UNIVERSITÄT LEIPZIG

Fakultät für Mathematik und Informatik

Masterarbeit

Entwicklung von zSeries-Anwendungen mit Hilfe der Enterprise Generation Language und dem Websphere Developer for zSeries

Studiengang: Informatik

Abgabetermin: Prüfer: Prof. Dr. Wilhelm G. Spruth

Stefan Erras

96 37 47 3

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbst verfasst und nur die angegebenen Quellen benutzt habe. Ferner habe ich die Arbeit noch keinem anderen Prüfungsamt vorgelegt.

Leipzig, den 29.04.2007

Danksagung

An dieser Stelle danke ich allen, die durch ihre Unterstützung zum Gelingen dieser Masterarbeit beigetragen haben.

Mein besonderer Dank gebührt meinem Betreuer Herrn Prof. Dr. Wilhelm G. Spruth, der mir diese Arbeit erst ermöglicht hat und sie stets mit freundlicher Unterstützung begleitete.

Bedanken möchte ich mich auch beim Leiter des Prüfungsamts der Fakultät für Mathematik und Informatik, Herrn Werner Reutter, der immer ein offenes Ohr hatte und alle Probleme, die ich an ihn herangetragen habe, ausnahmslos beseitigt hat.

Inhaltsangabe

Thema der vorliegenden Arbeit ist die Softwareentwicklung mit Hilfe der Enterprise Generation Language (EGL) und dem Websphere Developer for zSeries. Hierzu werden zunächst die wichtigsten Eigenschaften der zSeries-Plattform und des Websphere Developers erläutert. Es folgt eine Einführung in die Entwicklung mit Enterprise Generation Language und eine Diskussion der Vor- und Nachteile, die ihr Einsatz mit sich bringt. Die wichtigsten Aspekte sind hierbei die indirekte Kompilierung der EGL, die Auswirkungen auf den Softwareentwicklungsprozess, das verwendete Characterset und die Möglichkeiten zur Fehlerbehandlung. Außerdem wird in Form eines Lehrgangs gezeigt, wie sich die Enterprise Generation Language zur effizienten Entwicklung von Businesslogik einsetzen lässt. Es wird eine datenbankgestützte Anwendung entwickelt, die einen Teil der Anforderungen des Sabanes & Oxley Acts (US-Gesetz zur verbindlichen Regelung der Unternehmensberichterstattung) umsetzt. Ziel dieser Anwendung ist es, die Abdeckung von Prozessrisiken durch so genannte Kontrollaktivitäten zu ermitteln. Die Anwendung wird dabei zweifach implementiert: Version 1 zeigt, wie Datenbankinhalte direkt mit der EGL abgerufen und verarbeitet werden können. In Version 2 wird aufgezeigt, welche Möglichkeiten zur Einbindung von Java-Komponenten bestehen und wie sich diese anwenden lassen. Abschließend werden Fehler und Probleme, die bei der Entwicklung mit dem Websphere Developer auftreten können, aufgezeigt.

Die benötigte Software liegt der Arbeit in Form eines VM-Ware-Images bei und ist darin bereits vorinstalliert.

Inhaltsverzeichnis

Abkürzungsverzeichnis

4GL	4th Generation Language
AWT	Advanced Widget Toolkit
CF	Coupling Facility
CICS	Customer Information Control System
DL/1	Data Language/1
EGL	Enterprise Generation Language
GPL	General Public License
IBM	International Business Machines Corporation
ID	Identifikationsnummer
IDE	Integrated Development Environment
IMS	Information Management System
IT	Informationstechnologie
JDBC	Java Database Connectivity
JSF	Java Server Faces
PDF	Portable Document Format
PHP	PHP: Hypertext Preprocessor
PWC	Price Waterhouse Coopers
SOA	Serviceorientierte Architektur
SOX	Sabanes & Oxley Act
SQL	Structured Query Language
UTF-16	16-bit Unicode Transformation Format
UTF-8	8-bit Unicode Transformation Format
VSAM	Virtual Storage Access Method
W3C	World Wide Web Consortium
WDz	Websphere Developer for zSeries
XML	Extensible Markup Language
XSL-FO	Extensible Stylesheet Language–Formatting Objects

Abbildungsverzeichnis

Anlage

DVD mit einem komprimierten VM-Ware-Image (EGL-Tutorial.exe) mit folgendem Inhalt:

- Apache FOP 0.92 Beta
- Create-Skript für die Datenbank
- Foxit Reader 2.0
- IBM Rational Websphere Developer for zSeries 6.01
- IBM DB2 Express Edition 9.1.0.356
- Java-Komponente ReportingPersistence.jar
- Microsoft Windows XP Professional mit Service Pack 2 (Benutzer: UNILP, Passwort: unilp)
- XSLT-FO-Skript für die Transformation

Außerdem enthält die DVD im Ordner „Literatur“ alle verwendeten Internetquellen.

1 Einleitung

Seit Jahrzehnten werden viele unternehmenskritische Anwendungen auf Mainframe-Servern ausgeführt. Diese Rechner sind für ihre hohe Zuverlässigkeit, Sicherheit, Ausführungsgeschwindigkeit und Abwärtskompatibilität bekannt. 90 % der 2000 größten Unternehmen weltweit setzen zSeries-Rechner der International Business Machines Corporation (IBM) oder kompatible Geräte ein [HER1], um von diesen Eigenschaften zu profitieren. Trotzdem wird oft die (nicht rational begründbare) Behauptung aufgestellt, der Mainframe sei vom Aussterben bedroht. Tatsächlich haben zSeries-Rechner und das zugehörige Betriebssystem z/OS auf vielen Gebieten eine Technologieführerschaft inne. Beispiele hierfür sind der Workloadmanager, Virtualisierung, der Parallel-Sysplex oder der Transaktionsmanager Customer Information Control System (CICS) [ZDNET]. Auch stieg der Umsatz im Bereich der zSeries-Rechner im vierten Quartal 2006 um 5 % im Vergleich zum Vorjahr [IBM4].

Aufgrund der Vorzüge der zSeries-Plattform werden in vielen Unternehmen Programme verwendet, die teilweise schon seit Jahrzehnten im Einsatz sind. Andererseits werden immer noch neue Programme für diese Plattform entwickelt. Um die Softwareentwicklung für die zSeries zu erleichtern, stellt die Firma IBM die Entwicklungsumgebung Websphere Developer for zSeries (WDz) zur Verfügung. Diese Umgebung ist für die Entwicklung von zSeries-Anwendungen optimiert und stellt eine Reihe spezieller, für zSeries abgestimmter Funktionen bereit (siehe [HER2]).

Im Rahmen dieser Arbeit sollen zunächst die wesentlichen Eigenschaften der zSeries dargestellt werden. Anschließend wird ein wichtiger Bestandteil des WDz in Form eines Lehrgangs beschrieben: die Enterprise Generation Language. Es soll hierbei aufgezeigt werden, wie sich Anwendungen der zSeries mit Hilfe der Enterprise Generation Language (EGL) sinnvoll und zukunftssicher erweitern oder neu entwickeln lassen. Außerdem werden die Eigenschaften der EGL und die Vorzüge, die ihr Einsatz im Softwareentwicklungsprozess mit sich bringt, beschrieben. Dieser Lehrgang wird Teil des Ausbildungsprogramms von Prof. Dr. Wilhelm Spruth, das mittlerweile neben der Universität Leipzig an elf weiteren Hochschulen im In- und Ausland angeboten wird.

2 Eigenschaften der zSeries

Dieses Kapitel gibt einen Überblick der wichtigsten Eigenschaften der zSeries (Hard- und Software).

2.1 Abwärtskompatibilität

Von Prof. Dr. Kurt Hoffmann wird Software (frei nach Phil G. Armour) mit aufgeschriebenem Wissen verglichen [HOF]:

> *Software ist eigentlich ein Medium, um bestimmtes Wissen zu speichern, ähnlich wie auch ein Buch ein Medium zur Wissensspeicherung ist.*

In einem Unternehmen handelt es sich dabei um das Wissen, wie ein bestimmtes Problem zu lösen ist bzw. wie Daten zu verarbeiten sind. Ist dieses Wissen einmal in Form von Software abgelegt, sollte diese Software – schon wegen der Wirtschaftlichkeit – möglichst lang einsetzbar sein.

Um dieses Ziel zu erreichen, wurde bei der Weiterentwicklung der zSeries-Rechner immer auf die Kompatibilität mit den vorherigen Modellen geachtet. Es ist so möglich, bestehenden Code, der in den letzten Jahrzehnten geschrieben wurde, ohne Änderungen oder Neukompilierung auf ein neues System zu überspielen und auszuführen.

Forschungsarbeiten im Bereich der Compiler- und Prozessorentwicklung haben allerdings gezeigt, dass es teilweise nicht sinnvoll ist, diese Abwärtskompatibilität auf Microcodeebene zu erhalten. Bei der zSeries werden deshalb selten genutzte Maschinenbefehle, die nur zum Erhalt der Kompatibilität des Binärcodes dienen, durch den so genannten Licenced Internal Code nachgebildet [HER5]. Dies ist sinnvoll, da gezeigt werden konnte, dass ein reduzierter Befehlssatz mit einfachen Einzeloperationen, im Vergleich zu einem komplexen Befehlssatz mit vielen Spezialoperationen, im Endeffekt zu höherer Performance führt [FRÖ]. Um dies zu erreichen, müssen allerdings hochwertige Compiler

eingesetzt werden, die entsprechende Optimierungen vornehmen. Neuere Anwendungen profitieren von dieser Optimierungsmöglichkeit, während ältere Anwendungen dennoch ausführbar bleiben.

2.2 Virtualisierung

Unter Virtualisierung versteht man die Fähigkeit, auf einem Rechner mehrere Betriebssysteme gleichzeitig zu betreiben. Die jeweiligen Gastsysteme laufen dabei völlig getrennt voneinander ab.

Auf zSeries-Systemen gibt es grundsätzlich zwei Arten der Virtualisierung. Einerseits kann das System in so genannte Logical Partitions unterteilt werden, andererseits steht mit z/VM zusätzlich ein Virtualisierungsserver bereit [HER4].

Das Aufsetzen mehrerer Betriebsysteminstanzen kann aus verschiedenen Gründen sinnvoll sein:

- **Verwendung unterschiedlicher Betriebsysteme**: In manchen Umgebungen müssen wegen der benötigten Software unterschiedliche Betriebssysteme eingesetzt werden, im Fall der zSeries z. B. z/OS und zLinux. Durch die hohe Performance heutiger (zSeries-) Rechner werden durch Virtualisierung Kosten, Energie und Platz gespart, wenn mehrere Betriebsysteme gleichzeitig auf einem Server zur Ausführung kommen. Auch wird die Kommunikationsgeschwindigkeit im Vergleich zu getrennten Servern erhöht, weil die interne Kommunikation zwischen virtualisierten Instanzen in der Regel performanter durchgeführt werden kann als die Kommunikation über ein Netzwerk. Dies ist z. B. in einer serviceorientierten Architektur (SOA) sinnvoll, da hier ein hoher Kommunikationsbedarf zwischen den einzelnen Services besteht.

- **Abschottung von Anwendungen:** Ebenso sinnvoll ist die Abschottung der Anwendungen voneinander. In einem Unternehmen könnte ein System z. B. in drei unterschiedliche Partitionen zergliedert werden: Entwicklungs-, Test- und Produktivsystem. Zum Testen und Entwickeln kann so die exakt gleiche Konfiguration

wie im Produktivsystem bereitgestellt werden, ohne zusätzliche Hardware anschaffen zu müssen. Probleme, die bei der Migration vom Entwicklungssystem auf das Produktivsystem durch unterschiedliche Konfigurationen entstehen können, werden so ausgeschlossen. Zusätzlich ergibt sich die Möglichkeit, dem Produktivsystem in Stoßzeiten den vollen Leistungsumfang der Maschine zur Verfügung zu stellen, da die Ressourcen dynamisch zwischen den Teilsystemen aufgeteilt werden können. Würde auf getrennten Systemen entwickelt und getestet, wäre eine solche Lastverteilung mit höherem Aufwand verbunden (das ist auf zSeries-Systemen dank der Coupling Facility aber trotzdem möglich).

Auch in Multi-Tier-Konfigurationen kann sich die Abschottung als Vorteil erweisen. So können beispielsweise ein Webserver, ein Transaktionsmonitor und ein Datenbanksystem getrennt voneinander ablaufen, um die Sicherheit zu erhöhen. Da die Anzahl von Angriffen aus dem Internet stetig steigt, ist es nicht auszuschließen, dass ein Hacker über den Webserver Zugriff auf das darunterliegende System erlangt. Würden nun Webserver und Datenbank auf der gleichen Betriebssysteminstanz ablaufen, könnte ein Hacker mit Zugriff auf den Webserver im schlimmsten Fall gleichzeitig auch Zugang zur Datenbank erhalten. Bei getrennter Konfiguration wird dies erschwert, da die Datenbank aus Sicht des Hackers auf einem getrennten System ausgeführt wird. Zur Erlangung des Datenbankzugriffs müsste er in diesem Fall erst eine weitere Hürde überwinden.

2.3 Hohe Ausfallsicherheit

Aufgrund ihrer Architektur gelten zSeries-Rechner als sehr ausfallsicher. IBM nennt dafür den Wert 99,999 % für eine einzelne Maschine. Eine möglichst hohe Verfügbarkeit der Computersysteme ist besonders bei unternehmenskritischen Anwendungen von hoher Bedeutung. Laut [CI] belaufen sich die Kosten für eine Minute Downtime im unternehmenskritischen Bereich auf bis zu 10 000 $.

2.4 Workloadmanager

Ein wichtiger Bestandteil des z/OS-Betriebssystems ist der Workloadmanager. Dieser ermöglicht die dynamische Last- und Ressourcenverteilung zwischen einzelnen Instanzen des Betriebssystems. Hierbei können verschiedene Arten von Last definiert und bestimmte Verarbeitungsprogramme priorisiert werden. Der Workloadmanager kümmert sich automatisch um die Optimierung der Lastverteilung im System. Dabei kommen verschiedene Lösungsansätze zum Einsatz [HER1]. So wird im Forschungslabor von IBM in Böblingen an einem Workloadmanager gearbeitet, der auf Basis eines neuronalen Netzwerkes funktioniert.

2.5 Skalierbarkeit und Clustering mittels Coupling Facility (nach [HER3])

Das Clustering mehrerer zSeries-Rechner erfolgt über die so genannte Coupling Facility (CF). Es handelt sich dabei um eine separate Recheneinheit, die die Kontrolle des Clusters übernimmt und für die Synchronisation der einzelnen Knoten sorgt. Weiterhin werden Aufgaben wie Locking oder Caching der Daten von der CF übernommen bzw. von ihr an Teilknoten delegiert.

Die einzelnen Knoten sind dabei untereinander über ein Netzwerk verbunden. Weiterhin ist jeder Knoten zusätzlich an die CF angebunden. Jeder zSeries-Rechner verfügt über eine entsprechende Schnittstelle zur Anbindung der CF (Coupling Facility Support).

Die Auslagerung der Verwaltungsaufgaben im Cluster auf die CF führt dazu, dass die Leistung eines zSeries-Clusters fast linear mit der Anzahl der eingebunden Knoten skaliert (siehe Abbildung 1).

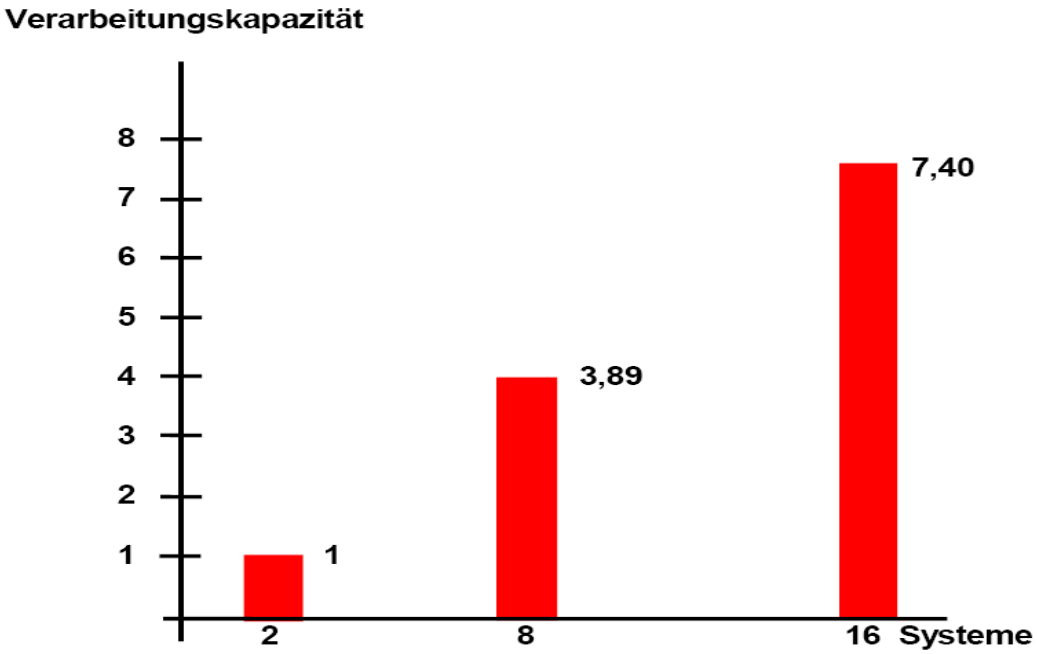

Abbildung 1: Leistungsverhalten des Parallel Sysplex [HER3]

3 Verwandtschaft des WDz zu Eclipse

Der WDz setzt auf die beliebte Anwendungsplattform Eclipse auf, bei der es sich um ein modular aufgebautes Anwendungsframework handelt, welches in Java implementiert wurde und unter einer Open-Source-Lizenz für jeden frei zugänglich ist. Bekanntester Vertreter der Eclipse-Plattform ist die gleichnamige Java-Entwicklungsumgebung, die laut [CAS] von über 50 % der Java-Programmierer eingesetzt wird. Entwickler, die bereits Erfahrungen mit der Eclipse-IDE (Integrated Development Environment) haben, finden sich deshalb schnell im WDz zurecht, da er der Eclipse-IDE sehr ähnlich ist.

Zusätzlich finden sich eine Reihe anderer Anwendungen, die ebenso auf das Eclipse-Projekt aufsetzen (z. B. JBuilder von Borland [HEI] oder der Workplace Managed Client von IBM [SEW]).

Wissen über diese Plattform kann somit sinnvoll auch in anderen Projekten angewendet werden und ist nicht an einen Hersteller gebunden.

An dieser Stelle sei noch angemerkt, dass die Oberfläche des WDz und der Eclipse-Plattform nicht mit Hilfe der javaeigenen Oberflächen-Toolkits Advanced Widget Toolkit (AWT) oder Swing realisiert wurden. Diesen wird häufig nachgesagt, nicht besonders performant zu sein (Swing) bzw. eine ungenügende Anzahl an Kontrollelementen bereitzustellen (AWT). Stattdessen wurde ein neues Oberflächen-Toolkit entwickelt, welches im Gegensatz zu Swing die nativen Controls des jeweiligen Betriebssystems verwendet. Hierdurch soll eine höhere Performance und eine bessere Integration in die Anwendungslandschaft erreicht werden, da sich das Look-And-Feel einer Eclipse-Anwendung nicht von dem einer herkömmlichen Anwendung unterscheidet [SWT1, SWT2].

4 Entwicklung mit Hilfe der Enterprise Generation Language

Ziel dieses Kapitels ist es, dem Leser eine Grundlage für die Entwicklung von zukunftssicheren Anwendungen mit Hilfe der EGL zu geben. Hierbei sollen die Vor- und Nachteile dieser Sprache diskutiert werden. In den Kapiteln 5 bis 11 folgt dann eine Einführung in die praktische Anwendung der EGL.

4.1 Einführung

Bei der EGL handelt es sich um eine so genannte 4th Generation Language (4GL), also eine Sprache der 4. Generation. Im Gegensatz zu Sprachen der dritten Generation (wie C++, Java oder Cobol) beschreibt der Programmierer in einer solchen Sprache nicht „WIE“ ein bestimmtes Problem zu lösen ist, sondern „WAS“ der Rechner tun soll, um ein Problem zu lösen [KOL]. Ziel ist es, Programme mit möglichst wenig Code auf leicht verständliche Weise zu erstellen, wobei die konkrete technische Umsetzung in den Hintergrund tritt. Die Entwicklung wird so vereinfacht und beschleunigt und geht mit einer gleichzeitigen Verbesserung der Lesbarkeit des Codes einher, was zusätzlich die Wartbarkeit erleichtert.

Um dies zu erreichen, sind Sprachen der 4. Generation häufig auf ein bestimmtes Themengebiet zugeschnitten. Sie sollten deshalb nur zur Lösung von Problemen aus dem jeweiligen Themengebiet verwendet werden, da sich Probleme anderer Gebiete mit einer spezialisierten Sprache häufig nur schwer oder gar nicht lösen lassen.

Beispiele für solche Sprachen sind:

- **Structured Query Language (SQL):** Abfragesprache für relationale Datenbanken.
- **MATLAB:** Bietet eine Sprache zur Implementierung von mathematischen Algorithmen.
- **Macrosprachen:** Zur Automatisierung von Abläufen in Desktopanwendungen.

4.2 EGL im Softwareentwicklungsprozess

Die EGL eignet sich besonders zur gezielten Verarbeitung von Datenbankinhalten und kann z. B. in Kombination mit Java Server Faces (JSF) zum dynamischen Befüllen von Internetseiten verwendet werden. Zur Erstellung von Webauftritten mit der EGL finden sich deshalb auf der Internetseite von IBM einige englischsprachige Beispiele.

Da für die Entwicklung von dynamischen Internetseiten eine Reihe von Frameworks (z. B. Struts oder JSF) und Sprachen (z. B. Visual Basic Script, PHP[1], Java) zur Verfügung stehen, die mittlerweile einen hohen Verbreitungsgrad erreicht haben, sollte das Hauptaugenmerk bei der EGL-Entwicklung nach Ansicht des Autors auf die Implementierung von Businesslogik gelegt werden.

In vielen Bereichen der Wirtschaft sind große Datenmengen in relationalen Datenbanken abgelegt, die nach komplizierten Regeln verarbeitet werden müssen. Werden diese Verarbeitungsregeln mit einer Sprache der dritten Generation gelöst, enthält der Code, bezogen auf die reinen Anwendungslogik, häufig einen großen Anteil an technischem „Overhead“, d. h. Codezeilen, die nicht direkt mit der eigentlichen Logik in Verbindung stehen, sondern die technischen Rahmenbedingungen realisieren. Als Beispiel sei an dieser Stelle etwa das Aufbauen einer Datenbankverbindung mittels Java Database Connectivity (JDBC) genannt. Bei der Programmierung mit EGL wird bewusst versucht, diesen technischen Anteil

[1] Rekursives Backronym für „**P**HP: **H**ypertext **P**reprocessor“, ursprünglich „**P**ersonal **H**ome **P**age Tools“.

im Code auf ein Minimum zu reduzieren und so die reine Anwendungslogik in den Vordergrund zu rücken.

Dies bietet den Vorteil, dass die Anzahl potentieller Fehlerquellen und Sicherheitslücken, die bei der Implementierung des technischen Codes entstehen können, minimiert wird.

Außerdem werden auch technisch weniger versierte Entwickler in die Lage versetzt, Daten automatisiert verarbeiten zu lassen oder zumindest den Code anderer Entwickler zu verstehen. In der Zukunft dürfte sich dies als erheblicher Vorteil in der Softwareentwicklung erweisen.

Im Moment gibt es eine Unzahl von Softwareentwicklungsmodellen, die das Ziel haben, die Softwareentwicklung effizienter, planbarer und zielgerichteter zu gestalten. Trotzdem ist die Anzahl der Softwareprojekte, die mit einem Projektabbruch enden oder nur mit stark erhöhtem Aufwand zum Ziel geführt werden können, viel zu hoch.

Prof. Dr. Gottfried Koch von der Universität Leipzig hat hierzu folgende Zahlen verschiedener Studien zusammengetragen [KOC]:

- *Gemäss einer internationalen Umfrage von PWC (PriceWaterhouse-Coopers) sind 65% aller Organisations- und Informatikprojekte nicht erfolgreich.*

- *Gemäss Gartner Group scheitern 40 Prozent aller IT-Projekte.*

- *Im Bereich Electronic Business ist die Ernüchterung noch radikaler: 75 Prozent der Projekte erreichen nicht das geplante Ziel (Mummert +Partner).*

- *Eine Untersuchung von der "Standish Group" zeigt bei einer Befragung von amerikanischen IT-Managern, dass nur 1/4 aller IT Projekte erfolgreich abgeschlossen werden:*
  - *erfolgreiche Projekte 26 %*
  - *abgebrochene Projekte 28 %*
  - *nicht erfolgreiche Projekte 46 %*

Als Hauptgrund für dieses Problem werden häufig die mangelnde Planung und Spezifikation angeführt. Prof. Dr. Koch nennt hierzu weiterhin folgende Zahlen [KOC]:

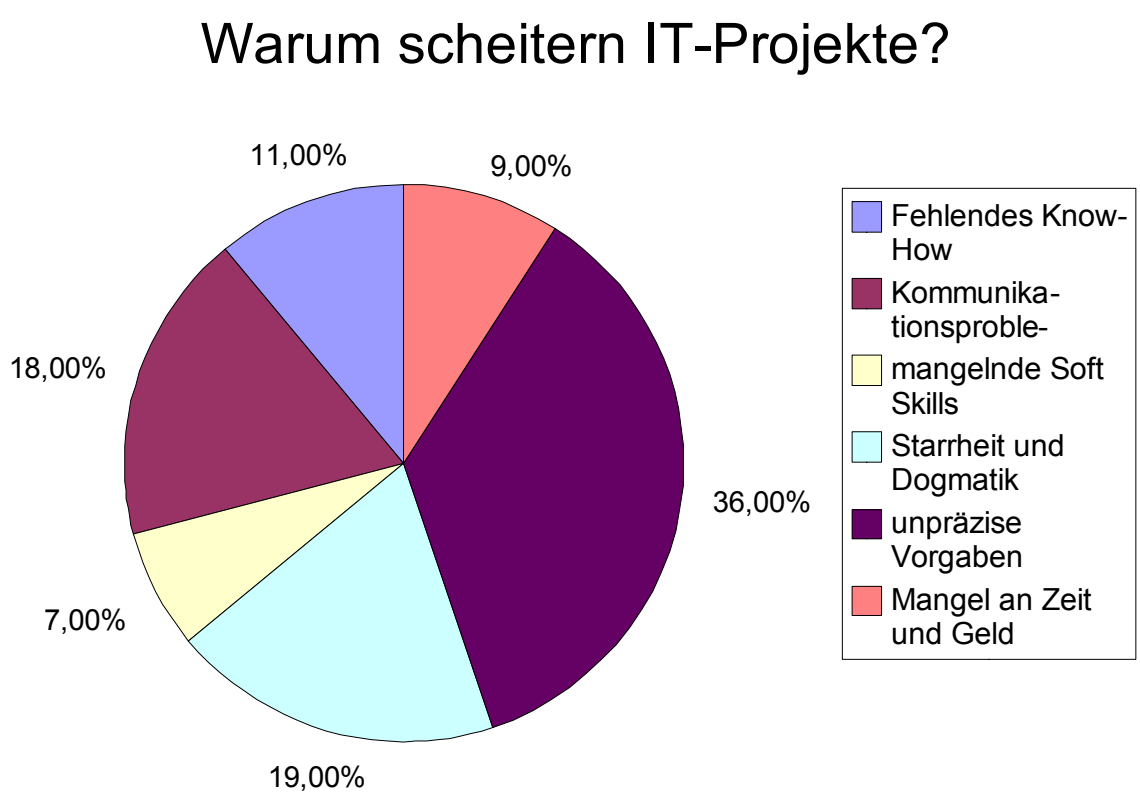

Abbildung 2: Umfrageergebnis: Warum scheitern IT-Projekte?

In Anbetracht des obigen Diagramms wird klar, dass die exakte fachliche Beschreibung einer Software durch den Auftraggeber die wichtigste Grundlage für die Entwicklung eines

Programms ist. Nur damit kann der Auftraggeber die komplexen fachlichen Zusammenhänge an das Entwicklungsteam weitergeben. Dies gestaltet sich umso schwieriger, je weniger fachliches Wissen im Entwicklerteam bzw. je weniger Wissen über Informationsverarbeitung beim Auftraggeber vorhanden ist.

Entwicklungsmodelle wie „Extreme Programming“ versuchen, die Trennung zwischen technischem Fachwissen[2] einerseits und fachlichem Wissen[3] andererseits durch den Einsatz eines so genannten On Sight Customers zu überbrücken. Dabei wird vom Auftraggeber ein Fachspezialist für die Beantwortung von Fragen und die Bewertung der Softwarelösung während des gesamten Projekts beim Entwicklungsteam abgestellt.

Doch auch bei fachlicher Unterstützung des Entwicklerteams während der gesamten Projektlaufzeit hat der Auftraggeber lediglich die Möglichkeit, Unklarheiten zu klären, die dem Entwicklungsteam auffallen bzw. die vom On Sight Customer entdeckt werden. Fehlerhafte Spezifikationsteile oder Fehlinterpretationen durch die Entwickler bleiben weiterhin so lange unentdeckt, bis ein Mitarbeiter des Auftraggebers in der Lage ist, diesen Teil der Software zu testen. Das kann bei komplexen Zusammenhängen oft erst sehr spät in der Projektlaufzeit der Fall sein, weil Abhängigkeiten zu anderen Modulen bestehen oder entsprechende Oberflächen erstellt werden müssen. Zusätzlich müssen auch passende Testdaten zur Verfügung stehen.

Außerdem kann auch eine umfangreiche Testphase evtl. nicht jedes Problem aufdecken. Die Testszenarien werden teilweise so kompliziert, dass nicht jeder Fall in angemessener Zeit simuliert werden kann.

Diese Problematik wird durch die Verwendung der EGL zur Implementierung der Businesslogik verkleinert. Fachspezialisten des Auftraggebers, die in dieser (leicht erlernbaren) Sprache geschult sind, werden in die Lage versetzt, eine aktive Rolle bei der Softwareentwicklung zu übernehmen. Dies kann einerseits durch Entwicklung von Codeteilen oder andererseits über die Durchführung von Codereviews geschehen, bei denen der Code be-

[2] Unter technischem Fachwissen wird in diesem Zusammenhang das Wissen über Informationsverarbeitung und Softwareentwicklung verstanden.

[3] Wissen über die Aufgabenstellung.

gutachtet und die Logik direkt nachvollzogen wird. Der Auftraggeber kann so nicht nur das Ergebnis der Entwicklungsarbeiten, sondern auch deren Umsetzung überprüfen. Dies führt zur besseren und frühzeitigeren Aufdeckung von Missverständnissen und Problemen. Weiterhin wird sichergestellt, dass der Code auch für Dritte verständlich ist und somit leicht geändert oder erweitert werden kann.

Eine andere Möglichkeit zur Verbesserung des Entwicklungsprozesses mit der EGL ist die Implementierung von Modultests[4] bzw. Testklassen und -funktionen durch Fachspezialisten. Das Fachwissen kann dann direkt zur Implementierung der Testfälle genutzt werden, was sich positiv auf die Anzahl der entdeckten Fehler auswirken dürfte. Die eigentliche Implementierung der Anwendung wird aber weiterhin von Informatikern auf hohem technischen Niveau durchgeführt, während der Testcode, bei dem es häufig weniger auf Sicherheit, Performance und Qualität/Strukturierung ankommt, von den Fachspezialisten programmiert werden kann. Die Fachspezialisten hinterlegen ihr fachliches Wissen in diesem Fall also in den Prüfungsroutinen für die Anwendung.

4.3 Nachteile der Verwendung einer 4GL, wie EGL oder SQL

Da in einer Sprache der vierten Generation meist beschrieben wird, was, aber nicht wie etwas zu tun ist, hat der Entwickler weniger Einfluss auf die genaue Umsetzung seiner Anweisungen. Er ist zwar einerseits von technischen Aspekten entlastet, muss sich dafür aber auf die Sprache und ihre Umsetzung verlassen. Eine händische Optimierung der Ausführung ist dann häufig nur mit genauem Wissen der internen Abläufe möglich. Wie komplex solche Abläufe sein können, sei an einem Beispiel der Sprache SQL gezeigt:

[4] Hier wird zu jedem Modul (z. B. Klasse oder Funktionssammlung) Code geschrieben, der das Verhalten des Moduls überprüft.

Eine einfache SQL-Abfrage mit mehreren Left-Joins und einer statischen Where-Bedingung:

```
SELECT * FROM TUTORIAL.PROZESSGRUPPE AS PROZESSGRUPPE
   LEFT JOIN TUTORIAL.PROZESSGRUPPE_RISIKO AS PROZESSGRUPPE_RISIKO
             on PROZESSGRUPPE.ID = PROZESSGRUPPE_RISIKO.PG_ID
   LEFT JOIN TUTORIAL.RISIKO AS RISIKO
             on RISIKO.ID = PROZESSGRUPPE_RISIKO.R_ID
   LEFT JOIN TUTORIAL.PROZESS AS PROZESS
             on PROZESS.PG_ID = PROZESSGRUPPE.ID
   LEFT JOIN TUTORIAL.KONTROLLAKTIVITAET_PROZESS AS
                   KONTROLLAKTIVITAET_PROZESS
             on KONTROLLAKTIVITAET_PROZESS.P_ID = PROZESS.ID
             and KONTROLLAKTIVITAET_PROZESS.PG_ID =
             PROZESSGRUPPE.ID
    LEFT JOIN TUTORIAL.KONTROLLAKTIVITAET AS KONTROLLAKTIVITAET
             on KONTROLLAKTIVITAET.ID =
                   KONTROLLAKTIVITAET_PROZESS.KA_ID
   WHERE PROZESS.ID = 1
```

Von der DB2-Datenbank generierter SQL-Code nach einer Optimierung obiger Abfrage :

```
SELECT Q11.$C9 AS "ID", Q11.$C7 AS "BESCHREIBUNG",
   Q11.$C6 AS "NAME", Q11.$C11 AS "PG_ID", Q11.$C10
   AS "R_ID", Q11.$C13 AS "ID", Q11.$C12 AS "BESCHREIBUNG",
   Q11.$C9 AS "PG_ID", Q11.$C8 AS "NAME", 1 AS "ID",
   Q11.$C0 AS "KA_ID", Q11.$C2 AS "P_ID", Q11.$C1 AS
   "PG_ID", Q11.$C5 AS "ID", Q11.$C4 AS "BESCHREIBUNG",
   Q11.$C3 AS "NAME"
FROM
(SELECT Q9.$C2, Q9.$C3, Q9.$C4, Q10.NAME, Q10.BESCHREIBUNG,
   Q10.ID, Q9.$C0, Q9.$C1, Q9.$C5, Q9.$C6, Q9.$C7,
   Q9.$C8, Q9.$C9, Q9.$C10
FROM
(SELECT Q7.$C4, Q7.$C5, Q8.KA_ID, Q8.PG_ID, Q8.P_ID,
   Q7.$C6, Q7.$C7, Q7.$C0, Q7.$C1, Q7.$C2, Q7.$C3
FROM
(SELECT Q5.$C0, Q5.$C1, Q6.BESCHREIBUNG, Q6.ID, Q5.$C2,
   Q5.$C3, Q5.$C4, Q5.$C5
FROM
(SELECT Q4.R_ID, Q4.PG_ID, Q3.$C1, Q3.$C0, Q3.$C2,
   Q3.$C3
FROM
(SELECT Q2.BESCHREIBUNG, Q2.NAME, Q1.NAME, Q1.PG_ID
FROM TUTORIAL.PROZESS AS Q1, TUTORIAL.PROZESSGRUPPE
```

```
   AS Q2
WHERE (Q1.ID = 1) AND (Q1.PG_ID = Q2.ID)) AS Q3 LEFT
   OUTER JOIN TUTORIAL.PROZESSGRUPPE_RISIKO AS Q4 ON
   (Q3.$C3 = Q4.PG_ID)) AS Q5 LEFT OUTER JOIN TUTORIAL.RISIKO
   AS Q6 ON (Q6.ID = Q5.$C0)) AS Q7 LEFT OUTER JOIN
   TUTORIAL.KONTROLLAKTIVITAET_PROZESS AS Q8 ON (Q8.P_ID
   = 1) AND (Q8.PG_ID = Q7.$C7)) AS Q9 LEFT OUTER JOIN
   TUTORIAL.KONTROLLAKTIVITAET AS Q10 ON (Q10.ID =
   Q9.$C2)) AS Q11
```

Bei Betrachtung des optimierten SQL-Codes fällt auf, dass er aus mehreren Einzelabfragen besteht, die durch eine deutlich umfangreichere Where-Bedingung miteinander kombiniert werden. Diese Optimierung wird von der Datenbank im Normalfall transparent im Hintergrund durchgeführt, was sich negativ auf die Optimierungsmöglichkeiten durch den Nutzer auswirkt. Weiterhin ist zu bedenken, dass es sich bei obigem Code lediglich um ein optimiertes SQL-Statement handelt. Wie die einzelnen Bestandteile des Statements letztendlich ausgeführt bzw. weiter optimiert werden, bleibt dem Nutzer wiederum verborgen.

Abbildung 3 zeigt den Ausführungsplan des SQL-Statements mit den entstehenden Kosten.

