UNIVERSITÄT LEIPZIG

Wirtschaftswissenschaftliche Fakultät
Fachbereich Wirtschaftsinformatik

Extraktion statischer SAP-Strukturinformationen in FAMIX als Grundlage für die Softwarevisualisierung

Bachelorarbeit zur Erlangung des akademischen Grades
Bachelor of Science – Wirtschaftsinformatik

vorgelegt von:

Tilmann Sager

ts99nami@studserv.uni-leipzig.de

Matrikelnummer 3705682

Erstgutachter Prof. Dr. Ulrich Eisenecker
Zweitgutachter Dipl.-Wirt.-Inf. Pascal Kovacs

Leipzig, den 5. September 2018

Abstrakt

Softwarevisualisierung stellt als Teil der Informationsvisualisierung Struktur, Verhalten und Evolution einer Software dar. In der Softwareentwicklung bietet sie für jeden Akteur ein nützliches Werkzeug. So können neben Entwicklern auch Berater und Kunden Eigenschaften einer Software beurteilen, ohne höhere Programmierkenntnisse zu besitzen.

Die Professur für Softwareentwicklung für Wirtschaft und Verwaltung der Universität Leipzig forscht im Bereich der Softwarevisualisierung. Neben einem Softwarevisualisierungsgenerator existieren bisher Extraktoren für C#, Java und Ruby. Für die SAP-eigene Programmiersprache ABAP und weitere SAP-Strukturinformationen sind in der Literatur und im kommerziellen Bereich wenige Konzepte veröffentlicht worden.

Das Ziel dieser Arbeit ist die Entwicklung eines Prototypen zur Extraktion von SAP- Strukturinformationen in das Metamodell FAMIX als Grundlage für die Softwarevisualisierung. Als Werkzeug kommt dabei der SAP Code Inspector zum Einsatz, um Informationen über Quelltextelemente zu sammeln. Die zu extrahierenden Informationen werden durch eine Analyse der *Recursive Disk Metaphor* ermittelt, bei der SAP-Elemente in die bestehende Version integriert werden.

Keywords SAP, ABAP, SAP Code Inspector, Softwarevisualisierung, Recursive Disk Metaphor, City Metaphor, Extraktion, FAMIX

Danksagung

Ich möchte an dieser Stelle in erster Linie meinem Betreuer Pascal Kovacs für seine Geduld, die zahlreichen Verbesserungen und die guten Diskussionen danken.

Weiterhin danke ich den Mitgliedern des Teams GSOU-D. Erst als Praktikant und dann als Werkstudent wurde ich mit vielen Hinweisen, Gesprächen und Wissen unterstützt.

Außerdem möchte ich der GiSA GmbH für die Kooperation und die technische Ermöglichung dieser Arbeit danken. Zuguterletzt spreche ich meinen Dank allen Teilnehmern des Forschungskolloquiums für die konstruktive Kritik, die Hinweise und die Verbesserungsvorschläge aus.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

3D	Dreidimensional
4GL	Fourth Generation Language
ABAP	Advanced Business Application Programming
CRM	Costumer Relationship Management
DDIC	Data Dictionary
EE	Enterprise Edition
ERP	Enterprise Resource Planning
FAMIX	FAMOOS Information Exchange Model
GUI	Graphic User Interface
HCM	Human Capital Management
ICI	Imperial Chemical Industries
ISU	Industry Solution Utilities
IT	Informationstechnik
LOC	Lines Of Code
RDM	*Recursive Disk Metaphor*
RFC	Remote Function Call
SAP	Systeme, Anwendungen, Produkte
SCI	SAP Code Inspector
SCM	Supply Chain Management
SE	Societas Europaea
SQL	Structured Query Language
WebGL	Web Graphics Layer
X3DOM	Extensible 3-DOM
X3D	Extensible 3D
XML	Extensible Markup Language

1 Einleitung

Im Zuge der Digitalisierung setzt eine wachsende Zahl von Unternehmen IT-Standardlösungen zur Unterstützung von Geschäftsprozessen ein. SAP, der weltweit größte Hersteller für Unternehmensstandardsoftware, bietet einen großen Umfang an Softwarelösungen. Die gewählte Standardsoftware wird bei der Einführung in das Unternehmen an die individuellen internen Vorgaben und Prozesse angepasst. Die Anpassung übernimmt entweder die unternehmenseigene IT-Abteilung oder ein externes Software-Unternehmen wird mit dieser Aufgabe betraut. Um den Auftraggeber über die entstehende Softwarestruktur, der konkreten oder abstrakten Ausführung und ihrer Evolution zu informieren, bietet sich Softwarevisualisierung als Werkzeug an (vgl. S. Diehl, 2003, S.257f). Als Teil der Informationsvisualisierung ermöglicht es Softwarevisualisierung, die Komplexität existierender Softwaresysteme zu veranschaulichen, indem sie sich verschiedene Formen der Darstellung zu Nutze macht (vgl. Knight & Munro, 1999, S.2).

1.1 Motivation

Das Institut für Wirtschaftsinformatik an der Universität Leipzig, insbesondere die Professur für Softwareentwicklung für Wirtschaft und Verwaltung, forscht im Gebiet der Softwarevisualisierung. Innerhalb dieser Forschung werden unter anderem zwei Darstellungsformen (Metaphern) untersucht: die *Recursive Disk Metaphor* (RDM) und die City Metaphor. Die *Recursive Disk Metaphor* (RDM), wie in Abbildung 1.1 (A) dargestellt, stellt eine am Institut, speziell für die Abbildung aller Aspekte von Software, entwickelte Metapher dar. Sie stellt Entitäten als kreisförmige Glyphen dar. Eine Klasse wird als Kreis abgebildet, welcher sich wiederum in innere Kreise unterteilen kann, die beispielsweise innere Klassen darstellen (vgl. Müller & Zeckzer, 2015, S.172f).

(A) (B)

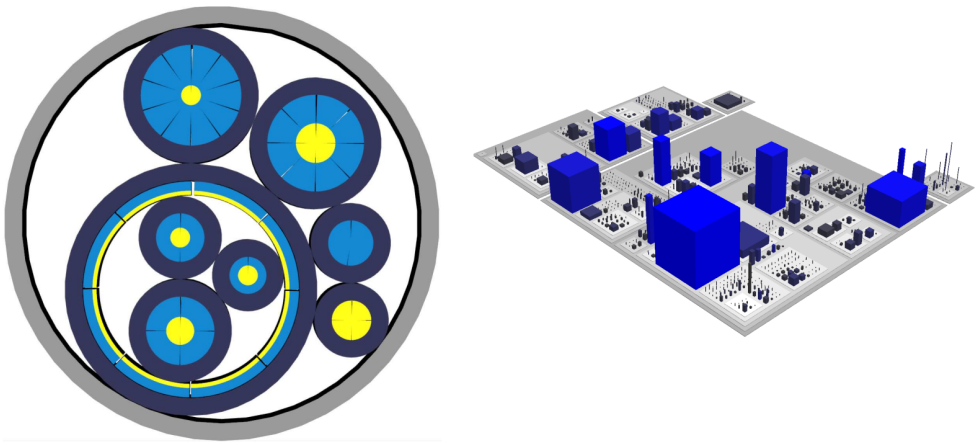

Abb. 1.1: (A) *Recursive Disk Metaphor* (aus Müller & Zeckzer, 2015, S.173, Abb. 1),
(B) City Metaphor (aus Zilch, 2015, S.10, Abb. 3.1)

Die City Metaphor, auch Stadtmetapher, eine in der Forschungsgemeinde verbreitete Metapher, siehe Stephan Diehl (2007), Balogh et al. (2016) oder Wettel et al. (2011), bildet Entitäten auf Gegenstände ab, die dem Menschen aus seinem realen Umfeld einer Stadt bekannt sind (vgl. Zilch, 2015, S.9). Wie in Abbildung 1.1 sichtbar, werden Klassen in der City Metaphor als Hochhäuser dargestellt. Ihre Höhe wird durch die Anzahl der in der Klasse enthaltenen Methoden beschrieben. In Müller (2015) wird ein Programm beschrieben, nachfolgend Generator genannt, welches aus den extrahierten Strukturinformationen ein X3D-Model generiert. Der Generator basiert auf Xtext und Xtend, der in mehreren Transformationen die Visualisierung erstellt (vgl. Kovacs, 2010, S.69).

