung verschiedener DataSets und der daraus resultierenden neuen DataSets werden die ursprünglich gesetzten Filter jedesmal mit übernommen.

Da die einzelnen Links zwischen den Tabellen explizit definiert werden müssen, ist ebenfalls die Möglichkeit gegeben, auf dieser Ebene benutzerabhängige Rechte zu vergeben. Ein Benutzer kann nur Verknüpfungen über die in seiner Basisbibliothek modellierten DataSets durchführen, für die ihm Rechte zugewiesen worden sind. Hierdurch wird vermieden, daß ein Benutzer über eine Telefonnummer die dazugehörige Adresse abfragen kann. Der Link von Adresse zu Telefonnummer wird ihm zugänglich gemacht. Der umgekehrte Link wird jedoch nicht definiert.

Zusätzlich werden allen Attributen der DataSets Eigenschaften wie z.B. “suchbar“ zugewiesen. Hierdurch wird ebenfalls gewährleistet, daß eine Adresse nicht über die in der gleichen Tabelle abgespeicherte Telefonnummer lokalisiert werden kann.

3.2.4 Festdefinierte SQL-Statements

Da die bestehende Funktionalität des SQL-Generators eine Einschränkung bedeutet, wurde als weitere Entwicklung die Integration festdefinierter SQL-Statements in das DataSet-Konzept realisiert [Wink96]. Für einzelne DataSets können durch die Hinterlegung vordefinierter SQL-Anweisungen komplexere Anfragen, die der SQL-Generator nicht erzeugen kann, an das System gestellt werden.

Der Anwender wählt bei der Informationsabfrage einen DataSet aus. Bevor der SQL-Generator aufgerufen wird, der anhand des DataSets (Attribute und Filter) ein SQL-Statement generiert, wird geprüft, ob ein vordefiniertes SQL-Statement im System hinterlegt wurde. Trifft dies zu, wird als Ergebnis das gefundene SQL-Statement an das Darstellungswerkzeug der Reportingplattform übergeben. Der SQL-Generator wird in diesem Fall umgangen.

Auf der Basis der Attribute eines DataSets werden SQL-Anweisungen erzeugt. Es bietet sich deshalb an, Attributmengen den vordefinierten SQL-Statements zuzuordnen, damit anhand von Attributen erkannt werden kann, ob eine Anweisung im System hinterlegt wurde:

Attribute ® hinterlegte SQL-Anweisung

Diese Vorgehensweise ist jedoch mit Problemen verbunden, da eine Attributmenge ein hinterlegtes verschachteltes SQL-Statement beschreibt und eine Erzeugung durch den SQL-Generator zulassen kann. Eine Attributmenge kann so unterschiedliche SQL-Anweisungen identifizieren. Um dieses zu vermeiden, könnten vordefinierte SQL-Statements gemäß ihres Einsatzgebiets klassifiziert werden. Auch eine weitere Unterteilung in Gruppen und Untergruppen erscheint sinnvoll, um die Zahl der potentiellen SQL-Anweisungen, die durch eine Attributmenge identifiziert werden, zu verringern.
Die Klassifikation könnte durch die Angabe eines Pfades innerhalb des DataSets erfolgen. Je konkreter die Angaben innerhalb des Pfad sind, desto kleiner ist der Suchraum für die Attributmenge und die dafür hinterlegten SQL-Anweisungen. Folglich muß der DataSet erweitert werden, um Mehrdeutigkeiten aufzulösen.

$$DataSet = Attribute \times Filter \times Pfad$$
$$Pfad = Char^*$$

SQL-Anweisungen werden hierarchisch, gemäß ihrer Zugehörigkeit zu einem Bereich, im Repository hinterlegt (siehe Abbildung 18). Die Darstellungsform ist eine Baumstruktur. Das Setzen des DataSet-Pfads "Disposition/Kontrolle" würde den Suchraum auf den unteren linken Teilbaum mit der Wurzel "Kontrolle" beschränken. Navigiert man von der Wurzel zu einem Blatt, so läßt sich für jedes

hinterlegte SQL-Statement eine ID bestimmen, wobei die Bezeichnungen der besuchten Knoten (sowie des Blattes) zusammengesetzt werden. Formal läßt sich die ID so beschreiben:

Klasse[/Gruppe[/Untergruppe]]/StmtName*

Die eckigen Klammern stehen für optionale Angaben, der Stern für eine beliebige Wiederholung. Erforderlich sind die Angabe einer Klasse und des *Statementnamens*.

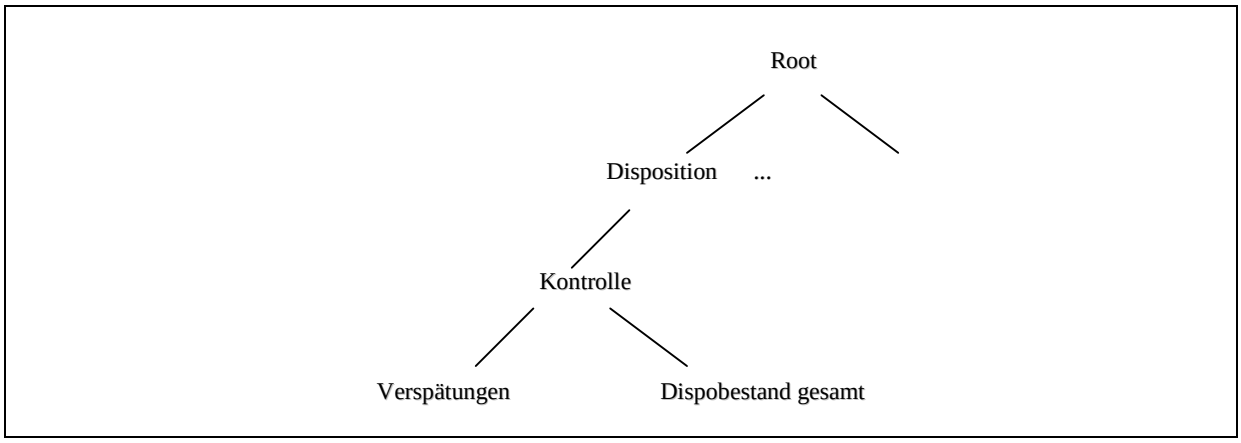


Abbildung 18: Hierarchische Einordnung festdefinierter SQL-Statements

So könnte jede Anweisung über einen Editor eingegeben und ihr eine ID und ein Pfad zugeordnet werden. Dies ist allerdings zu starr und unflexibel. Deshalb werden folgende Anforderungen gestellt:

a) *Einbringen zusätzlicher Filterinformationen* - Das Filtern von Attributen ist im Zusammenhang mit dem SQL-Generator bereits möglich. Filter werden im Mengentool erzeugt und lassen sich im DataSet hinterlegen. Filterinformationen müssen sich dynamisch in hinterlegte SQL-Anweisungen integrieren lassen.
b) *Änderung der Anzeigenreihenfolge* - Die Reihenfolge der Spaltenauswahl bzw. Darstellung sollte nicht fest vorgegeben sein, sondern der Reihenfolge der Hinterlegung der Attribute im DataSet entsprechen. Attribute können auch "unsichtbar" sein, werden also nicht angezeigt.
c) *Wiederverwendbarkeit* - Viele SQL-Anweisungen gleichen sich und sind nur in einigen Details voneinander unterscheidbar. Deutlich wird das bereits bei den SQL-Anfragen zur Dispositionskontrolle. Diese Thematik (SQL-Templates) wird im folgenden entwickelt.
d) Abgesehen von der Wiedereinsetzbarkeit läßt sich durch Anwendung von Templates auch eine redundanzarme Hinterlegung von SQL-Anweisungen erreichen.

An dieser Stelle sollen keine weiteren Erläuterungen zum Konzept der festdefinierten SQL-Statements erfolgen, sondern gemäß dem Fokus dieser Arbeit im folgenden auf die Umsetzung des Templatekonzepts eingegangen werden. Für eine detaillierte Darstellung der festdefinierten SQL-Statements sei auf [Wink96] verwiesen.

3.2.5 Zusammenfassung

Durch die Verwendung von DataSets in Verbindung mit den Metadaten des Repositorys, als modellierte Abfragen, die der Anwender einfach adaptieren kann und so bei Bedarf mittelbar neue SQL-Statements erstellt, wird ein Teil des möglichen Informationsbedarfs des Anwenders mit der gewünschten Flexibilität abgedeckt. Der Anwender kommt nicht mit SQL in Berührung ist aber in der Lage die gewünschten Abfragen zusammenstellen. Die Verwendung festdefinierter SQL-Statements ermöglicht dem Modellierer die Bereitstellung des gesamten Informationsbedarfs für den Anwender, allerdings mit dem erhöhten Modellierungsaufwand und dem Manko, daß die festdefinierten Statements nicht bei Bedarf durch den Anwender zu adaptieren sind. Aus diesen Gründen sind weitgehendere Konzepte notwendig, die die hohe Flexibilität bei der Zusammenstellung neuer Informationsabfragen beibehalten und insbesondere die Wiederverwendbarkeit bereits modellierter SQL-Statements gewährleisten (siehe *Kapitel 3.5.2 Das SQL-Templatekonzept*).

3.3 VERWENDUNG VON COMPONENTWAREKONZEPTEN

Nachfolgend wird nun eine Erweiterung des Systems um Componentwarekonzepte entwickelt. Dazu wird eine Componentwarestruktur als Softwarearchitektur entworfen, die die generische Verwendung von Prozeßbausteinen und Informationsabfragen sowie externen Programmen als Komponenten im Sinne von Componentware ermöglicht und auf den Grundlagen des entwickelten Modells aufbaut.

3.3.1 Anforderungen an die Componentwarekonzepte

Für die Verwendbarkeit der Konzepte aus dem Componentwarebereich muß das System die vom Benutzer gewünschten Verknüpfungsarten der Komponenten ermöglichen, wobei dies für den Benutzer transparent zu geschehen hat. Sofern notwendig, ist ein Datenfluß zwischen den Komponenten zu gewährleisten. Komponenten sollen auch ohne eine Kooperation mit anderen Komponenten verwendbar sein. Innerhalb einer Komponente sollten Funktionalitäten zusammengefaßt werden, die i.a. gemeinsam verwendet werden.
Verschiedene Nebenbedingungen können sich auf die Verwendbarkeit von Komponenten in einem bestimmten Kontext auswirken. Das System muß deshalb mittels Metadaten / Integritätsbedingungen bzgl. der möglichen Verwendbarkeit der einzelnen Komponenten unterstützen können.
Es muß die ad hoc-Verknüpfung der Komponenten zur Laufzeit möglich sein, ohne daß ein Neukompilieren erforderlich ist. Damit nicht für jeden Teilprozeß / jedes SQL-Statement eine eigene Komponente modelliert werden muß, ist eine Möglichkeit zu schaffen, Komponenten zur Laufzeit entsprechend der Anforderungen anzupassen. Für diese *Adaption* existieren unterschiedliche Alternativen [GuPo98], wie *Vererbung*, *Erweiterung durch Extension* oder *Parametrisierung*.
Diese Anforderungen lassen sich nochmals wie folgt zusammenfassen:

- Formale Spezifikation, Beschreibungssprachen.
- Komposition von Komponenten.
- Lose Kopplung und maximale Kohäsion.
- Beherrschung der Nebenbedingungen.
- Flexible Adaptierbarkeit zur Laufzeit.

3.3.2 Rahmenbedingungen zur Verwendung von Componentware

Componentware ist für unterschiedlichste Anwendungsbereiche gut geeignet, birgt jedoch auch einige Risiken und Probleme. Wenn man z.B. die drei Sichten *Anwendungssicht*, *Benutzersicht* und *Systemsicht* auf ein Informationssystem [Thal97] in Betracht zieht (siehe Abbildung 19), so sind folgende Fragestellungen zu berücksichtigen:

- Können alle Belange der Anwendung durch die Kombination der Komponenten erschlossen werden (*Anwendungssicht*) ?
- Paßt alles so zusammen, wie es geplant worden ist (*Systemsicht*) ?
- Ist das entstandene System einfach und intuitiv im beabsichtigten Rahmen benutzbar (*Benutzersicht*) ?
- Ist die Stabilität und Robustheit des Systems bei beliebiger Kombinierbarkeit gewährleistet (*Orthogonalität*) ?

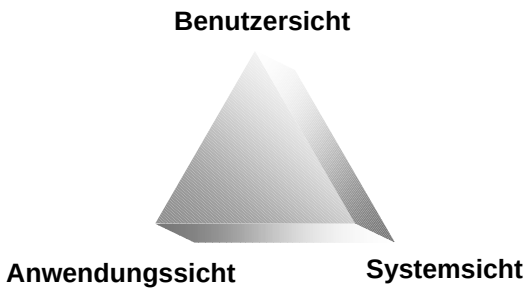


Abbildung 19: Verschiedene Sichten auf ein Informationssystem

Ebenso hat die Wahl der Größe zu verwendender Komponenten einen entscheidenden Einfluß darauf, wie exakt eine Anpassung der Componentware auf den Anwender möglich ist. Je größer die

verwendeten Bauteile sind, desto geringer ist die Paßgenauigkeit und desto seltener werden sie in unterschiedlichen Domains verwendet [Küff94]. Zu kleine Komponenten lassen sich nur schwer von angrenzenden Komponenten differenzieren. Ein hierdurch aufgebautes Anwendungssystem verfügt über zu viele Schnittstellen zwischen den Komponenten, die eine ausreichende Entkopplung der Komponenten verbietet und ist aufgrund fehlender Transparenz nicht mehr handhabbar.

So ist die Auswahl von Komponenten einer Componentware als Mechanismus für die Anpassung von großen ERP-Anwendungssystemen, wie z.B. SAP R/3, ungeeignet und muß durch geeignete Mechanismen weiter unterstützt werden. Die Vorkonfiguration des Anwendungssystems mittels geeigneter Komponenten ist jedoch sinnvoll [GuPo98]. Der modulare Aufbau und wohldefinierte Schnittstellen erlauben auch die Einbindung von Komponenten anderer Hersteller.

Sofern Komponenten kleiner werden, muß der Anwender bei der Auswahl geeigneter Komponenten unterstützt werden, mit deren Erhöhung der Transparenz eine schnelle Montage möglich ist. Sollen Komponenten nicht nur in dem Anwendungssystem verwendet werden, für das sie konzipiert wurden, so sind weitere Anforderungen an das Design zu stellen. Dazu sollte der Bedarf nach Wiederverwendbarkeit bestehen und die Komponente sollte von ihrem Aufbau her den Einbau in anderen Bereichen unterstützen. Wichtigste Voraussetzung ist jedoch eine *lose Kopplung* innerhalb der Componentware. Dazu könnte die Kopplung nicht über Objektklassen, sondern über eine Parameterübergabe und die Verwendung von Design Patterns erfolgen.

3.3.3 Umsetzung der Componentwarekonzepte

Die Struktur der Komponentenarchitektur wurde in Form des Design Patterns Template modelliert, damit die Komposition der verschiedenen Komponenten flexibel ermöglicht wird. Die Steuerung der gesamten Anwendung wird über ein komponentenbasiertes Framework realisiert. Damit nicht für jede Variante eine Komponente modelliert werden muß, erfolgt die Adaption der Komponenten durch Parametrisierung als eine Technik der Spezialisierung. Parametrisierung umfaßt die Auswahl von Implementierungsvarianten eines Anwendungssystems. Dies wird durch das Setzen bestimmter Parameter mit vordefinierten Wertebereichen durchgeführt [GuPo98].

Die jeweils eingehenden Parameter initiieren eine Adaption der Templates, indem verschiedene Komponenten in das Algorithmusskelett (mit "Löchern" für die Einschubmethoden, den Hot Spots), eingefügt werden. Grundlage für die Adaption bilden neben den Parametern die im Rahmen der Domainmodellierung abgelegten *Metadaten* und *Integritätsbedingungen*. Dabei werden durch den IT-Spezialisten die anwendungsneutralen Komponenten als Basis für die spätere Verwendung erzeugt (Frozen-Spots). Die Überführung in anwendungsspezifische Komponenten erfolgt zur Laufzeit über die Parametrisierung.

Für die Verknüpfung der Komponenten wurde als Glue eine Skriptsprache definiert, die zur Steuerung des Prozeßverhaltens *Kontrollkonstrukte* nutzt, wie *Bedingungen* und *Iterationen*. Eine Skriptkomponente kapselt eine Menge partiell gebundener Komponenten und determiniert die für weitere Skripte sichtbaren Schnittstellen (*Ports*). Diese Ports *parametrisieren* sozusagen das Verhalten der Skripte / Komponenten [Nie+90].

Bei der Anpassung der SQL-Statements wird durch die Skriptsprache die Parametrisierung der SQL-Templates gesteuert. Die möglichen Verknüpfungsarten zwischen den Komponenten bzw. deren Adaptierbarkeit werden ebenfalls über Integritätsbedingungen und Metadaten determiniert. Die in ein Skript eingebetteten Komponenten bilden nun in ihrer Gesamtheit wiederum neu verwendbare Komponenten, die mit anderen Skriptkomponenten verknüpft werden können. Die Kopplung ermöglichen Input- und Outputports, über die insbesondere auch der mögliche Datenfluß zwischen den Komponenten abgewickelt wird. Jede Komponente wird mit der Granularität modelliert, daß sie zusammengehörige Funktionen enthält, die einem gemeinsamen Zweck dienen (maximale Kohäsion). Alle Komponenten oder auch Komponententeile sind für sich allein ablauffähig (minimale Kopplung).

Die Übergabe dieser im Sourcecode vorgehaltenen Komponenten an das Laufzeitsystem erfolgt erst zum Zeitpunkt des Komponentenaufrufs. Hierdurch kann zur Laufzeit auf eintretende Exceptions

reagiert werden, da keine Instanziierung des gesamten Prozesses durchgeführt wird und vorab kein Kompilieren von SQL notwendig ist. So kann ein vollständig spezifiziertes Prozeßtemplate zur Laufzeit verändert werden, da nicht alle Komponenten zu Beginn in eine Gesamtprozeßinstanz übernommen wurden, sondern erst zum Zeitpunkt ihres Aufrufs ggf. adaptiert und übersetzt werden.

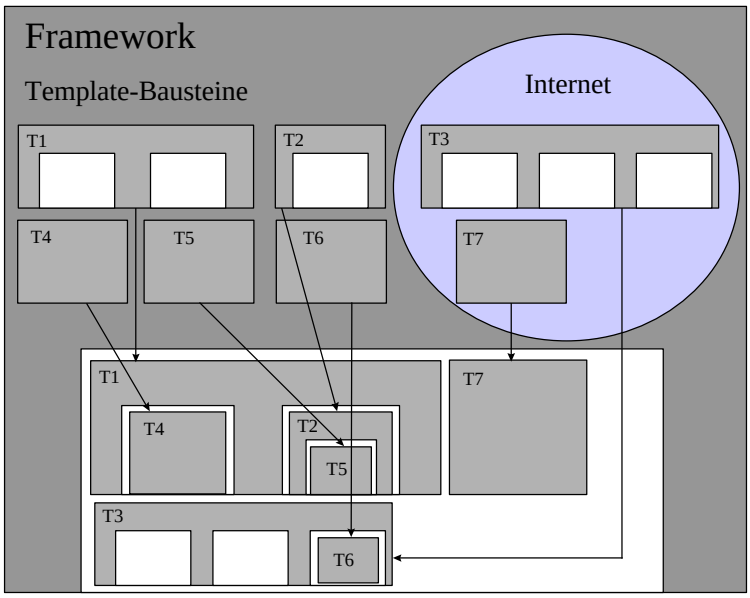


Abbildung 20: Templatebausteine des Frameworks

Die Zusammenstellung der einzelnen Komponenten in Abbildung 20 veranschaulicht, daß Templates auch aus anderen Templates aufgebaut sein können. Templatebausteine, die keine weitere Möglichkeit der Integration anderer Bausteine haben, können nur als eine Einheit verwendet werden (T4 bis T7 in Abbildung 20). Hierbei handelt es sich um Prozeßbausteine und festdefinierte SQL-Statements mit kleinster Granularität (z.B. ein Elementarworkflow oder eine SQL-Subquery) sowie externe Programme, die keine Spezialisierung mehr zulassen. Für zukünftige Anforderungen sollte eine lokationstransparente Kooperation z.B. via Internet ermöglicht werden. In diesem Falle sind entsprechende Middlewarekonzepte wie CORBA zu integrieren.

Die Adaption der Frameworkbausteine / -templates erfolgt über Parametrisierung. Die Parameter werden über die Problemspezifikation bzw. weitere Anpassungswünsche in den jeweiligen Dialogen mit dem Anwender erzeugt. Der Parametereingang im jeweiligen Template führt mit gleichzeitiger Bezugnahme auf Metadaten zu der gewünschten Veränderung bzw. Komposition. Metadaten umfassen im Bereich der Prozeßbausteine z.B. die Modellierung des Datenflusses in Form einer Schnittstellendefinition zur möglichen Datenübergabe zwischen den einzelnen Komponenten sowie Regeln zur Verknüpfung von Prozeßbausteinen mit der entsprechenden Konsistenzprüfung.

Das Framework kontrolliert als Steuerungskomponente den Gesamtprozeß. Die vorhandenen anwendungsneutralen Komponenten in Form modellierter Prozeßbausteine / SQL-Statements werden durch die Parametrisierung in anwendungsspezifische Komponenten transformiert (siehe Abbildung 21). Prozeßbausteine werden nicht immer ein Startereignis und ein Endereignis besitzen, sondern auch Alternativen bei den einzuschlagenden Prozeßpfaden zulassen bzw. AND / OR / XOR-Verknüpfungen enthalten, die nicht wieder zusammenfließen. Wenn jetzt eine Verknüpfung mehrerer Prozeßbausteine erfolgt, muß ein konsistenter Übergang von einem Prozeßbaustein zum nächsten gewährleistet sein. Bei mehreren Alternativen ist der Anwender über Dialogboxen mit einzubeziehen. Zur Überwachung dieser Übergänge dienen die Integritätsbedingungen.

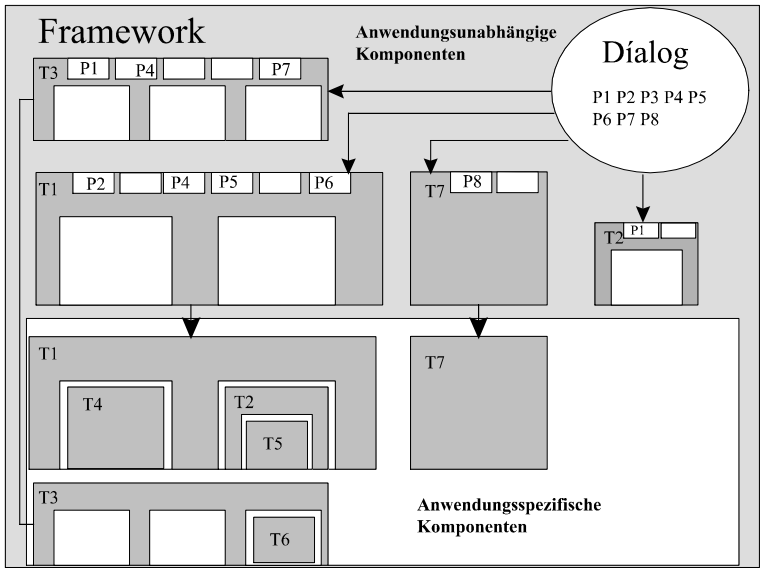


Abbildung 21: Templatebausteine des Frameworks mit Parametrisierung

3.4 TEMPLATES

Durch die Verwendung von Templates als interne Struktur der verschiedenen Komponenten wird eine hohe Flexibilität in bezug auf die Behandlungen von Exceptions zur Laufzeit erreicht. In die Komponenten eingehende Parameter bilden den jeweiligen Kontext und initiieren die Adaption der Templates. Die Templates werden in einer Skriptsprache definiert und können anschließend ebenfalls wieder als Komponenten verwendet werden. Die allgemeine Templatestruktur gliedert sich wie folgt:

DO	<Typ >	SQL- oder Prozeßtemplate
	<Inputparameter>	Datenelemente, Prozeßelemente, Spalten, Komponenten
WITH	<Komponenten>	Komponenten, Prozesse, Tabellen im <from-clause>
UNDER CONDITIONS	<Bedingungen>	<where-clause + order-by-clause + having-clause |, Kontrollflußkonstrukte, Integritätsbedingungen

Der <Typ> bestimmt die weitere Spezialisierung bei der Adaption des Templates. Dabei kann es sich um SQL- oder Prozeßtemplates handeln, die auf unterschiedliche Arten übersetzt werden müssen. Oder es handelt sich um elementare SQL- oder Prozeßbausteine, die nicht weiter adaptiert werden können. Im Rahmen einer Verallgemeinerung des Konzepts könnten auch weitere Design Patterns mit Verwendung finden, deren Adaption jedoch noch zu spezifizieren wäre. In <Inputparameter> werden die einzelnen Parameter spezifiziert, die die Adaption steuern. Diese bestehen aus Komponenten, Datenelementen und Metadaten z.B. in Form zusätzlicher Bedingungen, die über den Dialog mit dem Benutzer spezifiziert wurden (-> DataSet). Nachfolgend werden die einzelnen möglichen Strukturelemente nochmals in einer *BNF-Notation* spezifiziert.

Typ::=	Template | Elementarprozeß | Elementar-SQL-Statement | Patterntyp
Inputparameter::=	Spalten | Komponenten | Datenelement | Prozeßelement
Datenelement::=	Datenobjekte, die durch Komponenten fließen
Prozeßelement::=	eingehende Prozeßwegweiser, die ausgehende Prozeßpfade repräsentieren | Boolsche Verknüpfung der eingehenden Pfade
Komponente::=	Komponente | Template
Template::=	Komponente | Template | Elementarprozeß | Elementar-SQL-Statement

Bedingungen::=	Constraints | <where-clause + order-by-clause + having-clause | Kontrollflußkonstrukte | Integritätsbedingungen

Für die weiteren Ausführungen wird die o. g. Darstellung der Templates in eine parametrisierte Form überführt, da sich so die einzelnen Mechanismen besser erläutern lassen.

Template-Name	*Template-x*	
DO	[T]	<Typ>: Prozeß / SQL
	$[P_1]$, $[P_2]$,[...],$[P_n]$	<Inputklasse>:Komponenten, Datenelemente etc.
WITH	$[B_i]$, $[B_{i+1}]$,[...],$[B_k]$,	<Komponenten>: festdefiniert
	$[B_m]$, $[B_{m+1}]$,[...],$[B_q]$	<Komponenten>: variabel
UNDER CONDITION	$[C_i]$, $[C_{i+1}]$,[...],$[C_k]$,	<Bedingungen>: festdefiniert
	$[C_m]$, $[C_{m+1}]$,[...],$[C_q]$	<Bedingungen>: variabel

Festdefinierte Komponenten und festdefinierte Bedingungen sind Teile des Templates, die nicht verändert bzw. weggelassen werden können, da sie im Rahmen der Domainmodellierung für diesen Bereich spezifiziert wurden.

Der dynamische Aspekt der Adaption wird durch die *variablen Bausteine* und *variablen Bedingungen* abgedeckt. Bei den Bausteinen kann es sich wiederum um Templates handeln, so daß im Bereich der Prozeßtemplates eine Kaskadierung möglich ist. Ebenso kann der Anwender an jedem Prozeßknoten zusätzliche SQL-Statements in Form von SQL-Templates hinterlegen, die ihn im Rahmen seiner erfahrungsbasierten Problemlösung unterstützen. Bei der Anpassung bzw. dem Einfügen der verschiedenen Bausteine ist es i.d.R. notwendig, zusätzliche Bedingungen / Bausteine mit aufzunehmen, die durch das System determiniert werden.

Nachfolgend wird das Templatekonzept zunächst anhand von SQL-Templates erläutert. Anschließend folgt die Beschreibung der Prozeßtemplates.

3.5 SQL-TEMPLATES

3.5.1 Übersicht

Konzeptuelle Grundlage des SQL-Templatekonzepts bilden *query forms* [Thal00, S.253 ff.], die zur Modularität und Wiederverwendbarkeit von SQL-Abfragen beitragen. Query forms sind parametrisierte Spezifikationen bestimmter Anfrageklassen, die über Import- und Exportschnittstellen verfügen. Dies ist im Rahmen der Softwareentwicklung und der damit verbundenen Modulspezifikation als Konzept bereits allgemein bekannt. Eine query form kann in ihrer einfachsten Form wie folgt aussehen:

query (data) oder $query_1\ (query_2)$

Des weiteren kann eine Anfrage q mit beispielsweise drei Parametern wie folgt dargestellt werden:

$$q: \text{Data} \times \text{Data} \times \text{Data} \rightarrow \{\text{Data}\}$$

Die Schnittstelle einer Anfrage mit drei Parametern und einem Ausgabeparameter in Form einer Menge von Strings könnte dann beispielsweise den folgenden Aufbau haben:

$$q_1(q_2) : \text{String} \times \text{SmallInt} \times \text{String} \rightarrow \{\text{String}\}$$

Nachfolgend wird der Gedanke der query forms hin zum SQL-Templatekonzept erweitert.

So wird dem Begriff SQL-Template innerhalb dieses Konzepts die Bedeutung einer *Typschablone* zugewiesen, die in der Lage ist, ein parametrisiertes SQL-Statement als einen Datentyp aufzunehmen.

Ein SQL-Template ist also eine Schablone in Form eines Basisstatements, die aus den bekannten SELECT- / FROM- / WHERE-Klauseln mit bereits vorbelegten Spalten-, Tabellenbezeichnungen und Bedingungen besteht und aufgrund ihres Kontexts zu vollständigen Statements ergänzt werden. Der Aufbau eines SQL-Statements wird nachfolgend genauer spezifiziert.

Dem Templatekonzept kommt die Funktion einer Erweiterung bzw. Flexibilisierung der o.g. festdefinierten Statements zu. Die höhere Flexibilität der Templates im Unterschied zu festdefinierten Statements wird durch die Parametrisierung der Attribute (implizit der zugehörigen Tabellenlinks und Joins), sowie der attributbezogenen Filter und der Einbindung von über Mengenoperatoren verknüpften Subqueries erreicht. Durch die Parametrisierung der Templatestatements können Statements, die in ihrer grundlegenden Struktur Gemeinsamkeiten aufweisen zu einem komplexen Template als Basis zusammengefaßt werden, selbst wenn sie nur in einigen Teilen des Statements variieren. Die damit verbundene Aufgabe beinhaltet das Problem, zunächst einmal alle aus einem bestimmten Informationsbedarf resultierenden Statements zu spezifizieren. Im Anschluß daran sind die Statements auf das Vorhandensein gleicher Bestandteile zu untersuchen.

An den Templatemechanismus sind die Anforderungen zu stellen, ein konkretes Statement nach vorhergehenden Tests in die einzelnen Templatebausteine zerlegen und parametrisieren zu können. Die so parametrisierten Bausteine sind anschließend durch individuelles Zusammenstellen zu einem syntaktisch und semantisch korrekten Statement zusammenzuführen.

DataSets bilden ebenfalls ein Template / eine Schablone auf höherer Ebene, da aufgrund der jeweils ausgesuchten bzw. zugeordneten Attribute / Filter / Sortierung ein entsprechendes SQL-Statement erzeugt wird. Der Anwender kann aufgrund von Eingaben an speziell vorgehaltenen Slots Einfluß auf das zu generierende SQL-Statement nehmen. Sofern in dem Slot für SQL-Templates (aufgrund der vorhergehenden Modellierung) systemseitig ein SQL-Template eingefügt wird, ist dieses in die SQL-Generierung einzubeziehen.

Ebene: DataSets (Anwenderseite)

```
Name:            DataSet_X              auswählbar bzw. veränderbar

Attribute:       [1], [...], [n]        auswählbar

  [SQL-Template: [y]   ]                vom Anwender nicht veränderbar

Filter:          [k], [..], [n]         auswählbar bzw. veränderbar

Sortierung:      [1], ..., [n]          auswählbar bzw. veränderbar
```

Der Anwender hat die Möglichkeiten, Attribute zu verändern bzw. auszuwählen, zusätzliche Filter einzufügen und ggf. mittels Konstanten zu erweitern, sowie die Sortierung aufgrund der gewählten Attribute zu verändern. Des weiteren muß er nicht unbedingt einen DataSet auswählen, sondern kann auch nur Attribute wählen und zu einem neuen DataSet zusammenfügen.

Das ggf. zugrundeliegende SQL-Template kann er jedoch nicht verändern. Dieses wird kontextbezogen systemseitig adaptiert und anschließend mit den von ihm ausgewählten bzw. verändernden Eingaben verknüpft. Die Metadaten im Repository definieren die Abbildung der DataSet-Attribute auf die Tabellenspalten und den dazugehörigen Joins zwischen den einzelnen Tabellen.

Ebene: SQL-Templates
Sofern zu den ausgewählten Attributen bzw. dem gewählten DataSet kein passendes SQL-Template existiert, wird durch den SQL-Generator, auf den Metadaten des Repository basierend, ein SQL-Statement erzeugt.

Die SQL-Templates haben den folgenden Aufbau:

Template-Name	*SQL-Template-y*	
DO	[T]	<Typ>: SQL
	$[P_1]$, $[P_2]$,[...],$[P_n]$	SELECT-CLAUSE, Tabellenspalten,
WITH	$[B_i]$, $[B_{i+1}]$,[...],$[B_k]$,	FROM-CLAUSE, festdefinierte Tabellen
	$[B_m]$, $[B_{m+1}]$,[...],$[B_q]$	FROM-CLAUSE, variable Tabellen
UNDER CONDITION	$[C_i]$, $[C_{i+1}]$,[...],$[C_k]$,	WHERE-CLAUSE, festdefinierte Filter
	$[C_m]$, $[C_{m+1}]$,[...],$[C_q]$	WHERE-CLAUSE, variable Filter
		[GROUP BY][HAVING][ORDER BY] variabel, je nach Parametereingang

Abbildung 22 zeigt anhand eines Beispiels die Übersetzung eines DataSets mittels zugeordnetem SQL-Template und Metadaten in ein SQL-Statement [Riec98]. Die Mechanismen für diese Übersetzung werden in den folgenden Kapiteln erläutert.

Der *DataSet1* wird seitens des Benutzers ausgewählt. Das System hat das zugeordnete *Template12* ermittelt und paßt dieses nun mittels der modellierten Metadaten des Repositorys an. Die Regeln I - IV umfassen die Zuordnung der Attribute zu den SELECT-Spalten der zugrundeliegenden Tabellen. Da die SELECT-Spalte S7 nicht im Template vorhanden ist, wird die Metaregel VI ausgewertet. Diese bestimmt, daß im Falle der aufgenommenen Spalte S7 die Tabelle Tab3 in die FROM-Klausel aufgenommen werden muß. Zusätzlich ist der Join zwischen den Tabellen Tab1 und Tab3 über die Tabellenspalten S9 (Tab1) und S11 (Tab3) in der WHERE-Klausel mit aufzunehmen.

Ebenfalls muß bei Auftreten von Filter1 (z.B. „LKW NICHT einsatzbereit„) eine Subquery mit eingefügt werden (Metaregel VII). Im Rahmen der ODER-Verknüpfung von Attribut4 wird der HAVING-Operator verwendet (Metaregel VIII), wobei ‘ODER’ durch ‘OR’ ersetzt wird (Metaregel V) und die vom Benutzer eingegebenen Werte in die Variablen x und y eingesetzt werden.

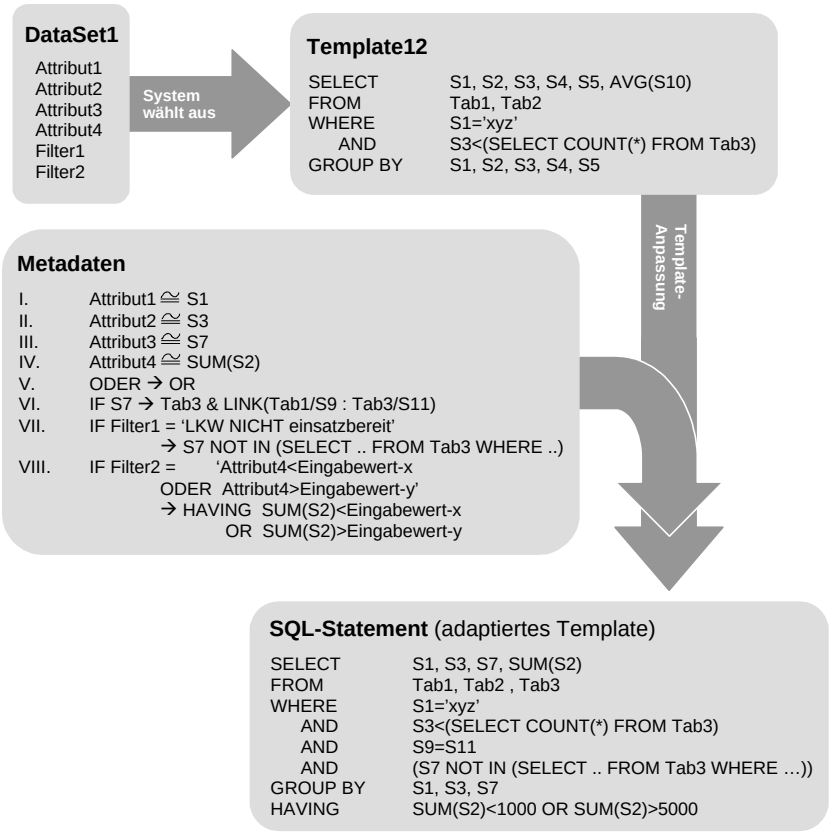


Abbildung 22: Übersetzung eines DataSets in ein SQL-Statement

3.5.2 Das SQL-Templatekonzept

3.5.2.1 Parametrisierung eines Basisstatements

Zu Beginn der Ausführungen wird die grundlegende Struktur eines SQL-Statements betrachtet:

- In der *SELECT-Klausel* sind Attribute enthalten, die direkt aus einer Basistabelle (atomar) stammen, oder indirekt über ein Fremdattribut aus einer Fremdtabelle referenziert werden.
- In der *FROM-Klausel* sind die Tabellenbezeichnungen enthalten, die für die ausgewählten Attribute als Basis- oder Fremdtabelle dienen.
- In der *WHERE-Klausel* werden bei Auswahl von Attributen, die eine Referenz auf ein Attribut einer Fremdtabelle aufweisen, die daran beteiligten Schlüssel- bzw. Fremdschlüsselbeziehungen zur Verknüpfung der betroffenen Tabellen aufgeführt.

Die aufgeführten Komponenten eines SQL-Statements erzeugen eine Anfrage in minimalistischer Form. Deshalb werden sie als parametrisierbarer Basisbaustein eines SQL-Templates definiert. Diese Struktur eines Basisstatements ist insofern parametrisierbar, als daß die in der SELECT-Klausel eingebundenen Attribute fest mit ihrer direkten oder indirekten Tabelle verknüpft sind, während die für die beteiligten Tabellen notwendigen Verknüpfungen in der WHERE-Klausel enthalten sind. So ist eine generische Auswahl von Attributen seitens des Benutzers möglich, da automatisch die dazugehörigen Tabellen und deren Verknüpfungen in den Basisbaustein eingefügt werden können. Zuvor muß diese interne Verknüpfungsstruktur analysiert werden, um eine korrekte Zuordnung zur Laufzeit vornehmen zu können. Um die Informationen über die interne Struktur zu erhalten, ist die Tabellenstruktur des Repositorys nachzuvollziehen, da in diesem die erforderlichen Informationen gespeichert sind. Hierzu kann der bereits entwickelte SQL-Generator zur Lösung herangezogen werden, da über dessen Funktionalität (Erzeugung eines *Defaultstatements* aus einer Menge ausgewählter Attribute) ein aus den Repositorydaten abbildbares Basisstatement generierbar ist. Hierdurch wird das Auslesen der Repositorystrukturen bzw. -tabellen umgangen.

3.5.2.2 Erweiterung um komplexere parametrisierbare Bausteine

Damit komplexere SQL-Statements über Templates erzeugt werden können, wird auf die parametrisierbare Basiskomponente aufsetzend geprüft, welche sinnvollen Möglichkeiten und Forderungen es an weitergehende parametrisierbare Bausteine gibt. Innerhalb dieses Konzepts werden nur solche Bausteine betrachtet, die aufgrund ihrer Komplexität in ein Template integrierbar sind, da durch das Templatekonzept nicht die gesamte Bandbreite aller potentiell generierbaren SQL-Statements erzeugbar ist. Bei Bedarf können die bereits integrierten festdefinierten SQL-Statements verwendet werden.

Attribute in Form einer Aggregationsfunktion

Es besteht die Möglichkeit Attribute auszuwählen, die im Repository als Aggregationsfunktionen spezifiziert worden sind. Des weiteren könnte sich bei der Modellierung eines Statements auch der Bedarf an einer individuell gestalteten Aggregationsfunktion ergeben, die nicht zur Auswahl über Attribute im Repository zur Verfügung steht. In diesem Falle sind Mechanismen zur Spezifikation sowie deren Abbildung auf SQL bereitzustellen.

Integration einer GROUP BY-Klausel

Eine Vielzahl von Informationswünschen bezieht sich auf gruppierte Daten, die gleiche Schlüsselwerte besitzen und daher zusammengefaßt auszugeben sind. Deshalb besteht die Forderung nach einer integrierbaren GROUP BY-Klausel, deren Parameter sich aus den Attributen der SELECT-Klausel ergeben, ohne Einbeziehung von Spalten, die eine Aggregationsfunktion enthalten.
Bei der Modellierung eines Templates muß deshalb geprüft werden, ob die Attribute, welche in die SELECT-Klausel eingebunden worden sind, eine Aggregationsfunktion darstellen. In diesem Fall ist die Integration einer GROUP BY-Klausel erforderlich.

Integration einer HAVING-Klausel

Die HAVING-Klausel erweitert ein Statement um die Möglichkeit der bedingten Gruppierung. Die HAVING-Bedingung kann nur im Zusammenhang mit einer in der SELECT-Klausel enthaltenen Aggregationsfunktion verwendet werden, um für den Ergebniswert dieser Aggregationsfunktion bzgl. einer Gruppe einen Vergleichswert zu suchen. In dem Fall, daß außer einer Aggregationsfunktion noch weitere nicht aggregierte Attribute in der SELECT-Klausel integriert sind, muß eine GROUP BY-Klausel vor der HAVING-Klausel in das Statement eingebunden werden, in der die nicht aggregierten Attribute als Parameter enthalten sind.
Da in dem Repository die Attribute beschreibenden Informationen hinterlegt sind, können diese Attributdeskriptoren bei der Modellierung eines Templates verwendet werden. So kann bei der Integration bestimmter Attribute in ein Template aus dem Repository die Information ausgelesen werden, ob es bei der Verwendung in einem Filterausdruck in dem HAVING-Teil des Statements plaziert werden muß.

Integration einer ORDER BY-Klausel

Bei der Ausgabe der Attribute, die durch die Spaltenbezeichner in der SELECT-Klausel bestimmt werden, besteht die Anforderung, diese in sortierter Reihenfolge auszugeben. Hieraus resultiert die Forderung zur Einbindung der ORDER BY-Klausel in ein SQL-Template. Als Sortierkriterium kann jede Spalte in der SELECT-Klausel vereinbart werden, wobei zwischen der Sortierung von Ziffern und Zeichenfolgen unterschieden wird.
Die Integration der ORDER BY-Klausel in ein SQL-Template sollte durch den Modellierer bzw. Anwender individuell entschieden werden, wobei insbesondere über die konkrete Auswahl der zu sortierenden Spalten in der SELECT-Klausel zu entscheiden ist. Da aber bereits im Rahmen des Konzepts der DataSets die Funktionalität implementiert worden ist, eine ORDER BY-Klausel dynamisch in eine Menge zu integrieren, wird bei den Anforderungen an den Funktionsumfang des Templatekonzepts darauf verzichtet, eine ORDER BY-Klausel bereits bei der Modellierung eines Templates mit zu integrieren.

Formulieren von Bedingungen in Subqueries

Durch die Integration von Subqueries in die Templates besteht eine breit gefächerte Möglichkeit, weitere Bausteine als Parameter zu erzeugen. Diese finden in den Fällen ihre Verwendung, in denen die Suchbedingung von der Lösungsmenge einer anderen Abfrage abhängt.
Eine Subquery stellt eine in Klammern eingeschlossene SELECT-Anweisung dar, die innerhalb der WHERE-Klausel einer anderen SELECT-Anweisung auftritt. Subqueries können aber auch in mehreren Ebenen geschachtelt sein, wobei bei dieser Konstellation die Reihenfolge der Abarbeitung von unten nach oben erfolgt. Alle im Zugriff befindlichen Tabellen, auch die Tabelle der Hauptabfrage, können über die Subquery angesprochen werden. Die Ausnahme bildet der EXISTS-Operator, da sich hier die Subquery stets auf eine Spalte bezieht.
Innerhalb des Templatekonzepts sollen die folgenden Subqueryvarianten realisierbar sein:

- Subqueries mit relationalen Operatoren.
- Subqueries mit *ALL* und *ANY.*
- Subqueries mit Operator *[NOT] IN.*
- Subqueries mit Operator *[NOT] EXISTS.*

Die Integration von geschachtelten Subqueries stellt gerade eine bedeutende Erweiterung des Systems dar, da diese mit der Funktionalität des entwickelten SQL-Generators nicht automatisch realisierbar sind. Durch diesen eingeschränkten Funktionsumfang kann eine durch den Modellierer manuell eingegebene Subquery auch nicht durch den SQL-Generator überprüft oder optimiert werden. Eine erste Erweiterung wurde bereits durch die Möglichkeit der manuellen Einbindung von Subqueries in eine vordefinierte Anweisung erreicht, wobei sich die Struktur einer vordefinierten Anweisung nicht dynamisch an die aktuelle Attributselektionsmenge eines DataSets anpassen kann, da diese Anweisungsform nicht veränderbar ist, da sie als konstantes Statement in einem DataSet fest kodiert hinterlegt wird. So ist auch für das Templatekonzept notwendig, Subqueries als Bausteine zu integrieren, die sich allerdings dynamisch an die aktuelle Struktur eines Templates anpassen müssen.

3.5.2.3 Definition der konkreten Templatebausteine

In diesem Kapitel werden die SQL-Bausteine bestimmt, die in ein durch das Templatekonzept generierbares SQL-Template integriert werden können. Die nachfolgend aufgeführten Bausteine werden dabei aus den einzelnen Klauseln einer SQL-Anfrage abgeleitet. Die in den jeweiligen Klauseln formulierbaren Ausdrücke werden in ihre elementare Struktur zerlegt und dem Templatekonzept als Parameter zur Verfügung gestellt. Des weiteren wird in diesem Zusammenhang eine Unterscheidung zwischen konstanten und variablen Bausteinen eines SQL-Templates vorgenommen.

3.5.2.3.1 Festlegung von konstanten Filterausdrücken

Als *konstante Filterausdrücke* werden im folgenden solche Filterbausteine bezeichnet, die in die Hauptanfrage eines Templates fest integriert werden. Die Einbindung dieses Filtertyps ist daher unabhängig von den konkret gewählten Attributspalten in der die Ergebnismenge darstellenden SELECT-Klausel zu betrachten. Es kann ein konstanter Filterausdruck erstellt werden, dessen Filterattribut nicht als ein Ergebnisattribut in der SELECT-Klausel eingebunden worden ist. Derartige Filterausdrücke stellen bezüglich der im System bereits verfügbaren dynamischen Filter eine Erweiterung in der Art dar, als daß die zur Laufzeit durch den Anwender formulierbaren Filter nur auf Attributspalten der SELECT-Klausel bezug nehmen dürfen. Im konkreten Anwendungsfall kann der Benutzer nur die tatsächlich selektierten Attribute einer Menge mit einem Filter belegen bzw. als ein Filterattribut in einen Filterausdruck integrieren.
Da jede Menge aber auch aus einer Definitionsattributmenge besteht, die alle über Tabellenlinks erreichbaren Attribute enthält, die zwar zueinander kompatibel, aber nicht in die Selektionsmenge gewählt worden sind, erscheint es sinnvoll, auch die nicht selektierten Attribute in Filterausdrücke zu integrieren. Dadurch ist es zusätzlich möglich, komplexere Filterausdrücke (Filterbäume) in ein Template einzubinden, die bereits bei der Modellierung durch einen mit der SQL-Syntax vertrauten Spezialisten erzeugt und dem Template zugeordnet wurden.

Nachfolgend wird untersucht, aus welchen Bestandteilen sich ein Filterbaum bzw. ein darin enthaltener Filterausdruck zusammensetzt und wie diese elementaren Bestandteile als Bausteine für das Templatekonzept bereitgestellt und verwaltet werden können. Die Verwaltung der einzelnen Bausteine ist notwendig, da ein Template dynamisch ein SQL-Statement generieren soll, wobei die Struktur von den durch den Benutzer gewählten Attributen hängt. Diese sind als Parameter in die SELECT-Klausel einzubinden. Das folgende Datenmodell veranschaulicht den Aufbau eines Filterbaums.

$$Filterbaum = Filterid \xrightarrow[m]{} Filtertyp$$
$$Filtertyp = elmentarerFilter \mid komplexerFilter$$
$$elementarerFilter = Filterattribut \times Vergl\_Operator \times Vergleichswert$$
$$Attribut = char^{+}$$
$$Vergl\_Operator = <|>|=|\leq|\geq|!=|like$$
$$Vergleichswert = Attribut \mid Konstante$$
$$Konstante = token$$
$$KoplexerFilter = Linker\ Filter \times Log\_Operator \mid Rechter Filter$$
$$Linker\ Filter = FilterID$$
$$Rechter Filter = FilterID$$
$$Log\_Operator = AND \mid OR \mid NOT$$

Das Templatekonzept parametrisiert die entsprechenden Bausteine eines Filters und materialisiert diese im Repository. Hierdurch kann die Struktur eines Filters dynamisch generiert werden. Die Parametrisierung der einzelnen elementaren Bestandteile und die anschließende Verwaltung eines (komplexen) Filterausdrucks ist notwendig, da sich die Struktur eines aus einem SQL-Template zur Laufzeit abgeleiteten konkreten SQL-Statements dynamisch an die vom Anwender ausgewählten Attribute anpassen muß. Zur Laufzeit wird also in Abhängigkeit der ausgewählten Attribute aus der Menge der zulässigen Attribute eines Templates der Filterbaum vollständig neu aufgebaut.

3.5.2.3.2 Festlegung von Gruppenfilterausdrücken

Die Einführung von *Gruppenfilterausdrücken* basiert auf der Annahme, daß Informationsbedürfnisse innerhalb einzelner Unternehmensbereiche grundlegend ähnlich sein können. Informationen werden

aus SQL-Statements gewonnen, die aufgrund ihrer Basisstrukturen identisch sind. Der Unterschied der Anfrageergebnisse könnte also nur von abweichend gesetzten Filterausdrücken abhängen, die eine für einen bestimmten Anwendungsbereich speziellere Datenverdichtung erzeugen.

Daher ist bei diesem Konzept der Ansatz gewählt worden, ein Template hinsichtlich der konstanten und variablen Bausteine zu unterscheiden. Als *konstante Bausteine* eines Templates werden die konstanten Filterausdrücke und die Subqueries bezeichnet, während ein Gruppenfilter bezüglich eines Templates als variabel (*variabler Baustein*) anzusehen ist. Bei der Modellierung eines Templates werden alle als konstant definierten Bausteine in die für jeden Bausteintyp vorgesehenen Tabellen unter einer Templatenummer (*TemplateID*) gespeichert. Soll nun dieses Template für spezielle Informationsbedürfnisse um zusätzliche Filter erweitert werden, um die Ergebnisdaten, welche die SQL-Anfrage aus den konstanten Bausteinen generiert, zu verfeinern bzw. zu verdichten, so sind diese Filter als *Gruppenfilter* zu definieren. Bei der dynamischen Erzeugung eines SQL-Statements aus dem um die Gruppenfilter erweiterten Basistemplates werden die variablen Filterausdrücke aus den entsprechenden Tabellen *dazugelinkt* und in den Gesamtfilter des SQL-Statements integriert. Diese Vorgehensweise hat einerseits den Vorteil der Mehrfachverwendung von Basistemplates und der damit verbundenen Vermeidung eines erhöhten Modellierungsaufwands. Andererseits wird eine redundante Speicherung gleicher Basisbausteine verhindert.

Da ein Gruppenfilterausdruck einen zum konstanten Filterausdruck identischen Aufbau hat, soll an dieser Stelle nicht näher auf die zur Generierung notwendigen Bausteine eingegangen werden.

3.5.2.3.3 Struktur einer Subquery als Templatebaustein

Ein Template muß ebenfalls Subqueries als Bausteine enthalten können, so daß zunächst untersucht wird, aus welchen Bestandteilen sich eine Subquery in der SQL-Syntax zusammensetzt. Des weiteren werden die verschiedenen Typen einer Subquery betrachtet, um die für jeden Typ erforderliche Parametrisierung zu gewährleisten.

Eine Unterabfrage setzt sich generell aus einem Mengenoperator und einer Anfrage zur Erzeugung der Unterergebnismenge zusammen. Durch den gewählten Mengenoperator wird darüber entschieden, ob die Notwendigkeit einer Vergleichsspalte aus den Tabellen der oberen Anfrage besteht, da diese über den spezifizierten Mengenoperator mit der Unterergebnismenge verglichen wird. Eine Vergleichsspalte ist nicht erforderlich, falls die Mengenoperatoren NOT EXIST oder EXISTS gewählt worden sind, da hierbei lediglich die (Nicht-) Existenz einer in der Unterabfrage spezifizierten Tabellenspalte abgefragt wird. Weiterhin ist zu prüfen, ob in den Filterausdrücken der Unteranfragen Attribute (Tabellenspalten) in den Vergleichswerten eingebunden werden, die aus Tabellen der oberen Anfragen zu realisieren sind. Diese müßten dann hinsichtlich der dynamischen Generierbarkeit eines SQL-Templates ebenfalls parametrisiert werden, um zu gewährleisten, daß bei einer Strukturänderung der oberen Anfrage in jedem Fall die Tabelle verfügbar ist, aus der das Vergleichsattribut der Unteranfrage zu gewinnen ist. Eine Strukturänderung der oberen Anfrage ergibt sich aus der Möglichkeit, Attribute der Hauptanfrage dynamisch zur Laufzeit in das SQL-Template einzubinden / zu entfernen. Hierdurch können bei bestimmten Attributkonstellationen Tabellen der oberen Anfrage entfernt werden, da die hieraus zu gewinnenden Attribute nicht ausgewählt worden sind. Sollte aber auch ein Vergleichsattribut der unteren Anfrage in einer der zuvor entfernten Tabellen enthalten sein, so muß dafür gesorgt werden, daß diese Tabelle dennoch in die FROM-Klausel der Hauptanfrage eingebunden wird.
Dem geschilderten grundlegenden Aufbau einer Subquery lassen sich mehrere erforderliche Parameter für den dynamischen Aufbau eines SQL-Templates entnehmen:

Parametertyp 1: Die Vergleichsspalte muß insofern parametrisiert werden, als daß von vornherein nicht ersichtlich ist, welche Tupelvariable die Tabelle erhält, in der die Vergleichsspalte enthalten ist.

Parametertyp 2: Die Vergleichsattribute in den Filterausdrücken der Unterabfrage, die aus den Tabellen der oberen Anfrage zu gewinnen sind.

Parametertyp 3: Die SQL-Struktur der Unteranfrage wird als String gespeichert, in den bei Verwendung von externen Vergleichsattributen in Filterausdrücken, zur Laufzeit die aktuellen Vergleichsspaltennamen und die dazugehörigen Tupelvariablen eingefügt werden müssen.

Das Ausgangsstatement, welches eine zu parametrisierende Subquery enthält, weist folgende Struktur auf:

```
SELECT a1.Name,a2.Ort,a3.Gehalt
FROM   Tabelle a1, Tabelle a2, Tabelle a3
WHERE  a3.Gehalt < ALL (   Select s1.Gehalt
                           From Tabelle s1
                           Where s1.Ort = a2.Ort  )  ;
```

Die *Vergleichsspalte* aus der Tabelle mit der Korrelationsvariablen *a3* und das Vergleichsattribut in der Unterabfrage *a2.Ort* stellen die ersten beiden Parameter der Subquery dar. Diese werden daher innerhalb der Bausteinverwaltung des Templatekonzepts durch die Platzhalter *{H1}* für die Vergleichsspalte (*Parametertyp 1*) und *{H2}* für das externe Vergleichsattribut (*Parametertyp 2*) ersetzt. Intern wird ein Verweis zwischen den gesetzten Platzhaltern und den Attributnamen vorgenommen, welche parametrisiert worden sind. Die gesamte Subquery (*Parametertyp 3*) wird in folgender Form in den Bausteintabellen verwaltet:

```
{H1} < ALL(  SELECT s1.Gehalt
             FROM Tabelle s1
             WHERE s1.Ort = {H2} )   ;
```

Zum Zeitpunkt der Generierung eines SQL-Statements aus diesem Template wird dann unter Abhängigkeit der gewählten Attribute, welche in die Hauptanfrage eingebunden werden, die parametrisierte Subquery-Struktur mit den konkreten Spaltennamen und den dazugehörigen Korrelationsvariablen aktualisiert.

```
SELECT a1.Gehalt
FROM   Tabelle a1, Tabelle a2
WHERE  a1.Gehalt < ALL ( SELECT s1.Gehalt
                         FROM Tabelle s1
                         WHERE s1.Ort = a2.Ort )  ;
```

3.5.2.4 Klassifikation von Templates

Bei der Klassifikation von Templates geht es im wesentlichen darum, die modellierten Templates den entsprechenden Unternehmensbereichen zuzuordnen, deren spezielle Informationsbedürfnisse über die Templates realisiert werden sollen. Es ist daher notwendig, einen Auswahlkatalog bereitzustellen, der sich in die einzelnen Unternehmensbereiche untergliedert und in jeder Hierarchiestufe die für die Stufe relevanten Templates verwaltet. Da aber aufgrund der Ähnlichkeit der Informationsbedürfnisse innerhalb einer Unternehmensebene grundlegende Bausteine der Templates identisch sein können, werden *virtuelle Templates* eingeführt bzw. zur Auswahl bereitgestellt, die alle auf ein einmal definiertes Basistemplate zugreifen, aber dieses um abweichende Filterausdrücke für ihren speziellen Informationsbedarf erweitern. Die einem virtuellen Template zugeordneten Filtererweiterungen stellen die zuvor bereits erläuterten *Gruppenfilter* dar.

Um aber eine Mehrfachverwendung von Basistemplates bei der Integration in virtuelle Templates zu ermöglichen, ist es erforderlich, das Wiederauffinden dieser bereits modellierten und hinterlegten Basistemplates zu gewährleisten. Daher ist innerhalb des Modellierungstools ein Suchwerkzeug (siehe *Kapitel: 3.5.4 Wiederauffinden modellierter Templates*) bereitgestellt worden, über welches mit Attributen als Suchparameter nach Basistemplates gesucht werden kann. So kann ein gefundenes Basistemplate um individuelle Gruppenfilter erweitert und als ein neues virtuelles Template unter dem entsprechenden Unternehmensbereich im Auswahlkatalog hinterlegt werden. Der Anwender des Systems kann darauf auf das jeweilige im Auswahlkatalog hinterlegte Template (virtuelles Template) zugreifen, welches seine speziellen Informationsanfragen an das System realisiert (siehe *Kapitel 3.5.5 Integration des SQL-Templatekonzepts*).

3.5.3 Modellierung von SQL-Templates

Nachfolgend wird ein für das Templatekonzept geeigneter Bausteinkatalog definiert, der sämtliche parametrisierbaren Bestandteile eines SQL-Statements umfaßt und diese in einer adäquaten Form für die dynamische Anpassung an individuelle Informationsbedürfnisse des Anwenders bereitstellt. Der Anwender des Systems hat nur über DataSets eine Einflußnahme auf die Struktur eines Templates, indem er die für ihn relevanten Attribute eines Templates, die seinen Informationsbedarf darstellen, über das Mengentool selektiert. Der Templatemechanismus hat darauf die Aufgabe, ein SQL-Statement aus der selektierten Attributkombination und denen dem Template (virtuellen Template) zugeordneten Bausteinen zu generieren. Der Anwender wird also von der Integration von Filterausdrücken und Subqueries ferngehalten, da diese nur durch den Modellierer eines SQL-Templates bei der Erstellung des Templates erzeugt werden.

3.5.3.1 Anforderungen an eine Modellierungsoberfläche

Ein SQL-Statement wird normalerweise über eine Maske als Textstring eingegeben, gespeichert und an die Datenbank übergeben. Da aber bei dem Templatemechanismus Bestandteile eines SQL-Statements parametrisiert und als solche vom System erkannt werden sollen, ist eine derartige Vorgehensweise bei der Formulierung eines Templates nicht möglich. Daher werden alle Bausteine eines SQL-Templates, die der Modellierer formulieren kann, über spezielle Bausteinmasken in das System eingegeben. Dadurch ist es sowohl möglich, die einzelnen Bausteine syntaktisch korrekt zuzuordnen, als auch die Kompatibilität der Bausteine innerhalb des Templates zu überprüfen und zu gewährleisten. Des weiteren braucht der Modellierer ausschließlich nur mit Attributnamen zu arbeiten, so daß er unabhängig von den im Repository definierten Tabellenspaltennamen ist, und sich nicht tiefergehend mit der Repositorystruktur auseinandersetzen muß. Er kann sich auf der Modellierungsoberfläche die Filterausdrücke und Subqueries zusammenstellen, ohne sich um eventuell erforderliche Tabellenlinks oder der Vergabe von Korrelationsvariablen kümmern zu müssen.

In dem gleichem Maße, in dem der Modellierungsaufwand sinkt, steigt die Anforderung an die Funktionalität der Modellierungsoberfläche. Zum einen muß über die Oberfläche ein Dialog erzeugt werden, der den Arbeitsschritten bei der Erstellung eines SQL-Statements entspricht, so daß der Modellierer die Reihenfolge seiner Vorgehensweise über die Modellierungsmasken einhält. Zum anderen muß eine derartige Modellierungsoberfläche viele Aufgaben übernehmen, die sonst üblicherweise der Modellierer eines SQL-Statements übernehmen muß. So werden nur die Attribute zur Integration in die einzelnen Bausteine zur Auswahl angeboten, die auch kompatibel zum bisherigen Templateaufbau sind, da sie aus gemeinsamen Tabellen gewonnen werden können. Generell müssen nach jeder Integration eines Bausteins alle erforderlichen Statementklauseln in einer syntaktisch korrekten Form aktualisiert werden.

Daher ist eine Modellierungsoberfläche zu entwickeln, die es dem Modellierer ermöglicht, alle innerhalb des Templatekonzepts spezifizierten Bausteine in das System einzugeben. Die Oberfläche wird dabei in einzelne Eingabemasken untergliedert, deren jeweilige Aufgabe darin besteht, speziell eine der zulässigen Templatebausteine bzw. deren elementare Bestandteile aufzunehmen. So existieren Eingabemasken zur Aufnahme der konstanten Filterausdrücke, der Gruppenfilter und der Subqueries sowie deren Filterausdrücken. Nach jeder Eingabe eines Bausteins wird darauf die Struktur des Templates um die neuen Ausdrücke erweitert und in der momentanen Arbeitsmaske angezeigt. Des weiteren existieren die üblichen Pflegefunktionen, wie das Löschen und Verändern zuvor eingegebener Bausteine, die ein komfortables Arbeiten bei der Modellierung ermöglichen.

Abbildung 23 stellt die Schnittstellen des Templatemechanismus zu den über die Modellierungsoberfläche eingehenden Bausteinen, zu den Generierungsfunktionen, zur Templatetabellenstruktur und zum Repository dar.

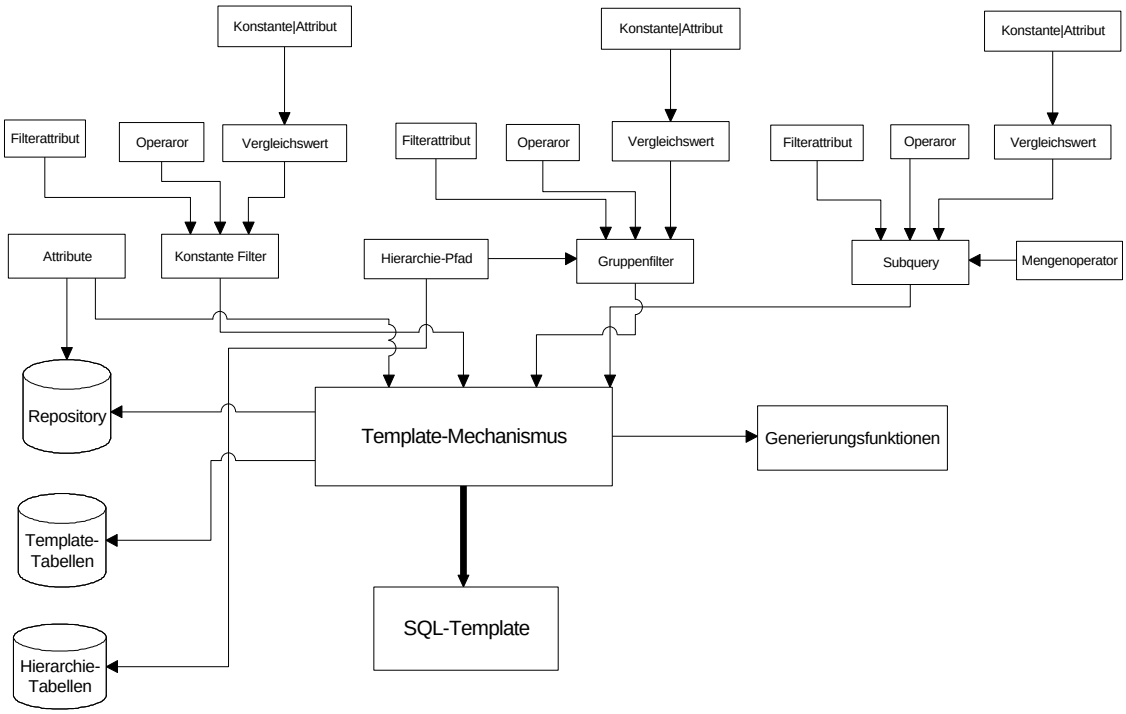


Abbildung 23: Bausteinstruktur des Templatemechanismus

3.5.3.2 Voraussetzungen für die automatische Erstellung eines SQL-Statements

Nachfolgend wird der Zusammenhang eines in einer formalisierten Darstellung veranschaulichten SQL-Statements und einem parametrisierten SQL-Template erläutert. Bei der Darstellung des SQL-Statements wird nicht der Anspruch auf die Beschreibung der vollständigen Syntaxdiagramme erhoben, sondern es wird durch die Verwendung eines abstrakt formulierten Datenmodells der Zusammenhang von Attributen bzw. Spalten, Tabellen und Links gezeigt. Des weiteren wird die Verwendung der Attribute und kompatiblen Spalten in Filtern, Subqueries, der GROUP BY-Klausel (mit oder ohne HAVING-Klausel) und der ORDER BY-Klausel dargestellt.

Invarianten für ein korrektes SQL-Statement

Zunächst wird ein Datenmodell entwickelt, welches die Komplexität der SQL-Statements widerspiegelt, die durch das SQL-Templatekonzept realisierbar sind. Anschließend werden für dieses Datenmodell Invarianten spezifiziert, die die Korrektheit der über Templates generierten SQL-Statements gewährleisten. Die durch die Invarianten sichergestellte Konsistenz des Statements bezieht sich allerdings nicht auf die korrekte SQL-Syntax, sondern auf die Kompatibilität der in den einzelnen Statementklauseln enthaltenen Bausteine.

Das Datenmodell setzt in seiner obersten Hierarchiestufe allgemein auf den Klauseln eines SQL-Statements auf, wobei auf die Einbeziehung der ORDER BY-Klausel bei dieser Betrachtung verzichtet wird, da diese nicht im Funktionsumfang der Templates enthalten ist, sondern optional durch den Anwender nachträglich in jede Menge (*DataSet*) integriert werden kann. Weiterhin wird bei dem nachfolgenden Datenmodell nicht die Verwendung des UNION-Operators betrachtet, obwohl dieser für die Verwendung bestimmter Attribute im Repository definiert wird. Der UNION-Operator bestimmt, welche Attribute über mehrere Spalten aus den konkreten Datenbanktabellen gewonnen werden können, wie z.B. „ID“. Falls nicht ein zusätzliches Attribut in einem DataSet gewählt wird, so werden alle Tabellen, in denen dieses Attribut als konkrete Spalte enthalten ist, über ein UNION verbunden. Erst durch die Hinzunahme eines weiteren Attributs aus dem Repository wird die genaue Tabelle spezifiziert und UNION ist als Operator nicht mehr notwendig. Da aber bei dem Einsatz der SQL-Templates davon ausgegangen wird, daß mehr als ein Attribut in die SELECT-Klausel

eingebunden werden soll, wird bei dem Datenmodell auf die Einbindung eines UNION-Operators verzichtet.

Über das Datenmodell wird ein SQL-Statement spezifiziert, welches aus einer Hauptanfrage besteht, die eine Subquery der Schachtelungstiefe 2 enthalten kann. Diese Darstellung entspricht der maximalen Komplexität eines SQL-Templates. Die einzelnen Klauseln des modellierten SQL-Statements werden innerhalb des Datenmodells sukzessive in ihre elementaren Bestandteile untergliedert.