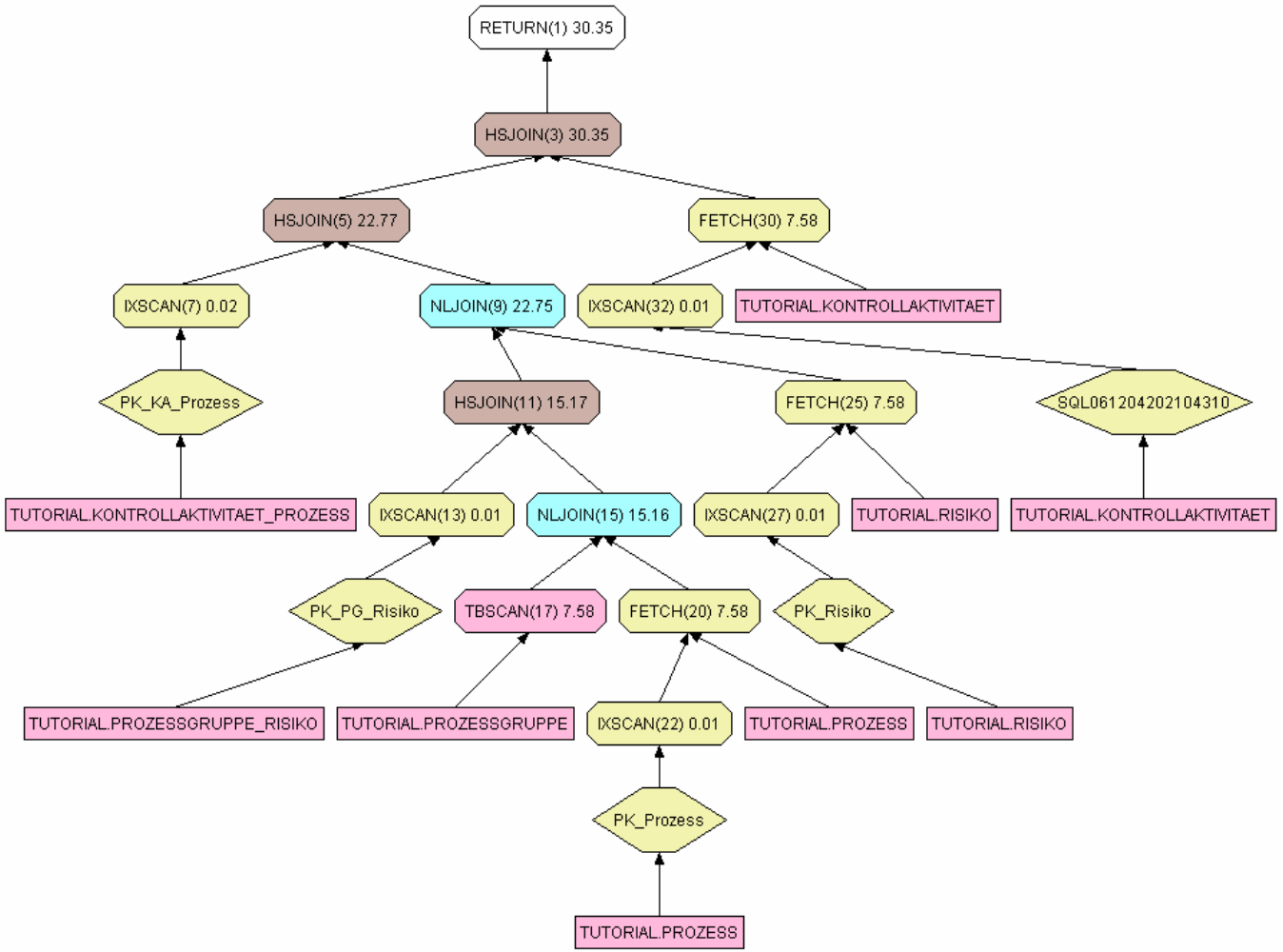

Abbildung 3: Ausführungsplan einer einfachen SQL-Abfrage auf einer DB2-Datenbank

Der Nachteil einer 4GL-Sprache, der durch die transparente Optimierung entsteht, wird bei der EGL teilweise dadurch kompensiert, dass der EGL-Code vor seiner endgültigen Kompilierung in eine andere Sprache übersetzt wird. Experten dieser Sprache können so die entsprechenden Abläufe auf einer technischen Ebene analysieren und ggf. Engpässe finden und beheben.

Wird zur EGL-Entwicklung der WDz verwendet, kann der generierte Code leider nur modifiziert werden, wenn anschließend keine Änderungen am ursprünglichen EGL-Code mehr gemacht werden, da der generierte Ziel-Code bei jedem Durchlauf des Generators überschrieben wird. Hier sollte nach einer Möglichkeit gesucht werden, Änderungen am generierten Code beizubehalten.

4.4 Die wichtigsten Eigenschaften der EGL

Im Folgenden sind die beiden wichtigsten Eigenschaften der EGL beschrieben:

- **EGL-Code wird in Java- oder Cobol-Source-Code übersetzt:** Im Gegensatz zu anderen Programmiersprachen wird EGL-Code nicht direkt kompiliert. Stattdessen wird der Code in eine andere Sprache umgewandelt. Im Moment stehen als Zielsprachen im WDz Java und Cobol zur Verfügung. Java steht einerseits für eine moderne Plattform mit hohem Verbreitungsgrad und die Unabhängigkeit von Hardware und verwendetem Betriebssystem. Dies wird durch die Bereitstellung einer unabhängigen Virtual Machine zur Ausführung von Java-Programmen erreicht. Cobol andererseits ist vor allem im Mainframebereich seit Jahren im Einsatz und wird dort wahrscheinlich in absehbarer Zeit nicht verschwinden. Mit dem Einsatz der EGL erreicht man somit einen hohen Grad an Investitionssicherheit: Eine bestehende Cobol-Umgebung kann um neue Bestandteile erweitert werden, ohne auf eine neue Plattform wie Java setzen zu müssen. Gleichzeitig kann die gleiche EGL-Implementierung aber auch in modernen Java-Code übersetzt werden. Sollte zu einem späteren Zeitpunkt ein Wechsel auf die Java-Plattform notwendig werden, können alle Komponenten, die mittels EGL entwickelt wurden, wieder verwendet werden.

Der Umweg über eine andere Sprache ist außerdem von Vorteil, falls die Weiterentwicklung der EGL eingestellt würde. Der generierte Java- oder Cobol-Quellcode ließe sich dann trotzdem noch in anderen Projekten nutzen und erweitern.

- **Umfangreiche Möglichkeiten zum Zugriff auf relationale Datenbanken:** Die EGL stellt eine Reihe von Funktionen zur Verfügung, um Daten in relationalen Datenbanken mit Hilfe so genannter SQLRecords zu verarbeiten oder abzurufen. Dies kann vielfach sogar ohne die Verwendung von SQL-Statements erfolgen. Nähere Informationen zum Zugriff auf relationale Datenbanken finden sich in Kapitel 9 dieser Arbeit.

4.5 Abgrenzung zu Skriptsprachen

Skriptsprachen beschleunigen die Softwareentwicklung, indem bewusst auf übliche Konstrukte herkömmlicher Programmiersprachen verzichtet wird. In fast allen Skriptsprachen kann ohne Typisierung oder Variablendeklaration programmiert werden. Außerdem werden Skriptsprachen nicht kompiliert und stehen so in der Regel immer als Source-Code zur Verfügung.

Die fehlende Variablendeklaration und Typisierung kann aber bei größeren Projekten zu Problemen führen. Durch das Fehlen von Variablendeklarationen werden Schreibfehler oft nicht entdeckt. Dies sei im Folgenden durch ein Beispiel in der Sprache PHP gezeigt:

```
for ($zaehler = 0; $zaehler < 5; $zahler++)
   echo $zaehler;
```

Obiger Code führt zu einer Endlosschleife, da die Variable $zaehler nie inkrementiert wird.

Zu Problemen bei der Lesbarkeit des Codes kann das Fehlen der Typisierung führen:

```
…
$obj = $obj2->getReference();
$obj->rotate();
…
```

Dem Leser des Codeausschnitts wird dabei nicht klar, um welche Art von Objekt es sich bei $obj nach der Zuweisung handelt. Vor allem bei komplexen Objekt-Hierarchien erweist sich dies als Nachteil, wenn der Code nicht genauestens kommentiert ist.

Weiterhin ist für die Ausführung einer Skriptsprache in der Regel ein Interpreter notwendig, der auf jedem Zielsystem installiert sein muss.

Bei der EGL kommen diese Nachteile nicht zum Tragen, weil dort alle Variablen deklariert und typisiert sind. Auch ist kein spezieller Interpreter notwendig, da EGL zunächst in eine andere Sprache umgewandelt und anschließen kompiliert wird.[5]

4.6 Abgrenzung zu Stored Procedures

Stored Procedures sind Prozedueren, die auf einem Datenbanksystem abgelegt werden und direkt dort zur Ausführung kommen. Zur Definition dieser Prozeduren bieten einige Datenbanksysteme spezielle Sprachen an, die besonders für Manipulation von Daten ausgelegt sind. Beispiele hierfür sind die proprietären Sprachen PL/SQL (Oracle-Datenbanksysteme) und Transact-SQL (Microsoft SQL-Server).

In Kapitel 9 dieser Arbeit wird gezeigt, dass auch die EGL eine Reihe von Möglichkeiten bietet, um Datenbankabfragen effizient zu formulieren. Im Vergleich zu Stored Procedures

[5] Falls Java-Code erzeugt wird, wird dieser beim Kompilieren in der Regel in Bytecode umgewandelt, für dessen Ausführung man eine Java Virtual Machine benötigt. Soll keine Virtuelle Maschine verwendet werden, muss der Java-Code zunächst mit Hilfe eines so genannten Ahead Of Time Compilers in nativen Maschinencode umgewandelt werden.

bietet sie jedoch den Vorteil, dass sie unabhängig vom jeweiligen Datenbanksystem arbeitet, während Sprachen zur Definition von Stored Procedures häufig an ein bestimmtes Datenbanksystem gebunden sind und nur schwerlich auf Systeme anderer Hersteller übertragen werden können.

Außerdem können mit der EGL sowohl Client- als auch Server-Anwendungen erstellt werden, da sie nicht direkt an ein Datenbanksystem gebunden ist. Stored Procedures hingegen stehen in der Regel nur serverseitig zur Verfügung.

5 Lehrgang

In den Kapiteln fünf bis elf soll dem Entwickler konkretes Wissen über die Softwareentwicklung mit Hilfe der EGL vermittelt werden. Der praktische Teil des Lehrgangs gliedert sich in drei Abschnitte: Einrichtung der nötigen Software, Erstellung eines Hello-World-Programms und Implementierung einer datenbankgestützen Reportinganwendung auf zwei verschiedene Arten. Weiterhin sind zwei theoretische Kapitel über den verwendeten Zeichnsatz innerhalb eines EGL-Programms und die Möglichkeiten zur Fehlerbehandlung enthalten. Als Zielplattform wurde für diesen Kurs die Java gewählt. Dafür spricht eine Reihe von Gründen:

- Java ist mittlerweile als Open-Source-Software unter der General Public License (GPL)[6] veröffentlicht [GOL]. Dies erhöht den Schutz der Investitionen in Software, da alle Bestandteile der Java-Umgebung im Source-Code erhältlich sind und entsprechend den Bestimmungen der GPL frei weiterentwickelt werden dürfen. Die Java-Plattform kann so herstellerunabhängig an künftige Gegebenheiten angepasst werden. Außerdem ist es möglich, den Code (z. B. der Java-Klassenbibliothek oder der Virtual Maschine) auf evtl. Sicherheitslücken oder andere Probleme zu überprüfen.

- Die Java-Plattform steht auf vielen Zielsystemen zur Verfügung. Dazu zählen unter anderem: z/OS, Linux / Unix, aber auch Windows. Man muss sich somit nicht an einen bestimmten Betriebssystemtyp oder -hersteller binden. Außerdem bietet sich dem Entwickler die Möglichkeit, dass er seine Software auf einem lokalen Rechner installieren und testen kann, auch wenn diese letztlich für eine ganz andere Plattform geschrieben wird.

- Neben der Verfügbarkeit als Open-Source-Software haben sich zahlreiche Softwarehersteller auf diese Plattform spezialisiert. IBM bietet beispielsweise neben

[6] Die GPL ist eine weit verbreitete Open Source Lizenz, unter der viele Programme (wie z. B. Linux) veröffentlicht wurden. Nähere Informationen finden sich unter anderem in [GNU].

umfangreicher Software auch Prozessoren für ihre zSeries-Rechner an, die ausschließlich für die Ausführung von Java-Programmen bereitgestellt werden [IBM2].

- Weiterhin ist es möglich Java-Objekte innerhalb eines EGL-Programms zu verwenden. Es stehen somit bereits unzählige fertige Java-Komponenten zur Verfügung, die lediglich eingebunden werden müssen. Auch die Entwicklung im Team profitiert von der Möglichkeit der Java-Einbindung. Java- und Datenbankspezialisten können in einem solchen Modell komplizierte Persitenzlogiken direkt in der objektorientierten Sprache entwickeln, während Fachspezialisten diese Objekte dann in einem EGL-Programm zur Implementierung der Businesslogik verwenden können.

5.1 Notation von Menüabläufen

Um die Schreibweise dieses Lehrgangs möglichst kompakt zu halten, werden aufeinander folgende Menüaktionen, die vom Nutzer durchgeführt werden müssen, nicht mit Sätzen in normaler Sprache sondern mit einer Pfeilnotation formuliert. Eine Abfolge wie „Klick auf Menü X, Klick auf Menü Y“ wird notiert als X → Y, wobei es sich bei X und Y um die Beschriftung der entsprechenden Menüpunkte oder Buttons handelt.

5.2 Installation

In diesem Kurs wird die komplette Entwicklung innerhalb einer virtuellen Maschine auf Basis von VM-Ware durchgeführt. So kann den Teilnehmern die umfangreiche Installation der Software abgenommen werden. Zum Ausführen des Images muss lediglich der kostenlose VM-Ware-Player installiert werden. Die aktuelle Version des VM-Ware-Players ist unter [VMP] im Internet zu finden. Er steht im Augenblick für Linux und Windows zu Verfügung.

Nach der Installation wird die selbstentpackende Datei EGL-Tutorial.exe (Windows Executable) auf der beigelegten DVD geöffnet und ihr Inhalt in ein beliebiges Verzeichnis auf

der Festplatte entpackt. Die virtuelle Maschine wird durch einen Doppelklick auf die Datei „Windows XP Professional.vmx“ gestartet.

Das Image enthält folgende Produkte:

- Apache FOP 0.92 Beta
- Foxit Reader 2.0
- IBM Rational Websphere Developer for zSeries 6.01
- IBM DB2 Express Edition 9.1.0.356
- Microsoft Windows XP Professional mit Service Pack 2 (Betriebssystem)

5.3 Aufbau eines EGL-Projekts für Java

Grundsätzlich besteht ein EGL-Projekt aus mehreren Hauptbestandteilen:

- **EGL-Source-Code Dateien**: Diese enthalten die eigentliche Logik.
- **EGL-Build-Descriptor**: Enthält eine Beschreibung der Umgebung. Hier wird beispielsweise festgelegt, welche Zielplattform verwendet wird oder welche Parameter zum Ansprechen einer evtl. genutzten Datenbank verwendet werden.
- **Java-Source-Code Dateien**: Werden nach der Generierung automatisch aus dem EGL-Source-Code und der dazugehörigen EGL-Beschreibung erstellt.

5.4 Vorbedingungen

Aufgrund des hohen Funktionsumfangs des WDz hat IBM die Funktionen in einzelne Rollen untergliedert, die sich auf bestimmte Arbeitsgebiete beziehen (z. B. Java- oder EGL-Entwicklung). Wird eine solche Rolle aktiviert, werden alle Funktionen, die zur Entwicklung auf einem bestimmten Arbeitsgebiet notwendig sind, in die Entwicklungsumgebung

integriert. Der Nutzer wird so nicht von einer Unzahl überflüssiger Funktionen „erschlagen“. Für die Entwicklung mit der EGL muss deshalb die Rolle „EGL Development“ aktiviert werden. Dies geschieht durch Öffnen der Einstellungen im Menü Window → Preferences, unter Workbench → Capabilites → EGL Developer. Dort ist ein Haken bei EGL Development zu setzen (siehe Abbildung 4) und mit OK zu bestätigen.

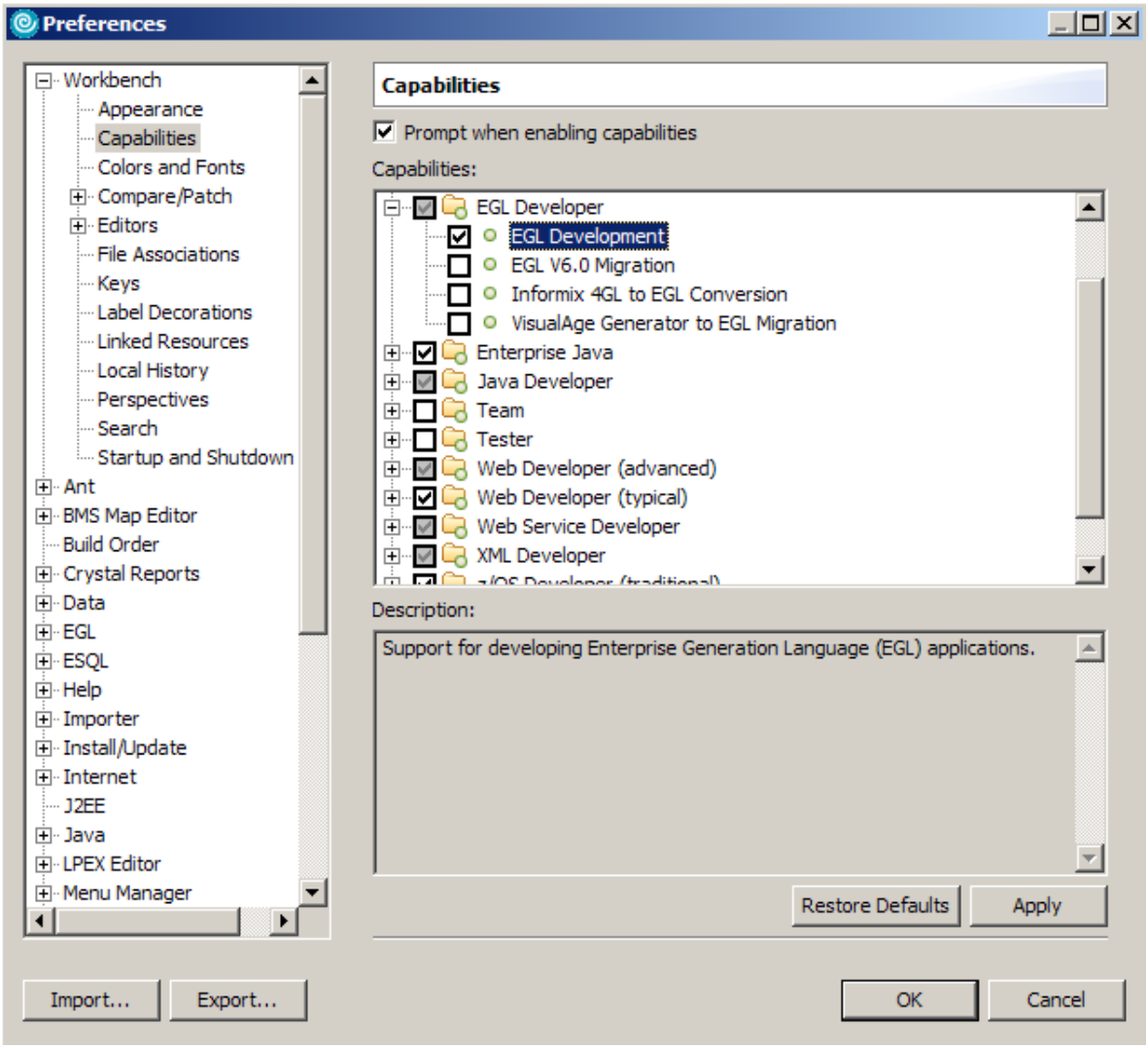

Abbildung 4: Auswahl der Rolle EGL Developer im Menü Window → Preferences

6 Hello World

Um jedem Kursteilnehmer die Möglichkeit zu geben, mit der EGL zu experimentieren, wird in dieser Lektion lediglich ein Hello-World-Programm erstellt, welches eine Ausgabe auf der Konsole erzeugt.

Durch dieses einfache Beispiel soll ein Einstiegspunkt in die Welt der EGL gegeben werden, ohne sich mit der Dokumentation der EGL vertraut machen zu müssen.

Um das Tutorial zu starten, muss zunächst in die EGL-Perspektive gewechselt werden. Dies geschieht (wie bei Eclipse üblich) durch einen Klick auf das Wort EGL in der rechten Menüliste oberhalb des Editorfensters (Abbildung 5). Alternativ: Window → Open Perspective → Other… → EGL.

Abbildung 5: Wechseln einer Perspektive

Nun kann ein neues EGL-Projekt erstellt werden: File → New → EGL Project.

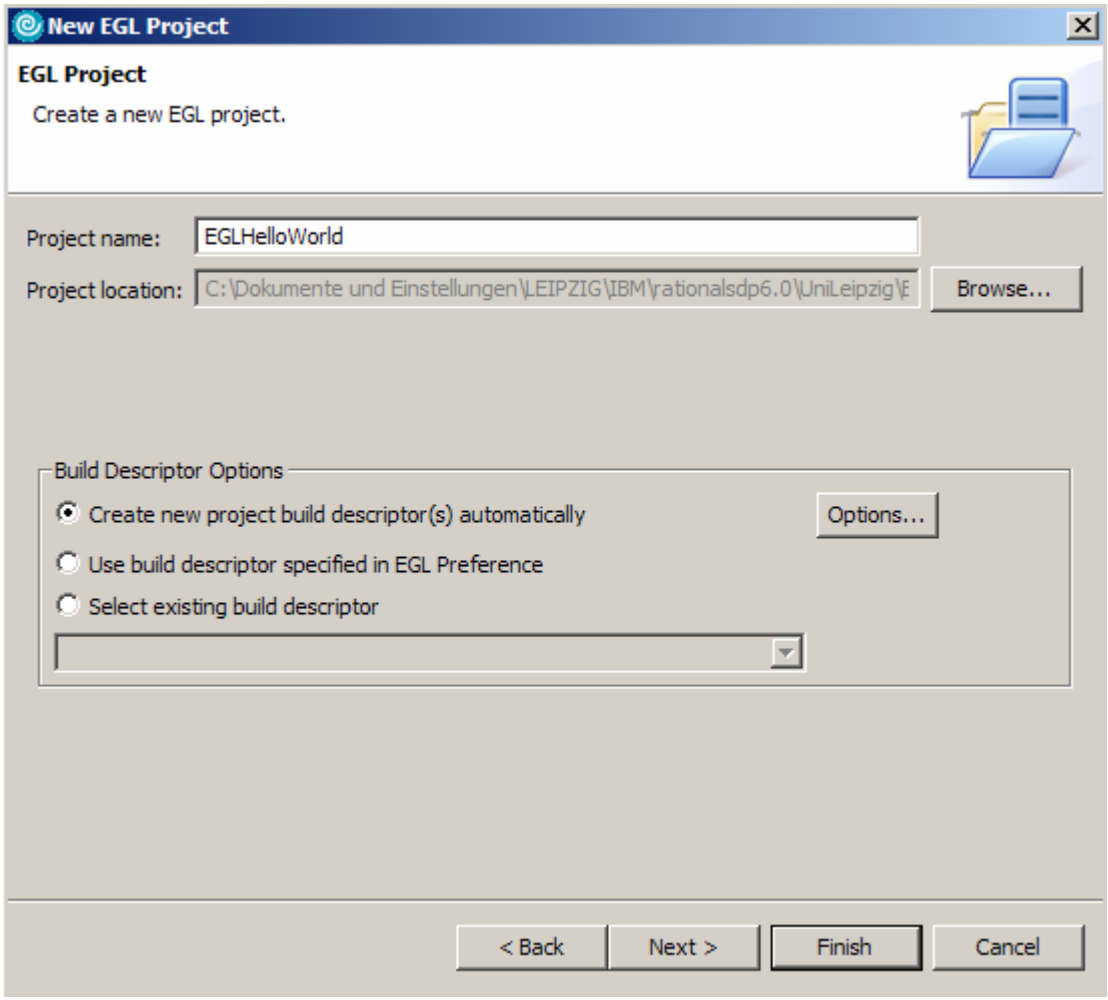

Abbildung 6: New EGL Project Wizzard

Es öffnet sich ein Wizzard. Dort wird der Name des Projekts angegeben (siehe Abbildung 6). In diesem Fall beispielsweise „EGLHelloWorld“. Weiterhin sollte darauf geachtet werden, dass die Option „Create new project build descriptor(s) automatically“ aktiviert ist. Durch einen weiteren Klick auf Finish wird das Projekt erzeugt.

Das neue Projekt ist nun im Project Explorer auf der linken Bildschirmseite unter EGL Projects zu finden.

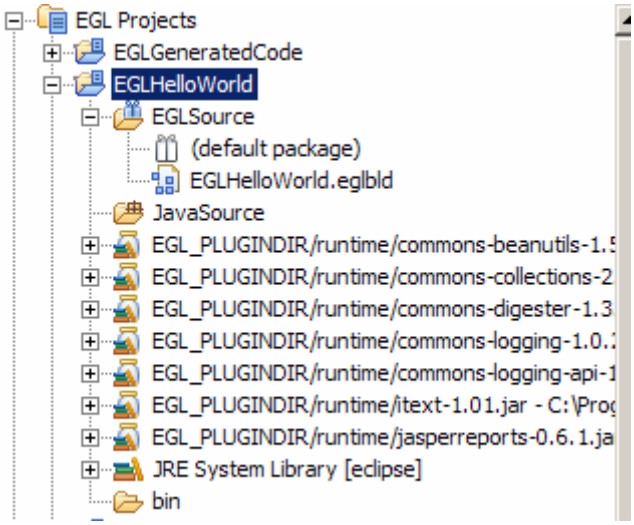

Abbildung 7: Project Explorer mit EGL-Projekt

Es enthält zahlreiche Java-Bibliotheken und zwei Ordner für EGL- und Java-Quellcode. Im EGLSource-Ordner ist bereits die Datei EGLHelloWorld.eglbld enthalten. Hierbei handelt es sich um den erwähnten EGL-Build-Descriptor (siehe Kapitel 5.3 dieser Arbeit). In dieser Datei sind zahlreiche Einstellungen, die zum Generieren des Java-Codes notwendig sind, enthalten. Diese Einstellungen wurden vom WDz teilweise automatisch vorbelegt und können nun angepasst werden. Um die Einstellungen zu ändern, genügt ein Doppelklick auf den Dateinamen. Es öffnet sich die in Abbildung 8 dargestellte Ansicht der Buildoptionen. Da in diesem Fall keine Einstellungen nötig sind, kann die Datei ohne Änderungen wieder geschlossen werden.

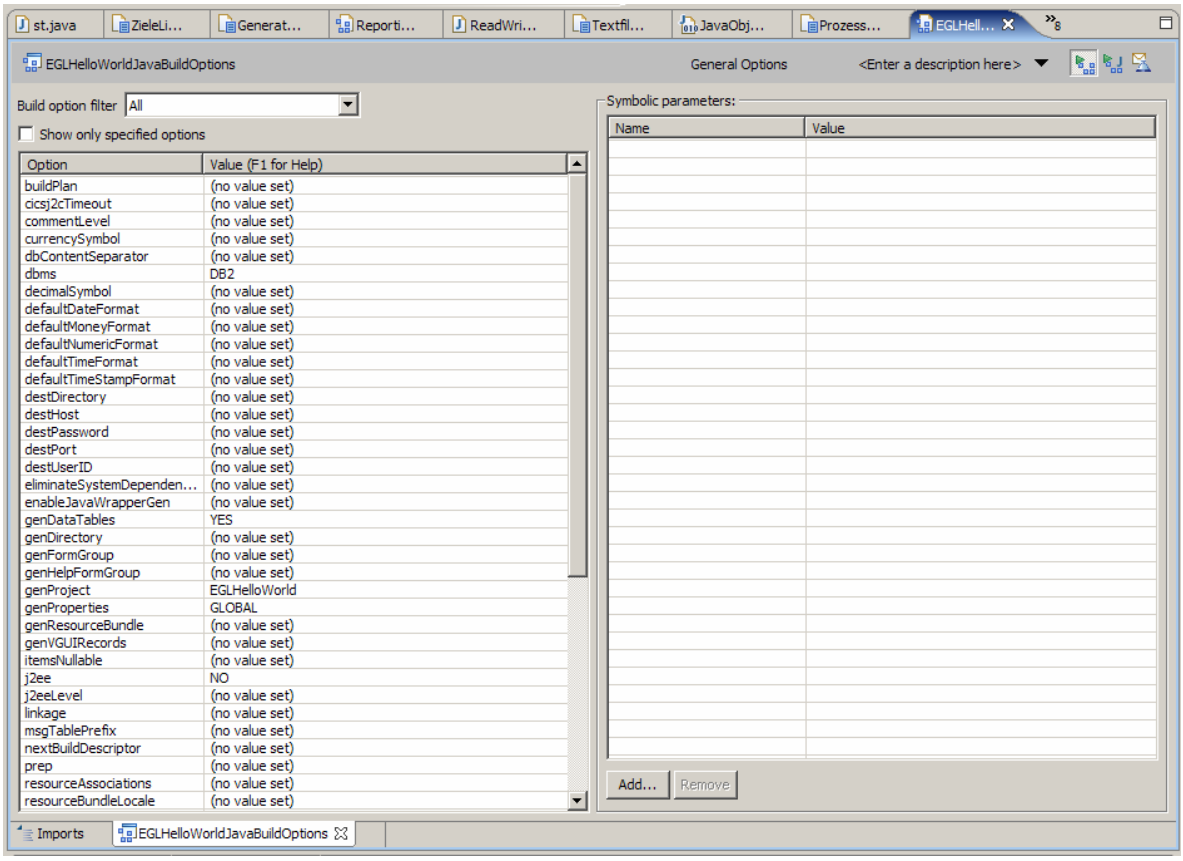

Abbildung 8: Buildoptionen

Als nächstes muss eine neue Source-Code-Datei erstellt werden, die später die eigentliche Ausführungslogik enthalten soll: File → New → Program. Wie in Abbildung 9 gezeigt, werden nun Dateiname (EGL source file name = HelloWorldMain) und Programmtyp (EGL Program Type = Basic – Create a basic program) festgelegt. Weitere Einstellungsmöglichkeiten sind der Zielordner, in dem die Datei abgelegt werden soll, und das Package. Bei einem Package handelt es sich (wie bei Java) um eine Art Namespace, der Funktionen enthalten kann. Gleichartige Funktionen können so zur besseren Strukturierung in einem gemeinsamen Package zusammengefasst werden. Für das „Hello World“-Beispiel ist keine Änderung der Einstellungen notwendig. Der Dialog ist über Finish zu schließen.

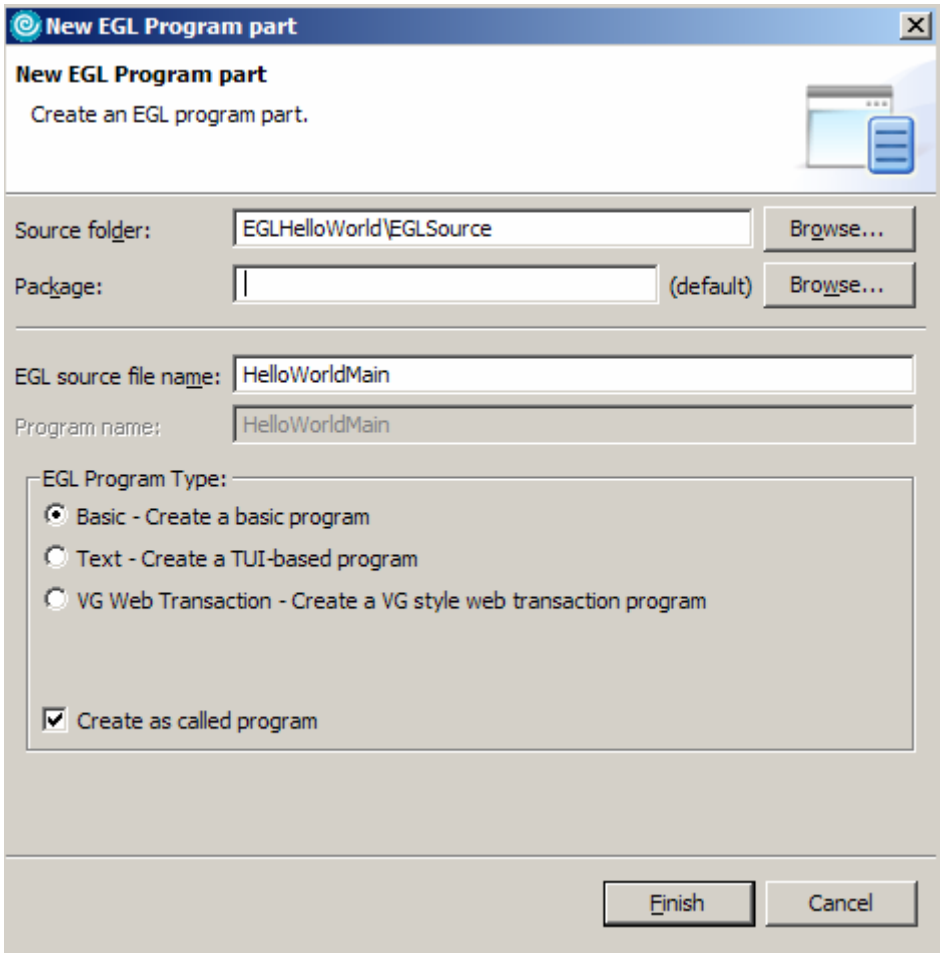

Abbildung 9: Neues EGL-Programm erstellen

Bevor die Source-Code-Datei erstellt wird, wird beim Nutzer noch einmal nachgefragt, ob wirklich das Default-Package verwendet werden soll. Diese Frage ist mit Yes zu beantworten.

Die neue Datei (HelloWorldMain.egl) wird jetzt automatisch im Source-Code-Editor geöffnet und ist ebenso im Project Explorer sichtbar.

Die Datei hat folgenden Inhalt:

```
// basic called program
//
program HelloWorldMain type BasicProgram
   (parm1 typeDefOrPrimitive1, parm2 typeDefOfPrimitive2)
   {msgTablePrefix = "msgTableIdentifierPrefix"}
```

```
    // Data Declarations
    identifierName declarationType;
    // Use Declarations
    use usePartReference;

    function main()
    end
end
```

Dieser Code ist nicht lauffähig und enthält unnötige Bestandteile, die der WDz zur Verdeutlichung der Syntax generiert. Der Code wird deshalb durch den folgenden ersetzt:

```
program HelloWorldMain type BasicProgram

    function main()
          sysLib.writeStdout("Hello World");
    end

end
```

Dieses Programm besteht lediglich aus einer Hauptfunktion (main()), die beim Starten des Programms ausgeführt wird.

Innerhalb der Funktion wird mit Hilfe der Standardbibliothek sysLib ein String in den Standard-Out-Stream ausgegeben.

Um das Programm kompilieren zu können, wird es in Java-Source-Code umgewandelt. Dies geschieht durch einen Rechtsklick auf den Ordner EGLSource im Project Explorer. Im daraufhin geöffneten Kontextmenü wird Generate gewählt (siehe Abbildung 10).

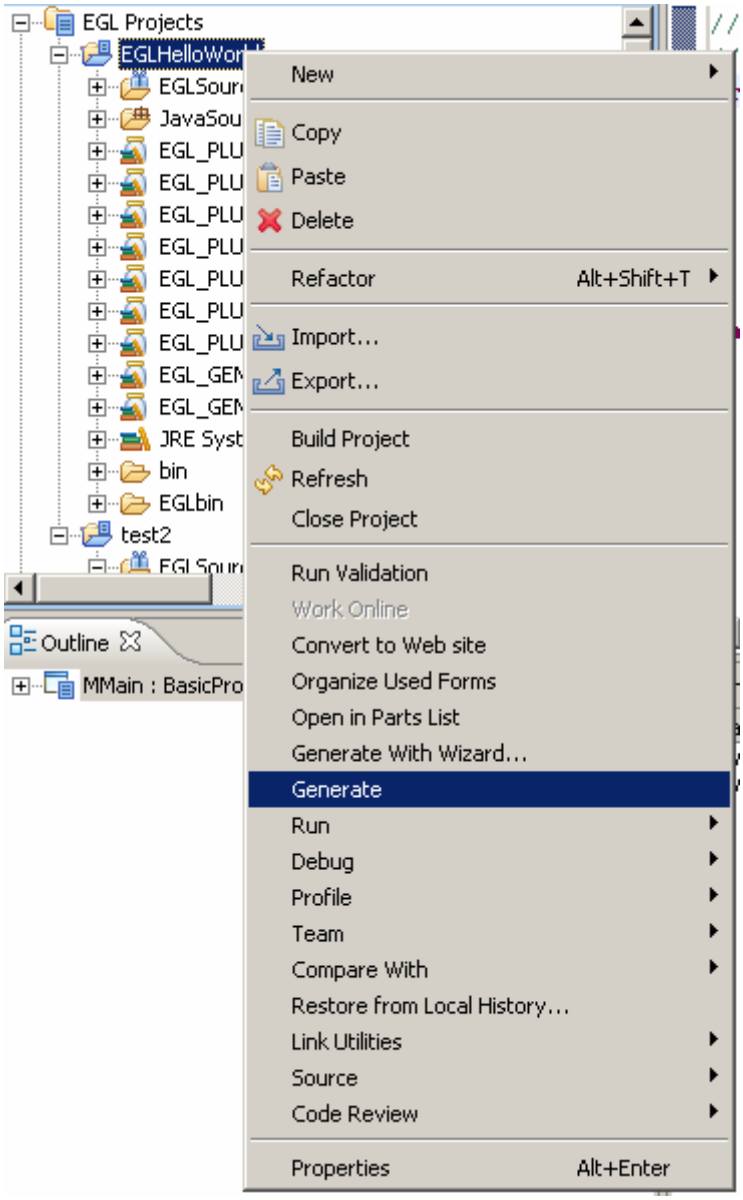

Abbildung 10: Generierung des Java-Sourcecodes

Falls die Source-Code-Datei zu diesem Zeitpunkt noch nicht gespeichert wurde, wird vom WDz gefragt, ob die Datei jetzt gespeichert werden soll. Diese Frage ist mit OK zu beantworten. Der WDz erzeugt nun selbstständig den Java-Source-Code und legt ihn im Ordner JavaSource (default package) ab. In diesem Fall handelt es sich dabei um die Datei namens HelloWorldMain.java.

