Ein universelles Software-Repository

Harry M. Sneed
CC GmbH, Flachstraße 13, 65197 Wiesbaden,
Harry.Sneed@caseconsult.com

Abstract: The following contribution describes a tool supported approach to creating a universal software repository. A universal repository is based on a data model which accomodates heterogeneous systems written in both procedural and object-oriented languages as well as database and interface description languages. In it the various semantic models – procedural, object-oriented, hierarchical and relational, map-based and web-based – are integrated into a single common architectural model. As a result, it is possible to depict all systems with UML diagrams, including the legacy systems. The common model is object-based and component-oriented. The advantage of this repository is the possibility of modelling a mixed system environment, an important prerequisite to Enterprise Application Integration.

1 Gründe für ein universales Software Repository

Große Anwenderbetriebe wie Banken, Versicherungen, Handelsgesellschaften und Behörden müssen Softwaresysteme aus verschiedenen Technologiegenerationen warten und weiterentwickeln. Das Alter der eingesetzten Anwendungen reicht von drei bis 30 Jahren. Assembler-, COBOL-, Natural-, C^{++}- und Java-Systeme existieren nebeneinander. Hinzu kommen die unterschiedlichen Datenbanksprachen von DLI über DDL und ADAWAN bis zu SQL und die verschiedensten Schnittstellensprachen wie MFS, BMS, IDL und zuletzt XML. Jede Generation einer neuen Softwaretechnologie hat ihren eigenen Baustil und jede hat ihre Bauwerke hinterlassen. Demzufolge ist die IT-Landschaft inzwischen äußerst bunt geworden. Sie ist voll mit alten Bauwerken und es kommen immer mehr neue dazu. Das wäre alles nicht so schlimm, wenn es nicht erforderlich wäre, sämtliche Anwendungssysteme – die Alten wie die Neuen – zu pflegen und sogar weiterzuentwickeln. Außerdem stehen die Betriebe heute unter Druck, die alten Systeme mit den neuen zu integrieren. Enterprise Application Integration (EAI) ist die Parole, nach der alles zusammenspielen muß, die Assembler mit Java, die VSAM-Dateien mit den relationalen Datenbanken, die CICS-BMS-Masken mit den HTML-Webseiten. Kurzum, es gibt nichts, was nicht integrierbar ist. XML und vor allem XMI, die eXtensible Metadata Language, sollen es möglich machen [WHB01].

Daraus ergibt sich aber die Notwendigkeit, alle Programm-, Oberflächen- und Datenbankarten in einem einzigen allumfassenden Metamodell zusammenzufassen. Die Darstellung soll für alle Sprachen und alle Architekturansätze gelten, denn es kommt darauf an, jedes Anwendungssystem in der gleichen, allgemeingültigen Weise abzubilden. Das Modell, das hier gewählt wurde, ist das Model Driven Architecture der OMG. Dieses ist ein objektorientiertes Modell auf der Basis von XML und wird mit XMI implementiert. Es

verbindet die Objektmodellierungsansätze von UML mit den Datenbeschreibungsansätzen von XML [GDB02]. Damit können IT-Systeme in ihrem vollen Umfang samt Datenbanken, Schnittstellen und Komponenten dargestellt werden. Sie werden unter anderem nachdokumentiert, visualisiert und nach Bedarf abgefragt. [HWS00]

Mit Hilfe eines Repository werden ausgewählte Dokumente wie z. B. Klassendiagramme, Abhängigkeitsbäume, Sequenzdiagramme und Datenverwendungsnachweise generiert. Zwecks der Systemverständigung bzw. Program Comprehension wird die Systemarchitektur grafisch dargestellt bzw. visualisiert. Es werden im Online-Modus diverse grafische Sichten auf das System angeboten – Bäume, Netze und Diagramme. Zur Beantwortung bestimmter Fragen bezüglich der Systemwartung, wie z. B. welche anderen Module betroffen sind, wenn ein bestimmtes Modul geändert wird, bieten Repositories ein Abfrage Facility an. Hier werden SQL-ähnliche Abfragen der Art SELECT * WHEN (Bedingung) behandelt und die Ergebnisse als geordnete Listen ausgegeben. Schließlich dient das Repository auch der Aufwandsschätzung von Wartungsprojekten und zwar in Verbindung mit einer Impactanalyse der betroffenen Elemente. In jeder Hinsicht ist also ein Repository ein wichtiges Hilfsmittel für die Wartung und Weiterentwicklung von IT-Systemen, aber nur, wenn alle Systemelemente und Beziehungen darin enthalten sind und wenn sie alle zum gleichen Metamodell gehören. [Mc92]