Der Generator ermöglicht bisher die Visualisierung der Programmiersprachen Java, Ruby und C# (vgl. Müller et al., 2011, S.4). Diese Arbeit beleuchtet die Extraktion und Bereitstellung der SAP-Strukturinformationen, sowie ein Konzept einer möglichen Darstellung in der *Recursive Disk Metaphor*, um eine Möglichkeit aufzuzeigen, wie eine Auswahl an SAP-Strukturinformationen dargestellt werden könnte. Auf der Grundlage der extrahierten SAP-Strukturinformationen wird im Rahmen einer weiteren Bachelorarbeit von Johannes Roth in Roth (2017) ein Ansatz zur Visualisierung in der City Metaphor inklusive der Implementierung untersucht.

Zusammenfassend bietet die Extraktion der SAP-Strukturinformationen und der Visualisierungsansatz in der *Recursive Disk Metaphor* Einsatzmöglichkeiten in Unternehmen und in der Forschung. Es ermöglicht, Strukturen im SAP-System ohne hohe Programmierkenntnisse zu erkennen und zu analysieren. Dieser Prototyp kann als Grundlage weiter ausgebaut werden.

1.2 Zielstellung

Die Zielstellung dieser Arbeit gliedert sich in zwei Bereiche. Zuerst soll untersucht werden, welche Informationen für eine hinreichende und vollständige Visualisierung in der *Recursive Disk Metaphor* benötigt werden. Darauf aufbauend sollen diese SAP-Strukturinformationen aus einem SAP-System extrahiert werden können.

Im ersten Bereich werden folgende Teilziele definiert:

1. Integration ABAP-Entitäten in die vorhandene RDM-Version und
2. Erweiterung der *Recursive Disk Metaphor* um SAP spezifische Entitäten.

Teilziel 1 beschreibt die Untersuchung der benötigten Informationen für eine vollständige Darstellung in der *Recursive Disk Metaphor*. Dazu müssen erst relevante SAP-Strukturinformationen selektiert werden. Die zu ermittelnden Informationen werden in Zusammenarbeit mit Johannes Roth festgelegt. Diese Arbeit ist Grundlage für Roth (2017), in der die Erweiterung des Generators nach Müller et al. (2011) für SAP-Strukturinformationen untersucht wird. Im Anschluss werden die Informationen in die vorhandene RDM-Version integriert. Aufbauend auf Teilziel 1 ist das Teilziel 2 die Erweiterung der *Recursive Disk Metaphor* um die SAP-Strukturinformationen, die nicht in die bestehende RDM-Version integriert werden konnten.

Der zweite Bereich beinhaltet folgende Teilziele:

1. Anbindung des Prototypen an den SAP Code Inspector, um SAP-Strukturinformationen zu sammeln,
2. Anreicherung um DDIC- und Metainformationen,
3. Umsetzung einer iterativen Ermittlung und
4. Erweiterung der FAMIX-Grammatik.

Teilziel 1 bildet die Grundlage für alle anderen Teilziele. Der Prototyp extrahiert die durch den SAP Code Inspector gesammelten Strukturinformationen. Diese werden in Teilziel 2 um spezifische Informationen ergänzt, die nicht durch den SAP Code Inspector ermittelt werden. Teilziel 3 sieht einen iterativen Aufruf des SAP Code Inspector vor. Dieser dient zur Ermittlung von umfassenderen SAP-Strukturinformationen, beispielsweise Aufrufen von Entitäten außerhalb der selektierten Menge. In Teilziel 4 wird die Grammatik des Extraktionsformats definiert. Unter Verwendung des Meta-Models FAMIX wird die bestehende Grammatikdefinition von Müller (2015) um ABAP-Entitäten erweitert.

1.3 Methodisches Vorgehen

Die Entwicklung des Prototypen teilt sich in zwei Schritte. Der erste Schritt beinhaltet die Ermittlung aller relevanten Informationen. Auf Grundlage dieser Erkenntnisse wird im zweiten Schritt der Prototyp entwickelt. Im ersten Schritt wird mithilfe eines Konzeptansatzes für die Visualisierung der SAP-Strukturinformationen in der *Recursive Disk Metaphor* untersucht, welche Daten der Prototyp ermitteln und in das FAMIX-Format schreiben muss. Der Ansatz richtet sich nach Roth (2017), der eine Integration von SAP-Elementen in die City Metaphor untersucht. Auf Grundlage von Müller & Zeckzer (2015) werden die Erkenntnisse von Roth (2017) über Visualisierungsform und Anordnung in den Konzeptansatz integriert.

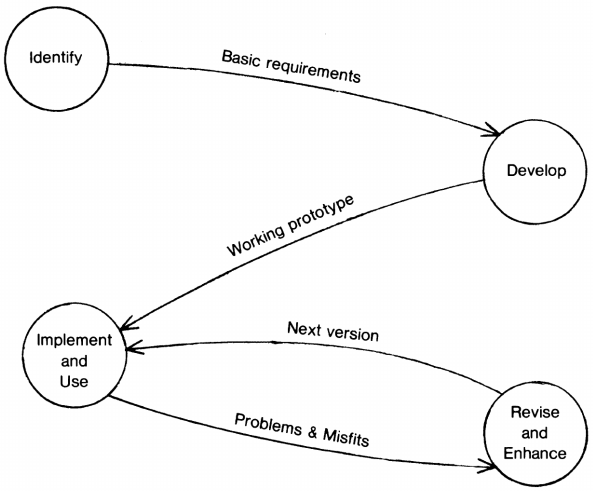

Abb. 1.2: Vorgehen Prototyping (aus Naumann & Jenkins, 1982, S.2, Fig. 1)

Auf Grundlage der Analyseergebnisse aus Schritt eins wird im zweiten Schritt der Prototyp entwickelt. Für die Entwicklung des Prototypen wird das Prototyping nach Naumann & Jenkins (1982) angewendet. Das Prototyping beschreibt ein Vorgehen, bei welchem zunächst die essentiellen Funktionalitäten eines späteren Systems entwickelt werden. Darauf folgend wird dieses System je nach Anforderung erweitert, ergänzt oder ersetzt (vgl. Carr & Verner, 2004, S.4f). Dadurch bildet es eine Grundlage für spätere Einsatzmöglichkeiten und gibt eine Lösungsmöglichkeit vor. Die Abbildung 1.2 zeigt die einzelnen vier Teilschritte:

1. Identifikation der Anforderungen,
2. Entwicklung eines elementaren Prototypen,
3. Implementierung und Nutzung und
4. Verbesserung und Korrektur des Prototypen.

Die Anforderungsanalyse stellt die Grundlage des gesamten Prozesses dar. Die Anforderungen leiten sich aus folgenden Fragen ab:

- Welche Entitäten sind vorhanden?
- Welche Entitäten sollen visualisiert werden?
- Welche Informationen werden benötigt, um eine veranschaulichende und möglichst vollständige Darstellung dieser Entitäten im Modell mittels einer Metapher zu ermöglichen?
- Wie können diese Informationen, insbesondere ABAP-spezifische Informationen, im FAMIX-Format definiert werden?
- Wie wird dieses spezielle FAMIX-Format erzeugt?

Auf den Anforderungen aufbauend wird ein elementarer Prototyp mit grundlegenden Funktionalitäten entwickelt. Dieser Prototyp ist an den SAP Code Inspector angebunden und kann SAP- Strukturinformationen extrahieren und in ein lokales FAMIX-Dokument schreiben. Zudem wird eine eine erste Version der FAMIX-Grammatik definiert, welche die Struktur und genügend Informationen beinhaltet, um eine Visualisierung der extrahierten Informationen zu ermöglichen. In den nächsten beiden Schritten wird der elementare Prototyp stetig erweitert, getestet und gegebenenfalls refaktorisiert. Mit jeder Iteration und jeder neuen Funktionalität wird der Umfang des FAMIX-Formats ausgebaut. Neue Anforderungen können während der Überarbeitung, durch Recherche oder in Zusammenarbeit mit Johannes Roth identifiziert werden. In der nächsten Iteration werden die Neuanforderungen implementiert und anschließend getestet. Neben dem Testen der Funktionalitäten wird in größeren Abständen auch die Darstellung der Entitäten in der City Metaphor getestet, um weitere mögliche Anforderungen zu identifizieren.