Als nächstes wird das Programm im Debugger ausgeführt und ein Breakpoint gesetzt, um es an einer bestimmten Stelle anzuhalten.

Das Setzen eines Breakpoints geschieht durch Doppelklick auf die graue Leiste auf der linken Seite des Source-Code-Editors. Bei späteren Debuggerdurchläufen wird der Programmablauf in jeder Zeile angehalten, die einen solchen Breakpoint enthält. Hierbei ist zu beachten, dass nur in den Zeilen Breakpoints gesetzt werden können, die ausführbare Logik enthalten. In diesem Beispiel ist dies nur in der Zeile mit dem Aufruf von sysLib.writeStdout() möglich.

Nach Setzen des Breakpoints wird eine neue Debugger-Konfiguration angelegt. Das geschieht durch Run → Debug ... und muss nur einmal durchgeführt werden, da die Konfiguration im WDz gespeichert wird.

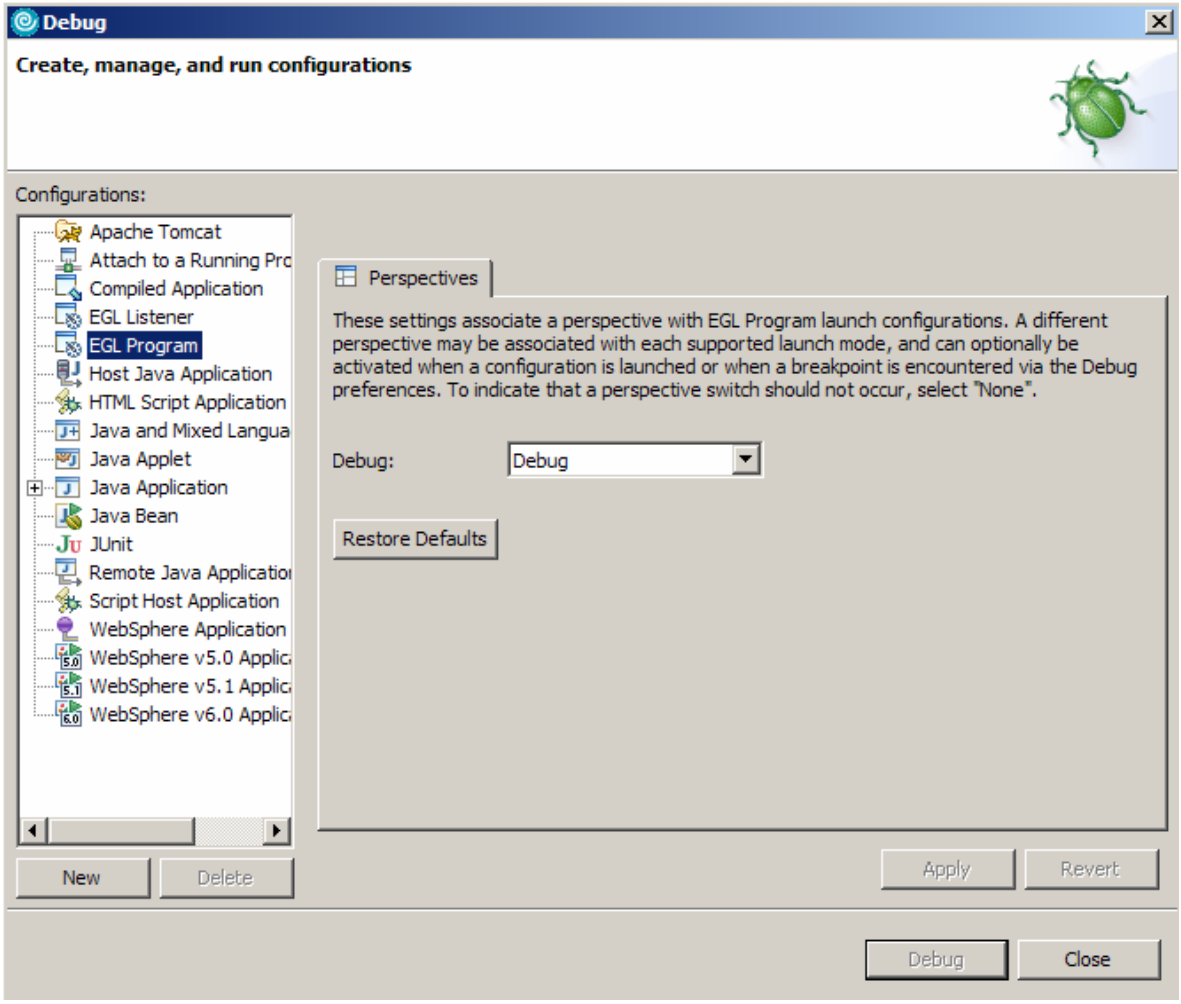

Abbildung 11: Neue Debuggerkonfiguration anlegen

Man wählt nun EGL-Program und klickt auf New. Über das Eingabefeld Name kann jetzt der Name der Konfiguration eingeben werden (z. B. HelloWorldDebug). Dann wird das

Projekt ausgesucht, welches ausgeführt werden soll. Zu diesem Zweck wird der Browse-Button gedrückt und das Projekt EGLHelloWorld selektiert. Abschließend muss der Einstiegspunkt in das Programm angegeben werden. Hierzu wird der Search-Button angeklickt, EGLSource/HelloWorldMain.egl ausgewählt und OK angeklickt. Weitere Einstellungen sind nicht nötig.

Ein Klick auf Debug startet den Debugger mit der neuen Konfiguration. Falls die Source-Code-Datei noch nicht gespeichert wurde, wird nachgefragt, ob diese gespeichert werden soll. Dies wird mit OK bejaht.

Der WDz wechselt jetzt die Perspektive von EGL nach Debug. Diese Perspektive gliedert sich in 5 Hauptbestandteile:

1. **Links oben:** Enthält Informationen über den DebugThread
2. **Rechts oben**: Informationen über Variablen
3. **Mitte links:** Ansicht des Quellcodes, an dem sich die Ausführung gerade befindet. Die aktuelle Zeile ist dabei mit einem kleinen Pfeil markiert.
4. **Mitte rechts**: Struktur der geöffneten Source-Code-Datei.
5. **Unten:** Ausgabe auf der Konsole (das Register Console muss ausgewählt sein).

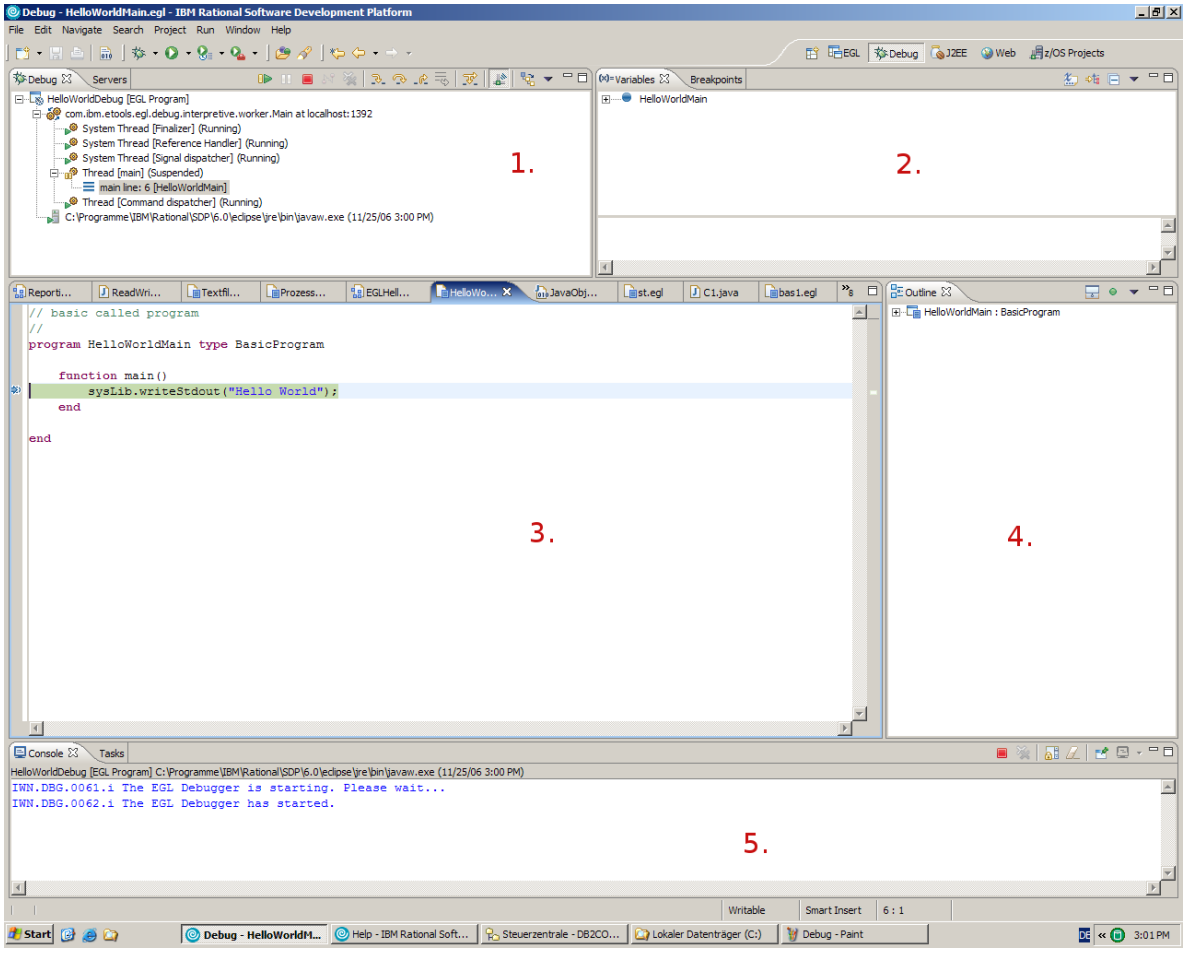

Abbildung 12: Debuggerperspektive

In der Quellcodeansicht ist bereits das Hauptprogramm geöffnet. Außerdem ist die Codezeile, in der der Breakpoint gesetzt wurde, markiert. Durch Drücken der Taste F6 kann zur nächsten Anweisung gesprungen werden.

Wie erwartet wird "Hello World" auf der Konsole (Bereich 5) ausgegeben. Ein weiterer Tastendruck auf F6 beendet das Programm, da die letzte Anweisung ausgeführt wurde.

Es folgt eine Übersicht der wichtigsten Debuggerbefehle:

- **F8 Resume:** Das Programm wird bis zum nächsten Breakpoint ausgeführt (in einem Programm können mehrere Breakpoints gesetzt werden).
- **F6 Step Over:** Die aktuelle Anweisung wird ausgeführt. Das Programm wird nach der Ausführung angehalten.

- **F5 Step Into:** Falls es sich bei der aktuellen Anweisung um eine Funktion handelt, für die der Source-Code zur Verfügung steht, wird die entsprechende Source-Code Datei geöffnet. Als nächster Schritt wird die erste Anweisung innerhalb dieser Funktion angezeigt.

Nach Beendigung des Durchlaufs kann die Perspektive wieder gewechselt werden. Hierzu muss man den Knopf "EGL" in der Leiste am oberen Bildschirmrand drücken (siehe Abbildung 5).

Um das Programm auf einem beliebigen Computer ausführen zu können, muss eine (für Java übliche) JAR-Datei[7] erzeugt werden. Diese Datei enthält den kompilierten Bytecode der Anwendung. Zum Erzeugen der JAR-Datei wird File → Export ausgewählt. Es öffnet sich ein neuer Dialog, in dem JAR File gewählt und Next gedrückt wird (siehe Abbildung 13).

[7] Im Grunde handelt es sich hierbei um eine ZIP-Datei.

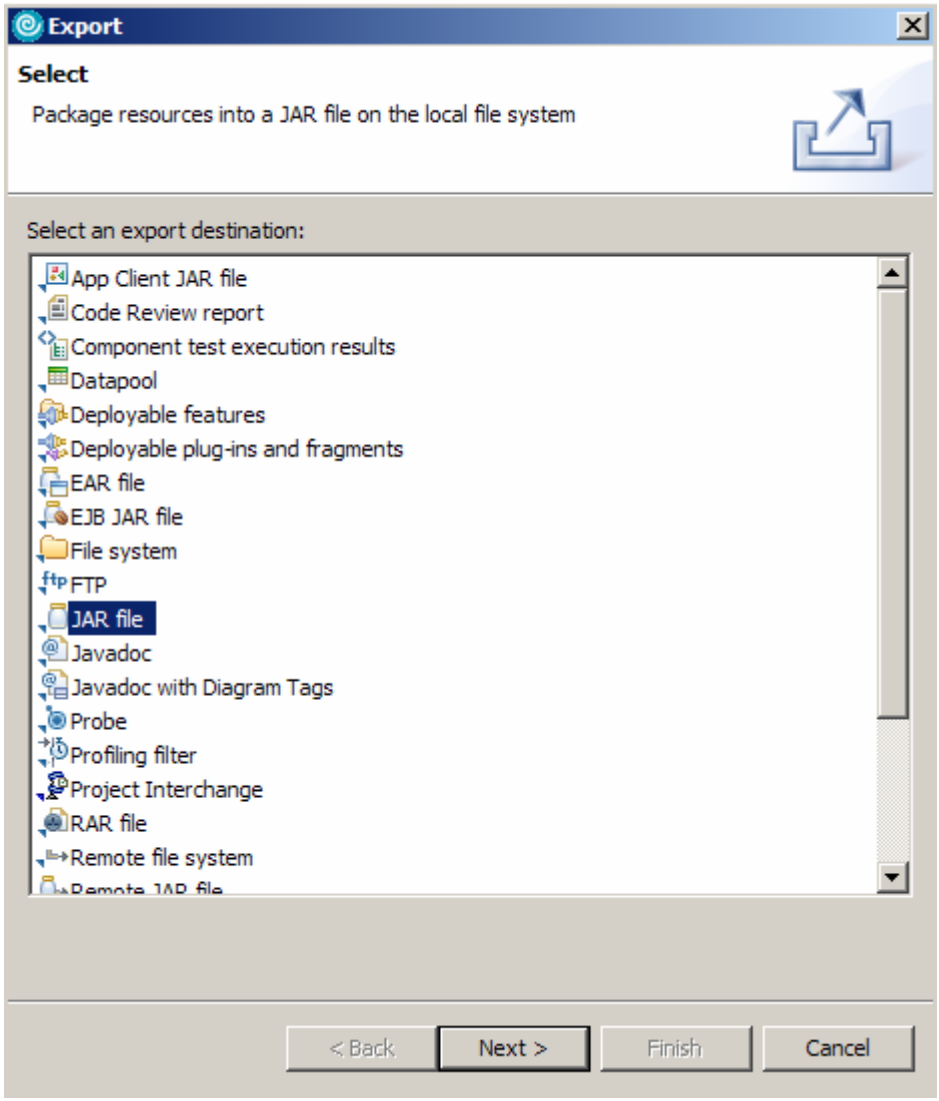

Abbildung 13: Export einer JAR-Datei

Als nächstes wird das Projekt ausgesucht, das exportiert werden soll (vor EGLHelloWorld ist ein Haken zu setzen). Über den Browse Button werden Zielverzeichnis und Name der JAR-Datei festgelegt (beispielsweise c:\EGLHelloWorld.jar). Man klickt nun zweimal auf Next und wählt die Funktion, die beim Programmstart als erstes ausgeführt werden soll (Main class). Dies geschieht durch einen Klick auf den Browse Button und durch Auswahl von HelloWorldMain (Doppelklick auf den Namen, siehe Abbildung 14).

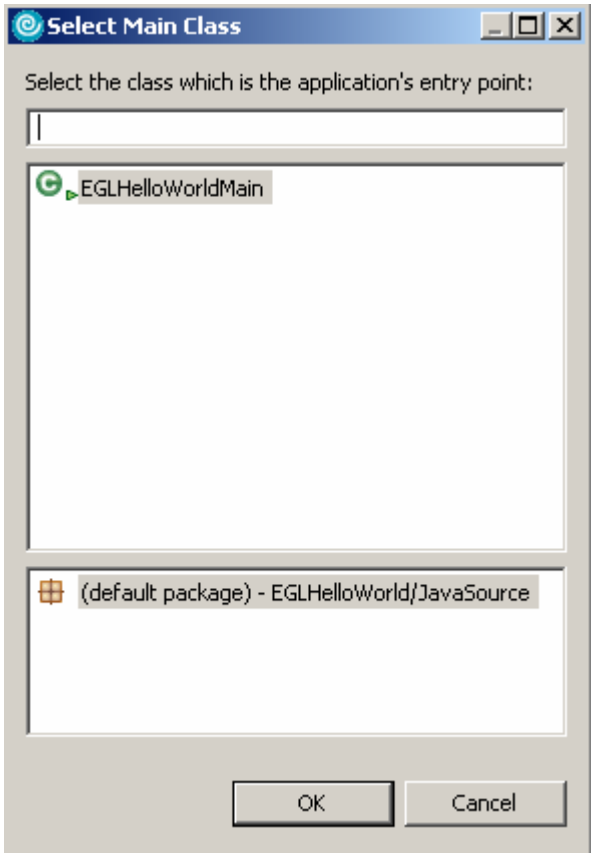

Abbildung 14: Auswahl des Einstiegspunktes

Mit einem finalen Klick auf Finish wird die JAR-Datei erzeugt. Um das Programm ohne WDz auszuführen, öffnet man die Eingabeaufforderung und gibt folgenden Befehl ein:

```
java -jar c:\EGLHelloWorld.jar
```

Nach kurzer Zeit erscheint der Schriftzug "Hello World" in der Konsole (Abbildung 15).

```
C:\Dokumente und Einstellungen\LEIPZIG>java -jar c:\EGLHelloWorld.jar
Hello World
```

Abbildung 15: Ergebnis: Lauffähiges Hello World Programm

Die Erstellung des ersten funktionsfähigen EGL-Programms ist damit abgeschlossen.

Um weitere Abläufe in das Hello World Programm integrieren zu können, folgt eine Übersicht der grundlegenden Programmkonstrukte der EGL:

- **Variablendeklaration**

Syntax	Erklärung
Variablenname Typ;	Die Deklaration erfolgt durch Angabe des gewünschten Namens und des Typs der Variablen.

- **Bedingte Ausführung**

Syntax	Erklärung
If (*boolscher Ausdruck***)** *Anweisungen…* **end**	Der Code zwischen *If* und *end* wird ausgeführt, wenn der eingeklammerte boolsche Ausdruck *true* zurückliefert. Mögliche Operatoren: ==, !=, >=, <=, >, <, in, is, like, matches, not. Eine Beschreibung dieser Operatoren findet sich in [IBM1].

- **Schleifen**

Syntax	Erklärung
for (*Zählvariable* **from** *Startwert* **to** *Endwert* **by** *Inkrement***)** *Anweisungen…* **end** **While (***boolscher Ausdruck***)** *Anweisungen…* **end**	Wie in anderen Sprachen üblich, wird bei der for-Schleife eine Zählvariable inkrementiert, bis ein Zielwert erreicht ist. Die while-Schleife wird hingegen so lange ausgeführt, bis der boolsche Ausdruck im Schleifenkopf *false* ist.

- **Funktionsdeklaration**

Syntax	Erklärung
Function *Funktionsname* **(***Parameterliste***) [returns** *Datentyp***]** *Anweisungen*… **end**	Deklaration einer Funktion. Die Parameter werden per Referenz übergeben und können sowohl als Ein- als auch als Ausgabeparameter fungieren. Falls die Funktion einen Wert zurückgibt, muss das optionale Schlüsselwort returns gefolgt vom Datentyp des Rückgabewerts angegeben werden. Innerhalb der Funktion wird der Rückgabewert durch das Schlüsselwort return gefolgt vom gewünschten Wert, gesetzt. Der Funktionsaufruf endet nach der return-Anweisung.

- **Ende eines Aufrufs**

Syntax	Erklärung
;	Grundsätzlich wird jede vollständige Anweisung mit einem Semikolon abgeschlossen.

- **Aufruf einer Funktion, die sich in einer Bibliothek befindet**

Syntax	Erklärung
*Bibliotheksname***.***Funktionsname*	Um eine Funktion, die sich in einer Bibliothek befindet, aufzurufen, muss vor dem Funktionsnamen der Name der Bibliothek angegeben werden. Die beiden Bezeichner werden dabei durch einen Punkt voneinander getrennt.

- **Das Schlüsselwort „in“**

Syntax	Erklärung
If (*Suchwert* **in** *Array***)** Anweisungen… **end**	In vielen Programmen muss häufig untersucht werden, ob ein bestimmter Wert in einem Array enthalten ist. Bei den meisten Programmiersprachen muss hierzu eine Schleife implementiert werden, die jedes Array-Element mit dem Suchwert vergleicht. In der EGL kann dieses Problem durch Verwendung des Schlüsselworts „in“ ohne Verwendung einer Schleife gelöst werden. Zusätzlich wird der Array-Index der Fundstelle in eine globale Variable geschrieben (sysVar.ArrayIndex), um an späterer Stelle auf das Array-Element zugreifen zu können.

- **Einbinden eines Packages**

Syntax	Erklärung
Import *Packagename*;	Um den Inhalt eines Packages, das Teil eines Projektes ist, in einer Quellcodedatei eines anderen Packages verfügbar zu machen, wird an den Anfang der Datei das Schlüsselwort Import, gefolgt vom Packagenamen gesetzt. Das Importieren mehrerer Packages ist durch mehrfaches Einfügen des Importstatements möglich.

Weitere Informationen finden sich in der Onlinehilfe des WDz oder in [IBM1].

7 Fehlerbehandlung in der EGL

Da man meist davon ausgeht, dass im Regelbetrieb eines Programms keine Fehler auftreten, wird der Fehlerbehandlung bei der Softwareentwicklung häufig nur niedrige Priorität eingeräumt, da die eigentliche Funktionalität im Vordergrund steht. Verstärkt wird diese Vernachlässigung oft noch durch Personalmangel und Zeitdruck.

Andererseits werden im Rahmen einer SOA häufig einzelne Softwarekomponenten als Services angeboten, die möglichst ohne Nutzerinteraktion funktionieren und trotzdem eine hohe Verfügbarkeit bieten sollen. Deshalb muss der Behandlung von Fehlern in serviceorientierten Umgebungen immer eine hohe Bedeutung beigemessen werden. Jede Software sollte in der Lage sein, Probleme weitestgehend zu kompensieren oder zumindest genauestens zu protokollieren und an entsprechende Stellen weiterzuleiten.

Ein Konzept zur Behandlung von Fehlern und Problemen muss schon in den Entwurf eines Programms einfließen, um zu gewährleisten, dass die Fehlerbehandlung aller Programmteile kompatibel zueinander ist und alle Fehler/Probleme an einer einzigen Stelle konsolidiert werden. Dies ist nicht nur für die Behebung von Fehlern, sondern auch für die Überwachung der Software von großer Bedeutung.

Im Folgenden wird auf die Besonderheiten der Fehlerbehandlung der EGL eingegangen. Die EGL bietet grundsätzlich drei Ansätze:

- Funktionsrückgabewerte
- Globale Variablen
- Ausnahmen

7.1 Fehlerbehandlung über Funktionsrückgabewerte

Die Fehlerbehandlung über Rückgabewerte ist universell einsetzbar und in nahezu jeder Programmiersprache möglich, die die Definition von Funktionen erlaubt. Hierzu werden bestimmte Rückgabewerte als Fehlercodes interpretiert, die dann im späteren Programmverlauf ausgewertet werden können.

EGL Beispiel:

```
// Diese Funktion teilt a durch b und gibt das
// Ergebnis über die Variable ergebnis zurück.
// Im Fehlerfall ist der Rückgabewert < 0, sonst 1.

function div(a float, b float,ergebnis float ) returns int

   // Division durch 0 als Fehler melden.
   if (b == 0) return -1

   ergebnis = a / b;

   return 1;
end
```

Ein Nachteil dieser Methode ist, dass zur genauen Behandlung aller Fehler nach jedem Funktionsaufruf eine If-Anweisung implementiert werden muss, die überprüft, ob ein Fehler aufgetreten ist und davon abhängig die Ausführung fortsetzt oder eine Fehlerbehandlung anstößt. Der Code wird dadurch sehr lang und unübersichtlich, da keine klare Trennung zwischen Fehlerbehandlung und Ausführungslogik besteht. Außerdem kann diese umständliche Art der Fehlerbehandlung dazu führen, dass Fehlercodes (vor allem bei Funktionen, bei denen sehr selten ein Fehler auftritt) überhaupt nicht ausgewertet werden.

7.2 Fehlerbehandlung über globale Variablen

Diese Art der Fehlerbehandlung ähnelt der Verwendung von Rückgabewerten, wird aber in anderen Programmiersprachen selten verwendet. Statt Funktionsrückgabewerte für die Fehlerbenachrichtigung zu verwenden, werden von den einzelnen Funktionen vordefinierte globale Variablen mit entsprechenden Werten belegt.

Im Vergleich zur Behandlung über Rückgabewerte, hat dies den Nachteil, dass sich alle Entwickler auf die Verwendung der selben Variablen einigen müssen, um Kollisionen zu vermeiden. Wenn Code aus verschiedenen Projekten integriert werden muss, führt das vor allem bei Programmiersprachen, die diese Art der Fehlerbehandlung nicht vorsehen, zu Problemen. Außerdem muss bei der Verwendung von globalen Variablen immer beachtet werden, dass sich dadurch Kopplung und Fehleranfälligkeit erhöhen, wobei gleichzeitig Wartbarkeit verschlechtert wird [GLI].

Da die EGL die Verwendung von globalen Variablen explizit vorsieht, wird diesem Problem bereits Rechnung getragen. Für die Auswertung von Fehlern ist die Variable sysVar.errorCode vorgesehen, auf die an allen Stellen innerhalb eines EGL-Programms zugegriffen werden kann. Es handelt sich bei sysVar.errorCode um ein 8 Byte großes Character-Feld, das im Normalfall den Wert „00000000“ enthält. Im Falle eines Fehlers wird dieser Wert durch einen anderen ersetzt (wird beispielsweise beim Aufruf einer externen Java-Komponente eine Ausnahme geworfen, so wird sysVar.errorCode auf den Wert „00001000“ gesetzt).

Von der Verwendung dieser Art der Fehlerbehandlung ist dann abzuraten, wenn der generierte Code in Komponenten eingebunden werden soll, die nicht in EGL implementiert sind, da diese Komponenten die verwendeten globalen Variablen ebenfalls berücksichtigen müssen.

Beispiel zum Verarbeiten einer Java-Ausnahme, mit Hilfe der globalen Variable sysVar.errorCode:

```
// Wurde im vorhergehenden Code eine JavaException geworfen?
if (sysVar.errorCode == "00001000")

   // Fehlermeldung abfragen
   errMsg = JavaLib.invoke( (objId)"caughtException",
                "getMessage" );

   // Fehlermeldung ausgeben
   sysLib.writeStderr(errMsg);

   // Programm beenden
   exit program;
end
```

Hinweis: Die globale Variable sysVar.ErrorCode wird auch bei Verwendung von Ausnahmen zur Fehlerbehandlung gesetzt.

7.3 Fehlerbehandlung über Ausnahmen

Die Verwendung von Ausnahmen ist nur in Programmiersprachen möglich, die dies auch unterstützen. Sie hat aber im Vergleich zu Rückgabewerten und globalen Variablen eine Reihe von Vorteilen.

Bei der Ausnahmebehandlung wird die eigentliche Ausführungslogik innerhalb eines so genannten Try-Blocks[8] implementiert. Tritt in diesem ein Fehler auf, wird eine Ausnahme geworfen und die Ausführung des Try-Blocks abgebrochen. Die geworfene Ausnahme kann von einem Catch-Block gefangen werden kann. Der Code innerhalb des Catch-Blocks wird nur beim Fangen einer Ausnahme ausgeführt und daher nur für die Fehlerbehandlung benötigt. Wird keine Ausnahme geworfen, wird der Catch-Block übersprungen.

[8] Unter einem Block wird in diesem Zusammenhang eine Reihe von Anweisungen verstanden, die zu einer Einheit zusammengefasst werden.

Für das Werfen von Ausnahmen steht in der Regel ein eigener Befehl zur Verfügung, der vom Entwickler in entsprechenden Fehlersituationen eingesetzt werden kann.

Mit Hilfe von Ausnahmen lässt sich der Code zum Behandeln von Fehlern leicht von der eigentlichen Logik trennen. Außerdem steht beim Fangen von Exceptions in der Regel ein kompletter Stacktrace von der auslösenden Funktion bis hin zur Funktion mit dem entsprechenden Catch-Block zur Verfügung. Dies erleichtert das Finden der Ursache. Nähere Informationen zur Ausnahmebehandlung finden sich in [ULL].

Pseudocode:

```
try
{
   // Ausführungslogik
   // Ausnahmen werden hier entweder durch aufgerufene Funktionen,
   // oder direkt geworfen. Z.B.:
   If (Datenbankverbindung.geschlossen == wahr)
         throw new DatabaseException(„Die Datenbankverbindung ist
               geschlossen“);
}
catch (Exception e)
{
   // Fehlerbehandlung
}
```

Es gibt allerdings auch Programmiersprachen, bei denen diese Vorteile nicht oder nur teilweise zum Tragen kommen. Die Sprache Visual Basic (nicht Visual Basic.NET) beispielsweise bietet zwar ein der Ausnahmebehandlung sehr ähnliches Konzept der Fehlerbehandlung, allerdings ohne selbstständig einen Stacktrace zu erzeugen. Soll ein Stacktrace erzeugt werden, muss die Ausnahme (in diesem Fall das Error-Objekt) in jeder einzelnen Funktion gefangen und um die gewünschten Informationen erweitert werden. Anschließend wird es erneut geworfen.

Auch bei der EGL gibt es in der aktuellen Version einige Einschränkungen bei der Fehlerbehandlung mittels Ausnahmen. So bietet die EGL die Möglichkeit Code nach der üblichen Vorgehensweise in Try- und Catch-Blöcke (zur Fehlerbehandlung) zu verteilen. Eine Anweisung zum Werfen einer Ausnahme ist allerdings nicht im Sprachumfang vorhanden.

Das Werfen von Ausnahmen kann somit nicht durch den Programmierer veranlasst oder gesteuert werden. Stattdessen werden Ausnahmen nur vom System innerhalb von systemeigenen oder externen Funktionen geworfen (z. B. Ein-/Ausgabe-Fehler oder Ausnahmen, die von externem Java-Code geworfen werden).

Es ist deshalb z. B. üblich, alle Operationen, die direkt von einer Datenbank oder einer anderen Datenquelle abhängig sind, mit entsprechendem Code zur Ausnahmebehandlung zu umschließen.

Hierzu wird folgende Syntax verwendet:

```
try
   // Anweisungen…
onException
   // Anweisungen zur Fehlerbehandlung…
end
```

Während der Programmausführung wird später „versucht“, alle Anweisungen im Try-Block auszuführen. Falls hierbei ein Systemfehler auftritt, springt das Programm automatisch in den Code nach dem Schlüsselwort onException (dieser Block wird bei fehlerfreien Durchläufen nicht ausgeführt). Da der onException-Block optional ist, kann er auch weggelassen werden. In diesem Fall würde das Programm bei einem Systemfehler innerhalb des Try-Blocks nach dem Ende des Try-Blocks fortgesetzt. Das Weglassen des onException-Blocks ist allerdings nicht zu empfehlen, da Fehler immer behandelt werden sollten.

Durch das Fehlen der Möglichkeit, eigene Ausnahmen zu definieren bzw. zu werfen, ergeben sich einige Probleme:

- **Vermischung von Ansätzen zur Fehlerbehandlung:** Da nur externe Ausnahmen behandelt werden können, muss ein EGL-Entwickler bei allen Funktionen, die innerhalb eines Programms oder einer Bibliothek definiert sind bzw. eine Eigenentwicklung darstellen, eine Fehlerbehandlung mittels Rückgabewert oder globaler

Variable realisieren. Dies führt zur einer Vermischung beider Ansätze, was sich negativ auf Lesbarkeit und Wartbarkeit auswirkt.

- **Schlechte Integration in die Java-Welt:** Im Bereich der Java-Programmierung wird die Fehlerbehandlung in der Regel ausschließlich über Ausnahmen abgewickelt, weil die Programmiersprache die nötigen Voraussetzungen hierfür von Haus aus mitbringt. Wird nun die EGL verwendet, um Java-Code zu erzeugen, ergibt sich ein Bruch im Fehlerbehandlungsprozess. Falls dieser Code später direkt in ein Java-Programm integriert werden soll, muss bei der Programmierung berücksichtigt werden, dass die Fehlerbehandlung innerhalb des EGL-Java-Codes nicht über Ausnahmen abgehandelt wird.

- **Verlust von Information:** Wie bereits beschrieben, bieten die meisten Programmiersprachen mit Ausnahmebehandlung die Möglichkeit, einen Stacktrace von der werfenden Funktion bis zum entsprechenden Fänger abzufragen. In Programmiersprachen ohne Ausnahmebehandlung muss ein entsprechender Stacktrace häufig durch den Entwickler in das Programm eingebaut werden, falls diese Information benötigt wird. Dies zieht letztlich einen höheren Entwicklungsaufwand nach sich und führt zu neuen Fehlerquellen. Wird kein Stacktrace implementiert, ist es (vor allem im Wirkbetrieb) schwer eine Fehlerquelle genau zu identifizieren, wenn nicht jedes Problem mit einer eindeutigen Fehlermeldung beschrieben wurde.

Nach Ansicht des Autors sollte die Definition und Behandlung von eigenen Ausnahmen in den Sprachumfang der EGL integriert werden. Die Logik zur Fehlerbehandlung ließe sich dadurch klarer strukturieren. Die oben genannten Nachteile würden so bei zukünftigen Implementierungen nicht auftreten.

Auch das Auswerten des Fehlertyps stellt bei der EGL im Vergleich zu anderen Sprachen eine Besonderheit dar: In vielen Sprachen werden verschiedenartige Fehler durch eigenständige Ausnahmetypen unterschieden, die von einem allgemeineren Ausnahmetyp abgeleitet sind. Dies ist in Abbildung 16 verdeutlicht.