Datenmodell des SQL-Statements:

$$SQL\_Statement = Select\_Klausel \times From\_Klausel \times Where\_Klausel \times GroupBy\_Klausel \times Having\_Klausel$$
$$Select\_Klausel = Attribut - set$$
$$Attribut = Aggregationsfunktion \mid selbstdefinierteFunktion \mid ElementaresAttribut$$
$$Aggregationsfunktion = ElementaresAttribut \times Funktion$$
$$ElementaresAttribut = token$$
$$Funktion = AVG \mid SUM \mid MIN \mid MAX \mid COUNT$$
$$selbstdefinierteFunktion = token$$
$$From\_Klausel = Tabelle - set$$
$$Tabelle = Attribut - set$$
$$Where\_Klausel = Joinmenge \times Filtermenge_w \times Subquery_1$$
$$Joinmenge = Join - set$$
$$Join = ElementaresAttribut \times rel\_Operator \times ElementaresAttribut$$
$$rel\_Operator = \prec \mid \succ \mid = \mid \leq \mid \geq \mid != $$
$$Filtermenge_w = Filtermenge$$
$$Filtermenge = Filterausdruck - set$$
$$Filterausdruck = Attribut \times rel\_Operator \times Vergleichswert \times \log\_Operator$$
$$Vergleichswert = Attribut \mid Konstante$$
$$Konstante = token$$
$$\log\_Operator = AND \mid OR \mid NOT$$
$$Subquery_1 = Vergleichsattribut \times SET\_Operator \times SQL\_Statement_1$$
$$Vergleichsattribut = Attribut$$
$$SET\_Operator = rel\_Operator \mid [rel\_Operator]ANY \mid [rel\_Operator]ALL \mid [NOT]IN \mid [NOT]EXISTS$$
$$SQL\_Statement_1 = Select\_Klausel \times Where\_Klausel_1 \times Having\_Klausel_1$$
$$Where\_Klausel_1 = Joinmenge \times Filtermenge_w \times Subquery_2$$
$$Subquery_2 = VergleichsAttribut \times SET\_Operator \times SQL\_Statement_2$$
$$SQL\_Statement_2 = Select\_Klausel \times From\_Klausel \times Where\_Klausel_2 \times Having\_Klausel_2$$
$$Where\_Klausel_2 = Joinmenge \times Filtermenge_w$$
$$Filtermenge_{Hh} = Filtermenge$$
$$Having\_Klausel_2 = Filtermenge_h$$
$$Having\_Klausel_1 = Filtermenge \times Subquery_1$$
$$GroupBy\_Klausel = GroupAttribut - set$$
$$GroupAttribut = ElementaresAttribut \mid selbstdefinierteFunktion$$
$$Having\_Klausel = Filtermenge_h \times Subquery_1$$

Die Invariante für das gesamte Statement wird durch die Invariante *inv_SQL*_Statement repräsentiert, die ihrerseits die Invarianten für die Hauptanweisung des SQL-Statements aufruft (die *SELECT*-, die *FROM*-, die *WHERE*-, die *HAVING*- und die *GROUP BY*-Klausel).

$$inv\_SQL\_Statement((selectklausel, fromklausel, whereklausel, havingklausel, groupbyklausel)) \triangleq$$
$$inv\_Select\_Klausel(selectklausel) \wedge inv\_From\_Klausel(fromklausel) \wedge$$
$$inv\_Where\_Klausel_1(whereklausel) \wedge inv\_Having\_Klausel_1(havingklausel) \wedge$$
$$inv\_GroupBy\_Klausel(groupbyklausel)$$

Durch die Vorbedingung *inv_Select_Klausel* wird überprüft, ob alle Attribute in der SELECT-Klausel zueinander kompatibel sind. Der Kompatibilitätstest erfolgt über die Hilfsfunktion *Attribut_Is_Kompatibel*, die überprüft, ob eine Menge von Attributen zu einem weiteren Attribut kompatibel ist.

$$inv\_Select\_Klausel(selectklausel) \triangleq$$
$$\forall a \in selectklausel \bullet (Attribut\_Is\_Kompatibel(selectklausel - \{a\}, a))$$

Da alle in einem Statement integrierten Attribute, die in der SELECT-Klausel, in Filterausdrücken oder in der GROUP BY-Klausel enthalten sind, auch aus gemeinsamen Tabellen gewonnen werden können, muß gewährleistet sein, daß die erforderlichen Tabellen in der From-Klausel integriert werden.

$$inv\_From\_Klausel(fromklausel) \triangleq$$
$$\forall a \in selectklausel \bullet (a \in union\ \ fromklausel) \wedge$$
$$\forall f \in filtermenge \bullet \begin{pmatrix} f.Attribut \in union\ \ fromklausel \wedge \\ if\ \ is-Attribut(f.Vergleichswert) \\ then\ \ f.Vergleichswert \in union\ \ fromklausel \end{pmatrix}$$

Die Bestandteile einer WHERE-Klausel setzen sich aus einer Menge von *Joins*, *Filterausdrücken* und einer *Subquery* zusammen. Daher wird in der WHERE-Vorbedingung *inv_Where_Klausel* auch für jeden einzelnen Bestandteil eine Invariante aufgerufen.

$$inv\_Where\_Klausel(whereklausel) \triangleq$$
$$inv\_Joinmenge(whereklausel.joinmenge) \wedge$$
$$inv\_Filtermenge_w(whereklausel.filtermenge) \wedge$$
$$inv\_Subquery_1(whereklausel.subquery)$$

Ein Nachweis für die Korrektheit der notwendigen bzw. integrierten Joins ist bei dieser Betrachtung nicht erforderlich, da unter Verwendung des SQL-Generators alle benötigten Joins bezüglich einer verwendeten Attributmenge für die SELECT-Klausel und die Filterausdrücke automatisch ermittelt werden.

$$inv\_Join(\ ) \triangleq$$
$$TRUE$$

Durch die Invariante *inv_Filtermenge*$_w$ werden die einzelnen Filterausdrücke in der WHERE-Klausel auf ihre Kompatibilität überprüft. Ein Filterattribut eines Filterausdrucks in der WHERE-Klausel darf nur ein *ElementaresAttribut* oder eine *selbstdefinierteFunktion* sein. Falls der Vergleichswert auch ein *ElementaresAttribut* oder eine *selbstdefinierteFunktion* darstellt, muß sichergestellt werden, daß das Filterattribut und das Vergleichsattribut hinsichtlich ihres Datentyps zueinander kompatibel sind.

$$inv\_Filtermenge_w(filtermenge) \triangleq$$
$$\forall f \in filtermenge \bullet \begin{pmatrix} (is-ElementaresAttribut(f.Attribut) \vee is-selbsdefinierteFunktion(f.Attribut)) \wedge \\ if\ \ NOT\ \ is-Konstante(f.Vergleichswert) \\ then\ \ Attribut\_Is\_Kompatibel(f.Attribut,\ f.Vergleichswert) \\ else\ \ TRUE \end{pmatrix}$$

Da innerhalb dieses Modells eine Subquery integriert werden kann, die wiederum eine Subquery beinhaltet, wird die Subquery zur genaueren Betrachtung in eine obere Subquery (*Subquery*$_1$) und in eine untere Subquery (*Subquery*$_2$) aufgeteilt. Die Subquerybausteine werden in Anlehnung an die technische Realisierung im Templatekonzept in das Vergleichsattribut, falls sich auf die Subquery ein Mengenoperator ungleich EXISTS oder IN bezieht, in den Mengenoperator an sich und in die die Subquery darstellende SQL-Anweisung aus den bekannten Statementklauseln unterteilt. In dem Fall, daß sich ein Vergleichsattribut über einen angegeben Mengenoperator auf die Ergebnismenge der Subquery bezieht, ist es notwendig zu überprüfen, ob das Ergebnisattribut der Subquery kompatibel zu dem Vergleichsattribut der oberen Anfrage ist. Des weiteren muß gewährleistet sein, daß der Mengenoperator einen zulässigen Operator darstellt. Durch den Aufruf der Invariante *inv_SQL-Statement*$_1$ wird das, die Subqueryergebnismenge erzeugende, SQL-Statement überprüft.

$$inv\_Subquery_1((vergleichsattribut, set\_operator, sql\_statement_1))\underline{\Delta}$$
$$Attribut\_Is\_Kompatibel(vergleichsattribut, sqlstatement_1.selectklausel)\wedge$$
$$SET\_Operator\_Korrekt(set\_operator)\wedge$$
$$inv\_SQL\_Statement_1(sql\_statement_1)$$

Das SQL-Statement, das die Subquery$_1$ realisiert, wird ebenso, wie das obere Statement auf die Korrektheit der Tabellenmenge in der FROM-Klausel, der Bestandteile in der WHERE-Klausel und der Bestandteile der HAVING-Klausel untersucht. Allerdings unterscheiden sich die aufzurufenden Invarianten bei der WHERE- und HAVING-Klauseluntersuchung von denen der oberen Anfrage, da Filterattribute in den Filterausdrücken bezug auf kompatible Vergleichsattribute der oberen Anfrage nehmen können und eine Subquery in einer der beiden Klauseln auftreten kann, die sich in ihrem Aufbau grundlegend von der Subquery$_1$ unterscheidet (siehe *inv_SQL_Statement$_2$*).

$$inv\_SQL\_Statement_1((selectklausel, fromklausel, whereklausel, havingklausel), vergleichsattribut)\underline{\Delta}$$
$$Select\_Klausel(selectklausel)\wedge inv\_From\_Klausel(fromklausel)\wedge$$
$$inv\_Where\_Klausel_1(whereklausel)\wedge inv\_Having\_Klausel_1(havingklausel)$$

Die WHERE-Klausel der oberen Subquery besteht wiederum aus einer Filtermenge und einer weiteren Subquery (*Subquery$_2$*).

$$inv\_Where\_Klausel_1(whereklausel)\underline{\Delta}$$
$$inv\_Joinmenge(whereklausel.Joinmenge)\wedge$$
$$inv\_Filtermenge_W(whereklausel.Filtermenge)\wedge$$
$$inv\_Subquery_2(whereklausel.Subquery)$$

Der Unterschied der Invarianten *inv_Subquery$_2$* besteht in dem Aufruf der Vorbedingung *inv_SQL_Statement$_2$*, da der Aufbau des die zweite Subquery umfassenden SQL-Statements von dem der oberen Subquery abweicht.

$$inv\_Subquery_2((vergleichsattribut, set\_operator, sql\_statement_2))\underline{\Delta}$$
$$Attribut\_Is\_Kompatibel(vergleichsattribut, sqlstatement_1.selectklausel)\wedge$$
$$SET\_Operator\_Korrekt(set\_operator)\wedge$$
$$inv\_SQL\_Statement_2(sql\_statement_2)$$

Die Invariante *inv_SQL_Statement$_2$* weicht insofern von der Invariante *inv_SQL_Statement$_1$* ab, als daß sich die Untersuchung der in den *SQL-Statement$_2$* befindenden WHERE- und HAVING-Klausel unterscheidet. Denn zum einen entfällt hierbei die Untersuchung einer weiteren Subquery und zum anderen erweitert sich der Umfang der Filterausdrücke um die Möglichkeit, daß sich ein Filterattribut nicht nur auf ein Vergleichsattribut der oberen Anfrage (*Subquery$_1$*) beziehen kann, sondern auch auf ein Vergleichsattribut der Hauptanfrage.

$$inv\_SQL\_Statement_2((selectklausel, fromklausel, whereklausel, havingklausel), vergleichsattribut)$$
$$Attribut\_Is\_Kompatibel(selectklausel, vergleichsattribut)\wedge inv\_From\_Klausel(fromklausel)\wedge$$
$$inv\_Where\_Klausel_2(whereklausel)\wedge inv\_Having\_Klausel_2(havingklausel)$$

$$inv\_Where\_Klausel_2(whereklausel)\underline{\Delta}$$
$$inv\_Joinmenge(whereklausel.Joinmenge)\wedge$$
$$inv\_Filtermenge_W(whereklausel.Filtermenge)$$

$$inv\_Having_2(havingklausel) \underline{\Delta}$$
$$inv\_Filtermenge_H(havingklausel.Filtermenge)$$

Durch die Invariante *inv_Filtermenge$_H$* werden die einzelnen Filterausdrücke in der HAVING-Klausel auf ihre Kompatibilität überprüft. Ein Filterattribut eines Filterausdrucks in der WHERE-Klausel darf nur eine *Aggregationsfunktion* sein.

$$inv\_Filtermenge_H(filtermenge) \underline{\Delta}$$
$$\forall f \in filtermenge \bullet \begin{pmatrix} is-Aggregationsfunktion(f.Attribut) \wedge \\ if\ NOT\ is-Konstante(f.Vergleichswert) \\ then\ Attribut\_Is\_Kompatibel(f.Attribut, f.Vergleichswert) \\ else\ TRUE \end{pmatrix}$$

In der HAVING-Klausel dürfen nur Filterausdrücke integriert werden, deren Attribut(e) aggregiert werden. Es darf nur eine Subquery integriert werden, die eine aggregierte Vergleichsspalte hat.

$$inv\_Having_1(havingklausel) \underline{\Delta}$$
$$inv\_Filtermenge_H(havingklausel.Filtermenge) \wedge$$
$$inv\_Subquery_2(havingklausel.Subquery)$$

Alle Attribute der GROUP BY-Klausel müssen auch in der SELECT-Klausel integriert sein, wobei in der SELECT-Klausel ein aggregiertes Attribut vertreten sein muß, welches wiederum nicht in der GROUP BY-Klausel integriert sein darf.

$$inv\_GroupBy\_Klausel(groupbyklausel) \underline{\Delta}$$
$$card(groupbyklausel) \prec card(Select\_Klausel) \wedge$$
$$\forall a \in groupbyklausel \bullet (a \in Select\_Klausel \wedge Not\ is-Aggregationsfunktion(a)) \wedge$$
$$\exists a \in Select\_Klausel \bullet (is-Aggregationsfunktion(a))$$

In der HAVING-Klausel der Hauptanweisung können sowohl Filterausdrücke als auch eine Subquery integriert sein, die untersucht werden müssen.

$$inv\_Having(havingklausel) \underline{\Delta}$$
$$inv\_Filtermenge_H(havingklausel.Filtermenge) \wedge$$
$$inv\_Subquery_1(havingklausel.Subquery)$$

3.5.4 Wiederauffinden modellierter Templates

In diesem Kapitel werden die implementierten Mechanismen erläutert, die es ermöglichen, auf bereits modellierte und materialisierte Templates zuzugreifen und zur Modellierung weiterer Templates zu verwenden. Die Suchparameter dieses Retrievalverfahrens bilden die vom Modellierer gewählten Attributkombinationen, welche die Ergebnismenge eines zu erstellenden Templates repräsentieren. Dabei kann der Modellierer bestimmen, ob die Suchattribute konjunktiv oder disjunktiv verknüpft werden. Die Verknüpfung der Attribute kann für die jeweilige Zielsetzung der Suche interessant sein. So könnte der Modellierer beispielsweise bei einer disjunktiven Suche ein Interesse daran haben, in welchen hinterlegten Templates überhaupt mindestens eines der Suchattribute enthalten ist. Bei der konjunktiven Suche kann das Ziel sein, ein oder mehrere Templates zu finden, die alle gewählten Attribute realisieren bzw. in deren SELECT-Klausel einbinden können.

Nachdem die Suche vom System durchgeführt worden ist, hat der Modellierer die Möglichkeit, sich die Struktur jedes der gefundenen Templates anzeigen zu lassen, um darüber zu entscheiden, ob die bereits

darin enthaltenen Bausteine für die Realisierung seines Problems nützlich sind. Falls ein Template ermittelt wird, welches den Anforderungen des Modellierers genügen sollte, kann dieses zur weiteren Bearbeitung in das Modellierungstool übernommen werden. Bei der Übernahme des Templates wird unterschieden, ob das Template nur um Gruppenfilter erweitert, oder ob aus den gefundenen Templatebausteinen ein neues Template erstellt werden soll. Die erste Variante dient der Erstellung der bereits genannten virtuellen Templates. In diese Templates können lediglich Gruppenfilter integriert werden, wodurch eine Wiederverwendung des gesamten Templates erreicht wird. Wenn aber nur einige der Templatebausteine für das aktuelle Problem interessant sein sollten, kann das Template auch in der Art übernommen werden, daß bei der Modellierung alle Bausteine verändert werden können. Die aus dem alten Template adaptierten bzw. geänderten Bausteine werden danach als ein neues Template hinterlegt.

Zur Wiederauffindung bereits modellierter Templates ist eine Indexstruktur erzeugt worden, über welche die lösungsrelevanten Templates mittels der gewählten Attribute ermittelt werden können. Die Indexstruktur wird aus den Inhalten zweier Tabellen aufgebaut.

In der Tabelle *RPT_Template_TMenge* wird jedem *Attributnamen* eines in einer oder mehreren Template-SELECT-Klausel(n) enthaltenen Attributs die Menge aller Templates, genauer deren *TemplateID*, zugeordnet, in welcher das Attribut vorkommt. Durch die Tabelle kann ein indizierter Zugriff auf alle Templates erfolgen, deren Attributmengen das gewählte Attribut mit dem Attributnamen enthalten, wodurch der Suchraum nur auf die relevanten Templates eingeschränkt wird.

$$RPT\_Template\_TMenge = Attributname \xrightarrow[m]{} TemplateID - set$$
$$Attributname = char^{+}$$
$$TemplateID = N_1$$

Die über den *Attributnamen* aus der Tabelle Templatemenge ausgelesene Menge an Templates kann nun elementweise in der Tabelle *Attributmenge* abgearbeitet werden, indem ein Zugriff für jede *TemplateID* der ermittelten Templatemenge auf die gesamte Attributmenge des Templates erfolgt, die durch das Feld *(Attributname)-set* spezifiziert wird. Ein für die Geschwindigkeit der Suche wesentlicher Punkt ist, daß von allen Templatemengen, die durch einen Attributnamen aus der Tabelle Templatemenge ausgelesen worden sind, die jeweilige abgearbeitete TemplateID aus der Menge entfernt wird, da zuvor die Schnittmenge aller Templatemengen gebildet wird. Durch diese Vorgehensweise wird die doppelte Abarbeitung eines Templates vermieden, falls mehr als ein gewähltes Attribut in der Attributmenge eines Templates enthalten ist.

$$RPT\_Attribut\_TMenge = TemplateID \xrightarrow[m]{} Attributname - set$$
$$TemplateID = N_1$$
$$Attributname = char^{+}$$

Die Hauptfunktion zur Auffindung der Templates stellt die Funktion *Ermittle_Templates* dar, die als Aufrufparameter die gewählten Attribute und deren Verknüpfungsoperator erhält.

$$Attribut = char^{+}$$
$$SuchOp = AND \mid OR$$
$$TemplateID = N_1$$

$$type\ \ Ermittle\_Templates: \quad Attribut - set \times SuchOp \rightarrow TemplateID - set$$

$$Ermittle\_Templates(attr, suchop) \underline{\Delta}$$
$$\quad Let\ \ TemplateIDMenge = Lese\_Template\_Menge(attr)\ \ in$$
$$\quad\quad Let\ \ TemplateIDMengeSel = Selektiere\_Templates(attr, TemplateIDMenge, suchop)\ \ in$$
$$\quad\quad\quad (TemplateIDMengeSel)$$

In einem ersten Schritt werden alle Templatemengen aus der Tabelle *RPT_Template_Menge* ausgelesen, in denen jeweils mindestens eines der gewählten Attribute enthalten ist. Die so gewonnenen

Templatemengen werden darauf zu einer Menge vereinigt, so daß jedes Template nur einmal in der Gesamttemplatemenge auftritt. Diese Vorgehensweise ist notwendig, da mehr als ein Attribut auf dasselbe Template (TemplateID) zeigen kann bzw. in einem Template mehr als eines der gewählten Attribute enthalten ist.

$type\ Lese\_Template\_Menge: \ Attribut-set \times TemplateID-set \rightarrow TemplateID-set$

$Lese\_Template\_Menge(attr, tempidvereinigt) \underline{\Delta}$

$if\ card(attr) \succ 0$

$then\ Let\ a = a \in attr\ be\ such\ that\ a \in dom\ RPT\_Template\_Menge\ in$

$if\ a \neq \{\ \}$

$then\ Lese\_Template\_Menge(attr - \{a\}, tempidvereinigt \cup RPT\_Template\_Menge(a))$

$else\ tempidvereinigt$

$else\ tempidvereinigt$

Durch die Funktion *Lese_Template_Menge* wurden zunächst einmal alle Templates ermittelt, die mindestens eines der gewählten Attribute enthalten. Da aber das Suchergebnis der zuvor bestimmten Suchbedingung entsprechen muß, wird in einem weiteren Schritt die Menge der ausgelesen Templates auf die Templatemenge reduziert, deren einzelne Templateelemente jeweils die Suchbedingung erfüllen. Diese Aufgabe kommt der Funktion *Selektiere_Templates* zu, die wiederum je nach der gewählten Suchbedingung die Hilfsfunktionen *Konjunktive_Suche* oder *Disjunktive_Suche* zur Ermittlung der zulässigen Templatemenge aufruft.

$type\ Selektiere\_Templates: Attribut-set \times TemplateID-set \times SuchOp \rightarrow TemplateID-set$

$Selektiere\_Templates(attr, tempidmenge, suchop) \underline{\Delta}$

$if\ is-AND(suchop)$

$then\ Kunjuntive\_Suche(attr, tempidmenge, \{\ \})$

$else\ Disjunktive\_Suche(attr, tempidmenge, \{\ \})$

$type\ Konjunktive\_Suche: \ Attribut-set \times TemplateID-set \times TemplateID-set \rightarrow TemplateID-set$

$Konjunktive\_Suche(attr, tempidmenge, tempidmengesel) \underline{\Delta}$

$if\ card(tempidmenge) \succ 0$

$then\ Let\ tid = tid \in tempidmenge\ be\ such\ that\ attr \subseteq RPT\_Attribut\_TMenge(tid)\ in$

$if\ t \neq \{\ \}$

$then\ Konjunktive\_Suche(attr, tempidmenge - \{t\}, tempidmengesel \cup \{t\})$

$else\ tempidmengesel$

$else\ tempidmengesel$

$type\ Disjunktive\_Suche: \ Attribut-set \times TemplateID-set \times TemplateID-set \rightarrow TemplateID-set$

$Disjunktive\_Suche(attr, tempidmenge, tempidmengesel) \underline{\Delta}$

$if\ card(tempidmenge) \succ 0$

$then\ Let\ tid = tid \in tempidmenge\ be\ such\ that\ attr \cap RPT\_Attribut\_TMenge(tid) \neq \{\ \}\ in$

$if\ t \neq \{\ \}$

$then\ Disjunktive\_Suche(attr, tempidmenge - \{t\}, tempidmengesel \cup \{t\})$

$else\ tempidmengesel$

$else\ tempidmengesel$

3.5.5 Integration des SQL-Templatekonzepts

Im folgenden wird erläutert, wie die technische Integration des Templatekonzepts in das bisher entwickelte System auf Basis der DataSets erfolgt. Dabei ist zunächst das SQL-Templatemodellierungstool innerhalb des Systems zu implementieren, so daß über diese Komponenten verschiedenartige SQL-Templates für die speziellen Informationsbedürfnisse erstellt werden können. Der nächste Entwicklungsschritt umfaßt die Möglichkeit des eigentlichen Einsatzes der SQL-Templates, so daß ein SQL-Template einer Menge (*DataSet*) zugewiesen werden kann. Über das Mengentool soll die Möglichkeit bestehen, ein Template aus dem Templatekatalog auszuwählen und der aktuellen Menge, welche das Mengentool gerade beschreibt, zuzuweisen. Die Zuweisung eines Templates zu einer Menge erfolgt dabei in der Art, daß zum einen ein direkter Verweis auf die Identifikationsnummer des Templates in die Menge integriert wird und zum anderen wird das SQL-Statement, welches das Template in Abhängigkeit von den aktuell selektierten Attributen abbildet, fest in der Menge hinterlegt.

Die Vorgehensweise der fest kodierten Integration des aus dem Template abgeleiteten SQL-Statements ist deshalb gewählt worden, da sich der Aufbau eines aus einem Template generierten SQL-Statements nur dann notwendigerweise ändern muß, wenn die Attribute der Selektionsmenge eines DataSets bezüglich des vorherigen Zustands entweder verringert oder erweitert werden. Daher muß der Templatemechanismus zur Generierung eines SQL-Statements nur dann ein den aktuellen Parametern angepaßtes Statement dynamisch generieren, wenn einer der beiden zuvor genannten Zustandsänderungen eintritt. Des weiteren besteht der Vorteil darin, daß bei der Speicherung (in das Archiv der Reportingplattform, siehe *Kapitel 4.4.3 Das Archiv*) sowohl der Verweis auf den Templateidentifikator als auch das aktuelle SQL-Statement selbst mit archiviert (*serialisiert*) werden, worauf diese Informationen beim Auslesen einer Menge aus dem Archiv (*deserialisieren*) sofort wieder in der Mengenbeschreibung bereitstehen.

Ein durch ein Template generiertes SQL-Statement wird nicht durch den SQL-Generator erzeugt, sondern durch den Templatemechanismus. Daher muß der Aufruf des SQL-Generators bei der Verwendung von Templates redefiniert werden, was aber nicht bedeutet, daß der Generator nicht mehr benötigt wird, denn zum einen hat der Anwender immer noch die Möglichkeit, einer Menge dynamische Filter über das Mengentool zuzuweisen, und zum anderen kann er sich die Selektionsattribute sortiert in der Ergebnismenge anzeigen lassen. Hierdurch kann jedes SQL-Statement einer Menge optional mit einer ORDER BY-Klausel versehen werden.

Da der Benutzer des Systems die Möglichkeit haben soll, sich zur Laufzeit dynamisch einen DataSet für seine individuellen Informationsbedürfnisse zu erstellen, wurde ein Mengentool implementiert, über welches sich eine solche Datenmenge erzeugen läßt. Innerhalb dieses Tools werden zu Beginn alle aus dem Repository verfügbaren Attribute angezeigt, aus denen sich der Anwender die Attributmenge zusammenstellen kann, welche die Ergebnismenge der Datenbankanfrage repräsentiert. Die so über das Mengentool erzeugte Menge kann darauf an jedes beliebige Werkzeug (Kreuztabellen-, Graphik- und Tabellentool) übergeben werden, um innerhalb dieses Werkzeugs die mittels SQL-Generator erzeugte Ergebnismenge entsprechend der jeweiligen Tooldarstellung anzuzeigen.

Einem DataSet wird über die Identifikationsnummer ein Template direkt zugewiesen und fest in den DataSet eingefügt. Dazu werden drei Schnittstellenfunktionen bereitgestellt, die aus einem DataSet eine TemplateID auslesen, eine TemplateID in einen DataSet schreiben und testen, ob eine Menge einen Templateeintrag enthält.

$$type\ DataSet\_GetTemplateID: Handle \rightarrow TemplateID$$

$$type\ DataSet: \_SetTemplateID: Handle \times TemplateID \rightarrow Handle$$

$$type\ DataSet\_HasTemplateID: Handle \rightarrow Bool$$

Da das SQL-Statement eines mit einem Template verknüpften DataSets nicht über den SQL-Generator erstellt wird, muß der bestehende SQL-Generatoraufruf modifiziert werden. Der eigentliche SQL-Generator wird nur aufgerufen, wenn weder eine vordefinierte Anweisung noch ein Template mit dem

DataSet verknüpft sind. Daher muß die Menge bzw. das Handle der Menge, die an den Generator übergeben wird, hinsichtlich des realisierenden SQL-Anweisungstyps untersucht werden. Der nachfolgend spezifizierte SQL-Generatoraufruf entspricht dem redefinierten Aufruf, der zwischen einem Template, einer vordefinierten Anweisung und einer durch den SQL-Generator zur erzeugenden SQL-Anfrage unterscheidet.

$$type\ SQLMgr\_BeginSqlStatement: Handle \rightarrow Bool$$
$$type\ SQLMgr\_GetSqlStatement: Handle \rightarrow Bool$$
$$type\ SQLMgr\_GetSqlStatement_2: Handle \rightarrow Bool$$

$$SQL\_Mgr\_GetSqlStatement(handle) \underline{\Delta}$$
$$if\ DataSet\_HasTemplateID(handle)$$
$$Let\ Tid = DataSet\_GetTemplateID(handle)\ in$$
$$Generiere\_Template(handle, Tid)$$
$$else\ if\ DataSet\_HasDefStmt(handle)$$
$$then\ DataSet\_GetDefStmt(handle)$$
$$else\ Let\ ok = SQLMgr\_BeginSqlStatement(handle)\ in$$
$$SQLMgr\_GetSqlStatement_2(handle)$$

Wird eine Menge ermittelt, die einen Templateverweis enthält, so wird die Identifikationsnummer des Templates ausgelesen und zusammen mit der Menge über die Funktion *Generiere_Template* an den Templatemechanismus übergeben. Dieser liest darauf die tatsächlich selektierten Attribute des DataSets aus, um daraus zusammen mit den entsprechenden, in den Templatetabellen verwalteten Bausteinen, das aktuelle SQL-Statement zu generieren.

Die wesentlichen Funktionsaufrufe zur Generierung eines SQL-Statements aus einem DataSet mit den darin enthaltenen Selektionsattributen und der Templatereferenznummer werden anschließend in abstrakter Syntax formuliert.

Zuvor werden aber noch einige benötigte Datentypen und im System bereits verfügbare Hilfsfunktionen, die zum Auslesen der Selektionsattribute eines DataSets dienen, eingeführt:

$$SqlStmt = char^*$$
$$Handle = token$$
$$Attribute = Attribut - set$$
$$Attribut = char^+$$
$$TemplateID = N_1$$
$$type\ DataSet\_Attr\,Read\_Begin: Handle \rightarrow Bool$$
$$type\ DataSet\_Attr\,Read\_Next: Handle \rightarrow Bool$$
$$type\ DataSet\_Attr\,Read\_Get: Handle \rightarrow Attribut$$
$$type\ DataSet\_Attr\,Read\_IsValid: Handle \rightarrow Bool$$

Ein SQL-Statement wird nur generiert, wenn auch tatsächlich Attribute in der Menge selektiert worden sind, denn ansonsten wird ein leeres SQL-Statement zurückgegeben. Das Auslesen der Selektionsattribute einer nicht leeren Selektionsattributmenge erfolgt durch den Aufruf der Hilfsfunktion *Lese_Selektionsattribute*, welche die ausgelesenen Attribute an die Funktion *Generiere_Template* zurückgibt. Nachdem die Selektionsattribute ermittelt worden sind, wird die Funktion *Generiere_SQL_Statement* aufgerufen, die das eigentliche SQL-Statement aus den Templatebausteinen und den selektierten Attributen erzeugt.

$$type\ \ Generiere\_Template: \ \ Handle \times TemplateID \rightarrow SqlStmt$$

$$Generiere\_Template(handle, vtid) \underline{\Delta}$$
$$if\ \ DataSet\_Attr\,\mathrm{Re}ad\_Begin(handle)$$
$$then\ \ Let\ \ attrsel = \{\ \}\ \ in$$
$$Let\ \ attrSel = Lese\_Selektionsattribute(handle, attrSel)\ \ in$$
$$Generiere\_SQL\_Statement(attrsel, vtid)$$
$$else\ \ \{\ \}$$

$$type\ \ Lese\_Selektionsattribute: \ \ Handle \times Attribute \rightarrow Attribute$$

$$Lese\_Selektionsattribute(handle, attrSel) \underline{\Delta}$$
$$if\ \ DataSet\_Attr\,\mathrm{Re}ad\_IsValid(handle)$$
$$then\ \ Let\ \ attrSel = DataSet\_Attr\,\mathrm{Re}ad\_Get(handle)\ \ in$$
$$Let\ \ ok = DataSet\_Attr\,\mathrm{Re}ad\_Next(handle)\ \ in$$
$$Lese\_Selektionsattribute(handle, attrSel)$$
$$else\ \ attrSel$$

Beim Entwurf der Funktion *Generiere_SQL_Statement* ist allerdings innerhalb dieser Darstellung darauf verzichtet worden, zu testen, ob einzelne Bausteine auch wirklich in dem verwiesenen Template enthalten sind, so daß hierbei davon ausgegangen wird, jedes Template enthielte *Konstante Filterausdrücke*, *Gruppenfilterausdrücke* und eine *Subquery*.

$$type\ \ Generiere\_SQL\_Statement: \ \ Attribute \times TemplateID \rightarrow SQLStmt$$

$$Generiere\_SQL\_Statement(attrSel, tid) \underline{\Delta}$$
$$Let\ \ sqlstmt_H = Generiere\_Haup\,\mathrm{tan}\,frage(attrSel, tid)\ \ in$$
$$Let\ \ subquery = Generiere\_Subquery(tid)\ \ in$$
$$Generiere\_Gesam\,\mathrm{tan}\,frage(sqlstmt_H, subquery)$$

Der Aufruf der Funktion *Generiere_Hauptanfrage* dient der Erzeugung der SELECT-, FROM und WHERE-Klausel der oberen Anfrage, in welche die selektierten Attribute in die SELECT-Klausel integriert werden. Weiterhin werden in die WHERE-Klausel alle Filterausdrücke der konstanten und Gruppenfilter integriert, deren Filterattribute nicht aggregiert werden. Sollten in Filterausdrücken zu aggregierende Attribute auftreten, so sind diese als Parameter in die HAVING-Klausel einzufügen. Die notwendigen Funktionalitäten zur Erstellung eines SQL-Statements durch den Templatemechanismus wurden bereits in *Kapitel 3.5.3.2 Voraussetzungen für die automatische Erstellung eines SQL-Statements* durch Invarianten spezifiziert.

Die Vorgehensweise bei der Erzeugung eines SQL-Statements unterteilt sich in die Erstellung der Hauptanfrage inklusive der Filterausdrücke, in die Anpassung der Subquery und in die Integration der Subquery in die obere Anfrage. Zur Erstellung der Hauptanfrage wird die bereits vorhandene Funktionalität des SQL-Generators verwendet, aus den Selektionsattributen und den aus den Templatetabellen gewonnenen Filterausdrücken ein Defaultstatement zu erzeugen. Dazu wird eine temporäre Menge erzeugt, deren statischen Filter ein dynamisches Filterhandle zugewiesen wird, welches wiederum den aus den Tabellen gewonnenen Filterbaum enthält.

$$type\ \ Generiere\_Haup\tan frage: \ \ Attribute \times TemplateID \rightarrow SqlStmt$$

$$Generiere\_Haup\tan frage(attrSel, tid)\underline{\Delta}$$
$$\quad Let\ \ handle = DataSet\_Create(\ )\ \ in$$
$$\quad\quad Let\ \ handle = Weise\_Attribute\_Zu(handle, attrSel)\ \ in$$
$$\quad\quad\quad Let\ \ handle = Lese\_Filter(tid)\ \ in$$
$$\quad\quad\quad\quad Let\ \ ok = SQLMgr\_BeginSqlStatement(handle)\ \ in$$
$$\quad\quad\quad\quad\quad SQLMgr\_GetSqlStatement(handle)$$

Die Hilfsfunktion *Weise_Attribute_Zu* hat die Aufgabe, eine Attributmenge einer Menge (Mengenhandle) zuzuweisen. Innerhalb dieser Funktion wird die vom System bereitgestellte Funktion *DataSet_AddSelAttribute* verwendet, deren Aufgabe es ist, eine Menge jeweils um ein Attribut in der Selektionsmenge zu erweitern.

$$type\ \ DataSet\_AddSelAttribute: \ \ Handle \times Attribut \rightarrow Handle$$
$$type\ \ Weise\_Attribute\_Zu: \ \ Handle \times Attribute \rightarrow Handle$$

$$Weise\_Attribute\_Zu(handle, attrSel)\underline{\Delta}$$
$$\quad if\ \ card(attrSel) \succ 0$$
$$\quad then\ \ Let\ \ attr = attr \in attrSel\ \ in$$
$$\quad\quad Let\ \ handle = DataSet\_AddSelAttribute(handle, attr)\ \ in$$
$$\quad\quad\quad Weise\_Attribute\_Zu(handle, attrSel - \{attr\})$$
$$\quad else\ \ handle$$

Eine Subquery wird nach der Modellierung durch das Templatemodellierungstool als ein String in einer Tabelle hinterlegt, der je nach Struktur der Subquery parametrisiert wird. Die Parametrisierung ist dann erforderlich, wenn eine Subquery erzeugt wird, auf die ein Mengenoperator angewendet wird, der eine Vergleichsspalte aus der oberen Anfrage bedingt oder innerhalb der Vergleichsattribute in Filterausdrücken Attribute verwendet werden, die aus Tabellenspalten der Hauptanfrage gewonnen werden. Die zuletzt genannten Filterausdrücke müssen deshalb parametrisiert werden, da die Struktur der Hauptanfrage entscheidend von den konkret gewählten Selektionsattributen abhängig ist, denn diese entscheiden neben den integrierten Filterausdrücken darüber, welche Tabelleneinträge in der FROM-Klausel erforderlich sind. So kann es notwendig sein, eine aufgrund der Reduktion der Selektionsattribute entfernte Tabelle dennoch wieder einzufügen, da ein Vergleichsattribut eines Filterausdrucks der Subquery aus dieser gewonnen wird.

Der zuvor beschriebene Sachverhalt läßt die Schlußfolgerung zu, daß nur dann eine Subquery an das aktuelle SQL-Statement angepaßt werden muß, wenn weder eine Vergleichsspalte noch ein Vergleichsattribut eines Filterausdrucks aus einer Tabelle der Hauptanfrage ermittelt werden muß. Dazu muß eine Subquery [NOT] EXISTS als Mengenoperator aufweisen und keinen externen Attributvergleich beinhalten.

$$Subqueries = id \xrightarrow[m]{} Vergleichsattribut \times Subquerystring \times FilterVerglExtern$$
$$Vergleichsattribut = Attribut$$
$$Subquerystring = char^{+}$$
$$FilterVerglExtern = Attribute$$

$$type\ Generiere\_Subquery:\ TemplateID \rightarrow char^{+}$$

$$Generiere\_Subquery(tid)\underline{\Delta}$$
$$Let\ (verglattr, subquerystring, attrextern) = Subqueries(tid)\ in$$
$$if\ verglattr \neq NIL \vee attrextern \neq NIL$$
$$then\ Let\ subquerystring = Aktualisiere\_Subquery(tid)\ in$$
$$(subquerystring)$$
$$else\ subquerystring$$

Nachdem sowohl die Hauptanfrage als auch die Subquery generiert worden sind, wird die Subquery in die entsprechende WHERE- oder HAVING-Klauseln des SQL-Statements integriert.

$$type\ Generiere\_Gesamtanfrage:\ SqlStmt \times Subquerystring \rightarrow SqlStmt$$

In Abbildung 24 wird die Erweiterung des Systems um das Templatekonzept dargestellt. Dabei sollen insbesondere die Kommunikationsschnittstellen zu einem DataSet, dem SQL-Generator, den Templatetabellen und den Repositorytabellen hervorgehoben werden.

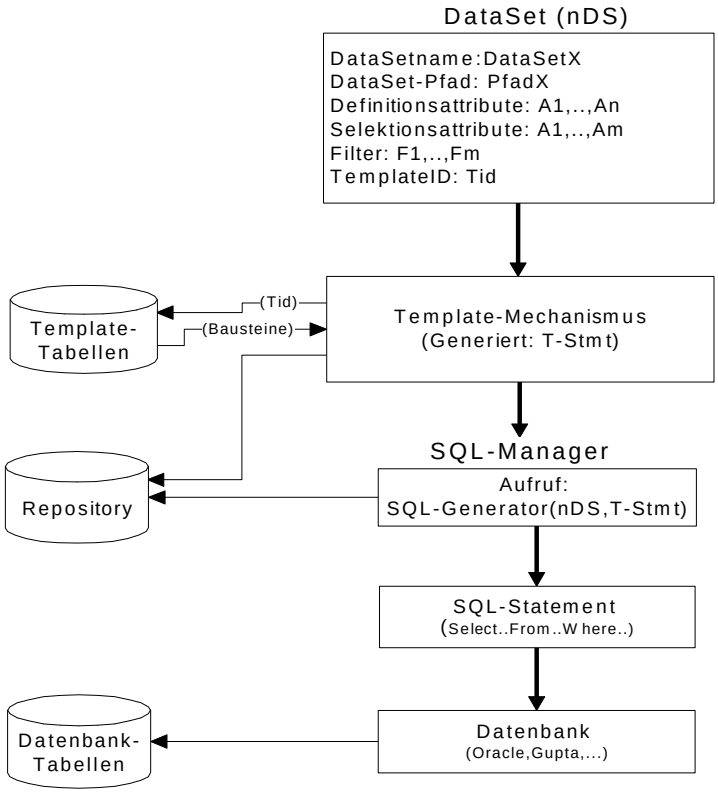


Abbildung 24: Transformationsweg von DataSets über den Templatemechanismus zu SQL-Statements

3.5.6 Herleitung des Datenmodells und der Algorithmen zur Bearbeitung von SQL-Templates

Die folgenden Kapitel erläutern die im Repository hinterlegte Tabellenstruktur zur Bearbeitung der SQL-Templatebausteine und die darauf angewendeten Algorithmen.

3.5.6.1 Klassifikation der Templates

Bei der Templateklassifikation werden (als Erweiterung der bereits in *Kapitel 3.2.4 Festdefinierte SQL-Statements* spezifizierten Hierarchie) die Templates in der obersten Ebene zunächst einem *Unternehmensbereich* indirekt zugeordnet, welcher aus einem *Bereichsnamen*, der den jeweiligen Unternehmensbereich spezifiziert, und der zugehörigen BereichsID (Identifikator) besteht. Durch diese Struktur werden die Templatebausteine genau dem Unternehmensbereich zugewiesen, für dessen bestimmtes Sachgebiet sie den Informationsumfang darstellen.

$$1.\ Unternehmensbereich = ID \xrightarrow[m]{} Bereichsname \times BereichsID$$

Die zuvor genannten Sachgebiete einzelner Unternehmensbereiche werden nun in der Tabelle *Bereichsgruppierung* über das Tupel *Gruppenname* beschrieben. Dabei wird das zweistellige Tupel (Gruppenname × TemplateID) wie folgt unterschieden:

- in die Beschreibung eines Gruppennamens im Sinne eines Sachgebiets, bei der das Feld *TemplateID* leer ist.
- in die Beschreibung eines Gruppennamens, der den Namen eines Templategruppenfilters darstellt und bei dem über die *TemplateID* direkt auf die Identifikationsnummer des Basistemplates verwiesen wird.

Die Tabelleneinträge, auf die eine *GruppenID* zeigt, deren *TemplateID* leer ist, können als (sub-) strukturierende Pfadangaben der ihnen zugewiesenen Unternehmensbereiche verstanden werden.

$$2.\ Bereichsgruppierung = Gruppen\ ID \xrightarrow[m]{} Gruppenname \times TemplateID$$

In der Tabelle *Bereichshierarchie* wird die Hierarchie im Sinne einer Pfadstruktur innerhalb der einzelnen Unternehmensbereiche über die Sachgebiete, die in Tabelle Bereichsgruppierung spezifiziert wurden, beschrieben. Der jeweilige Unternehmensbereich wird durch die *BereichsID* bestimmt. Die Beschreibung der Pfadstruktur wird durch eine Zweifachverkettung vorgenommen. Innerhalb dieser wird das erste Glied der Kette durch die *Ebene1*, welche den oberen Pfad der ihr untergeordneten *Ebene2* angibt, beschrieben.

$$3.\ Bereichshierarchie = BereichsID \times Ebene1\ (direkt) \times Ebene2\ (indirekt)$$

3.5.6.2 Hierarchie der Templates

Die an der Hierarchisierung von Templates beteiligten Tabellen haben die Funktion der Speicherung der Templateparameter für die SELECT-Klausel des Templatestatements. Zusätzlich werden die in der WHERE-Klausel enthaltenen Gruppenfilter und konstanten Filter in eigens dafür vorgesehenen Tabellen gehalten. In der Tabelle *Templates* wird jedes parametrisierte Template unter seiner Identifikationsnummer *TemplateID* abgelegt. Die über die Positionsvariablen parametrisierten Templateskeletts können später bei der Adaption durch die Auswahl der interessierenden Attribute und Filter konkret in ein SQL-Statement transformiert und durch den SQL-Generator zur Laufzeit an das RDBMS übergeben werden.

$$4.\ Templates = TemplateID \xrightarrow[m]{} Template\text{-}Statement$$

Die in der SELECT-Klausel potentiell integrierbaren Attribute des Templates werden unter der Identifikationsnummer des Templates *TemplateID* in der Tabelle *Template_Attribute* gehalten. Der Eintrag, der unter der *TemplateID* abgespeicherten Tabellenzeile, enthält das zweistellige Tupel *(Positionsmenge × Attributmenge)*, durch welches elementweise die Position des konkreten Attributnamens beschrieben wird.

$$5.\ Template\_Attribute = TemplateID \xrightarrow[m]{} Positionsmenge \times Attributmenge$$

In der Tabelle *Filterparts* werden die Gruppenfilter zu den *Gruppennamen* aus der Tabelle *Bereichsgruppierung* gehalten, denen direkt ein Template über dessen TemplateID zugeordnet worden ist. Der Bezug wird durch das Feld *BereichsID* hergestellt, so daß bei Auswahl eines Templatestatements über den angebotenen Gruppennamen automatisch der entsprechende Filterparameter in das Statement integriert werden kann. Damit Filtereinschränkungen vermieden werden, kann das Feld *Filterparameter* auch leer sein.

Das Auftreten eines leeren Filterparameters bei einem Gruppenfilter hat die generelle Auswirkung, daß andere optional in das Statement zu integrierende Filter nicht mehr in das Statement eingebunden werden können. Diese Einschränkung findet sicherlich nur in dem Fall Anwendung, falls der Benutzer ein Templatestatement nicht direkt über die Templateklassifikation auswählt, sondern durch eine indizierte Suche über ausgewählte Attribute, da ihm bei dieser Vorgehensweise zu Beginn der Filterauswahl zunächst alle verfügbaren Gruppenfilter zur Integration angeboten werden.

Das Datenfeld Filtermenge beinhaltet die Verweisnummern auf die Filterausdrücke, die zusammengesetzt den Gruppenfilter ergeben. Die Verweisnummern stellen die jeweilige *FilterID* der Tabelle *Filter_Var* dar, unter der die Bestandteile des Filterausdrucks abgespeichert worden sind. Die Spalte *TemplateID* ist notwendig, wenn ein Template direkt unter seiner Identifikationsnummer ausgewählt wurde, um nachträglich die zu dem Template gehörigen Gruppenfilter wählen zu können.

$$6.\ Filterparts = GFilterID \xrightarrow[m]{} Filtermenge \times Filtername \times BereichsID \times TemplateID$$

In der Tabelle *Filter_Var* sind alle Bestandteile der Filterausdrücke enthalten, die wiederum Bestandteile eines Gruppenfilters sind. Die Spalte *Filterattribut* bezeichnet den linken Teilausdruck und die Spalte *Operator* den Vergleichsoperator eines Filterausdrucks. Durch die Spalte *VerglFlag* wird spezifiziert, ob es sich bei dem rechten Teilausdruck des Filters um eine Konstante (*VerglFlag*=0) oder um ein Vergleichsattribut (*VerglFlag*=1) handelt. Diese Information wird später bei der Generierung eines SQL-Statements aus dem Template benötigt, um die Korrektheit des Filterausdrucks zu erhalten. Über die Spalte *Position* wird die Parameterposition des Filterausdrucks innerhalb des Templates bestimmt.

$$7.\ Filter\_Var = FilterID \xrightarrow[m]{} Position \times Filterattribut \times Operator \times VerglWert \times VerglFlag$$

Die Tabelle *Filter_Konst* dient der Verwaltung von Filterausdrücken, die konstant in ein Template eingebunden werden müssen. Dies bedeutet, daß der jeweilige Filterausdruck, unabhängig davon ob ein oder welcher Gruppenfilter integriert worden ist, in das Template mit einbezogen werden muß. Die Spaltenstruktur entspricht bis auf die Spalte *TemplateID* der der Tabelle Filter_Var, denn diese Spalte stellt die Verknüpfung zum Template dar, so daß bei Auswahl des Templates dessen ID identisch zu der in der Spalte *TemplateID* ist, automatisch der Filterausdruck referenziert und integriert werden kann.

$$8.\ Filter\_Konst = FilterID \xrightarrow[m]{} Position \times Filterattribut \times Operator \times VerglWert \times VerglFlag \times TemplateID$$

In der Tabelle *Subqueries* wird der parametrisierte Subquerystring und dessen Verweise auf die in der Tabelle *Filter_Sub* eingetragenen Vergleichsattribute gehalten. Die Verweise auf die Vergleichsattribute lassen sich in der Spalte *Attributmenge* wiederfinden, denn diese enthält die Menge der *FilterIDs*, um indiziert auf die in der Subquery integrierten Vergleichsattribute über die Tabelle *Filter_Sub* zugreifen zu können. Die Vergleichsattribute stellen innerhalb einer Subquery Parameter dar, denn sie bezeichnen eine Tabellenspalte der oberen Anfrage, die mit einer zulässigen Tabellenspalte der Subquery in Form eines *Outer Joins* verknüpft worden ist. Die Parametrisierung des Vergleichsattributs ist deshalb notwendig, weil aufgrund der generischen Auswahl der Attribute der oberen SELECT-Klausel die Struktur des Basisstatements stark variieren kann, was sich auch auf die Anzahl der Tabellen in der FROM-Klausel und somit auf deren Tupelvariablen auswirkt.

Gerade die Tupelvariablen sind für die Referenzierung des an einem Outer Join in der Subquery beteiligten Vergleichsattributs aus seiner in der oberen Anfrage stehenden Basistabelle insofern von Bedeutung, als daß sie zur Laufzeit vor das Vergleichsattribut gestellt werden müssen. Daher wurde der Ansatz gewählt, nur den Namen des Vergleichsattributs zu speichern, ohne eine Tupelvariable von vornherein fest vorzugeben.

$$9.\ Subqueries = TemplateID \xrightarrow[m]{} Position \times Attributmenge \times Subquery\_String$$

Die für die Subquery benötigten Parameter (die Vergleichsattribute) werden in der Tabelle *Filter_Sub* abgespeichert. In dem Datenfeld *Vergleichsattribut* steht der konkrete Name des Vergleichsattributs und unter *Position* wird die Parameterposition in der Subquery bestimmt.

$$10.\ Filter\_Sub = FilterID \xrightarrow[m]{} Position \times Vergleichsattribut$$

3.5.6.3 Indextabellen zur Auffindung von Templates über Attributkombinationen

Die Indextabellen ermöglichen eine für den Benutzer komfortable Suchmöglichkeit nach Templates. Bei dieser Suche bilden die vom Benutzer ausgewählten Attribute die Suchparameter.
In der Tabelle *Templatemenge* wird jeder *AttributID* eines in einer Template-SELECT-Klausel enthaltenen Attributs die Menge aller Templates, genauer deren *TemplateID*, zugeordnet, in welcher das Attribut vorkommt. Durch die Tabelle kann also ein indizierter Zugriff auf alle Templates erfolgen, deren Attributmengen das gewählte Attribut mit der AttributID enthalten, wodurch der Suchraum nur auf die relevanten Templates eingeschränkt wird.

$$11.\ Templatemenge = AttributID \xrightarrow[m]{} (TemplateID)\text{-}set$$

Die über die AttributID aus der Tabelle Templatemenge ausgelesene Menge an Templates kann nun elementweise in der Tabelle *Attributmenge* abgearbeitet werden, indem ein Zugriff für jede *TemplateID* der ermittelten Templatemenge auf die gesamte Attributmenge des Templates erfolgt, die durch das Feld *(AttributID)-set* spezifiziert wird.

Ein für die Geschwindigkeit der Suche wesentlicher Punkt ist, daß von allen Templatemengen, die durch eine AttributID aus der Tabelle Templatemenge ausgelesen worden sind, die jeweilige abgearbeitete TemplateID aus der Menge entfernt wird, da zuvor die Schnittmenge aller Templatemengen gebildet wird. Durch diese Vorgehensweise wird die doppelte Abarbeitung eines Templates vermieden, falls mehr als ein gewähltes Attribut in der Attributmenge eines Templates enthalten ist.

$$12.\ Attributmenge = TemplateID \xrightarrow[m]{} (AttributID)\text{-}set$$

3.5.7 Algorithmen zur Generierung eines Templates

Die Algorithmen zur Erzeugung eines Templates ermitteln die parametrisierbaren Anteile eines Templates und hinterlegen diese in aufbereiteter Form in den Repositorytabellen. Zusätzlich werden außer den konkret benötigten Templateparametern auch noch die Informationen aufbereitet und in den Tabellen mit abgespeichert, die die Mechanismen für die spätere Erstellung eines SQL-Statements aus einem Template unterstützen (Metadaten).
Die Reihenfolge der Beschreibung der Generierungsmechanismen soll der sukzessiven Vorgehensweise bei der Modellierung eines Templates durch den Modellierer entsprechen und die nach Ablauf der einzelnen Modellierungsphasen entstandenen Parameter und Verwaltungsinformationen herausstellen.

Zu Beginn der Modellierung wird die Attributmenge bestimmt, die in die obere SELECT-Klausel des Templates integriert werden soll. Die Reihenfolge, die der Modellierer bei der Auswahl der Attribute vornimmt, gilt auch weiterhin, wenn der Anwender sich die Attributmenge des Templates dynamisch in einer abweichenden Reihenfolge zusammenstellt. Daher ist die Information über die Position eines jeden Attributs in der SELECT-Klausel notwendig, die zusammen mit dem jeweiligen Attribut in der

Tabelle *Template_Attribute* hinterlegt wird. Die Funktion *Speichere_Template_Attribute* beschreibt den zur Abspeicherung notwendigen Algorithmus:

Type Speichere_Template_Attribute: Attributmenge $\times$ *TemplateID* $\longrightarrow$ *Bool*

Type inv_Speichere_Template_Attribute: Attributmenge $\longrightarrow$ *Bool*

inv_Speichere_Template_Attribute(attrmenge)

3.5.7.1 Herleitung eines SQL-Templates aus strukturähnlichen SQL-Statements

Die Herleitung eines Templates soll nachfolgend anhand eines durchgängigen Beispiels aufgezeigt werden, in dem verdeutlicht wird, wie nach dem Templatekonzept von mehreren konkreten SQL-Statements aus der Personalabteilung zu einem entsprechendem SQL-Template abstrahiert wird. Innerhalb des Beispiels wird gezeigt, wie die Vorgehensweise eines Modellierers bei der Erstellung eines Templates aussieht und wie die Templatemechanismen die vom Modellierer eingegeben SQL-Statementbestandteile um zusätzliche systeminterne Informationen erweitern und parametrisieren. Im ersten Schritt der Templategenerierung werden zunächst zwei Statements (Punkt: a - b) beispielhaft aufgeführt, deren Generierung einem bestimmten Informationsbedarf aus der Personalabteilung zugrundeliegt und die hinsichtlich ihrer grundlegenden Struktur Gemeinsamkeiten aufweisen.

Diese Aufgabe wird in der Praxis durch einen Modellierer vorgenommen, der mit der Anfragesprache SQL entsprechend vertraut ist, um auch sehr komplexe Anfragen formulieren zu können.

a) Das Statement ermittelt alle Mitarbeiter der Personalabteilung (Personalnummer, Name, Abteilungsnummer, Gehalt, Postleitzahl, Wohnort und Straße), was durch den Filter *a1.Abtnr=4* ausgedrückt wird, die mehr als das gesamtdurchschnittliche Gehalt vom Februar 2000 verdienen.

b) Das Statement ermittelt alle Mitarbeiter der Controllingabteilung (Personalnummer, Name, Abteilungsnummer, Gehalt, Postleitzahl, Wohnort und Straße), was durch den Filter *a1.Abtnr=5* ausgedrückt wird, die mehr als das gesamtdurchschnittliche Gehalt vom Februar 2000 verdienen.

```
SELECT    a1.Persnr, a1.Pname,a1.Abtnr, a2.Mgehalt, a3.PPlz, a3.POrt, a3.PStrasse
FROM      Persdat a1, Mgehalt a2, PAdresse a3
WHERE     a1.Persnr = a2.Persnr        AND
          a1.Persnr = a3.Persnr        AND
          a1.Abtnr  = 5                AND
          a2.MJahr  = 2000             AND
          a2.MMonat = 2                AND
          a2.Mgehalt > (SELECT AVG(s1.Mgehalt)
                        FROM Mgehalt s1
                        WHERE s1.Jahr    = 2000    AND
                              s1.Monat   = 2)
ORDER BY  1                                                    ;
```

Bei der syntaktischen Analyse der Strukturgemeinsamkeiten der beiden Statements wird deutlich, daß sowohl die Attributmengen, die Tabellenlinks, die Joins als auch die Subqueries, auf die jeweils ein relationaler Operator angewendet wird, in beiden Anfragen identisch sind. Des weiteren sind in beiden Statements die konstanten Filter zur Selektion des Februars 2000 enthalten. Die unterschiedlichen Bestandteile der Statements werden durch das Vorhanden- bzw. Nichtvorhandensein konstanter Filter, in Form der Abteilungsnummern für die Personalabteilung (*a1.Abtnr=4*) und für das Controlling (*a1.Abtnr=5*), spezifiziert.

Den durch die Analyse ermittelten Filtern für die Personal- und Controllingabteilung wird innerhalb des Konzepts eine Gruppierungsfunktion zugeordnet, da sie das Statement hinsichtlich der Semantik gruppieren und somit der Generierbarkeit jedes Ursprungsstatements der Personalabteilung aus dem Templatestatement Rechnung tragen. Durch die Gruppierung der Filter aus den strukturähnlichen Statements besteht also die Möglichkeit über die Auswahl verschiedener Gruppenfilter, diese a priori in ein Templatestatement einzubinden, um die direkte Auswahl eines in das Template verschmolzenen Statements zu erreichen.

Nachdem die Analyse der Strukturgleichheit bzw. -abweichung erfolgt ist, beginnt nun die Phase der Parametrisierung des Statements zu einem Template, welche sich prinzipiell je nach Komplexität des Templatestatements in mehrere Schritte unterteilt. Die Komplexität eines Statements drückt sich in der Summe der aus der SQL-Syntax verwendeten Ausdrücke aus. Allerdings muß hierbei in die zwingend notwendigen Ausdrücke, die sich aus der Struktur der Haupt-SELECT-Klausel ergibt (Basistemplate), und in die optional integrierbaren Ausdrücke, die sich aus der Integration der erweiterten Bausteine des Modellierers ergeben, unterschieden werden.

3.5.7.1.1 Generierung des Basistemplates

Die Generierung des Basistemplates im Beispiel erfolgt automatisch nach der Auswahl der Templateattribute über das Modellierungstool, da der Modellierer hierdurch die maximal darstellbare Attributmenge des SQL-Templates definiert. Die ausgewählten Attribute der Attributselektionsmenge eines DataSets werden zugewiesen und an den SQL-Generator übergeben. Das von dem SQL-Generator gelieferte Ergebnis entspricht dem Basistemplate bei Auswahl der maximal darstellbaren Attributmenge.

Das Basistemplate besteht aus der mit den Attributen versehenen SELECT-Klausel, aus der FROM-Klausel, welche die Fremd- und Basistabellen mit Tupelvariablen versehen enthält und bei Erfordernis von Joins aus der WHERE-Klausel, in der die notwendigen Attributreferenzierungen über einen Fremdschlüssel einer Fremdtabelle mit der Basistabelle formuliert werden. Nachfolgend wird die Vorgehensweise zur Erstellung des Basistemplates aufgezeigt:

Der Modellierer wählt Attributmenge A über das Mengentool aus:

$$A = \begin{Bmatrix} Personalnummer, Personalname, Abteilungsnummer, Monatsgehalt, Postleitzahl, \\ Wohnort, Straße \end{Bmatrix}$$

⇨ Das System generiert automatisch das Basisstatement (Defaultstatement):

```
SELECT a1.Persnr, a1.Pname,a1.Abtnr, a2.Mgehalt
FROM   Persdat a1, Mgehalt a2
WHERE  a1.Persnr = a2.Persnr     ;
```

⇨ Das System ermittelt automatisch die Positionsnummer für jedes Attribut in der SELECT-Klausel, um einen korrekten Bezug zu einer eventuell einzubindenden GROUP BY- oder ORDER BY-Klausel herzustellen.

⇨ Das System erweitert automatisch die Tabelle *Template_Attribute* um das neue Template, das unter einer TemplateID mit seiner Attributmenge einzutragen ist:

Positionsangaben in den geschweiften Klammern bezeichnen Attribute, Tabellen und Joins.

Template_Attribute	TemplateID	Attributname	Position
	1	Personalnummer	1
	1	Personalname	2
	1	Abteilungsnummer	3
	1	Monatsgehalt	4

Tabelle 8: Zustand der Tabelle Template_Attribute

⇨ Das System generiert automatisch das *Basistemplate* aus dem Ergebnis der zuvor ermittelten Positionsnummern:

```
SELECT {1,} {2,} {3,} {4,}
FROM   {5}
WHERE  {6 }       ;
```

Die Parameter, die in den geschweiften Klammern des Basistemplates dargestellt werden, entsprechen der Position des konkreten Ausdrucks an dieser Stelle. In der SELECT-Klausel stellt jeder Parameter die Position des Attributs dar, welches in dem durch den Generator erzeugten Statement geliefert wurde. Die Attribute werden anschließend unter der aktuellen Templatenummer in eine Tabelle abgespeichert, in der zu jedem Template die maximal darstellbaren Attributnamen gehalten werden.

Die Notwendigkeit der Speicherung der Templateattribute besteht darin, daß die SELECT-Klausel eines Templates im Verhältnis zu den vordefinierten Statements keine konstante Struktur hat, da zur Laufzeit die Möglichkeit besteht, daß sich der Anwender (oder Modellierer) die ihn interessierenden Attribute in beliebiger Kombination zusammenstellt.

Die Konsequenz aus diesen anwenderspezifischen Attributkombinationen ist, daß sich eine vom Basistemplate abweichende Struktur ergeben kann, da die Struktur des Basistemplates auf der maximalen Attributmenge beruht. Eine Strukturabweichung könnte durch die Reduktion der Attributmenge ausgelöst werden, denn sofern für die Attribute unterschiedliche Basistabellen benötigt werden, wird auch eine Verringerung der Anzahl der Basis- und Fremdtabellen erforderlich.

Die bereits vorhandene Funktionalität des SQL-Generators wird genutzt, indem für jedes Template unmittelbar nur die Attributnamen gespeichert werden, um daraus zur Laufzeit, nach Auswahl der Attributkonstellation über einen DataSet, durch den SQL-Generator ein neues, der aktuellen Attributmenge angepaßtes Basisstatement zu erzeugen.

3.5.7.1.2 Generierung von Filtern innerhalb des Basistemplates

Das Basistemplate kann in der WHERE-Klausel um die Filterausdrücke erweitert werden, deren Filterattribute kompatibel zu den Attributen in der Haupt-SELECT-Klausel sind. Die Kompatibilität ist dann gewährleistet, wenn die Filterattribute aus denselben Basistabellen stammen, die bereits für die Attributmenge des Basistemplates benötigt und integriert wurden, oder wenn aufgrund der Repositoryinformationen weitere Basistabellen über einen Join mit den schon vorhandenen Basistabellen verknüpft werden können. Daraus ergibt sich eine zulässige Filterattributmenge, die aus den Spalten der bereits referenzierten Tabellen der Templateattributmenge und den Spalten, die wiederum aus korrespondierenden Tabellen zu gewinnen sind, besteht.
Dazu ist es erforderlich, auch die korrespondierenden Tabellen mit deren Tupelvariablen der FROM-Klausel und die Joins der WHERE-Klausel in das Basistemplate zu integrieren. Da die Integration von Filterausdrücken aber seitens des Modellierers eine sehr tiefgehende Kenntnis der Repositorystruktur voraussetzt, wird dieser bei der praktischen Realisierung von Filterausdrücken durch systeminterne Funktionen entlastet, die ihm vorgeben, welche Attribute für die Integration in Filterausdrücke zulässig sind. Es ist dann lediglich die Kenntnis über die Datentypen der Vergleichskonstanten und der Bestimmung der Vergleichsoperatoren notwendig. Auch die Vergleichsoperatoren werden vom System zur Auswahl angeboten, um dadurch von vornherein nur gültige Operatoren für den Aufbau des gesamten Filterausdrucks zuzulassen.

3.5.7.1.3 Generierung von Gruppenfiltern

Ein durch den Modellierer festgelegter Gruppenfilter wird zunächst in einer Gruppenfiltertabelle abgespeichert. Der komplett zu speichernde Datensatz besteht aus folgenden Daten:

Datensatz = ({Position × FilterAttribut × Operator ×
VerglWert × VerglFlag } × Filtername × BereichsID × TemplateID)

Definition der einzelnen Datenfelder:

Position: Beschreibt die Position des Gruppenfilters in dem Basistemplate.
Filterattribut: Datenfeld beinhaltet den Attributnamen des linken Filterausdrucks.
Operator: Datenfeld beinhaltet den Operator des Filterausdrucks.
VerglWert: Datenfeld beinhaltet den Vergleichswert, der im rechten Filterausdruck steht.
VerglFlag: Datenfeld stellt ein Flag dar, welches dann gesetzt (=1) ist, wenn der rechte Filterausdruck ein Vergleichsattribut ist, sonst wenn es nicht gesetzt (=0) ist, dann ist der rechte Filterausdruck eine Vergleichskonstante.

Filtername: Datenfeld beschreibt die Filterbedeutung.

Um aber auch der Situation in der Praxis gerecht zu werden, daß ein Gruppenfilter aus mehreren Filterausdrücken bestehen kann, wird dem Modellierer die Möglichkeit geboten, eine Menge von Filterausdrücken zu generieren und einem Gruppenfilter zuzuweisen. Dazu bildet der Templatemechanismus intern eine Filtermenge, in der jeder Filter durch eine FilterID repräsentiert wird. Eine FilterID stellt einen Schlüssel für einen Filterausdruck in der Tabelle *Filter_Var* dar, in der jeder Eintrag aus dem Filterattribut, dem Operator, dem Vergleichswert und dem VergleichsFlag besteht. Der erforderliche Datensatz beinhaltet folgende Datenfelder:

Datensatz: (*Position* × *FilterAttribut* × *Operator* × *VerglWert* × *VerglFlag*)

Die jeweilige FilterID ergibt sich aus dem Inkrement der zuletzt eingetragenen FilterID der Tabelle. Nachdem alle Filter in der Tabelle eingetragen worden sind, wird der Gruppenfiltertabelle in der Spalte *Filtermenge* die Menge der FilterIDs (Filtermenge = {1..FilterID(n)}) zugewiesen.
Für das praktische Beispiel aus der Personalabteilung ergibt sich folgender Ablauf zur Integration der Gruppierungsfilter Personalabteilung (*a1.Abtnr=4*) und der Controllingabteilung (*a1.Abtnr=5*):

⇨ Der Modellierer wählt Gruppenfilter für die Personalabteilung (*a1.Abtnr=4*) aus:

⇨ *Abteilungsnummer = 4* (Personalabteilung).

⇨ Das System erweitert die Tabelle Filter_Var um Gruppenfilter *Personalabteilung.*

Filter_Var	FilterID	Filterattribut	Operator	VerglWert	VerglFlag	Position
	1	Abteilungsnummer	=	4	0	8

Tabelle 9: Zustand der Tabelle *Filter_Var* nach Festlegung des Filters für die Personalabteilung

⇨ Der Modellierer weist Gruppenfilter dem Unternehmensbereich Personal (*BereichsID=4*) zu.

⇨ Der Modellierer bestimmt die Ebene des Personalzweigs unter der das Template abgespeichert werden soll (GruppenID = X).

Gruppenfilter	VTemplateID	BereichsID	TemplateID	Filtername	Filtermenge
		4	1	Personal	{1}

Tabelle 10: Zustand der Tabelle *Gruppenfilter*

⇨ Der Modellierer wählt Gruppenfilter für die Controllingabteilung (*a1.Abtnr = 5*) aus:

⇨ *Abteilungsnummer = 5* (Controlling).

⇨ Das System erweitert Tabelle Filter_Var um Gruppenfilter *Controlling.*

Zustand der Tabelle *Filter_Var* nach Festlegung des Filters für die Controllingabteilung:

Filter_Var	FilterID	Filterattribut	Operator	VerglWert	VerglFlag	Position
	1	Abteilungsnummer	=	4	0	7
	2	Abteilungsnummer	=	5	0	7

Tabelle 11: Tabelle Filter_Var (neuer Zustand)

⇨ Der Modellierer weist Gruppenfilter dem Unternehmensbereich Controlling (*BereichsID=5*) zu.

⇨ Der Modellierer bestimmt die Ebene des Personalzweigs, unter der das Template abgespeichert werden soll (GruppenID = X).

Zustand der Tabelle *Gruppenfilter*:

Gruppenfilter	VTemplateID	BereichsID	TemplateID	Filtername	Filtermenge
		4	1	Personal	{1}
		5	1	Controlling	{2}

Tabelle 12: Tabelle Gruppenfilter (neuer Zustand)

Tabelle *Unternehmensbereich*:

Unternehmensbereich	UnterID	Bereichsname	BereichsID
	1	Disposition	1
	2	Vertrieb	2
	3	Einkauf	3
	4	Personal	4
	5	Controlling	5

Tabelle 13: Tabelle Unternehmensbereich

⇨ Das System aktualisiert die Tabellen *Bereichsgruppierung* und *Bereichshierarchie.*

Tabelle *Bereichsgruppierung:*

Bereichsgruppierung	GruppenID	Gruppenname	TemplateID
	1	Personalbearbeitung	NULL
	2	Personal	1
	3	Controlling	1

Tabelle 14: Tabelle Bereichsgruppierung

Tabelle *Bereichshierarchie*:

Bereichshierarchie	BereichsID	Direkt	Indirekt
	4	1	2

Tabelle 15: Tabelle Bereichshierarchie

Das parametrisierte Template weist nach der Erweiterung des Basistemplates um die Gruppierungsfilter folgende Gestalt auf:

```
SELECT {1,} {2,} {3,} {4,}
FROM   {5}
WHERE  {6 }{AND 7}  ;
```

3.5.7.1.4 Generierung von konstanten Filtern

Im Gegensatz zu den variablen bzw. Gruppenfiltern wird jeder konstante Filter fest in das Template integriert, so daß dieser unabhängig von der Auswahl des Gruppenfilters immer in das aus dem Template konkret generierte SQL-Statement eingebunden wird. Jeder konstante Filterausdruck wird mit der Nummer des Basistemplates in der Tabelle *Filter_Konst* hinterlegt, da diese bei der Erzeugung eines konkreten Statements aus einem Template über die entsprechende TemplateID abgearbeitet wird, um die zu integrierenden Filterausdrücke zu ermitteln und einzubinden. Der Datensatz besteht aus folgenden Datenfeldern:

Datensatz: (*Position* × *FilterAttribut* × *Operator* × *VerglWert* × *VerglFlag* × *TemplateID*)

Für das Beispiel aus der Personalabteilung ergibt sich folgender Ablauf zur Integration der konstanten Filter für die Jahreszahl (a2.Jahr=2000) und der Monatsangabe (a2.Monat=2):

⇨ Der Modellierer wählt aus:

⇨ Jahr = 2000.