2 Die Entitäten eines universalen Repository

Das Metamodell des universalen Repository ist ein objektbasiertes Entity/Relationship-Schema, in dem alle wesentlichen Software-Systemelemente als Entitäten samt ihren Beziehungen abgebildet sind. In Anbetracht der Tatsache, daß auch nicht objektorientierte Systeme im Repository beschrieben werden, kann das Modell nicht rein objektorientiert sein. Ein solches Modell muß sehr allgemeingültig gehalten werden, um sämtliche semantischen Konstruktionen aufnehmen zu können, und zwar prozedurale wie auch objektorientierte und komponentenorientierte.

Es geht hier vorrangig darum, prozedurale und objektorientierte Anwendungssysteme einer heterogenen IT-Landschaft in einem gemeinsamen Metamodell darzustellen. Das Modell muß daher die Eigenschaften beider Welten in sich vereinigen. Reine objektorientierte Eigenschaften wie Klassen, Koppelung, Vererbung und Polymorphie müssen mit dem klassischen, strukturierten Modell auf einen gemeinsamen Nenner reduziert werden. Andererseits müssen prozedurale Eigenschaften wie Datenflüsse und Aufrufsbäume auf ein Niveau gebracht werden, damit sie mit dem Objektmodell kompatibel sind. Prozedurale, objektorientierte und auch webbasierte Architekturen müssen zusammenschmelzen.

Das Schema, das hier geschildert wird, ist nicht deduktiv nach einem abstrakten Prozeß- oder Produktmodell, sondern induktiv aus der Analyse diverser Programmiersprachen entstanden. In Anlehnung an die antike Philosophie ist dieser Entstehungsprozeß eher mit dem Bottom-Up-Ansatz von Aristoteles zu vergleichen, im Gegensatz zum deduktiven Top-Down-Ansatz von Platon. Aristoteles basierte seine Lehre auf dem Studium der physikalischen Wirklichkeit, aus der er dann allgemeingültige Schlüsse zog. [Ar55] Im Falle der Software ist die Wirklichkeit der Code. Die einzige richtige Beschreibung eines Programmes ist das Programm selbst. [DLP79] Also wird hier von den Programmen und deren syntaktischen Einheiten ausgegangen, um zu dem übergeordneten Beschreibungs-

modell zu gelangen. Syntaktische Einheiten, die alle Sprachen gemeinsam haben, sind Operanden, Operatoren, Anweisungen, Datengruppen, Variabeln und Konstanten. Prozedurale Programme haben außerdem Paragraphen bzw. Labeled Codeblöcke, Abschnitte und Module. Objektorientierte Programme haben zusätzlich Klassen, Methoden bzw. Funktionen, Schnittstellen bzw. Interfaces und Komponenten bzw. Pakete. Diese syntaktischen Bausteine ergeben die prozeduralen Entitäten des Modells. Hinzu kommen die Entitäten aus dem Datenmodell:

- Datenbanken
- Tabellen
- Attribute
- Dateien
- Satzarten
- Felder

Datenbanken und Dateien werden gleich behandelt, ebenso Tabellen und Satzarten sowie Attribute und Felder. Schließlich folgen die Entitäten aus dem Kommunikationsmodell. Module und Komponenten kommunizieren über Schnittstellen miteinander. In der Sprache C^{++} haben Funktionen eine Signatur oder einen Prototyp. In Java gibt es ausdrücklich ein Interface, hinter dem mehrere Funktionen stecken können. In den prozeduralen Sprachen haben in der Regel nur Module eine Schnittstelle wie die Linkage Section in COBOL, aber in allen prozeduralen Sprachen gibt es zusätzliche ENTRY-Anweisungen, die auf der Funktionsebene vorkommen können. Somit sind Schnittstellen auf allen prozeduralen Ebenen möglich.