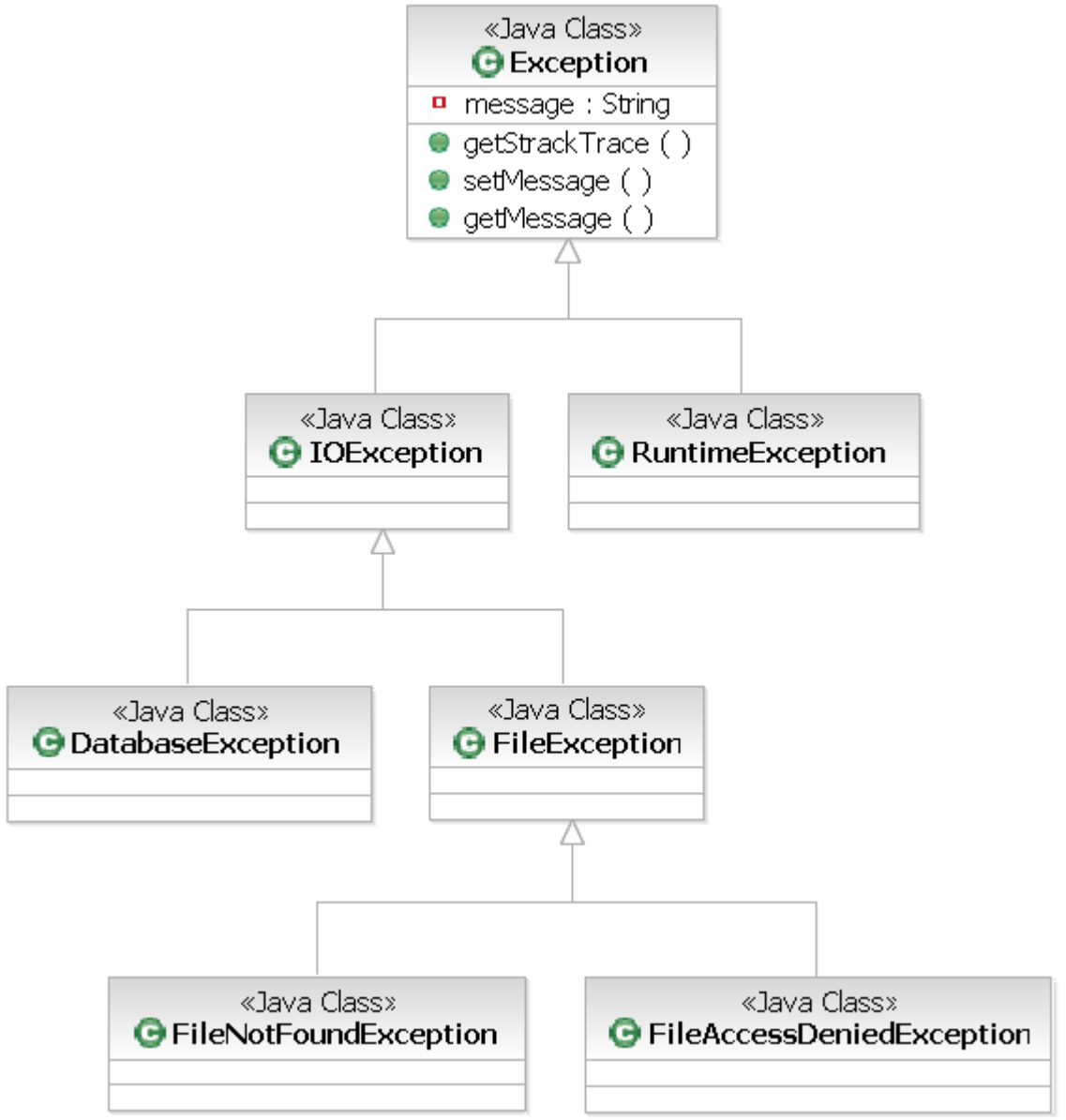

Abbildung 16: Ausschnitt einer Ausnamehierarchie in UML-Notation

Bei der eigentlichen Fehlerbehandlung ist es möglich, in einem bestimmten Catch-Block nur eine bestimmte Art von Ausnahmen zu behandeln. Weiterhin ist es durch mehrfaches Ableiten möglich, einzelne Fehlertypen weiter zu untergliedern. Auf diese Weise kann auch die Detailliertheit der Fehlerbehandlung gesteuert werden. Sollen alle Fehlertypen mit demselben Code behandelt werden, so reicht es, einen einzigen Catch-Block zu implementieren, der alle Exceptions auffängt, die vom Basistyp Exception abgeleitet sind. Soll zwischen Datenbank und Dateifehlern unterschieden werden, wird ein Catch-Block für alle Ausnahmen, die von DatabaseException abgeleitet sind, implementiert. Für die Behand-

lung der Dateifehler wird ein zusätzlicher Catch-Block zum Fangen aller FileExceptions erstellt. Natürlich ist eine weitere Untergliederung möglich.

Die EGL bietet diese Möglichkeit der hierarchischen Vererbung nicht. Stattdessen gibt es nur einen Ausnahmetyp, bei dem sich die Fehlerart über einen String ermitteln lässt. Dieser String kann über die globale Variable sysLib.currentException.code innerhalb eines onException-Blocks abgefragt werden.

Außerdem bieten einige Programmiersprachen eine sinnvolle Erweiterung des Try-Catch-Konzeptes: Bei diesen Sprachen ist noch ein dritter Block vorhanden, der nach dem Durchlaufen des Try- oder Catch-Blocks immer ausgeführt wird. In einigen Sprachen wird dieser Block durch das Schlüsselwort Finally eingeleitet. In diesem Block können nötige „Aufräumarbeiten“, wie das Schließen von Dateien oder Datenbankverbindungen, einmalig implementiert werden.

Dies sei mit folgendem Pseudocode verdeutlicht:

```
try
{
   // Datei zum Lesen öffnen
   // Dateiinhalt verarbeiten
}
catch (Exception e)
{
   // Fehler behandeln
}
finally
{
   // Datei schließen
}
```

Wird z. B. innerhalb des Try-Blocks eine Datei zum Schreiben geöffnet, so muss diese an späterer Stelle im Code wieder geschlossen werden. Falls sich das Kommando zum Schließen der Datei ebenfalls im Try-Block befindet, müsste bei der herkömmlichen Try-Catch-Kombination das Schließen zusätzlich im Catch-Block implementiert werden, da das Kommando im Try-Block bei einem Fehler evtl. nicht erreicht wird. Der Finally-Block bietet nun die Möglichkeit, derartige Aufgaben einmalig zu implementieren, da er in jedem

Fall durchlaufen wird (wenn das Programm nicht vorher komplett beendet wurde). Bei der Weiterentwicklung der EGL sollte auch dieser Aspekt berücksichtigt werden, weil die Anzahl potentieller Fehlerquellen und der Implementierungsaufwand durch Einführung eines Finally-Blocks gemindert werden kann.

8 Entwicklung einer Reportingkomponente

Die Entwicklung einer Reportingkomponente mit Hilfe der EGL ist sinnvoll, weil umfangreiche Auswertungen sehr schnell eine hohe Komplexität erreichen. Hier ist ein umfangreiches Fachwissen über die Art und Verarbeitung der Daten notwendig. Auch müssen solche Reports häufig an neue Gegebenheiten angepasst werden. Es ist deshalb zweckmäßig, Fachpersonal mit der EGL vertraut zu machen, um es in die Lage zu versetzen, die entsprechende Logik selbst zu implementieren.

Zur Verdeutlichung der Möglichkeiten, die die EGL bietet, und um Lösungswege für unterschiedliche Szenarien zu liefern, soll die Komponente auf zwei Arten realisiert werden:

- Implementierung der kompletten Logik mittels EGL.
- Verwendung von Java-Komponenten, die in einem EGL-Programm angesprochen werden.

8.1 Aufgabenstellung

Um den Lehrgang möglichst praxisnah zu gestalten, wurde ein Thema gewählt, das in der Zukunft sehr an Aktualität gewinnen dürfte und nicht schon mehrfach beschrieben wurde. Die Wahl fiel dabei auf ein Teilgebiet des Sabanes & Oxley Act (SOX). SOX ist ein Gesetz zur Unternehmensberichterstattung, dem sich alle Unternehmen, die an US-Börsen gehandelt werden[9], unterwerfen müssen. Ziel von SOX ist es, die Berichterstattung von Firmen transparenter zu gestalten und Möglichkeiten zur Bilanzfälschung einzudämmen.

Teil von SOX ist unter anderem die Überwachung von Geschäftsprozessen durch die Definition von damit verbundenen Risiken, denen über so genannte Kontrollaktivitäten entgegen gewirkt werden soll.

[9] Hierzu zählen auch Unternehmen, die lediglich eine Tochter am US-Kapitalmarkt haben.

In Anlehnung an SOX wurde ein (vereinfachtes) Modell entwickelt, welches aus folgenden Bestandteilen besteht:

- **Prozessgruppe:** Entspricht einer Prozesskategorie. D. h. alle gleichartigen Prozesse werden einer Prozessgruppe zugeordnet.
- **Prozess:** Konkrete Beschreibung eines Ablaufs, um ein bestimmtes Ziel zu erreichen. Jeder Prozess ist (genau) einer Prozessgruppe zugeordnet.
- **Risiko:** Risiken können den erfolgreichen Durchlauf eines Prozesses verhindern oder sich auf sonstige Weise negativ auf ein Unternehmen auswirken. Risiken werden allgemein definiert und einer Prozessgruppe zugeordnet. Jeder Prozess erbt die Risken seiner Prozessgruppe.
- **Kontrollaktivität:** Aktivitäten, die durchgeführt werden, um die Eintrittswahrscheinlichkeit eines Risikos zu minimieren. Kontrollaktivitäten werden definiert und ihrerseits mit allen Risiken verknüpft, denen die Aktivität entgegenwirkt. Zusätzlich sind sie Prozessen zugeordnet.

Vorgabe ist es nun, jedem Prozess so viele Kontrollaktivitäten zuzuordnen, dass alle Risiken, die ein Prozess von seiner Prozessgruppe geerbt hat, vollständig abgedeckt werden.

Die Reportingkomponente hat die Aufgabe, diese Abdeckung für einen bestimmten Prozess zu ermitteln und in Form eines Berichts aufzubereiten.

Die nötigen Daten werden in einer relationalen Datenbank (IBM DB2) abgelegt. Zur Erstellung des Reports wird der XSL-FO (Extensible Stylesheet Language – Formatting Objects) Transformator Apache FOP verwendet (eine passende XSL-Datei ist dem Lehrgang beigelegt). Die eigentlichen Ausgabedaten müssen durch die Komponente in Form von Extensible-Markup-Language (XML) Daten bereitgestellt werden.

8.2 Modellierung der Beziehungen (Entity-Relationship-Modell)

Im Folgenden wird der oben beschriebene Zusammenhang graphisch als Entitiy-Relationship-Modell dargestellt. Für die Darstellung wurde die weit verbreitete IDEF1X-Notation gewählt.

Nähere Informationen zur gewählten Notation und zum Datenbankdesign sind unter anderem zu finden in [IDE] und [ULM].

8.3 Logische Ansicht

In dieser Ansicht sind nur die eigentlichen Nutzdaten zu sehen. Many-to-Many-Beziehungen sind direkt zwischen den einzelnen Entitäten eingezeichnet (siehe Abbildung 17). Ein derartiges Modell kann allerdings nicht direkt auf eine relationale Datenbank abgebildet werden, da Many-to-Many-Beziehungen nur durch die Verwendung von zusätzlichen Hilfstabellen abgebildet werden können (siehe Kapitel 8.4 dieser Arbeit).

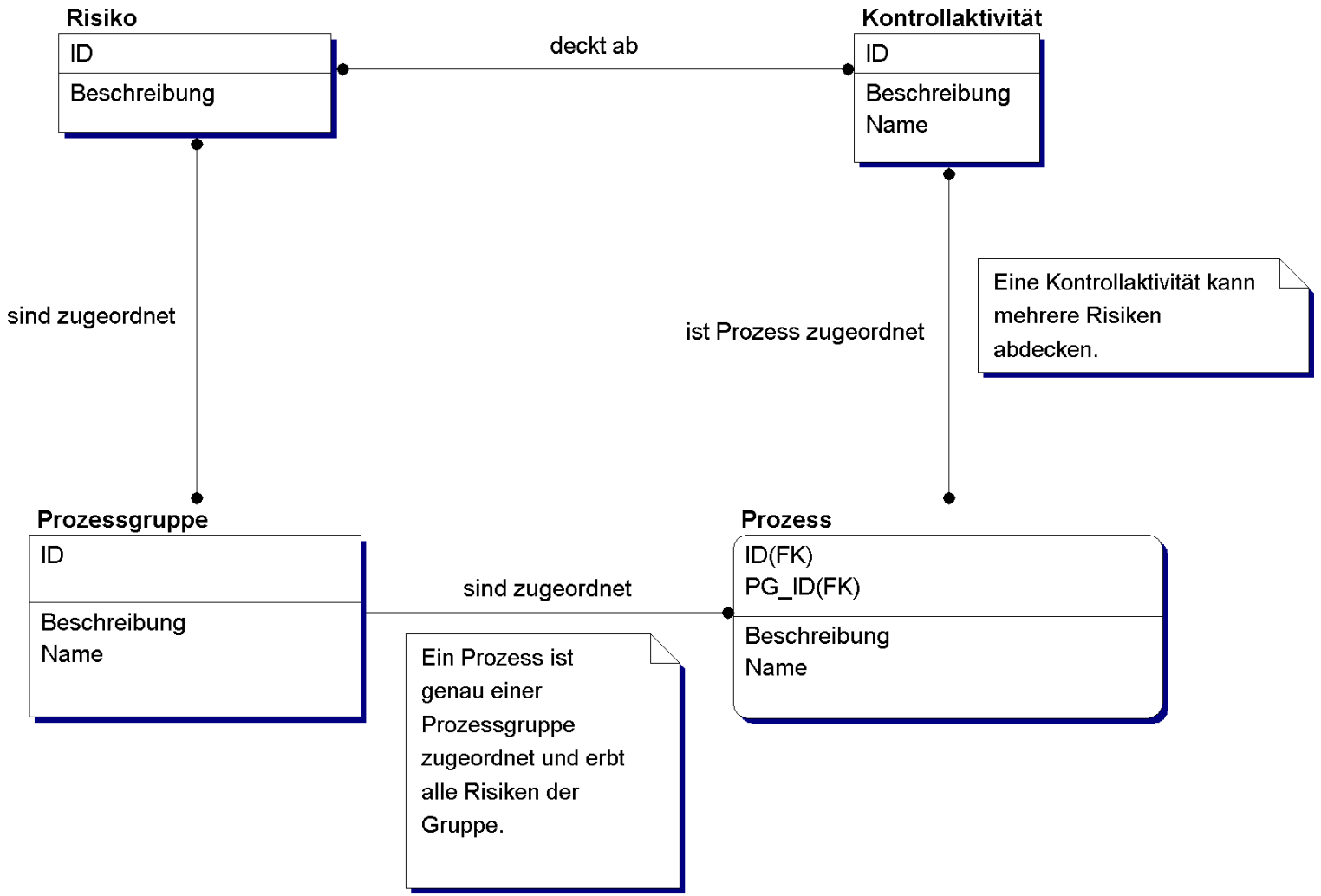

Abbildung 17: Logische Ansicht des ER-Modells

8.4 Physikalisches Modell

Dieses Modell dient zur Veranschaulichung des tatsächlichen Datenbankaufbaus. Neben den eigentlichen Datentabellen sind zusätzliche Hilfstabellen dargestellt, über die die Many-to-Many-Beziehungen abgebildet werden (Abbildung 18).

Für jede Many-to-Many-Beziehung muss eine zusätzliche Hilfstabelle eingeführt werden, da diese in einer relationalen Datenbank nicht direkt modelliert werden kann. Ein einzelner Datensatz einer Hilfstabelle hat dann die Aussage Datensatz X (aus Tabelle A) steht in Beziehung mit Datensatz Y (aus Tabelle B). Die Datensätze der Tabellen A und B lassen sich so beliebig kombinieren.

Lediglich die Beziehung zwischen dem Prozess und seiner Prozessgruppe kann direkt über einen Fremdschlüssel abgebildet werden, da ein Prozess immer genau einer Prozessgruppe zugeordnet ist. Insgesammt besteht die Datenbank aus sieben Tabellen.

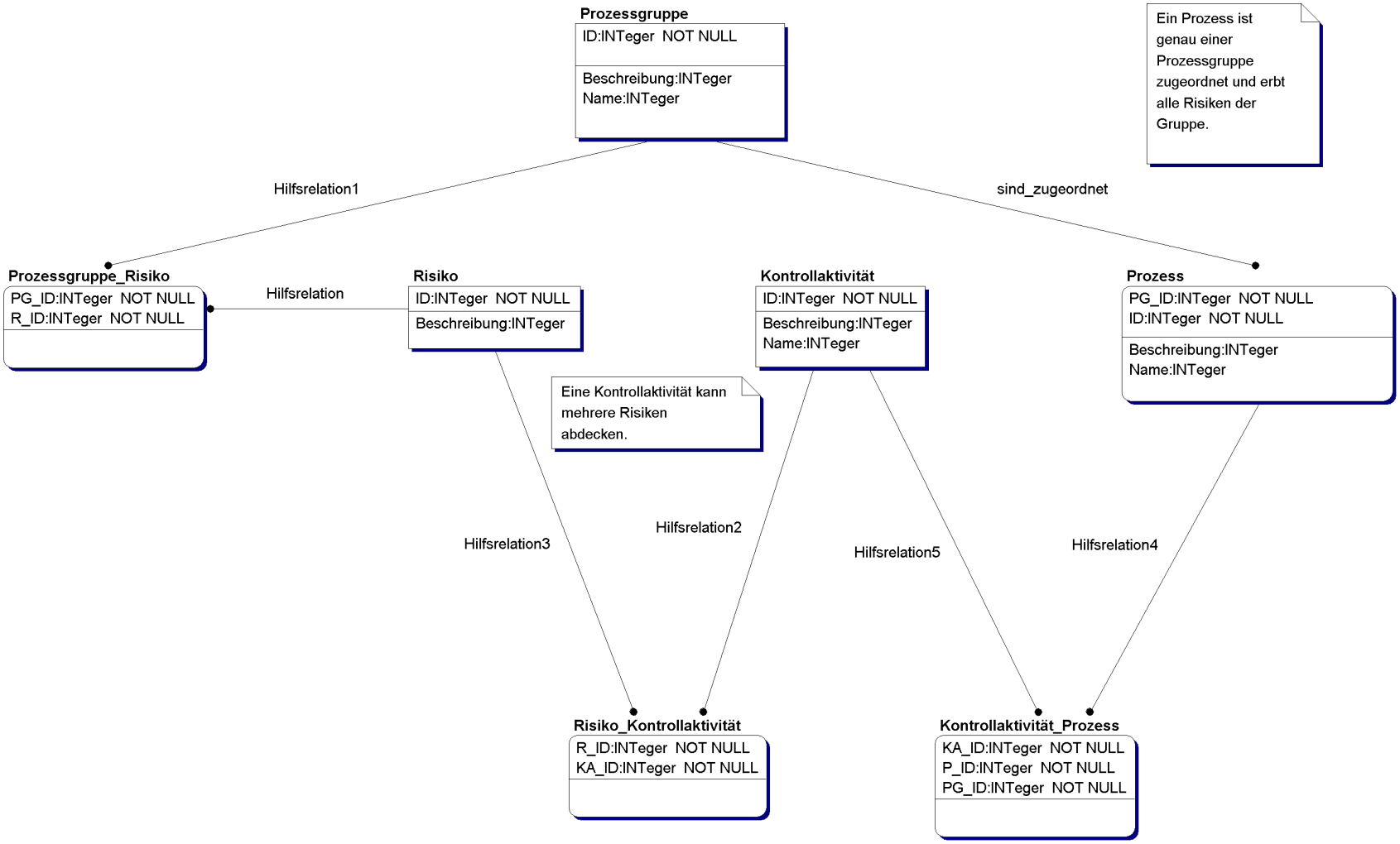

Abbildung 18: Physikalisches Modell

8.5 Aufbau der XML-Datei

Weil die XSL-Transformation nur durchgeführt werden kann, wenn die Quell-Daten in einer passenden XML-Struktur vorliegen, wurde das in Abbildung 19 dargestellte XML-Schema entwickelt. Über das EGL-Programm soll eine XML-Datei erzeugt werden, die diesem Schema entspricht.

Das Schema folgt der W3C-Empfehlung von 2001 und beschreibt die Struktur der XML-Daten. Ein solches Schema dient einerseits zur Verdeutlichung der XML-Struktur und kann andererseits zur Validierung der erzeugten XML-Daten verwendet werden.

Neben XML-Schema gibt es noch weitere Beschreibungssprachen für XML-Daten, wie RelaxNG oder Document Type Definition (DTD). Im Rahmen dieser Arbeit wird XML-Schema verwendet, da es mehr Möglichkeiten bietet als DTD und einen höheren Verbreitungsgrad erreicht hat als RelaxNG [THR]. Außerdem wird XML-Schema direkt durch den WDz unterstützt.

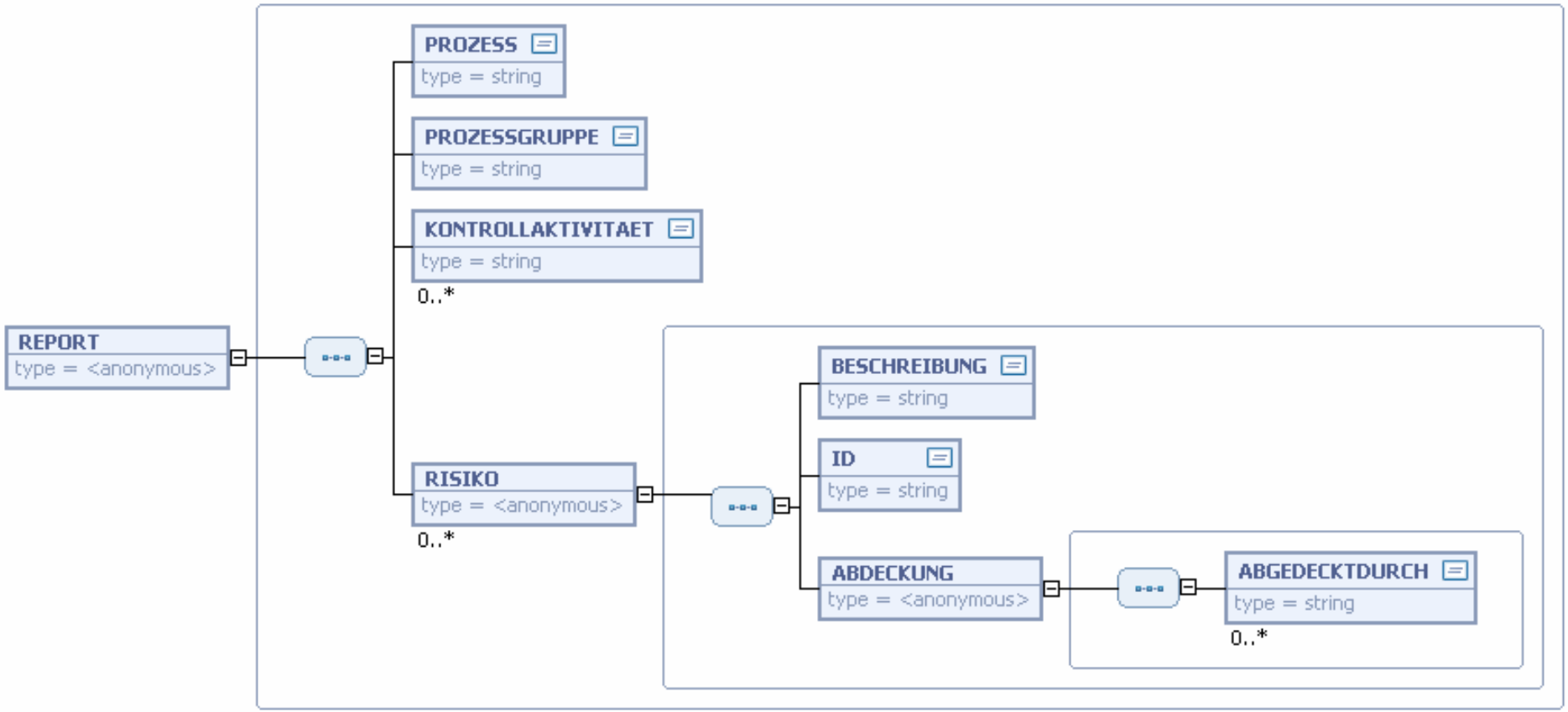

Abbildung 19: Grafische Darstellung des XML-Schemas

Es folgt nun der Quellcode des Schemas:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="REPORT">
    <complexType>
      <sequence>
        <element name="PROZESS" type="string" maxOccurs="1"
          minOccurs="1">
        </element>
        <element name="PROZESSGRUPPE" type="string"
          maxOccurs="1" minOccurs="1">
        </element>
        <element name="KONTROLLAKTIVITAET" type="string"
          minOccurs="0" maxOccurs="unbounded">
        </element>
        <element name="RISIKO" maxOccurs="unbounded"
          minOccurs="0">
          <complexType>
            <sequence>
              <element name="BESCHREIBUNG" type="string"
                maxOccurs="1" minOccurs="1">
              </element>
              <element name="ID" type="string" maxOccurs="1"
                minOccurs="1">
              </element>
              <element name="ABDECKUNG" maxOccurs="1"
                minOccurs="1">
                <complexType>
                  <sequence>
                    <element name="ABGEDECKTDURCH"
                      type="string" maxOccurs="unbounded"
                      minOccurs="0">
                    </element>
                  </sequence>
                </complexType>
              </element>
            </sequence>
          </complexType>
        </element>
      </sequence>
    </complexType>
  </element>
</schema>
```

In herkömmlicher Sprache bedeutet dies:

Jede Datei hat den Wurzelknoten <REPORT>. Dieser hat genau einen Kindknoten <PROZESS>, der den Namen des Prozesses enthält, für den der Report erstellt werden

soll, und genau einen Kindknoten <PROZESSGRUPPE> für die Beschreibung der Prozessgruppe, dem der Prozess zugeordnet ist. Weiterhin gibt es 0 bis n Kindknoten <KONTROLLAKTIVITAET> zur Speicherung des Names aller Kontrollaktivitäten, die dem Prozess zugeordnet sind. Als Letztes folgen 0 bis n <RISIKO>-Knoten, die jeweils genau einen <BESCHREIBUNG>-Knoten, einen <ID>-Knoten und einen <ABDECKUNG>-Knoten enthalten. <BESCHREIBUNG> enthält die Risikobeschreibung und <ID> die eindeutige Nummer eines Risikos. <ABDECKUNG> wiederum hat 0 bis n Kindknoten vom Typ <ABGEDECKTDURCH>. Jeder dieser Knoten enthält den Namen einer Kontrollaktivität, die dem Prozess zugeordnet ist und dieses Risiko abdeckt.

8.6 Ablauf der Transformation

Zur Verdeutlichung des Ablaufs der XSLT-Transformation dient das Flussdiagramm aus Abbildung 20. Die eigentliche XSL-Datei befindet sich im Ordner c:\fop und trägt den Namen report.xsl (der Inhalt dieser Datei ist im Anhang dieser Arbeit abgedruckt). Zur Syntax und zur Erstellung von XSL-Dateien finden sich im Internet zahlreiche Informationen (z. B. unter [W3C] oder [PIN]).

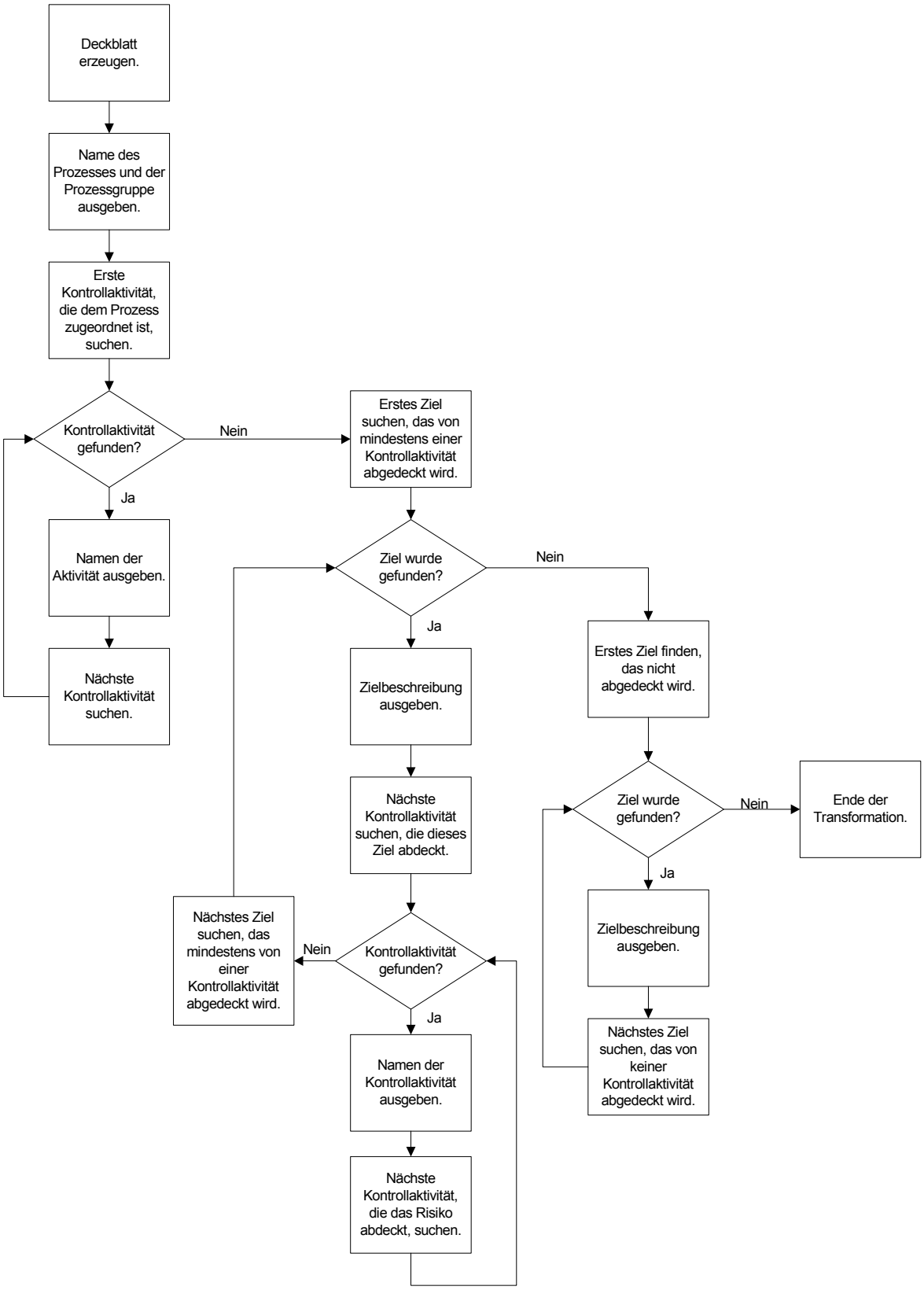

Abbildung 20: Ablaufdiagramm der XSLT-Transformation zur Reportgenerierung

8.7 Befüllung der Datenbank

Da das primäre Ziel dieses Lehrgangs die Softwareentwicklung mit Hilfe der EGL ist, wurde im VM-Ware-Image bereits eine leere Datenbank namens PROZESSE erzeugt. Um in dieser Datenbank die nötigen Tabellen zu erstellen und mit Daten zu füllen, enthält das Image zusätzlich ein Create-Skript. Dieses Skript legt die Tabellen mit Hilfe von Data Definition Language Statements an und fügt anschließend einige Datensätze über SQL-Befehle ein. Alle Tabellen werden im Schema TUTORIAL angelegt. Das Skript befindet sich in der Datei c:\Reporting\Create.sql. Zum Ausführen des Skripts wird die DB2-Steuerzentrale geöffnet (Windows Startmenü → Programme → IBM DB2 → DB2COPY1 (Standard) → Allgemeine Verwaltungstools → Steuerzentrale).

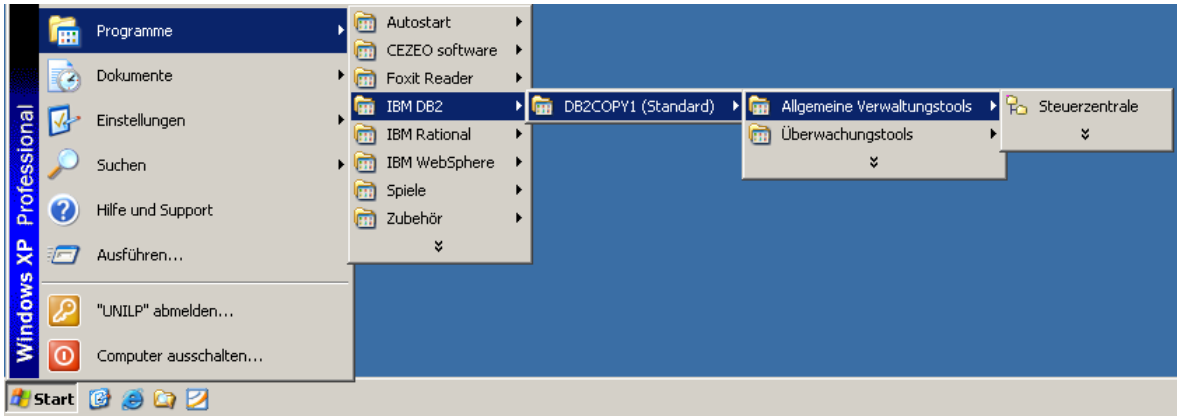

Abbildung 21: Start der DB2-Steuerzentrale

Nach dem Programmstart wird die erweiterte Sicht geöffnet und das Create-Skript ausgeführt. Dies geschieht durch eine Abfrage (Rechtsklick auf PROZESSE in der Baumansicht auf der linken Seite der Steuerzentrale → Abfragen…, siehe Abbildung Abbildung 22). Es öffnet sich nun ein neuer Bildschirm (siehe Abbildung 23).

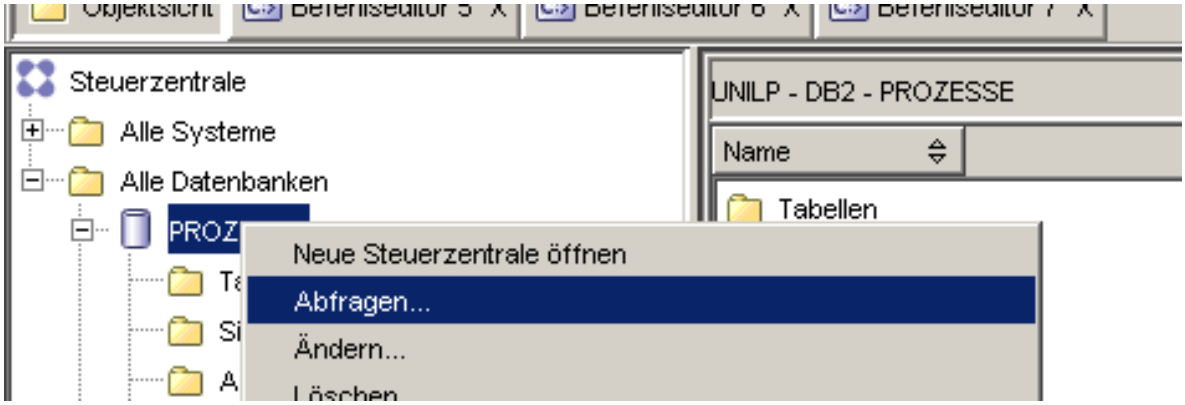

Abbildung 22: Öffnen des Abfrageeditors

Durch einen Klick auf das Ordnersymbol in der fünften Menüleiste von oben kann die Skriptdatei geöffnet werden (siehe Abbildung 23). Es werden jetzt die enthaltenen SQL-Anweisungen angezeigt. Zum Ausführen des Skripts wird der grüne Pfeil angeklickt.

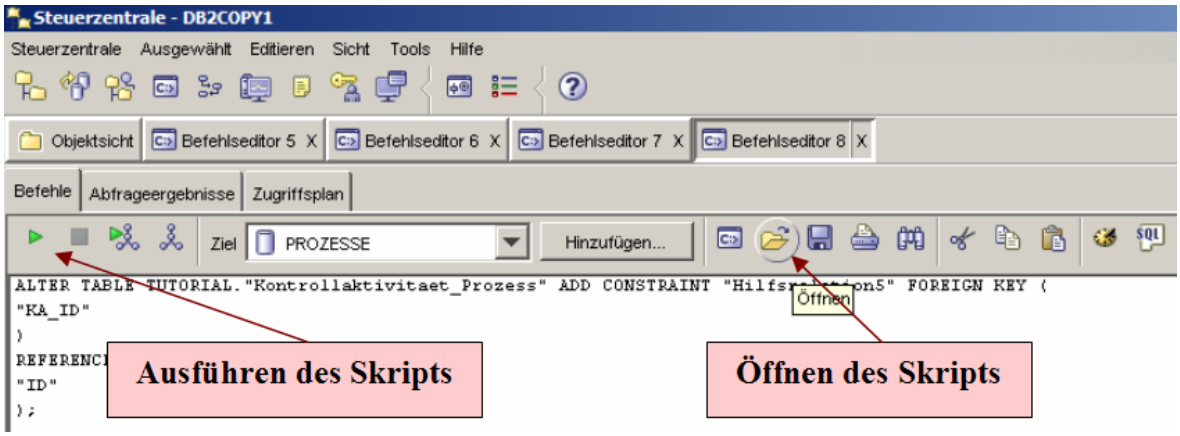

Abbildung 23: Abfrageeditor

Nach wenigen Sekunden sind die folgenden Tabellen erstellt (vgl. Kapitel 8.4 dieser Arbeit) und einige Datensätze eingefügt:

- Prozess
- Prozessgruppe
- Risiko
- Kontrollaktivität

- Prozessgruppe_Risiko (Zuordnungstabelle)
- Risiko_Kontrollaktivität (Zuordnungstabelle)
- Kontrollaktivität_Prozess (Zuordnungstabelle)

Weitere Informationen zu SQL und DB2 finden sich unter anderem in [WIN] und [DB2].

9 Erstellen der kompletten Logik mittels EGL

In diesem Abschnitt wird die konkrete Entwicklung einer Reportingkomponente beschrieben. Ziel der Komponente ist es, einen Report zu generieren, der für einen bestimmten Prozess angibt, ob alle Risiken des Prozesses abgedeckt sind, und durch welche Kontrollaktivitäten diese abgedeckt werden.

Die Logik wird dabei komplett mittels EGL realisiert. Der Aufbau des Programms ist prozedural.

Zunächst ist ein neues Projekt in Anlehnung an Kapitel 6 dieser Arbeit zu erstellen. Dieses Projekt sollte den Namen „ReportingEGL“ tragen.

9.1 Definieren von SQL-Records und passenden Libraries

Neben einfachen Datentypen, wie Int, Double oder String, können in der EGL zusammengesetzte Datentypen definiert werden. Diese werden Records genannt und sind mit Structs aus der Programmiersprache C vergleichbar. D. h. ein Record kann gleichzeitig Daten verschiedener Datentypen aufnehmen. Dies ist sinnvoll, da so zusammengehörige Daten in eine Einheit integriert werden können (z. B. Adressdaten).

Es folgt ein einfaches Beispiel für einen Record namens NameRecord, der die beiden Felder Vorname und Nachname besitzt, die jeweils den Datentyp string haben:

```
record NameRecord type Basicrecord
   Vorname string;
   Nachname string;
end
```

Das Schlüsselwort Basicrecord sagt aus, dass es sich um einen einfachen Record handelt, der an keine spezielle Funktion geknüpft ist. Ist ein solcher Record einmal definiert, können einzelne Instanzen, wie normale Variablen erzeugt werden.

Um eine Instanz mit Werten zu belegen bzw. um Werte abzurufen, wird (wie in vielen Programmiersprachen üblich) ein Punkt verwendet, der den Namen der Record-Instanz vom Namen des Kindelements trennt.

Beispiel:

```
EinName NameRecord; // Instanz erzeugen

EinName.Vorname = "Stefan"; // Instanz mit Werten belegen
EinName.Nachname = "Erras";
```

Neben den Basicrecords gibt es noch weitere Recordtypen, die zur Erfüllung bestimmter Aufgaben konzipiert wurden. Grundsätzlich wird hierbei zwischen Records mit fester und variabler Größe unterschieden. Bei Records mit fester Größe wird der Speicherplatzverbrauch bereits bei der Definition angegeben und kann während der Programmlaufzeit nicht variieren. Diese Typen werden hauptsächlich für den Zugriff auf serielle Datenbestände verwendet (z. B.: VSAM[10], DL/1[11]-Datenbanken, MQSeries Message Queues). Bei Records mit variabler Größe kann sich die Feldgröße der einzelnen Elemente während der Programmausführung ändern [IBM1].

Da in diesem Lehrgang lediglich Basic- und SQLRecords von Bedeutung sind, sei an dieser Stelle nur ein kurzer Überblick der verschiedenen Typen gegeben (die folgenden Angaben stammen aus [IBM1], dort sind auch weitere Informationen zu finden):

[10] Virtual Storage Access Method.

[11] Data Language/1.

Typ	Beschreibung
BasicRecord	Wird nur zur Verarbeitung innerhalb eines Programms verwendet und hat keine besondere Funktion.
DLISegment	Kann für den Zugriff auf eine DL/I-Datenbank verwendet werden.
IndexedRecord	Record mit fester Größe, mit dem auf indexsequentielle Dateien zugegriffen werden kann.
MQRecord	Dient dem Zugriff auf MQSeries Message Queues.
PSBRecord	Enthält Informationen, die benötigt werden, wenn ein EGL-Programm mit einer IMS[12]- oder DL/I-Datenbank kommunizieren muss.
RelativeRecord	Record mit fester Größe, der den Zugriff auf sequenzielle Daten ermöglicht, wobei alle Datensätze durch ihre Positionsnummer (Integer-Wert) ansprechbar sind.
SerialRecord	Wie der RelativeRecord, wird dieser für den Zugriff auf serielle Daten verwendet. Allerdings kann bei diesem Typ die Größe der einzelnen Datensätze variieren.
SQLRecord	Dient dem Zugriff auf eine relationale SQL-Datenbank.
VGUIRecord	Wird für den Datenaustausch zwischen einem Programm und einer Webseite verwendet.

Im Folgenden soll gezeigt werden, wie ein SQL-Record zum Zugriff auf eine relationale Datenbank genutzt werden kann, und welche Funktionen der WDz bietet, um die nötige Arbeit zu vereinfachen.

[12] Information Management System.

Zum Erstellen der Records und zugehöriger Libraries, wird File → New → Other → EGL → EGL Data Parts ausgewählt und Next angeklickt. Es öffnet sich ein Dialog, wie er in Abbildung 24 dargestellt ist.

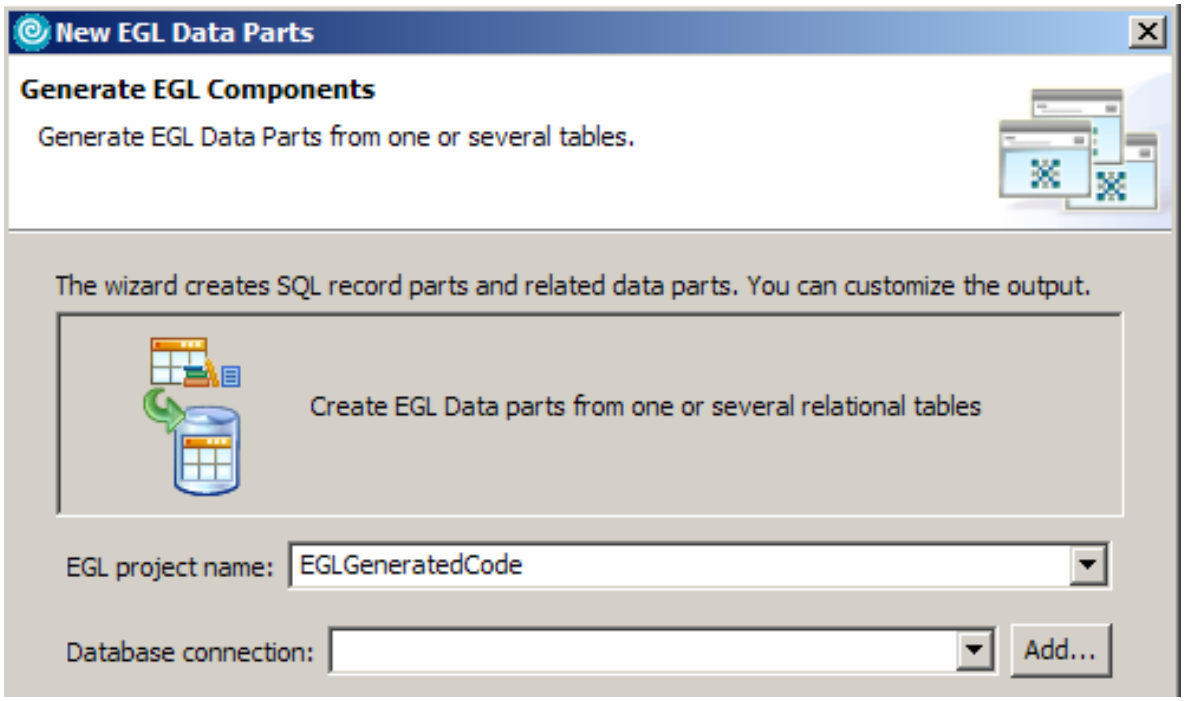

Abbildung 24: DataPart-Erstellung

Dort muss zunächst eine neue Datenbankverbindung erstellt werden (Klick auf Add…). Diese wird im folgenden Menü ReportingEGLCon genannt (Connection name). Außerdem wird die Option „Choose a database manager and JDBC driver“ ausgewählt und Next angeklickt. Die Einstellungen sind anschließend wie in Abbildung 25 zu wählen:

- JDBC driver: C:\Programme\IBM\SQLLIB\java\db2java.zip
- Alias: PROZESSE
- Class location: C:\Programme\IBM\SQLLIB\java\db2java.zip

Ein Klick auf Finish erstellt die Verbindung.

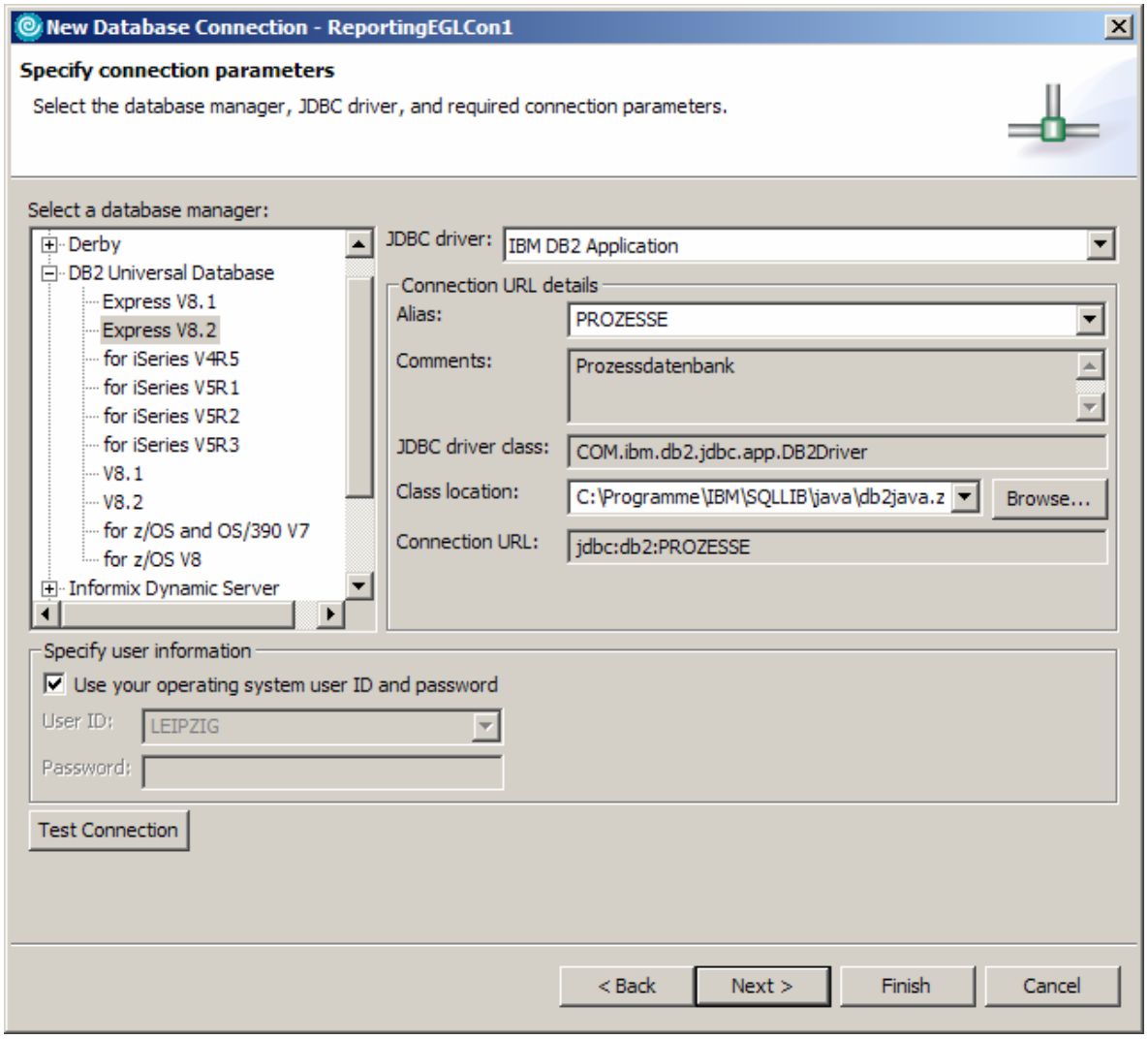

Abbildung 25: Neue Datenbank-Verbindung

Im Folgebildschirm können die Tabellen ausgewählt werden, für die SQLRecords und Libraries erstellt werden sollen. Es sind die Tabellen aus Kapitel 8.7 dieser Arbeit auszuwählen: Prozess, Prozessgruppe, Prozessgruppe_Risiko, Risiko, Risiko_Kontrollaktivität, Kontrollaktivität, Kontrollaktivität_Prozess (siehe Abbildung 26). Außerdem wird ein Haken bei „Record and library in same file“ gesetzt, bei EGL project name „ReportingEGL“ eingetragen und auf Finish geklickt.

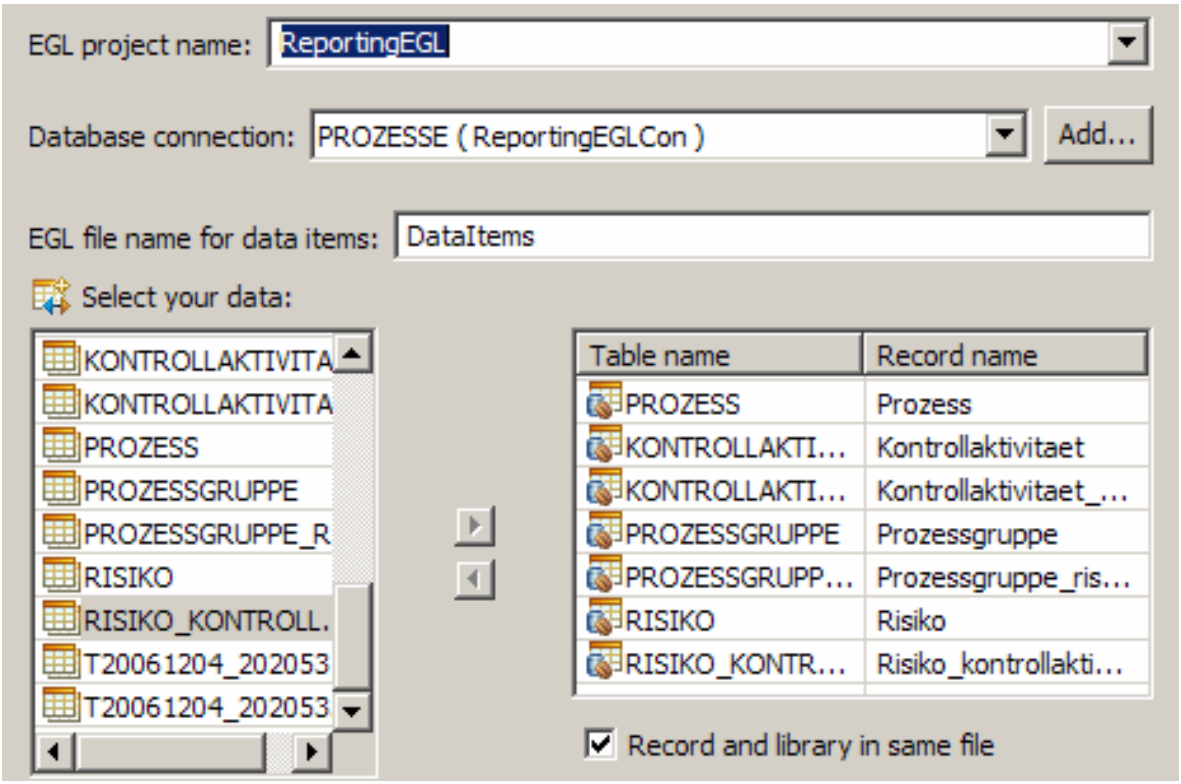

Abbildung 26: Auswahl der Tabellen

Nach kurzer Zeit erscheint eine Erfolgsmeldung auf dem Bildschirm. Das Projekt enthält nun die beiden neuen Packages „data“ und „libraries“ (im Project Explorer unter ReportingEGL im Unterordner EGLSource). Data enthält dabei Basisdatentypen und einen BasicRecord namens StatusRecord, der zur Übergabe von Statusinformationen, die bei den Datenbankzugriffen entstehen, verwendet wird. Im Package libraries hingegen sind die Records für die einzelnen Datenbanktabellen enthalten. Außerdem sind für jeden dieser Records Funktionen zum Einlesen, Ändern, Einfügen und Löschen von Datensätzen einer Tabelle vorhanden.

Der SQLRecord für die Prozess-Tabelle sieht beispielsweise so aus (Datei ProzessLibrary.egl):

```
record Prozess type SQLRecord { tableNames = [["TUTO-
RIAL.PROZESS"]], keyitems = ["pg_id", "id"] }
   pg_id PG_ID {column = "PG_ID"};
   name NAME {column = "NAME", sqlVariableLen = yes, maxlen = 100,
      isNullable = yes};
```

```
    id ID {column = "ID"};

    // used in arrays: index of the row
    indexInArray int {persistent = no};
end
```

Es sind hier Angaben über die verwendete Tabelle und die einzelnen Datenfelder enthalten.

Neben einem Record ist in jeder der erzeugten Quellcodedateien auch eine Library mit Funktionen vorhanden. In der Datei RisikoLibrary.egl sind dies z. B. readRisiko, readRisikoFromKeyRecord, createRisiko, deleteRisiko, updateRisiko und selectRisiko, die zu einer Library namens RisikoLibrary zusammengefasst sind. Der Beginn einer Library wird mit dem Schlüsselwort Library, gefolgt vom Namen der Library, eingeleitet. Danach folgen die zugehörigen Funktionsdefinitionen. Abgeschlossen wird die Library durch das Wort end.

Im Folgenden wird nun exemplarisch der Inhalt von RisikoLibrary erklärt.

Funktion readRisiko:

```
function readRisiko (risiko Risiko, sqlStatusData StatusRecord)

    try
        get risiko;
        sqlStatusData.sqlStatus = sysvar.sqlCode;
    onException
        sqlStatusData.sqlStatus = sysvar.sqlCode;
        sqlStatusData.description =
            syslib.currentException.description;
    end
end
```

Diese Funktion liest einen Datensatz mit einer bestimmten Identifikationsnummer (ID) aus der Datenbank. Der Parameter Risiko dient dabei sowohl zur Eingabe als auch zur Ausgabe. sqlStatusData hingegen ist ein reiner Ausgabeparameter, der nach Anweisungsausführung Statusinformationen der SQL-Abfrage enthält. Um einen Datensatz zu lesen, wird an

die Funktion ein gültiger Risiko-Record übergeben, bei dem das Attribut ID mit einem Wert vorbelegt ist. Die Funktion sucht bei ihrer Ausführung einen Datensatz, der diese ID hat und belegt die restlichen Felder des Records mit Daten.

Der Aufruf get innerhalb von readRisiko führt die eigentliche Datenbankabfrage durch. Die Angabe eines SQL-Statements wäre an dieser Stelle zwar möglich, ist aber unnötig, da dieses implizit bereits mit der Record-Definition festgelegt wurde.

Da sich die Funktion in einer Library befindet, muss bei einem Aufruf der Funktion (innerhalb eines Programms) der Name der Library mit angegeben werden. Dabei wird im Programm zunächst der Name der Library, gefolgt von einem Punkt und dem Namen der Funktion angegeben. Falls sich der Funktionsaufruf in einem anderen Paket befindet, muss zusätzlich ein entsprechendes import-Statement am Anfang des Progamms eingefügt werden.

Beispiel zur Verwendung der Funktion readRisiko:

```
import RisikoLibrary;

…

einRisiko Risiko;
sqlStatus sqlStatusData;
einRisiko.ID = 1;
RisikoLibrary.readRisiko(einRisiko, sqlStatus);
```

Die Funktion readRisikoFromKeyRecord erfüllt im Grunde den gleichen Zweck wie readRisiko, allerdings wird die ID bei dieser Funktion über einen separaten Parameter (RisikoKeys) übergeben. RisikoKeys hat hierbei einen eigenen Datentyp, der am Anfang der Datei definiert ist und nur zur Übergabe von Schlüsselfeldern dient.

```
function readRisikoFromKeyRecord (risiko Risiko, RisikoKeys
     risikoKeys, sqlStatusData StatusRecord)

         risiko.id = risikoKeys.id;

         readRisiko (risiko, sqlStatusData);

end
```

createRisiko erzeugt einen neuen Datensatz in der Datenbank, in dem ein gültiger Risiko-Record übergeben wird. Auch hier ist kein SQL-Statement notwendig, da dieses implizit definiert ist.

```
function createRisiko (risiko Risiko, sqlStatusData StatusRecord)

   try
         add risiko;
         sqlStatusData.sqlStatus = sysvar.sqlCode;
   onException
         sqlStatusData.sqlStatus = sysvar.sqlCode;
         sqlStatusData.description =
           syslib.currentException.description;
   end
end
```

deleteRisiko löscht einen Datensatz mit einer bestimmten ID, die wie bei readRisiko über einen Risiko-Record übergeben wird. Hier ist das SQL-Statement allerdings nicht implizit definiert, sondern wird explizit angegeben und mit dem Kommando execute ausgeführt. Ebenso verhält es sich bei der Funktion updateRisiko, mit der ein bestimmter Datensatz geändert werden kann.

```
function deleteRisiko (risiko Risiko, sqlStatusData StatusRecord)

   try
         execute #sql{
         DELETE FROM TUTORIAL.RISIKO WHERE RISIKO.ID = :risiko.id
         };
         sqlStatusData.sqlStatus = sysvar.sqlCode;
   onException
```

```
        sqlStatusData.sqlStatus = sysvar.sqlCode;
        sqlStatusData.description =
            syslib.currentException.description;
   end
end
```

Mit der Funktion selectRisiko kann der komplette Inhalt der Risiko-Tabelle in ein dynamisches Array gelesen werden. D. h. jedes Element des Arrays enthält nach dem Aufruf der Funktion einen Datensatz. Der Einsatz dieser Funktion ist nur bedingt empfehlenswert, da es vor allem bei großen Tabellen wenig sinnvoll ist, eine komplette Tabelle bedingungslos in den Hauptspeicher zu laden.

```
function selectRisiko (risiko Risiko[], sqlStatusData StatusRe-
cord)

   try
        get risiko;
        sqlStatusData.sqlStatus = sysvar.sqlCode;
   onException
        sqlStatusData.sqlStatus = sysvar.sqlCode;
        sqlStatusData.description =
            syslib.currentException.description;
   end
end
```

Im Gegensatz zu vielen anderen Programmiersprachen müssen Arrays in der EGL nicht mit einer bestimmten Größe dimensioniert werden. Elemente können einfach hinzugefügt oder entfernt werden. Eine Redimensionierung erfolgt automatisch.

Definiert werden Arrays wie folgt:

```
dynamischesArray Risiko[];
statischesArray Risiko[5];
```

Die eckigen Klammern zeigen hierbei an, dass es sich um ein Array handelt. Wird innerhalb der Klammern eine Zahl angegeben, handelt es sich um ein statisches Array (d. h. mit fester Elementanzahl). Wird die Zahl weggelassen, wird ein dynamisches Array erzeugt. Selbstverständlich ist auch die Definition mehrdimensionaler Arrays möglich. Da in diesem Beispiel nur eindimensionale Arrays benötigt werden, sei für nähere Informationen zur Definition an [IBM1] verwiesen. Die Verwendung der Arrays wird an späterer Stelle gezeigt.

Über updateRisiko kann ein Datensatz in der Datenbank aktualisiert werden. Hierzu wird eine Risiko-Instanz, die mit Daten gefüllt ist, an die Funktion übergeben. Diese Daten werden dann für das Update verwendet. Das Update funktioniert allerdings nur, wenn es bereits einen Datensatz mit der entsprechenden ID in der Datenbank gibt.

```
function updateRisiko (risiko Risiko, sqlStatusData StatusRecord)
   try
         execute #sql{
         UPDATE TUTORIAL.RISIKO SET BESCHREIBUNG =
             :risiko.beschreibung WHERE RISIKO.ID = :risiko.id
         };
         sqlStatusData.sqlStatus = sysvar.sqlCode;
   onException
         sqlStatusData.sqlStatus = sysvar.sqlCode;
         sqlStatusData.description =
             syslib.currentException.description;
   end
end
```

Die anderen Dateien im Paket libraries sind analog aufgebaut.

9.2 Implementierung eigener Funktionen

Da die generierten Lesefunktionen nicht für die Aufgabenstellung geeignet sind, müssen nun eigene Funktionen zum Einlesen der Daten geschrieben werden.

Grundsätzlich wird die Definition einer Funktion durch das Schlüsselwort function eingeleitet. Danach folgen der Name der Funktion und die Parameterliste samt Datentypen.

Nach der Parameterliste kann über das Schlüsselwort returns noch der Datentyp des Rückgabewerts angegeben werden (falls vorhanden).

Als erstes soll eine Funktion implementiert werden, die alle Kontrollaktivitäten, die einem Prozess zugeordnet sind, einliest.

Hierzu öffnet man die Datei KontrollaktivitaetLibrary.egl und fügt vor dem letzten Schlüsselwort end folgenden Code ein:

```
function selectKontrollaktivitaetFuerProzess(ka
     Kontrollaktivitaet[],PRid int, sqlStatusData StatusRecord)
   try

         get ka with
         #sql{
               select * from tutorial.Kontrollaktivitaet
               inner  join
               tutorial.Kontrollaktivitaet_prozess on
               tutorial.Kontrollaktivitaet_prozess.KA_ID =
               tutorial.Kontrollaktivitaet.ID
               where p_id = :PRid
         } ;

   onException
         sqlStatusData.sqlStatus = sysvar.sqlCode;
         sqlStatusData.description =
            syslib.currentException.description;
   end


end
```

Beim Parameter ka handelt es sich um ein dynamisches Array vom Record-Typ Kontrollaktivitaet. Über den zweiten Parameter wird die ID des Prozesses übergeben, dessen Kontrollaktivitäten gesucht werden. Mit dem dritten Parameter werden im Fehlerfall Statusinformationen zurückgegeben.

Der Aufruf von get besagt, dass das Array ka mit Hilfe des angegebenen SQL-Statements mit Daten befüllt werden soll. Das SQL-Statement führt hierbei einen Inner-Join zwischen den Tabellen Kontrollaktivitaet und der Zuordnungstabelle Kontrollaktivitaet_Prozess durch. Über die where-Klausel des Statements wird die Rückgabemenge auf alle Kontroll-

aktivitäten eingeschränkt, die über die Zuordnungstabelle mit dem gewünschten Prozess verknüpft sind. Da die ID des Suchprozesses in der Variablen PRid enthalten ist, muss sie in die where-Klausel integriert werden. Dies geschieht durch Angabe eines Doppelpunktes, gefolgt vom Namen der Variable.

Analog dazu müssen die folgenden Funktionen in die Dateien RisikoLibrary.egl eingefügt werden:

```
function selectRisikenFuerKontrollaktivitaet(risiko Risiko[],KAid
       int, sqlStatusData StatusRecord)
   try

         get risiko    with
               #sql{
                     select * from tutorial.risiko inner join
                     tutorial.risiko_Kontrollaktivitaet on
                     tutorial.risiko_Kontrollaktivitaet.R_ID =
                     tutorial.risiko.ID
                     where ka_id = :KAid
               } ;
   onException
         sqlStatusData.sqlStatus = sysvar.sqlCode;
         sqlStatusData.description =
            syslib.currentException.description;
   end


end

function selectRisikenFuerProzessgruppe(risiko Risiko[],PGid int,
   sqlStatusData StatusRecord)
   try
         get risiko    with
               #sql{
                     select * from tutorial.risiko inner join
                     tutorial.prozessgruppe_risiko on
                     tutorial.prozessgruppe_risiko.R_ID =
                     tutorial.risiko.ID
                     where pg_id = :PGid
                     } ;
   onException
         sqlStatusData.sqlStatus = sysvar.sqlCode;
         sqlStatusData.description =
            syslib.currentException.description;
   end

end
```

Diese Funktionen dienen dazu, alle Risiken einer Kontrollaktivität bzw. einer Prozessgruppe abzurufen.

Weiterhin wird eine Funktion benötigt, die einen oder mehrere Prozesse anhand des Prozessnamens einliest. Bei dieser Funktion wird ebenfalls ein dynamisches Array als Zieldatenstruktur verwendet, da es theoretisch vorkommen könnte, dass mehrere Prozesse mit dem gleichen Namen in der Datenbank abgelegt werden. Als zusätzliche Parameter werden Prozessname (name string) und ein StatusRecord übergeben.

```
function selectProzessByName (prozess Prozess[],name string in,
         sqlStatusData StatusRecord)

         try
               get prozess
                     with #sql{
                           select * from TUTORIAL.PROZESS where
                           NAME = :name
                     };
               sqlStatusData.sqlStatus = sysvar.sqlCode;
         onException
               sqlStatusData.sqlStatus = sysvar.sqlCode;
               sqlStatusData.description =
                  syslib.currentException.description;
         end
end
```

Diese Funktion wird ebenfalls oberhalb des letzten end-Schlüsselworts in die Datei ProzessLibrary.egl eingefügt.

Um den Programmierstil weiter zu verbessern, wird bei dieser Funktion der Parameter name zusätzlich mit dem Schlüsselwort „in“ versehen. Name kann so nur als Eingabeparameter verwendet werden. Im Normalfall sollten alle Parameter, die nicht zur Ausgabe verwendet werden, als Eingangsparameter deklariert werden, um ungewollte Veränderungen der übergebenen Variablen zu verhindern.

Will man eine Funktion A direkt mit dem Rückgabewert einer anderen Funktion B aufrufen, schreibt der Compiler vor, dass der entsprechende Eingabeparameter von A mit „in“ deklariert wird. Dies ist sinnvoll, da auf den Rückgabewert von B nach dem Funktionsaufruf von A nicht mehr zugegriffen werden kann.

```
function A(a int in) returns int
   return (a*a);
end

function B() returns int
   return 2;
end

…
// Der folgende Aufruf ist nur möglich, wenn der Parameter a der
// Funktion A als Eingabeparameter deklariert wurde.

A(B());
…
```

9.3 Implementierung des Hauptprogramms

Analog zu Kapitel 6 dieser Arbeit wird nun eine neue Programmdatei für das Hauptprogramm angelegt. Diese muss den Namen ReportingEGLMain tragen und sich im Default-Package befinden. In dieser Datei wird das eigentliche Programm abgelegt.

Ablauf des Programms:

1. Einlesen eines oder mehrerer Prozesse anhand des Namens. Der Prozessname wird dabei als Kommandozeilen-Parameter übergeben.

2. Einlesen der Prozessgruppe, zu der der Prozess gehört.

3. Einlesen aller Risiken, die zur Prozessgruppe gehören.

4. Einlesen aller Kontrollaktivitäten, die dem Prozess zugeordnet sind.

5. Einlesen aller Risiken, die von den einzelnen Kontrollaktivitäten abgedeckt werden. Dafür wird eine geeignete Datenstruktur erzeugt, die eine Kontrollaktivität und alle zugehörigen Risiken enthält.

6. Prüfung, ob die einzelnen Risiken der Prozessgruppe von den Kontrollaktivitäten abgedeckt werden (verschachtelte Schleife). In diesem Schritt wird gleichzeitig der XML-Code erzeugt.

7. Schreiben der XML-Datei.

8. Transformation der XML-Datei durch einen Kommandozeilenaufruf.

9. Programmende.

Für Punkt 5 ist eine Datenstruktur notwendig, die den Zusammenhang einer Kontrollaktivität und der zugehörigen Risiken darstellt. Diese Struktur wird in Form eines BasicRecords implementiert. Da die Anzahl der zugeordneten Risiken variabel ist, muss dieser Record ein dynamisches Array enthalten. Außerdem muss er einen Verweis auf eine Kontrollaktivität besitzen. Der Record ist im Folgenden definiert und in der Datei ReportingEGLMain oberhalb der Programmdefinition zu sehen.

```
record KARisiko type basicrecord
   ka Kontrollaktivitaet;
   risiko Risiko[];
end
```

Danach wird das eigentliche Hauptprogramm implementiert. Erklärungen zum Programm sind als Kommentare direkt im Quellcode enthalten.

Es folgt nun der Abdruck des kompletten Hauptprogramms. Der Inhalt der Datei ReportingEGLMain ist durch diesen zu ersetzen.

```
import libraries.*;
import data.*;

record KARisiko type basicrecord
   ka Kontrollaktivitaet;
   risiko Risiko[];
end


program ReportingEGLMain type BasicProgram


   function main()
         const SQLSTATUSOK int = 0; // alle SQL-Codes kleiner 0 sind Fehler

         Prozesse Prozess[]; // enthält die einzelnen Prozesse
         PG Prozessgruppe; // enthält die Prozessgruppe, die zum Prozess gehört
         PGRisiken Risiko[]; // Risiken, die zur Prozessgruppe gehören
         KAs Kontrollaktivitaet[];  // alle KAs eines Prozesses
         KAsUndRisiken KARisiko[]; // alle KAs + Risiken
         KAUndRisiken KARisiko; // eine KA und ihre Risiken
         Status      StatusRecord; // Statusdaten der Abfragen
         i int; // Zähler
         j int; // Zähler
         k int; // Zähler
         l int; // Zähler
         xml string; // String, welcher die XML-Struktur enthalten soll
         tempRisiken Risiko[]; //temporäres Array
         DateiInhalt clob; // Textdatentyp zur Ausgabe in eine Datei
         Filename String;

         // falls kein Parameter übergeben wurde, Programm verlassen.
         if (sysLib.getCmdLineArgCount() < 1)
               exit program;
         end
```

```
// Prozess einlesen (der zweite Parameter von selectProzessByName
// ist als Eingabeparameter deklariert und kann somit den Rückgabe-
// wert von getCmdLineArg direkt aufnehmen.)
ProzessLibrary.selectProzessByName(Prozesse,sysLib.getCmdLineArg(1), Status);

// falls ein Datenbankfehler aufgetreten ist, Meldung ausgeben und Programm beenden.
if (Status.sqlStatus < SQLSTATUSOK )
      sysLib.writeStderr("Fehler beim Datenbankzugriff: " + Status.description);
      exit program;
end

// alle gefundenen Prozesse durchlaufen
for (i from 1 to SysLib.size(Prozesse) by 1)

      // XML-String und Arrays leeren.
      xml = "";
      KAsUndRisiken.removeAll();
      KAs.removeAll();
      PGRisiken.removeAll();

      // Prozessgruppe einlesen
      PG.id = Prozesse[i].pg_id;
      ProzessgruppeLibrary.readProzessgruppe(PG, Status);

      // falls ein Datenbankfehler aufgetreten ist, Meldung ausgeben und Programm beenden.
      if (Status.sqlStatus < SQLSTATUSOK )
            sysLib.writeStderr("Fehler beim Datenbankzugriff: " + Status.description);
            exit program;
      end

      // Risiken für Prozessgruppe einlesen
      RisikoLibrary.selectRisikenFuerProzessgruppe(PGRisiken, PG.id, Status);

      // falls ein Datenbankfehler aufgetreten ist, Meldung ausgeben und Programm beenden.
```

```
if (Status.sqlStatus < SQLSTATUSOK )
      sysLib.writeStderr("Fehler beim Datenbankzugriff: " + Status.description);
      exit program;
end

// Kontrollaktivitäten für Prozess einlesen
KontrollaktivitaetLibrary.selectKontrollaktivitaetFuerProzess(KAs, Prozesse[i].id, Status);

// falls ein Datenbankfehler aufgetreten ist, Meldung ausgeben und Programm beenden.
if (Status.sqlStatus < SQLSTATUSOK )
      sysLib.writeStderr("Fehler beim Datenbankzugriff: " + Status.description);
      exit program;
end

// für jede Kontrollaktivität Risiken einlesen und in Datenstruktur übertragen.
for (j from 1 to SysLib.size(KAs) by 1)
      RisikoLibrary.selectRisikenFuerKontrollaktivitaet(KAUndRisiken.risiko,
                              KAs[j].id, Status);

      // falls ein Datenbankfehler aufgetreten ist, Meldung ausgeben und Programm beenden.
      if (Status.sqlStatus < SQLSTATUSOK )
            sysLib.writeStderr("Fehler beim Datenbankzugriff: " + Status.description);
            exit program;
      end

      KAUndRisiken.ka = KAs[j];
      KAsUndRisiken.appendElement(KAUndRisiken);
end

// XML beginnen.
xml = "<?xml version='1.0' encoding='iso-8859-15' ?><REPORT>";

xml = xml + "<PROZESS>" + Prozesse[i].name + "</PROZESS>";
```

```
xml = xml + "<PROZESSGRUPPE>" + PG.beschreibung + "</PROZESSGRUPPE>";

// alle zugeordnete KAs ausgeben.
for (j from 1 to SysLib.size(KAsUndRisiken) by 1)
      xml = xml + "<KONTROLLAKTIVITAET>" + KAsUndRisiken[j].ka.beschreibung +
                  "</KONTROLLAKTIVITAET>";
end

// Abdeckung für jedes Risiko der Prozessgruppe prüfen
for (j from 1 to SysLib.size(PGRisiken) by 1)

      // Informationen zum Risiko in XML schreiben.
      xml = xml + "<RISIKO>";
      xml = xml + "<BESCHREIBUNG>" + PGRisiken[j].beschreibung
            + "</BESCHREIBUNG>";
      xml = xml + "<ID>" + PGRisiken[j].id + "</ID>";
      xml = xml + "<ABDECKUNG>";

      // Abdeckung prüfen:

      // Array der KAs durchlaufen
      for (k from 1 to SysLib.size(KAsUndRisiken) by 1)

            // Risiken der aktuellen KA in Temp-Array übertragen.
            tempRisiken  = KAsUndRisiken[k].risiko;

            // für jedes Risiko der KA die Abdeckung prüfen
            for (l from 1 to SysLib.size(tempRisiken) by 1 )

                  // falls eine Übereinstimmung gefunden wurde,
                  // den Namen der KA, die das Risiko abdeckt in
                  // das XML-Dokument schreiben und Schleife mit
                  // exit for verlassen.
```

```
                        if (PGRisiken[j].id == tempRisiken[l].id)
                              xml = xml + "<ABGEDECKTDURCH>";
                              xml = xml + KAsUndRisiken[k].ka.beschreibung;
                              xml = xml + "</ABGEDECKTDURCH>";
                              exit for;
                        end
                  end

            end
            xml = xml + "</ABDECKUNG>";
            xml = xml + "</RISIKO>";
      end

      xml = xml + "</REPORT>";

      // Inhalt des XML-String in Clob-Datentyp übertragen.
      // Dies ist nötig, da nur der Inhalt eines Clobs,
      // in eine Text-Date geschrieben werden kann.
      loblib.setClobFromString(DateiInhalt, xml);

      // Pfad + Dateinamen erzeugen (ohne Endung).
      // Um das Programm nicht unnötig zu verkomplizieren,
      // wird der Ausgabepfad hart kodiert.