⇨ Monat = 2.

⇨ Das System erweitert automatisch die Tabelle Filter_Konst um die Filter:

⇨ Datensatz1: (1,7,Jahr,=,2000,0,1); Vergl=0, da Vergleichskonstante.

⇨ Datensatz2: (2,8,Monat,=,2,0,1); Vergl=0, da Vergleichskonstante.

Zustand der Tabelle *Filter_Konst*:

Filter_Konst	FilterID	Position	Filterattribut	Operator	VerglWert	VerglFlag	TemplateID
	1	8	Jahr	=	2000	0	1
	2	9	Monat	=	2	0	1

Tabelle 16: Tabelle Filter_Konst

Das parametrisierte Template weist nach der Erweiterung des Basistemplates um die Gruppierungsfilter und die konstanten Filter folgend Struktur auf:

```
SELECT {1,} {2,} {3,} {4,}
FROM   {5}
WHERE  {6 } {AND 7} {AND 8} {AND 9}     ;
```

3.5.7.1.5 Generierung von Subqueries

Die Erzeugung von Unterabfragen muß hinsichtlich der Auswahl des Mengenoperators unterschieden werden, da sich aufgrund des Typs des Operators die Struktur bzw. die Komplexität sehr unterscheiden kann. So ist es gemäß der SQL-Syntax nur bei der Verwendung des EXISTS-Operators möglich, einen Kandidaten aus allen Attributen des Repositorys für das Attribut der SELECT-Klausel der Subquery zu wählen, während sich bei allen anderen Mengenoperatoren die Menge der wählbaren Attribute der SELECT-Klausel nur aus Vergleichsspalten zusammensetzt, die den gleichen Namen wie die Vergleichsspalte aus der oberen Anfrage vor dem Mengenoperator aufweisen.

Daher werden dem Modellierer systemseitig nach Festlegung des Mengenoperators und der Vergleichsspalte, außer bei Wahl des EXISTS-Operators, nur die zulässigen Attribute für die Integration in die SELECT-Klausel zur Auswahl angeboten. Nach Auswahl des Attributs wird analog zum Mechanismus der oberen Anfragegenerierung ein Defaultstatement unter Verwendung des SQL-Generators erzeugt.

Damit der Modellierer auch Erweiterungen an der Subquery in Form von Filterausdrücken vornehmen kann, werden diesem alle kompatiblen Attribute für den linken Filterausdruck (als Filterattribut) angeboten. Zusätzlich werden auch die zulässigen Vergleichsoperatoren vom System bereitgestellt. Der rechte Teilausdruck des Filters muß allerdings in die Auswahl einer Vergleichskonstante und eines Vergleichsattributs unterschieden werden. Soll der rechte Teilausdruck aus einer Konstante bestehen, so gibt der Modellierer diesen Vergleichswert unter Berücksichtigung des korrekten Datentypformats in ein dafür vorgesehenes Datenfeld ein.

Für das praktische Beispiel aus der Personalabteilung ergibt sich folgender Ablauf zur Integration der Subquery:

⇨ Der Modellierer wählt den Mengenoperator aus: $O_s = "\succ"$.

⇨ Der Modellierer wählt die Vergleichsspalte aus, die über den Operator mit dem Ergebnis der Subquery verglichen werden soll: *Vergleichspalte = Monatsgehalt* .

⇨ Der Modellierer wählt das Attribut für die SELECT-Klausel der Subquery aus:
$A_s = \{Durchschnittsgehalt\}$

⇨ Das System generiert automatisch das Basisstatement:

```
Mgehalt > (SELECT AVG(s1.Gehalt)
           FROM Mgehalt s1)
```

⇨ Das System erweitert automatisch die Tabelle *Filter_Sub* um die Vergleichsspalte *Monatsgehalt*:

⇨ Datensatz: (1, Monatsgehalt).

Filter_Sub	FilterID	Position	Vergleichsattribut
	1	1	Monatsgehalt

Tabelle 17: Zustand der Tabelle Filter_Sub:

⇨ Der Modellierer wählt konstante Filter in der Subquery aus:

⇨ Jahr = 2000.

⇨ Monat = 2.

⇨ Das System erweitert automatisch die Basissubquery:

```
(SELECT AVG(s1.Gehalt)
 FROM Mgehalt s1
 WHERE  s1.Jahr  = 2000  AND
        s1.Monat = 2)
```

Die parametrisierte Subquery weist nach der Erstellung folgende Struktur auf:

```
{H1} > (SELECT AVG(s1.Gehalt)
        FROM    Mgehalt s1
        WHERE   s1.Jahr  = 2000 AND
                s1.Monat = 2)
```

⇨ Der Parameter *{H1}* bestimmt die Position für das Vergleichsattribut, an der zur Laufzeit der Filter Monatsgehalt mit seiner Tupelvariablen eingefügt wird.

⇨ Das System erweitert die Tabelle Subqueries um die aktuelle Subquery.

⇨ Datensatz: (1,10,{1},Templatesubquery).

Zustand der Tabelle *Subqueries* nach der Erstellung der aktuellen Subquery:

Subqueries	TemplateID	Subquery_String	Attributmenge	Position
	1	{H1} > (SELECT AVG(s1.Gehalt) FROM Mgehalt s1 WHERE s1.Jahr = 2000 AND s1.Monat = 2)	{1}	10

Tabelle 18: Tabelle Subqueries

3.5.7.1.6 Generierung von virtuellen Templates

Das zuvor erstellte Basistemplate wird um die beiden Gruppenfilter erweitert, welche das Template an die individuellen Informationsbedürfnisse der Personalabteilung und der Controllingabteilung anpassen. Die Anpassung erfolgt in der Art, daß für jeden Bereich ein künstliches Template erzeugt wird, welches einen Verweis auf das Basistemplate und den jeweiligen Gruppenfilter enthält. Aus diesen *virtuellen Templates* wird in der späteren Anwendung das konkrete SQL-Statement generiert, welches zur Gewinnung der Ergebnismenge an die Datenbank abgesetzt wird. Um dem System diese Templates zur Verfügung zu stellen, müssen sie zuvor in einem Templatekatalog hinterlegt werden.
Für das praktische Beispiel aus der Personalabteilung ergibt sich folgender Ablauf zur Integration der Gruppierungsfilter Personalabteilung (*a1.Abtnr=4*) und der Controllingabteilung (*a1.Abtnr=5*) in das zuvor erstellte Basistemplate:

⇨ Der Modellierer wählt Gruppenfilter für die Personalabteilung (*a1.Abtnr=4*) aus:

⇨ *„Abteilungsnummer = 4“*.

⇨ Der Modellierer speichert das virtuelle Template unter dem Unternehmensbereich Personal / Personalgehälter:

⇨ Der Modellierer wählt Gruppenfilter für die Controllingabteilung (*a1.Abtnr=5*) aus:

⇨ *„Abteilungsnummer = 5“*.

⇨ Der Modellierer weist dem Unternehmensbereich Controlling / Personalgehälter Gruppenfilter zu.

Durch die Erweiterung der Bereichsgruppierung um das Feld VTemplateID kann der Gruppenfilter mit dem jeweiligen Filter direkt gewählt und in das Basistemplate eingefügt werden.

Neuer Zustand der Tabelle *Bereichsgruppierung:*

Bereichsgruppierung	GruppenID	Gruppenname	TemplateID	VTemplateID
	1	Personalbearbeitung	NULL	
	2	Personal	1	
	3	Controlling	1	
	4	Personal/Personalgehälter		2
	5	Controlling/Personalgehälter		3

Tabelle 19: Tabelle Bereichsgruppierung

Neuer Zustand der Tabelle *Gruppenfilter*:

Gruppenfilter	VTemplateID	BereichsID	TemplateID	Filtername	Filtermenge
	2	4	1	Personal	{1}
	3	5	1	Controlling	{2}

Tabelle 20: Tabelle Gruppenfilter (neuer Zustand)

Das parametrisierte Template weist nach der Erweiterung des Basistemplates um die Gruppierungsfilter aus dem virtuellen Template folgende Struktur auf:

```
SELECT {1,} {2,} {3,} {4,}
FROM   {5}
WHERE  {6 }{AND 7}  ;
```

3.5.7.2 Ableitung eines SQL-Statements aus einem hinterlegten SQL-Template

Um eine Datenbankanfrage über ein Template zu realisieren, muß das benötigte Template aus dem Katalog der verfügbaren bzw. modellierten Templates innerhalb des Mengentools ausgewählt werden. Dem aktuellen DataSet, der über das Mengentool erstellt werden soll, wird darauf das ausgewählte Template zugewiesen, indem die Definitions- und Selektionsattributen des DataSets mit den Attributen des Templates initialisiert werden. Des weiteren wird die ReferenzID in den DataSet eingetragen, mit der das Template hinterlegt worden ist, so daß zur Realisierung der Datenmenge über die verschiedenen Darstellungstools, ein SQL-Statement generiert werden kann. Dieses besteht aus den Bausteinen des eingetragenen Templates mit den aktuellen Selektionsattributen des DataSets.

Als Beispiel für eine praktische Anwendung wird einem DataSet das im vorherigen Abschnitt modellierte Template (Ermittlung aller Mitarbeiter aus der Controllingabteilung, die im Februar 2000 mehr als das durchschnittliche Gehalt verdienen) zugewiesen. Dabei sind für den Anwender bezüglich seiner momentanen Problemstellung allerdings nur die Personalnummer und der Mitarbeitername interessant.

In einem ersten Schritt wählt der Anwender oder Modellierer über das Mengentool das entsprechende Template aus, wodurch dieses automatisch dem aktuellen DataSet zugeordnet wird. Da die Definitions- und Selektionsattribute des DataSets bei der Zuweisung des Templates, in der Defaulteinstellung mit allen durch das Template darstellbaren Attributen initialisiert werden, muß die Selektionsmenge des DataSets in einem nächsten Schritt an den aktuellen Informationsbedarf angepaßt werden. Dies geschieht dadurch, daß der Anwender die nicht erforderlichen Attribute aus der Selektionsmenge deselektiert.
Zur Realisierung der Informationen, die durch den DataSet und dem korrespondierenden Template bereitgestellt werden, ruft der Anwender eines der Darstellungstools zur Auswertung der Menge auf. Das dynamisch generierte SQL-Statement, welches an den momentanen Informationsbedarf des Anwenders angepaßt ist, weist nach dessen Generierung durch den Templatemechanismus folgende Struktur auf:

```
SELECT a1.Persnr, a1.Pname
FROM   Persdat a1, Mgehalt a2
WHERE  a1.Persnr = a2.Persnr            AND
       a1.Abtnr  = 5 AND a2.Jahr=2000   AND
       a2.Monat  = 2                    AND
       a2.Gehalt > (SELECT AVG(s1.Gehalt)
                    FROM   Mgehalt s1
                    WHERE  s1.Jahr = 2000  AND s1.Monat = 2)
ORDER BY 1   ;
```

3.5.8 Zusammenfassung

Es wurde mit dem Templatekonzept eine sinnvolle Erweiterung der zunächst aus modellierten DataSets sowie den festdefinierten SQL-Statements realisierten SQL-Generierungsfunktionen durchgeführt, so daß ein überaus flexibler Mechanismus zum Eingrenzen der Abbildung des Informationsumfangs auf einen überschaubaren Umfang an zu spezifizierenden SQL-Statements entwickelt werden konnte. Der Fokus lag dabei auf der Möglichkeit der effizienten Generierung von komplexeren SQL-Statements, da dieses über das Konzept der DataSets nicht möglich wäre. Schwerpunkt der Umsetzung des Templatekonzepts bildete die Parametrisierung eines SQL-Statements. Dazu mußten die einzelnen Bestandteile eines SQL-Statements lokalisiert und in einem Bausteinkatalog abgelegt werden, um zur Laufzeit ein dynamisch an die jeweils selektierten Attribute angepaßtes Statement zu erzeugen. Ausgehend von der Modellierung eines Basis-SQL-Statements über die Spezifikation verschiedener Filtertypen (konstante Filter, Gruppenfilter) oder die Integration von Subqueries bis hin zu einem virtuellen SQL-Template, das lediglich kontextbezogen materialisiert wird, wurde durch die Umsetzung des Templatekonzepts ein sehr flexibler Mechanismus zur variablen Informationsbereitstellung für den Anwender entwickelt. Des weiteren wird hierdurch eine Reduzierung des Umfangs zu entwerfender bzw. später zu pflegender SQL-Statements realisiert.

Allerdings mußte in diesem Zusammenhang ein Modellierungstool entwickelt werden, durch das der Modellierer in der Lage ist, ein SQL-Template in geeigneter Form in das System einzugeben. Die Reportingplattform muß die Bausteine erkennen und dem entsprechenden SQL-Template syntaktisch korrekt zuordnen können. Die für diesen Bausteinkatalog notwendigen Tabellen wurden in das Repository eingepflegt.

Zusätzlich wurde ein Suchalgorithmus entwickelt, der den Modellierer bei der Ermittlung bereits modellierter Templates durch gewählte Attributkombinationen unterstützt. Die Suchparameter (Attribute) können dabei konjunktiv oder disjunktiv verknüpft werden, um den Suchraum einzuschränken.

Durch die vollständige Integration der SQL-Templates in das DataSet-Konzept erhält der Anwender eine äußerst flexible Möglichkeit zur Adaption der verschiedenen, auch komplexeren, Informationsabfragen, da bei neuen Anforderungen das System die notwendigen SQL-Bausteine kontextbezogen in das jeweilige SQL-Template einfügt.

3.6 PROZESSTEMPLATES

3.6.1 Übersicht

Analog des vorigen Kapitels zur Entwicklung des SQL-Konzepts wird nachfolgend die Umsetzung des Templatekonzepts für Prozesse an Beispielen aus dem Bereich Transportlogistik dargestellt.

Zielsetzung der Ausarbeitungen in diesem Kapitel ist die Entwicklung eines Werkzeugs, das den Anwender durch die einzelnen Arbeitsschritte der verschiedenen Entscheidungsprozesse führt und insbesondere die im vorhergehenden Abschnitt entwickelten Mechanismen zur Informationsbereitstellung integriert, um ihm die für Entscheidungen benötigten Informationen aus der Unternehmensdatenbank zu präsentieren oder externe Programme aufrufen zu lassen.

Die notwendige Funktionalität eines solchen Tools beinhaltet das in *Kapitel 2.4 Workflowmanagement* vorgestellte Workflowkonzept, denn die konkrete Aufgabe des zu entwickelnden Werkzeugs entspricht prinzipiell der einer auf seine elementaren Funktionen reduzierten Ausführungskomponente eines Vorgangssteuerungssystems. Der wesentlichste konzeptuelle Unterschied besteht darin, daß hier keine getrennten Server- und Clientkomponenten zum Einsatz kommen, sondern sich die Ablaufsteuerung und das Benutzerinterface auf dem lokalen Rechner befinden. Zusätzlich entfällt hier die i.a. der Serverkomponente zugewiesene Aufgabe der dynamischen Zuordnung von personellen und technischen Ressourcen zu Funktionen der Prozesse, da die Prozesse der Reportingplattform von einzelnen Mitarbeitern an einem Arbeitsplatz bearbeitet werden.

Folgende grundlegende Anforderungen werden an das zu entwickelnde Werkzeug gestellt.

- Zu jeder Funktion des Prozesses sollen automatisch die benötigten Informationen aus der Datenbank bereitgestellt sowie Aufrufe externer Programme integriert werden. Zunächst wird ausschließlich lesend auf die Informationen der Datenbank zugegriffen. Eine Weiterleitung von Daten zwischen den Funktionen der Prozesse ist sinnvoll, um z.B. Einschränkungen auf DataSets in Form von zusätzlichen Filtern mit zu übergeben und den Anwendern Aufwand zu ersparen.
- Der Anwender soll auf Ausnahmesituationen so reagieren können, daß er zur Laufzeit die Prozesse anpassen kann. Dazu können neue Subprozesse eingefügt, Prozesse herausgelöst oder Prozesse aneinandergefügt werden.
- Stehen zu einer Funktion keine Informationen aus der Datenbank zur Verfügung oder ist vom Prozeßbearbeiter eine manuelle Tätigkeit zu verrichten, so erhält er vom System einen Hinweis, welchen Arbeitsschritt er zur Fortsetzung des Prozesses an dieser Stelle vollziehen muß.
- Anhand der inhaltlichen Ausprägung der dargestellten Informationen soll mit Hilfe zu hinterlegender Entscheidungsregeln die Ablaufsteuerung innerhalb des Prozesses weitgehend automatisiert erfolgen.
- Wenn sich der Inhalt der Daten nicht systemgestützt auswerten läßt, muß der Anwender nach Aufforderung durch das System selbsttätig zwischen den alternativen Verzweigungsmöglichkeiten des Prozesses wählen können.
- Der Anwender soll Einflußmöglichkeiten auf den Fortgang der Prozeßbearbeitung in der Form erhalten, daß er manuell an bestimmte Stellen im Prozeßablauf springen, einen beliebigen Bearbeitungsstand eines Prozesses sichern und die Arbeit zu gegebener Zeit an dieser Stelle wieder fortsetzen kann.

Zusätzlich zum Prozeßwerkzeug ist analog zur Modellierungskomponente von Workflowmanagementsystemen ein Editor zu entwickeln, um die auszuführenden Prozeßmodelle im System abzubilden.

3.6.2 Das Konzept der Prozeßtemplates

Der Prozeß *Disposition* läßt sich stark vereinfacht anhand der in Abbildung 25 dargestellten *EPK* beschreiben: Bevor ein Disponent die Verplanung seines zu transportierenden Warenbestands durchführt, verschafft er sich zunächst einen Überblick hinsichtlich des Transportvolumens. Hieran schließt sich die Verplanung an, die sich meist über den ganzen Arbeitstag hinzieht. Ein großer Teil der Arbeitszeit wird jedoch für die Reaktion auf spezielle Problemstellungen verwendet, wie z.B. *Fahrer fällt aus, Zug fällt aus, Kapazitäten reichen nicht* etc. Die Abarbeitung dieser Exceptions erfolgt

parallel neben der eigentlichen Verplanung, die dann unterbrochen wird. In Abbildung 25 wird dies durch die Funktion *Exceptions* dargestellt.

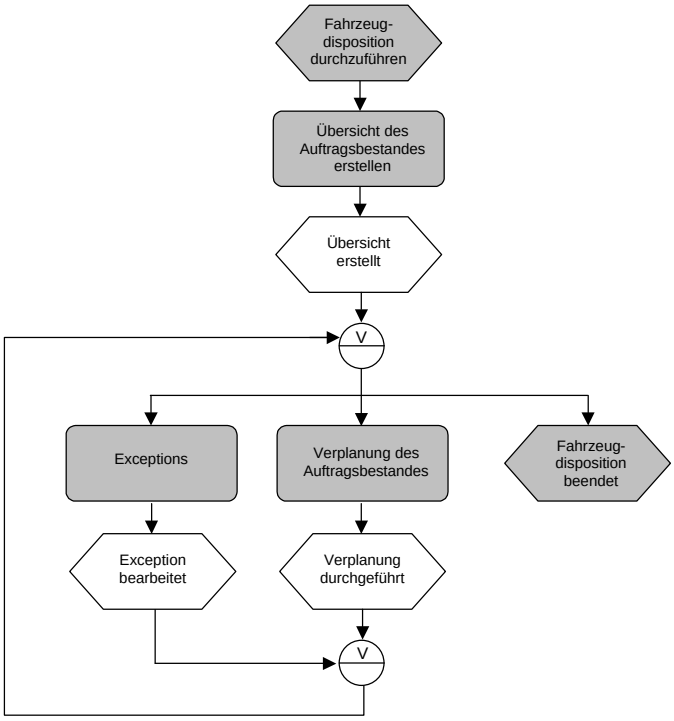


Abbildung 25: Hauptprozeß Disposition

Bevor die Spezifikation des Hauptprozesses *Disposition* durch Prozeßtemplates erfolgt soll zunächst eine Definition der Verhaltensaspekte für die Prozeßsteuerung gegeben werden. Für die folgenden Beispiele wird die Spezifikation der Ablaufbedingungen anhand der in *Kapitel 2.4.3 Das Referenzmodell MOBILE* dargestellten Terminologie beschrieben:

Serielle Ausführung:	→ (B_1, B_2)	Erst Prozeßbaustein B_1, dann B_2
Parallele Ausführung:	|| (B_1,B_2)	|| (B_1,B_2) ist erst dann beendet, wenn B_1 und B_2 beendet wurden. Bei sequentieller Ausführung ist jede Reihenfolge erlaubt.
Nichtdeterministische Ausführung:	[]	Der Prozeß ist beendet, sowie einer der Teilprozesse beendet ist.
Iterative Ausführung:	Ψ(B_1)	Ψ(B_1) == || (B_1, B_1, B_1, ...)
Bedingte Anweisung:	if (cond(), B_1; ~ cond(), B_2)	Hier entscheidet eine Ablaufbedingung *cond()*, welcher von alternativen Bausteinen [B_1] und [B_2] aufgerufen wird. Die Vorbedingung für diese Form der Ablaufkontrolle besteht darin, daß die *cond()* entscheidbar ist.
Optionale Anweisung:	{}	Der Prozeß kann ausgeführt oder auch übergangen werden.
Input:	?	
Output:	!	

Der Dispositionsprozeß wird mittels der folgenden zwei Prozeßtemplates analog der in *Kapitel 3.4 Templates* festgelegten allgemeinen Templatestruktur spezifiziert [Riec99].

Das folgende Template-1 spezifiziert den Prozeß Disposition.

Template-1	*Disposition*
DO	[Typ: Prozeß]
	[P_1:Prozeß_Bestandsübersicht_erstellen], [P_2:*Verplanung*],...,[P_i]
WITH	[B_1:Prozeß_Bestandsübersicht_erstellen],
	[B_2:*Template-2*],..., [B_i]
UNDER CONDITION	[C_1: →(B_1, Ψ(if{Beenden ?,Dispositionsende}[] B_2))],...,[C_i]

Template-1 umfaßt die Funktion für die Erstellung einer Übersicht sowie die nachfolgende Iteration eines Bausteins B_2, bis der Anwender den Prozeß beendet. Zur Spezifikation des Dispositionsprozesses wird mittels des Parameters P_2 das Template-2 gewählt, das als Baustein B_2 zu verwenden ist. Die zusätzlichen Platzhalter P_i, B_i und C_i sollen verdeutlichen, daß neben dem Parameter zur Integration von Template-2 jederzeit zusätzlich eingehende Parameter eine weitere Adaption auslösen können. So könnte durch den Benutzer z.B. per Drag & Drop an einem Prozeßknoten ein weiterer DataSet hinzugezogen werden. Das integrierte SQL-Template würde dann als ein zusätzlicher Baustein B_i eingefügt und die Bedingungen um die Sequenz-Condition Ψ (';' Template-y ';') erweitert werden. Das Template-2 zur Spezifikation der *Verplanung* mit möglichen Exceptions hat den folgenden Aufbau.

Template-2	*Verplanung*
DO	[Typ: Prozeß]
	[P_1:Prozeß_Verplanung_Auftragsbestand], [P_2:Variabel],...,[P_i]
WITH	[B_1:Prozeß_Verplanung_Auftragsbestand], [B_2:Variabel],...,[B_i]
UNDER CONDITION	[C_1:(B_1 [] B_2:Variabel)],...,[C_i]

In dem Beispielprozeß wird Baustein 2 in Template-1 durch Template-2 ersetzt, was die Reaktion auf Exceptions ermöglicht. Ein Parameter spezifiziert den Prozeßbaustein, der dann zur Behandlung der Exception eingefügt wird. Hierdurch wird die Lösung der Problemstellungen mittels der Spezialisierung des Hauptprozesses Disposition durchgeführt.

Das so aus Template-1 und Template-2 neu zusammengesetzte Prozeßtemplate hat den folgenden Aufbau.

Template-1 + Template-2	*Disposition + Verplanung*
DO	[Typ: Prozeß]
	[P_1:Prozeß_Bestandsübersicht_erstellen], [P_2:Prozeß_Verplanung_Auftragsbestand], [P_3:Variabel],...,[P_i]
WITH	[B_1:Prozeß_Bestandsübersicht_erstellen], [B_2:Prozeß_Verplanung_Auftragsbestand], [B_3:Variabel],...,[B_i]
UNDER CONDITION	[C_1: →(B_1 , Ψ(if{Beenden ?,Dispositionsende} [] (B_2 [] B_3:Variabel)))], ...,[C_i]

In Abbildung 26 wird eine Teilmenge möglicher Exceptions dargestellt, deren Prozeßbausteine je nach auftretendem Ereignis einzufügen sind.

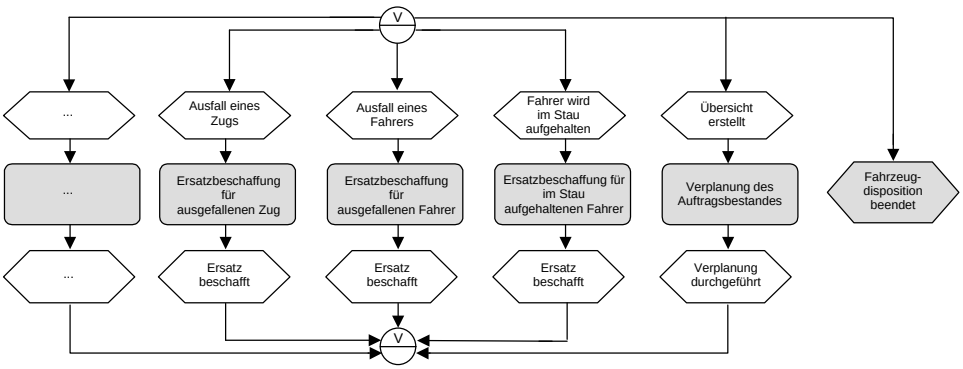


Abbildung 26: Mögliche Exceptions im Prozeß Disposition

In diesem Beispiel wird nun der Ausfall eines Zugs angenommen, so daß ein komplexerer Prozeß zur Lösung des Problems zu durchlaufen ist. Der Prozeß wird der Übersichtlichkeit halber nachfolgend als ein Prozeßbaustein eingefügt, auch wenn er aufgrund seiner Komplexität wieder aus mehreren Templates besteht. Der Mechanismus zum Kombinieren der verschiedenen Templates / Bausteine ist generisch, so daß sich beliebig tiefe Verschachtelungen darstellen lassen. Abbildung 27 stellt den Prozeß für die Exception: Ausfall eines Zugs dar.

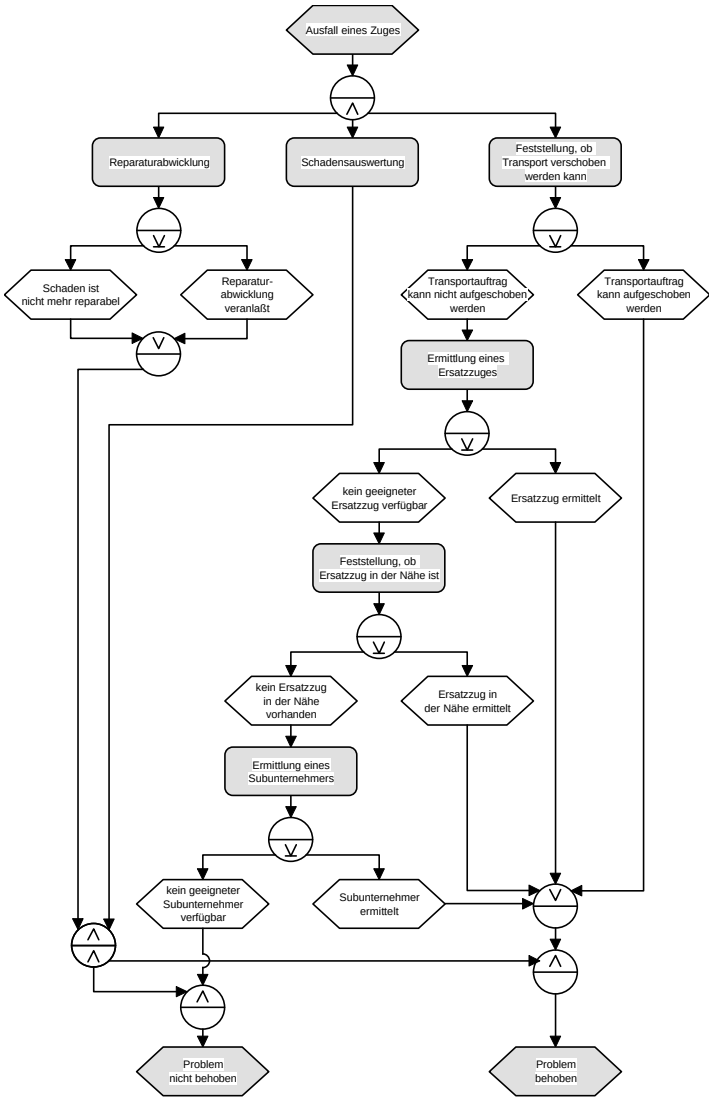


Abbildung 27: Prozeßbaustein: Ausfall eines Zugs

Die Spezifikation des ablauffähigen Entscheidungsprozesses sieht wie folgt aus:

Template-1 + Template-2 + Zugausfall	*Disposition + Verplanung + Ausfall eines Zugs*
DO	[Typ:Prozeß] [P_1:Prozeß_Bestandübersicht_erstellen], [P_2:Prozeß_Verplanung_Auftragsbestand], [P_3:Ausfall_eines_Zugs],...,[P_i]
WITH	[B_1:Prozeß_Bestandübersicht_erstellen], [B_2:Prozeß_Verplanung_Auftragsbestand], [B_3:Ausfall_eines_Zugs], ...,[B_i]
UNDER CONDITION	[C1: → (B1 , Ψ(if{Beenden ? , Dispositionsende} [] B2 [] B3))], ...,[Ci]

3.6.3 Verknüpfen von Prozeßkomponenten

Nachfolgend wird ein schematischer Überblick zu den Verknüpfungsmechanismen der Komponenten gegeben [Riec99]. Abbildung 28 verdeutlicht die hohe Komplexität und Dynamik im Tagesgeschäft einer Disposition. Der dort dargestellte Geschäftsprozeß ist nicht vorhersehbar, sondern ergibt sich aufgrund des jeweiligen Kontexts und wird durch den Anwender aus den verschiedenen Komponenten in Form von Teilprozessen zusammengefügt.

Je nach anfallenden Aufgabenstellungen übernimmt der Anwender Komponenten aus dem Gesamtangebot (rechter Kasten), paßt sie an seine Bedürfnisse an und positioniert sie aufgrund ihrer Priorität innerhalb des Gesamtprozesses.

Prozeßtemplates lassen folgende Adaptionen zu:
- Sequentielles Aneinanderfügen von Prozessen.
- Einfügen von Subprozessen.
- Löschen von Subprozessen.
- Einfügen von Iterationen.
- Aufnahme zusätzlicher Programme oder DataSets an einzelnen Prozeßknoten.
- Löschen von DataSets / Programmen an einzelnen Prozeßknoten.

Da es sich um jeweils eigenständig ablauffähige Komponenten handelt, ist der Anwender nicht gezwungen, zunächst einen Prozeßbaustein vollständig zu durchlaufen, um anschließend bei dem nächsten, mittels UND-Verknüpfung auf der gleichen Prozeßhierarchieebene liegenden, Teilprozeß anzuschließen (Serialisierung). Vielmehr kann er einzelne Prozeßstände "einfrieren" und zu anderen, "wieder aufzutauenden" oder neuen Prozessen wechseln.

Hierdurch werden kurze Reaktionszeiten für die Berücksichtigung von Exceptions gewährleistet, so daß bei Änderungen der jeweiligen Prozeßprioritäten sofort zum nächsten Prozeß gewechselt bzw. eine neue Komponente in den Gesamtablauf übernommen werden kann. Diese Flexibilität läßt sich nur durch die verwendeten Konzepte aus dem Bereich Componentware erzielen, da keine Gesamtprozeßinstanz initiiert wird, sondern die Komponenten erst bei Aufruf an das Laufzeitsystem übergeben werden und jederzeit zu anderen Komponenten gewechselt werden kann, sofern keine Integritätsbedingungen verletzt werden.

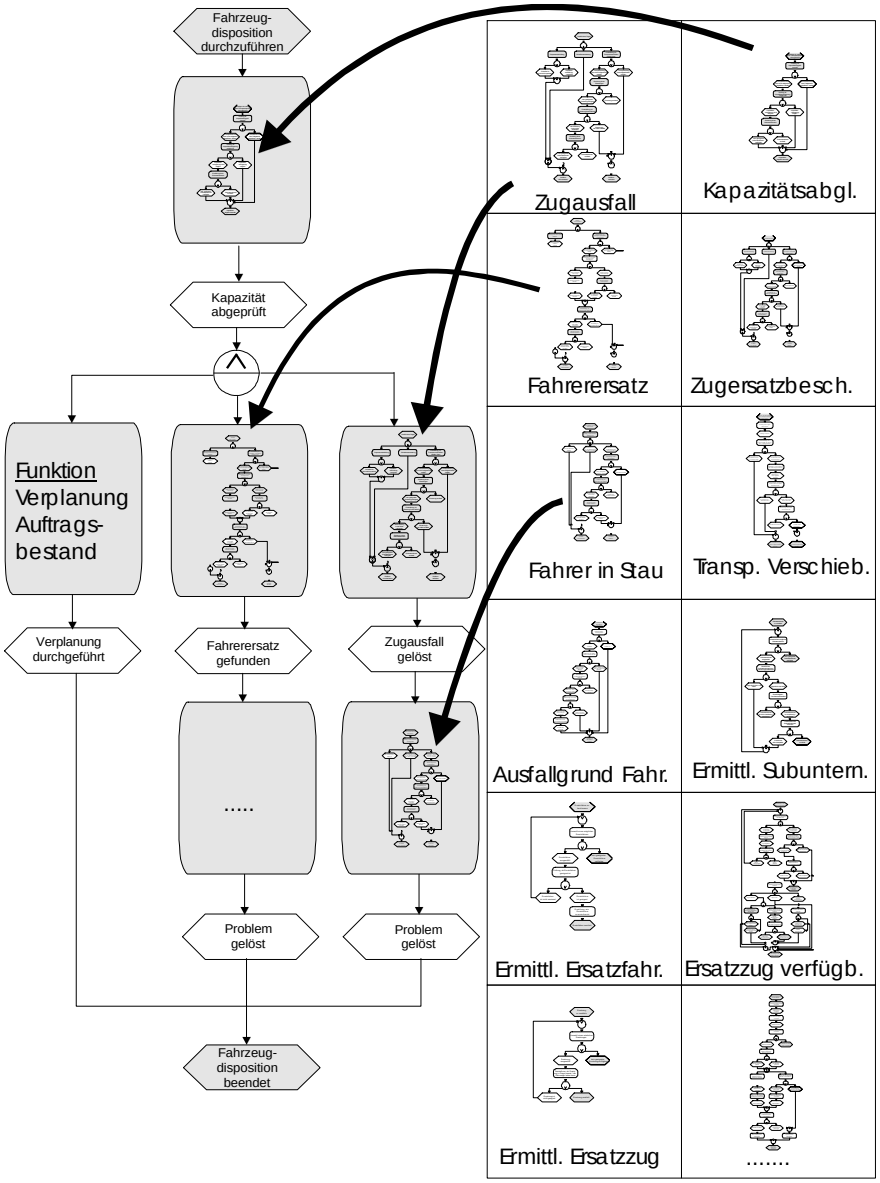


Abbildung 28: Aus unterschiedlichen Prozeßkomponenten gebildeter Geschäftsprozeß in der Disposition[2]

Die Verknüpfung der einzelnen Komponenten ist für den Anwender problemlos am Bildschirm per Drag & Drop auszuführen. Die interne Umsetzung ist für ihn transparent. Sollte keine eindeutige Verknüpfung möglich sein, da ggf. kein durchgängiger Datenfluß gewährleistet werden kann, so wird der Anwender über Dialogboxen in die Entscheidung mit einbezogen.

Abbildung 29 und Abbildung 30 zeigen beispielhaft eine Konfliktsituation, die zur Laufzeit aufgelöst werden muß. Die Aussparungen am unteren Ende sollen die Möglichkeit des *Andockens* anderer Komponenten illustrieren. Mit Andocken bezeichnet man die Fähigkeit der Komponenten über Standardschnittstellen die anwendungsspezifischen Schnittstellen anderer Komponenten zu ermitteln, um dann miteinander zu kooperieren [Jenz98, S.242].

Als Ausgangsschnittstelle bietet die Prozeßkomponente-1 drei, in Form von Ellipsen dargestellte, DataSets *Fahrer* oder (*Tour* und *Zug*) an, die als Eingangsparameter für einen weiteren Prozeßbaustein dienen können. Als Beispiel für eine anzudockende Komponente soll der in Abbildung 30 dargestellte Prozeßbaustein dienen. Dabei müssen die eingehenden Parameter nicht zwingend 1:1 mit den ausgehenden Parametern der anzudockenden Komponente übereinstimmen.

[2] Auf die Darstellung der einzufügenden ODER-Bedingungen, die bei nicht gelungener Problemlösung einen anderen Prozeßverlauf initiieren würden, wurde zur Vereinfachung verzichtet.

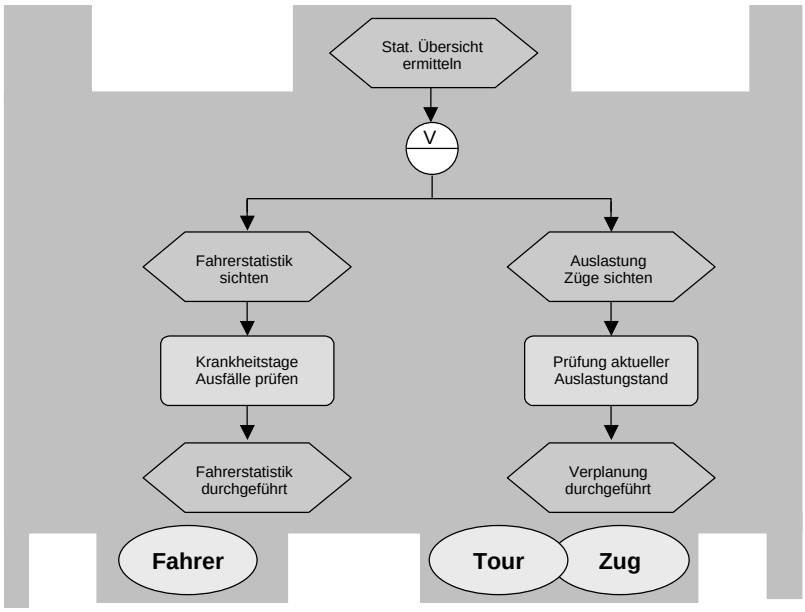


Abbildung 29: Prozeßkomponente-1

Damit die Verknüpfung verschiedener Prozeßbausteine immer zu einem konsistenten Gesamtprozeß führt, sind verschiedene generische Integritätsbedingungen zu überprüfen. So ist nicht für jeden Übergang zwischen den verschiedenen Komponenten eine Integritätsbedingung zu definieren. Die Prüfung erfolgt mittels der einmal spezifizierten Bedingungen und jeweils zu verwendenden Metadaten.

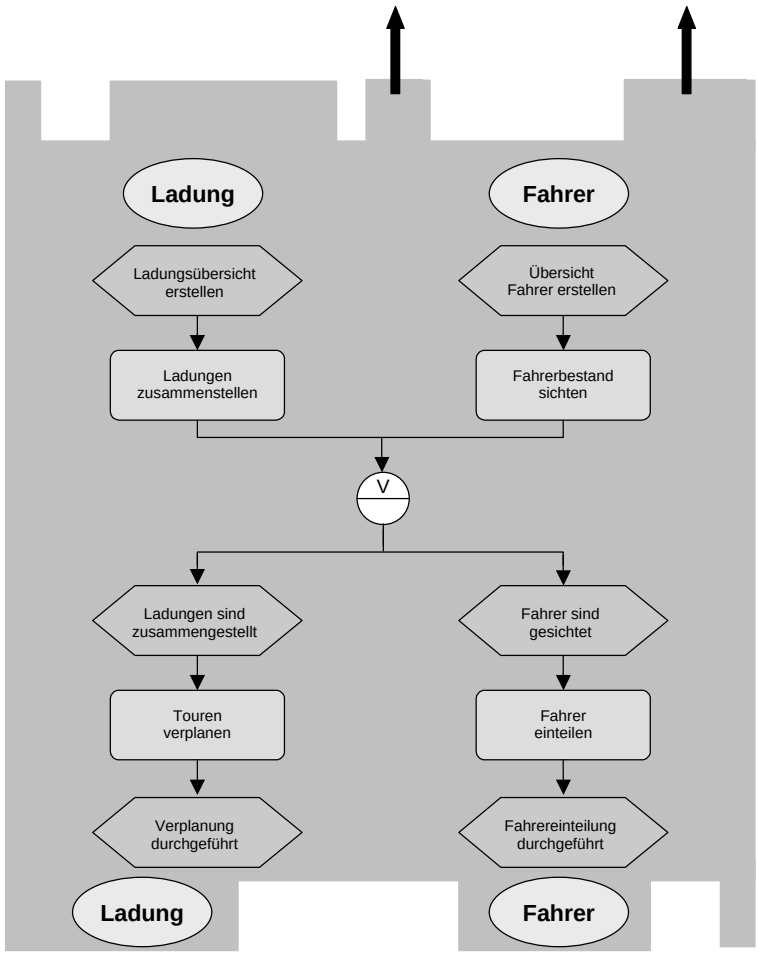


Abbildung 30: Prozeßkomponente-2

Eine *optimale Kopplung* wird durch die Einhaltung sämtlicher nachfolgender Bedingungen erzielt.

1. Die ausgehenden Prozeßpfade der ersten Komponente "passen" genau zu den eingehenden Prozeßpfaden der nächsten Komponente.
2. Die Ausgangsbedingung des ersten Prozesses bzgl. der Funktionsdurchführung (UND / ODER / XOR) stimmt mit der Eingangsbedingung des zweiten Prozesses überein.
3. Der Datenfluß ist in den Komponenten modelliert und stimmt in beiden Prozeßkomponenten überein, so daß eine Datenübergabe zwischen den beiden Prozessen erfolgen kann.
4. Die Prozesse gehören zu einer Domain.

In diesem Fall ist es sinnvoll, die Möglichkeit der Verbindung beider Komponenten zuzulassen, auch wenn keine eindeutige Übereinstimmung der Schnittstellen besteht. Sofern der Anwender im Rahmen des Prozeßdurchlaufs in Prozeßkomponente-1 (Abbildung 29) den Pfad über die Fahrerstatistik genommen hat, besteht eine partielle Übereinstimmung zu der Schnittstelle von Prozeßkomponente-2, die dem Anwender ggf. ausreicht. Vielleicht hat er in dieser Situation kein Interesse an einer Verfolgung der Ladung, sondern möchte über Abfragen zur Fahrereinteilung etc. Informationen zu bestimmten Fahrern weiter untersuchen? Im vorigen Prozeß gesetzte Filter würden so direkt an den DataSet im folgenden Prozeßbaustein mit übergeben werden, um so eine durchgängige Bearbeitung zu gewährleisten.

Wenn der Anwender zur Laufzeit in Prozeßkomponente-1 Abfragen bzgl. des Auslastungsgrads durchgeführt hat, besteht keine Übereinstimmung mehr zur nachfolgenden Schnittstelle. Dies sollte jedoch nicht zu einer Fehlermeldung führen (Robustheit), sondern über eine Dialogbox zusammen mit dem Anwender gelöst werden. Dazu sind mehrere Alternativen denkbar:

- Bei partieller Übereinstimmung der nachfolgenden Knoten ist eine mögliche Iteration über den 1. Pfad in Prozeßkomponente-1 durchzuführen, um den notwendigen Datenfluß des "passenden" Prozeßpfads zu realisieren.
- Keine Übergabe von Daten notwendig bzw. möglich; keine weitere Berücksichtigung der vorigen Daten erwünscht.
- Durchlauf der Komponente 2 als eigenständiger Prozeß, so daß vorhergehende Daten für neuen Kontext nicht relevant sind.

Die relativ hohe Freiheit bei der Kooperation verschiedener Prozeßkomponenten versetzt den Anwender in die Lage, seine Lösungsstrategien individuell zusammenzustellen. Da nicht jede Verknüpfung antizipiert werden kann, sind hierdurch vielfältige Kombinationsmöglichkeiten gegeben. Eine mögliche Kopplung der beiden Prozesse wird nochmals in Abbildung 31 schematisch dargestellt.

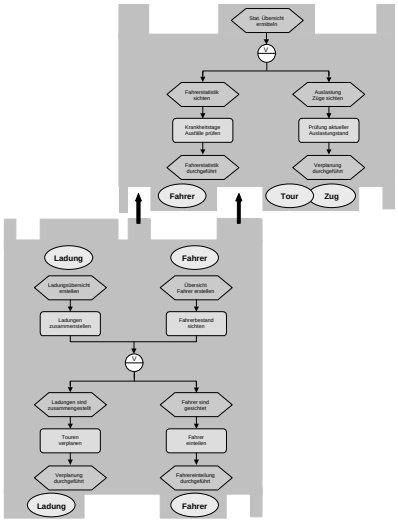


Abbildung 31: Kopplung zweier Prozeßkomponenten

Wenn für die einzelnen Prozeßkomponenten die Metapher von (Prozeß-)Bausteinen herangezogen wird, so läßt sich die Kopplung der einzelnen Bausteine schematisch wie in Abbildung 32 darstellen (Prozesse *Kapazitätsabgleich* und *Zugausfall* aus Abbildung 28). Die Tabellen an den Aussparungen stellen in diesem Fall eine Erweiterung der o.g. DataSet-Übergabemöglichkeit dar. In einer späteren Ausbaustufe der Reportingplattform wird die Schnittstelle auch für andere Datenobjekte erweitert.

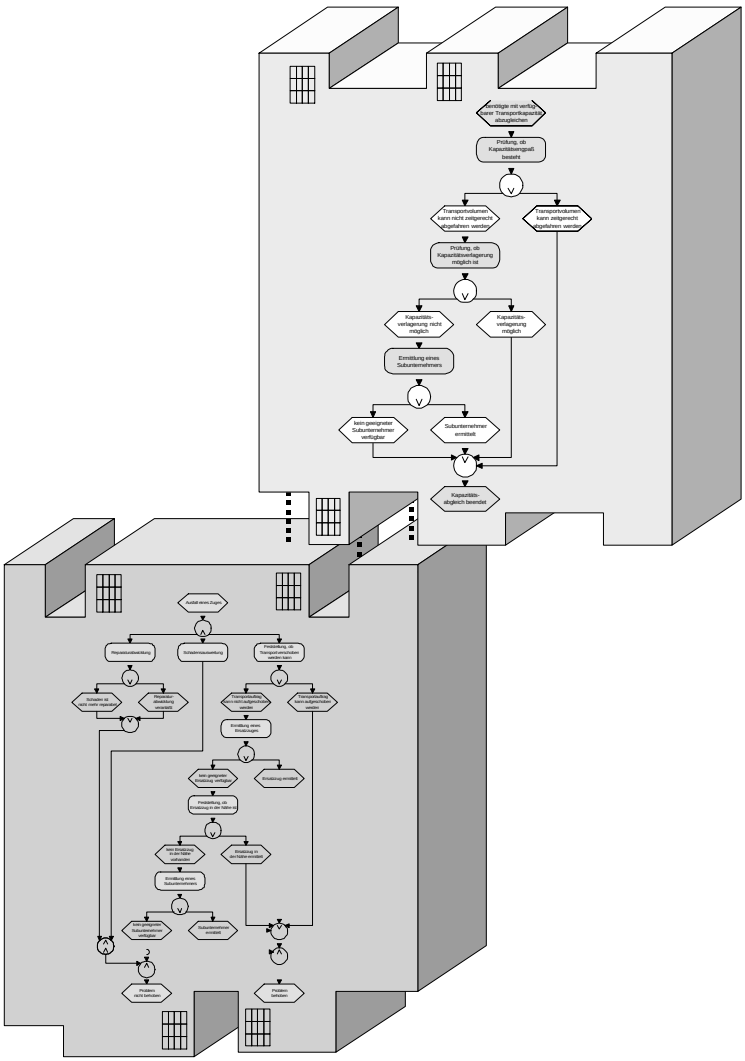


Abbildung 32: Schematische Darstellung der Kopplung zweier Prozeßkomponenten aus Abbildung 28

Wenn man die einzelnen Komponenten in Form der in Abbildung 32 gezeigten Bausteine sieht, so würde das gesamte Informationsangebot der modellierten und neu verknüpften Komponenten einem Baukasten gleichen, aus dem der Anwender sich je nach Bedarf passende Bausteine heraussucht, miteinander verknüpft und nach der Nutzung wieder abspeichert, um sie bei späteren Problemstellungen erneut zu verwenden. In Abbildung 33 wird dieses anhand der beiden o. g. Bausteine dargestellt, die nach der Verwendung als ein neuer Baustein abgelegt werden und so zukünftig auch als eine Einheit zur Verfügung stehen.

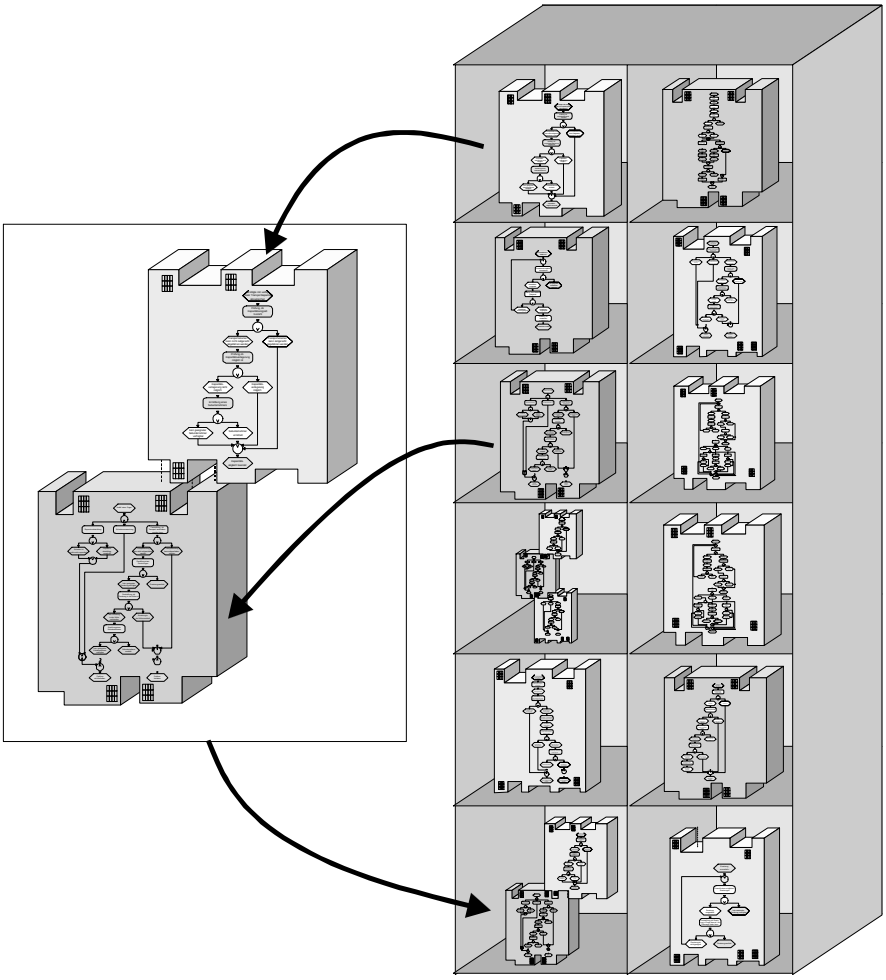


Abbildung 33: Komponentenbaukasten

Die in diesem Kapitel beschriebenen Mechanismen zur Kopplung von Prozeßbausteinen werden nachfolgend im Einzelnen spezifiziert.

3.6.4 Abbildung der Prozesse in der Reportingplattform

3.6.4.1 Herleitung des Datenmodells

Die essentielle Prozeßstruktur ist die eines Geschäftsprozesses / Arbeitsvorgangs. Zu dessen Beschreibung dienten in *Kapitel 2.4 Workflowmanagement* die ereignisgesteuerten Prozeßketten (EPK), die nachfolgend direkt in eine äquivalente Datenstruktur umgesetzt werden sollen, damit sich die erstellten Prozeßmodelle ohne großen Änderungsaufwand unmittelbar übernehmen lassen.

Eine EPK, wie sie Abbildung 34 beispielhaft zeigt, stellt einen gerichteten Graphen dar. Die Besonderheit besteht hier zum einen darin, daß es drei verschiedene Arten von Knoten gibt, *Ereignisse*, *Funktionen* sowie *Verknüpfungen* von Ereignissen und Funktionen in Form logischer Operatoren. Zum anderen sind den Funktionen die darzustellenden Informationen aus der Datenbank zugeordnet, aber auch die anderen Knotentypen werden im weiteren um zusätzliche Beschreibungen zu ergänzen sein, so daß sich die vorliegende Datenstruktur noch präziser als gerichteter Graph mit markierten Knoten beschreiben läßt.

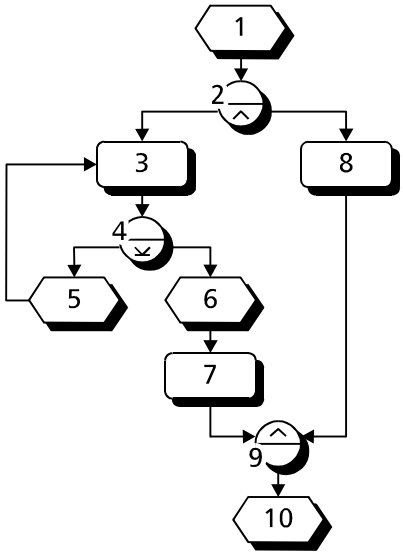


Abbildung 34: Beispielhafte Struktur einer ereignisgesteuerten Prozeßkette

Der Aufbau eines solchen Graphen läßt sich durch eine Menge von Knoten spezifizieren, die zu ihrer Identifikation eine eindeutige Nummer erhalten sollen und zu jedem Knoten eine Menge von Nachfolgeknoten abbilden. Der Beispielprozeß stellt sich dann wie folgt dar.

Prozeß= { 1 → {2}
, 2 → {3, 8}
, 3 → {4}
, 4 → {5, 6}
, 5 → {3}
, 6 → {7}
, 7 → {9}
, 8 → {9}
, 9 → {10}
, 10 → {} }

Nachstehend wird nun diese Grundstruktur eines Graphen entsprechend den speziellen Erfordernissen einer EPK weiter ausgebaut und verfeinert (siehe [Pete96]).
Da in die zukünftige Anwendung eine größere Anzahl verschiedener Prozesse integriert werden soll, benötigen diese eine eindeutige Identifikationsnummer sowie auf Seiten der Benutzerschnittstelle eine textliche Bezeichnung und erläuternde Beschreibung der auswählbaren Prozesse.

Prozesse = *ProzeßNr* $\xrightarrow{m}$ *(Prozeß* × *Prozeßbezeichnung* × *Prozeßbeschreibung)*
ProzeßNr = N_1
Prozeßbezeichnung = *Char**
Prozeßbeschreibung = *Char**

Die einzelnen Prozesse benötigen neben der Darstellung ihrer Struktur in Form von Nachfolgebeziehungen zwischen Knoten auch eine je nach den drei verschiedenen Typen unterschiedliche Beschreibung der einzelnen Knoten.

Prozeß = *KnotenNr* $\xrightarrow{m}$ *(Nachfolgeknoten* × *Knotenbeschreibung)*
KnotenNr = N_1
Nachfolgeknoten = *KnotenNr-set*
Knotenbeschreibung = *Funktionsknoten* | *Ereignisknoten* | *Operatorknoten*

Die Ausführung einer Funktion soll so erfolgen, daß dem Anwender die zur Ausführung der Tätigkeit notwendigen Informationen aus der Datenbank bereitgestellt werden. Es wird daher außer dem Namen und einem erklärenden Hilfstext für den Anwender insbesondere die Beschreibung des DataSets

benötigt, aus welchem das SQL-Template zur Informationsgewinnung generiert werden soll, sowie der Name des Tools, mit dem die Ergebnisdatenmenge darzustellen ist.

Funktionsknoten = *Funktionsbezeichnung* × *Funktionsbeschreibung* × *DataSet* × *Werkzeug*
Funktionsbezeichnung = *Char**
Funktionsbeschreibung = *Char**
Werkzeug = *Char**

Als weiterer Knotentyp existieren Ereignisse, die hier als Bedingungen zur Steuerung des Prozeßablaufs verstanden werden sollen. Wird eine Funktion von nur einem Ereignis gefolgt, so weist dieses Ereignis lediglich auf die Beendigung der Funktion hin, treten aber mehrere Folgeereignisse in Verbindung mit einem logischen Operator auf, so stellen diese Ereignisse Regeln für die Ablaufsteuerung des Prozesses dar. Welches Ereignis eingetreten ist und welches nicht, hängt dann von der inhaltlichen Ausprägung der in der vorangegangenen Funktion eingelesenen Daten ab. Dementsprechend ist für ein Ereignis eine Bedingung zu hinterlegen, wann es als eingetreten gilt. Wenn eine solche automatische Auswertung nicht möglich ist, muß der Anwender anhand von Beschreibungstexten der alternativen Ereignisse selbst entscheiden, mit welchem dieser Ereignisse der Prozeß fortzusetzen ist.

Ereignisknoten = *Eintrittsbedingung*
Eintrittsbedingung = *AutomatischeBedingung* | *ManuelleBedingung*
AutomatischeBedingung = *token*
ManuelleBedingung = *Ereignisbeschreibung*
Ereignisbeschreibung = *Char**

Logische Operatoren verknüpfen die Funktionen und Ereignisse miteinander. Diesem Typ von Knoten können mehrere Eingangs- wie auch Ausgangsknoten zugeordnet sein, so daß er durch die gegebenenfalls notwendigen Eingangs- und Ausgangsoperatoren gekennzeichnet ist.

Operatorknoten = *(Eingangsoperator)* × *(Ausgangsoperator)*
Eingangsoperator = *AND* | *OR*
Ausgangsoperator = *AND* | *XOR*

Nachdem das Datenmodell soweit spezifiziert ist, sind im folgenden eine Reihe von Konsistenzbedingungen festzulegen, um zu gewährleisten, daß sich die Struktur eines Prozeßgraphen in einer syntaktisch korrekten Form befindet sowie den speziellen Anforderungen der verarbeitenden Module entspricht. Die Überwachung dieser Bedingungen wird zunächst bei der Modellierung der Prozesse im ebenfalls zu entwickelnden *Prozeßeditor* durchgeführt. Des weiteren erfolgt die Kontrolle bzgl. der Einhaltung dieser Konsistenzbedingungen zur Laufzeit nach einer anwenderseitig durchgeführten Adaption eines Prozesses (siehe *Kapitel 3.6.3 Verknüpfen von Prozeßkomponenten*).
Zwei Hilfsfunktionen kommen bei der Überprüfung der Konsistenzbedingungen zum Einsatz, die der Feststellung dienen, ob einem oder einer Kombination von Operatorknoten ein Funktions- oder aber ein Ereignisknotentyp vorangeht bzw. nachfolgt.

type vorKnotenTyp: KnotenNr x Prozeß ⟶ Knotenbeschreibung
type nachKnotenTyp: KnotenNr x Prozeß ⟶ Knotenbeschreibung
vorKnotenTyp(node, proc) ≙
  if Øis-Operatorknoten(proc(node).Knotenbeschreibung)
    then proc(node).Knotenbeschreibung
    else let prev Î {n|n Î dom proc · node Î rng proc(n).Nachfolgeknoten} in vorKnotenTyp(prev, proc)
nachKnotenTyp(node, proc) ≙
  if Øis-Operatorknoten(proc(node).Knotenbeschreibung)
    then proc(node).Knotenbeschreibung
    else let next Î proc(node).Nachfolgeknoten in nachKnotenTyp(next, proc)

Ein Knoten sollte nicht nur innerhalb seines eigenen Prozesses, sondern über alle Prozesse hinweg eindeutig identifizierbar sein, damit nicht bei jeder Bezugnahme auf einen Knoten die Prozeßnummer mitgeführt werden muß. Zusätzlich lassen sich so hierarchische Prozesse leichter abbilden, da sich eine

Funktion relativ einfach durch einen detaillierteren Unterprozeß ersetzen läßt, indem die Nummern ihrer Vorgänger- und Nachfolgerereignisse durch die Nummern der Ein- und Ausgangsereignisse des Unterprozesses ersetzt werden.

inv-Prozesse(procs) ≙
card union{dom proc|(proc, procname, procdescr) Î rng procs}
=sum{card dom proc|(proc, procname, procdescr) Î rng procs} Ù

Eine grundsätzliche Bedingung für jeden Graphen besteht darin, daß in die Menge der Nachfolgeknoten nur tatsächlich vorhandene Knoten aufgenommen werden.

"(proc, procname, procdescr) Î rng procs ·
(union{next|(next, descr) Î rng proc} Í dom proc) Ù

Eine EPK besitzt genau einen Startknoten und mindestens einen Endknoten, die alle vom Typ Ereignis sein müssen. Sie zeichnen sich dadurch aus, daß sie keinen Vorgänger- bzw. keine Nachfolgeknoten besitzen. Für alle übrigen Knoten wird gefordert, daß sie vollständig in den Prozeßablauf eingebunden sind, also mindestens ein Vorgänger und ein Nachfolger existiert.

(let start Í dom proc be such that ("s Î start · Ø($n Î dom proc · s Î proc(n).Nachfolgeknoten)) in
let end Í dom proc be such that ("e Î end · proc(e).Nachfolgeknoten = {}) in
(card start = 1) Ù (card end ³ 1) Ù (start Ç end = {}) Ù
("n Î (start È end) · is-Ereignisknoten(proc(n).Knotenbeschreibung)) Ù
("n Î (dom proc - start - end) · (proc(n).Nachfolgeknoten¹{} Ù n Î union{next|(next, descr) Î rng proc}))) Ù

Auf eine Funktion hat genau ein Knoten zu folgen, der entweder vom Typ Ereignis oder Operator, gefolgt von ein oder mehr Ereignissen, sein muß. Analog dazu darf auf ein Ereignis nur eine einzelne Funktion oder ein Operator mit anschließender Funktion folgen.

"node Î dom proc · let(next, descr)=proc(node) in
(is-Funktionsknoten(descr) Þ
(card(next)=1 Ù let n Î next in is-Ereignisknoten(nachKnotenTyp(n, proc)))) Ù
(is-Ereignisknoten(descr) Þ
(card(next)£1 Ù "n Î next is-Funktionsknoten(nachKnotenTyp(n, proc)))) Ù

Ein Operatorknoten ist nur dann sinnvoll eingesetzt, wenn er mehrere Ein- oder Ausgangsknoten, also einen Ein- oder einen Ausgangsoperator oder aber beides besitzt.

(is-Operatorknoten(descr) Þ
((descr.Eingangsoperator¹NIL Ú descr.Ausgangsoperator¹NIL) Ù

Verfügt er über keinen Ein- oder Ausgangsoperator, darf er auch nur einen einzelnen Ein- bzw. Ausgangsknoten haben, anderenfalls muß er mindestens zwei Ein- bzw. Ausgangsknoten besitzen.

(let countprev=card{n|n Î dom proc · node Î proc(n).Nachfolgeknoten} in
if descr.Eingangsoperator=NIL then countprev=1 else countprev>1) Ù
(let countnext=card next in
if descr.Ausgangsoperator=NIL then countnext=1 else countnext>1) Ù

Bei einem Operatorknoten müssen die direkten (oder bei einem weiteren Operator als Nachbarknoten die indirekten) Vorgänger und Nachfolger jeweils einheitlich vom Typ Funktion oder Ereignis sein.

("n_1, n_2 Î {n|n Î dom proc · node Î proc(n).Nachfolgeknoten} · vorKnotenTyp(n_1, proc)
=vorKnotenTyp(n_2,proc)) Ù
("n_1, n_2 Î next · nachKnotenTyp(n_1, proc)=nachKnotenTyp(n_2, proc)) Ù

Die folgenden Einschränkungen auf zulässige Typkombinationen von Operatoren und ihren Nachbarknoten stellen keine grundsätzlichen Forderungen der EPK-Syntax dar, sondern werden hier explizit

formuliert, um die notwendige Verarbeitungslogik für die Prozeßgraphen in ihrer Komplexität zu reduzieren.

So sollen Verzweigungen zwischen zwei oder mehr Funktionen, sei es als Ein- oder als Ausgang eines Operatorknotens, stets nur UND-verknüpft definiert werden dürfen, was also einer Parallelverarbeitung der Funktionen entspricht. Für den Fall einer ODER-Verzweigung existieren keine Entscheidungsregeln für den zu wählenden Weg, weil diese nur in den Ereignisknoten hinterlegt sind.

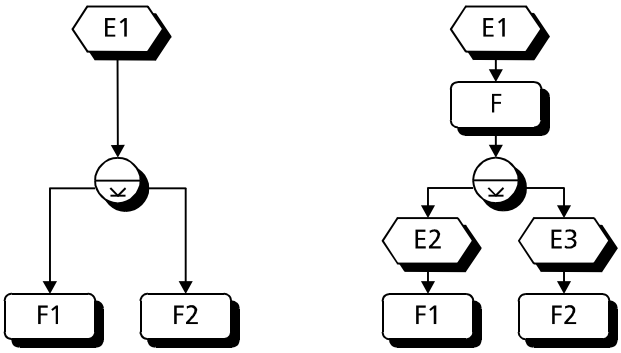


Abbildung 35: Formulierung alternativer Funktionsaufrufe in einer EPK

Eine alternative Verarbeitung von Funktionen läßt sich daher, wie in Abbildung 35 veranschaulicht, in der Weise formulieren, daß den betreffenden Funktionen eine zusätzliche Entscheidungsfunktion vorgeschaltet wird (Vgl. [Sche94, S.49 ff.].

(let prev={n|n Î dom proc · node Î proc(n).Nachfolgeknoten} in
(card prev>1 Ù "n Î prev · is-Funktionsknoten(vorKnotenTyp(n, proc)) Þ
descr.Eingangsoperator=AND) Ù

(card next>1 Ù "n Î next · is-Funktionsknoten(nachKnotenTyp(n, proc)) Þ
descr.Ausgangsoperator=AND) Ù

Handelt es sich bei den Nachfolgern eines Operators um mindestens zwei Ereignisse, dann soll nur eine XOR-Verzweigung zulässig sein, da eine eindeutige Bedingung für das eintretende Ereignis und damit den weiteren Prozeßablauf notwendig ist.

(card next>1 Ù "n Î next · is-Ereignisknoten(nachKnotenTyp(n, proc)) Þ
descr.Ausgangsoperator=XOR) Ù

Dementsprechend dürfen Ereignisse als Ausgang eines Operators auch nicht mit einem weiteren Operator kombiniert werden, da sonst eventuell eine mehrstufige Suche nach dem nächsten Ereignis, das einer Reihe von Operatoren folgt, notwendig wäre.

($n Î next · is-Ereignisknoten(proc(n).Knotenbeschreibung)) Þ
Ø($n Î next · is-Operatorknoten(proc(n).Knotenbeschreibung))) Ù

Schließlich sollen mehrere Funktionen als Eingang eines Operators nicht direkt mit alternativen Ereignissen als Ausgangsknoten verknüpft werden, weil sich in einem solchen Fall die inhaltliche Auswertung dargestellter Informationen aus der Datenbank über mehrere Funktionen gleichzeitig erstrecken müßte.

(let prev={n|n Î dom proc · node Î proc(n).Nachfolgeknoten} in
(card prev>1 Ù card next>1 Ù "n Î prev · is-Funktionsknoten(vorKnotenTyp(n, proc))) Þ
"n Î next · Øis-Ereignisknoten(nachKnotenTyp(n, proc))))

3.6.4.2 Verarbeitung einer Prozeßstruktur

Nachdem das Datenmodell einer EPK entwickelt worden ist, werden nun die elementaren Algorithmen zur Verarbeitung dieser Struktur spezifiziert. Dazu muß vorab ein zusätzlicher Datentyp eingeführt werden, der im Gegensatz zu den obigen nicht den statischen Aufbau eines Prozesses abbildet, sondern den aktuellen Bearbeitungsstatus einer Instanz dieses Prozesses beschreibt. Jeder bereits durchlaufene Knoten wird dazu in einer Statustabelle eingetragen, bei Ereignisknoten wird zudem vermerkt, ob das Ereignis als eingetreten gilt oder nicht. Außerdem werden alle Knoten in der Reihenfolge ihres Aufrufs aufsteigend durchnumeriert, um an gegebener Stelle die Aufrufhierarchie möglichst einfach nachvollziehen zu können.

Einige Hilfsfunktionen unterstützen die eigentliche Hauptfunktion zur Prozeßausführung: die Bestimmung des Startknotens eines Prozesses, die Ermittlung des Typs, die Menge der Vorgänger und der Nachfolger eines Knotens sowie der Überprüfung, ob ein Knoten überhaupt noch Nachfolger besitzt.

Wenn ein Prozeßgraph zyklische Strukturen enthält, also von einem unteren wieder zu einem höher angesiedelten Knoten verzweigt wird, dann muß der Bearbeitungsstatus der erneut zu durchlaufenden Knoten zurückgesetzt werden. Es ist dann folglich der Status all derjenigen Knoten zu löschen, die in der Aufrufreihenfolge hinter dem aktuellen Knoten liegen.

Eine Voraussetzung für das einwandfreie Funktionieren dieses Vorgehens besteht allerdings darin, daß nicht von außerhalb einer Struktur aus parallelläufigen Teilprozessen inmitten einen solchen Teilprozeß verzweigt wird, da ansonsten möglicherweise auch der Status der übrigen Teilprozesse gelöscht, sie jedoch nicht erneut ausgeführt werden, um ihren Status neu zu setzen.

