Universität Kaiserslautern
Fachbereich Informatik
AG Datenbanken und Informationssysteme
Prof. Dr. Theo Härder

# X-Translate – Benutzerdefinierte Spracherweiterungen für SQL:1999

**Diplomarbeit**

von

Boris Stumm

**Betreuer:**

Jernej Kovse, Wolfgang Mahnke

Januar 2003

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig und unter ausschließlicher Verwendung der angegebenen Literatur angefertigt habe.

Kaiserslautern, den 28. Januar 2003

Boris Stumm

# Inhaltsverzeichnis

# Kapitel 1 Einleitung

In dieser Arbeit wird ein Ansatz für ein Softwaresystem vorgestellt, mit dem SQL:1999 modular um neue Sprachmittel ergänzt werden kann. Der Ansatz hat viele Gemeinsamkeiten, aber auch viele Unterschiede, mit *Intentional Programming* (IP). IP wird in Abschnitt 5.3 vorgestellt, und die Unterschiede und Gemeinsamkeiten zu X-Translate dargestellt. Abschnitt 1.1 gibt einen Überblick über den Zweck der Existenz verschiedener Sprachen und Spracherweiterungen. Ein kurzer Einblick in die Datenbank-Anfragesprache SQL folgt in Abschnitt 1.2. Abschnitt 1.3 beschreibt kurz das in der Arbeit besprochene System. In Abschnitt 1.4 wird der weitere Aufbau dieser Arbeit erläutert.

## 1.1 Sprachen und Spracherweiterungen

Zur Entwicklung von Softwaresystemen wird eine Vielzahl von Sprachen eingesetzt. Das liegt darin begründet, dass die zu lösenden Probleme bei der Entwicklung je nach Anwendung stark voneinander abweichen. Es gibt jedoch keine echte Universalsprache; je nach Problemstellung eignet sich die eine oder die andere Sprache besser zur Implementierung. Das hängt normalerweise nicht von der Mächtigkeit der Sprache (im Sinne der Turing-Vollständigkeit) ab; alle gängigen Programmiersprachen, von C, C++, Java über SQL:1999 bis hin zu XSLT sind Turing-vollständig.

Ein Kriterium für die Auswahl einer Sprache ist neben der Anzahl der unterstützten Plattformen und dem Zugriff auf Netzwerk, Ein- und Ausgabegeräte, Dateisystem usw. auch die *Anwenderfreundlichkeit*. Die *Anwender* sind hier Softwareentwickler und Programmierer. Die *Freundlichkeit* drückt sich in mehreren Punkten aus.

- Unterstützung der in Design und Entwurf verwendeten allgemeinen Konzepte und Abstraktionen durch die Sprache, z.B. Objektorientierung.
- Angebot von Abstraktionen für die Lösung anwendungsspezifischer Probleme.
- Geeignete Werkzeugunterstützung wie Compiler, Debugger und integrierte Entwicklungsumgebungen.

### 1.1.1 Universelle Sprachen

Universelle Programmiersprachen (engl. general purpose languages, GPL) dienen als Grundlage zur Implementierung von Anwendungen. Sie stellen Sprachabstraktionen zur Verfügung, die zur Lösung der meisten Probleme gebraucht werden. Um effiziente Anwendungen zu entwickeln, wird oft C++ benutzt. In den letzten Jahren gewinnt Java als plattformunabhängige Sprache immer mehr Bedeutung.

SQL ist eine universelle Datenbank-Anfragesprache. Alle bedeutende Datenbankverwaltungssysteme (DBVS) unterstützen SQL. Im Gegensatz zu den Programmiersprachen ist SQL immer im Verbund mit einem DBVS zu sehen.

### 1.1.2 Domänenspezifische Sprachen

Domänenspezifische Sprachen (engl. domain specific languages, DSL) werden zur Lösung spezieller Probleme in einem relativ eng umrissenen Feld eingesetzt. Eine Domäne beschränkt sich möglicherweise auf eine einzelne Anwendung, kann aber auch weiter gefasst sein. SQL wird z.B. oft als domänenspezifische Sprache zur Abfrage von Datenbanken gesehen. Andere Beispiele für domänenspezifische Sprachen sind HTML, BNF und LaTeX. Eine ausführlichere Diskussion über die Definition von Domänen findet sich in [CE00].

Statt die domänenspezifischen Abstraktionen in eine Sprache zu fassen können auch Bibliotheken mit den entsprechenden Funktionen benutzt werden. Im Unterschied zu Bibliotheken haben DSLs den entscheidenden Vorteil, das die in der Domäne verwendeten Abstraktionen direkt Elemente der Sprache sind. Das vereinfacht die Handhabung erheblich. Der Code wird übersichtlicher und intentionaler, wie Simonyi das für IP beschreibt [Si96].

### 1.1.3 Erweiterung von Sprachen

Domänenspezifische Abstraktionen werden oft als Erweiterung einer bereits vorhandenen Sprache bereitgestellt. GPLs können auch um allgemeine, nicht domänenspezifische Konzepte erweitert werden. Hier ist zu unterscheiden zwischen eingebetteten Spracherweiterungen und Erweiterungen mit externem Einfluss [CE00]. Eingebettete Spracherweiterungen haben keinen Einfluss auf die Semantik der erweiterten Sprache. Ein Beispiel hierfür ist SQLJ. Erweiterungen mit externem Einfluss wirken sich hingegen auf die Semantik der erweiterten Sprache aus, z. B. die in Abschnitt 2.2 vorgestellte Unterstützung für Mehrfachvererbung. Die Nutzung einer Spracherweiterung schließt meist die Nutzung weiterer Erweiterungen aus, da beide Spracherweiterungen die Konstrukte der anderen jeweils als Syntaxfehler abweisen und somit eine Kompilierung unmöglich machen.

## 1.2 SQL und SQL:1999

SQL ist eine Anfragesprache für relationale Datenbanksysteme. 1970 legte Dr. E. F. Codd den Grundstein für SQL mit der Beschreibung eines relationalen Modells für Datenbanken. Darauf folgten um 1980 kommerzielle Implementierungen von Datenbanksystemen. Der erste SQL-Standard wurde 1989 vom American National Standards Institute (ANSI) veröffentlicht. Drei Jahre später folgte SQL-92. Der derzeit aktuelle Standard ist SQL:1999 [ISO99a]. SQL:1999 ist wesentlich umfangreicher als SQL-92 und enthält außer den relationalen Merkmalen objektrelationale Erweiterungen sowie PSM, eine prozedurale Sprache, die SQL Turing-vollständig macht. Weitere Merkmale des aktuellen Standards sind die Unterstützung von benutzerdefinierten Routinen in verschiedenen Sprachen und ein im Vergleich zu SQL-

92 geändertes Konformitäts-Konzept: in SQL-92 gab es mehrere Stufen der Konformität zum Standard, die Hersteller erfüllen konnten. In der Praxis zeigte sich jedoch, das die Hersteller, wenn überhaupt, nur zur ersten Stufe voll konform waren, jedoch ausgewählte Merkmale aus anderen Stufen unterstützten. SQL:1999 besteht hingegen nur aus einem Sprachkern (engl. Core SQL) und optionalen Zusatzmerkmalen (Features). Das soll es einfacher machen, standard-konform zu sein. Hersteller müssen lediglich den Sprachkern voll implementieren, und können dann eine Liste der unterstützten Merkmale angeben. Bis jetzt unterstützt aber noch kein Hersteller den SQL:1999-Standard, auch nicht den Sprachkern. Andererseits bieten praktisch alle Hersteller eigene Erweiterungen an. Das führt zu einigen Problemen mit der praktischen Verwendbarkeit von SQL:1999-Spracherweiterungen. Auf diese Probleme wird in dieser Arbeit an den entsprechenden Stellen eingegangen.

Der jeweils aktuelle SQL-Standard ist deshalb eher als Empfehlung denn als Standard zu sehen; bis er von Herstellern implementiert wird, ist schon die Folgeversion des Standards verabschiedet. Als Basis dieser Arbeit dient der SQL:1999-Standard. Es wird auch ein Ansatz zur Abbildung von SQL:1999 auf Herstellerdialekte vorgestellt.

Der nächste SQLStandard wird vermutlich 2003 veröffentlicht und SQL:2003 heißen. Der aktuelle Entwurf zeigt, das es wenig Änderungen sondern eher Erweiterungen geben wird. Die Ergebnisse dieser Arbeit lassen sich daher problemlos auch auf SQL:2003 anwenden.

## 1.3 X-Translate

Das in dieser Arbeit vorgestellte X-Translate-System bietet eine Umgebung zur Entwicklung und zum Einsatz von Spracherweiterungen für SQL:1999. Sehr wichtig ist die Möglichkeit, *mehrere* Erweiterungen *gleichzeitig* einzusetzen.

So können Erweiterungen von allgemeinem Nutzen (z.B. Schemamodularität, Abschnitt 2.3) zusammen mit domänenspezifischen Erweiterungen eingesetzt werden. Spracherweiterungen, die herstellerspezifische Erweiterungen simulieren, können benutzt werden, um schon existierende SQL-Schemas und -Skripte auf andere DBVS zu portieren.

Das X-Translate-Laufzeitsystem transformiert alle Anweisungen, die Spracherweiterungen nutzen, in SQL:1999-konforme Anweisungen. Danach werden diese auf den jeweiligen Herstellerdialekt abgebildet. Spezielle Hersteller-Spracherweiterungen bleiben bei entsprechenden Einsatz von Herstellermodulen und Herstelleranpassungen davon unangetastet. Das erweiterte SQL:1999 wird, unabhängig davon, wie viele und welche Spracherweiterungen genutzt werden, als XSQL bezeichnet.

Die Erweiterbarkeit von SQL:1999 hat Grenzen, und es gibt eine Vielzahl von Konfliktmöglichkeiten zwischen zwei Spracherweiterungen. Einige dieser Konflikte können mit X-Translate umgangen bzw. gelöst werden. Es bleiben jedoch einige Punkte, an denen Konflikte nicht vermieden werden können. Das ist teilweise abhängig von der Qualität (bezüglich der Implementierung) der eingesetzten Spracherweiterungen, und teilweise von ihrer Größe und Komplexität. Hersteller von

Spracherweiterungen werden darauf achten, möglichst gute und kompatible Erweiterungen zu erzeugen. Eine Spracherweiterung, die mit möglichst vielen anderen Erweiterungen zusammenarbeitetet, hat einen marktwirtschaftlichen Vorteil gegenüber anderen, weniger kompatiblen Spracherweiterungen mit gleichen oder ähnlichen Funktionen. Das entspricht dem von Czarnecki und Eisenecker [CE00] beschriebenen *Markt für Intentionen* für IP.

## 1.4 Aufbau der Arbeit

In Kapitel 2 werden Beispiele für mögliche Spracherweiterungen vorgestellt. Drei ausgewählte Spracherweiterungen werden ausführlich und an Beispielen vorgestellt: semantisch reichhaltige Beziehungen [**?**], Mehrfachvererbung für SQL und eine Erweiterung für den modularen Schemaentwurf in SQL. Anhand dieser Beispiele werden in Kapitel 3 die Anforderungen an die verschiedenen involvierten Systembestandteile in einem Anforderungskatalog zusammen gestellt. Es wird beschrieben, wie ein die Anforderungen erfüllendes System aussehen und arbeiten muss. Weiterhin werden die zu lösenden Hauptprobleme diskutiert. Kapitel 4 stellt das X-Translate-System vor. Nach dem Erläutern der verwendeten Konzepte und einer Übersicht über die Architektur werden die drei X-Translate-Systemumgebungen Entwicklung, Installation und Laufzeit vorgestellt. Auf verwandte Arbeiten aus verschiedenen Bereichen der Informatik wird in Kapitel 5 eingegangen. Abschließend wird in Kapitel 6 die Arbeit zusammengefasst und kritisch hinterfragt. Vor- und Nachteile des X-Translate-Systems werden einander gegenübergestellt und diskutiert. Weiterhin wird ein Ausblick auf noch offene Fragen im Bezug zu X-Translate gegeben.

# Kapitel 2 Anwendungsbeispiele

In diesem Kapitel werden einige mögliche Spracherweiterungen für SQL:1999 vorgestellt und an Beispielen erläutert. Nach einer Motivation zur jeweiligen Erweiterung wird diese kurz beschrieben. Danach werden anhand einiger Beispiele die im Zusammenhang mit XSQL wichtigen Punkte herausgestellt.

## 2.1 Semantisch reichhaltige Beziehungen

Ein wichtiges Konzept in objektorientierten Datenmodellen sind die Beziehungen zwischen Objekten. In SQL:1999 können Objekte per Referenz oder per PK/FK-Paar (Primärschlüssel/Fremdschlüssel-Paar) miteinander assoziiert werden. Eine solche Assoziation hat nur eine schwache Semantik, die fest vorgegeben ist.

Um ternäre oder höhergradige Beziehungen zu modellieren, müssen zusätzliche Hilfsmittel, wie z.B. Tabellen zum Halten der Beziehungsinstanzen, benutzt werden. SQL:1999 unterstützt lediglich eine Art von Beziehung mit impliziter Semantik: die Spezialisierungs-Beziehung bei Typen und Tabellen. Wenn Typ A von Typ B spezialisiert wird, können Objekte vom Typ B anstelle von Objekten vom Typ A treten, umgekehrt ist das jedoch nicht der Fall.

Beziehungen mit komplexer Semantik können im Nachhinein oft nur mit Hilfe der Dokumentation ausgemacht werden, da sie im Modell nicht sichtbar sind. Dadurch wird ein solches Datenmodell unübersichtlich und schlecht wartbar.

Zhang [Zh01] stellt in ihrer Dissertation *ORIENT* (**O**bject-based **R**elationship **I**ntegration **EN**vironmen**T**), eine Umgebung zur Integration von semantisch reichhaltigen Beziehungen (engl. semantically rich relationships) in ORDBVS vor.

Dort führt sie *semantisch reichhaltige Beziehungen* als neue Sprachkonstrukte in SQL ein. Durch das explizite Definieren von Beziehungen inklusive ihrer Semantik entsteht ein leichter verständliches und übersichtlicheres Datenmodell. Das Vorhandensein von versteckten Abhängigkeiten zwischen Objekten wird weniger wahrscheinlich, weiterhin ist die Semantik einer Beziehung im Modell definiert und nicht ausschließlich in der Dokumentation.

### 2.1.1 Beschreibung

Hauptaufgabe von OrientSQL ist das Bereitstellen von Sprachkonstrukten für semantisch reichhaltige Beziehungen. Bei der Definition solcher Beziehungen wird nicht nur die Struktur, sondern auch die Bedingungen für das Verhalten bei verschiedenen Anweisungen der Datenmanipulationssprache (engl. data manipulation language, DML) festgelegt.

Um die Definition von anwendungsspezifischen Beziehungen zu vereinfachen, gibt es eine Menge von vordefinierten Beziehungen, die spezialisiert werden können. Das geschieht durch einen Vererbungsmechanismus.

### Semantik der Beziehungen

Im Folgenden werden die Begriffe *Beziehung* und *Teilnehmer* zur Bezeichnung von Beziehungstypen und Teilnehmertypen gebraucht. Für Instanzen werden die Bezeichnungen *Beziehungsinstanz* und *Teilnehmerinstanz* verwendet.

#### Attribute

Für jede Beziehung können, ähnlich wie bei benutzerdefinierten Typen, Attribute definiert werden.

#### Strukturelle Eigenschaften

Es lassen sich mehrere strukturelle Eigenschaften von Beziehungen identifizieren.

- Der *Grad* einer Beziehung definiert die Anzahl der Teilnehmer. Es gibt binäre, ternäre und höhergradige (n-stellige) Beziehungen.
- Die *Kardinalität* eines Teilnehmers bestimmt, wie viele Instanzen dieses Teilnehmers mit einer bestimmten Instanz der Beziehung assoziiert sein können. Weiterhin können minimale und maximale Kardinalitäten angegeben werden, z.B. [1, 10].
- *Kompositionen* oder Verbünde sind Beziehungen, die einen ausgezeichneten Teilnehmer, den *Eigner* haben. Außer dem Eigner gibt es Teilnehmer und optionale Teilnehmer. Operationen auf dem Eigner werden an die anderen Teilnehmer weitergegeben.
- Die *Exklusivität* (engl. sharability) gibt bei Kompositionen an, ob eine Teilnehmerinstanz in mehr als einer Beziehungsinstanz teilnehmen kann. Wenn z.B. eine Teilnehmerinstanz vom Typ `Abbildung` (ein Teilnehmer) in mehreren `Kapiteln` (den Eignern) vorkommen kann, ist sie nicht exklusiv. Wenn eine Abbildung maximal einmal vorkommen darf, ist sie exklusiv.

#### Operationale Eigenschaften

Neben den strukturellen gibt es noch operationale Eigenschaften von Beziehungen, die festlegen, in wie weit sich Operationen auf einer Teilnehmerinstanz auf assoziierte Teilnehmerinstanzen und die Beziehungsinstanzen auswirken. Hier lassen sich drei Typen von Eigenschaften feststellen:

- Die *existenzielle Abhängigkeit* einer Teilnehmerinstanz von anderen Teilnehmerinstanzen gibt an, ob diese Teilnehmerinstanz unabhängig existieren kann oder ob das Vorhandensein anderer Teilnehmerinstanzen vorausgesetzt wird.
- Mit *operationaler Transitivität* ist die Propagation von Operationen von einem Teilnehmer auf andere Teilnehmer der Beziehung gemeint.

- Die *Verletzung der Kardinalitäten*, d.h. Über- oder Unterlauf der maximalen/minimalen Kardinalität eines Teilnehmers kann sich auf andere Teilnehmer auswirken.

Die operationalen Eigenschaften können als Folge von Aktionen verstanden werden, die durch eine Operation auf einem Teilnehmer angestoßen wird: die Operation wird propagiert. Abhängig von dem Typ der Operation und dem Operand lassen sich jeweils verschiedene Alternativen für die Art der Propagation erkennen:

- *Löschen einer Teilnehmerinstanz*:
  - Isoliertes (engl. isolated) Löschen: Das Löschen einer Teilnehmerinstanz hat keinen Einfluss auf andere Teilnehmerinstanzen.
  - Obligatorisches (engl. mandatory) Löschen: Beim Löschen einer Teilnehmerinstanz werden alle assoziierten Teilnehmerinstanzen gelöscht sowie die zugehörigen Beziehungsinstanzen.
  - Bedingtes (engl. conditional) Löschen: Beim Löschen einer Teilnehmerinstanz werden alle assoziierten Teilnehmerinstanzen gelöscht, die nicht auch noch in einer anderen Beziehungsinstanz teilhaben. Die betroffene Beziehung wird auch gelöscht.
  - Beschränktes (engl. restricted) Löschen: Eine Teilnehmerinstanz kann nur im Verbund mit allen assoziierten Teilnehmerinstanzen gelöscht werden.
- *Löschen einer Beziehungsinstanz*:
  - Isoliertes Löschen: Das Löschen einer Beziehungsinstanz hat keinen Einfluss auf die Teilnehmerinstanzen.
  - Obligatorisches Löschen: Beim Löschen einer Beziehungsinstanz werden alle Teilnehmerinstanzen gelöscht.
  - Bedingtes Löschen: Beim Löschen einer Beziehungsinstanz werden alle Teilnehmerinstanzen gelöscht, falls sie nicht noch in einer anderen Beziehungsinstanz teilnehmen.
- *Einfügen einer Teilnehmerinstanz*:
  - Isoliertes Einfügen: Das Einfügen einer Teilnehmerinstanz hat keinen Einfluss auf andere Teilnehmerinstanzen.
  - Bedingtes Einfügen: Beim Einfügen einer Teilnehmerinstanz wird die zugehörige Beziehungsinstanz erzeugt und für die anderen Teilnehmerinstanzen Platzhalter angelegt.
  - Beschränktes Einfügen: Beim Einfügen einer Teilnehmerinstanz wird die zugehörige Beziehungsinstanz erzeugt. Alle zugehörigen Teilnehmerinstanzen müssen schon existieren.
- *Selektion von Teilnehmerinstanzen*:
  - Isolierte Selektion: Nur die angegebene Teilnehmerinstanz wird selektiert.

  - Obligatorische Selektion: Außer der angegebenen Teilnehmerinstanz werden alle assoziierten Teilnehmerinstanzen mit selektiert.
  - Bedingte Selektion: Außer der angegebenen Teilnehmerinstanz werden alle Teilnehmerinstanzen der angegebenen Beziehungsinstanz mit selektiert.

- *Homogene Änderungen*: Bei homogenen Änderungen werden die aktualisierten Attribute an andere Teilnehmerinstanzen weitergegeben.
  - Isolierte Änderung: Die Änderung einer Teilnehmerinstanz hat keine Auswirkungen auf andere Teilnehmerinstanzen.
  - Obligatorische Änderung: Bei der Änderung einer Teilnehmerinstanz werden die gemeinsamen Attribute in den anderen Teilnehmerinstanzen auch aktualisiert (z.B. das Datum der letzten Änderung).
  - Bedingte Änderung: Die Änderung einer Teilnehmerinstanz wirkt sich nur auf den Eigner aus.
  - Beschränkte Änderung: Es ist keine Änderung von Teilnehmerinstanzen erlaubt.

- *Heterogene Änderungen*: Bei heterogenen Änderungen verursacht die Aktualisierung einer Teilnehmerinstanz die Erzeugung einer neuen Teilnehmerinstanz. Zhang nennt hier als Beispiel die Versionierung von Datenbankobjekten. Alle Versionen sind Teilnehmerinstanzen einer Versionierungsbeziehung, und bei einer Änderung einer Version wird automatisch eine Folgeversion generiert.
  - Änderung ohne Einfügen (engl. update/no insertion): Die Änderung einer Teilnehmerinstanz hat keine Auswirkungen auf andere Teilnehmerinstanzen.
  - Änderung mit Kopieren (engl. update/copy insertion): Die aktualisierte Teilnehmerinstanz und alle anderen Teilnehmerinstanzen werden in eine neue Beziehungsinstanz kopiert. Die alte Beziehungsinstanz und deren Teilnehmerinstanzen bleiben unverändert.
  - Änderung mit Beibehaltung (engl. update/preserved insertion): Die aktualisierte Teilnehmerinstanz wird in eine neue Beziehungsinstanz aufgenommen. Die unveränderten Teilnehmerinstanzen werden in diese Beziehungsinstanz übernommen.

- *Kardinalitätsunterlauf*:
  - Obligatorisches Löschen: Alle assoziierten Teilnehmerinstanzen, sowie die betroffene Beziehungsinstanz, werden gelöscht.
  - Bedingtes Löschen: Nur die assoziierten Teilnehmerinstanzen, die nicht in anderen Beziehungsinstanzen teil haben, sowie die betroffene Beziehungsinstanz, werden gelöscht.
  - Beschränktes Löschen: Falls ein Kardinalitätsunterlauf auftreten würde, wird das Löschen der Teilnehmerinstanz verboten.

- *Kardinalitätsüberlauf*:
  - Einfügen mit Beibehaltung: Es wird eine neue Beziehungsinstanz erzeugt, und die Teilnehmerinstanzen der alten Beziehungsinstanz so aufgeteilt, dass keine weiteren Über- oder Unterläufe auftreten.
  - Einfügen mit Kopieren: Es wird eine neue Beziehungsinstanz erzeugt, und ein Teil der Teilnehmerinstanzen der alten Beziehungsinstanz kopiert.
  - Beschränktes Einfügen: Falls ein Kardinalitätsüberlauf auftreten würde, wird das Einfügen der Teilnehmerinstanz verboten.

#### Insert-Blöcke

Je nach Semantik einer Beziehung müssen mehrere Teilnehmerinstanzen gleichzeitig eingefügt werden. Dazu wird das Konzept der Insert-Blöcke eingeführt. In einem solchen Block können mehrere Einfügeoperationen hintereinander ausgeführt werden, die Konsistenz wird jedoch erst am Schluss des Blocks geprüft.

Konzeptionell ist ein solches Konstrukt nicht erforderlich, da SQL:1999 mit der Transaktionsunterstützung bereits passende Mittel bereitstellt. Der Grund für die Einführung von Insert-Blöcken war die einfachere Implementierung. Dieses Konstrukt ist auch für eine XSQL-Spracherweiterung sinnvoll: es ist einfacher, eine neue Semantik an eine neue Syntax zu binden, als ein schon vorhandenes syntaktisches Konstrukt zu erweitern.

#### Spezialisierung von Beziehungen

Wie auch bei Typen und Klassen kann eine Beziehung eine Spezialisierung einer anderen Beziehung sein. So können generische Beziehungen an Anwendungsbedürfnisse angepasst werden. Der Vererbungsmechanismus funktioniert etwas anders als in der Objektorientierung üblich. Es lassen sich Parallelen ziehen zu den C++-Templates: dort wird der Typ erst bei der Instanziierung eines Objektes festgelegt. Auf die Vererbung bei den semantisch reichhaltigen Beziehungen wird in Abschnitt 2.1.2 weiter eingegangen.

### 2.1.2 Beispiele

Anhand einiger Beispiele wird im Folgenden die OrientSQL-Syntax vorgestellt. Weiterhin wird auch auf die zu beachtenden Punkte bei Umsetzung von OrientSQL auf SQL:1999 mittels XSQL eingegangen.

#### Anlegen einer Beziehung

In Beispiel 2.1 werden zwei Beziehungen erzeugt. In den Zeilen 1-6 wird `aggregation` als generische Beziehung mit den zwei Teilnehmern `aggregate` und `part` angelegt.

In Zeile 2 wird der Eigner der Beziehung definiert. Zeile 3 spezifiziert, dass beim Löschen des Eigners alle anderen Teilnehmerinstanzen gelöscht werden sollen. Das Verhalten bei einem `SELECT` wird in Zeile 4 als bedingte Selektion spezifiziert. Für alle nicht angegebenen operationalen Eigenschaften werden Vorgabewerte benutzt.

Beispiel 2.1: Anlegen der Beziehungen `aggregation` deren Spezialisierung `includes`

```
 1  CREATE RELATIONSHIP aggregation (
 2      aggregate OWNER
 3          ON DELETE MANDATORY DELETION
 4          ON SELECT CONDITIONAL SELECTION,
 5      part PARTICIPANT
 6  );
 7
 8  CREATE RELATIONSHIP includes UNDER aggregation (
 9      chapter OWNER (aggregate)
10      figure PARTICIPANT (part)
11          NON SHARABLE
12          CARDINALTIY [0,50]
13          ON INSERT CONDITIONAL INSERTION
14      table PARTICIPANT (part)
15          NON SHARABLE
16          CARDINALITY [0,20]
17          ON INSERT CONDITIONAL INSERTION
18  );
```

Die `includes`-Beziehung (Zeilen 8-18 in Beispiel 2.1) ist eine Spezialisierung von `aggregation`. Es wird in den Zeilen 9, 10 und 14 ersichtlich, dass die Spezialisierung etwas anders funktioniert als Vererbung in SQL, Java oder C++. Jedem Teilnehmer der spezialisierten Beziehung wird die Rolle eines Teilnehmers der übergeordneten Beziehung zugewiesen. Hierbei können verschiedene Teilnehmer in der Spezialisierung die gleiche Rolle übernehmen; so sind `figure` und `table` beide ein `part`.

Auch in der `includes`-Beziehung werden operationale Eigenschaften spezifiziert. Darüberhinaus sind jedoch auch zwei strukturelle Eigenschaften angegeben: die Kardinalität und die Exklusivität (Zeilen 11-12, 15-16). `NON SHARABLE` bedeutet hier, dass eine Abbildung oder eine Tabelle maximal in einer `includes`-Beziehung teilnehmen kann.

Die Abbildung der bloßen Datenstruktur auf SQL:1999 ist unproblematisch. Ein `CREATE RELATIONSHIP` kann in ein `CREATE TABLE` umgewandelt werden. Die so erzeugte Tabelle kann später die Beziehungsinstanzen aufnehmen. Hier ist jedoch zu beachten, dass für eine einzelne Beziehungsinstanz möglicherweise mehrere Tupel in diese Tabelle eingefügt werden, je nach den angegebenen Kardinalitäten der Teilnehmer. Um die Datenstrukturen zu normalisieren, kann es erforderlich sein, mehr als eine Tabelle anzulegen. Wenn die Kardinalität nach oben beschränkt ist (wie im Beispiel der Fall), kann man auch Arrays benutzen[1].

Weiterhin muss SQL-Code generiert werden, der für die Einhaltung der operationalen Eigenschaften verantwortlich ist. Hierzu kann man beispielsweise Constraints und Assertions benutzen. Eine andere Möglichkeit ist das Prüfen bzw. Anwenden

[1] Bei Arrays muss in SQL:1999 schon bei der Deklaration die maximale Größe angegeben werden.

der Bedingungen zur Übersetzungszeit. Dies ist unter anderem für die Selektions-Eigenschaften notwendig, da bei einer Selektion die Anfrage möglicherweise umgeschrieben werden muss.

Der dritte Punkt ist die Behandlung des Informationsschemas: Zum einen muss dieses um entsprechende Sichten auf die Beziehungs-Metadaten erweitert werden. Zum anderen müssen aber auch die durch die Abbildung auf SQL:1999 entstehenden Einträge im Informationsschema versteckt werden. Wenn ein `CREATE RELATIONSHIP` auf ein `CREATE TABLE` abgebildet wird, darf die dadurch erzeugte Tabelle nicht im Informationsschema auftauchen.

### Einfügen einer Beziehungsinstanz

In Beispiel 2.2 wird in die in Beispiel 2.1 erzeugte Beziehung eine Instanz eingefügt. Die in spitzen Klammern eingefassten Bezeichner stehen für die Referenzen[2] auf die einzufügenden Teilnehmerinstanzen. Die geschweiften Klammern dienen zum Gruppieren mehrerer Instanzen des gleichen Teilnehmers.

Auch dieses Beispiel lässt sich leicht in eine Folge von SQL:1999-Anweisungen umsetzen. Neben der direkten Übersetzung von `INSERT INTO RELATIONSHIP` in eine oder mehrere SQL-`INSERT`-Anweisungen müssen hier möglicherweise auch die Bedingungen, die für die Beziehung spezifiziert wurden, geprüft und eventuell eine Fehlermeldung ausgegeben werden.

Beispiel 2.2: Einfügen einer neuen Beziehungsinstanz

```
INSERT INTO RELATIONSHIP includes (chapter, figure, table)
VALUES (<Kapitel2>, {<Figur1>, <Figur2>}, {<Tabelle1>, <Tabelle2>});
```

### Erweiterte `SELECT`-Anweisung

In Beispiel 2.3 wird ein Kapitel anhand seines Namens selektiert. Es wird die in der `includes`-Beziehung spezifizierte Semantik zur Selektion benutzt. Da `includes` von `aggregation` erbt, wird eine bedingte Selektion durchgeführt. Die Ergebnismenge dieser Anweisung enthält also nicht nur alle passenden `chapter`-Teilnehmerinstanzen, sondern auch die assoziierten `figure` und `table`-Teilnehmerinstanzen.

Beispiel 2.3: Selektion eines Kapitels inklusive seiner Abbildungen und Tabellen

```
SELECT * FROM chapter (includes) c
WHERE c.name = 'Kapitel 2';
```

Im Gegensatz zu den vorigen Beispielen wird hier keine neue Anweisung eingeführt, sondern eine schon existierende syntaktisch erweitert. Für die Übersetzung muss jede `SELECT`-Anweisung auf das Vorhandensein dieser Syntaxerweiterung geprüft werden. Nur die Anweisungen, die die Erweiterungen nutzen, müssen umgeschrieben werden.

[2] Zhang [Zh01] verwendet den Begriff *Object Identifier* (OID).

SELECT-Anweisungen in SQL können beliebig komplex sein, was die Übersetzung erschwert. Zu beachten sind beispielsweise die Möglichkeiten von Subqueries und rekursiven Anfragen.

#### Unveränderte SQL:1999-Anweisung

Beispiel 2.4 zeigt eine normale SQL:1999-Anweisung, die selbst nicht übersetzt werden muss. Jedoch müssen eventuell weitere Anweisungen eingefügt werden (für die Anweisung im Beispiel sind auch alle assoziierten `table`s und `figure`s zu löschen), oder es muss eine Fehlermeldung ausgegeben werden. Das kann z.B. der Fall sein, wenn nach dem Löschen einer Teilnehmerinstanz die Kardinalitätsrestriktionen einer Beziehung nicht mehr erfüllt würden.

Beispiel 2.4: Löschen eines Kapitels

```
DELETE FROM chapter WHERE name = 'Kapitel 2';
```

Hier wird ersichtlich, dass die OrientSQL-Erweiterung sich auf einen großen Teil der SQL-Anweisungen auswirkt, auch wenn das syntaktisch nicht sichtbar ist. Diese Tatsache gilt auch für die anderen im Folgenden vorgestellten Spracherweiterungen, was eine Überlappung der verschiedenen Erweiterungen bedeutet. Der Umgang mit dieser Art von Überlappung ist ein wichtiges Kriterium für die Nützlichkeit von XSQL.

## 2.2 Mehrfachvererbung

In objektorientierten Sprachen spielt die Vererbung als Mechanismus zur Implementierung von Spezialisierungsbeziehungen und anderen Konzepten, z.B. Mixins [Vi98], eine wichtige Rolle. Es kann hier unterschieden werden zwischen der *Vererbung der Implementierung* und der *Vererbung der Schnittstellen* [Vi98]. Ein Typ, der von einem anderen Typ (dem *Supertyp*) die Schnittstelle erbt, kann an jeder Stelle im Code stehen, an der der Supertyp auch stehen kann. Die Vererbung der Implementierung dagegen ist prinzipiell nichts anderes als die Wiederverwendung von Implementierungsteilen. Der Begriff *Vererbung* (oder *Mehrfachvererbung*) ohne Zusätze steht meistens für die Kombination aus Schnittstellen- und Implementierungsvererbung; so wird er auch im Folgenden gebraucht. Neben dem Realisieren von Spezialisierungen findet der Vererbungsmechanismus auch andere Anwendungen (siehe Abschnitt *Pro und Kontra*).

Im Unterschied zur einfachen Vererbung kann bei der Mehrfachvererbung ein Typ von mehreren Supertypen erben. Java unterstützt nur die einfache Vererbung von Implementierungen, aber die Mehrfachvererbung von Schnittstellen.

Mit SQL:1999 wurde die Unterstützung der Objektorientierung in den SQL-Standard mit aufgenommen, und somit auch ein Mechanismus zur Vererbung von Methoden und Attributen eines Typs. SQL:1999 unterstützt jedoch nur die einfache Vererbung, auch ein Schnittstellenkonzept wie in Java gibt es nicht.

### Pro und Kontra

Die Mehrfachvererbung ist ein strittiges Thema, aufgrund der damit verbundenen Probleme. Eine ausführliche Diskussion über Vor- und Nachteile von Mehrfachvererbung macht Viega [Vi98]. In diesem Abschnitt werden einige wichtige Punkte daraus skizziert.

Beim Entwurf einer Anwendung oder eines Datenbankschemas kommt es des Öfteren vor, dass ein Typ eine Spezialisierung zweier verschiedener Typen ist: z.B. ist `InputOutputStream` eine Spezialisierung von `OutputStream` und `InputStream`. Eine solche Beziehung kann am besten mit Mehrfachvererbung ausgedrückt werden. Eine weitere Anwendung sind Mixins. Mixins sind Klassen, die eine bestimmte Funktionalität, die für viele andere Klassen nützlich ist, kapseln. Die Mixin-Funktionalität kann mit (Mehrfach)Vererbung in diese Klassen eingebaut werden.

Viega nennt auch Nachteile der Mehrfachvererbung. Zum einen können Namenskonflikte entstehen, wenn in zwei oder mehreren Supertypen Methoden mit gleicher Signatur existieren. Dieses Problem kann auf verschiedene Weise angegangen werden, mehr dazu in Abschnitt 2.2.1. Zum anderen kann ein Typ mehrfach Supertyp eines anderen sein. Hier ist die Frage, ob z.B. Attribute mehrfach oder nur einmal geerbt werden. Auch dieses Problem wird in Abschnitt 2.2.1 behandelt. Zuletzt kann die Mehrfachvererbung auch missbraucht werden, z. B. zur Abbildung von Aggregationen. Ein solcher Missbrauch kann nur durch Programmierdisziplin vermieden werden.

Trotz der möglichen Probleme ist die Mehrfachvererbung oft das Mittel der Wahl, um komplexe Beziehungen möglichst natürlich auf eine Programmiersprache abzubilden. In SQL gewinnt dies an Bedeutung, da hier normalerweise mit einer Wirtssprache zusammengearbeitet wird. Wenn die Wirtssprache Mehrfachvererbung unterstützt muss mit Workarounds gearbeitet werden, um Objekte zwischen Anwendung und Datenbank auszutauschen.

## 2.2.1 Beschreibung

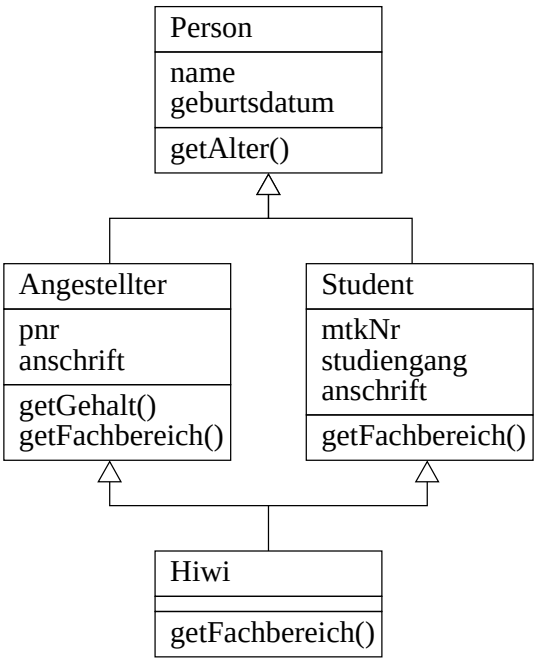
Eine Spracherweiterung für die Unterstützung der Mehrfachvererbung muss semantisch dem schon in SQL:1999 vorhandenen Vererbungsmechanismus entsprechen. Weiterhin sollte bei einfachen Vererbungsbeziehungen der eingebaute Mechanismus eingesetzt werden, da eine Simulation immer eine Performanzeinbuße bedeutet. Für die in Abschnitt 2.2 beschriebenen Probleme müssen Lösungen gefunden werden.

Zuerst wird die Mehrfachvererbung von Typen vorgestellt, danach wird kurz auf die Mehrfachvererbung bei Tabellen und Sichten eingegangen.

Abbildung 2.1 zeigt eine Vererbungshierarchie mit Mehrfachvererbung. In Beispiel 2.5 findet sich der dazugehörige SQL-Code.

Die Semantik der Mehrfachvererbung ist prinzipiell klar: der Subtyp enthält die Methoden und Attribute seiner Supertypen. In Ausdrücken und Anweisungen kann er an Stelle jedes Supertypen treten. Bei Konflikten muss jedoch eine Vorgehensweise zur Auflösung festgelegt werden, oder die Konflikte müssen durch die Sprache vermieden werden. Nachfolgend sind vier verschiedene Arten von Konflikten aufgeführt. Wie sie in dieser Arbeit behandelt werden ist ebenfalls erläutert.

Abbildung 2.1: Eine Vererbungshierarchie mit Mehrfachvererbung




Beispiel 2.5: Beispiel einer Vererbungshierarchie mit Mehrfachvererbung

```
CREATE TYPE Person AS (
    name VARCHAR(30),
    geburtsdatum DATE)
    NOT FINAL
    METHOD getAlter() RETURNS INTEGER;

CREATE TYPE Student UNDER Person AS (
    mtkNr INTEGER,
    studiengang VARCHAR(20),
    anschrift ADRESSE)
    NOT FINAL
    METHOD getFachbereich() RETURNS VARCHAR(20);

CREATE TYPE Angestellter UNDER Person AS (
    pnr INTEGER,
    anschrift ADRESSE)
    NOT FINAL
    METHOD getGehalt() RETURNS Euro,
    METHOD getFachbereich() RETURNS VARCHAR(20);

CREATE TYPE Hiwi UNDER Student, Angestellter
    NOT FINAL
    OVERRIDING METHOD getFachbereich() RETURNS VARCHAR(20);
```

**Mehrfaches Erben eines Attributs von unterschiedlichen Supertypen**
Ein Attribut wird über seinen Namen identifiziert. Falls mehrere Supertypen ein Attribut gleichen Namens haben, muss der Typ des Attributs bei allen Supertypen identisch sein. In Beispiel 2.5 erbt `Hiwi` das Attribut `anschrift` zwei mal. Das ist erlaubt, weil `anschrift` jeweils den gleichen Typ hat. `Hiwi` besitzt somit *ein* Attribut mit Namen `anschrift`. Wenn der Typ der gleichbenannten Attribute nicht identisch ist, kann kein Subtyp abgeleitet werden. Auch wenn ein Attribut mehrfach geerbt wird, kommt es im Subtyp nur einmal vor.

**Mehrfaches Erben eines Attributs vom gleichen Supertypen**
Das funktioniert analog zu dem Erben von Attributen von unterschiedlichen Supertypen. Im Beispiel sind das die Attribute `name` und `geburtsdatum`.

**Mehrfaches Erben einer Methode von unterschiedlichen Supertypen**
Methoden werden über ihre Signatur identifiziert. Die Signatur besteht aus dem Methodennamen und den Ein- und Ausgabeparametern. Methoden mit gleichem Namen, aber unterschiedlicher Signatur stehen somit nicht in Konflikt miteinander, da Methoden überladen werden können. Wenn jedoch mehrere Supertypen eine Methode mit identischer Signatur haben, *muss* diese vom Subtyp überschrieben werden.

Im Beispiel 2.5 gibt es in den Typen `Angestellter` und `Student` eine Methode `getFachbereich()`. Diese Methode liefert den Fachbereich, in dem der Angestellte arbeitet bzw. der Student eingeschrieben ist. Hier ist keine sinnvolle automatische Auflösung des Konflikts möglich, deshalb muss in einem solchen Fall der Entwickler des Subtyps die Methode überschreiben und entscheiden, welche Semantik sie im Subtyp haben soll.

**Mehrfaches Erben einer Methode vom gleichen Supertypen**
Wenn ein Subtyp die gleiche Methode mehrfach erbt, gibt es keinen Konflikt. Beispiel: `Hiwi` erbt `Person.getAlter()`[3] zweimal, weil `Person` Supertyp von `Student` *und* von `Angestellter` ist. Das ist kein Konflikt, weil es dieselbe Methode ist.

Die Mehrfachvererbung für Tabellen und Sichten basiert auf der für Typen, da Tabellen, um die Vererbung zu nutzen, getypt sein müssen. Bei Tabellen werden jedoch auch Constraints und Trigger geerbt, analog zu den Attributen bei Typen. Die Vererbung von Triggern gestaltet sich schwierig. Es muss darauf geachtet werden, dass kein Trigger doppelt geerbt wird, falls ein Typ mehrfach Supertyp ist. Weiterhin muss möglicherweise der ursprüngliche Trigger angepasst werden, um auch die Subtabelle mit zu erfassen.

## 2.2.2 Beispiele

### Anlegen eines Typs

Zugrunde liegt der Typ `Hiwi` aus Beispiel 2.5. Die anderen Typen in diesem Beispiel können ohne Mehrfachvererbung, d.h. mit SQL:1999-Mitteln, erzeugt werden.

[3] Zur Trennung von Typ- und Methodenbezeichner wird hier die Punkt-Notation von SQL:1999 benutzt.

Für die Übersetzung nach SQL:1999 muss entschieden werden, welcher Supertyp der SQL:1999-Supertyp ist und welcher simuliert wird. In diesem Beispiel ist `Student` der SQL:1999-Supertyp, das ergibt sich aus der Position in der Typdefinition: der erste nach `UNDER` genannte Supertyp ist der SQL:1999-Supertyp. Für den simulierten Supertyp `Angestellter` muss ein Cast vom Subtyp in diesen geschrieben werden, der SQL:1999-Modifikator `AS ASSIGNMENT` bewirkt, dass in Ausdrücken dieser Cast automatisch verwendet wird. Weiterhin müssen alle Methoden von `Angestellter` hinzugefügt werden. Der Methodenrumpf enthält jeweils den genannten Cast sowie einen Aufruf der originalen Methode. Ein kleiner Ausschnitt der übersetzten Anweisung findet sich in Beispiel 2.6.

Beispiel 2.6: Ausschnitt einer übersetzten XSQL-Anweisung

```
CREATE TYPE Hiwi UNDER Student AS (
    pnr INTEGER)
    NOT FINAL
    OVERRIDING METHOD getFachbereich() RETURNS VARCHAR(20);
CREATE CAST (Hiwi AS Angestellter)
    WITH SPECIFIC FUNCTION castFkt AS ASSIGNMENT;
CREATE FUNCTION castFkt (IN Hiwi)
    RETURNS Angestellter
    LANGUAGE SQL DETERMINISTIC
    BEGIN
        [...]
    END;
[...]
```

Das Informationsschema muss angepasst werden, z.B. muss ein
`SELECT * FROM INFORMATION_SCHEMA.DIRECT_SUPERTYPES`
`WHERE UDT_NAME = 'Hiwi'`
beide Supertypen liefern.

Wie auch bei den semantischen Beziehungen müssen alle intern von der Erweiterung genutzten Datenstrukturen (Cast-Funktionen, zusätzliche Methoden usw.) im Informationsschema versteckt werden.

### Anlegen einer Tabelle

In SQL:1999 muss die Vererbungshierarchie von Tabellen der Typhierarchie entsprechen, das ändert sich auch bei der Mehrfachvererbung nicht. Nur die letzte Anweisung in Beispiel 2.7 muss geändert werden, in dem die simulierte Supertabelle entfernt wird und stattdessen ein Test auf das Vorhandensein von `AngestellterT` angefügt wird.

Weiterhin müssen alle für `AngestellterT` definierten Constraints und Trigger auf `HiwiT` übertragen werden. In vielen Fällen kann das bedeuten, dass auch der ursprüngliche Trigger durch einen neuen ersetzt werden muss, da auch die simulierte Subtabelle in die Aktionen mit einbezogen werden muss. Wenn ein Trigger auf eine Tabelle zugreift, für die *nach* der Triggerdefinition eine Subtabelle definiert wird, kann er nicht automatisch auch auf die Subtabelle zugreifen (z.B. bei einem

Beispiel 2.7: Anlegen einer Tabellenhierarchie

```
CREATE TABLE PersonT OF TYPE Person;
CREATE TABLE StudentT OF TYPE Student UNDER PersonT;
CREATE TABLE AngestellterT OF TYPE Angestellter UNDER PersonT;
CREATE TABLE HiwiT OF TYPE Hiwi UNDER StudentT, AngestellterT;
```

SELECT auf die Supertabelle). Eine Lösung ist es, alle Trigger als Aufruf einer benutzerdefinierten Funktion zu implementieren. Diese Funktion kann dann mit dem X-Translate-System interagieren.

#### INSERT und Fremdschlüssel

Eine Anweisung wie `INSERT INTO HiwiT (name, ...) VALUES ('Hans', ...)` muss nicht geändert werden, in diesem Fall ist ein INSERT unproblematisch.

Anders sieht das jedoch beim Einfügen in die Tabelle von Beispiel 2.8[4] aus. Zur Verdeutlichung zeigt Abbildung 2.2 eine Übersicht über die Vererbungs- und Fremdschlüsselbeziehungen. Das INSERT wird nicht akzeptiert, da HiwiT keine SQL:1999-Subtabelle von AngestellterT ist. Hier muss die Erstellung der Fremdschlüsselbedingung durch eine adäquate selbst implementierte Lösung ersetzt werden.

Beispiel 2.8: Tabelle mit Fremdschlüsselreferenz auf simulierte Supertabelle

```
CREATE TABLE "hatSchlüssel" (
    "schlüsselNr" INTEGER,
    name VARCHAR(30) REFERENCES AngestellterT NOT NULL
);
INSERT INTO "hatSchlüssel" VALUES (1, 'Hans');
```

### Selektionen

In Selektionsanweisungen muss UNION verwendet werden, um alle Subtabellen einzubinden.

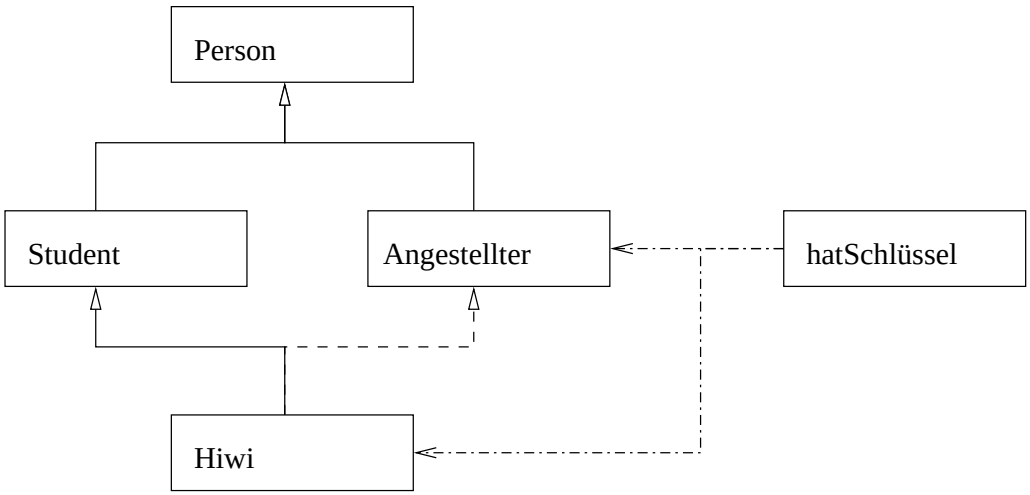
In Beispiel 2.9 sind eine XSQL-Anweisung und die dazu gehörige Übersetzung dargestellt.

### Löschen von Tupeln

Ein `DELETE FROM AngestellterT WHERE name='Hans'` muss auch in HiwiT alle korrespondierenden Einträge löschen. Eine Lösch-Anweisung wird in einem solchen Fall in eine Folge von entsprechenden Anweisungen umgewandelt.

[4] In Beispiel 2.8 werden die in SQL:1999 eingeführten *delimited identifiers* benutzt. So können alle Sonderzeichen in einem Bezeichner benutzt werden, im Beispiel das „ü“.

Abbildung 2.2: Fremdschlüsselreferenzen und Mehrfachvererbung




Beispiel 2.9: Übersetzung einer SELECT-Anweisung

```
SELECT * FROM AngestellterT;
```

wird ersetzt durch

```
SELECT * FROM AngestellterT
UNION CORRESPONDING BY (...)
SELECT * FROM HiwiT;
```

### 2.2.3 Anmerkungen

Die Mehrfachvererbung ist auf den ersten Blick eine triviale Erweiterung. Die Grammatik von SQL:1999 muss nur um wenige Symbole erweitert werden. Die vorgegangenen Beispiele zeigen jedoch, dass praktisch jede SQL-Anweisung von der Mehrfachvererbung betroffen ist. Nicht immer müssen die Anweisungen umgeschrieben werden, eine Prüfung ist jedoch in jeder Anweisung, die mit Typen oder Tabellen arbeitet, nötig. Das erhöht wiederum die Konfliktwahrscheinlichkeit mit anderen Spracherweiterungen. Beim Neuanlegen von Tabellen, die mehrere direkte Supertabellen haben, kann es nötig sein, schon vorhandene Elemente wie z.B. Trigger oder Fremdschlüsselreferenzen durch andere zu ersetzen.

## 2.3 Schemamodularität

Modularität ist eines der wichtigsten Paradigmen bei der Entwicklung von Softwaresystemen. Nur durch modularen Aufbau der Programme können größere Projekte bewältigt werden. Komponententechnologien und Frameworks für deren Einsatz, wie

Suns J2EE [Sun01a] oder Microsofts .NET [Ch02] gewinnen aufgrund dessen in den letzten Jahren immer mehr an Aufmerksamkeit. Komponenten sind voneinander unabhängige Softwaremodule, die parametrisiert und wie Bausteine zusammengesetzt werden können. Somit können Anwendungen bestehenden Code auf einfache Art und Weise wiederverwenden.

Auch bei der Entwicklung von Datenbankschemas wird eine solche modulare Vorgehensweise immer wichtiger, da die Komplexität von Datenbankschemas und Datenbankanwendungen nicht zuletzt durch die neuen Möglichkeiten, die SQL:1999 bietet, immer weiter ansteigt.

SQL:1999 bietet jedoch keine adäquate Unterstützung für die modulare Entwicklung von Datenbankschemas. Die einzige Möglichkeit, Schemaelemente logisch voneinander zu trennen, ist deren Unterbringung in verschiedenen SQL-*Schemas*. SQL-Schemas sind aber eher für administrative Zwecke geeignet als zur Komponentenentwicklung, da wichtige Konzepte wie z.B. Information Hiding mit Schemas nicht verwirklicht werden können.

Mahnke und Steiert [MS01a, MS01b, Ma02] beschreiben eine Spracherweiterung für SQL, die es ermöglicht, Datenbankschemas modular zu entwickeln. Es können verschiedene Arten von DB-Komponenten erstellt werden, die untereinander und mit Anwendungskomponenten interagieren. Eine Analyse der Konzepte für einen modularen Schemaentwurf macht Neumann [Ne02].

### 2.3.1 Beschreibung

Mit der Schemamodularität-Spracherweiterung können verschiedene Arten von Modulen und Schnittstellen definiert werden, abhängig von dem Verwendungszweck. Die Modultypen werden unter dem Oberbegriff *Schema-Module* zusammengefasst.

#### Schema-Module

Schema-Module müssen im Allgemeinen vor der Verwendung instanziiert werden. Jede Instanz hat einen eigenen Namensraum, über den die Elemente des Schema-Moduls angesprochen werden können.

##### Schema-Packages

Ein Schema-Package enthält ausschließlich benutzerdefinierte Typen (engl. user defined types, UDTs) und benutzerdefinierte Routinen (engl. user defined routines, UDRs) und hat keinen Zustand[5]. Aufgrund dieser Tatsache gibt es nur eine automatisch erstellte Instanz eines Schema-Package, die den Namen des Schema-Packages trägt.

##### Schema-Komponenten

Eine Schema-Komponente kann alle möglichen SQL-Schema-Elemente enthalten und mehrfach instanziiert werden. In der Regel werden für eine Anwendung eine oder sehr wenige Instanzen einer Komponente ausreichen.

[5]Neumann [Ne02] bespricht auch Schema-Packages mit Zustand, auf die aber hier nicht eingegangen wird.

Beispiele für Komponenten sind eine Benutzerverwaltung oder ein Katalogsystem. Eine Versandfirma kann in ihrem Internetauftritt beide Komponenten zusammen benutzen, um den Benutzern eine komfortable Einkaufsmöglichkeit zu bieten.

#### Schema-Frameworks

Schema-Frameworks können als Vorlagen für Schema-Komponenten angesehen werden: sie sind nicht selbst instanziierbar und somit nicht direkt nutzbar. Stattdessen werden sie von Schema-Komponenten vervollständigt. Die Basis-Infrastruktur für ein Web-Portalsystem kann als Schema-Framework implementiert werden. Um die Framework-Funktionalität zu nutzen, muss der Betreiber des Portals eine Schema-Komponente schreiben, die das Schema-Framework vervollständigt und z.B. die Anbindung an die betriebsinternen Datenbestände vornimmt.

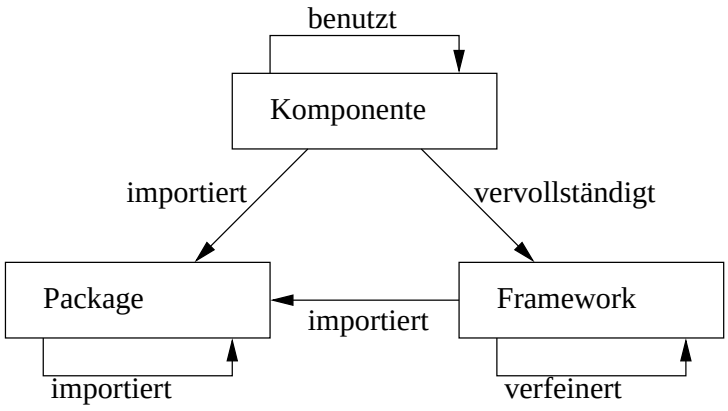
### Schnittstellen

Auf Schema-Module wird über Schnittstellen zugegriffen. Für ein Modul können mehrere Schnittstellen definiert werden, und eine Schnittstelle kann von mehreren Schema-Modulen implementiert werden. Bei der Instanziierung wird die zu benutzende Schnittstelle festgelegt. Die Spracherweiterung definiert zwei Arten von Schnittstellen: *APIs* (Application Programming Interfaces) und *Konnektoren* (engl. Connectors). APIs sind Schnittstellen für das Anwendungsprogramm, Konnektoren dienen zur Kommunikation zwischen Schema-Modulen. Diese Unterscheidung ist notwendig, da Schnittstellen für Anwendungen und für Module teilweise andere Möglichkeiten bieten, so können z.B. Fremdschlüssel auf eine Tabelle eines Moduls nur in Konnektoren definiert werden.

### Beziehungen zwischen Schema-Modulen

Beziehungen zwischen Schema-Modulen müssen explizit angegeben werden. Die Art der Beziehung wird bei der Modul-Definition festgelegt, die konkrete Beziehung aber erst bei der Instanziierung des Moduls. Abbildung 2.3 gibt einen Überblick über die möglichen Beziehungen zwischen Schema-Modulen.

Abbildung 2.3: Beziehungen zwischen Schema-Modulen




Die verschiedenen Typen von Schema-Modulen können unterschiedliche Beziehungen zueinander haben, wie in Abbildung 2.3 ersichtlich. Packages können von allen

drei Modularten *importiert* werden. Das importierende Modul kann dann die Definitionen des Packages nutzen. Frameworks können entweder von einem anderen Framework *verfeinert* oder von einer Komponente *komplettiert* werden. Auch ein verfeinerndes Framework kann nur über die angebotenen Schnittstellen auf das zu verfeinernde Framework zugreifen. Zuletzt können Komponenten von anderen Komponenten *benutzt* werden.

### 2.3.2 Beispiele

#### Definition einer Anwendungsschnittstelle

In Beispiel 2.10 ist ein Ausschnitt aus einer API-Definition zu sehen. Auffällig ist in Zeile 3 und 11 das Fehlen des `CREATE`-Schlüsselwortes, das in SQL zum Anlegen von Typen und Tabellen benutzt wird. Es handelt sich hier aber um eine Schnittstelle, und es werden keine Tabellen oder Typen angelegt. Um dies auch in der Syntax auszudrücken, wird kein `CREATE` verwendet.

Beispiel 2.10: Definition eines API

```
 1  DECLARE INTERFACE "UniversitätsInterface"
 2  BEGIN
 3      TYPE StudentTyp AS (
 4          mtkNr INTEGER,
 5          studiengang VARCHAR(30),
 6          ...)
 7          NOT FINAL
 8          METHOD getFachbereich() RETURNS VARCHAR(30),
 9          ...;
10
11      TABLE studentTable OF StudentTyp;
12
13      ...
14  END;
```

Bei der Übersetzung dieses Beispiels nach SQL:1999 wird kein Typ und keine Tabelle angelegt. Stattdessen wird die Schnittstellendefinition im Informationsschema aufgenommen. Wenn später über diese Schnittstelle auf eine Komponente zugegriffen wird, wird bei der Übersetzung der XSQL-Anweisung die Schnittstellendefinition verwendet, um sicher zu stellen, dass der Komponentenzugriff nur auf die in der Schnittstelle freigegebenen Elemente erfolgt.

#### Definition einer Komponente

Die in Beispiel 2.10 definierte Schnittstelle wird durch die in Beispiel 2.11 definierte Komponente implementiert (Zeile 3). Auch hier wird nicht das `CREATE`-Schlüsselwort benutzt, da bei der Komponentendefinition noch nicht die in der Komponente definierten Schemaelemente angelegt werden. Sie werden lediglich *deklariert*, was durch die Benutzung des `DECLARE`-Schlüsselwortes deutlich wird.

Beispiel 2.11: Anlegen und Instanziieren einer Komponente

```
 1  DECLARE COMPONENT "UniversitätsBeispiel"
 2  BEGIN
 3      IMPLEMENTS "UniversitätsInterface";
 4
 5      DECLARE TYPE StudentTyp AS (
 6          mtkNr INTEGER,
 7          studiengang VARCHAR(30),
 8          ...)
 9          NOT FINAL
10          METHOD getFachbereich() RETURNS VARCHAR(30),
11          METHOD updateStuff() RETURNS BOOLEAN,
12          ...;
13
14      DECLARE TABLE studentTable OF StudentTyp;
15
16      DECLARE METHOD getFachbereich()
17          FOR StudentTyp
18          BEGIN
19              -- Bestimmung des Fachbereich
20          END;
21  END;
22
23  CREATE COMPONENT Uni OF "UniversitätsBeispiel";
```

Ähnlich wie bei der Schnittstellendefinition wird die Komponentendefinition in das Informationsschema aufgenommen, ohne tatsächlich Typen und Tabellen anzulegen.

In Zeile 23 des Beispiels 2.11 wird die Schema-Komponente *instanziiert*. Im Namensraum `Uni` werden alle Tabellen und Typen, die definiert wurden, erzeugt. Die Umsetzung des Konzepts der Namensräume auf SQL:1999 kann durch ein Umschreiben der Elementnamen erfolgen: Der Name `StudentTyp` wird zu `"__UNI::STUDENTTYP"`. Das Umschreiben muss vom X-Translate-System verdeckt werden.

Kommentar 2.1:

SQL:1999 unterscheidet bei normalen Bezeichnern nicht zwischen Groß- und Kleinschreibung, `StudentTyp` und `STUDENTTYP` referenzieren den gleichen Typen. Bei Bezeichnern in Anführungsstrichen wird aber zwischen Groß- und Kleinschreibung unterschieden. Beim Umschreiben von normalen Bezeichnern in Bezeichner mit Anführungsstrichen müssen alle Buchstaben groß geschrieben werden. Beim hier vorgeschlagenen Umschreiben müssen die Bezeichner in Anführungsstriche geschrieben werden, da auch Sonderzeichen (::) benutzt werden.

### Benutzen einer Komponente

Die von einer Komponente bereitgestellten Typen und Tabellen können nach der Instanziierung der Komponente wie normale Schemaelemente genutzt werden, mit

der Einschränkung, dass lediglich die in der Schnittstelle erlaubten Operationen ausgeführt werden dürfen. In unserem Beispiel 2.10 sind keine Einschränkungen spezifiziert, deshalb ist das Erzeugen eines Subtyps in Beispiel 2.12 eine legale Operation.

Der einzige Unterschied zu einer SQL:1999-konformen Anweisung ist die Voranstellung des Namensraums bei der Benutzung eines Elements einer Komponente (Zeile 1).

Beispiel 2.12: Benutzung einer Komponente

```
1 CREATE TYPE HiwiTyp UNDER Uni::StudentTyp AS (
2     Aufgabe VARCHAR(30),
3     Arbeitszeit INTEGER)
4     NOT FINAL;
```

Bei der Übersetzung nach SQL:1999 muss lediglich `Uni::StudentTyp` durch `"__UNI::STUDENTTYP"` ersetzt werden, sowie die Schnittstelle befragt werden, ob die Operation erlaubt ist.

## 2.4 Weitere Spracherweiterungen

In der Literatur finden sich neben den drei vorgestellten Beispielen noch viele weitere Vorschläge für SQL-Erweiterungen:

- Kollektionsunterstützung: In SQL:1999 gibt es als einzigen Kollektionstyp einen nicht schachtelbaren `ARRAY`-Typ. Lufter [Lu02] schlägt die Einführung verschiedener Kollektionstypen wie `LIST`, `SET` und `MULTISET` vor. Auch die Schachtelung von Kollektionstypen ist vorgesehen.

- Preference-SQL: Kießling und Köstler [KK02] stellen eine Erweiterung von SQL um unscharfe Anfragen vor. Beispiele:

  ```
  SELECT * FROM trips PREFERRING duration AROUND 14;
  SELECT * FROM apartments PREFERRING HIGHEST(area);
  ```

  Preference-SQL ist für den Einsatz in Suchmaschinen gedacht. Dort sind unscharfe Anfragen sinnvoll.

- Versionierung: Auch die Versionsverwaltung von Objekten ist eine denkbare Spracherweiterung für SQL:1999; zur Zeit muss eine Versionierung durch die Anwendung oder durch Middleware erfolgen.

- Herstellerspezifische Erweiterungen: Der SQL:1999-Standard wird zur Zeit von keinem Hersteller vollständig implementiert. Dies wird sich vermutlich auch in Zukunft nicht ändern. Stattdessen bieten die Hersteller jeweils eigene SQL-Dialekte an, die sich vom Standard in zwei Dingen unterscheiden. Zum einen werden Teile des Standards nicht implementiert, und zum anderen werden herstellerspezifische Erweiterungen eingebaut.

Hierdurch wird es sehr schwierig, portable SQL-Schemas und Anfragen zu schreiben. Wenn man die Funktionen eines objekt-relationalen DBVS (ORDBVS) voll ausnutzen will, ist man oft auf herstellerspezifischen Spracherweiterungen angewiesen. Ein Wechsel auf ein anderes DBVS wird dadurch sehr kostenintensiv.

Diese Problematik kann mit XSQL elegant gelöst werden, in dem die Differenz eines SQL-Dialekts zu SQL:1999 in eine XSQL-Spracherweiterung gefasst wird.

In diesem Zusammenhang treten drei weitere Probleme auf:

- Herstellerspezifische Erweiterungen sollten auf dem nativen System *nicht* nach SQL:1999 transformiert werden, sondern müssen durchgereicht werden.
- Die Abbildung XSQL → SQL:1999 reicht in der Praxis nicht aus, da bei allen zur Zeit existierenden ORDBVS der SQL:1999-Standard nicht vollständig implementiert ist. Das resultierende SQL:1999 muss also in der Regel auf den Herstellerdialekt des eingesetzten DBVS abgebildet werden.
- Viele Datenbanksysteme haben auch Erweiterungen, die nicht auf Sprachkonstrukte des SQL:1999-Standards abgebildet werden können. Darunter fallen unter anderem Befehle zur Indexgenerierung oder Optimierer-Direktiven. Hier kann XSQL zum Ausfiltern dieser Erweiterungen genutzt werden, um zumindest die Lauffähigkeit auf anderen Systemen zu gewährleisten. Bei vielen Anwendungen wird dies jedoch zu Performanzeinbrüchen führen.

## 2.5 Anmerkungen

Die drei ausführlicher beschriebenen Beispiele sind sehr stark zentriert auf die Daten-Definitionssprache (engl. data definition language, DDL). Da durch Änderungen an den SQL-Datentypen viele Anweisungen betroffen sind, ist die Implementierung dieser Beispiele sehr komplex. Spracherweiterungen, die keine neuen Datenstrukturen einführen bzw. schon vorhandene ändern, sind weitaus einfacher zu handhaben. Bei Preference-SQL [KK02] sind lediglich die `SELECT`-Anweisungen betroffen. Die `PREFERRING`-Subklausel muss in eine passende `WHERE`-Subklausel übersetzt werden. Generell kann der Aufwand für das Entwickeln einer Spracherweiterung sehr stark schwanken, das hängt unter anderem von der DDL-Zentrierung der Erweiterung ab.

# Kapitel 3 Anforderungen und Probleme

An den Beispielen in Kapitel 2 wurden bereits einige Anforderungen an das X-Translate-System deutlich. In Abschnitt 3.1 werden alle Anforderungen detailliert beschrieben. Abschnitt 3.2 beschreibt die allgemeine Struktur und die Bestandteile von X-Translate anhand der Anforderungen. Diese Anforderungen beschreiben ein optimales System; einige Punkte sind in der Praxis nicht realisierbar. Das wird an den betreffenden Stellen diskutiert.

## 3.1 Anforderungskatalog

### 3.1.1 Benutzergruppen

**B1** Es lassen sich drei Gruppen von Systembenutzern identifizieren. *Entwickler* bezeichnet im Folgenden die Personen, die eine Spracherweiterung entwickeln. Mit *Benutzer* sind alle Personen, die erweiterte SQL-Anweisungen nutzen, gemeint. Ein *Administrator* ist für die Installation von Spracherweiterungen in die Laufzeitumgebung von X-Translate zuständig.

### 3.1.2 Funktionale Anforderungen

Die funktionalen Anforderungen sind getrennt in Basisanforderungen und erweiterte Anforderungen. Die Basisanforderungen beschreiben die allgemeine Systemfunktionalität. Ein System zur Erweiterung von SQL:1999 hat im Moment keine praktische Relevanz, da kein Hersteller den SQL:1999-Standard vollständig implementiert. Auch in Zukunft ist zu erwarten, dass die meisten Hersteller nicht alle SQL-Features anbieten. Die erweiterten Anforderungen tragen dem Rechnung und gehen auf die Behandlung von SQL-Dialekten der Hersteller ein.

**F1** Mit dem System sollen beliebige Spracherweiterungen für SQL:1999 entwickelt werden können. Das schließt (a) die Erweiterung der Sprachsyntax und (b) des Metadatenmodells ein, sowie (c) die Spezifikation der Abbildung der Erweiterungen auf SQL:1999. Zur Abbildung gehört auch die Definition von (d) erweiterungsspezifischen Fehler- und Statusmeldungen. Die Sprache, die durch eine Spracherweiterung entsteht, wird mit *XSQL* bezeichnet. XSQL ist somit ein Sammelbegriff für alle möglichen Erweiterungen von SQL:1999. Die Spezifikation einer Spracherweiterung wird *Sprachmodul* genannt.

**F2** Es gibt ein Laufzeitsystem, das vor ein DBVS geschaltet werden kann und XSQL-Anweisungen auf SQL:1999-Anweisungen abbildet, ausführt und die

Ergebnisse an den Benutzer zurückliefert. Dazu muss vorher ein Sprachmodul in das Laufzeitsystem geladen werden. Als Ergebnis zählt auch jede Art von Fehler- und Statusmeldung.

**F3** Es sollen gleichzeitig mehrere unabhängig voneinander entwickelte Sprachmodule in das Laufzeitsystem geladen und benutzt werden können.

**F4** Eine einzelne XSQL-Anweisung kann aus verschiedenen Erweiterungen zusammengesetzt sein. Das kann sich in (a) syntaktischer Hinsicht bemerkbar machen, oder (b) in semantischer Hinsicht.

**F5** Die Spezifikation von Spracherweiterungen muss so einfach wie möglich gestaltet werden. Dem Entwickler müssen passende Entwicklungswerkzeuge (z.B. Editor und Debugger) zur Verfügung gestellt werden.

**F6** Konflikte in (a) Syntax, (b) Semantik, (c) Metadaten und (d) Abbildungen sind bei unabhängig entwickelten Sprachmodulen möglich. Diese Konflikte sollen möglichst automatisch erkannt werden.

**F7** Konflikte sollen, soweit möglich, automatisch aufgelöst werden.

**F8** Wenn eine automatische Konfliktauflösung nicht möglich ist, kann der Administrator eingreifen und die Sprachmodule an den Konfliktstellen manuell zusammenführen.

**F9** Falls ein Konflikt nicht gelöst werden kann, muss das Laufzeitsystem den gleichzeitigen Einsatz der Spracherweiterungen verbieten.

**F10** Sprachmodule sollen im laufenden Betrieb in das Laufzeitsystem geladen werden können und dann sofort benutzbar sein.

**F11** Durch das Laden eines neuen Sprachmoduls darf sich die Bedeutung bereits existierender Sprachkonstrukte von SQL:1999 nicht verändern. Eine beliebige SQL:1999-Anweisung muss sich immer gleich verhalten.

**F12** Sprachmodule dürfen sich auch gegenseitig nicht beeinflussen. Das Ausführen von Anweisungen einer Erweiterung darf nicht von einer anderen Erweiterung abhängen.

Erweiterte Anforderungen:

**H1** Das System muss auf einfache Art und Weise an SQL-Dialekte von Datenbankherstellern angepasst werden können. Dass heißt, das die Abbildung von XSQL nicht mehr wie in **F1** und **F2** beschrieben auf SQL:1999, sondern auf den jeweiligen Dialekt erfolgt. Die Anpassungen werden *Herstelleranpassungen* genannt und wie Sprachmodule ins Laufzeitsystem geladen.

**H2** SQL-Dialekte können als Sprachmodul, das die Differenz zu SQL:1999 spezifiziert, in das System geladen werden. Diese Sprachmodule werden Herstellermodule genannt. Herstellermodule dürfen die Anforderung **F10** nicht verletzen.

**H3** Herstellermodule sollen in zwei Modi arbeiten: *Originalmodus* und *Emulationsmodus*. Im Originalmodus transformieren sie nicht, sondern reichen die herstellerspezifischen SQL-Anweisungen nach unten durch. Im Emulationsmodus arbeiten sie exakt wie andere Sprachmodule und transformieren die herstellerspezifischen Erweiterungen auf SQL:1999.

**H4** Wenn ein SQL:1999-Sprachkonstrukt von einer Herstelleranpassung nicht verarbeitet werden kann, muss eine Fehlermeldung ausgegeben werden.

### 3.1.3 Andere Anforderungen

#### Schnittstellen

**S1** Es sollen die gleichen Schnittstellen wie bei DBVS üblich angeboten werden. Das schließt JDBC- [Sun01b], ODBC-Treiber [Ge95] und ähnliche Programmierschnittstellen, sowie die von den Herstellern angebotenen Werkzeuge, ein.

#### Performanz

**P1** Die Performanz der Abarbeitung von SQL-Anweisungen darf nicht wesentlich beeinträchtigt werden.

#### Rahmen dieser Arbeit

**D1** Das in dieser Arbeit beschriebene System beschränkt sich auf die Bereitstellung eines JDBC-Treibers als Schnittstelle zum Laufzeitsystem. Der Treiber kann dann vor einen beliebigen, herstellerspezifischen JDBC-Treiber geschaltet werden. Andere DB-Schnittstellen wie z.B. ODBC oder herstellerspezifische Werkzeuge werden nicht besprochen, somit kann Anforderung **S1** von dem in dieser Arbeit vorgestellten System nicht erfüllt werden.

**D2** Die kompilierten Sprachmodule bestehen aus einer Reihe von Java-Funktionen, die in der Datenbank ausgeführt werden. Damit bricht X-Translate mit dem 1999 veröffentlichten SQL:1999-Standard, in dem das Ausführen von Java-Funktionen in der DB nicht spezifiziert ist. Mittlerweile existiert jedoch eine Zusatzspezifikation zur Integration von Java in die Datenbank [ISO02], die allgemein SQL:1999 zugerechnet wird.

**D3** In dieser Arbeit wird hauptsächlich die Entwicklung von Spracherweiterungen und deren Abbildung auf SQL:1999 behandelt. Die Behandlung von Fehlern bei der Ausführung von XSQL-Anweisungen wird aufgrund der Komplexität nicht behandelt.

**D4** Es ist denkbar, das ein Sprachmodul ein anderes benutzt, um sich selbst zu implementieren. Diese Möglichkeit wird in dieser Arbeit nicht besprochen, da das System dadurch wesentlich komplexer würde.

## 3.2 Systembeschreibung

X-Translate kann anhand der Anforderungen in drei Teilsysteme zerlegt werden.

- Eine *Laufzeitumgebung*, in die Sprachmodule geladen werden können und die als Schnittstelle zwischen Benutzer und DBVS steht.
- Eine *Installationsumgebung* mit Werkzeugen zur Installation von Sprachmodulen und für die Konfliktbehandlung.
- Eine *Entwicklungsumgebung* zum Entwickeln von Sprachmodulen.

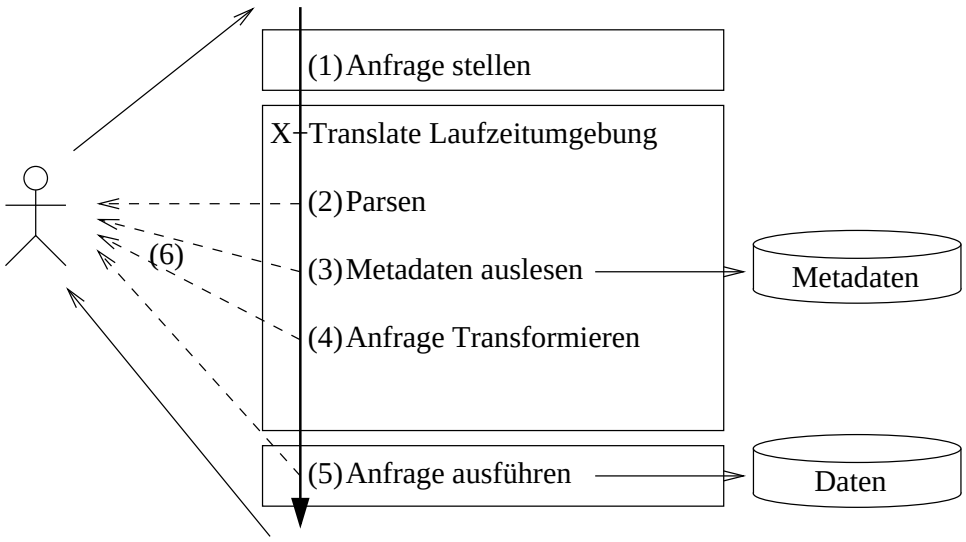
Die Systemaufteilung in Laufzeitumgebung, Installation und Konfliktauflösung und Entwicklungsumgebung entspricht der Benutzergruppenaufteilung in Benutzer, Administrator und Entwickler in Anforderung **B1**.

### 3.2.1 Die Laufzeitumgebung

#### Stellen von Anfragen

Die Laufzeitumgebung deckt die Anforderungen **F2**-**F4**, **F10**, **H3** und **H4** ab. In Abbildung 3.1 ist der Weg einer SQL-Anweisung vom Benutzer bis zum DBVS schematisch dargestellt. Die vom Benutzer gestellte XSQL-Anfrage (1) muss zuerst

Abbildung 3.1: Weg einer erweiterten SQL-Anfrage durch das X-Translate-System.




geparst werden (2). Danach kann das Laufzeitsystem auf Metadaten zugreifen, um alle zur Transformation nötigen Informationen zu erhalten (3). Nach der Transformation (4) wird die Anfrage dann im DBVS ausgeführt (5). Bei (2) bis (5) kann durch auftretende Fehler die weitere Bearbeitung abgebrochen werden, was durch die mit (6) markierten gestrichelten Pfeile angedeutet ist. Besondere Beachtung verdienen die Fehler, die in (5) auftreten. X-Translate muss hier möglicherweise einen Statuscode des DBVS in einen XSQL-Statuscode übersetzen. Die Fehlerbehandlung wird in dieser Arbeit nicht diskutiert (Entscheidung **D3**).

Das Laufzeitsystem ist vergleichbar mit einem Interpreter oder einem Compiler für eine Programmiersprache. Die erweiterten SQL-Anweisungen werden zwar kompiliert, jedoch kann bereits bei der Kompilierung auf die Metadaten zugegriffen werden, was eher der Arbeitsweise eines Interpreters entspricht.

#### Metadatenmodell

SQL:1999 [ISO99b] definiert ein Metadatenmodell für in der Datenbank gespeicherte Daten als eine Reihe von Sichten im Informationsschema (`INFORMATION_SCHEMA`). Basis dieser Sichten bildet das Definitionsschema (`DEFINITION_SCHEMA`). Das Definitionsschema existiert nur konzeptionell und muss von Herstellern nicht implementiert werden; lediglich das Informationsschema dient als Quelle von Metadaten.

SQL:1999 sieht keine benutzerdefinierte Erweiterung des Informationsschemas vor. Das X-Translate-System muss daher ein erweitertes Informationsschema bereitstellen, welches einerseits die SQL:1999-Informationen aus dem Informationsschema enthält und andererseits alle Erweiterungen, die in einem erweiterten Definitionsschema gespeichert sind. Für den Benutzer ist das transparent, Zugriffe auf das Informationsschema werden an das erweiterte Informationsschema weitergeleitet.

### 3.2.2 Installation von Sprachmodulen und Konfliktauflösung

In den Anforderungen **F3** und **F6-F9** werden die Aufgaben der administrativen Komponente von X-Translate beschrieben. Außerdem ergeben sich indirekt weitere Aufgaben bzw. Anforderungen, die im Folgenden in einer groben Reihenfolge der Abarbeitung beschrieben sind.

1. Automatische Konflikterkennung: Hier versucht das X-Translate-System, möglichst viele der Konflikte zwischen dem neuen und den bereits installieren Sprachmodulen zu erkennen. Auf die verschiedenen Arten von Konflikten wird in Abschnitt 3.3 eingegangen.

2. Manuelle Konflikterkennung: Es ist nicht zu erwarten, dass das X-Translate-System alle möglichen Konflikte zwischen zwei Spracherweiterungen automatisch erkennen kann. Deshalb muss der Administrator vor der Installation eines neuen Sprachmoduls alle zusammen eingesetzten Module einer manuellen Prüfung unterziehen, um mögliche Konflikte aufzudecken.

3. Automatische Konfliktauflösung: Nach dem Identifizieren der Konflikte versucht das System, diese automatisch aufzulösen.

4. Manuelle Konfliktauflösung: Wie schon bei der Erkennung von Konflikten, werden auch bei deren Auflösung manuelle Eingriffe nötig. Für jeden Konfliktfall hat der Administrator mehrere Optionen: Er kann den Konflikt als Scheinkonflikt markieren, falls die automatische Erkennung falsch gearbeitet hat. Weiterhin kann er die konfliktären Sprachmodule an den betreffenden Stellen manuell aneinander anpassen. Zuletzt gibt es die Möglichkeit, Funktionalität eines Moduls zu verwerfen oder die Syntax zu ändern. Nach der Änderung eines Sprachmoduls ist die Kompatibilität zur Original-Erweiterung möglicherweise nicht mehr gegeben.

5. Anlegen der Verwaltungsdaten und Laden des Sprachmoduls: Nachdem alle Konflikte beseitigt sind, können die Verwaltungsdaten angelegt werden und das Sprachmodul in das Laufzeitsystem geladen werden.

6. Erweiterung des Metadatenmodells: Als letzter Installationsschritt müssen die zur Erweiterung gehörigen Metamodell-Erweiterungen installiert werden.

7. Migration der DB-Schemas: Da eine Spracherweiterung sich auch auf schon existierende Schemaobjekte auswirken kann, müssen möglicherweise alle Schemaobjekte geprüft und angepasst werden.

8. Aktivieren der Spracherweiterung: Nach dem Abschluss der Installation wird das Sprachmodul aktiviert, so dass die Erweiterungen vom Benutzer verwendet werden können.

### 3.2.3 Die Entwicklungsumgebung

Relevant für die Entwicklungsumgebung sind die Anforderungen **F1**, **F11**, **H1** und **H3**. Gemäß dieser Anforderungen besteht ein Sprachmodul aus mehreren logischen Teilen: Erweiterungen für die Grammatik von SQL:1999, Erweiterungen des Metadatenmodells und Transformationsregeln. Ein Aspekt, der in allen Teilen betrachtet werden muss, ist die Behandlung von XSQL-Fehler- und Statusmeldungen. Diese drei logischen Teile werden in den Folgenden Abschnitten beschrieben.

#### Grammatikerweiterungen

Der SQL:1999-Standard [ISO99b] spezifiziert die Sprache mit einer Reihe von Grammatik-Produktionen in einer BNF-Notation. Ein Spracherweiterungs-Modul führt meistens neue Produktionen in die Grammatik ein oder fügt zu schon vorhandenen Produktionen neue Alternativen hinzu. Die Spezifikation der Grammatik-Erweiterungen geschieht in dieser BNF-Form. Neben den Produktionen muss die Spezifikation der Grammatik-Erweiterung möglicherweise auch Hinweise und Meldungen für den Parser enthalten, um sinnvolle Meldungen bei Parse-Fehlern auszugeben. Dies wird in dieser Arbeit jedoch nicht weiter behandelt (Entscheidung **D3**).

#### Erweiterung des Metadatenmodells

Da Spracherweiterungen generell keine existierende Funktionalität zerstören (Anforderung **F11**), beschränkt sich die Erweiterung des Metadatenmodells auf das Hinzufügen von Spalten und Sichten im erweiterten Informationsschema. Zu bemerken ist, dass sich die Anforderung **F11** nicht vollständig erfüllen lässt: Ein `SELECT * FROM INFORMATION_SCHEMA.<SICHT>` verhält sich nach dem Hinzufügen einer Spalte anders, was dazu führen kann, dass Anwendungen ungültige Anweisungen an die Datenbank schicken.

Es lassen sich zwei Fälle der Erweiterung unterscheiden.

- Durch die Einführung neuer Datenstrukturen oder das Ändern bereits vorhandener ändern sich die Metadaten. Es müssen neue Sichten oder bei bestehenden Sichten neue Spalten hinzugefügt werden.

- Zur Abbildung der Spracherweiterung auf SQL:1999 werden zusätzliche Schemaobjekte angelegt. Darunter fallen auch benutzerdefinierte Funktionen, in denen Sprachfunktionalität gekapselt wird. Diese nur zum internen Gebrauch vorgesehenen Objekte müssen für den Benutzer unsichtbar sein.

### Spezifikation der Abbildung auf SQL:1999

Ein Sprachmodul enthält für jedes von ihr betroffene Sprachelement eine oder mehrere Transformationsregeln, die XSQL-Anweisungen in SQL:1999 übersetzen. Der Parser liefert einen XSQL-Syntaxbaum, welcher von den Transformationsregeln in einen SQL:1999-Syntaxbaum übersetzt wird. Die Transformationsregeln werden so spezifiziert, dass sie so lokal wie möglich arbeiten. Das heißt, dass eine Transformation so wenig Knoten im Syntaxbaum wie möglich ändert.

### Fehler- und Statusmeldungen

Fehler- bzw. Statusmeldungen entstehen an verschiedenen Stellen im Ablauf der Bearbeitung einer Anfrage. Abbildung 3.1 verdeutlicht dies.

1. Syntaxfehler fallen beim Parsen auf. Schon bei dem Erweitern der Grammatik müssen möglicherweise spezielle Meldungen mit angegeben werden.

2. Semantikfehler werden nach dem Parsen entdeckt. Semantikfehler sind z.B. viele der im SQL:1999-Standard beschriebenen Regeln (engl. rules).

3. Metadatenbedingte Fehler werden, wie die Semantikfehler, bei der Transformation der SQL-Anweisung entdeckt, da zu diesem Zeitpunkt auf die Metadaten zugegriffen wird. Ein Beispiel hierfür ist das Ansprechen einer nicht vorhandenen Tabelle.

4. Datenbedingte Fehler werden erst beim Ausführen der transformierten Anfrage erkannt, wie z.B. nicht erfüllte referentielle Bedingungen.

Typ eins kann in der erweiterten Grammatik beschrieben werden. Typ zwei und drei können in der Regelsprache spezifiziert werden. Problematisch wird Typ vier. Da dieser erst nach dem Ausführen einer Anweisung entdeckt wird, müssen möglicherweise SQL:1999-Fehlermeldungen in erweiterte Fehlermeldungen transformiert werden.

Die Spezifikation von Fehlermeldungen wird in dieser Arbeit nicht behandelt (Entscheidung **D3**).

## 3.3 Realisierbarkeit und Grenzen

Wenn Spracherweiterungen unabhängig voneinander entwickelt und dann zusammen eingesetzt werden, sind Konflikte unausweichlich. Auch wenn X-Translate Möglichkeiten bietet, Konflikte zu vermeiden oder aufzulösen, wird es immer Fälle geben, in denen zwei Spracherweiterungen nicht zusammenarbeiten können.

In den Folgenden Abschnitten werden Punkte diskutiert, die entweder die Anwendbarkeit von X-Translate einschränken oder mit denen manche Einschränkungen umgangen werden können.

Konflikte sind oft vermeidbar, wenn Spracherweiterungen sorgfältig geplant und implementiert werden. Gute Spracherweiterungen zeichnen sich nicht nur durch die angebotenen Sprachabstraktionen, sondern auch durch die Qualität der Implementierung aus.

Kommentar 3.1:

Die *Qualität* eines Sprachmoduls ist schwer messbar; ein naives Maß dafür könnte beispielsweise der Prozentsatz der Sprachmodule sein, mit denen das gemessene Sprachmodul ohne Probleme zusammenarbeitet. Hierbei wird jedoch unter anderem außer acht gelassen, das Spracherweiterungen unterschiedlich komplex sind, und dass nicht jede Kombination von Spracherweiterungen sinnvoll ist.

### 3.3.1 Konflikte in der Grammatik

Wenn zwei Erweiterungen an der gleichen Stelle die Syntax von SQL:1999 erweitern, entsteht eine Mehrdeutigkeit. Ein solcher Konflikt ist unwahrscheinlich, wenn sich die Erweiterungen nicht auch semantisch überlappen (siehe Abschnitt 3.3.4). Gelöst werden kann dieses Problem durch den Verzicht auf eine der beiden Syntax-Erweiterungen. Möglicherweise kann auch die Syntax einer Erweiterung abgewandelt werden. Im ersten Fall verliert man einen Teil der Funktionalität einer Spracherweiterung, die zweite Lösung geht auf Kosten der Portierbarkeit.

### 3.3.2 Konflikte im Metadatenmodell

Ebenso wie in der Grammatik können auch im Metadatenmodell Konflikte auftreten. Das ist der Fall, wenn zwei Spracherweiterungen die gleiche Tabelle oder die gleiche Spalte zum Originalmodell hinzufügen. Hier ist eine gleichzeitige Nutzung der konfliktären Erweiterungen nur möglich, wenn die mehrfach bereitgestellten Tabellen und Spalten exakt die gleiche Bedeutung haben. Das kann nur vom Administrator entschieden werden.

### 3.3.3 Konflikte bei der Abbildung

Jeder Knoten des Quellbaumes darf nur von einer Erweiterung transformiert werden. Wenn mehrere Erweiterungen den gleichen Knoten transformieren, dürfen sie nicht gleichzeitig installiert sein. Im besten Fall kann ein Administrator mit ausreichenden Entwicklerkenntnissen und ausreichenden Kenntnissen von beiden Spracherweiterungen die problematischen Transformationen manuell zusammenführen. Damit werden die Spracherweiterungen abhängig voneinander.

### 3.3.4 Überlappende Semantik

Das Vorhandensein der gleichen oder ähnlicher Semantik in mehreren Spracherweiterungen führt nicht zwangsläufig zu Konflikten, das passiert erst, wenn auch die

Syntax gleich ist. Wenn aber die gleiche Semantik auf völlig unterschiedliche Weise implementiert werden kann, führt das zur Konfusion der Benutzer.

### 3.3.5 Performanz

Die Performanz einer Kombination von Sprachmodulen lässt sich nur schwer voraussagen. Die Übersetzung von Anweisungen im X-Translate-System benötigt mehr Zeit als bei einem DBVS ohne X-Translate, da ein Übersetzungsschritt hinzukommt. Wenn insgesamt mehr Transformationsregeln im System geladen sind, steigt der Zeitaufwand im Durchschnitt, da pro Anweisungen mehr Transformationen abgearbeitet werden müssen. Weiterhin sind im Allgemeinen Zugriffe auf die Metadaten notwendig, was unter Umständen erhebliche Performanz-Verluste aufgrund von Plattenzugriffen zur Folge haben kann.

Für die Ausführungszeit von XSQL-Anweisungen lassen sich ebensowenig wie die für die Übersetzungszeit allgemeine Regeln finden. Eine XSQL-Anweisung wird oft in mehrere SQL:1999-Anweisungen übersetzt, dadurch dauert die Ausführung möglicherweise wesentlich länger. Insbesondere beim Einsatz von mehreren Sprachmodulen hängt die Ausführungszeit sehr davon ab, inwieweit die transformierten Anweisungen vom DBVS-Optimierer optimiert werden können. Da Sprachmodule unabhängig voneinander entwickelt werden, ist im X-Translate-System eine modulübergreifende Optimierung von Anweisungen höchstens in Einzelfällen möglich.

### 3.3.6 Daten- und Operationszentrierte Erweiterungen

Ein starkes Kriterium für die Entwicklung von Spracherweiterungen und die Konfliktwahrscheinlichkeit zwischen ihnen ist ihre Zentrierung auf Daten oder auf Operationen.

- Datenzentrierte Erweiterungen ändern bzw. erweitern meistens die schon vorhandenen Datenstrukturen. Die drei Hauptbeispiele aus Kapitel 2 sind datenzentriert. Das hat zur Folge, dass sich praktisch alle SQL-Anweisungen ändern. Das Konfliktpotential beim gleichzeitigen Einsatz mehrerer datenzentrierter Erweiterungen ist sehr hoch.
- Im Gegensatz dazu stehen die operationszentrierten Spracherweiterungen. Operationszentrierte Sprachmodule stellen hauptsächlich Erweiterungen bereit, die keine Datenstrukturen ändern, sondern nur die möglichen Operationen auf diesen erweitern. Ein Beispiel hierfür ist Preference-SQL [KK02]. Operationszentrierte Erweiterungen sind weniger komplex und wirken sich meist nur auf wenige SQL-Anweisungen aus. Damit sinkt die Wahrscheinlichkeit von Konflikten mit anderen Spracherweiterungen erheblich.

## 3.4 Zusammenfassung

In diesem Kapitel wurde ein Anforderungskatalog zusammen gestellt, den ein System zur Entwicklung und Benutzung von Spracherweiterungen für SQL:1999 erfüllen

sollte. Besonders wichtig ist die Unabhängigkeit der Sprachmodule voneinander. Um ein fehlerfreies Arbeiten mit mehreren Sprachmodulen zu gewährleisten, muss auf die Konflikte, die zwischen zwei Sprachmodulen auftreten können, geachtet werden. Auch wenn ein Konflikt auftritt, kann er möglicherweise durch das System oder durch einen manuellen Eingriff aufgelöst werden.

In dieser Arbeit wird kein vollständig den Anforderungen entsprechendes System vorgestellt, das würde den gegebenen Rahmen sprengen. Einige Aspekte, wie beispielsweise die Rückgabe von Fehler- und Statusmeldungen, werden ausgelassen.

# Kapitel 4 Das X-Translate-System

In diesem Kapitel wird ein Ansatz zur Implementierung des X-Translate-Systems vorgestellt. In Abschnitt 4.1 werden Konzepte und Begriffe erläutert, die in diesem Kapitel verwendet werden. Abschnitt 4.2 gibt einen Überblick über die Architektur des X-Translate-Systems. Die Abschnitte 4.3 bis 4.6 beschreiben detailliert die Systembestandteile. Der Transformationssprache ist aufgrund des Umfangs ein eigener Abschnitt (4.4) gewidmet.

## 4.1 Konzepte

### 4.1.1 Grammatik

Das X-Translate-System hat eine vorgegebene Grammatik, die von Sprachmodulen erweitert werden kann. Diese *Basisgrammatik* entspricht der in SQL:1999 [ISO99b] vorgegebenen. Grammatikalische Produktionen werden mit der im Standard [ISO99a] definierten BNF-Variante notiert.

Mit der BNF (Backus-Naur Form) werden allgemein kontextfreie Grammatiken spezifiziert. Es gibt *Terminalsymbole*, *Nonterminalsymbole* und *Produktionen*. Eine Produktion ist eine Regel, die spezifiziert, wie ein Nonterminalsymbol durch null oder mehrere Symbole, Terminale sowie Nonterminale, ersetzt werden kann. Angefangen von einem *Startsymbol* wird solange ersetzt, bis lediglich Terminalsymbole übrigbleiben. Die Menge aller Terminalsymbolreihen, die so erzeugt werden können, bildet die von der Grammatik spezifizierte Sprache. Ein Element der Sprache wird als *Wort* bezeichnet.

### 4.1.2 Syntaxbäume

Im Compilerbau wird im Allgemeinen zwischen Parse-Bäumen und abstrakten Syntaxbäumen (engl. abstract syntax trees, ASTs) unterschieden [ASU86]. Parse-Bäume enthalten vom Startsymbol an alle Produktionen, die nötig sind, um das entsprechende Wort der Sprache zu konstruieren. ASTs hingegen stellen eine Vereinfachung der Parse-Bäume dar, in der viele Produktionen weggelassen werden. Ein AST hat im Allgemeinen weniger Knoten und eine geringere Höhe als der Parse-Baum, aus dem er entstanden ist.

Aufgrund der weniger komplexen Struktur von ASTs sind diese für Compiler einfacher zu handhaben. Die Syntaxbäume in X-Translate sind jedoch Parse-Bäume und keine ASTs. Das hat den Grund, das in ASTs lediglich die für den Compiler notwendigen Produktionen und Informationen vorhanden sind. Was für den Compiler notwendig ist und wie ein Parse-Baum auf einen AST abgebildet wird, kann bei

einer nicht erweiterbaren Sprache im vorhinein festgestellt werden, nicht jedoch bei XSQL. Nur die Sprachmodule selbst können wissen, welche Teile des Parse-Baums sie brauchen. Deshalb müssten die Sprachmodule außer der Erweiterung der Grammatik auch die Abbildung auf einen AST definieren. Dadurch können neue Konflikte mit anderen Sprachmodulen entstehen. Weiterhin arbeiten Transformationen auf einzelnen Baumknoten, und da ein Parse-Baum weitaus differenzierter ist, sinkt die Konfliktwahrscheinlichkeit für Transformationen dort. Aus diesen Gründen haben wir uns dafür entschieden, als Eingabe für den X-Translate-Compiler keine ASTs, sondern Parse-Bäume zu benutzen. Statt dem Begriff *Parse-Baum* wird hier *Syntaxbaum* benutzt. Es gibt zwei Ausnahmen: Die Nonterminalsymbole <literal> und <identifier> aus der SQL:1999-Grammatik werden vom Parser direkt aufgelöst und haben nur einen Kindknoten mit dem entsprechenden Wert.

Jeder Knoten in einem Syntaxbaum entspricht einer grammatikalischen Produktion, deshalb werden die Begriffe *Produktion* und *Knoten* synonym benutzt. Analog dazu entspricht der Typ eines Knotens immer einem Nonterminal- oder Terminalsymbol in der Grammatik. Auch die Begriffe *Knotentyp* und *(Non)terminalsymbol* werden deshalb synonym benutzt.

Als Quell-Syntaxbaum oder *Quellbaum* wird im Folgenden der durch das Parsen der vom Benutzer abgesandten XSQL-Anweisung entstandene Syntaxbaum bezeichnet. Im Gegensatz dazu ist der Ziel-Syntaxbaum oder *Zielbaum* derjenige, der transformiert wird. Wenn die Transformation fertig ist, kann der Ziel-Syntaxbaum durch einfache Aneinanderreihung der Blattknoten in eine SQL:1999-Anweisung umgewandelt werden.

### 4.1.3 Knotenmuster

Derselbe Knotentyp kann, je nach dem, wo er im Syntaxbaum steht, unterschiedlich behandelt werden. In Beispiel 4.1 wird das deutlich: In einer `CREATE TYPE`-Anweisung wird vorausgesetzt, dass der Typname noch nicht existiert (a). Beim Anlegen einer getypten Tabelle hingegen muss der referenzierte Typ schon existieren (b). Eine Spracherweiterung muss bei solchen Knotentypen eindeutig feststellen können, welche Möglichkeit zutrifft, und danach ihre Aktionen ausrichten.

Um solche Unterscheidungen zu ermöglichen, werden Knoten anhand von *Knotenmustern* (engl. node patterns) identifiziert. Ein Knotenmuster besteht aus zwei Teilen: dem Subjekt und dem Kontext. Das Subjekt ist der Knotentyp, und der Kontext beschreibt die Umgebung, in der das Subjekt liegt. Durch den Vergleich eines Knotenmusters mit dem Syntaxbaum werden die Knoten gefunden, die in dem entsprechenden Kontext liegen. Das kann unter Umständen mehr als ein Knoten sein.

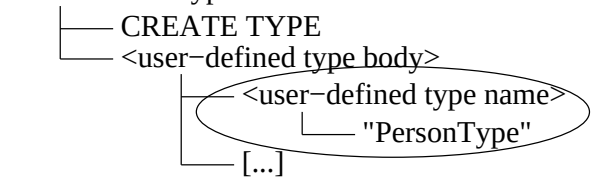
Abbildung 4.1 zeigt einen unvollständigen Syntaxbaum, der beim Einsatz des Sprachmoduls *Mehrfachvererbung* entstehen kann. Mit Fettschrift hervorgehoben ist das Subjekt des Knotenmusters. Der Kontext besteht aus den kursiv geschriebenen Elementen und den dick gedruckten Linien. Auf dieses Knotenmuster passen nur <user-defined type definition>-Produktionen, die mindestens zwei Supertypen haben und nicht als `FINAL` markiert sind. Diese müssen vom Modul Mehrfachvererbung besonders behandelt werden.

Beispiel 4.1: Unterschiedliche Behandlung von gleichen Knotentypen (vereinfachte Syntax)

```
-- Bsp. (a)
CREATE TYPE PersonTyp ( ... );

-- Bsp. (b)
CREATE TABLE PersonTable AS PersonTyp;
```

**Bsp. (a)**




**Bsp. (b)**

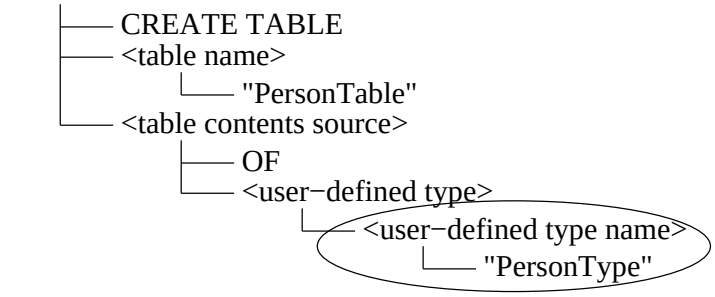
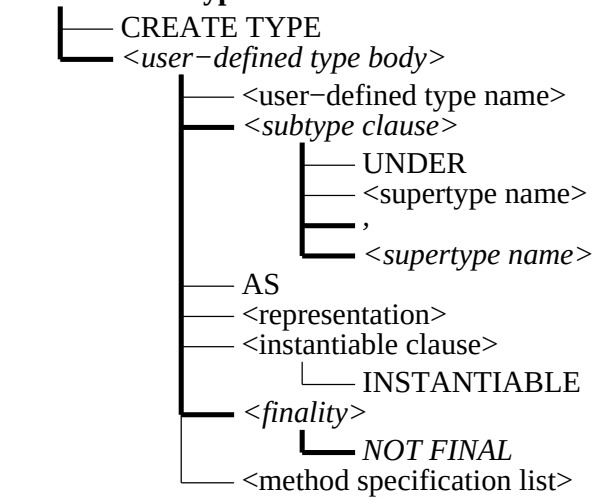



Abbildung 4.1: Ein Knotenmuster einer Transformation des Sprachmoduls Mehrfachvererbung



#### Entscheidbarkeit der Überlappung von Knotenmustern

Eine wichtige Frage ist die nach der eindeutigen Bestimmbarkeit des passenden Knotenmusters für einen Knoten. Das ist bei überlappenden Knotenmustern nicht immer möglich. Es ist deshalb wichtig, sich überlappende oder sich einschließende Knotenmuster bereits bei der Installation von Sprachmodulen zu identifizieren, und nicht erst bei der Ausführung von SQL-Anweisungen.

### 4.1.4 Transformationen

Um eine Spracherweiterung auf SQL:1999 abzubilden, werden von dem Sprachmodul bereitgestellte Transformationsregeln, oder einfach *Transformationen*, benutzt. Sie erzeugen einen SQL:1999-Syntaxbaum (dem *Zielbaum* aus einem XSQL-Syntaxbaum (dem *Quellbaum*).

Mit einer Transformation ist immer ein Knotenmuster assoziiert. Die Transformation wird immer auf Knoten, die dem Subjekt des Knotenmusters entsprechen, angewandt. Während der Transformation heißt dieser Subjektknoten *aktueller Knoten*.

Die Transformationen werden prinzipiell gleichzeitig angewendet. Die Reihenfolge spielt keine Rolle, weil jede Transformation den unveränderten Syntaxbaum als Eingabe bekommt. Es wird also nicht der Quellbaum direkt transformiert, sondern aus dem Quellbaum wird ein neuer Baum generiert. Weiterhin wird auf den Quellbaum nicht direkt, sondern über *Methoden* der *Klassen* (siehe Abschnitt 4.1.5) zugegriffen.

Eine Transformation kann im Teilbaum unter dem aktuellen Knoten die Reihenfolge von Knoten verändern, neue Knoten einfügen und Knoten löschen oder ändern. Weiterhin greifen Transformationen auf Metadaten zu. Nicht jede Transformation muss tatsächlich einen Knoten strukturell verändern. Auch wenn lediglich Metadaten gelesen oder geschrieben werden, ist das eine Transformation. Transformationen werden detailliert in Abschnitt 4.3 besprochen.

### 4.1.5 Objektorientierung

Die verschiedenen Elemente der Transformationssprache können auch von einem objektorientierten Standpunkt aus gesehen und erklärt werden. Für die Implementierung der Transformationen spielt diese Sichtweise nur eine untergeordnete Rolle, da die Klassen, Objekte und Methoden nicht direkt auf die entsprechenden Elemente einer objektorientierten Programmiersprache abgebildet werden können (siehe Abschnitt 4.6).

**Klassen und Objekte** Klassen werden durch Knotenmuster definiert. Die Objekte einer Klasse sind alle Knoten, die im passenden Kontext liegen und deren Typ dem Subjekt des Knotenmusters entspricht.

Der Typ eines Knotens bestimmt somit noch nicht dessen Klasse. Je nach dem, an welcher Stelle im Syntaxbaum ein Knoten auftaucht, kann er zu verschiedenen Klassen gehören. Es kann auch sein, das ein Knoten gleichzeitig verschiedenen Klassen angehört.

**Attribute und Methoden** Das X-Translate-Äquivalent zu Attributen von Objekten sind die Kindknoten und der Elternknoten eines Knotens. Objekte einer Klasse müssen nicht immer die gleichen Attribute haben: pro Alternative, die in der Grammatik als Produktion zugelassen ist, kann ein Knoten unterschiedliche Kindknoten besitzen.

Der Zugriff auf Knoten erfolgt über Methoden. Die Attribute des Knotens sind nicht direkt sichtbar. Sprachmodule müssen Methoden bereitstellen, die die Änderungen des Sprachmoduls nach außen hin verdecken. So nehmen zwei unterschiedliche Sprachmodule nichts voneinander wahr, da der Quellbaum immer als SQL:1999-Syntaxbaum erscheint.

Zur Signatur einer Methode gehört unter anderem auch der *Aufrufer*. Der Aufrufer ist immer die aktuelle Transformation. Ein Knotenmuster legt fest, von wo aus eine Methode aufgerufen werden darf, d. h., in welchem Knotenmuster der aktuelle Knoten liegen muss. In der Regel passt das Aufrufs-Knotenmuster auf alle Knoten, so dass nicht zwischen unterschiedlichen Aufrufern unterschieden wird.

Methoden haben immer den Namen des Knotens, für den sie aufgerufen werden, und liefern eine Sicht auf diesen.

**Polymorphie** Wenn mehrere Methoden für einen Knoten aufgerufen werden können, entscheidet das Aufrufs-Knotenmuster und andere Faktoren (siehe Abschnitt 4.4.4) darüber, welche davon aufgerufen wird. Auch die Klasse eines Objektes ist polymorph, wie schon im Absatz *Klassen und Objekte* erwähnt. Czarnecki und Eisenecker [CE00] nennen dies im Zusammenhang mit IP *polymorph über Knotenmustern im Quell-Graphen*[1].

Die Verwendung von Aufrufs-Knotenmustern macht Sinn, um verschiedene Spracherweiterungen aneinander anzupassen. Dadurch entsteht eine Abhängigkeit zwischen diesen Erweiterungen, die sich aber nur auswirkt, wenn die Erweiterungen tatsächlich zusammen eingesetzt werden.

### 4.1.6 Datenmodell

Als Mittel der Einteilung und Abgrenzung von Schemaelementen dienen in SQL:1999 die Schemas. SQL:1999 definiert zwei spezielle Schemas, mit denen die Metadaten verwaltet werden. X-Translate definiert drei weitere. Die folgende Liste beschreibt die speziellen Schemas und deren Inhalte.

`DEFINITION_SCHEMA` (Abk. DS): Das in SQL:1999 beschriebene Definitionsschema. Hier befinden sich Tabellen mit Metadaten zu den Inhalten aller Schemas. Das DS ist lediglich eine Hilfskonstruktion zur Definition der Sichten im Informationsschema. Ein direkter Zugriff ist nicht möglich.

`INFORMATION_SCHEMA` (Abk. IS): Das in SQL:1999 beschriebene Informationsschema. Es enthält eine Reihe von Sichten auf das Definitionsschema. Die Sichten können jedoch nicht verändert bzw. gelöscht werden.

[1] IP benutzt statt Syntaxbäumen Graphen mit baumartigen und graphenartigen Kanten [CE00].

`DEFINITION_SCHEMA_EXTENDED` (Abk. DSX): Das erweiterte Definitionsschema. Da X-Translate weitere Metadaten verwalten muss, wird das DSX als Ergänzung zum DS bereitgestellt. Alle zusätzlichen Metadaten, die von den Sprachmodulen erzeugt werden, sind hier gespeichert.

Da das DS (und somit auch das IS) Zugriff auf die Metadaten *aller* Schemas und Schemaobjekte bietet, ist es nötig, die Metadaten für die nur intern von X-Translate genutzten Daten zu maskieren. Hierzu gibt es im DSX spezielle Tabellen, in denen alle zu maskierenden Metadaten aufgelistet sind. Maskierte Metadaten sind für den Benutzer nicht sichtbar.

`INFORMATION_SCHEMA_EXTENDED` (Abk. ISX): Das erweiterte Informationsschema. Es tritt an Stelle des vom DBVS bereitgestellten Informationsschemas. In den Sichten des ISX werden die Metadaten im IS mit den Metadaten im DSX vereinigt, unter Auslassung der maskierten Metadaten.

`XT_SYSTEM_SCHEMA` (Abk. XSS): Das Verwaltungsschema mit den Systemdaten. Hier liegt der Programmcode für Parser und Compiler. Die installierten Sprachmodul-Archive werden für Verwaltungszwecke auch gespeichert.

Daten sind in der Regel in mehreren Ebenen strukturiert, wobei die Daten auf Ebene $n+1$ die Daten auf Ebene $n$ beschreiben. Die Daten auf Ebene $n+1$ sind die *Metadaten* für die Daten auf Ebene $n$. Tabelle 4.1 verdeutlicht die Datenaufteilung in X-Translate. Die vertikale Aufteilung ist angelehnt an die Meta Object Facility (MOF) [OMG02].

Tabelle 4.1: Datenmodell des X-Translate-Systems

| **Ebene nach MOF** | **SQL:1999** | **X-Translate-Benutzerdaten** | **X-Translate-Systemdaten** |
|---|---|---|---|
| 3. Meta-Metamodell | – | Metamodell-Sprache | – |
| 2. Metamodell | IS (DS) | ISX (IS, DS, DSX) | IS |
| 1. Modell | IS-Daten | ISX-Daten | IS-Daten |
| 0. Daten | DB-Daten | DB-Daten | XSS- und IS-Daten |

Die erste Spalte enthält die Bezeichnungen in der MOF-Spezifikation. In der zweiten Spalte steht die Aufteilung, wie sie in SQL:1999 vorgenommen ist. In den letzten beiden Spalten wird die Einteilung der X-Translate-Daten verdeutlicht.

Spalte drei teilt die Benutzerdaten auf verschiedene Ebenen auf. Das ISX ersetzt das von der Datenbank bereitgestellte IS. Der Benutzer kann es weiterhin unter dem Namen `INFORMATION_SCHEMA` ansprechen. Sprachmodule können das IS mit der auf Ebene vier angesiedelten Erweiterungssprache erweitern.

Auf der Metamodell-Ebene wird für SQL:1999 und das X-Translate-System neben dem IS(X) auch das DS(X) genannt. Die Definitionsschemas sind zwar für den Benutzer nicht sichtbar, bilden aber die Basis für die Sichten in den Informationsschemas. Das SQL:1999-IS ist in X-Translate unsichtbar, bildet aber die Basis für das ISX.

Die vierte Spalte gibt einen Überblick über die Systemdaten des X-Translate-Systems. Diese Spalte wurde nur der Vollständigkeit halber in die Tabelle aufgenommen; sinnvoller ist hier eine Einteilung in Programmcode und Programmdaten. Der Programmcode findet sich im IS in Form von benutzerdefinierten Funktionen. Die Programmdaten werden im XSS gespeichert. Auch das IS gehört zu den Programmdaten, da Transformationen darauf zugreifen können. ISX und DSX wurden nicht in der vierten Spalte erwähnt, da sie nicht zum Betrieb des X-Translate-Systems benötigt werden.

### 4.1.7 Sprachmodule

Als Sprachmodul wird die Gesamtheit der zur Implementierung einer Spracherweiterung verwendeten Dateien und Programme bezeichnet. Die Entwicklungsumgebung stellt Werkzeuge bereit, um Sprachmodule in einfach zu installierende Archive zu packen.

### 4.1.8 X-Translate und herstellerspezifische SQL-Dialekte

Um einen praktischen Einsatz von X-Translate zu ermöglichen, muss der Tatsache Rechnung getragen werden, das zur Zeit kein Hersteller den SQL:1999-Standard voll unterstützt. X-Translate stellt dafür zwei Mittel zur Verfügung.

#### Herstellermodule

*Herstellermodule* sind spezielle Sprachmodule. Sie bieten die herstellerspezifischen Erweiterungen von SQL:1999 als Spracherweiterung an. Wird ein Herstellermodul auf einem anderen als dem originären DBVS eingesetzt, verhält es sich wie ein normales Sprachmodul und übersetzt die Erweiterungen nach SQL:1999. Beim Einsatz auf dem korrespondierenden DBVS jedoch reicht es die Erweiterungen durch, d. h. der Quellbaum wird nicht nach SQL:1999, sondern nach SQL:1999 mit Herstellererweiterungen transformiert. So können beispielsweise Befehle zur Indexerstellung weiter genutzt werden. Ohne Herstellermodule könnten solche Befehle nicht ausgeführt werden, was die Performanz auch bei relativ kleinen Datenbeständen erheblich beeinflussen kann.

So können zwar Herstellererweiterungen nutzbar gemacht werden, es ist jedoch nicht immer möglich, den SQL-Dialekt eines Herstellers hundertprozentig zu simulieren, falls Semantik oder Syntax im Widerspruch zu SQL:1999 stehen.

#### Herstelleranpassungen

Um die von einem DBVS-Hersteller nicht unterstützten SQL:1999-Merkmale auf den Herstellerdialekt abzubilden, werden *Herstelleranpassungen* eingesetzt. Es kann immer nur eine Herstelleranpassung installiert sein: eine, die für das eingesetzte System geschrieben wurde. Die Übersetzung des Quellbaumes verläuft in zwei Phasen: Zuerst wird XSQL nach SQL:1999 (eventuell mit Herstellererweiterungen) übersetzt. In einem zweiten Schritt wird das Ergebnis dann von der Herstelleranpassung in den Herstellerdialekt übersetzt. Herstelleranpassungen funktionieren ähnlich wie Sprachmodule. Sie sind aber einfacher zu handhaben, weil es hier keine Konflikte geben kann.

Kommentar 4.1:

In dieser Arbeit werden in einigen internen X-Translate-Anweisungen SQL:1999-Merkmale, die nicht von allen DBVS unterstützt werden, genutzt. Das geschieht vor allem deshalb, weil SQL:1999 eine elegantere und verständlichere Lösung für einige Probleme bietet. Zum praktischen Einsatz müssen diese Anweisungen umgeschrieben werden, um X-Translate auf möglichst vielen DBVS einsetzbar zu machen.

### 4.1.9 Fehler- und Statusmeldungen

Aus Gründen der Komplexität wird hier auf die genaue Besprechung der Behandlung von Fehlermeldungen verzichtet. In einem produktiven System ist es jedoch wichtig, dem Benutzer im Fehlerfall Informationen zu geben, die das Problem genau beschreiben. Nur so kann die Fehlerursache schnell und effizient behoben werden.

## 4.2 Systemarchitektur

Das komplette X-Translate-System besteht aus einer Anzahl verschiedener Programme und Sprachen. In diesem Abschnitt werden alle Komponenten vorgestellt und in Beziehung zueinander gesetzt. Der in Kapitel 3 gemachten Dreiteilung in Entwicklungs-, Installations- und Laufzeitumgebung wird gefolgt.

X-Translate arbeitet auf der Basis von Syntaxbaum-*Transformationen*. XSQL-Anweisungen werden von einem *Parser* eingelesen und daraus ein Syntaxbaum erstellt. Der Syntaxbaum wird dann von einem *Compiler* Knoten für Knoten abgearbeitet und es werden jeweils die Transformationen angewandt, für die ein passendes *Knotenmuster* existiert. Durch die Anwendung der Transformationen entsteht ein neuer Syntaxbaum, der SQL:1999-konform ist. Er wird in eine Zeichenkette umgewandelt und diese wird an das darunter liegende DBVS gegeben.

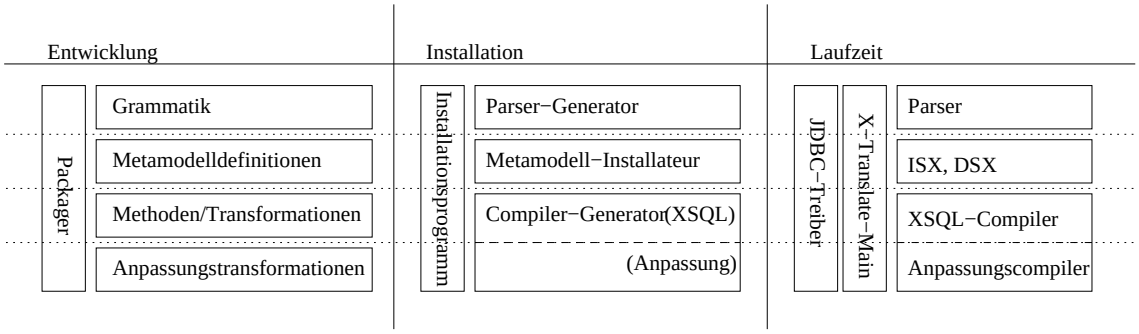
Abbildung 4.2 gibt einen schematischen Überblick über die verschiedenen Systemkomponenten. Die Trennung in Entwicklungs- und Installationskomponenten ist nicht fest; die Installationskomponenten werden auch schon bei der Entwicklung zu Debuggingzwecken eingesetzt. Die vertikale Trennung der Komponenten verdeutlicht die verschiedenen Aspekte von Sprachmodulen. Der Packager generiert aus Grammatik, Metamodelldefinitionen, Methoden, Transformationen und Anpassungstransformationen ein Installationsarchiv. Dieses wird vom Installationsprogramm installiert. Bei jeder Installation eines Sprachmoduls werden der Parser und die Compiler vom jeweiligen Generator neu generiert und das Metamodell erweitert.

Der JDBC-Treiber dient als Schnittstelle zwischen dem Benutzer und X-Translate-Main. X-Translate-Main ist verantwortlich für die Koordination der verschiedenen Übersetzungsschritte.

### 4.2.1 Komponenten der Entwicklungsumgebung

Der Entwicklungsumgebung zugeordnet sind die Methoden und Sprachen, die zur Spezifikation der Grammatik- und Metamodell-Erweiterungen, der Transformationen und dem Maskieren der Systemdaten benutzt werden. Die Trennung zwischen

Abbildung 4.2: Komponenten von X-Translate




Entwicklungs-, Installations- und Laufzeitumgebung ist nicht fest eingegrenzt, so werden die der Installations- bzw. Laufzeitumgebung zugerechneten Programme sinnvollerweise auch zur Entwicklung eines Sprachmoduls eingesetzt.

**Grammatik-Erweiterungen** Da die Abwärtskompatibilität immer gewährleistet sein muss, gibt es lediglich zwei Möglichkeiten, die SQL:1999-Grammatik zu erweitern: das Hinzufügen neuer Alternativen zu bereits vorhandenen Produktionen und das Hinzufügen neuer Terminal- und Nonterminalsymbole. Eine separate Sprache zur Erweiterung der Grammatik wird somit nicht benötigt; es genügt, die Erweiterungen als Produktionen in der von X-Translate genutzten BNF-Variante zu schreiben. Der Parser-Generator überprüft unter anderem, dass keine Zweideutigkeiten gibt.

**Metamodell-Sprache** Neben der Grammatik kann auch das Metamodell von SQL:1999 erweitert werden. Das Metamodell entspricht den Definitionen der Sichten im IS bzw ISX. Auch hier kommt wegen der Abwärtskompatibilität nur das Hinzufügen von neuen Sichten oder das Erweitern von bereits vorhandenen Sichten um neue Spalten in Frage. Neben dem ISX kann auch auf dem DSX gearbeitet werden.

**Transformationssprache** Kern des X-Translate-Systems ist die Abbildung von XSQL auf SQL:1999. Die Entwicklungsumgebung stellt dafür die Transformationssprache zur Verfügung. Mit ihr kann man Methoden und Transformationen definieren.

Methoden haben lesenden Zugriff auf den Quellbaum. Transformationen können durch Methodenaufrufe auf den Quellbaum zugreifen. Sie haben auch Lese- und Schreibzugriff auf Benutzer- und Systemdaten in der Datenbank.

Der Zielbaum wird von den Transformationen aus dem Quellbaum generiert. Eine Transformation kann den aktuellen Knoten und dessen Kindknoten verändern.

Transformationen können auch Status- und Fehlermeldungen spezifizieren, die an den Benutzer zurückgegeben werden.

In Herstellermodulen können Transformationen, je nach dem, ob sie auf dem eigenen oder auf einem fremden DBVS eingesetzt werden, unterschiedliche Ausgaben produzieren. Im Regelfall bedeutet das, dass nur auf fremden Systemen die Erweiterung nach SQL:1999 transformiert wird. Auf dem eigenen System wird die Erweiterung durchgereicht, da sie vom DBVS verstanden wird.

**Anpassungs-Sprache** In einem zweiten Schritt wird der durch die Transformationen erhaltene SQL:1999-Syntaxbaum auf den SQL-Dialekt des Datenbankherstellers abgebildet. Das funktioniert ähnlich wie der erste Transformationsschritt. Der Zugriff auf den Quellbaum kann jedoch direkt erfolgen, Methoden werden nicht benötigt.

Mit *Transformation* ist in dieser Arbeit immer eine in der Transformationssprache definierte Transformation gemeint. Eine in der Anpassungssprache definierte Transformation heißt *Anpassungs-Transformation*.

Auch hier können Status- und Fehlermeldungen spezifiziert werden.

**Modul-Packager** Der Modul-Packager ist ein Werkzeug, das dem Entwickler hilft, ein Sprachmodul in ein installierbares Archiv zu packen. Es prüft die Abhängigkeiten zwischen den verschiedenen Sprachmodul-Teilen, beispielsweise ob alle von den Transformationen genutzten Produktionen auch in der erweiterten Grammatik definiert sind. Wenn alle Prüfungen erfolgreich verlaufen, werden die Dateien in ein Archiv gepackt, welches dann zur Installation im Laufzeitsystem verwendet werden kann.

### 4.2.2 Komponenten der Installationsumgebung

Die Installationsumgebung besteht aus einem Installationsprogramm, das die Schnittstelle zum Administrator bildet. Weiterhin gehören Compiler und andere Komponenten zur Installation der verschiedenen Bestandteile dazu.

**Installationsprogramm** Das Installationsprogramm dient als Schnittstelle zwischen Administrator und den anderen Komponenten. Wenn Probleme bei der Installation auftauchen, werden sie angezeigt, und der Administrator kann darauf reagieren, in dem er Änderungen vornimmt oder Entscheidungen trifft, die das X-Translate-System nicht selbst treffen kann. Das Installationsprogramm startet den Parser-Generator und den Compiler-Generator, und lädt die fertigen Programme in die Datenbank.

**Parser-Generator** Der Parser-Generator fügt alle Grammatikerweiterungen zusammen und generiert daraus einen Parser. Er hat auch die Aufgabe, Konflikte und Inkonsistenzen in der Grammatik anzuzeigen.

**Compiler-Generator** Der Compiler-Generator generiert aus den Methoden und Transformationen den XSQL-Compiler und den Anpassungscompiler für das Laufzeitsystem. Konflikte werden, soweit möglich, angezeigt.

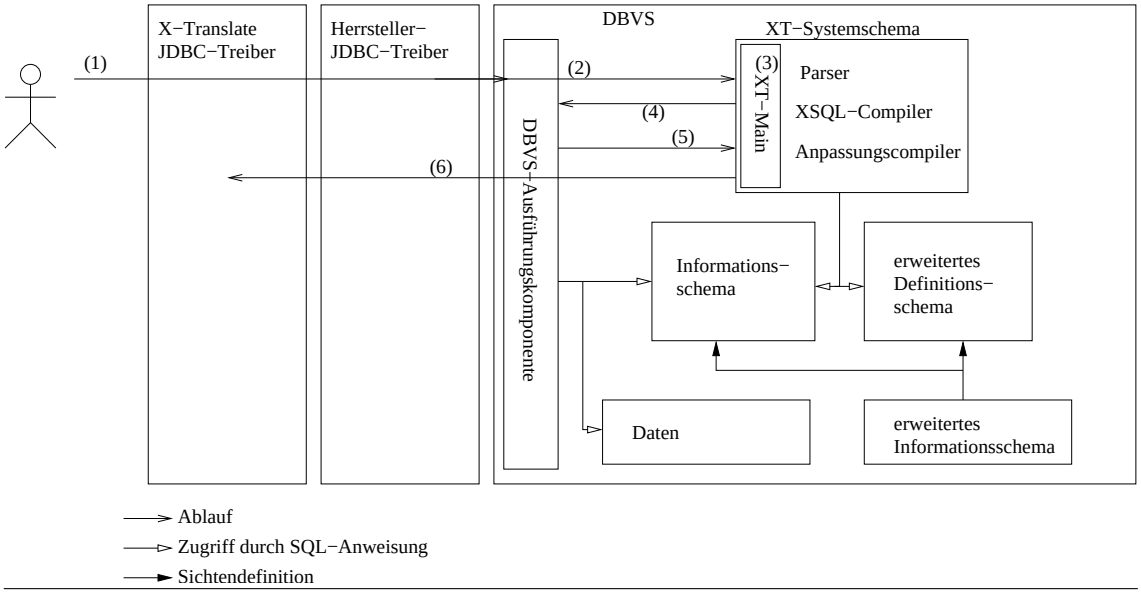
**Metamodell-Installateur** Der Metamodell-Installateur ist für das Erweitern des ISX und DSX anhand der Metamodell-Definitionen verantwortlich.

### 4.2.3 Komponenten der Laufzeitumgebung

Die Laufzeitumgebung setzt sich zusammen aus einem JDBC-Treiber und Java-Klassen in der Datenbank. Zusätzlich werden im XSS alle benötigten Verwaltungsdaten gespeichert.

Abbildung 4.3 gibt einen Überblick über die Funktionsweise des Gesamtsystems. Die einzelnen Komponenten sowie der Weg einer SQL-Anfrage durch das System werden in diesem Abschnitt erläutert.

Abbildung 4.3: Das X-Translate-Laufzeitsystem: Komponenten und Arbeitsweise




### Komponenten

**JDBC-Treiber** Es gibt zwei JDBC-Treiber. Zum einen der X-Translate-JDBC-Treiber, und zum anderen einen speziellen für das jeweilige DBVS.

Der X-Translate-Treiber dient lediglich als Wrapper, um SQL-Anweisungen vom Benutzer in Aufrufe des X-Translate-Laufzeitsystems umzuwandeln. Der Herstellertreiber bietet die Schnittstelle zum DBVS.

Eine weitere, in dieser Arbeit nicht besprochene Aufgabe des X-Translate-JDBC-Treibers kann die Transformation von Fehlermeldungen sein.

**DBVS-Ausführungskomponente** Diese Komponente steht für die Anfrage-Verarbeitungskomponente des DBVS.

**XT-Main** Das Laufzeit-Hauptprogramm, X-Translate-Main, dient lediglich als Koordinator zwischen Parser und Compiler. Es ruft Parser, XSQL-Compiler und, falls nötig, den Anpassungscompiler auf.

**Parser** Der Parser überführt den vom Benutzer gelieferten String mit SQL-Anweisungen in den Quellbaum. Dazu bedient er sich der Basisgrammatik und allen Erweiterungen.

**XSQL-Compiler (XSQL→ SQL:1999+Hersteller)** Der XSQL-Compiler besteht aus einer Steuereinheit und den Transformationen aller installierten Sprachmodule. Es wird für jeden Knoten des Syntaxbaums geprüft, welche Transformationen passen, d.h. ob der jeweilige Knoten in das Knotenmuster der Transformation passt. Dann werden alle passenden Transformationen nacheinander angewendet.

Kommentar 4.2:

In dieser Arbeit wird davon ausgegangen, dass das Laufzeitsystem tatsächlich in der Datenbank gespeichert und ausgeführt wird. Es gibt nur eine Methode, die als SQL-Routine von außen angesprochen werden kann. Dadurch wird es auch für andere als den X-Translate-JDBC-Treiber einfach, das X-Translate-System zu benutzen. Wenn das Laufzeitsystem in den X-Translate-JDBC-Treiber gesetzt wird, muss er in jedem Fall benutzt werden.
Aktuell fällt die Java-Unterstützung bei den Herstellern unterschiedlich aus. Teils wird eine externe Java Virtual Machine (JVM) benutzt, teils eine interne. Die Javaklassen werden entweder in der Datenbank oder im Dateisystem gespeichert. Der Standard drückt sich in dieser Hinsicht leider nicht klar aus. Für die Arbeitsweise und die Performanz sind die Unterschiede jedoch nicht relevant, lediglich die Installationsprozedur muss an jedes DBVS angepasst werden.

Der Quellbaum wird mittels einer Tiefensuche durchlaufen und die Transformationen angewendet. Normalerweise ist die Reihenfolge der Transformationen irrelevant. Durch Abhängigkeiten in den Metadaten kann es je nach Ausführungsreihenfolge zu unterschiedlichen Ergebnissen kommen, das ist jedoch ein Konflikt und sollte bei einer korrekten Installation nicht vorkommen. Für die Implementierung ist eine festgelegte Reihenfolge jedoch einfacher, so kann der Zielbaum von oben nach unten aufgebaut werden.

**Anpassungs-Compiler (SQL:1999+Hersteller → Herstellerdialekt)** Der Anpassungs-Compiler ist ähnlich aufgebaut wie der XSQL-Compiler. Die Arbeitsweise ist jedoch nicht so komplex, da hier nicht auf mögliche spätere Spracherweiterungen geachtet werden muss. Dieser Compiler wird nur verwendet, wenn eine Herstelleranpassung installiert ist. In der Regel wird das der Fall sein, da zur Zeit kein Hersteller den vollen SQL:1999-Standard unterstützt.

### Ablauf einer SQL-Anfrage

Der Weg einer SQL-Anfrage durch das DBVS und X-Translate (Abbildung 4.3) wird jetzt schrittweise erläutert:

1. Der Benutzer setzt über den X-Translate-JDBC-Treiber eine Anfrage an das DBVS ab. Der X-Translate-Treiber wandelt die Anfrage um in einen SQL-Funktionsaufruf mit der Anfrage als Parameter. Die aufgerufene Funktion ist das X-Translate-Main-Programm.
2. Die DBVS-Ausführungskomponente ruft die Funktion auf.
3. Das X-Translate-Laufzeitsystem transformiert die Anfrage. Währenddessen kann auf das IS und das DSX zugegriffen werden.
4. Das X-Translate-Laufzeitsystem ruft die generierte SQL:1999-Anweisung auf.
5. Das X-Translate-Laufzeitsystem erhält das Ergebnis der Anfrage.
6. Das Ergebnis der SQL-Anfrage wird an den Benutzer zurückgegeben.

Während Punkt drei können Fehler auftreten, die behandelt werden müssen, und zwischen Punkt fünf und Punkt sechs muss möglicherweise eine Fehlermeldung des DBVS in eine Fehlermeldung eines Sprachmoduls umgewandelt werden. Nach Entscheidung **D3** wird dies hier nicht besprochen.

## 4.3 Entwicklung von XSQL-Sprachmodulen

In diesem Abschnitt wird detailliert auf die Entwicklung von Sprachmodulen eingegangen. Die meisten Beispiele haben als Grundlage die in Beispiel 4.2 dargestellte XSQL-Anweisung. Die Anweisung ähnelt der in den Beispielen 2.5 und 2.12 vorgestellten. Allerdings benutzt diese Anweisungen zwei Sprachmodule gleichzeitig: zum einen die Mehrfachvererbung, und zum zweiten die Schemamodularität. Beide Erweiterungen wurden in Kapitel 2 vorgestellt.

Beispiel 4.2: XSQL-Anweisung mit zwei Erweiterungen

```
CREATE TYPE Hiwi UNDER Uni::Student, Org::Angestellter AS (
    Aufgabe VARCHAR(30),
    Arbeitszeit INTEGER)
    NOT FINAL;
```

Der Syntaxbaum für die Anweisung findet sich in Abbildung 4.4. Der Syntaxbaum entspricht nicht vollständig der SQL:1999-Norm, einige grammatikalische Produktionen wurden der Einfachheit halber ausgelassen.

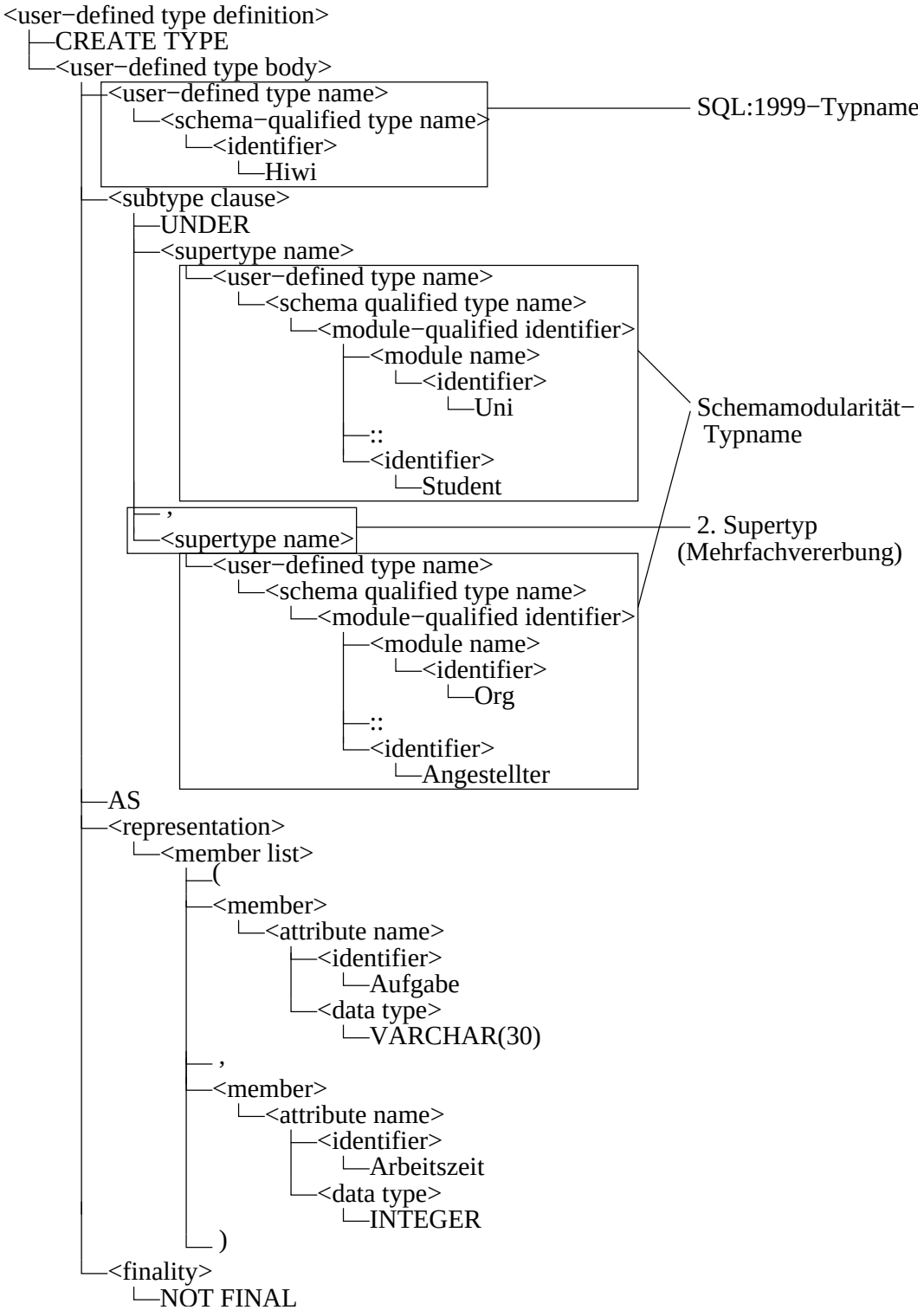
### 4.3.1 BNF für die Grammatik

Die Grammatikerweiterungen werden in der im SQL-Standard [ISO99a] benutzen BNF-Variante spezifiziert. Tabelle 4.2 enthält eine Liste der Symbole mit spezieller Bedeutung[2].

Wenn ein spezielles Symbol einzeln auf einer linken Seite steht, verliert es die spezielle Bedeutung. Klammern können beliebig tief geschachtelt werden, und Alternativen können auf jeder Tiefe benutzt werden.

[2]Sinngemäß übernommen aus [ISO99a], Seite 35.

Abbildung 4.4: Vereinfachter Syntaxbaum für Beispiel 4.2




Neue Produktionen und neue Alternativen für bereits existierende Produktionen werden auf die gleiche Art definiert: indem in der beschriebenen BNF-Variante die Produktion notiert wird. Der Parser-Generator fasst dann alle Produktionen, die die gleiche linke Seite haben, zu einer Produktion mit den entsprechenden Alternativen zusammen. Die Grammatik kann nur erweitert werden, und bleibt in jedem Fall abwärtskompatibel.

Beispiel 4.3 zeigt, welche Grammatikerweiterungen für die Anweisung aus Beispiel 4.2 notwendig sind.

Tabelle 4.2: Spezielle BNF-Symbole

| Symbol | Bedeutung |
|---|---|
| <> | In spitzen Klammern eingeschlossene Zeichenketten sind Namen von Nonterminalsymbolen. |
| ::= | Der Definitionsoperator trennt linke Seite (das zu definierende Nonterminalsymbol) von der rechten Seite (der Ersetzung für das Nonterminalsymbol) der Produktion. |
| [] | In eckigen Klammern werden optionale Elemente eingefasst. |
| {} | Geschweifte Klammern dienen zur Gruppierung mehrerer Elemente. |
| \| | Der Alternativenoperator trennt zwei alternative Ersetzungen. Wenn er nicht in eckigen oder geschweiften Klammern eingeschlossen ist, trennt er zwei komplette Alternativen für das von der Produktion definierte Element. Andernfalls spezifiziert er Alternativen für den Inhalt der innersten umschließenden Klammer. |
| ... | Das Element direkt vor der Ellipse kann beliebig oft wiederholt werden. Wenn die Ellipse direkt auf eine schließende geschweifte Klammer (}) folgt, gilt sie für die komplette Gruppe. |
| !! | Nach zwei Ausrufezeichen folgt normaler Text. Damit können Syntaxelemente, die aus irgend einem Grund nicht in BNF ausgedrückt werden können, in natürlicher Sprache definiert werden. Das kann beispielsweise bei der Definition von Zeichen, die im benutzten Zeichensatz nicht vorhanden sind, notwendig werden. |
| Whitespace | Jede Form von Whitespace trennt syntaktische Elemente. |

### 4.3.2 Die Metamodell-Sprache

Die Metamodell-Sprache dient zur Definition der Sichten im ISX und der Tabellen im DSX.

**Definition der Tabellen im DSX** Die Tabellendefinitionen für das DSX sind unkompliziert. Die Syntax entspricht im wesentlichen der einer normalen SQL-`CREATE TABLE`-Anweisung. Der Tabellenname darf aber keinen Schemanamen als Bestandteil haben, da die Tabellen immer im DSX angelegt werden. Um die Eindeutigkeit von Namen zu gewährleisten, stellt X-Translate dem Tabellennamen einen eindeutigen Bezeichner für das Sprachmodul voran. Das geschieht systemintern und muss vom Sprachmodul-Entwickler nicht berücksichtigt werden.

Für jede Sicht im IS wird im DSX eine Tabelle mit den gleichen Spalten angelegt. Die Tabellen enthalten eine zusätzliche Spalte, in der für jede Zeile das zugehörige Sprachmodul eingetragen wird.

**Definition der Maskierungstabellen im DSX** In die Maskierungstabellen werden die nur intern genutzten Metadaten eingetragen, die für den Benutzer unsichtbar sein sollen. Durch eine `WHERE NOT EXISTS...`-Klausel (Beispiel 4.4)

Beispiel 4.3: Grammatikerweiterungen für Beispiel 4.2

Im Mehrfachvererbungs-Modul:

⟨subtype clause⟩ ::= UNDER ⟨supertype name⟩ [{, ⟨supertype name⟩}...]

Im Schemamodularität-Modul:

⟨schema-qualified type name⟩ ::= [⟨schema-qualified identifier⟩]
    ⟨module-qualified identifier⟩

⟨module-qualified identifier⟩ ::= ⟨module name⟩ ::
    ⟨qualified indentifier⟩

⟨module name⟩ ::= ⟨identifier⟩

Kommentar 4.3:

In Beispiel 4.2 wird zur Trennung von Modul- und Typnamen ein zweifacher Doppelpunkt (::) benutzt. Eleganter wäre es, der SQL:1999-Vorgabe zu folgen, und, genau wie zur Trennung von Schemanamen und Typnamen, einen Punkt (.) zu benutzen. Dann lässt sich jedoch nicht mehr in jedem Fall entscheiden, ob ein Modul oder ein Schemaname zur Qualifizierung des Typnamens genutzt wird. Hier stoßen wir auf eine Grenze der Erweiterbarkeit in syntaktischer Hinsicht. IP (Abschnitt 5.3) hat dieses Problem nicht, da IP ohne eine Syntax auskommt.
Tatsächlich hat SQL:1999 dieses Problem bereits: Schema- und Typnamen werden durch einen Punkt getrennt, ebenso Typnamen und Attributnamen. Falls eine Zweideutigkeit vorhanden ist, muss der voll qualifizierte Name des Elements angegeben werden.

werden diese Metadaten von Benutzeranfragen ausgeschlossen. Die Maskierungstabellen werden automatisch angelegt. Für jede Sicht im IS legt das X-Translate-System eine Tabelle im DSX an. Die Tabellen enthalten die gleichen Spalten wie die Sichten. Ein leeres X-Translate-System, d. h. ohne installierte Sprachmodule, definiert die vom IS ins ISX übernommenen Sichten wie in Beispiel 4.4 dargestellt. In vielen Sichten des IS werden Schemas referenziert. Da die Schemas DSX, XSS und IS komplett maskiert werden müssen, ist es nicht sinnvoll, für jedes Schemaobjekt dieser Schemas einen Eintrag in die Maskierungstabellen zu machen. Die Schemas werden durch eine einfache Zusatzklausel, ebenfalls in Beispiel 4.4 ersichtlich, ausgeschlossen.

**Definition neuer Sichten im ISX** Neue Sichten im ISX werden auf der Basis der bereits vorhandenen Sichten und Tabellen im ISX und DSX definiert. Somit entfällt hier die Maskierung von internen Metadaten. Es kann eine normale SQL:1999-`CREATE VIEW`-Anweisung benutzt werden.

**Erweiterung von Sichten im ISX** Um Sichten zu erweitern, bietet die Metamodell-Sprache einen neuen Befehl. Beispiel 4.5 zeigt eine mögliche Anwendung. In Zeile 1 wird die zu erweiternde Sicht angegeben. Danach folgt eine normale `SELECT`-Anweisung, um alle Tupel auszuwählen. Wie in einer SQL:1999-`SELECT`-Anweisung kann auch hier eine `WHERE`-Klausel zum

Beispiel 4.4: Sichtendefinition für das ISX in einem leeren X-Translate-System

```
CREATE VIEW INFORMATION_SCHEMA_EXTENDED.DIRECT_SUPERTYPES AS
    SELECT UDT_CATALOG, UDT_SCHEMA, UDT_NAME
    FROM INFORMATION_SCHEMA.DIRECT_SUPERTYPES AS v
    WHERE NOT EXISTS ( -- Maskierung
        SELECT UDT_CATALOG, UDT_SCHEMA, UDT_NAME
        FROM DEFINITION_SCHEMA_EXTENDED.DIRECT_SUPERTYPES_mask AS m
        WHERE v.UDT_CATALOG = m.UDT_CATALOG
        AND v.UDT_SCHEMA = m.UDT_SCHEMA AND v.UDT_NAME = m.UDT_NAME
    )
    AND UDT_SCHEMA NOT IN ('INFORMATION_SCHEMA',
                  'DEFINITION_SCHEMA_EXTENDED', 'XT_SYSTEM_SCHEMA')
    UNION
    SELECT UDT_CATALOG, UDT_SCHEMA, UDT_NAME
    FROM DEFINITION_SCHEMA_EXTENDED.DIRECT_SUPERTYPES;
```

Einschränken der Auswahl angegeben werden, was in diesem Fall nicht erforderlich ist. In den Zeilen 4-6 wird angegeben, wie der Verbund mit der bereits existierenden Sicht vollzogen werden soll. Es muss darauf geachtet werden, dass die verwendeten Spaltennamen in der SELECT-Klausel eindeutig sind, da sie später referenziert werden (siehe auch Beispiel 4.6).

Beispiel 4.5: Erweiterung einer Sicht

```
1 EXTEND VIEW DIRECT_SUPERTYPES AS c WITH
2    SELECT UDT_MODULE
3    FROM DEFINITION_SCHEMA_EXTENDED.SM_TABLE_TYPES AS x,
4    JOIN PREDICATE
5       c.UDT_CATALOG = x.UDT_CATALOG AND c.UDT_SCHEMA = x.UDT_SCHEMA
6           AND c.UDT_NAME = x.UDT_NAME
```

Bei der Installation des Sprachmoduls wird die alte Sicht gelöscht und neu angelegt. Dabei wird die Basis-Definition und alle Sicht-Erweiterungen zusammengefügt. Als einfache Art der Implementierung bieten sich abgeleitete Tabellen (engl. derived tables) aus SQL:1999 an. Beispiel 4.6 zeigt in abgekürzter Form, wie eine erweiterte Sichtendefinition aussehen könnte.

Beispiel 4.6: Erweiterte Definition einer Sicht

```
CREATE VIEW DIRECT_SUPERTYPES AS
SELECT UDT_CATALOG, UDT_SCHEMA, UDT_NAME
FROM ( /* SELECT-Statement aus Beispiel 4.4 */) AS c,
     ( /* SELECT-Statement aus Beispiel 4.5 */) AS x,
WHERE
     c.UDT_CATALOG = x.UDT_CATALOG AND c.UDT_SCHEMA = x.UDT_SCHEMA
         AND c.UDT_NAME = x.UDT_NAME;
```

Wenn viele Sprachmodule die gleiche Sicht erweitern, kann das zu einer unnötig komplexen Sichtendefinition führen. Hier kann eine manuelle Optimierung durch den Administrator helfen.

### 4.3.3 Definition der Systemdaten

Falls ein Sprachmodul Daten benötigt, die vor dem Einsatz des Moduls vorhanden sein müssen, können diese mit normalen SQL-Anweisungen spezifiziert werden. Alle anderen Operationen auf der Datenbank werden durch die Transformationen vorgenommen.

## 4.4 Die Transformationssprache

In diesem Abschnitt wird die Transformationssprache vorgestellt. Mit ihr können Methoden und Transformationen für eine Spracherweiterung entwickelt werden. Die Gesamtheit der Methoden und Transformationen aller installierten Sprachmodule werden zur Generierung des Compilers genutzt.

Es wird hauptsächlich die allgemeine Form der Transformationssprache beschrieben und an Beispielen veranschaulicht. Die Syntaxregeln und Details zur Sprache finden sich in Anhang B.

Um die Beschreibung zu vereinfachen, werden hier die im Folgenden benutzten Begriffe definiert.

**Aktueller Knoten** Der aktuelle Knoten ist der Subjektknoten des Knotenmusters der Transformation, die gerade ausgeführt wird.

**Methodenknoten** Eine Methode wird immer auf einem bestimmten Knoten ausgeführt. Dieser wird als Methodenknoten bezeichnet.

**Klasse** Die Klasse, der eine Methode angehört, bestimmt sich durch das bei der Methodendefinition angegebene Knotenmuster. Statt *Knotenmuster* wird im Folgenden der Begriff *Klasse* verwendet. Der Methodenknoten ist ein Objekt der Methodenklasse.

### 4.4.1 Angabe von Bäumen und Knotenmustern

In Methoden und Transformationen müssen des Öfteren Bäume oder Knotenmuster angegeben werden. Hierzu wird die im Folgenden beschriebene Klammerschreibweise verwendet. Eine Entwicklungsumgebung für Sprachmodule kann zur intuitiveren Handhabung auch eine grafische, baumorientierte Eingabemöglichkeit vorsehen.

In der Klammerschreibweise wird jeder Knoten in der Form $(k, c_1, c_2, \ldots, c_n)$ ausgedrückt. Mit $k$ ist der Knoten selbst gemeint, $c_i$ bezeichnet die Kindknoten. Falls ein Knoten keine Kinder hat, kann die Klammer weggelassen werden. Jedem Knoten ist ein Typ zugewiesen, dieser Typ entspricht einem Symbol aus der XSQL-Grammatik.

Für Knotenmuster gibt es noch spezielle Wildcard-Knotentypen und eine Markierung für den Subjektknoten. Statt Komma (,) kann bei Knotenmustern auch ein Längsstrich (|) zur Trennung der Kindknoten verwendet werden. Damit werden, wie in BNF, Alternativen gekennzeichnet.

Der Datentyp für Bäume und Knotenmuster heißt `Tree`. Die Unterscheidung zwischen Baum und Knotenmuster wird über das Vorhandensein von Wildcards und über die Syntax gemacht. Ein `Tree` mit Wildcard-Symbolen ist ein Knotenmuster. Wenn keine Wildcards vorkommen, ist es ein Baum oder ein Knotenmuster. Das ist kein Konflikt, da in der Syntax immer *entweder* ein Baum *oder* ein Knotenmuster erwartet wird.

Für die verschiedenen Knotentypen gibt es die in Tabelle 4.3 gezeigten Schreibweisen. Mit (K) markierte Regeln gelten nur für Knotenmuster.

Tabelle 4.3: Schreibweisen für Knotentypen

| | |
|---|---|
| <name> | Ein Nonterminalsymbol aus der XSQL-Grammatik. |
| <name>* | Der Subjektknoten bei Knotenmustern. Bei Bäumen wird so der Bezugsknoten markiert. |
| <name>+ | Knoten mit direktem Zugriff. Wird bei den Zugriffsmustern in Transformationen benutzt (K). |
| <name>- | Knoten ohne direktem Zugriff. Wird bei den Zugriffsmustern in Transformationen benutzt (K). |
| [name] | Ein Terminalsymbol aus der XSQL-Grammatik. |
| ? | Ein beliebiges Nonterminalsymbol (K). |
| $?x..y$ | Zwischen $x$ und $y$ beliebige Nonterminalsymbole (K). |
| $?*$ | Beliebig viele Nonterminalsymbole (K). |
| $@x..y$ | Ein Nonterminalsymbol zwischen $x$ und $y$ Stufen weiter unten (K). |
| $@*$ | Ein Nonterminalsymbol beliebig viele Stufen weiter unten (K). |

Die Wildcards können alleine oder mit einem Terminal-/Nonterminalsymbol zusammen stehen. Wenn sie alleine stehen, gelten sie für beliebige Knotentypen, andernfalls nur für die spezifizierten. Wenn zwischen zwei Knoten *kein* Wildcardsymbol steht ist das äquivalent zu einem $?*$.

Nach einem Nonterminalsymbol oder einem Terminalsymbol kann, durch Doppelpunkt (:) abgetrennt, ein Kennzeichen stehen, um den Knoten später zu einfacher zu referenzieren.

Das Schlüsselwort `this` steht in einer Methode für den Methodenknoten, und in einer Transformation für den Subjektknoten.

In Abbildung 4.5 sind zwei Knotenmuster einmal in Klammerschreibweise und einmal in Baumdarstellung zu sehen.

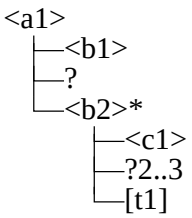
### 4.4.2 Erstellen und manipulieren von Bäumen

Methoden und Transformationen haben als Ausgabe Teilbäume. Um diese zu erstellen und zu manipulieren bietet X-Translate mehrere Möglichkeiten, die im Folgenden erklärt werden. In Beispiel 4.7 werden diese Möglichkeiten dargestellt.

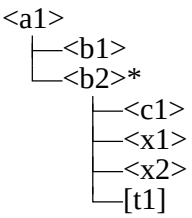
Abbildung 4.5: Knotenmuster

Knotenmuster (<a1>, <b1>, (<b2>*, ?, <c1>, ?2..3, [t1]))

Als Baum

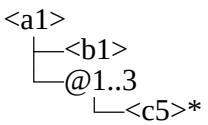



Passender Teilbaum

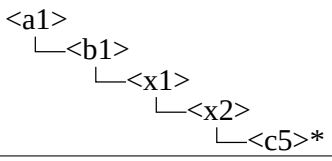



Knotenmuster (<a1>, <b1>, (@1..3, <c5>*))

Als Baum




Passender Teilbaum




### Erstellen von Bäumen

Bäume haben den Datentyp `Tree`. Durch die Deklarierung einer Variablen und Zuweisung eines Literals (Beispiel 4.7, Zeilen 2-5) kann ein Baum erstellt werden.

### Manipulation von Bäumen

Die Manipulation von Bäumen umfasst das Zuweisen von Teilbäumen an Knoten. Einzelne Knoten können auf zwei Arten angegeben werden.

- Als Pfadausdruck: Pfadausdrücke starten vom `this`-Knoten aus oder vom Wurzelknoten eines Baumes. Die Schreibweise ähnelt der von Verzeichnisnamen eines Dateisystems: am Anfang steht eine Raute (#), dann `this` oder der Baumname, gefolgt von „:/“. Danach folgen alle Knoten bis zum Ziel, durch „/“ getrennt. Elternknoten werden mit „..“ bezeichnet. Wenn es mehrere Kindknoten mit gleichem Namen gibt, kann durch eine dem Namen nachgestellte, in eckigen Klammern eingefasste Zahl der richtige ausgewählt werden. Es kann auch nur eine Zahl ohne Namen angegeben werden, falls der Name des Kindknoten nicht bekannt ist. Das ist bei Terminalsymbolknoten oft der Fall.

  Transaktionen können auf ihre Kindknoten entweder per Methodenaufruf oder direkt zugreifen. Um auf einen Knoten direkt zuzugreifen, muss im Pfadausdruck der Knotenname von einem Ausrufezeichen (!) gefolgt werden. Auf alle Knoten vom aktuellen Knoten bis hin zum mit „!“ gekennzeichneten Knoten wird direkt zugegriffen.

- Direkt: Die direkte Angabe von Knoten ist nur möglich, wenn sie vorher gekennzeichnet wurden. Dann kann ein Knoten mit dem Baumnamen oder `this`, gefolgt vom Kennzeichen in eckigen Klammern, angegeben werden.

In den Zeilen 7-8 wird einem direkt angegebenen Knoten ein neuer (Teil)Baum zugewiesen. In den Zeilen 11-12 wird einem per Pfadausdruck angegebenen Knoten ein neuer, ebenfalls per Pfadausdruck angegebener, Baum zugewiesen.

Zeile 15 zeigt, wie einzelne Terminalknoten geändert werden können.

Beispiel 4.7: Beispiele für Operationen auf Bäumen

```
1    // Zuweisung eines Baumes an eine Variable
2    Tree x = #(<supertype name>,
3                (<user-defined type name>,
4                    <schema-qualified type name>:tmp))
5
6    // Zuweisen eines Baumes an einen bestehenden Knoten (direkte)
7    #x[tmp] = #(<schema-qualified type name>,
8                         (<identifier>, [Angestellter]:tmp2));
9
10   // Zuweisung über Pfadausdrücke, direkter Knotenzugriff
11   #x:/<supertype name>/<user-defined type name> =
12                                  #this:/<user-defined type name>!
13
14   // Ändern von "Angestellter" auf "Beamter"
15   #x[tmp2] = #[Beamter];
```

### 4.4.3 Quellbaum: Struktur und Zugriff

Die interne Struktur des Quellbaumes entspricht der eines normalen Baumes. Der Zugriff auf die Baumknoten unterliegt jedoch bestimmten Regeln: eine Transformation kann nur auf den aktuellen Knoten und dessen Kinder direkt zugreifen. Auf alle anderen Knoten wird nur per Methodenaufruf zugegriffen. Auch auf die Kindknoten kann so zugegriffen werden. Zugriffe über Methoden sind direkten Zugriffen vorzuziehen, da alle Knoten, auf die direkt Zugegriffen wird, nicht mehr von anderen Spracherweiterungen transformiert werden können.

Durch die Verwendung von Methoden wird die tatsächliche Struktur des Quellbaumes verborgen.

X-Translate stellt bereits alle Methoden bereit, die nötig sind, um in Bäumen mit ausschließlich SQL:1999 zu navigieren.

Bei Zugriffen auf Kindknoten muss zusätzlich auch die Zugriffsart (Methodenaufruf oder direkt) angegeben werden. In den anderen Fällen werden automatisch Methoden verwendet.

### 4.4.4 Methoden

Methoden dienen dazu, den Quellbaum für Transformationen anderer Sprachmodule als SQL:1999-Syntaxbaum erscheinen zu lassen. Eine Methode ist nichts anderes als eine Sicht auf den Knoten, für den sie aufgerufen wird. Methoden können überladen

werden, so dass je nach Aufrufkontext und Knotenklasse verschiedene Methoden ausgeführt werden.

### Funktionsweise einer Methode

Für jeden Knoten im Quellbaum existiert mindestens eine Methode. Wenn kein Sprachmodul eine Methode für einen bestimmten Knotentyp in einem bestimmten Knotenmuster bereitstellt, so wird dafür eine eingebaute Methode benutzt. Je nach dem Knotenmuster des aktuellen Knotens wird eine Methode ausgewählt und ausgeführt.

Der Zugriff auf einen Knoten über eine Methode unterscheidet sich von einem direkten Zugriff dadurch, das der Rückgabewert der Methode ein *virtueller Knoten* ist. Die Methode passt die *Attribute* des ursprünglichen Knotens an, so dass er nach außen hin voll SQL:1999-kompatibel erscheint. Als Attribute werden hier alle direkt benachbarten Knoten gesehen. Das sind die Kindknoten und der Elternknoten. Geschwisterknoten sind nur Attribute des Elternknotens.

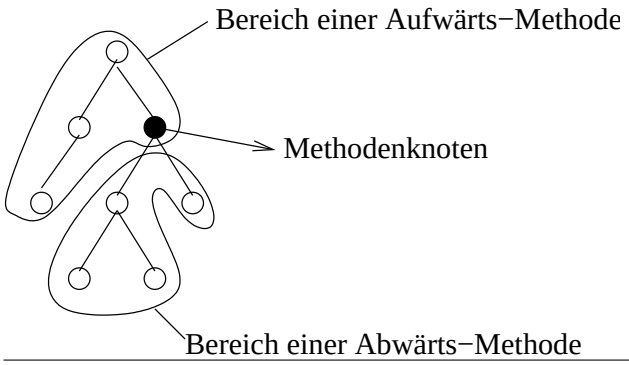
Im einfachsten Fall, der auch bei den eingebauten Methoden zutrifft, entspricht der von der Methode generierte virtuelle Knoten exakt dem Ursprungsknoten.

Eine Methode kann jedoch auch Kindknoten auslassen, deren Reihenfolge ändern oder neue Kindknoten hinzufügen. Beim Hinzufügen muss auch der Teilbaum unter dem neuen Kindknoten von der Methode generiert werden. Es können auch Teile des Quellbaumes komplett übernommen werden. Eine solche Methode wird für Zugriffe *von oben*, d. h. einer höheren Hierarchieebene im Baum, aufgerufen und liefert eine Sicht auf den Methodenknoten und dessen Kindknoten. Sie verändert die Sicht auf den Quellbaum vom Methodenknoten aus *abwärts*.

Der aktuelle Knoten kann auch ein Nachfahre des Methodenknotens sein. Dann wird auf die Methoden *von unten* zugegriffen. Es wird möglicherweise eine Methode gebraucht, die die Sicht auf dem Quellbaum vom Methodenknoten an *aufwärts* verändert. Hierfür gibt es ähnliche Operationen wie bei den Abwärts-Methoden.

Methoden sind immer entweder Aufwärts- oder Abwärts-Methoden. Eine Methode darf nie die Sicht auf den Quellbaum in die Richtung ändern, aus der die Anfrage kommt. Abbildung 4.6 veranschaulicht, welche Bereiche jeweils von einer Aufwärts- bzw. Abwärts-Methode geändert werden können.

Abbildung 4.6: Bereiche von Aufwärts- und Abwärts-Methoden




Die Ausgabe einer Methode wird nur einmal berechnet und in einem Zwischenspeicher abgelegt. Um Seiteneffekte zu vermeiden, kann eine Methode nur lesend auf die

Datenbank zugreifen, und die Ausgaben aller Methoden werden vor dem Beginn der Transformation berechnet.

### Aufbau einer Methode

Eine Methode hat mehrere Bestandteile.

- Der *Name* der Methode gibt an, für welchen Knotentyp sie geschrieben ist.
- Der *Typ* einer Methode ist entweder `up` (aufwärts) oder `down` (abwärts).
- Die *Klasse* der Methode ist das Knotenmuster mit dem Namen der Methode als Subjekt. Die Methode wird aufgerufen, wenn der Knoten in die Klasse gehört, d. h. auf das Knotenmuster passt.
- Das *Aufrufsknotenmuster* bestimmt, auf welches Knotenmuster der aktuelle Knoten passen muss. Es ist optional; ohne Angabe gilt die Methode für alle Aufrufer.
- Die Angabe der *aufrufenden Spracherweiterung* erlaubt eine beschränkte Interaktion zwischen verschiedenen Sprachmodulen, indem für bestimmte Erweiterungen spezielle Methoden geschrieben werden können, um eventuelle Inkompatibilitäten zu reparieren. Diese Angabe ist optional; wenn keine Spracherweiterung angegeben wird, gilt die Methode für alle Erweiterungen.
- Im Rumpf der Methode stehen Anweisungen, um die Knoten des virtuellen Baumes zu generieren.

Anhand der *Signatur* einer Methode kann festgestellt werden, ob zwei Methoden im Konflikt zueinander stehen, weil sie für denselben Knoten eingesetzt werden. Zur Signatur gehören Name, Typ, Klasse, Aufrufsknotenmuster und die aufrufende Spracherweiterung.

Beispiel 4.8 zeigt eine einfache Methode. In Zeile 1 wird die Methode als `down`-Methode für den Knoten <schema-qualified type name> deklariert. Die Zeilen 2-6 bilden das die Klasse definierende Knotenmuster. Danach könnte in der Form `from (<...>)` das Aufrufs-Knotenmuster stehen, gefolgt von `of <...>` um die aufrufende Spracherweiterung anzugeben. Das ist bei dieser Methode nicht der Fall, da sie von überall aus aufgerufen werden kann. Im Rumpf der Methode wird zuerst ein Gerüst für den Rückgabewert angelegt (Zeilen 9-13). Hier erkennt man, das das Nonterminalsymbol <module-qualified type name> nicht mehr auftaucht. Genau das ist die Aufgabe der Methode. Das <module-qualified type name>-Symbol ist aus der Schemamodularität-Spracherweiterung. Intern bildet diese Erweiterung Typnamen in Modul-Namensräumen auf einfache Typnamen nach einem einfachen Schema ab. Für andere Spracherweiterungen ist nur dieser Name sichtbar. Im Ergebnisbaum stehen einige Fragezeichen, die noch ersetzt werden müssen. In Zeile 15 wird der Teil des Quellbaumes, für den das Fragezeichen in Zeile 5 steht, in den Ausgabebaum kopiert. Falls sich im Quellbaum an dieser kein Knoten befindet, wird auch nichts kopiert. In den Zeilen 17-20 wird der Modulname und der Typname aus dem Quellbaum gelesen und in einem String gespeichert. Danach wird mit einer SQL-Anweisung in der Abbildungstabelle der kodierte Typname gesucht und

der Variablen `mappedName` zugewiesen. Wenn kein Eintrag in der Abbildungstabelle vorhanden ist, ist das ein Fehler. Das drückt sich durch die Rückgabe eines `null`-Wertes aus (Zeile 25). Eine für den praktischen Einsatz geeignete Methode müsste hier einen Fehlercode zurückgeben oder eine Ausnahme werfen. Der kodierte Name wird in Zeile 27 den Ausgabebaum eingesetzt, und in Zeile 28 wird der generierte Teilbaum zurückgegeben.

Beispiel 4.8: Eine Methode, die die Kindknoten umstrukturiert

```
1   down method <schema-qualified type name>
2       #(<supertype name>,
3           (<user-defined type name>,
4               (<schema-qualified type name>*,
5                   ?[0..1]:x1,
6                   <module-qualified type name>)))
7       // from und of braucht nicht angegeben werden
8       {
9       Tree result =
10          #(this,
11              ?:x2,
12              (<qualified identifier>,
13                  <identifier>, x3?));
14
15      #result[x2] = #this[x1];
16
17      String mName = #this:/<module-qualified type name>/
18              <module name>/<identifier>/[1]
19      String tName = #this:/<module-qualified type name>/
20              <qualified identifier>/<identifier>/[1]
21
22      String mappedName = null;
23      #sql { SELECT mapping FROM smMappings INTO mappedName
24              WHERE module = mName AND type = tName };
25      if (mappedName == null) return null;
26
27      #result[x3] = mappedName;
28      return result;
29  }
```

### Auswahl von Methoden

Bevor eine Methode aufgerufen wird, muss sie erst ausgewählt werden. Das erfolgt, in dem eine Methode gefunden wird, für die die Knotenmuster des aktuellen und des Methodenknotens passen. Hier gibt es ein Problem: um zu prüfen, welche Knotenmuster passt, müssen möglicherweise andere Methoden aufgerufen werden, und diese rufen bei der Prüfung die erste Methode wieder auf. Um diese Zyklen zu vermeiden, wird anders vorgegangen: es werden alle möglichen Kombinationen ausprobiert und

jeweils geprüft, ob daraus eine gültige Methodenauswahl resultiert. Wenn mehrere Kombinationen gültig sind, ist das ein Konflikt.

Theoretisch kann es passieren, das die Methoden so voneinander abhängen, das jede Kombination von Methoden auf allen Knoten ausprobiert werden muss. Es gibt zwar nur eine endliche Anzahl von Möglichkeiten, aber in der Praxis wäre hier der Rechenaufwand zu hoch, was den sinnvollen Einsatz von X-Translate verhindern würde.

In der Regel dürften solche Abhängigkeiten nach unserer Einschätzung jedoch sehr selten auftreten. Fast alle Knoten im Quellbaum werden nur eine Methode haben, und die fast alle Knotenmuster werden die Form $(< k > *)$ haben. Bei Knotentypen, für die es mehrere Methoden gibt, sind die Knotentypen im Knotenmuster normalerweise einfache Knoten (d.h. Knoten mit nur einer Methode).

### 4.4.5 Zielbaum: Struktur und Zugriff

So wie auf den Quellbaum ausschließlich lesend zugegriffen werden kann, kann man auf den Zielbaum nur schreibend zugreifen. Die Ausgabe der Transformationen wird vom System automatisch an die richtige Stelle im Zielbaum gesetzt.

### 4.4.6 Transformationen

Methoden dienen zum Zugriff auf den Quellbaum, und die Transformationen generieren aus dem Quellbaum einen den Zielbaum. Man kann auch sagen, sie *transformieren den Quellbaum*, daher ihr Name. Diese Aussage ist aber nur im übertragenen Sinne richtig, da der Quellbaum tatsächlich nicht verändert wird.

#### Typen

Es gibt verschiedene Typen von Transformationen. Die Einteilung in Typen erfolgt nach der Art, inwieweit die jeweilige Transformation den Quellknoten strukturell verändert. Nicht berücksichtigt bei der Einteilung wird der Lese- und Schreibzugriff auf die Metadaten. Es können drei Typen unterschieden werden.

**Lesende Transformationen** Dieser Typ von Transformation verändert den Syntaxbaum nicht. Lesende Transformationen können aber auf die Metadaten lesend und schreibend zugreifen, und somit den Zustand der Datenbank verändern. Deshalb wird auch hier der Begriff *Transformation* verwendet. Da lesende Transformationen bezüglich des Syntaxbaums keine Konflikte verursachen können, darf es für jeden Knoten beliebig viele lesende Transformationen geben.

**Änderungstransformationen** Dieser Typ von Transformation ändert die Struktur eines Teilbaums. Um Konflikte zu vermeiden, darf für jeden Knoten maximal eine Änderungstransformation existieren.

**Ergänzungstransformationen** Hier wird die Struktur der vorhandenen Knoten im Syntaxbaum nicht verändert, sondern es werden lediglich neue Geschwisterknoten hinzugefügt. Das ist der Fall, wenn aus einer XSQL-Anweisung mehrere

SQL:1999-Anweisungen entstehen. Bei einer Ergänzungstransformation wird der betreffende Knoten eine Stufe tiefer in die Syntaxbaumhierarchie gestellt und an seinen vorigen Platz wird eine Verbundanweisung (<compound statement>) [ISO99c] mit atomarem Ausführungskontext (engl. atomic execution context) [ISO99b, ISO99c], im Folgenden kurz *Atomic-Block*, eingefügt. Das setzt voraus, das die Syntaxregeln des Elternknotens dies erlauben.

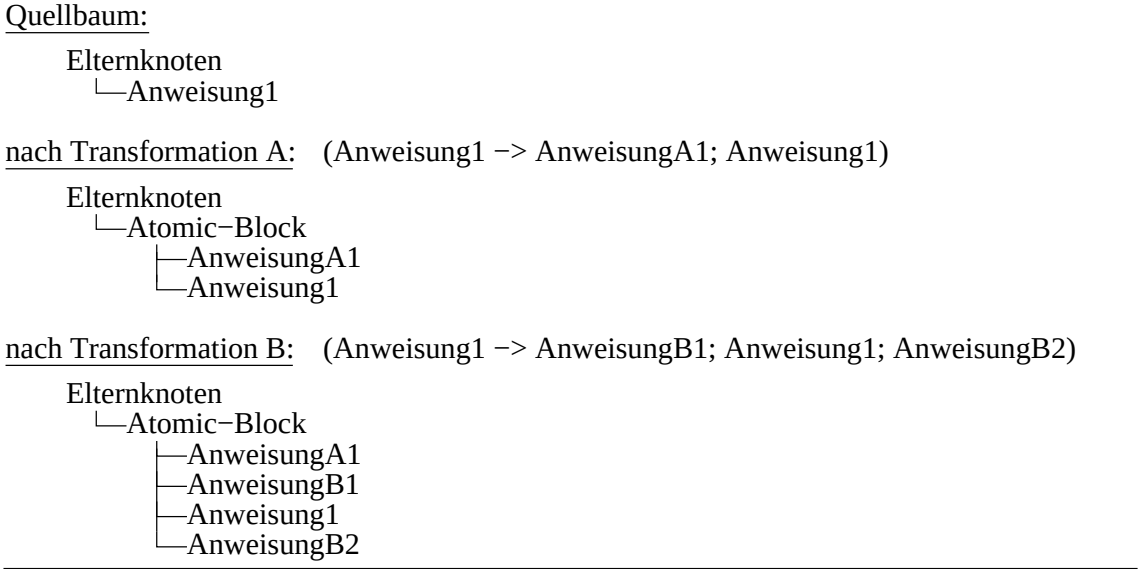
Für einen Knoten kann es beliebig viele Ergänzungstransformationen geben, das Einfügen des Atomic-Blocks geschieht nur einmal.

In Abbildung 4.7 wird das Vorgehen bei einer Ergänzungstransformation noch einmal deutlich.

Auch für andere Syntaxelemente ist diese Vorgehensweise denkbar. Voraussetzung ist die Möglichkeit, den bearbeiteten Knoten eine Ebene tiefer zu setzen und stattdessen eine Art Listenknoten einzufügen. Das hier besprochene System beschränkt sich aber auf die beschriebene Art von Ergänzungstransformation.

Ergänzungs- und Änderungstransformation können auch kombiniert auftreten.

Abbildung 4.7: Einfügen eines Atomic-Blocks




### Funktionsweise

Vom Wurzelknoten des Quellbaumes angefangen, werden alle Knoten von oben nach unten und von links nach rechts (Tiefensuche) durchlaufen. Für jeden Knoten wird geprüft, ob eine oder mehrere Transformationen existieren und diese dann ausgeführt. Das bedeutet, es werden auch die Knoten besucht, die im Zielbaum nicht mehr auftauchen. Das kann passieren, wenn ein Änderungsknoten einen seiner Kindknoten nicht mit in den Zielbaum übernimmt. Bei lesenden Transformationen kann es in einigen Fällen trotzdem sinnvoll sein, sie auszuführen. Falls Änderungs- oder Ergänzungstransformationen auf nicht mehr vorhandenen Knoten ausgeführt werden, ist das ein Fehler bzw. Konflikt. Bei der Installation müssen solche möglichen

Konflikte untersucht werden, und bei der Möglichkeit des Auftretens Gegenmaßnahmen getroffen werden.

In Herstellermodulen gibt es eine globale Variable `DBVS`, in der der Name des DBVS gespeichert ist. Transformationen können anhand dieser Variablen ihre Aktionen steuern.

### Aufbau

Die drei Transformationstypen haben einen ähnlichen Aufbau. Die Elemente einer Transformation sind im Folgenden aufgeführt.

- Der *Typ* der Transformation kann `reading` (lesend), `appending` (ergänzend) oder `changing` (ändernd) sein.
- Ein *Knotenmuster* bestimmt die Klasse der Knoten, auf die die Transformation angewendet wird.
- Ein spezielles Knotenmuster, das *Zugriffsmuster*, dient zur Spezifikation, auf welche Kindknoten direkt zugegriffen wird und auf welche nicht. Wenn dieses Muster nicht angegeben wird, nimmt das System an, das auf alle Kindknoten direkt zugegriffen wird, und fragt bei der Installation den Administrator, falls ein Konflikt auftritt. *Direkter Zugriff* bedeutet in diesem Zusammenhang den direkten Zugriff auf einen Kindknoten ohne über eine Methode zu gehen, oder das Nicht-Übernehmen des Knotens in den Zielbaum.
- Im Rumpf der Transformation stehen die Transformationsanweisungen.

Beispiel 4.9 zeigt den Ausschnitt einer Transformation aus dem Mehrfachvererbungs-Sprachmodul.

In Zeile 1 wird der Typ der Transformation spezifiziert. In diesem Fall handelt es sich um eine kombinierte Ergänzungs- und Änderungstransformation. Die Zeilen 2-6 spezifizieren das Knotenmuster für die Transformation. Das Knotenmuster passt nicht auf alle <user-defined type definition>-Symbole, sondern nur auf die mit mehr als einem Supertyp. In den Zeilen 9-16 ist ein Teil des Zugriffsmusters zu sehen. So wird durch das Minuszeichen in Zeile 12 spezifiziert, dass diese Transformation den <user-defined type name>-Knoten nicht ändert, und dass andere Transformationen auf diesem Knoten arbeiten dürfen.

Die Zeilen 17-30 beinhalten einen Ausschnitt aus dem Rumpf der Transformation. Ähnlich wie in den Methoden kann hier Javacode stehen, es kann lesend, aber auch schreibend auf die Datenbank zugegriffen werden und es werden Ausgabebäume erzeugt. In Zeile 19 wird die Liste der zu ergänzenden Knoten *vor* dem Subjektknoten erzeugt. Die Liste ist leer, d.h. es werden keine Knoten vorher eingefügt. In Zeile 20 wird die Danach-Liste mit $x$ Elementen erzeugt ($x$ ist irgendwo vorher im Rumpf berechnet worden). Danach (Zeilen 21-23) werden Bäume für Cast-Anweisungen eingefügt (siehe auch Abschnitt 2.2). In Zeile 25 wird der Ausgabebaum angelegt. Möglicherweise werden einige SQL-Anweisungen nicht in den Zielbaum geschrieben, sondern direkt ausgeführt (Zeile 27). In Zeile 29 werden schließlich die generierten Knoten zurückgegeben.

Beispiel 4.9: Transformation einer Typdefinition mit zwei Supertypen

```
 1  appending, changing transformation
 2      #(<user-defined type definition>*,
 3          ?,
 4          (<user-defined type body>,
 5              ?,
 6              (<subtype clause>, ?[2..*]<supertype name)))
 8      accesses
 9          #(<user-defined type definition>*,
10              ?-,
11              (<user-defined type body>+,
12                  <user-defined type name>-,
13                  <subtype clause>+,
14                 // ...
15              )
16          )
17  {
18      // ...
19      Tree[] before = new Tree[0]; // Keine Ergänzung vorher.
20      Tree[] after = new Tree[x];
21      for (int i = 0; i < noOfSuperTypes - 1; i++) {
22          after[i] = (/* CREATE CAST für i+1ten Supertyp */);
23      }
24      // ...
25      Tree result = #(<user-defined type definition>, ...)
26      // ...
27      #sql { ... };
28
29      return (before, after), result;
30  }
```

Die Entscheidung, ob eine SQL-Anweisung in den Quellbaum übernommen wird oder während der Transformation ausgeführt wird, ist abhängig vom Schema, auf das zugegriffen wird. Zugriffe auf das DSX müssen während der Transformation erfolgen, Zugriffe auf Benutzerdaten sind Teil des Zielbaumes.

### 4.4.7 Anpassungssprache

Die Anpassungssprache hat die gleiche Syntax wie die Transformationssprache. Es dürfen jedoch keine Methoden definiert werden. Weiterhin gibt es keinen Unterschied zwischen direktem Zugriff auf Kindknoten und einem Zugriff über einen Methodenaufruf, da die Vorgabe-Methoden die Originalsicht auf den Quellbaum bieten.

## 4.5 Die Installationsumgebung

Die Installationsumgebung dient zum Einsetzen von Sprachmodulen in ein bereits in der Datenbank vorhandenes X-Translate-System. Auf die Basisinstallation des X-Translate-Systems wird nicht näher eingegangen, da hierfür lediglich einige Daten in die Datenbank geladen werden müssen und sich die Vorgehensweise von Datenbank zu Datenbank unterscheidet.

### 4.5.1 Das Installationsprogramm

Das Installationsprogramm ist die Schnittstelle zwischen Administrator und allen anderen Komponenten in der Installationsumgebung. Im einfachsten Fall erhält das Programm als Eingabe das vom Packager generierte Archiv und ruft Compiler-Generator, Parser-Generator und Metamodell-Installateur auf.

Bei der Installation werden Parser, Compiler und teilweise das ISX neu erzeugt. Beim Parser und Compiler geschieht das aus Performanzgründen. Beim ISX hat es einen anderen Grund: Da SQL-Sichten nicht erweitert werden können, müssen alle zu erweiternden Sichten zuerst gelöscht und dann neu angelegt werden.

Die Installation erfolgt in einer Transaktion, um sicherzustellen, das kein Sprachmodul nur teilweise installiert wird und es zu Inkonsistenzen kommt.

Bei einem Fehler wird die Installation unterbrochen. *Fehler* bedeutet hier Übersetzungsfehler oder Parsefehler, und nicht, dass ein Konflikt zwischen Sprachmodulen entdeckt wurde. In einem solchen Fall muss der Fehler erst behoben werden, um mit der Installation fortzufahren. Fehler sollten nur während der Entwicklung eines Sprachmoduls auftreten.

Wenn ein Konflikt entdeckt wird, wird die Installation soweit möglich fortgesetzt, um weitere Konflikte zu finden. Die Installation wird jedoch nicht vollendet, sondern der Administrator erhält am Ende eine Liste mit allen Konfliktbeschreibungen.

Es kann Konflikte geben, die nicht automatisch bei der Installation entdeckt werden. Das Installationsprogramm stellt deshalb auch Möglichkeiten zur manuellen Prüfung durch den Administrator bereit. Im einfachsten Fall kann das ein einfaches Programm zur Anzeige von Quellcode sein.

Abschnitt 4.5.2 beschreibt die Möglichkeiten, die ein Administrator hat, um Konflikte zu behandeln. In den Abschnitten 4.5.3 bis 4.5.5 werden die Komponenten der Installationsumgebung vorgestellt und die dazugehörigen Konflikte besprochen.

### 4.5.2 Auflösung von Konflikten

Zur Auflösung eines Konflikts hat der Administrator im Allgemeinen mehrere Möglichkeiten. Nicht alle sind in jedem Fall anwendbar.

- Akzeptieren des Konflikts. Ein einfaches Akzeptieren eines Konflikts kann sinnvoll sein bei *Scheinkonflikten*. Scheinkonflikte sind von X-Translate erkannte Konflikte, die tatsächlich keine sind. Beispielsweise kann es passieren, dass sich zwei Methodenklassen theoretisch überlappen können, dies aber aufgrund

von nicht in der Syntax erfassten Regeln niemals passiert. Das Akzeptieren von echten Konflikten führt zu einer nicht determinierten Arbeitsweise des X-Translate-Systems bei Auftreten des Konflikts.

- Ändern des Sprachmoduls an der konfliktären Stelle. Hierdurch ist das geänderte Sprachmodul möglicherweise nicht mehr Kompatibel zum Original. Schemas, die das geänderte Sprachmodul benutzen, sind nicht mehr portabel. Das muss nicht immer der Fall sein; bei schlecht implementierten Sprachmodulen kann der Administrator beispielsweise durch genauere Spezifizierung von Knotenmustern einen Konflikt lösen, ohne das sich Syntax und Semantik der Spracherweiterung ändert.

- Zusammenführen von Sprachmodulen an der konfliktären Stelle. Eine solche Vorgehensweise ist nötig, wenn zwei Sprachmodule Transformationen mit überlappendem oder gleichem Knotenmuster definieren. Der Administrator kann die beiden Transformationen zu einer einzigen zusammen führen.

- Deaktivieren der konfliktären Stelle. Falls der den Konflikt verursachende Teil des neuen Sprachmoduls nicht unbedingt gebraucht wird, kann er deaktiviert werden. Dadurch kann möglicherweise die Spracherweiterung unbrauchbar werden und sich unerwartet verhalten.

- Abweisen des neuen Sprachmoduls. Als letzte Möglichkeit bleibt der Abbruch der Installation, das ist immer möglich. In diesem Fall bleibt die vor Installationsbeginn bestehende Konfiguration des X-Translate-Systems erhalten.

### 4.5.3 Der Parser-Generator

Aus der Gesamtheit aller BNF-Syntaxregeln wird ein Parser für XSQL generiert. Eingabe für den Parser sind die SQL-Anweisungen vom Benutzer, die Ausgabe ist im Regelfall ein Syntaxbaum, der anschließend transformiert wird.

Auf die genaue Vorgehensweise des Parser-Generators und die Funktionsweise des Parsers wird hier nicht näher eingegangen. Es gibt mehrere frei verfügbare Parser-Generatoren [LMB92, SM00], die als Beispiel für eine X-Translate-Implementierung eines Parser-Generators dienen können.

Mehrdeutigkeiten in einem Sprachmodul sind ein Fehler, und das Sprachmodul wird zurückgewiesen. Mehrdeutigkeiten zwischen zwei Sprachmodulen oder doppelte Terminalsymbole sind ein Konflikt, der vom Administrator behandelt werden kann. Eine weitere Konfliktmöglichkeit, die doppelte Verwendung von Nonterminalsymbolen wird vermieden durch ein Präfix, welches für jedes Sprachmodul vergeben wird.

#### Mehrdeutigkeiten in der Grammatik

Bei Mehrdeutigkeiten in der Grammatik kann der Administrator die Grammatik ändern oder die betreffende Produktion deaktivieren. Bei der Änderung der Grammatik müssen möglicherweise auch Methoden und Transformationen geändert werden, da diese von der Struktur des Quellbaumes abhängen.

#### Doppelte Terminalsymbole

Wenn Terminalsymbole doppelt vergeben werden, kann das ein Scheinkonflikt oder eine Mehrdeutigkeit sein. Im Falle eines Scheinkonflikts kann der Konflikt akzeptiert werden; im Falle einer Mehrdeutigkeit kann diese durch Umbenennen des Terminalsymbols aufgelöst werden. Parser und Compiler sorgen dafür, dass an allen anderen relevanten Stellen, wie in Knotenmustern, der Name ebenfalls geändert wird.

### 4.5.4 Der Compiler-Generator

Der Compiler-Generator generiert aus den Methoden und Transformationen den XSQL-Compiler und den Anpassungscompiler. Der Anpassungscompiler arbeitet ähnlich wie der XSQL-Compiler. Der Quellbaum-Zugriff geschieht jedoch direkt und nicht über Methoden, bzw. nur über die Vorgabe-Methoden. Da nur eine Herstelleranpassung gleichzeitig genutzt wird, kann es auch keine Konflikte geben. Somit kann der Anpassungscompiler als im XSQL-Compiler enthalten angenommen werden. Wir konzentrieren uns deshalb im Folgenden auf die Implementierung des XSQL-Compilers.

Für die Generierung des Compilers wird in Abschnitt 4.6 ein Ansatz vorgestellt.

Die Generierung erfolgt in zwei Schritten: zuerst werden die Sprachmodule auf Konflikte geprüft. Möglicherweise greift dann der Administrator ein um die Konflikte zu behandeln. Danach wird der Compiler generiert.

#### Überlappende Methodenklassen

Wenn sich die Klassen zweier Methoden überlappen, kann zur Laufzeit nicht entschieden werden, welche Methode aufzurufen ist. Neben der Methodenklasse kann das auch vom Aufrufsknotenmuster abhängig sein.

Das Problem ist möglicherweise ein Scheinkonflikt. Dann kann der Administrator den Konflikt akzeptieren. Falls eine Überlappung aber praktisch auftreten kann, können die die Klasse definierenden Knotenmuster verfeinert werden, bis es keine Überlappung mehr geben kann. In selteneren Fällen kann das Hinzufügen oder Ändern von Aufrufsknotenmustern den Konflikt lösen. Als letzte Möglichkeit bleibt das Zusammenführen der konfliktären Methoden. Das ist im Allgemeinen aber nur möglich, wenn beide Methoden identisch sind.

#### Überlappende Knotenmuster bei Änderungstransformationen

Das gleiche Problem wie mit Methoden kann bei Änderungstransformationen auftreten. Die anderen Transformationstypen sind nicht betroffen, da davon beliebig viele pro Knoten ausgeführt werden können.

Hier kann analog zu den überlappenden Klassen bei Methoden vorgegangen werden. Allerdings ist das Zusammenführen von Transformationen praktikabler als bei Methoden.

#### Verbotene Knoten

Änderungstransformationen arbeiten auf dem Subjektknoten ihres Knotenmusters sowie möglicherweise auf dessen Kindknoten. Falls auf einen Kindknoten direkt, d. h.

nicht über einen Methodenaufruf, zugegriffen wird, ist dieser Knoten für Transformationen anderer Sprachmodule nicht mehr zugänglich. Weiterhin kann eine Transformation einen Kindknoten nicht in den Zielbaum übernehmen, wodurch eine Transformation desselben nicht mehr in Frage kommt. Diese Knoten werden *verbotene Knoten* genannt.

Wenn eine Transformation auf einen verbotenen Knoten zugreift, kann der Administrator das akzeptieren oder die Transformation, die den Knoten sperrt, ändern. Ein Akzeptieren des Konflikts ist wiederum nur bei einem Scheinkonflikt sinnvoll. In allen anderen Fällen sollte das Sperrmuster der den Knoten sperrenden Transformation verfeinert werden.

### 4.5.5 Der Metamodell-Installateur

Konflikte, die im Metamodell auftreten können, beschränken sich auf doppelte Namensvergabe von Spalten- und Sichtennamen. Hier ist die Nutzung eines Präfixes für jedes Sprachmodul nicht möglich, da das Metamodell für den Benutzer sichtbar ist.

Die einzige Möglichkeit, diesen Konflikt zu beheben, ist das Umbenennen der Spalten und Sichten. X-Translate ändert automatisch alle Referenzen auf die umbenannten Elemente.

Im Fall, dass sich Spalten auch im Typ bzw. Sichten in ihrer kompletten Struktur gleichen, kann auch ein Zusammenführen der betreffenden Spalten- und Sichtendefinitionen sinnvoll sein.

## 4.6 Generierung des Compilers

Um aus den in der Transformationssprache definierten Methoden und Transformationen einen Compiler zu generieren wird Java verwendet. Die Syntax der Transformationssprache wurde an Java angelehnt, um die Übersetzung zu vereinfachen.

Die Übersetzung der Transformationssprache nach Java-Bytecode erfolgt in drei Schritten. Um Aufwand zu sparen, wird die SQLJ-Syntax [ISO00], mit der aus Methoden und Transformationen auf die Datenbank zugegriffen werden kann, in den Java-Quellcode übernommen. Der resultierende Javacode mit SQLJ-Einschlüssen wird dann durch einen SQLJ-Präprozessor geleitet, um dann im Anschluss in Bytecode kompiliert zu werden.

Kommentar 4.4:

Für den Benutzer ist es nur möglich, den JDBC-Treiber direkt zu verwenden, da SQLJ-Präprozessoren kein XSQL verstehen. Hierzu müsste ein XSQL-SQLJ-Prozessor entwickelt werden.

Die Transformationssprache hat, wie in Abschnitt 4.1.5 erläutert, viele objektoriente Konzepte. Diese lassen sich jedoch nicht 1:1 nach Java übersetzen; eine X-Translate-Methode oder eine X-Translate-Klasse kann nicht in eine Java-Methode oder eine Java-Klasse übersetzt werden, und für Transformationen gibt es kein gleichlautendes

Konstrukt in Java. In diesem Abschnitt wird für die Elemente der Transformationssprache gezeigt, wie sie in Java realisiert werden können.

Um Verwirrung zu vermeiden, werden Methoden im Java-Sinne als Java-Methoden bezeichnet, und Methoden der Transformationssprache als X-Translate-Methoden.

Der Beispielcode in diesem Abschnitt ist stark vereinfacht. Die Mechanismen, um Fehler und Konflikte zu behandeln, sind komplett weggelassen worden, und nur einige wichtige Methoden werden gezeigt. Der tatsächlich generierte Code ist um ein vielfaches komplexer und unübersichtlicher, da er ja nicht von Menschen gelesen werden muss.

### 4.6.1 Methoden und Knoten

Knoten können mit einer einfachen Knotenklasse, die Referenzen auf Eltern- und Kindknoten hat, realisiert werden.

Im Quellbaum muss eine etwas kompliziertere Struktur angewendet werden, da der Zugriff über Methoden erfolgt. Hierzu muss die verwendete Knotenklasse eine `getSelf()`-Methode bereitstellen, die die passende Methode der Transformationssprache aufruft und einen virtuellen Knoten zurück gibt. In Beispiel 4.10 wird das veranschaulicht.

In den Zeilen 1-13 des Beispiels ist die Standard-Knotenklasse zu sehen. Die Zeilen 4-6 zeigen die Anwendung der `getSelf()`-Methode. In der Standard-Knotenklasse hat sie keine Funktion, da die Standard-X-Translate-Methoden nichts tun. Die `matches`-Methode in der Zeile 8 prüft, ob das übergebene Knotenmuster auf den Methodenknoten passt. In Zeile 9 gibt es eine Methode, die die Spracherweiterung des Knotens zurück gibt, und die Methode in Zeile 10 prüft, ob der übergebene Knoten ein Nachfahre des Methodenknotens ist.

Für jeden Knotentyp, d. h. jedes Symbol der Grammatik, wird eine Java-Klasse `Knoten_Typ_Symbolname` erzeugt. In den Zeilen 15-45 ist ein Ausschnitt einer Klasse für den fiktiven Knotentyp $X$ angezeigt. Die Klasse erweitert die Basis-Knotenklasse (Zeile 15) und kann somit dessen Methoden benutzen.

In den Zeilen 18-33 wird die `getSelf()`-Methode überschrieben. Je nach Klasse des Methodenknotens, Position des aktuellen Knotens, Spracherweiterung des aktuellen Knotens und Aufrufsmuster wird eine X-Translate-Methode aufgerufen, die einen Knoten gemäß der Definition in der Transformationssprache zurückliefert. Wenn in Zeile 19 festgestellt wird, dass der Methodenknoten die Klasse mit dem Index 0 hat und der aktuelle Knoten ein Nachfahre des Methodenknotens ist, wird in Zeile 21 die Methode `up_0_null_null()` aufgerufen. Der Name wird automatisch generiert und setzt sich aus der Methodenrichtung (up/down), dem Index der Klasse, dem Index des Aufrufsmusters und der Spracherweiterung zusammen. In den Zeilen 23-27 und 28-32 wird jeweils für eine andere Methode geprüft, ob sie aufgerufen werden soll.

Die Zeilen 35-42 enthalten ein Beispiel einer nach Java übersetzten X-Translate-Methode. In Zeile 36 wird ein neuer Knoten $k$ erzeugt. Da es eine Abwärts-Methode ist, darf der Elternknoten nicht verändert werden. Deshalb wird in Zeile 37 der Elternknoten des Methodenknotens übernommen. In den Zeilen 38-40 werden die

Beispiel 4.10: Codeausschnitt der Java-Klassen

```
 1  public class Knoten {
 2      // ...
 3
 4      public Knoten getParent() { return parent.getSelf(); }
 5      public Knoten getChild(int i) { return child[i].getSelf(); }
 6      protected Knoten getSelf() { return this; }
 7
 8      public boolean matches(Knotenmuster m) { /* ... */ }
 9      public String getSpracherweiterung() { /* ... */ }
10      public boolean isBelow(Knoten k) { /* ... */ }
11
12      public static Knoten getAktuellerKnoten() { /* ... */ }
13  }
14
15  public class Knoten_Typ_X extends Knoten {
16      Knotenmuster[] klassen;
17
18      protected Knoten getSelf() {
19          if (matches(klassen[0])
20                && getAktuellerKnoten().isBelow(this)) {
21              return up_0_null_null();
22          }
23          else if (matches(klassen[1]) &&
24              && !getAktuellerKnoten().isBelow(this)
25              && getAktuellerKnoten().getSpracherweiterung = "mv"){
26              return down_1_null_mv();
27          }
28          else if (matches(klassen[1]
29              && getAktuellerKnoten().isBelow(this)
30              && getAktuellerKnoten.matches(aufrufsMuster[2]))) {
31              return up_1_2_null();
32          }
33      }
34
35      private Knoten down_1_null_mv() {
36          Knoten k = new Knoten();
37          k.setParent(getParent());
38          k.addChild(getChild(1));
39          // 2. Kindknoten wird ausgelassen.
40          k.addChild(getChild(3));
41          return k;
42      }
43
44  // ...
45  }
```

Kindknoten gesetzt. Ein Kindknoten wird ausgelassen. Zu beachten ist, dass die Kindknoten nicht einfach in den neuen Knoten eingehängt werden, sondern dass sie dorthin *kopiert* werden. In der `addChild()` Methode wird der Elternknoten der hinzugefügten Knoten automatisch angepasst.

Neben den im Beispiel ersichtlichen hat die Knotenklasse noch mehr Attribute und Java-Methoden, beispielsweise den Knotentyp. Weiterhin gibt es noch Methoden zum direkten Zugriff auf Kindknoten, wie `getChildDirect(int)`.

### 4.6.2 Knotenmuster

Die Java-Klasse für Knotenmuster kann als eine Subklasse von `Knoten` realisiert werden, die als Typ der Knoten auch Wildcards akzeptiert.

### 4.6.3 Matching von Knotenmustern

Um zu testen, ob ein Knoten in ein bestimmtes Knotenmuster passt, muss vom fraglichen Knoten aus schrittweise zu seinen Vor- und Nachfahren gegangen werden und jeweils geprüft werden, ob diese den Vor- und Nachfahren des Subjektknotens gleichen. Etwas komplizierter wird es, wenn im Knotenmuster Wildcards benutzt werden bzw. wenn es mehrere Möglichkeiten der Übereinstimmung gibt. Dann müssen in jedem Schritt alle Kombinationen getestet werden, solange, bis alle Knoten im Knotenmuster gefunden werden konnten oder bis eine Kombination nicht mehr passt.

### 4.6.4 Transformationen

Transformationen werden ähnlich implementiert wie Methoden. Für jeden Knotentyp gibt es eine Java-Klasse, die von der Standard-Transformationsklasse erbt. Das Gerüst für eine Transformation ist in Beispiel 4.11 dargestellt.

Die Zeilen 1-8 enthält die Standard-Transformationsklasse. Die `transform()`-Methode packt den übergebenen Knoten in ein Hilfsobjekt vom Typ `TResult` und gibt dieses zurück. Die Hilfsklasse `TResult` ist in den Zeilen 9-13 definiert. Da Transformationen mehrere Rückgabewerte haben können, Java aber nur einen Rückgabewert erlaubt, ist diese Klasse nötig.

Die Zeilen 15-44 enthalten ein Gerüst für eine Transformationsklasse für einen fiktiven Knotentyp `X`. In den Zeilen 17-32 wird die `transform()`-Methode überschrieben. Sie ruft für jeden Transformationstyp die X-Translate-Transformationen auf. Zuerst werden die lesenden Transformationen ausgeführt (Zeilen 20-21). Die Methodennamen für die Transformationen werden automatisch generiert, `rt_0` steht für die erste Transformation, `rt_1` für die zweite usw. In den Zeilen 24-25 werden die Knotenmuster der Ergänzungstransformationen getestet und die Transformationen bei einem Match ausgeführt. In den Zeilen 28-29 wird die Änderungstransformation für den Knoten ausgeführt. Im Unterschied zu den vorigen Anweisungen wird hier eine `if ... else if ...`-Konstruktion verwendet, da es maximal eine Änderungstransformation geben darf.

In den Zeilen 34-43 steht ein Beispiel für die Java-Implementierung einer kombinierten Änderungs- und Ergänzungstransformation. Die Implementierungen für die

Beispiel 4.11: Codeausschnitt der Java-Klassen

```
 1  public class Transformation {
 2
 3      public TResult transform(Knoten k) {
 4          TResult r = new TResult();
 5          r.knoten = k;
 6          return r;
 7      }
 8  }
 9  public class TResult {
10      Knoten knoten = null;
11      Knoten[] before = null;
12      Knoten[] after = null;
13  }
14
15  public class Transformation_Typ_X extends Transformation {
16
17      public TResult transform(Knoten k) {
18          TResult r = new TResult();
19          // Lesende Transformationen ausführen
20          if (k.matches(rMuster[0])) { rt_0(k); }
21          if (k.matches(rMuster[1])) { rt_1(k); }
22
23          // Ergänzungstransformationen ausführen
24          if (k.matches(aMuster[0])) { at_0(k, r); }
25          if (k.matches(aMuster[1])) { at_1(k, r); }
26
27          // Änderungstransformationen ausführen
28          if (k.matches(aMuster[0])) { ct_0(k, r); }
29          else if (k.matches(aMuster[1])) { ct_1(k, r); }
30
31          return r;
32      }
33
34      private void ct_0(Knoten k ,TResult r) {
35          // kombinierte Änderungs- und Ergänzungstransformation
36          // ...
37          r.k = kStrich;
38          r.before = new Knoten[0];
39          r.after = new Knoten[1];
40          Knoten updStmt = new Knoten("update_statement_searched")
41          // ...
42          r.after[0] = updStmt;
43      }
44  }
```

anderen Transformationstypen sehen ähnlich aus. Die Anweisung in Zeile 37 weist dem Hilfsobjekt den zuvor veränderten aktuellen Knoten zu. Die Zeilen 38-39 initialisieren die Knotenlisten, die vor und nach dem aktuellen Knoten eingefügt werden. Vorher wird kein Knoten eingefügt (Zeile 38), und nach dem aktuellen Knoten wird noch ein weiterer Knoten eingefügt (Zeile 39). Wie in den Zeilen 40-42 ersichtlich, ist dieser eingefügte Knoten ein <update statement searched> aus der SQL:1999-Syntax.

## 4.7 Zusammenfassung

In diesem Kapitel wurden Konzepte, Architektur und Komponenten des X-Translate-Systems besprochen. Eine Implementierung des XSQL-Compilers in Java wurde diskutiert.

Mit dem vorgestellten System lassen sich mehrere unabhängig entwickelte Sprachmodule gleichzeitig einsetzen. Dazu müssen möglicherweise bei der Installation eines Sprachmoduls verschiedene Konflikte aufgelöst werden.

Die Anforderungen aus Kapitel 3 wurden weitgehend erfüllt. Die nicht erfüllten Anweisungen werden im Folgenden aufgeführt.

- Nicht erfüllt wurde Anforderung **F1d**, die Definition erweiterungsspezifischer Fehler- und Statusmeldungen.
- Anforderung **F11** (die Bedeutung existierender Sprachkonstrukte darf sich nicht verändern), ist nur schwer erfüllbar. X-Translate kann im Allgemeinen nicht feststellen, ob eine Operation in den Transformationen die SQL:1999-Semantik ändert.
- Die Erfüllung der Anforderung **P1** (die Performanz von SQL-Anweisungen darf nicht wesentlich beeinträchtigt werden) hängt von den jeweiligen Sprachmodulen ab. Vor allem bei hoher Last wird sich der Perfomanzverlust bemerkbar machen, da zusätzlich Speicher und Rechenkapazität für X-Translate benötigt wird.

Die Grenzen von X-Translate liegen in der syntaktischen Erweiterbarkeit von SQL:1999 und in der Begrenzung auf eine Änderungstransformation pro Knoten.

# Kapitel 5 Alternative Lösungsansätze und verwandte Arbeiten

In diesem Kapitel werden mit X-Translate verwandte Arbeiten diskutiert.

## 5.1 Metaprogrammierung

Mit *Metaprogrammen* sind im Allgemeinen Programme, die sich selbst oder andere Programme verändern oder generieren, gemeint. Die *Metaprogrammierung* beschäftigt sich mit der Entwicklung von Metaprogrammen. Compiler und Generatoren sind Beispiele für Metaprogramme. Sie generieren aus Quellcode oder einer Spezifikation ein lauffähiges Programm oder eine Komponente. Auch der Parser-Generator und der Compiler-Generator sind Metaprogramme. Das X-Translate-System ist eine Umgebung zur Metaprogrammierung, mit der Transformationssprache kann der X-Translate-Compiler erweitert werden.

Es gibt Sprachen mit eingebauten Metaprogrammierung-Eigenschaften, wie beispielsweise Smalltalk [GR89]. Dort können während der Laufzeit eines Programms neue Klassen erzeugt oder schon bestehende Klassen und Objekte verändert werden. Die Veränderungen beschränken sich nicht auf Zustandsänderungen; ein Objekt kann neue Attribute und Methoden erhalten oder seine Klassenzugehörigkeit ändern. Hier besteht die Gefahr, das Programmierung und Metaprogrammierung bei der Entwicklung einer Anwendung vermischt werden [CE00], was die Verständlichkeit des Quellcodes beeinträchtigen kann.

## 5.2 Aktive Bibliotheken

*Aktive Bibliotheken* (engl. active libraries) [CEG+00] stellen im Gegensatz zu herkömmlichen Bibliotheken nicht nur Funktionen oder Klassen bereit, sondern partizipieren auch in der Code-Generierung. Sie stellen Sprachabstraktionen bereit, die sie selbst optimieren können. Möglicherweise prüfen sie den Quellcode auf Korrektheit und stellen Methoden zum Debugging bereit. Auch Aufgaben wie Anzeige von Code-Teilen und Eingabehilfen können von aktiven Bibliotheken bereitgestellt werden.

Es gibt mehrere Stufen von aktiven Bibliotheken, je nach den von der Bibliothek übernommenen Aufgaben [CEG+00].

1. Erweiterung des Compilers. Aktive Bibliotheken können den Compiler erweitern, in dem sie für die bereitgestellten Sprachabstraktionen auch die Optimierung des Codes vornehmen. Blitz++ [Ve98] ist ein Beispiel dafür. Blitz++

bietet verschiedene optimierte Operationen für Array-Objekte und generiert je nach Zielplattform unterschiedlichen Code.

2. Domänenspezifische Werkzeugunterstützung. Aktive Bibliotheken können die Entwicklungsumgebung um domänenspezifische Werkzeuge zum Debugging, zur Analyse und ähnlichem erweitern.

3. Erweiterte Metaprogrammierung. Aktive Bibliotheken können Metacode zur Kompilierung, Optimierung, Debugging, Analyse, Visualisierung und Eingabe von domänenspezifischen Abstraktionen enthalten.

Ein Beispiel für eine Entwicklungsumgebung, die sowohl domänenspezifische Werkzeuge sowie erweiterte Metaprogrammierung unterstützt ist das in Abschnitt 5.3 vorgestellte IP.

Aktive Bibliotheken werden als miteinander interagierende, intelligente (engl. knowledgeable) Agenten beschrieben, die die konventionelle Sichtweise auf Compiler, Bibliotheken und Anwendungen redefinieren [CEG+00]. Zur Interaktion untereinander und mit dem Entwickler brauchen sie eine passende Infrastruktur. Eine solche Infrastruktur liefert IP.

## 5.3 Intentional Programming

*Intentional Programming* (Abk. IP) [Si95, Si96, Si99, WMS+01, CE00] ist eine Entwicklungsumgebung zur Programmierung von Anwendungen und zur Metaprogrammierung und ein früheres Forschungsprojekt von Microsoft Research.

Ziel von IP ist es, die in Programmiersprachen verwendeten Abstraktionen von den Sprachen unabhängig zu machen. Zur Zeit braucht eine Abstraktion eine Sprache als Träger, um sich zu verbreiten. Neue Sprachen verbreiten sich jedoch langsam, und in einer neuen Sprache finden sich neben sinnvollen, neuen Abstraktionen oft auch Fehler, die im Nachhinein nicht mehr zu korrigieren sind. Will man die Abstraktionen nutzen, muss man sich auch mit den Problemen der Programmiersprache herumschlagen.

Wenn Abstraktionen unabhängig von Programmiersprachen existierten, würden sie sich viel schneller verbreiten, und somit könnte die Forschung auf diesem Gebiet viel schneller voran schreiten. IP bietet diese Möglichkeit: es können beliebige neue Abstraktionen in das System geladen werden und dann genau wie die schon vorhandenen genutzt werden.

Für die Anwendungsprogrammierung können Erweiterungsbibliotheken in die Umgebung geladen werden, die allgemeine und domänenspezifische Spracherweiterungen enthalten. Diese Bibliotheken sind eine Form der in Abschnitt 5.2 beschriebenen aktiven Bibliotheken, die Compiler, Debugger, Editor, Versionierung usw. erweitern können. Diese Erweiterungsbibliotheken können mit IP selbst entwickelt werden, IP bietet hierfür eine adäquate Umgebung.

Eine der Hauptideen von IP ist die Abwendung von der konventionellen Repräsentierung von Quellcode als einfachem Text. In IP wird der Quellcode als *aktiver*

*Quellcode* (engl. active source), d.h. als einen Graph mit Verhalten schon zur Zeit der Programmierung, dargestellt. Das Verhalten des Quellcode wird von Methoden, die auf dem Quellgraph operieren, realisiert.

Anwendungsprogramme werden geschrieben, in dem verschiedene Sprachabstraktionen in das System geladen werden. Es können beispielsweise Java-Abstraktionen geladen werden, um Java-Code zu schreiben. Die Sprachabstraktionen in IP werden Intentionen genannt. Es können jederzeit neue Intentionen entwickelt werden, in dem sie deklariert und ihre Methoden geschrieben werden. Die Methoden werden in eine Erweiterungsbibliothek gepackt und sind dann als Spracherweiterung nutzbar. Es gibt Methoden zur Reduktion, d.h. Kompilierung von Code, Anzeige, Editieren, Debugging usw.

Durch die Methoden können auch domänenspezifische Notationen für Intentionen bereitgestellt werden, in dem spezielle Anzeige- und Eingabemethoden geschrieben werden. So wird kann beispielsweise eine echte zweidimensionale Notation für mathematische Formeln mit unteren und oberen Indizes, Matrizen usw. verwendet werden. Das erhöht die *Intentionalität* des Quellcodes; was in einer konventionellen Programmiersprache als Kommentar gefasst werden muss, kann in IP als Intention auf einer höheren Stufe direkt im Quellcode ausgedrückt werden. Das reduziert die Fehleranfälligkeit und erhöht die Verständlichkeit des Quellcodes.

Um den Umstieg von Programmiersprachen wie C, C++ oder Java auf IP zu erleichtern, gibt es Erweiterungsbibliotheken, die entsprechende Intentionen und Methoden anbieten. So kann mit IP beispielsweise Java-Code geschrieben werden. Intern wird weiterhin der Quellgraph genutzt. Die Anzeigemethoden stellen den Quellgraphen als Java-Code dar, und die Eingabemethoden akzeptieren nur Java-Code zur Eingabe der Intentionen. Schrittweise kann das IP-System durch neue Intentionen erweitert werden.

Um Legacy-Code weiterzuverwenden, werden Import-Parser eingesetzt. Diese Parsen ein Programm in C, C++ oder einer anderen Programmiersprache und erzeugen daraus einen IP-Quellgraphen. In Verbindung mit der entsprechenden Erweiterungsbibliothek kann somit ein schrittweiser Umstieg auf IP erfolgen.

Um aus dem Quellgraphen Code zu generieren, wird die IP-Reduktionskomponente benutzt. Die Kompilierung eines IP-Programms erfolgt durch den Aufruf der von den Intentionen bereitgestellten Reduktionsmethoden. Eine Reduktionsmethode transformiert eine Intention in eine einfachere Form. Das kann mehrmals iteriert werden, so das eine Spracherweiterung eine andere nutzen kann, um sich selbst zu implementieren. Dies ist in X-Translate nicht möglich. Im letzten Reduktionsschritt entsteht ein reduzierter Code, oder *R-Code*, eine maschinennahen Sprache, die von einem Compiler in die Maschinensprache der Zielplattform kompiliert werden kann. Reduktionsmethoden fügen den generierten Code in den Quellbaum ein, dieser wächst also monoton, bis für jede Intention R-Code erzeugt wurde. Auch hier lässt sich ein Unterschied zu X-Translate ausmachen: in X-Translate wird ein neuer Zielbaum erzeugt, und nicht der Quellbaum verändert. Das ist nötig, weil beim X-Translate-System der Quellbaum einem Wort einer XSQL-Grammatik entspricht, was durch Hinzufügen von neuen Knoten ungültig würde.

IP ist eine interessante Technologie, die die Intentionalität von Programmen erhöht und dadurch deren Verständlichkeit. Verständliche Programme lassen sich besser

warten, neue Entwickler benötigen weniger Einarbeitungszeit und Fehler werden weniger wahrscheinlich, da lediglich ausgedrückt werden muss *was* zu tun ist, und nicht mehr so sehr *wie* es zu tun ist.

## 5.4 Wrapper

X-Translate ist nicht nur ein System, um Spracherweiterungen für SQL:1999 anzubieten. Es kann auch als System zur Anpassung von SQL-Dialekten genutzt werden. Dadurch können einerseits sinnvolle Spracherweiterungen von einzelnen Herstellern allgemein nutzbar gemacht werden, und andererseits können Anwendungen mit einem einheitlichen Befehlssatz auf unterschiedliche DBVS zugreifen.

Ein verwandtes Problem ist die heterogene Struktur von Datenbankschemas. In größeren Firmen liegen Daten meistens auf mehreren verschiedenen Datenhaltungssystemen verteilt. Teilweise werden Informationen dupliziert. Um diese heterogene Struktur vor den Anwendungen zu verbergen gibt es *Wrapper*.

Wrapper werden vor allem genutzt, um auf Legacy-Systeme zuzugreifen. Wrapper bieten eine einheitliche Schnittstelle, z.B. SQL, um auf ein System zuzugreifen. So können beliebige Datenhaltungssysteme (hierarchische Datenbanken, Dateisysteme, usw.) mit einer Schnittstelle angesprochen werden.

*Garlic* [RS97, Ca+95] ist Beispiel für ein Middleware-System zum Einsatz von Wrappern. Mit Garlic kann durch den Einsatz von Wrappern eine einheitliche Sicht auf mehrere heterogene Datenquellen realisiert werden.

Garlic besteht aus einer Middleware-Komponente, an die Anfragen in einem erweiterten SQL gestellt werden können. Diese Anfragen können sich über mehrere Datenquellen erstrecken. Das System verteilt die Anfrage auf die Wrapper für die beteiligten Datenquellen. Die Wrapper teilen mit, welche Operationen von ihm unterstützt werden. So kann das Garlic-System die Anfrage optimieren, in dem die Datenquellen selbst möglichst viel der Arbeit unternehmen. Nicht unterstützte Operationen muss Garlic selbst ausführen.

Ein Wrapper hat vier Hauptaufgaben [RS97].

- Er modelliert die Inhalte der darunterliegenden Datenquelle als Objekte.
- Er ermöglicht das Ausführen von Methoden zum Zugriff auf die Attribute der Objekte.
- Er nimmt an der Anfrageplanung (engl. query planning) teil. Garlic teilt dem Wrapper den für ihn relevanten Teil der Anfrage mit und der Wrapper antwortet, welchen Teil der Arbeit er übernehmen kann.
- Bei der Anfrageausführung ist der Wrapper für die Ausführung der während der Anfrageplanung festgelegten Operationen verantwortlich.

Wrapper für Garlic können sehr schnell entwickelt werden [RS97]. Sie können inkrementell entwickelt werden, indem in einer ersten Version lediglich Basisfunktionalitäten implementiert und später auch die speziellen von der Datenquelle angebotenen Operationen unterstützt werden.

## 5.5 SQL-Templates

Marder, Ritter und Steiert [MRS99, RS00] stellen im Rahmen des SENSOR[1]-Projektes einen Ansatz zum automatischen Testen von OCL[2]-Bedingungen (engl. OCL constraints) vor. OCL ist eine Sprache, um Bedingungen (Invarianten, Prä- und Postkonditionen) in UML-Modellen formal zu beschreiben. In OCL spezifizierte Bedingungen werden auf SQL-Bedingungen abgebildet, um in der Datenbank ausgeführt zu werden.

Die Abbildung der OCL-Bedingungen auf SQL erfolgt automatisch. Nach dem Parsen der OCL-Bedingung wird eine Zwischendarstellung, der *Übersetzungsgraph* (engl. translation graph), erzeugt. Dieser Graph enthält zwei Arten von Knoten: Übersetzungsknoten und Metadatenknoten. Die Übersetzungsknoten enthalten die Algorithmen, die zur Übersetzung nach SQL notwendig sind, und die Metadatenknoten liefern die Daten dazu.

Die Übersetzungsknoten generieren SQL-Code, genannt *SQL-Templates*, der generische Parameter enthalten kann. Um die Parameter zu ersetzen, fragt der Übersetzungsknoten seine Kindknoten nach einem passenden SQL-Fragment. In Beispiel 5.1 ist ein SQL-Template mit generischen Parametern dargestellt.

Beispiel 5.1: SQL-Template mit generischen Parametern

```
NOT EXISTS (SELECT *
            FROM ($SetNode$) AS $VariableNode$
            WHERE NOT ($PredicateNode$))
```

Die Ersetzung erfolgt rekursiv, d. h. die von den Kindknoten gelieferten Fragmente können wiederum generische Parameter enthalten.

Die Arbeitsweise der Übersetzung von OCL nach SQL ähnelt der Arbeitsweise des X-Translate-Systems.

## 5.6 XSLT

XSLT [W3C02b] arbeitet auf der Basis von Vorlagen (engl. templates). Anhand eines Musters (ähnlich den in Kapitel 4 beschriebenen Knotenmustern) werden zur Vorlage passende Knoten im Quell-Dokument gefunden und anhand der in der Vorlage beschriebenen Regeln transformiert. Begonnen wird mit dem Wurzelknoten, auf den die am besten passende Vorlage angewendet wird. Danach wird rekursiv mit den Kindknoten fortgefahren. In der Vorlage kann spezifiziert werden, das nicht alle Kindknoten bearbeitet werden sollen. So kann eine XSLT-Vorlage aus einem Schriftstück ein Inhaltsverzeichnis generieren, in dem nur die Überschrift-Knoten abgearbeitet werden.

In den Vorlagen können auch die in Programmiersprachen üblichen Kontrollkonstrukte wie Iteration über Elemente und bedingte Anweisungen verwendet werden.

[1] Supporting Software Engineering Processes by Object-Relational Database Technology.
[2] Object Constraint Language [OMG97].

XSLT ist jedoch eine funktionale Sprache, d. h. Variablen kann nach deren Initialisierung kein neuer Wert zugewiesen werden. Der Begriff *Variable*, wie in prozeduralen Sprachen gebraucht, mag hier etwas irreführend erscheinen.

In XSLT können nicht nur Vorlagen, sondern auch Funktionen definiert werden. Im Unterschied zu Vorlagen können diese in Ausdrücken verwendet werden.

Beispiel 5.2 zeigt ein einfaches XSLT-Programm zur Transformation von XML-Daten in ein HTML-Format sowie die Ein- und Ausgabedatei dazu.

Die Eingabedatei ist eine XML-Adressenliste mit offensichtlichem Format. Das Wurzelelement ist `addressbook` und für jede Adresse gibt es ein `address`-Element. Aus Platzgründen hat unser Beispiel-Adressbuch nur einen Eintrag.

Das XSLT-Stylesheet beginnt in Zeile 1 mit der Deklaration der benutzten XML-Version. Die Zeilen 2-3 enthalten das Wurzelelement des Stylesheets. Dort wird der Namensraum für die XSLT-Befehle auf `xsl` festgelegt. In Zeile 4 wird als Ausgabemethode HTML gewählt. Da ein XSLT-Stylesheet ein XML-Dokument ist, muss es wohlgeformt sein, d. h. unter anderem, es muss für jeden Start-Tag auch ein Ende-Tag vorhanden sein. In HTML gilt das nicht für alle Elemente, beispielsweise hat `br` keinen Ende-Tag. Bei Wahl von HTML als Ausgabemethode wird deshalb aus einem `<br/>` ein `<br>`.

Die Zeilen 6-13 enthalten die erste Vorlage unseres Beispiels. Es wird auf den Wurzelknoten angewendet, und der Inhalt der Vorlage wird in die Ausgabe geschrieben. In Zeile 9 werden die Vorlagen für alle Kindknoten aufgerufen, und in Zeile 11 wird die Gesamtzahl der Einträge mittels der `count()`-Funktion ermittelt.

In den Zeilen 14-20 steht die Vorlage für das `address`-Element. Die Inhalte der Sub-Elemente `name`, `strasse` und `ort` werden mit dem `value-of`-Befehl ausgelesen und in die Ausgabe eingefügt.

In der Ausgabedatei steht das Ergebnis der Transformation. Hier fällt auf, das die Telefonnummer, die in der Eingabedatei angegeben ist, nicht mehr auftaucht. Das ist korrekt, da die Vorlage für das `address`-Element den Wert nicht ausliest.

Beispiel 5.3 zeigt weitere Elemente der Sprache. Das Eingabedokument kann aus einer Liste von Zahlen bestehen, von denen das XSLT-Programm im Beispiel jeweils die Fakultät berechnet und ausgibt.

In Zeile 4 wird ein weiterer Namensraum deklariert, das ist nötig, um später die Funktion zu definieren, da Funktionen immer in einem Namensraum liegen müssen. In Zeile 6 wählen wir diesmal kein HTML, sondern Text als Ausgabeformat. Die Vorlage in den Zeilen 8-13 ruft für jede angegebene Zahl im Eingabedokument die Fakultätsfunktion auf. In Zeile 9 steht das in XSLT vorhandene `for-each` Element, um über eine Menge von Elementen zu iterieren. Es gibt in XSLT keine `for` oder `while`-Schleifen, da Variablen ihren Wert nicht ändern können. Um solche Konstrukte zu simulieren, ist Rekursion nötig. Das unschön umgebrochene `value-of`-Element (Zeile 10-11) dient zur Verhinderung eines Zeilenumbruchs nach „von“. In Zeile 11 wird die Fakultätsfunktion aufgerufen. Diese in den Zeilen 15-23 definierte Funktion berechnet die Fakultät rekursiv. In Zeile 16 wird der einzige Parameter der Funktion deklariert, in den Zeilen 17-22 das Ergebnis berechnet. Die Zeilen 18-21 enthalten zwei sich gegenseitig ausschließende `if`-Anweisungen. Es gibt in XSLT kein `else`, lediglich eine Fallunterscheidung ähnlich der `switch`-Anweisung in Java.

Beispiel 5.2: XSLT-Stylesheet zur Generierung einer HTML-Datei

Eingabedatei:

```
<addressbook>
    <address>
        <name>Fritz Klein</name>
        <strasse>Musterstrasse 1</strasse>
        <ort>67655 Kaiserslautern</ort>
        <telefon>0631 1234567</telefon>
    </address>
</addressbook>
```

XSLT-Programm:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:transform
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">
    <xsl:output method="html"/>
    <xsl:template match="addressbook"/>
        <html><body>
            <h3>Adressbuch</h3><hr/>
            <xsl:apply-templates/>
            Gesamtzahl der Einträge:
                <xsl:value-of select="count(./address)"/>
        </body></html>
    </xsl:template>
    <xsl:template match="address">
        <p>
        Name: <xsl:value-of select="./name"/><br/>
        Strasse: <xsl:value-of select="./strasse"/><br/>
        PLZ/Ort: <xsl:value-of select="./ort"/>
        </p>
    </xsl:template>
</xsl:transform>
```

Ausgabedatei:

```
<html><body>
    <h3>Adressbuch</h3><hr>
    <p>
    Name: Fritz Klein<br>
    Strasse: Musterstrasse 1<br>
    Ort: 67655 Kaiserslautern
    </p>
    Gesamtzahl der Einträge: 1
</body></html>
```

Außer den in den Beispielen gezeigten gibt es noch einige weitere Sprachkonstrukte in XSLT. Es lassen sich mit XSLT beliebige Berechnungen durchführen, wobei die Stärke von XSLT in der Transformation von XML-Dokumenten und damit in der Transformation von Bäumen liegt.

Beispiel 5.3: XSLT-Stylesheet zur Generierung einer HTML-Datei

```
 1 <?xml version="1.0"?>
 2 <xsl:transform version="2.0"
 3         xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 4         xmlns:math="mathe-funktionen">
 5
 6     <xsl:output method="text"/>
 7
 8     <xsl:template match="/">
 9         <xsl:for-each select="root/number">
10             Fakultaet von <xsl:value-of select="."
11             /> ist <xsl:value-of select="math:fac(.)"/>
12         </xsl:for-each>
13     </xsl:template>
14
15     <xsl:function name="math:fac">
16         <xsl:param name="n"/>
17         <xsl:result>
18             <xsl:if test="$n = 1">1</xsl:if>
19             <xsl:if test="$n > 1">
20                 <xsl:value-of select="$n * math:fac($n - 1)"/>
21             </xsl:if>
22         </xsl:result>
23     </xsl:function>
24 </xsl:transform>
```

## 5.7 Abbildung von SQL:1999 auf Herstellerdialekte

Die Abbildung von SQL:1999 auf die unterschiedlichen SQL-Dialekte der DBVS-Hersteller ist Bestandteil des X-Translate-Systems. Mit den Anpassungstransformationen kann eine SQL:1999-Anweisung, möglicherweise mit herstellerspezifischen Erweiterungen, auf den Dialekt des DBVS-Herstellers abgebildet werden.

Kauhaus et al. [KLS02] stellen einen weiteren Ansatz zur Abbildung von SQL:1999 auf Herstellerdialekte vor. Als Basis wird kein reines SQL:1999 benutzt, sondern ein um eine Kollektionsunterstützung [Lu02] erweitertes. Das um Mengen, Multimengen und Listen (engl. sets, multisets, lists) sowie Operationen darauf erweiterte SQL:1999 wird SQL:1999+ genannt. Der Ansatz hat starke Ähnlichkeit mit der Anpassungssprache von X-Translate. Interessanterweise wird XSLT zur Implementierung des Anpassungscompiler-Äquivalentes verwendet. Der XSLT-Ansatz wird in dieser Arbeit abgelehnt (Abschnitt 5.8.3). Für die Übersetzung von SQL:1999+ in

Herstellerdialekte muss jedoch nicht auf Erweiterbarkeit geachtet werden, wie das bei X-Translate der Fall ist. Deshalb kann in diesem Fall XSLT durchaus eine sinnvolle Möglichkeit zur Implementierung des Compilers sein.

## 5.8 Bewertung der Alternativen

In diesem Kapitel wurden verschiedene mit X-Translate verwandte Arbeiten und mögliche alternative Lösungsansätze angesprochen. In diesem Abschnitt werden die Alternativen kurz bewertet und mit dem X-Translate-System verglichen.

### 5.8.1 IP als Alternative zu X-Translate

IP und X-Translate sind sich in der dahinter liegenden Idee sehr ähnlich. Um IP als Alternative zu X-Translate abzulehnen, gibt es aber mehrere Gründe.

- Ein IP-System ist zur Zeit nicht erhältlich, somit ist es nicht möglich, die mögliche Anwendbarkeit von IP im Problemfeld dieser Arbeit zu untersuchen.
- Das IP-System ist zur Entwicklung von Anwendungen und Erweiterungsbibliotheken ausgelegt. Die on-the-fly-Kompilierung von SQL-Anweisungen mit Datenbankzugriff während der Kompilierung bzw. Reduktion ist eine Aufgabe, an die das IP-System erst angepasst werden muss.
- Um die von X-Translate ausgeführten Aufgaben zu Übernehmen, muss IP zuerst mit einem Import-Parser die XSQL-Anweisung importieren, dann mittels der Reduktionsmethoden auf SQL:1999 abbilden und danach an das DBVS schicken. X-Translate ist spezialisiert auf die Übersetzung von XSQL-Anweisungen und somit vermutlich schneller als das IP-System.
- Es müssen Import-Parser und SQL:1999-Intentionen für IP entwickelt werden. Bei X-Translate ist das nicht nötig, die schon vorliegende Grammatik für SQL:1999 reicht aus.

Leider lag uns kein lauffähiges IP-System vor, so das nicht mit letzter Sicherheit eine positive oder negative Bewertung von IP als Alternative zu X-Translate vorgenommen werden kann.

### 5.8.2 SQL:1999+ als Alternative zum Anpassungscompiler

Das von Kauhaus et al. [KLS02] vorgestellte System zur Abbildung von SQL:1999+ auf Herstellerdialekte ist vergleichbar mit dem Anpassungscompiler von X-Translate. Die Entwicklung des Anpassungscompilers aus dem XSQL-Compiler kann jedoch mit relativ geringem Aufwand geschehen, im einfachsten Fall kann der XSQL-Compiler direkt dafür genutzt werden. Es werden keine zusätzlichen Schnittstellen zu einem weiteren Programm nötig, und es wird nur ein System anstatt zwei vor die Datenbank geschaltet. Die Verwendung eines eigenen Anpassungscompiler für X-Translate ist deshalb die bessere Wahl.

### 5.8.3 XSLT als Alternative Compiler-Implementierungssprache

Oberflächlich betrachten arbeitet XSLT sehr ähnlich wie X-Translate, und scheint sich deshalb für die Implementierung des XSQL-Compilers zu eignen. In Java muss die Funktionalität des Pattern-Matching erst nachgebildet werden. Bei näherer Betrachtung zeigt sich jedoch, das es hier einige Probleme gibt.

- Der Quellbaum ist von überall in seiner Gänze sichtbar, es wird sehr schwierig zu überwachen, dass Transformationen und Methoden alle Regeln einhalten.
- Bei Java ist die Implementierung der Methoden und Transformationen nicht intuitiv, da es jeweils eine Dispatcher-Methode geben muss, die alle Knotenmuster testet und dann die passende X-Translate-Methode oder Transformation aufruft. Die Annahme, dass dies in XSLT eleganter gelöst werden könnte durch die eingebaute Pattern-Matching-Funktionalität erwies sich als nicht berechtigt: in XSLT gibt es für jeden Knoten im Quellbaum nur eine Vorlage, die aufgerufen wird. Will man polymorphe Methoden und mehrfache Transformationen eines Knotens zulassen, wird das genauso komplex wie mit Java.
- XSLT-Prozessoren und -Stylesheets sind nicht zur Implementierung eines Compilers entwickelt worden. Dadurch ist es möglich, dass die Performanz eines in XSLT implementierten Compilers nicht optimal ist.
- Um sinnvoll in XSLT programmieren zu können, ist es zur Zeit noch erforderlich, bestimmte Herstellererweiterungen für XSLT zu benutzen. Damit würde X-Translate dann sprichwörtlich „den Teufel mit dem Belzebub austreiben“, in dem die DBVS mit XSQL herstellerunabhängig werden, aber X-Translate selbst von bestimmten Produkten abhängt.
- Längere Codeblöcke mit prozeduralem Code, wie sie in Transformationen oft benutzt werden, lassen sich in einer prozeduralen Programmiersprache wesentlich einfacher schreiben.

Kommentar 5.1:

Die Syntax der in dieser Arbeit vorgestellten Transformationssprache ist näher an Java als an XSLT. Es fällt deshalb leichter, sie nach Java zu übersetzen. Das ist jedoch kein durch XSLT verursachtes Problem. Aufgrund der Entscheidung für Java als Implementierungssprache wurde die Transformationssprache an Java angelehnt.

# Kapitel 6 Zusammenfassung und Ausblick

In diesem Kapitel werden die Hauptpunkte der Arbeit zusammengefasst und kritisch hinterfragt. Danach wird ein Ausblick gegeben auf die noch offenen Fragen.

## 6.1 Zusammenfassung

In dieser Arbeit wurde ein System zur Entwicklung und zum Einsatz von Sprachmodulen für die Erweiterung von SQL:1999 vorgestellt. Mehrere Sprachmodule können unabhängig voneinander entwickelt werden, sind dann aber gleichzeitig benutzbar. Um dies zu ermöglichen, muss dafür gesorgt werden, das sich die Sprachmodule nicht gegenseitig beeinflussen, aber trotzdem jedes Sprachmodul möglichst großen Handlungsfreiraum hat. Das wurde durch die Verwendung von Methoden für den Quellbaumzugriff und die Einführung von drei verschiedenen Transformationstypen erreicht.

Die mit X-Translate erreichte Erweiterbarkeit von SQL:1999 erstreckt sich von der Grammatik über das Metamodell bis hin zur Semantik. Je komplexer Erweiterungen sind, und je mehr Erweiterungen gleichzeitig installiert sind, desto größer ist die Konfliktwahrscheinlichkeit zwischen zwei verschiedenen Sprachmodulen. Diese Konflikte können teilweise vom Administrator bei der Installation der Sprachmodule behoben werden. Bei nicht auflösbaren Konflikten entsteht eine Inkompatibilität zwischen Sprachmodulen.

Um auf kommerziellen DBVS eingesetzt zu werden, ist es wichtig, dass XSQL nicht nur SQL:1999 abgebildet wird, sondern auch auf die SQL-Dialekte der Hersteller. Das wurde durch einen zweiten Übersetzungsschritt mit dem Anpassungscompiler erreicht. Wenn Herstellermodule und Herstelleranpassungen zusammen eingesetzt werden, können Erweiterungen des Herstellers, wie beispielsweise Anweisungen zur Indexgenerierung und -benutzung, an das DBVS ohne Transformation durchgereicht werden.

Die Komplexität eines Sprachmoduls kann sehr unterschiedlich sein. Wenn eine Erweiterung die Datenstrukturen erweitert, auf denen die meisten SQL:1999-Operationen arbeiten, wird es ein sehr großes und komplexes Sprachmodul mit vielen Transformationen. Dadurch steigt die Konfliktwahrscheinlichkeit. Ist die Erweiterung jedoch nur lokal, wie die Erweiterung der `SELECT`-Anfrage bei Preference-SQL so ist weitaus seltener mit Konflikten zu rechnen, da eine solche Spracherweiterung nur sehr wenige Transformationen hat.

## 6.2 Ergebnisse

In diesem Abschnitt wird ein Überblick über die Ergebnisse dieser Arbeit gegeben.

- Um SQL:1999 zu erweitern, musste ein erweitertes Informationsschema und ein erweitertes Definitionsschema eingeführt werden, da das von SQL:1999 bereitgestellte Informationsschema nicht erweiterbar ist. Da im Informationsschema die Metadaten zu *allen* Schemaelementen, also auch den von X-Translate intern genutzten, sichtbar sind, wurden Maskierungstabellen eingeführt, um die Daten zu verdecken.

- Um das Metamodell (also das erweiterte Informationsschema) zu erweitern, wurde ein neuer Befehl `EXTEND VIEW` eingeführt, um Sichten zu erweitern. Alle anderen Befehle, wie Erzeugung von Sichten und Tabellen, gibt es in SQL:1999 bereits. Das Metamodell wird somit in einer SQL-ähnlichen DDL spezifiziert.

- Es wurde eine Sprache entworfen, mit der sich die Abbildung von Spracherweiterungen auf SQL:1999 spezifizieren lässt. Diese Sprache ist genau für diesen Zweck entworfen und bietet einfache Möglichkeiten, durch Transformation von Knoten aus einem Quellbaum einen Zielbaum zu erstellen.

- Zur Übersetzung der in der Transformationssprache geschriebenen Methoden und Transformationen wurde zwei Ansätze für einen Compiler-Generator untersucht. Aufgrund der beschränkten Möglichkeiten von XSLT wurde sich für Java entschieden. Die Transformationssprache wird also zuerst in Java-Quellcode, und mit einem normalen Java-Compiler in Java-Bytecode übersetzt.

- Um die Grammatik zu erweitern, wurde eine BNF-Notation festgelegt, in der die Erweiterungen zu spezifizieren sind. Der Parser muss einen kompletten Parsebaum, und keinen AST, ausgeben.

- Es wurden verschiedene Stellen identifiziert, an denen Konflikte zwischen Spracherweiterungen auftreten können, nämlich in Grammatik, Metamodell, und Transformationen. Um diese Konflikte aufzulösen, wurde die Rolle des Administrators für die Installation eingeführt. Der Administrator kann durch Änderung der Spracherweiterungen Konflikte auflösen.

- Um, soweit möglich, alle Konflikte zu finden und keinen zu übersehen, wurde eine vorsichtige Herangehensweise gewählt. Deshalb kann X-Translate Konflikte finden, die in Wirklichkeit keine sind. Dies kann vom Administrator dem System bekanntgemacht werden. Andererseits kann nicht ausgeschlossen werden, das einige Konflikte, vor allem semantische, dem System verborgen bleiben. Hier ist man auf den Administrator angewiesen, der solche Konflikte in einer manuellen Prüfung erkennen muss.

- Die Arbeit beschäftigte sich hauptsächlich mit der Umsetzung des X-Translate-Systems als Java-Programm. Der Zugriff für Anwendungsprogramme erfolgt über einen JDBC-Treiber, die Entwicklungswerkzeuge werden in Java implementiert, und das von Parser-Generator und Compiler-Generator generierte

Laufzeitsystem ist auch ein Javaprogramm. Somit ist eine hohe Portabilität des X-Translate-Systems gewährleistet.

- Es wurde festgestellt, das es relativ einfach ist, kleine Spracherweiterungen oder eingebettete Spracherweiterungen [CE00] zu entwickeln. „Klein“ bedeutet, das wenig Transformationen spezifiziert werden müssen. Bei größeren Erweiterungen, wie beispielsweise die Mehrfachvererbung, müssen sehr viele Transformationen geschrieben werden, weil sich die Änderung auf sehr viele SQL:1999-Anweisungen auswirkt. Im Falle der Mehrfachvererbung ist jede Anweisung, in der ein benutzerdefinierter Typ oder eine Tabelle angesprochen wird, betroffen, was das Ausmaß der Spracherweiterung verdeutlicht. Dadurch steigt das Risiko des Konflikts mit anderen Spracherweiterungen.

## 6.3 Kritik

Der erste Kritikpunkt setzt an der Komplexität von Sprachmodulen an. Für kleine Erweiterungen ist X-Translate eine sinnvolle Möglichkeit. Bei komplexen Erweiterungen, wie der Einführung neuer, domänenspezifischer Datentypen und Operationen müssen für fast jedes Nonterminalsymbol Transformationen geschrieben werden. Somit sind, wenn zwei solcher Erweiterungen gleichzeitig benutzt werden sollen, Konflikte fast unvermeidbar.

Der zweite Punkt der Kritik bezieht sich auf die Verwendung von SQL:1999 als Abbildungssprache. Der SQL:1999-Standard ist ein sehr großer, komplexer Standard. Vermutlich wird kein Hersteller den Standard jemals vollständig implementieren, zur Zeit wird selbst der Sprachkern von niemandem implementiert. Die Verwendung von Herstelleranpassungen ist eher ein Notbehelf; es wird ein zweiter Übersetzungsschritt notwendig. Durch die Komplexität von SQL:1999 wird die Entwicklung von Spracherweiterungen sehr erschwert, da XSQL immer abwärtskompatibel bleiben muss. Tatsächlich wird diese Abwärtskompatibilität schon beim Erweitern einer IS-Sicht um eine Spalte zerstört: ein `SELECT * FROM <sichtname>` liefert eine Spalte mehr zurück, wodurch im schlimmsten Fall eine Anwendung nicht mehr funktioniert. X-Translate selbst benutzt intern auch nirgendwo implementierte SQL:1999-Anweisungen, somit ist das System, so, wie es hier vorgestellt wurde, nicht einsetzbar. Die internen Anweisungen müssen zuerst auf eine von allen Herstellern unterstützte Teilmenge von SQL:1999 abgebildet werden.

Ein dritter Kritikpunkt ist die Performanz. Da X-Translate auf dem DBVS aufsetzt und nicht in den DBVS-Anfragecompiler integriert ist, wird die Performanz beeinträchtigt, unabhängig davon, wie schnell oder wie langsam das X-Translate-Laufzeitsystem arbeitet. Bei Ad-Hoc Anfragen wirkt sich das nicht sonderlich aus. Bei DBVS, die unter hoher Last arbeiten, kann die Benutzung von X-Translate unmöglich sein. Das X-Translate-System unterstützt keine prepared Statements, da der Anfragestring immer komplett transformiert wird. Auch dadurch kommt es zu Perfomanzeinbußen.

Als vierter Kritikpunkt ist die Entwicklung von Sprachmodulen zu nennen. Das X-Translate-System kann einen Entwickler nur minimal in seiner Arbeit unterstützen,

womit die Qualität eines Sprachmoduls sehr stark von den Fähigkeiten des Entwicklers abhängt. Es können zwar Knotenmuster für Transformationen und Methoden spezifiziert werden, um festzulegen, auf welchen Knoten die Transformation bzw. Methode arbeiten soll. Das System kann aber nicht feststellen, ob ein Knotenmuster wirklich minimal ist in dem Sinne, das es möglichst nur auf die Knoten passt, die transformiert werden. Es kann sein, das durch ein zu allgemein spezifiziertes Knotenmuster sich mit einem anderen an Stellen überlappt, die in einem korrekten Syntaxbaum aufgrund anderer, nicht in der Grammatik erwähnten Regeln gar nicht vorkommen können. Hier entsteht ein Scheinkonflikt; je nach den Fähigkeiten des Administrators, und je nach Komplexität der Sprachmodule kann ein solcher Konflikt nicht unbedingt erkannt werden, und prinzipiell ist es immer besser, Scheinkonflikte erst gar nicht entstehen zu lassen. Um also maximale Kompatibilität zu anderen Sprachmodulen zu gewährleisten, muss sehr umsichtig programmiert werden.

Der fünfte Punkt bezieht sich auf die vorgestellten Parser- und Compiler-Generatoren. Es wird immer der Quellcode aller Spracherweiterungen benötigt, um Parser und Compiler zu generieren. Das ist ein Problem für kommerzielle Sprachmodul-Hersteller, da sie so ihre eigenen Entwicklungen der Konkurrenz offenlegen.

Zuletzt bleibt die Frage, ob der Nutzen von unabhängigen Spracherweiterungen sich nicht ins Gegenteil verkehrt. Durch X-Translate ist es möglich, eine Quelle von Inkompatibilitäten in der Datenbank-Welt, die der unterschiedlichen SQL-Dialekte von DBVS-Herstellern, zu entschärfen. Aber durch die vielen Konfliktmöglichkeiten, die es zwischen Sprachmodulen gibt, kann auch eine neue Quelle der Inkompatibilität entstehen, wenn die Hersteller von Sprachmodulen untereinander inkompatible Produkte vertreiben.

## 6.4 Offene Fragen

Nach der Zusammenfassung und dem Ansprechen einiger Kritikpunkte werden in diesem Abschnitt einige Punkte angesprochen, die weiterer Forschung bedürfen, und deren Beantwortung möglicherweise einige der kritischen Punkte von X-Translate entkräften kann.

- Durch die Komplexität von SQL:1999 wird die Entwicklung von Erweiterung erschwert; hier wäre die Festlegung einer von allen Hersteller unterstützten Teilmenge von SQL:1999 sinnvoll, auf die abgebildet wird. Der komplette SQL:1999-Funktionsumfang kann als Sprachmodul angeboten werden, und schritthaltend mit der Entwicklung der Herstellerdialekte in Richtung SQL:1999 kann mehr SQL:1999-Funktionalität in das Basissystem aufgenommen werden.

- Viele Aspekte von SQL:1999, die unter Umständen Probleme verursachen können, wurden in dieser Arbeit nicht oder nur flüchtig betrachtet. SQL-Routinen müssen möglicherweise nach der Ausführung von XSQL-Anweisungen umgeschrieben werden. Das kann beispielsweise passieren, wenn

eine Tabelle die Mehrfachvererbung nutzt. Alle SQL-Routinen, die auf simulierte Supertabellen zugreifen, müssen angepasst werden. Besonders problematisch wird das bei externen SQL-Routinen, da X-Translate hier keine Möglichkeit hat, sie zu ändern.

Um dieses Problem zu lösen, muss X-Translate direkt in das DBVS-System integriert werden. Es gibt einige DBVS, deren Quellcode frei zugänglich ist. Dort könnte man eine Integration des X-Translate-Systems erforschen. Im ersten Schritt kann man X-Translate soweit in die DB-Engine integrieren, das alle SQL-Anweisungen zuerst durch das X-Translate-System laufen, jedoch das X-Translate-System weiterhin unabhängig ist. Im zweiten Schritt stünde die *tiefe* Integration von X-Translate in die DB-Engine, d. h. X-Translate wird ein echter Teil des DBVS. Bei DBVS, die dies nicht unterstützen, könnte weiterhin die externe Implementierung von X-Translate eingesetzt werden.

- Zugriffsrechte für Benutzer und Rollen wurden in dieser Arbeit nicht besprochen. Das X-Translate-System braucht selbst Administrator-Rechte, der Benutzer darf aber nur die Operationen ausführen, zu denen er berechtigt ist. Hierzu müssen an vielen Stellen des X-Translate-Systems noch entsprechende Prüfungen und Zusatzklauseln in Anfragen eingefügt werden. Eine Möglichkeit der Implementierung ist die Verwendung von speziellen Tabellen und Spalten, ähnlich den Maskierungstabellen zum Verstecken der internen Metadaten.

- Sprachmodule sollten wahlweise im Quellcode oder in kompilierter Form, d. h. einer nicht menschenlesbaren Form, vorliegen, um XSQL auch für kommerzielle Hersteller interessant zu machen. Hier stellt sich die Frage, inwieweit der Administrator bei der Installation zur Konfliktbehebung noch eingreifen kann. Die Knotenmuster der Transformationen bzw. die Klassen der Methoden könnten beispielsweise offengelegt, die Rümpfe von Transformationen und Methoden hingegen in kompilierter Form vertrieben werden.

- Um die Performanz der mehrfachen Ausführung von SQL-Anweisungen zu steigern, ist die Verwendung von prepared Statements sehr sinnvoll. Hierzu muss ein Mechanismus entworfen werden, der die ursprünglichen Platzhalter während der Transformation verfolgt, um die Werte bei der nächsten Anfrage direkt in der transformierten Anweisung einzusetzen. Hier kann es Probleme geben, wenn ein Platzhalter während der Transformation wegfällt bzw. durch einen Metadatenzugriff verändert wird.

- Um die Qualität von Sprachmodulen weniger abhängig von den Fähigkeiten des Entwicklers zu machen, muss es klare, vom X-Translate-System verarbeitbare Regeln oder Heuristiken geben, die z.B. für ein Knotenmuster erkennen, ob es minimal ist oder ob möglicherweise ein Scheinkonflikt enstehen kann. Vermutlich wird man sich hier auf Heuristiken beschränken müssen, da die Semantik der Spracherweiterung erst durch das Definieren von Transformationen, Knotenmustern usw. festgelegt wird, aber zur Bewertung von Knotenmustern diese Semantik schon bekannt sein muss.

# Anhang A Abkürzungen

**AST** Abstrakter Syntaxbaum (abstract syntax tree)

**BNF** Backus-Naur Form

**DBVS** Datenbank-Verwaltungssystem

**DDL** Daten-Definitionssprache (data definition language)

**DML** Daten-Manipulationssprache (data manipulation language)

**DS** Definitionsschema (`DEFINITION_SCHEMA`)

**DSL** Domänenspezifische Sprache (domain specific language)

**DSX** Erweitertes Definitionsschema (`DEFINITION_SCHEMA_EXTENDED`)

**GPL** Universelle Sprache (general purpose language)

**IP** Intentional Programming

**IS** Informationsschema (`INFORMATION_SCHEMA`)

**ISX** Erweitertes Informationsschema (`INFORMATION_SCHEMA_EXTENDED`)

**XSL** Extensible Stylesheet Language

**XSLT** XSL Transformations

**XSS** X-Translate-Systemschema (`XT_SYSTEM_SCHEMA`)

**XSQL** erweitertes SQL:1999

**XT** X-Translate

# Anhang B Details der Transformationssprache

Dieser Anhang enthält die Details zur Transformationssprache. Abschnitt B.1 beschreibt, wie Transformationen und Methoden auf mehrere Dateien aufgeteilt werden. In Abschnitt B.2 wird die Syntax der Sprache in BNF dargestellt.

## B.1 Dateien

Die Definitionen der Transformationen und Methoden können auf mehrere Dateien und Verzeichnisse verteilt werden. Der Modul-Packager durchsucht das angegebene Verzeichnis und alle Unterverzeichnisse nach Dateien mit der Endung `.ts`.

## B.2 Syntax

Die Syntax der Transformationssprache ist an Java angelehnt. In den Rümpfen von Transformationen und Methoden kann generell Java-Code, der auch in einer Java-Methode stehen kann, benutzt werden. Das heißt, es sind keine Klassen- und Methodendefinitionen erlaubt. Die Benutzung von Java-Klassen ist möglich. Am Anfang einer Datei können Java-Import-Anweisungen stehen.

Die verwendete BNF-Notation entspricht der in Kapitel 4 vorgestellten.

### B.2.1 Bäume und Knotenmuster

Bäume und Knotenmuster sind Literale vom Typ `Tree` und können dort stehen, wo Java-Literale erlaubt sind.

⟨`baum`⟩ ::= '#' { ⟨`knoten`⟩ | ⟨`knotenmuster`⟩ }

⟨`knoten`⟩ ::= '(' ⟨`symbol`⟩, ⟨`kindknoten`⟩ ')' | ⟨`symbol`⟩ | '(' ⟨`symbol`⟩ ')'

⟨`kindknoten`⟩ ::= ⟨`knoten`⟩ [ { ',' ⟨`knoten`⟩ }... ]

⟨`symbol`⟩ ::= {⟨`nonterminal`⟩ ('*') | ⟨`terminal`⟩} [ ':' ⟨`label`⟩ ]

⟨`nonterminal`⟩ ::= '<' ⟨`identifier`⟩ '>'

⟨`terminal`⟩ ::= '[' ⟨`identifier`⟩ ']'

⟨`this`⟩ ::= 'this'

⟨label⟩ ::= ⟨identifier⟩

⟨knotenmuster⟩ ::= '(' ⟨mustersymbol⟩, ⟨kindmuster⟩ ')' | ⟨mustersymbol⟩ | '(' ⟨mustersymbol⟩ ')'

⟨kindmuster⟩ ::= ⟨knotenmuster⟩ [ { { ',' | '|' } ⟨knotenmuster⟩}... ]

⟨mustersymbol⟩ ::= [ ⟨wildcard⟩ ] ⟨symbol⟩

⟨wildcard⟩ ::= ⟨horizontal wildcard⟩ | ⟨vertical wildcard⟩

⟨horizontal wildcard⟩ ::= '?' [ ⟨quantifier⟩ ]

⟨vertical wildcard⟩ ::= '@' ⟨quantifier⟩

⟨quantifier⟩ ::= ⟨range⟩ | '*'

⟨range⟩ ::= ⟨number⟩ '..' ⟨number⟩

### B.2.2 Knotenangaben

Knotenangaben sind Objekte vom Typ `Tree` und können dort stehen, wo laut Java-Syntax solche Objekte erlaubt sind.

⟨knotenangabe⟩ ::= ⟨pfadausdruck⟩ | ⟨direkte angabe⟩

⟨pfadausdruck⟩ ::= '#' { 'this' | ⟨identifier⟩ } ':/' [ ⟨pfadschritt⟩... ]

⟨pfadschritt⟩ ::= { ⟨identifier⟩ [ '[' ⟨number⟩ ']' ] | '..' } '/'

⟨direkte angabe⟩ ::= '#' { 'this' | ⟨identifier⟩ } '[' ⟨identifier⟩ ']'

### B.2.3 Methoden

⟨methode⟩ ::= ⟨methodenkopf⟩ '{' ⟨methodenrumpf⟩ '}'

⟨methodenkopf⟩ ::= ⟨typ⟩ 'method' ⟨methodenname⟩ ⟨klasse⟩ [ ⟨from⟩ ] [ ⟨of⟩ ]

⟨typ⟩ ::= 'up' | 'down'

⟨methodenname⟩ ::= ⟨identifier⟩

⟨klasse⟩ ::= ⟨knotenmuster⟩

⟨from⟩ ::= 'from' ⟨knotenmuster⟩

⟨of⟩ ::= 'of' ⟨identifier⟩

⟨methodenrumpf⟩ ::= ⟨javacode⟩

### B.2.4 Transformationen

⟨transformation⟩ ::= ⟨ttyp⟩ 'transformation' ⟨knotenmuster⟩ 'accesses' ⟨knotenmuster⟩ '{' ⟨transformationsrumpf⟩ '}'

⟨ttyp⟩ ::= 'appending' | 'changing' | 'reading'

⟨transformationsrumpf⟩ ::= ⟨javacode⟩

# Literaturverzeichnis


[ASU86] Aho, A. V., Sethi, R., Ullman, J. D.:
*Compilers – Principles, Techniques, and Tools*
Addison Wesley, 1986

[Ca+95] Carey, M., et al.:
*Towards Heterogeneous Multimedia Inforamtion Systems: The Garlic Approach*
Proc. IEEE RIDE-DOM, Taipei, Taiwan, März 1995

[CE00] Czarnecki, K., Eisenecker, U.:
*Generative Programming*
Addison Wesley, 2000

[CEG+00] Czarnecki K., Eisenecker, U., Glück, R., Vandevoorde, D., Veldhuizen, T.:
*Generative Programming and Active Libraries*
Proceedings of the Dagstuhl Seminar 98171 on Generic Programming, Schloß Dagstuhl, Germany, April/Mai, 1998

[Ch02] Chappel, D.:
*Understanding .NET: A Tutorial and Analysis*
Addison Wesley Professional, 2002

[Ge95] Geiger, K.:
*Inside Odbc*
Microsoft Press, 1995

[GR89] Goldberg, A., Robson, D.:
*Smalltalk 80: The Language*
Addison Wesley Professional, 1989

[ISO99a] ANSI/ISO/IEC 9075-1-1999:
*Information Systems — Database Languages — SQL — Part 1: Framework (SQL/Framework)*
American National Standard Institute, Inc., 1999

[ISO99b] ANSI/ISO/IEC 9075-2-1999:
*Information Systems — Database Languages — SQL — Part 2: Foundation (SQL/Foundation)*
American National Standard Institute, Inc., 1999

[ISO99c] ANSI/ISO/IEC 9075-4-1999:
*Information Systems — Database Languages — SQL — Part 4: Persistent Stored Modules (SQL/PSM)*
American National Standard Institute, Inc., 1999

[ISO00] ISO/IEC 9075-10:2000:
*Information technology — Database languages — SQL — Part 10: Object Language Bindings (SQL/OLB)*
International Organization for Standardisation, 2000

[ISO02] ISO/IEC 9075-13:2002:
*Information technology — Database languages — SQL — Part 13: SQL Routines and Types Using the Java™ Programming Language (SQL/JRT)*
International Organization for Standardisation, 2002

[KK02] Kießling, W., Köstler, G., :
*Preference SQL — Design, Implementation, Experiences*
Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

[KLS02] Kauhaus, C., Lufter, J., Skatulla, S.:
*Eine Transformationsschicht zur Realisierung objektrelationaler Datenbankkonzepte mit erweiterter Kollektionsunterstützung*
Datenbank-Spektrum Jahrgang 2, Heft 4, November 2002

[LMB92] Levine, J., Mason, T., Brown, D.:
*lex & yacc, 2nd Edition*
O'Reilly, 1992

[Lu02] Lufter, J.:
*Kollektionsunterstützung für SQL:1999*
in: Jenaer Schriften zur Mathematik und Informatik Math/Inf/05/02, Institut für Informatik, Friedrich-Schiller-Universität Jena, Januar 2002

[Ma02] Mahnke, W.:
*Towards a modular, object-relational schema design*
in: Proc. 9th Doctoral Consortium of the 14th Int. Conf. on Advanced Information Systems Engineering (CAiSE'2002), Toronto, Mai 2002

[MRS99] Marder, U., Ritter, N., Steiert, H.-P.:
*A DBMS-based Approach for Checking OCL Constraints*
in: OOPSLA'99-Workshop „Rigourous Modeling and Analysis with the UML: Challenges and Limitations", Denver, Co., 1999.

[MS01a] Mahnke, W., Steiert, H.-P.:
*Es ist an der Zeit, die Prinzipien komponentenbasierten Software Engineerings im objekt-relationalen Datenbankentwurf anzuwenden!*
in: Tagungsband der GI/OCG-Jahrestagung Informatik 2001, Wien, September 2001

[MS01b] Mahnke, W., Steiert, H.-P.:
*Modularity in ORDBMSs - A new Challenge*
in: Tagungsband 13. Workshop „Grundlagen von Datenbanken", GI-FG 2.5.1, Magdeburg, Juni 2001

[Ne02] Neumann, D.:
*Konzepte für einen modularen Schemaentwurf in objekt-relationalen Datenbanksystemen*
Diplomarbeit, Universität Kaiserslautern, November 2002

[OMG97] Object Management Group (OMG):
*Object Constraint Language Specification, Version 1.1*
OMG Document ad/97-08-08, September 1997

[OMG02] Object Management Group (OMG):
*Meta Object Facility (MOF) Specification, Version 1.4*
Object Management Group, Inc., April 2002

[RS97] Roth, M.-T., Schwarz, P.:
*Don't Scrap It, Wrap it! — A Wrapper Architecture for Legacy Data Sources*
Proceedings of the 23rd VLDB Conference, Athen, Griechenland, 1997

[RS00] Ritter, N., Steiert, H.-P.:
*Enforcing Modeling Guidelines in an ORDBMS-based UML Repository*
in: International Resource Management Association Conference 2000 (Information Modeling Methods and Methodologies Track of IRMA 2000) Anchorage, Alaska, Mai 2000, Seiten 269-273.

[Si95] Simonyi, C.:
*The death of computer languages, the birth of Intentional Programming*
Tech. Rep. MSR-TR-95-52, Microsoft Research, 1995

[Si96] Simonyi, C.:
*Intentional Programming — Innovation in the Legacy Age*
Präsentiert auf dem IFip WG 2.1 Meeting, Juni 1996

[Si99] Simonyi, C.:
*The Future is Intentional*
IEEE Computer 5/1999, S. 56-57

[SM00] Sun Microsystems, Metamata Inc.:
*JavaCC Version 2.1*
Oktober 2000
URL: http://www.webgain.com/products/java_cc/

[Sun01a] Sun Microsystems:
*Java™ 2 Platform, Enterprise Edition (J2EE™) Specification, Version 1.3*
Sun Microsystems, Palo Alto, August 2001

[Sun01b] Sun Microsystems:
*Java™ JDBC™ Data Access API Specification, Version 3.0*
Sun Microsystems, Palo Alto, Dezember 2001

[Ve98] Veldhuizen, T. L.:
*Arrays in Blitz++*
in: ISCOPE'98, Lecture Notes in Computer Science Band 1505, 1998

[Vi98] Viega, J.:
*Automated Delegation*
Masters Thesis, University of Virginia, August 1998

[W3C02a] World Wide Web Consortium:
*Extensible Markup Language (XML) 1.1*
W3C Candidate Recommendation, Oktober 2002
URL: http://www.w3.org/TR/xml11/

[W3C02b] World Wide Web Consortium:
*XSL Transformations (XSLT) Version 2.0*
W3C Working Draft, November 2002
URL: http://www.w3.org/TR/xslt20/

[WMS+01] Van Wyk, E., de Moor, O., Sittampalam, G., Sanabria-Piretti, I., Backhouse, K., Kwiatkowski, P.:
*Intentional Programming: a Host of Language Features*
Tech. Rep. PRG-RR-01-21, Computing Laboratory, University of Oxford, 2002

[Zh01] Zhang, N.:
*Supporting Semantically Rich Relationships in Extensible Object-Relational Database Mangagement Systems*
Shaker-Verlag, September 2000