1.4 Aufbau dieser Arbeit

Nachdem in Kapitel 1 das Ziel definiert wurde, SAP-Strukturinformationen zu extrahieren und einen Ansatz zur Visualisierung in *Recursive Disk Metaphor* zu finden, werden im Kapitel 2 und 3 die Grundlagen zu SAP und der Softwarevisualisierung aufgezeigt. Das Kapitel 2 vermittelt die Grundlagen eines SAP-Systems, der SAP-eigenen Programmiersprache ABAP und dem SAP-Werkzeug SAP Code Inspector, um die Lesenden an die Struktur und den Aufbau von SAP heranzuführen. Darauf folgen in Kapitel 3 neben einer Einführung in die Softwarevisualisierung eine kurze Vorstellung der verwendeten Technologien, der FAMIX-Grammatik und der *Recursive Disk Metaphor*. Aufbauend auf den Grundlagen aus dem zweiten und dritten Kapitel werden im Kapitel 4 die zu extrahierenden Strukturinformationen analysiert. Dazu wird die Einbindung von SAP-Elementen in die *Recursive Disk Metaphor* untersucht, um die zu ermittelnden Informationen zu spezifizieren und die Informationsermittlung abschätzen zu können. Die Analyseergebnisse bilden den Ausgangspunkt für die Beschreibung des Prototypen in Kapitel 5. Dieses Kapitel erläutert das Konzept und die Umsetzung des Prototypen in einem SAP-System. Abschließend werden der Prototyp evaluiert und noch bestehende Probleme aufgezeigt. Kapitel 6 gibt einen Ausblick, welche Verbesserungen und Erweiterungen noch in den Prototypen implementiert werden können.

2 SAP-Grundlagen

Die SAP SE ist eine deutsche Aktiengesellschaft mit Sitz in Walldorf. Ihr Hauptprodukt, SAP ERP, welches aus dem SAP R/3 hervorging, ist mit 14% Marktanteil eine der verbreitetsten Standardsoftware zur Unterstützung unternehmerischer Prozesse (Statista, 2013). Das Enterprise Resource Planning (ERP)-System, siehe Abbildung 2.1, ist in verschiedene Module gegliedert: Modul MM für die Materialwirtschaft, Modul SD für den Verkauf und den Vertrieb, Modul PP für die Produktionsplanung und Modul HCM, auch HR genannt, für die Personalverwaltung eines Unternehmens (vgl. Mormann, 2014, S.75f). Die einzelnen Module sind über Schnittstellen verknüpft und bilden zusammen ein komplexes System. Die Grundidee dieses Systems entstand aus dem Gedanken, dass betriebswirtschaftliche Prozesse unterschiedlicher Unternehmen eine Schnittmenge in ihrer Wertschöpfungskette haben (vgl. Mormann, 2014, S. 76f). 1972 wurde auf einem Großrechner von Imperial Chemical Industries (ICI) das erste Modul RF für Rechnungs- und Finanzwesen entwickelt, welches die Grundlage für das SAP R/1 war, wobei R für *Realtime* (engl. Echtzeit) steht. Mit der 1982 beginnenden Internationalisierung wurde die Version R/2 vor allem in grenzüberschreitend tätigen Konzernen eingesetzt. Bis Mitte der 1990er entwickelte sich die dritte Version SAP R/3 zum *"de facto-Standard für mittlere und große Unternehmen"* (s. Mormann, 2014, S.78), was bis zur Abgabe dieser Arbeit gilt.

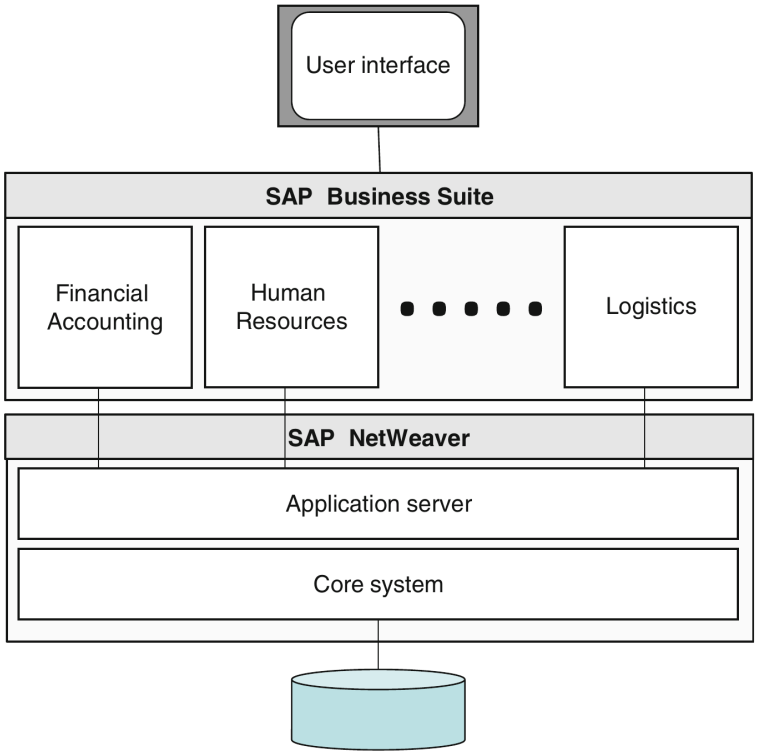

Abb. 2.1: SAP-Architektur am Beispiel der SAP Business Suite (aus Kappauf et al., 2012, S.6, Abb. 2.1)

Neben dem ERP-System bietet SAP noch weitere Systeme zur Abbildung von Unternehmensprozessen an. Zu nennen ist das Costumer Relationship Management (CRM)-System, welches vor allem für die Kundenbetreuung ausgelegt ist und dafür Daten mit dem ERP-System austauscht. Das Supply Chain Management (SCM)-System bindet die Lieferkette des Unternehmens an das ERP-System an.

Genauso existieren branchenspezifische Lösungen wie das Industry Solution Utilities (ISU)-System, welches vor allem für die Wasser-, Gas- und Stromindustrien sowie für die Abfallwirtschaft entwickelt wurde.

2.1 SAP-System

Eine SAP-Systemlandschaft ist ein Verbund aus mehreren SAP-Systemen. Ein SAP-System besteht wiederum aus mehreren Modulen. Ein Modul kapselt eine bestimmte Funktionalität. Beispielsweise übernimmt das Modul Human Capital Management (HCM) spezielle Aufgaben im Personalwesen. Diese Module sind über die SAP-NetWeaver-Plattform miteinander verbunden. Der SAP-NetWeaver bildet ein effizientes Bindeglied zwischen allen Modulen des SAP-Systems (vgl. Jens Präkelt (GiSA GmbH), 2014, S.6).