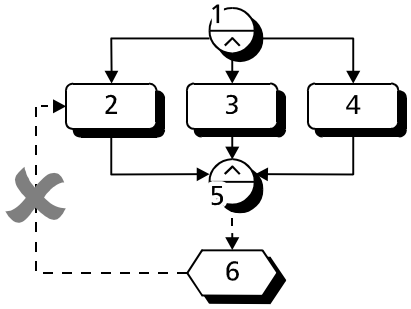


Abbildung 36: Unzulässige Verzweigung in einen parallelen Teilprozeß

Sofern in Abbildung 36 die angegebenen Knoten- mit den Aufrufnummern identisch sind, ist beispielsweise ein Rücksprung von Knoten 6 zu Knoten 2 unzulässig, weil der Bearbeitungsstatus für alle Knoten mit einer höheren Aufrufnummer zurückgesetzt wird. Es verlieren also auch die Knoten 3 und 4 ihren Status, womit die Bedingung in Knoten 5 nicht ausgewertet werden kann. Eine korrekte Verzweigung wäre daher von Knoten 6 zu Knoten 1, da nun neben der Funktion 2 auch die Funktionen 3 und 4 erneut durchlaufen werden.

Bei einer Verzweigungsmöglichkeit zwischen mehreren alternativen Ereignissen darf nur genau eines der Ereignisse eintreten. Ist das gültige Ereignis gefunden, so kann der Status aller parallelen Ereignisse auf *ungültig* gesetzt werden, ohne daß für diese noch eine explizite Überprüfung der Eintrittsbedingung erforderlich ist.

Um an einem Operatorknoten mit einer UND-Bedingung als Eingangsoperator den Prozeßablauf fortsetzen zu können, ist zu prüfen, ob alle direkten Vorgängerknoten einen gültigen Bearbeitungsstatus besitzen.

Sind mehrere parallele Ereignisse zu überprüfen oder Funktionen auszuführen, so muß hierfür eine sequentielle Bearbeitungsfolge gefunden werden, da keine interne bzw. externe Parallelverarbeitung möglich ist. Der einzelne Prozeßbearbeiter vor dem Bildschirm kann nur eine Tätigkeit zur Zeit erledigen. Unter der Prämisse, daß die Vergabe der Knotennummern bei der Definition des Prozesses in der typischen Abarbeitungsfolge von oben nach unten und links nach rechts vorgenommen worden

ist, kann diese Reihenfolge auch bei der Ausführung des Prozesses reproduziert werden, indem parallele Knoten in aufsteigender Reihenfolge ihrer Nummern bearbeitet werden.

Eine vor die Prozeßausführung geschaltete Hilfsfunktion dient der Auswahl des vom Anwender gewünschten Prozesses sowie der Bereitstellung des Startknotens und eines initialisierten Prozeßstatus.

Die eigentliche Abarbeitung eines Prozeßablaufs erfolgt knotenweise. Es werden die für einen Knoten erforderlichen Operationen ausgeführt und anschließend seine Nachfolgeknoten aufgerufen. Vorab wird anhand der Statusinformationen überprüft, ob ein zyklischer Aufruf eines bereits bearbeiteten Knotens vorliegt, um gegebenenfalls den Status seiner Nachfolgeknoten für einen erneuten Durchlauf zurückzusetzen.

Handelt es sich bei dem aktuellen Knoten um eine Funktion, dann wird die Anzeige der mit ihr verbundenen Informationen aus der Datenbank ausgelöst, die Funktion wird als abgearbeitet gekennzeichnet, und die Bearbeitung des Nachfolgeknotens wird aufgerufen.

Ist ein Ereignisknoten bereits als ungültig markiert oder liefert der Aufruf seiner Auswertungsfunktion eben dieses Resultat, dann erübrigt sich eine Fortsetzung des Prozesses an dieser Stelle. Anderenfalls wird sein Status auf *gültig* und für alle alternativen Ereignisse auf *ungültig* gesetzt. Sofern es sich nicht um ein Endereignis eines Prozesses handelt, wird die Prozeßausführung mit seinem Nachfolgeknoten fortgesetzt.

Bei einem Operatorknoten ist im Falle einer UND-Verknüpfung als Eingangsoperator zunächst zu prüfen, ob die vorgelagerten Knoten bearbeitet oder gültig sind, um den Prozeß an dieser Stelle fortsetzen zu dürfen. Bei einer ODER-Verknüpfung oder fehlendem Eingangsoperator ist diese Bedingung automatisch erfüllt. Je nachdem, ob ein Ausgangsoperator existiert oder nicht, wird anschließend die Bearbeitung mehrerer oder nur eines Folgeknotens ausgelöst.

3.6.4.3 Ausführung einer Funktion

Da die Ausführung einer Funktion i.a. darin besteht, die zugehörigen Informationen aus der Datenbank anzuzeigen, muß lediglich der im Knoten hinterlegte DataSet an den SQL-Generator übergeben werden bzw. ist das dem DataSet zugeordnete SQL-Template mit den in den vorigen Kapiteln beschriebenen Mechanismen zur Erzeugung eines SQL-Templates zusammenzusetzen. Der Aufruf eines externen Programms erfolgt durch Übergabe des Programmnamens und ggf. einzelnen Parametern.

3.6.4.4 Auswertung eines Ereignisses

Die automatische Auswertung eines Ereignisses besteht darin, zu prüfen, ob die inhaltliche Ausprägung eines oder mehrerer Felder einer in der vorausgehenden Funktion angezeigten Datenmenge mit einer vorgegebenen Bedingung korrespondiert. Dementsprechend gilt das Ereignis dann als eingetreten oder nicht. Wie eine solche Bedingung formuliert und ausgewertet werden kann, wird nachfolgend erläutert.

Eine Datenmenge, wie sie Abbildung 37 beispielhaft zeigt, wird vom RDBMS als Ergebnis einer Datenbankabfrage in Form einer zweidimensionalen Tabellenstruktur zurückgeliefert. Um nun ein bestimmtes Feld dieser Tabelle zum inhaltlichen Vergleich mit einem Vorgabewert zu lokalisieren, genügen im einfachsten Fall seine Zeilen- und Spaltennummer. Da aber die Position der gewünschten Spalte nicht unbedingt bekannt sein muß und diejenige der Zeilen mit Anzahl und Inhalt gelesener Datensätze variieren kann, ist auch eine Möglichkeit der Identifizierung eines Felds, unabhängig von seiner absoluten Lage, notwendig. Die Spalten lassen sich dabei relativ einfach über die korrespondierenden Attributnamen beschreiben, welche hier die Spaltenüberschriften bilden, und die Zeilen lassen sich durch die inhaltliche Ausprägung eines oder mangels Eindeutigkeit auch mehrerer Attribute bestimmen. So könnte eine Zeile identifiziert werden als "diejenige Zeile, in der das Attribut *Auftraggeber* den Wert *Volkswagen* annimmt".

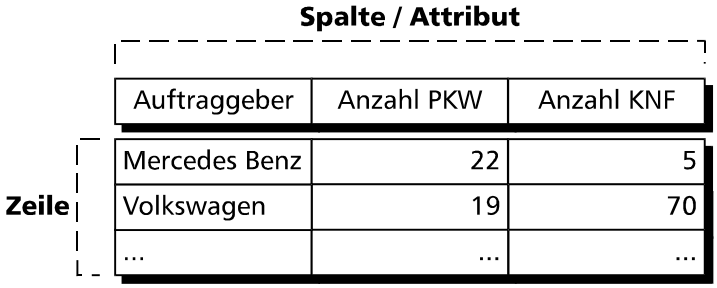


Abbildung 37: Beispielhaftes Ergebnis einer Datenbankabfrage

Neben dieser Betrachtung einzelner Felder sollen auch Summen über ganze Zeilen und Spalten als Vergleichsgrundlage herangezogen werden, wobei so aggregierte Werte in der Folge wie ein elementares Feld behandelt werden können.

Feld = *Summe* × *(Spalte)* × *(Zeile)*
Summe = *Bool*
Spalte = N_1 | *Attribut*
Zeile = N_1 | *Attribut* $\xrightarrow[m]{}$ *AttrAusprägung*
Attribut = *Char**
AttrAusprägung = *Char**

inv-Feld(feld) ≙
let (sum, sp, zl)=feld in
(sum=false Þ sp¹NIL Ù zl¹NIL) Ù (sum=true Þ sp¹NIL Å zl¹NIL)
inv-Zeile(attr) ≙
card dom attr³1

Über Rechenoperationen lassen sich derartige Felder schließlich zu einem Ausdruck zusammensetzen, so daß sich ein Vergleichswert auf eine beliebige Kombination von Feldern beziehen kann. Hier werden einfache Addition und Subtraktion dargestellt. Eine Erweiterung hinsichtlich der Integration komplexer Kennzahlensysteme ist in [Spir96] realisiert worden.

Ausdruck = *((RechenOperator)* × *Feld)*$^+$
RechenOperator = *PLUS* | *MINUS*
inv-Ausdruck(aus) ≙
aus(1).RechenOperator=NIL Ù "i Î inds tl aus · aus(i).RechenOperator¹NIL

Ein Ausdruck findet nun in einer Vergleichsbedingung seine Verwendung. Als Vergleichswerte sind hier nur numerische Konstanten vorgesehen, doch wäre eine Erweiterung um Vergleiche auf Textbasis oder um Vergleiche von Feldern untereinander prinzipiell möglich. Bedingungen können bei Bedarf negiert und per UND- oder ODER-Verknüpfung miteinander kombiniert werden. Die spätere Auswertung einer so zusammengesetzten Bedingung findet der Einfachheit halber von links nach rechts statt, da der Aufwand für die Berücksichtigung einer Klammerung oder verschiedener Prioritäten der Operatoren für diesen Einsatzzweck unangemessen erscheint.

ZusBedingung = *((VerknOperator)* × *Negation* × *Bedingung)*$^+$
VerknOperator = *AND* | *OR*
Negation = *Bool*
Bedingung = *Ausdruck* × *VerglOperator* × *VerglWert*
VerglOperator = *GROESSSER* | *KLEINER* | *GLEICH*
VerglWert = *R*
inv-ZusBedingung(zusBed) ≙
zusBed(1).VerknOperator=NIL Ù "i Î inds tl zusBed · zusBed(i).VerknOperator¹NIL

Die Auswertung zum Zeitpunkt der Prozeßausführung erfolgt mehrstufig. Das Ergebnis einer zusammengesetzten Bedingung ergibt sich aus den Ergebnissen seiner einzelnen Bedingungen. Eine einzelne Bedingung wiederum benötigt das Resultat des in ihr enthaltenen Ausdrucks. Dieser Ausdruck

wird berechnet aus den beteiligten Feldern, deren Inhalte schließlich aus den gelesenen Datensätzen entnommen werden. Die sich hierarchisch aufrufenden Module sind nachfolgend beschrieben; lediglich die Funktion zum Auslesen eines Datenfeldes oder einer Summe von Datenfeldern wird an dieser Stelle nicht näher spezifiziert, da ihr Aufbau zu sehr von der konkreten Implementierung abhängig ist.

```
type zusBedingungAuswerten:  ZusBedingung        ® Bool
type bedingungAuswerten:     Bedingung           ® Bool
type ausdruckAuswerten:      Ausdruck            ® R
type feldAuswerten:          Feld                ® R

zusBedingungAuswerten(zusBed) ≙
let bedErg=bedingungAuswerten(zusBed(1).Bedingung) in
      let bedErgNeu=if Ølen tl zusBed
                        then bedErg
                        else    (let op=zusBed(2).VerknOperator in
                            ( op=AND     ® bedErg Ù zusBedingungAuswerten(tl zusBed)
                            , op=OR   ® bedErg Ú zusBedingungAuswerten(tl zusBed)) in
          let neg=zusBed(1).Negation in
              if Øneg Ù ØbedErgNeu
                  then true
                  else neg Ù bedErgNeu

bedingungAuswerten(bed) ≙
    let (aus, op, wert)=bed in
        let ausErg=ausdruckAuswerten(aus) in
            ( op=GROESSER    ® ausErg>wert
            , op=KLEINER     ® ausErg<wert
            , op=GLEICH      ® ausErg=wert  )

ausdruckAuswerten(aus) ≙
    let feldErg=feldAuswerten(aus(1).Feld) in
        if Ølen tl aus
            then feldErg
            else let op=aus(2).RechenOperator in
                ( op=PLUS      ® feldErg+ausdruckAuswerten(tl aus)
                , op=MINUS  ® feldErg - ausdruckAuswerten(tl aus))
```

3.6.4.5 Überführung des Datenmodells in konkrete Speicherungsstrukturen

Das entwickelte Datenmodell soll nun in konkrete Strukturen zur Speicherung des statischen Prozeßaufbaus (funktionaler Aspekt) und zum einfachen Zugriff auf diese Daten zur Laufzeit des Systems überführt werden. Für die persistente Speicherung der Prozeßabläufe und auch für den Zugriff zur Ausführungszeit werden Datenbanktabellen als Erweiterung des Repositorys angelegt.

Um einen Graphen in einer tabellarischen Form abzubilden, muß die Struktur aus Knoten und der Menge seiner Nachfolger zu einer Anzahl paarweiser Kombinationen aus jeweils einem Knoten und einem Nachfolger aufgebrochen werden. Ermittelt man per entsprechender Datenbankabfrage alle Nachfolger eines Knotens, so erhält man das gesuchte Ergebnis allerdings wie gewünscht als Menge zurückgeliefert. Eine tabellarische Darstellung des Beispielgraphs ist in Abbildung 34 dargestellt.

EPK_Prozeß	VorKnoten	NachKnoten
	1	2
	2	3
	2	8
	3	4
	4	5
	4	6
	...	...

Tabelle 21: Tabelle EPK_Prozeß

Die Zuordnung der zu einem bestimmten Prozeß gehörigen Knoten muß dementsprechend ebenfalls paarweise erfolgen.

EPK_Prozesse	ProzessNr	KnotenNr
	1	1
	1	2
	1	3
	...	...

Tabelle 22: Tabelle EPK_Prozesse

Weiterhin gehören zu einem Prozeß sein Name und eine Beschreibung.

EPK_ProzeßInfo	ProzeßNr	Name	Beschreibung
	1	Fahrerausfall	Suche nach Ersatzfahrer oder –zug
	...	...	...

Tabelle 23: Tabelle EPK_ProzeßInfo

EPK_KnotenTyp	KnotenNr	Typ
	1	Ereignis
	2	Operator
	3	Funktion
	...	...

Tabelle 24: Tabelle EPK_KnotenTyp

Den einzelnen Knoten wird zunächst ihr jeweiliger Typ zugeordnet, um dann getrennt für jeden Knotentyp die für ihn benötigte Beschreibung festzulegen. Eine automatisch auszuwertende Bedingung eines Ereignisknotens wird dabei nicht in ihren einzelnen Komponenten und möglicherweise über mehrere Datenbanktabellen abgelegt, sondern der wesentlich kompakteren Speicherung wegen als eine zusammenhängende Zeichenkette, deren Syntax im folgenden Abschnitt genauer beschrieben wird. Auf diese Weise lassen sich außerdem automatisch und manuell auszuwertende Bedingungen einheitlich in derselben Tabellenspalte speichern, da letztere ebenfalls im Textformat vorliegen.

EPK_Operator	KnotenNr	EinOperator	AusOperator
	2		UND
	4		XOR
	9	UND	

Tabelle 25: Tabelle EPK_Operator

EPK_Ereignis	KnotenNr	Bedingung	Typ
	1	...	automatisch
	5	...	manuell
	6	...	manuell
	10	...	automatisch

Tabelle 26: Tabelle EPK_Ereignis

EPK_Funktion	KnotenNr	Name	Beschreibung	Tool	FilterNr
	3	...	...	Diagramm	1
	7	...	...	Tabelle	
	8	...	...	Tabelle	

Tabelle 27: Tabelle EPK_Funktion

In Abbildung 38 sind die Abhängigkeiten zwischen den neu im Repository eingeführten Datenbanktabellen im Überblick dargestellt.

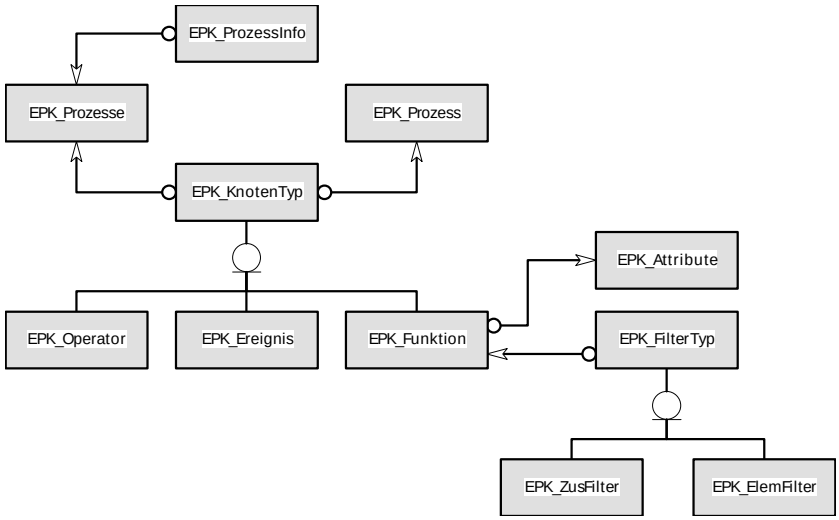


Abbildung 38: Struktur der Prozeßtabellen

3.6.4.6 Steuerung von Prozessen

3.6.4.6.1 Auslesen der statischen Prozeßinformationen

Der Zugriff auf die Informationen der oben beschriebenen Tabellen erfolgt über Hilfsfunktionen, die die gewünschten Daten zur Laufzeit des Systems über entsprechende SQL-Abfragen anhand der identifizierenden Nummer eines Prozesses, Knotens oder Filters aus der Datenbank gewinnen. Sofern das Ergebnis dieser Abfragen eine Menge bildet, speziell bei der Ermittlung von Vorgänger- und Nachfolgeknoten, werden die Daten als Array zurückgeliefert. Da die Daten bereits in sortierter Folge in dieses Array eingelesen werden können, entfällt damit eine gesonderte Sortierung, wenn parallele Knoten in einer bestimmten Reihenfolge durchlaufen werden sollen.

Die erforderlichen SQL-Abfragen sind in der Regel von simpler Struktur; lediglich bei der Bestimmung vorangehender, nachfolgender und paralleler Knoten ist besonders zu berücksichtigen, daß die gesuchten Knoten dem gleichen Prozeß wie der gegebene angehören, da aufgrund der mehrfachen Verwendbarkeit von Teilprozessen auch Nachfolger und Vorgänger für einen Knoten definiert sein können, die aus unterschiedlichen Prozessen stammen. Der erforderliche SQL-Ausdruck ist nachfolgend einmal für die Bestimmung aller parallelen eines gegebenen Knotens beispielhaft aufgezeigt.

```
SELECT NachKnoten
FROM   EPK_Prozess, EPK_Prozesse
WHERE  NachKnoten = KnotenNr          AND
       ProzessNr  = gegebene ProzeßNr AND
       VorKnoten IN (SELECT VorKnoten
                     FROM   EPK_Prozess, EPK_Prozesse
                     WHERE  NachKnoten = gegebene KnotenNr AND
                            VorKnoten  = KnotenNr          AND
                            ProzessNr  = gegebene ProzeßNr) ;
```

3.6.4.6.2 Überführung der rekursiven Prozeßausführung in eine sequentielle Struktur

Die Hauptfunktion zur Abarbeitung der einzelnen Prozeßschritte ist sinnvollerweise als rekursive Struktur modelliert worden. Bei der konkreten Überführung in die eingesetzte Programmiersprache kann diese Form allerdings nicht vollständig beibehalten werden, weil die Kontrolle über die Programmausführung jeweils bei Erreichen eines Funktionsknotens an den Anwender zurückgegeben werden muß; denn der Prozeß soll schließlich schrittweise von einem Funktionsknoten zum nächsten durchlaufen werden, damit der Benutzer sich an diesen Stellen die mit der Funktion verknüpften Informationen anzeigen lassen, Bearbeitungsstände speichern, den Prozeß adaptieren oder auf Wunsch den Prozeß fortsetzen kann.
Die rekursive Ausführung wird daher immer nur bis zum Erreichen des nächsten Funktionsknotens fortgesetzt. Da somit aber die in der rekursiven Aufrufstruktur implizit enthaltene Information darüber,

welche anderen parallelen Zweige nach Beendigung eines Teilprozesses im weiteren noch abzuarbeiten sind, verlorengeht, müssen diese Daten explizit zwischengespeichert werden. Dieser Zwischenpuffer wird als *Stapelspeicher* implementiert, da immer der zuletzt abgelegte Funktionsknoten als nächster wieder bearbeitet wird.

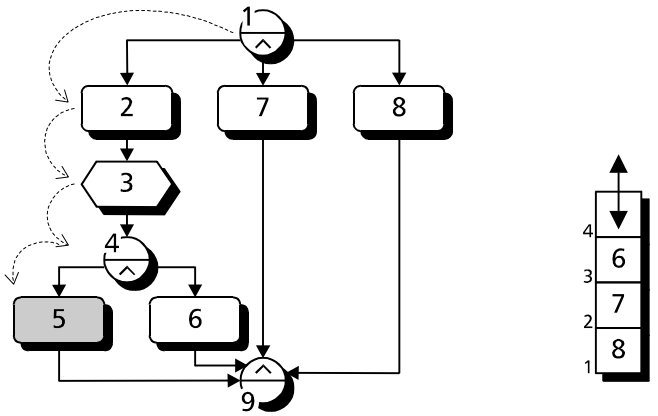


Abbildung 39: Rekursive Aufrufstruktur paralleler Funktionen

Abbildung 39 verdeutlicht diese Vorgehensweise anhand eines kleinen Ausschnitts aus einem Prozeß. Im Anschluß an den Operatorknoten 1 sind drei parallele Zweige zu durchlaufen, so daß die Knotennummern 7 und 8 vorübergehend auf dem Stack abgelegt werden, um zunächst mit Knoten 2 fortzufahren. Die gleiche Situation ergibt sich am Operatorknoten 4, wo Knoten 6 zwischengespeichert und der Prozeß mit Knoten 5 fortgesetzt wird. Am Knoten 9 angelangt, kann der Prozeß nicht fortgeführt werden, weil noch nicht alle Zweige bearbeitet worden sind. Daher werden jetzt nacheinander die Funktionen 6, 7 und 8 vom Stack geholt und ausgeführt, so daß schließlich die Bedingung 9 erfüllt ist.

3.6.4.6.3 Prozeßfortsetzung bei alternativen Verzweigungen

Alternative Ereignisse, die einem Operatorknoten mit XOR-Verzweigung folgen, werden in der Form verarbeitet, daß, sobald eines dieser Ereignisse durchlaufen wird, sämtliche parallelen Ereignisse daraufhin überprüft werden, welches von ihnen das gültige ist. Mit diesem wird die Prozeßausführung dann fortgesetzt, während alle übrigen als ungültig markiert werden.

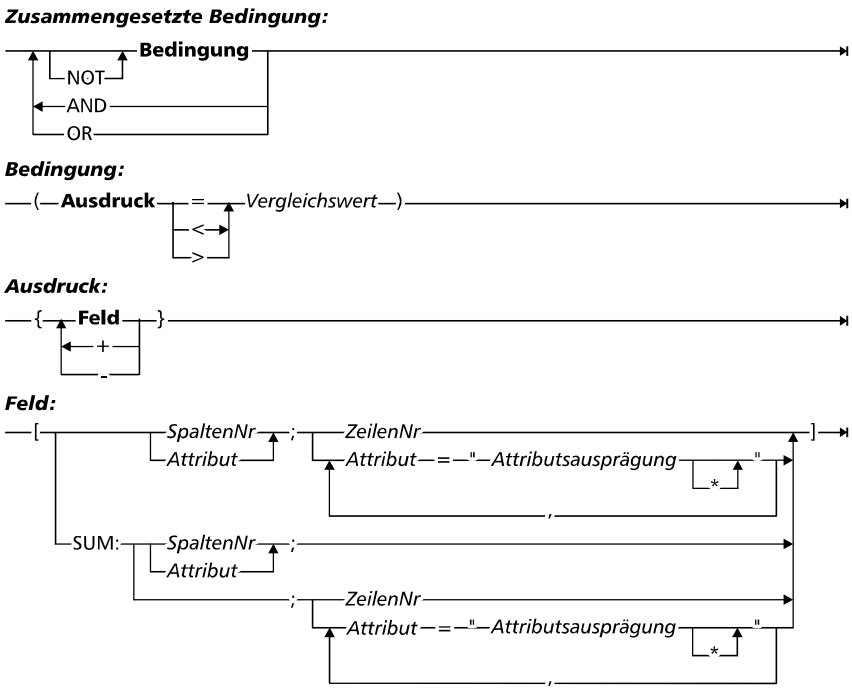


Abbildung 40: Syntax einer Ereignisbedingung

Handelt es sich um manuell auszuwertende Ereignisse, so wird die Auswahl des eingetretenen Ereignisses direkt per Dialogbox durch den Prozeßbenutzer vorgenommen, anderenfalls muß eine Auswertung anhand der hinterlegten Bedingungen durch das Programm erfolgen. Diese Bedingungen, deren Struktur oben beschrieben worden ist, sind in Form von Zeichenketten abgelegt, so daß vor der Berechnung ihres Ergebnisses ein Parsen der Zeichenfolge erforderlich wird. Anhand von Schlüsselzeichen werden dabei die verschiedenen Komponenten identifiziert und aus der Zeichenkette extrahiert, um anschließend inhaltlich ausgewertet zu werden. Die Syntax einer solchen Zeichenkette kann anhand der in Abbildung 40 dargestellten Syntaxdiagramme nachvollzogen werden.

Die *SpaltenNr* und *ZeilenNr* sind dabei natürliche Zahlen größer Null, der Vergleichswert ist eine reelle Zahl, und *Attribut* sowie *Attributausprägung* sind beliebige Zeichenfolgen.
Einfache Ereignisse, die direkt einem Funktionsknoten folgen und keiner Auswertung bedürfen, werden auch als Typ *automatisch* gespeichert, enthalten aber zu ihrer speziellen Kennzeichnung als Bedingung die Zeichenfolge „TRUE“.

3.6.4.6.4 Speicherung von Bearbeitungsständen

Um einen beliebigen Bearbeitungsstand einer Prozeßausführung abzuspeichern, müssen alle Informationen, die zur Laufzeit den aktuellen Status des Prozesses beschreiben, gesichert werden. Dabei handelt es sich neben der Nummer des aktuellen Prozesses und Knotens insbesondere um den Status der einzelnen Knoten und um den Inhalt des Stacks mit noch zu verarbeitenden Parallelfunktionen. Gespeichert werden diese Daten direkt in einer gesonderten Tabelle auf dem Datenbankserver, womit sich eine einfache Möglichkeit ergibt, von jedem Rechnerarbeitsplatz auf die Bearbeitungsstände zuzugreifen und diese somit zwecks Delegation oder Weiterleitung zwischen mehreren Bearbeitern auszutauschen. Sicherheitsbedenken bestehen bei diesem Vorgehen keine, weil benutzergebundene Zugriffsrechte auf sensible Informationen bereits Bestandteil der Reportingplattform sind und dann zum Tragen kommen, wenn der tatsächliche Zugriff auf die Daten erfolgt.

EPK_Sicherung	Name	ProzeßNr	KnotenNr	Stack	ProzeßStatus
	...	1	9	6,7,8	1,1,1;2,2,1;3,3,1;4,4,1;5,5,1

Tabelle 28: Tabelle EPK_Sicherung

Die Inhalte der Stack- und der Prozeßstatusarrays werden zur Speicherung in eine Zeichenkette konvertiert, so daß sich jeder Bearbeitungsstand in nur einem Datensatz festhalten läßt. Ansonsten bräuchten diese Daten zusätzliche Tabellen, bei denen für jeden Index jener Arrays ein eigener Datensatz benötigt würde. Außerdem befinden sich die Daten damit bereits in einem geeigneten Format, sollte sich die Speicherungsform einmal ändern, z.B. in eine Datei auf dem lokalen Rechner.

Der Stackstring setzt sich inhaltlich aus den enthaltenen Knotennummern zuzüglich benötigter Trennzeichen zusammen. Der Prozeßstatus wird in Einheiten von jeweils drei zusammengehörigen Zahlenwerten abgelegt, von denen der erste die Knotennummer und gleichzeitig den Arrayindex wiedergibt. Dieser wird deshalb ausdrücklich benötigt, weil hier nicht der Inhalt des gesamten Prozeßstatusarrays gespeichert wird, sondern nur die Felder, die einen Knoten repräsentieren, der bereits einen Status erhalten hat. An zweiter Stelle folgt die Aufrufnummer, die die Reihenfolge beschreibt, in der die Knoten durchlaufen worden sind, und an dritter Stelle wird durch die Werte 0 und 1 ausgedrückt, daß ein Knoten als *ungültig* bzw. *gültig* gekennzeichnet ist.

Um einen gespeicherten Stand wiederherzustellen, muß neben dem Einlesen der gesicherten Variablen- und Arraywerte auch die Datenmenge des zuletzt ausgeführten Funktionsknotens neu erzeugt werden. Da die aktuelle Knotennummer bereits auf den als nächstes zu bearbeitenden Knoten verweist, muß hierzu aus der protokollierten Aufrufreihenfolge der letzte Knoten ermittelt werden, der eine Funktion darstellt.

3.6.4.6.5 Darstellung der Prozeßstruktur

Während eine graphische Abbildung die gesamte Prozeßstruktur in ihrem Zusammenhang zeigt, ist in einer reinen Textform eine solch übersichtliche Darstellung schwer zu realisieren. Es kann daher jeweils nur ein einzelner zu bearbeitender Knoten dargestellt werden, dem zur besseren Identifikation eine Kurzbeschreibung beigefügt wird, die je nach Knotentyp aus dem Namen einer Funktion, der

Bedingung eines Ereignisses oder den Ein- und Ausgangsoperatoren besteht. Eine solche Beschreibung ist aber keineswegs über den ganzen Prozeß hinweg eindeutig, da sicherlich verschiedene gleichartige Operatorknoten Verwendung finden und einfache Ereignisse, in denen keine Ablaufbedingung hinterlegt ist, überhaupt nicht weiter beschrieben werden.

Unter der Prämisse, daß der Anwender nicht mit den intern vergebenen Knotennummern konfrontiert werden soll, ist eine direkte Auswahl eines zu bearbeitenden Knotens folglich nicht möglich, da die Knoten teilweise nur in Verbindung mit ihren Nachbarknoten identifizierbar sind. Zum aktuell ausgewählten Knoten werden daher jeweils alle erreichbaren Vorgänger und Nachfolger aufgelistet, die zudem die einzig mögliche Form der Navigation durch den Prozeßgraphen darstellen, indem durch schrittweise Auswahl eines Vorgänger- oder Nachfolgeknotens ein gewünschter Knoten im Prozeß angesteuert wird. Bei der praktischen Arbeit wird sich dieses Vorgehen als weniger umständlich erweisen, da ein Prozeß überwiegend sequentiell in der Reihenfolge der Knotenanordnung modelliert wird, also Sprünge innerhalb des Prozesses eher selten notwendig werden. Dabei wird allerdings vorausgesetzt, daß der Prozeßablauf nicht erst zu diesem Zeitpunkt entworfen wird, sondern idealerweise bereits als graphisches Modell vorliegt, welches mit Hilfe des Prozeßeditors nur noch in das System übertragen werden soll.

3.6.4.6.6 Einfügen neuer Knoten

Mit Ausnahme vom Startknoten eines Prozesses läßt sich ein neuer Knoten grundsätzlich nur als Nachfolger eines bereits vorhandenen Knotens in den Prozeß einfügen. Damit ist sichergestellt, daß er sofort in den Prozeß integriert ist, also keine isolierten Knoten zustande kommen können. Weiterhin läßt sich auf diese Weise eine unzulässige Kombination von Knotentypen ausschließen, da dem Anwender nur die jeweils erlaubten Typen als Nachfolger zur Auswahl gestellt werden, so daß eine wichtige Konsistenzbedingung des Prozesses bereits von vornherein erfüllt ist. Soll der Prozeß dabei an einem Operatorknoten fortgesetzt werden, so ist zur Bestimmung des zulässigen Nachfolgers zu überprüfen, welcher Knotentyp, also Funktion oder Ereignis, dem Operator oder einer Reihe von Operatoren unmittelbar vorausgeht.

Ob diese Knotentypen auch in der richtigen Anzahl miteinander verknüpft sind oder beispielsweise ein logischer Operator mit den korrekten Vorgänger- oder Nachfolgertypen kombiniert ist, kann zum Zeitpunkt der Modellierung nicht überprüft werden, solange das Prozeßmodell noch in Bearbeitung ist. Es kann nicht zu jedem Zeitpunkt konsistent sein, da noch nicht alle notwendigen Knoten plaziert wurden und an den Beschreibungen der bestehenden Knoten noch jederzeit Änderungen durch den Anwender möglich sind. Bei einer Adaption zur Laufzeit erfolgt sofort die Überprüfung der Prozeßkonsistenz.

Jeder neu eingefügte Knoten erhält bei der Definition automatisch seine interne Knotennummer, über die er eindeutig identifiziert werden kann. Es wird dabei immer die den Nummern der bestehenden Knoten folgende, nächst höhere Zahl vergeben, so daß die Nummern eventuell gelöschter Knoten nicht wieder neu vergeben werden. Auf diese Weise ist sichergestellt, daß parallele Knoten bei der Prozeßausführung in der gleichen Reihenfolge wie bei der Prozeßdefinition bearbeitet werden, da ihr Aufruf in aufsteigender Reihenfolge der Knotennummern vorgenommen wird. Dieselbe Vorgehensweise bei der Nummernvergabe erfolgt auch für neue Prozesse und Filter.

3.6.4.6.7 Einfügen neuer Verbindungen zwischen Knoten

Da es auch möglich sein muß, verschiedene Zweige eines Prozesses in einem Knoten wieder zusammenzuführen, kommen nicht nur neu zu erstellende Knoten als Nachfolger in Frage, sondern es muß auch eine Möglichkeit geben, Verknüpfungen zwischen bestehenden Knoten herzustellen, also einen bereits existierenden Knoten als Nachfolger eines anderen zu definieren. Die Prüfung, ob die richtigen Knotentypen miteinander kombiniert werden, muß dann auch auf den einzufügenden Nachfolger ausgedehnt werden. Es ist also bei einem Operatorknoten als potentiellem Nachfolger zu überprüfen, ob ihm ein Ereignis oder eine Funktion folgt.

Bei diesem Test ist zu berücksichtigen, daß anders als bei der Feststellung der Vorgänger in der Modellierungsphase noch nicht alle oder auch gar keine Nachfolgeknoten des Operators festgelegt sein

müssen, da sich das Prozeßmodell noch in Bearbeitung befindet. Die in *Kapitel 3.6.4.1 Herleitung des Datenmodells* beschriebene Testfunktion, die sich auf einen fertiggestellten Prozeß bezieht, muß daher modifiziert werden.

```
type nachKnotenTyp1:   KnotenNr × Prozeß                  ~→ Knotenbeschreibung
type nachKnotenTyp2:   KnotenNr × Prozeß × KnotenNr-set   ~→ Knotenbeschreibung
nachKnotenTyp1(node, proc) ≙
    nachKnotenTyp2(node, proc, proc(node).Nachfolgeknoten)
nachKnotenTyp2(node, proc, nextnodes) ≙
    if Øis-Operatorknoten(proc(node).Knotenbeschreibung)
        then proc(node).Knotenbeschreibung
        else if nextnodes ¹ {}
            then let next Î nextnodes in
                let type=nachKnotenTyp2(next, proc, proc(next).Nachfolgeknoten) in
                if Øis-Operatorknoten(type)
                    then type
                    else nachKnotenTyp2(node, proc, nextnodes-{next})
            else proc(node).Knotenbeschreibung
```

3.6.4.6.8 Löschen von Knoten

Soll ein beliebiger Knoten aus einem Prozeß gelöscht werden, so ist zunächst zu prüfen, ob diese Löschung insofern zulässig ist, als daß ein Prozeß dadurch nicht in zwei unabhängige Teile zertrennt wird. Bei der vorliegenden Form der Navigation durch den Prozeß, also der schrittweisen Anwahl eines Nachbarknotens, wäre der abgetrennte Teilprozeß dann nicht mehr erreichbar. Jeder Nachfolger des zu löschenden Knotens muß daher mindestens einen weiteren Vorgänger besitzen, um den Zusammenhalt eines Prozesses zu gewährleisten. Will der Anwender also beispielsweise einen Teilprozeß entfernen, so muß er zuvor eine neue Verbindung zwischen dessen Vorgänger- und Nachfolgeknoten erstellen.

Bei der eigentlichen Löschung des Knotens aus den entsprechenden Datenbanktabellen ist zu berücksichtigen, daß derselbe Knoten und auch dieselbe Verbindung zu seinen Nachbarknoten in verschiedenen Prozessen Verwendung finden kann (s.u.). Sollte er also noch in anderen Prozessen wiederzufinden sein, so wird er lediglich aus der Beschreibung des aktuellen Prozesses entfernt, und es wird die Verbindung zu seinen Nachbarknoten nur dann aufgehoben, wenn eine identische Verbindung in keinem anderen Prozeß enthalten ist; ansonsten kann die gesamte Knotenbeschreibung gelöscht werden, da sie nicht mehr anderweitig benötigt wird. Dazu gehört auch, daß im Falle eines Funktionsknotens ebenfalls die Referenz auf den DataSet oder ein externes Programm entfernt werden.

3.6.4.6.9 Löschen einer Verbindung zwischen Knoten

Bei der Löschung einer Verbindung zwischen zwei Knoten ist analog zur Löschung eines Knotens zu prüfen, ob eine identische Verknüpfung derselben Knoten nicht noch in einem anderen Prozeß benötigt wird und ob die Nachfolgerknoten noch Verbindungen zu weiteren Vorgängern besitzen, damit der Prozeß nicht zerteilt wird. Da die beiden Knoten selbst erhalten bleiben sollen, wird aus der Datenbank lediglich der Datensatz entfernt, der die Vorgänger-Nachfolgerbeziehung zwischen ihnen beschreibt.

3.6.4.6.10 Einfügen von Prozessen

Damit sich auch mehrfach verwendbare Unterprozesse definieren und in andere Prozesse integrieren lassen, besteht die Möglichkeit, in einem Knoten nicht nur einen weiteren Knoten, sondern einen kompletten Prozeß als Nachfolger einzufügen. Dabei stehen zwei verschiedene Methoden des Einfügens zur Auswahl.

Einfügen eines Prozesses als Referenz

Wird ein Prozeß als Referenz in einen anderen eingefügt, so werden lediglich die Knotennummern des einzufügenden Prozesses in die Prozeßbeschreibung des Zielprozesses aufgenommen, und es wird eine Verbindung zwischen dem Anknüpfknoten im Zielprozeß und dem Startknoten des eingefügten Prozesses hergestellt. Im Zielprozeß sind jetzt dieselben Knoten mit den identischen Nummern wie im Quellprozeß enthalten, so daß sich Änderungen dieser Knoten oder ihrer Verknüpfungen untereinander

auf jeden Prozeß auswirken, in dem sie vertreten sind. Neue Verbindungen zu Knoten, die zum Zeitpunkt des Einfügens nicht Bestandteil des Quellprozesses waren, bleiben hingegen lokal begrenzt.
Ein Prozeß darf in sich selbst nicht eingefügt werden, da ein mehrfaches Auftreten identischer Knotennummern innerhalb eines Prozesses nicht zulässig ist. Ansonsten wäre mangels Eindeutigkeit der Verknüpfungen zwischen den Knoten eine korrekte Prozeßausführung nicht gewährleistet.

Einfügen eines Prozesses als Kopie

Beim Einfügen eines Prozesses als Kopie werden sämtliche Knoten des einzufügenden Prozesses dupliziert. Es werden vollständig neue Knoten generiert, die lediglich eine inhaltliche Kopie der Quellknoten darstellen, aber über eine neue und eigenständige Identifikationsnummer verfügen. Änderungen an einem Knoten haben somit auf die Kopie keinerlei Auswirkung. Die Nummern der neuen Knoten errechnen sich dabei aus der Nummer des Quellknotens plus einem Offsetwert, der sich aus der Differenz zwischen der nächst höheren zu vergebenden Knotennummer und der kleinsten Knotennummer des Quellprozesses ergibt. Auf diese Weise läßt sich auch die Beschreibung der Vorgänger-Nachfolgerbeziehungen des Quellprozesses direkt kopieren, auch wenn dieser Lücken im Durchnumerieren seiner Knoten aufweist. Würden fortlaufend neue Knotennummern vergeben, so müßte für den Zielprozeß aufwendig die Anordnungsstruktur der neuen Knoten analysiert werden.
Das Kopieren der Knoten stellt in der Regel ein einfaches Duplizieren der Datensätze mit der Knotenbeschreibung dar, die nur mit ihrer neuen Knotennummer versehen werden müssen.

3.6.4.6.11 Bearbeiten einer Knotenbeschreibung

Die Knotenbeschreibungen, also je nach Knotentyp z.B. der Funktionsname, die Ereignisbedingung oder die logischen Operatoren, müssen bereits bei der Erstellung eines neuen Knotens zwingend vom Anwender eingegeben werden, weil sie ihm während der Prozeßmodellierung zur Identifikation der Knoten dienen; sie können aber auch im Nachhinein noch editiert werden. Lediglich das Startereignis eines Prozesses sowie Ereignisse, die direkt einem einzelnen Funktionsknoten folgen, erhalten keine weitere Beschreibung, da sie für die Ablaufsteuerung des Prozesses ohne Bedeutung sind. Die Beschreibungen werden während der Modellierungsphase im wesentlichen über einfache Dialogfenster eingelesen und in den entsprechenden Datenbanktabellen gespeichert.

3.6.4.6.12 Erstellung einer Gesamtübersicht des Prozesses

Da während des Editierens oder während der Ausführung immer nur ein kleiner Ausschnitt des Prozesses (der aktuelle Knoten mitsamt seiner Vorgänger und Nachfolger) sichtbar ist, wird eine einfache Darstellungsmöglichkeit für den Prozeß in seiner Gesamtheit implementiert. Diese Gesamtübersicht soll zum einen in der Form Verwendung finden, daß sie zur Druckausgabe in einer Textdatei abgelegt wird; zum anderen soll sie auch als alternative Möglichkeit zur direkten Auswahl eines Knotens zum Einsatz kommen, da bei der gezielten Suche nach einem Knoten das schrittweise Durchhangeln durch den Prozeß unter Umständen zu zeitaufwendig werden kann.
Die Gestaltungsmöglichkeiten für die Prozeßübersicht sind bei der Beschränkung auf eine reine Textausgabe recht begrenzt, doch lassen sich zumindest die Typen und Beschreibungen der Knoten sowie ihre Vorgänger-Nachfolgerbeziehungen abbilden. Dazu werden die Knoten in ihrer Anordnungsreihenfolge zeilenweise untereinander aufgeführt, wobei jeweils alle direkten Nachfolger eines Knotens um eine Spalte versetzt aufgelistet werden. Parallele Knoten lassen sich somit dadurch identifizieren, daß sie bezüglich ihrer Zeilenposition auf gleicher Ebene dargestellt werden. Für den Beispielprozeß in Abbildung 34 gestaltet sich dieses wie folgt:

```
(1) Ereignis
   (2) Operator (---/AND)
       (3) Funktion (...)
          (4) Operator (---/XOR)
              (5) Ereignis (...)
                  (3) Funktion (...)
              (6) Ereignis (...)
                  (7) Funktion (...)
                      (9) Operator (AND/---)
       (8) Funktion (...)
          (9) Operator (AND/...)
              (10) Ereignis
```

Die algorithmische Beschreibung der zur Erzeugung dieser Übersicht erforderlichen Funktionen wird nachstehend vorgenommen. Zuvor wird der Verwendungszweck der an dieser Stelle neu einzuführenden Datentypen erläutert.

Die zu erstellende *ProzeßÜbersicht* besteht aus einer geordneten Folge von Zeilen, die jeweils die Beschreibung eines Knotens enthalten. Zur Erstellung dieser Übersicht wird der Prozeß ähnlich wie bei der Prozeßausführung knotenweise durchlaufen, wobei jeweils der aktuelle Knoten in die *ProzeßÜbersicht* aufgenommen wird und sich dieser Vorgang dann rekursiv für alle seine Nachfolger wiederholt. In der Menge *KnotenFertig* werden alle Knoten vermerkt, die bereits in der *ProzeßÜbersicht* enthalten sind. Dieses ist zum einen notwendig, um bei zyklischen Verkettungen die Auswertung eines Prozeßzweigs abzubrechen, sobald ein Knoten eine zweites Mal durchlaufen wird; zum anderen kann so festgestellt werden, ob schon alle Vorgänger eines Operatorknotens durchlaufen wurden und die Prozeßauswertung somit an dieser Stelle fortgesetzt werden kann, oder ob zuvor noch ein vorangehender Teilprozeß zu bearbeiten ist.

Der Bearbeitungsstand der Vorgänger ist kein alleiniges Kriterium für die Prozeßfortsetzung, da im Falle eines Zyklus ein Vorgänger eines Knotens möglicherweise erst im Anschluß an diesen durchlaufen wird. Daher gibt die Anzahl *ParallelÜbrig* an, wie viele parallele Teilprozesse noch rekursiv abzuarbeiten sind. Sind noch nicht alle Vorgänger eines Knotens bearbeitet, aber auch kein vorangehender Teilprozeß mehr zu durchlaufen, so muß die Prozeßauswertung an dieser Stelle trotzdem fortgesetzt werden, sofern nicht ein Endknoten des Prozesses erreicht ist. Bei der Zeichenkette *Einrückung* handelt es sich um eine je nach Rekursionstiefe variable Folge von Leerzeichen, die für die zeilenweise versetzte Darstellung der Knotenbeschreibungen verantwortlich zeichnet.

ProzeßÜbersicht = *KnotenInfo**
KnotenInfo = *Char**
KnotenFertig = *KnotenNr-set*
ParallelÜbrig = N_0
Einrückung = *Char**

Neben der Hauptfunktion zur Erstellung der Prozeßübersicht sowie einer Funktion zu ihrer Initialisierung kommen zwei Hilfsfunktionen zum Einsatz, von denen die eine für die sequentielle Auswertung aller Nachfolger eines Knotens zuständig ist; die andere, die hier nicht weiter spezifiziert werden soll, dient der Erstellung einer Zeichenfolge mit einer Knotenbeschreibung.

```
type prozeßübersichtErstellen:  Prozeß                ® ProzeßÜbersicht
type prozeßübersichtErstellen2: KnotenNr × Prozeß ×KnotenFertig × ParallelÜbrig × Einrückung ×
                                                                          ProzeßÜbersicht
                                                      ® KnotenFertig × ParallelÜbrig × ProzeßÜbersicht
type nachfolgerDurchlaufen:     KnotenNr-set x Prozeß x KnotenFertig × ParallelÜbrig × Einrückung ×
                                                                          ProzeßÜbersicht
                                                      ® KnotenFertig × ParallelÜbrig × Prozeßübersicht
type knoteninfoErzeugen:    KnotenNr × Prozeß ® KnotenInfo
prozeßübersichtErstellen(proz) ≙
    if dom proz={}
        then [ ]
        else let (kFertig, pÜbrig, pÜbers)=prozeßübersichtErstellen2(startKnoten(proz), proz, {}, 0, "", []) in
            pÜbers
prozeßübersichtErstellen2(kNr, proz, kFertig, pÜbrig, einrück, pÜbers) ≙
    let pÜbersNeu=pÜbers...[einrück...knoteninfoErzeugen(kNr, proz)] in
        if kNr Ï kFertig Ù (vorKnoten(kNr, proz) Í kFertig Ú (ØpÜbrig Ù nachKnoten(kNr, proz)¹{}))
            then let pÜbrigNeu=pÜbrig+card nachKnoten(kNr, proz) in
                let kFertigNeu=kFertigÈ{kNr} in
                    nachfolgerDurchlaufen( nachKnoten(kNr, proz), proz, kFertigNeu,
                                           pÜbrigNeu, einrück..."_", pÜbersNeu)
            else (kFertigNeu, pÜbrigNeu, pÜbersNeu)
```

```
nachfolgerDurchlaufen(kNrNach, proz, kFertig, pÜbrig, einrück, pÜbers) ≙
    if kNrNach={}
        then (kFertig, pÜbrig, pÜbers)
        else let kNr=min kNrNach in
            let (kFertigNeu, pÜbrigNeu, pÜbersNeu)=prozeßübersichtErstellen2(kNr, proz, kFertig,
                                                                    pÜbrig-1, einrück, pÜbers) in
                nachfolgerDurchlaufen(kNrNach-{kNr}, proz, kFertigNeu, pÜbrigNeu, einrück, pÜbersNeu)
```

3.6.4.6.13 Prüfung der Konsistenz eines Prozesses

Einige syntaktische Regeln der Prozeßstruktur werden bereits zur Bearbeitungszeit überwacht, und zwar insbesondere die Verbindung nur zulässiger Knotentypen und die Verhinderung der Entstehung isolierter Knoten. Eine Reihe weiterer Prüfungen kann während der Modellierungsphase erst nach Fertigstellung des Prozeßmodells durchgeführt werden, denn solange noch Knoten hinzugefügt, andere wieder gelöscht und Knotenbeschreibungen modifiziert werden können, ist die permanente Konsistenz eines Prozesses nicht gewährleistet.

Bevor der Anwender also während der Laufzeit den Prozeß nach einer Adaption weiter durchlaufen oder während der Modellierung den Editor verlassen kann, wird der aktuelle Prozeß zunächst bestimmten Prüfungen unterzogen, um gegebenenfalls eine Warnmeldung an der Benutzer auszugeben. Diese entsprechen denjenigen der in *Kapitel 3.6.4.1 Herleitung des Datenmodells* aufgeführten Konsistenzbedingungen, die noch nicht während der Bearbeitungsphase getestet werden konnten, also beispielsweise, ob jeder Knoten mit der richtigen Anzahl von Nachbarknoten verknüpft ist oder ob die Endknoten des Prozesses vom richtigen Typ sind.

3.6.5 Zusammenfassung

Das Prozeßtemplatekonzept in Kombination mit Componentwaremechanismen hat sich als äußerst flexibler Mechanismus für die Spezifikation von Prozeßbausteinen und deren mögliche Adaption, insbesondere auch zur Systemlaufzeit, erwiesen. Es wurden zunächst, analog der Entwicklung der SQL-Templates, Prozeßtemplates mittels einer Skriptsprache spezifiziert, die die Adaption durch Parametrisierung unterstützt und die Prozeßabläufe mittels entsprechender Ablaufbedingungen analog dem Verhaltensaspekt von Workflowmanagementsystemen steuert.

Die Prozesse wurden als EPK modelliert, ebenfalls in Bausteine separiert und in Tabellen des Repositorys abgelegt. Bei Adaptionen der Prozesse durch den Anwender werden verschiedene Konsistenzbedingungen geprüft, um zu gewährleisten, daß die Prozesse von einem konsistenten Zustand in einen ebenfalls wieder konsistenten Zustand überführt werden.

Die Prozeßtemplates sind als Komponenten im Sinne von Componentware modelliert worden und kommunizieren mit anderen Komponenten über ihre Schnittstellen. Bei der Übergabe einer Prozeß-komponente, die aus mehreren Prozeßkomponenten besteht, an das Laufzeitsystem erfolgt keine Instanziierung des gesamten Prozesses, sondern nur der aktuell bearbeiteten Komponente. Hierdurch können Komponenten auch zur Laufzeit ausgetauscht bzw. um andere Komponenten ergänzt werden. Hierbei auftretende Integritätsverletzungen können teilweise nur durch Einbeziehung des Anwenders vermieden werden.

3.7 KONZEPT ZUM WIEDERAUFFINDEN ÄHNLICHER PROBLEMBESCHREIBUNGEN

3.7.1 Übersicht

Das im folgenden beschriebene Konzept ermöglicht mittels der Mechanismen des fallbasierten Schließens über Indexmechanismen (*Retrieve*) Fälle, die hinsichtlich ihrer problembeschreibenden Parameter ähnlich sind, aus einer Fallbasis zu selektieren und anschließend zu adaptieren (siehe [Egge97]). Bei der Adaption der Fälle geht es im wesentlichen darum, die Lösungen der mit einem Ähnlichkeitsmaß gefundenen Fälle an die aktuelle Problemstellung anzupassen (*Reuse*). Das Ähnlichkeitsmaß liefert eine Aussage, wie hoch der Aufwand nach erfolgter Adaption wäre, die Lösung des ähnlichen Problems an die des aktuellen Problems anzupassen. Die in der Reusephase veränderte Lösung wird hierauf auf inhaltliche Konsistenz getestet und als neuer Fall abgespeichert.

Der neue Fall ist als Erweiterung des Wissens der Fallbasis zu sehen, weil hierdurch bei einer neuen Suche eventuell Problembeschreibungen eingegeben werden, die ein hohes Ähnlichkeitsmaß zu dem neu abgespeicherten Fall aufweisen und somit die Grundlage für dessen Lösungsprozeß bilden.

Innerhalb der Reportingplattform findet das fallbasierte Schließen seine Anwendung bei dem Wiederauffinden von ähnlichen DataSets oder Prozessen, die als Grundlage einer Lösung zu einer den Anwender interessierenden Problemstellung dienen. Die Problemstellung wird durch den Anwender über *Artefakte* (Attribute, DataSets und Entscheidungsprozesse) spezifiziert. Das kleinste problembeschreibende Element aus der Menge der Artefakte ist das Attribut, welches in einem oder mehreren DataSets integriert ist.

Für jeden DataSet wird zur Laufzeit durch das System entweder ein SQL-Statement über den SQL-Generator erzeugt, ein festdefiniertes und dem DataSet zugewiesenes SQL-Statement verwendet oder ein SQL-Template adaptiert. Das kontextbezogen generierte SQL-Statement wird dann an das RDBMS übergeben.

Über die jeweilig im DataSet integrierte SQL-Anweisungsform wird dann der durch die Selektionsattribute und die dynamischen Filter spezifizierte Informationsgehalt realisiert. Daraus ergibt sich die Folgerung, daß die erste Fallbasis aus Fällen bestehen muß, die sich aus Attributen als problembeschreibende Parameter (Symptome) und aus den die Attribute in der SELECT-Klausel enthaltenen SQL-Statements als Lösungsmenge (-raum) ergibt. Eine zweite Fallbasis dient dann dem Retrieval nach Geschäftsprozessen über die aus der ersten Fallbasis ermittelten DataSets (siehe *Kapitel 4.2.2.1 Die Prozeßfallbasis*).

3.7.2 Schnittstelle zur Anwenderoberfläche

Die Schnittstelle zur Anwenderoberfläche besteht darin, daß die über die Benutzerschnittstelle ausgewählten Artefakte (hier nur die Artefakte Attribute und/oder DataSets) als Eingangsparameter für die Indexsuche dienen. Dabei müssen die Übergabeparameter in verschiedene Konstellationen (Situation 1-3) unterschieden werden.

Situation 1: Der Anwender wählt nur Attribute aus, die UND-verknüpft sind.
Bei der Auswahl einer UND-verknüpften Attributmenge werden nur DataSets ermittelt, die alle Attribute der gewählten Menge enthalten. Die Ähnlichkeitsmaßberechnung bezieht dabei lediglich auf die Abweichungen der Attributmenge der gefundenen DataSets zur gewählten Attributmenge.

Situation 2: Der Anwender wählt nur Attribute aus, die ODER-verknüpft sind.
Bei der Auswahl einer ODER-verknüpften Attributmenge werden alle DataSets ermittelt, die mindestens ein Attribut der gewählten Anfragemenge enthalten. Die Ähnlichkeitsmaßberechnung bezieht sich auf die Abweichungen der Attributmenge der gefundenen DataSets zur Attributkonstellation der gewählten Attributmenge.

Situation 3: Anwender wählt nur DataSets aus.
Durch die Auswahl eines DataSets aus dem Glossar (s.u.) wird dieser DataSet zum einen implizit mit einer Ähnlichkeit von 100% versehen und zum anderen bekundet der Anwender sein Interesse an allen in diesem DataSet gewählten Selektionsattributen. Es müssen alle Selektionsattribute des DataSets ausgelesen und in der Problembeschreibung angezeigt werden. Daher kann diese Datenkommunikation mit der Benutzeroberfläche nicht im Sinne des CBR gesehen werden, sondern eher in der Vorteilhaftigkeit, über die Indexstruktur auf die Selektionsattribute des gewählten DataSets zuzugreifen.

3.7.3 Konzeption des Retrieveprozesses

3.7.3.1 Grundlagen

Das Ziel des zu entwickelnden Retrieveprozesses besteht in der Organisation und Realisierung der notwendigen Zugriffsverfahren auf die in der Fallbasis gespeicherten Fallbeispiele. Daher soll zu Beginn der Ausführungen des in der Reportingplattform integrierten Retrieveprozesses die Struktur der

Fallbasis, die Struktur eines Fallbeispiels und die Struktur einer Anfragemenge aufgezeigt werden. Ein Fallbeispiel, im Sinne des CBR, wird innerhalb der Reportingplattform durch einen DataSet repräsentiert, da dieser sowohl die, den spezifischen Informationsbedarf beschreibenden, Attribute (Problemparameter) in der Selektionsmenge enthält, als auch einen Verweis auf die diesen Informationsbedarf realisierenden Templates oder festdefinierten SQL-Statements. Die festdefinierten Statements stellen im Sinne des CBR eigentlich keine besonders vorteilhaften Fallbeispiele dar, da sie einmalig in einer starren Form fest kodiert einem DataSet zugewiesen werden, so daß nicht die Möglichkeit besteht, die Attribute der SELECT-Klausel zu verändern, wodurch eine Anpassung an die eventuell aktuell interessierende Attributmenge ausgeschlossen wird. Es kann so keine Reusephase durchgeführt werden. Daher kann das SQL-Template also als der eigentliche Lösungsmechanismus für die in der Problembeschreibung aufgeführten Attribute angesehen werden, der eine aus den Selektionsattributen bestehende Lösungsmenge generiert, nachdem dieser zuvor die Selektionsattribute in seine spezifische Struktur integriert und daraus ein SQL-Statement transformiert, welches darauf über den SQL-Generator an das RDBMS übergeben wird.

Die Fallbasis des Systems hat nun die Funktion, alle bisher generierten DataSets zu verwalten, um sie den im Retrieveprozeß implementierten Funktionen im Anfragefall zur Ähnlichkeitsberechnung zur Verfügung zu stellen. Eine Anfragemenge setzt sich aus den durch den Anwender spezifizierten Attributen zusammen, die seinen Informationsbedarf charakterisieren. Diese Anfragemenge dient also als Vergleichskriterium zu den Fallbeispielen aus der Fallbasis (DataSets) bei der Ähnlichkeitsberechnung, um einen DataSet zu finden, welcher der Anfragemenge hinsichtlich seiner Attributkonstellation sehr ähnlich ist und somit zu dessen Lösungsrealisierung dienen kann.

Innerhalb dieser Betrachtung soll allerdings nicht davon ausgegangen werden, daß die SQL-Statements nur aus SQL-Templates gebildet werden. Es sollen auch DataSets gefunden werden, die eine vordefinierte Anweisung enthalten oder ein Defaultstatement erzeugen, da dem Anwender des Systems nach wie vor noch die Möglichkeit gegeben wird, auch DataSets zu modellieren, die auf der Basis der beiden zuletzt genannten Anweisungsformen arbeiten. Für ein geeignetes Retrieveverfahren erscheint es deshalb sinnvoll zu sein, einen Index über die Attribute in allen im System archivierten DataSets zu legen, so daß dadurch ein indizierter Zugriff auf alle DataSets durchgeführt werden kann, die ein jeweils gesuchtes Attribut enthalten. Durch die Kenntnis darüber, welche DataSets die gesuchten Attribute realisieren können, kann eine Einschränkung des Suchraums in der Art vorgenommen werden, daß jetzt nur noch diese DataSets hinsichtlich der Ähnlichkeitsbestimmung untersucht werden müssen. Außerdem wird durch diese Vorgehensweise das Auftreten von $a - Fehlern$ [Jan+93] vermieden, die daraus resultieren könnten, daß die Attributselektionsmenge eines DataSet zwar keines der Attribute aus der durch den Anwender spezifizierten Anfragemenge enthält, dennoch wird die Lösungsmenge des DataSets durch ein Template erzeugt, welches durchaus eine Teilmenge der Attribute der Anfragemenge realisieren kann. Das entsprechende DataSet, auf das diese Situation zutrifft, würde im Rahmen der Vorauswahl der Fallbeispiele (DataSets) in der Retrievephase also nicht in die Menge der potentiellen Lösungskandidaten übernommen werden. Die Bedeutung der Selektionsmenge liegt, wie innerhalb des Templatekonzepts ausführlich beschrieben, darin, daß hierin nur die Attribute enthalten sind, die den augenblicklichen Informationsumfang eines DataSets darstellen, nämlich die n der SELECT-Klausel des korrespondierenden SQL-Templates zur Ausführungszeit zu integrierenden Attribute. Allerdings wird einem DataSet auch eine Attributdefinitionsmenge zugewiesen, die eine Obermenge der Selektionsmenge bildet, da alle Attribute der Selektionsmenge auch in der Definitionsmenge enthalten sind (siehe *Kapitel 3.2.2 DataSet und Mengenwerkzeug*).

Die Differenz zwischen der Attributselektionsmenge und der Attributdefinitionsmenge repräsentieren die Attribute, die durch das korrespondierende Template potentiell noch in die SELECT-Klausel eingesetzt werden könnten. Dies geschieht dadurch, daß der Anwender zur Laufzeit die Attribute der Definitionsmenge, die bisher noch nicht der Selektionsmenge vertreten waren, dieser zuweisen kann. Diese Erweiterbarkeit der Selektionsmenge eines DataSets hat für die Suche nach geeigneten Lösungskandidaten im Retrieval folglich die Konsequenz, auch die Fallbeispiele in die Ähnlichkeitsbetrachtung mit einzubeziehen, bei denen die Schnittmenge von den Attributen der Anfragemenge und der Menge

der Selektionsattribute zwar leer ist, dagegen die Schnittmenge von den Attributen der Anfragemenge und denen der Definitionsmenge nicht leer ist.

Der Suchweg zu denen den Anwender interessierenden DataSets wird ausgehend von jedem Attribut zu der DataSet-Menge gebildet, in der das Attribut vertreten ist. Die so gefundenen DataSets können hinsichtlich ihrer Attributkonstellation untersucht werden. Jede Attributkonstellation dient jetzt als Grundlage für die Berechnung des Ähnlichkeitsmaßes, das angibt, inwieweit die SELECT-Klausel des bisherigen im DataSet integrierten Statements mit der benötigten bzw. relevanten Attributmenge übereinstimmt. Große Attributabweichungen werden besonders „bestraft“, da hierbei der Aufwand der Neugenerierung des Statements mit der gewählten Attributmenge aus dem Template bzw. durch den SQL-Generator bei einem Defaultstatement wesentlich höher ist, als bei geringeren Abweichungen. Die Ausnahme bei diesem Verfahren bilden die vordefinierten Anweisungen, weil bei dieser Anweisungsform die Selektionsattribute immer den Definitionsattributen entsprechen.

3.7.3.2 Organisation des Retrieveprozesses

Der Retrieveprozeß unterteilt sich, wie auch in den klassischen Verfahren (z.B. sequentielles Retrieval, relationales Retrieval oder Retrieval über kd-Bäume [Wess95]), in die zwei Phasen *Vorauswahl* der zur Lösung dienlichen DataSets und in das *eigentliche Retrieval* mittels Ähnlichkeitsbewertung.

Eine Vorauswahl erweist sich als sehr sinnvoll, da die so erreichte Einschränkung des Suchraums auf nur lösungsrelevante DataSets den Aufwand der eigentlichen Ähnlichkeitsberechnungen, im Verhältnis zu einem sequentiellen Ansatz, erheblich verringert. Um aber die Vorauswahl durchführen zu können, ist zu prüfen, wie die Fallbasis strukturiert werden kann, damit eine reduzierte Menge an Lösungskandidaten erreicht wird. Die einzige Information, die über die Anfragemenge erreichbar ist, stellen die in dieser Menge enthaltenen Attribute dar, woraus sich der Ansatz entwickelte, auch eine Vorauswahl über die tatsächlich interessierenden Attribute aufzubauen. Es ist also von Interesse, in welchen DataSets mindestens eines der Attribute der Anfragemenge vertreten ist, um diese der Menge der potentiellen Lösungskandidaten zuzuführen. Um diese Information zu erhalten, wird eine Tabelle im Repository genutzt, in der die im System vorhandenen Attribute, die in einem oder mehreren DataSet(s) eingebunden wurden, als Referenz auf die Menge der diese enthaltenen DataSets zeigen.

$$RPT\_DataSet\_Menge = Attributname \xrightarrow[m]{} DataSetID - set$$

$$Attributname = char^{+}$$

$$DataSetID = N_1$$

In der Tabelle *RPT_DataSet_Menge* wird jedem *Attributnamen* eines in einer beliebigen SELECT-Klausel enthaltenen Attributs die Menge aller DataSets, genauer deren *DataSetIDs*, zugeordnet, in denen das Attribut vorkommt. Durch die Tabelle kann also ein indizierter Zugriff auf alle DataSets erfolgen, deren Attributmengen das gewählte Attribut mit dem Attributnamen enthalten, wodurch der Suchraum nur auf die relevanten DataSets eingeschränkt wird.

Die über den Attributnamen aus der Tabelle *RPT_DataSet_Menge* ausgelesene Menge an DataSets kann nun elementweise in der Tabelle *RPT_Attribut_Menge* abgearbeitet werden, indem für jede *DataSetID* der ermittelten DataSet-Menge ein Zugriff auf die aktuelle Selektionsmenge und Definitionsmenge des DataSets erfolgt, die durch die Felder *(*Selektionsattribut*)*-set und (Definitionsattribut)-set spezifiziert werden. Ein für die Geschwindigkeit der Suche wesentlicher Punkt ist, daß von allen *DataSet-Mengen*, die durch einen Attributnamen aus der Tabelle *RPT_DataSet_Menge* ausgelesen worden sind, die jeweilige abgearbeitete *DataSetID* aus der Menge entfernt wird, da zuvor die Schnittmenge aller DataSet-Mengen gebildet wird. Durch diese Vorgehensweise wird die doppelte Abarbeitung eines DataSets vermieden, falls mehr als ein gewähltes Attribut in der Attributmenge eines DataSets enthalten ist.

$$RPT\_Attribut\_Menge = DataSetID \xrightarrow[m]{} Definitionsattribute \times Selektionsattribute$$
$$DataSetID = N_1$$
$$Definitionsattribute = Attribut - set$$
$$Selektionsattribute = Attribut - set$$
$$Attribut = char^+$$

3.7.3.3 Beschreibung der implementierten Retrieveverfahren

Im folgenden werden die für das Retrieval notwendigen Algorithmen zur Realisierung der Vorauswahl und dem eigentlichen Retrieval beschrieben. Dies umfaßt die Erzeugung der reduzierten Kandidatenmenge für die Berechnung des Ähnlichkeitsmaßes.

3.7.3.3.1 Retrievephase: Vorauswahl

Die Vorauswahl der relevanten Fälle wird teilweise bereits implizit durch die verwendete Tabellenstruktur erreicht, und zwar durch die vorhandenen Informationen aus den Tabellen *RPT_DataSet_Menge* und *RPT_Attribut_Menge*. Wenn also eine Anfragemenge vom Anwender bestimmt worden ist, dann werden alle Attribute der Anfragemenge als Schlüsselattribute auf die Tabelle *RPT_DataSet_Menge* angewendet, um für jedes Attribut die Menge DataSets zu erhalten, in denen es tatsächlich vorhanden ist. Nachdem für alle Attribute der Anfragemenge die jeweilige DataSet-Menge erzeugt worden ist, wird aus allen DataSet-Mengen die Schnittmenge gebildet, um ein mehrfaches Untersuchen eines DataSets zu vermeiden, falls in diesem mehr als ein Attribut der Anfragemenge enthalten ist. In einem weiteren Schritt werden alle zu den DataSets gehörigen Datensätze über die zuvor ermittelte DataSetID aus der Tabelle *RPT_Attribut_Menge* ausgelesen, wodurch gewährleistet ist, daß nur die relevanten DataSets (Fallbeispiele) in die Kandidatenmenge aufgenommen werden. Diese Vorgehensweise garantiert also die Ermittlung einer *vollständigen* und *korrekten* Kandidatenmenge, deren Erzeugung auf der *partiellen Gleichheit* [StWe89] beruht, da nur DataSets in die Menge mit aufgenommen wurden, bei denen mindestens ein Attribut mit einem Attribut der Anfragemenge identisch ist.
Bevor ein DataSet D und eine Anfragemenge A als Datentypen im weiteren Verlauf verwendet werden, sollen diese zunächst spezifiziert werden.

$$D = D_S \times D_D$$
$$D_S = Attribut - set$$
$$D_D = Attribut - set$$
$$Attribut = char^+$$
$$A = Attribut - set$$

Folgende Äquivalenzrelation gilt für jeden DataSet $D.D_D = \{a_{d1},..,a_{dn}\}$ bezüglich der Anfragemenge $Q = \{a_1,..,a_k\}$:

$$SIM(Q, D.D_D) \Leftrightarrow \exists i \in \{1..k\}; a_i = a_{di}$$

3.7.3.3.2 Retrievephase: Ähnlichkeitsmaßberechnung

Die in dieser Retrievephase durchzuführende Ähnlichkeitsmaßberechnung bezieht sich auf die in der Vorauswahl ermittelte Kandidatenmenge. Diese gewährleistet, wie im vorherigen Abschnitt erklärt, daß nur Fallbeispiele in die Ähnlichkeitsberechnungen einbezogen werden, die zumindest der partiellen Gleichheit genügen. Dadurch wird der Sachverhalt ausgedrückt, daß durch jeden DataSet aus der Kandidatenmenge mindestens ein Attribut aus der Anfragemenge realisiert werden kann. Bei der Berechnung der Ähnlichkeit sollen aber mehrere Faktoren in das Ähnlichkeitsmaß einfließen. Es soll so ein differenzierterer Vergleich zwischen der Anfragemenge und dem jeweiligen Fallbeispiel (DataSet) erfolgen. Daher werden nachfolgend die möglichen Zustände aufgezeigt, die während der Untersuchung der Übereinstimmungen der Attribute der Anfragemenge und der Selektionsattribute der DataSets der Kandidatenmenge auftreten können:

Zustand 1: Ein Attribut der Anfragemenge $A = \{a_1,..,a_i\}$ist auch in der Selektionsmenge eines DataSets $D.D_S = \{a_{s1},..,a_{sn}\}$vorhanden, was sich formal als folgender Ausdruck $\exists j \in \{1..i\} a_j = a_{sj}$ darstellen läßt.
Zustand 2: Ein Attribut der Anfragemenge $A = \{a_1,..,a_i\}$ist nicht in der Selektionsmenge eines DataSets $D.D_S = \{a_{s1},..,a_{sn}\}$vorhanden, was sich formal als folgender Ausdruck $\exists j \in \{1..i\} a_j \notin D.D_S$ darstellen läßt.
Zustand 3: Ein Attribut in der Selektionsmenge eines DataSets $D.D_S = \{a_{s1},..,a_{sn}\}$ist nicht in der Anfragemenge $A = \{a_1,..,a_i\}$vorhanden, was sich formal als folgender Ausdruck $\exists j \in \{1..n\} a_j \notin A$ darstellen läßt.