Eine Schnittstelle enthält einen oder mehrere Parameter. Es kann sich um einzelne Werte, Zeiger oder ganze Datenstrukturen handeln. Um Datenflüsse im Sinne des strukturierten Designs darstellen zu können, müssen die Schnittstellen und deren Parameter erfaßt werden. In OO-Sprachen wie Java und IDL spielen die Schnittstellen eine entscheidende Rolle. Ihnen sind auch Ausnahmebedingungen zugewiesen.

Die Schnittstelle zum Benutzer ist von besonderer Art. Benutzerschnittstellen sind in der objektorientierten Welt ein Behälter mehrerer Objekte, die der Benutzer aktivieren kann (Controls), oder Objekte, die ihm gezeigt werden. In der prozeduralen Welt sind Benutzerschnittstellen Masken mit festen Feldern. Manche Felder sind Titel, manche Eingabefelder und andere sind Ausgabefelder. In beiden Welten sind aber Benutzerschnittstellen Sichten auf das darunter liegende System, Sichten mit Attributen und Beziehungen zu den internen Entitäten.

Daraus folgt, daß auch Schnittstellen diskrete Softwareentitäten mit erkennbaren Attributen sind. Sie lassen sich in externe Schnittstellen und interne Schnittstellen teilen. Interne Schnittstellen haben Parameter und Ausnahmebedingungen. Externe Schnittstellen haben Controls, Felder und Feldattribute bzw. Objekte und Objektattribute.

Interne Schnittstellen	**Externe Schnittstellen**
Parameter	Controls
Ausnahmebedingungen	Felder bzw. Objekte
	Attribute

Zunächst einmal stellt sich die Frage des Überbaus. Anwendungssysteme setzen sich aus einem oder mehreren Subsystemen und Subsysteme aus einer oder mehreren Kompo-

nenten zusammen. Ein Subsystem ist eine reine logische Unterteilung bzw. eine Menge logisch zusammenhängender Komponenten. Eine Komponente ist im klassischen Sinne ein Lademodul bzw. ein Rununit. In Online-Systemen entspricht sie einer Transaktion, in Batchsystemen einem Jobstep. Komponenten in neueren verteilten Systemen sind binär ausführbare Einheiten wie exe-Dateien oder dll's in der Windows-Umgebung. Sie sind in beiden Fällen das Ergebnis eines Compilier- und Linkprozesses. In Java werden sie als Packages bezeichnet. In klassischen Hostsystemen gelten sie als Programme, Online- oder Batch-Programme. In einem Client/Server-Umfeld finden wir Client- und Server-komponenten. In einem webbasierten System gibt es Website-Komponenten, Webserver-Komponenten, Application Server-Komponenten und Datenserver-Komponenten. In der letzten UML-Spezifikation wird eine Komponente als: „a physical, replacable part of a system that packages implementation and provides the realization of a set of interfaces. A component represents a physical piece of implementation of a system, including software code (source, library or executable) or equivalents such as scripts or command files.“ [oV99]

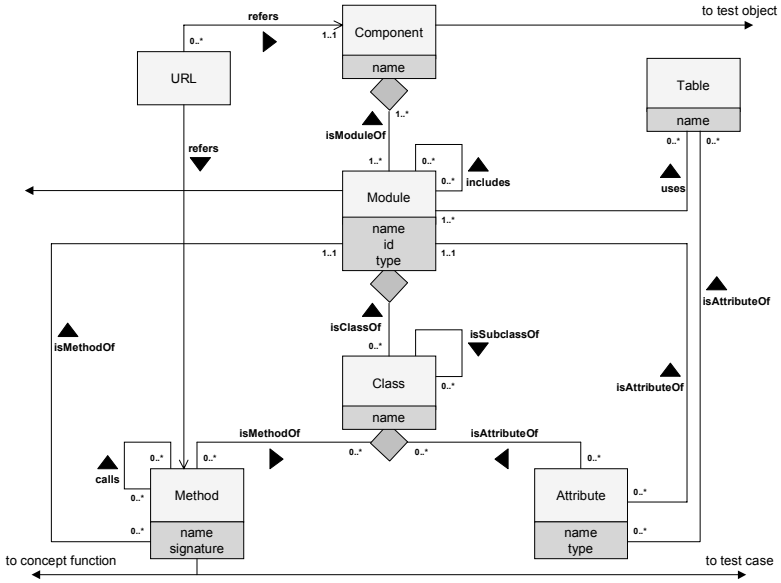
Innerhalb einer Komponente kann es ein oder mehrere Module geben. Ein Modul ist eine separate Sourcedatei. Es kann in der Regel getrennt kompiliert werden, muß aber nicht. Module haben einen globalen Datenbereich, zu dem alle Codeabschnitte des Moduls Zugriff haben. Hier hört jedoch die Gemeinsamkeit auf. In objektorientierten Sprachen sind Module in Klassen unterteilt. Ein Modul hat eine oder mehrere Klassen. In prozeduralen Sprachen können Module in Abschnitte oder Prozeduren unterteilt sein, z. B. CSECTS in ASSEMBLER, interne Prozeduren in PL/I oder Sections in COBOL. So, wie in objektorientierten Systemen ein Modul oft nur eine Klasse hat, haben Module in prozeduralen Systemen oft nur einen Abschnitt. Falls sie keinen haben, gilt das ganze Modul gleich als Abschnitt.

Klassen dienen in dem Objektmodell dazu, Attribute und Funktionen zu kapseln. Die Abschnitte oder Sections in prozeduralen Sprachen wie Assembler, COBOL und Natural haben keine eigenen Daten, wohl aber eigene Funktionen, nämlich die Paragraphen. In Blocksprachen wie ADA und PL/I können sie auch eigene Daten haben. Um prozedurale Module objektorientierten Modulen semantisch gleichzustellen, müssen ihre Abschnitte oder Sections in Klassen umgewandelt werden. Ihre kleinsten prozeduralen Einheiten, das sind ihre Paragraphen oder Labeled Blöcke, sind als Funktionen bzw. Methoden zu behandeln. Da diese über Verzweigungen im Code erreichbar sind, haben sie auch eine ähnliche Bedeutung wie Funktionen in einer Klasse mit dem Unterschied, daß sie bei GO TO-Verzweigungen nicht zum Ursprungspunkt zurückkehren, sondern weiter zur nächsten Funktion in der Sequenz durchfallen. Die globalen Daten eines Moduls müssen auf die Abschnitte verteilt werden, die sie verwenden. Nur die Daten, die von allen Abschnitten benutzt werden, bleiben in einem globalen Datenbereich bestehen, dieser wird jedoch in eine extra Datenklasse umgewandelt. Somit umfaßt jedes Modul, auch die prozeduralen Module, am Ende mindestens eine Klasse. Es entsteht mit Hilfe der syntaktischen Gleichschaltung folgende Entitätenhierarchie:

Systeme
→ Subsysteme
→ Komponente
→ Module

→ Klassen/Abschnitte
→ Funktionen

Bild 1: Das Repository Metamodel



3 Die Beziehungen eines universalen Repository

Die Entitäten eines Softwaresystems haben zwei Arten von Beziehungen zueinander: vertikale und horizontale.

Vertikale Beziehungen zeigen Zugehörigkeiten. Eine übergeordnete Entität besteht aus mehreren untergeordneten Entitäten. Eine untergeordnete Entität gehört zu einer übergeordneten Entität. Diese Beziehungsart entspricht der Komposition in dem UML-Klassendiagramm. Horizontale Beziehungen zeigen Nutzungen. Eine Entität benutzt eine andere bzw. eine Entität wird von einer anderen benutzt. Sie sind aber, was die Zugehörigkeit betrifft, unabhängig voneinander. Diese Beziehungsart entspricht einer Assoziation in UML.

In einem komplexen Softwaresystem gibt es je nach Sprache eine hohe Anzahl potenzieller Beziehungen. In einem Repository Metamodell kommt es darauf an, eine sinnvolle, aussagekräftige Untermenge dieser Beziehungen darzustellen. Die Kriterien für die Auswahl der Beziehungen ergeben sich aus der Sicht des Informationsbedarfs. Man fragt sich, was will einer wissen. Die erforderliche Einschränkung der möglichen Beziehungen beruht auf einer Einschätzung der Benutzeranforderungen.

Die häufigsten Benutzerfragen bezüglich einer Softwarearchitektur sind:

- Was gehört wohin?
- Woraus besteht etwas?

- Wer benutzt was?

Man möchte nämlich wissen, welche anderen Entitäten dadurch betroffen sind, wenn eine Entität sich ändert, oder wenn man eine Entität testet, welche anderen mit getestet werden müssen. Damit man den Aufwand schätzen kann, will man bei einem Änderungsantrag auch wissen, wie viele Entitäten betroffen sind und zu welchem Grad. Diese Fragen stehen auf der Tagesordnung eines jeden Wartungsbetriebes. Der Hauptzweck eines Repository ist, dem Wartungsbetrieb zu dienen [Le91]. Also müssen zunächst diese Fragen beantwortet werden.

Bezüglich der Zugehörigkeit der Entitätentypen gibt es folgende Beziehungen:

- Systeme enthalten Subsysteme
- Subsysteme enthalten Komponenten
- Komponenten enthalten Module
- Module enthalten Klassen/Abschnitte
- Klassen enthalten Funktionen
- Klassen enthalten Attribute
- Module enthalten Funktionen
- Module enthalten Attribute
- Module haben Schnittstellen
- Funktionen haben Schnittstellen
- Schnittstellen haben Parameter
- Schnittstellen haben Ausnahmebedingungen
- Datenbanken enthalten Tabellen
- Dateien enthalten Sätze
- Tabellen enthalten Attribute
- Sätze enthalten Felder

Diese Beziehungen beschreiben die statische Komposition eines prozeduralen und/oder objektorientierten Systems einschließlich des Funktions-, Daten- und Kommunikationsmodells.

Hinzu kommt die dynamische Sicht bezüglich der gegenseitigen Benutzung der Entitätentypen:

- Komponenten stoßen andere Komponenten an
- Module rufen andere Module auf
- Klassen erben von übergeordneten Klassen
- Klassen vererben an untergeordnete Klassen
- Funktionen rufen andere Funktionen auf
- Funktionen greifen auf Tabellen bzw. Sätze
- Funktionen bedienen Schnittstellen

- Funktionen senden Parameter
- Funktionen empfangen Parameter
- Funktionen lösen Ausnahmebedingungen auf
- Funktionen setzen, befragen oder benutzen Attribute

Natürlich lassen sich alle diese Beziehungen invertieren, um zu erfahren, was mit einer Entität geschieht, z. B. wer von wem wird aufgerufen, welcher Parameter von welcher Funktion empfangen oder welches Attribut von welcher Funktion geändert wird.

Da alle untergeordneten Entitäten einer übergeordneten Entität zugehören, ist es mittels der Transposition jederzeit möglich, zu bestimmen, welche untergeordneten Elemente von welchen übergeordneten Elementen benutzt werden, z. B. welche Attribute in welchem Modul geändert oder welche Schnittstellen von welchen Komponenten bedient werden.

Die Erfahrung in mehreren Projekten hat gezeigt, daß es mit den oben genannten statischen und dynamischen Beziehungen möglich ist, fast alle Fragen bezüglich der Architektur bzw. bezüglich der Auswirkung von Änderungen zu beantworten. Was nicht ohne weiteres zu beantworten ist, ist eine Frage nach dem fachlichen Inhalt bzw. nach den Geschäftsregeln. Solche Fragen sind deshalb nicht leicht zu beantworten, weil der fachliche Inhalt nicht unbedingt mit der technischen Struktur übereinstimmt. Manche technischen Funktionen sind 1 : 1 identisch mit fachlichen Funktionen, viele aber nicht. Oft sind fachliche Funktionen auf mehrere technische Funktionen verteilt oder umgekehrt, eine technische Funktion bedient mehrere fachliche Funktionen. Nur eine Untersuchung der Codeinhalte und Kommentare läßt darauf schließen, welche Fachlichkeiten dort behandelt werden. Die Lösung liegt in einer Zuordnungstabelle, die fachliche Funktionen und Datenentitäten mit den Codeentitäten verbindet. Diese Tabelle muß jedoch aufgebaut und gepflegt werden. Im Moment ist der Autor dabei, dies zu realisieren [Sn01a].