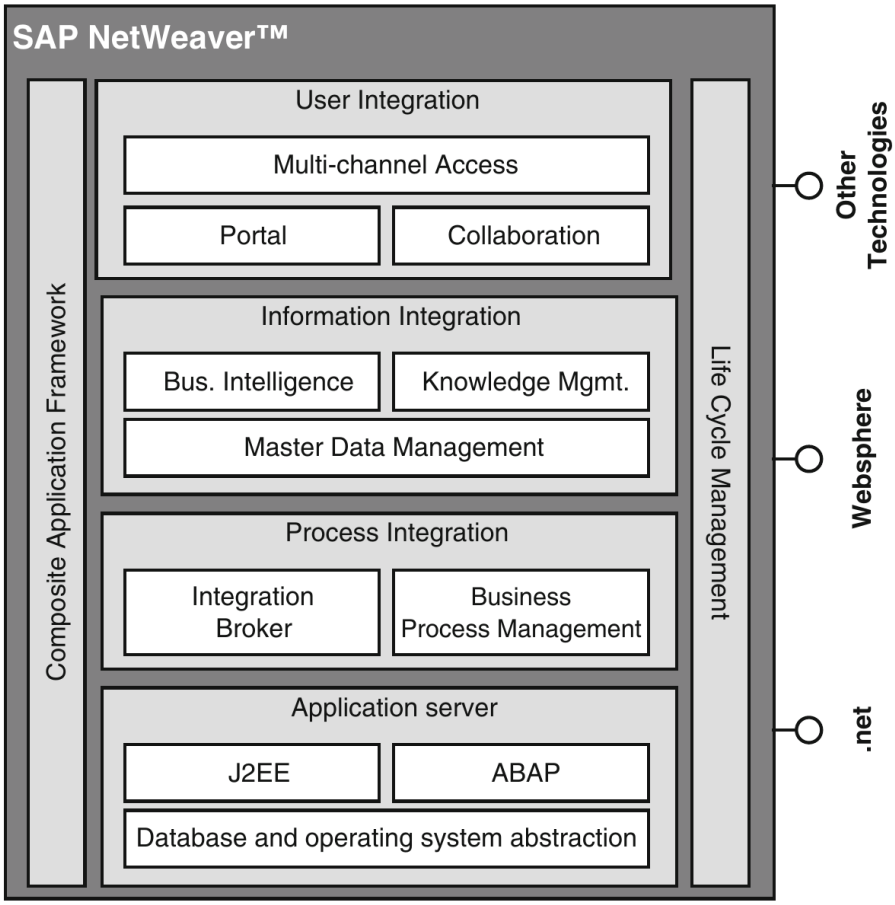

Abb. 2.2: Funktionen des SAP-NetWeaver (aus Kappauf et al., 2012, S.13, Abb. 2.3)

Die Abbildung 2.2 verdeutlicht den Aufbau und Aufgaben des SAP-NetWeavers. Der *Application Server* bildet die Grundlage des SAP-Systems. Dafür abstrahiert er die Datenbank- und Betriebssystemfunktionen und unterteilt sich in einen ABAP- und einen Java EE-Stack. *Process Integration* vereint die Prozesse verschiedener Applikationen, während *Information Integration* sich um die Datenhaltung und Informationsbereitstellung kümmert. *User Integration* bereitet das Graphic User Interface für den Anwender unter Nutzung der darunter liegenden Schichten auf. Die SAP-NetWeaver-Plattform ermöglicht zudem die Einbindung SAP-fremder Software, um weitere Funktionalitäten

dritter Anbieter nutzen zu können. Ein Beispiel ist VirtualForge (2017), welches den ABAP-Entwickler bei der Identifikation und Korrektur von Fehlern und Schwachstellen unterstützt.

SAP empfiehlt für die Entwicklung und Erweiterung des SAP-Standards eine dreistufige Systemlandschaft mit den Komponenten Entwicklungs-, Qualitätssicherungs- und Produktivsystem. Jeder dieser drei eigenständigen Stufen bildet die komplette Ziellandschaft ab (vgl. Jens Präkelt (GiSA GmbH), 2014, S.6).

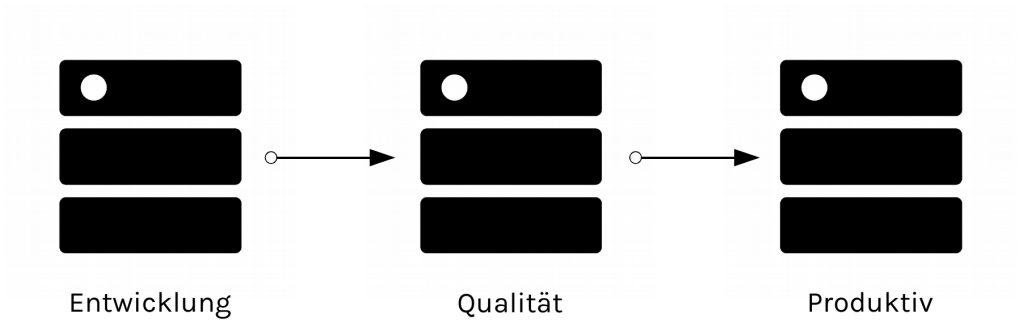

Abb. 2.3: SAP-Systemlandschaft für Entwicklung (erstellt nach Jens Präkelt (GiSA GmbH), 2014, S.6, Abb. 1)

Im Entwicklungssystem (auch D-System) werden die Erweiterungen des SAP-Standards entwickelt. Über Transportwege werden die Erweiterungen in das Qualitätssystem (auch Q-System) übertragen und getestet. Hier sollen keine neuen Änderungen mehr vorgenommen werden, sondern ausschließlich die implementierten Erweiterungen getestet werden, um eine Konsistenz des Quelltextversionen in den verschiedenen Systemen sicherzustellen. Nach erfolgreichem Testen werden die Erweiterungen in das Produktivsystem (auch P-System) transportiert, wo sie in den produktiven Geschäftsprozess des Unternehmens eingebunden werden.

2.2 ABAP

Advanced Business Application Programming (ABAP) ist die proprietäre Programmiersprache des SAP-Systems. Ihren Ursprung besitzt die Programmiersprache im R/1-System bei ICI. Der Allgemeine Berichtsaufarbeitungsprozessor, kurz ABAP, sollte das Auswerten von Daten mit festgelegten Kriterien erleichtern (vgl. Mormann, 2014, S.79). Zunächst prozedural gehalten, wurde ABAP mit der Zeit zu einer höheren Programmiersprache ausgebaut, die mittlerweile objektorientiertes Programmieren ermöglicht. Im Zuge dieser Weiterentwicklungen fand eine Umbenennung in Advanced Business Application Programming (ABAP) statt.

Das gesamte SAP-System ist auf Basis von ABAP geschrieben. Das bringt mehrere Vorteile mit sich. Einerseits besitzt SAP die volle Kontrolle über das Ausmaß der Programmiersprache und deren Implementation, was eine hohe Qualität sicherstellt. Andererseits ist der Quelltext der ABAP-basierten Applikationen und der Basissysteme vollständig zugänglich. Das heißt, auch SAP-Standardprogramme

können direkt im eigenen Quelltext aufgerufen und genutzt werden. Verbesserungen der Programmiersprache können einfach integriert werden, ohne Gefahr zu laufen, mit SAP-Wartungen und Korrekturen zu kollidieren (vgl. Kappauf et al., 2012, S.15).

ABAP besitzt zudem eine Abwärtskompatibilität. Das heißt, ein neuer Befehl ersetzt einen älteren Befehl nicht, sondern beide Befehlsvarianten sind verwendbar. Befehle werden nur als obsolet gekennzeichnet, wenn sie sicherheitskritisch sind. Daher müssen Entwickler auch darauf achten, mit welcher Version von ABAP, beziehungsweise SAP, sie entwickeln. Die Abwärtskompatibilität ergibt sich aus SAP-Systemen, die aus betriebstechnischen Gründen nicht aktualisiert werden können. Eine Bank kann ihr SAP-Buchungssystem nicht kurzzeitig außer Betrieb nehmen, um ihren Zahlungsverkehr nicht zu gefährden.

ABAP besitzt zudem den Status einer Fourth Generation Language (4GL), die mit wenig Quelltext eine hohe Funktionalität ermöglichen soll. Eine Fourth Generation Language definiert sich vor allem über einheitliche Sprachelemente, um den Wechsel zwischen Programmiersprachen innerhalb eines Projekts zu vereinfachen. Zu diesen einheitlichen Sprachelementen zählen (vgl. Jens Präkelt (GiSA GmbH), 2014, S.9):

- OpenSQL als eine in ABAP integrierte Datenbanksprache,
- eine SAP-Pufferung zur Performanzoptimierung von Datenbankzugriffen,
- interne Tabellen für die dynamische Speicherung und Bearbeitung von tabellarischen Massendaten im Arbeitsspeicher,
- eine integrierte Schnittstelle zu anderen Programmierumgebungen über Remote Function Call (RFC) und
- eine integrierte Schnittstelle zu XML.

.

```
1  REPORT z_example.
2
3  *********************************************************
4  * DECLARATION SECTION
5  *********************************************************
6
7  DATA: lv_cond TYPE abap_bool,
8        lv_message TYPE string.
9
10 FIELD-SYMBOLS: <lv_example> TYPE string.
11
12 CLASS lcl_example DEFINITION.
13   PUBLIC SECTION.
14     METHODS get_data
15       RETURNING VALUE(rv_data) TYPE string.
16
17 ENDCLASS.
18
19 *********************************************************
```

```
20 * IMPLEMENTATION SECTION
21 ***********************************************************
22
23 CLASS lcl_example IMPLEMENTATION.
24
25   METHOD get_data.
26     DATA lv_data TYPE string.
27
28     lv_data = 'Test'.
29     rv_data = lv_data.
30
31   ENDMETHOD.
32
33 ENDCLASS.                    "LCL_EXAMPLE
```

Code 2.1: Programm in ABAP

Der Code 2.1. zeigt ein Programmbeispiel in ABAP. Die ausführbaren Objekte in ABAP teilen sich in den Definitions- und in den Implementationsteil auf. Der Definitionsteil beginnt mit der Definition des Programmtypen und mit den programmtypischen Eigenschaften. Darauf folgt die Deklaration von Datenobjekten, Objekttypen und Feldsymbolen. Der Implementationsteil beinhaltet die Funktionalitäten eines Programms. Diese sind wiederum in Verarbeitungsblöcke aufgeteilt. Verarbeitungsblöcke stellen Methoden, Prozeduren, Unterprogramme und Dialogmodule dar (vgl. Jens Präkelt (GiSA GmbH), 2014, S.12).

2.2.1 Report

Der Report stellt ein prozedurales Programm in ABAP dar. Das heißt, Statements werden in einer definierten Reihenfolge abgearbeitet. Es ist die älteste Variante der ABAP-Programme, auch *ausführbare Programme* genannt (vgl. Theobald, 2007, S.19). Der Report in SAP verfügt über mehrere Arten von Modularisierung, um größere Stücken Quelltext zu organisieren (s. Theobald, 2007, S.26). Ein Include dient ausschließlich dazu, Quelltext in einen Container zu packen, der in einem anderen Programm eingefügt werden kann. Eine andere Art der Modularisierung stellt die Formroutine dar, auch Unterprogramm genannt. Ähnlich einer Methode innerhalb einer Klasse wird eine bestimmte Funktionalität gekapselt, die über Ein- und Ausgabeparameter verfügt.

2.2.2 Funktionsgruppe

Die Funktionsgruppe stellt eine Sammlung von Funktionsbausteinen dar. Sie kann selber Attribute und per Include auch Quelltext enthalten, welche für alle Funktionsbausteine verfügbar sind. Ein Funktionsbaustein ist ein prozedurales Programm, welches einer freien Methode in C++ ähnelt.

```
1 int function(int a, int b)
2 {
3   return (a+b);
4 }
```

Code 2.2: Freie Methode in C++

Ein Funktionsbaustein importiert, ändert und exportiert Parameter, die sie in einem prozeduralen Ablauf verarbeiten können. Der Unterschied ist, dass Funktionsbausteine auch ohne Angabe ihrer Funktionsgruppe aufgerufen werden können. Das heißt, der Name eines Funktionsbausteins muss innerhalb eines SAP-Systems eindeutig sein. Die Eindeutigkeit bei Methoden wird über ihre Beziehung zur Klasse gewährleistet.

2.2.3 Klasse

Wie alle ausführbaren ABAP-Objekte teilt sich die Klasse in zwei Bereiche: die Definition und die Implementation. In der Definition werden Methoden und Memberattribute mit ihren Eigenschaften deklariert. Die Statements `PUBLIC`, `PROTECTED` oder `PRIVATE` bestimmen den Zugriffsspezifizierer für die nachfolgenden Klassenelemente. Der Implementationsteil enthält die eigentliche Funktionalität der Methoden. Da ABAP ursprünglich prozedural entwickelt wurde, war die Einführung von Klassen aufgrund der Abwärtskompatibilität eine Herausforderung. Das Problem wurde so gelöst, dass eine Klasse intern in ein prozedurales Programm umgewandelt wird. Daher besitzt ein Programm immer einen Objekt- und einen Programmnamen.

2.2.4 Datentyp

In einem SAP-System existieren drei Arten von Datentypen: elementare, komplexe und Referenztypen. Die elementaren Datentypen, wie sie in der Programmiersprache ABAP übersetzt werden, bestehen nicht aus anderen Typen. Ein elementarer Datentyp ist immer durch einen der zehn ABAP-Datentypen spezifiziert (vgl. Krüger, 2009, S. 231):

- `c` für alphanumerische Werte,
- `n` für numerische Werte,
- `d` für Datumsangaben,
- `t` für Zeitangaben,
- `i` für ganzzahlige Werte,
- `f` für Gleitpunktzahlen,
- `p` für gerundete Werte,
- `x` für Byte-Felder,
- `string` für Text und
- `xstring` für Byte-Strings.

Komplexe Datentypen können aus Kombinationen anderer Datentypen bestehen. Sie erlauben eine semantisch unterschiedliche Verwendung gleicher Datentypen. Ein komplexer Datentyp muss aus existierenden Datentypen bestehen, beispielsweise einem spezifischen elementaren Datentyp. In der Tabelle 2.2 werden einige komplexe Datentypen des Data Dictionary (DDIC) und einfache Datentypen aus ABAP miteinander verglichen. Hier wird deutlich, dass alle komplexen Datentypen einen spezifischen Datentypen als Grundlage besitzen x(vgl. Krüger, 2009, S.232).

Data Dictionary	ABAP	Bemerkung
ACCP	N(6)	Buchungsperiode in der Form JJJJMM
CHAR n	C(n)	Beliebige Zeichenfolge definierter Länge
CLNT	C(3)	Mandant
CUKY	C(5)	Währungsschlüssel
CURR n,m	P((n+1)/2) DECIMAL m	Betrag (zum Währungsschlüssel)
DATS	D(8)	Datum (JJJJMMTT)
FLTP	F(8)	Fließkommazahl
INT1	X(1)	1 Byte
INT2	X(2)	2 Byte
INT4	X(4)	4 Byte
LANG	C(1)	Sprachenschlüssel
NUMC n	N(n)	Zeichenfolge, ausschließlich Ziffern
PREC	X(2)	Genauigkeit eines QUAN-Feldes
QUAN n,m	P((n+1)/2) DECIMAL m	Mengenfeld mit beliebiger Genauigkeit.
RAW n	X(n)	Folge beliebiger Bytes

Tab. 2.2: Vergleich ABAP- und DDIC-Datentypen (aus Theobald, 2007, S.308)

Zu den Referenztypen gehören auch Strukturen. Eine Struktur beschreibt eine Zeilenstruktur, deren Spalten durch eine Referenz auf einen Datentypen definiert werden. Strukturen können verschachtelt sein, das heißt, Strukturen können andere Strukturen enthalten.

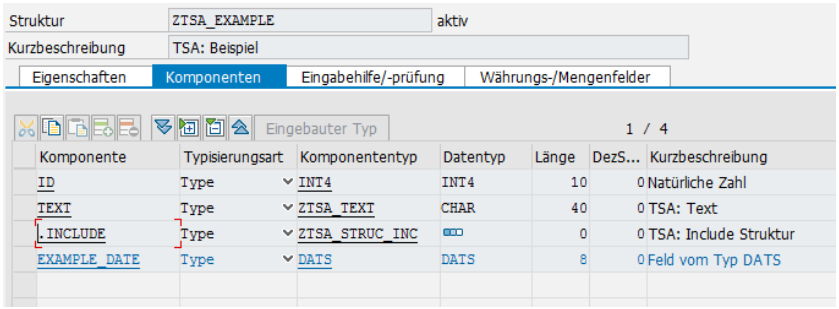

Abb. 2.4: Struktur mit Include (Screenshot 11.06.2017)

Der SAP-Anwender hat außerdem die Möglichkeit, eigene komplexe Datentypen anzulegen (vgl. Gupta, 2011, S.35). In einem SAP-System werden dafür Datenelemente genutzt. Datenelemente sind eigenständige Objekte, die bei globaler Definition innerhalb des gesamten SAP-Systems verwendet werden können.

2.2.5 Tabelle

In SAP gibt es drei Arten von Tabellen: transparente Tabellen, Pooled-Tabellen und Cluster-Tabellen. Eine transparente Tabelle hat eine 1:1-Beziehung zu einer Datenbanktabelle. Das heißt, die Struktur der transparenten Tabelle ist dieselbe wie die der Datenbank (vgl. Gupta, 2011, S.77). Demnach ähnelt eine transparente Tabelle einem Datenbankschema, welches physisch umgesetzt wird, sobald die Datenbank erzeugt wird.

Pooled-Tabellen besitzen 1:n-Beziehungen mit der Datenbanktabelle. Das heißt, es gibt mehrere Tabellen im Data Dictionary für eine Tabelle in der Datenbank. Im Unterschied zur transparenten Tabelle besitzt eine Pooled-Tabelle einen anderen Namen, eine unterschiedliche Anzahl an Feldern und verschiedene Feldnamen als die der Datenbanktabelle (vgl. Gupta, 2011, S.78).

Eine Cluster-Tabelle ähnelt einer Pooled-Tabelle. Sie besitzt eine n:1-Beziehung mit der Datenbanktabelle. Viele Cluster-Tabellen des Data Dictionary sind in einem Tabellen-Cluster einer Datenbank gespeichert. Genau wie die Pooled-Tabelle hat die Cluster-Tabelle einen anderen Namen, eine unterschiedliche Anzahl an Feldern und verschiedene Feldnamen im Vergleich zur Datenbanktabelle. Ein Tabellen-Cluster stellt ein ähnliches Konstrukt wie der Tabellen-Pool dar, bei dem verschiedene logisch zusammenhängende Daten von verschiedenen Cluster-Tabellen in einem physischen Abschnitt gespeichert werden (vgl. Gupta, 2011, S.80).

2.3 SAP Code Inspector

"Der Code Inspector [...] ist ein Werkzeug zur Prüfung von Repository-Objekten hinsichtlich verschiedenster statischer Codeaspekte." (s. Eilenberger et al., 2011, S.31). Der SAP Code Inspector überprüft ABAP-Objekte auf bestimmte, teilweise auch individuell einstellbare Richtlinien. Es ist ein entwicklungsbegleitendes Werkzeug, welches hilft, Fehler oder Fehlentwicklungen in einem frühzeitigen Stadium zu erkennen. Allerdings bietet der SAP Code Inspector nur statische Testverfahren von ABAP-Objekten. Dynamische Tests und Laufzeitanalysen können nur bedingt durchgeführt werden, beispielsweise mit dem SQL-Trace. Als Teil des Performanzprüfungen können mit der SQL-Trace-Analyse OpenSQL-Aufrufe in ABAP-Objekten überprüft werden.

Bezeichnung	**Beschreibung**
Allgemeine Prüfungen	Aufbau des Programms, Entwickler des Programms
Performanzprüfungen	Potenzielle verlangsamende Komponenten
Sicherheitslücken	Schwachstellen im Programm
Syntaxprüfungen	Korrekter Quelltext
Robuste Programmierung	Syntaktisch korrekte, aber problematische Programmierung
Programmierkonvention	Namensgebung
Metriken und Statistiken	Genereller Aufbau, Modularisierung
Dynamische Tests	Modultests

Tab. 2.3: Auswahl an SCI-Standardprüfungen (aus Eilenberger et al., 2011, S.66)

Der SAP Code Inspector besitzt Standardprüfungen, die mit dem SAP NetWeaver *Application Server*, siehe Abschnitt 2.1, ausgeliefert werden. Mit den *Allgemeinen Prüfungen* kann vor allem ein Überblick des Programms gewonnen werden. Dafür wird der Aufbau analysiert, der Entwickler eines Programmabschnitts ermittelt oder Datenbankaufrufe ausgelesen (vgl. Eilenberger et al., 2011, S.66). Die *Performanzprüfungen* sollen vor allem *"Bremsblöcke"* (vgl. ebd.) im Programm erkennen. Unter anderem werden SQL-Zugriffe auf Datenbanktabellen analysiert, deren Performance mit der Anzahl der Aufrufe durch das Erstellen von Indizes steigt. *Sicherheitsprüfungen* untersuchen den Quelltext auf mögliche Sicherheitslücken, wie beispielsweise SQL-Injections (vgl. ebd.). Syntaxprüfungen sichern die Korrektheit und die Lauffähigkeit des Quelltexts ab. Die Kategorie *Robuste Programmierung* weist auf problematische Programmierungen hin, die in der späteren Aufrufkette zu Fehlern führen kann. Mit der Prüfung auf *Programmierkonventionen* können vor allem Namenskonventionen überprüft werden, welche sich in ABAP vor allem nach Sichtbarkeit(global/lokal), Art (Variable/Konstante/Feldsymbol) und Typ (Import/Export/Return) richtet. *Metriken und Statistiken* geben Informationen zum generellen Aufbau des Programms. Dynamische Tests, auch bekannt als ABAP Unit, ermöglichen automatisiertes, auf Funktion ausgerichtetes, Testen.

Durch die Beschreibung der SCI-Standardprüfungen wird deutlich, dass sich der SAP Code Inspector zwischen der Entwicklung und der Qualitätssicherung eines Softwareprojekts einordnet, siehe Abbildung 2.3. Er ist einerseits ein Entwicklungswerkzeug, um Fehler von Anfang zu minimieren. Andererseits hilft er durch Prüfungen wie *Dynamische Tests* oder *Metriken und Statistiken* einen generellen Eindruck des Programms zu bekommen. Letzteres ist nicht nur für Entwickler, sondern auch für SAP-Berater oder Kunden interessant.

2.3.1 Anwendung des SAP Code Inspectors

Der SAP Code Inspector kann auf verschiedene Wege in einem SAP-System verwendet werden. Einerseits kann er über die Startseite SAP EASY ACCESS durch die Auswahl von SAP-Menü > WERKZEUGE > ABAP WORKBENCH > TEST > SCI – CODE INSPECTOR aufgerufen werden. Die Transaktion SCI ermöglicht die Verwaltung von Tests, Varianten, Objektmengen und Inspektionen beziehungsweise SCII für die direkte Ausführung ohne vorherige Speicherung. In der SAP-Entwicklungsumgebung ABAP Workbench mit der Transaktion SE80 findet man unter dem Eintrag REPOSITORY BROWSER den Punkt REPOSITORY-OBJEKT > PRÜFEN > CODE INSPECTOR oder per Rechtsklick auf das entsprechende Objekt unter PRÜFEN > CODE INSPECTOR.

Die in Tabelle 2.3 gezeigten Standardprüfungen können in einer *Prüfvariante* individuell zusammengestellt werden. Dadurch kann ein Entwickler ausschließlich die Performanzprüfungen, die Syntaxprüfungen und die Namenskonventionsprüfungen ausführen, um seinen eben geschriebenen Quelltext auf syntaktische Fehler und performanzkritische Stellen zu überprüfen. Eine *Objektmenge* fasst die SAP-Repository-Objekte zusammen, die in einer SCI-Prüfung analysiert werden sollen. Objektmengen können auf verschiedene Arten zusammengestellt werden. Hier werden die beiden wichtigsten Möglichkeiten vorgestellt. Die erste Möglichkeit ist, Objekte über ihre Objektzuordnung zur *Objektmenge* hinzuzufügen, zum Beispiel über die Zuordnung zu einem Paket oder einem Verantwortlichen. Die zweite Möglichkeit bietet eine freie Objektwahl, zum Beispiel eines einzelnen Funktionsbausteins. Letztendlich kombiniert eine *Inspektion Prüfvariante* und *Objektmenge* miteinander

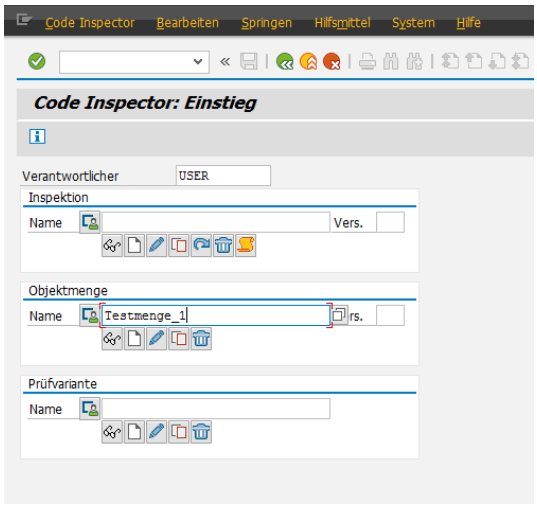

Abb. 2.5: SAP Code Inspector Start (Screenshot 03.07.2017)

und bildet einen Container für die Prüfergebnisse. So können leicht wiederverwendbare individuelle Prüfungen durchgeführt werden. Der SCI kann auch automatisiert zum Einsatz kommen. Über eine administrative Einstellung im *Transport Organizer Tool* kann bei Auftragsfreigabe eine vorher festgelegte Prüfung gestartet werden. Je nach Konfiguration werden die Ergebnisse als Inspektionsversion gespeichert oder dem Entwickler bei der nächsten Anmeldung am System angezeigt werden (vgl. Jens Präkelt (GiSA GmbH), 2014, S.18).

2.3.2 Aufbau eigener Prüfklassen

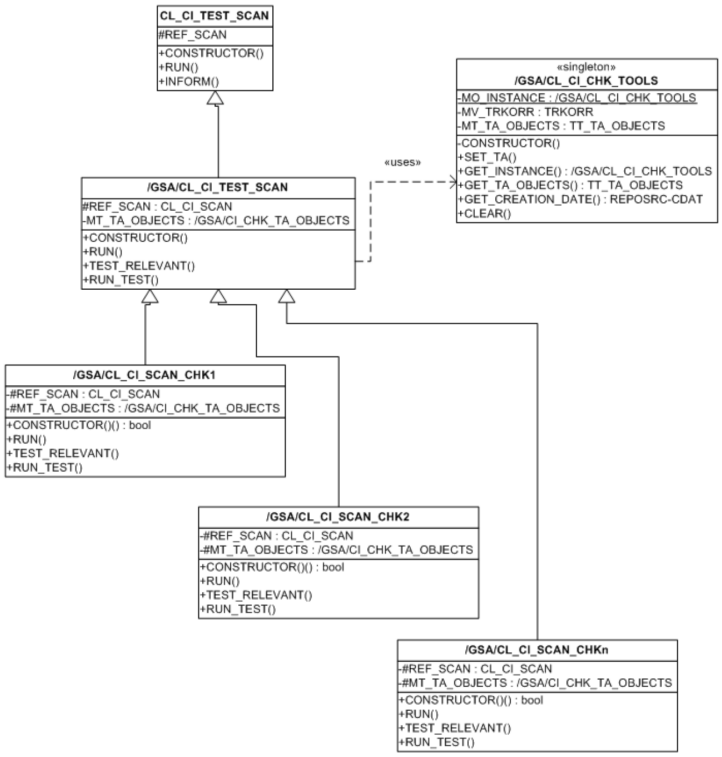

Abb. 2.6: Umgebungsarchitektur für eine eigene Prüfung (aus Jens Präkelt (GiSA GmbH), 2014, S.41, Abb.13)

Zusätzlich zu den Standardprüfungen, siehe Tabelle 2.3, können auch eigene Prüfungen in den SAP Code Inspector implementiert werden. Dadurch kann die Funktionalität des SCI individuell, und auf den Anwendungsfall zugeschnitten, erweitert werden. Eilenberger et al. (2011, S.159ff) und Jens Präkelt (GiSA GmbH) (2014, S.38ff) beschreiben ausführlich die Anbindung eigener Prüfungen an den SAP Code Inspector, was hier zusammengefasst dargestellt wird.

Wie in der Abbildung 2.6 zu sehen, besteht eine Prüfklasse wie andere Klassen aus Methoden und Attributen. Die für den Rahmen relevanten Methoden und Attribute sind bereits durch eine Vererbung der Root-Klasse `CL_CI_TEST_SCAN` über zwei Ebenen vorgegeben. In der Methode `constructor()` werden die Standardparameter für die Klasse bei ihrer Initialisierung festgelegt. Zu den Parametern gehören unter anderem `description`, `category`, `position`, `has_documentation` und `has_attributes`. `description` beschreibt, worauf die Prüfung die Repository-Objekte prüft. Die `category`-Variable legt fest, um welche Kategorie einer Prüfklasse es sich handelt. Zum Beispiel können in der Kategorie Namenskonventionen eigene Prüfklassen zur Einhaltung einer bestimmten Entwicklungsrichtlinie erstellt werden. Die Methode `inform()` meldet einen Fund einer Prüfung, um diesen später in der Ergebnistabelle anzuzeigen.

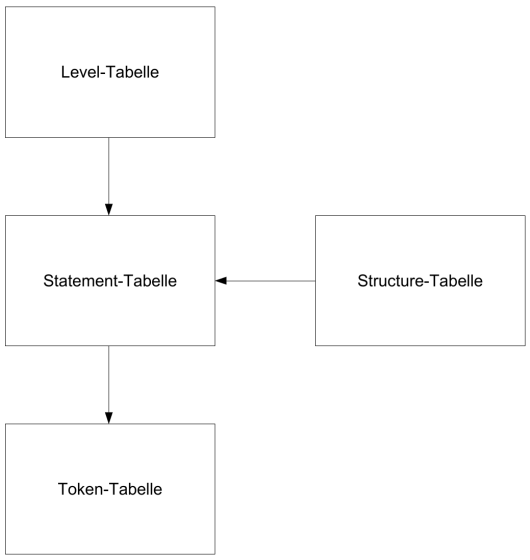

Abb. 2.7: SAP Code Inspector-Tabellen (erstellt nach Eilenberger et al., 2011, S.218, Abb.4-29)

Bereits vor dem Ausführen der ersten Prüfung werden die zu analysierenden Objekte durch die ABAP Scan Engine in vier Tabellen zerlegt: Token, Statement, Structure und Level.

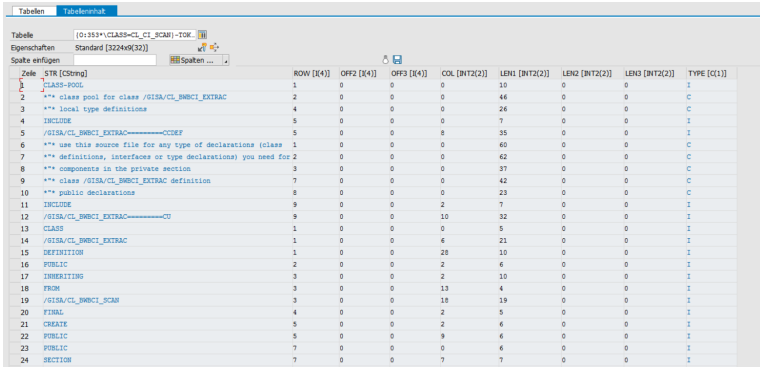

Abb. 2.8: Token-Tabelle des SAP Code Inspector (Debugger-Ansicht Screenshot 09.08.2017)

Jede Zeile der Token-Tabelle enthält einen Token, welches einen einzelnen elementaren Teil des Quelltexts repräsentiert. Die Statement-Tabelle zeigt den Zusammenhang zwischen einzelnen Tokens auf, beispielsweise ob mehrere Tokens zu einem Aufruf gehören. Die Structure-Tabelle stellt nähere Informationen zu den einzelnen Statements bereit, beispielsweise über deren Art und Aufbau. Die Level-Tabelle zeigt die hierarchische Struktur des zu analysierenden Quelltexts mit all seinen untergeordneten Quelltextteilen (vgl. Eilenberger et al., 2011, S.217).

Da der Prototyp dieser Arbeit auf die Architektur aus Jens Präkelt (GiSA GmbH) (2014) aufbaut, wird noch eine weitere, nicht zum Standard gehörende, Methode genannt. In Abbildung 2.6 wird eine eigene Prüfung `/GSA/CL_CI_SCAN_CHK1` als erbende Klasse von `/GSA/CL_CI_TEST_SCAN` erstellt. Die Methode `RUN_TEST()` startet die Ausführung der eigentlichen Prüfungen nach ihrer Initialisierung. `RUN_TEST()` übergibt auch die von der ABAP Scan Engine erzeugten Tabellen an die verarbeitenden Methoden des Prototypen.

3 Softwarevisualisierung

3.1 Grundlagen der Softwarevisualisierung

Der Brockhaus (2006) definiert Visualisierung als *"Bezeichnung für bildliche Formulierung und Kommunikation, das heißt für die Aufbereitung von Informationen mit bildlichen Mitteln"*. Dem Betrachter werden Informationen nicht nur als Text, sondern auch visuell veranschaulicht dargestellt. Stephan Diehl (s. 2007, S.3) stellt zwei Disziplinen in der Visualisierung heraus: die wissenschaftliche Visualisierung und die Informationsvisualisierung. Die wissenschaftliche Visualisierung verarbeitet physische Daten, während die Informationsvisualisierung mit abstrakten Daten arbeitet. Die Softwarevisualisierung ordnet sich in den zweiten Bereich ein. Stephan Diehl (s. 2007, S.3) definiert Softwarevisualisierung "*als die Visualisierung von Artefakten im Zusammenhang mit Software und dem Entwicklungsprozess*". Durch die weite Definition umfasst die Softwarevisualisierung einen großen Bereich, der Struktur, Verhalten und Evolution einer Software umfassen kann. In dieser Arbeit liegt der Fokus auf den Strukturinformationen.

3.2 Visualisierungsprozess

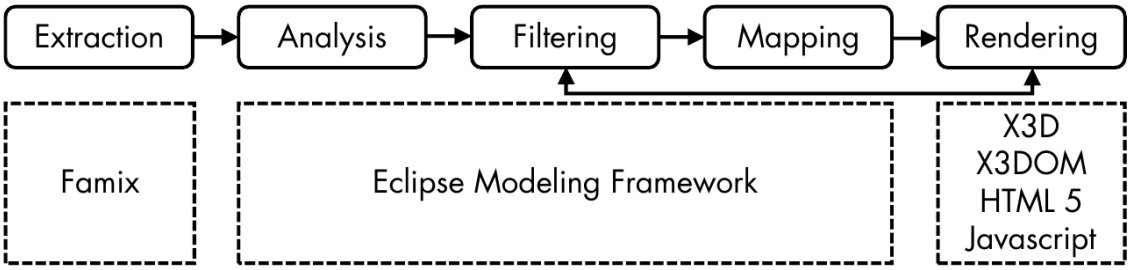

Abb. 3.1: *"Visualization pipeline and implementation techniques"* (s. Müller & Zeckzer, 2015, S.173, Fig.2)

In diesem Abschnitt wird der Visualisierungsprozess beschrieben, wie ihn Müller et al. (2011) vorgestellt hat. Die Abbildung 3.1 zeigt die einzelnen Schritte des Visualisierungsprozesses. Im ersten Schritt werden statische Informationen über ein Softwaresystem im FAMIX-Format extrahiert. Die extrahierten Informationen werden in FAMIX gespeichert. Während der Analyse werden die einzelnen Elemente auf ihre syntaktische Korrektheit und die semantische Validität geprüft. Die Elemente müssen ihrem Meta-Model, der Grammatik-Definition siehe Abschnitt 3.3, entsprechen. Die Filterung ermöglicht es, während der Generierung Elemente auszuschließen, die nicht in der Visualisierung abgebildet werden soll. Das Mapping wird durch Transformations-und Modifikationsfunktionen des Eclipse Modeling Framework realisiert (vgl. Steinberg et al., 2009) . Zuerst werden die validen gefilterten Entitäten der Input-Datei auf in ein plattformunabhängiges Model transformiert. Hier werden die jeweiligen Formen, ihre Größe und Anordnung berechnet. Im zweiten Schritt wird ein Plattform-spezifisches Format, vorrangig Extensible 3D (X3D), umgewandelt (s. Igd, 2014). Am Schluss wird das X3D-Model für die Webanzeige optimiert und in Extensible 3-DOM (X3DOM) konvertiert. Die Visualisierung selber wird von einem 3D-fähigen Browser mit einer WebGL-Schnittstelle

gerendert und durch ein in Javascript geschriebenes Graphic User Interface gesteuert (vgl. Müller et al., 2011, S.173).

Diese Arbeit untersucht die Extraktion statischer SAP-Strukturinformationen in ein FAMIX-Format. Dafür wird die FAMIX-Grammatik nach Müller (2015) um SAP-spezifische Informationen erweitert. Daher sind die weiteren Schritte und Technologien des oben genannten Visualisierungsprozesses für diese Arbeit nicht relevant.

3.3 FAMIX-Grammatik

FAMIX besteht aus sprachunabhängigen Meta-Modellen, die auf eine vereinheitlichte Weise mehrere objektorientierte und prozedurale Sprachen darstellen können (vgl. Ducasse et al., 2011, S.12). Die FAMIX-Familie wurde für die Analyse von Historien und Aspekten, das Finden von Duplikaten, koevolutionärer Elemente und Subversionen erstellt. FAMIX ermöglicht es, Informationen in einer selbst definierten Syntax zu speichern. In dieser Arbeit wird mithilfe des Frameworks Xtext eine erweiterte FAMIX-Grammatik beschrieben, welche es ermöglicht, SAP-Strukturinformationen zu extrahieren und einzulesen. Das Open-Source Framework Xtext ist für die Entwicklung von unter anderem Programmiersprachen entwickelt worden (vgl. Efftinge, 2014, S.113). Eine vollumfängliche Beschreibung des Frameworks findet sich ebenda.

```
1  grammar org.svis.xtext.Famix with org.eclipse.xtext.common.Terminals
2  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
3  generate famix "http://www.svis.org/famix"
4
5  Root:
6      document=Document?;
7  Document:
8      {Document}
9      '(' elements+=FAMIXElement* ')';
10 FAMIXElement:
11     FAMIXStructure | FAMIXFunctionModule;
12 FAMIXStructure:
13     FAMIXFunctionGroup;
14
15 FAMIXFunctionGroup:
16     '(FAMIX.FunctionGroup'
17     '(' 'id: ' name=INT_ID ')'
18     '(' 'name' value=MSESTRING ')'
19     '(' 'container' container=IntegerReference ')'')';
20 FAMIXFunctionModule:
21     '(FAMIX.FunctionModule'
22     '(' 'id: ' name=INT_ID ')'
23     '(' 'name' value=MSESTRING ')'
24     '(' 'parentType' parentType=IntegerReference ')'
25     ('(' 'numberOfStatements' numberOfStatements=INT ')')?')';
26
27 IntegerReference:
28     '(' 'ref: ' ref=[FAMIXElement|INT_ID] ')';
29 INT_ID returns ecore::EString:
```

```
30      '^'? INT;
31  terminal MSESTRING:
32      ("'"->"'")*;
```

Code 3.1: Beispiel einer FAMIX-Grammatik

Der Code 3.1 zeigt einen Ausschnitt einer Grammatik für FAMIX in Xtext einer Funktionsgruppe mit einem Funktionsbaustein. Zuerst wird die Grammatik festgelegt, in der alle nachfolgenden Elemente geschrieben werden. Danach werden die Abhängigkeiten der Elemente beschrieben. In diesem Fall sind `FAMIXStructure` und `FAMIXFunctionModule` Unterelemente von `FAMIXElement`. Unter `FAMIXElement` müssen alle Elemente, die später definiert werden, aufgelistet sein. Nach der Auflistung folgt die eigentliche Syntax des Elements. Die Definition des Elements `FAMIXFunctionModule` wird mit seinem Namen innerhalb des Xtext-Dokuments eingeleitet. Alle nachfolgenden, in Hochkommata gehaltenen, Zeilen beschreiben die spätere Syntax in dem FAMIX-Dokument. Das heißt, jedes in Hochkommata gesetzte Literal kommt im eigentlichen Dokument vor. Das Element `FAMIXFunctionModule` besitzt mehrere Attribute: `id`, `name` und `parentType` und `numberOfStatements`. Die `id` muss eindeutig sein, um jedes Element zuordnen zu können. Das Attribut `parentType` enthält den Wert der Eltern-`id`, in ABAP die `id` einer Funktionsgruppe.

3.4 Recursive Disc Metaphor

Die *Recursive Disk Metaphor* ist eine von Müller & Zeckzer (2015) entwickelte Softwarevisualisierungsmetapher. Sie besteht aus mehreren Glyphen, die wiederum Glyphen enthalten können. Durch die rekursive Generierung der Elemente leitet sich der Name der *Recursive Disk Metaphor* ab. In der *Recursive Disk Metaphor* wird jede Entität, das heißt, Attribut, Methode, Klasse oder Paket, als Glyphe visualisiert (s. