UNIVERSITÄT MANNHEIM

Design und Implementierung eines Modulkonzepts für Demaq

Diplomarbeit

vorgelegt bei
Prof. Dr. Carl-Christian Kanne
Juniorprofessur für Praktische Informatik
(Informationssysteme)

von
Andreas Krämer
im April 2009

Inhaltsverzeichnis

1 Einleitung

Durch die steigende Relevanz des Internet gewinnen so genannte verteilte Anwendungen eine immer größere Bedeutung. Eine verteilte Anwendung ist eine Anwendung, die in einem verteilten System ausgeführt wird. Diese Systeme bestehen aus mehreren Rechnern, die durch ein Netzwerk wie etwa das Internet miteinander verbunden sind. Ein wichtiger Einsatzbereich solcher verteilter Anwendungen stellt dabei das automatische Bearbeiten von unternehmensübergreifenden Geschäftsprozessen dar. Innerhalb dieser Geschäftsprozesse müssen Daten zwischen den beteiligten Rechnern ausgetauscht werden. Um diesen Austausch zu erleichtern wurde mit XML vom W3C ein standardisiertes Format für den Datenaustausch in verteilten Systemen entwickelt. Die zu den Geschäftsprozessen gehörigen Anwendungen sind in der Regel in einer Programmiersprache wie Java oder C++ realisiert. Ein Problem dieser Sprachen ist, dass sie nicht direkt auf XML-Dokumenten arbeiten können, sondern die ausgetauschten Daten zunächst in ein geeignetes Format konvertieren müssen. Diese notwendige Konvertierung wirkt sich negativ auf die Performance aus und erhöht außerdem die Komplexität sowie den Entwicklungsaufwand dieser Anwendungen.

Aus diesem Grund wurde von der Forschungsgruppe Datenbanken der Universität Mannheim das Projekt *Demaq* ins Leben gerufen. Dabei handelt es sich um ein System, mit welchem verteilte Anwendungen entwickelt und ausgeführt werden können. Dieses System bietet eine native Unterstützung des XML-Formats, sodass direkt auf XML-Dokumenten gearbeitet werden kann. Anwendungen in Demaq werden mit einer eigens für dieses System eingeführten neuartigen Sprache entwickelt. Mit dieser Sprache wird die Anwendungslogik von verteilten Anwendungen auf Basis von XML-verarbeitenden Regeln spezifiziert.

1.1 Zielsetzung der Arbeit

Bei der Entwicklung von Anwendungen wird häufig wiederkehrende Funktionalität mehrfach innerhalb derselben oder in verschiedenen Anwendungen benötigt. Beispiele für solche Funktionalität sind Zugriffe auf das Dateisystem, mathematische Berechnungen oder die Erstellung von grafischen Oberflächen. Die meisten Programmiersprachen bieten für solche Aufgaben üblicherweise System-Bibliotheken an, welche die gängigsten Funktionalitäten bereitstellen. Auf diese Bibliotheken kann bei der Entwicklung von Anwendungen zugegriffen werden. Wird über diese Bibliotheken hinaus gehende Funktionalität benötigt, stehen dem Benutzer normalerweise Werkzeuge zur Verfügung, mit welchen er eigene Bibliotheken erstellen und benutzen kann.

Ein solches Konzept fehlt bislang in der sich noch in der Entwicklung befindlichen Demaq-Sprache. In der vorliegenden Arbeit geht es darum eine Erweiterung für Demaq zu entwerfen und zu implementieren, mit der solche benutzerdefinierten Bibliotheken erstellt und verwendet werden können.

1.2 Gliederung der Arbeit

Um die gesteckten Ziele zu erfüllen, werden in Kapitel 2 die Grundlagen für die eigentliche Arbeit gelegt. Kapitel 3 beschäftigt sich mit einer Reihe von Anwendungsfällen, aus welchen die funktionalen und nicht-funktionalen Anforderungen für die Erweiterung von Demaq abgeleitet werden. Aus diesen Anforderungen ergeben sich für einen Entwurf der Spracherweiterung verschiedene Alternativen, welche in Kapitel 4 diskutiert werden. In Kapitel 5 wird darauf die Implementierung dieses Entwurfs beschrieben. Die Evaluation in Kapitel 6 prüft, ob die gestellten Anforderungen erfüllt werden konnten. Abschließend enthält Kapitel 7 eine Zusammenfassung der Ergebnisse sowie einen Ausblick auf zukünftige Möglichkeiten zur Weiterentwicklung von Demaq auf Basis dieser Ergebnisse.

2 Grundlagen

Dieses Kapitel erläutert die zum Verständnis der vorliegenden Arbeit notwendigen Grundlagen. Hauptsächlich wird dabei Demaq vorgestellt, wozu zunächst die dafür benötigten Konzepte *XML*, *XPath*, *XQuery* sowie Grundlagen des Compilerbaus besprochen werden. Den Abschluss bildet ein Unterkapitel über die Wiederverwendung von Anwendungs-Code.

2.1 XML

XML [15] steht für *Extensible Markup Language* und ist ein Dateiformat, um semi-strukturierte Daten zu speichern. Ein XML-Dokument der Version 1.1 beginnt mit der *XML-Deklaration*, in Version 1.0 ist diese Angabe optional. Diese Deklaration sieht folgendermaßen aus, die Angabe der Zeichenkodierung ist optional: `<?xml version=“1.1” enoding=“UTF-8”?>`. Erst danach kommt der eigentliche Inhalt des Dokuments, welcher zumindest aus einem *Wurzelelement* bestehen muss, welches weitere Elemente enthalten kann. Auf der Ebene des Wurzelelements dürfen sich keine weiteren Elemente befinden, die erste Ebene enthält nur das Wurzelelement.

Ein *XML-Element* besteht entweder aus einem *Start-Tag* und einem *End-Tag*, oder aus einem selbst schließenden *Tag* (*Empty-Element-Tag*). *Start-Tags* bestehen aus dem Namen des Elements, eingeschlossen von den Symbolen < und >. Das *End-Tag* enthält ebenfalls den Namen des Elements, umrahmt von den Symbolen </ und >. *Empty-Element-Tags* bestehen aus dem Elementnamen, der von < und /> eingeschlossen wird.

Zwischen *Start-* und *End-Tag* befindet sich der Inhalt des Elements. Dieser Inhalt besteht in der Regel aus einem einzelnen Datensatz oder aus XML-Elementen. Somit lassen sich Elemente verschachteln, wodurch sich in XML-Dokumenten beliebige Baumstrukturen abbilden lassen. Verschachtelte Elemente müssen in der richtigen Reihenfolge wieder geschlossen werden.

Ein Beispiel für ein XML-Element ist `<Land>Deutschland</Land>`, wobei „Deutschland“ den Inhalt darstellt und das Tag die Bezeichnung „Land“ trägt, welche in *Start-* sowie *End-Tag* vorkommt.

Für ein leeres Element ohne Inhalt wird entweder das *Empty-Element-Tag* (`<Land/>`) verwendet, oder das Element besteht nur aus *Start-* und *End-Tag*, ohne dass Inhalt dazwischen steht (`<Land></Land>`).

Elemente können beliebig viele Attribute enthalten, welche in die *Start-* bzw. *Empty-Element-Tags* geschrieben werden. Diese Attribute bestehen aus einem Namen und einem Wert und werden durch Whitespace getrennt nach dem Element-Namen definiert, z. B. :

```
1 <Land Hauptstadt="Berlin" Vorwahl="0049">Deutschland</Land>
```

Hierbei stellen `Hauptstadt` und `Vorwahl` die Namen der Attribute dar und *Berlin* bzw. *0049* die entsprechenden Werte.

Listing 2.1 zeigt ein vollständiges XML-Dokument und beginnt in Zeile 1 mit der Versionsdeklaration. Danach folgt in Zeile 2 das Wurzelelement des Dokuments (`<Laender>`), welches in Zeile 18 geschlossen wird. Innerhalb des Wurzelelements (Zeilen 3-17) befinden sich beispielhaft drei `<Land>`-Elemente, welche die Elemente `Name`, `Hauptstadt` und `Flaeche` enthalten. Erst diese Elemente enthalten auch Datensätze. Die Elemente `Hauptstadt` sowie `Flaeche` besitzen zudem jeweils ein Attribut namens `Kennung` bzw. `Einheit`.

Listing 2.1: Ein XML-Dokument.

```
1  <?xml version="1.1"?>
2  <Laender>
3    <Land>
4      <Name Kennung="GER">Deutschland</Name>
5      <Hauptstadt>Berlin</Hauptstadt>
6      <Flaeche Einheit="km2">357104.07</Flaeche>
7    </Land>
8    <Land>
9      <Name Kennung="EN">England</Name>
10     <Hauptstadt>London</Hauptstadt>
11     <Flaeche Einheit="km2">130395</Flaeche>
12   </Land>
13   <Land>
14     <Name Kennung="FR">Frankreich</Name>
15     <Hauptstadt>Paris</Hauptstadt>
16     <Flaeche Einheit="km2">674843</Flaeche>
```

```
17   </Land>
18 </Laender>
```

Namensräume (Namespaces) Zur Vermeidung von Namenskonflikten in Attributen und Elementen können in XML Namensräume [4] definiert werden. Ein Namensraum wird wie ein Attribut innerhalb des Start-Tags (oder eines Empty-Element-Tags) definiert und dabei an einen eindeutigen *Uniform Resource Identifier* (*URI*) gebunden. Die Reichweite des Namensraums erstreckt sich vom Start-Tag, in dem er definiert wurde, bis zum dazu gehörigen End-Tag bzw. bei einem Empty-Element-Tag nur über den Tag.

Mit `xmlns:prefix=''URI''` wird ein Präfix namens *prefix* für den Namensraum definiert. Die Elemente und Attribute, auf die der Namensraum angewandt werden soll, müssen diesen Präfix vorangestellt bekommen. Dabei kommt es auf die Eindeutigkeit des URIs und nicht des Präfixes an, unterschiedliche Präfixe mit demselben URI bezeichnen denselben Namensraum.

Durch die Angabe von `xmlns=''URI''` kann ein Standard-Namensraum angegeben werden, der sich über seine ganze Reichweite auf alle Elemente und Attribute bezieht. Innerhalb eines Standard-Namensraums kann einem Element oder Attribut durch einen Präfix ein anderer als der Standard-Namensraum zugewiesen werden.

In Listing 2.2 werden im Wurzelelement `book` ein Standard-Namensraum (`urn:loc.gov:books`) sowie der Namensraum `urn:loc.gov:authors` mit dem Präfix *author* definiert. Der Standard-Namensraum bezieht sich auf die Elemente `book`, `title` und `author`. Da sich innerhalb des `author`-Elements ein weiteres `title`-Element befindet, wird dafür zur Unterscheidung der vom Standard-Namensraum abweichende Namensraum `urn:loc.gov:authors` verwendet.

Listing 2.2: Verwendung von Namensräumen.

```
1 <?xml version="1.1"?>
2 <book xmlns="urn:loc.gov:books" xmlns:author="urn:loc.gov:authors">
3   <title>Some title</title>
4   <author>
5     <author:title>Sir</author:title>
6     <author:name>Kraemer</author:name>
7   </author>
8 </book>
```

2.2 XPath

XPath [9] ist eine Sprache, mit der Teile eines XML-Dokuments adressiert werden können. Das Dokument wird dabei als Baum betrachtet und die einzelnen XML-Objekte als Knoten. XML-Objekte sind etwa Elemente, Attribute, Namensräume oder Texte [23].

Der Name XPath kommt von den wie Pfadausdrücke (*path expressions*) aufgebauten Anfragen. Diese Pfadausdrücke bestehen aus einer Reihe von *Lokalisierungsschritten*, welche jeweils durch '/' voneinander getrennt werden und auch mit '/' beginnen können. Ein einzelner Lokalisierungsschritt besteht aus der *Achse*, dem *Knotentest* sowie beliebig vielen *Prädikaten* und ist folgendermaßen aufgebaut: `Achse::Knotentest[Prädikat]`.

Die Achsen sind unterteilt in die *Vorwärtsachse* und die *Rückwärtsachse* und bestimmen die Richtung des Lokalisierungsschritts. Zu den Vorwärtsachsen gehören unter anderem `child`, `attribute` oder `descendant-or-self`, zu den Rückwärtsachsen etwa `parent` oder `ancestor-or-self`. Die `child`-Achse bezieht sich auf die Kindknoten und `attribute` auf die Attributknoten des aktuellen Knotens. `descendant-or-self` adressiert den aktuellen und alle darunter liegenden Knoten, `parent` den Elternknoten und `ancestor-or-self` den aktuellen und alle übergeordneten Knoten. Wird keine Achse angegeben bezeichnet das Trennzeichen '/' die `child`-Achse und '//' die `descendant-or-self`-Achse. Weitere Abkürzungen sind etwa '@' für die `attribute`-Achse oder '..' für die `parent`-Achse.

Durch die Angabe der Achse wird in jedem Lokalisierungsschritt eine bestimmte Knotenmenge selektiert, welche durch den Knotentest weiter eingeschränkt wird. Beim Knotentest können durch die Angabe eines Knotennamens alle Knoten mit diesem Namen gefiltert werden, oder es werden z. B. mit dem Knotentest `text()` alle Textknoten ausgewählt. Mit `*` oder `node()` werden in einem Knotentest alle Knoten der gewählten Achse angesprochen.

Als Abschluss eines Lokalisierungsschritts können beliebig viele Prädikate angegeben werden. Prädikate stehen hinter dem Knotentest in eckigen Klammern [] und enthalten einen logischen Ausdruck, der zu „wahr“ oder „falsch“ ausgewertet wird. Innerhalb eines logischen Ausdrucks können die gängigen

Vergleichsoperatoren <, <=, =, !=, >=, > verwendet und mit den Operatoren `and` bzw. `or` verknüpft werden. Als Ausnahme kann das Prädikat statt aus einem Ausdruck auch aus einer Zahl bestehen, welche den Knoten an der entsprechenden Position auswählt (z. B. [3] für den dritten Knoten).

Listing 2.3: Ein XML-Dokument.

```
1  <?xml version="1.1" encoding="UTF-8" ?>
2  <order id="34219">
3    <customer id="983" />
4    <items>
5      <item>
6        <id>32485</id>
7        <quantity>1</quantity>
8      </item>
9      <item>
10       <id>908345</id>
11       <quantity>3</quantity>
12     </item>
13   </items>
14   <payment>
15     <method>3</method>
16     <totalPrice>324.27</totalPrice>
17   </payment>
18 </order>
```

Zur Veranschaulichung werden im Folgenden eine Reihe von Beispiel-Anfragen betrachtet, welche alle auf dem in Listing 2.3 dargestellten XML-Dokument ausgeführt werden.

- `/*` selektiert das Wurzelelement *order*.
- `//item` besteht aus einem Lokalisierungsschritt, welcher vollständig ausgeschrieben `/descendant-or-self::item` lautet. Er bezieht sich auf alle Elemente mit dem Namen *item* der `descendant-or-self`-Achse beginnend beim Wurzelelement und selektiert die beiden *item*-Elemente in den Zeilen 5-8 sowie 9-12.
- `//item[1]` oder `//item[position() = 1]` selektiert das erste *item*-Element der `descendant-or-self`-Achse (Zeilen 5-8).

- `/order/items/item[/id/text() = 908345]` besteht aus drei Lokalisierungsschritten und selektiert durch das Durchlaufen von `child`-Achsen ausgehend vom Wurzelelement ein bestimmtes *item*-Element. Dieses muss einen Kindknoten *id* besitzen, welcher den Wert *908345* enthält (Zeilen 9-12).

- `//customer/@id/text()` selektiert die Werte aller Attribute mit dem Namen *id* aller *customer*-Elemente der `descendant-or-self`-Achse.

2.3 XQuery

Die von XPath bereitgestellten Möglichkeiten eignen sich bereits sehr gut, um Daten aus XML-Dokumenten zu selektieren. Allerdings fehlt in XPath die nötige Funktionalität, um neue XML-Dokumente generieren oder mithilfe von Variablen, Schleifen und Bedingungen komplexere Anfragen ausführen zu können. Um diese Funktionalität zu bieten wurde *XQuery* (*XML Query Language* [19]) entwickelt. Eine Anfrage in XQuery ist dabei in den optionalen *Prolog* und den *QueryBody* unterteilt.

Prolog Der Prolog enthält eine Reihe von Deklarationen und Importen, wobei zu den Deklarationen Variablen- und Funktionsdeklarationen gehören, welche zum Verständnis dieser Arbeit benötigt werden. Ein Beispiel für eine Variablendeklaration lautet:

```
1 declare variable $x as xs:integer := 9;
```

Dabei wird die Variable `$x` vom Typ Integer deklariert und ihr der Wert 9 zugewiesen. Die Angabe eines Typs ist dabei optional. Statt einen Wert für eine Variable anzugeben, kann diese auch als *external* definiert werden, wodurch der Wert von einer externen Umgebung bereit gestellt werden muss.

```
1 declare variable $y external;
```

Auch für externe Variablen kann wie zuvor ein Typ angegeben werden, der in diesem Beispiel allerdings nicht festgelegt wird.

Funktionen besitzen einen Namen und müssen in einem Namensraum liegen. Für eine Funktion können beliebige Parameter definiert werden, für die optional ein Typ angegeben werden kann. Alle Parameter sind dabei nur im *Körper*

(*Body*) der Funktion gültig. Der *Funktions-Body* darf aus einer beliebigen Anfrage bestehen, welche erst im nächsten Abschnitt näher erläutert werden. Das folgende Beispiel zeigt eine sehr einfache Funktion namens *add*. Diese liegt innerhalb des Namensraums *local* und definiert zwei Integer-Parameter `$a` und `$b`, deren Summe im *Body* der Funktion berechnet und zurück gegeben wird.

```
1 declare function local:add($a as xs:integer, $b as xs:integer) {
2   return $a + $b;
3 };
```

Alle im Prolog definierten Variablen und Funktionen können innerhalb des *QueryBody* verwendet werden.

QueryBody Im *QueryBody* befindet sich die eigentliche Anfrage, welche aus mehreren Ausdrücken bestehen kann. Den wichtigsten Ausdruck stellt dabei die so genannte *FLWOR-Expression* dar. Zur Veranschaulichung soll eine Beispiel-Anfrage auf das bereits für XPath verwendete XML-Dokument dienen (siehe Listing 2.3 auf Seite 7):

```
1 for $items in doc("order.xml")//item
2 let $itemID := $items/item/id
3 where $items/quantity/text() > 2
4 order by $itemID
5 return $itemID
```

Im Detail besteht diese Anfrage aus den folgenden Teilen:

- `for` startet eine Iteration durch alle *item*-Knoten des Dokuments *order.xml*. Die restliche *FLWOR-Expression* wird dabei für jeden gefundenen Knoten ausgeführt.

- `let` ermöglicht eine Variablendefinition. In diesem Beispiel wird der Variable `$itemID` als Wert das Ergebnis des angegebenen Pfadausdrucks (Zeile 2) zugewiesen.

- `where` unterstützt eine Filterung der gefundenen Knoten, indem eine Bedingung für diese angegeben wird. In diesem Fall sollen nur Artikel mit einer Bestellmenge größer als 2 in die Ergebnisliste aufgenommen werden.

- `order by` lässt eine Sortierung der Ergebnisse zu, hier wird nach der Artikel-ID sortiert.
- `return` gibt die Ergebnisse zurück.

Das Ergebnis der gestellten Anfrage sieht dabei folgendermaßen aus:

```
1 <id>908345</id>
```

Auch ein konditionaler Ausdruck steht in XQuery zur Verfügung:
`if(`*Condition*`) then` *Expression* `else` *Expression*
Damit können wie in anderen Sprachen anhand von Bedingungen unterschiedliche Zweige eines Programms erreicht werden. Als Ausdruck (*Expression*) kann dabei ein einzelner Ausdruck oder eine Sequenz von Ausdrücken stehen, wobei Sequenzen durch Kommata voneinander getrennt und in () eingeschlossen werden. Dabei muss zwingend ein `else`-Statement angegeben werden, auch wenn dieses leer ist und somit aus der leeren Sequenz () besteht.

Des Weiteren bietet XQuery vieles mehr, was für ein Verständnis dieser Arbeit nicht benötigt wird. Dazu gehören zahlreiche vordefinierte Funktionen wie etwa `doc(,,URI“)`, reguläre Ausdrücke, das Arbeiten mit Daten, Uhrzeiten und Zeitspannen, etc.

2.4 Compilerbau

Aho und andere [1] beschreiben ausführlich den Aufbau und die Funktionsweise eines Compilers. Dieser Abschnitt fasst daraus die zum Verständnis dieser Arbeit wesentlichen Punkte zusammen.

Die Aufgabe eines Compilers besteht hauptsächlich darin, ein Quellprogramm, das in einer bestimmten Sprache geschrieben wurde, in ein gleichwertiges Zielprogramm einer anderen Sprache umzuwandeln (siehe Abb. 2.1). Eine wichtige Aufgabe des Compilers besteht unter anderem darin, den Benutzer auf Fehler innerhalb seines geschriebenen Quellprogramms hinzuweisen.

Die Umwandlung eines Quellprogramms in ein Zielprogramm ist in die beiden Teile *Analyse* und *Synthese* untergliedert. Die Analyse zerlegt ein Quellprogramm in seine Bestandteile und generiert eine Zwischendarstellung des Quellprogramms. In der Synthese wird anschließend das Zielprogramm aus

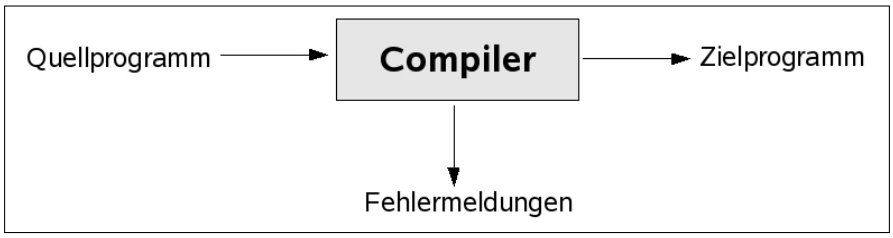

Abbildung 2.1: Ein Compiler. (In Anlehnung an [1])

dieser Zwischendarstellung erzeugt. Die Analyse wird auch als *Frontend* des Compilers bezeichnet und die Synthese als *Backend*.

In Abbildung 2.2 auf Seite 12 wird der Aufbau eines Compilers samt der Unterteilung in die verschiedenen Phasen dargestellt. Die einzelnen Phasen eines Compilers werden im Folgenden näher erläutert.

Frontend

Das Frontend besteht aus der *lexikalischen Analyse*, der *Syntaxanalyse* und der *semantischen Analyse*. Diese Phasen werden in den nächsten Abschnitten genauer beschrieben.

Lexikalische Analyse Diese Phase wird von einem so genannten *Lexer* durchgeführt. Sie liest den Quellcode des Quellprogramms ein und konvertiert diesen in einzelne Symbole (*Tokens*), welche an die Syntaxanalyse weitergegeben werden. Ein Token stellt eine logisch zusammen gehörende Folge von Zeichen dar, etwa ein Schlüsselwort der Sprache wie z. B. `for` in XQuery.

Innerhalb der lexikalischen Analyse kann zudem bereits Whitespace (Leerzeichen, Kommentare, etc.) aus dem Quellcode entfernt sowie eine einfache syntaktische Analyse durchgeführt werden.

Syntaxanalyse Die Syntaxanalyse wird auch als *Parsen* des Quellprogramms bezeichnet und wird durch den so genannten *Parser* vorgenommen. Sie fasst die ihr übergebenen Tokens zu grammatikalischen Sätzen der Quellsprache zusammen, welche in der Regel durch einen *Abstrakten Syntaxbaum* (*AST*) dargestellt werden. Lässt sich der übergebene Token-Strom nicht mit der Grammatik der Sprache produzieren, wird ein entsprechender Syntaxfehler ausgegeben. Ein Beispiel für einen Syntaxfehler ist der Aufruf einer Funktion mit weniger Parametern als in der Signatur der Funktion definiert sind.

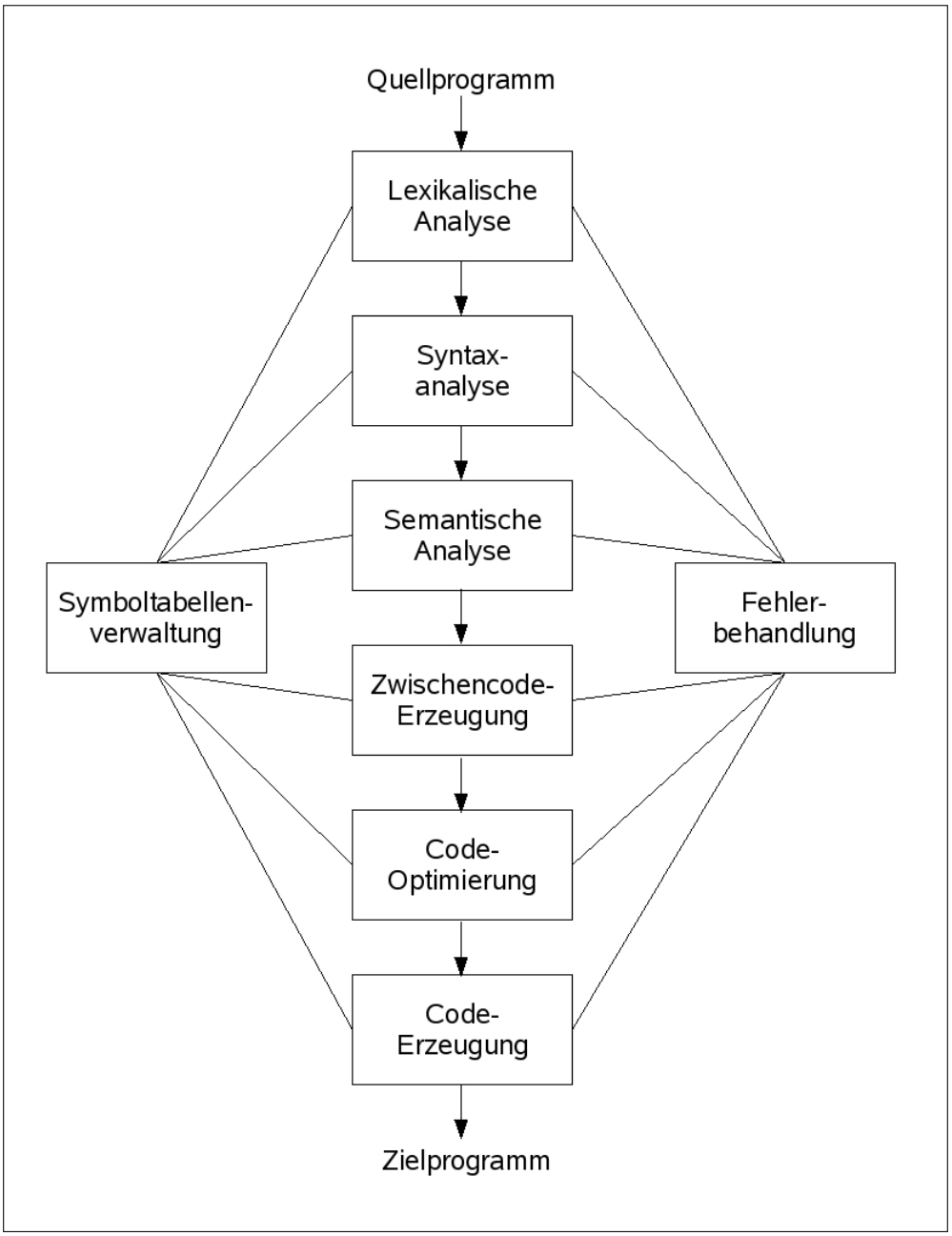

Abbildung 2.2: Die Phasen eines Compilers. (In Anlehnung an [1])

Semantische Analyse In dieser Phase wird das Quellprogramm auf semantische Fehler untersucht. Diese Fehlersuche wird auf dem AST ausgeführt, der durch den Parser erstellt wurde. Ein Beispiel für einen semantischen Fehler ist z. B. ein Funktionsparameter, welcher einen falschen Typ hat (etwa *String* statt *Integer*).

Backend

Im Backend sind die Phasen *Zwischencode-Erzeugung*, *Code-Optimierung* und *Code-Erzeugung* enthalten, welche im Folgenden vorgestellt werden.

Zwischencode-Erzeugung In manchen Compilern wird nach der syntaktischen und semantischen Analyse eine Zwischendarstellung des Quellprogramms erzeugt. Dieser Zwischencode wird so gewählt, dass die nachfolgenden Phasen damit möglichst effektiv arbeiten können.

Code-Optimierung Diese Phase versucht den Zwischencode zu optimieren, um ein effizienteres Zielprogramm zu erhalten. Aufwändige Verbesserungen können dabei viel Zeit kosten, im Optimalfall wird die Laufzeit des Zielprogramms deutlich verbessert ohne dabei die Kompilierzeit zu stark zu verlangsamen. Das Optimierungs-Potenzial hängt dabei allerdings wesentlich von der Quellsprache ab.

Code-Erzeugung Die letzte Phase erzeugt aus dem optimierten Zwischencode den Zielcode. Dieser wird für die verwendete Laufzeitumgebung erzeugt und besteht gewöhnlich aus Maschinen- oder Assemblercode.

Weitere Aufgaben

Zwei weitere Bereiche des Compilers sind die Symboltabellenverwaltung und die Fehlerbehandlung. Diese stehen in einer Wechselbeziehung zu den bereits genannten Compiler-Phasen.

Symboltabellenverwaltung Die Aufgabe eines Compilers ist es unter anderem, die im Quellprogramm enthaltenen Bezeichner (Variablen, Funktionsnamen, etc.) samt zugehörigen Informationen abzuspeichern. Diese Informationen sind z. B. Typ, Speicherbedarf oder Gültigkeitsbereich eines Bezeichners.

In der lexikalischen Analyse erkannte Bezeichner werden in die Symboltabelle eingetragen. Die restlichen Phasen des Compilers tragen weitere Informationen zu den Bezeichnern in die Symboltabelle ein und lesen bei Bedarf benötigte Informationen daraus aus.

Fehlerbehandlung In jeder Phase eines Compilers können Fehler auftreten, die dem Benutzer mitgeteilt werden sollten. Im Optimalfall läuft die Übersetzung nach Entdeckung eines Fehlers weiter, um noch andere Fehler im Quellprogramm entdecken zu können. Ein Großteil der Fehler wird dabei von der syntaktischen sowie der semantischen Analyse behandelt. Manche Fehler treten allerdings erst zur Laufzeit eines Programms auf und können daher durch die Fehlerbehandlung des Compilers nicht erkannt werden.

2.5 Demaq

Demaq [2] steht für **De**clarative **M**essaging **A**nd **Q**ueuing und stellt eine Umgebung zur Entwicklung verteilter Anwendungen bereit. Diese Demaq-Anwendungen basieren dabei auf dem Austausch von asynchronen XML-Nachrichten und weisen somit Ähnlichkeiten mit Web Services auf. Die Besonderheit dabei ist, dass Demaq mit *DQL* (*Demaq Query Language*) eine deklarative Sprache verwendet. Andere deklarative Sprachen sind beispielsweise *SQL* (*Structured Query Language* [6]) oder *XQuery*, wobei Letztere als Grundlage für DQL verwendet wird.

In den folgenden Abschnitten werden die einzelnen Bestandteile von Demaq genauer beschrieben. Diese Abschnitte sind dazu unterteilt in das Demaq-Programmiermodell, die Demaq Query Language, die Fehlerbehandlung und das Laufzeitsystem von Demaq.

2.5.1 Programmiermodell

Die Anwendungsentwicklung in Demaq unterscheidet sich deutlich von einer Entwicklung mit imperativen Programmiersprachen wie Java oder C++. Das liegt zum einen an der direkten Unterstützung von XML. Das Grundkonzept von Demaq besteht darin, aus eingehenden XML-Nachrichten neue XML-Nachrichten zu generieren, welche anschließend an externe Teilnehmer gesendet

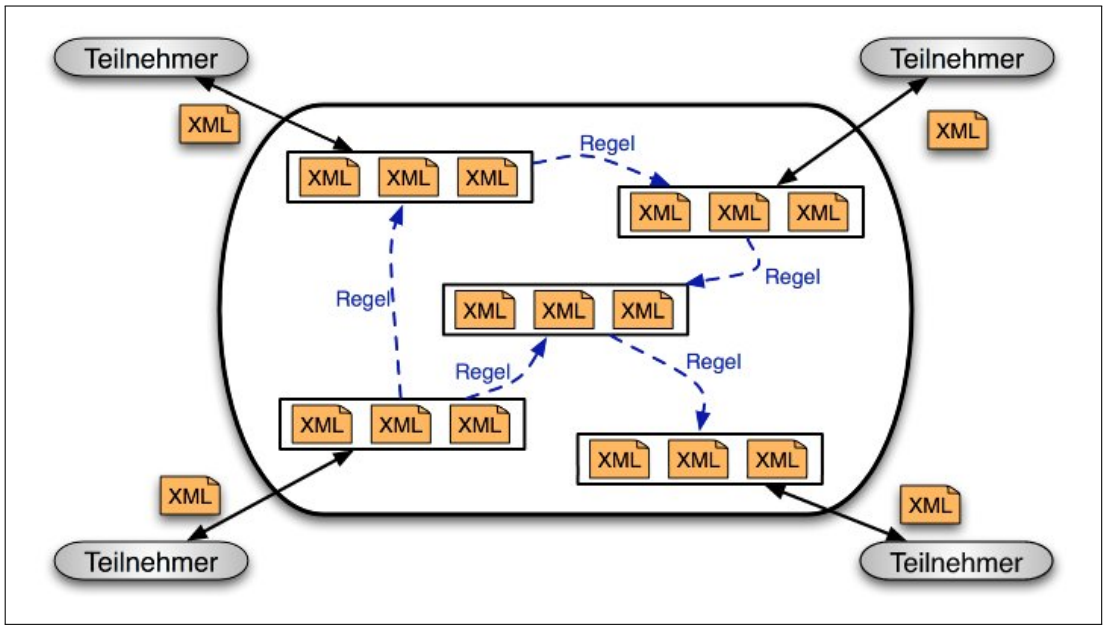

Abbildung 2.3: Demaq-Programmiermodell (aus [2])

werden können. Die Verwaltung der gesamten Nachrichten wird durch *Warteschlangen* (*Queues*) realisiert, während das Generieren neuer Nachrichten mithilfe von *Regeln* umgesetzt ist (siehe Abb. 2.3). Dabei werden ankommende Nachrichten zunächst in Queues gespeichert. Zum Versenden neuer Nachrichten an externe Teilnehmer müssen diese Nachrichten in ausgehende Queues eingefügt werden.

Zum anderen liegt der Unterschied in der deklarativen Programmierung mit DQL, welche sich grundsätzlich von einer imperativen Programmierung unterscheidet. Im Gegensatz zu imperativen Programmiersprachen wird beschrieben *was* eine Anwendung leisten soll, aber nicht zwingend *wie* sie es leisten soll. Unter anderem ermöglicht dieser Ansatz eine Optimierung von Anwendungen durch das Laufzeitsystem, wodurch die Laufzeit von Anwendungen verbessert werden kann.

2.5.2 Demaq Query Language (DQL)

Die *Demaq Query Language* ist in zwei Bereiche unterteilt, in *QDL* (*Queue Definition Language*) und *QRL* (*Queue Rule Language*). Mit QDL wird die benötigte Infrastruktur einer Demaq-Anwendung angelegt und QRL realisiert

die Anwendungslogik. Im Folgenden werden diese beiden Teile der Demaq-Sprache genauer vorgestellt.

Queue Definition Language (QDL)

Die QDL besteht aus den Konstrukten *Queue*, *Property* und *Slicing*. Die Verwendung dieser Konstrukte zur Anlegung einer Infrastruktur für eine Demaq-Anwendung wird in den kommenden Abschnitten beschrieben.

Queues Als Datenstruktur werden in Demaq ausschließlich Queues verwendet, welche in zwei verschiedene Arten unterteilt sind. *Basic-Queues* dienen als interner Nachrichtenspeicher während *Gateway-Queues* zur Kommunikation mit externen Systemen verwendet werden. Für alle Queues muss dabei angegeben werden, ob die enthaltenen Daten persistent oder flüchtig (*transient*) gespeichert werden sollen. Flüchtig gespeicherte Nachrichten können beim Auftreten von Fehlern, bei Systemabstürzen, etc. verloren gehen während persistent gespeicherte Nachrichten erhalten bleiben. Jeder Queue kann weiterhin eine Priorität sowie eine Schema-Definition zugewiesen werden. Mit der Angabe einer Priorität wird gefordert, dass die Nachrichten in einer Queue mit einer höheren Priorität vor den Nachrichten in einer Queue mit einer geringeren Priorität verarbeitet werden. Wird keine Priorität angegeben wird sie auf den Standardwert 0 gesetzt. Durch die Angabe eines XML-Schemas [10] kann zudem verlangt werden, dass alle in diese Queue eingefügten Nachrichten diesem Schema entsprechen müssen. In den folgenden Abschnitten wird eine einfache Basic-Queue vorgestellt sowie die Funktionsweise der Gateway-Queues näher erläutert.

Basic-Queues Das folgende Beispiel erzeugt eine einfache Basic-Queue mit dem Namen *customers* und persistenter Nachrichtenspeicherung.

```
1 create queue customers kind basic mode persistent priority 12 validate schema.xsd;
```

Um eine flüchtige Datenspeicherung zu erreichen müsste die Queue mit „`mode` transient“ erzeugt werden. Zudem wird der Queue die Priorität 12 zugewiesen und alle darin eingehenden Nachrichten müssen dem durch die zugewiesene .xsd-Datei vorgegebenen Schema entsprechen. Innerhalb derselben Anwendung

darf keine weitere Queue namens *customers* definiert werden, da alle gleichartigen Konstrukte einer Anwendung eindeutig sein müssen.

Gateway-Queues Mithilfe von Gateway-Queues wird die Kommunikation einer Demaq-Anwendung mit externen Systemen ermöglicht. Ankommende Nachrichten von externen Systemen werden dabei durch eingehende Gateway-Queues (*incoming*) behandelt. Für den Versand von Nachrichten an externe Systeme werden diese Nachrichten innerhalb einer Demaq-Anwendung in ausgehende Gateway-Queues (*outgoing*) eingefügt. Der folgende Code-Ausschnitt zeigt ein Beispiel einer *incoming*-Queue namens *requests*.

```
1 create queue requests kind incoming interface "interface.wsdl" port "somePort"
    mode persistent;
```

Im Gegensatz zu einer Basic-Queue sind mit *interface* und *port* zwei neue Attribute hinzu gekommen und der *Queuetyp* (*kind*) wurde auf *incoming* geändert. Um festzulegen, welche Art von Nachrichten von dieser Queue wie empfangen werden, ist im gezeigten Beispiel eine *WSDL* (*Web Service Description Language* [8]) - Datei angegeben. In der aktuellen Version von Demaq werden WSDL-Dateien allerdings nicht unterstützt. Deshalb wird das *interface* zur Angabe eines Transportprotokolls und der *port* zur Angabe eines Ports benutzt. Als Transportprotokoll können dabei *HTTP* (*Hypertext Transfer Protocol*) oder *SMTP* (*Simple Mail Transfer Protocol*) verwendet werden.

Nach demselben Schema werden durch eine Änderung des Queuetyps auf *outgoing* ausgehende Gateway-Queues definiert. Die Werte der benötigten Versandparameter können den in *outgoing*-Queues eingefügten Nachrichten auch in der Form von *Properties* (siehe übernächster Abschnitt) zur Laufzeit übergeben werden. Dabei muss immer das zu verwendende Transportprotokoll angegeben werden. Weiterhin werden für einen HTTP-Versand ein *Uniform Resource Locator* (*URL*) und ein Port benötigt, für einen Versand über SMTP müssen zusätzlich eine Absender- und eine Empfänger-Adresse übergeben werden.

Response-Queues Im Gegensatz zum eigentlich asynchronen Datenaustausch in Demaq-Anwendungen wird von manchen Transportprotokollen (z. B. HTTP) ein synchroner Datenaustausch verlangt. Zur Unterstützung dieser Art des Nachrichtenaustauschs kann zu einer *incoming*-Queue eine *Response*-Queue

definiert werden. In diese Response-Queue eingefügte Nachrichten werden automatisch mit der eingehenden Anfrage in Beziehung gesetzt und über dieselbe Verbindung verschickt. Die Response-Queues werden automatisch erzeugt, somit darf innerhalb einer Anwendung keine weitere Queue mit demselben Namen existieren. Im folgenden Beispiel lauscht die Gateway-Queue *requests* am Port 80 auf eingehende HTTP-Anfragen. Antwortnachrichten werden über die zugehörige Response-Queue *replies* verschickt.

```
1 create queue requests kind incoming interface "http" port "80" response replies
      mode persistent;
```

Properties Jede von Demaq gespeicherte Nachricht kann zusätzlich mit *Properties* versehen werden. Eine solche Property stellt ein Paar aus einem Wert und einem eindeutigen Schlüssel dar und wird für alle Nachrichten in einer bestimmten Queue definiert. Das folgende Beispiel zeigt die einfachste Art, eine Property zu definieren.

```
1 create queue customers kind basic mode persistent;
2 create property customerID queue customers;
```

Es wird eine Queue *customers* erzeugt und darauf die Property *customerID* definiert. Hierbei muss der Property innerhalb einer Regel (siehe späterer Abschnitt zu QRL) ein Wert zugewiesen werden. Weitere Arten sind *computed*, *fixed* und *inherited* Properties.

Computed Properties müssen nicht manuell definiert, sondern können vom System berechnet werden. Berechnete Properties können allerdings durch Manuelle überschrieben werden. Enthalten die in die Queue *customers* eingefügten Nachrichten bereits die Kunden-ID, so kann diese mit einem entsprechenden XQuery-Ausdruck der Property als Wert zugewiesen werden:

```
1 create property customerID queue customers value /customer/ID;
```

Properties können für mehrere Queues auf einmal definiert werden. Dies kann wie folgt für denselben XQuery-Ausdruck geschehen:

```
1 create property customerID queue customers, orders value //customer/ID;
```

Ist der Property-Wert in den verschiedenen Queues nur über unterschiedliche XQuery-Ausdrücke zu erreichen, kann dies folgendermaßen realisiert werden:

```
1 create property customerID
2   queue customers value /customer/ID
3   queue orders value /order/customerID;
```

Fixed Properties können das Überschreiben von *Computed Properties* verhindern, indem diese zusätzlich als *fixed* definiert werden:

```
1 create property customerID
2   queue customers fixed value /customer/ID
3   queue orders value /order/customerID;
```

Mithilfe von *Inherited Properties* kann erreicht werden, dass alle Properties einer Nachricht an die davon abgeleiteten Nachrichten vererbt werden.

```
1 create property customerID queue customers inherited;
```

Inherited Properties können zusätzlich als *fixed* definiert werden, damit sie nicht manuell überschrieben werden können.

Slicings In einem *Slicing* können Nachrichten aus einer Queue anhand ihrer Properties zu einer Sequenz von Nachrichten (*Slice*) zusammengefasst werden. Die einzelnen Slices fassen dabei jeweils Nachrichten zusammen, deren Properties denselben Wert haben. Das folgende Code-Beispiel zeigt die Erstellung eines Slicings *itemSlice* basierend auf der Property *category*.

```
1 create property category queue items;
2 create slicing itemSlice on category require fn:false();
```

Enthält die Queue *items* beispielsweise 3 Nachrichten, so ist jeder Nachricht eine eindeutige Nachrichten-ID sowie eine Kategorie zugeordnet. Das kann sich etwa folgendermaßen darstellen: (1, book), (2, book), (3, music). Somit werden basierend auf den unterschiedlichen Properties *book* und *music* zwei Slices generiert. Das Slice für den so genannten *Slicekey book* enthält die Nachrichten-Sequenz (1, book), (2, book) und das Slice für den Slicekey *music* die Sequenz (3, music).

Slicings können somit benutzt werden, um miteinander in Beziehung stehende Nachrichten zu gruppieren, die auf verschiedene Queues verteilt sind. Als Beispiel könnten in einem Shop alle Nachrichten eines bestimmten Kunden in einem Slicing zusammengefasst werden. Realisieren lässt sich dies, indem

ein Slicing auf eine Property definiert wird, welche wiederum allen benötigten Queues zugewiesen wurde.

Für jedes Slicing muss zudem mit dem *require*-Attribut durch einen DQL- oder XQuery-Ausdruck festgelegt werden, wie lange die Nachrichten in diesem Slicing aufbewahrt werden sollen. Die im Beispiel zuvor gewählte Variante *fn:false()* hebt die Nachrichten für immer auf. Mithilfe der Funktion *qs:history()* kann z. B. eine bestimmte Anzahl von Nachrichten festgelegt werden oder mit *fn:current-dateTime* eine gewisse Dauer.

Queue Rule Language (QRL)

Mit QRL wird die Anwendungslogik einer Demaq-Anwendung durch die Definition von Regeln erstellt. Um auf existierende DQL-Konstrukte zuzugreifen, können innerhalb von Regeln verschiedene Funktionen verwendet werden, welche einen Datenzugriff auf diese Konstrukte ermöglichen.

Regeln Regeln werden immer auf eine Queue oder ein Slicing definiert und werden jedes Mal ausgewertet, sobald eine neue Nachricht in diese Queue oder dieses Slicing eingeführt wird. Die Nachricht, welche die Auswertung einer Regel auslöst, wird innerhalb der Regel als *Kontext-Nachricht* bezeichnet. Als Resultat einer Regel wird immer eine (möglicherweise leere) Sequenz von XML-Nachrichten erstellt, welche in bestehende Queues eingefügt werden. Durch das Einfügen einer Nachricht in eine Gateway-Queue kann dabei die Kommunikation mit einem externen System angestoßen werden. Ein Beispiel für eine Regel sieht folgendermaßen aus:

```
1  create rule processOrder for orders
2    let $custID := /order/customerID/text()
3    let $items := /order/items
4    for $itemID in $items/item/ID/text()
5    return
6      enqueue message
7        <invokeShipping>
8          <itemID>{$itemID}</itemID>
9          <customerID>{$custID}</customerID>
10       </invokeShipping>
11     into shipping
12 ;
```

Für die Regel wird zunächst ein eindeutiger Name vergeben (*processOrders*) sowie die Queue (oder das Slicing), auf der (dem) sie definiert ist (*orders*). Anschließend folgt der *Regel-Body*, welcher aus einem XQuery-Ausdruck samt Demaq-spezifischen Erweiterungen (z. B. *enqueue message*) besteht. Diese Erweiterungen sind nötig, da mit XQuery keine Daten verändert werden können. Deshalb ist die *XQuery Update Facility* [5] in Demaq eingebunden, um eine Veränderung von Daten zu ermöglichen. Der XQuery-Ausdruck der vorgestellten Regel wird verwendet, um eine XML-Nachricht (*Message*) zu erzeugen, welche mithilfe von „`enqueue message` *Message* `into` *TargetQueue*“ in eine Queue eingefügt werden kann. Im angeführten Beispiel werden aus einer eingehenden Bestellung zunächst die Kunden-ID (Zeile 2) sowie die bestellten Artikel (Zeile 3) in Variablen gespeichert. Dabei kann durch die Angabe von Pfadausdrücken auf die Kontext-Nachricht zugegriffen werden. Anschließend (Zeilen 4-11) werden alle bestellten Artikel durchlaufen und für jeden aus der Artikel-ID zusammen mit der Kunden-ID eine neue XML-Nachricht erzeugt, welche in die Queue *shipping* eingefügt wird. Innerhalb des XQuery-Ausdrucks können auch im Prolog definierte Variablen und Funktionen verwendet werden.

Das folgende Beispiel demonstriert das Hinzufügen von Properties zu einer Nachricht, wie es im Abschnitt QDL bereits angesprochen wurde.

```
1  create queue soap kind basic mode persistent;
2  create queue outgoingSOAP kind outgoing interface "" port "" mode persistent;
3  create rule sendSOAP for soap
4    enqueue message
5      <s:Envelope xmlns:s="http://www.w3.org/2001/12/soap-envelope">
6        <s:Header>
7        </s:Header>
8        <s:Body>
9          {.}
10       </s:Body>
11     </s:Envelope>
12   into outgoingSOAP
13   with comm:URL value "http://www.example.org"
14   with comm:TransportProtocol value "comm:HttpPost"
15   with comm:DestinationPort value "8080"
16 ;
```

Die Regel *sendSOAP* (Zeilen 3-16) erzeugt aus allen in die Queue *soap* (Zeile 1) eingefügten Nachrichten eine SOAP-Nachricht [14], welche anschließend in

die Gateway-Queue *outgoingSOAP* (Zeile 2) eingefügt wird. In den *Body* der SOAP-Nachricht wird dabei die Kontext-Nachricht eingefügt (Zeile 9), welche durch "." angesprochen werden kann. Mithilfe von „`with` *PropertyName* `value` *PropertyValue*“ können einer erzeugten Nachricht manuell Properties mitgegeben werden. Dazu wird den Properties (*PropertyName*) ein Wert (*PropertyValue*) zugewiesen. In diesem Beispiel werden der Nachricht die nötigen Versandparameter (URL, Port, Protokoll) mitgegeben (Zeilen 13-15), weshalb sie innerhalb der Gateway-Queue nicht definiert werden müssen.

Datenzugriff Um innerhalb von Regeln Kontextinformationen zu bekommen ist es nötig auf beliebige Queues, Slicings oder Properties der Anwendung zugreifen zu können. Dazu stehen eine Reihe Funktionen zur Verfügung, welche in der folgenden Auflistung vorgestellt werden.

- Mit `qs:queue(''QueueName'')` können alle Nachrichten einer Queue ausgelesen werden.
- `qs:slicekey(''SlicingName'')` ermöglicht den Zugriff auf den *Slicekey* der Kontext-Nachricht.
- Die Funktion `qs:slice(''Slicekey'', ''SlicingName'')` liefert alle Nachrichten von dem *Slice* des Slicings *SlicingName*, welches sich auf den angegebenen *Slicekey* bezieht.
- Mithilfe von `qs:property(''PropertyName'', ContextMessage)` kann der Wert der Property *PropertyName* der angegebenen Kontext-Nachricht ausgelesen werden.
- `qs:message()` enthält die aktuelle Kontext-Nachricht und kann verwendet werden, falls sich innerhalb einer Regel der Kontext verändert (etwa in einem Pfadausdruck) und somit "." nicht mehr eindeutig ist.

2.5.3 Fehlerbehandlung

Wird durch eine Nachricht ein Fehler ausgelöst, wird eine Fehlernachricht in die entsprechende *Errorqueue* eingefügt. Diese Errorqueue muss wie eine gewöhnliche Queue erzeugt werden. Die folgenden Abschnitte behandeln die

drei möglichen Arten, auf welche diese zugehörige Errorqueue definiert werden kann. Trifft keiner dieser Fälle zu, wird die Fehlernachricht in der Demaq System-Queue (*demaq:systemMessages*) gespeichert, welche alle systembezogenen Meldungen enthält.

Errorqueue für Regeln

Zur Fehlerbehandlung aller von einer Regel verarbeiteten Nachrichten kann für eine Regel eine Errorqueue angegeben werden. Dadurch werden Fehler, die beim Verarbeiten einer Nachricht durch diese Regel auftreten, als Fehlernachricht in diese Errorqueue eingefügt. Im folgenden Beispiel werden alle von der *someRule*-Regel verursachten Fehler als Fehlernachricht an die Errorqueue *errors* gesendet.

```
1 create queue errors kind basic mode persistent;
2 create queue someQueue kind basic mode persistent;
3
4 create rule someRule for someQueue errorqueue errors
5   ...
6 ;
```

Errorqueue für Queues

Um die auftretenden Fehler für alle Nachrichten in einer Queue zu behandeln, kann jeder Queue eine Errorqueue zugewiesen werden. Das heißt die Fehlernachrichten für alle durch Nachrichten dieser Queue verursachten Fehler werden in dieser Errorqueue gespeichert. Wurde der Fehler in einer Regel verursacht, welche eine eigene Errorqueue definiert, dann wird die Fehlermeldung in die Errorqueue dieser Regel eingefügt. Das folgende Beispiel demonstriert die Fehlerbehandlung der Queue *someQueue* durch die Errorqueue *errors*.

```
1 create queue errors kind basic mode persistent;
2 create queue someQueue kind basic mode persistent errorqueue errors;
```

Globale Errorqueue

Im XQuery-Prolog, der in jeder Demaq-Anwendung enthalten ist, kann eine globale Errorqueue für die Anwendung definiert werden. In diese Errorqueue werden die Fehlernachrichten aller Queues und Regeln eingefügt, für welche

keine eigene Errorqueue angegeben ist. Die Definition einer globalen Errorqueue namens *globalErrors* geschieht mit folgendem Befehl:

```
1 declare default errorqueue globalErrors;
```

Diese Queue muss durch ein entsprechendes `create queue`-Statement in der Anwendung auch erzeugt werden:

```
1 create queue globalErrors kind basic mode persistent;
```

2.5.4 Der Demaq-Compiler

Der Demaq-Compiler unterscheidet sich in den verschiedenen Phasen und deren Aufbau von herkömmlichen Compilern. Dieser Abschnitt erläutert die Funktionsweise des Demaq-Compilers indem auf seine einzelnen Phasen eingegangen wird, welche unten im Text durch Fettschrift hervorgehoben sind. Danach wird auf die Unterschiede zu einem gewöhnlichen Compiler, wie er in Unterkapitel 2.4 vorgestellt wurde, kurz eingegangen.

Der Demaq-Compiler beginnt ebenfalls mit der **lexikalischen Analyse**. Hierbei wird das in DQL geschriebene Quellprogramm in einzelne Tokens zerlegt, welche an die **Syntaxanalyse** weitergegeben werden. Diese erzeugt daraus einen AST, auf welchem die restlichen Phasen arbeiten. Die anschließende **Rewrite-Phase** durchläuft diesen AST mehrfach und nimmt dabei die semantische Analyse sowie Optimierungen und Normalisierungen vor. Eine Optimierung ist dabei etwa das Zusammenfassen von Regeln, welche auf dieselbe Queue definiert sind (sofern diese eine gemeinsame Errorqueue haben). Eine Normalisierung ist z. B. das Hinzufügen des Queuenamen zu den `qs:queue()`-Aufrufen, bei denen dieser ausgespart wurde, da es sich dabei um eine erlaubte syntaktische Abkürzung handelt. Durch die Optimierung und Normalisierung wird der AST in einen neuen, logisch äquivalenten AST umgeschrieben. In der **Code-Erzeugungs-Phase** wird dieser AST schließlich in eine XML-Repräsentation (*DQLX*) umgewandelt. Diese enthält für jedes DQL-Konstrukt eine entsprechende Darstellung in XML und kann anschließend durch das Demaq Laufzeitsystem weiter verarbeitet werden.

Auch der Demaq-Compiler besitzt eine **Symboltabellenverwaltung** und eine **Fehlerbehandlung**, welche mit den oben genannten Phasen in einer

Wechselbeziehung stehen. Die Fehlerbehandlung gibt gefundene Fehler an den Benutzer aus und die Symboltabellenverwaltung verwaltet die in der Demaq-Sprache verfügbaren Bezeichner (Queuenamen, Regelnamen, etc.).

Besonders auffällig ist, dass hier keine klare Trennung zwischen Frontend und Backend des Compilers vorhanden ist. Stattdessen wird die semantische Analyse mit der Code-Optimierung in der Rewrite-Phase vermischt. Im Vergleich zu dem vorgestellten Aufbau eines Compilers (siehe Abb. 2.2 auf Seite 12) fehlt daher eine Zwischencode-Erzeugung.

2.5.5 Laufzeitsystem

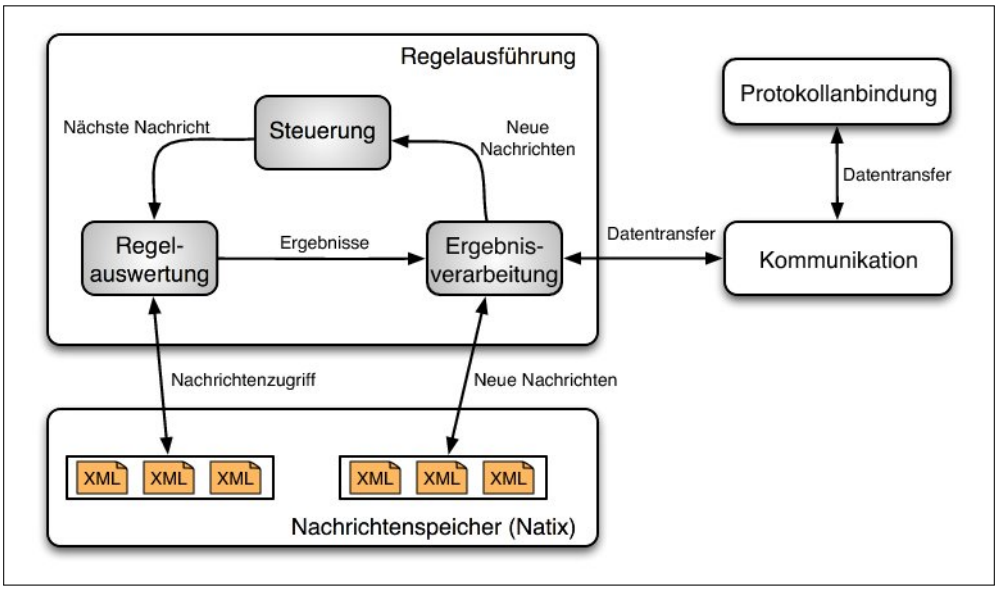

Abbildung 2.4: Das Demaq-Laufzeitsystem. (aus [2])

Um eine lauffähige Demaq-Instanz zu erzeugen, muss Demaq-Anwendungscode durch den Demaq-Compiler (vgl. 2.5.4) in einen Ausführungs-Plan übersetzt werden, den das Laufzeitsystem versteht. Die Architektur des Laufzeitsystems von Demaq ist in Abb. 2.4 dargestellt und besteht aus folgenden drei Bestandteilen:

- Einem Speicher für die XML-Nachrichten (hier: *Natix* [11]), welcher die Queues verwaltet und eine persistente Speicherung der Nachrichten ermöglicht.

- Der Regelausführung, welche aus der *Steuerung*, der *Regelauswertung* und der *Ergebnisverarbeitung* besteht. Die Steuerung bestimmt dabei, welche Nachricht als nächstes bearbeitet wird. Darauf wertet die Regelauswertung die Regeln aus, welche durch diese Nachricht ausgelöst werden, und erstellt eine Liste aller daraus resultierenden Aktionen. Diese Liste wird an die Ergebnisverarbeitung weitergegeben, in der alle Aktionen, wie z. B. das Einfügen einer Nachricht in eine Queue, ausgeführt werden.

- Einer Kommunikations-Komponente, welche den Datenaustausch mit externen Systemen über die Protokolle HTTP und SMTP ermöglicht.

2.6 Wiederverwendung von Anwendungs-Code

Bei der Entwicklung von Anwendungen wird an vielen Stellen wiederkehrende Funktionalität benötigt. Ein Anliegen bei der Anwendungsentwicklung ist dabei, diese Funktionalität wiederverwendbar zu gestalten.

Die Wiederverwendung von Anwendungs-Code kann laut Rumbaugh und anderen [18] auf zwei unterschiedliche Arten erfolgen. Eine Variante ist die Wiederverwendung von Code innerhalb eines Projekts, wozu redundante Anwendungsteile zunächst erkannt werden müssen. Diese können anschließend durch in der verwendeten Programmiersprache enthaltene Mechanismen wiederverwendbar umgesetzt werden. Die andere Möglichkeit ist die Wiederverwendung von existierendem Code in neuen Projekten. Dabei werden Teilanwendungen wiederverwendet, in welchen für mehrere Projekte benutzbare Funktionalität realisiert wurde.

Bei der Wiederverwendung von Code spielt die Datenkapselung (*Information Hiding*) eine wichtige Rolle. Bereits 1971 wurde von Parnas [16] erkannt, wie wichtig die Kapselung von Daten ist. Anhand von Beispielen zeigt er, dass wiederverwendbare Anwendungsteile unabhängig sein sollten, wodurch auch eine bessere Verständlichkeit dieser Anwendungsteile erreicht wird. Ist diese Unabhängigkeit nicht gegeben, sind laut Parnas durch die Änderung eines Anwendungsteils ggf. Anpassungen in anderen Anwendungsteilen oder in der eigentlichen Anwendung nötig, von welcher diese Anwendungsteile eingebunden werden. Für ein besseres Verständnis wird diese Anwendung im Folgenden

Hauptanwendung genannt. Nach Wirth [25] kann durch diese Kapselung zudem die Korrektheit von Anwendungsteilen gewährleistet werden, da sie nicht durch einen unvorhersehbaren Zugriff von „Außen“ manipuliert werden können.

Um diese Datenkapselung zu erreichen, sollten die Anwendungsteile in einen sichtbaren und einen unsichtbaren Teil untergliedert sein. Diese Unterteilung wird etwa von Szyperski [22] auch als Blackbox beschrieben. Dabei ist ein Zugriff aus dem Hauptprogramm ausschließlich über die sichtbaren Teile möglich, während auf den unsichtbaren Teil nicht zugegriffen werden kann. Die sichtbaren Teile werden dabei in der Regel als Schnittstellen oder Spezifikation eines Anwendungsteils bezeichnet.

Im Folgenden werden einige Beispiele für Programmiersprachen und die darin verfügbaren Sprachkonzepte zur Wiederverwendung von Code vorgestellt. Betrachtet werden dabei exemplarisch die Sprachen Ada, Modula-2, XQuery, C++ und Java.

Ada Ada [3] bietet die Möglichkeit so genannte Pakete zu erstellen, welche in beliebigen Anwendungen oder anderen Paketen eingebunden werden können. Diese Pakete sind unterteilt in eine Paket-Spezifikation und einen Paket-Body, wobei diese Teile physikalisch voneinander getrennt und separat übersetzt werden können. Die Spezifikation beschreibt die Schnittstellen eines Pakets, welche die benutzbaren Teile eines Pakets definieren, während der Body die eigentliche Funktionalität bereitstellt.

Modula-2 In Modula-2 [25], dem Nachfolger von *Pascal* [24], ist jedes Programm ein Modul. Ein solches Modul besteht aus einem *Definitionsteil* und einem *Implementationsteil*, wobei der Implementationsteil gekapselt ist und der Definitionsteil die zugreifbaren Inhalte eines Moduls spezifiziert. Innerhalb des Implementationsteils müssen die dort definierten Inhalte implementiert sein. In die Hauptanwendung werden in der Regel mehrere Module eingefügt, welche wiederum selbst Module enthalten können. Eingebundene Module müssen dabei durch eine Import-Liste bekannt gegeben werden. Weiterhin besteht die Möglichkeit, Module in einer Programmbibliothek zusammenzufassen.

XQuery Auch XQuery [19] bietet die Möglichkeit, Module zu erzeugen und unter einem *URI* zur Verfügung zu stellen. Diese Module bestehen aus dem

Prolog von XQuery, wodurch alle Möglichkeiten des Prologs, wie etwa die Definition von Variablen und Funktionen, zur Verfügung stehen. Module enthalten zudem eine Moduldeklaration, die einen Namensraum für das Modul definiert, an welchen alle Inhalte des Moduls gebunden werden. Dadurch können Namenskonflikte mit gleichnamigen Inhalten in der Hauptanwendung oder anderen Modulen vermieden werden. Eingebunden werden die Module über einen Import-Befehl innerhalb des Prologs, somit können Module weitere Module importieren. XQuery-Module unterstützen allerdings keine Kapselung, auf alle Inhalte eines Moduls kann direkt zugegriffen werden. Die Module sind aber trotzdem unabhängig, da sie lediglich einer Sammlung von Variablen, Funktionen, etc. zur Verfügung stellen.

C++ Um in einem C++-Programm [21] Anwendungs-Code aus mehreren Quellen zusammenzufassen besteht die Möglichkeit durch *Includes* Code aus dem Dateisystem einzubinden. Dabei wird lediglich das *include*-Statement durch den Inhalt der jeweiligen Datei ersetzt. Hierdurch wird zwar eine Wiederverwendung von Code ermöglicht, allerdings keine Kapselung unterstützt. Das ist allerdings nicht nötig, da C++ eine objektorientierte Entwicklung [18] ermöglicht, welche bereits ausreichende Möglichkeiten zur Kapselung von Inhalten bietet. In den so genannten Klassen werden alle Inhalte (Variablen, Methoden) mit einer Zugriffsberechtigung versehen. Dabei erlaubt `public` beliebigen Zugriff, während z. B. `private` nur einen Zugriff durch Konstrukte aus derselben Klasse gestattet. Mit der *Standard Template Library* (*STL* [20]) steht in C++ zudem eine umfassende Bibliothek für häufig benötigte Funktionalität zur Verfügung.

Java Wie C++ bietet Java [13] durch die Objektorientierte Entwicklung bereits genügend Möglichkeiten zur Kapselung sowie zur Zugriffskontrolle von Inhalten. Um zusätzlich eine Wiederverwendbarkeit von Anwendungs-Code zu ermöglichen, können Implementierungen in Java einem Paket zugeordnet werden. Ein Paket kann Unterpakete enthalten, daher ist eine beliebig tiefe Schachtelung dieser Pakete zugelassen. In jeder Java-Anwendung können gezielt bestimmte Inhalte der verfügbaren Pakete importiert und verwendet werden. Mit der *Java Class Library* [7] wird zudem eine Reihe von System-Paketen zur Verfügung gestellt. Diese ermöglichen z. B. den Zugriff auf Zeit-

und Datums-Funktionen oder die Verarbeitung von XML-Dokumenten.

In den verschiedenen Sprachen werden unterschiedliche Möglichkeiten zur Wiederverwendung von Code bereitgestellt. Diese Möglichkeit sind elementar für eine effektive Anwendungsentwicklung, weshalb auch zahlreiche andere Sprache ähnliche Konzepte zur Verfügung stellen.

In den Beispielen reichen diese Konzepte von einer strengen Einhaltung der Datenkapselung bis zu einer bloßen Bereitstellung einer Funktionssammlung. Bestehende Konzepte zur Datenkapselung wurden zudem um die Funktionalität erweitert, Anwendungsteile aus verschiedenen Quellen einzubinden. Alle Konzepte ermöglichen eine Wiederverwendung von Anwendungsteilen innerhalb derselben sowie in verschiedenen Anwendungen. Weiterhin können Anwendungen mit allen Konzepten besser strukturiert und somit übersichtlicher gestaltet werden.

Wie bereits gesehen ist die Realisierung der Kapselung und Wiederverwendbarkeit dabei von der jeweiligen Sprache abhängig.

3 Anforderungsanalyse

Um die bestehenden Schwierigkeiten sowie die zu entwerfende Spracherweiterung von Demaq besser spezifizieren zu können, werden in diesem Kapitel in Unterkapitel 3.1 zunächst eine Reihe von Anwendungsfällen vorgestellt. Diese verdeutlichen jeweils verschiedene Probleme bei ihrer Umsetzung mit Demaq. Anhand dieser Anwendungsfälle werden in Unterkapitel 3.2 die funktionalen Anforderungen an die Spracherweiterung hergeleitet. Im Anschluss werden die nicht-funktionalen Anforderungen an die Spracherweiterung in Unterkapitel 3.3 aufgeführt.

3.1 Anwendungsfälle

Im Folgenden werden einige Anwendungsfälle aufgelistet, welche mit den bestehenden Möglichkeiten von Demaq nur sehr umständlich oder überhaupt nicht umgesetzt werden können.

Für jeden vorgestellten Anwendungsfall wird zunächst ein allgemeines Szenario angegeben, um das es sich bei dem Anwendungsfall handelt. Zur Konkretisierung des Szenarios werden dazu jeweils verschiedene Beispiele beschrieben. Zu jedem dieser Beispiele werden unterschiedliche Umsetzungen mit Demaq vorgestellt. Abschließend werden die ggf. entstehenden Probleme der einzelnen Umsetzungsalternativen beschrieben.

Um die verschiedenen Umsetzungsvarianten besser referenzieren zu können, werden die Anwendungsfälle mit arabischen Ziffern, die zugehörigen Beispiele mit Kleinbuchstaben und die jeweiligen Umsetzungen mit großen römischen Ziffern durchnummeriert. Beispielsweise bezeichnet 2.a.III die dritte Umsetzung des ersten Beispiels von Anwendungsfall 2.

Das Ziel dieser Anwendungsfälle besteht darin eine Auflistung der bestehenden Probleme in Demaq zu erhalten, um aus dieser Liste konkrete Anforderungen für die Erweiterung von Demaq herzuleiten.

3.1.1 Anwendungsfall 1: Wiederkehrende Anwendungsteile

Szenario

Eine Anwendung enthält an verschiedenen Stellen dieselbe Funktionalität, aber jeweils mit geringfügigen Unterschieden. Bei der Umsetzung sollte darauf geachtet werden Redundanz soweit als möglich zu vermeiden.

Beispiel a) Benachrichtigung von Lieferanten

In einem Online-Shop wird jeder Artikel von einem bestimmten Lieferanten geliefert. Geht ein Artikel zur Neige soll automatisch eine Nachricht an den entsprechenden Lieferanten gesendet werden. Die Schnittstelle zum System des Lieferanten erwartet dabei eine SOAP-Nachricht.

Umsetzung I: Realisierung mit DQL

Für jeden Lieferanten existiert eine ausgehende Gateway-Queue (Zeilen 1 u. 2), über welche die SOAP-Nachrichten an ihn gesendet werden, sowie eine Basic-Queue (Zeilen 4 u. 5), in welche für jeden zur Neige gehenden Artikel eine entsprechende Nachricht eingefügt wird. Eine zugehörige Regel (Zeilen 7-17 bzw. 19-29) generiert die SOAP-Nachricht, welche über die Gateway-Queue an den Lieferanten gesendet wird.

Listing 3.1: SOAP-Benachrichtigung von Lieferanten.

```
1  create queue HttpOutSupplierX kind outgoing interface "http" port "8010" mode
       persistent;
2  create queue HttpOutSupplierY kind outgoing interface "http" port "8030" mode
       persistent;
3
4  create queue lowStockSupplierX kind basic mode persistent;
5  create queue lowStockSupplierY kind basic mode persistent;
6
7  create rule briefSupplierX for lowStockSupplierX
8    enqueue message
9      <s:Envelope xmlns:s="http://www.w3.org/2001/12/soap-envelope">
10       <s:Header>
11       </s:Header>
12       <s:Body>
```

```
13          {.}
14        </s:Body>
15      </s:Envelope>
16    into HttpOutSupplierX
17    with comm:URL value "supplier-x.com";
18
19  create rule briefSupplierY for lowStockSupplierY
20    enqueue message
21      <s:Envelope xmlns:s="http://www.w3.org/2001/12/soap-envelope">
22        <s:Header>
23        </s:Header>
24        <s:Body>
25          {.}
26        </s:Body>
27      </s:Envelope>
28    into HttpOutSupplierY
29    with comm:URL value "supplier-y.com";
```

Probleme

Das Zusammensetzen der SOAP-Nachricht wird für jeden Lieferanten separat definiert (Zeilen 9-15 sowie 21-28). Dadurch entstehen redundante Anwendungsteile, was die Wartung der Anwendung erschwert und die Fehleranfälligkeit erhöht. Weiterhin ergibt sich ein größerer Zeitaufwand beim Erstellen der Anwendung, da dieselbe Funktionalität mehrfach erzeugt oder kopiert werden muss.

Umsetzung II: Realisierung mit einer XQuery-Funktion

Die Umsetzung gestaltet sich wie in Umsetzung I. Allerdings wird die SOAP-Nachricht durch eine XQuery-Funktion erzeugt (Zeilen 1-9), welche in den Regeln aufgerufen wird (Zeilen 18 u. 22).

Listing 3.2: SOAP-Benachrichtigung mit einer XQuery-Funktion.

```
1  declare function local:soap($message) {
2    <s:Envelope xmlns:s="http://www.w3.org/2001/12/soap-envelope">
3      <s:Header>
4      </s:Header>
5      <s:Body>
6        {$message}
```

```
7       </s:Body>
8     </s:Envelope>
9   };
10
11  create queue HttpOutSupplierX kind outgoing interface "http" port "8010" mode
        persistent;
12  create queue HttpOutSupplierY kind outgoing interface "http" port "8030" mode
        persistent;
13
14  create queue lowStockSupplierX kind basic mode persistent;
15  create queue lowStockSupplierY kind basic mode persistent;
16
17  create rule briefSupplierX for lowStockSupplierX
18    enqueue message local:soap(.) into HttpOutSupplierX
19    with comm:URL value "supplier-x.com";
20
21  create rule briefSupplierY for lowStockSupplierY
22    enqueue message local:soap(.) into HttpOutSupplierY
23    with comm:URL value "supplier-y.com";
```

Probleme

Mithilfe der XQuery-Funktion kann nur das Zusammensetzen der SOAP-Nachricht ausgelagert werden. Der Rest der Regel (`enqueue`-Target und Properties) sowie die benötigten Queues können nicht innerhalb der Funktion umgesetzt werden. Deshalb weist die Umsetzung (siehe Listing 3.2) immer noch Redundanz auf, wenn auch nicht mehr so stark wie in Umsetzung I.

Beispiel b) Versand von E-Mails

In einem Online-Shop werden in verschiedenen Situationen E-Mails versendet, welche sich bestimmten Kategorien wie z. B. *Kontakt* oder *Bestellung* zuordnen lassen. Die E-Mails der unterschiedlichen Kategorien sollen von verschiedenen Absender-Adressen versendet werden, etwa *service@demaq.net* für alle E-Mails der Kategorie *Kontakt*. Der Versand der E-Mails soll über eine zentrale ausgehende Queue per SMTP mit festgelegtem Port (25) erfolgen.

Umsetzung I: Realisierung mit DQL

Zum Versand der E-Mails kann eine gemeinsame Outgoing-Queue (Zeile 1) definiert werden. Für jeden Absender wird eine Basic-Queue (z. B. Zeile 3) sowie eine darauf definierte Regel (z. B. Zeilen 6-11) benötigt. Die Basic-Queues verlangen dabei eine Nachricht in der Betreff, Empfänger und Nachricht der E-Mail angegeben werden (siehe Listing 3.4).

Listing 3.3: Variabler E-Mail-Versand mit DQL.

```
1  create queue outgoingMails kind outgoing interface "smpt" port "25" mode
       persistent;
2
3  create queue mailsService kind basic mode persistent;
4  create queue mailsOrder kind basic mode persistent;
5
6  create rule sendMailService for mailsService
7    enqueue message //msg into outgoingMails
8    with comm:URL value "www.demaq.net"
9    with comm:From value "service@demaq.net"
10   with comm:To value "//to/text()"
11   with comm:Subject value "//subject/text()";
12
13 create rule sendMailOrder for mailsOrder
14   enqueue message //msg into outgoingMails
15   with comm:URL value "www.demaq.net"
16   with comm:From value "order@demaq.net"
17   with comm:To value "//to/text()"
18   with comm:Subject value "//subject/text()";
```

Listing 3.4: Struktur der Nachrichten.

```
1  <mail>
2    <msg>Message goes here.</msg>
3    <subject>Subject goes here.</subject>
4    <to>recipient@example.org</to>
5  </mail>
```

Probleme

Da erneut redundante Anwendungsteile entstehen, treten dieselben Probleme wie in der DQL-Umsetzung von Beispiel a) auf.

Umsetzung II: Realisierung mit einer XQuery-Funktion

Im Gegensatz zu Beispiel a) lässt sich in diesem Fall keine Funktionalität in eine XQuery-Funktion auslagern. Sinnvoll wäre eine Ausgliederung der Properties (Zeilen 8-11 sowie 15-18 in Listing 3.3) in eine Funktion. Das ist semantisch aber nicht möglich, da XQuery-Funktionen keine Updates erlauben und somit nicht nach einem `enqueue`-Target stehen dürfen.

3.1.2 Anwendungsfall 2: Auslagerung von Anwendungsteilen

Szenario

Ein bestimmter Anwendungsteil wird in mehreren Anwendungen benötigt. Dazu soll dieser Anwendungsteil auf eine Weise realisiert werden, die eine möglichst einfache Integration in verschiedene Programme ermöglicht.

Beispiel a) Erzeugung von SOAP-Nachrichten

Die Erzeugung von SOAP-Nachrichten aus Anwendungsfall 1 a) soll in verschiedenen Programmen eingebunden werden.

Umsetzung I: Realisierung mit DQL

Die SOAP-Generierung wird wie in 1.a.I direkt in den entsprechenden Regeln realisiert. Der dazu gehörige Code muss zur Wiederverwendung in anderen Anwendungen in diese hinein kopiert und angepasst werden.

Probleme

- Es besteht keine Möglichkeit auf diese Weise Anwendungsteile in anderen Programmen wiederzuverwenden.
- Nötige Wartungen können nicht an einer zentralen Stelle vorgenommen werden. Stattdessen müssen sie in allen bestehenden Programmen, welche den Anwendungsteil einbinden, durchgeführt werden.
- Potenzielle Fehlerquellen vermehren sich durch die Verteilung des Codes auf verschiedene Programme.

Umsetzung II: Realisierung mit XQuery-Modulen

Die XQuery-Funktion aus 1.a.II wird in ein XQuery-Modul (siehe Listing 3.5) ausgelagert, welches von jeder beliebigen Anwendung eingebunden (siehe Listing 3.6, Zeile 1 und Listing 3.7, Zeile 1) werden kann. Nach dem Import des Moduls kann auf die enthaltenen Funktionen zugegriffen werden (siehe Listing 3.6, Zeilen 10 u. 14 sowie Listing 3.7, Zeile 9).

Bei den beiden Listings handelt es sich zum einen um die Benachrichtigung von Lieferanten aus Beispiel a) von Anwendungsfall 1 sowie um ein weiteres Beispiel. In diesem werden Anfragen bearbeitet, indem sie als SOAP-Nachricht an einen externen Service weitergeleitet und von dort Antworten zurückgeschickt werden.

Listing 3.5: XQuery-Funktion zur Erzeugung von SOAP-Nachrichten.

```
1 declare function generate($message) {
2   <s:Envelope xmlns:s="http://www.w3.org/2001/12/soap-envelope">
3     <s:Header>
4     </s:Header>
5     <s:Body>
6       {$message}
7     </s:Body>
8   </s:Envelope>
9 };
```

Listing 3.6: Anwendung 1 mit SOAP-Generierung durch ein XQuery-Modul.

```
1  import module namespace soap = "http://www.demaq.net/dql" at "modules/soap.xql";
2
3  create queue HttpOutSupplierX kind outgoing interface "http" port "8010" mode
       persistent;
4  create queue HttpOutSupplierY kind outgoing interface "http" port "8030" mode
       persistent;
5
6  create queue lowStockSupplierX kind basic mode persistent;
7  create queue lowStockSupplierY kind basic mode persistent;
8
9  create rule briefSupplierX for lowStockSupplierX
10   enqueue message soap:generate(.) into HttpOutSupplierX
11   with comm:URL value "supplier-x.com";
12
13 create rule briefSupplierY for lowStockSupplierY
```

```
14   enqueue message soap:generate(.) into HttpOutSupplierY
15   with comm:URL value "supplier-y.com";
```

Listing 3.7: Anwendung 2 mit SOAP-Generierung durch ein XQuery-Modul.

```
1  import module namespace soap = "http://www.demaq.net/dql" at "modules/soap.xql";
2
3  create queue requests kind basic mode persistent;
4  create queue answers kind basic mode persistent;
5  create queue forward kind outgoing interface "http" port "8020" mode persistent;
6  create queue reply kind incoming interface "http" port "8030" mode persistent;
7
8  create rule forwardRequests for requests
9    enqueue message soap:generate(.) into forward
10   with comm:URL value "example.com";
11
12 create rule replyRequests for reply
13   enqueue message
14     <answer>//s:Body/text()</answer>
15   into answers;
```

Probleme

Keine. Die Erzeugung von SOAP-Nachrichten kann durch ein XQuery-Modul ausgelagert und von verschiedenen Anwendungen eingebunden werden.

Umsetzung III: Realisierung in einer eigenen Anwendung

Durch die Auslagerung der SOAP-Generierung in eine externe Anwendung könnte sie von verschiedenen Programmen eingebunden werden. Für die Umsetzung der Generierung einer SOAP-Nachricht ist eine eigene Anwendung allerdings zu aufwändig, zumal dies mit Umsetzung II bereits sehr gut gelöst wurde.

Beispiel b) Kundenverwaltung

Eine Kundenverwaltung soll erstellt werden, sodass sie in verschiedenen Anwendungen eingebunden werden kann. Die Kundendaten sollen persistent gespeichert werden, damit sie zu einem späteren Zeitpunkt bei Bedarf wieder

abgefragt werden können. Diese Kundendaten werden als XML-Nachricht geliefert und enthalten eine Kunden-ID, den Namen samt Adresse des Kunden sowie dessen eindeutige E-Mail-Adresse (siehe Listing 3.8). Zum Auslesen eines Kunden muss eine Kunden-ID übermittelt werden. Es soll weiterhin nicht möglich sein verschiedene Kunden mit derselben E-Mail-Adresse zu speichern.

Listing 3.8: Struktur der Kundendaten.

```
1 <customer>
2   <id>89235</id>
3   <firstname>Max</firstname>
4   <lastname>Mustermann</lastname>
5   <address>Musterstr. 123</address>
6   <zip>123456</zip>
7   <city>Musterhausen</city>
8   <email>mail@example.com</email>
9 </customer>
```

Umsetzung I: Realisierung mit DQL

Das Problem kann wie in 2.a.I nur mithilfe von Copy&Paste gelöst werden. Dadurch entstehen erneut die bereits erwähnten Probleme durch redundante Anwendungsteile.

Umsetzung II: Realisierung mit XQuery-Modulen

Da die Kundendaten persistent gespeichert werden sollen ist eine Umsetzung mit einer XQuery-Funktion nicht möglich, da innerhalb dieser Funktionen keine Zustände gespeichert und verwaltet werden können.

Umsetzung III: Realisierung in einer eigenen Anwendung

Es wird eine eigene Anwendung für die Kundenverwaltung erstellt, welche über Gateway-Queues von bestehenden Anwendungen angesprochen werden und Daten an diese zurückliefern kann. Dazu enthält die Kundenverwaltung *incoming*-Queues (Zeilen 2 u. 3) für eingehende Anfragen, welche über die definierten *Response*-Queues (Zeilen 2 u. 3) beantwortet werden. Des Weiteren existiert eine Queue zur Speicherung der Kundendaten (Zeile 1), Slicings für die Kunden-ID (Zeile 6) und die E-Mail-Adresse (Zeile 5) sowie Regeln zum Speichern (Zeilen 8-20) als auch zum Auslesen (Zeilen 22-26) der Kunden.

Listing 3.9: Eine Kundenverwaltung als eigene Anwendung.

```
create queue customers kind basic mode persistent;
create queue newCustomer kind incoming interface "http" port "8010" response
    newCustomerResponse mode persistent;
create queue getCustomer kind incoming interface "http" port "8020" response
    getCustomerResponse mode persistent;

create slicing property customerMail queue customers value //email/text() require
    fn:false();
create slicing property customerID queue customers value //id/text() require fn:
    false();

create rule saveCustomer for newCustomer
  let $custExists := qs:slice(//email/text(), "customerMail")[position() = last()]
  return
    if(not empty($custExists)) then
      enqueue message
        <result>Error: The e-mail {//email/text()} already exists.</result>
      into newCustomerResponse
    else (
      enqueue message . into customers,
      enqueue message
        <result>Success: The data has been saved.</result>
      into newCustomerResponse
    );

create rule getCustomerByID for getCustomer
  enqueue message {
    let $customer := qs:slice(//id/text(), "customerID")[position() = last()]
    return $customer
  } into getCustomerResponse;
```

Probleme

- Um die Kundenverwaltung aus einer Anwendung aufzurufen, müssen Nachrichten zwischen den beiden Anwendungen ausgetauscht werden. Dadurch wird die gesamte Anwendung langsamer im Gegensatz zu einer Realisierung in einer Anwendung. Dies begründet sich durch den zusätzlich benötigten Austausch von Nachrichten über das die Anwendungen verbindende Netzwerk.

- Für jede Anwendung, welche die Kundenverwaltung einbindet, muss eine Instanz der Kundenverwaltung gestartet werden, bevor diese Anwendung erfolgreich ausgeführt werden kann. Die eigentliche Anwendung kann also nicht ausgeführt werden, sofern die Kundenverwaltung nicht zuvor gestartet wurde.

3.1.3 Anwendungsfall 3: Getrennte Entwicklung einzelner Anwendungsteile einer Anwendung

Szenario

Ein großes Projekt soll umgesetzt werden. Dazu wird es in mehrere Anwendungsteile untergliedert, welche von verschiedenen Entwicklern programmiert werden sollen. Das heißt jeder Anwendungsteil soll eigenständig programmiert und anschließend in die eigentliche Anwendung integriert werden.

Beispiel a) Online-Shop

Ein Online-Shop soll erstellt werden und wird dazu aufgeteilt in vier Anwendungsteile. Die *Kundenverwaltung* wurde bereits in Anwendungsfall 2 b) in einer stark vereinfachten Form dargestellt. Für den Online-Shop müsste zudem die Aktualisierung von Kunden sowie ein Kunden-Login bereitgestellt werden. Die *Artikelverwaltung* speichert Artikel in verschiedene Kategorien, die zuvor angelegt werden, und kann die Artikel gefiltert nach Kategorie oder anderen Kriterien auch wieder ausgeben. Dem *Warenkorb* kann beim Einkauf eine Menge verschiedener Artikel hinzugefügt werden, woraus im Anschluss die *Bestellung* generiert werden kann.

Umsetzung I: Realisierung mit DQL

Da es nicht möglich ist DQL-Code einer Demaq-Anwendung auf verschiedene Resourcen aufzuteilen, kann zum Erreichen des Ziels eine Versionsverwaltung eingesetzt werden, wodurch alle Entwickler an derselben Datei arbeiten. Eine Alternative ist, dass die Entwickler ihre Anwendungsteile separat erstellen und die Anwendung durch Zusammenfügen aller Anwendungsteile in einer Datei fertig gestellt wird.

Probleme:

- Es entsteht eine sehr unübersichtliche Anwendung, da der komplette Shop innerhalb einer Datei realisiert werden muss.
- Die entstandene Anwendung ist schwer zu warten, weil die Änderungen nicht an losgelösten Anwendungsteilen vorgenommen werden können. Stattdessen sind durch die Vermischung aller Anwendungsteile mit der Hauptanwendung bei Änderungen an einzelnen Anwendungsteilen ggf. zusätzliche Anpassungen der Hauptanwendung nötig.
- Da die einzelnen Anwendungsteile weder separat erstellt noch getrennt bearbeitet werden können, gestaltet sich eine Arbeitsteilung umständlich.
- Eine einfache Möglichkeit zur Wiederverwendung der benutzten Anwendungsteile geht verloren durch die fehlende Möglichkeit Anwendungsteile auszugliedern.
- Die einzelnen Entwickler haben keine Möglichkeit ihre Anwendungsteile zu testen oder zu debuggen bevor diese in die Hauptanwendung eingebunden werden, da die Anwendungsteile nur in Verbindung mit der Hauptanwendung funktionieren.

Umsetzung II: Realisierung mit XQuery-Funktionen

Innerhalb von XQuery-Funktionen können weder DQL-Sprachkonstrukte erzeugt noch Zustände gespeichert werden. Deshalb eignen sie sich nicht für die Umsetzung von komplexen Anwendungsteilen, wie sie für den Shop benötigt werden.

Umsetzung III: Verteilung der Anwendungsteile auf eigene Anwendungen

Wie bereits die Kundenverwaltung (vgl. 2.b.III) werden auch die anderen Anwendungsteile in eigenen Anwendungen realisiert. Über Gateway-Queues werden alle Anwendungsteile an die eigentliche Hauptanwendung angebunden, indem Nachrichten über diese Queues an die entsprechenden Anwendungsteile gesendet und von dort auch wieder Nachrichten zurückgesendet werden.

Probleme:

- Von der Hauptanwendung müssen viele Anfragen zunächst an einen der externen Anwendungsteile weitergeleitet und anschließend ggf. die zurückkommenden Antworten ausgewertet werden. Durch den erhöhten Nachrichtenaustausch verringert sich die Leistungsfähigkeit und damit auch die Geschwindigkeit der gesamten Anwendung.
- Vor dem Start der Hauptanwendung muss sichergestellt werden, dass alle Anwendungsteile gestartet wurden und korrekt laufen. Dadurch gestaltet sich das Starten der Anwendung komplizierter als bei einer einzelnen Anwendung.

3.2 Funktionale Anforderungen

In diesem Unterkapitel werden die erläuterten Probleme der zusammen getragenen Anwendungsfälle analysiert und daraus die funktionalen Anforderungen formuliert.

Das Ziel dabei ist einen Anforderungenkatalog für die Erweiterung von Demaq zu erstellen, welcher im folgenden Kapitel in einen passenden Entwurf umgesetzt wird.

3.2.1 Unabhängige Entwicklung

In 3.a.I wird bemängelt, dass es in Demaq keine Möglichkeit gibt, mit der mehrere Entwickler gleichzeitig verschiedene Anwendungsteile eines Online-Shops entwickeln können. Daher soll die Erweiterung von Demaq ermöglichen, dass einzelne Anwendungsteile unabhängig von der Anwendung, in der sie benötigt werden, erstellt und debuggt werden können.

3.2.2 Unabhängige Wartung

Bei einem so umfangreichen Projekt wie dem vorgestellten Online-Shop (Anwendungsfall 3) fallen in regelmäßigen Abständen Wartungsarbeiten an. Dabei werden Fehler behoben, Funktionalität verbessert und ggf. neue Funktionalität hinzugefügt. Beziehen sich solche Wartungsarbeiten auf die Funktionalität innerhalb eines Anwendungsteils, so sollen diese Arbeiten durchgeführt werden

können, ohne dass zusätzlich die Hauptanwendung, welche den Anwendungsteil einbindet, geändert werden muss.

3.2.3 Zulassung der kompletten DQL-Syntax

Der Online-Shop aus Anwendungsfall 3 enthält sehr komplexe Anwendungsteile. Allein zur Umsetzung der Kundenverwaltung (Anwendungsfall 2 b)) werden mit Queues, Regeln, Properties und Slicings alle in DQL verfügbaren Sprachkonstrukte zur Umsetzung benötigt. Separat entwickelte Anwendungsteile (vgl. 3.2.1) sollen deshalb alle Möglichkeiten einer normalen Demaq-Anwendung bieten.

3.2.4 Wiederverwendbarkeit

Die Beispiele aus Anwendungsfall 2 (SOAP-Nachricht und Kundenverwaltung) sollen in mehreren Anwendungen eingesetzt werden. Die Erweiterung von Demaq muss daher ermöglichen, dass derselbe Anwendungsteil in verschiedene Anwendungen eingebunden werden kann.

3.2.5 Individualisierbarkeit

In Anwendungsfall 1 (1.a.I und 1.b.I) entstehen redundante Codefragmente, welche sich nur geringfügig voneinander unterscheiden. Solche Codefragmente sollen deshalb in einem vorgegebenen Rahmen individuell anpassbar sein. Damit soll ermöglicht werden, dass unterschiedliche Varianten desselben Codes auf Basis einer identischen Kernfunktionalität erzeugt werden können.

3.2.6 Mehrfaches Einbinden

Um die in Anwendungsfall 1 bemängelte Redundanz (vgl. 1.a.I und 1.b.I) zu vermeiden soll die Erweiterung von Demaq gewährleisten, dass ein Anwendungsteil mehrfach derselben Anwendung hinzugefügt werden kann.

3.2.7 Zugriff

Werden die verschiedenen Bereiche des Online-Shops aus Anwendungsfall 3 auf unabhängig voneinander erstellbare Anwendungsteile (vgl. 3.2.1) verteilt,

müssen die einzelnen Anwendungsteile mit der Hauptanwendung des Shops verbunden werden können. Etwa bei der Kundenverwaltung muss die Hauptanwendung eingehende Kundendaten zur Verwaltung an den entsprechenden Anwendungsteil weitergeben und bei Bedarf die entsprechenden Kundendaten zurückgeliefert bekommen.

Die Erweiterung von Demaq muss daher eine Möglichkeit bieten, sodass von einer Hauptanwendung auf Anwendungsteile zugegriffen werden kann als auch umgekehrt.

3.3 Nicht-funktionale Anforderungen

In diesem Unterkapitel werden die nicht-funktionalen Anforderungen an die Erweiterung von Demaq aufgelistet und kurz erläutert.

3.3.1 Laufzeitumgebung

Die an Demaq vorzunehmenden Anpassungen sollen die Laufzeitumgebung des bestehenden Systems nicht verändern.

3.3.2 Compiler

Die Architektur des Demaq-Compilers soll bestehen bleiben.

3.3.3 Sprache

Alle Änderungen an der DQL, sei es die Einführung eines neuen Befehls oder die Erweiterung eines Bestehenden, sollen zum bisherigen Konzept von Demaq passen.

3.3.4 Funktionalität

Bereits mit den bisherigen Möglichkeiten erstellte Anwendungen sollen nach der Erweiterung von Demaq noch funktionieren wie zuvor.

3.3.5 Laufzeit

Die Laufzeit von Anwendungen soll sich nicht ändern.

3.3.6 Benutzbarkeit

Die Erweiterung von Demaq soll einfach benutzt werden können.

4 Entwurf

In Kapitel 3 wurden die Anforderungen für die Erweiterung von Demaq aufgestellt. In diesem Kapitel wird nun ein Entwurf erarbeitet, welcher die gestellten Anforderungen umsetzt. Zur Erfüllung der Anforderungen sind ggf. Änderungen an der bestehenden Demaq-Sprache (DQL) notwendig. Das heißt es müssen möglicherweise bestehende Sprachkonstrukte erweitert oder neue Sprachkonstrukte eingeführt werden.

Ob und wie umfangreich eine solche Anpassung der DQL nötig ist wird in Unterkapitel 4.1 diskutiert. Bei Modifizierungen der Sprache müssen zudem alle neuen und geänderten Konstrukte in den Compiler integriert werden. Die ggf. notwendigen Änderungen am Compiler werden daher in Unterkapitel 4.2 erläutert.

4.1 Sprachentwurf

In diesem Unterkapitel wird ausgeführt, wie eine Erweiterung der DQL aussehen muss, um die Anforderungen aus Kapitel 3 zu erfüllen. Dazu wird im Folgenden ein neues Sprachkonzept eingeführt und dessen Funktionsweise festgelegt.

4.1.1 Erweiterung von Demaq um Module

Dieser Abschnitt liefert unter Beachtung bereits vorhandener Sprachkonzepte ein Konzept, mit welchem die geforderten Anforderungen erfüllt werden können. Untersucht werden dazu die bereits in den Anwendungsfällen (vgl. 3.1) mit einbezogenen Umsetzungsalternativen. Das sind zum einen XQuery-Funktionen, welche in XQuery-Module ausgelagert werden können, und zum anderen die Realisierung von Anwendungsteilen in eigenen Anwendungen. Ab-

schließend wird eine Kombination aus beiden Varianten auf ihre Tauglichkeit analysiert.

4.1.1.1 XQuery-Module

Die in Anforderung 3.2.1 geforderte unabhängige Entwicklung von Anwendungsteilen läßt sich mit XQuery-Modulen realisieren. In XQuery-Modulen können XQuery-Funktionen deklariert werden, welche mehrfach in derselben Anwendung wiederverwendet (vgl. 3.2.6) und mithilfe von Parametern an verschiedene Szenarien angepasst (vgl. 3.2.5) werden können. Durch die Auslagerung von XQuery-Funktionen in ein XQuery-Modul können sie zudem in beliebigen Anwendungen eingebunden werden (vgl. 3.2.4).

Allerdings können XQuery-Funktionen keine DQL-Konstrukte enthalten, weshalb sie wegen Anforderung 3.2.3 in dieser Form als Lösung nicht in Frage kommen.

4.1.1.2 Anwendungsteile als eigene Anwendung

Die in den Anwendungsfällen (vgl. 3.1) benutzte Umsetzungsalternative, Anwendungsteile auf eine separate Demaq-Anwendung auszulagern, lässt hingegen alle Möglichkeiten von Demaq in externen Anwendungsteilen zu. Die Kommunikation mit der eigentlichen Anwendung muss dabei durch das Senden und Empfangen von Nachrichten über Gateway-Queues realisiert werden.

Demaq-Anwendungen können wie auch XQuery-Module getrennt entwickelt (vgl. 3.2.1) sowie durch die Erzeugung mehrerer Anwendungsinstanzen in verschiedene Anwendungen eingebunden werden (vgl. 3.2.4). Weiterhin lässt sich in ihnen, im Gegensatz zu den XQuery-Funktionen, DQL in vollem Umfang einsetzen (vgl. 3.2.3).

Allerdings ist es nicht möglich eine eigene Anwendung an verschiedene Szenarien anzupassen (vgl. 3.2.5), da Demaq-Anwendungen nicht parametrisiert werden können. Doch selbst wenn sich das realisieren ließe, würde sich aufgrund des benötigten Nachrichtenaustauschs massiv die Laufzeit einer Anwendung ändern, was Anforderung 3.3.5 widerspricht.

4.1.1.3 Kombination aus XQuery-Modul und eigener Anwendung

Die Kombination aus der Flexibilität von XQuery-Modulen und der Mächtigkeit einer eigenen Anwendung liefert einen brauchbaren Lösungsansatz. Eine Idee hierzu ist die XQuery-Module insofern zu erweitern, dass darin vollständige Anwendungsteile unter Verwendung aller DQL-Konstrukte erstellt werden können. Damit bestehende Anwendungen nach der Erweiterung der XQuery-Module immer noch wie bisher funktionieren (vgl. 3.3.4), müssen die XQuery-Module in ihrer bestehenden Form nach der Erweiterung weiterhin zur Verfügung stehen. Zusätzlich zum XQuery-Prolog, aus welchem ein XQuery-Modul im Wesentlichen besteht, müssen in einer solchen Erweiterung DQL-Konstrukte erzeugt werden können. Dadurch sind diese Erweiterungen wie Demaq-Anwendungen aufgebaut, welche ebenfalls den XQuery-Prolog enthalten können. Der einzige Unterschied ist, dass in XQuery-Modulen eine Moduldeklaration enthalten ist.

Da eine solche Erweiterung der XQuery-Module durch das Hinzufügen der DQL-Konstrukte allerdings stark von den XQuery-Modulen abweicht, soll für diese Erweiterung ein neues Sprachkonstrukt eingeführt werden. Bei diesem Sprachkonstrukt handelt es sich um ein an die XQuery-Module angelehntes Modulkonzept, mit welchem vollständige Demaq-Anwendungen in Module ausgelagert werden können.

Dieses neue Sprachkonstrukt wird im weiteren Verlauf dieser Arbeit als Modul oder in unklaren Fällen als DQL-Modul bezeichnet. Um es von den Modulen in XQuery unterscheiden zu können, werden diese ausdrücklich als XQuery-Module bezeichnet. Wie diese Module im Detail realisiert werden sollen wird in den folgenden Unterkapiteln näher erläutert.

4.1.2 Modul-Spezifikation

Dieser Abschnitt beschäftigt sich mit der Adressierung und den Inhalten von Modulen.

4.1.2.1 Adressierung der Module

Wie die Pakete in Java oder die Includes in C/C++ könnten DQL-Module ebenfalls aus dem Dateisystem eingebunden werden. Gerade im Bereich verteilte Anwendungen und Web Services ist es allerdings sinnvoll, ein Modul

unter einem URL ablegen zu können. Wie bei den XQuery-Modulen bietet sich deshalb zur Adressierung der Module die Auslagerung in einen URI an. Dadurch kann ein Modul in einer lokalen Datei, unter einem URL, etc. zur Verfügung gestellt werden. Somit kann ein Modul, wie von Anforderung 3.2.4 verlangt, problemlos in verschiedenen Anwendungen eingebunden sowie getrennt von der eigentlichen Anwendung entwickelt werden (vgl. 3.2.1).

4.1.2.2 Sprache innerhalb der Module

Als Sprache in den Modulen bieten sich nicht viele Möglichkeiten, da sich die Spracherweiterung in das bestehende Konzept von Demaq einfügen soll (vgl. 3.3.3). Weiterhin verlangt Anforderung 3.2.3, dass in den Anwendungsteilen die vollständige DQL-Syntax zugelassen wird. Daraus ergibt sich als Möglichkeit für die Sprache innerhalb der Module der komplette Umfang der DQL.

Als Alternative für eine Sprache innerhalb der Module kommt DQL in einer erweiterten Form mit einer Reihe neuer Befehle und Sprachkonstrukte in Frage. Um die Möglichkeiten der Module auszunutzen müssen sich Entwickler dadurch zunächst in diese erweiterte Sprache einarbeiten. Dadurch sind die Module nicht mehr einfach zu benutzen (vgl. 3.3.6) und werden somit evtl. nicht verwendet. Zusätzlich gibt es dadurch mehr Möglichkeiten innerhalb der Module als in gewöhnlichen Anwendungen. Deswegen kann es dazu kommen, dass Anwendungen nur noch die Ausführung verschiedener Module beinhalten, da diese einen größeren Funktionsumfang bieten. Dies ist jedoch nicht die Intention bei der Einführung eines Modulkonzepts.

Fazit Sofern es zur Erfüllung der Anforderungen nicht zwingend benötigt wird, sollen innerhalb der Module nicht mehr Sprachkonstrukte als zuvor zur Verfügung stehen.

4.1.3 Modul-Zugriff

In Anforderung 3.2.7 wird gefordert, dass ein Datenfluss sowohl von der Anwendung zu einem Modul als auch umgekehrt möglich sein muss. Im Folgenden werden dazu verschiedene Ansätze vorgestellt und miteinander verglichen.

Ein erster Ansatz ist, dass sowohl die Anwendung auf alle Inhalte des Moduls zugreifen kann als auch das Modul auf alle Inhalte der Anwendung. Dadurch können Module und Anwendungen optimal aufeinander abgestimmt werden und den Möglichkeiten eines Moduls sind keine Grenzen gesetzt. Ein solcher Zugriff kann auf verschiedene Arten erfolgen:

- Einfügen einer Nachricht aus der Anwendung in eine Queue des Moduls.
- Einfügen einer Nachricht aus dem Modul in eine Queue der Anwendung.
- Definition einer Regel in der Anwendung auf eine Queue oder ein Slicing innerhalb des Moduls.
- Definition einer Regel im Modul auf eine Queue oder ein Slicing der Anwendung.
- Aufruf von `qs:queue(“QueueName”)` aus der Anwendung auf eine Queue des Moduls.
- Aufruf von `qs:queue(“QueueName”)` aus dem Modul auf eine Queue der Anwendung.
- Aufruf einer der folgenden Funktionen aus dem Modul auf ein Slicing der Anwendung oder umgekehrt.
  - `qs:slicekey(“SlicingName”)`
  - `qs:slice(“SlicingKey”, “SlicingName”)`

Ein Modul soll laut Anforderung 3.2.4 in verschiedene Anwendungen eingebunden werden können. Falls ein Modul auf Konstrukte der Anwendung zugreift kann es sein, dass das Modul in eine Anwendung eingebunden wird, welche diese Konstrukte nicht enthält. Dadurch kann es bei einer Einbindung des Moduls zu einem Fehler kommen. Ein solcher Fehler tritt z. B. auf, falls ein Modul durch den Aufruf der Funktion `qs:slice(“123”, “customerID”)` auf das Slicing *customerID* aus der Anwendung zugreifen will, dieses dort aber nicht existiert. Dieses Szenario wird aber durch die bestehende Fehlerbehandlung erkannt. Ungünstiger ist es, falls das Slicing vorhanden ist, es allerdings auf einer anderen Nachrichtenstruktur oder Property definiert wurde, als vom Modul erwartet wird. Dadurch kann es zu einer Fehlfunktion kommen, ohne dass diese vom Benutzer bemerkt wird.

Nach Anforderung 3.2.1 sollen Module zudem unabhängig debuggt werden können. Das ist nicht mehr der Fall, wenn Module auf Konstrukte der Anwendung zugreifen können. Da diese Konstrukte erst nach der Anbindung an die Anwendung zur Verfügung stehen, wird das Debuggen des Moduls einen Fehler liefern. Es kann nämlich nicht davon ausgegangen werden, dass die nötigen Konstrukte in der jeweiligen Anwendung existieren.

Falls ein Modul überarbeitet werden muss und sich damit Inhalte, auf welche von der Anwendung zugegriffen wird, verändern, müssen alle Benutzer des Moduls ihre Anwendung an die neue Beschaffenheit des Moduls anpassen. Von einer Änderung des Moduls sollen seine Benutzer aber nicht betroffen sein, das heißt die bestehende Anbindung eines Moduls an eine Anwendung soll sich nicht verändern (vgl. 3.2.2).

Außerdem muss durch die Freiheit bei dieser Art des Zugriffs berücksichtigt werden, dass keine unerwünschten Seiteneffekte entstehen. Falls etwa die Anwendung eine Nachricht in eine Queue des Moduls einfügt muss darauf geachtet werden, was dadurch ausgelöst wird (Regeln, zusätzliches Einfügen der Nachricht in ein Slicing, etc.), um nicht gewollte Nebeneffekte zu vermeiden. Dies widerspricht jedoch der geforderten einfachen Benutzbarkeit (vgl. 3.3.6), da sich ein Benutzer des Moduls intensiv mit dem Modulcode beschäftigen muss, bevor er es einbinden kann.

Eine Einschränkung des Zugriffs insofern, dass nur von der Anwendung auf das Modul zugegriffen werden kann und nicht umgekehrt, bietet ebenfalls die geforderte Funktionalität (vgl. 3.2.7). Mittels *enqueue* können Daten an Queues des Moduls gesendet werden und durch die Definition von Regeln auf Modul-Queues oder den direkten Zugriff auf Konstrukte des Moduls kann die Anwendung auf Daten aus dem Modul zugreifen.

Die geschilderten Probleme bzgl. der Benutzbarkeit (vgl. 3.3.6) sowie der unabhängigen Wartbarkeit eines Moduls (vgl. 3.2.2) bleiben allerdings auch bei dieser Lösung bestehen.

Eine weitere Alternative besteht darin, die Inhalte und somit Implementierungsdetails eines Moduls zu kapseln. Der Zugriff auf ein Modul wird realisiert, indem der Anwendung auf bestimmte Konstrukte eines Moduls Zugriff gewährt werden kann. Ähnlich wird dies auch in objektorientierten Sprachen

(z. B. Java oder C++) mithilfe der Angabe von *public*, *private* oder *protected* für die Variablen und Methoden einer Klasse realisiert. Der Zugriff muss durch das Modul vorgegeben werden und bei einer Veränderung des Moduls in dieser Form bestehen bleiben. Dadurch wird die Wartung eines Moduls ermöglicht (vgl. 3.2.2), ohne dass alle Anwendungen angepasst werden müssen, in denen das Modul eingebunden ist. Mit dieser Alternative wird zudem die Benutzbarkeit von Modulen erleichtert, da nur noch auf bestimmte Konstrukte des Moduls zugegriffen werden darf.

Trotzdem treten auch bei dieser Variante verschiedene Probleme auf. Es muss z. B. geklärt werden, welche in der DQL verfügbaren Konstrukte für einen Zugriff zugelassen werden sollen, um einen Datenaustausch mit der Anwendung zu ermöglichen. Sinn machen dabei nur die QDL-Konstrukte (Queues, Slicings, Properties), da in Regeln keine Daten gespeichert werden können. Eine geringere Anzahl zugelassener Konstrukte vereinfacht dabei die Benutzbarkeit, schränkt dafür allerdings die Möglichkeiten von Modulen ein. Sind die zugelassenen Konstrukte festgelegt können weiterhin potenzielle Fehlerquellen entstehen, solange auf die Konstrukte freier Zugriff gestattet wird. Beispielsweise könnte eine Queue für den Zugriff aus der Anwendung zugelassen werden, auf welche das Modul verschiedene Regeln definiert. Obwohl diese Queue dazu gedacht ist, dass die Anwendung Daten daraus auslesen kann, ist es trotzdem möglich aus der Anwendung Nachrichten in diese Queue einzufügen. Dadurch entsteht das Problem, dass dadurch ungewollt Regeln angestoßen werden können.

Um dies zu verhindern, könnten den Konstrukten Bedingungen (*Constraints*) mitgegeben werden, die bestimmen, wie auf das jeweilige Konstrukt zugegriffen werden darf. Vor allem könnte mit Constraints festgelegt werden, in welche Richtung der Datenfluss erfolgen darf, das heißt ob ein schreibender, eingehender oder ein lesender, ausgehender Zugriff von der Anwendung auf das Modul möglich sein soll. Da die Constraints somit in den meisten Fällen entweder einen eingehenden oder einen ausgehenden Datenfluss zulassen, bietet es sich an, statt der Constraints die Konstrukte gezielt für einen ein- oder ausgehenden Datenfluss zu definieren. Dadurch wird der jeweilige Verwendungszweck eines Konstrukts eindeutig bestimmt und potenzielle Fehlerquellen können vermieden werden, indem ein- und ausgehende Konstrukte jeweils mit unterschiedlichen Einschränkungen verbunden werden.

Der Ansatz, den Zugriff über vorgegebene Konstrukte zu realisieren, welche zudem in Konstrukte für den ein- oder ausgehenden Datenfluss unterteilt sind, stellt eine akzeptable Lösung dar. Ein großer Nachteil dabei ist, dass die Zugriffs-Beschränkungen kontrolliert und eingehalten werden müssen, wodurch eine aufwändige Schnittstellen-Spezifikation benötigt wird.

Im Folgenden wird eine Idee vorgestellt, bei der diese Beschränkungen nicht nötig sind, da die Anwendung und das Modul völlig unabhängig nebeneinander existieren. Diese Idee bezieht sich auf eine nachrichtenbasierte Kommunikation, wodurch ein Modul vollständig von der Anwendung gekapselt werden kann. Das heißt die Anwendung greift nicht auf Konstrukte des Moduls zu sondern kommuniziert über XML-Nachrichten mit dem Modul. Dazu müssen Nachrichten einerseits von der Anwendung an das Modul und andererseits vom Modul an die Anwendung gesendet werden können (siehe Abb. 4.1).

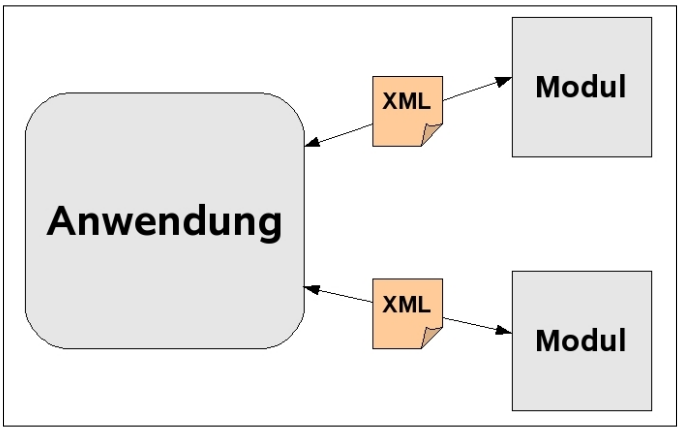

Abbildung 4.1: Kommunikation durch den Austausch von XML-Nachrichten.

Diese Lösung orientiert sich am bestehenden Konzept der Gateway-Queues, welche die Kommunikation einer Demaq-Anwendung mit externen Services ermöglicht. Dadurch fügt sich dieser Ansatz besonders gut in die vorhandene Sprache ein (vgl. 3.3.3). Wie bei einer Kommunikation über Gateway-Queues müssen für einen Nachrichtenaustausch auf beiden Seiten Kommunikations-Punkte erstellt werden, über welche die Nachrichten ausgetauscht werden können. Die genaue Beschaffenheit dieser Punkte wird in Unterkapitel 4.1.5 beschrieben.

Fazit Durch die vollständige Kapselung eines Moduls und die damit verbundenen Vorteile stellt eine nachrichtenbasierte Kommunikation die beste Lösung dar. Diese Vorteile bestehen unter anderem in der unabhängigen Entwicklung (vgl. 3.2.1) sowie Wartung (vgl. 3.2.2) der Module. Durch die nachrichtenbasierte Kommunikation gestaltet sich der Zugriff auf ein Modul zudem sehr einfach (vgl. 3.3.6) und passt dieser Ansatz auch in das bestehende Konzept von Demaq (vgl. 3.3.3).

4.1.4 Modul-Import

Nach der Modul-Spezifikation (vgl. 4.1.2.1) werden Module unter einem URI zur Verfügung gestellt. Um ein Modul in einer Anwendung zu verwenden muss daher angegeben werden, unter welcher URI es abgelegt ist. Im Folgenden werden dazu verschiedene Ansätze vorgestellt und miteinander verglichen.

Prinzipiell genügt es an den entsprechenden Stellen, an denen ein Modul benötigt wird, das Modul über den URI anzusprechen, unter dem das Modul gespeichert ist. Wird ein Modul mehrfach eingebunden (vgl. 3.2.6) muss allerdings bei jeder Anbindung dieser URI angegeben werden. Das stellt bei einer späteren Änderung des URIs eine potenzielle Fehlerquelle dar und verursacht einen erhöhten Wartungsaufwand. Zur Bestimmung aller in einer Anwendung verwendeten Module muss sie zudem komplett auf Modul-URIs durchsucht werden.

Der XQuery-Prolog enthält für den Import der XQuery-Module eine Import-Anweisung, welche möglicherweise in ihrer bestehenden oder einer leicht angepassten Form auch für die DQL-Module verwendet werden kann.

Die *import module*-Anweisung von XQuery fügt allerdings den Modulinhalt in den Prolog einer Demaq-Anwendung ein, in dem keine DQL-Konstrukte zugelassen sind, welche in einem DQL-Modul aber erlaubt sind (vgl. 4.1.2.2). Das heißt die *import module*-Anweisung muss überladen oder in einer anderen Art erweitert werden, um zwischen XQuery- und DQL-Modulen zu unterscheiden und die Module auf eine andere Weise einzubinden. Weiterhin sind alle Inhalte eines XQuery-Moduls (Variablen, Funktionen, etc.) an einen Namensraum gebunden. Die DQL-Konstrukte innerhalb der DQL-Module sollen allerdings

nicht einzeln angesprochen werden können (vgl. 4.1.3), wie das bei den Inhalten eines XQuery-Moduls der Fall ist, weshalb die Bindung der Modulinhalte an einen Namensraum nicht benötigt wird.

Als dritte Variante kommt die Einführung eines neuen Import-Befehls für die Module in Frage. Ein solcher Befehl muss einem Modul einen innerhalb der Anwendung eindeutigen Namen zuweisen und zudem den URI angeben, unter dem das Modul zu finden ist. Mithilfe des Namens kann das Modul innerhalb der Anwendung gezielt angesprochen werden und der URI kann an einer zentralen Stelle geändert werden.

Fazit Es ist nicht zwingend erforderlich einen neuen Befehl zum Import der Module einzuführen. Um Anwendungen übersichtlicher gestalten zu können ist es in vielen Sprachen üblich die benutzten externen Quellen an einer zentralen Stelle einzubinden, bevor sie verwendet werden (vgl. etwa Pakete in Java [13]). Durch die Anlehnung der Module an die XQuery-Module bietet es sich an den Import auf eine daran angepasste Weise umzusetzen, zumal die Spracherweiterung zum Konzept der bisherigen Sprache passen soll (vgl. 3.3.3). Aufgrund der genannten Nachteile und der dadurch entstehenden umfangreicheren Anpassung des Modul-Imports von XQuery, soll für den Import der Module ein neuer Befehl eingeführt werden. Dazu muss die DQL allerdings um diesen neuen Befehl erweitert werden. Diese Änderung gestaltet sich aber nicht so umfangreich wie die Realisierung eines aus dem Modul-Import von XQuery abgeleiteten Befehls, der für den Import beider Modul-Typen zuständig ist.

Der Import der Module soll außerdem an einer zentralen Stelle innerhalb einer Anwendung und auf jeden Fall vor der Verwendung des jeweiligen Moduls stattfinden. Dazu empfiehlt sich ein Ort möglichst zu Beginn einer Anwendung, also entweder noch vor oder unmittelbar nach dem XQuery-Prolog. Da keine der beiden Varianten signifikante Vorteile bietet, wird für den Modul-Import die Stelle direkt nach dem Prolog gewählt.

4.1.5 Modul-Schnittstellen

Als Lösung für den von Anforderung 3.2.7 geforderten Datenaustausch zwischen einem Modul und einer Anwendung wurde in Unterkapitel 4.1.3 eine nachrichtenbasierte Kommunikation gewählt. Dieses Unterkapitel beschäftigt

sich mit der Beschaffenheit der dort angesprochenen Kommunikations-Punkte. Für diese Punkte hat sich der Begriff *Schnittstelle* (*Interface*) durchgesetzt. Diese Begrifflichkeit wird daher im weiteren Verlauf dieser Arbeit dafür verwendet.

4.1.5.1 Queues als Schnittstellen

Für einen Nachrichtenaustausch mit externen Systemen lassen sich *incoming*- sowie *outgoing*-Queues anlegen. Der Austausch zwischen einer Anwendung und einem Modul kann in einer ähnlichen Form umgesetzt werden.

Bei einer nachrichtenbasierten Kommunikation müssen Nachrichten gesendet und empfangen werden können. Deshalb müssen in einem Modul ein- und ausgehende Schnittstellen angelegt werden, um darüber Nachrichten empfangen bzw. senden zu können. In der Anwendung müssen zu diesen Schnittstellen entsprechende Gegenstücke angelegt werden, welche bei der Anbindung eines Moduls den Schnittstellen des Moduls zugewiesen werden.

Damit die zur Verfügung stehenden Schnittstellen für Benutzer des Moduls problemlos herauszufinden sind, sollen diese Schnittstellen am Anfang eines Moduls zentral zusammengefasst werden. Dies lässt sich mit der geforderten einfachen Benutzbarkeit der Module begründen, da ansonsten der gesamte Modulcode nach Schnittstellen durchsucht werden muss.

Als Datenstruktur für diese Schnittstellen soll ein Konstrukt gewählt werden, welches Nachrichten empfangen und speichern kann. Für eine möglichst präzise Schnittstellendefinition soll zudem ein Schema für die erwarteten Nachrichten angegeben werden können. Queues sind für eine Speicherung von XML-Dokumenten vorgesehen und ihnen kann außerdem eine Schema-Datei zugewiesen werden. Daher ist es nahe liegend Queues als Datenstruktur zu verwenden, immerhin sind die Schnittstellen zu externen Services auch mit Queues realisiert. Diese Entscheidung fügt sich somit gut in das bestehende Konzept von Demaq ein. Für die Gegenstücke in der Anwendung sollen aus denselben Gründen ebenfalls Queues gewählt werden, da diese Gegenstücke dieselbe Funktionalität bereitstellen müssen.

In der Anwendung und im Modul sollten diese Queues jeweils ohne Einschränkungen verwendet werden können, ohne dass die Gegenseite durch das Arbeiten mit diesen Queues tangiert wird. Bei der Anbindung eines Moduls

an eine Anwendung (vgl. 4.1.7.1) muss dann dafür gesorgt werden, dass die Nachrichten entsprechend verarbeitet werden. Eingehende Nachrichten in eine Queue der Anwendung, welche einer eingehenden Schnittstelle eines Moduls zugeordnet wurde, müssen an diese Schnittstelle weitergeleitet werden. Dasselbe muss umgekehrt beim Einfügen einer Nachricht durch das Modul in eine ausgehende Schnittstelle geschehen.

Da es verschiedene Queue-Typen gibt stellt sich die Frage, ob z. B. eine *outgoing*-Queue als eingehende Schnittstelle Sinn macht. Sofern ein Modul mit einem entfernten Service kommuniziert kann es durchaus nützlich sein, wenn Anfragen aus der Anwendung direkt an diesen externen Service durchgereicht werden können. Wenn ein Modul keine eigene Funktionalität enthält, sondern nur die Verbindung zu diesem Service herstellt, kann das Modul sehr schlank gehalten werden.

Die Definition einer *incoming*-Queue als ausgehende Schnittstelle macht in diesem Zusammenhang ebenfalls Sinn, um die Antworten des Services direkt an die Anwendung weiterleiten zu können. Daher besteht kein Grund die Möglichkeiten eines Moduls in dieser Hinsicht einzuschränken, weshalb für die Schnittstellen beliebige Queues zugelassen werden.

4.1.5.2 Definition der Schnittstellen

Es bleibt noch zu klären, wie innerhalb eines Moduls diese Queues als ein- und ausgehende Schnittstellen gekennzeichnet werden sollen. Da es sich um gewöhnliche Queues handelt ist es nahe liegend hierfür den `create queue`-Befehl entsprechend zu erweitern. Diese Erweiterung muss ermöglichen, dass eine Queue als ein- oder ausgehende Schnittstelle definiert werden kann. Wie auch bei den Gateway-Queues kann dies durch ein zusätzliches Attribut gelöst werden. Konzeptuell ist hierbei das Problem, dass diese Queues einerseits eine modulinterne Queue als andererseits auch eine Schnittstelle zur Anwendung darstellen. Die Funktionalität als Schnittstelle wird durch diese Erweiterung allerdings nicht so deutlich hervorgehoben, wie das bei einer Schnittstellendefinition wünschenswert ist.

Eine Lösung dafür ist, dass Schnittstellen innerhalb des Moduls als gewöhnliche Queues erzeugt und benutzt werden können. In einer davon getrennten

Schnittstellendefinition kann dann innerhalb eines neu eingeführten Befehls bestimmt werden, welche Queues als Schnittstellen dienen und ob es sich dabei um eine ein- oder eine ausgehende Schnittstelle handelt. Dadurch lässt sich die Angabe der Schnittstellen sehr kurz halten, da die entsprechenden Queues hintereinander aufgelistet werden können. Für einen Benutzer des Moduls ist allerdings interessant, welchem Schema die in eine Schnittstelle eingehenden Nachrichten entsprechen müssen. Diese Schema-Angabe befindet sich in der Definition der Queue und nicht in der Schnittstellendefinition, wodurch die entsprechende Queue aus dem Modulcode herausgesucht werden muss.

Als weitere Idee kommt die Einführung eines neuen Befehls für die Schnittstellendefinition in Frage. Innerhalb dieses Befehls muss der Schnittstelle ein Name gegeben und festgelegt werden, ob es sich um eine ein- oder eine ausgehende Schnittstelle handelt. Zudem muss die Beschaffenheit der zu erzeugenden Queue angegeben werden, weshalb diese Schnittstellendefinition an den Befehl zur Erstellung einer Queue angelehnt werden kann.

Fazit Bei einer Erweiterung des `create queue`-Befehls entsteht der Nachteil, dass sich die Schnittstellendefinition nicht ohne Weiteres an den Anfang eines Moduls verschieben lässt, da DQL-Konstrukte bislang erst nach dem Prolog erlaubt werden. Zudem muss ein bestehendes DQL-Statement geändert werden, wodurch sich eine der beiden anderen Varianten anbietet, obwohl dabei ein neuer Befehl eingeführt werden muss.

Beide verbleibenden Möglichkeiten haben Vor- und Nachteile. Auf der einen Seite entsteht eine sehr schlanke Schnittstellendefinition, in welcher auf existierende Queues innerhalb des Moduls verwiesen wird. Auf der anderen Seite werden die Schnittstellen wie Queues definiert, wodurch aus der Schnittstellendefinition die modulinterne Beschaffenheit der Schnittstellen ersichtlich ist.

Die übrig gebliebenen Varianten stellen beide gute Lösungen dar. Um zu unterstreichen, dass es sich bei den jeweiligen Queues um Schnittstellen handelt, wird als Lösung die ausführliche Definition gewählt.

4.1.6 Modul-Parameter

Da es möglich sein soll ein Modul an verschiedene Szenarien anzupassen (vgl. 3.2.5) werden in diesem Abschnitt verschiedene Konzepte vorgestellt, mit wel-

chen die Erzeugung unterschiedlicher Varianten eines Moduls ermöglicht werden können.

Eine Möglichkeit ist, dass in der Anwendung Konstrukte aus einem Modul überschrieben oder sogar neue Konstrukte hinzugefügt werden können. Auf diese Art kann ein Benutzer ein Modul vollständig an seine persönlichen Bedürfnisse anpassen.

Allerdings wurde in Unterkapitel 4.1.3 bewusst eine Kapselung der Module beschlossen. Der Grund dafür war unter anderem, dass ein Modul verändert werden kann ohne dass die das Modul einbindende Hauptanwendung angepasst werden muss. Als Beispiel überschreibt ein Benutzer eine bestimmte Regel eines Moduls. Dazu muss er sich unter Umständen zunächst intensiv in den Modulcode einarbeiten, was nicht der geforderten einfachen Benutzbarkeit der Module entspricht (vgl. 3.3.6). Durch das Überschreiben einer Regel ist es sehr wahrscheinlich, dass dabei auf weitere modulinterne Konstrukte zugegriffen wird. Zum einen ist das eine Queue oder ein Slicing, worauf die Regel definiert ist und zum anderen eine Queue, in welche von der Regel Nachrichten eingefügt werden. Wird das Modul durch eine neue Version ausgetauscht oder in einer anderen Form modifiziert, so kann es sein, dass sich die interne Struktur des Moduls ändert und eines (oder mehrere) der erwähnten Konstrukte in seiner (ihrer) vorherigen Form nicht mehr vorhanden ist (sind).

Wird eine solche Situation rechtzeitig bemerkt genügt es die Anwendung an die geänderten Implementierungsdetails anzupassen. Nach Anforderung 3.2.2 soll ein solcher Fall allerdings vermieden werden. Problematisch wird es vor allem, falls die Änderung eines Moduls nicht bemerkt wird und die Anwendung dadurch nicht mehr richtig funktioniert.

Ein anderer Ansatz ist, dass in einem Modul Parameter verwendet werden können, welchen bei der Anbindung des Moduls an eine Anwendung Werte zugewiesen werden müssen. Dadurch lassen sich Module in einem streng vorgegebenen Rahmen anpassen, der von den Entwicklern eines Moduls vorgeschrieben wird.

Um die Fehler des vorherigen Ansatzes zu vermeiden soll es mit den Parametern nicht möglich sein auf Modul-Interna, wie z. B. die Namen von Queues oder Regeln, zuzugreifen oder gar neue Konstrukte in das Modul einzufü-

gen. Dadurch soll die Kapselung eines Moduls beibehalten werden. Problemlos kann eine Verwendung von Parametern in Regel-Bodies und *with ... value*-Statements erlaubt werden. Zusätzlich wäre es nützlich, wenn sich Slicing *require*-Ausdrücke sowie Werte von Properties variabel gestalten ließen. Bei einem Modul zur Speicherung von Nachrichten könnte die Dauer der Nachrichten-Aufbewahrung als *require*-Parameter angegeben werden. Beide Stellen sind in Bezug auf ihre Fehleranfälligkeit nicht unproblematisch. Da sie aber die Möglichkeiten von Modulen sinnvoll erweitern, sollen sie für eine Parametrisierung zugelassen werden. Auch eine Parametrisierung des *enqueue*-Targets von Regeln könnte erlaubt werden, um von einer Regel generierte Nachrichten gezielt in eine Queue der Anwendung einfügen zu können. Allerdings wird hierbei die strenge Kapselung des Moduls gelockert. Da sich dieses Szenario ebenfalls durch die Definition einer weiteren ausgehenden Schnittstelle lösen lässt, soll ein *enqueue*-Target nicht aus einem Parameter bestehen dürfen.

Weiterhin könnten die Parameter innerhalb eines Moduls mit einem Basis-Wert initialisiert werden. Den Parametern müssen dann nur in speziellen Fällen Werte übergeben werden, wodurch sich die Benutzbarkeit von Modulen erhöht. Da die Übergabe von Werten zu diesen Parametern dadurch nicht mehr erforderlich ist entsteht eine potenzielle Fehlerquelle. Ein solcher Fehler tritt auf falls die Übergabe eines Werts wichtig ist, aber aufgrund eines eingestellten Initialwerts bei einer fehlenden Übergabe keine Fehlermeldung ausgegeben wird. Aufgrund der nur leicht erhöhten Benutzbarkeit und der Entstehung einer potenziellen Fehlerquelle soll daher auf die Möglichkeit der Angabe eines Initialwerts verzichtet werden.

Es stellt sich zudem die Frage welche Datentypen für die Parameter zugelassen werden. Möglich sind zum einen alle in XQuery verfügbaren Literale, also Strings, Integer, Double, etc. Zum anderen kann es nützlich sein, für die Parameter zusätzlich Pfadausdrücke verwenden zu können. Wird ein Pfadausdruck in einem *Regel-Body* einer Regel verwendet, die auf eine modulinterne Queue definiert ist, greift der Pfadausdruck auf Nachrichten in dieser modulinternen Queue zu. Dadurch wird die Kapselung des Moduls durchbrochen (vgl. 4.1.3), weshalb Pfadausdrücke nicht zugelassen werden sollen.

Wie auch die Schnittstellen sollen die innerhalb eines Moduls benutzten Parameter zu Beginn des Moduls angegeben werden können, um einem Benutzer des Moduls die Parameter bekannt zu machen.

Für die Angabe der Parameter kann wie für die Schnittstellendefinitionen ein neuer Befehl eingeführt werden. Die Anforderungen an einen solchen Befehl bestehen darin einen Namen für den Parameter zu definieren, welcher innerhalb des Moduls verwendet werden kann. Anhand dieses Namens kann dem Parameter bei der Anbindung des Moduls zudem ein Wert zugewiesen werden. Um die Parameter genauer definieren zu können sollte zum Namen zusätzlich eine Typdefinition (`xs:string`, `xs:integer`, etc.) angegeben werden können.

Innerhalb des XQuery-Prologs können externe Variablen definiert werden (vgl. 2.3), welche alle genannten Anforderungen erfüllen. Diese externen Variablen sind zudem genau dafür vorgesehen, um von einer externen Umgebung übergeben zu werden. Zudem können in einer Demaq-Anwendung definierte Variablen ausschließlich an den geforderten Stellen (Regel-Body, Property-Value, etc.) eingesetzt werden, was sich auch auf externe Variablen bezieht. Externe Variablen definieren einen Parameternamen, dem optional eine Typdefinition mitgegeben werden kann. Ein Wert für den Parameter kann dabei nicht angegeben werden. Durch das Schlüsselwort `external` anstelle der Zuweisung eines Werts können die Parameter zudem von lokalen Variablen unterschieden werden.

Fazit Die externen Variablen genügen im Gegensatz zur ersten Alternative exakt den gestellten Anforderungen. Aus diesem Grund sollen diese externen Variablen für die Parameterdefinition verwendet werden, wodurch der Sprache kein neuer Befehl hinzugefügt werden muss.

4.1.7 Modul-Instanziierung

Dieses Unterkapitel beschäftigt sich mit der Instanziierung von Modulen. Dazu wird zunächst die Anbindung eines Moduls an eine Anwendung besprochen. Anschließend wird eine Mehrfach-Instanziierung desselben Moduls untersucht und im letzten Abschnitt wird behandelt, ob eine einzelne Queue als Schnittstelle für verschiedene Instanzen verwendet werden darf.

4.1.7.1 Anbindung eines Moduls

Bei der Anbindung eines Moduls an eine Anwendung müssen die innerhalb des Moduls bereitgestellten Schnittstellen mit der Anwendung verknüpft werden. Das heißt um eine Instanz des Moduls zu erstellen müssen alle im Modul definierten Schnittstellen den entsprechenden Gegenstellen aus der Anwendung zugeordnet, sowie den im Modul vorhandenen Parametern Werte zugewiesen werden. Der vorliegende Abschnitt stellt dazu verschiedene Lösungsansätze vor. Zur Herstellung einer Verbindung zwischen den Queues der Anwendung und den Schnittstellen des Moduls kommen dabei dieselben Lösungsmöglichkeiten in Frage, welche auch für die Definition der Schnittstellen des Moduls in Erwägung gezogen wurden.

Die erste Möglichkeit besteht darin, den `create queue`-Befehl insofern zu erweitern, dass die erzeugte Queue einem Modul sowie einer darin enthaltenen Schnittstelle zugewiesen werden kann. Das Problem hierbei ist, dass für die Zuweisung von Werten zu den Modul-Parametern zusätzlich eine Lösung gefunden werden muss. Weiterhin erstreckt sich die Instanziierung eines Moduls dadurch über verschiedene `create queue`-Befehle und es existiert kein expliziter Instanziierungs-Punkt. Wie auch bei den Modul-Schnittstellen (vgl. 4.1.5) soll die Instanziierung deutlich abgegrenzt und nicht durch eine Erweiterung bestehender Konstrukte realisiert werden. Schließlich handelt es sich bei der Instanziierung um eine neue Funktionalität, welche nicht mit dem Erzeugen von Queues vermischt werden soll.

Für die Definition der Schnittstellen (vgl. 4.1.5.2) wird ein neuer Befehl verwendet, der eine Queue erzeugt und zusätzlich den Verwendungszweck der Schnittstelle (ein- oder ausgehend) bestimmt. Durch den neuen Befehl wird der Unterschied einer Schnittstelle zu einer gewöhnlichen Queue hervorgehoben. Um diese Lösung für das hier vorliegende Problem zu übertragen kann für die Anwendung ein ähnlicher Befehl eingeführt werden. Dieser muss ebenso eine Queue erzeugen sowie angeben, welcher Schnittstelle von welchem Modul diese Queue zugewiesen werden soll.

Allerdings wäre die Instanziierung eines Moduls immer noch auf mehrere Befehle verteilt und auch die Übergabe der Parameter müsste separat gelöst werden. Zusätzlich wäre es angenehm, wenn ein Modul mit relativ wenig Auf-

wand an eine bestehende Anwendung, also an darin bereits vorhandene Queues, angebunden werden könnte. Mit dieser Lösung müssen für eine Anbindung eines Moduls diese Queues allerdings zunächst umgeschrieben werden, was die Benutzbarkeit von Modulen erschwert (vgl. 3.3.6).

Ein weiterer Ansatz ist die Verwendung eines Instanziierungs-Blocks innerhalb der Anwendung, in welchem gebündelt die benötigten Queues für eine Modulinstanz zusammengefasst und den entsprechenden Schnittstellen zugewiesen werden können. Ein solcher Instanziierungs-Block könnte durch einen neu eingeführten Befehl eingeleitet werden. Dieser Befehl muss das Modul angeben sowie die Übergabe der Werte für die Parameter ermöglichen. Danach können innerhalb des Befehls die benötigten Queues aufgelistet werden, wodurch ihre Zugehörigkeit zu dem entsprechenden Instanziierungs-Block dargestellt wird.

Diese Lösung erzeugt einen klaren Instanziierungs-Punkt und es kann nicht passieren, dass die Instanziierung wie in der vorhergehenden Lösung über die gesamte Anwendung verteilt wird. Auch der Modulname muss nicht jedes Mal angegeben werden, wodurch den Queues nur noch der Name der entsprechenden Schnittstelle mitgegeben werden muss. Dadurch stellt dieser Ansatz bereits eine gute Lösung dar, aber der `create queue`-Befehl muss um ein weiteres Attribut erweitert werden und die Anbindung an eine bestehende Anwendung ist weiterhin mit einer Umstrukturierung dieser Anwendung verbunden.

Als letzte Lösung bleibt die Variante, welche bei der Schnittstellendefinition nur knapp zurückstehen musste (vgl. 4.1.5.2). Bei dieser Variante wird ein neuer Befehl eingeführt, welcher die Schnittstellenzuweisung mithilfe bestehender Queues vornimmt. Dieser Befehl gibt an, für welches Modul eine Instanz erstellt werden soll und anschließend können bereits vorhandene Queues der Anwendung beliebigen Schnittstellen dieses Moduls zugewiesen werden. Innerhalb desselben Befehls kann auch die Übergabe der Werte für die Parameter realisiert werden.

Ein großer Vorteil dieser Lösung ist, dass die `create queue`-Anweisung nicht erweitert werden muss um die verlangte Funktionalität zu gewährleisten. Stattdessen wird die komplette Instanziierung innerhalb eines neuen Befehls an einer Stelle realisiert, ohne dass die Instanziierung auf verschiedene Befehle verteilt ist. Bestehende Anwendung müssen für die Einbindung eines Moduls nicht um-

geschrieben werden und auch die Zuweisung von Werten zu den Parametern kann innerhalb desselben Befehls erfolgen. Diese Vorteile machen diesen Vorschlag zur flexibelsten und zugleich am einfachsten zu benutzenden (vgl. 3.3.6) Alternative.

4.1.7.2 Mehrfach-Instanziierung

Nach Anforderung 3.2.6 soll ein Modul mehrfach in derselben Anwendung eingebunden, also mehrfach instanziiert werden können. Dazu müssen für jede Instanz jeweils alle bereitgestellten Schnittstellen mit Queues aus der Anwendung verbunden sowie allen Parametern Werte zugewiesen werden. Mit der im vorherigen Abschnitt vorgestellten Lösung für die Schnittstellenzuweisung ist die Realisierung der Mehrfach-Instanziierung eines Moduls kein Problem. Zur Unterscheidung der einzelnen Instanzen muss mit dem Instanziierungs-Befehl allerdings ein eindeutiger Instanzname vergeben werden. Dies ist besonders wichtig, um beim Debuggen und bei der Fehlerbehandlung bestimmen zu können, in welcher Instanz ein Fehler aufgetreten ist.

4.1.7.3 Verwendung derselben Queue für mehrere Instanzen

Die Ermöglichung der Mehrfach-Instanziierung desselben Moduls (vgl. 4.1.7.2) wirft die Frage auf, ob eine Queue als Schnittstelle für mehrere Instanzen verwendet werden darf. Das kann sinnvoll sein, falls z. B. die Antwort-Nachrichten aller Instanzen eines Moduls in der Anwendung in einer zentralen Queue gesammelt werden sollen. In diesem Fall muss jeder Instanz eines Moduls als ausgehende Schnittstelle immer dieselbe Queue in der Anwendung zugewiesen werden können.

Auf die beteiligten Modulinstanzen hat es dabei keine Auswirkungen, ob eine ihrer Schnittstellen mit derselben oder unterschiedlichen Queues der Anwendung verbunden ist. Innerhalb der Anwendung muss die Verwendung einer Queue für verschiedene Stellen allerdings bewusst geschehen, da dies ansonsten zu fehlerhaftem Verhalten der Anwendung führen kann. Falls etwa eine Queue der Anwendung den eingehenden Schnittstelle verschiedener Module zugeordnet wird, müssen die darin eingefügten Nachrichten auch allen geforderten Schemata entsprechen, wenn mit einer Nachricht alle Module angesprochen werden sollen. Ein fehleranfälliges Szenario entsteht dann, wenn Queues als

ausgehende Schnittstelle des einen und eingehende Schnittstelle eines anderen Moduls verwendet werden. Allerdings entstehen dadurch offensichtlich sehr viele Einsatzmöglichkeiten, um mehrere Instanzen gleichzeitig anzusprechen oder verschiedene Instanzen miteinander zu verknüpfen.

Es sollte bei dem Benutzer eines Moduls liegen, ob er dieselbe Queue für verschiedene Instanzen verwenden möchte oder nicht und er sollte sich daher auch den sich daraus ergebenden Konsequenzen bewusst sein. Um dem Benutzer diese Einsatzmöglichkeiten zu gestatten sollte die Verwendung von Queues für verschiedene Instanzen nicht untersagt werden. Dabei spielt es keine Rolle, ob diese Instanzen von demselben oder einem anderen Modul erzeugt werden.

Mit dem neuen Befehl zur Schnittstellenzuweisung, wie er in Abschnitt 4.1.7.1 beschlossen wurde, ist es problemlos möglich eine mehrfache Zuweisung derselben Queue zu Schnittstellen unterschiedlicher Instanzen umzusetzen, da innerhalb der Schnittstellen-Zuweisung lediglich existierende Queues referenziert werden.

4.2 Compilerentwurf

Zur Realisierung des im vorherigen Unterkapitel entworfenen Modulkonzepts für Demaq müssen der DQL neue Befehle hinzugefügt werden (vgl. 4.1). Diese Befehle werden für den Modul-Import (vgl. 4.1.4), zur Schnittstellendefinition innerhalb eines Moduls (vgl. 4.1.5) und zur Erzeugung einer Modulinstanz (vgl. 4.1.7) samt Schnittstellenzuweisung und Parameter-Übergabe benötigt.

Damit das Demaq-System diese neuen Befehle verstehen und auswerten kann, muss der *Compiler* angepasst werden. In diesem Kapitel wird daher erklärt inwiefern der Compiler verändert werden muss, um mit dem Modulkonzept umgehen zu können. Das ist die Voraussetzung, um den Compiler entsprechend erweitern zu können und damit die Verwendung von Modulen in Demaq zu ermöglichen.

4.2.1 Umgang mit Modulen

Zunächst muss geklärt werden, wie der Compiler mit den Modulen umgehen soll. Dazu wird in Abschnitt 4.2.1.1 analysiert, wie aus einer modularisierten Anwendung eine funktionsfähige Demaq-Anwendung erstellt werden soll.

In Abschnitt 4.2.1.2 wird anschließend untersucht, wie diese Anbindung im Compiler realisiert werden kann.

4.2.1.1 Anbindung an eine Anwendung

Da die Anbindung von Modulen an eine Anwendung stark an die Verbindung zwischen einem externen Service und einer Demaq-Anwendung angelehnt wurde (vgl. 4.1.3) besteht ein erster Ansatz darin, ein Modul wie einen externen Service zu behandeln. Das heißt für alle in einer Anwendung erzeugten Modulinstanzen wird eine eigene Demaq-Instanz angelegt, welche über den Austausch von Nachrichten mit der Anwendung kommuniziert.

Diese Variante wurde allerdings bereits im Sprachentwurf (vgl. 4.1.1) ausgeschlossen, da die benötigte Netzwerk-Kommunikation eine massive Erhöhung der Laufzeit verursachen würde. Dies soll nach Anforderung 3.3.5 allerdings vermieden werden. Die Erzeugung mehrerer Demaq-Instanzen wirft an dieser Stelle zusätzlich weitere Probleme auf, z. B. unter welchen Ports die Anwendungen gestartet werden oder wie die Fehlerbehandlung realisiert wird. Somit ist diese Möglichkeit für eine Anbindung der Module an eine Anwendung nicht geeignet.

Eine vorteilhaftere Lösung ist, dass der Compiler aus einer modularisierten Anwendung wieder eine gewöhnliche Demaq-Anwendung erstellt. Das heißt alle in einer Anwendung enthaltenen Module, bzw. die davon erzeugten Instanzen werden in die bestehende Anwendung eingefügt und über die Schnittstellen an die Anwendung angebunden.

Ein großer Vorteil dieser Variante ist, dass aus einer Anwendung mit Modulen eine Anwendung erzeugt wird, welche zu einer Anwendung vor der Einführung des Modulkonzepts äquivalent ist. Wie von Anforderung 3.3.1 verlangt muss daher die Laufzeitumgebung nicht modifiziert werden, da sie weiterhin ausschließlich gewöhnliche Demaq-Anwendungen verarbeiten muss. Allerdings muss eine Lösung für ggf. entstehende Namenskonflikte zwischen einer Modulinstanz und der Anwendung oder anderen Modulinstanzen, z. B. bei sich überschneidenden Queuenamen, gefunden werden.

4.2.1.2 Realisierung der Anbindung

In diesem Abschnitt wird untersucht, wie eine solche Umschreibung von Anwendungen mit Modulen in äquivalente Anwendungen ohne Module realisiert werden kann.

Da der Demaq-Compiler bislang keine Phase zur Zwischencode-Erzeugung enthält besteht eine erste Möglichkeit darin, ihn um eine entsprechende Phase zu erweitern. Innerhalb dieser Phase werden Anwendungen mit Modulen umgeschrieben. Zusätzlich kann der erzeugte Zwischencode so gewählt werden, dass die nachfolgenden Phasen einfacher durchgeführt werden können als auf der bisherigen Repräsentation durch einen AST. Allerdings soll nach Anforderung 3.3.2 die Architektur des Compilers nicht verändert werden, wodurch diese Alternative auszuschließen ist.

Eine weitere Variante ist die Module in der Code-Erzeugungs-Phase in die Anwendung einzufügen. Dazu muss für jede Modulinstanz ein AST erzeugt werden, der wie auch die Anwendung serialisiert wird. Der erzeugte DQLX-Code der Anwendung und allen Modulinstanzen kann anschließend zu einer vollständigen Anwendung zusammengefügt werden.

Eine bessere Idee besteht allerdings darin, die Module mit einem vorangestellten Compiler (*Precompiler*) vor dem Aufruf des Demaq-Compilers der Anwendung hinzuzufügen. Das heißt bereits vor dem eigentlichen Kompilieren einer Demaq-Anwendung wird diese auf enthaltene Module untersucht. Alle gefundenen Module werden anschließend zur Anwendung hinzugefügt. Dies kann realisiert werden, indem der DQL-Code der Module in den der Anwendung eingefügt wird und die Module zudem mit der Anwendung verbunden werden. Um die Korrektheit der Module zu gewährleisten, müssen sie vor dem Einfügen in die Anwendung zumindest auf syntaktische und im Optimalfall auch auf semantische Fehler untersucht werden.

Ein großer Vorteil dieser Variante ist zum einen, dass eine Anwendung mit Modulen von den Optimierungen und Normalisierungen der Rewrite-Phase des Demaq-Compilers profitieren kann. Zum anderen muss die Phase der Code-Erzeugung nicht angepasst werden, da ihr bereits der AST einer Anwendung ohne Module übergeben wird. Zudem ist das Einfügen von Modulen streng

von der restlichen Funktionalität des Compilers getrennt. Allerdings muss durch diese Variante bestehende Funktionalität erneut bereitgestellt werden. Der Demaq-Compiler verfügt schließlich bereits über eine syntaktische sowie semantische Fehleranalyse. Da ein Modul außer den Schnittstellendefinitionen aus bestehenden Demaq-Konstrukten besteht und für das Einbinden eines Moduls lediglich zwei neue Befehle benötigt werden, muss diese existierende Fehlerbehandlung nur leicht erweitert werden, um zusätzlich mit Modulen umgehen zu können.

Eine weitere Möglichkeit besteht darin, das Zusammenfügen von Anwendung und Modulen innerhalb der Rewrite-Phase des Demaq-Compilers umzusetzen. Dadurch kann die im vorherigen Abschnitt angedachte Vorverarbeitung der Module in die bestehende Infrastruktur integriert werden. Hierbei wird für jede Modulinstanz ein AST erzeugt, welcher zum AST der Anwendung hinzugefügt wird. Wurde dies für alle erzeugten Modulinstanzen durchgeführt, kann der daraus entstehende AST mit der unveränderten Code-Erzeugung serialisiert werden.

Auch bei dieser Alternative kann von den Optimierungen und Normalisierungen des Compilers profitiert werden, sofern das Zusammenfügen bereits in einem frühen Durchlauf der Rewrite-Phase erfolgt. Weiterhin muss die Phase der Code-Erzeugung nicht verändert werden. Eine Einbindung der Module während der Rewrite-Phase passt zudem konzeptuell sehr gut, da das Einfügen von Modulen aus einer Umschreibung einer Anwendung mit Modulen in eine Anwendung ohne Module besteht. Da somit eine Verdoppelung bestehender Funktionalität vermieden werden kann, stellt diese Möglichkeit die beste Lösung dar.

4.2.2 Anpassung der Compiler-Phasen

In den folgenden Abschnitten wird beschrieben, wie die einzelnen Phasen des Demaq-Compilers (vgl. 2.5.4) angepasst werden müssen, um den im vorherigen Unterkapitel beschriebenen Umgang mit den Modulen zu realisieren.

Lexikalische Analyse

Damit der Compiler die neu eingeführten Befehle für den Import, die Instanzierung sowie die Schnittstellendefinition verarbeiten kann, muss zunächst der Lexer angepasst werden. Dadurch wird gewährleistet, dass dieser die neuen Befehle verarbeiten kann und sie nicht als Fehler bemängelt.

Parser

Bei der Verwendung von neuen Befehlen übergibt der Lexer dem Parser diesem bislang unbekannte Tokens. Daher muss die Grammatik dieser neuen Befehle dem Parser hinzugefügt werden. Da vom Parser zudem der AST erzeugt wird muss definiert werden, wie aus diesen Befehlen die entsprechenden Knoten für den AST generiert werden, welche Kindknoten diese wiederum besitzen und an welcher Stelle diese Knoten eingefügt werden.

Demaq-Rewrite-Phase

Hierbei handelt es sich um die wesentliche Phase, da in ihr zu Beginn die Module eingelesen und in die bestehende Anwendung eingebunden werden. Beim Einlesen eines Moduls werden für das Modul die Phasen der lexikalischen Analyse und der Syntaxanalyse sowie die semantische Analyse und die Normalisierungen aus der Rewrite-Phase durchgeführt. Die Optimierungen des Modul-AST werden dabei deaktiviert, da diese noch nach dem Zusammenfügen von Modul und Anwendung in der Rewrite-Phase der Anwendung durchgeführt werden.

Beim Einlesen wird das Modul auf syntaktische und semantische Korrektheit überprüft. Liegt kein Fehler vor, wird für die einzelnen Modulinstanzen jeweils ein eigener AST erzeugt, der allerdings wie bereits erwähnt nicht optimiert wird. Anschließend wird jeder dieser ASTs an den AST der Anwendung angehängt. Dies geschieht bereits zu einem frühen Zeitpunkt der Rewrite-Phase der Anwendung, sodass dieser zusammengesetzte AST anschließend noch von den Optimierungen profitiert.

Code-Erzeugung

Diese Phase ist von der Einführung eines Modulkonzepts nicht betroffen, da die gesamte Behandlung der Module in der Rewrite-Phase realisiert wird. Zu-

dem wird eine modularisierte Anwendung bewusst in eine Anwendung ohne Module umgeschrieben. Da diese Umschreibung bereits in der Rewrite-Phase stattfindet, müssen die Modul-Konstrukte (Import, Instanziierung, Schnittstellendefinition) bei der Code-Erzeugung nicht berücksichtigt werden.

5 Implementierung

Im vorherigen Kapitel wurde ein detaillierter Entwurf für das Modulkonzept erstellt. In diesem Kapitel wird die vorgenommene Implementierung beschrieben, welche sich an den veränderten Phasen des Demaq-Compilers orientiert. Im Einzelnen wurde die lexikalische Analyse, der Parser sowie die Demaq-Rewrite-Phase verändert.

Die Implementierung wurde im Rahmen der bestehenden Infrastruktur des Demaq-Compilers vorgenommen.

5.1 Lexikalische Analyse

Um Demaq den Umgang mit Modulen zu ermöglichen, wurden der DQL drei neue Befehle hinzugefügt (siehe Anhang A). Diese Befehle betreffen den Import, die Instanziierung und die Schnittstellendefinition eines Moduls. Der Lexer des Demaq-Compilers wurde um die entsprechenden Tokens erweitert, die sich aus den neu hinzugefügten Befehlen ergeben. Dadurch werden die für die neuen Befehle benötigten Schlüsselwörter erkannt und können an den Parser weitergegeben werden.

5.2 Parser

Der Parser wurde um die Grammatik der neuen Befehle erweitert, damit er aus den übergebenen Tokens die neuen Ausdrücke zusammensetzen kann. Zudem wurden ihm die Erstellungsvorschriften der neuen Befehle in AST-Knoten hinzugefügt, damit er die neuen Befehle als Knoten in den AST einfügen kann. Für die Schnittstellendefinition konnte dabei die bestehende Darstellung von Queues übernommen werden. Durch die Grammatik ist außerdem festgelegt, an welcher Stelle des AST die neuen Knoten einzufügen sind.

5.3 Demaq-Rewrite-Phase

Die Rewrite-Phase besteht aus mehreren Durchläufen durch den AST. Dabei werden zu allen DQL-Konstrukten (Queues, Regeln, etc.) Kontextinformationen gespeichert, damit in späteren Durchläufen darauf zugegriffen werden kann. Die Rewrite-Phase wurde derart erweitert, dass die neu hinzu gekommenen AST-Knoten bei den Durchläufen beachtet werden. Dadurch konnten die neuen Befehle mit Funktionalität versehen und Kontextinformationen zu den einzelnen Import- und Instanziierungs-Befehlen gespeichert werden. Die Funktionalität der neuen Befehle ermöglicht ein Umschreiben von Anwendungen mit Modulen in dazu äquivalente Anwendungen ohne Module. Des Weiteren wurde eine Prüfung auf semantische Fehler realisiert.

Die Umsetzung dieser Funktionalität wird in den folgenden Abschnitten beschrieben.

5.3.1 Semantische Analyse

Mit den neu eingeführten Befehlen entsteht eine Reihe potenzieller semantischer Fehler, die auftreten können. Die neuen Fehlerquellen betreffen ausschließlich den Import und die Instanziierung, da die Schnittstellendefinition semantisch mit der Erzeugung einer Queue identisch und die semantische Prüfung dafür im bestehenden System bereits implementiert ist.

Die semantischen Analyse wurde daher um eine Prüfung der Semantik der neuen Befehle erweitert. Diese Analyse reicht von einfachen Prüfungen, ob ein eingebundenes Modul existiert, bis hin zu umfangreicheren Prüfungen bei der Instanziierung eines Moduls. Nachdem sicher gestellt wurde, dass ein Modul importiert und ein eindeutiger Instanzname vergeben wurde, wird z. B. geprüft, ob alle Schnittstellen des Moduls mit Queues aus der Anwendung verbunden werden. Zudem wird untersucht, ob für alle im Modul angegeben Parameter bei der Instanziierung Werte übergeben werden.

Weiterhin müsste eine Typ-Prüfung der Parameter stattfinden. Das ist in der aktuellen Implementierung allerdings nicht enthalten, bzw. alle Parameter werden bisher als String behandelt. Der Grund dafür ist, dass der Demaq-Compiler bis zur Fertigstellung dieser Arbeit keine Typ-Prüfung konnte, wodurch eine Typ-Analyse der Parameter nicht möglich war.

5.3.2 Umschreiben von Anwendungen mit Modulen

Zur Realisierung der Umschreibung wurde im Compilerentwurf entschieden, aus jeder Modulinstanz einen AST zu erzeugen, welcher dem AST der Anwendung hinzugefügt wird. Vor dem Hinzufügen der Module bestand ein aus Demaq-Quellcode erzeugter AST aus dem Wurzelknoten *DQLApplicationModule*. Angelehnt an XQuery enthält dieser Wurzelknoten die Kindknoten *Prolog* und *Body*. Der Prolog-Knoten stellt den Elternknoten aller Prolog-Elemente dar, während der Body-Knoten als Kindknoten alle DQL-Konstrukte einer Demaq-Anwendung enthält.

Für die Realisierung des Modulkonzepts wurde dem AST einer Demaq-Anwendung ein Knoten für den Modul-Import hinzugefügt. Dieser befindet sich zwischen Prolog und Body, damit die Modul-Importe einer Anwendung an einer zentralen Stelle zu finden sind und die Instanziierung innerhalb des Quellcodes erst nach dem Import erfolgt. Die Instanziierung wurde als Kindknoten des Bodies in den AST eingefügt.

Für die Module wurde ein neuer Wurzelknoten definiert (*DQLLibraryModule*). Dieser Knoten enthält als ersten Kindknoten die Moduldeklaration. Diese enthält alle Schnittstellendefinitionen eines Moduls und ist unterteilt in ein- und ausgehende Schnittstellen. Anschließend folgen mit Prolog, Modul-Import und Body die Kindknoten einer gewöhnlichen Anwendung.

Das eigentliche Umschreiben von Anwendungen mit Modulen wurde folgendermaßen realisiert. Für jedes Modul, das auch instanziiert wird, wird ein AST erzeugt. Dieser AST wird für jede Instanz eines Moduls kopiert, umgeschrieben und anschließend dem AST der Anwendung hinzugefügt. Dieser Vorgang wird innerhalb der Rewrite-Phase der Anwendung rekursiv durchgeführt, damit auch in Modulen enthaltene Module beachtet werden. Das Umschreiben und Hinzufügen der Instanz-ASTs wird in den folgenden Abschnitten detailliert beschrieben. Die neuen AST-Knoten für den Import, die Instanziierung und die Schnittstellendefinition werden lediglich für das Einfügen der Module benutzt und nicht für die Code-Erzeugung benötigt. Daher werden sie nach dem Zusammenfügen von Anwendung und Modulen aus dem AST entfernt bzw. von der Code-Erzeugung nicht bearbeitet.

Zum Einfügen der Module müssen einerseits die Prolog-Knoten aller Module mit dem Prolog-Knoten der Anwendung verbunden werden. Der Prolog

muss pro Modul nur einmalig hinzugefügt werden, da sich die Prologe verschiedener Modulinstanzen desselben Moduls nicht unterscheiden. Andererseits müssen die Knoten aller DQL-Konstrukte einer Instanz als Kindknoten dem Body-Knoten der Anwendung hinzugefügt werden. Alle DQL-Konstrukte einer Instanz schließt auch die Schnittstellen mit ein. Diese müssen aus dem Moduldeklarations-Knoten kopiert und dem Body-Knoten des Moduls hinzugefügt werden, bevor dieser an den Body-Knoten der Anwendung angehängt wird.

Das Zusammenfügen der ASTs wurde dabei in den ersten Durchläufen der Rewrite-Phase umgesetzt. Dadurch wird ermöglicht, dass auf dem resultierenden AST Optimierungen durchgeführt werden können, weil die Optimierungen erst in den darauf folgenden Durchläufen ausgeführt werden. Durch das Verbinden der ASTs benötigen diese folgenden Durchläufe die Kontextinformationen aus den einzelnen Instanzen. Deswegen werden diese Kontextinformationen nach dem Zusammenfügen der ASTs denen der Anwendung hinzugefügt.

5.3.3 Umschreiben eines Moduls

Das Umschreiben eines Moduls wurde realisiert, indem beim Erreichen eines Import-Knotens für jedes Modul die Phasen der lexikalischen Analyse, der Syntaxanalyse sowie der Rewrite-Phase durchgeführt werden. Momentan wird dabei lediglich eine Einbindung der Module aus dem lokalen Dateisystem unterstützt. Die Optimierungen der Rewrite-Phase werden deaktiviert, da nach dem Zusammenfügen aller ASTs eine Optimierung des Gesamt-ASTs durchgeführt wird. Die erzeugten ASTs aller Module werden zwischengespeichert, um bei der Behandlung der Instanziierungs-Knoten zur Verfügung zu stehen. Wird ein Instanziierungs-Knoten erreicht und wurde der Modul-AST bereits erzeugt, so wird dieser AST kopiert, mit den Instanziierungs-Informationen umgeschrieben und für die spätere Anbindung an die Anwendung zwischengespeichert. Diese Instanziierungs-Informationen bestehen aus dem Instanznamen, dem Modulnamen, den Schnittstellenzuweisungen sowie den übergebenen Parameter-Werten. Die vorzunehmenden Umschreibungen betreffen die Schnittstellenzuweisung, das Einfügen der Parameter-Werte, die Umbenennung der Modul-Inhalte sowie die Fehlerbehandlung. Das Umbenennen der Inhalte eines Moduls ist deswegen nötig, um Namenskonflikte zu vermeiden.

Realisiert wurde das Umschreiben eines Modul-ASTs in einen Instanz-AST mithilfe eines *Mutators* [17]. Dabei handelt es sich um ein gebräuchliches *Entwurfsmuster* (*Design Pattern* [12]), welches die Möglichkeit bietet Modifikationen auf einer Datenstruktur vorzunehmen. Dabei wurde sich ein bestehendes Besucher-Muster (*Visitor-Pattern*) zunutze gemacht, welches den AST einer Demaq-Anwendung vollständig durchläuft. Die davon benötigten Besuchs-Methoden wurden durch den *Mutator* überschrieben.

Schnittstellenzuweisung Bei den Schnittstellen handelt es sich um gewöhnliche Queues, die mit den restlichen DQL-Konstrukten der Anwendung hinzugefügt werden sollen. Beim Zusammenfügen von Modul und Anwendung wird allerdings nur der Body-Knoten des Moduls dem Body-Knoten der Anwendung hinzugefügt. Da die Schnittstellen eines Instanz-ASTs dem Moduldeklarations-Knoten zugeordnet sind, werden sie durch den *Mutator* als Kindknoten an den Body-Knoten der Instanz angehängt. Dazu werden die Schnittstellen beim Durchlaufen des Moduldeklaration-Knotens gespeichert und beim Besuch des Body-Knotens an diesen angehängt.

Weiterhin muss der nachrichtenbasierte Datenaustausch realisiert werden. Zum einen müssen dazu alle in eine ausgehende Schnittstelle des Moduls eingefügte Nachrichten an die dieser Schnittstelle zugewiesenen Queue der Anwendung weitergeleitet werden. Zum anderen müssen alle Nachrichten, die in eine Queue der Anwendung eingefügt werden, welche einer eingehenden Schnittstelle eines Moduls zugewiesen ist, an diese Schnittstelle weitergeleitet werden. Zu diesem Zweck wird für jede Schnittstelle eine Regel erzeugt. Diese Regel realisiert die Weiterleitung aller Nachrichten, die in die Schnittstelle bzw. in die Queue der Anwendung eingefügt werden, in die entsprechende Richtung. Dabei wird der AST-Knoten für die Regel generiert und der Body-Knoten der Instanz um den erzeugten Regel-Knoten erweitert. Die Generierung dieser Regel geschieht unmittelbar nachdem die entsprechende Schnittstelle dem Body-Knoten der Instanz hinzugefügt wurde. Da die Moduldeklaration in ein- und ausgehende Schnittstellen unterteilt ist kann bestimmt werden, in welche Richtung die Weiterleitung erfolgen muss. Die zugehörige Queue aus der Anwendung wird aus den Kontextinformationen des jeweiligen Instanziierungs-Befehls ausgelesen, für welchen die aktuelle Umschreibung durchgeführt wird. Zu beachten war dabei, dass im Gegensatz zu den anderen DQL-Konstrukten

der Instanz innerhalb dieser Regeln der Name der Queue der Anwendung nicht umbenannt werden darf (siehe übernächster Abschnitt).

Einsetzen der Parameter-Werte Die bei der Instanziierung übergebenen Werte der Parameter werden im Zuge der Umschreibung für die entsprechenden Variablen-Referenzen innerhalb des Instanz-ASTs eingesetzt. Dieses Einsetzen wurde ebenfalls mithilfe des *Mutators* gelöst. Dabei wird für alle gefundenen Variablen-Referenzen anhand der Kontextinformation der aktuellen Instanziierung geprüft, ob ein gleichnamiger Parameter bei der Instanziierung übergeben wird. Ist das der Fall, wird die Variablen-Referenz durch den zugewiesenen Wert ersetzt. Wie bereits erwähnt werden dabei bislang ausschließlich Parameter vom Typ String unterstützt (vgl. 5.3.1).

Ein Problem tritt auf, falls sich eine Variablen-Definition innerhalb eines `for`- oder `let`-Statements befindet. Dabei kann innerhalb des ASTs nicht zwischen einer Variablen-Referenz oder der Definition einer `for`- bzw. einer `let`-Variable unterschieden werden. In diesem Fall kann es sein, dass innerhalb eines dieser Statements eine Variable definiert wird, welche den gleichen Namen wie ein Parameter hat. Werden alle gefundenen Parameter-Referenzen ersetzt, sind somit auch diese Variablen betroffen. Daher muss eine Möglichkeit gefunden werden, diese Variablen unberührt zu lassen. Leider gestaltet es sich sehr umständlich innerhalb des *Mutators* zu prüfen, ob es sich um eine solche `for`- oder `let`-Variable handelt. Als Lösung wurde daher eine semantisch Prüfung eingeführt. Dabei wird getestet, ob eine Variable innerhalb eines `for`- oder `let`-Statements mit dem gleichen Namen wie ein Parameter definiert wurde. Ist das der Fall, wird dem Benutzer eine entsprechende Fehlermeldung ausgegeben.

Vermeidung von Namenskonflikten Alle gleichartigen DQL-Konstrukte in einer Anwendung oder einem Modul müssen jeweils unterschiedliche Namen haben (vgl. 2.5.2). Zum Beispiel müssen alle Queuenamen innerhalb einer Anwendung eindeutig sein. Durch die Module werden der Anwendung DQL-Konstrukte hinzugefügt, deren Namen nicht bekannt sind. Zudem enthalten Instanzen desselben Moduls gleichnamige DQL-Konstrukte, was in jedem Fall zu Konflikten führt. Im Rahmen der Umschreibung muss daher gewährleistet werden, dass keine Namenskonflikte auftreten.

Als Lösung für dieses Problem wurde entschieden, dass allen DQL-Konstrukten einer Instanz der Instanzname als Präfix vorangestellt wird. Diese Instanznamen müssen in der Anwendung oder dem Modul, in der (dem) die Instanz erzeugt wird, eindeutig sein. Dabei müssen evtl. definierte Namensräume beachtet werden, welche den DQL-Konstrukten zugewiesen sein können. Der Name einer Queue kann z. B. *shop:customers* lauten, wobei *shop* den Namensraum darstellt, dem in einer Namensraumdeklaration im Prolog ein eindeutiger URI zugewiesen werden muss. Daher darf nur dem eigentlichen Namen (hier *customers*) der Instanzname als Präfix vorangestellt werden, um die Zuordnung zu einem Namensraum nicht zu verlieren. In der vorliegenden Implementierung wird dieser Fall allerdings nicht beachtet sondern dem gesamten Namen wird der Instanzname als Präfix vorangestellt.

Zur Umsetzung dieser Lösung werden durch den *Mutator* alle DQL-Konstrukte sowie alle Stellen, an denen diese DQL-Konstrukte benutzt werden, umbenannt. Beim Anlegen einer Regel wird z. B. auf eine Queue oder ein Slicing verwiesen, auf welcher (welchem) die Regel definiert ist. Innerhalb der Zugriffsfunktionen `qs:slicekey("SlicingName")`, `qs:queue("QueueName")`, etc. wird ebenfalls auf existierende DQL-Konstrukte zugegriffen.

Alle Namen von DQL-Konstrukten sind in der Symboltabelle (vgl. 2.4) unter eindeutigen IDs gespeichert, welche innerhalb der AST-Knoten referenziert werden. Anhand dieser IDs können die Namen der DQL-Konstrukte ausgelesen und ihnen anschließend der Instanzname als Präfix vorangestellt werden. Daraufhin wird dieser neue Name der Symboltabelle hinzugefügt und die zugeordnete ID in den AST-Knoten gespeichert, wobei die alte ID überschrieben wird. Wie bereits erwähnt sind dabei die erstellten Weiterleitungs-Regeln für die Schnittstellen zu beachten (siehe Abschnitt *Schnittstellenzuweisung* zuvor). Innerhalb dieser Regeln wird der Name der Anwendungs-Queue nicht umgeschrieben.

Ebenfalls umgeschrieben werden müssen die Zugriffsfunktionen. Weil diese Funktionen auch mit Variablen-Referenzen oder Pfadausdrücken aufgerufen werden können, kann der Typ des Parameters nicht als String angenommen werden. Da es sich aber um einen Ausdruck handelt, der zu einen String ausgewertet wird, kann die Voranstellung des Instanznamens durch die XQuery-Funktion `fn:concat(String, String)` erreicht werden. Diese Funktion fügt die beiden String-Parameter zu einem String zusammen.

Fehlerbehandlung Innerhalb eines Moduls können wie in normalen Demaq-Anwendungen für Queues und Regeln Errorqueues definiert werden. Wird innerhalb des Moduls eine globale Errorqueue definiert muss dafür eine Lösung gefunden werden, damit keine Konflikte mit der globalen Errorqueue der Anwendung oder den globalen Errorqueues anderer Module entstehen. Dazu muss bei der Umschreibung des Moduls allen Queues und Regeln, denen keine eigenen Errorqueue zugewiesen ist, diese globale Errorqueue explizit hinzugefügt werden. Anschließend muss die Deklaration der globalen Errorqueue aus dem Prolog des Moduls entfernt werden. Das ist in der aktuellen Implementierung allerdings nicht enthalten.

5.3.4 Zusammenfügen der ASTs

Nachdem die ASTs aller Modulinstanzen umgeschrieben und zwischengespeichert wurden, können sie dem AST der Anwendung hinzugefügt werden. Diese umgeschriebenen ASTs aller Modulinstanzen beinhalten auch innerhalb von Instanzen enthaltene Module, die bereits den ASTs der Instanzen hinzugefügt wurden. Nach der erfolgten Umschreibung der Modulinstanzen werden beim Durchlaufen des Body-Knotens der Anwendung alle Body-Knoten der Instanzen ausgelesen und ihm als Kindknoten hinzugefügt. An dieser Stelle werden auch die Kontextinformationen der Anwendung um die Kontextinformationen der Instanzen erweitert. Wird der Prolog-Knoten der Anwendung erreicht, wird er um die Prolog-Knoten aller importierten Module erweitert (sofern diese Module auch instanziiert werden). Die Prolog-Knoten der Module werden in der aktuellen Implementierung unverändert übernommen, weshalb es zu Namenskonflikten mit Prolog-Inhalten aus der Anwendung oder anderen Modulen kommen kann. Ein Problem dabei ist die Behandlung von Namensräumen innerhalb des Prologs. Die Freilegung der Namensräume eines Moduls bedeutet, dass der Anwendung Zugriff auf das Modul erlaubt wird, was im Entwurf allerdings untersagt wurde.

Die Instanziierungs-Knoten werden ausschließlich für das Umschreiben und Einfügen der Instanz-ASTs benötigt und sind für die Code-Erzeugung irrelevant. Deswegen werden sie nach dem Zusammenfügen aus dem AST der Anwendung entfernt, indem sie durch einen leeren AST-Knoten ersetzt werden. Die Modul-Import-Knoten werden im AST gelassen und von der Code-

Erzeugung nicht beachtet, wodurch sie im erzeugten DQLX-Code nicht vorkommen. Die Knoten für die Moduldeklaration sowie für die Modul-Importe innerhalb des ASTs einer Instanz werden dadurch entfernt, dass dem AST der Anwendung lediglich die Prolog- und Body-Knoten des Instanz-ASTs hinzugefügt werden.

6 Evaluation

Im bisherigen Verlauf dieser Arbeit wurden die Anforderungen, der Entwurf und die Implementierung eines Modulkonzepts für Demaq betrachtet. In diesem Kapitel wird dieses entworfene Modulkonzept evaluiert, um zu prüfen, ob alle in Kapitel 3 aufgestellten Anforderungen erfüllt wurden.

In Unterkapitel 6.1 werden dazu zunächst die Anwendungsfälle (vgl. 3.1) mit dem entwickelten Modulkonzept umgesetzt. Danach wird in Unterkapitel 6.2 die Performance von modularisierten Anwendungen gemessen und in Unterkapitel 6.3 findet eine abschließende Diskussion statt.

6.1 Umsetzung der Anwendungsfälle

Dieses Unterkapitel untersucht die in Unterkapitel 3.1 vorgestellten Anwendungsfälle darauf, wie sie mithilfe der Module umgesetzt werden können. Dazu wird jeweils der Modulcode sowie der Code zur Anbindung in einer Anwendung vorgestellt und erläutert. Der Anwendungs-Code enthält außer der Einbindung des Moduls keine weitere Funktionalität.

6.1.1 Anwendungsfall 1, Beispiel a)

In Beispiel a) von Anwendungsfall 1 ging es um die Benachrichtigung von Lieferanten per SOAP-Nachricht, sobald ein Artikel in einem Online-Shop zur Neige geht. Bei der Umsetzung mit DQL entstanden dabei redundante Codefragmente. Mithilfe einer XQuery-Funktion konnte zumindest ein Teil dieser Fragmente faktorisiert werden, in Anwendungsfall 2, Beispiel a) wurde diese Funktion zudem in ein XQuery-Modul ausgelagert.

6.1.1.1 Umsetzung des Moduls

Im Gegensatz zu XQuery-Modulen ist es mit DQL-Modulen möglich, den gesamten redundanten Bereich in ein Modul auszulagern (siehe Listing 6.1). Somit kann die gesamte Regel zur Benachrichtigung eines Lieferanten durch ein Modul bereitgestellt werden (Zeilen 9-21). Damit die Anwendung das Modul ansprechen kann wird eine eingehende Schnittstelle definiert (Zeile 1), welche die Benachrichtigung anstößt, sobald von der Anwendung eine Nachricht darin eingefügt wird. Über die modulinterne Gateway-Queue (Zeile 7) wird die Benachrichtigung an den Lieferanten gesendet. In den Zeilen 3 bis 5 werden die verwendeten Parameter definiert, welche bei der Instanziierung des Moduls von der Anwendung übergeben werden müssen. Dies sind der Port sowie der URL des Lieferanten und das zu verwendende Transportprotokoll. Die Werte werden als Properties innerhalb der Regel (Zeilen 17-18) der Gateway-Queue mitgegeben. Dadurch kann das Modul für verschiedene Lieferanten eingesetzt werden.

Listing 6.1: Modulcode zu Anwendungsfall 1 - Beispiel a).

```
1  declare input queue lowStockSupplier kind basic mode persistent validate supplier.
       xsd;
2
3  declare variable $supplierPort external;
4  declare variable $supplierURL external;
5  declare variable $supplierInterface external;
6
7  create queue out kind outgoing interface "" port "" mode persistent;
8
9  create rule briefSupplier for lowStockSupplier
10   enqueue message
11     <s:Envelope xmlns:s="http://www.w3.org/2001/12/soap-envelope">
12       <s:Header>
13       </s:Header>
14       <s:Body>
15         {.}
16       </s:Body>
17     </s:Envelope>
18   into out
19   with comm:URL value $supplierURL
20   with comm:DestinationPort value $supplierPort
21   with comm:TransportProtocol value $supplierInterface;
```

6.1.1.2 Umsetzung der Anbindung

In der Anwendung (siehe Listing 6.2) wird das erstellte Modul zunächst unter dem Namen *briefSupplier* eingebunden (Zeile 1), damit später mehrere Instanzen davon erzeugt werden können. Im Gegensatz zur ursprünglichen Version ohne Module (vgl. Listing 3.1 aus Kapitel 3) muss hier keine *outgoing*-Queue angelegt werden, da diese bereits im Modul vorhanden ist. Die Basic-Queues (Zeilen 3 u. 4) für die zentrale Anbindung der Benachrichtigungs-Funktionalität an die Anwendung müssen jedoch wie zuvor erzeugt werden. Allerdings wird diesmal die Funktionalität durch ein angebundenes Modul statt innerhalb der Anwendung realisiert. Dazu werden mithilfe des neuen `create binding`-Statements zwei Instanzen des Moduls erzeugt (*briefSupplierX*, Zeile 6 sowie *briefSupplierY*, Zeile 10), welchen die jeweils entsprechende Basic-Queue als eingehende Schnittstelle zugewiesen wird. Nach den Schnittstellenzuweisungen müssen den Instanzen zudem noch die benötigten Parameter *supplierPort*, *supplierURL* sowie *supplierInterface* übergeben werden. Dadurch werden erfolgreich zwei Instanzen des Moduls erzeugt und an die Anwendung angebunden.

Listing 6.2: Programmcode zu Anwendungsfall 1 - Beispiel a).

```
1  import dqlModule briefSupplier at "modules/briefSupplier.dql";
2
3  create queue lowStockSupplierX kind basic mode persistent;
4  create queue lowStockSupplierY kind basic mode persistent;
5
6  create binding briefSupplierX for briefSupplier
7    assign input lowStockSupplierX as lowStockSupplier
8    with supplierPort value 8020 with supplierURL value "www.supplier-x.com" with
         supplierInterface value "comm:HttpPost";
9
10 create binding briefSupplierY for briefSupplier
11   assign input lowStockSupplierY as lowStockSupplier
12   with supplierPort value 8030 with supplierURL value "www.supplier-y.com" with
         supplierInterface value "comm:HttpPost";
```

6.1.2 Anwendungsfall 1, Beispiel b)

Dieses Beispiel behandelte E-Mails, welche von unterschiedlichen Absender-Adressen versendet werden. Die Umsetzungsvariante in DQL enthielt erneut redundante Fragmente und eine Auslagerung in eine XQuery-Funktion war in diesem Fall aus semantischen Gründen nicht möglich.

6.1.2.1 Umsetzung des Moduls

Als Modul lässt sich die Funktionalität des E-Mail-Versands samt Individualisierung der Absender-Adresse wie in Listing 6.3 dargestellt realisieren. Das Modul definiert zunächst eine eingehende Schnittstelle (Zeile 1), über welche der E-Mail-Versand aus der Anwendung heraus angestoßen werden kann. Diese Schnittstelle verlangt eingehende Nachrichten in denen der Empfänger, der Betreff und die Nachricht der E-Mail angegeben werden (vgl. Listing 3.4 aus Abschnitt 3.1.1). Der Versand der E-Mails erfolgt über eine *outgoing*-Queue, welche als modulinterne Gateway-Queue definiert ist (Zeile 6). In den Zeilen 3 und 4 werden zudem Parameter definiert, damit der URL (*$url*) und die Absender-Adresse (*$from*), über welche die E-Mails versendet werden, aus der Anwendung übergeben werden können bzw. müssen. Anschließend wird in der Regel *sendMail* (Zeilen 8-13) die eigentliche Funktionalität des Moduls umgesetzt, welche durch in der eingehenden Schnittstelle ankommende Nachrichten angestoßen wird. Dabei wird die in dieser Nachricht übergebene E-Mail-Nachricht mit den Properties, die zum einen aus den Parametern und zum anderen aus der Kontext-Nachricht kommen, in die ausgehende Gateway-Queue des Moduls eingefügt.

Listing 6.3: Modulcode zu Anwendungsfall 1 - Beispiel b).

```
1  declare input queue mails kind basic mode persistent validate mail.xsd;
2
3  declare variable $url external;
4  declare variable $from external;
5
6  create queue out kind outgoing interface "smtp" port "25" mode persistent;
7
8  create rule sendMail for mails
9    enqueue message //msg into out
10   with comm:URL value $url
11   with comm:From value $from
```

```
12   with comm:To value //to/text()
13   with comm:Subject value //subject/text();
```

6.1.2.2 Umsetzung der Anbindung

Zunächst wird das E-Mail-Modul unter dem Namen *sendMail* in die Anwendung (siehe Listing 6.4) importiert (Zeile 1), damit innerhalb der Anwendung Instanzen davon erzeugt werden können. Danach werden in den Zeilen 3 und 4 die Queues für die eingehenden Schnittstellen der beiden zu erzeugenden Instanzen definiert. Dadurch wird die benötigte Infrastruktur geschaffen, um in den Zeilen 6-8 bzw. 10-12 zwei Instanzen des Moduls erzeugen zu können. Den jeweiligen Instanzen wird als eingehende Schnittstelle die entsprechende Basic-Queue (Zeile 9 bzw. 14) zugewiesen. Eine ausgehende Schnittstelle ist wie im vorherigen Beispiel nicht vorhanden. Weiterhin werden die Instanzen durch die Übergabe der Parameter *url* und *from* individualisiert, was den Versand von verschiedenen Absender-Adressen ermöglicht.

Listing 6.4: Programmcode zu Anwendungsfall 1 - Beispiel b).

```
1  import dqlModule sendMail at "modules/sendMail.dql";
2
3  create queue mailsService kind basic mode persistent;
4  create queue mailsOrder kind basic mode persistent;
5
6  create binding mailsService for sendMail
7    assign input mailsService as mails
8    with url value "www.demaq.net" with from value "service@demaq.net";
9
10 create binding mailsOrder for sendMail
11   assign input mailsOrder as mails
12   with url value "www.demaq.net" with from value "order@demaq.net";
```

6.1.3 Anwendungsfall 2, Beispiel b)

Dieses Beispiel bestand aus einer Kundenverwaltung, welche derart umgesetzt werden sollte, dass sie in verschiedenen Anwendungen verwendet werden kann. Die Funktionalität bestand darin, Kunden persistent zu speichern und bei Bedarf wieder auszugeben. Als mögliche Lösung wurde in Kapitel 3 die Realisierung in einer eigenen Anwendung vorgestellt.

6.1.3.1 Umsetzung des Moduls

Mithilfe des Modulkonzepts lässt sich die Kundenverwaltung als Modul (siehe Listing 6.5) realisieren. Dabei ähnelt die Umsetzung stark der Variante mit der eigenen Anwendung aus Abschnitt 3.1.2 (vgl. Listing 3.9). Allerdings mit dem Unterschied, dass die Kommunikation zwischen der Anwendung und der Kundenverwaltung durch Schnittstellen gelöst wird statt durch Gateway-Queues. Diese Schnittstellen bestehen aus zwei eingehenden (Zeilen 1 u. 2) und zwei ausgehenden Schnittstellen (Zeilen 3 u. 4). Über die eingehenden Schnittstellen kann aus der Anwendung ein neuer Kunden-Datensatz zur Verwaltung übergeben (Zeile 1) und die Ausgabe eines zuvor eingefügten Kunden durch die Übergabe der Kunden-ID angestoßen werden (Zeile 2). Die ausgehenden Schnittstellen übergeben zum einen eine Meldung, ob der Kunde gespeichert wurde oder nicht (Zeile 3), und gegeben zum anderen die angefragten Kundendaten aus (Zeile 4). In der in Zeile 6 definierten Basic-Queue *customers* werden die eingehenden Kundendaten gespeichert. Anschließend werden in den Zeilen 8 und 9 zwei Slicings erzeugt, um einzelne Kunden-Datensätze anhand der E-Mail oder der ID ansprechen zu können. Mithilfe der beiden Regeln (Zeilen 11-23 sowie 25-29) werden eingehende Kunden gespeichert bzw. ausgegeben.

Listing 6.5: Modulcode zu Anwendungsfall 2 - Beispiel b).

```
1  declare input queue newCustomer kind basic mode persistent validate newCustomer.
       xsd;
2  declare input queue getCustomer kind basic mode persistent validate getCustomer.
       xsd;
3  declare output queue newCustomerResponse kind basic mode persistent;
4  declare output queue getCustomerResponse kind basic mode persistent;
5
6  create queue customers kind basic mode persistent;
7
8  create slicing property customerMail queue customers value //email/text() require
       fn:false();
9  create slicing property customerID queue customers value //id/text() require fn:
       false();
10
11 create rule saveCustomer for newCustomer
12   let $custExists := qs:slice(//email/text(), "customerMail")[position() = last()]
13   return
14     if(not empty($custExists)) then
```

```
15        enqueue message
16          <result>Error: The e-mail {//email/text()} already exists.</result>
17        into newCustomerResponse
18      else (
19        enqueue message . into customers,
20        enqueue message
21          <result>Success: The data has been saved.</result>
22        into newCustomerResponse
23      );
24
25  create rule getCustomerByID for getCustomer
26    enqueue message {
27      let $customer := qs:slice(//id/text(), "customerID")[position() = last()]
28      return $customer
29    } into getCustomerResponse;
```

6.1.3.2 Umsetzung der Anbindung

Zur Anbindung des Moduls an eine Anwendung muss das Modul zunächst importiert werden (Zeile 1). Weiterhin müssen entsprechende Queues (Zeilen 3-6) vorhanden sein, an welche das Modul angebunden werden kann. Anschließend wird durch das `create binding`-Statement (Zeilen 8-10) eine Instanz des Moduls erzeugt, indem die angelegten Queues den entsprechenden ein- bzw. ausgehenden Schnittstellen zugewiesen werden. Dabei werden zuerst die Queues für die eingehenden (Zeile 9) und anschließend die Queues für die ausgehenden Schnittstellen (Zeile 10) zugewiesen. Mehrere Zuweisungen können dabei durch ein Komma getrennt werden.

Listing 6.6: Programmcode zu Anwendungsfall 2 - Beispiel b).

```
1  import dqlModule customers at "modules/customers.dql";
2
3  create queue saveCustomer kind basic mode persistent;
4  create queue getCustomerByID kind basic mode persistent;
5  create queue saveCustomerResponse kind basic mode persistent;
6  create queue getCustomerResponse kind basic mode persistent;
7
8  create binding customerManagement for customers
9    assign input saveCustomer as newCustomer, getCustomerByID as getCustomer
10   assign output saveCustomerResponse as newCustomerResponse,
         getCustomerResponse as getCustomerResponse;
```

6.2 Performance

Im Sprachentwurf wurde die Entscheidung getroffen, den Zugriff von einer Anwendung auf ein Modul als auch umgekehrt durch einen nachrichtenbasierten Datenaustausch zu realisieren (vgl. 4.1.3). Das hat zur Folge, dass pro Modulinstanz für jede Modul-Schnittstelle im Modul und in der Anwendung eine Queue existiert. Bei einer eingehenden Schnittstelle muss beim Einfügen einer Nachricht in die Queue auf der Anwendungs-Seite diese Nachricht kopiert und an die Queue des Moduls weitergeleitet werden. Bei einer ausgehenden Schnittstelle verhält es sich umgekehrt, wenn eine Nachricht in die Queue des Moduls eingefügt wird. Durch diese Entscheidung konnte die Benutzbarkeit der Module erhöht werden, weil die entsprechenden Queues in der Anwendung (dem Modul) völlig unabhängig von einem daran angebundenen Modul (einer das Modul einbindenden Anwendung) benutzt werden können. Durch die zusätzlich benötigte Kommunikation entsteht allerdings ein erhöhter Nachrichtenaustausch, welcher in Konflikt mit der Anforderung nach Erhaltung der Laufzeit steht (vgl. 3.3.5).

In diesem Abschnitt wird untersucht, inwiefern sich diese Entscheidung auf die Laufzeit von Anwendungen, welche Module verwenden, auswirkt. Dazu wird eine mit Modulen realisierte Anwendung mit einer äquivalenten Anwendung ohne Module verglichen. Der Code der gewöhnlichen Anwendung ist in Listing 6.7 dargestellt und erzeugt eine Anwendung, die eingehende Nachrichten lediglich wieder zurückschickt und ansonsten keine Funktionalität bietet. Diese Anwendung wird im Folgenden als Ping-Pong Anwendung bezeichnet.

Listing 6.7: Code der Ping-Pong Anwendung ohne Module.

```
1 create queue ping kind incoming interface "http" port "2342" response pong mode
      persistent;
2
3 create rule pingpong for ping
4   enqueue message . into pong;
```

Die modularisierte Anwendung wird durch den Code in Listing 6.8 erzeugt und stellt zum einen die Infrastruktur zum Einbinden des Moduls zur Verfügung (Zeile 3). Zum anderen wird das Modul importiert (Zeile 1) und instanziiert (Zeilen 5-7). Der dazu gehörige Modulcode ist in Listing 6.9 dargestellt und realisiert die Ping-Pong Funktionalität.

Listing 6.8: Code der Ping-Pong Anwendung mit Modul.

```
1 import dqlModule pingpong at "pingpong_module.dql";
2
3 create queue ping kind incoming interface "http" port "2342" response pong mode
      persistent;
4
5 create binding ping for pingpong
6   assign input ping as ping
7   assign output pong as pong;
```

Listing 6.9: Code des zugehörigen Moduls.

```
1 declare input queue ping kind basic mode persistent;
2 declare output queue pong kind basic mode persistent;
3
4 create rule pingpong for ping
5   enqueue message . into pong;
```

Diese einfache Anwendung wurde bewusst gewählt, weil sie sehr deutlich das bestehende Problem zeigt, da in dieser Anwendung durch die Modularisierung ein sehr großer Overhead entsteht. Das hat den Grund, dass es sich dabei um einen Extremfall handelt, weil alle in der Anwendung existierenden Queues zur Anbindung des Moduls verwendet werden. Somit werden doppelt so viele Nachrichten im Vergleich zu derselben Anwendung ohne Modul versendet. Dieses erhöhte Nachrichtenaufkommen relativiert sich allerdings in größeren Anwendungen, welche nicht ausschließlich aus der Anbindung von Modulen bestehen, da in diesen der Overhead durch Module weitaus weniger ins Gewicht fällt.

Tabelle 6.1: Messtabelle Ping-Pong. (LZ = Laufzeit)

Nachrichtengröße	*LZ ohne Modul (s)*	*LZ mit Modul (s)*
5kB	95	174
10kB	137	259
20kB	227	437

In Tabelle 6.1 werden die Ergebnisse der Testläufe dargestellt. Es wurden dazu jeweils 1000 XML-Nachrichten in unterschiedlichen Größen (5kB, 10kB und 20kB) an die Anwendung gesendet und die dafür benötigte Laufzeit in

Sekunden gemessen. Alle Messungen wurden dabei auf einer optimiert übersetzten Version [2] des Demaq-Laufzeitsystems (Revision 2242) ausgeführt. Das Testsystem bestand aus einem AMD Athlon(tm) 64 X2 4600+ @ 2.4 GHz mit 3.8 Gb RAM. Als Betriebssystem wurde openSuse 11.0 eingesetzt, die Version des Kernels war 2.6.25.18.

Die Ergebnisse bestätigen deutlich das angesprochene Problem. Die Anwendung mit Modul benötigt in allen Fällen ungefähr die doppelte Ausführungszeit wie dieselbe Anwendung ohne Modul. Dies erklärt sich mit dem zusätzlichen Versand der Nachrichten zwischen den Modul-Schnittstellen und den Queues der Anwendung.

Beim Entwurf des Modulkonzepts in Kapitel 4 wurde diese Verlängerung der Laufzeit allerdings bewusst in Kauf genommen, um eine komfortable Benutzbarkeit der Module zu ermöglichen und somit Anforderung 3.3.6 zu erfüllen.

6.3 Diskussion

Diese Diskussion untersucht, ob die funktionalen und nicht-funktionalen Anforderungen aus Kapitel 3 durch das entwickelte Modulkonzept erfüllt werden. Dazu werden in Abschnitt 6.3.1 in diesem Unterkapitel zunächst die funktionalen und anschließend in Abschnitt 6.3.2 die nicht-funktionalen Anforderungen untersucht.

6.3.1 Erfüllung der funktionalen Anforderungen

Die Anwendungsfälle aus Kapitel 3 konnten in Unterkapitel 6.1 ohne Schwierigkeiten mit dem Modulkonzept realisiert werden. Alle vorgestellten Umsetzungen lassen erkennen, dass ein Modul wie verlangt unabhängig von der eigentlichen Anwendung entwickelt werden kann (vgl. 3.2.1). Wie von Anforderung 3.2.3 verlangt, können dazu alle in Demaq zur Verfügung stehenden DQL-Konstrukte verwendet werden. Allein das Modul aus Beispiel 6.1.3 enthält mit Queues, Slicings, Properties und Regeln alle verfügbaren DQL-Konstrukte. Da die Module nur über die definierten Schnittstelle von der Anwendung angesprochen werden dürfen, kann eine unabhängige Wartung (vgl. 3.2.2) der Module jederzeit durchgeführt werden. Durch die Schnittstellen wird weiterhin ermöglicht, dass zwischen einem Modul und einer Anwendung beliebig Nachrichten

ausgetauscht werden können (vgl. 3.2.7). Wurde ein Modul erstellt, lässt es sich durch den Import-Befehl problemlos in verschiedene Anwendungen einbinden (vgl. 3.2.4). Anschließend lassen sich eine (vgl. 6.1.3) oder mehrere (vgl. 6.1.1 und 6.1.2) Instanzen innerhalb einer Anwendung davon erzeugen. Ein Modul kann somit wie von Anforderung 3.2.6 verlangt mehrfach in eine Anwendung eingebunden werden. Weiterhin können die Module auf unterschiedliche Szenarien zugeschnitten werden, wodurch die Forderung nach Individualisierbarkeit (vgl. 3.2.5) erfüllt ist. Dies geschieht über die Parameter, mit welchen das Modul individualisiert werden kann, und welche in den ersten beiden Beispielen verwendet werden. In Beispiel 6.1.1 wurde damit der Versand von Nachrichten an unterschiedliche Lieferanten realisiert und in Beispiel 6.1.2 konnte der Versand von E-Mails von unterschiedlichen Absendern sowie über unterschiedliche URLs umgesetzt werden.

6.3.2 Erfüllung der nicht-funktionalen Anforderungen

Wie von Anforderung 3.3.1 verlangt, wurde die Laufzeitumgebung von Demaq nicht verändert. Das ist vor allem der Entscheidung im Compilerentwurf zu verdanken, dass aus Anwendungen mit Modulen äquivalente Anwendungen ohne Module erzeugt (vgl. 4.2.1) werden. Dadurch müssen von der Laufzeitumgebung keine neuen Sprachkonstrukte behandelt werden.

Auch die Architektur des Compilers wurde nicht verändert (vgl. 3.3.2), da die nötigen Änderungen am Compiler der bereits existierenden Rewrite-Phase hinzugefügt wurden.

Für die Realisierung des Modulkonzepts wurden der DQL drei neue Befehle hinzugefügt, welche sich allesamt an bestehenden Befehlen orientieren. Der Import-Befehl wurde dem bestehenden Import von XQuery-Modulen angepasst und unterscheidet sich davon nur in der Syntax und der fehlenden Namespace-Angabe. Die Befehle zur Erzeugung von ein- und ausgehenden Schnittstellen wurden an die Erstellung gewöhnlicher Queues angelehnt und besitzen nur deswegen eine unterschiedliche Syntax, um ihre Funktion als Schnittstellendefinition zu verdeutlichen. Der neue Befehl zur Modul-Instanzierung wurde ebenfalls an bestehende DQL-Konstrukte angepasst. Für die Übergabe der Parameter wurde etwa das `with ...value ...`-Statement übernommen, welches bislang zur Übergabe von Properties innerhalb des `enqueue-`

Statements benutzt wird. Auch der Zugriff einer Anwendung auf ein Modul sowie umgekehrt wurde unter anderem aus dem Grund durch einen nachrichtenbasierten Datenaustausch realisiert, um sich in das bestehende Konzept von Demaq einzufügen.

Durch die Einführung des Modulkonzepts wurde zudem die Funktionalität (vgl. 3.3.4) von bestehenden Anwendungen nicht beeinträchtigt. Das lässt sich damit begründen, dass durch das Modulkonzept bestehende Anwendungen nicht verändert werden müssen, sondern Module optional zu beliebigen neuen oder bereits existierenden Anwendungen hinzugefügt werden können.

Auf die Laufzeit (vgl. 3.3.5) von modularisierten Anwendungen wurde in Unterkapitel 6.2 näher eingegangen. Diese Anforderung konnte aufgrund des gewählten nachrichtenbasierten Datenaustauschs (vgl. 4.1.3) nicht erfüllt werden. Allerdings wurde dadurch eine einfache Benutzbarkeit (vgl. 3.3.6) der Module erreicht, da somit keine Zugriffsbeschränkungen auf Modul-Schnittstellen oder auf die zugehörigen Queues in der Anwendung benötigt werden. Im Compilerentwurf wurde zudem entschieden, dass Anwendungen mit Modulen zu Beginn der Rewrite-Phase des Demaq-Compilers in äquivalente Anwendungen ohne Module umgeschrieben werden (vgl. 4.2.1). Dadurch könnte der durch Module entstehende Nachrichten-Overhead mit geeigneten Optimierungs-Rewrites reduziert werden, da die Nachrichten-Weiterleitungen immer nach demselben Muster erstellt werden.

7 Zusammenfassung & Ausblick

In diesem Kapitel werden die Ergebnisse dieser Arbeit in Unterkapitel 7.1 zusammengefasst sowie ein Ausblick auf mögliche Weiterentwicklungen des entwickelten Modulsystems in Unterkapitel 7.2 gegeben.

7.1 Zusammenfassung

Im Rahmen der vorliegenden Arbeit wurde dem existierenden Demaq-System ein Modulkonzept hinzugefügt, mit welchem Anwendungsteile in Module ausgelagert werden können. Diese Module können in beliebigen Demaq-Anwendungen eingebunden werden, wodurch die Anwendungsentwicklung erheblich vereinfacht wird. Wiederkehrende Anwendungsteile können somit modularisiert sowie individualisiert und in verschiedenen Anwendungen eingesetzt werden.

In Kapitel 3 wurden mithilfe verschiedener Anwendungsfälle die Anforderungen für das Modulkonzept erarbeitet. Die Ziele hierbei waren, dass die Module trotz der Einbindung in einer Anwendung wartbar bleiben und zum bestehenden Konzept passen. Weiterhin sollte sich ihre Benutzbarkeit möglichst einfach gestalten und eine Anwendung mit Modulen sollte sich in ihrer Laufzeit nicht von bisherigen Anwendungen unterscheiden.

Kapitel 4 befasste sich anschließend mit dem Entwurf des Modulkonzepts. Dieser ist in den Sprachentwurf und den Compilerentwurf unterteilt.

Im Sprachentwurf wurde die Beschaffenheit der Module bestimmt. Dabei wurde entschieden, dass Module mit allen verfügbaren DQL-Konstrukten erstellt und unter einer beliebigen URI abgelegt werden können. Die Anbindung eines Moduls an eine Anwendung geschieht sowohl über neue Befehle für den Import sowie die Instanziierung von Modulen als auch über die Schnittstellen,

welche innerhalb des Moduls festgelegt werden müssen. Weiterhin wurde festgelegt, dass der Zugriff auf Module durch einen Austausch von Nachrichten zwischen Queues der Anwendung und den Schnittstellen des Moduls stattfindet, wobei diese Schnittstellen als Queues dargestellt werden.

Im Compilerentwurf wurde diskutiert, wie beim Übersetzen einer Demaq-Anwendung mit den Modulen umgegangen wird. Es wurde entschieden, dass Anwendungen mit Modulen in dazu äquivalente Anwendungen ohne Module umgeschrieben werden. Daraus entsteht der Vorteil, dass die Laufzeitumgebung von Demaq nicht verändert werden muss. Für dieses Umschreiben von modularisierten Anwendungen wurde die Rewrite-Phase des Demaq-Compilers gewählt. Dadurch können die Module von den Optimierungen des Compilers profitieren, zudem muss die bestehende Code-Erzeugungs-Phase nicht verändert werden.

Der erstellte Entwurf diente als Grundlage für die Implementierung des Modulkonzepts, welche in Kapitel 5 beschrieben wurde. Es wurden die nötigen Änderungen im Lexer und Parser angesprochen und anschließend das Umschreiben von Anwendungen mit Modulen in der Rewrite-Phase ausführlich beschrieben.

In der Evaluation in Kapitel 6 wurden die in Kapitel 3 vorgestellten Anwendungsfälle mit dem Modulkonzept umgesetzt. Daran konnte gezeigt werden, dass alle funktionalen Anforderungen umgesetzt werden konnten. Die nichtfunktionalen Anforderungen konnten ebenfalls erfüllt werden. Die einzige Ausnahme ist hierbei die Erhaltung der Laufzeit, welche durch den gewählten nachrichtenbasierten Datenaustausch nicht eingehalten werden konnte. In typischen Anwendungen entsteht dadurch aber lediglich ein geringer Overhead.

7.2 Ausblick

Durch das Modulkonzept wird die Modularisierung von Demaq-Anwendungen ermöglicht. Dieses Unterkapitel bietet einen Ausblick, wie dieses Modulkonzept in der Zukunft sinnvoll erweitert werden kann.

Binärmodule Durch eine Unterstützung von Binärmodulen könnte es ermöglicht werden, in anderen Sprachen geschriebene Module in eine Demaq-

Anwendung einzubinden. Damit wäre es möglich, bislang in Demaq nicht vorhandene oder nur schwer umsetzbare Funktionalität zu realisieren. Ein Beispiel dafür könnte ein Modul sein, welches eine Zufallszahl zurückliefert oder ein Kryptographie-Modul, mit welchem Nachrichten ver- und entschlüsselt werden können.

Modul-Bibliothek Wie die *Standard Template Library* (*STL* [20]) in C++ oder die Klassenbibliothek (*Java Class Library* [7]) in Java könnten von Demaq verschiedene Module in einer internen Modul-Bibliothek zum Einbinden bereitgestellt werden. Somit wäre es möglich bei der Entwicklung von Anwendungen auf eine Sammlung von Modulen zurückzugreifen, welche in Demaq-Anwendungen häufig benötigte Funktionalität bereitstellen. Beispiele für solche Funktionalitäten sind das Versenden und Empfangen von SOAP-Nachrichten oder der Versand von E-Mails.

Globale Parameter Eine weitere Idee ist, dass die Parameter eines Moduls bereits beim Import mit Werten versehen werden können. Diese Werte sind somit für alle erzeugten Instanzen des Moduls identisch und müssen bei der Instanziierung nicht mehr übergeben werden. Es sollte aber dennoch möglich sein, bei Bedarf den beim Import gesetzten Wert bei der Instanziierung zu überschreiben. Ein Beispiel hierfür könnte der Absender für einen E-Mail-Versand sein, welcher in allen Instanzen eines Moduls gleich sein soll.

Literaturverzeichnis

1 Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers - Principles, Techniques and Tools*. Addison-Wesley, 1986. 2.4, 2.1, 2.2

2 Böhm, A. *Demaq System Documentation and User Manual*, März 2009. 2.5, 2.3, 2.4, 6.2

3 Booch, G., and Bryan, D. *Software engineering with Ada (3rd ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1993. 2.6

4 Bray, T., Tobin, R., Hollander, D., and Layman, A. Namespaces in XML 1.0 (second edition). W3C recommendation, W3C, August 2006. 2.1

5 Chamberlin, D., Robie, J., Melton, J., Florescu, D., and Siméon, J. XQuery update facility 1.0. Candidate recommendation, W3C, März 2008. 2.5.2

6 Chamberlin, D. D., and Boyce, R. F. Sequel: A structured english query language. In *FIDET '74: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control* (New York, NY, USA, 1974), ACM Press, pp. 249–264. 2.5

7 Chan, P., Kramer, D., and Lee, R. *The Java Class Libraries: Supplement for the Java 2 Platform*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. 2.6, 7.2

8 Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. Web services description language (wsdl) 1.1. W3C recommendation, W3C, März 2001. 2.5.2

9 DeRose, S., and Clark, J. XML path language (XPath) version 1.0. W3C recommendation, W3C, November 1999. 2.2

10 Fallside, D. C., and Walmsley, P. XML schema part 0: Primer second edition. W3C recommendation, W3C, Oktober 2004. 2.5.2

11 Fiebig, T., Kanne, C.-C., and Moerkotte, G. Natix - ein natives xml-dbms. *Datenbank Spektrum*, 1 (2001), 5–13. 2.5.5

12 Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns*. Addison-Wesley Professional, Januar 1995. 5.3.3

13 Gosling, J., Joy, B., Steele, G., and Bracha, G. *The Java Language Specification, Third Edition*, 3 ed. Addison-Wesley Longman, Amsterdam, Juni 2005. 2.6, 4.1.4

14 Lafon, Y., and Mitra, N. SOAP version 1.2 part 0: Primer (second edition). Tech. rep., W3C, April 2007. 2.5.2

15 Paoli, J., Cowan, J., Bray, T., Yergeau, F., Maler, E., and Sperberg-McQueen, C. M. Extensible markup language (XML) 1.1 (second edition). W3C recommendation, W3C, August 2006. 2.1

16 Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15*, 12 (Dezember 1972), 1053–1058. 2.6

17 Raner, M. The mutator pattern. In *PLoP '06: Proceedings of the 2006 conference on Pattern languages of programs* (New York, NY, USA, 2006), ACM, pp. 1–6. 5.3.3

18 Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorenson, W. *Object-Oriented Modeling and Design*, 1 ed. Prentice Hall, Inc., October 1991. 2.6, 2.6

19 Siméon, J., Chamberlin, D., Florescu, D., Boag, S., Fernández, M. F., and Robie, J. XQuery 1.0: An XML query language. W3C recommendation, W3C, Januar 2007. 2.3, 2.6

20 Stepanov, A., and Lee, M. The standard template library. Tech. rep., WG21/N0482, ISO Programming Language C++ Project, 1994. 2.6, 7.2

21 Stroustrup, B. *The C++ Programming Language*, third ed. Addison-Wesley Professional, Februar 2000. 2.6

22 Szyperski, C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. 2.6

23 Walsh, N., Fernández, M., Malhotra, A., Nagy, M., and Marsh, J. XQuery 1.0 and XPath 2.0 data model (XDM). W3C recommendation, W3C, Januar 2007. 2.2

24 Wirth, N. The programming language pascal. *Acta Inf. 1* (1971), 35–63. 2.6

25 Wirth, N. *Programmieren in Modula-2. übers. aus dem Engl. von Guido Pfeiffer. (Programming in Modula-2)*. Springer-Verlag, Berlin, 1985. 2.6, 2.6

Ehrenwörtliche Erklärung

Ich versichere, dass ich die beiliegende Diplomarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Forst, den 30. April 2009

Andreas Krämer

A Anhang

Listing A.1: BNF der geänderten Befehle.

```
1 [203] DQLLibraryModule      ::= ModuleDecl DQLApplicationModule
2 [204] DQLApplicationModule ::= Prolog DQLModuleImport* (DQLDefinition | DQLRule
      )*
3 [207] QDLExpression         ::= (QueueDefinition | PropertyDefinition |
      SlicingDefinition | BindingDefinition)
4 [208] QueueDefinition       ::= "create" "queue" QueueBody
```

Listing A.2: BNF der hinzugefügten Befehle.

```
1 [301] DQLModuleImport          ::= "import" "dqlModule" DQLModuleName ("at"
      URILiteral)? ";"
2 [302] BindingDefinition        ::= "create" "binding" QName "for" DQLModuleName
      "assign" "input" ModuleQueueAssignment ("assign" "output"
      ModuleQueueAssignment)? ModuleParamsAssign*
3 [303] ModuleQueueAssignment    ::= QueueName ("as" QueueName)? ("," QueueName
      ("as" QueueName)?)*
4 [304] ModuleParamsAssignment ::= "with" ParameterName "value" (Literal | VarRef)
5 [305] ModuleDecl               ::= ModuleInputQueueDecl+ ModuleOutputQueueDecl
      *
6 [306] ModuleInputQueueDecl     ::= "declare" "input" "queue" QueueBody ";"
7 [307] ModuleOutputQueueDecl    ::= "declare" "output" "queue" QueueBody ";"
8 [308] QueueBody                ::= QueueName QueueKind QueueMode QueuePriority?
      ErrorHandler? QueueValidation?
9 [311] DQLModuleName            ::= QName
10 [312] ParameterName           ::= QName
```