Dasselbe trifft für die Geschäftsregeln zu. Geschäftsregeln stecken in der Entscheidungslogik und in den einzelnen Algorithmen. Diese liegen jedoch unterhalb der Funktionsebene. Sie aufzunehmen würde das Repository unnötig aufblähen. Daher gibt es Verweise von den Funktionen zu ausgelagerten Struktogrammen, die aus dem Code generiert werden. Wer Geschäftsregeln sucht, muß sich mit den Detaildiagrammen auseinandersetzen. Geschäftsregeln sind ohnehin nicht einfach zu definieren. Erst die Nutzung einer formalen Spezifikationssprache wie OCL, die Object Constraint Language, macht es möglich, Geschäftsregeln in dem Code zu identifizieren [WK99].

4 Der Aufbau des universalen Repository

Das universale Repository ist in einer universalen DB2-Datenbank mit 20 einzelnen Tabellen gespeichert. Sie wird in regelmäßigen Abständen erneuert. Die alten Tabellen werden gelöscht und neue kreiert. Der ganze Prozeß zum Aufbau des Repository dauert drei Tage und vollzieht sich in drei Schritten.

Im ersten Schritt werden die Source-Module, die Datenbankschemen und die Schnittstellenbeschreibungen aller Anwendungssysteme analysiert und eine CSV Import File für jedes System generiert. Dies ist die Aufgabe der Sourceanalyse-Werkzeuge wie ASMAnal, COBAnal, DLIAnal, SQLAnal, CPPAnal, JAVAnal, IDLAnal und

XMLAnal. Das aus der Sourceanalyse hervorgehende CSV-File enthält alle Entitäten und ihre Beziehungen zueinander. Die Komponenten darin besitzen Module, die Module besitzen Klassen oder Sections und Datengruppen, die Klassen oder Sections besitzen Attribute und Funktionen. Die Datenbanken besitzen Tabellen, die Tabellen besitzen Schlüssel und Attribute. Die Schnittstellen besitzen Parameter, Datengruppen und Ausnahmeregeln. Die Datengruppen besitzen einzelne Datenattribute. Funktionen invokieren andere Funktionen und übergeben ihnen Parameter. Funktionen verwenden auch Daten als Prädikate, Argumente und Ergebnisse. Klassen erben von anderen Klassen und benutzen Schnittstellen. Die CSV-Dateien sind für alle Sprachen gleich aufgebaut. Es gibt immer die gleichen fünf Spalten in jeder Zeile:

- Quellenentität
- Quellentitätentyp
- Beziehungstyp
- Zielentität
- Zielentitätentyp

Bild 2: Ausschnitt aus einer Repository Import Datei

```
Type;BaseEntityName                    ;Rel ;Type;TargetEntityName
SECT;PQMC01K_DATENBANKSCHNITTSTELLE    ;OWNS;FUNC;PQMC01K_R01_FETCH310
FUNC;PQMC01K_R01_FETCH310              ;PRED;DATA;SQLCODE
FUNC;PQMC01K_R01_FETCH310              ;INPT;DATA;SQLCODE
FUNC;PQMC01K_R01_FETCH310              ;OUTP;DATA;ISS10_DB2NR
FUNC;PQMC01K_R01_FETCH310              ;OUTP;DATA;ISS10_DB2TAB
FUNC;PQMC01K_R01_FETCH310              ;OUTP;DATA;ISS10_DB2OPER
FUNC;PQMC01K_R01_FETCH310              ;OUTP;DATA;ISS10_SQLCODE
SECT;PQMC01K_DATENBANKSCHNITTSTELLE    ;OWNS;FUNC;PQMC01K_R01_COUNT
FUNC;PQMC01K_R01_COUNT                 ;PRED;DATA;ITA01_QTBRC
FUNC;PQMC01K_R01_COUNT                 ;INPT;DATA;ITA01_QTBPLA
FUNC;PQMC01K_R01_COUNT                 ;OUTP;DATA;TQ0275K
FUNC;PQMC01K_R01_COUNT                 ;PRED;DATA;SQLCODE
FUNC;PQMC01K_R01_COUNT                 ;INPT;DATA;T0275_QVLK1
```