Der *Zustand 1* repräsentiert den sicherlich am erstrebenswertesten Zustand bei der Auffindung eines ähnlichen (oder des ähnlichsten) Falls, da eines der in die Attributmenge gewählten Attribute über den DataSet realisiert werden kann. Dieser Zustand läßt sich systemseitig über die Ermittlung der Kardinalität der Schnittmenge der Attribute der Anfragemenge und der Attribute der Definitionsmenge des DataSets umsetzen. Diese Berechnung soll im folgenden als $card(A \cap D.D_D)$ formuliert werden. Durch *card* wird eine Funktion bezeichnet, die die Kardinalität einer Menge berechnet. Das Argument der Funktion bezeichnet gerade die Schnittmengenbildung von den Attributen der Anfragemenge A und den Attributen der Definitionsmenge des DataSets D.

Der *Zustand 2* stellt die negative Situation dar, daß ein Attribut der Anfragemenge nicht durch den DataSet realisiert werden kann, wodurch ein Informationsverlust bezüglich des in der Anfragemenge spezifizierten Attributs entsteht. Daher wird dieser Zustand innerhalb der Ähnlichkeitsberechnung im Verhältnis zu den anderen beiden Zuständen auch sehr hart „bestraft“. Dies geschieht durch den Faktor h, der mit der Anzahl der nicht realisierten Attribute multipliziert wird. Die Anzahl der nicht in der Selektionsmenge eines DataSets vorhandenen Attribute ergibt sich durch $card(A - D.D_S)$, in der das Argument die Differenz der Attribute der Anfragemenge und der Selektionsmenge des DataSets darstellt.
Unter Berücksichtigung des "Bestrafungsfaktors" h ergibt sich der Ausdruck $h * card(A - D.D_S)$.

Durch den *Zustand 3* wird die Situation dargestellt, daß in der Selektionsmenge eines DataSets ein Attribut vorhanden ist, welches nicht in der Anfragemenge spezifiziert ist, also einen Informationsüberschuß repräsentiert. Da aber generell die Möglichkeit besteht, ein Attribut aus der Selektionsmenge eines DataSets zu entfernen, kann dieser Umstand als nicht so negativ wie der Zustand 2 gewertet werden. Dennoch stellt das Entfernen eines Attributs aus der Selektionsmenge eines DataSets Aufwand dar, weshalb hier der "Bestrafungsfaktor" g eingeführt wird, der mit der Kardinalität der Differenz der Attribute Selektionsmenge eines DataSet und der Attribute der Anfragemenge multipliziert wird. Dies wird durch den Ausdruck $g * card(D.D_S - A)$ berechnet.

Die Formel des Ähnlichkeitsmaßes $sim(A,D)$, welches in einem Intervall von [0;1] liegt, setzt sich nun aus den Berechnungsausdrücken der drei Zustände zusammen:

$$sim(A,D) = \frac{card(A \cap D.D_D)}{card(A \cap D.D_D) + \gamma * card(D.D_S - A) + \eta * card(A - D.D_S)}$$

Für die "Bestrafungsfaktoren" g, h sind folgende Werte gewählt worden: $g = 0{,}5$ und $h = 2$.

Nachweis für die Einhaltung der Intervallgrenzen des Ähnlichkeitsmaßes sim:
Einhaltung der unteren Intervallgrenze 0

Für die Erreichung der unteren Intervallgrenze müssen folgende Voraussetzungen gelten:

Die Kardinalität der Selektionsattributmenge eines DataSets sei unendlich groß, was impliziert, daß auch die Kardinalität der Definitionsmenge unendlich groß ist.
Die Kardinalität der Anfragemenge sei 1, woraus unter Einbeziehung der vorherigen Voraussetzung folgt: $card(A) \prec\prec card(D.D_s) \Rightarrow card(A) \prec\prec card(D.D_D)$.

Aus den zuvor genannten Voraussetzungen lassen sich folgende Erkenntnisse ableiten:
Der Grenzwert des Ausdrucks $card(D.D_s - A)$ läuft bei einer unendlich großen Selektionsmenge und einer Anfragemenge mit der Kardinalität 1 gegen unendlich: $\lim_{\substack{card(D.D_s)\to\infty \wedge \\ card(A)=1}} card(D.D_s - A) = \infty$

Der Wert der Kardinalität des Ausdrucks $card(A - D.D_s)$ hängt davon ab, ob die Anfragemenge *A* in der Selektionsmenge $D.D_S$ enthalten ist oder nicht. Falls *A* nicht in $D.D_S$ enthalten ist, so liefert der Ausdruck eine Kardinalität von 1, sonst 0.

Daraus ergibt sich für die gesamte Formel *sim* folgender Wert für die untere Intervallgrenze:

Fall 1: $A \notin D.D_S$:

$$\lim_{\substack{card(D.D_s)\to\infty \wedge \\ card(A)=1}} sim(A, D) = \frac{card(A \cap D.D_D)}{card(A \cap D.D_D) + g * card(D.D_S - A) + h * card(A - D.D_S)} = \frac{n}{n + \infty + 1} = \frac{1}{2 + \infty} = 0; n \in N_1$$

Fall 2: $A \in D.D_S$:

$$\lim_{\substack{card(D.D_s)\to\infty \wedge \\ card(A)=1}} sim(A, D) = \frac{card(A \cap D.D_D)}{card(A \cap D.D_D) + g * card(D.D_S - A) + h * card(A - D.D_S)} = \frac{n}{n + \infty + 0} = \frac{1}{1 + \infty} = 0; n \in N_1$$

Einhaltung der oberen Intervallgrenze 1:

Für die Erreichung der oberen Intervallgrenze muß folgende Voraussetzung gelten:
Die Anfragemenge *A* muß identisch zu der Selektionsattributmenge $D.D_S$ sein: $A = D.D_S$

Aus der zuvor genannten Voraussetzung lassen sich folgende Erkenntnisse ableiten:

Der Ausdruck $card(D.D_s - A)$ errechnet bei $A = D.D_S$ eine Kardinalität von 0.
Der Ausdruck $card(A - D.D_s)$ errechnet bei $A = D.D_S$ eine Kardinalität von 0.
Daraus ergibt sich für die gesamte Formel *sim* folgender Wert für die obere Intervallgrenze:

$$sim(A, D) = \frac{card(A \cap D.D_D)}{card(A \cap D.D_D) + g * card(D.D_S - A) + h * card(A - D.D_S)} = \frac{n}{n + 0 + 0} = 1; A = D.D_S \wedge n \in N_1$$

3.7.3.3.3 Integration der Methoden des fallbasierten Schließens

Nachfolgend werden die notwendigen Funktionen zur Realisierung der Vorauswahlphase, der eigentlichen Ähnlichkeitsmaßberechnung und die Pflegeroutinen für die Indexstruktur spezifiziert. Zu Beginn der Systemintegration soll jedoch die Verwendung der Methoden des fallbasierten Schließens innerhalb der Reportingplattform verdeutlicht werden. Wie bereits an vorheriger Stelle erwähnt, unterteilt sich die gesamte Fallbasis des Systems in zwei separate Fallbasen, wobei in der zunächst beschriebenen alle DataSets enthalten sind, die in Prozessen des Systems integriert sind. Diese werden hinsichtlich ihrer Attributmengen für eine geeignete Indexstruktur aufbereitet, um diese in späteren

Retrievephasen wieder aufzufinden. Die zweite Fallbasis hat die Aufgabe der Verwaltung aller Prozesse und wird in *Kapitel 4.2.2 Strukturelle Sicht* ausführlich erörtert.
Die konkrete Aufgabe dieser Methoden besteht zunächst also darin, für ein an der Benutzeroberfläche ausgelöstes Anfrageereignis, welches in Form einer gewählten Attributkombination formuliert wird, alle DataSets zu ermitteln, die den angegebenen Suchbedingungen genügen. Dabei wird für jeden gefundenen DataSet-Kandidaten das Ähnlichkeitsmaß berechnet, welches den Aufwand für die Anpassung des gefundenen DataSets an die durch die Attributkombination spezifizierte aktuelle Problemlösung widerspiegelt.

Da für den Zugriff auf die DataSets eine Indexstruktur aufgebaut bzw. verwendet wird, sind weitere Funktionen bereitzustellen, welche die Pflege dieser Strukturen realisieren. Eine Funktion dient der Aktualisierung der Indexstruktur. Diese muß gewährleisten, daß nach jeder Erzeugung eines DataSets auch ein Verweiseintrag in den Indextabellen vorgenommen wird. Im Gegensatz dazu kann auch eintreten, daß ein DataSet aus dem System gelöscht wird, worauf die für diesen DataSet vorhanden Einträge aus den Indexstrukturen gelöscht werden müssen.

3.7.3.3.3.1 Integration der Funktionen zur Berechnung der Ähnlichkeiten

Die Funktion zur Auffindung der DataSets über Attributkombinationen stellt die Funktion *Ermittle_DataSet_Kandidaten* dar, die als Aufrufparameter die gewählten Attribute und deren Verknüpfungsoperator erhält.

$$type\ Bestimme\_DataSet\_Kandidaten:\ Attribut-set \times SuchOp \rightarrow DataSetMap \times SimilarityMap$$
$$Attribut = char^{+}$$
$$SuchOp = AND \mid OR$$
$$DataSetMap = id \xrightarrow[m]{} DataSetID$$
$$id = N_1$$
$$DataSetID = N_1$$
$$SimilarityMap = id \xrightarrow[m]{} Similarity$$
$$Similarity = R_1^{+}$$

$$Bestimme\_DataSet\_Kandidaten(Anfrageattr, suchop)$$
$$Let\ DataSetID\_Menge = Ermittle\_DataSet\_Mengen(Anfrageattr)\ in$$
$$Let\ DataSetIDSel = Selektiere\_DataSets(attr, DataSetID\_Menge, suchop)\ in$$
$$Let\ Definitionsattr = Lese\_Definitionsattribute(DataSetIDSel)\ in$$
$$Let\ Selektionsattr = Lese\_Selektionsattribute(DataSetIDSel)\ in$$
$$Let\ SimilarityMap = Berechne\_Similarity\begin{pmatrix} DataSetIDSel, Definitionsattr, \\ Selektionsattr, Anfrageattr \end{pmatrix}\ in$$
$$(DataSetIDSel, SimilarityMap)$$

In einem ersten Schritt werden alle DataSet-Mengen aus der Tabelle *RPT_DataSet_Menge* ausgelesen, in denen jeweils mindestens eines der gewählten Attribute enthalten ist. Die so gewonnenen DataSet-Mengen werden darauf zu einer Menge vereinigt, so daß jeder DataSet nur einmal in der Gesamt-DataSet-Menge auftritt. Diese Vorgehensweise ist notwendig, da mehr als ein Attribut auf denselben DataSet (DataSetID) zeigen kann bzw. in einem DataSet mehr als eines der gewählten Attribute enthalten ist.

$$type\ Lese\_DataSet\_Menge: \ Attribut-set \times DataSetID-set \rightarrow DataSetID-set$$

$$Lese\_DataSet\_Menge(attr, datasetidvereinigt) \triangleq$$
$$\quad if\ card(attr) \succ 0$$
$$\quad then\ Let\ a = a \in attr\ be\ such\ that\ a \in dom\ RPT\_Template\_Menge\ in$$
$$\qquad if\ a \neq \{\ \}$$
$$\qquad then\ Lese\_Template\_Menge(attr - \{a\}, datasetidvereinigt \cup RPT\_DataSet\_Menge(a))$$
$$\qquad else\ datasetidvereinigt$$
$$\quad else\ datasetidvereinigt$$

Durch die Funktion *Lese_DataSet_Menge* wurden zunächst alle DataSets ermittelt, die mindestens eines der gewählten Attribute in der Definitionsmenge enthalten. Da aber das Suchergebnis der zuvor bestimmten Suchbedingung genügen muß, wird in einem weiteren Schritt die Menge der ausgelesen DataSets auf die DataSet-Menge reduziert, deren einzelne DataSet-Elemente jeweils die Suchbedingung erfüllen. Diese Aufgabe kommt der Funktion *Selektiere_DataSets* zu, die wiederum je nach der gewählten Suchbedingung die Hilfsfunktionen *Konjunktive_Suche* oder *Disjunktive_Suche* zur Ermittlung der zulässigen DataSet-Menge aufruft.

$$type\ Selektiere\_Templates: Attribut-set \times TemplateID-set \times SuchOp \rightarrow TemplateID-set$$

$$Selektiere\_Templates(attr, tempidmenge, suchop) \triangleq$$
$$\quad if\ is-AND(suchop)$$
$$\quad then\ Kunjuntive\_Suche(attr, tempidmenge, \{\ \})$$
$$\quad else\ Disjunktive\_Suche(attr, tempidmenge, \{\ \})$$

$$type\ Konjunktive\_Suche: \ Attribut-set \times DataSetID-set \times DataSetID-set \rightarrow DataSetID-set$$

$$Konjunktive\_Suche(attr, datasetidmenge, datasetidmengesel) \triangleq$$
$$\quad if\ card(datasetidmenge) \succ 0$$
$$\quad then\ Let\ datasetid = did \in datasetidmenge\ be\ such\ that\ attr \subseteq RPT\_Attribut\_Menge(did)\ in$$
$$\qquad if\ datasetid \neq \{\ \}$$
$$\qquad then\ Konjunktive\_Suche(attr, datasetidmenge - \{datasetid\}, datasetidmengesel \cup \{datasetid\})$$
$$\qquad else\ datasetidmengesel$$
$$\quad else\ datasetidmengesel$$

$$type\ Disjunktive\_Suche: \ Attribut-set \times DataSetID-set \times DataSetID-set \rightarrow DataSetID-set$$

$$Disjunktive\_Suche(attr, datasetidmenge, datasetidmengesel) \triangleq$$
$$\quad if\ card(datasetidmenge) \succ 0$$
$$\quad then\ Let\ datasetid = did \in datasetidmenge\ be\ such\ that\ attr \cap RPT\_Attribut\_Menge(did) \neq \{\ \}\ in$$
$$\qquad if\ datasetid \neq \{\ \}$$
$$\qquad then\ Disjunktive\_Suche(attr, datasetidmenge - \{datasetid\}, tempidmengesel \cup \{datasetid\})$$
$$\qquad else\ datasetidmengesel$$
$$\quad else\ datasetidmengesel$$

Für jeden der DataSets, der über die Funktion *Selektiere_DataSets* durch die einschränkende Suchbedingung ermittelt worden ist, werden nun aus der Tabelle *RPT_Attribute_Menge* die Selektions- und Definitionsattribute ausgelesen, um diese für die nachfolgende Ähnlichkeitsmaßberechnung *Similarity* bereitzustellen.

$$type\ Lese\_Definitionsattribute: \ DataSetID-set \rightarrow AttributeMap$$
$$AttributeMap = id \xrightarrow[m]{} Attribut-set$$

$$Lese\_Definitionsattribute(datasetidmenge)\underline{\Delta}$$
$$\left\{datasetmap \cup \left\{\begin{matrix} card\ dom\ datasetmap+1 \mapsto RPT\_Attribut\_Menge(d).Definitionsattribute \mid \\ d \in datasetidmenge \wedge d \in dom\ RPT\_Attribut\_Menge \end{matrix}\right\}\right\}$$

$$type\ Lese\_Selektionsattribute: \ DataSetID-set \rightarrow AttributeMap$$
$$AttributeMap = id \xrightarrow[m]{} Attribut-set$$

$$Lese\_Selektionsattribute(datasetidmenge)\underline{\Delta}$$
$$\left\{datasetmap \cup \left\{\begin{matrix} card\ dom\ datasetmap+1 \mapsto RPT\_Attribut\_Menge(d).Selektionsattribute \mid \\ d \in datasetidmenge \wedge d \in dom\ RPT\_Attribut\_Menge \end{matrix}\right\}\right\}$$

Die Funktion *Berechne_Similarity* berechnet die Ähnlichkeit von der vom Anwender gewählten Attributmenge zu den Attributen der über die Kandidatenmenge spezifizierten DataSets und gibt für jeden DataSet dieser Menge die berechnete Ähnlichkeit zurück. Als Aufrufparameter werden die vom Anwender bestimmten Anfrageattribute (*AnfrageAttribute*), die für jeden gefundenen DataSet aus den Funktionen *Lese_Definitionsattribute* und *Lese_Selektionsattribute* ermittelten Definitions- (*DefinitionsattrMap*) und Selektionsattribute (*SelektionsattrMap*) an die Funktion übergeben. Innerhalb der Berechnung wird nun die zuvor beschriebene Formel *Sim(A,D)* umgesetzt. Für jeden gefundenen DataSet wird der Wert des Ähnlichkeitsmaßes errechnet und in den Array (*SimilarityMap*) gespeichert, welcher auch den Rückgabewert der Funktion repräsentiert.

$$type\ Berechne\_Similarity: \ AnfrageAttribute \times DefinitionsattrMap \times SelektionsattrMap \rightarrow SimilarityMap$$
$$AnfrageAttribute = Attribut-set$$
$$DefinitionsattrMap = id \xrightarrow[m]{} Attribut-set$$
$$SelektionsattrMap = id \xrightarrow[m]{} Attribut-set$$
$$SimilarityMap = id \xrightarrow[m]{} R_0^+$$

$$Berechne\_Similarity(anfrageattr, definitionsattrmap, selektionsattrmap)$$
$$\left\{\begin{matrix} similaritymap \cup \left\{card\ (dom\ similaritymap)+1 \mapsto sim \middle| d \in dom\ definitionsattrmap \wedge \right. \\ Let\ EnthaltenInSel = card\ (anfrageattr \cap definitionsattrmap(d))\ in \\ Let\ NotInSel = card\ (selektionsattrmap(d) - anfrageattr)\ in \\ Let\ NotInDef = card\ (anfrageattr - selektionsattrmap(d))\ in \\ Let\ sim = (EnthaltenInSel / (EnthaltenInSel + g * NotInSel + h * NotInDef))\ in \\ sim \end{matrix}\right\}$$

3.7.3.3.3.2 Integration der Funktionen zum Aufbau der Indexstruktur

Zunächst wird eine Funktion vorgestellt, deren Aufgabe darin besteht, einen über die Benutzeroberfläche vom Anwender gespeicherten DataSet, der bezüglich der verwendeten fallbasierten Methodologie einen neuen Fall darstellt, in die Indexstruktur zu integrieren. Dadurch wird gewährleistet, daß dieser Fall in einer späteren Retrievephase zur Lösung eines aktuellen Problems wieder aufgefunden wird. Daher muß eine Funktion bereitgestellt werden, die beliebig nach jeder Speicherung aufgerufen werden kann. Die Aufrufparameter dieser Funktion setzen sich aus der Nummer des Speicherhandles und dem Handle der Menge (*DataSets*) zusammen. Dabei repräsentiert

das Speicherhandle den Wert, unter dem die aktuell übergebene Menge im Archiv abgespeichert worden ist. Das Mengenhandle ist daher von Bedeutung, da aus diesem die Selektions- und Definitionsattribute des DataSets zu entnehmen sind.
Durch die Funktion *Aktualisiere_DataSet_Zuordnung* wird die vorhergehend erläuterte Problematik gelöst.

$$type\ Aktualisiere\_DataSet\_Zuordnung: \ DataSetID \times Mengenhandle \times RPT\_Attribut\_Menge \times RPT\_DataSet\_Menge \xrightarrow{\sim} (RPT\_Attribut\_Menge \times RPT\_DataSet\_Menge)$$

$$DataSetID = N_1$$
$$Mengenhandle = N_1$$

$$Aktualisiere\_DataSet\_Zuordnung(dataid, mhandle, rpt\_attribut\_menge, rpt\_dataset\_menge) \underline{\Delta}$$
$$Let\ rpt\_attribut\_menge = Speichere\_DataSet(dataid, defattrmenge, selattrmenge, rpt\_attribut\_menge)\ in$$
$$Let\ rpt\_dataset\_menge = Speichere\_Attribute\_Zurodnungen(dataid, defattrmenge, rpt\_dataset\_menge)\ in$$
$$(rpt\_attribut\_menge, rpt\_dataset\_menge)$$

Durch die Funktion *Speichere_DataSet_Zuordnung* wird die Indextabelle *RPT_Attribut_Menge* um den Eintrag des DataSets erweitert. Der DataSet wird unter der *DataSetID* mit den Definitions- und Selektionsattributen als neuer Eintrag in die Tabelle gespeichert.

$$type\ Speichere\_DataSet\_Zuordnung: \ DataSetID \times Definitionsattribute \times Selektionsattribute \times RPT\_Attribut\_Menge \xrightarrow{\sim} RPT\_Attribut\_Menge$$

$$Definitionsattribute = Attribut - set$$
$$Selektionsattribute = Attribut - set$$

$$Speichere\_DataSet\_Zuordnung(dataid, defattrmenge, selattrmenge, rpt\_attribut\_menge) \underline{\Delta}$$
$$Let\ rpt\_attribut\_menge = rpt\_attribut\_menge \cup \{dataid \mapsto (defattrmenge, selattrmenge)\}\ in$$
$$(rpt\_attribut\_menge)$$

Die Funktion *Speichere_Attribut_Zuordnung* aktualisiert die Indextabelle *RPT_DataSet_Menge*. Für jedes Attribut der Menge der Definitionsattribute des zu speichernden DataSets wird ein Eintrag in der Tabelle vorgenommen. Dabei muß die Aktualisierung allerdings in zwei Fälle unterschieden werden:

Fall 1: Für ein Attribut der Definitionsmenge des einzutragenden DataSets existiert noch kein Eintrag. Es wurde bisher keine Menge modelliert, die dieses Attribut in seiner Definitionsmenge beinhaltet. Daher muß ein komplett neuer Eintrag für den DataSet unter dem noch nicht gespeicherten Definitionsattribut vorgenommen werden.

Fall 2: Für das einzutragende Definitionsattribut existiert bereits ein Eintrag in der Tabelle, wodurch nur die Notwendigkeit besteht, diesen Eintrag um die neue *DataSetID* zu erweitern.

$$type\ Speichere\_Attribut\_Zuordnung:\ DataSetID \times Definitionsattribute \times RPT\_DataSet\_Menge \xrightarrow{\sim} RPT\_DataSet\_Menge$$

$$Speichere\_Attribut\_Zuordnung(dataid, defattrmenge, rpt\_dataset\_menge) \underline{\Delta}$$

$$if\ card(defattrmenge) \succ 0$$

$$then\ Let\ attr = a \in defattrmenge\ in$$

$$if\ attr \in dom\ rpt\_dataset\_menge$$

$$then\ Let\ datasetmenge = rpt\_dataset\_menge(attr)\ in$$

$$Let\ datasetmenge_{neu} = datasetmenge \cup \{dataid\}\ in$$

$$Speichere\_Attribut\_Zuordnung\begin{pmatrix} dataid, defattrmenge - \{attr\}, rpt\_dataset\_menge + \\ \{attr \mapsto datasetmenge_{neu}\} \end{pmatrix}$$

$$else\ Speichere\_Attribut\_Zuordnung\begin{pmatrix} dataid, defattrmenge - \{attr\}, rpt\_dataset\_menge \cup \\ \{attr \mapsto \{dataid\}\} \end{pmatrix}$$

$$else\ (rpt\_dataset\_menge)$$

Da über die Anwendungsoberfläche aber auch die Möglichkeit besteht, einen gespeicherten DataSet, nach einer Überprüfung hinsichtlich bestehender Referenzen, zu löschen, muß eine weitere Funktion bereitgestellt werden. Diese entfernt für den zu löschenden DataSet die vorgenommenen Einträge wieder aus der Indexstruktur. Es dürfen jedoch nur DataSets gelöscht werden, für die keine Referenz mehr besteht und die auch in keinem anderen Geschäftsprozeß enthalten sind.

$$type\ Lösche\_DataSet:\ DataSetID \xrightarrow{\sim} (RPT\_Attribut\_Menge \times RPT\_DataSet\_Menge)$$

$$Lösche\_DataSet(dataid) \underline{\Delta}$$

$$Let\ defattributmenge = Lese\_Definitionsattribute\{dataid\}\ in$$

$$Let\ RPT\_Attribut\_Menge = RPT\_Attribut\_Menge - \{did\}\ in$$

$$Let\ RPT\_DataSet\_Menge = Aktualisiere\_Attributeintr\ddot{a}ge(defattributmenge, dataid)\ in$$

$$(RPT\_Attribut\_Menge, RPT\_DataSet\_Menge)$$

$$type\ Aktualisiere\_Attributeintr\ddot{a}ge:\ Attribut - set \times DataSetID \xrightarrow{\sim} RPT\_DataSet\_Menge$$

$$Aktualisiere\_Attributeintr\ddot{a}ge(attrmenge, dataid) \underline{\Delta}$$

$$if\ card(attrmenge) \succ 0$$

$$then\ Let\ attr = a \in attrmenge\ in$$

$$Let\ datasetmenge = RPT\_DataSet\_Menge(attr)\ in$$

$$Let\ datasetmenge_{neu} = datasetmenge - \{dataid\}\ in$$

$$if\ datasetmenge_{neu} \neq \{\ \}$$

$$then\ Let\ RPT\_DataSet\_Menge = RPT\_DataSet\_Menge + \{attr \mapsto datasetmenge_{neu}\}\ in$$

$$Aktualisiere\_Attributeintr\ddot{a}ge(attrmenge - \{attr\}, dataid)$$

$$else\ Let\ RPT\_DataSet\_Menge = RPT\_DataSet\_Menge - \{attr\}\ in$$

$$Aktualisiere\_Attributeintr\ddot{a}ge(attrmenge - \{attr\}, dataid)$$

$$else\ (RPT\_DataSet\_Menge)$$

3.7.4 Zusammenfassung

Aufgrund der vielfältigen und komplexen im System gespeicherten Informationen hat der Nutzer keine vollständige Übersicht, ob bereits systemseitig vorhandene Lösungen zu aktuellen Problemstellungen verwendet werden können. Deshalb wurde eine Technologie entwickelt, die den Anwender beim

Informationsretrieval unterstützt. Dazu werden durch ihn formulierte Problemspezifikatoren in Form von Attributen, DataSets und Prozessen genutzt. Durch die realisierten Mechanismen des fallbasierten Schließens besteht die Möglichkeit, über die im Repository hinterlegten Attribute alle DataSets zu finden, in denen diese Attribute enthalten sind. Der Vorteil der Auffindung der DataSets, welche die vom Anwender spezifizierten Attribute enthalten, besteht darin, daß so auch Lösungen für schwach strukturierte bzw. sehr allgemein formulierte Problemstellungen gefunden werden können. Der Anwender kann hierdurch auch bei Problemstellungen, die nicht in letzter Konsequenz vollständig durchdacht worden sind, mittels Systemvorschlägen, in Form der den Informationsbedarf realisierenden DataSets, zu dem konkreten Lösungsprozeß geführt werden.

Eine andere wichtige Zielsetzung wird mit der Wiederverwendung von lösungsrelevanten DataSets verfolgt, so daß zuvor modellierte DataSets, die mit einer festdefinierten SQL-Anfrage oder einem SQL-Template korrespondieren, nicht wiederholend erstellt werden müssen.

Weiterhin wurden Routinen bereitgestellt, die eine Aktualisierung der Indexstruktur erlauben, so daß sich diese bezüglich der aktuell gespeicherten DataSets immer in einem konsistenten Zustand befindet. Innerhalb der Retrievephase wurden Verfahren implementiert, die eine Vorauswahl der Lösungskandidaten und die darauf angewendeten Ähnlichkeitsberechnungen realisieren. Die Anpassung einer Lösung kann über das Mengentool erfolgen, in dem weitere Attribute der Definitionsmenge selektiert bzw. bereits selektierte Attribute wieder deselektiert werden.

Für die Weiterentwicklung dieses Konzepts wären sicherlich noch Erweiterungen im Bereich der Suche denkbar, so daß nicht nur allgemein über Attribute gesucht werden kann, sondern zusätzlich auch über wählbare Attributausprägungen. Dies würde zur Folge haben, daß die Indexstruktur nicht nur über die Attribute der DataSets gelegt werden muß, sondern auch über die Filterausdrücke aus den SQL-Statements, über welche die Informationen eines DataSets realisiert werden. Dadurch könnte auch eine ganz neue Form der Indexierung erforderlich sein, wie beispielsweise ein *kd-Baum* [Wess95], in dessen Knoten wiederum ähnliche DataSets gruppiert werden können. Das Navigieren durch die inneren Knoten würde dann über die Attributausprägungen des jeweiligen Attributs laufen, wobei die Auswahl danach erfolgt, ob die Ausprägung des gewählten Attributs größer oder kleiner als die des Teilknotenattributs ist. Außerdem könnte noch für den generellen Fall, daß zu einer Attributanfrage keine vormodellierte Lösung im System vorliegt, ein Automatismus entwickelt werden, der einen temporären DataSet, gemäß der gewählten Attribute und den darauf bezogenen Filtern (Attributausprägungen), im Mengentool bereitstellt, worauf der Anwender zumindest durch die Ausführung dieser Menge testen kann, ob die aus der Datenbank gewonnenen Informationen für dessen Lösung nützlich sind.

3.8 ZUSAMMENFASSUNG DER ENTWICKLUNG DER REPORTINGPLATTFORM

Zunächst wurden Basismechanismen zur Bereitstellung flexibler Informationsabfragen in Form der DataSets entwickelt, die benutzerseitig Anpassungen erlauben, ohne daß der Anwender direkt mit SQL in Berührung kommt. Die während der Modellierungsphase erstellten Metadaten erlauben es, mittels eines entwickelten SQL-Generators zur Laufzeit SQL-Statements zu erzeugen. Aufgrund der Anforderung nach komplexeren SQL-Statements, als diese mittels des SQL-Generators erzeugbar sind, wurde die Möglichkeit geschaffen, den DataSets festdefinierte SQL-Statements zuzuweisen. Hierdurch wird der gesamte SQL-Umfang nutzbar.

Durch die Verwendung festdefinierter SQL-Statements konnten komplexe Domains modelliert werden, was jedoch zu einem sehr hohen Wartungsaufwand führte und mangelnde Flexibilität bedeutet. Des weiteren bestand der Bedarf nach einer systemseitig unterstützten Benutzerführung, die bislang nur durch Briefing Book Funktionalitäten gegeben war und deshalb zur Entscheidungsunterstützung bei unbekannten Problemstellungen nicht ausreichte. Aus diesem Grunde wurde mit dem Design Pattern Template eine Technologie genutzt, um einerseits einen verbesserten Grad an Wiederverwendung bestehenden SQL-Codes zu erhalten und das System wartbar zu halten. Andererseits dient das Templatekonzept auch zur Spezifikation von Prozessen. Hierdurch werden Adaptionen der Prozesse möglich, so daß sie zur systemgestützten, erfahrungsbasierten Entscheidungsunterstützung dienen können.

Die Parametrisierung der virtuellen SQL-Templates und der Prozeßtemplates hat sich als gutes Konzept zur überschaubaren Anpassung der einzelnen Bausteine erwiesen. Anpassungen zur Laufzeit werden bei den SQL-Templates mittels des SQL-Generators auf korrekte Syntax hin überprüft. Die Mechanismen zum Zusammenfügen der einzelnen Bestandteile eines SQL-Templates zu einem korrekten SQL-Statement werden durch die Metadaten im Repository sichergestellt. Komplexer stellte sich diese Problematik bei der Adaption der verschiedenen Prozeßtemplates dar, da hierfür nicht sämtliche Kombinationsmöglichkeiten im Vorwege zu überprüfen sind. Aus diesem Grunde wurden Konsistenzbedingungen spezifiziert, die bei Adaptionen zur Laufzeit gewährleisten, daß ein Prozeß von einem konsistenten Zustand wieder in einen konsistenten Zustand übergeht.

Damit insbesondere die Prozeßtemplates zur Laufzeit adaptiert werden können und somit die Ausnahmebehandlung quasi zur Regel werden kann, wurden sie als Komponenten im Sinne von Componentware modelliert. Hierdurch kann die hohe Flexibilität bei der Prozeßsteuerung erzielt werden, die den Austausch einzelner Komponenten per Point & Click zur Laufzeit ermöglicht. Integritätsbedingungen wachen u.a. darüber, daß der Datenfluß in Form von DataSets von einer Prozeßkomponente zur nächsten sichergestellt ist. Bei Konfliktsituationen wird der Anwender in die Auflösung mit einbezogen, so daß ein robustes Gesamtsystem entsteht.

Zur Domainmodellierung der Metadaten sowie der SQL- und Prozeßtemplates wurden Werkzeuge entwickelt, die den Modellierer unterstützen und insbesondere von Routineaufgaben entlasten. Aufgrund der allgemein gehaltenen Modellierungswerkzeuge und des Repository ist die Reportingplattform im Gegensatz zu anderen entscheidungsunterstützenden Systemen in unterschiedlichen Domains einsetzbar, da ihre Funktionalitäten auf den während der Modellierungsphase erstellten Metadaten, DataSets, SQL- und Prozeßtemplates aufbauen.

Da der Anwender keine Übersicht zu der Vielzahl modellierter Lösungsbausteine des Systems haben kann, wurden Mechanismen des fallbasierten Schließens verwendet, ihn bei der Auffindung von Attributen, DataSets und Prozessen zu unterstützen. Während der Modellierungsphase wird der Modellierer bei der Erstellung der SQL-Templates dahingehend unterstützt, daß er bereits modellierte und zum aktuellen SQL-Template ähnliche Templates suchen kann.

Der Anwender der Reportingplattform spezifiziert im Rahmen von Sitzungen zur Entscheidungsunterstützung die aktuelle Problemstellung in Form von Attributen, DataSets und Prozessen, die mit Namen der aktuellen Domainterminologie versehen sind. Aufgrund dieser Spezifikation ermittelt die Reportingplattform eine Menge ähnlicher Lösungen zur aktuellen Problemstellung, die der Anwender anschließend adaptieren und bei Bedarf zur Lösung späterer Problemfelder speichern kann.

Gerade die Funktionalitäten über Mechanismen des fallbasierten Schließens adäquate Lösungsbausteine zu finden, ermöglichen den Einsatz der Reportingplattform in komplexen Entscheidungssituationen. Ohne die systemseitige Unterstützung beim Wiederauffinden bereits vorhandener Lösungen wäre die Reportingplattform zwar flexibler einsetzbar als z.B. EIS- oder Workflowmanagementsysteme, allerdings mit denselben Nachteilen bzgl. der notwendigen Übersicht zu vorhandenen Lösungen behaftet.

Im folgenden *Kapitel 4 Implementierung* werden die zuvor entwickelten Mechanismen implementiert und am Beispiel des Werkzeugs für die Entscheidungsunterstützung sowie einzelnen Modellierungstools dargestellt.

4 IMPLEMENTIERUNG

4.1 ÜBERSICHT

In diesem Kapitel wird insbesondere die Entwicklung des DSS-Werkzeugs zur Entscheidungsunterstützung dargestellt (siehe *Kapitel 4.2 Das DSS-Werkzeug*). Bei der Entwicklung des Tools wurden die Konzepte des fallbasierten Schließens (siehe *Kapitel 2.6 Fallbasiertes Schliessen*) dahingehend umgesetzt, daß die Funktionalitäten des DSS-Werkzeugs sämtliche Phasen unterstützen (siehe *Kapitel 2.6.2.1 Prozeßmodell des fallbasierten Schließens*).

Die in *Kapitel 3.7 Konzept zum Wiederauffinden ähnlicher Problembeschreibungen* entwickelte Methodologie, die zunächst auf den Bereich des Lokalisierens von DataSets und SQL-Templates ausgerichtet war, wird nun hinsichtlich des Wiederauffindens von Attributen, DataSets und (Entscheidungs- oder) Geschäftsprozessen erweitert, so daß der Anwender über seine Benutzerschnittstelle bei der Entscheidungsfindung durch ein entsprechendes Angebot an im System vorhandenen und angebotenen Lösungen unterstützt wird.

Die Entwicklung der ergonomischen Benutzeroberfläche wurde nach dem MUSE-Konzept sowie der Werkzeug-Material-Metapher durchgeführt (siehe *Kapitel 2.1.1 MUSE - Method for User Interface Engineering* und *Kapitel 2.1.2 Die Werkzeug-Material-Metapher*).
Nach einem kurzen Überblick zur konzeptuellen Sicht des DSS-Werkzeugs (siehe *Kapitel 4.2.1 Konzeptuelle Sicht des DSS-Tools*) wird dessen strukturelle Sicht anhand von Dialognetzen beschrieben (siehe *Kapitel 4.2.2 Strukturelle Sicht des DSS-Tools*). Anschließend wird eine Einführung in die Struktur und die Funktionalitäten der Prozeßfallbasis gegeben (siehe *Kapitel 4.2.2.1 Die Prozeßfallbasis*). Aufgrund der Verwendung eines flexiblen Speichermediums für das effiziente Wiederauffinden einzelner Fälle (siehe *Kapitel 4.4.3 Das Archiv*) erfolgt die Abgrenzung der neuen Datenstruktur zur Tabellenstruktur des Repositorys.

Kapitel 4.2.2.2 Dialog zur Problemspezifikation gibt einen Einblick in die Spezifikation von Problemstellungen unter Verwendung des *Glossars*, als Schnittstelle zur Fallbasis. Die Suche nach in der Fallbasis enthaltenen Lösungen mit der Berechnung des jeweiligen Ähnlichkeitsmaßes wird in *Kapitel 4.2.2.3 Dialog zur Spezifikation der Retrievephase* dargestellt. Die daran anschließenden Phasen (Reuse-, Revise- und Retainphasen) zur Adaption und ggf. Übernahme der neuen Lösung in die Fallbasis werden in *Kapitel 4.2.2.4 Dialog zur Bearbeitung, Beurteilung und Übernahme einer Lösung* erläutert.

In *Kapitel 4.3 Entwicklung der Benutzerschnittstelle* wird ein Überblick zu den entwickelten Werkzeugen der Benutzerschnittstelle gegeben. *Kapitel 4.3.1.1 SQL-Templatemodellierung* zeigt die Umsetzung des in *Kapitel 3.5 SQL-Templates* entwickelten Konzepts dahingehend, daß eine Auswahl an Masken zur Modellierung von SQL-Templates beschrieben werden. Die Umsetzung des in *Kapitel 3.6 Prozesstemplates* entwickelten Konzepts zur flexiblen Modellierung von Prozessen in Form von Templates wird anhand einer kurzen Darstellung des Prozeßmodellierungstools in *Kapitel 4.3.1.2 Der Prozeßeditor* beschrieben.

Anschließend wird anhand einer beispielhaften Beschreibung eines Sitzungsszenarios mit dem DSS-Werkzeug das Lösungsangebot in Form des Glossars (siehe *Kapitel 4.3.2.1 Das Informationsangebot*) und die Suche nach Lösungen (siehe *Kapitel 4.3.2.2 Informationssuche*) erläutert.
Die darauffolgende Adaption, den Durchlauf des Entscheidungsprozesses sowie die Speicherung der neuen Lösung in der Fallbasis werden in *Kapitel 4.3.2.3 Informationsadaption* dargestellt. Hierbei wird deutlich, wie sich die Modellierung der einzelnen Prozeßbausteine in Form von Komponenten (siehe *Kapitel 3.3 Verwendung von Componentwarekonzepten*) und deren Adaptierbarkeit (siehe *Kapitel 3.6.3 Verknüpfen von Prozeßkomponenten*) als flexibler Mechanismus bei der Prozeßadaption und der Behandlung von Ausnahmefällen bewährt. Die Anpassungen der Prozesse erfolgen seitens des Anwenders durch einfache Drag & Drop Mechanismen.

Einen Überblick zu den zusätzlich entwickelten Werkzeugen der Reportingplattform, die u.a. die Umsetzung einer EIS-Komponente widerspiegeln, gibt *Kapitel 4.3.3 Werkzeuge der Analysekomponente*.

Die Beschreibungen der komponentenbasierten Systemarchitektur des Frameworks und der funktionalen Architektur schließen sich in *Kapitel 4.4 Architektursichten der Reportingplattform* an. *Kapitel 4.4.3 Das Archiv* erläutert das Speicherungskonzept für komplexe Objekte der Reporting-plattform. Die Informationen des Repositorys werden ebenfalls über einen Serialisierungsmechanismus in das Archiv übertragen, so daß seitens der Werkzeuge lediglich auf ein Speichermedium zugegriffen werden muß.
Die Erfahrungen bei der Einführung der Reportingplattform werden in *Kapitel 4.5 Anwendungserfahrungen* erläutert.

4.2 DAS DSS-WERKZEUG

Im folgenden wird die Spezifikation des Werkzeugs für die Entscheidungsunterstützung (Decision Support System – DSS) beschrieben. Das Design erfolgte nach dem MUSE-Sichtenkonzept zur Entwicklung von Benutzerschnittstellen. Dabei wurden die komplexen Teilgebiete des Entwicklungs-vorgangs in überschaubare Teilaufgaben zerlegt. Neben der Gestaltung der Dialogstruktur der Anwendung werden ebenfalls die verwendeten Interaktionselemente der Benutzeroberfläche erläutert (siehe [Hamp97]). Das Werkzeugkonzept wurde analog der Werkzeug-Material-Metapher entwickelt.

Bezüglich der Möglichkeit, ähnliche Applikationen fallbasierter Systeme als Ausgangspunkt bzw. "Ideengeber" zur Gestaltung einer ergonomischen Benutzerschnittstelle zu nutzen, ist anzumerken, daß aufgrund des speziellen Anwendungsgebiets (Bereitstellung vormodellierter Informationen in Form von Prozessen und DataSets) keine verwandten Systeme in der gängigen Literatur vorzufinden sind. Konzeptuell arbeiten diese Anwendungssysteme nach dem gleichen Verfahren, stellen aber hinsichtlich eines Anwendungsgebiets in der Regel Individuallösungen dar. Darüber hinaus konnten Anregungen zu den Adaptionsmechanismen dem fallbasierten System INRECA [INRE95] entnommen werden.

4.2.1 Konzeptuelle Sicht des DSS-Tools

Die Werkzeugfunktionen des zu entwickelnden Systems ergeben sich aus den in den vorigen Kapiteln erläuterten Anforderungen zur Verknüpfung der DataSets und Prozesse mit Mechanismen des fallbasierten Schließens und einer ergonomischen Benutzeroberfläche.

Anwendungs-Funktionen	Steuer-funktionen	Adaptier-funktionen	Meta-funktionen
Fallbeispiele • Suchen • Erzeugen • Ändern • Löschen • Testen • Ausschneiden • Kopieren • Einfügen **Artefaktsuche** • Volltextsuche	**Retrievephase** • Beginnen • vorzeitig beenden • neu initiieren **Automatische Übernahme** • aller Lösungen • der besten Lösung **Papierkorb** • Wiederherstellen gelöschter Fallbeispiele	**Fensterfunktionen** • Größenänderung (gegeben) **Darstellungsform** • Schrifttyp • Schriftgröße • Schriftfarbe • Markierungsfarbe gefundener Lösungen	**<keine>**

Tabelle 29: Funktionalitäten der Benutzerschnittstelle

Dabei beziehen sich die *Anwendungsfunktionen* auf die notwendige Unterstützung in den einzelnen Phasen des fallbasierten Schließens, wobei die Funktionen *Ausschneiden, Kopieren* und *Einfügen* eine Alternative zu den ebenso vorhandenen Drag & Drop-Mechanismen der Anwendung darstellen.

Zu den *Steuerfunktionen* gehört neben den notwendigen Regelungsmöglichkeiten zur Retrievephase auch eine komfortable Methode zur automatischen Zusammenstellung gefundener Lösungen aus der Fallbasis. Darüber hinaus werden gelöschte Fallbeispiele zunächst nicht physisch entfernt, sondern zeitgemäß in einen Papierkorb verschoben, um sie bei Bedarf wiederherstellen zu können.

Die *Adaptierfunktionen* (nicht zu verwechseln mit dem Begriff der Adaption im fallbasierten Schließen) erlauben ein gewisses Maß der Anpassung der Benutzungsoberfläche an die persönlichen Bedürfnisse des Anwenders. Gegeben sind dabei die grundlegenden Fenstermechanismen (minimieren, maximieren, verschieben usw.) durch die verwendete Systemplattform. Darüber hinaus verfügt das Werkzeug über die Möglichkeiten zur Beeinflussung der Informationsdarstellung durch Schrifttyp, Schriftgröße und Schriftfarbe. Zunächst wurden keine *Metafunktionen* implementiert; ein aussagekräftiges Hilfesystem, kontextbezogene Hilfe und eine Tutoringkomponente wären aber zur Unterstützung der Verwendung einzelner Funktionen sinnvoll.

4.2.2 Strukturelle Sicht des DSS-Tools

Die Spezifikation der Dialogstruktur des DSS-Werkzeugs erfolgt durch ein Dialognetz [BuJa96], da hierdurch in geeigneter Weise die Parallelität der Dialoge direkt ausgedrückt werden kann. Hierbei wird lediglich eine Grobstruktur der Benutzerschnittstelle beschrieben. Eine Übersicht zur Darstellungsform und Bedeutung der verwendeten Dialognetzkomponenten ist im *Anhang Kapitel 11.3 Dialognetze zur Ablaufbeschreibung von Benutzerdialogen* gegeben.

Das in Abbildung 41 dargestellte Dialognetz (mit unspezifizierten Beschriftungen) stellt die grobe Dialogstruktur der Benutzerschnittstelle des DSS-Werkzeugs dar. Dabei wurden zugunsten einer besseren Übersicht die Dialoge der einzelnen Phasen als komplexe Stellen dargestellt, deren detailliertere Beschreibung noch folgt.

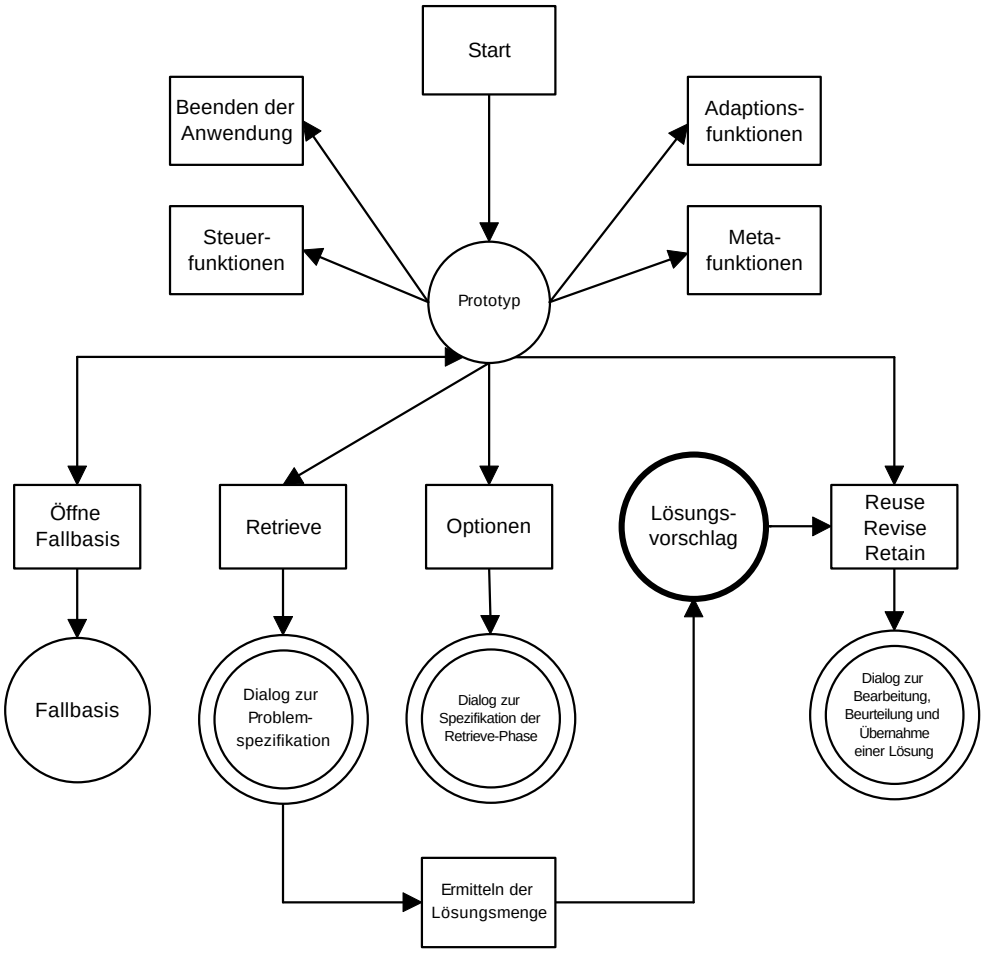


Abbildung 41: Dialogstruktur der Benutzerschnittstelle

Zu Beginn einer Sitzung wird die Fallbasis geöffnet. Durch die bidirektionale Flußrelation wird verdeutlicht, daß die Fallbasis stets sichtbar bleibt, auch wenn andere Stellen des Dialognetzes markiert werden. Wie aus der Abbildung bereits ersichtlich wird, kann der normale Verlauf im Prozeßmodell

des fallbasierten Schließens vollständig nachvollzogen werden. Um eine größere Flexibilität der Dialogfolge zu erreichen, werden aber die einzelnen Phasen nicht durch das System fest vorgegeben, sondern können in beliebiger Reihenfolge durchlaufen werden.

Diese Flexibilität wird aus ergonomischen Gründen gefordert, ergibt sich zudem auch aus der Tatsache heraus, daß bereits vor einem Retrieval bekannt sein kann, daß es keine geeignete Lösung in der Fallbasis gibt. Folglich ist die Suche nach einer Lösung über eine Problemspezifikation überflüssig. Daher wurde ein direkter Einstieg in die Reusephase eingerichtet, in der der Benutzer *sofort* eine neue Lösung mittels der Bearbeitungsmöglichkeiten dieser Phase erzeugen kann. Es ist aber jederzeit möglich, während der eigentlichen Reusephase neue Teillösungen zu suchen, die wiederum innerhalb der aktuellen Bearbeitung verwendet werden können.

4.2.2.1 Die Prozeßfallbasis

Die Prozeßfallbasis besteht aus der persistenten Speicherung der Fallbeispiele auf einem externen Speichermedium und einer Datenstruktur, die ein effizientes Wiederauffinden dieser Fälle erlaubt.

$$Fallbasis = Archiv \times Knotenstruktur$$

Die allgemeine Invariante dieses Datenmodells besteht aus der Konsistenz der physisch angelegten Fallbasis im Archiv und der logischen Betrachtung der Fallbasis durch die sie beschreibende Datenstruktur.

$$inv\text{-}Fallbasis(fb) \triangleq (inv\text{-}Archiv(fb.archiv) \grave{U} inv\text{-}Knotenstruktur(fb.knotenstruktur)) \grave{U} (\forall nodeKS \hat{I} \underline{dom}\ fb.knotenstruktur \cdot (\exists! nodeAR \hat{I} \underline{dom}\ fb.archiv \cdot nodeAr = nodeKS))$$

Wie bereits erläutert, können im Archiv der Reportingplattform beliebige Objekte persistent gespeichert werden, so daß dort die Fallbeispiele der Fallbasis unter einer benutzerspezifischen ID hinterlegt werden. Zur Strukturierung der Fallbeispiele werden Kategorien (im folgenden *Ordner* genannt) verwendet, die ebenfalls im Archiv abgebildet werden.
Die allgemeine Form eines Archiveintrags lautet daher:

$$\begin{aligned}
Archiv &= ID \xrightarrow[m]{} Objekt \\
ID &= Kennung \times Benutzer \times KnotenNummer \\
Kennung &= token \\
Benutzer &= Char^{+} \\
Knotennummer &= N_1 \\
Objekt &= Ordner \mid Fallbeispiel \\
Ordner &= KnotenDaten \times KnotenNummer^{*} \\
KnotenDaten &= Typ \times Name \times Beschreibung \\
Typ &= \text{'Ordner'} \mid \text{'Prozeß'} \mid \text{'Funktion'} \mid \text{'Ereignis'} \mid \text{'Operator'} \\
Name &= Char^{*} \\
Beschreibung &= Char^{*} \\
Fallbeispiel &= Prozeß \mid Funktion \\
Prozeß &= KnotenDaten \times Prozeßstruktur \times Referenz \\
Prozeßstruktur &= Char^{*} \\
Referenz &= N_1 \\
Funktion &= KnotenDaten \times Material \times Referenz \\
Material &= Char^{*}
\end{aligned}$$

Da an späterer Stelle die Beschreibung der internen Darstellung der Struktur eines Prozesses und dessen Funktionsknoten erfolgt, wird hier zunächst die Invariante der physischen Repräsentation der Prozeßfallbasis im Archiv lediglich verbal erläutert.

Die Konsistenz der Fallbasis ist dann gewahrt, wenn die Vereinigungsmenge aller verwendeten Knotennummern innerhalb eines Ordner-, Prozeß- oder Funktionsknotens eine Teilmenge des Urbildbereichs des Archivs ist.

Neben dieser physischen Repräsentation von Fällen und Ordnern im Archiv, bedarf es einer Struktur, die die Beziehungen der Objekte der Fallbasis adäquat widerspiegelt. Adäquat bedeutet in diesem Zusammenhang (siehe Abbildung 42), daß

- *Bottom-Up*: von einem Attribut auf alle DataSets, von jedem DataSet auf alle Prozesse und von jedem Prozeß auf alle Ordner/Hauptprozesse
- *Top-Down:* von jedem Ordner/Hauptprozeß auf alle Prozesse, von jedem Prozeß auf alle DataSets und von jedem DataSet auf alle Attribute

direkt oder *indirekt* geschlossen werden kann.

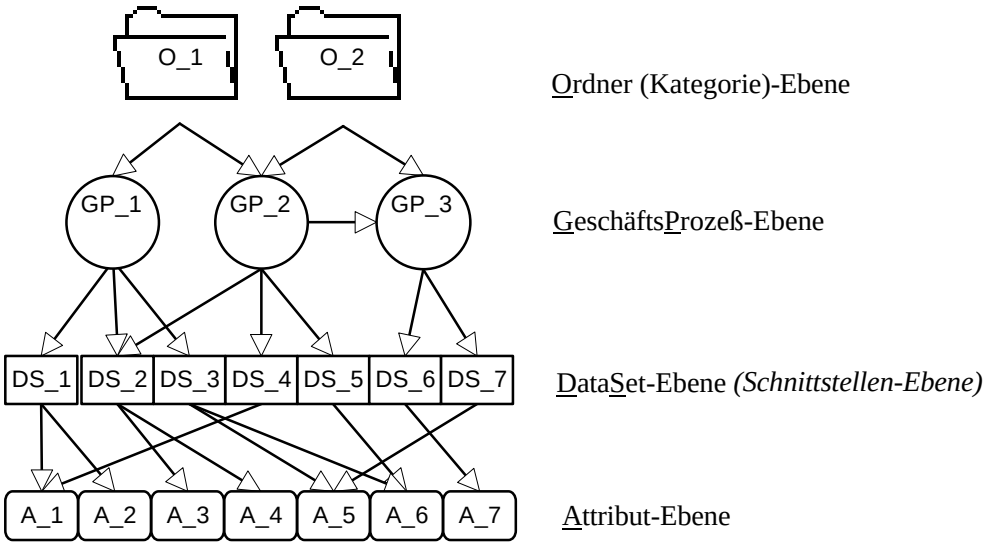


Abbildung 42: Verknüpfung von Informationsbausteinen

Die nachfolgende Datenstruktur ermöglicht gerade diese Anforderung bezüglich der geforderten Bottom-Up-Analyse bzw. Top-Down-Analyse eines beliebigen Objekts der Fallbasis.

$$
\begin{aligned}
&\textit{Knotenstruktur} && = \textit{ID} \xrightarrow[m]{} \textit{VorknotenID} \times \textit{NachknotenID} \\
&\textit{VorknotenID} && = \textit{KnotenSet} \\
&\textit{NachknotenID} && = \textit{KnotenSet} \\
&\textit{KnotenSet} && = \textit{ID-set}
\end{aligned}
$$

Die Invariante der Knotenstruktur stellt sicher, daß alle als Vorgänger oder Nachfolger verwendeten Knoten auch gültige Knoten dieser Struktur sein müssen. Darüber hinaus darf kein Knoten sich selber als Vorgänger- bzw. Nachfolgerknoten haben (direkte Rekursion). Zur Vermeidung einer indirekten Rekursion darf kein Vorgängerknoten eines Knotens A (oder Knoten A selber) in der durch A direkt oder indirekt erreichbaren Knotenmenge vorhanden sein. Die Menge der erreichbaren Knoten wird dabei durch eine Hilfsfunktion *getLinks* ermittelt (s.u.).

inv-Knotenstruktur(ks) ≙
(union { ks(id).VorknotenID È ks(id).NachknotenID | id Î dom ks } Í dom ks) Ù
node Ï ks(node).VorknotenID È ks(node).NachknotenID)) Ù
(" node Î dom ks · (Ø$ id Î ks(node).VorknotenIDÈ{node} · id Î getLinks(ks(node).NachknotenID, ks))

Konzeptuelle Abgrenzung

Die in Abbildung 42 dargestellten Verknüpfungen von Geschäftsprozessen, DataSets und Attributen zeigen die logische Sicht einer möglichen Kombination von Informationsbausteinen der Reporting-plattform, die ihrerseits durch Ordner strukturiert werden können.
Bedingt durch die interne Repräsentation eines DataSets und dessen zugehörigen Attributen können diese Verknüpfungen jedoch nicht vollständig in die oben beschriebene Knotenstruktur integriert

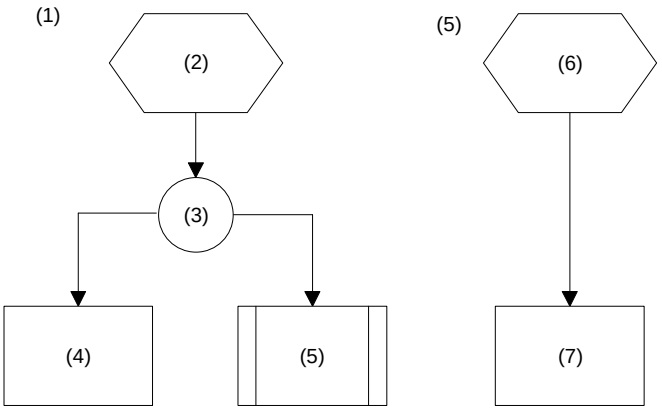
werden, so daß sich diese auf die Repräsentation der logischen Zusammenhänge innerhalb der Ordnerebene, Geschäftsprozeßebene und DataSet-Ebene beschränkt. Verknüpfungen zwischen der DataSet- und Attributebene werden auf der Ebene der SQL-Templates realisiert. Für die erforderliche Kommunikation an der Schnittstellenebene existieren eine Reihe von Funktionen, die über die verschiedenen Zusammenhänge der beiden Ebenen Auskunft geben.

So ergibt sich für die in Abbildung 42 beispielhaft vorgestellten Informationszusammenhänge eine interne Darstellung der beschriebene Knotenstruktur folgender Art (ohne Attribute):

Knotenstruktur = { O_1 ® ({}, {GP_1, GP_2}),
O_2 ® ({}, {GP_2, GP_3}),
GP_1 ® ({O_1}, {DS_1, DS_2, DS_3}),
GP_2 ® ({O_1, O_2}, {GP_3, DS_2, DS_4, DS_5}),
GP_3 ® ({O_2, GP_2}, {DS_6, DS_7}),
DS_1 ® ({GP_1}, {}),
DS_2 ® ({GP_1, GP_2}, {}),
DS_3 ® ({GP_1}, {}),
DS_4 ® ({GP_2}, {}),
DS_5 ® ({GP_2}, {}),
DS_6 ® ({GP_3}, {}),
DS_7 ® ({GP_3}, {})
}

4.2.2.1.1 Interne Darstellung von Prozessen im Archiv

Geschäftsprozesse haben wie alle Einträge der Fallbasis einen Namen und eine Beschreibung. Zudem können Prozesse eine beliebige Anzahl an Funktions-, Ereignis- und Steuerknoten besitzen, die letztlich die spezifische Struktur eines Geschäftsprozesses definieren. Die persistente Abbildung dieser Graphenstruktur erfolgte in Kapitel 3.6.4.5 *Überführung des Datenmodells in konkrete Speicherungsstrukturen* durch paarweise Kombination jeweils eines Knotens und seines Nachfolgers in Repositorytabellen.
Nun wird eine Anpassung hinsichtlich der Speicherung von Geschäftsprozessen notwendig, da sämtliche Objekte der Fallbasis im Archiv (siehe *Kapitel 4.4.3 Das Archiv*) der Reportingplattform hinterlegt werden und im wesentlichen die folgenden an einem Beispielprozeß verdeutlichten Änderungen beinhaltet:



(1): Hauptprozeß
(2), (6): Ereignisse
(3): Operator
(4), (7): Funktionen
(5): Subprozeß

Abbildung 43: Beispiel einer Prozeßstruktur

Vorknoten	Nachknoten
1	2
2	3
3	4
3	5
5	6
6	7

Tabelle 30: Ursprüngliche Speicherung der Prozeßstruktur

Wurde die Prozeßstruktur ursprünglich durch paarweise Kombination der Prozeßknoten in eine Repositorytabelle (wie in Tabelle 30 dargestellt) hinterlegt, so wird diese nun durch die Einführung einer Nachfolger- und Aufzählungsnotation

X-Y : Y ist direkter Nachfolger von X
X-Y,Z : Y und Z sind direkte Nachfolger von X

durch folgende Zeichenkette repräsentiert:

Struktur Prozeß 1 = "2-3;3-4,5"
Struktur (Sub-)Prozeß 5 = "6-7"

Die Zahlen stellen hier die Knotennummern der jeweiligen Prozeß- Ereignis-, Operator- oder Funktionsknoten dar, die ebenfalls im Archiv mit ihrer jeweiligen Struktur hinterlegt werden.

$$Ereignis = ID \xrightarrow[m]{} (Typ \times Beschreibung \times Bedingung)$$
$$Bedingung = Char^{*}$$
$$Operator = ID \xrightarrow[m]{} (Typ \times EinOperator \times AusOperator)$$
$$EinOperator = Char^{*}$$
$$AusOperator = Char^{*}$$

Aus der Sicht der Retrievephase stellen Ereignisse und Operatoren innerhalb eines Prozesses jedoch keine Fallbeispiele dar, so daß diese Knoten in diesem Sinne auch keine relevanten Informationen enthalten und daher nicht in die Indexstruktur integriert werden.

4.2.2.1.2 Interne Darstellung von Funktionen im Archiv

Funktionsknoten repräsentieren u.a. die DataSets der Reportingplattform. Sie finden innerhalb von Prozessen Verwendung, werden aber zudem aus der Sicht des fallbasierten Schließens als eigenständige Fallbeispiele angesehen, die im Rahmen der Retrievephase auf ihr Lösungspotential untersucht werden.

$$Funktion = ID \xrightarrow[m]{} (KnotenDaten \times Material \times Referenz)$$

Ein Funktionsknoten enthält neben einem Namen und einer Beschreibung einen sogenannten *Materialbeschreibungsstring*, der einen Verweis auf die eigentlichen Metadaten der darzustellenden Datenmenge enthält. Dieser Verweis wird ebenfalls als Zeichenkette abgebildet und hat folgenden Aufbau:

Material = "RecordID | Materialtyp | Materialname | Toolname"

Die RecordID repräsentiert letztlich die Knotennummer des Archiveintrags, auf den die Reportingplattform immer dann Bezug nimmt, wenn eine Datenmenge am Bildschirm dargestellt werden soll. Der Materialtyp besagt, ob die Datenmenge ein Werkzeug zur Darstellung besitzt (Typ = Tool) oder nicht; ist dies der Fall, so wird als Toolname der Name eben dieses Darstellungswerkzeugs gespeichert. Der Name des Materials ist ein optionaler Eintrag.

Referenzzähler:
Die Fallbeispiele der Fallbasis können aus logischer Sicht beliebig oft wiederverwendet werden. Funktionen und Prozesse können durch die Benutzer des Systems den verschiedenen Ordnern

(Kategorien) und Prozessen mehrfach zugeordnet werden. Da jedes Fallbeispiel jedoch nur einmal physisch im Archiv hinterlegt wird, ist es zur Sicherung der referentiellen Integrität notwendig, daß dieses nur dann aus dem Archiv entfernt werden darf, wenn es von keinem anderen Eintrag der Fallbasis verwendet wird. Die Prüfung dieser Bedingung erfolgt durch einen Referenzzähler, der für jedes Fallbeispiel des Archivs eingeführt wird und anzeigt, wie häufig ein Fallbeispiel *systemweit* verwendet wird. Somit wird bei jeder neuen Referenz auf dieses Fallbeispiel der Zähler inkrementiert, bzw. bei einer gelöschten Referenz dekrementiert. Ein Fall darf demnach nur dann physisch gelöscht werden, wenn der Wert des Referenzzählers 1 beträgt (die eigene Referenz).

4.2.2.1.3 Operationen auf Objekten der Fallbasis

Die Operationen, die auf die Objekte der Prozeßfallbasis im Zuge des Prozeßmodells des fallbasierten Schließens ausgeführt werden, sind Gegenstand der folgenden Ausführungen. Dabei werden die wichtigsten Funktionen, sowie deren Anwendbarkeit in Form von Vorbedingungen spezifiziert. Auf die Spezifikation von Hilfsfunktionen, die lediglich eine einfache Bedingung testen, wird bei diesen Ausführungen verzichtet. Die angesprochenen Hilfsfunktionen sind an dem allgemeinen Präfix "*is_*" im Funktionsnamen zu erkennen.

Erweitern der Fallbasis

Die Fallbasis muß erweiterbar sein (Retainphase), so daß eine Methode zum Einfügen eines Objekts (Ordner oder Fallbeispiel) erforderlich ist, wobei grundsätzlich die Unterscheidung zu treffen ist, ob es sich bei der anstehenden Erweiterung der Fallbasis um das Etablieren gänzlich neuen Wissens handelt, oder ausschließlich um eine Verknüpfung bestehender Wissenseinträge.
Die nachfolgende Tabelle verdeutlicht, welche Objekte aus logischer Sicht hierarchisch gegliedert werden dürfen:

Hauptobjekt **Subobjekt**	**Ordner**	**Prozeß**	**Funktion**
Ordner	✓		
Prozeß	✓	✓	
Funktion	✓	✓	

Tabelle 31: Hierarchisierung von Objekten der Fallbasis

Integration neuen Wissen in die Fallbasis

Einzufügendes neues Objekt ist ein Ordner:
Ein neuer Ordner kann nur als Subeintrag eines bestehenden Ordners erstellt werden. Der neue Ordner hat nach der Erstellung zunächst keine eigenen Einträge, die Liste der Einträge des alten Ordners wird um die ID des neuen erweitert. Mit dem physischen Anlegen des neuen Ordners im Archiv geht ebenfalls die Aktualisierung der die Fallbasis beschreibenden Knotenstruktur einher:

type createNewFolder: Fallbasis×ID×Name×Beschreibung $\xrightarrow{\sim}$ Fallbasis
pre-createNewFolder(fb, id, name, text) ≙ is_Folder(fb.archiv, id)
createNewFolder(fb, id, name, text) ≙
let newID = saveNewFolder(fb.archiv, id, name, text) in
establishNewNode(fb.knotenstruktur, id, newID)

Physisches Anlegen des neuen Ordners:

type saveNewFolder: Archiv×ID×Name×Beschreibung ® ID
pre-saveNewFolder(ar, id, name, text) ≙ id Î dom ar
saveNewFolder(ar, id, name, text) ≙
let newID = getNextID() in
ar È { newID ↦ (('Ordner', name, text), [ ]) } Ù ar(id).Knotennummer ↛ [newID]

Aktualisierung der Knotenstruktur:

type establishNewNode: Knotenstruktur×ID×ID $\xrightarrow{\sim}$ Knotenstruktur
pre-establishNewNode(ks, parentID, childID) ≙ parentID Î dom ks Ù childID Ï dom ks
establishNewNode(ks, parentID, childID) ≙
ks È { childID ® ({parentID}, {}) } Ù
ks + { parentID ® (ks(parentID).VorknotenID, ks(parentID).NachknotenID È childID) }

Einzufügendes neues Objekt ist ein Fallbeispiel:
Ein Fallbeispiel kann nur einem Ordner oder einem Prozeß untergeordnet werden. In Abhängigkeit davon, erfolgt entweder die Erweiterung der Ordnereinträge um die ID des neuen Fallbeispiels, oder die Aktualisierung der Prozeßstruktur.

type createNewCase: $Fallbasis \times ID \times Typ \times Name \times Beschreibung \xrightarrow{\sim} Fallbasis$
pre-createNewCase(fb, id, typ, name, text) ≙ *is_Folder(fb.archiv, id)Ú is_Process(fb.archiv, id)*
createNewCase(fb, id, typ, name, text) ≙
let newID = saveNewCase(fb.archiv, id, typ, name, text) in
if id Î dom fb.Knotenstruktur then establishNewNode(fb.knotenstruktur, id, newID)

Physisches Anlegen des neuen Fallbeispiels:

type saveNewCase: $Archiv \times ID \times Typ \times Name \times Beschreibung \to ID$
pre-saveNewCase(ar, id, typ, name, text) ≙ *id Î dom ar*
saveNewCase(ar, id, typ, name, text) ≙
let newID = getNextID() in
if typ = 'Prozeß' then ar È { newID ↦ (('Prozeß', name, text), "", 1) }
else let material = MatExt_ComfortSerialize(ar) in ar È {newID ↦ (('Funktion', name, text), material, 1)} Ù
if is_Folder(ar, id) then ar(id).Knotennummer ↷ [newID]
else insertProcessNode(ar, id, newID)

Wird ein Prozeß in seiner Struktur durch Einfügen eines neuen Knotens (Subprozeß oder Funktionsknoten) geändert, so erfolgt die Anpassung der Prozeßstruktur ebenfalls auf der Archivebene. Das Einfügen und auch Löschen eines Knotens in bzw. aus einen(m) Prozeß wurde bereits hinreichend mit den damit verbundenen Konsistenzprüfungen in Kapitel 3.6.4.6 *Steuerung von Prozessen* spezifiziert, so daß an dieser Stelle lediglich die Signatur der Hilfsfunktion dargestellt wird.

type insertProcessNode: $Archiv \times ID \times ID \to Bool$
type deleteProcessNode: $Archiv \times ID \times ID \to Bool$

Verknüpfung bestehender Wissenseinträge der Fallbasis
Bei der Verknüpfung bestehender Wissenseinträge entfällt das physische Anlegen eines Fallbeispiels. Es wird lediglich eine Aktualisierung der entsprechenden Ordner- bzw. Prozeßstruktur im Archiv vorgenommen und bzgl. der Knotenstruktur eine neue Kante zwischen dem Knoten des Haupteintrags und dem Knoten des Subeintrags etabliert.

Durch die mehrfache Verwendung von Fallbeispielen ergibt sich zudem die Erhöhung des Referenzzählers eines jeden, dem Subeintrag (und dessen Subeinträge usw.) untergeordneten Fallbeispiels.

type establishNewEdge: $Fallbasis \times ID \times ID \xrightarrow{\sim} Fallbasis$
pre-establishNewEdge(fb, parentID, childID) ≙ *parentID Î dom fb.knotenstruktur Ù childID Î dom fb.knotenstruktur*
establishNewEdge(fb, parentID, childID) ≙
let ks = fb.knotenstruktur in
(ks + { parentID ® (ks(parentID).VorknotenID, ks(parentID).NachknotenIDÈ{childID}) } Ù
ks + { childID ® (ks(childID).VorknotenID È {parentID}, ks(childID).NachknotenID) }) Ù
if is_Folder(fb.archiv, parentID) then fb.archiv(parentID).Knotennummer ↷ [childID]
else insertProcessNode(fb.archiv, parentID, childID)
Ù (" nodeID Î getLinks({childID}, ks) · incrementReference(fb.archiv, nodeID))

Die Funktion *getLinks* ermittelt alle Fallbeispiele, die mit einer gegebenen Menge von Fällen in einer *Part-Of-Relation* stehen. Für Funktionen ist diese Nachfolgermenge naturgemäß leer, da diese laut Tabelle 31 keine Subeinträge enthalten.

type getLinks: $KnotenSet \times Knotenstruktur \to KnotenSet$
pre-getLinks(s_0, ks) ≙ true
getLinks(s_0, ks) ≙ let $s_1 = s_0 \cup$ union { ks(node).NachfolgerID | node ∈ s_0 } in
if $s_1 = s_0$ then s_1 else getLinks(s_1, ks)

Der Vollständigkeit halber sei ebenfalls die Signatur und Vorbedingung der Hilfsfunktion *incrementReference* spezifiziert.

type incrementReference: $Archiv \times ID \to Bool$
pre-incrementReference(ar, id) $\triangleq$ *is_Case(ar, id)*

Analog werden die im weiteren Verlauf der Ausführungen verwendeten Hilfsfunktionen *decrementReference* und *getReference* spezifiziert:

type decrementReference: $Archiv \times ID \to Bool$
pre-decrementReference(ar, id) $\triangleq$ *is_Case(ar, id)*

type getReference: $Archiv \times ID \to N_1$
pre-getReference(ar, id) $\triangleq$ *is_Case(ar, id)*

Entfernen eines Objekts aus der Fallbasis
Ein Objekt kann aus der Fallbasis ebenfalls entfernt werden. Handelt es sich bei dem Objekt um einen Ordner oder einen Prozeß, so führt dies zum rekursiven Löschen aller Subeinträge dieses Ordners bzw. Prozesses. Fallbeispiele dürfen nur dann physisch gelöscht werden, wenn diese von keinem anderen Eintrag benutzt werden (Referenz = 1), sonst wird lediglich ihr Referenzzähler dekrementiert.

type removeObject: $Fallbasis \times ID \xrightarrow{\sim} Fallbasis$
pre-removeObject(fb, id) $\triangleq$ *true*
removeObject(fb, id) $\triangleq$
let childID $\in$ *fb.knotenstruktur(id).NachfolgerID in*
    removeEdge(fb, id, childID) $\circ$
    let childReference = getReference(fb.archiv, childID) in
    if childReference = 1
    then deleteObject(fb.archiv, childID) $\circ$ *removeNode(fb.knotenstruktur, childID)*
    else decrementReference(fb.archiv, childID)

Entfernen der Verbindung zweier Knoten:

type removeEdge: $Fallbasis \times ID \times ID \xrightarrow{\sim} Fallbasis$
pre-removeEdge(fb, parentID, childID) $\triangleq$ *parentID* $\in$ *dom fb.knotenstruktur* $\cap$ *childID* $\in$ *dom fb.knotenstruktur*
removeEdge(fb, parentID, childID) $\triangleq$
let ks = fb.knotenstruktur in
    (ks + { parentID $\to$ *(ks(parentID).VorknotenID, ks(parentID).NachknotenID -{childID}) }* $\circ$
      ks + { childID $\to$ *(ks(childID).VorknotenID - {parentID}, ks(childID).NachknotenID) })* $\circ$
     if is_Folder(fb.archiv, parentID) then deleteFolderEntry(fb.archiv,parentID, childID)
     else deleteProcessNode(fb.archiv, parentID, childID)

Physisches Löschen eines Objekts aus dem Archiv, dessen Referenzzähler 1 betragen muß (die eigene Referenz):

type deleteObject: $Archiv \times ID \xrightarrow{\sim} Archiv$
pre-deleteObject(ar, id) $\triangleq$ *getReference(ar, id)* $\in$ *1*
deleteObject(ar, id) $\triangleq$ *ar - { id }*

Entfernen eines Knotens aus der Knotenstruktur, dessen Menge an Vorgänger- und Nachfolgerknoten leer sein muß (isolierter Knoten):

type removeNode: $Knotenstruktur \times ID \xrightarrow{\sim} Knotenstruktur$
pre-removeNode(ks, id) $\triangleq$ *id* $\in$ *dom ks* $\circ$ *(ks(id).VorknotenID* $\cup$ *ks(id).NachknotenID* $\in$ *{})*
removeNode(ks, id) $\triangleq$ *ks - { id }*

Die Funktionen zur Änderung eines Objekts (Name, Beschreibung usw.) erfolgen analog und werden daher an dieser Stelle nicht weiter spezifiziert.

Damit sind die grundlegenden Manipulationsmöglichkeiten der Fallbasis beschrieben worden, so daß im folgenden die komplexen Stellen des Dialognetzes aus Abbildung 41 erläutert werden können.

4.2.2.2 Dialog zur Problemspezifikation

Die zentrale Rolle bei der Problemspezifikation spielt das *Glossar*. Innerhalb des Glossars werden alle dem System bekannten Problemfaktoren (Artefakte: Prozesse, DataSets, Attribute) alphabetisch nach Namen sortiert angezeigt, und dienen dem Benutzer als eine Auswahl zur Beschreibung des gegebenen Problems.

Glossar = *Artefakt-set*
Problem = *Artefakt-set*
Artefakt = *Fallbeispiel* | *Attribut*
Attribut = *ID*

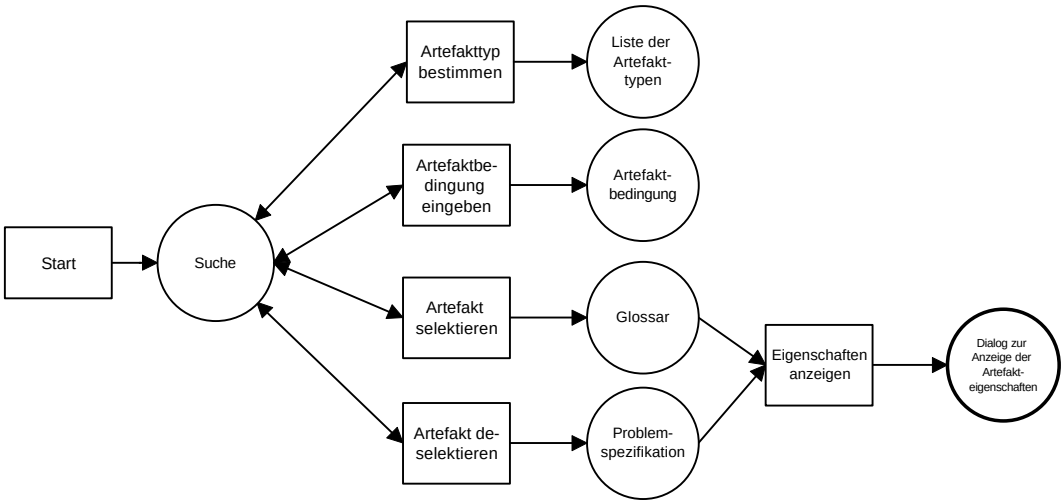


Abbildung 44: Unterdialognetz zur Retrievephase

Durch die verschiedenen, in Abbildung 44 dargestellten Interaktionsmöglichkeiten des Anwenders, ergeben sich zahlreiche potentielle Sichten auf das Glossar, die im folgenden näher betrachtet werden müssen.

Sichten des Glossars
Die Sicht auf das Glossar kann durch verschiedene Aktionen des Benutzers beeinflußt werden.

Artefakttyp bestimmen:
Bei Auswahl dieser Sicht werden im Glossar nur solche Artefakte angezeigt, deren Typ mit dem ausgewählten Artefakttyp übereinstimmt (z.B.: Alle Artefakte des Typs *Prozeß*).

Artefaktbedingung eingeben:
Diese Sicht ermöglicht es dem Benutzer, nur solche Artefakte im Glossar anzeigen zu lassen, die einem textlichen Vergleich mit der Artefaktbedingung genügen (z.B.: Alle Artefakte, die den Begriff "fahrer" im Namen tragen).

*Domain bestimmen (*Siehe *Kapitel 4.2.2.3 Dialog zur Spezifikation der Retrievephase):*
Eine weitere Sicht ergibt sich durch die Festlegung einer Domain. Dabei werden im Glossar nur solche Artefakte berücksichtigt, die auch wirklich mit dieser Domain in Zusammenhang stehen, also darin vorkommen (z.B.: Alle Artefakte der Domain ″Logistik″).

Artefakt auswählen
Diese Aktion dient der eigentlichen Problembeschreibung. Bei Auswahl eines Artefakts im Glossar wird durch diese Aktion das gewählte Artefakt in die Problembeschreibung überführt.

Da jede dieser Aktionen zu jedem Zeitpunkt und in kombinierter Form durchgeführt werden kann, führt dies zu einer Änderung der aktuellen Sicht auf das Glossar, so daß die Liste der anzuzeigenden Artefakte dahingehend aktualisiert werden muß, daß sich innerhalb des Glossars nur solche Artefakte befinden, die allen gegebenen Bedingungen genügen. Die oben aufgeführten Beispiele ergeben in kombinierter Form also die globale Glossarbedingung

"Zeige alle Prozesse aus dem Bereich Logistik, die den Begriff 'fahrer' im Namen enthalten".

Aus dieser Forderung läßt sich die folgende Invariante des Glossars ableiten:

$$inv\text{-}Glossar(FB, D, T, B) = (\forall a \in A_G \cdot a \in \{ A_D \cap A_T \cap A_B \} \cup a \in A_P)$$

wobei

FB die Fallbasis,
D die aktuelle Domain (einen Ordner) der Fallbasis,
T den ausgewählten Typ der Artefakte,
B die formulierte Bedingung,
A_G *die Menge der Artefakte im Glossar,*
A_D *die Menge der Artefakte der aktuellen Domain,*
A_T *die Menge der Artefakte eines Typs*
A_B *die Menge der Artefakte, die einer formulierten Bedingung genügen*
A_P *die Menge der Artefakte der Problembeschreibung (bereits ausgewählte Artefakte)*

darstellen.

Die zur Bildung der erforderlichen Mengen notwendigen Funktionen werden wie folgt spezifiziert:

A_D, *Menge der* Artefakte *der aktuellen Domain:*
Die Menge der Artefakte der aktuellen Domain stellen bezüglich der Fallbeispiele eine Teilmenge der gesamten Fallbasis dar. Es werden sämtliche Fallbeispiele ermittelt, die einem Ordner (oder dessen Subordnern usw.) zugeordnet sind.

type getDomainSet: $Fallbasis \times ID \rightarrow KnotenSet$
pre-getDomainSet(fb, id) $\triangleq$ *is_Folder(fb.archiv, id)*
getDomainSet(fb, id) $\triangleq$ $\underline{let}$ *ks = fb.knotenstruktur* $\underline{in}$
{ nodeID | nodeID $\in$ $\underline{dom}$ *ks* $\cdot$ *nodeID* $\in$ *getLinks(ks(id).NachfolgerID, ks)* $\cup$ $\neg$*is_Folder(fb.archiv, nodeID) }*

Diese Vorgehensweise ist aber nur bedingt anwendbar, da der Anwender die Möglichkeit hat, die Artefakte der zu erstellenden Problembeschreibung verschiedenartig zu verknüpfen (siehe Kapitel *4.2.2.3 Interne Darstellung von Funktionen im Archiv*) und dieses sich ebenfalls regelnd auf die anzuzeigenden Artefakte des Glossars auswirkt. So reicht es bei einer UND-verknüpften Problembeschreibung nicht aus, allein alle Fallbeispiele eines Ordners zu ermitteln, vielmehr impliziert diese Verknüpfungsart, nur solche Fallbeispiele in Betracht zu ziehen, die allen Artefakten der Problembeschreibung genügen und diese realisieren können.

Bezogen auf die Knotenstruktur ist hierzu die Existenz einer Kante zwischen dem Knoten des potentiellen Fallbeispiels und jedem Knoten der Problembeschreibung, der einen zu berücksichtigenden Problemfaktor darstellt, erforderlich, bzw. einem Knoten, der bereits als Kandidat der Lösungsmenge ermittelt wurde.

type getCompatibleNodes: $Fallbasis \times KnotenSet \rightarrow KnotenSet$
pre-getCompatibleNodes(fb, problem) $\triangleq$ $\underline{card}$ *problem > 0*
getCompatibleNodes(fb, problem) $\triangleq$
{ compNode | compNode $\in$ $\underline{dom}$ *fb.knotenstruktur* $\cdot$ $\neg$*is_Folder(fb.archiv, compNode)* $\cup$
problem $\cap$ *getLinks(fb.knotenstruktur(compNode).NachfolgerID, fb.knotenstruktur) }*

Zur Veranschaulichung dieser Vorgehensweise soll beispielhaft die bereits in Abbildung 42 vorgestellte Knotenstruktur dreier einfach strukturierter Geschäftsprozesse betrachtet werden.

fallbasis.knotenstruktur = { O_1 ® ({}, {GP_1, GP_2}),
O_2 ® ({}, {GP_2, GP_3}),
GP_1 ® ({O_1}, {DS_1, DS_2, DS_3}),
GP_2 ® ({O_1, O_2}, {GP_3, DS_2, DS_4, DS_5}),
GP_3 ® ({O_2, GP_2}, {DS_6, DS_7}),
DS_1 ® ({GP_1}, {}),
DS_2 ® ({GP_1, GP_2}, {}),
DS_3 ® ({GP_1}, {}),
DS_4 ® ({GP_2}, {}),
DS_5 ® ({GP_2}, {}),
DS_6 ® ({GP_3}, {}),
DS_7 ® ({GP_3}, {})
}

Beispielhaft sei ferner eine Problembeschreibung bestehend aus dem DataSet DS_7 angenommen. So ergibt sich hieraus bei einer UND-verknüpften Problembeschreibung eine kompatible Menge von potentiellen Fallbeispielen

getCompatibleNodes(fallbasis, { DS_7 }) = { GP_3, GP_2 },

da GP_3 alle Problemfaktoren als direkte und GP_2 als indirekte Nachfolger beinhaltet.

A_T, Menge der Artefakte eines Typs:
Aus einer gegebenen Knotenmenge werden alle Artefakte herausgefiltert, die einem gegebenen Typ entsprechen.

type getTypSet: Fallbasis×KnotenSet×Typ ® KnotenSet
pre-getTypSet(fb, s, t) ≙ " a Î s· a Î dom fb.archiv
getTypSet(fb, s, t) ≙ { nodeID | nodeID Î s · fb.archiv(nodeID).Typ = t }

A_B die Menge der Artefakte, die einer formulierten Bedingung genügen*:*
Aus einer gegebenen Menge werden alle Artefakte herausgefiltert, in deren Name ein gegebener Begriff enthalten ist (der Begriff kann dabei auch Platzhalter enthalten).

type getBedingungSet: Fallbasis×KnotenSet×Bedingung ® KnotenSet
pre-getBedingungSet(fb, s, b) ≙ " a Î s· a Î dom fb.archiv Ù b <> ""
getBedingungSet(fb, s, b) ≙ { nodeID | nodeID Î s · is_Matched(fb.archiv(nodeID).Name, b) }

A_G die Menge der zulässigen Artefakte im Glossar:
type getGlossarSet: Fallbasis×ID×Typ×Bedingung×Verknüpfung×Problem ® KnotenSet
pre-getGlossarSet(fb, id, t, b, vk, problem) ≙ true
getGlossarSet(fb, id, t, b, vk, problem) ≙
let s_1 = getDomainSet(fb, id) in
if vk = 'UND' then let s_1 = s_1 Ç getCompatibleNodes(fb, problem) in
let s_2 = getTypSet(fb, s_1, t) in { nodeID | nodeID Î getBedingungSet(fb, s_2, b) }

Analog darf die Problembeschreibung nur solche Artefakte enthalten, die durch die aktuelle Domain realisiert werden können. Diese Forderung ergibt sich aus der Tatsache, daß der Anwender jederzeit den zu untersuchenden Problembereich ändern kann und daher getestet werden muß, ob ein bereits zur Problemspezifikation verwendetes Artefakt der vorherigen Domain auch tatsächlich in der neuen Domain vorkommt.

Die Invariante für diesen Sachverhalt lautet daher:

inv-Problemspezifikation(D) = (" a Î A_P · a Î A_D Ù a Ï A_G)

Eine weitere Möglichkeit stellt laut Abbildung 44 die Anzeige von *Eigenschaften* eines Artefakts dar. Diese Möglichkeit bietet den Anwender eine detailliertere Auskunft (Beschreibung, Verwendungs-

zweck etc.) über das Artefakt und hat keinen Einfluß auf das Glossar oder die Problembeschreibung. Dieser Dialog wurde als eine modale Stelle dargestellt, so daß der Anwender vor einer weiteren Fortführung der Anwendung dieses Fenster wieder schließen muß.

4.2.2.3 Dialog zur Spezifikation der Retrievephase

In dem in Abbildung 45 dargestellten Dialogfenster werden Optionen zur Steuerung der Retrievephase gesetzt. So kann durch die Bestimmung einer Domain -aus der Domainliste- der zu untersuchende Problembereich innerhalb der Retrievephase eingeschränkt bzw. erweitert werden. Dies ist immer dann sinnvoll, wenn dem Anwender bewußt ist, daß sich eine mögliche Lösung seines Problems ausschließlich in einem bestimmten Bereich der Fallbasis befinden kann (z.B. Betrachtung aller Fallbeispiele der Domain "*Logistik"*). Mit dieser Einschränkung des Suchraums wird naturgemäß auch eine Erhöhung der Performance des Systems innerhalb des eigentlichen Retrievals erreicht.

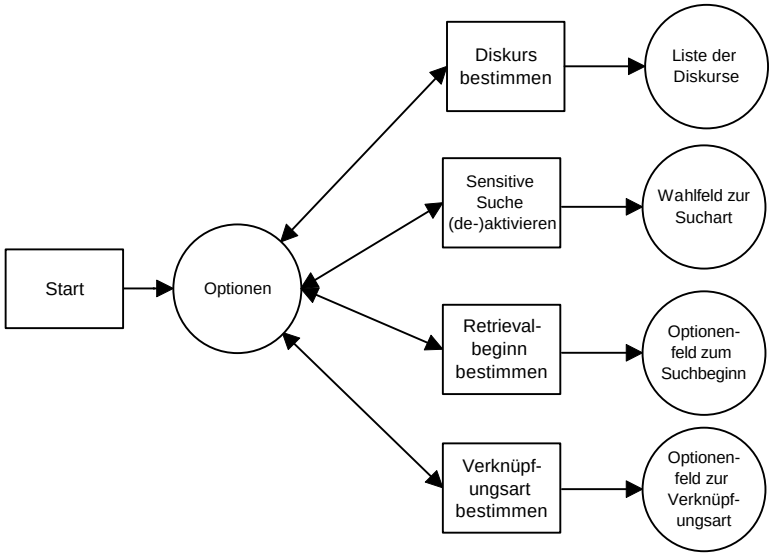


Abbildung 45: Unterdialognetz zur Spezifikation der Retrievephase

Eine weitere Möglichkeit dieses Dialogs ist die Aktivierung / Deaktivierung der *sensitiven Suche*. Diese Einstellung wirkt sich regelnd auf die Anzeige der Artefakte des Glossars aus (siehe *Kapitel 4.2.2.2 Dialog zur Problemspezifikation*), indem bei aktivierter *sensitiver Suche* alle Artefakte innerhalb des Glossars ausgeblendet werden, die der gegebenen Artefaktbedingung nicht entsprechen. Soll keine sensitive Suche erfolgen, so wird bei Eingabe einer textlichen Bedingung lediglich im Glossar geblättert und es wird das Artefakt markiert, welches der Bedingung am Wortanfang entspricht. Diese Regelungsmöglichkeit lehnt sich stark an die der Suche über den "Index" in Hilfesystemen an.
Durch die Einstellung des eigentlichen Retrievalstarts wird bestimmt, ob die Suche nach möglichen Lösungen immer dann automatisch gestartet werden soll, wenn ein Artefakt der Problembeschreibung hinzugefügt bzw. entnommen wurde, oder ob dieser Vorgang explizit durch den Anwender (per Knopfdruck) ausgelöst werden muß.

Die letzte Einstellungsmöglichkeit besteht in der Festlegung einer Verknüpfungsart der Artefakte (Problemfaktoren) der Problembeschreibung. Hierbei kann zwischen einer UND- und ODER-Verknüpfung der Artefakte gewählt werden. Eine ODER-Verknüpfung der Artefakte führt in der Regel zu einer größeren Lösungsmenge, da jede in Frage kommende Lösung nur mindestens eines der ausgewählten Artefakte erfüllen können muß. Diese Verknüpfungsart ist dann zu wählen, wenn es für den Anwender von Interesse ist, mögliche Fälle zu mindestens einem Symptom zu erhalten.

Bei einer UND-verknüpften Problembeschreibung muß jeder potentielle Fall allen in der Problembeschreibung ausgewählten Artefakten voll genügen. Diese Verknüpfungsart schließt also Fälle aus, die einer gegebenen Problembeschreibung nur teilweise entsprechen.

Da der Anwender für eine Problembeschreibung die Artefakte des Glossars nutzt, wird nach jeder Selektion eines Artefakts und Übernahme in die Problembeschreibung das Glossar auf kompatible Einträge überprüft. Es werden so nur Artefakte zur weiteren Problemspezifikation angeboten, deren Lösungspotential die bisherige Problembeschreibung umfaßt.

Ermittlung der Lösungsmenge
Nachdem das eigentliche Retrieval ausgelöst wurde (automatisch oder per Knopfdruck) erfolgt systemseitig die Ermittlung relevanter Lösungen. Die hierzu intern notwendigen Arbeitsschritte sind folgende:

1. Analyse der Problembeschreibung (Ermittlung relevanter Problemfaktoren)
2. Ermittlung und Bewertung relevanter Lösungen der Fallbasis

Zunächst erfolgt eine Analyse der Problembeschreibung hinsichtlich der zu untersuchenden Ebenen der Informationsverknüpfung (siehe Abbildung 42). Je nach Kombination der zur Problembeschreibung verwendeten Artefakttypen ist eine entsprechende Bottom-Up-Analyse erforderlich. Zur Zeit können die drei Artefakttypen Prozesse, DataSets und Attribute zur Problembeschreibung verwendet werden, so daß sich hieraus theoretisch acht Kombinationen (2^3) einer Problembeschreibung ergeben können.

Variante	Attribut	DataSet	Prozeß	Analyse beginnend ab
I				keine
II			✓	Prozeßebene
III		✓		DataSetebene
IV		✓	✓	DataSetebene
V	✓			Attributebene
VI	✓		✓	Attributebene
VII	✓	✓		Attributebene
VIII	✓	✓	✓	Attributebene

Tabelle 32: Kombinationen einer Problembeschreibung aus Artefakten

Die Ebene der Bottom-Up-Analyse ergibt sich demnach aus dem Vorhandensein eines Vertreters dieser Ebene in der Problembeschreibung.
Die Ermittlung und Bewertung relevanter Lösungen der Fallbasis involviert bei einer Analyse ab der Attributebene die Benutzung der bereits in Kapitel *3.7.3.3.3 Integration der* Methoden spezifizierten Schnittstellenfunktionen. Hierzu werden alle in der Problembeschreibung vorkommenden Attribute und deren Verknüpfungsart als Argumente an die Funktion *Ermittle_DataSet _Kandidaten* übergeben. Diese Funktion hat die Ermittlung jener DataSets zur Aufgabe, deren Selektions- oder Definitionsattribute die übergebene Attributmenge ganz oder teilweise (in Abhängigkeit der Verknüpfungsart) realisieren können. Darüber hinaus ermittelt die Funktion für jeden gefundenen DataSet ein Ähnlichkeitsmaß, welches ausdrückt, wie gut dieser DataSet den Umfang der gesuchten Attribute tatsächlich erfüllt.

Für die weiteren Ebenen (Verknüpfungen der DataSet- und Prozeßebene) wird analog verfahren.

type SolutionMap: ID ® R_0
getSolution: Knotenstruktur×KnotenSet×KnotenSet×KnotenSet×Verknüpfungsart® SolutionMap

pre-getSolution(ks, processSet, datasetSet, attributSet, vk) ≙
card union { processSet, datasetSet, attributSet } > 0

getSolution(ks, processSet, datasetSet, attributSet, vk) ≙
let datasetByAttrib = ***Ermittle_DataSet_Kandidaten(attributSet, vk)*** *in //DataSets durch Attribute ermitteln*
let processByDataSet = getProcessByDataSet(union { datasetSet, dom datasetByAttrib }, vk) in
let processByProcess = getProcessByProcess(union { processSet, processByDataSet }, vk) in
let solutionID Î union { dom datasetByAttrib, datasetSet, processSet, processByDataSet, processByProcess)} in
SolutionSimilarity È { solutionID ® getSolutionSimilarity(ks, solutionID, datasetSet, datasetByAttrib) }

Die oben verwendeten Hilfsfunktionen zur Ermittlung übergeordneter Prozesse spezifizieren sich wie folgt:

Prozesse durch DataSets ermitteln
Durch die Ermittlung aller Vorknoten eines DataSets erhält man nach Eliminierung der Ordnerknoten alle übergeordneten Prozesse, die als potentielle Lösungskandidaten zu sehen sind. Eine zusätzliche Forderung an den Lösungskandidaten stellt bei einer UND-verknüpften Problembeschreibung das Vorhandensein aller geforderten DataSets in diesem Prozeß dar.

getProcessByDataSet: Fallbasis ×KnotenSet×Verknüpfungsart® KnotenSet
pre-getProcessByDataSet(fb, ,datasetSet, vk) ≙ true
getProcessByDataSet(fb, ,datasetSet, vk) ≙
let ks = fb.knotenstrukur in
{ processID | processID Î { prenode| prenode Î union { getPreNode(ks, nodeID) | nodeID Î datasetSet }· is_Process(fb.archive, prenode) } ·
if vk = 'UND' then {id | id Î getPostNode(ks, processID)· is_DataSet(fb.archiv, id)} Í datasetSet
else true

Hauptprozesse durch Subprozesse ermitteln
Durch die Ermittlung aller Vorknoten eines Prozesses erhält man wiederum nach Eliminierung der Ordnerknoten alle übergeordneten Prozesse, die dadurch ebenfalls als potentielle Lösungskandidaten in Frage kommen. Eine zusätzliche Forderung an den Lösungskandidaten ergibt sich bei einer UND-verknüpften Problembeschreibung. Alle geforderten Prozesse der Problembeschreibung stellen Subprozesse dieses Kandidaten dar.

getProcessByProcess: Fallbasis ×KnotenSet×Verknüpfungsart® KnotenSet
pre-getProcessByProcess(fb, ,datasetSet, vk) ≙ true
getProcessByProcess(fb, ,processSet, vk) ≙
let ks = fb.knotenstrukur in
{ processID | processID Î { prenode| prenode Î union { getPreNode(ks, nodeID) | nodeID Î processSet }· is_Process(fb.archive, prenode) } ·
if vk = 'UND' then {id | id Î getPostNode(ks, processID)· is_Process(fb.archiv, id)} Í processSet
else true

Berechnung des Ähnlichkeitsmaßes einer Lösung
Bei der folgenden eigentlichen Berechnung des Ähnlichkeitsmaßes einer Lösung werden zwei grundsätzliche Annahmen getroffen.

1. Wird ein DataSet direkt in die Problembeschreibung aufgenommen, so wird davon ausgegangen, daß den Anwender ebenfalls sämtliche Selektionsattribute dieses DataSets interessieren, so daß dessen Ähnlichkeitsmaß stets den Wert 1 erhält.
2. Wird ein Geschäftsprozeß in die Problembeschreibung aufgenommen, so wird davon ausgegangen, daß den Anwender ebenfalls sämtliche DataSets und Subprozesse dieses Prozesses und damit auch sämtliche Attribute der DataSets bzw. DataSets der Subprozesse usw. interessieren. Das Ähnlichkeitsmaß dieses Prozesses ist also ebenfalls stets 1.

Werden hingegen DataSets über Attribute (siehe ebenfalls Kapitel *3.7.3.3.3 Integration der* Methoden*)* und Prozesse über DataSets (bzw. Subprozesse) gefunden, berechnet sich deren Ähnlichkeitsmaß wie folgt:

Gefundene Lösung ist ein Geschäftsprozeß

$$SIM_p = \frac{1}{fcard(p)} * \sum_{i=1}^{fcard(p)} SIMsub_i \text{ , } SIMsub_i \text{ Î } fsel(p),$$

wobei die Hilfsfunktionen

fcard(p) die Anzahl der in einem Prozeß p vorhandenen direkten Subeinträge ermittelt

und

fsel(p) die in einem Prozeß p vorhandenen direkten Subeinträge als Menge zurückgibt.

getSolutionSimilarity: Knotenstruktur×ID×KnotenSet×KnotenSet ® R_0
pre-getSolutionSimilarity(ks, solutionID, dataSet, datasetByAttrib) ≙ true

getSolutionSimilarity(ks, solutionID, dataSet, datasetByAttrib) ≙
let postNode Î ks(solutionID).NachfolgerID in
if postNode Î datasetSet then 1 else datasetByAttrib.Similarity
else let similarity = similarity + getSolutionSimilarity(postNode, datasetSet, datasetByAttrib) in
similarity / card ks(solutionID).NachfolgerID

4.2.2.4 Dialog zur Bearbeitung, Beurteilung und Übernahme einer Lösung

Das Unterdialognetz in Abbildung 46 stellt die Spezifikation zur Bearbeitung, Beurteilung und Übernahme einer Lösung dar. Unter einer Lösung kann sowohl ein bereits vorhandenes Fallbeispiel aus der Fallbasis als auch eine mittels der Werkzeuge der Reportingplattform neu erzeugte Lösung verstanden werden.

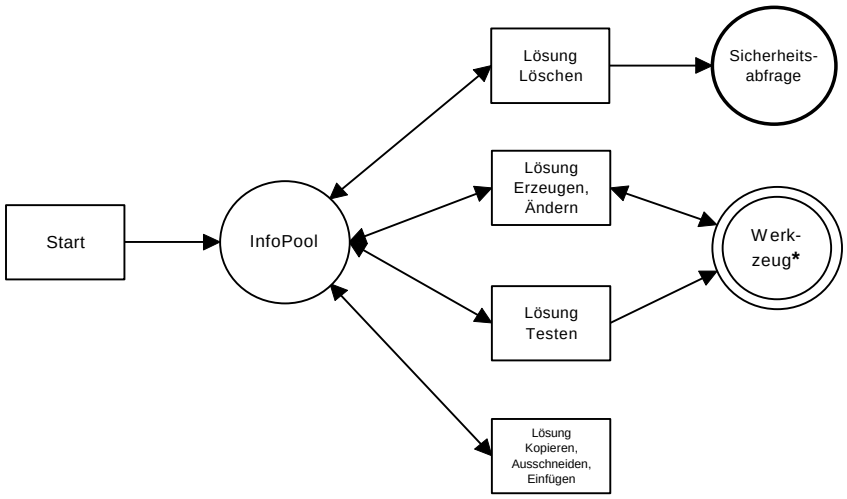


Abbildung 46: Unterdialognetz zur Reuse-, Revise- und Retainphase

Die Reuse-, Revise- und Retainphase
Die Bearbeitungsmöglichkeiten der Reusephase entsprechen im wesentlichen den Methoden, die auf die Objekte der Fallbasis angewendet werden können. Der grundsätzliche Unterschied besteht darin, daß es im Bearbeitungsfenster (*InfoPool*) keine Ordner gibt. Es können nur Prozesse und Funktionen (DataSets) erzeugt, bearbeitet und gelöscht werden. Eine Trennung von Objekten der Fallbasis und denen des Bearbeitungsfensters wurde vorgenommen, um dem Anwender des Systems die Möglichkeit zu geben, neue Informationen zunächst evaluieren zu können, bevor diese persistent in die Fallbasis übernommen werden.

Damit entspricht das Bearbeitungsfenster einer *"temporären"* Ablage von Fallbeispielen, deren Informationen in der Retrievephase solange nicht berücksichtigt werden, bis diese im Rahmen der Retainphase in die Fallbasis übernommen werden. Dennoch werden die Fallbeispiele des Bearbeitungsfensters physisch im Archiv hinterlegt, um die Evaluierung einer Lösung durch Testen innerhalb der Revisephase durchführen zu können, da die verschiedenen Werkzeuge auf diese Archivdaten zugreifen.

Werkzeuge des Informationssystems
Unter der komplexen Stelle *Werkzeuge* werden alle bereits erläuterten Modellierungs- und Darstellungswerkzeuge der Reportingplattform verstanden.

Werden im Rahmen des Testens einer Lösung diese Werkzeuge benötigt, so erfolgt zunächst eine Differenzierung der Lösung nach deren Typ, da jede Lösung ihre individuelle Umgebung benötigt. Geschäftsprozesse werden demnach durch das Prozeßtool, DataSets entsprechend ihrer Darstellungswerkzeuge (Menge, Tabelle, Kreuztabelle, Suche, Graphik) evaluiert.

Durch die Funktion *openObject* wird das zur Darstellung benötigte Werkzeug der Reportingplattform gestartet. Dabei werden (die nicht weiter spezifizierten) Funktionen *startProcessTool* und *startDataSetTool* verwendet.

```
type openObject: Fallbasis×ID
pre-openObject(fb, id)   ≙ true
openObject(fb, id)       ≙
let typ = fb.archiv(id).KnotenDaten.Typ in
if typ = 'Prozeß'
then startProcessTool( id )
else let materialHandle = MatExt_ComfortUnserialize( fb.archiv, fb.archiv(id).Material.RecordID )in
        startDataSetTool( materialHandle )
```

Gilt eine Lösung als geeignet (evaluiert), so kann sie zur späteren Wiederverwendung persistent in die Fallbasis übernommen werden. Dieser Vorgang besteht lediglich in der Bestimmung einer Domain der Fallbasis, dem die Lösung logisch zugeordnet werden kann. Dabei kann es vorkommen, daß sich eine Lösung aus logischer Sicht durchaus mehreren Domains zuordnen läßt, so daß durch die Mechanismen der Übernahmeart (*Kopieren, Ausschneiden, Einfügen bzw. Drag & Drop*) auch dieser Aspekt berücksichtigt wurde.

4.3 ENTWICKLUNG DER BENUTZERSCHNITTSTELLE

Bevor beispielhaft anhand einer Sitzung mit dem DSS-Werkzeug die Funktionalitäten zur Entscheidungsunterstützung der Reportingplattform dargestellt werden, wird zunächst in *Kapitel 4.3.1 Modellierungstools* ein Überblick zur Erstellung von SQL-Templates und den Funktionalitäten des Prozeßeditors gegeben.

4.3.1 Modellierungstools

4.3.1.1 SQL-Templatemodellierung

Die Struktur der in Abbildung 47 dargestellten Hauptmaske zur SQL-Templatemodellierung orientiert sich an der schrittweisen Vorgehensweise eines Modellierers bei der Erstellung eines SQL-Statements. Hierzu wird die Maske im wesentlichen in die einzelnen Bausteine eines SQL-Statements unterteilt, die innerhalb eines Templates parametrisiert werden können. Des weiteren werden alle anderen Fenster über diese Hauptmaske aufgerufen, die mit der Templatemodellierung in Zusammenhang stehen.

Die Registerkarten werden deshalb meist sequentiell aufgerufen. Mit *SQL-Template* wird das aktuelle Template angezeigt, nachdem es über *Template-Attribute* durch die zugehörigen Attribute bestimmt wurde. Im linken Fenster *Wählbare Template-Attribute* in Abbildung 47 werden sämtliche Attribute gelistet, die zur Spezifikation des Templates verwendet werden können. Sobald ein Attribut gewählt wurde, erfolgt im Fenster ein Update, so daß nur noch die, zu den gewählten Attributen im rechten Fenster kompatiblen, Attribute angezeigt werden. In diesem Beispiel werden Attribute für einen Quartalsbericht gewählt.

Gruppenfilter und *Filter (konst.)* ermöglichen die Modellierung der beiden Filtertypen. Über *Subquery* können entsprechende Subqueries bis zur Tiefe zwei erstellt werden.

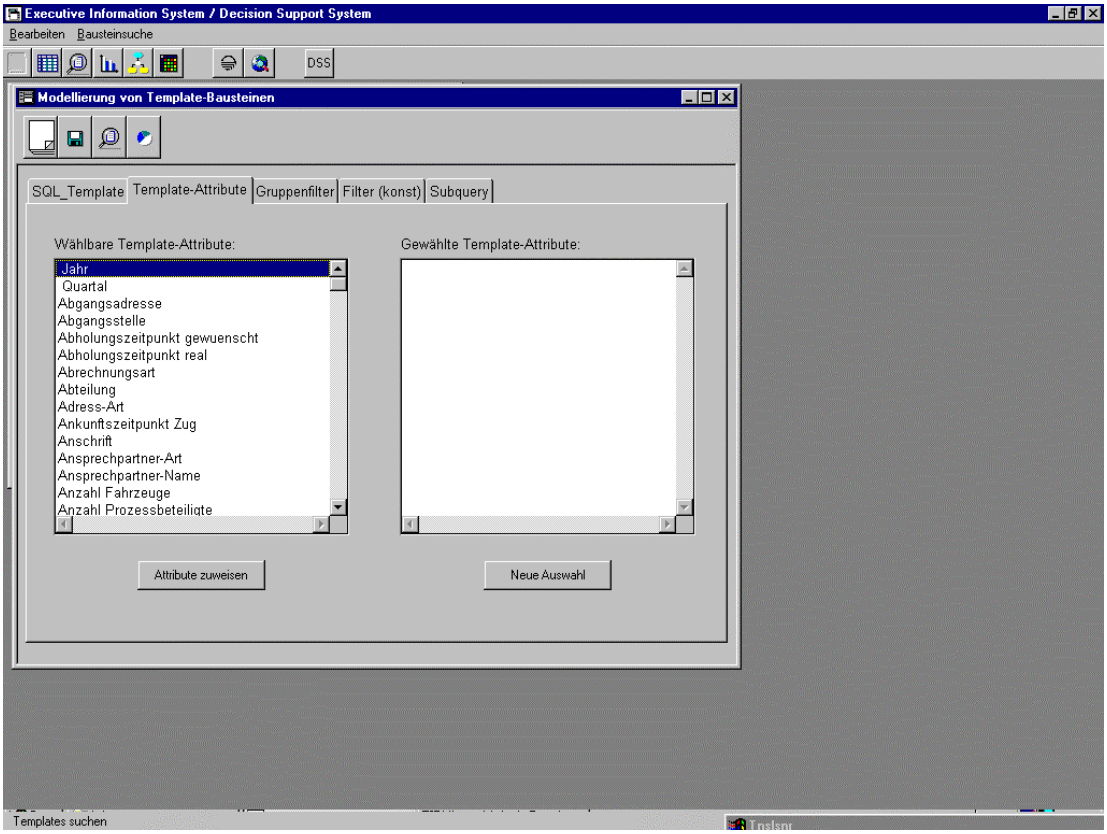


Abbildung 47: Hauptmaske der Templatemodellierung

Über die in Abbildung 48 dargestellte Maske werden die *konstanten Filterausdrücke* für ein Template eingegeben, die sich aus einem *Filterattribut*, einem *Vergleichsoperator* und einem *Vergleichswert* zusammensetzen. In der Combobox *Filterattribute* werden alle zulässigen Filterattribute bereitgestellt, die aus der Hauptanfrage bzw. aus deren Tabellen gewonnen werden können. Anschließend kann aus der Combobox *Operatoren* ein *Vergleichsoperator* für den zu erstellenden *Filterausdruck* ausgewählt werden. Die Art des Vergleichswerts des Filterausdrucks hängt von der Radiobuttoneinstellung ab, die für den Vergleichswert vorgesehen ist. Wenn der Radiobutton *Konstante* aktiviert ist, kann nur eine Vergleichskonstante eingegeben werden. Bei Aktivierung des Radiobuttons *Attribut* kann aus der Combobox *Vergleichswert* ein Vergleichsattribut für den Filterausdruck gewählt werden. In dem Beispiel in Abbildung 48 wurde ein Filter mit einer Konstante als Vergleichswert zur Einschränkung des Templates auf Daten des Jahres 1999 gesetzt. Die RadioButtons *(AND, OR, NOT)* aus der Gruppenbox *Filter-Verknüpfungen* bestimmen den Verknüpfungsoperator zwischen dem aktuellen und dem nachfolgenden Filterausdruck.

Für den Fall, daß bereits erstellte Filterausdrücke geändert werden sollen, kann ein zu ändernder Filterausdruck aus der Combobox *Gewählte Filterausdrücke* gewählt werden, worauf jeder einzelne Bestandteil des gewählten Ausdrucks geändert und neu in das Template integriert werden kann. Es ist darauf zu achten, daß nach der Erstellung der einzelnen Filterausdrücke der Pushbutton *Filter speichern* zu aktivieren ist, da erst zu diesem Zeitpunkt die zuvor temporär integrierten Filterausdrücke dem aktuellen Template tatsächlich zugeordnet werden. Da aber auch bereits gesetzte Filterausdrücke wieder gelöscht werden können, besteht die Möglichkeit über den Pushbutton *Filter löschen* eine *Dialogbox* aufzurufen, über die einzeln ausgewählte Filter aus dem Template gelöscht werden.

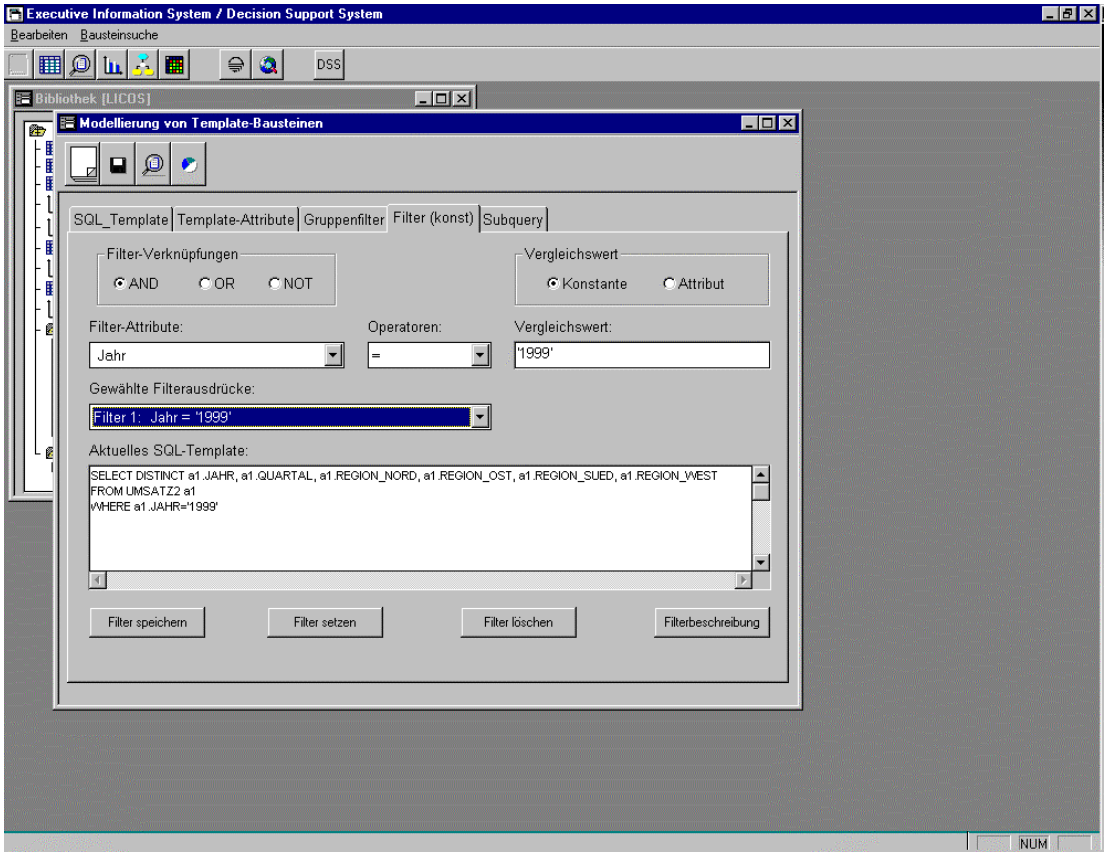


Abbildung 48: Erstellung von konstanten Filterausdrücken

Nach der Spezifikation des Templates muß es einem DataSet zugewiesen werden. Dazu kann es über das Mengentool (s.u.) und dem in Abbildung 49 dargestellten SQL-Editor einem DataSet zugeordnet werden. Die Speicherung des Templates erfolgt innerhalb der Funktionsbereichshierarchie. Bei Bedarf kann im SQL-Editor das Template zusätzlich in ein festdefiniertes SQL-Statement umgewandelt und ebenfalls in einem Funktionsbereich abgelegt werden.

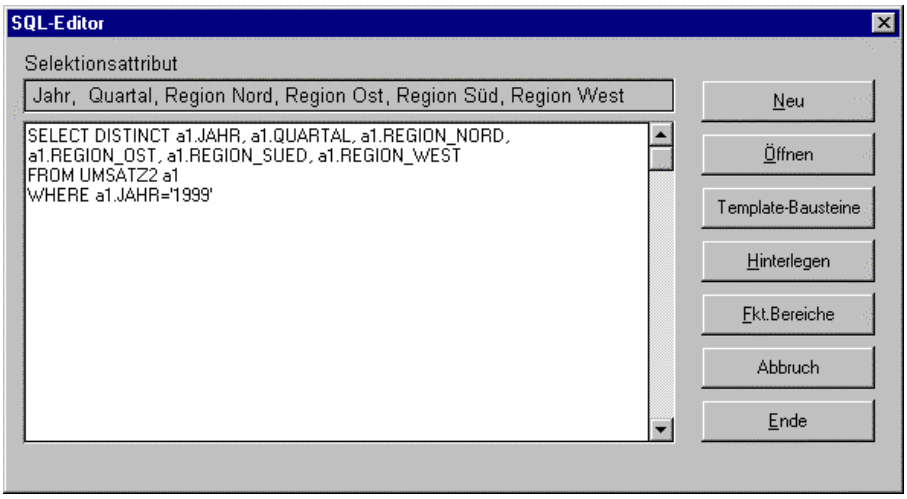


Abbildung 49: Dialogbox zur Anzeige des Statementtyps und dessen Struktur

4.3.1.2 Der Prozeßeditor

Der *Prozeßeditor* ist ein Tool zur Erstellung von Prozeßmodellen in der Syntax der ereignisgesteuerten Prozeßketten, die mit Hilfe des Prozeßwerkzeugs der Reportingplattform abgearbeitet werden können. Alle Funktionen des Editors sind vom Hauptfenster aus erreichbar, welches in Abbildung 50 dargestellt ist.

In der Gruppenbox *Prozeß* befinden sich alle Funktionen zum Verwalten verschiedener Prozesse. Soll ein neuer Prozeß erstellt werden, so muß er nach Mausklick auf die Taste *Neu* zunächst einen Namen und wahlweise eine erläuternde Beschreibung erhalten. Alle auf diese Weise erzeugten Prozesse lassen sich anschließend aus der Prozeßauswahlbox aufrufen, die jeweils den gerade aktiven Prozeß anzeigt. Der Name und die Beschreibung des aktiven Prozesses lassen sich auch im Nachhinein durch Anwahl der Funktion *Bearbeiten* noch modifizieren. Mit der Funktion *Löschen* wird ein Prozeß mitsamt Namen, Beschreibung und sämtlichen Knoten wieder entfernt.

Im Feld *Aktueller Knoten* wird jeweils der gerade ausgewählte Knoten eines Prozesses mit einer textlichen Kurzbeschreibung angezeigt, und in den Auswahllisten *Vorgänger-Knoten* und *Nachfolger-Knoten* sind alle seine direkten Nachbarknoten aufgeführt. Durch Mausklick auf einen dieser Nachbarn wird der ausgewählte zum aktuellen Knoten; so ist es möglich, sich knotenweise durch den Prozeß zu bewegen, um zu einem bestimmten Knoten zu gelangen.

Bis auf den Startknoten eines Prozesses werden neue Knoten immer als Nachfolger des aktuellen Knotens in den Prozeß eingefügt, damit automatisch die Verbindung zwischen den Knoten hergestellt werden kann. Durch Druck auf die Taste *Erstellen* in der Gruppenbox *Nachfolger-Knoten* wird ein solcher neuer Knoten generiert. Dabei ist in einer Dialogbox als erstes der Typ des neuen Knotens festzulegen, um anschließend eine je nach Typ unterschiedliche Beschreibung des Knotens einzugeben. Einzig für den Startknoten und für Ereignisknoten, die direkt auf einen Funktionsknoten folgen, wird diese Beschreibung nicht benötigt und entfällt daher. Der neu eingefügte Knoten findet sich anschließend in der Liste der Nachfolger des aktuellen Knotens wieder.

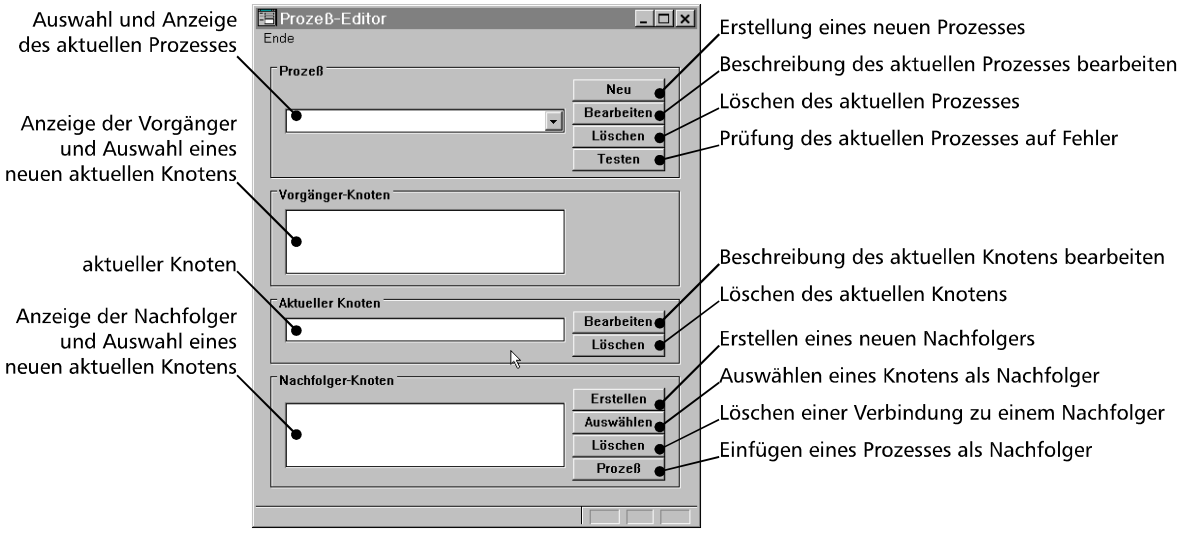


Abbildung 50: Prozeßeditor - Bedienungselemente des Hauptfensters

Die Funktion *Auswählen* ermöglicht das Einfügen eines bereits vorhandenen Knotens als Nachfolger des aktuellen, also das Herstellen einer Verbindung zwischen den beiden Knoten. Die Auswahlmethode für den einzufügenden Nachfolgerknoten korrespondiert mit der für den aktuellen Knoten. In der entsprechenden Dialogbox ist deshalb solange von einem Nachbarknoten zum nächsten zu springen, bis der gesuchte Knoten gefunden ist.

Anstelle einzelner Knoten kann mit der Funktion *Prozeß* auch einer der verfügbaren Prozesse an den aktuellen Knoten angehängt werden. Dabei stehen zwei verschiedene Methoden zur Auswahl, wie ein ausgewählter in den aktiven Prozeß eingefügt werden soll. Beim Einfügen als *Referenz* wird lediglich ein Verweis auf die Knoten im Originalprozeß vorgenommen. Eine Änderung dieser Knoten wirkt sich in allen Prozessen aus, in denen sie als Referenz enthalten sind. Das Einfügen als *Kopie* hingegen erzeugt gänzlich neue Knoten, so daß eine Änderung auf den jeweils aktuellen Prozeß begrenzt bleibt.
Die Funktion *Löschen* in der Gruppenbox *Aktueller Knoten* entfernt eben diesen aktuellen Knoten mitsamt seiner Beschreibung aus dem aktiven Prozeß. Diese Löschung ist nur dann zulässig, wenn seine Vorgänger- und Nachfolgerknoten noch über andere Knotenverbindungen in den Prozeß

eingebunden sind. Soll also etwa ein Teilabschnitt eines Prozesses durch einen neuen ersetzt werden, so muß zuerst die neue Knotenfolge definiert werden, bevor die alte gelöscht werden kann.

Die Funktion *Löschen* in der Gruppenbox *Nachfolger-Knoten* dient der Aufhebung einer Verbindung zwischen dem aktuellen Knoten und einem auszuwählenden Nachfolger. Die beteiligten Knoten selbst werden nicht gelöscht, sondern bleiben im Prozeß erhalten. Das Entfernen einer Verbindung ist ebenfalls nur dann möglich, wenn beide Knoten noch über andere Verknüpfungen in den Prozeß integriert sind.

Die Funktion *Bearbeiten* in der Gruppenbox *Aktueller Knoten* gestattet das Editieren der Beschreibung des aktuellen Knotens innerhalb eines typabhängigen Dialogfensters. Beim Erstellen eines neuen Knotens wird der entsprechende Dialog automatisch aufgerufen, da die Eingabe einer Knotenbeschreibung obligatorisch ist.

Für einen Ereignisknoten muß die Art der Ereignisauswertung, also *automatisch* oder *manuell*, erfaßt werden sowie die zugehörige Bedingung für den Ereigniseintritt. Ein Operatorknoten verlangt nach der Festlegung seines Ein- und/oder Ausgangsoperators, und ein Funktionsknoten erhält einen Namen und wahlweise einen erläuternden Zusatztext sowie die Beschreibung der Datenmenge, die später an diesem Knoten angezeigt werden soll.

4.3.2 Sitzungsszenario des DSS-Werkzeugs

Die im folgenden beschriebenen zwei Masken der Benutzerschnittstelle für das Informationsretrieval und die Lösungsadaption geben einen Überblick hinsichtlich der Umsetzung der in den vorigen Kapiteln entwickelten Konzepte. Die Basis hierfür bilden die modellierten DataSets und Prozeßbausteine.

4.3.2.1 Das Informationsangebot

In dem Fenster *Verfügbare Informationen* (linkes Fenster in Abbildung 51) werden alle vorhandenen Systeminformationen in ihren Anwendungsbereichen (*Domains*) graphisch angeordnet, wobei an einem Knoten (Ordner) ähnliche Fälle mit gleichem Abstraktionsniveau abgelegt sind. Dem Anwender ist somit leicht möglich, den für ihn relevanten Informationsbereich zu erkennen. Durch einen Eintrag über die Auswahlbox *Domain* im Suchfenster (rechtes Fenster in Abbildung 51) kann der aktuelle Domainbereich (innerhalb der dem Anwender zugewiesenen Rechte) jederzeit erweitert bzw. eingeschränkt werden.

Der Anwender kann durch den Baum navigieren, um so die ihn interessierenden Informationen zu finden. Innerhalb der Hierarchie läßt sich "per Mausklick" der Detaillierungsgrad jeder Information verfeinern (Spezialisierung) bzw. vergröbern (Generalisierung), so daß die Komplexität der Information ersichtlich wird. So können Subprozesse und die Informationsabfragen an den einzelnen Knoten sichtbar gemacht werden.

4.3.2.2 Informationssuche

Das Suchfenster ist ähnlich der allgemeinen Indexsuche bei Hilfesystemen aufgebaut. In einem *Glossar* werden alle Informationen bezüglich des aktuellen Informationsumfangs und des ausgewählten Informationselements (Attribute, Informationsabfragen, Prozesse) in alphabetischer Reihenfolge angeboten. Das Glossar umfaßt neben den Informationen aus dem Fenster *verfügbare Informationen* auch *Deskriptoren*, die als zusätzliche Beschreibung auf einer abstrakteren Ebene dienen. Jeder Eintrag im Glossar verweist auf n ($n \geq 1$) Einträge im Fenster *verfügbare Informationen*. Das Informationsangebot läßt sich über eine Combobox auf bestimmte Informationselemente beschränken.

Unterhalb des Glossars befindet sich die *Problembox*, in der die Problembeschreibung als Kombination von verschiedenen Informationselementen zusammengestellt werden kann. Zusätzlich können Deskriptoren in die Problembeschreibung aufgenommen werden. Die erstellte Problembeschreibung wird in eine geeignete Ausgangsbasis für die Suchstruktur der Fallbasis transformiert, um über eine Indexstruktur den Lösungsraum eingrenzen zu können. Über die ausgewählten Attribute werden

zugehörige DataSets gefunden, über DataSets werden Prozesse und über DataSets und Prozesse wiederum übergeordnete Prozesse. Basierend auf dieser *Inputmenge* werden mittels der entwickelten Mechanismen des fallbasierten Schließens ähnliche Fälle ermittelt.

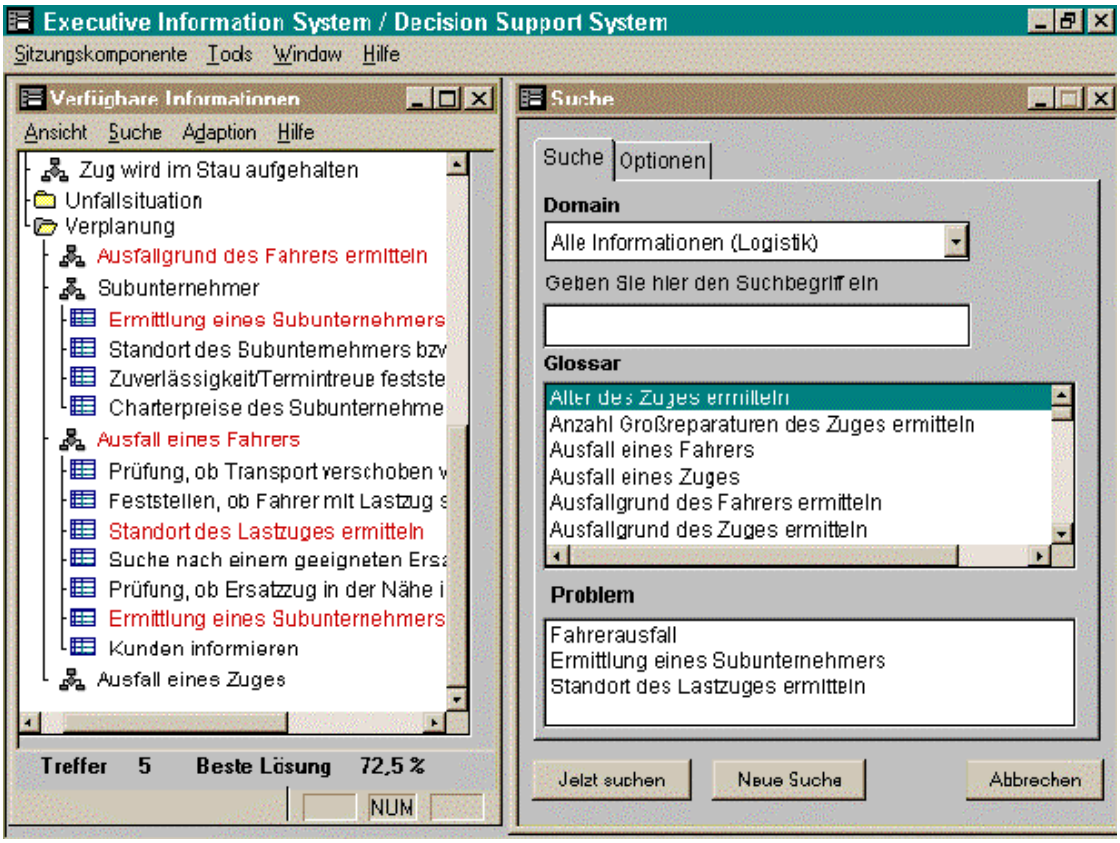


Abbildung 51: Ergebnis einer Suche mittels fallbasierten Schließens

In Abbildung 51 wurden drei Einträge zur Problembeschreibung getätigt (Problembox). Der Eintrag *Fahrerausfall* ist ein *Deskriptor* und verweist auf zwei Prozesse (*Ausfallgrund des Fahrers ermitteln*, *Ausfall eines Fahrers*, die im Fenster *Verfügbare Informationen* jeweils farblich gekennzeichnet werden) mit zunächst gleicher Wahrscheinlichkeit zur Lösung des Problems.

Eine zusätzliche Problembeschreibung durch die Informationsabfragen (*Ermittlung eines Subunternehmers*, *Standort des Lastzuges ermitteln*) ergibt eine neue Lösungsmenge. Die Analyse der Treffer ergibt, daß der Prozeß *Ausfall eines Fahrers* zusätzlich die beiden ausgewählten Informationsabfragen enthält, so daß diesem Prozeß eine höhere Wahrscheinlichkeit zuzuordnen ist. Durch eine besondere Gewichtung der einzelnen Treffer (Informationsabfragen, die innerhalb eines bereits gefundenen Prozesses ermittelt wurden, erhalten ein höheres Gewicht) bildet dieser Prozeß letztlich die beste Lösung bzgl. der Problembeschreibung.

4.3.2.3 Informationsadaption

Das Ergebnis der Suche wird farblich gekennzeichnet im Fenster *Verfügbare Informationen* angezeigt (linkes Fenster in Abbildung 52). Dabei erhält jeder gefundene Fall eine Wahrscheinlichkeit, mit der bestimmt wird, wie gut dieser das aktuelle Problem lösen kann. In einer Statuszeile im unteren Bereich des Fensters werden die Anzahl der gefundenen Lösungen und die Prozentzahl der besten Lösung dargestellt.

Das Ergebnis der Suche kann per Drag & Drop in das *Informationspoolfenster* (rechtes Fenster in Abbildung 52) übernommen und bearbeitet werden. Es ist aber auch möglich, alle gefundenen Fälle automatisch in das Informationspoolfenster zu übernehmen (Menüpunkt Adaption). Bei dieser Übernahmeart werden die gefundenen Fälle kombiniert und automatisch zu einem neuen Fall zusammengestellt. Dabei wird der Fall mit der höchsten Wahrscheinlichkeit als erster angezeigt (hier *Ausfall eines Fahrers*).

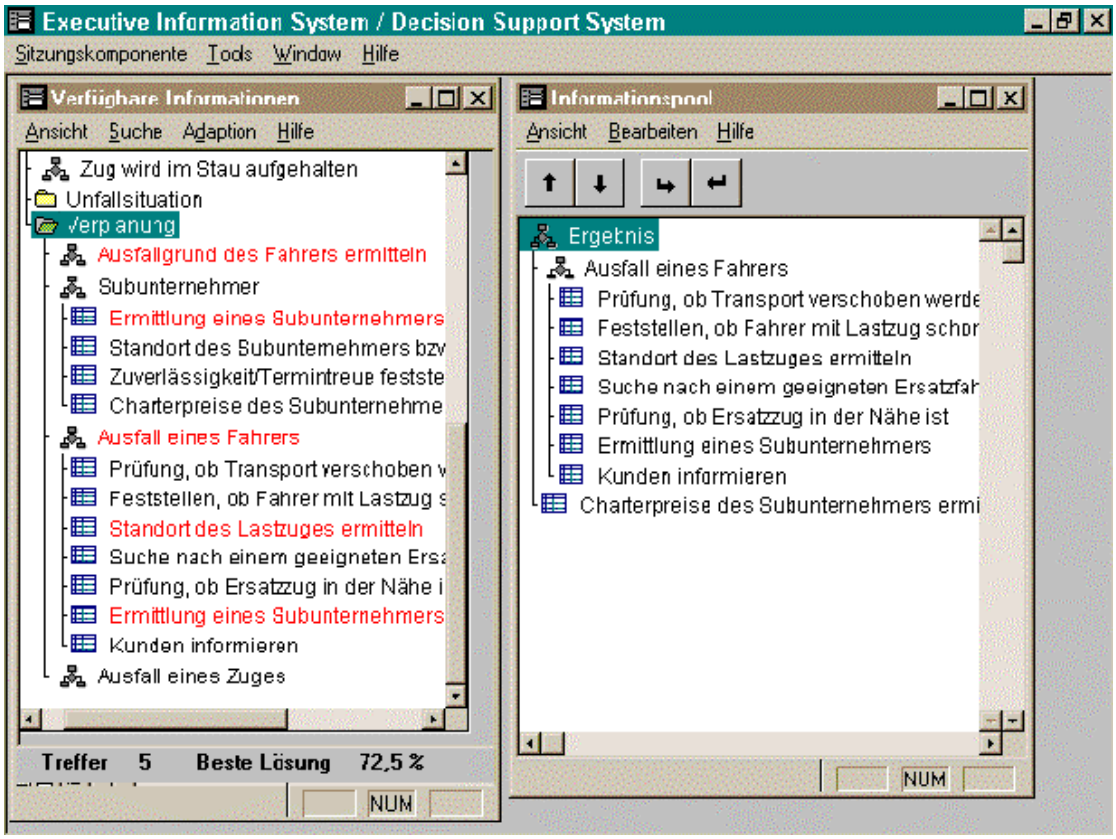


Abbildung 52: Automatische Adaption eines neuen Prozesses

Dieser neue Fall kann anschließend durch den Anwender weiter bearbeitet werden. In Abbildung 52 wurde per Drag & Drop zusätzlich aus dem Prozeß *Subunternehmer* die Informationsabfrage *Charterpreise des Subunternehmers ermitteln* in den Ergebnisprozeß übernommen.

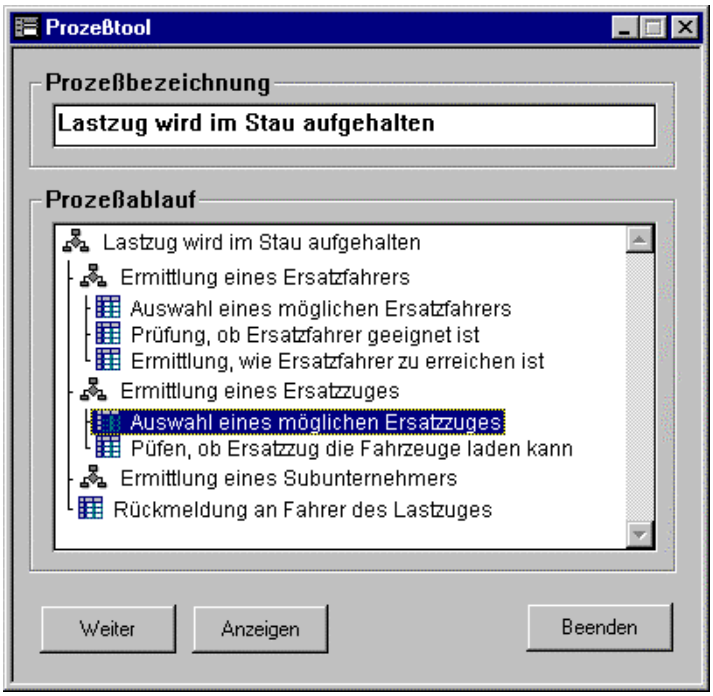


Abbildung 53: Prozeßtool mit Beispielprozeß

Der auf diese Weise neu geschaffene Entscheidungsprozeß kann nun auf Knopfdruck mittels des in der Reportingplattform enthaltenen *Prozeßtools* durchlaufen werden. Dabei kann der Startpunkt durch den

Benutzer gewählt werden. Abbildung 53 stellt dieses anhand einer jederzeit aufrufbaren Ausnahmebehandlung mit dem Beispielprozeß "Lastzug wird im Stau aufgehalten" dar.
Stellt der so adaptierte Prozeß neues Wissen dar, so kann der Anwender diesen per Mausklick in die Fallbasis übernehmen. Hierdurch wird der neu adaptierte Prozeß zu einer weiteren Lösung für zukünftige Problemstellungen.

Beim Programmstart werden die mit dem Repositorymanager modellierten und abgespeicherten Metadaten in den Hauptspeicher geladen und stehen dem Benutzer ständig zur Verfügung. Die Benutzeroberfläche setzt sich aus verschiedenen Werkzeugen (Tools) zusammen, die zur Zusammenstellung und Spezifikation der Informationswünsche sowie der Darstellung der ermittelten Daten dienen. Im folgenden werden die im System zur Verfügung stehenden Werkzeuge kurz erläutert.

4.3.3 Werkzeuge der Analysekomponente

Die Darstellungs- und Analysekomponente stellt dem Benutzer die Oberfläche und Funktionalität zur Verfügung (siehe Abbildung 54), die er für die Informationssuche innerhalb der Reportingplattform benötigt. Mittels einer graphischen Benutzeroberfläche, welche die unter graphischen Betriebssystemen bekannten Bedienungselemente (Maussteuerung, Pull-Down-Menüs, Schaltflächen, etc.) aufweist, kann sich der Anwender auf einfache Art und Weise durch das Informationsangebot navigieren.

Mengentool (DataSet-Modellierung)
Mit Hilfe dieses Werkzeugs kann der Modellierer (oder auch der erfahrenere Anwender) sich sämtliche Attribute, die seinem Informationsbedarf gerecht werden, zusammenstellen und mit einem Namen versehen. Aus einer Auswahlliste (Selektionsliste) können die gewünschten Attribute in eine Zielliste (Definitionsliste) übertragen werden. Bei der Selektion einzelner Attribute wird die Auswahlliste auf die noch zulässigen Attribute eingeschränkt. Filter können direkt eingeben werden.

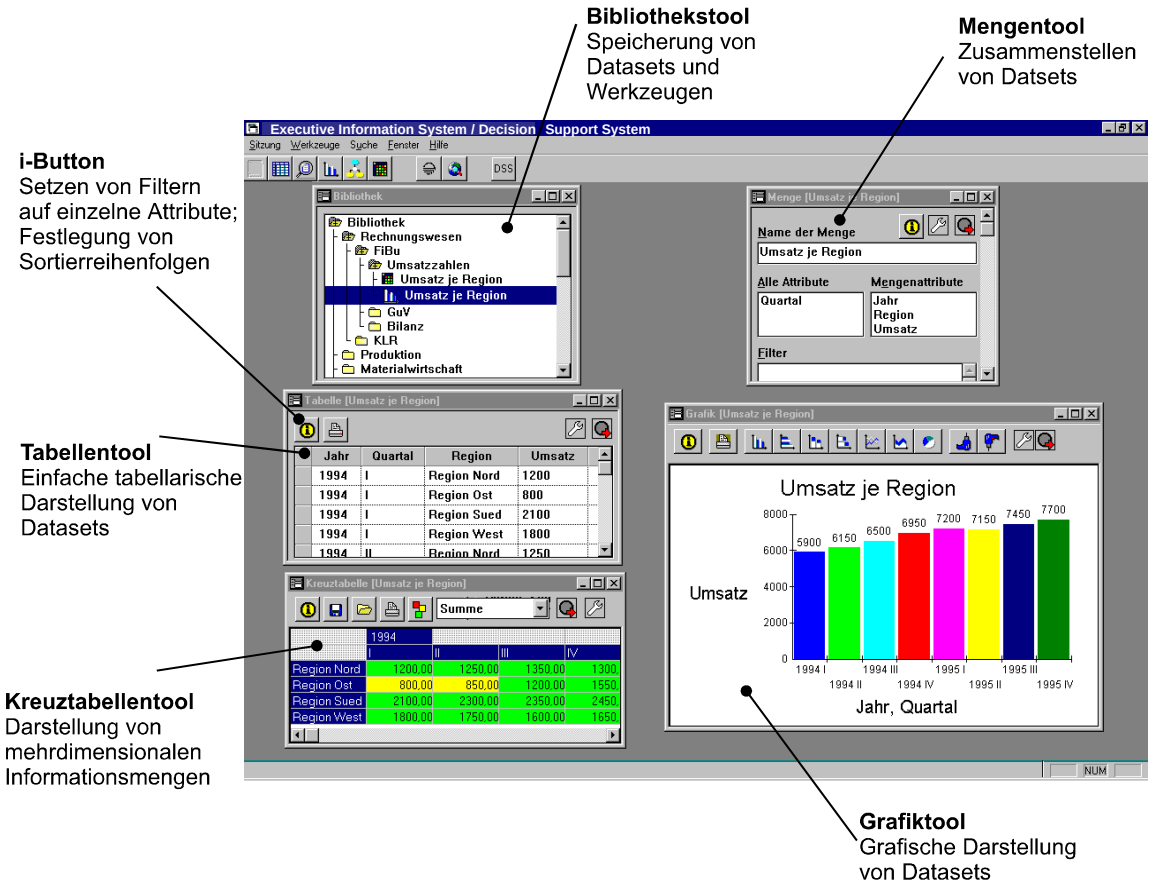


Abbildung 54: Oberfläche der Reportingplattform mit den wichtigsten Werkzeugen

Tabellentool
Dieses Werkzeug stellt den Inhalt einer Datenmenge in einfacher tabellarischer Form dar. Hierbei lassen sich die Tabellenspalten frei anordnen und die Spaltenbreite variabel verändern. Über den i-Button läßt sich ein weiteres Fenster öffnen (siehe Abbildung 55), in dem Attribute mit Filtern belegt, bei Bedarf ausgeblendet und Sortierreihenfolgen festgelegt werden können. So sind individuelle Sichtweisen auf Datenmengen generierbar. Ein mit Filtern belegter DataSet läßt sich wiederum in die anderen Werkzeuge des Systems übertragen. Der i-Button steht in allen DataSet verarbeitenden Werkzeugen zur Verfügung.

Bibliothekstool
Die Bibliothek bietet die Möglichkeit, DataSet-Spezifikationen und Werkzeuge zu speichern. Hat sich der Anwender z.B. ganz bestimmte Informationen aus dem Angebot zusammengestellt und zur Visualisierung ein beliebiges Darstellungswerkzeug gewählt, kann diese Kombination in der Bibliothek abgespeichert und bei Bedarf abgerufen werden. Neben der DataSet-Spezifikation wird auch die Größe und Lage des Werkzeugfensters mit gesichert ("eingefroren"). Zur strukturierten Verwaltung der Objekte ist die Bibliothek in einer Baumstruktur organisiert. Der Benutzer kann sich innerhalb dieser Struktur eigene Verzeichnisse anlegen, in denen er die jeweiligen Objekte anordnet.

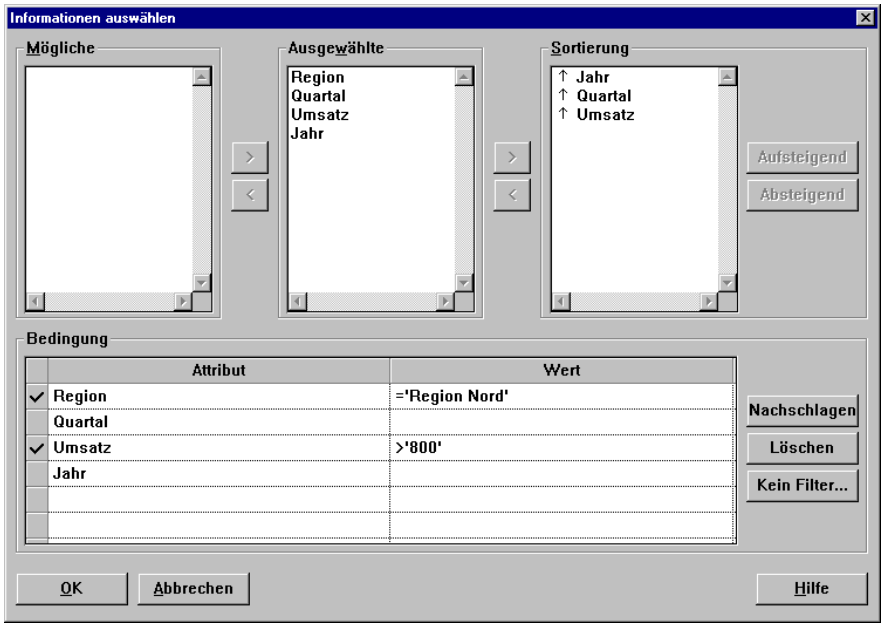


Abbildung 55: Maske zur Festlegung von Informationskriterien (Filter, Sortierreihenfolgen)

Kreuztabellentool
Das Kreuztabellenwerkzeug ermöglicht eine mehrdimensionale Datenanalyse in zweidimensionaler Darstellung. So kann jeder Achse eine oder eine beliebige Kombination von Dimensionen zugeordnet werden. Durch einen Informationsauswahldialog sind durch einfaches Drag & Drop den Achsen die gewünschten Dimensionen zuzuordnen. Genauso lassen sich die Fakten für den Wertebereich einer Kreuztabelle und zusätzlich die gewünschte Auswertungsfunktion auswählen. Als Auswertungen stehen Mittelwert, Maximum, Minimum, Anzahl und keine Auswertung zur Verfügung.

Zur weiteren Unterstützung der Datenbewertung bietet die Kreuztabelle dem Benutzer die Möglichkeit, eine Abweichungsanalyse durchzuführen. In dem dafür vorgesehenen Dialog kann der Anwender den Sollwert sowie den Toleranzbereich einer Größe festlegen. Abweichungen können entweder nach oben, unten oder in beide Richtungen betrachtet werden. Nach der Durchführung der Abweichungsanalyse besteht die Möglichkeit, die Zellen, je nach Lage in bezug auf den Toleranzbereich, farblich zu markieren. Diese farbliche Kennzeichnung im Rahmen des Traffic Lighting (grün: Werte im Sollbereich, gelb: Werte im Toleranzbereich, rot: Werte außerhalb des Toleranzbereichs) verschafft einen schnellen Überblick über die Situation bestimmter Werteentwicklungen.

Des weiteren wurde ein Kennzahlensystem eingeführt [Spir96], welches die Möglichkeit bietet, über einen Kennzahleneditor komplexe Kennzahlensysteme einzupflegen und diese über das Kreuztabellentool darstellen zu lassen.

Graphiktool
Das Graphiktool stellt die Funktionalitäten bereit, DataSets graphisch darzustellen. Es muß darauf geachtet werden, daß mindestens eins der Attribute numerische Werte enthält, da sonst eine sinnvolle graphische Aufbereitung nicht möglich ist. Als Diagrammarten werden Balken-, Säulen-, Linien-, Torten- und Punktgraphiken angeboten. Der Wechsel der Darstellungsart geschieht durch einfachen Mausklick. In einem Auswahldialog können die Attribute dem Wertebereich oder dem Dimensionsbereich frei zugeordnet werden. Ist dem Wertebereich mehr als ein Attribut zugeordnet, wird automatisch eine entsprechende Legende zur Unterscheidung im Diagramm eingefügt. Sind die Attribute im Dimensionsbereich hierarchisch geordnet (z.B. Jahr, Quartal, Monat), können die verschiedenen Aggregationsstufen der Wertattribute durch Mausklick durchlaufen werden (Drill-Down und Roll-Up).

Die Datenmengen können durch Drag & Drop zwischen allen DataSet-verarbeitenden Werkzeugen übertragen werden, so daß ein Wechsel der Darstellungsart (Tabelle ⇒ Diagramm) problemlos möglich ist. In den Darstellungswerkzeugen besteht die Möglichkeit, Abfrageergebnisse auszudrucken. Die Kreuztabelle bietet zusätzlich die Option, die selektierten Daten ins MS Excelformat zu exportieren. Hier können diese weiter bearbeitet und ausgewertet werden.

4.4 ARCHITEKTURSICHTEN DER REPORTINGPLATTFORM

4.4.1 Systemarchitektur des Frameworks

Nachfolgend wird die Systemarchitektur aus abstrakter Sicht dargestellt (Abbildung 56). Ein Modellierer erfaßt mittels der *Modellierungskomponente*, die z.Zt. noch aus verschiedenen Werkzeugen besteht (Mengentool, Prozeßeditor, Repositorymanager, SQL-Templatemodellierer, Modellierer für festdefinierte SQL-Statements), sämtliche Metadaten zu den Geschäftsprozessen, Prozeßtemplates, SQL-Templates, DataSets und Attributen der zu modellierenden Domain im Repository. Diese Metadaten bilden die Grundlage für die spätere Arbeit mit der Reportingplattform.

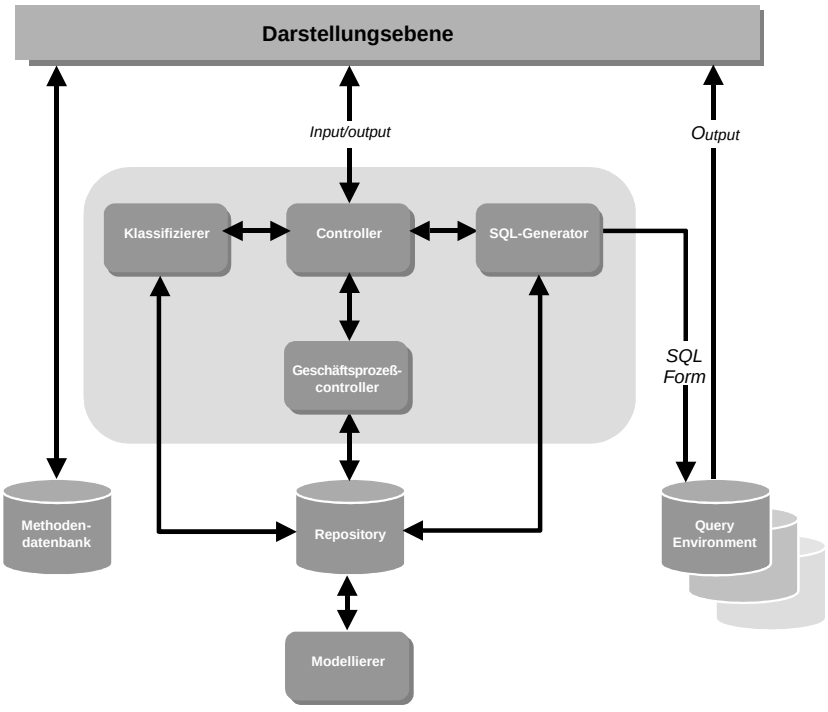


Abbildung 56: Systemarchitektur

Der *Systemkern* umfaßt vier Komponenten zur Sytemsteuerung des Frameworks. Der *Controller* ist für das Monitoring des gesamten Systems verantwortlich und wird bei jeder Transaktion auch der anderen Komponenten mit einbezogen. Jeder Benutzerinput über die Werkzeuge auf der Darstellungsebene wird hier entgegengenommen und direkt an den *Klassifizierer* geleitet, der aufgrund des verwendeten Darstellungswerkzeugs und der eingehenden Daten entscheidet, um welche Art von Input es sich handelt (Problembeschreibung, Auswahl eines Prozeßtemplates oder eines DataSets etc.). Im Falle einer benutzerseitigen Problembeschreibung für das Auffinden von Lösungen in Form von DataSets oder Prozessen stellt der Klassifizierer zunächst sämtliche, mit der Beschreibung übereinstimmende bzw. ähnliche Fälle aus der Fallbasis zusammen und übergibt diese mittels des Controllers an die Darstellungsebene. Sollte der Anwender sich für einen Fall in Form eines Prozesses entschieden haben, der zu durchlaufen ist, so wird diese Information an den *Geschäftsprozeßcontroller* übergeben, der dann den korrekten Ablauf des Workflows überwacht.

An jedem Knoten wird vom Geschäftsprozeßcontroller mit den dort plazierten DataSets erneut der Klassifizierer aufgerufen, der nun aufgrund der zu dem jeweils ausgewählten DataSet das ggf. zugeordnete festdefinierte SQL-Statement oder ein SQL-Template ermittelt. Der DataSet wird dann entweder allein zur Erzeugung eines Defaultstatements oder mit dem zugehörigen festdefinierten SQL-Statement / SQL-Template an den SQL-Generator übergeben, der unter Einbeziehung der Filter das vollständige SQL-Statement erzeugt. Das so generierte *SQL-Statement (SQL-Form)* wird an das *Query Environment* übergeben. Diese ist als zusätzliche Middleware zum Zieldatenbanksystem zu sehen, welches hierdurch unabhängig von seiner Lokation angesprochen wird. Die Ergebnisdatenmenge kann aus Vereinfachungsgründen vom Query Environment direkt an die Darstellungsebene und damit an das gewünschte Darstellungswerkzeug übergeben werden

Sollte der Anwender einen Prozeß zur Laufzeit durch Spezialisierung oder Generalisierung anpassen, so werden die neuen Bausteine über den Klassifizierer gesucht und an den Geschäftsprozeßcontroller übergeben, der diese dann in den laufenden Prozeß integriert. Sollten die Konsistenzbedingungen verletzt werden, so ist über Dialogboxen mit dem Anwender zu kommunizeren, bis eine Integration möglich ist. Sofern an einzelnen Geschäftsprozeßknoten externe Programme aufzurufen sind, wird der Programmname vom Geschäftsprozeßcontroller an den Controller übergeben, der das Programm anschließend mit entsprechenden Parametern (Name, Pfad, Daten etc.) versehen aufruft.

In der *Methodendatenbank* werden verschiedene Analysemethoden bereitgestellt, die bei Bedarf auf die jeweiligen Ergebnismengen angewendet werden können (ABC-Analyse, Gap-Analyse etc.).

4.4.2 Funktionale Architektur

Die Reportingplattform arbeitet mit mindestens zwei Datenbanken, der *Ziel*- und der *Archivdatenbank*. Aus der Zieldatenbank (Unternehmensdatenbank / Data Warehouse) werden die durch die Reportingplattform darzustellenden Datenmengen mittels SQL-Statements ausgelesen. Die Metadaten über die in der Unternehmensdatenbank enthaltenen Relationen / Indexe etc. und der DataSets, Prozesse, festdefinierten SQL-Statements, SQL-Templates sind wie bereits erläutert im Repository abgelegt, welches für die persistente Speicherung seiner Tabellen ebenfalls das RDBMS der Unternehmensdatenbank nutzt. Die Daten des Repositorys werden aus Gründen einer erhöhten Flexibilität in die Archivdatenbank eingelesen und dort gespeichert.

Der Architekturteil *Informationssystem* in *Abbildung 57* umfaßt das *Basissystem* und *Basiswerkzeug*, von denen die jeweils entwickelten Darstellungswerkzeuge ihre Basisfunktionen erben. Die Werkzeuge des Informationssystems greifen nun auf verschiedene *Materialien* sowie weitere *Systembibliotheken* in Form von DLLs zu. Durch diese Trennung in verschiedene Bereiche wird eine hohe Flexibilität in bezug auf zukünftige Erweiterungen gewährleistet. Die Intention dieser Trennung von Quell- und Metadaten bei der Entwicklung der Reportingplattform war, eine möglichst große Unabhängigkeit vom verwendeten Zieldatenbanksystem zu erreichen.

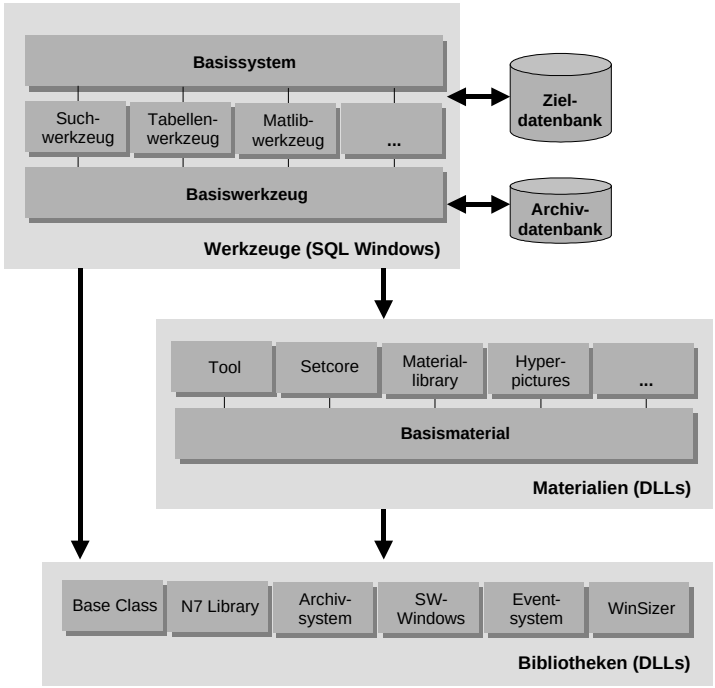


Abbildung 57: Funktionaler Aufbau der Reportingplattform

4.4.3 Das Archiv

Neben den Repositoryinformationen können auch allgemeine Objekte persistent im *Archiv* hinterlegt werden. So werden beispielsweise die Metadaten aller DataSets und ihrer Darstellungswerkzeuge mit Namen, Selektions- und Definitionsattributen, Filtern, Bildschirmpositionen usw. sowie die Bibliotheken für jeden Benutzer des Systems in das Archiv gespeichert. Das Archivsystem ermöglicht alle komplexen Daten mit wenig Aufwand abzubilden und in eine einfach verwaltbare Form zu bringen.

Die Auslagerung des Archivsystems bietet den Vorteil gegenüber anderen Speichermedien, die im System verwendete Datenstruktur abzubilden und persistent zu halten. Dies vereinfacht die Auslagerung erheblich, denn es müssen keine zusätzlichen Datenstrukturen entworfen werden, wie z.B. bei einer relationalen Datenbank oder bei einem Filesystem. Zudem müssen bei Programmerweiterungen keine Datenstrukturen nachgepflegt werden, was die Wartungskosten verringert.

Die Basisstrukturen des Archivsystems bilden *Datensätze*. Datensätze können von der sie verwendenden Applikation konstruiert und ausgelesen werden. Des weiteren können sie nur als Ganzes gelesen oder geschrieben werden. Innerhalb eines Datensatzes ist es vollständig die Aufgabe der Applikation, eine Struktur zu erzeugen und die Daten zu formulieren.

Das Archiv läßt sich in einer beliebigen relationalen Datenbank aufbauen. Dies kann in demselben RDBMS erfolgen, in dem auch die operativen Daten abgelegt werden. Das Archivsystem besteht aus einer zweispaltigen Tabelle. Die erste Spalte ("ID") ist das Schlüsselattribut und dient zur Identifikation eines Datensatzes. Damit dieses benutzerabhängig geschieht umfaßt die ID eine Kennung (IS) + Benutzer (LoginName) + eindeutiger Knotennummer (ID). Hierdurch erhält jeder Benutzer der Reportingplattform seine spezielle Sicht auf das Archiv. In der zweiten Spalte wird die Objektbeschreibung in einem ASCII-Stream abgelegt. Um auch umfangreiche Objektbeschreibungen speichern zu können, ist das Feld vom Typ VARCHAR oder MEMO und somit von einem Typ, der die Speicherung beliebig langer Zeichenketten zuläßt. Da das Repository ebenfalls in das Archiv geschrieben wird, können durch benutzerabhängige Repositories Einschränkungen der Modellierungsmöglichkeiten für die Endanwender erzielt werden.

Es ist erforderlich einem Benutzer neben dem ihm zugeteilten Archivausschnitt, auch den Zugriff auf weitere Daten der Reportingplattform zu ermöglichen. Diese umfassen z.B. die durch die Prozeß-

modellierung anfallenden Entscheidungsprozesse und Funktionen, die für jeden (zugriffsberechtigten) Benutzer verfügbar sein müssen. Diese Verfügbarkeit wird im entwickelten System durch einen dynamischen Wechsel der Archivsicht realisiert. Sobald eine Information dargestellt werden soll, die nicht von dem aktuellen Benutzer erzeugt wurde, erhält dieser automatisch für den Zeitraum der Darstellung das Profil des Informationseigentümers. Durch die Benutzung gemeinsamer Daten ergeben sich allerdings Beschränkungen bezüglich deren Manipulation. Diese Daten können deshalb lediglich gelesen (read-only), nicht aber geändert oder gelöscht werden (write-protect).

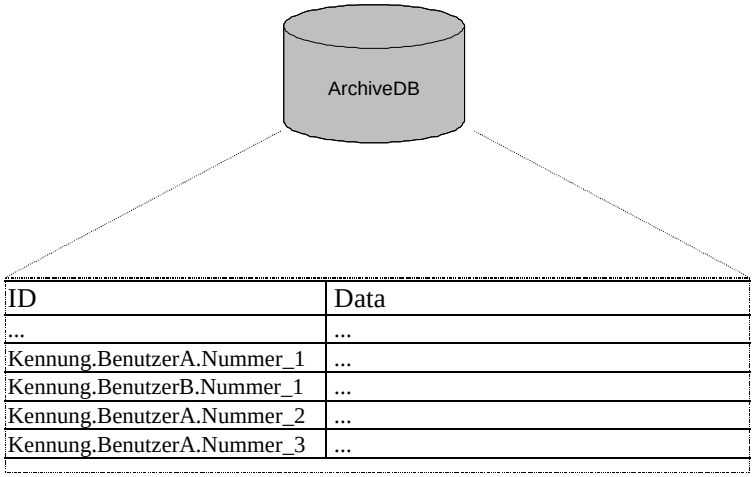


Abbildung 58: Ausschnitt einer Archivtabelle

Wenn Daten über das Archivsystem ausgelagert werden, ist es irrelevant, wo die Daten wirklich gespeichert werden. Die endgültigen Datenquellen sind austauschbar und vor der Anwendung verborgen. Dies erhöht die Flexibilität und Erweiterbarkeit der verwendenden Applikation.

Datensätze können zudem in verschiedenen Formen archiviert und verwaltet werden. In jeder speziellen Form können die Datensätze anders identifiziert und über unterschiedliche Zugriffsarten referenziert werden. Komplexe Problemstellungen wie benutzer- oder mandantenabhängige Konfigurationen und Daten können mit Hilfe des Archivsystems auf einfache Weise gelöst werden.

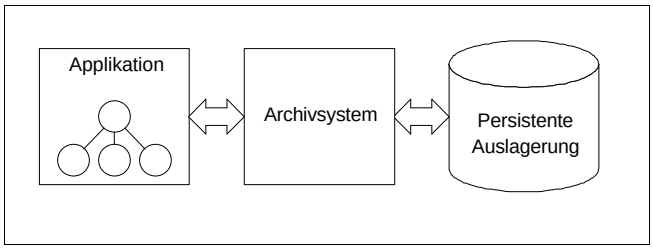


Abbildung 59: Persistente Auslagerung von Objekten in das Archivsystem

Datensätze sind die elementaren Einheiten innerhalb des Archivsystems. Alle Daten werden in dieser Einheit ausgelagert, eingelesen und verwaltet. Die Granularität der Datensätze wird vom verwendenden System bestimmt. Datensätze setzen sich hierarchisch aus Komponenten zusammen. Es gibt drei Formen von Komponenten, aus denen sich alle Daten in ihrer Form auslagern und wieder rekonstruieren lassen:

- *Datencontainer* können eine beliebig lange Zeichenkette aufnehmen. Andere Datentypen müssen von der Applikation in Zeichenketten konvertiert werden. Datencontainer sind die einzige Komponentenart, die konkrete Daten speichert.
- *Tupel* können mehrere Komponenten zu einer neuen zusammenfassen. Jeder der Teilkomponenten kann ein Name innerhalb des Tupels zugewiesen und so beim Auslesen referenziert werden.
- *Listen* ermöglichen es, beliebig viele Komponenten zusammenfügen. Die Reihenfolge, in der die Komponenten zusammengebracht wurden, bleibt erhalten und entspricht der des Auslesens. Mengen können mit Hilfe von Listen realisiert werden, indem die Reihenfolge der Elemente einfach nicht beachtet wird.

Besonders in objektorientierten Programmiersprachen ist es sinnvoll, für die Auslagerung die gleiche Struktur zu verwenden, wie sie auch zur Laufzeit des Systems verwendet wird. Das Entwickeln einer neuen Struktur erhöht den Entwicklungsaufwand meistens nur unnötig. Datensätze haben keine theoretische Maximalgröße. Die verwendende Applikation sollte jedoch eine sinnvolle Größe benutzen, da Datensätze nur als Ganzes verwaltet werden.

Jeder Datensatz kann sich in einen *Datenstream* umwandeln, der dann als Text in der Datenquelle gespeichert werden kann. Genauso kann ein Datensatz seine Komponenten aus einem solchen Datenstream generieren.

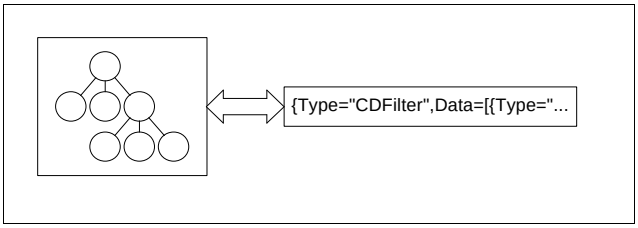


Abbildung 60: Umwandlung eines komplexen Datensatzes in einen Textstream

Als Format für den Stream wurde das ASCII-Format verwendet. Dieses kann von den unterschiedlichsten Systemen eingelesen, gespeichert oder transportiert werden. Jede der drei Komponentenarten *CData*, *CTupel* und *CList* speichert seine Daten auf verschiedene Art und Weise in dem Stream. Anhand des Streams kann eindeutig erkannt werden, um welche Komponentenart es sich handelt:

- Eine Datenkomponente speichert ihre Daten, indem sie sie in doppelten Anführungszeichen einbettet.
- Ein Tupel schließt seine Daten in geschweifte Klammern ein. Jede Teilkomponente wird darin durch Kommata voneinander getrennt gespeichert. Vor jeder Teilkomponente wird der Name der Komponente gespeichert, gefolgt von einem "=". Danach folgt die Teilkomponente.
- Eine Liste wird in eckige Klammern eingebettet. Darin enthalten sind alle Teilkomponenten der Liste, getrennt durch Kommata.

Streams sind eine rein interne Repräsentation von Datensätzen, die dem Entwickler nur für Verständnis- und Debuggingzwecke bekannt werden sollte. Die Funktionalitäten des Archivs werden insbesondere zum Einfrieren einzelner Sitzungsstände genutzt, da so Werkzeuge und Materialien serialisiert persistent gemacht werden können.

4.5 ANWENDUNGSERFAHRUNGEN

4.5.1 Vorgehensweise bei der Modellierung

Die Modellierung der Domain erfolgt durch einen IT-Spezialisten, der sich mit der Funktionalität der Reportingplattform detailliert auskennt und gleichzeitig über Erfahrungen in der interviewbasierten Wissensextraktion von Anwendern ohne weiterführende IT-Erfahrungen verfügt. Dazu muß im Vorwege das Vokabular der Anwendungsterminologie exakt festgelegt werden. Es ist ebenfalls zu fixieren, wie mit Erweiterungen bzw. neu zusammengesetzten Informationsbausteinen hinsichtlich der Bennennung zu verfahren ist, da dies die einzige Referenzierungsmöglichkeit darstellt. Zusätzlich ist eine *Taxonomie* zu entwickeln, die bei der Vielzahl der zu modellierenden Bausteine eine Gruppierung aufgrund der fachlichen Zusammengehörigkeit ermöglicht. Dies hat zwar keine Auswirkungen auf die Möglichkeit des Wiederauffindens, erleichtert jedoch den Umgang und die Pflege mit der modellierten Domain im täglichen Geschäft, da auch das Abspeichern neuer Lösungen möglichst strukturiert erfolgen sollte.

Diese außerhalb der Reportingplattform umzusetzenden Beschränkungen ermöglichen es dem späteren Anwender, die modellierten Informationsartefakte zu strukturieren und die Benennung neuer Komponenten möglichst anhand der festgelegten Vorgaben durchzuführen, ohne ihn aber in der Freiheit bzgl. der anderweitigen Verwendung einzuschränken.

Datensicht
Nachdem diese Zusicherungen mit dem späteren Anwender abgestimmt wurden, können im ersten Schritt die Metadaten (Bezeichnungen, Aggregationen, Links zwischen Tabellen etc.) zu den zu modellierenden Attributen / DataSets entworfen werden. Komplexere Abfragen, die nicht über DataSets und den SQL-Generator abzubilden sind, werden durch festdefinierte SQL-Statements umgesetzt.
Im nächsten Step erfolgt das Clustern ähnlicher SQL-Statements zur Modellierung der einzelnen SQL-Templates, die wiederum einzelnen DataSets zugewiesen werden.

Prozeßsicht
Nachdem die Datensicht in der ersten Version modelliert wurde, geht es daran, die verschiedenen Entscheidungsprozesse zu spezifizieren. Dazu werden mit den Anwendern aufgrund von Use Cases verschiedene Szenarien lokalisiert, aus denen wiederum mittels (intuitiver) Abstraktionsmechanismen versucht wird, allgemeingültige Prozeßbausteine in kleinster Granularität (*Elementarprozesse*) zu erzeugen. Aufgrund der Verwendung von EPK können auch nicht IT-versierte Anwender relativ zügig die einzelnen Prozeßschritte darstellen und neue Abläufe integrieren bzw. Sub- und Hauptprozesse beschreiben. Nachdem die Elementarprozesse in Form von EPK existieren erfolgt die Zusammenführung zu komplexeren Prozessen, die systemseitig in Form der Prozeßtemplates abgelegt werden.
Dieser Modellierungsprozeß sowie die Modellierung der Datensicht erfolgt nicht sequentiell sondern iterativ, da im Rahmen der Modellierungsphase seitens der späteren Anwender immer wieder Anforderungen bzgl. neuer Daten und (Sub-) Prozesse gestellt werden, so daß prinzipiell von einem *integrierten Entwurf* gesprochen werden kann.

Nachdem diese beiden Sichten modelliert wurden, wird dem Anwender die Domain in Form der bereits erwähnten Basisbibliothek zur Verfügung gestellt.

Die Dauer der Modellierung hängt von der Komplexität der jeweiligen Domain ab sowie von der verfügbaren Zeit der einzelnen Mitarbeiter. Die Modellierung der Datensicht für die EIS-Komponente in der Beispieldomain benötigte ca. 20 Manntage, innerhalb denen auf das bestehende Datenbankschema aufgesetzt werden konnte. Zusätzliche Anforderungen sowie Schnittstellen zu externen Systemen, die eine Erweiterung um zusätzliche Tabellen notwendig machten, führten zu einer entsprechenden Verlängerung.

Die Modellierung der Prozeßsicht der EIS-Komponente war aufgrund der gewünschten statischen Prozesse mittels Briefing Book Funktionalität für eine erste Version innerhalb weniger Tage modelliert.

Umfangreicher hat sich die Modellierung der Prozesse im Rahmen des prototypischen Einsatzes im Bereich der LKW-Disposition erwiesen, in der die Ausnahmebehandlung i.a. die Regel bildet. Insbesondere die Lokalisierung der Elementarprozesse und das Zusammenfügen zu komplexeren Entscheidungsprozessen erwies sich als sehr zeitaufwendig, da permanent weitere notwendige Ausnahmebehandlungen eruiert wurden. Durch die bottom-up Vorgehensweise konnte jedoch sehr strukturiert vorgegangen werden und die bereits modellierten Komponenten konnten so in neu lokalisierten Prozessen auf höherem Level eingefügt werden.

4.5.2 Voraussetzungen der Anwender

Der Anwender der Reportingplattform muß über Erfahrungen in seiner modellierten Domain verfügen und insbesondere mit der vereinbarten Struktur der verwendeten Terminologie sowie der Taxonomie für die verschiedenen Artefakte vertraut sein. Zur Erläuterung der einzuhaltenden Struktur ist eine kleine Schulung notwendig. Der Anwender muß anschließend aufgrund der Klassifizierung der Informationen, die sich in der Eingruppierung auf unterschiedlichsten Ebenen widerspiegelt, einen Überblick hinsichtlich der Gesamtstruktur bekommen.

Neben dieser fachlichen Einweisung, die aufgrund der gemeinsamen Modellierung relativ kurz ist, benötigt der Anwender eine Schulung zur Funktionsweise des Gesamtsystems. Durch die einfach zu bedienenden Darstellungswerkzeuge, mit durchgängiger, gleicher Funktionsweise, die die Interaktion

zwischen Anwender und System auf Point & Click reduziert, ist der Umgang mit der Reportingplattform innerhalb eines Tages erlernbar, sofern der Anwender im Umgang mit einem Personalcomputer und Windowsoberfläche vertraut ist.

Bei der Zusammenstellung der Problemspezifikation wird er systemseitig dahingehend unterstützt, daß er verschiedene Sichten auf die modellierten Artefakte nutzen kann, um den Informationsumfang zunächst weiter einzuschränken. Bei UND-verknüpften Problemspezifikationen erhält der Anwender auch über das Glossar nur ein verbleibendes Informationsangebot noch kompatibler (also aufzufindender) Artefakte, so daß die Überschaubarkeit verbessert wird. Diese Funktionalitäten ermöglichen die schrittweise Einarbeitung für die Anwender.

4.5.3 Verwendung der Reportingplattform

Die Reportingplattform befindet sich mit der EIS-Komponente für die Geschäftsführung eines international tätigen Unternehmen aus dem Logistikbereich im Einsatz. Dazu wurde die Domain mittels DataSets und Metadaten, festdefinierten SQL-Statements für besonders komplexe Abfragen sowie SQL-Templates modelliert und in Form einer Basisbibliothek den Anwendern zur Verfügung gestellt. Prozesse wurden mit Briefing Book Funktionalitäten zunächst relativ statisch modelliert. Die Modellierung komplexerer Prozesse erfolgte in einem zweiten Step. Der Zugriff wurde gleichzeitig auf ein Data Warehouse und die operativen Datenbestände ermöglicht.

Bei der Verwendung der Reportingplattform hat sich gezeigt, daß die hohe Flexibilität bei der möglichen Zusammenstellung neuer Informationsabfragen zu einer guten Benutzerakzeptanz, auch bei den weniger versierten Anwendern, führte. Insbesondere die Funktionalität, die DataSets auf einfache Art und Weise dem jeweiligen Informationsbedarf anpassen zu können, indem zusätzliche Attribute mit aufzunehmen oder wegzulassen sind, erhielt Zuspruch. Die transparente Verwendung von SQL-Templates zur dynamischen Generierung des jeweiligen SQL-Statements hat sich als mächtiges Konzept erwiesen, um einen umfangreichen Informationsbedarf zu modellieren und dem Anwender für die jeweils problembezogene Verwendung zur Verfügung zu stellen.

Die Möglichkeit der Übergabe der DataSets per Drag & Drop von einem Darstellungswerkzeug zum nächsten sowie die damit verbundene dynamische Zuweisung von Filtern wurde zum Durchhangeln durch ganze Datenbestände genutzt. Sofern z.B. im Tabellentool einige Zeilen markiert sind und der DataSet an ein weiteres Werkzeug übergeben wird, erfolgt eine Einschränkung auf die markierte Datenmenge mittels eines zuzuweisenden Filterausdrucks. Dieser wird anschließend in das jeweilige SQL-Template integriert.
Die Modellierung der SQL-Templates führte zu einem geringen Wartungsaufwand, da sich 90 % der zukünftig gewünschten Informationsabfragen durch die Kombination der verschiedenen Templatebausteine abbilden ließen.

Allerdings hat sich die SQL-Templatemodellierung als umfangreiche Aufgabe erwiesen, da hierfür trotz guter systemseitiger Unterstützung ein besonders geschulter SQL-Experte benötigt wurde, der einen Überblick zum Datenmodell der Domain besitzt. Die spätere Flexibilität des Systems wird so mittels eines hohen Aufwands für die Modellierung "erkauft". Der Modellierer sollte deshalb zukünftig eine noch weitergehende Unterstützung seitens der Modellierungswerkzeuge erhalten, die ihn so von allen manuellen Tätigkeiten befreien.

Der Piloteinsatz des DSS-Werkzeugs erfolgte in einer LKW-Disposition, in der die Ausnahmebehandlung i.a. die Regel bildet. Hierbei war die Modellierung ungleich aufwendiger, da neben den DataSets und den SQL-Templates komplexere Entscheidungsprozesse lokalisiert, die optimale Granularität ermittelt und als Bausteine im System abgelegt werden mußten. Die Spezifikation der Problemstellungen über das Glossar kann sehr schnell und bequem erfolgen. Ebenso waren die gefundenen Lösungsvorschläge mit den jeweiligen Ähnlichkeitsmaßen mit der gewünschten Performanz und Qualität ermittelt worden. Dies führte bei den Anwendern bzgl. der angebotenen Funktionalitäten zu der gewünschten Akzeptanz. Die Speicherung neuer Lösungen ließ die Fallbasis schnell um ein Vielfaches des ursprünglichen Umfangs anwachsen.

Das System hat sich als robust erwiesen, da es selbst bei der Zusammenstellung inkompatibler Prozeßbausteine eine Auflösung dieser Konflikte durch die Einbeziehung des Anwenders über Dialogboxen ermöglichte. Die Suche nach Lösungen zu Problemstellungen erfolgte schnell und die Speicherung neuer Lösungen in der Fallbasis war ebenfalls unproblematisch. Durch die adäquate Informationsbereitstellung konnte der Zugriff auf Datenbestände für die operative Abwicklung ebenso wie für komplexere Auswertungen erfolgen. Die eigenständige Adaption der Abfragen / Prozesse wurde grundsätzlich ebenfalls für gut befunden.

Durch den pilotierten Einsatz der Reportingplattform wurde sich zunächst auf einen Teil (ca. 50 Prozent) der eruierten Dispositionsprozesse beschränkt. Durch die starke Integration der Anwender in der Modellierungsphase waren diese mit dem Vokabular der Terminologie vertraut und konnten die Probleme zügig spezifizieren. Nur durch diese Vertrautheit mit der Begriffswelt kann die Semantik der einzelnen Artefakte schnell genug aufgefaßt werden, um zeitnah auf Problemstellungen reagieren zu können.

Die Funktionsfähigkeit der Reportingplattform liegt zusätzlich darin begründet, daß der Anwender einen hohen Freiheitsgrad hinsichtlich der Kompositionsmöglichkeit hat. In Zusammenhang mit dem notwendigen Anwenderwissen über die modellierte Domain ermöglicht die Reportingplattform die Lösung verschiedenster Problemstellungen. Weitere Voraussetzung für die Verwendbarkeit war, daß neue Prozeßbausteine nicht auf Zustände vorhergehender Prozesse aufsetzen müssen und ebenfalls kein Datenfluß durch die Prozesse zu realisieren war, da Werkzeuge und DataSets jederzeit getauscht werden können.

Hierdurch wird selbst die schnelle Lösung konkreter telefonischer Anfragen möglich. Allerdings ist eine umfassende Integration der operativen Anwendungssysteme, inklusive eines Datenflusses, notwendig, um insbesondere kurzfristige Probleme des Tagesgeschäfts zu lösen. Sofern dies nicht der Fall ist, muß seitens der Anwender gleichzeitig mit mehreren Systemen gearbeitet werden und die Anwendungsumgebung wird gesamtheitlich zu unflexibel. Aus diesen Gründen ist die Nutzung der Reportingplattform auf dem jetzigen Entwicklungsstand eher für komplexere Problemstellungen mit langfristiger Lösungsfindung über mehrere Tage geeignet. Dieses macht jedoch eher 20 – 30 Prozent des Tagesgeschäftes in einer LKW-Disposition aus. In anderen Branchen, wie z.B. Containerreedereien wäre dies sicher ein höherer Anteil.

Im "rauhen Tagesgeschäft" der Disposition wurde zusätzlich evident, daß die Suche nach Lösungen und eine anschließende Adaption dieser Bausteine relativ genau durch den Anwender begleitet werden mußte und sich die zunächst entwickelte DSS-Benutzeroberfläche nicht optimal für den Alltag eignet. Es muß sehr sorgfältig überwacht werden, wenn aus einzelnen Prozeßbausteinen neue Gesamtprozesse zusammengesetzt werden und diese mit zusätzlich gewünschten Informationsabfragen, sowie weiteren notwendigen Anpassungen zu versehen sind. Des weiteren war das Zusammenspiel zwischen Prozeßwerkzeug und DSS-Werkzeug nicht vollständig transparent, so daß diese Funktionalitäten in einer zukünftigen Version des Systems zusammenfließen sollten. Dies geht konform mit dem Wunsch der Anwender bei Bedarf eine verbesserte Visualisierung der einzelnen Entscheidungsprozesse zu erhalten. Das neu zu implementierende integrierte Werkzeug sollte deshalb direkt nach Auffindung der möglichen Lösungsmengen bzw. nach deren Adaption den Prozeß durchlaufen können, ohne daß zwischen zwei Werkzeugen hin- und hergeschaltet werden muß. Die hierfür zu entwickelnde ergonomische Benutzeroberfläche stellt sicher eine Herausforderung dar.

Der Vorteil des Archivs als einfach zu strukturierendes, persistentes Speichermedium hat sich bei dem stark angewachsenen Volumen der Fallbasis durch ein überproportionales Ansteigen der Ladezeit beim Programmstart bemerkbar gemacht, da die CBR-Indexstrukturen und die benutzerspezifischen Daten aus dem Archiv in den Hauptspeicher geladen werden.

Es ist festzuhalten, daß die Implementierung des entwickelten Templatekonzepts bzgl. SQL- und Prozeßbausteinen zu den gewünschten Ergebnissen bzgl. der flexiblen Verwendbarkeit der Komponenten sowie der Wiederverwendung von (Programm-) Code geführt hat. Insbesondere die Realisierung der Prozeßtemplates als *Components* ermöglichte die hohe Flexibilität bei der Adaption,

wobei die verwendete Entwicklungsumgebung (4GL-Sprache) nicht die vollständige Umsetzung in Form einer stabilen und damit im Unternehmen einsatzfähigen Applikation zuließ.

Die Integration des CBR-Konzepts ermöglichte die Entwicklung einer lösungsorientierten Benutzerunterstützung für das erfahrungsbasierte Problemlösen.
Im Rahmen der prototypischen Umsetzung konnten die entwickelten Mechanismen dahingehend evaluiert werden, daß sich das System in weiteren Entwicklungsstufen zu der gewünschten Produktreife ausbauen läßt, da die Benutzerakzeptanz hinsichtlich der Konzepte gegeben ist.

5 WEITERENTWICKLUNG DER REPORTINGPLATTFORM

5.1 AKTUELLER STAND

Das im Rahmen dieser Arbeit entwickelte Konzept verwendet Design Patterns zur Modellierung von Komponenten im Sinne von Componentware. Das dazu konzipierte Framework steuert die Abläufe bzgl. der Komposition, Kopplung und Kooperation der Komponenten. Die Interaktion zwischen den Komponenten ist so flexibel gehalten, daß auch zur Laufzeit noch Adaptionen und Änderungen in der Komposition der Komponenten erfolgen können, selbst wenn Komponenten in aggregierter Form existieren. Hierzu wurden Mechanismen entwickelt, die eine Übergabe der im Sourcecode vorliegenden Komponenten an das Laufzeitsystem erst bei Aufruf der (auch jeweiligen Sub-) Komponente ermöglichen. Damit das Gesamtsystem überschaubar bleibt, sind die einzelnen Komponenten durch Parametrisierung an die jeweiligen Verwendungszwecke anpaßbar.

Nachfolgend wird anhand der in Kapitel *3.3.2 Rahmenbedingungen zur Verwendung von Componentware* dargestellten drei Sichten auf ein Informationssystem die erfolgreiche Umsetzung der o.g. Konzepte erläutert. Gleichzeit wird gezeigt, daß die in *Kapitel 1.2 Anforderungen an die Reportingplattform* gestellten Anforderungen an das zu entwickelnde System erfüllt wurden.

Systemsicht:

Die Umsetzbarkeit der vorgenannten Konzepte wurde im Rahmen der Entwicklung der Reportingplattform beispielhaft gezeigt. Mittels des Design Patterns *Template* wurden SQL-Statements und Prozesse über eine Skriptsprache als Komponenten modelliert. Aufgrund einer benutzerfreundlichen Oberfläche wurde dem Anwender kein direkter Zugriff auf die SQL-Templates ermöglicht, sondern indirekt über das DataSet-Konzept. Die DataSets wurden wiederum in einer Templateform modelliert (Attribute und Filter), so daß die Übersetzung von DataSet zu SQL zweistufig erfolgt.

DataSets enthalten Attribute und Filter, Prozesse enthalten DataSets, externe Programme und andere Prozesse. Durch die Modellierung als Komponenten können diese nun flexibel miteinander kooperieren. Auch die Komposition der verschiedenen Prozeßkomponenten ermöglicht eine Adaption zur Laufzeit. Jede Komponente wird erst bei Aufruf an das Laufzeitsystem übergeben, so daß zuvor noch Möglichkeiten der Adaption des Gesamtprozesses z.B. durch Entfernen oder Hinzufügen weiterer Prozeßbausteine oder DataSets / externer Programme möglich ist. Hierdurch kann zeitnah auf Exceptions reagiert werden. Die Reportingplattform bildet mit ihrem Systemkern das Framework zur Ablaufsteuerung. Die Templates enthalten jeweils festdefinierte Bausteine, die nicht veränderbar sind, sowie einen durch Parameter anpaßbaren Teil. Somit muß nicht für jede Problemstellung eine neue Komponente modelliert werden, was wiederum zu einer erhöhten Wiederverwendung beiträgt.

Die über verschiedene Modellierungstools spezifizierten DataSets und Prozesse bilden mögliche Lösungen für einzelne Domains. Über eine Problemspezifikation seitens des Anwenders ermittelt die Reportingplattform über Mechanismen des fallbasierten Schließens mögliche Lösungen, die mit einer Bewertung der Relevanz (Ähnlichkeitswert) zur gesuchten Lösung versehen sind. Die so vorgeschlagenen Lösungen können seitens des Anwenders adaptiert werden. Bei Bedarf speichert der Anwender die neuen kompositen Lösungen, so daß sie bei neuen Problemspezifikationen mit in die Lösungssuche einbezogen werden und hierdurch ein "lernendes System" entsteht.

Anwendungssicht:

Durch die Bereitstellung einer Modellierungskomponente und der damit gegebenen Möglichkeit verschiedenste Lösungen in Form von DataSets und Prozessen zu entwerfen, werden die Anforderungen aus Anwendungssicht an die Reportingplattform erfüllt. Die Kombinations- und Adaptionsfunktionalitäten für Komponenten in Verbindung mit der Unterstützung bei der Suche nach ähnlichen Lösungen ermöglichen die Verwendung der Reportingplattform selbst in Domains mit teilstrukturierten Problemstellungen. Die Datensicherheit wird durch Mechanismen im Repository sowie auf Ebene des RDBMS gewährleistet.

Benutzersicht:

Die Bereitstellung geeigneter und besonders ergonomisch gestalteter Werkzeuge zur Aufbereitung und Verarbeitung von Daten und Prozessen, die sich an den Funktionalitäten der EIS-Systeme orientieren, werden den Bedürfnissen des Anwenders zur flexiblen, erfahrungsbasierten Problemlösung gerecht. Der Benutzer kann in der Terminologie seines Anwendungsbereichs mit der Reportingplattform interagieren. Hierfür sind keine systemspezifischen Kenntnisse notwendig, da der Anwender stets auf der Ebene des Fachkonzepts mit dem System kommuniziert. Dazu werden den modellierten DataSets und Prozessen "sprechende" Namen in der Terminologie der Domain zugewiesen, die bei Bedarf angepaßt werden können. Durch einfaches Point & Click können Problemspezifikationen zusammengestellt werden, oder aber sich durch Datenbestände durchgehangelt werden.

Sofern bei der Adaption z.B. Prozesse nicht zusammengefügt werden können, wird der Anwender über Dialogboxen mit in die Lösungsfindung einbezogen. Diese Robustheit des Systems führt zu einer erhöhten Akzeptanz bei den Anwendern. Die Speicherung gefundener Lösungen und die Einbeziehung dieser in die Suche zukünftiger Lösungen induziert beim Anwender den Gedanken, daß jeder Arbeitsschritt mit dem System eine Erleichterung in der Zukunft bedeutet. Die Möglichkeit der schnellen Reaktion auf Exceptions zur Laufzeit hebt das System von anderen ab. Der aktuelle Stand bzgl. der Problemlösung oder der Abfragen kann über eine Serialisierung in das Archiv persistent gemacht werden, um z.B. im Rahmen einer notwendigen Ausnahmebehandlung eine andere Sitzung durchzuführen oder das System zu verlassen. Die Wiederherstellung der verschiedenen Stände ist jederzeit möglich.

Allgemein:

Die Orthogonalität ist bei den Prozeßtemplates gewährleistet. Sofern keine Konsistenzbedingungen bei der Komposition von Prozessen verletzt werden und es sich nicht um festdefinierte Prozeßbausteine, als kleinste Granularität, handelt, können diese vollständig orthogonal miteinander verknüpft werden. Bei den SQL-Templates ist dieses aufgrund der bekannten Struktur von SQL nicht möglich. Die Einbeziehung genesteter Statements ist erst für weitere Ausbaustufen geplant. Die Einschränkung hinsichtlich der ausschließlichen Verwendung von SQL als Anfragesprache ist im Rahmen der Reportingplattform nicht problematisch, da der größte Teil kommerziell eingesetzter Systeme auf relationalen Datenbanksystemen basieren. Fehlender Sprachumfang von SQL (z.B. keine Berechnung der transitiven Hülle, Stücklistenproblem etc. [LiPa94]) kann durch Ergänzung der Komponenten um weitere Methoden kompensiert werden. Dies wären z.B. OLAP-Funktionalitäten [Rugg96] oder die Möglichkeit der Verknüpfung von mehren Ergebnismengen.
Die Beschränkung auf teilstrukturierbare Entscheidungsprobleme durch die Nutzung von mit EPK modellierten Prozessen als Lösungsstrategien erweist sich nicht als Nachteil. Durch die mögliche Kombination der unterschiedlichsten Bausteine und die lose Kopplung kann der Anwender die gewünschten Informationen / Lösungsstrategien zusammenstellen. Die Bereitstellung der passenden Granularität ermöglicht eine ausreichende Flexibilität.

Je feiner die Granularität der Bausteine ist, desto besser kann reagiert werden. Diese Flexibilität determiniert allerdings einen erhöhten Aufwand für den Anwender bei der Auswahl und dem Zusammenfügen der Komponenten.

Die Verwendung der Templates hat sich in Zusammenhang mit der Kombinationsmöglichkeit der Komponenten ebenfalls als sinnvolles Konzept erwiesen. Die Parametrisierung der Komponenten läßt die Einschränkung der Variantenzahl auf eine beherrschbare Größenordnung zu.

5.2 DOMAINMODELLIERUNG

In den vorangegangenen Kapiteln wurde die Entwicklung der Reportingplattform mit den verwendeten Konzepten beschrieben. In diesem Rahmen sind die verschiedenen Bausteine, wie Attribute, DataSets und Prozesse spezifiziert und mittels der notwendigen Verknüpfungsmechanismen versehen worden. In Abbildung 61 wird nochmals die Systemsicht der verschiedenen Modellierungsbausteine (Artefakte) der Reportingplattform in vergröberter Struktur dargestellt. Dies verdeutlicht wie einzelne

Modellierungskomponenten aus weiteren Modellierungskomponenten aufgebaut werden und welche Bestandteile zur Konsistenzerhaltung bei der Komposition dienen.

Struktur der Modellierungskomponenten
(vergröberte Systemsicht)

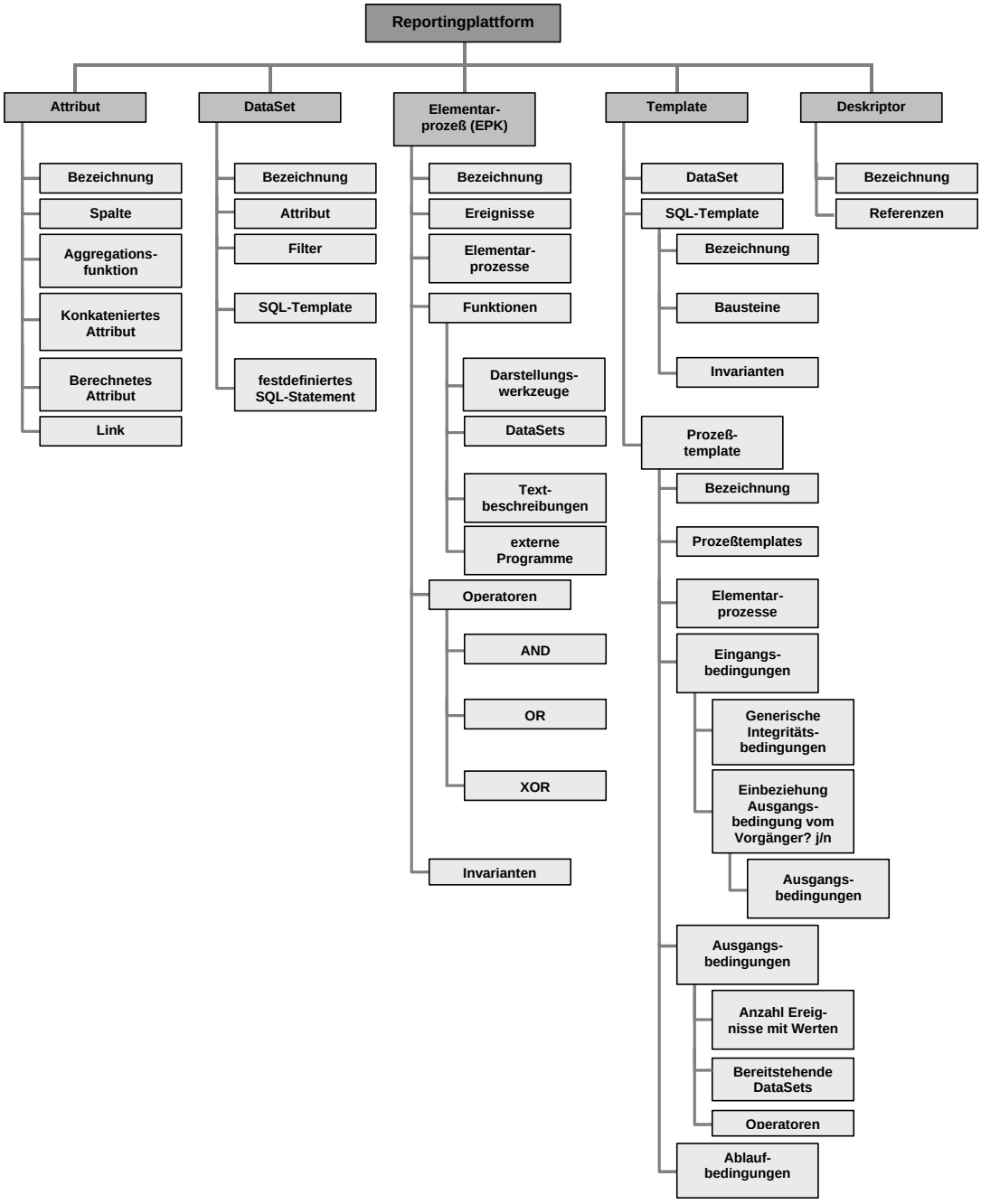


Abbildung 61: Struktur der Modellierungskomponenten

Das systemseitige *Monitoring* bzgl. der möglichen Verknüpfungen/Adaptionen erfolgt durch:

- Metadaten zur Modellierung der DataSets.
- Constraints für die Verknüpfung von Attributen.
- Integritätsbedingungen für die Verknüpfung von Prozeßtemplates.
- Konsistenzbedingungen zur Verknüpfung von Komponenten.
- Invarianten für die Modellierung von EPK und SQL-Templates.

Attribute werden mittels Links zwischen Tabellen und Berechnungsvorschriften für Attribute modelliert. Die Konsistenz bei EPKs und SQL-Templates wird durch Invarianten gewährleistet. Die Verknüpfung der Prozeßtemplates wird durch Integritätsbedingungen überwacht. Deskriptoren dienen der reinen Verschlagwortung und referenzieren andere Artefakte, auf die der Anwender direkt zugreifen kann.

Die modellierten Artefakte werden in Form von Komponenten (i.S. von Componentware) in der Laufzeitumgebung des Frameworks bereitgestellt. Zur Komposition der Komponenten muß die Integrität / Konsistenz der neu entstehenden Komponenten sichergestellt werden. Hierzu ist im Rahmen der Entwicklung einer Composition Language eine systemseitig verwendbare formale Spezifikationssprache notwendig. Diese formalisiert die Zusammensteckmechanismen und gewährleistet eine Robustheit sowie den Übergang von einem konsistenten Komponentenzustand in einen wiederum konsistenten Zustand. Hierzu sind Universen, Components, Schnittstellen etc. zu formalisieren [Szyp98].

Es ergeben sich damit mehrere Ebenen der Konsistenz- / Integritätsprüfung:

1. Metadaten, Invarianten.
2. Formale Komponentenspezifikation.
3. Integritätsbedingungen zur weiteren Strukturüberprüfung.

Diese Mechanismen gewährleisten systemseitig eine hinrechende Einschränkung bzgl. der Verbindung unsinniger Kompositionen, die jedoch notwendigerweise durch das beim Anwender vorhandene Wissen über sinnvolle Verknüpfungen ergänzt sein muß. Grundsätzlich kann in diesem Zusammenhang davon ausgegangen werden, daß durch den Anwender bewußt keine Komposition unsinniger Bausteine durchgeführt wird. Der soweit realisierte Umfang der Reportingplattform gewährleistet somit die Robustheit des Systems, jedoch besteht die Möglichkeit, daß neue Kompositionen auf fachlicher Ebene nicht zueinander passen oder eine unsinnige Semantik aufweisen.

Die hohe Freiheit, die die Reportingplattform dem Anwender bei der Zusammenstellung neuer Entscheidungsprozesse / Informationsabfragen läßt, transferiert die Verantwortung für die Synthese semantisch korrekter, neuer Komponenten ebenfalls auf die Anwenderseite. Der Benutzer muß also schon wissen, was er wie zusammenfügt. Das System kann nicht gewährleisten, daß neue Prozesse auch immer fachlich sinnvolle Abläufe darstellen. Die dahinterstehende Semantik wird ausschließlich durch die Namensgebung der modellierten Bausteine festgelegt und basiert damit auf einem einfachen Textstring. Hierdurch wird die Rolle der Komponente im Gesamtsystem festgelegt [Grif98, S. 451 ff.]. Die *Rolle* bestimmt im Zusammenhang mit Componentware aus technischer Sicht die Einsatzmöglichkeit in einem neu zu entwickelndem System und sollte deshalb insbesondere die verschiedenen syntaktischen und ablaufsemantischen Aspekte beschreiben. Dazu wäre es notwendig, die Schnittstellen formal zu spezifizieren und ggf. die Verarbeitungssemantik mit in die Rollenklassifikation aufzunehmen. Dies wäre sicher für die Umsetzung in einer komplexen Entwicklungsumgebung für Komponentenframeworks notwendig und auch wünschenswert. Für den Anwendungsbereich der Reportingplattform sind die realisierten Verifikationsmechanismen jedoch ausreichend, da sich auf bestimmte Typen modellierbarer Komponenten fokussiert wurde. Aus diesem Grund wird systemseitig mittels o.g. Mechanismen sichergestellt, daß passende Komponenten zusammengefügt werden oder nicht passende Verknüpfungen abgewiesen werden. Bei Unstimmigkeiten wird der Anwender über Dialogboxen zur Auswahl von Alternativen einbezogen.

Nebend den o.g. Formalismen zur strukturierten Spezifikation der Komponenten sowie deren Kompositionen besteht die Möglichkeit das Wissen der zu modellierenden Domain strukturiert in einer *Knowledge Base* abzulegen und mittels unterschiedlicher Mechanismen den Anwender bei der Suche nach Zusammenhängen sowie der Erzeugung neuen Wissens zu unterstützen, so daß unsinnige Verknüpfungen von Informationsbausteinen auszuschließen sind. Hierzu können Mechanismen aus dem Bereich *Ontologie* [Grub93] zur Beschreibung der Begriffswelt und des verwendbaren Vokabulars genutzt werden. Eine Ontologie ist die Spezifikation eines Konzepts mittels eines deklarativen Formalismus. Dazu dienen Definitionen für die Namen der verwendeten Entitäten und formale Axiome zur Beschränkung der Interpretation sowie der wohlgeformten Verwendung der *Terme*. Verallge-

meinert definiert eine Ontologie das Vokabular mit dem Anfragen und Zusicherungen z.B. zwischen Agenten ausgetauscht werden können.

Durch die *Wissensmodellierung* der Domain in Form einer Ontologie wird die Möglichkeit geschaffen, die Semantik des Anwendungsbereichs zu spezifizieren und zu strukturieren. Hierdurch würde die Reportingplattform dahingehend erweitert werden, daß neben der Strukturierung der Modellierungskomponenten auch eine Strukturierung des Wissens möglich ist und der Anwender eine noch umfassendere Unterstützung bei der Entscheidungsfindung erhält. Ansonsten müßte er über einen umfangreichen Erfahrungsschatz verfügen und insbesondere Kenntnis über das implizit in den Rollennamen der Artefakte enthaltene Wissen haben.

Bei Verwendung einer normierten Spezifikation der Wissensmodellierung, z.B. in Form der *KIF – Knowledge Interchange Format* [GeFi92], kann dieses modellierte Wissen auch in andere Knowledge Bases übertragen werden bzw. andere Knowledge Bases könnten ebenfalls darauf zugreifen.

Die Ontologie ermöglicht es nun aufbauend auf den o.g. Prüfleveln einen weiteren Level einzuführen, der die Integrität der Semantik der modellierten Domain mit den neu zu komponierenden Prozessen / Informationsabfragen sicherstellt. Hierbei bieten die Konzepte der Axiome den Beitrag zur Integritätssicherung.

4. Spezifikation der Ontologie

Wenn z.B. zwei Prozeßbausteine zusammengefügt werden, prüfen die Ontologiemechanismen, ob dieses semantisch korrekt ist und ob ein fachlich konsistenter Gesamtprozeß entsteht. Diese Vorgehensweise ist in einer weiteren Ausbaustufe der Reportingplattform notwendig, wenn auch schreibend auf RDBMS zugegriffen wird und die folgenden Prozeßschritte auf zuvor persistent gemachte Daten der vorhergehenden Prozesse / Informationsabfragen aufbauen. Das derzeit implementierte System selektiert lediglich Daten (read-only) und benötigt deshalb nicht in jedem Fall eine semantisch strukturierbare Knowledge Base.

Der Modellierungsaufwand würde durch die mögliche Realisierung der Ontologiemechanismen allerdings nicht unerheblich zunehmen. Diesem Nachteil steht ein weiterer Vorteil gegenüber, daß bereits modellierte Ontologien auch die Grundlage zur Modellierung neuer Ontologien bilden können und damit nicht jedesmal ein vollständig neuer Entwurf durchzuführen wäre.

Zur Modellierung der Ontologie sind zunächst verschiedene Abstrakionen der Domain durchzuführen, da die Grundlage der Ontologie eine Taxonomie mittels Klassen und Subklassen bildet. Abbildung 62 gibt ausschnittsweise einen Überblick zu einer möglichen Modellierung der Domain "Transportlogistik", speziell den Diskurs "Disposition".

Innerhalb der Klassenhierarchie werden die verschiedenen Artefakte in zuvor abstrahierten Knoten abgelegt und ermöglichen die Lösung gleichartiger Problemstellungen, wie dies in Abbildung 62 anhand der Prozeßbausteine für den Umgang mit Havariesituationen, in die Fahrzeuge involviert sind, dargestellt ist (siehe Abbildung 62, Knoten *Havariesituationen/Straße/Fahrzeug*). Die im Rahmen der CBR-Mechanismen der Reportingplattform entwickelte Bewertung der Ähnlichkeit zu bestimmten Problemspezifikationen kann ebenfalls verwendet werden, um die Lösung einzelner Problemstellungen zu gewichten, die ggf. durch einen Knoten beschrieben werden. Grundlage der Ähnlichkeitsberechnung ist das Enthaltensein einzelner Artefakte, die über den Vergleich ihrer Bezeichnungen (Text) in die Berechnung einfließen.

Damit die abzulegenden Bausteine automatisiert bzgl. der Zugehörigkeit zu einem Knoten bewertet werden können, müßte den Knoten im Rahmen der Ontologiespezifikation Bezeichnungen von Attributen, DataSets und Prozeßbausteinen zugeordnet werden, die als Problemspezifikation fungieren. Mittels dieser knotenspezifischen Problemspezifikation kann dann die Ähnlichkeitsberechnung erfolgen und die abzulegenden Artefakte können hinsichtllich der Zugehörigkeit pro Knoten bewertet werden.

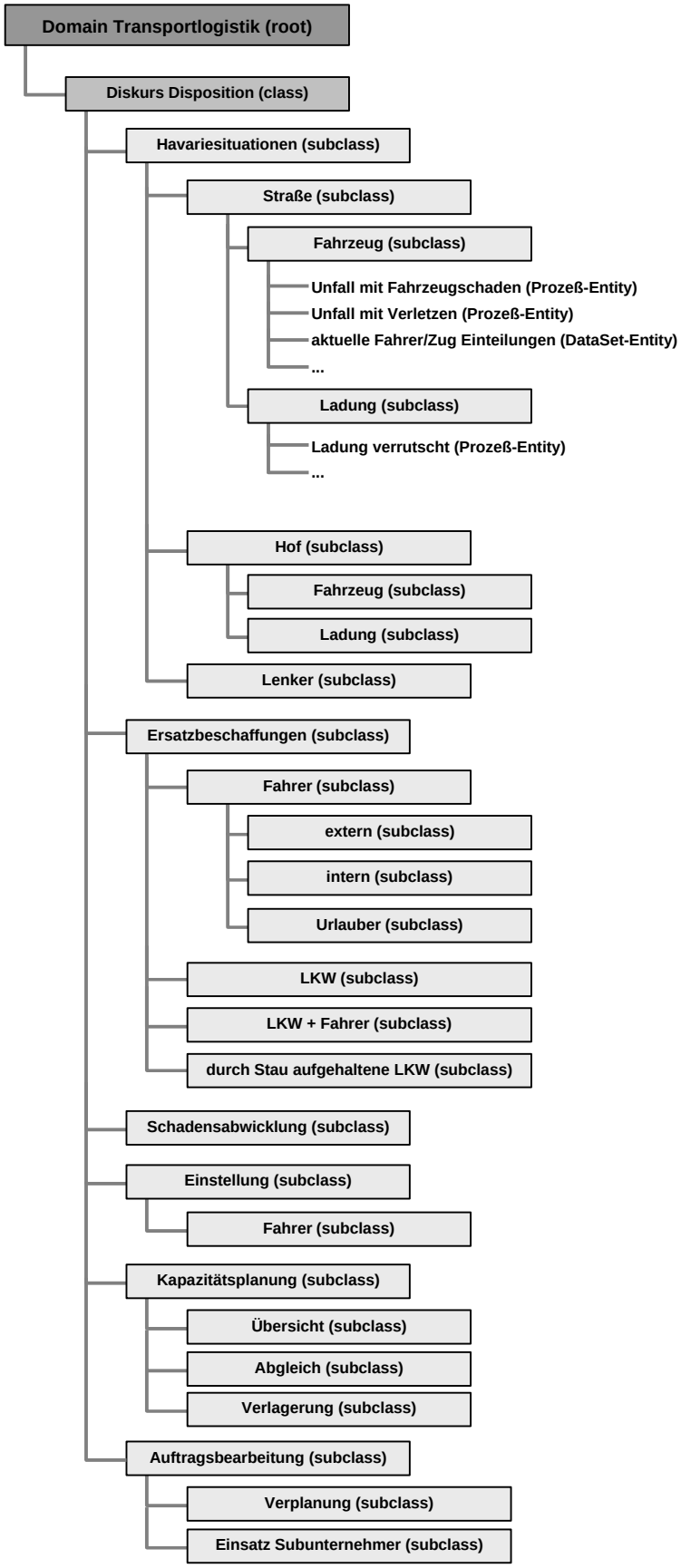


Abbildung 62: Beispielontologie der Domain Transportlogistik / Diskurs Disposition

Zusätzlich könnte der Anwender mit in die Zuweisung einer Gewichtung zur Problemlösung einbezogen werden, indem er systemseitig auf Basis der Ähnlichkeitsberechnung eine Auswahl in

Form eines Wertintervalls angeboten bekommt, so daß er gemäß seiner subjektiven Einschätzung die Gewichtung des jeweiligen Artefaktes an dem einzelnen Knoten bestimmt.

Da gleiche Prozeßbausteine auch gleichzeitig in verschiedenen Knoten enthalten sein können, ermöglicht die Ähnlichkeitsberechnung / Gewichtung unterschiedliche Bewertungen, da der Prozeß ggf. in beiden Knoten unterschiedlich gut zur Lösung beitragen kann. Bei der Neukomposition von Prozeßbausteinen können die einzelnen Ähnlichkeiten / Gewichtungen mittels einer (auch möglichen heuristischen) Berechnungsvorschrift neu bewertet werden und die neu entstandenen Prozesse hierdurch automatisch an neuen Knoten einsortiert werden, sofern die Gewichtungen z.B. einen bestimmten Wert nicht unterschreiten.

Sollte durch Wertunterschreitung ein neuer Prozeß nicht zuordbar sein, so muß der Anwender mit in die Lösung des Problems einbezogen werden, indem er entweder eine Zuordnung explizit vornimmt oder auch einen neuen Knoten erzeugt, der die neu komponierten Prozesse aufnehmen kann. Hierdurch ist gewährleistet, daß die einmal spezifizierte Taxonomie auch durch neu erzeugte Bausteine weiter verwendbar ist. Sofern jedoch die Synthese neuer Bausteine bestehende Axiome verletzt, da diese nicht im Vorwege sämtliche möglichen Problemlösungsprozesse erfassen können, müssen weitere Lösungsmechanismen gefunden werden.

Beispiele für Axiome wären:

- *Einsatz von Subunternehmern nur erlaubt, wenn keine eigenen Züge in der Nähe sind.*
- *Ein Fahrer fährt auf einem Zug.*

Wenn ein Zug beim Transport von Gütern durch einen Stau aufgehalten wurde, muß eine *Ersatzbeschaffung* erfolgen. Hierbei hilft es nicht, einen Ersatzfahrer zu suchen, da im Rahmen der Ersatzbeschaffung für im Stau aufgehaltene Züge ein Zug UND ein Fahrer gefunden werden muß. Aus diesem Grund sollte in der Klasse *Ersatzbeschaffung/durch Stau aufgehaltene Züge* gesucht werden. Die Ontologie sorgt dafür, daß die notwendigen Funktionalitäten systemseitig bereitgestellt weren und auch zur Laufzeit überprüfbar sind. Durch das so modellierte Wissen der gesamten Domain soll ausgeschlossen werden, daß "falsche" Informationen gefunden werden oder falsches Wissen entsteht.

Beispiele:
Deskriptoren als Modellierungsstruktur dienen der Verschlagwortung von Entitäten und können über eine Referenzrelation in KIF spezifiziert werden.

```
(defrelation REFERENCE
; eine Referenz ist eine Beschreibung einer Entität (Attribut, DataSet, Prozeß), die sie eindeutig identifiziert. Beinhaltet alle Informationen zum Wiederauffinden der Entität.
(=>     (Reference ?ref)
        (and    (defined        (ref.entity  ?ref))
                (defined        (ref.name  ?ref)))))

(deffunction  REF.ENTITY
; ref.entity ist eine Abbildung der reference auf die Entität.
(=>     (and    (defined        (REF.ENTITY  ?ref)
                (=              (REF.ENTITY  ?ref)  ?ent)
        (and    (reference  ?ref)
                (entity       ?ent))))

(deffunction  REF.NAME
; REF.NAME ist eine Abbildung auf den Entitätennamen
(=>     (and    (defined        (REF.NAME  ?ref))
                (=              (REF.NAME  ? ref)  ?Name))
        (and    (reference  ?ref)
                (entity.Name  ?name))))
```

Axiom: *Einsatzbereiter Zug besteht aus Zug UND Fahrer*.

```
(=>     (einsatzbereiterZug  ?zug)
            (and    (exists-zug       ?zug)
                    (exists-fahrer    ?fahrer))))
```

Zu klären gilt es in diesem Zusammenhang, wie die Modellierung neuen Wissens durch den Anwender im System der Ontologie abzubilden wäre. Ein Vorteil der Reportingplattform liegt in der Verwendung der entwickelten Mechanismen des fallbasierten Schließens in Verbindung mit den Kompositionsmöglichkeiten der Komponenten zur Laufzeit, die insbesondere die benutzergesteuerte Modellierung und Speicherung neuer Informationen ermöglicht. Inwiefern eine Modifikation der spezifizierten Ontologie zur Laufzeit möglich ist, muß noch geprüft werden. Die Speicherung neuer Artefakte innerhalb der Klassenhierarchie ist nach o.g. Vorgehensweise lösbar. Sofern Axiome die Verbindung zu unsinnigen neuen Informationen beschränken, impliziert dieses jedoch wiederum die Vollständigkeit der Domainmodellierung, was aufgrund der hohen Komplexität, analog der bereits implementierten Funktionalitäten, nicht möglich ist. Somit ist wiederum ein Mittelweg zu finden, der auf der einen Seite eine sinnvolle Einschränkung der Möglichkeiten des Anwenders bedeutet, ihm aber auf der anderen Seite genügend Freiheit bei der Zusammenstellung ggf. auch unorthodoxer Lösungsprozesse läßt.

5.3 WERKZEUGZENTRIERTE ANSÄTZE

Wie bereits erwähnt sollte das DSS-Tool und das Prozeßtool für die Ablaufsteuerung der Entscheidungsprozesse zukünftig transparent für den Benutzer anwendbar sein. Für die enge Integration externer Programme in die Reportingplattform müßten zusätzliche generische Schnittstellen zur Übergabe komplexerer Datenstrukturen (Datenfluß, Kontrollfluß etc.) geschaffen werden. Die Visualisierung einzelner Prozeßkomponenten ist ebenfalls mit einer verbesserten graphischen Aufbereitung zu versehen, um so flexibler und schneller die gewünschten Informationen zu finden.

Derzeit werden SQL-Templates lediglich für lesende Zugriffe genutzt. Es wäre sinnvoll ebenfalls einen schreibenden Zugriff auf die Datenbestände zu ermöglichen, sofern Informationen des Data Dictionarys und Metadaten des Repositorys dieses ermöglichen. Zur Verwendung von Was-wäre-wenn- oder Simulationsfunktionalitäten müßten über SQL-Statements abgefragte Datenmengen zwischengepuffert werden.

5.4 ENTWICKLUNG DER NÄCHSTEN GENERATION DER WERKZEUGE

Im Rahmen der Wiederverwendbarkeit von Softwarebausteinen lassen sich die Mechanismen zur Komposition und Adaption von Komponenten sowie der Wiederauffindbarkeit von Design Patterns und Komponenten in *Patternbrowser* integrieren oder für den Aufbau von Pattern- / Komponentenbibliotheken nutzen. Eine interessante Erweiterung wäre die Entwicklung *generischer Patterns* [Berz96] für bestimmte Domains, die wiederum zur Modellierung domainspezifischer Komponenten, wie z.B. für den Bereich der Transportwirtschaft [OMG97], verwendet werden können.

Wenn die Design Patterns und die daraus modellierten Komponenten als Lösungen zu bestimmten Problemstellungen betrachtet werden, so könnten verschiedenste Lösungen in Form einer *Design Pattern-Bibliothek* mit zugehörigen Komponenten zusammengefaßt werden, die je nach Problemstellung zu direkt ausführbaren Programmcode übersetzt und ausgeführt werden können. Die flexible Anpassung an die jeweilige Problemstellung ist durch die entwickelten Mechanismen gegeben. Hierdurch wird ein hohes Maß an Wiederverwendbarkeit erreicht.

Zur Modellierung der Patterns und Komponenten müssen komfortable Werkzeuge zur Verfügung stehen. Um der trotz Parametrisierung existierenden Variantenvielfalt Herr zu werden, muß das System über die Eingabe der Problemspezifikation in einen o.g. Patternbrowser beim Finden möglicher Lösungsalternativen unterstützend mitwirken. Hierzu ist die Einteilung der Lösungen in verschiedene Domains und Abstraktionsebenen sinnvoll.

Der Mechanismus für eine persistente Speicherung von Werkzeugen und Materialien im Archiv kann dahingehend erweitert werden, daß die gespeicherten Komponenten nicht einfach im Archiv abgelegt werden, sondern an einen anderen Netzwerkknoten weitergeleitet werden können. Dabei ist unerheblich, ob der Informationsaustausch innerhalb des entwickelten Prozeßtools über gemeinsame Speicherbereiche oder per Email über das Internet erfolgt. Bei dem nächsten Knoten wird die Komponente erneut zugreifbar und kann aufbauend auf dem "eingefrorenen" Arbeitsstand entsprechend weiter bearbeitet werden. Es müßten nicht einmal sämtliche Werkzeuge an den einzelnen Knoten verfügbar sein, da innerhalb eines zu versendenden Objekts alle Komponenten mitgegeben werden können (natürlich könnten auch nur DataSets oder Prozesse versendet werden). Als Arbeitsplatzumgebung wäre nur die Reportingplattform erforderlich. Es könnten so auch verschiedene Versionen der Reportingplattform miteinander kommunizieren, die nicht immer sämtliche Funktionalitäten bzw. Werkzeuge umfassen müssen.
Zur Realisierung dieser Erweiterungsmöglichkeiten könnten die Konzepte der *migrierenden Threads* [Math+95] [MaSc94] als Erweiterung des *Tycoon-Projekts* [Matt93] genutzt werden. Die auszutauschenden Komponenten (Werkzeuge und DataSets / Prozesse mit ihren jeweils aktuellem Prozeßstatus) müßten dazu als migrierende Threads modelliert werden und können so über das Netzwerk zu heterogenen Hard- und Softwaresystemen versendet werden, auf denen die Reportingplattform installiert ist.

Beim Durchlaufen der einzelnen Entscheidungsprozesse könnte an jedem Knoten erneut eine Bewertung des aktuellen Prozeßstands inkl. Lösungsraum erfolgen, so daß dem Anwender ggf. jeweils neue Lösungsvorschläge in Form weiterer Komponenten unterbreitet werden.

Sollten aufgrund einer Problemspezifikation durch das System keine modellierten oder nur mit sehr geringer Relevanz zum Problem passenden Lösungen gefunden werden, so wäre der Einsatz von Multiagentensystemen [Kirn94] [Müll93] oder Planungssystemen [Yang97] zur ad hoc Generierung von Prozessen sinnvoll.

Die Mechanismen des fallbasierten Schließens haben sich als äußerst sinnvoll zum Auffinden von Templates oder Prozessen erwiesen. Damit läßt sich diese Methodik relativ einfach auf bestehende Geschäftsprozeßmodellierungssysteme (z.B. ARIS) oder Workflowmanagementsysteme (z.B. Staffware) übertragen, um dem Modellierer wiederverwendbare Bausteine anzubieten und so die Modellierungsphase innerhalb von Projekten zu verkürzen.

Zur Erhöhung der Systemflexibilität wäre es sinnvoll, für die konsistente Spezifikation der Geschäftsprozeß- / Arbeitsvorgangs- und Workflowmodelle am Markt erhältliche Modellierungstools bzw. Workflowmanagementsysteme zu nutzen, wobei letztere ebenfalls die Ablaufsteuerung übernehmen könnten. Solange die Workflowmanagementsysteme jedoch mit den Nachteilen eines mangelnden Exceptionhandlings behaftet sind und keinesfalls die hohen Anforderungen der Reportingplattform an ein komponentenbasiertes System erfüllen, ist die Integration nur schwer realisierbar. Sicherlich bieten aktuelle Forschungsprojekte zur objektorientierten Entwicklung von Workflowmanagementsystemen (siehe [Wesk+98]) hierzu zukünftig Lösungen.

Sollte dies der Fall sein, so könnte die mögliche Integration externer Systeme in die Reportingplattform wie folgt umgesetzt werden [Riec98]. Dazu wird in Abbildung 63 die Architektur des Systemkerns in einer anderen Form dargestellt. Das System besteht aus einer objektorientierten- / Komponenten-Wissensbasis (OO-Wissensbasis) und mehreren Schnittstellen zur Außenwelt. Der modulare Aufbau des Systems soll insbesondere zu einer möglichen Verwendbarkeit auch in anderen Bereichen, als der Reportingplattform dienen.

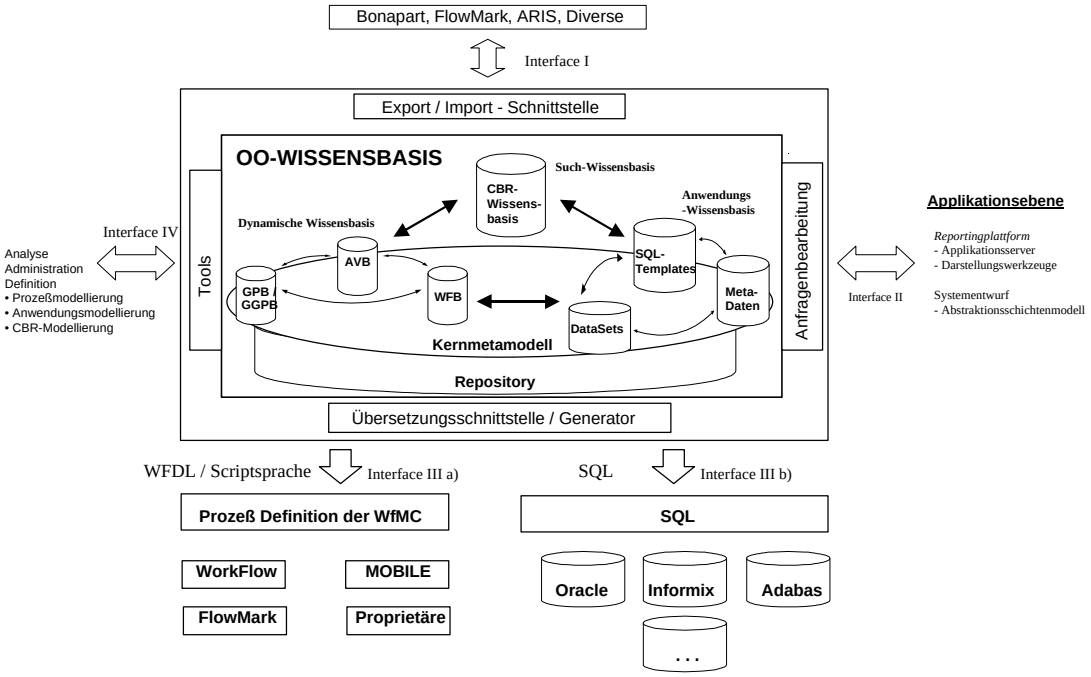


Abbildung 63: Architektur des erweiterten Systemkerns

Die *Anwendungswissensbasis* (DataSets, SQL-Templates und Metadaten) kommuniziert über eine *interne Schnittstelle* mit der *Dynamischen Wissensbasis* (Geschäftsprozeßbausteine-GPB, Arbeitsvorgangsbausteine-AVB und Workflowbausteine-WFB), um die notwendige Kopplung zwischen Workflow- / Entscheidungsprozessen und Informationsabfragen zu gewährleisten. Die Informationsabfragen können auch ohne zugehörige Workflowinstanzen seitens des Benutzers aufgerufen werden. Die CBR-Wissensbasis kommuniziert nun wiederum mit den beiden anderen Wissensbasen, um einerseits die notwendigen Strukturen für das Wiederauffinden aufzubauen und andererseits aufgrund von Problemspezifikationen Fälle zu finden und anschließend dem Anwender zu präsentieren.
Der Systemkern steht über *externe Schnittstellen* mit der Außenwelt in Verbindung. Es werden vier Schnittstellen unterschieden, die die Kommunikation ermöglichen.

Schnittstelle I ist für den Import oder Export von Informationen aus externen (Modellierungs-) Tools, wie z.B. ARIS, Bonapart oder anderen Werkzeugen, vorgesehen. So werden durch den Import von ARIS-Spezifikationen neue AVB erzeugt. Sollte ein Import von einem Workflowmanagementsystem durchgeführt werden, so erzeugt das System neue WFB.

Schnittstelle II ermöglicht die Kommunikation des Systems mit Applikationen in verschiedenen Anwendungsgebieten. So kommunizieren die Applikationen der Reportingplattform über diese Schnittstelle mit dem Metamodell. Ein weiteres Anwendungsbeispiel wäre ein Tool für die Systementwicklung auf Basis vorhandener Referenzprozesse.

Schnittstelle III
a) generiert aus den WFB entsprechende Workflowspezifikationen in einer der WFMC konformen Skriptsprache, die über ein Interface an unterschiedliche Workflowmanagementsysteme übergeben werden können.

b) generiert aus den SQL-Templates und den Metadaten kontextbezogene SQL-Statements, die an unterschiedliche RDBMS übergeben werden, die wiederum die Daten bereitstellen.

Schnittstelle IV umfaßt Modellierungskomponenten für die drei Wissensbasen sowie Administrations- und Analysetools.

6 ABKÜRZUNGSVERZEICHNIS

4GL	Fourth Generation Language
API	Application Programming Interface
AVB	Arbeitsvorgangsbausteine
BNF	Backus-Naur-Form
bzgl.	bezüglich
bzw.	beziehungsweise
CBR	Case-based Reasoning
CORBA	Common Object Request Broker Architecture
DCOM	Distributed Common Object Model
DDL	Data Definition Language
DML	Data Manipulation
DSS	Decision Support System
EIS	Executice Information System
EPK	Ereignisgesteuerte Prozeßketten
ERP	Enterprise Resource Planning System
f. f.	ferner folgende
FIS	Führungsinformationssystem
GPB	Geschäftsprozeßbausteine
i. a.	im allgemeinen
IDL	Interface Definition Language
i. d. R.	in der Regel
i. S.	im Sinne
KIF	Knowledge Interchange Format
MSL	Mobile Script Language
MUSE	Method for User Interface Engineering
o. g.	oben genannte
OLAP	Online Analytical Processing
OMT	Object Modeling Technique
OO	Objektorientiert
QBE	Query by Example
RDBMS	Relational Database Management System
SQL	Structured Query Language
s. u.	siehe unten
VDM	Vienna Development Method
WF	Workflow
WFB	Workflowbausteine
WFM	Workflowmanagement
WfMC	Workflowmanagement Coalition
WFMS	Workflowmanagementsystem

7 ABBILDUNGSVERZEICHNIS

8 TABELLENVERZEICHNIS

9 LITERATURVERZEICHNIS

[AaPl94] Aamondt, A. / Plaza, E.: Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches, in: AI Communications 7(1), 1994.

[Abi+95] Abiteboul, S. / Hull, R. / Vianu, V.: *Foundations of Databases*, Addison-Wesley, 1995.

[AlAa96] Althoff, K. / Aamondt, A.*: Zur Analyse fallbasierter Problemlöse- und Lernmethoden in Abhängigkeit von Charakteristika gegebener Aufgabenstellungen und Anwendungsdomänen,* in: Künstliche Intelligenz, Heft 1/1996.

[Alt+92a] Althoff, K. / Wess, S.: *Ähnlichkeit in PATDEX,* Seiten 95-100, in: Althoff, K.-D. / Bartsch-Spörl, B. / Janetzko, D. (Hrsg): Workshop: Ähnlichkeit von Fällen beim fallbasierten Schließen. SEKI-Report., 1992.

[Alt+92b] Althoff, K. / Wess, S. / Bartsch-Spörl, B. / Janetzko, D. / Maurer, F. / Voß, A.: *Fallbasiertes Schließen,* in Expertensysteme: Welche Rolle spielen Fälle für wissensbasierte Systeme ? 1992.

[AlWe91] Althoff, K. / Wess, S.*: Fallbasiertes Problemlösen in Expertensystemen – begriffliche und inhaltliche Betrachtungen,* Fachbericht. Universität Kaiserslautern, 1991.

[Ambe96] Amberg, M.: *Transformation von Geschäftsprozeßmodellen des SOM-Ansatzes* in: workflow-orientierte Anwendungssysteme, Vortrag im Workshop „Workflow-management-State-of-the-Art aus Sicht von Theorie und Praxis“, Westf. Wilhelms-Universität Münster, 1996.

[BaWe96] Bartsch-Spörl, B. / Wess, S.: *Editorial.* In: Künstliche Intelligenz, Heft 1, Interdata, 1996.

[Berg96] Bergmann, R.: *Effizientes Problemlösen durch flexible Wiederverwendung von Fällen auf verschiedenen Abstraktionsebenen,* Dissertation, Universität Kaiserslautern, 1996.

[Berz96] Berztiss, A.: *Domains and Patterns in Conceptual Modeling,* in: Kangassalo, H.,Fischer Nilsson, J. (Hrsg.): Information Modelling and Knowledge Bases VIII, The Sith European – Japanese Seminar on Information Modelling and Knowledge Bases, Hornbaek, Dänemark, 1996.

[Booc96] Booch, G.: *Object Solutions –Managing the object-oriented Project,* Addison-Wesley Publishing Company, 1996.

[Bud+92] Budde, R. / Christ-Neumann, M.-L. / Sylla, K.-H.*: Tools and Materials, an Analysis and Design Metaphor,* in: Heeg, G. / Magnusson, B. / Meyer, B. (Hrsg.): TOOLS7, Prentice-Hall, 1992.

[BuJa96] Bullinger, H.-J. / Janssen, C.: *Ein Beschreibungskonzept für Dialogabläufe bei graphischen Benutzungsschnittstellen,* Informatik Forschung und Entwicklung, Springer-Verlag, 1996.

[Bus+96] Buschmann, F. / Meunier, R. / Rohnert, H. / Sommerlad, P. / Stal, M.*: Pattern-Oriented Software Architecture-A System of Patterns,* Wiley, 1996.

[BuZü90] Budde, R. / Züllighoven, H.: *Software-Werkzeuge in einer Programmierwerkstatt,* Oldenbourg Verlag, 1990.

[Cham74] Chamberlin, D. / Boyce, R.: SEQUEL: A structured English query language, Proceedings ACM SIGFIDET Workshop, 1974.

[Codd71] Codd, E.F.: *A data base sublanguage founded on the relational calculus,* Proceedings ACM SIGFIDET Workshop, 1971.

[CoSc95] Coplien, J.O. / Schmidt, D.C.: *Pattern Languages of Program Design,* Addision-Wesley, 1995.

[Dec+94] Decker, K. / Dvorak, J. / Rehmann, R.: *A tool environment for parallel programming – User-driven development of a novel programming environment for distributed memory parallel processor systems,* in: Priority Programme Informatics Research, Information Conference Module 3 on Massively parallel systems, 1994.

[DeOb96] Desel, J., Oberweis, A.: *Petri-Netze in der Angewandten Informatik,* in: Wirtschaftsinformatik, Vieweg-Verlag, Heft 4, 1996.

[DIN96] DIN-Fachbericht 50: *Geschäftsprozeßmodellierung und Workflow-Management.* Forschungs- und Entwicklungsbedarf im Rahmen der Entwicklungsbegleitenden Normung (EBN), DIN Deutsches Insitut für Normung e.V, Beuth Verlag, 1.Auflage 1996.

[DuRi97] Ducasse, S. / Richner, T.: *Executable Connectors: Towards Reusable Design Elements,* in: Proceedings ESEC '97, 1997.

[Egge97] Egge, R.: *Nutzung von Methoden des fallbasierten Schließens in einem Informationssystem,* Diplomarbeit, Fachhochschule Wedel, 1997.

[FeSi95] Ferstl, O.K. / Sinz, E.J.: *Der Ansatz des Semantischen Objektmodells (SOM) zur Modellierung von Geschäftsprozessen,* in: Wirtschaftsinformatik 37. Jg., Vieweg, 1995.

[Gai+94] Gaintanides, M. / Scholz, R. / Vrohlings, A. u. a., 1994: *Prozessmanagement: Konzepte, Umsetzungen und Erfahrungen des Reengineering*, Hanser, 1994.

[Gam+95] Gamma, E. / Helm, R. / Johnson, R. / Vlissides, J.: *Design Patterns*, Addison-Wesley, 1995.

[GaSc95] Galler, J. / Scheer, A.-W.*: Workflow-Projekte: Vom Geschäftsprozeßmodell zur unternehmensspezifischen Workflow-Anwendung*. IM-Information Management, S.20-28, 10, 1995.

[GeFi92] Geneserth, M.R. / Fikes, R.E.: *Knowledge Interchange Format, Vers. 3.0*, Reference Manual, Technical Report Logic-92-1, Computer Science Department, Stanford University, 1992

[Gorn94] Gorny, P.: *Design sofware-ergonomischer Benutzungsoberflächen*; Tutorial zur Fachtagung D-CSCW'94, Universität Marburg.

[Grif98] Griffel, F.: *Componentware – Konzepte und Techniken eines Softwareparadigmas*, dpunkt.verlag, 1998.

[Grub93] Gruber, T.R.: *Towards Principles for the Design of Ontologies Used for Knowledge Sharing*, Stanford Knowledge Systems Laboratory, 1993

[GuPo98] Guntermann, T. / Popp, K.: *Betrachtungen zur Anpassung objektorientierter Standardanwendungssysteme*, in: Informationssystem Architekturen, Rundbrief des GI-Fachausschusses 5.2, 5.Jg., Heft 1, 1998.

[Hamp97] Hampe, A.: *Entwicklung einer Benutzungsschnittstelle für ein entscheidungsunterstützendes Informationssystem*, Diplomarbeit, Fachhochschule Wedel, 1997

[Hann96] Hannig, U.: *Data Warehouse und Management-Informationssysteme*, Schaeffer-Poeschel, 1996.

[Humm92] Hummeltenberg, W.: *Realisierung von Management-Unterstützungssystemen mit Planungssprachen und Generatoren für Führungsinformationssysteme*, in: Hichert, R. / Moritz, M. (Hrsg.): Managementinformationssysteme, Springer, 1992.

[Humm98] Hummeltenberg, W.: *Information Management for Business and Competetive Intelligence and Excellence*, Vieweg, 1998.

[Inmo92] Inmon, W.H.: *Building the Data Warehouse*, Boston, Toronto, London: QED Technical Publishing Group, 1992.

[INRE95] *Documentation of the final integrated INRECA System*, Version 1. ESPRIT project 6322, Task 3.4, Deliverable D16, 1995.

[JaBu96] Bussler, C. / Jablonski, S.: *Workflow Management: Modeling Concepts, Architecture and Implementation*. International Thomson Computer Press, 1996.

[Jabl95a] Jablonski, S.: *Workflow-Management-Systeme: Motivation, Modellierung, Architektur,* Informatik Spektrum, S.13-24, Organ der Gesellschaft für Informatik e.V., 1995.

[Jabl95b] Jablonski, S.: *Workflow-Management-Systeme: Modellierung und Architektur*, Thomson Publishing, 1.Auflage, 1995.

[Jab+97] Jablonski, S. / Böhm, M. / Schulze, W.: *Workflow Management: Entwicklung von Anwendungen und Systemen; Facetten einer neuen Technologi*e, 1.Auflage, dpunkt-Verlag, 1997.

[Jac+92] Jacobson, I. / Christerson, M. / Jonsson, P. / Overgaard, G.*: Object-Oriented Software Engineering - A Use Case Driven Approach*, Addison-Wesley, 1992.

[Jahn94] Jahnke, B.: *Führungsinformationssysteme*, in: Preßmar, D. (Hrsg.): Informationsmanagement, Gabler, 1994.

[Jana82] Janas, J.: *Natürlichsprachliche Schnittstelle zu relationalen Datenbanken*, Dissertation, Hochschule der Bundeswehr Hamburg, 1982.

[Jan+93] Janetzko, D. / Melis, E. / Wess, S.: *System and Processing View*, in: Wess, S. / Althoff, K.-D. / Janetzko, D. / Bartsch-Spörl, B. (Hrsg.): SEKI – Working Paper, Universität Kaiserslautern, 1993.

[Jenz98] Jenz & Partner: *Komponentenbasierte Anwendungsentwicklung*, Studie 1998.

[Jone90] Jones, C.B.: *Systematic Software Development using VDM*, Prentice-Hall Int., 2nd Ed., 1990.

[Kirn94] Kirn, St.: *Zur Verbundintelligenz integrierter Mensch-Computer-Teams: Ein organisationstheoretischer Ansatz*, Arbeitsbericht Nr.28, Institut fürWirtschaftsinformatik der Westfälischen Wilhelms-Universität Münster, 1994

[KiUn94] Kirn, St./Unland, R.: *Workflow Management mit kooperativen Softwaresystemen: State of the Art und Problemabriß*, Arbeitsbericht. Nr.29, Institut für Wirtschaftsinformatik der Westfälischen Wilhelms-Universität Münster, 1994.

[Kolo93] Kolodner, J.: *Case-Based Reasoning*, Morgan-Kaufmann, 1993.

[Korn94] Kornblum, W.: *Einsatz innovativer Führungsinformationssysteme für eine effiziente Unternehmenssteuerung*, in: Klotz, M. / Wenzel, H. (Hrsg.): Führungsinformations-systeme im Unternehmen, Berlin: Erich Schmidt, 1994.

[Kral96] Krallmann, H. / Reichardt, K.: *Business Process Reengineering: Konsequente Prozessrestrukturierung unter Einsatz von modernen Methoden und Werkzeugen*, Universität Oldenburg, 09.02.96.

[Küff94] Küffmann, K.: *Software-Wiederverwendung – Konzept einer domänenorientierten Architektur*, Vieweg, 1994.

[Kuge96] Kugelberg, J.: *Entwicklung eines Prototypen für ein transportwirtschaftliches Informationssystem*, Diplomarbeit, Institut für Wirtschaftsinformatik, Universität Hamburg, 1996.

[Leme97] Lemenkühler, M.: *Einführung eines Executive Information System (EIS) / Data Warehouse zur Unterstützung der Geschäftsführung eines Speditionsunternehmens*, Diplomarbeit, Fachhochschule Wedel, 1997.

[Lew+95] Lewis, T. / Rosenstein, L. / Pree, W. / Weinand, A. / Gamma, E. / Calder, P. / Andert, G. / Vlissides, J. / Schmucker, K.: *Object Oriented Application Frameworks*, Manning Publications Co., 1995.

[Ley+95] Leymann, F. / Roller, D. / Vogt, E.: *White Paper: [Workflow Management]*, IBM Software Solutions Division, Workflow Management White Paper – Third Draft, März 1995.

[Lieb92] Liebelt, W.: *Methoden und Techniken der Ablauforganisation*, in: Frese, E.: Handwörterbuch der Organisation. Hrsg.:, 3.Auflage, Teubner, 1992.

[Lind96] Lindner, U.: *Massive Wiederverwendung: Konzepte, Techniken und Organisation*, Objektspektrum 2/96, SIGS Conferences GmbH, 1996.

[LiPa94] Linnemann, V. / Pampel, H., *Sprachliche Formulierung rekursiver und iterativer Anfragen in Datenbanksystemen*, in: Informatik-Spektrum (1994) 17, Springer Verlag, 1994.

[Math+95] Mathiske, B. / Matthes, F. / Schmidt, J.-W.: *On Migrating Threads*, Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems, Israel, Juni 1995.

[MaSc90] Matthes, F./Schmidt, J.: *Towards Database Application Systems: Types, Kinds and Other Open Invitations*, in: Schmidt, J./Stogny, B. (Hrsg.): Next Generation Information System Technology, Lecture Notes in Computer Science 504, Springer, 1990.

[MaSc94] Matthes, F. / Schmidt, J.-W.: *Persistent Threads*, Proceedings of the 20th VLDB Conference, Chile, September 1994.

[Matt93] Matthes, F.: *Persistente Objektsysteme*, Springer-Verlag, 1993.

[Moun96] Mountfield, Andrew: *Anforderungen an moderne Managementinformationssysteme*, in: Hannig, U. (Hrsg.): Data Warehouse und Managementinformationssysteme, Stuttgart: Schäfer-Poeschel, 1996.

[Mühl96] zur Mühlen, M.: *Der Lösungsbeitrag von Metamodellen und Kontrollflußprimitiven beim Vergleich von Workflowmanagement-Systemen*, Diplomarbeit, Westf. Wilhelms-Universität Münster, 18.September 1996.

[Müll93] Müller, J.*: Verteilte Künstliche Intelligenz*, BI Wissenschaftsverlag, 1993.

[Nie+90] Nierstrasz, O. / Dami, L. / de Mey, V. / Stadelmann, M. / Tsichritzis, D.: *Visual Scripting – Towards Interactive Construction of Oject-Oriented Applications*, Tsichritzis et ateurs, 1990.

[Nie+91] Nierstrasz, O. / Tsichritzis, D. / de Mey, V. / Stadelmann, M.: *Objects + Scripts = Applications*, in: Proceedings, Esprit 1991 Conference, Kluwer Academic Publishers, 1991.

[Nie+96] Nierstrasz, O. / Schneyder, J.-G. / Lumpe, M.: *Formalizing Composable Software Systems – A Research Agenda*, in: Proceedings 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems, FMOODS'96, Paris, 1996.

[Nie+97] Nierstrasz, O. / Schneyder, J.-G. / Lumpe, M. / Achermann, F.: *Towards a formal composition language*, in: Proceedings of ESEC ´97 Workshop on Foundations of Component-Based Systems, Zürich, 1997.

[Nier95] Nierstrasz, O.: *Research Topics* in Software Composition, in: Proceedings, Langes et Modèles à Objets, A. Napoli (Hrsg.), Nancy, 1995.

[NiMe95] Nierstrasz, O. / Meijler, T.*: Requirements for a Composition Language*, ECOOP 94, LNCS 924, Springer, 1995.

[NiTs95] Nierstrasz, O. / Tsichritzis, D. (Hrsg.): *Object-Oriented Software Composition*, Prentice Hall, 1995.

[Noac89] Noack, J.:*Ein transportables Front-End zur Interpretation deutschsprachiger Anfragen an ein relationales Datenbanksystem*, Dissertion, RWTH Aachen, 1989.

[Ober96] Oberweis, A.: *Modellierung und Ausführung von Workflows mit Petri-Netzen*. B.G. Teubner Verlagsgesellschaft Stuttgart, 1996.

[OMG96] *The Object Management Architecture Guide*, Rev. 2.0, OMG, 1996.

[OMG97] *Transportation Domain Reference Model*, Draft Rev. 1.0, Document transprt/97-11-01, Nov. OMG, 1997.

[Pete96] Peters, S.: *Geschäftsprozeßmodellierung im Speditionswesen*, Diplomarbeit, Fachhochschule Wedel, 1996.

[PiPe92] Pittman, T. / Peters, J.: *The Art of Compiler Design – Theory and Practice*, Prentice-Hall International, 1992.

[Poh+95, S. 55] Pohl, K. / Jarke, M./Dömges, R. (1995): *Unterstützung schwach strukturierter Geschäftsprozesse*. In: Informationssystem Architektur, Rundbrief GI-Fachausschuß 5.2, 2. Jg, Heft 2.

[Póly62] Pólya, G.: *Mathematik und Plausibles Schließen, Induktion und Analogie in der Mathematik*, Bd I, Birkhäuser Verlag, 1962.

[Póly67] Pólya, G.: *Vom Lösen mathematischer Aufgaben, Einsicht und Entdeckung, Lernen und Lehren*, Bd II, Birkhäuser Verlag, 1967.

[Pree97] Pree, W.: *Komponenten-basierte Softwareentwicklung mit Frameworks*, dpunkt.verlag, 1997.

[Pryc96] Pryce, N.: *Distributed Software Engineering Group*, Department of Computing, Imperial College, London, 1996.

[Qin 95] Qin, L.: *Zur wissensorientierten Unterstützung bei der Benutzungsoberflächengestaltung*. In: Menschengerechte Softwaregestaltung (Konzepte und Werkzeuge auf dem Weg in die Praxis), Teubner-Verlag, 1995.

[Raab96] Raab, J.: *MIS als Wegbereiter des Lean Management*, in: Hannig, U. (Hrsg.): Data Warehouse und Managementinformationssysteme, Schäfer-Poeschel, 1996.

[ReWi96] Reinartz, T. / Wilke, W.*: Fallbasiertes Schließen in der Finanzwelt: Eine echte Alternative zu Neuronalen Netzen?* Fachbericht, Universität Kaiserslautern, 1996.

[Riec98] Rieckmann, J.: *Die Unterstützung von Benutzerprozessen beim Einsatz von DSS/EIS durch Workflowmanagementsysteme*, in: Hummeltenberg, W. (Hrsg.): Information Management for Business and Competitive Intelligence and Excellence, Vieweg, 1998.

[Riec99] Rieckmann, J.: *Componentware for Supporting Transport Logistics*, in: Scholz-Reiter, B. / Stahlmann, H.-D. / Nethe, A. (Hrsg.): Process Modelling, Springer, 1999.

[Rieg90] Rieger, B.: *Executive Information Systems (EIS): Rechnergestützte Aufbereitung von Führungsinformationen*, in: Krallmann, H. (Hrsg.): Innovative Anwendungen der Informations- und Kommunikationstechnologien in den 90er Jahren, Oldenbourg, 1990.

[Riff98] Riffel, V.: *Implementierung einer Workflowanwendung zur Unterstützung der Geschäftsprozesse in einem Dienstleistungsunternehmen*, Diplomarbeit, Fachhochschule Wedel, 1998.

[Rugg96] Ruggaber, R.: *Konzepte des Online Analytical Processing (OLAP*), in: Breitner, C. / Herzog, U. / Müller, J. / Schlösser, J. (Hrsg.): Data Warehouse, Karlsruhe: Seminar Universität Karlsruhe, SS 1996.

[Rum+91] Rumbaugh, J. / Blaha, M. / Premerlani, W. / Eddy, F. / Lorensen, W.: *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[Saak97] Saake, G. / Türkner, C. / Schmitt, I.: *Objektdatenbanken: Konzepte, Sprache, Architekturen*. An International Thomson Publishing Company, Bonn, 1.Auflage 1997.

[Schee94] Scheer, A.-W.: *Wirtschaftsinformatik, Referenzmodelle für industrielle Geschäftsprozesse*, 5. Aufl., Springer, 1994.

[Schew96] Schewe, B.: *Kooperative Softwareentwicklung*, Deutscher Universitätsverlag, 1996.

[Schm87] Schmidt, J.-W.: Datenbankmodelle, in: Lockemann, P.: Datenbank-Handbuch, Springer, 1987

[Schm95] Schmidt, D.C.: *Using Design Patterns to Develop Reusable Object-Oriented Communication Software*, Communications of the ACM, Vol. 38, Nr. 10, 1995.

[Scho90] Scholz-Reiter, B.: *CIM-Informations- und Kommunikationssysteme*, Oldenbourg Verlag, 1990.

[Serw94] Serwas, G. / Gutzmann. J.: *Die (Weiter-) Entwicklung eines FIS - Eine unendliche Geschichte?*, in: Klotz, M. / Wenzel, H. (Hrsg.): Führungsinformationssysteme im Unternehmen, Erich Schmidt, 1994.

[Spir96] Spirius, P.: *Entwicklung eines Kennzahlen-Systems als Bestandteil eines Executive Information System*, Diplomarbeit, Fachhochschule Wedel, 1996.

[Stud98] Studzinski, M.: *Von der Geschäftsprozeß- zur Workflow-Modellierung -eine objektorientierte Methodik*, Diplomarbeit, Fachhochschule Hamburg, 1998.

[StWe89] Stadler, M. / Wess, S.: *PATDEX-Konzept und Implementierung eines fallbasierten analogieorientierten Inferenzmechanismus zur Diagnose eines CNC-Bearbeitungszentrums*, Projektarbeit, Universität Kaiserslautern, 1989

[Szyp98] Szyperski, C.: *Component Software*, Addison-Wesley, 1998.

[Thal94] Thalheim, B.: *Database Design Strategies*, Report, BTU Cottbus, 1994.

[Tha+95] Thalheim, B. / Schewe, B. / Schewe, K.-D.*: Objektorientierter Datenbankentwurf in der Entwicklung datenintensiver Informationssysteme*, Informatik Forschung und Entwicklung, Band 10, Heft 3, , Springer, 1995.

[Thal97] Thalheim, B.: *Codesign von Struktur, Funktionen und Oberflächen von Datenbanken*, in: Informatik-Bericht I-05/1997, BTU Cottbus, 1997.

[Thal00] Thalheim, B.: *Entity-Relationship Modeling, Foundations of Database Technology*, Springer, 2000

[Turb93] Turban, E.: *Decision support and expert systems: management support systems*, Macmillan 1993.

[Ullm88] Ullman, J.: *Principles of Database and Knowledge-Base Systems, Vol 1*, Computer Science Press, 1988.

[VDM] *VDM Tutorial an der Fachhochschule Wedel*, www.fh-wedel.de/~si/VDMClassLib/ vdmtutor/ node11.html.

[WeGl94] Wess, S. / Globig, C.: *Cased-Based and Symbolic Learning- A Case Study.* In: Wess, S. / Althoff, K. / Richter, M..(Hrsg), Topics in Cased-Based Reasoning - selected papers from the first European Workshop on Cased-Based Reasoning, Kaiserslautern, 1993. LNAI, Nr. 837, Springer Verlag, 1994.

[Weiß96] Weiß, D./ Krcmar, H.: *Workflow-Management: Herkunft und Klassifikation*, Wirtschaftsinformatik 38, S.503-513, 1996.

[Wink96] Winkler, A.: *Konzeptionelle Erweiterung eines Informationssystems*, Diplomarbeit, Fachhochschule Wedel, 1996.

[Wess93] Wess, S.: *PATDEX - Inkrementelle und wissensbasierte Verbesserung von Ähnlichkeits-urteilen in der fallbasierten Diagnostik*. In: Tagungsband 2. Deutsche Expertensystem-tagung XPS-93. Hamburg: Springer Verlag. 1993.

[Wess95] Wess, S.: *Fallbasiertes Problemlösen in wissensbasierten Systemen zur Entscheidungsunterstützung und Diagnostik*; Dissertation, Universität Kaiserslautern, 1995.

[Wesk+98] Weske, M. / Hündling, J. / Kuropa, D. / Schuschel, H.*: Objektorientierter Entwurf eines flexiblen Workflow-Management-Systems*, Informatik Forschung und Entwicklung, Bd. 13, Heft 4, 1998.

[WfMC96] Workflow Management Coalition: *Terminology & Glossary*. Document Number WFMC-TC-1011.Document Status-Issue 2.0. Brüssel, Juni 1996.

[WfMC94] Workflow Management Coalition: *The Workflow Reference Model*. Document Status-Issue 1.1. Brüssel, November 1994.

[WoHe93] Woodman, M. / Heal, B.: *Introduction to VDM*, Mc Graw Hill, 1993.

[Yang97] Yang, Q.: *Intelligent Planning*, Springer International, 1997.

[Zie+95] Ziegler, J. / Janssen, C.: *Aufgabenbezogene Dialogstruktur für Informationssysteme*, in: Software-Ergonomie '95, German Chapter of the ACM, Teubner-Verlag, Stuttgart 1995.

[Zloo75] Zloof, M., *Query by Example*, Proc. AFIPS Natl. Comp. Conf. 44, 1975.

10 STICHWORTVERZEICHNIS

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

11 ANHANG

11.1 VDM - VIENNA DEVELOPMENT METHOD - NOTATION

Nachfolgend wird in Anlehnung an [VDM] ein kurzer Überblick zur VDM-Notation gegeben.

Terminologie:

Eine Modellspezifikation besteht in VDM aus einer Klassen-Definition (der sogenannten Domain-Gleichung), der Bedingung für die Wohlgeformtheit eines Objekts dieser Klasse und der Definition aller Funktionen, die mit diesen Objekten umgehen. Der Begriff **Klasse** bzw. **Domain** wird meist synonym mit dem Begriff "abstrakter Datentyp" (ADT) verwendet, ist aber allgemeiner und umfaßt *alle* möglichen Datentypen.

Definition: Eine **Domain-Gleichung** gibt den Namen und die syntaktische Struktur einer Klasse von Objekten an, d.h. aus welchen Bestandteilen ein Objekt besteht und wie diese Bestandteile strukturiert sind. Sie dient also als "Bauplan" für die Objekte.
Mit Daten repräsentiert ein Programm Objekte der Realität.

Wohlgeformtheit:
Die Bedingung für die **Wohlgeformtheit** ist eine Funktion, die die realen Zustände eines Objekts formalisiert und somit erlaubt, die Korrektheit eines Datums im Programm diesbezüglich zu testen. Ist der Zustand erlaubt, ist das Datum wohlgeformt.
Die Domain-Gleichung enthält keine Informationen darüber; daher ist es sinnvoll, ein Kriterium dafür zu definieren. Alle Funktionen, die ein neues Objekt der Klasse erzeugen oder ein existierendes manipulieren, müssen diese Bedingung kennen und einhalten. Eine andere Bezeichnung für diese Bedingung ist **Datentyp-Invariante** .

Semantische Funktionen: Die Funktionen, die mit den Objekten einer Klasse umgehen, diese also manipulieren, werden auch **semantische Funktionen** genannt. Im Gegensatz zum allgemeinen modellorientierten Ansatz können dies auch komplexe Manipulationen sein, die die Applikation benötigt.
Hilfsfunktionen sollten in *allen* komplexen Funktionen identifiziert werden. Sie bilden in der Regel eine Menge von für den ADT "charakteristischen" Funktionen, passen also in die Definition für Operationen.

Konstruktor: *Eine* Funktion muß in jedem Fall definiert werden, den **Konstruktor** für ein wohlgeformtes Objekt der Klasse festlegen. Natürlich kann es mehrere verschiedene Konstruktoren für eine Klasse geben, die sich durch die Parameter unterscheiden.

Die VDM-Spezifikation:

Nachdem nun die Begriffe bekannt sind, wird eine Spezifikation in VDM beschrieben. VDM unterstützt die repräsentationale Abstraktion durch die *Domain-Gleichungen*, die operationale Abstraktion durch die *Funktions-Spezifikation*.

Domain-Gleichung
Set (Menge) enthält eine beliebige Anzahl von Elementen des Elementtyps ohne festgelegte Reihenfolge. Einen Set $\mathcal{M}$ von Elementen des Typs $\mathcal{A}$ wird definiert als

$$\mathcal{M} = \mathcal{A}\text{-set}$$

Das **Tuple** (Liste) ist eine eindimensionale Tabelle, also eine Sequenz von gleichartigen Elementen *ohne* Obergrenzen. Einen Tuple $\mathcal{L}$ von Elementen des Typs $\mathcal{A}$ wird definiert als

$$\mathcal{L} = \mathcal{A}^*$$

Die **Map** (Abbildung) ist eine Funktion mit endlicher Argumentmenge, also die Menge aller Paare (*a*,*b*), für die gilt: jedem *a* ist höchstens ein *b* zugeordnet. die *a* heißen *Argumentwerte* , die *b* heißen *Map-Werte* (analog zum Begriff "Funktionswert"). In der Terminologie von VDM heißt die Argumentmenge auch *Argument-Domain*, die Zielmenge *Co-Domain* .

Eine Map $\mathcal{F}$ von Paaren (*a*,*b*) mit $a \in \mathcal{A}$ und $b \in \mathcal{B}$ wird definiert als

$$\mathcal{F} = \mathcal{A} \xrightarrow{m} \mathcal{B}$$

Der **Tree** (Verbund) ist eine geordnete Sammlung von Elementen unterschiedlichen Typs, die einzeln benannt sind.. Ein Tree $\mathcal{T}$ der Elementtypen $\mathcal{A}_1$ ... $\mathcal{A}_n$ wird definiert als

$$\mathcal{T} :: \mathcal{A}_1 \ldots \mathcal{A}_n$$

Wird ein Tree nur als Elementtyp innerhalb eines anderen ADT, braucht dieser keinen Namen zu bekommen. Statt dessen wird die Folge der Elementtypen in runde Klammern eingeschlossen.
Ein solcher Tree heißt **anonym**.

$$(\mathcal{A}_1 \ldots \mathcal{A}_n)$$

Die **Union** (Variante) enthält *ein* Datenelement, das aber einer Reihe verschiedener Datentypen angehören kann und den Elementtyp auch während seiner Existenz wechseln kann
Eine Union $\mathcal{U}$ der Elementtypen $\mathcal{A}_1$ ... $\mathcal{A}_n$ wird definiert als

$$\mathcal{U} = \mathcal{A}_1 \mid \ldots \mid \mathcal{A}_n$$

Das **Optional** enthält zu einem bestimmten Zeitpunkt entweder ein Objekt des Elementtyps oder den Wert **nil.** Ein Optional $\mathcal{O}$ von Elementen des Typs $\mathcal{A}$ wird definiert als

$$\mathcal{O} = [\mathcal{A}]$$

Zur Modellierung von Datentypen können VDM-Basistypen beliebig verschachtelt werden.

Datentyp-Invariante:
Die Bedingung für die Wohlgeformtheit eines Objekts ist ein sogenanntes **Prädikat**: die Beschreibung einer Eigenschaft, die ein Objekt haben kann oder nicht. Sie läßt sich als Funktion auffassen, die einen logischen Wahrheitswert zurückliefert. Zur Beschreibung eignen sich demzufolge die logischen Verknüpfungen, die Vergleichsoperatoren und alle weiteren Operatoren, die **true** und **false** als Ergebnis haben (hier eine Auswahl):

- AND($\wedge$)
- OR($\vee$)
- NOT($\neg$)
- Implikation ($\supset$ oder $\Rightarrow$)-- „wenn... dann...``
- Äquivalenz ($\equiv$)-- „...genau dann, wenn...``
- Gleichheit / Ungleichheit
- $<$/ $\leq$
- $>$/ $\geq$
- $\in$ / $\notin$
- $\subset$ / $\subseteq$ / $\not\subset$/
- $\forall$ / $\exists$

Funktionsspezifikation:
VDM propagiert die beiden üblichen Möglichkeiten der **konstruktiven** und der **impliziten Spezifikation** von semantischen Funktionen. Bei beiden Arten wird zunächst die *Syntax* der Funktion und ihre *precondition* spezifiziert.
Syntaxdefinition: Zunächst muß die **Syntax** der Funktion mit der *TYPE*-Klausel definiert werden:

TYPE Name (Argumentliste) : Argumenttyp-Liste ® Resultattyp

Die *Argumentliste* enthält die Namen der Argumente, wie sie in der weiteren Spezifikation verwendet werden; werden sie nicht gebraucht, können sie weggelassen werden. Die Argumenttyp-Liste legt die Definitionsmenge der Funktion fest, der Resultattyp die Zielmenge. Handelt es sich um einen Konstruktor ohne Argumente oder um eine Prozedur ohne Resultat, kann der entsprechende Teil ebenfalls weggelassen werden.

***precondition*:** Die **precondition** stellt eine Vorbedingung für den Funktionsaufruf dar. Ist sie nicht erfüllt, darf die Funktion nicht aufgerufen werden, da sonst undefinierte Zustände entstehen können. Die *precondition* kann mit den gleichen Mitteln spezifiziert werden wie die Datentyp-Invariante, da beides Prädikate sind. Die Datentyp-Invariante ist eine implizite *precondition*, da die Funktion nur mit sinnvollen Zuständen der Objekte auch sinnvoll umgehen kann. Format der

pre: Bedingung

Die Bedingung wird als logischer Ausdruck formuliert, der entweder true oder false ergibt. Ist keine *precondition* nötig wird pre: true gesetzt oder weggelassen.

implizite Spezifikation fügt eine *postcondition* hinzu, ein weiteres Prädikat , das eine Aussage über den Programmzustand nach der Abarbeitung der Funktion macht, vorzugsweise über das Resultat der Funktion. Wie die Funktion dieses Resultat erreicht, wird nicht abgegeben. Format der *postcondition*:
post: Bedingung

Die **konstruktive Spezifikation** beschreibt die Arbeitsweise der Funktion möglichst exakt; VDM stellt dafür die Sprache ***META-IV*** mit einer Kombination aus funktionalen und prozeduralen Sprachelementen zur Verfügung. Das Funktionsergebnis wird durch *einen* Ausdruck definiert, der beliebig tief verschachtelt sein kann; innerhalb der Funktion können lokale Variablen erzeugt werden, mit denen in einer tieferen Stufe gearbeitet wird. Eine Sequenz von Aktionen kann nur durch diese hierarchische Schachtelung ausgedrückt werden. Komplexe Unterfunktionen werden ausgelagert und separat spezifiziert.

Es folgen nun die wichtigsten Konstrukte der Sprache:
Das Konstrukt **let** *Ausdruck* **in** wertet den Ausdruck aus (damit wird häufig ein strukturiertes Objekt in seine Elemente zerlegt oder aus Elementen konstruiert) und übergibt die dabei temporär erzeugten Objekte (im Beispiel eset) dem „Unterprogramm``, das auf **in** folgt.

*define * let define IN ;*

Sehr oft werden Elemente aus zusammengesetzten Objekten entnommen. Das sieht im einzelnen so aus:
Sets: ***let*** $elem \in Set$ ***in*** *-- zufällige Auswahl eines Elements*

Tuples: ***let***
** elem = hd Tup* ***in*** *-- Auswahl des ersten Elements*
** subtup = tl Tup* ***in*** *-- Auswahl des Tuples ohne „head``*
** elem = last Tup* ***in*** *-- Auswahl des letzten Elements*
** elem = Tup(i)* ***in*** *-- Auswahl des i.ten Elements*

Maps: ***let***

elem1 Î dom Map ***in*** *-- zufällige Auswahl eines Argumentwertes*

elem2 Î rng Map ***in*** *-- zufällige Auswahl eines Map-Wertes*
elem2 = Map(elem1) ***in*** *-- Auswahl eines bestimmten Map-Wertes*

Trees: ***let***
$Tree = mk-tree(a_1, \ldots, a_n)$ ***in*** *-- Konstruktion eines Tree aus Komponenten*
$mk-tree(a_1, \ldots, a_n) = Tree$ ***in*** *-- Zerlegung eines Tree in Komponenten*

Spezifikation eines **Funktiontyps** erfolgt:
A ® B
$A \xrightarrow{\sim} B$

Der **Funktionskonstruktor**:
f(a) ≙ expr

Mit **dcl** werden globale, statische Variablen definiert; ihnen wird mit dem Zuweisungsoperator := ein Wert zugewiesen, auf ihren Inhalt wird mit **c** (Contents-Operator) zugegriffen.
Beispiel: *Erzeugung mit Initialisierung und Datentyp:*

dcl *var := init-value* ***type*** *Typname*

Zuweisung:
var := new-value
Zugriff auf den Inhalt:
var := func(cvar)

if ***Ausdruck*** **then** ***Klausel*** **else** ***Klausel*** entspricht der bekannten **if**-Struktur.

Der **switch**-Struktur entspricht folgendes Konstrukt:
(cases *Ausdruck0:*
Ausdruck1 → Klausel1,
Ausdruck2 → Klausel2, ...
Ausdruck-n → Klausel-n,
T *→ Default-Klausel)*

Für verschachtelte **if...then...else if...**gibt es die **case**-ähnliche Struktur
(Ausdruck1 → Klausel1,
Ausdruck2 → Klausel2, ...
Ausdruck-n → Klausel-n,
T *→ Default-Klausel)*

Die **for**-Struktur werden durch folgende Konstrukte umgesetzt:

1. ([(*Ausdruck*)]) (*Klausel*) -- für alle Elemente eines Set, für die der (optionale) Ausdruck erfüllt ist (der Set kann z.B. auch die Argumentmenge einer Map sein)
2. **for** ***i*** **in** *Tuple* **do** (*Klausel*) -- für alle Elemente einer Liste

Die **while**-Struktur ist ebenfalls vorhanden:
while *Ausdruck* do (*Klausel*)

Für Tuples, Sets und Maps stehen Konstruktoren zur Verfügung, die die Elemente durch ein Prädikat beschreiben (im obigen Beispiel wurde ein solcher für *eset* verwendet):

```
Sets:     { a | Ausdruck(a) }

Tuples:   < a | Ausdruck(a) >

Maps:     [ a ↦ b | Ausdruck(a,b) ]
```

Es stehen die üblichen arithmetischen und logischen Verknüpfungen zur Verfügung:

+, -, x, ÷, sowie ∧, ∨, ¬

Test, ob alle Elemente einer Menge eine bestimmte Eigenschaft haben:

$(\forall e \in A)\ (Predicate(e))$

Test, ob mindestens ein Element einer Menge eine bestimmte Eigenschaft hat:

$(\exists e \in A)\ (Predicate(e))$

Undefinierte Zustände werden mit **undefined** gekennzeichnet.

11.2 EPK - EREIGNISGESTEUERTE PROZESSKETTEN - NOTATION

Um Geschäftsprozesse auf unterschiedlichen Beschreibungsebenen darstellen zu können, wurde innerhalb der ARIS-Architektur [Sche94] die Ereignisgesteuerte Prozeßkette (EPK) als Beschreibungstechnik entwickelt (siehe Abbildung 64). Die EPK ist in der Lage, die fachlichen Inhalte von Geschäftsprozessen realitätsgetreu abzubilden. Die in der Graphik dargestellten Objekttypen können aus fachkonzeptueller Sicht um Datenobjekte und Organisationseinheiten ergänzt werden. Diese Ergänzung erhöht die semantische Aussagekraft von Prozeßmodellen.

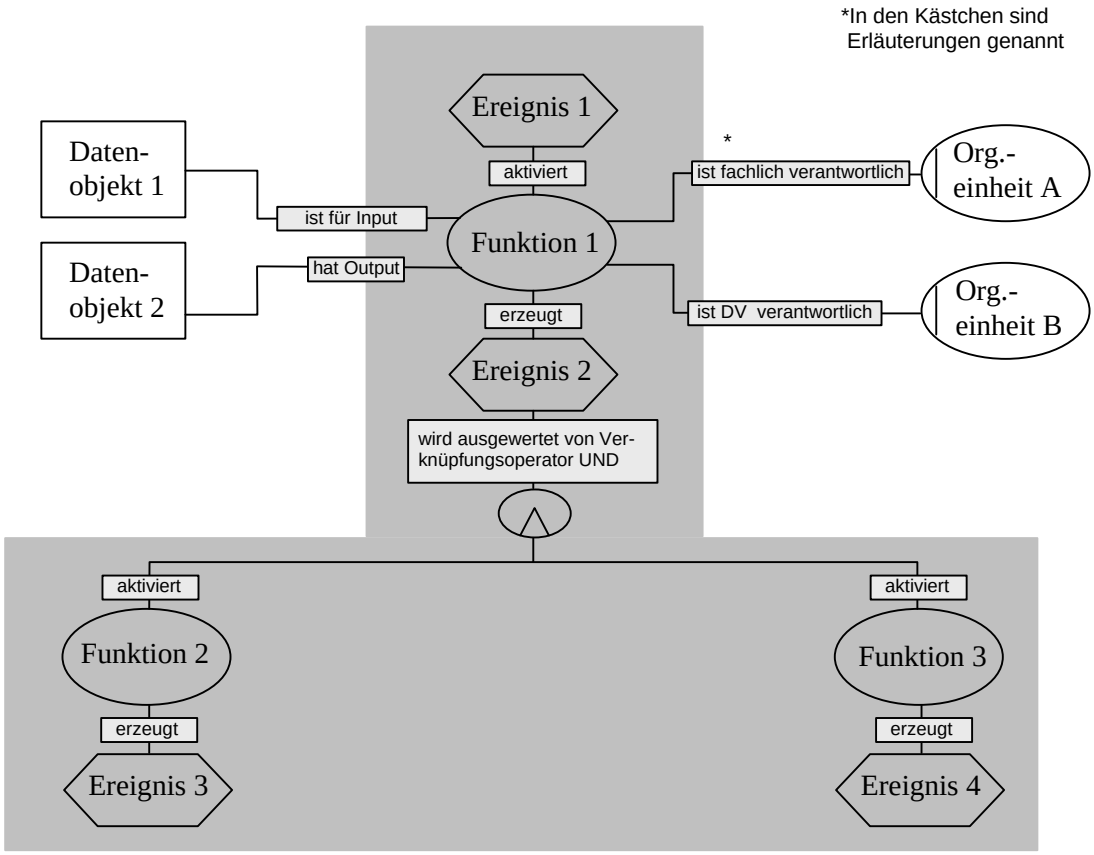


Abbildung 64: EPK mit Organisationseinheiten und Datenobjekten

Bei EPK handelt es sich um gerichtete, bipartite Graphen mit Ereignissen und Funktionen als Knotentyp. Ein Ereignis definiert das Eintreten eines Zustands, von dem der weitere Ablauf eines Geschäftsprozesses abhängt. Funktionen repräsentieren aktive Knoten und dienen zur Verarbeitung von Input- und Outputdaten. Zur Modellierung von nichtsequentiellen Abläufen werden Verknüpfungsoperatoren wie Konjunktion (AND), Adjuktion (OR) und Antivalenz (NOT) verwendet. Komplexe Bedingungen können mit Hilfe von Entscheidungs-tabellen realisiert werden. Durch die Verbindung der Elemente Ereignis, Funktion und Operator mit gerichteten Kanten wird ein Kontrollfluß repräsentiert. Die explizite Darstellung des Datenflusses wird bei EPK nicht unterstützt. Für jede Funktion können nur Input- und Outputdaten definiert werden.

11.3 DIALOGNETZE ZUR ABLAUFBESCHREIBUNG VON BENUTZERDIALOGEN

Nachfolgend werden Dialognetze, als spezielle Form von Petri-Netzen, für die adäquate Beschreibungstechnik von Dialogabläufen auf Fensterebene dargestellt [BuJa96].

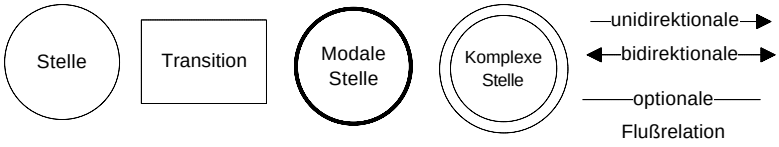


Abbildung 65: Komponenten eines Dialognetzes

Definition 1

Ein Dialognetz ist ein 6-Tupel DN = (S,T,F,t_0,b,B). Hierbei ist

S eine Menge von Stellen,

T eine Menge von Transitionen,

F eine Flußrelation, die nur zwischen Stellen und Transitionen oder umgekehrt besteht,

t_0 heißt Starttransition und hat keine Eingangsstelle, mindestens aber eine Ausgangsstelle,

B ist eine Menge von Beschriftungen und

b: $S \cup T \rightarrow B$ heißt Beschriftungsfunktion mit $b(t_0)$ = "Start".

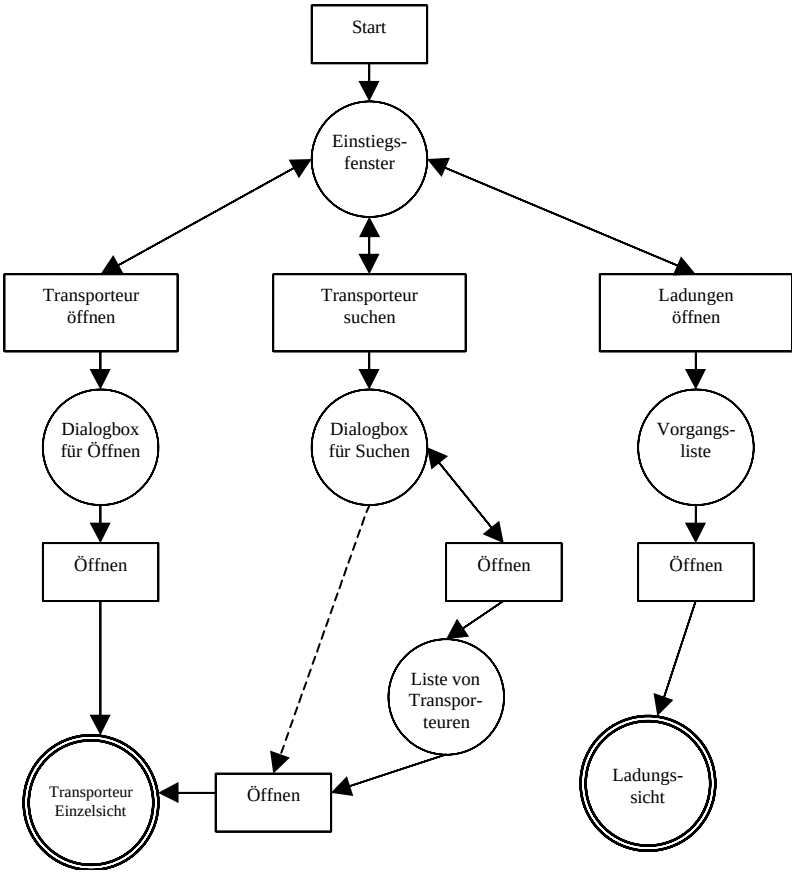


Abbildung 66: Dialognetz eines Grundmusters für Dialogabläufe

Das in Abbildung 66 dargestellte Dialognetz [Zie+95, S.403] beschreibt einen typischen Einstiegsdialog für objektorientierte Dialogabläufe. Hierbei ergeben sich die Eingangs- und Ausgangsstellen einer Transition $t \in T$ eines Dialognetzes DN = (S,T,F,t_0,b,B) zu:

${}^\bullet t = \{s \mid (s,t) \in F\}$ die Menge der Eingangsstellen von t,

$t^\bullet = \{s \mid (t,s) \in F\}$ die Menge der Ausgangsstellen von t,

${}^\bullet t \cap t^\bullet$ heißt die Menge der Nebenstellen von t (z.B. "Einstiegsfenster" als Nebenstelle der Transitionen "Transporteur öffnen", "Transporteur suchen" und "Ladungen öffnen").

Definition 3:

Sei DN = (S,T,F,t_0,b,B) ein Dialognetz.

Die möglichen Schaltfolgen in DN ergeben sich aus den folgenden Schaltregeln für Transitionen:

Schaltbedingung:

(1) Eine Transition t ∈ T kann schalten, wenn alle Eingangsstellen markiert und alle reinen Ausgangsstellen (s ∈ t•\•t) nicht markiert sind.

(2) t_0 ist zu Beginn eines Dialoges aktiviert.

(3) beim Schalten von t werden die Marken in den reinen Eingangsstellen (s ∈ •t\t•) entfernt und die Ausgangsstellen markiert.

Definition 4:

Wird eine komplexe Stelle markiert, so tritt das Start-Ereignis für das Unternetz ein. Wird die Marke aus einer komplexen Stelle abgezogen, so führt dies automatisch zum Abzug aller Marken aus dem Unternetz, also zu dessen Deaktivierung. Ebenso wird die Marke in einer komplexen Stelle entfernt, wenn innerhalb des Unternetzes alle Marken entfernt werden. Ansonsten werden Haupt- und Unternetz unabhängig voneinander geschaltet.

Definition 5:

Optionale Stellen beeinflussen nicht die Schaltbedingung. Beim Schalten einer Transition werden aber evtl. vorhandene Marken in optionalen Eingangsstellen entfernt und die optionalen Ausgangsstellen und Nebenstellen markiert, falls sie es noch nicht sind.

Definition 6:

Ist eine modale Stelle markiert, können systemweit (auch bei der hierarchischen Gliederung von Dialognetzen) nur noch Transitionen schalten, die diese als Eingangsstelle haben. Jede Transition kann maximal eine modale Stelle als Ausgangsstelle haben. Sind mehrere modale Stellen zur Zeit geöffnet, so können nur die Transitionen schalten, die die zuletzt geöffnete modale Stelle als Eingangsstelle haben.

Definition 7:

Ein voll spezifiziertes Dialognetz ist ein Dialognetz VDN = (S,T,F,t_0,b,B,b_S,b_T,E,C,A).

Hierbei ist

E eine Menge von Ereignissen,

C eine Menge von Bedingungen und

A eine Menge von Aktionen.

b_S: S → A×C×A heißt spezifizierende Stellenbeschriftung und

b_T: T → E×C×A spezifizierende Transitionenbeschriftung.

Es gelte stets $b_T(t_0)$ = ("Start","true",a), wobei "Start" das Start-Ereignis, "true" die immer wahre Bedingung und a eine Aktion aus A bezeichne.

Ferner sei b_s eindeutig bezüglich der Zuordnung eines c ∈ C für alle s ∈ S und b_T sei eindeutig für alle Transaktionen bezüglich der Zuordnung eines Paares (e,c) ∈ E×C.

Definition 8:

Sei VDN = (S,T,F,t_0,b,B,b_S,b_T,E,C,A) ein voll spezifiziertes Dialognetz. Für eine Transition t mit $b_T(t)$ = (e_t,c_t,a_t) werden die Schaltregeln aus Definition 3 wie folgt erweitert:

(1a) t kann nur schalten, wenn e_t eingetreten und c_t wahr ist.

(2a) Zu Beginn eines Dialoges trete das Startereignis "Start" ein.

(3a) Beim Schalten einer Transition t werden folgende Aktionen ausgeführt (a_{os} und a_{cs} bezeichnen die Öffneaktion, bzw. die Schließaktion, also $b_s(s)$ = (a_{os},c_s,a_{cs})):

- in nicht festgelegter Reihenfolge a_{cs} für alle reinen Eingangsstellen s ∈ •t\t• ,

- a_t,

- in nicht festgelegter Reihenfolge a_{os} für alle reinen Ausgangsstellen s ∈ t•\•t.