      Filename = "c:\\reporting\\" + Prozesse[i].name + "_" + i ;

      // Clob auf die Festplatte schreiben.
      LobLib.updateClobToFile(
      DateiInhalt,
        Filename +".xml");

      // FOP aufrufen
      sysLib.callCmd("c:\\fop\\fop.bat -xsl c:\\fop\\report.xsl -xml \"" +
            Filename + ".xml\" -pdf \"" + Filename +".pdf\"" );
```

```
        end
    end
end
```

Im nachfolgenden Abschnitt sind die wichtigsten neuen Konstrukte erklärt:

- **Einlesen von Kommandozeilen-Parametern:** Um Parameter, die beim Aufruf des Programms übergeben wurden, zu verarbeiten, werden die Funktionen sysLib.getCmdLineArgCount() und sysLib.getCmdLineArg(index) verwendet. Die erste der beiden Funktionen gibt dabei die Anzahl der Parameter zurück. Über die zweite Funktion können die einzelnen Parameter als String abgerufen werden. Diese sind durchnummeriert, wobei index die Position (beginnend mit 1) angibt.

  Das Aufnehmen von Kommandozeilen-Parametern ist u. a. in Modernisierungsszenarien von großer Bedeutung. Man kann so ein bestehendes Programm, das über einen Systemaufruf gestartet wird, durch ein neues EGL-Programm ersetzen. Dazu ist lediglich eine Nachbildung der evtl. vorhandenen Parameterstruktur des Vorgängers nötig.

- **(Vorzeitiges) Beenden eines Programms:** Um ein EGL-Programm (vorzeitig) zu beenden, wird der Aufruf exit program verwendet.

- **Datentyp Clob:** Dieser Datentyp dient dazu, Textdateien variabler Größe zu verarbeiten. Hierzu stehen in der Bibliothek lobLib einige Funktionen zur Verfügung. Um einen String in eine Datei zu schreiben, muss dieser String zuerst in einen Clob konvertiert werden. Dies geschieht über den Aufruf Loblib.setClobFromString(clob, string);, welcher den Ziel-Clob als ersten und den neuen Inhalt als zweiten Parameter übernimmt. Danach wird der Clob über LobLib.updateClobToFile(clob, dateinameString); in eine Textdatei geschrieben. Update bedeutet dabei, dass die Datei überschrieben wird, falls sie bereits vorhanden ist. Sollte die Datei noch nicht vorhanden sein, wird sie neu erstellt. Nähere Informationen finden sich in [IBM1] oder in der Onlinehilfe des WDz. Außerdem befinden sich dort auch Informationen zum Datentyp Blob, der eine ähnliche Aufgabe für Binärdaten übernimmt. Wegen dieser Ähnlichkeit zwischen Blob und Clob finden sich in der lobLib ebenfalls Funktionen zum Verarbeiten von Blobs (daher dürfte auch der Name lobLib stammen).

- **(Dynamische) Arrays:** Anders als bei vielen anderen Programmiersprachen, bringt ein EGL-Array eigene Methoden mit sich, die verwendet werden können, um Array zu manipulieren oder um Informationen über ein Array abzufragen. Die folgende Tabelle erläutert die verfügbaren Methoden [IBM1]:

Methode	Funktion
appendAll(appendArray)	Fügt alle Elemente des Übergebenen Arrays am Ende des aktuellen Arrays ein.
appendElement(element)	Fügt das übergebene Element am Ende des Arrays ein.
getMaxSize()	Gibt die Maximalgröße des Arrays zurück.
getSize()	Gibt die Anzahl der Elemente des Arrays zurück. Alternativ kann hierfür auch die Funktion sysLib.size(array) verwendet werden, der das zu vermessende Array als Parameter übergeben wird.
insertElement(element, index)	Fügt das übergebene Element an der angegebenen Stelle ein.
removeAll()	Löscht alle Elemente des Arrays.
removeElement(index)	Löscht das Element an der übergebenen Indexposition.
resize(size)	Ändert die Größe des Arrays.
resizeAll(size)	Ändert die Größe aller Dimensionen eines mehrdimensionalen Arrays.

setMaxSize(size)	Ändert die Maximalgröße eines Arrays.
setMaxSizes(size)	Ändert die Maximalgröße aller Dimensionen eines Mehrdimensionalen Arrays.

In anderen Sprachen müssen für obige Funktionen häufig spezielle Objekte, wie Collections, verwendet werden.

Leider gibt es im Zusammenhang mit Arrays eine Ungereimtheit im Sprachumfang der EGL: Zum Einlesen einer Menge von Datensätzen aus einer Datenbank kann die ForEach-Schleife verwendet werden. Zum Durchlaufen aller Array-Elemente ist dies aber nicht möglich. Da die ForEach-Schleife genau diese Funktion in anderen Sprachen übernimmt (z. B. C#, Visual Basic, PHP), könnte diese Einschränkung bei manchen Entwicklern im ersten Moment für Verwirrung sorgen. Nähere Informationen zu ForEach in [IBM1].

- **Verknüpfung von Strings:** Um mehrere Strings miteinander zu verknüpfen, wird der +-Operator verwendet. Zwei Strings werden dadurch in der angegebenen Reihenfolge aneinandergehängt.

- **Ausführung eines Systemkommandos:** Über sysLib.callCmd() kann ein Systemaufruf abgesetzt werden, also ein Befehl, der auch auf der Kommandozeile des jeweiligen Betriebssystems ausgeführt werden könnte. Die Ausführung wird synchron gestartet. D. h., das aufrufende EGL-Programm führt seine nächste Anweisung erst aus, wenn der gestartete Befehl die Kontrolle zurückgibt. Im konkreten Fall wird das Open-Source-Java-Programm FOP von der Apache Foundation gestartet. FOP ist in der Lage, eine XML-Datei mittels XSL-FO-Transformation in ein anderes (Ausgabe-)Format umzuwandeln. Das Programm benötigt hierzu mindestens drei Parameter: den Pfad zu einer XSL-Datei, die die Transformation steuert, den Pfad zu einer XML-Datei, welche die Nutzdaten enthält und das Ausgabeformat inkl. Pfad zur Ausgabedatei (hier Portable Document Format (PDF)). Für nähere Informationen zu XSL-FO siehe [W3C]. Die Dokumentation zu FOP findet sich im Internet unter [FOP].

Um ein Programm asynchron zu starten, wird der Befehl SysLib.startCmd() verwendet. Dieser ist allerdings nur in Java-Umgebungen verfügbar [IBM1].

Wie das Aufnehmen von Kommandozeilen-Parametern ist auch das Ausführen von Systemkommandos wichtig für die Modernisierung und Erweiterung bestehender Anwendungen, da vorhandene Programme einfach über einen Systemaufruf gestartet werden können. Die Kopplung zwischen beiden Programmen bleibt dabei auf niedrigem Niveau. Der Aufrufer muss lediglich über die Parameterstruktur des aufgerufenen Programms Bescheid wissen. Andere Aspekte wie Programmiersprache, Programmversion usw. spielen keine Rolle.

Zur weiteren Verringerung der Kopplung kann anstelle eines direkten Programmaufrufs auch eine Skript-Datei aufgerufen werden, die dann den Programmaufruf übernimmt. Sollten später Änderungen nötig werden, bzw. Zwischenschritte eingefügt werden müssen, lässt sich eine solche Skript-Datei schnell ändern oder erweitern.

9.4 Konfiguration des Build-Descriptors

Um das fertige Programm nun im Debugger starten zu können, muss zunächst der Build-Descriptor (ReportingEGL.eglbld) konfiguriert werden. Da in diesem Beispiel eine SQL-Datenbank verwendet werden soll, sind einige Attribute mit den Werten aus Abbildung 27 zu belegen.

sessionBeanID	(no value set)
sqlCommitControl	(no value set)
sqlDB	jdbc:db2://localhost:50000/PROZESSE
sqlID	TEST
sqlJDBCDriverClass	com.ibm.db2.jcc.DB2Driver
sqlJNDIName	(no value set)
sqlPassword	test
sqlValidationConnection...	(no value set)
synchOnTrxTransfer	(no value set)

Abbildung 27: Build-Descriptor

9.5 Konfiguration des Debuggers

Der Debugger für das neue Programm wird analog zu Kapitel 6 dieser Arbeit konfiguriert. Allerdings muss für einen sinnvollen Aufruf des Programms ein Kommandozeilen-Parameter an das EGL-Programm übergeben werden, der den Namen des gewünschten Prozesses enthält. Dies geschieht über den Karteireiter Arguments im Konfigurationsmenü des Debuggers. Dieses Argument wird später bei jedem Debuggeraufruf an das EGL-Programm übergeben.

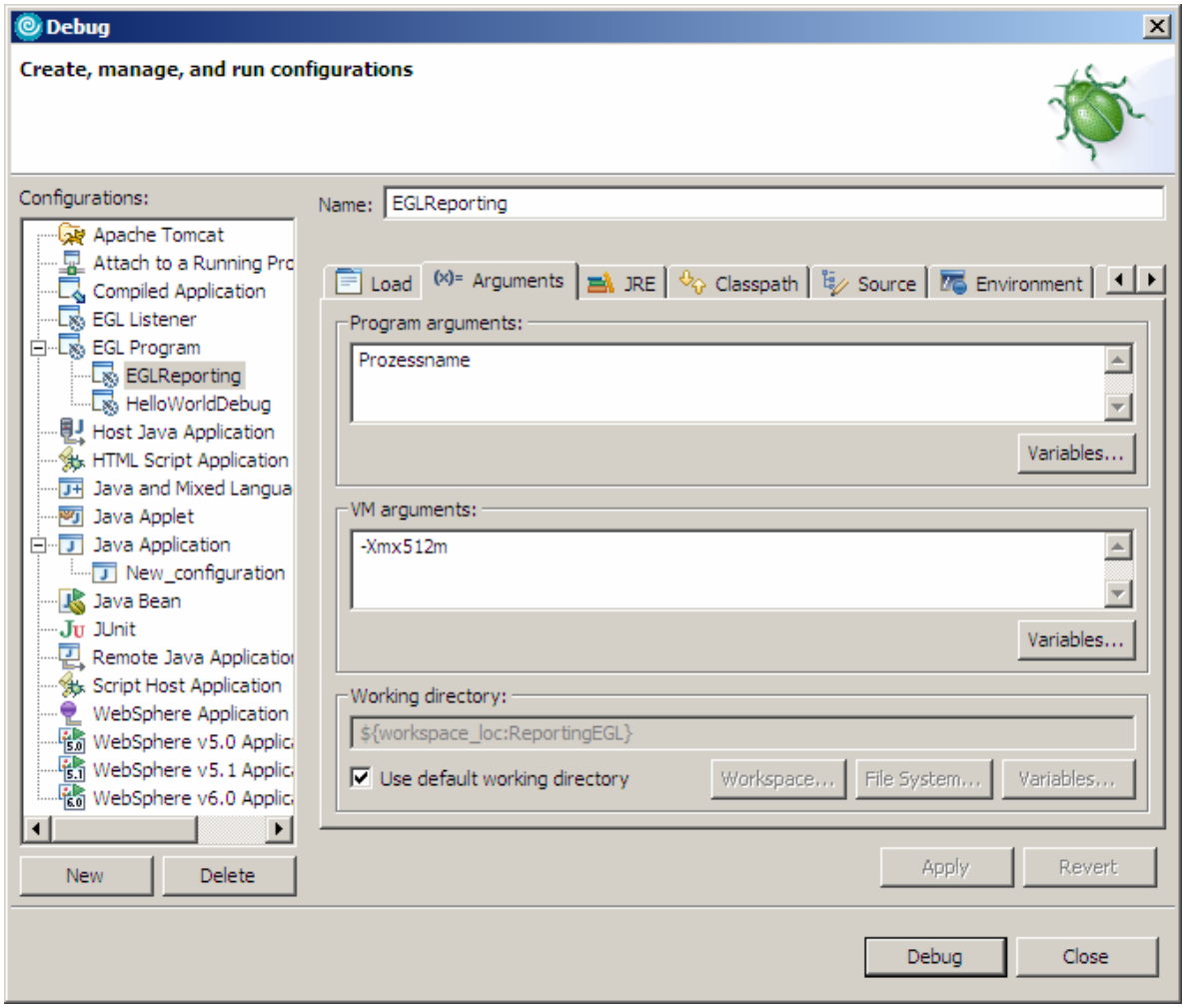

Abbildung 28: Kommandozeilen-Parameter übergeben

Weil in der Datenbank nur ein Prozess hinterlegt ist, muss als Argument der Name dieses Prozesses angegeben werden: „Rechnungsstellung Privatkunden“ (umschlossen von Anführungszeichen). Falls ein falscher Prozessname übergeben wird, endet das Programm

bereits nach der ersten Datenbankabfrage. Sollte bei der Ausführung ein Datenbankfehler auftreten, ist es ratsam als erstes die Einstellungen des Build-Descriptors zu überprüfen.

9.6 Ergebnis der Programmausführung

Nach einem kompletten Programmdurchlauf befinden sich im Verzeichnis c:\Reporting zwei neue Dateien: „Rechnungsstellung Privatkunden_1.pdf“ und „Rechnungsstellung Privatkunden_1.xml“. In den Abbildung 29 und Abbildung 30 ist die PDF-Datei dargestellt.

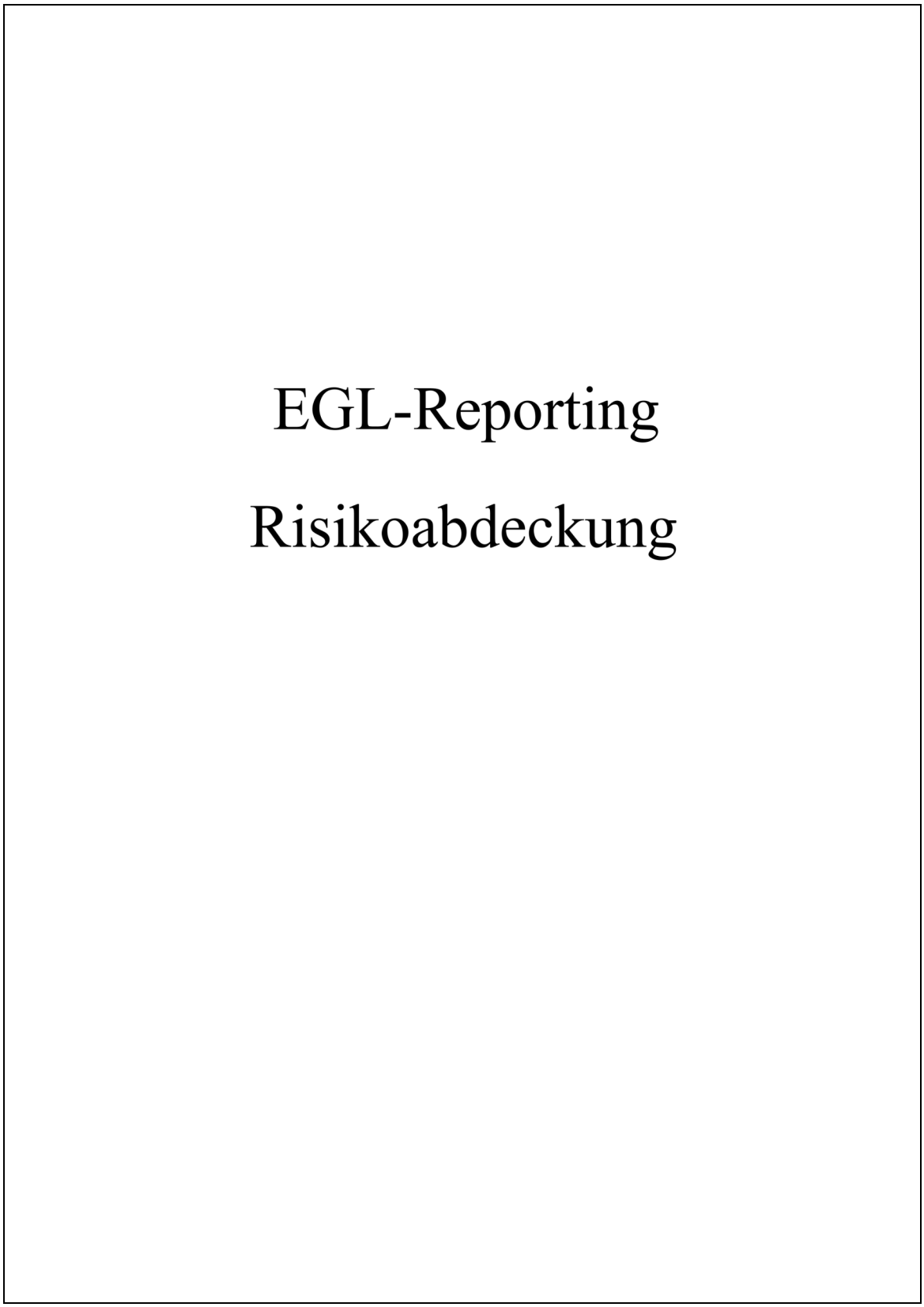

Abbildung 29: Deckblatt des fertigen Reports

EGL-Reporting

1. Prozessinformationen

Prozessname	Rechnungsstellung Privatkunden
Prozessgruppe	Rechnungsstellung

2. Zugeordnete Kontrollaktivitäten

Prüfung des Zahlungseingangs.
Prüfung der Rechnungsdaten.

3. Abdeckung der Risiken

3.1 Abgedeckte Risiken

Risiko	Kontrollaktivität
Rechnung enthält falsche Posten.	Prüfung der Rechnungsdaten.
Rechnung enthält falsche Kundendaten.	Prüfung der Rechnungsdaten.
Rechnung wird nicht korrekt bezahlt.	Prüfung des Zahlungseingangs.

3.2 Nicht abgedeckte Risiken

Rechnung wird nicht verschickt.

Universität Leipzig Seite 2

Abbildung 30: Inhalt des Reports

Wie in Abbildung 30 zu sehen ist, sind dem Prozess „Rechnungsstellung Privatkunden„ zwei Kontrollaktivitäten zugeordnet: „Prüfung des Zahlungseingangs“ und „Prüfung der Rechnungsdaten“. Weiterhin erbt der Prozess von der Prozessgruppe Rechnungsstellung folgende Risken:

- Rechnung wird nicht korrekt bezahlt.
- Rechnung enthält falsche Kundendaten.
- Rechnung enthält falsche Posten.
- Rechnung wird nicht verschickt.

Von den oben genannten Risken werden drei durch Kontrollaktivitäten abgedeckt.

Dass diese Angaben korrekt sind, kann durch schrittweises Ausführen des Programms im Debugger oder durch entsprechende Datenbankabfragen nachgeprüft werden.

10 Erstellung der Logik unter Verwendung von Java-Objekten

Wie bereits erwähnt, ist die Verwendung von bestehenden Java-Objekten häufig notwendig oder vorteilhaft. In diesem Kapitel wird deshalb aufgezeigt, wie das Ergebnis von Kapitel 9 dieser Arbeit mit Hilfe vorhandener Java-Objekte erzeugt werden kann.

10.1 Möglichkeiten der Java-Integration

Aus Sicht eines Entwicklers erfolgt die Java-Integration in der EGL über ein Schichtenmodell (Java- und EGL-Schicht). Hierbei läuft das EGL-Programm in der EGL-Schicht und kommuniziert mit Objekten und Klassen, die sich in der Java-Schicht befinden. Diese Kommunikation erfolgt über eine gemeinsame Schnittstelle. Da Java-Komponenten innerhalb eines EGL-Programms verwendet werden, wird diese Kommunikation durch EGL-Code eingeleitet.

Abbildung 31 verdeutlicht dies: Ein EGL-Programm schickt ein Kommando zur Erzeugung eines neuen Objektes A an die Java-Schicht und ruft im späteren Programmverlauf Methoden von A auf. Evtl. vorhandene Rückgabewerte dieser Methodenaufrufe werden an das EGL-Programm übermittelt oder ebenfalls in der Java-Schicht abgelegt. Letzteres ist der Fall, wenn eine Java-Methode ein weiteres Java-Objekt erzeugt (in der Grafik wird Objekt B von A erzeugt).

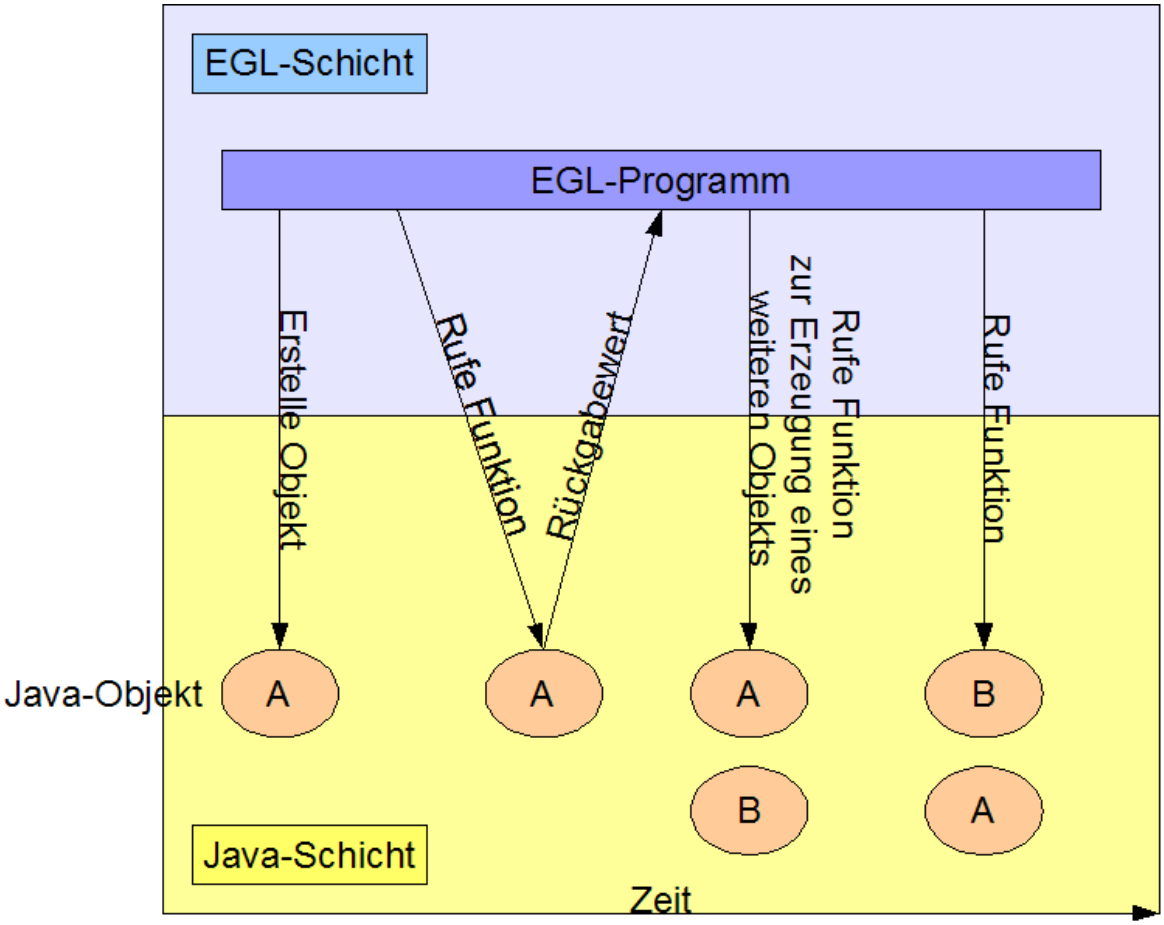

Abbildung 31: Trennung von EGL- und Java-Schicht

Die EGL bietet grundsätzlich zwei Wege, um die Kommunikation zwischen beiden Schichten durchzuführen: Interfaces und die Bibliothek JavaLib.

10.2 Kommunikation mittels Interface

Ein Interface ist eine Art Wrapper, welcher eine Java-Klasse auf die EGL-Welt abbildet. Ein solches Interface ähnelt einem EGL-Record, der neben der reinen Datenspeicherung zusätzlich auch andere Aufgaben in Form von Funktionen bzw. Methoden ausführen kann. Diese Methoden werden nicht vom Interface selbst implementiert, sondern z. B. von einer Java-Klasse.

Die Verwendung von Interfaces bietet den Vorteil, dass der Einsatz von Java-Objekten für den Entwickler vollständig transparent abläuft, sobald das Interface einmal implementiert ist, weil bei der Benutzung von Intefaces die gewohnte EGL-Syntax verwendet wird.

In der aktuellen Version der EGL erkauft man sich diesen Vorteil allerdings durch zwei Nachteile:

- Eine Java-Klasse kann nicht direkt über ein Interface instanziert werden, was dazu führt, dass zusätzlicher Java-Code implementiert werden muss: Zum Erzeugen einer neuen Instanz muss auf Java-Seite eine statische Methode erstellt werden, die eine Instanz einer Klasse zurückliefert (Fabrikmethode). Diese Methode kann dann ebenfalls in ein EGL-Interface aufgenommen und dort zur Instanzierung von Java-Objekten verwendet werden.

- Es können keine Interface-Arrays gebildet werden. Somit können auch keine Java-Methoden aufgerufen werden, die Objekt-Arrays als Parameter oder Rückgabewert verwenden. Dies macht die Verwendung von Interfaces im Beispiel der Reporting-komponente unmöglich, da hier unter anderem die Klasse Prozess implementiert wurde, welche eine Methode enthält, die ein Array aller Kontrollaktivitäten, die dem Prozess zugeordnet sind, zurückgibt (siehe Kapitel 10.4 dieser Arbeit).

Beispiel zur Definition eines Interfaces:

```
interface HelloWorld type JavaObject
  {javaName = "HelloWorld",
   packageName = "eu.erras.beispiel"}

  function sayHello() returns (String);
end
```

Zugehöriger Java-Code:

```
package eu.erras.beispiel;

  class HelloWorld
  {
    public String sayHello()
     {
       return "Hello World";
     }
  }
```

Obiges Beispiel erzeugt ein EGL-Interface für eine Java-Klasse namens HelloWorld, die sich im Paket eu.erras.beispiel befindet und eine einzige Methode besitzt, die einen String zurückliefert. Wie bereits erwähnt, besteht bei der Verwendung von Interfaces keine Möglichkeit, ein neues Java-Objekt zu erzeugen. Aus diesem Grund muss zusätzlicher Java-Code geschrieben werden, über den Instanzen erzeugt werden. Dies geschieht durch Implementierung einer statischen Methode. Falls der Source-Code der zu instanzierenden Klasse zur Verfügung steht, kann diese Methode direkt als Teil der Klasse implementiert werden. Sollte dies nicht der Fall sein, kann für diesen Zweck auch eine zusätzliche Klasse mit entsprechendem EGL-Interface angelegt werden. Der folgende Code zeigt, wie die Beispielklasse und das entsprechende Interface erweitert und verwendet werden können.

Erweiterung der Java-Klasse um eine statische Create-Methode:

```
package eu.erras.beispiel;

  class HelloWorld
  {
    public String sayHello()
     {
         return "Hello World";
     }
    public static HelloWorld create()
     {
         return new HelloWorld();
     }
   }
```

Erweiterung des EGL-Interfaces:

```
interface HelloWorld type JavaObject
  {javaName = "HelloWorld",
   packageName = "eu.erras.beispiel"}

  function sayHello() returns (string);
  static function create() returns (HelloWorld);
end
```

Verwendung des Interfaces in einem EGL-Programm:

```
program HelloWorld type BasicProgram

    function main()

      hello HelloWorld;

      hello = HelloWorld.create();

      syslib.writeStdout(hello.sayHello());

    end
end
```

10.3 Kommunikation über die JavaLib

Neben der Verwendung von Interfaces bietet die EGL in Form der JavaLib noch eine weitere Art zur Java-Integration. Die JavaLib enthält eine Reihe von Funktionen, mit denen Java-Objekte erzeugt, manipuliert und evtl. vorhandene Rückgabewerte abgefragt werden können. Die beiden genannten Einschränkungen der Verwendung von Interfaces treten hierbei nicht auf. Allerdings geht für den Entwickler ein Großteil der Transparenz verloren, da Java-Objekte in diesem Fall nicht über die gewohnte EGL-Syntax angesprochen werden. Stattdessen stellt die JavaLib eine Reihe von Funktionen bereit, die die Kommunikation zwischen den beiden Schichten über Strings ermöglichen. Diese Funktionen sind im Folgenden beschrieben [IBM1].

Hinweis zum Verständnis: Alle Instanzen von Java-Klassen, auf die per EGL zugegriffen wird, erhalten einen eindeutigen Identifizierer. Dies ist ein String, der vom Programmierer frei gewählt werden kann, sobald ein Objekt erzeugt wird. Bei allen folgenden Operationen, die im Zusammenhang mit einem Objekt stehen, muss dieser Identifizierer angegeben werden, um das entsprechende Objekt zu identifizieren. Obwohl es sich um einen String handelt, muss der Identifizierer bei jeder Verwendung in den Typ objId gecastet werden (siehe Beispiel). Wenn kein Objekt sondern eine Klasse angesprochen werden soll (z. B. um eine statische Methode aufzurufen), darf dieser Cast nicht durchgeführt werden. Stattdessen wird der Klassenname direkt als String übergeben.

Hinweis zur Notation: Optionale Funktionsparameter werden in eckige Klammern eingeschlossen. Parameter, die mehrfach übergeben werden können, sind durch ein Sternchen am Ende des Namens gekennzeichnet.

- **JavaLib.getField(string IdentifiziererOderKlassenname, string Feldbezeichnung):** gibt den Wert eines Feldes des angegebenen Objekts bzw. der angegebenen Klasse zurück. Ein Klassenname wird angegeben, wenn es sich um ein statisches Feld handelt. Soll das Feld eines bestimmten Objektes ausgelesen werden, wird der entsprechende Identifizierer übergeben.

- **JavaLib.invoke(string IdentifiziererOderKlassenname, string Methodenname [,Argument(e)*]):** führt die angegebene Methode mit den evtl. übergebenen Parametern aus.

- **JavaLib.isNull(string Identifizierer):** prüft, ob der übergebene Identifizierer auf eine Null-Referenz verweist.

- **JavaLib.isObjID(string Identifizierer):** prüft, ob der übergebene Identifizierer bereits besteht.

- **JavaLib.qualifiedTypeName(Identifizierer):** gibt den vollständig qualifizierten Klassen-Namen des Objektes, das sich hinter dem Identifizierer verbirgt, zurück.

- **JavaLib.remove(string Identifizierer):** löscht den Identifizierer und das evtl. mit ihm verbundene Java-Objekt. Dieser Befehl sollte immer ausgeführt werden, wenn

ein Java-Objekt nicht länger benötigt wird, da die automatische Speicherbereinigung der Java-Virtual-Machine sonst nicht arbeiten kann.

- **JavaLib.removeAll():** gibt alle Identifizierer und Objekte frei.
- **JavaLib.setField(string IdentifiziererOderKlassenname, string Fehlname, Wert):** setzt den Wert des angegebenen Feldes.
- **JavaLib.store(string ZielIdentifizierer, string IdentifiziererOderKlassenname, string Methodenname [,Argument(e)*]):** führt eine Methode von *IdentifiziererOderKlassenname* aus und legt den Rückgabewert in der Java-Schicht zusammen mit einem neuen Identifizierer (*Zielidentifizierer*) ab.
- **JavaLib.storeCopy(string QuellIdentifizierer, string Zielidentifizierer):** erzeugt einen neuen Identifizierer (*Zielidentifizierer***),** der auf dasselbe Objekt wie der *QuellIdentifizierer* zeigt.
- **JavaLib.storeField(string Zielidentifizierer, string IdentifiziererOderKlassenname, string Feldname):** speichert den Wert eines Feldes unter Verwendung eines neuen Identifizierers (*Zielidentifizierer*) in der Java-Schicht ab.
- **JavaLib.storeNew(string Zielidentifizierer, string Klassenname [, Argument(e)*]):** erzeugt ein neues Objekt der Klasse *Klassenname* unter Verwendung der optionalen Argumente und einen zugehörigen Identifizierer (*Zielidentifizierer***)**.

Wie oben beschrieben, erfolgt die Kommunikation ausschließlich unter Verwendung von Strings. Dies führt dazu, dass der Compiler keine Typprüfung der Java-Objekte vornimmt, weil sich der Wert eines String während der Laufzeit ändern kann (wenn sich der String in einer Variable befindet). Die Entwickler müssen deshalb besonderes Augenmerk auf die verwendeten Java-Typen und Methoden-/Feldnamen legen, da Fehler erst zur Laufzeit entdeckt werden können.

Weiterhin ist bei der Programmierung zu beachten, dass alle Java-Objekte, die nicht mehr verwendet werden, über die Funktionen remove() oder removeAll() manuell freigegeben

werden müssen. Dies dürfte vor allem für Java-Entwickler ungewohnt sein. Die automatische Speicherverwaltung der Java-Virtual-Maschine erledigt dies im Normalfall selbstständig.

Nachfolgend wird gezeigt, wie die erweiterte HelloWorld-Klasse aus Kapitel 10.2 dieser Arbeit mit Hilfe der JavaLib verwendet werden kann.

```
// Aufruf der statischen Methode create. Der Rückgabewert wird
// in der Java-Schicht abgelegt. Als Identifizierer für das neue
// Objekt wird „hello“ verwendet.
javaLib.store((objId)"hello","eu.erras.beispiel.HelloWorld",
              "create");

// Aufruf der Methode „sayHello“ und Ausgabe des Rückgabewerts auf
// der Konsole.
sysLib.writeStdout(javaLib.invoke((objId)"hello","sayHello"));
```

10.4 Fehlerbehandlung bei Verwendung der JavaLib

In Abschnitt 7 dieser Arbeit wurde dargestellt, dass die EGL mehrere Möglichkeiten der Fehlerbehandlung bietet. Dies ist auch bei Verwendung der JavaLib der Fall:

- **Try-Block**: Aufrufe der JavaLib können in Try-Blöcke gekapselt werden. Wird innerhalb des Java-Codes eine Ausnahme geworfen, die der Java-Code nicht selbst behandelt, wird ein evtl. vorhandener onException-Block im EGL-Programm angesprungen oder die Ausführung nach dem Try-Block fortgesetzt.

- **Globale Variable**: Sind die Java-Aufrufe nicht innerhalb eines Try-Blocks implementiert, kann eine Ausnahme entweder durch den automatischen Stopp des Programms oder durch Auswerten der globalen Variable sysVar.errorCode behandelt werden. Die Information über die jeweilige Variante erhält das Programm durch Setzen der Variable VGVar.handelSysLibraryError:

    - **VGVar.handelSysLibraryError = 0**: Programm endet bei einem Fehler.

- **VGVar.handelSysLibraryError = 1**: Die globale Variable sysVar.errorCode wird mit einem achtstelligen String befüllt, der Aufschluss über die Art des Fehlers gibt. Falls ein Java-Objekt bei einem Aufruf über JavaLib eine Ausnahme wirft, wird sysVar.errorCode mit dem Wert „00001000“ belegt. Um im EGL-Programm abzufragen, um welche Ausnahme es sich handelt, verwendet man den Aufruf JavaLib.qualifiedTypeName((objId)"caughtException");. Dies ist möglich, da die zuletzt geworfene Ausnahme in der Java-Schicht immer unter dem Identifizierer caughtException abgelegt wird. Ein möglicher Rückgabewert von JavaLib.qualifiedTypeName((objId)"caughtException"); wäre z. B. „java.lang.SecurityException“. Um die Fehlermeldung der Ausnahme abzufragen, wird der Aufruf JavaLib.invoke((objId)"caughtException", "getMessage"); verwendet.

Neben dem Wert „00001000“ kann sysVar.errorCode noch weitere Fehlercodes annehmen. Diese sind in der folgenden Tabelle beschrieben [IBM1]:

Fehlercode	Beschreibung
00001000	Es wurde eine Ausnahme geworfen.
00001001	Das Objekt war null oder der Identifizierer wurde in der Javaschicht nicht gefunden.
00001002	Es wurde keine öffentliche Methode / Feld / Klasse mit dem angegebenen Namen gefunden.
00001003	Der EGL-Datentyp passt nicht zum erwarteten Java-Typ.
00001004	Die Methode gab keinen Wert oder null zurück, bzw. der Wert des Feldes war null.
00001005	Der Rückgabewert passt nicht zum Typ der Rückgabevariable.

00001006	Die Klasse einer Null-Referenz konnte nicht ermittelt werden.
00001007	Beim Versuch Informationen über eine Methode oder ein Feld abzufragen, oder den Wert eines Feldes, das als final gekennzeichnet wurde, wurde entweder eine SecurityException oder eine IllegalAccessException geworfen.
00001008	Es konnte kein Konstruktor aufgerufen werden.
00001009	Statt eines Identifizieres wurde ein Klassenname angegeben.

10.5 Aufbau der Objektstruktur

Um die Vorteile der Objektorientierung nutzen zu können, wurde für jede Entität des Datenbankmodells eine eigene Java-Klasse erstellt (siehe Abbildung 32), wobei eine Instanz einer solchen Klasse jeweils einen Datensatz einer Datenbanktabelle (siehe Kapitel 8.2 dieser Arbeit) repräsentiert. Für den Zugriff auf die Datenfelder eines Datensatzes sind entsprechende Methoden vorhanden.

Abbildung 32: UML-Diagramm der Java-Klassen

Weiterhin gibt es Methoden, über die sich alle Objekte, die mit einem Objekt in Beziehung stehen, abrufen lassen. So hat die Klasse Prozess eine Methode namens getKontrollaktivitaeten(), die ein Array von Kontrollaktivitäten zurückgibt. Es handelt sich dabei um alle Kontrollaktivitäten, die mit dem Prozess verknüpft sind.

Außerdem enthält jede Klasse mindestens eine statische Methode GetInstance() (Fabrikmethode), über die die einzelnen Instanzen anhand eines Parameters (z. B. ID oder Name) erzeugt oder abgerufen werden. Risiko.GetInstance(5, con); gibt das Risiko mit der ID 5 zurück, falls dieses in der Datenbank vorhanden ist (eine geöffnete Verbindung zur gewünschten Datenbank wird über den Parameter con übergeben). Anders können Instanzen nicht abgerufen werden, da alle Konstruktoren der Klassen private deklariert wurden. Die Instanzierung der Java-Klasse über eine statische Methode hat den Vorteil, dass die Anzahl der Datenbankzugriffe verringert werden kann, wenn ein Objekt für einen bestimmten Da-

tensatz mehrfach instanziert wird. Dies wird dadurch erreicht, dass jede Klasse neben der statischen Methode zusätzlich eine statische Collection (in diesem Fall ein Hashtable) enthält, in dem Verweise auf alle bereits instanzierten Objekte des jeweiligen Klassentyps gehalten werden. Wird nun GetInstance aufgerufen, wird diese Collection zunächst nach einer bestehenden Instanz durchsucht, die zu den Eingabeparametern passt. Wird eine passende Instanz gefunden, wird diese zurückgegeben. Falls keine passende Instanz gefunden wird, wird ein neues Objekt erzeugt und ein Datenbankzugriff durchgeführt. Diese Vorgehensweise bietet außerdem bei Klassenmodellen, bei denen die Klassen zyklisch aufeinander verweisen, den Vorteil, dass keine weitere Logik implementiert werden muss, um einen Endloslauf beim Einlesen zu verhindern, da bereits eingelesene Objekte nicht mehrfach geladen werden. Evtl. erhöht sich hierdurch aber der Bedarf an Arbeitsspeicher.

Die Klasse Prozess hat zusätzlich eine statische Methode namens GetInstances. Diese liest alle Prozesse, die einen bestimmten Namen tragen und gibt sie als Array zurück. Weiterhin unterscheidet sie sich von den GetInstance-Methoden dadurch, dass sie keine geöffnete Datenbankverbindung erwartet. Stattdessen werden die nötigen Parameter zum Öffnen der Datenbankverbindung an die Methode übergeben. Die eigentliche Verbindung wird dann von der Klasse selbstständig aufgebaut und bei Aufrufen von GetInstance-Methoden übergeben (z. B. beim Aufruf von Prozessgruppe.GetInstance innerhalb der Methode getProzessgruppe).

10.6 Implementierung unter Verwendung von Java-Objekten

Zunächst legt man ein neues EGL-Projekt mit dem Namen ReportingEGLJava an. Damit die zu verwendenden Java-Klassen auch gefunden werden, muss dem Projekt mitgeteilt werden, wo sich die JAR-Datei mit dem Java-Bytecode befindet. Hierzu wird der Projektname im Project Explorer mit der rechten Maustaste angeklickt und Properties ausgewählt. Es öffnet sich ein neuer Dialog, in dem die Kategorie Java Build Path ausgewählt wird (siehe Abbildung 33).

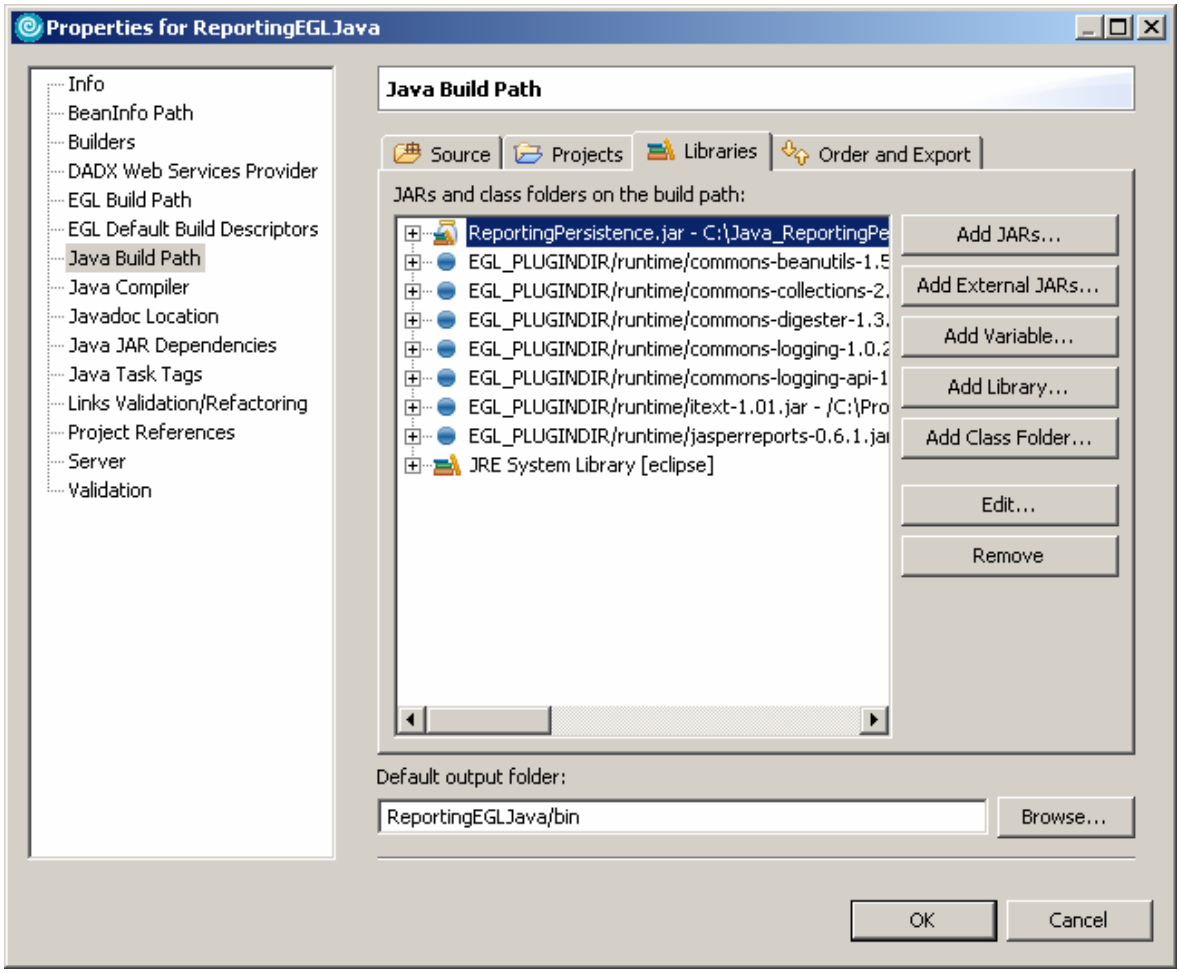

Abbildung 33: Auswahl einer externen JAR-Datei

Eine JAR-Datei kann durch einen Klick auf den Button Add External JARs… im Reiter Libraries hinzugefügt werden. Die benötigte JAR-Datei heißt ReportingPersistence.jar und befindet sich im Ordner c:\Reporting. Nach dem Einfügen der JAR-Datei wird der Dialog durch OK geschlossen. Weitere Projekteinstellungen sind nicht nötig.

Abschließend wird ein neues EGL-Programm vom Typ Basic Programm im Default-Package erzeugt. Der Name des Programms ist ReportingEGLJavaMain. Der darin bereits enthaltene Source-Code wird durch den folgenden ersetzt:

```
program ReportingEGLJavaMain type BasicProgram

   function main()
         // Zählvariablen:
         i int;
         j int;
         k int;
         l int;

         // Strings für die XML-Ausgabe:
         Prozessname string;
         Prozessgruppenname string;
         KAname string;
         RisikoBeschreibung string;

         // Variablen zur Aufnahme der IDs:
         KARisikoID int;
         PGRisikoID int;

         // Variablen zur Aufnahme der Arraygrößen
         lenProzesse int;
         lenKAs int;
         lenKARisiken int;
         lenPGRisiken int;

         // Rückgabewert
         xml string;
         DateiInhalt clob; // Textdatetyp zur Ausgabe in eine Datei
         Filename String; // Dateiname der Ausgabedatei
         Systemaufruf string; // temporärer String für den Systemaufruf
         errMsg String; // Fehlermeldungen
```

```
// falls kein Parameter übergeben wurde, Programm verlassen.
if (sysLib.getCmdLineArgCount() < 1)
      exit program;
end

Prozessname = sysLib.getCmdLineArg(1);

// Fehlerbehandlung über Ausnahmen
try
      // Array aller gefundenen Prozesse in der Java-Schicht ablegen.
      javaLib.store((objId)"arrProzesse", "eu.erras.Prozess", "GetInstances",Prozessname
                  ,"jdbc:db2://localhost:50000/PROZESSE","test","test");

      // Größe des Prozessarrays ermitteln.
      lenProzesse = javaLib.invoke( "java.lang.reflect.Array",  "getLength", (objId)"arrProzesse" );

      // alle gefundenen Prozesse durchlaufen
      for (i from 0 to lenProzesse -1 by 1)

            // aktuellen Prozess in eine Variable speichern.
            javaLib.store( (objId)"Prozess", "java.lang.reflect.Array", "get",
                        (objId)"arrProzesse", i );

            // Prozessgruppe des Prozesses abrufen.
            javaLib.store( (objId)"Prozessgruppe", (objId)"Prozess", "getProzessgruppe" );

            // Risiken der Prozessgruppe abrufen.
            javaLib.store( (objId)"arrPGRisiken", (objId)"Prozessgruppe", "getRisiken" );

            // Kontrollaktivitäten des Prozesses abrufen.
            javaLib.store( (objId)"arrKAs", (objId)"Prozess", "getKontrollaktivitaeten" );

            // Prozessnamen abrufen.
            Prozessname = javaLib.invoke((objId)"Prozess","getName");
```

```
// Namen der Prozessgruppe abrufen.
Prozessgruppenname = javaLib.invoke((objId)"Prozessgruppe","getName");

// XML erzeugen
xml = "<?xml version='1.0' encoding='iso-8859-15' ?>";
xml = xml + "<REPORT>";

xml = xml + "<PROZESS>" + Prozessname + "</PROZESS>";

xml = xml + "<PROZESSGRUPPE>" + Prozessgruppenname + "</PROZESSGRUPPE>";

// Anzahl der Kontrollaktivitäten ermitteln.
lenKAs = JavaLib.invoke( "java.lang.reflect.Array",  "getLength", (objId)"arrKAs" );

// Namen aller Kontrollaktivitäten in XML ausgeben.
for (j from 0 to lenKAs -1 by 1)
      javaLib.store( (objId)"KA", "java.lang.reflect.Array", "get", (objId)"arrKAs", j );

      KAname = javaLib.invoke((objId)"KA","getName");

      xml = xml + "<KONTROLLAKTIVITAET>" + KAname + "</KONTROLLAKTIVITAET>";
end

// Anzahl der Risiken der Prozessgruppe ermitteln.
lenPGRisiken = javaLib.invoke( "java.lang.reflect.Array",  "getLength",
                               (objId)"arrPGRisiken" );

// Risiken der Prozessgruppe durchlaufen
for (j from 0 to lenPGRisiken -1 by 1)

      // Aktuelles Risiko in Variable speichern
      javaLib.store((objId)"PGRisiko", "java.lang.reflect.Array", "get",
                    (objId)"arrPGRisiken", j );
```

```
// Risikobeschreibung abrufen
RisikoBeschreibung = javaLib.invoke((objId)"PGRisiko","getBeschreibung");

// Risiko ID abrufen
PGRisikoID = javaLib.invoke((objId)"PGRisiko","getId");

// Informationen zum Risiko in XML schreiben.
xml = xml + "<RISIKO>";
xml = xml + "<BESCHREIBUNG>" + RisikoBeschreibung
      + "</BESCHREIBUNG>";
xml = xml + "<ID>" + PGRisikoID + "</ID>";
xml = xml + "<ABDECKUNG>";

// alle Kontrollaktivitäten des Prozesses durchlaufen
for (k from 0 to lenKAs -1 by 1)
      javaLib.store( (objId)"KA", "java.lang.reflect.Array", "get",
                     (objId)"arrKAs", k );

      // Risiken der aktuellen Kontrollaktivität abrufen
      javaLib.store( (objId)"arrKARisiken", (objId)"KA", "getRisiken" );

      // Anzahl der Risiken der Kontrollaktivität ermitteln
      lenKARisiken = javaLib.invoke( "java.lang.reflect.Array",  "getLength",
                                     (objId)"arrKARisiken" );

      // Übereinstimmung der Risiken der Kontrollaktivität
      // mit dem aktuellen Risiko der Prozessgruppe ermitteln.
      for (l from 0 to lenKARisiken -1 by 1)
            javaLib.store((objId)"KARisiko", "java.lang.reflect.Array", "get",
                          (objId)"arrKARisiken",l );

            KARisikoID = javaLib.invoke((objId)"KARisiko","getId");

            // falls eine Übereinstimmung gefunden wurde,
```

```
                        // Abdeckung im XML speichern und Schleife abbrechen.
                        if (KARisikoID == PGRisikoID)
                              KAname = javaLib.invoke((objId)"KA","getName");

                              xml = xml + "<ABGEDECKTDURCH>";
                              xml = xml + KAname;

                              xml = xml + "</ABGEDECKTDURCH>";
                              exit for;
                        end
                  end
            end
            xml = xml + "</ABDECKUNG>";
            xml = xml + "</RISIKO>";
      end

      xml = xml + "</REPORT>";


      // alle Java-Objekte freigeben.
javaLib.removeAll();

// Inhalt des XML-String ins Clob-Datentyp übertrangen.
// Dies ist nötig, da nur der Inhalt eines Clobs
// in eine Text-Date geschrieben werden kann.
lobLib.setClobFromString(DateiInhalt, xml);

// Pfad + Dateinamen erzeugen (ohne Endung).
// Um das Programm nicht unnötig zu verkomplizieren,
// wird der Ausgabepfad hart kodiert.

Filename = "c:\\reporting\\Java_" + Prozessname + "_" + i ;

// Clob auf die Festplatte schreiben.
```

```
                    lobLib.updateClobToFile(
                    DateiInhalt,
                      Filename +".xml");

                    Systemaufruf = "c:\\fop\\fop.bat -xsl c:\\fop\\report.xsl -xml \"" +
                          Filename + ".xml\" -pdf \"" + Filename +".pdf\"";

                    // Fop aufrufen
                    sysLib.callCmd( Systemaufruf );
              end
        onException
              // ist eine Java-Ausnahme aufgetreten?
              if (sysVar.errorCode == "00001000")
                    // Fehlermeldung abfragen
                    errMsg = javaLib.invoke( (objId)"caughtException",  "getMessage" );
              else
                    errMsg = "Es ist eine unbehandelte Ausnahme aufgetreten!";
              end

              // Fehlermeldung ausgeben
              sysLib.writeStderr(errMsg);

              // alle Java-Objekte freigeben.
              javaLib.removeAll();

              // Programm beenden
              exit program;
        end
   end
end
```

Die Java-Variante des Programms orientiert sich an der bereits in Kapitel 9 dieser Arbeit implementierten Logik. Allerdings wurde die eigentliche Kommunikation mit der Datenbank in entsprechende Java-Klassen ausgelagert, die nun vom EGL-Programm instanziert und verwendet werden (der Java-Quellcode ist im Anhang abgedruckt). Hierzu werden die Funktionen JavaLib.Store(), JavaLib.Invoke() und JavaLib.RemoveAll() verwendet. Außerdem haben die Dateinamen aller Reports, die von der Java-Variante erzeugt werden, den Prefix „Java_“.

Zu beachten ist, dass das Programm nur lauffähig ist, wenn die JAR-Datei mit dem Java-Bytecode der Java-Klassen auch vom System gefunden wird. Dies ist sowohl während der Entwicklung (z. B. bei Debuggerläufen) als auch beim späteren Produktivbetrieb notwendig (die JAR-Datei muss also zusätzlich ausgeliefert werden). Weiterhin ist zu bedenken, dass ein Anwender oder Entwickler nur zur Laufzeit des Programms evtl. Fehler bei der Java-Kommunikation feststellen kann, da die angegebenen Java-Klassen und Methoden erst zur Laufzeit gesucht und instanziert bzw. aufgerufen werden. Selbst einfache Schreibfehler (z. B. falsche Klassennamen) werden erst zur Laufzeit erkannt. Bei allen Code-Teilen, die Java-Code verwenden, sollten deshalb umfangreiche Abdeckungstests durchgeführt werden.

Die Fehlerbehandlung des Programms ist über Ausnahmen realisiert. Der onException-Block enthält dabei eine Fallunterscheidung:

- Im Fall einer Java-Ausnahme wird die Fehlermeldung dieser Ausnahme aus der Java-Schicht abgefragt.
- Im Fall aller anderen Ausnahmen wird eine Meldung ausgegeben.

Alternativ wäre auch eine Fehlerbehandlung ohne Try-/onException-Blöcke möglich (vgVar.handleHardIOErrors muss dazu auf 1 gesetzt werden). Da in diesem Fall die globale Variable sysVar.errorCode zur Erkennung von Fehlern verwendet wird, kann beispielsweise folgende Funktion zur Fehlerbehandlung verwendet werden:

```
// Da die Java-Fehler über eine globale Variable ermittelt werden,
// benötigt die Funktion keine Parameter.
function handleJavaErrors()
   errMsg String;

   // Java-Fehler behandeln
   if (sysVar.errorCode == "00001000")
         // Fehlermeldung abfragen
         errMsg = JavaLib.invoke( (objId)"caughtException",
                  "getMessage" );
               // Fehlermeldung ausgeben
         sysLib.writeStderr(errMsg);
               // alle Java-Objekte freigeben.
         JavaLib.removeAll();
         // Programm beenden
         exit program;

         // alle anderen Fehler behandeln
   else if (sysVar.errorCode != "00000000")
         errMsg = "Es ist ein Fehler aufgetreten!"
               // Fehlermeldung ausgeben
         sysLib.writeStderr(errMsg);
               // alle Java-Objekte freigeben.
         JavaLib.removeAll();
         // Programm beenden
         exit program;
   end
end
```

Diese Funktion muss dann nach jeder Anweisung aufgerufen werden, um evtl. aufgetretene Fehler zu verarbeiten.

11 Characterset der EGL

Nicht zuletzt durch die Verbreitung des Internets muss bei der Softwareentwicklung (vor allem im Server-/Mainframe-Bereich) auf die Fähigkeit zur Internationalisierung geachtet werden. Eine Grundvoraussetzung zur Verarbeitung und Darstellung von Texten, die in verschiedenen Sprachen verfasst wurden, ist die Fähigkeit, die passenden Schriftzeichen zu verarbeiten.

Für die Unterstützung verschiedener Zeichensätze haben sich in der Informatik zwei Ansätze etabliert:

- **Codepages:** Unter einer Codepage versteht man die Zusammenfassung aller Zeichen, die in einer bestimmten Sprache oder in einem bestimmten Sprachraum verwendet werden. Zusätzliche Zeichen, die in diesem Sprachraum nicht verwendet werden, sind in solchen Codepages meist nicht enthalten. Ein Beispiel hierfür ist die standardisierte Codepage ISO[13] 8859-15, die den Zeichenvorrat für Westeuropa abdeckt. Insgesamt enthält ISO 8859-15 191 Zeichen, die mit einem Byte pro Zeichen kodiert werden können (d. h. der Speicherbedarf pro Zeichen variiert nicht). So kodierte Dateien sind also relativ klein. Außerdem kann bei einer einfachen Textdatei davon ausgegangen werden, dass die Dateigröße in Bytes der Anzahl der enthaltenen Zeichen entspricht, was vor allem beim Anspringen bestimmter Textstellen oder beim Speichern von Texten in Datenbanken von Vorteil ist.

  Die Verwendung von Codepages führt allerdings zu Problemen, wenn Texte mit unterschiedlichen Zeichensätzen in einem System verarbeitet werden müssen. Hierzu muss zu jedem Text die verwendete Codepage abgespeichert werden. Zusätzliche Probleme entstehen, wenn die Texte unterschiedlicher Codepages auf die gleiche Art und Weise verarbeitet werden müssen. Soll z. B. eine Suche in einer Da-

[13] ISO ist die Kurzbezeichnung der International Organization for Standardization. Es handelt sich dabei allerdings nicht um eine Abkürzung. ISO ist abgeleitet vom griechischen Wort „isos“ = „gleich“ [ISO].

tenbank implementiert werden, muss auch hier die Codepage berücksichtigt werden, was häufig die Verwendung von Standardkomponenten erschwert.

Es kann bei relationalen Datenbanken häufig nur ein Datensatz pro Tabellenspalte angegeben werden. D. h. die Datenbank geht bei einer Textspalte immer davon aus, dass alle Texte, die sich in dieser Spalte befinden, mit der gleichen Codepage kodiert wurden. Somit erfolgt auch die Suche in einer solchen Spalte immer mit der für diese Spalte konfigurierten Codepage. Werden nun einzelne Texte in dieser Spalte in einer anderen Codepage kodiert, schlägt diese Suche evtl. fehl oder liefert falsche Ergebnisse zurück, da die einzelnen Codepages miteinander kollidieren können.

Um dies zu umgehen, muss das Datenbanksystem so erweitert werden, dass mehrere Codepages pro Spalte erlaubt werden. Alternativ kann auch eine externe Suchlogik implementiert werden, die alle Zeichen entsprechend ihrer Codepage interpretiert und ggf. konvertiert. Beide Lösungsansätze führen zu verminderter Performance, da in jedem Schritt Maßnahmen zur Zeichenkonvertierung durchgeführt werden müssen.

Weiterhin führt die Verwendung von Codepages zu Problemen, wenn eine bestehende Anwendung, die bisher nur eine Codepage unterstützt, um weitere Codepages erweitert werden soll. Dies zieht je nach Architektur der Anwendung einen hohen Änderungsaufwand nach sich.

- **Unicode:** Um die oben beschriebenen Probleme zu vermeiden, wurde der Unicode-Standard geschaffen (ISO 10646), in dem die Zeichen aller Sprachen einheitlich durchnummeriert sind. Dies führt zwar, abhängig von der jeweiligen Kodierung des Unicode-Zeichensatzes, zu erhöhtem Speicherbedarf, da in vielen Texten nur eine sehr kleine Teilmenge des kompletten Zeichensatzes benötigt wird, erhöht aber die Kompatibilität zwischen verschiedenen Systemen und Sprachen.

  Die bedeutendsten Kodierungen für den Unicode-Zeichensatz sind UTF-8 und UTF-16. Es handelt sich hierbei um Multibyte-Kodierungen mit variabler Größe pro Schriftzeichen.

Bei UTF-8 werden die ersten 127 Zeichen des Unicodes mit genau einem Byte kodiert. Alle höherwertigen Zeichen benötigen, abhängig vom Zahlenwert, ein bis drei zusätzliche Bytes. Um anzuzeigen, aus wie vielen Bytes ein Zeichen besteht, werden die ersten ein bis drei Bits der einzelnen Bytes als Steuerbits verwendet. Eine genaue Beschreibung dieser Steuerbits und der Kodierung von UTF-8 findet sich in [UTF8]. Da die ersten 127 Zeichen von UTF-8 dem American Standard Code for Information Interchange entsprechen, der unter anderem das lateinische Alphabet enthält, kann UTF-8 für die effiziente Kodierung von Texten, die (hauptsächlich) aus diesen Zeichen bestehen, verwendet werden (ein solches Zeichen verbraucht in diesem Fall genau ein Byte). Werden allerdings Zeichen anderer Sprachen benutzt, ist UTF-8 ungeeignet, da höherwertige Zeichen wegen der benötigten Steuerbits vergleichsweise ineffizient abgespeichert werden.

Bei UTF-16 wird jedes Schriftzeichen mindestens mit 16 Bits kodiert, wobei Zeichen, die in diesen 16 Bits keinen Platz finden, mit 32 Bits kodiert werden. Dies führt dazu, dass weniger Steuerbits benötigt werden und somit einige Zeichen, die in UTF-8-Kodierung drei Bytes benötigen würden, bei UTF-16-Kodierung mit zwei Bytes auskommen. Andererseits benötigen auch die niederwertigen Zeichen immer 16 Bit-Speicher [UTF16].

Die EGL kennt folgende Datentypen zur Speicherung von Schriftzeichen [IBM1]:

- *CHAR refers to single-byte characters.*

- *DBCHAR refers to double-byte characters. dbchar replaces DBCS, which was a primitive type in EGL V5.*

- *MBCHAR refers to multibyte characters, which are a combination of single-byte and double-byte characters. mbchar replaces MIX, which was a primitive type in EGL V5.*

- *STRING refers to a field of varying length, where the double-byte characters conform to the UTF-16 encoding standards developed by the Unicode Consortium.*

- *UNICODE refers to a fixed field, where the double-byte characters conform to the UTF-16 encoding standards developed by the Unicode Consortium.*

- *HEX refers to hexadecimal characters.*

Innerhalb von Strings wird somit die UTF-16-Kodierung verwendet, wobei 4 Byte UTF-16-Zeichen nicht unterstützt werden. Beim Einlesen von Daten muss darauf geachtet werden, dass alle Zeichen korrekt in das UTF-16-Format konvertiert werden und keine 4-Byte-Zeichen enthalten sind.

Dank der UTF-16-Unterstützung sind EGL-Programme für internationale Anwendungen geeignet. Leider fehlen im Standardumfang Funktionen um die Konvertierung zwischen verschiedenen Kodierungsmethoden oder Codepages durchführen zu können. Außerdem muss beim Programmieren berücksichtigt werden, dass nicht alle Zeichendatentypen der EGL UTF-16 unterstützen.

12 Gefundene Fehler und Probleme im WDz 6.0.1

Im Rahmen der Vorbereitungen zu dieser Arbeit sind einige Fehler oder Probleme im Zusammenhang mit dem WDz aufgetreten. Dieses Kapitel soll dem Entwickler helfen, diese Probleme zu umgehen, da sich die Fehlersuche oft erschwert, wenn Fehler nicht durch den Benutzer, sondern durch die verwendete Software verursacht werden.

12.1 Generierung von DataParts auf Basis einer bestehenden Datenbank

12.1.1 Tabellen, deren Namen Anführungszeichen enthalten

Werden in der hier verwendeten DB2-Datenbank Tabellen angelegt, deren Namen von Anführungszeichen umschlossen sind (z. B.: „Prozessgruppe“), schlägt die automatische Generierung von DataParts fehl. Das Generieren von Dataparts für die betroffenen Tabellen war erst nach Entfernung der Anführungszeichen aus den Tabellennamen möglich.

12.1.2 Datenfelder, deren Namen Anführungszeichen enthalten

Falls in einer Tabelle Spaltennamen von Anführungszeichen umschlossen sind, schlägt die Generierung der entsprechenden EGL-Library zwar nicht fehl, führt aber zu fehlerhaftem EGL-Code.

So wurde beispielsweise folgende Zeile generiert, die sich nicht in gültigen Java-Code umwandeln lässt:

```
DataItem "Beschreibung"11 string {displayName = ""Beschrei-
bung"11"} end
```

Das Generieren von Java-Code wird in diesem Fall mit der Fehlermeldung „Syntax error on input ''Beschreibung''." beendet. Der Benutzer erhält keinen Hinweis darauf, dass dieser Fehler durch den Spaltennamen verursacht wird.

12.2 Einordnung von geschlossenen EGL-Projekten im WDz

Wird ein EGL-Projekt in der EGL-Perspektive geschlossen, verschwindet das Projekt aus der Liste der EGL-Projekte. Es wird nach dem Schließen in die Gruppe „Other Projects" eingeordnet. Dieser Umstand kann vor allem unerfahrene WDz-Entwickler verwirren, da sich der Projekt-Typ durch das Schließen eines Projekts eigentlich nicht ändert.

12.3 Markierung bereinigter Fehler

Im WDz kann man gefundene Syntaxfehler im Quellcode automatisch direkt neben der betroffenen Zeile durch einen roten Kreis markieren lassen. Diese Markierungen verschwinden zum Teil nicht automatisch, wenn die Fehler korrigiert werden. Vielfach muss erst eine andere Aktion (z. B. Neukompilierung) angestoßen werden, damit die Markierungen verschwinden.

12.4 Änderung des Build-Descriptors

Wird der Build-Descriptor geändert und danach der Debugger gestartet, wirkt sich die Änderung des Build-Descriptors erst nach einer Neugenerierung des Codes aus. Um die Arbeit zu erleichtern, wäre es besser, wenn der WDz den Code nach einer Änderung am Build-Descriptor automatisch neu generieren oder zumindest eine Meldung ausgeben würde.

13 Zusammenfassung

Bei der EGL handelt es sich um eine prozedurale Programmiersprache, die sich vor allem zur Datenverarbeitung in Kombination mit relationalen Datenbanken eignet, denn speziell hierfür stehen Zugriffsfunktionen zur Verfügung bzw. können automatisch durch den WDz generiert werden. Weiterhin bietet sie den Vorteil, dass sie nicht direkt in Maschinencode kompiliert wird. Stattdessen wird lauffähiger Cobol- oder Java-Code erzeugt, der sich einfach in bestehende Anwendungen integrieren lässt. Zusätzlich bietet dies die Möglichkeit, bei einem Technologiewechsel eine neue Zielsprache zu wählen. In Kapitel 10.1 dieser Arbeit wurde außerdem gezeigt, welche Möglichkeiten bestehen, um bestehende Java-Komponenten in ein EGL-Programm zu integrieren.

Für die Entwicklung von EGL-Programmen stellt IBM mit dem WDz eine leistungsfähige Entwicklungsumgebung bereit, welche auf Eclipse basiert und sich auch zur Realisierung von Anwendungen eignet, die nicht mit Hilfe der EGL umgesetzt werden (z.B. Java- oder Cobol-Programme).

Größter Nachteil der EGL ist die inkonsequente Umsetzung der Fehlerbehandlung über Ausnahmen und das Fehlen von Funktionen zur Konvertierung von Zeichensätzen. An diesen Stellen sollte die EGL noch erweitert werden. Auch der WDz hat in Version 6.0.1 ein paar kleinere Schwachstellen, die sich aber nicht nachteilig auf den Entwicklungsprozess auswirken, wenn diese im Entwicklerteam bekannt sind (siehe Kapitel 12).

An dieser Stelle sei darauf hingewiesen, dass die EGL ab Version 7.0 kein Teil der Websphere-Entwicklungsumgebung mehr sein wird. Stattdessen wird das Modul zur EGL-Entwicklung in ein Erweiterungsmodul ausgegliedert, welches separat erhältlich sein wird und wahrscheinlich im Laufe des Jahres 2007 auf den Markt kommt [IBM3].

Nach Ansicht des Autors ist die EGL vor allem dann für den Einsatz in Softwareprojekten geeignet, wenn hauptsächlich Daten verarbeitet werden sollen, und auf Objektorientierung verzichtet werden soll oder muss. Außerdem bietet sie eine Möglichkeit bestehende Java- oder Cobol-Programme sinnvoll zu erweitern.

Literaturverzeichnis

[CAS] F. Cassia: *Eclipse.org eclipsing Borland's Jbuilder*, VNU Business Publications, o. O., 2004, http://www.theinquirer.net/default.aspx?article=15862 [Stand 11.10.2006].

[CI] *IBM Weighs in With Uptime Guarantees*, Computergram International, o. O., 1999, http://findarticles.com/p/articles/mi_m0CGN/is_1999_March_25/ai_54207476/pg_ [Stand 12.10.2006].

[DB2] *DB2 Product Family*, IBM, o. O., o. J., http://www-306.ibm.com/software/data/db2/ [Stand 06.11.2006].

[FOP] *Apache FOP*, Apache Software Foundation, o. O., 2007, http://xmlgraphics.apache.org/fop/index.html [Stand 19.10.2006].

[FRÖ] G. Fröhlich: *Designprinzipien moderner Prozessoren*, Gesellschaft für Schwerionenforschung, Darmstadt, 2006, http://bel.gsi.de/designprinzipien.ppt [Stand 28.01.2007].

[GLI] M. Glinz, H. Gall: *Software Engineering - Systematisches Programmieren*, Universität Zürich, 2006, http://www.ifi.unizh.ch/rerg/fileadmin/downloads/teaching/courses/software_engineering_ws0607/folien/Kapitel_06_Syst_Prog.pdf [28.02.2007].

[GNU] *GNU General Public License*, Free Software Foundation, o. O., 1991, http://www.gnu.org/copyleft/gpl.html [Stand 09.01.2007].

[GOL] *Sun stellt Java unter die GPL*, Golem.de, Berlin, 2006, http://www.golem.de/0611/48906.html [Stand 23.02.2007].

[HEI] *Java-IDE JBuilder 2007 baut auf Eclipse auf*, Heise Online, Hannover, 2006, http://www.heise.de/newsticker/meldung/81333 [Stand 09.10.2006].

[HER1] P. Herrmann, U. Kebschull, W. G. Spruth: *Einführung in z/OS und OS/390*, 2. Auflage, Oldenbourg, 2004.

[HER2] P. Herrmann, W. G. Spruth: *Vorlesungsskript Einführung in z/OS und OS/390 - Teil 1*, Universität Leipzig, 2005.

[HER3] P. Herrmann, W. G. Spruth: *Vorlesungsskript Einführung in z/OS und OS/390 - Teil 11*, Universität Leipzig, 2006.

[HER4] P. Herrmann, W. G. Spruth: *Vorlesungsskript Einführung in z/OS und OS/390 - Teil 7*, Universität Leipzig, 2006.

[HER5] P. Herrmann, U. Kebschull, W. G. Spruth: *Vorlesungsskript Internet Anwendungen unter z/OS und OS/390 - Teil 4*, Universität Tübingen, 2004.

[HOF] K. Hoffmann: *Vorlesungsskript Software-Engineering 1 Einführung*, FH Amberg Weiden, 2005.

[IBM1] *EGL Reference Guide Version 6 Release 01*, IBM, o. O., 2005.

[IBM2] *System z Application Assist Processor (zAAP)*, IBM, o. O., o. J., http://www-03.ibm.com/systems/z/zaap/ [Stand 18.12.2006].

[IBM3] *Statement of direction - March 2007 Enterprise Generation Language,* IBM, o. O., 2007, http://www-304.ibm.com/jct03002c/software/awdtools/wdt400/news/direction700.html [Stand 31.03.2007].

[IBM4] *Quarterly earnings*, IBM, o. O., 2007, http://www.ibm.com/investor/4q06/4q06earnings.phtml [Stand 08.04.2007]

[IDE] *IDEF1X Data Modeling Method*, IDEF, o. O., o. J., http://www.idef.com/IDEF1x.html [Stand 10.12.2006].

[ISO] *Why is the short name of the International Organization for Standardization "ISO" rather than "IOS"?*, International Organization for Standardization, o. O., 2003, http://www.iso.org/iso/en/xsite/contact/01enquiryservice/013logo/query1.html [Stand 27.02.2007].

[KOC] G. Koch: *Vorlesungsskript Softwaremanagement in der Versicherungswirtschaft - Teil 1*, Universität Leipzig, 2005.

[KOL] H. Kolinsky: *Programmieren in Fortran 90/95*, Universität Bayreuth, o. J., http://www.rz.uni-bayreuth.de/lehre/fortran90/vorlesung/V01/V01.html [Stand 23.02.2007].

[PIN] M. M. Pineda: *XSL-FO – Einführung in die Sprache für Seitengestaltung und Umbruch*, dpunkt.verlag, Mannheim, 2004, http://www.data2type.de/xml/XSL-FO.html [Stand 11.01.2007].

[SEW] K. Sewell: *Introduction to the IBM Workplace Managed Client Developer Toolkit*, IBM, o. O., 2005, http://www-128.ibm.com/developerworks/lotus/library/wmc-toolkit/ [Stand 13.01.2007].

[SWT1] *SWT: The Standard Widget Toolkit*, o. O., o. J., http://www.eclipse.org/swt/ [Stand 13.01.2007].

[SWT2] *Why SWT*, o. O., o. J., http://eclipsewiki.editme.com/WhySWT [Stand 13.01.2007].

[THR] M. Thränert: *Einführung in XML - Zusammenfassung Schemasprachen: DSDL*, Universität Leipzig, 2004, http://bis2.informatik.uni-leipzig.de/studium/vorlesungen/2004_ws/xml/dsdl/2004w_xml_dsdl_2fach.pdf [Stand 05.02.2007].

[ULL] Ch. Ullenboom: *Java ist auch eine Insel*, 6. Auflage, Galileo Computing, Bonn, 2007.

[ULM] *Vorgehensweisen zur Einführung in das Datenbank-Design*, Universität Ulm, o. J., http://www.uni-ulm.de/uni/intgruppen/memosys/db-nrm05.htm [Stand 12.02.2007].

[UTF16] P. Hoffman: *UTF-16, an encoding of ISO 10646*, The Internet Society, o. O., 2000, http://tools.ietf.org/html/rfc2781 [Stand 17.01.2007].

[UTF8] F. Yergeau: *UTF-8, a transformation format of ISO 10646*, The Internet Society, o. O., 2003, http://tools.ietf.org/html/rfc3629 [Stand 23.02.2007].

[VMP] *VMware Player*, VMware, Palo Alto, o. J., http://www.vmware.com/de/products/player/ [Stand 24.02.2007].

[W3C] A. Berglund: *Extensible Stylesheet Language (XSL) Version 1.1*, W3C, o. O., 2006, http://www.w3.org/TR/xsl/ [Stand 12.03.2007].

[WIN] S. Winter: *Online-Kurs 'Datenbanken und Datenmodellierung'*, Universität Passau, 2002, http://www.tcs.informatik.uni-muenchen.de/lehre/lehrerausbildung/db_sql_anfragen.pdf [Stand 23.02.2006].

[ZDNET] A. Donoghue: *Mainframes - nach all den Jahren noch immer aktuell*, ZDNet Deutschland, München, 2004, http://www.zdnet.de/enterprise/print_this.htm?pid=39118680-20000003c [Stand 15.02.2007].

Anhang

A Java-Quellcode der Persistenzklassen

Beispielhafte Implementierung der Persistenzobjekte, die in Kapitel 10 verwendet werden. Alle Methoden sind in der Javadoc-Syntax kommentiert.

A.1 Klasse Kontrollaktivität

```
package eu.erras;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Hashtable;

/**
 *
 * Beispielhafte Implementierung
 *
 * Repräsentiert eine Kontrollaktivität
 */
public class Kontrollaktivitaet implements ReportingObject
{

  private static Hashtable instances = new Hashtable();

  private String Name;

  private String Beschreibung;

  private int id;

  private Hashtable Risiken = null;

  private Connection con = null;

  /**
   * Konstruktor
   *
   * @param con Geöffnete Datenbankverbindung
   * @throws SQLException
   */
  private Kontrollaktivitaet(Connection con)
      throws SQLException
  {
```

```
    this.con = con;
  }

  /**
   * Gibt eine bestimmte Instanz zurück.
   *
   * @param id ID der gewünschten Instanz
   * @param con Verbindung zur Datenbank
   * @return Gesuchte Instanz
   * @throws SQLException
   */
  public static Kontrollaktivitaet GetInstance(int id,
      Connection con) throws SQLException
  {

    if (instances.containsKey(new Integer(id)))
    {
      return (Kontrollaktivitaet) instances
          .get(new Integer(id));
    }

    Kontrollaktivitaet newKA = new Kontrollaktivitaet(con);
    instances.put(new Integer(id), newKA);
    newKA.read(id);
    return newKA;

  }

  /**
   * Liest die Instanz aus der Datenbank.
   *
   * @param id ID der Kontrollaktivität
   * @throws SQLException
   */
  private void read(int id) throws SQLException
  {

    if (id > 0)
    {
      // SQL-Query konstruieren
      Statement stmt;
      stmt = con.createStatement();
      String query = "SELECT * "
          + "FROM tutorial.kontrollaktivitaet where ID = "
          + id;

      // Anfrage ausführen
      ResultSet rs = stmt.executeQuery(query);

      while (rs.next())
      {

        this.Beschreibung = rs.getString("Beschreibung");
        this.Name = rs.getString("Name");
        this.id = id;

      }
```

```
      // Ergebnismenge schliessen
      rs.close();

      // SQL-Anweisung schliessen
      stmt.close();
    }

  }

  /**
   * Gibt die Beschreibung als String zurück.
   *
   * @return Beschreibung
   */
  public String getBeschreibung()
  {
    return Beschreibung;
  }

  /**
   * Gibt die ID zurück.
   *
   * @return ID
   */
  public int getId()
  {
    return id;
  }

  /**
   * Setzt die ID.
   *
   * @param id Zu setzende ID
   */
  public void setId(int id)
  {
    this.id = id;
  }

  /**
   * Gibt den Namen zurück.
   *
   * @return Name
   */
  public String getName()
  {
    return Name;
  }

  /**
   * Setzt den Namen.
   *
   * @param name Name
   */
```

```
  public void setName(String name)
  {
    Name = name;
  }

  /**
   * Gibt alle Risiken zurück, die zu dieser Instanz gehören.
   *
   * @return Risiken
   * @throws SQLException
   */
  public Risiko[] getRisiken() throws SQLException
  {
    if (this.Risiken == null)
    {
      this.Risiken = new Hashtable();

      Statement stmt;
      stmt = con.createStatement();
      String query = "SELECT R_ID "
        + "FROM tutorial.risiko_kontrollaktivitaet where KA_ID = "
        + this.id;

      // Anfrage ausführen
      ResultSet rs = stmt.executeQuery(query);

      while (rs.next())
      {

        int id = rs.getInt("r_ID");

        this.Risiken.put(new Integer(id), Risiko
            .GetInstance(id, this.con));

      }

      // Ergebnismenge schliessen
      rs.close();

      // SQL-Anweisung schliessen
      stmt.close();
    }
    return (Risiko[]) this.Risiken.values().toArray(
        new Risiko[this.Risiken.values().size()]);
  }
}
```

A.2 Klasse Prozess

```
package eu.erras;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Hashtable;
import java.util.Properties;

/**
 * Beispielhafte Implementierung
 *
 * Repräsentiert einen Prozess.
 */
public class Prozess implements ReportingObject
{

  private static Hashtable instances = new Hashtable();

  private String Name;

  private int id;

  private int PG_id;

  private Prozessgruppe Prozessgruppe = null;

  private Hashtable Kontrollaktivitaeten = null;

  private Connection con = null;

  /**
   * Konstruktor
   *
   * @param con Aktive Verbindung zur Datenabnk
   * @throws SQLException
   */
  private Prozess(Connection con) throws SQLException
  {
    this.con = con;
  }

  /**
   * Gibt alle Prozesse zurück, die den übergebenen Namen tragen.
   *
   * @param prozessname Name der Prozesse
   * @param URL URL der Datenbankverbindung
   * @param Username Benutzername für die Datenbank
   * @param Password Passwort der Datenbank
   * @return Alle Prozesse, die den gesuchten Namen tragen.
   * @throws Exception
```

```
 */
public static Prozess[] GetInstances(String prozessname,
    String URL, String Username, String Password)
    throws Exception
{

  try
  {
    // JDBC-Treiber laden
    try
    {
      Class.forName("com.ibm.db2.jcc.DB2Driver");
    } catch (ClassNotFoundException exc)
    {
      System.err.println("Could not load DB2Driver:"
          + exc.toString());
      System.exit(1);
    }

    // Properties mit UserID und Passwort laden
    Properties props = new Properties();

    props.put("password", Password);
    props.put("user", Username);

    // Verbindung zur Datenbank herstellen
    Connection con = null;
    try
    {
      con = DriverManager.getConnection(URL, props);
    } catch (SQLException exc)
    {
      System.err.println("getConnection failed:"
          + exc.toString());
      return null;
    }

    //       Alle Einträge ausgeben
    try
    {
      return Prozess.GetInstances(prozessname, con);

    } catch (SQLException exc)
    {
      System.out.println("JDBC/SQL error: "
          + exc.toString());
      exc.printStackTrace();
      return null;
    }

  } catch (Exception e)
  {
    e.printStackTrace();
  }

  return null;
```

```
    }

    /**
     * Sucht einen Prozess anhand seiner ID und der ID der
     * Prozessgruppe, zu der der Prozess gehört.
     *
     * @param id ID des Prozesses
     * @param pg_id ID der Prozessgruppe
     * @param con Aktive Verbindung zur Datenbank
     * @return Gesuchter Prozess
     * @throws SQLException
     */
    public static Prozess GetInstance(int id, int pg_id,
        Connection con) throws SQLException
    {

      String Key;
      Key = String.valueOf(id) + "_" + String.valueOf(pg_id);

      if (instances.containsKey(Key))
      {
        return (Prozess) instances.get(Key);
      }

      Prozess newP = new Prozess(con);
      instances.put(Key, newP);
      newP.read(id, pg_id);
      return newP;

    }

    /**
     * Gibt alle Prozesse mit einem bestimmten Namen zurück.
     *
     * @param prozessname Name der gesuchten Prozesse
     * @param con Aktive Verbindung zur Datenbank
     * @return Prozessarray
     * @throws SQLException
     */
    public static Prozess[] GetInstances(String prozessname,
        Connection con) throws SQLException
    {

      Hashtable HT = new Hashtable();

      Prozess newProzess = null;
      Statement stmt;
      stmt = con.createStatement();
      String query = "SELECT id,pg_id "
          + "FROM tutorial.prozess where Name = '"
          + prozessname + "'";

      //    Anfrage ausführen
      ResultSet rs = stmt.executeQuery(query);
      while (rs.next())
```

```
    {

      Integer Iid = new Integer(rs.getInt("id"));
      Integer Ipgid = new Integer(rs.getInt("PG_ID"));

      newProzess = GetInstance(Iid.intValue(), Ipgid
          .intValue(), con);
      HT.put(new String(Iid.toString() + Ipgid.toString()),
          newProzess);
    }
    rs.close();
    stmt.close();

    return (Prozess[]) HT.values().toArray(new Prozess[0]);
  }

  /**
   * Liest einen Prozess ein.
   *
   * @param id ID des Prozesses
   * @param pg_id ID der Prozessgruppe
   * @throws SQLException
   */
  private void read(int id, int pg_id) throws SQLException
  {

    if ((id > 0) && (pg_id > 0))
    {
      //     SQL-Query konstruieren
      Statement stmt;
      stmt = con.createStatement();
      String query = "SELECT * "
          + "FROM tutorial.prozess where ID = " + id
          + " and pg_id = " + pg_id;

      //    Anfrage ausführen
      ResultSet rs = stmt.executeQuery(query);

      while (rs.next())
      {

        this.Name = rs.getString("Name");
        this.PG_id = rs.getInt("PG_ID");
        this.id = id;

      }

      // Ergebnismenge schliessen
      rs.close();

      // SQL-Anweisung schliessen
      stmt.close();
    }

  }
```

```
/**
 * Gibt die ID des Prozesses zurück.
 *
 * @return Prozess-ID
 */
public int getId()
{
  return id;
}

/**
 * Setzt die Prozess-ID.
 *
 * @param id Zu setzende ID
 */
public void setId(int id)
{
  this.id = id;
}

/**
 * Gibt den Prozessnamen zurück.
 *
 * @return Prozessname
 */
public String getName()
{
  return Name;
}

/**
 * Setzt den Prozessnamen.
 *
 * @param name Prozessname
 */
public void setName(String name)
{
  Name = name;
}

/**
 * Gibt die Prozessgruppe zurück, zu der dieser Prozess gehört.
 *
 * @return Prozessgruppe
 * @throws SQLException
 */
public Prozessgruppe getProzessgruppe()
    throws SQLException
{
  if (this.Prozessgruppe == null)
  {
    this.Prozessgruppe = Prozessgruppe.GetInstance(
        this.PG_id, this.con);
  }
```

```
    return this.Prozessgruppe;
  }

  /**
   * Gibt alle Kontrollaktivitäten zurück, die diesem Prozess
   * zugeordnet sind.
   *
   * @return Kontrollaktivitäten
   * @throws SQLException
   */
  public Kontrollaktivitaet[] getKontrollaktivitaeten()
      throws SQLException
  {
    if (this.Kontrollaktivitaeten == null)
    {
      this.Kontrollaktivitaeten = new Hashtable();

      Statement stmt;
      stmt = con.createStatement();
      String query = "SELECT KA_ID "
        + "FROM tutorial.kontrollaktivitaet_prozess where P_ID = "
        + this.id + " and pg_id = " + this.PG_id;

      //    Anfrage ausführen
      ResultSet rs = stmt.executeQuery(query);

      while (rs.next())
      {

        int id = rs.getInt("KA_ID");

        this.Kontrollaktivitaeten.put(new Integer(id),
            Kontrollaktivitaet.GetInstance(id, this.con));

      }

      // Ergebnismenge schliessen
      rs.close();

      // SQL-Anweisung schliessen
      stmt.close();
    }

    return (Kontrollaktivitaet[]) this.Kontrollaktivitaeten
        .values().toArray(new Kontrollaktivitaet[0]);
  }
}
```

A.3 Klasse Prozessgruppe

```
package eu.erras;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Hashtable;

/**
 * Beispielhafte Implementierung
 *
 * Repräsentiert eine Prozessgruppe
 */
public class Prozessgruppe implements ReportingObject
{

  private static Hashtable instances = new Hashtable();

  private String Name;

  private String Beschreibung;

  private int id;

  private Hashtable Risiken = null;

  private Connection con = null;

  /**
   * Konstruktor
   *
   * @param con Geöffnete Datenbankverbindung
   * @throws SQLException
   */
  private Prozessgruppe(Connection con) throws SQLException
  {
    this.con = con;
  }

  /**
   * Gibt eine bestimmte Prozessgruppe zurück.
   *
   * @param id ID der gesuchten Prozessgruppe
   * @param con Aktive Verbindung zur Datenbank
   * @return Prozesgruppe
   * @throws SQLException
   */
  public static Prozessgruppe GetInstance(int id,
      Connection con) throws SQLException
  {
    if (instances.containsKey(new Integer(id)))
    {
```

```
      return (Prozessgruppe) instances.get(new Integer(id));
    }

    Prozessgruppe newPG = new Prozessgruppe(con);
    instances.put(new Integer(id), newPG);
    newPG.read(id);
    return newPG;

  }

  /**
   * Liest die Prozessgruppe mit der übergeben ID aus
   * der Datenbank.
   *
   * @param id ID der Prozessgruppe
   * @throws SQLException
   */
  private void read(int id) throws SQLException
  {
    if (id > 0)
    {
      //     SQL-Query konstruieren
      Statement stmt;
      stmt = con.createStatement();
      String query = "SELECT * "
          + "FROM tutorial.prozessgruppe where ID = " + id;

      //    Anfrage ausführen
      ResultSet rs = stmt.executeQuery(query);

      while (rs.next())
      {

        this.Beschreibung = rs.getString("Beschreibung");
        this.Name = rs.getString("Name");
        this.id = id;

      }

      // Ergebnismenge schliessen
      rs.close();

      // SQL-Anweisung schliessen
      stmt.close();
    }

  }

  /**
   * Gibt die Beschreibung zurück.
   *
   * @return Beschreibung
   */
  public String getBeschreibung()
  {
    return Beschreibung;
  }
```

```
/**
 * Setzt die Beschreibung.
 *
 * @param beschreibung Zu setzende Beschreibung
 */
public void setBeschreibung(String beschreibung)
{
  Beschreibung = beschreibung;
}

/**
 * Gibt die ID zurück.
 *
 * @return ID.
 */
public int getId()
{
  return id;
}

/**
 * Setzt die ID.
 *
 * @param id Zu setzende ID
 */
public void setId(int id)
{
  this.id = id;
}

/**
 * Gibt den Namen zurück.
 *
 * @return Name
 */
public String getName()
{
  return Name;
}

/**
 * Setzt den Namen.
 *
 * @param name Name
 */
public void setName(String name)
{
  Name = name;
}

/**
 * Gibt alle Risiken zurück, die mit der Prozessgruppe
 * verknüpft sind.
 *
```

```
     * @return Risiken
     * @throws SQLException
     */
    public Risiko[] getRisiken() throws SQLException
    {
      if (this.Risiken == null)
      {
        this.Risiken = new Hashtable();

        Statement stmt;
        stmt = con.createStatement();
        String query = "SELECT R_ID "
            + "FROM tutorial.prozessgruppe_risiko where PG_ID = "
            + this.id;

        //    Anfrage ausführen
        ResultSet rs = stmt.executeQuery(query);

        while (rs.next())
        {

          int id = rs.getInt("r_ID");

          this.Risiken.put(new Integer(id), Risiko
              .GetInstance(id, this.con));

        }

        // Ergebnismenge schliessen
        rs.close();

        // SQL-Anweisung schliessen
        stmt.close();
      }
      return (Risiko[]) this.Risiken.values().toArray(
          new Risiko[0]);
    }

  }
```

A.4 Interface ReportingObject

```
package eu.erras;

/**
 *
 * Beispielhafte Implementierung
 *
 * Allgemeines Interface für alle
 * Reporting-Objekte
 */
public interface ReportingObject {}
```

A.5 Klasse Risiko

```
package eu.erras;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Hashtable;

/**
 * Beispielhafte Implementierung
 *
 * Repräsentiert ein Risiko.
 */
public class Risiko implements ReportingObject
{

  private static Hashtable instances = new Hashtable();

  private String Beschreibung;

  private int id;

  private Hashtable Prozesse = null;

  private Hashtable Risiken = null;

  private Connection con = null;

  /**
   * Konstruktor
   *
   * @param con Verbindung zur Datenbank
```

```
     * @throws SQLException
     */
    private Risiko(Connection con) throws SQLException
    {
      this.con = con;
    }

    /**
     * Gibt die Instanz mit der übergebenen ID zurück.
     *
     * @param id Risiko-ID
     * @param con Geöffnete Datenbankverbindung
     * @return Gesuchtes Risiko
     * @throws SQLException
     */
    public static Risiko GetInstance(int id, Connection con)
        throws SQLException
    {
      if (instances.containsKey(new Integer(id)))
      {
        return (Risiko) instances.get(new Integer(id));
      }

      Risiko newRisiko = new Risiko(con);
      instances.put(new Integer(id), newRisiko);
      newRisiko.read(id);
      return newRisiko;

    }

    /**
     * Liest ein Risiko aus der Datenbank.
     *
     * @param id ID des Risikos, das gelesen werden soll.
     * @throws SQLException
     */
    private void read(int id) throws SQLException
    {
      if (id > 0)
      {
        //     SQL-Query konstruieren
        Statement stmt;
        stmt = con.createStatement();
        String query = "SELECT * "
            + "FROM tutorial.risiko where ID = " + id;

        //    Anfrage ausführen
        ResultSet rs = stmt.executeQuery(query);

        while (rs.next())
        {

          this.Beschreibung = rs.getString("Beschreibung");

          this.id = id;

        }
```

```
        // Ergebnismenge schliessen
        rs.close();

        // SQL-Anweisung schliessen
        stmt.close();
      }

    }

    /**
     * Gibt die Risikobeschreibung zurück.
     *
     * @return Risikobeschreibung
     */
    public String getBeschreibung()
    {
      return Beschreibung;
    }

    /**
     * Gibt die Risiko-ID zurück.
     *
     * @return Risiko-ID
     */
    public int getId()
    {
      return id;
    }

    /**
     * Setzt die Risiko-ID
     *
     * @param id Zu setzende ID
     */
    public void setId(int id)
    {
      this.id = id;
    }

  }
```

B Definition der Datenbanktabellen

Skript für die Erstellung der Datenbanktabellen und das Einfügen der nötigen Datensätze:

```
CREATE TABLE TUTORIAL.PROZESSGRUPPE
(
ID INTeger NOT NULL,
Beschreibung Varchar(500),
Name Varchar(100),
  CONSTRAINT "PK_Prozessgruppe" PRIMARY KEY (ID)
);
CREATE TABLE TUTORIAL.PROZESS
(
PG_ID INTeger NOT NULL,

Name Varchar(100),
ID INTeger NOT NULL,
  CONSTRAINT "PK_Prozess" PRIMARY KEY (PG_ID, ID)
);

CREATE TABLE TUTORIAL.RISIKO
(
ID INTeger NOT NULL,
Beschreibung Varchar(500),
  CONSTRAINT "PK_Risiko" PRIMARY KEY (ID)
);

CREATE TABLE TUTORIAL.KONTROLLAKTIVITAET
(
ID INTeger NOT NULL,
Beschreibung Varchar(500),
Name Varchar(100),
  CONSTRAINT "PK_KA" PRIMARY KEY (ID)
);

CREATE TABLE TUTORIAL.PROZESSGRUPPE_RISIKO
(
PG_ID INTeger NOT NULL,
R_ID INTeger NOT NULL,
  CONSTRAINT "PK_PG_Risiko" PRIMARY KEY (PG_ID, R_ID)
);

CREATE TABLE TUTORIAL.RISIKO_KONTROLLAKTIVITAET
(
R_ID INTeger NOT NULL,
KA_ID INTeger NOT NULL,
  CONSTRAINT "PK_Risiko_KA" PRIMARY KEY (R_ID, KA_ID)
);

CREATE TABLE TUTORIAL.KONTROLLAKTIVITAET_PROZESS
(
KA_ID INTeger NOT NULL,
```

```
P_ID INTeger NOT NULL,
PG_ID INTeger NOT NULL,
  CONSTRAINT "PK_KA_Prozess" PRIMARY KEY (KA_ID, P_ID, PG_ID)
);

ALTER TABLE TUTORIAL.PROZESS ADD CONSTRAINT
"sind_zugeordnet" FOREIGN KEY (
PG_ID
)
REFERENCES TUTORIAL.PROZESSGRUPPE (
ID
);

ALTER TABLE TUTORIAL.PROZESSGRUPPE_RISIKO ADD
CONSTRAINT "Hilfsrelation" FOREIGN KEY (
R_ID
)
REFERENCES TUTORIAL.RISIKO (
ID
);

ALTER TABLE TUTORIAL.PROZESSGRUPPE_RISIKO ADD
CONSTRAINT "Hilfsrelation1" FOREIGN KEY (
PG_ID
)
REFERENCES TUTORIAL.PROZESSGRUPPE (
ID
);

ALTER TABLE TUTORIAL.RISIKO_KONTROLLAKTIVITAET ADD
CONSTRAINT "Hilfsrelation2" FOREIGN KEY (
KA_ID
)
REFERENCES TUTORIAL.KONTROLLAKTIVITAET (
ID
);

ALTER TABLE TUTORIAL.RISIKO_KONTROLLAKTIVITAET ADD
CONSTRAINT "Hilfsrelation3" FOREIGN KEY (
R_ID
)
REFERENCES TUTORIAL.RISIKO (
ID
);

ALTER TABLE TUTORIAL.KONTROLLAKTIVITAET_PROZESS ADD
CONSTRAINT "Hilfsrelation4" FOREIGN KEY (
P_ID,
PG_ID
)
REFERENCES TUTORIAL.PROZESS (
ID,
PG_ID
);


ALTER TABLE TUTORIAL.KONTROLLAKTIVITAET_PROZESS ADD
```

```
CONSTRAINT "Hilfsrelation5" FOREIGN KEY (
KA_ID
)
REFERENCES TUTORIAL.KONTROLLAKTIVITAET (
ID
);


insert into TUTORIAL.KONTROLLAKTIVITAET
(ID,BESCHREIBUNG,NAME) values
(1,'Prüfung der Rechnungsdaten.', 'Prüfung der Rechnungsdaten.');
insert into TUTORIAL.KONTROLLAKTIVITAET (ID,BESCHREIBUNG,NAME)
values (2,'Prüfung des Zahlungseingangs.',
'Prüfung des Zahlungseingangs.');

insert into TUTORIAL.PROZESSGRUPPE
(id, beschreibung, name) values
(1, 'Rechnungsstellung', 'Rechnungsstellung');

insert into TUTORIAL.Prozess
(PG_ID,NAME,ID) values
(1,'Rechnungsstellung Privatkunden',1);


insert into TUTORIAL.RISIKO (ID,BESCHREIBUNG)
values (1,'Rechnung wird nicht korrekt bezahlt.');
insert into TUTORIAL.RISIKO (ID,BESCHREIBUNG)
values (2,'Rechnung enthält falsche Kundendaten.');
insert into TUTORIAL.RISIKO (ID,BESCHREIBUNG)
values (3,'Rechnung enthält falsche Posten.');
insert into TUTORIAL.RISIKO (ID,BESCHREIBUNG)
values (4,'Rechnung wird nicht verschickt.');


insert into TUTORIAL.KONTROLLAKTIVITAET_PROZESS
(KA_ID,P_ID,PG_ID) values (1,1,1);

insert into TUTORIAL.KONTROLLAKTIVITAET_PROZESS
(KA_ID,P_ID,PG_ID) values (2,1,1);

insert into TUTORIAL.PROZESSGRUPPE_RISIKO
(PG_ID, R_ID) values (1,1);
insert into TUTORIAL.PROZESSGRUPPE_RISIKO
(PG_ID, R_ID) values (1,2);
insert into TUTORIAL.PROZESSGRUPPE_RISIKO
(PG_ID, R_ID) values (1,3);
insert into TUTORIAL.PROZESSGRUPPE_RISIKO
(PG_ID, R_ID) values (1,4);

insert into TUTORIAL.RISIKO_KONTROLLAKTIVITAET
(R_ID,KA_ID) values (1,2);
insert into TUTORIAL.RISIKO_KONTROLLAKTIVITAET
(R_ID,KA_ID) values (2,1);
insert into TUTORIAL.RISIKO_KONTROLLAKTIVITAET
```

```
(R_ID,KA_ID) values (3,1);
```

C XSL-FO-Quellcode

Quellcode der XSL-FO-Datei für die Transformation des Reports:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0"
  xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <xsl:template name="ueberschrift">
    <xsl:param name="ueberschrift_text" />
    <xsl:param name="colspan" />
    <fo:table-row keep-with-next="always">
      <xsl:call-template name="zelle">
        <xsl:with-param name="text">
          <xsl:value-of
            select="$ueberschrift_text" />
        </xsl:with-param>
        <xsl:with-param name="fontsize">
          14pt
        </xsl:with-param>
        <xsl:with-param name="border">
          0
        </xsl:with-param>
        <xsl:with-param name="colspan">
          <xsl:value-of select="$colspan" />
        </xsl:with-param>
      </xsl:call-template>
    </fo:table-row>
  </xsl:template>
  <xsl:template name="zelle">
    <xsl:param name="text" />
    <xsl:param name="background_color">
      #FFFFFF
    </xsl:param>
    <xsl:param name="colspan">1</xsl:param>
    <xsl:param name="fontsize">9pt</xsl:param>
    <xsl:param name="border">
      0.5pt solid black
    </xsl:param>
    <xsl:param name="text_align">left</xsl:param>
    <xsl:param name="display_align">center</xsl:param>
    <fo:table-cell border="{$border}"
      background-color="{$background_color}"
      number-columns-spanned="{$colspan}"
      number-rows-spanned="1"
```

```
      display-align="{$display_align}"
      padding-left="1.5mm" padding-top="1.5mm"
      padding-bottom="1.5mm"
      padding-right="1.5mm">
      <fo:block text-align="{$text_align}"
        font-size="{$fontsize}"
        font-weight="normal" color="#000000">
        <xsl:value-of select="$text" />
      </fo:block>
    </fo:table-cell>
  </xsl:template>
  <xsl:output encoding="UTF-8" indent="yes" />
  <xsl:template match="REPORT">
    <fo:root
      xmlns:fo="http://www.w3.org/1999/XSL/Format">
      <fo:layout-master-set>
        <fo:simple-page-master
          master-name="default"
          page-height="29.7cm" page-width="21.0cm"
          margin-top="20mm">
          <fo:region-body margin-left="20mm"
            margin-right="20mm" margin-top="15mm"
            margin-bottom="50mm" />
          <fo:region-before extent="30mm"
            precedence="false" />
          <fo:region-after extent="20mm"
            precedence="false" />
          <fo:region-start extent="20mm"
            precedence="true" />
          <fo:region-end extent="20mm"
            precedence="true" />
        </fo:simple-page-master>
        <fo:simple-page-master master-name="cover"
          page-height="29.7cm" page-width="21.0cm"
          margin-top="20mm">
          <fo:region-body margin-left="20mm"
            margin-right="20mm" margin-top="20mm"
            margin-bottom="50mm" />
          <fo:region-before extent="30mm"
            precedence="false" />
          <fo:region-after extent="40mm"
            precedence="false" />
          <fo:region-start extent="20mm"
            precedence="true" />
          <fo:region-end extent="20mm"
            precedence="true" />
        </fo:simple-page-master>
      </fo:layout-master-set>
      <xsl:call-template name="cover" />
      <xsl:call-template name="pageTemplate" />
    </fo:root>
  </xsl:template>
  <xsl:template name="pageTemplate">
    <xsl:param name="pageContentTemplate" />
    <fo:page-sequence
```

```
        master-reference="default">
        <fo:static-content
          flow-name="xsl-region-before">
          <fo:table table-layout="fixed"
            width="100%" border="0"
            padding-top="0mm"
            padding-bottom="0mm">
            <fo:table-column column-width="170mm" />
            <fo:table-body>
              <fo:table-row>
                <xsl:call-template name="zelle">
                  <xsl:with-param name="text">
                    EGL-Reporting
                  </xsl:with-param>
                  <xsl:with-param name="border">
                    0
                  </xsl:with-param>
                </xsl:call-template>
              </fo:table-row>
            </fo:table-body>
          </fo:table>
          <fo:block font-size="1pt"
            line-height="1pt" padding-top="0mm"
            padding-bottom="0mm">
            <fo:leader leader-pattern="rule"
              rule-thickness="0.5pt"
              leader-length="100%" color="black" />
          </fo:block>
        </fo:static-content>
        <fo:static-content
          flow-name="xsl-region-after">
          <fo:table table-layout="fixed"
            width="100%" border="0"
            padding-top="0mm"
            padding-bottom="0mm">
            <fo:table-column column-width="85mm" />
            <fo:table-column column-width="85mm" />
            <fo:table-body>
              <fo:table-row>
                <xsl:call-template name="zelle">
                  <xsl:with-param name="text">
                    Universität Leipzig
                  </xsl:with-param>
                  <xsl:with-param name="fontsize">
                    8pt
                  </xsl:with-param>
                  <xsl:with-param name="border">
                    0
                  </xsl:with-param>
                </xsl:call-template>
                <fo:table-cell border="0"
                  display-align="center"
                  padding-left="1.5mm"
                  padding-top="1.5mm"
                  padding-bottom="1.5mm"
```

```
                padding-right="1.5mm">
                <fo:block text-align="right"
                  font-size="8pt">
                  Seite
                  <fo:page-number />
                </fo:block>
              </fo:table-cell>
            </fo:table-row>
          </fo:table-body>
        </fo:table>
      </fo:static-content>
      <fo:flow flow-name="xsl-region-body">
        <xsl:call-template name="prozess" />
      </fo:flow>
    </fo:page-sequence>
  </xsl:template>
  <xsl:template name="cover">
    <fo:page-sequence master-reference="cover">
      <fo:flow flow-name="xsl-region-body">
        <fo:block padding-top="50mm"
          font-size="40pt" text-align="center"
          font-family="Times" color="#000000">
          EGL-Reporting
        </fo:block>
        <fo:block padding-top="10mm"
          font-size="40pt" text-align="center"
          font-family="Times" color="#000000">
          Risikoabdeckung
        </fo:block>
        <fo:block padding-top="40mm"
          font-size="40pt" text-align="center"
          font-family="Times" color="#000000" />
      </fo:flow>
    </fo:page-sequence>
  </xsl:template>
  <xsl:template name="prozess">
    <fo:table table-layout="fixed" width="100%"
      border="0" padding-top="0mm"
      padding-bottom="0mm">
      <fo:table-column column-width="45mm" />
      <fo:table-column column-width="125mm" />
      <fo:table-body>
        <fo:table-row>
          <xsl:call-template name="zelle">
            <xsl:with-param name="text">
              1. Prozessinformationen
            </xsl:with-param>
            <xsl:with-param name="colspan">
              2
            </xsl:with-param>
            <xsl:with-param name="fontsize">
              14pt
            </xsl:with-param>
            <xsl:with-param name="border">
              0
```

```
            </xsl:with-param>
          </xsl:call-template>
        </fo:table-row>
        <fo:table-row>
          <xsl:call-template name="zelle">
            <xsl:with-param name="text">
              Prozessname
            </xsl:with-param>
            <xsl:with-param name="colspan">
              1
            </xsl:with-param>
            <xsl:with-param
              name="background_color">
              #d2d2ff
            </xsl:with-param>
          </xsl:call-template>
          <xsl:call-template name="zelle">
            <xsl:with-param name="text">
              <xsl:value-of
                select="/REPORT/PROZESS" />
            </xsl:with-param>
            <xsl:with-param name="colspan">
              1
            </xsl:with-param>
            <xsl:with-param
              name="background_color">
              #e5e5ff
            </xsl:with-param>
          </xsl:call-template>
        </fo:table-row>
        <fo:table-row>
          <xsl:call-template name="zelle">
            <xsl:with-param name="text">
              Prozessgruppe
            </xsl:with-param>
            <xsl:with-param name="colspan">
              1
            </xsl:with-param>
            <xsl:with-param
              name="background_color">
              #d2d2ff
            </xsl:with-param>
          </xsl:call-template>
          <xsl:call-template name="zelle">
            <xsl:with-param name="text">
              <xsl:value-of
                select="/REPORT/PROZESSGRUPPE" />
            </xsl:with-param>
            <xsl:with-param name="colspan">
              1
            </xsl:with-param>
            <xsl:with-param
              name="background_color">
              #e5e5ff
            </xsl:with-param>
```

```
          </xsl:call-template>
        </fo:table-row>
      </fo:table-body>
    </fo:table>
    <fo:table table-layout="fixed" width="100%"
      border="0" padding-top="25mm"
      padding-bottom="0mm">
      <fo:table-column column-width="170mm" />
      <fo:table-body>
        <fo:table-row>
          <xsl:call-template name="zelle">
            <xsl:with-param name="text">
              2. Zugeordnete Kontrollaktivitäten
            </xsl:with-param>
            <xsl:with-param name="fontsize">
              14pt
            </xsl:with-param>
            <xsl:with-param name="border">
              0
            </xsl:with-param>
          </xsl:call-template>
        </fo:table-row>
        <xsl:for-each
          select="/REPORT/KONTROLLAKTIVITAET">
          <fo:table-row>
            <xsl:call-template name="zelle">
              <xsl:with-param
                name="background_color">
                #e5e5ff
              </xsl:with-param>
              <xsl:with-param name="text">
                <xsl:value-of select="." />
              </xsl:with-param>
            </xsl:call-template>
          </fo:table-row>
        </xsl:for-each>
      </fo:table-body>
    </fo:table>
    <fo:table table-layout="fixed" width="100%"
      border="0" padding-top="25mm"
      padding-bottom="0mm">
      <fo:table-column column-width="85mm" />
      <fo:table-column column-width="85mm" />
      <fo:table-body>
        <fo:table-row>
          <xsl:call-template name="zelle">
            <xsl:with-param name="text">
              3. Abdeckung der Risiken
            </xsl:with-param>
            <xsl:with-param name="colspan">
              2
            </xsl:with-param>
            <xsl:with-param name="fontsize">
              14pt
            </xsl:with-param>
```

```
            <xsl:with-param name="border">
              0
            </xsl:with-param>
          </xsl:call-template>
        </fo:table-row>
        <fo:table-row>
          <xsl:call-template name="zelle">
            <xsl:with-param name="text" />
            <xsl:with-param name="colspan">
              2
            </xsl:with-param>
            <xsl:with-param name="fontsize">
              14pt
            </xsl:with-param>
            <xsl:with-param name="border">
              0
            </xsl:with-param>
          </xsl:call-template>
        </fo:table-row>
        <fo:table-row>
          <xsl:call-template name="zelle">
            <xsl:with-param name="text">
              3.1 Abgedeckte Risiken
            </xsl:with-param>
            <xsl:with-param name="colspan">
              2
            </xsl:with-param>
            <xsl:with-param name="fontsize">
              14pt
            </xsl:with-param>
            <xsl:with-param name="border">
              0
            </xsl:with-param>
          </xsl:call-template>
        </fo:table-row>
        <fo:table-row>
          <xsl:call-template name="zelle">
            <xsl:with-param name="text">
              Risiko
            </xsl:with-param>
            <xsl:with-param
              name="background_color">
              #d2d2ff
            </xsl:with-param>
          </xsl:call-template>
          <xsl:call-template name="zelle">
            <xsl:with-param name="text">
              Kontrollaktivität
            </xsl:with-param>
            <xsl:with-param
              name="background_color">
              #d2d2ff
            </xsl:with-param>
          </xsl:call-template>
        </fo:table-row>
```

```
        <xsl:for-each
select="/REPORT/RISIKO[count(./ABDECKUNG/ABGEDECKTDURCH)&gt;0]">
          <fo:table-row>
            <xsl:call-template name="zelle">
              <xsl:with-param name="text">
                <xsl:value-of
                  select="./BESCHREIBUNG" />
              </xsl:with-param>
              <xsl:with-param name="colspan">
                1
              </xsl:with-param>
              <xsl:with-param
                name="background_color">
                #e5e5ff
              </xsl:with-param>
            </xsl:call-template>
            <xsl:call-template name="zelle">
              <xsl:with-param name="text">
                <xsl:for-each
                  select="./ABDECKUNG/ABGEDECKTDURCH">
                  <xsl:value-of select="." />
                  &#xA;
                </xsl:for-each>
              </xsl:with-param>
              <xsl:with-param
                name="background_color">
                #e5e5ff
              </xsl:with-param>
              <xsl:with-param name="colspan">
                1
              </xsl:with-param>
            </xsl:call-template>
          </fo:table-row>
        </xsl:for-each>
        <fo:table-row>
          <xsl:call-template name="zelle">
            <xsl:with-param name="text" />
            <xsl:with-param name="colspan">
              2
            </xsl:with-param>
            <xsl:with-param name="fontsize">
              14pt
            </xsl:with-param>
            <xsl:with-param name="border">
              0
            </xsl:with-param>
          </xsl:call-template>
        </fo:table-row>
        <fo:table-row>
          <xsl:call-template name="zelle">
            <xsl:with-param name="text">
              3.2 Nicht abgedeckte Risiken
            </xsl:with-param>
            <xsl:with-param name="colspan">
              2
```

```
            </xsl:with-param>
            <xsl:with-param name="fontsize">
              14pt
            </xsl:with-param>
            <xsl:with-param name="border">
              0
            </xsl:with-param>
          </xsl:call-template>
        </fo:table-row>
        <xsl:for-each
select="/REPORT/RISIKO[count(./ABDECKUNG/ABGEDECKTDURCH)=0]">
          <fo:table-row>
            <xsl:call-template name="zelle">
              <xsl:with-param name="text">
                <xsl:value-of
                  select="./BESCHREIBUNG" />
              </xsl:with-param>
              <xsl:with-param
                name="background_color">
                #e5e5ff
              </xsl:with-param>
              <xsl:with-param name="colspan">
                2
              </xsl:with-param>
            </xsl:call-template>
          </fo:table-row>
        </xsl:for-each>
      </fo:table-body>
    </fo:table>
  </xsl:template>
</xsl:stylesheet>
```