Im zweiten Schritt werden die prozeduralen CSV-Dateistrukturen in objektorientierte CSV-Dateistrukturen umgesetzt. Aus den Paragraphen werden Methoden und aus den Abschnitten bzw. Sections werden Klassen. Außerdem wird eine Klassenhierarchie durch eine Inversion des PERFORM-Baumes erzeugt. Die Abschnitte, deren Funktionen aufgerufen werden, werden zu Basisklassen und die aufrufenden Abschnitte zu untergeordneten Klassen. Aufgrund einer Datenverwendungsanalyse werden die Daten auf die neu generierten Klassen verteilt, und zwar den Klassen auf der höchsten Stufe zugewiesen, die sie verwenden. Die darunter liegenden Klassen erben sie und die aufrufenden Klassen erben die aufgerufenen Funktionen. Auf diese Weise werden die prozeduralen Strukturen in OO-Strukturen umgesetzt. Über das genaue Verfahren dazu wurde auf dem GI Reengineering Workshop im Frühjahr 2002 berichtet [Sn02].

Im dritten Schritt werden die Entitäten und Beziehungen aus den einzelnen CSV-Dateien in die entsprechenden Datenbanktabellen geladen. Es gibt für jeden Entitätentyp und jeden Beziehungstyp eine eigene Tabelle. Die Entitätentabellen sind n-ary-Relationen mit einer Spalte für jedes Entitätenattribut. Die Beziehungstabellen sind binary-Relationen mit jeweils einer Basisentität und einer Zielentität. Zugegriffen werden sie über die Entitätennamen, die auch Ordnungsbegriffe sind. Insgesamt gibt es sieben Entitätentabellen :

- Systemtabelle
- Komponententabelle
- Modultabelle
- Klassentabelle
- Schnittstellentabelle
- Methodentabelle
- AttributenTabelle

Hinzu kommen 12 Beziehungstabellen:

- System → Komponente
- Komponente → Module
- Module → Klassen
- Klassen → Klassen
- Klassen → Schnittstellen
- Schnittstelle → Schnittstelle
- Schnittstelle → Attribute
- Klassen → Attribute
- Attribute → Attribute
- Klassen → Methoden
- Methoden → Methoden
- Methoden → Schnittstellen

Es kommen weitere Tabellen dazu, die diese Codeentitäten mit Konzeptentitäten verknüpfen, z. B. Methoden mit Anwendungsfällen und Komponenten mit Fachfunktionen, aber die zu beschreiben würde den Rahmen dieses Beitrages sprengen. Es genügt hier darauf hinzuweisen, daß Links existieren sowohl zu dem darüber liegenden Fachkonzept als auch zu den darunter liegenden Testfällen.

5 Der Nutzen des universalen Repository

Der Hauptnutzen eines derartig universalen Software Repository liegt in der Modellierung der alternativen Integrationsmöglichkeiten. Durch die Einbeziehung der Legacy-Systeme ist es möglich zu erkennen, welche darin begrabenen Bausteine wieder zu verwenden sind und wie sie eingebunden werden können. Die Wrapper-Software läßt sich sogar aus dem Repository heraus generieren. Auch die XML-Schnittstellen sind generierbar. In dieser Hinsicht dient das Repository dem System Softwrap als Datenbasis für die Kap-

selung. [Sn01b] Insofern dient das Repository nicht nur zur Information. Sie trägt aktiv dazu bei, die diversen Systeme miteinander zu integrieren.

Zum Zweiten dient das Repository als Datenbasis für die Aufwandsschätzung von Wartungsprojekten. Über das Repository werden die Entitäten identifiziert, die durch eine Änderung im System betroffen sind. Daraus errechnet das Tool RepoCalc die Anzahl Function-Points, Object-Points und Anweisungen in dem Impactbereich des geplanten Projektes, um zu einem Wartungsaufwand zu kommen. Zu diesem Thema wurde auf der ICSM 2001 bereits berichtet [Sn01c].

Zum Dritten dient das Repository als Data Warehouse für die Bearbeitung von Abfragen zur Struktur der Anwendungssysteme. Fragen nach der Verwendung von Attributen oder Funktionen werden über SQL-Abfragen beantwortet. Auch Fragen nach der Klassenvererbung und der Assoziation werden beantwortet. Ein Wartungsprogrammierer erhält also relativ schnell Auskunft über Systemzusammenhänge. Darüber wurde auf dem IWPC 1999 berichtet. [SD99]

Schließlich dient das Repository als Datenbasis für die Nachdokumentation der bestehenden Systeme und zwar ausgehend vom aktuellen Codezustand. Es werden nach Bedarf nicht nur verschiedene UML-Dokumentenarten sondern auch klassisch strukturierte Dokumentenarten bereitgestellt. Diese Dokumente bilden den Rahmen für die technische Systemdokumentation und können darüber hinaus als XMI-Dokumente eine Brücke zu anderen CASE-Werkzeugen herstellen. Mit XMI als Grundmodell gibt es zahlreiche Möglichkeiten, die im universalen Repository enthaltenen Metadaten wiederzuverwenden. Dies ist nur der erste Schritt in Richtung voll integrierter Systembeschreibungen [SS02].

Literaturverzeichnis

[Ar55] Aristoteles: Politik und Staat der Athener – Erstes Buch, Artemis, Zürich 1955

[DLP79] DeMillo,D./Lipton,G./Perlis,A.: „Social Processes and Proofs of Theorems and Programs“, Comm. of ACM, Vol. 22, No. 5, Mai 1979

[GDB02] Grose,T. J./Doney,G. C./Brodsky,S. A.: Mastering XMI, John Wiley & Sons, New York 2002

[HWS00] Holt,R./Winter,A./Schuerr,A.: „GXL – Toward a standard Exchange Format“, Proc. of WCRE 2000, IEEE Computer Society Press, Queensland, Nov. 2000, p. 162-171

[Le91] Levkovits,H.: IBM’s Repository Manager, QED Information Sciences, Wellesley, MA., 1991, p. 123-152

[Mc92] McClure,C.: The Three R’s of Software Automation – Reengineering, repository, reusability, Prentice-Hall, Englewood Cliffs, 1992, p. 157-220

[oV99] UML Revision Task Force, OMG Unified Modelling language Specification v. 1.3, document ad/99-06-08, Object Management Group, London, June 1999

[SD99] Sneed,H./Dombovari,T.: „Comprehending a complex, distributed, object-oriented Software System“, Proc. of IWPC 1999, Computer Society Press, Pittsburgh, May, 1999, p. 218-225

[Sn01a] Sneed, H.: „Extracting Business Logic from existing COBOL Programs as a Basis for Redevelopment“, Proc. of IWPC 2001, IEEE Computer Society Press, Toronto, May, 2001, p. 176-188

[Sn01b] Sneed,H.: „Wrapping COBOL Programs behind an XML-Interface“, Proc. of WCRE 2001, IEEE Computer Society Press, Stuttgart, Okt. 2001, p. 189-197

[Sn01c] Sneed, H.: „Impact Analysis of Maintenance Tasks in distributed systems“, Proc. of ICSM 2001, IEEE Computer Society Press, Florence, Nov. 2001, p. 180-189

[Sn02] Sneed,H.: „Transformierung prozeduraler Programmstrukturen in objektorientierte Klassenstrukturen zum Aufbau einer gemeinsamen Software

[SS02] Sneed, H./Sneed, S.: Web-basierte Systemintegration mit XML, dpunkt, Heidelberg 2002

[WHB01] Weitzel,T./Harder,T./Buxmann,P.: Electronic Business und EDI mit XML, dpunkt, Heidelberg 2001

[WK99] Warner, J./Kleppe,A.: The Object Constraint Language – Precise Modelling with UML, Addison-wesley, Reading, Mass. 1999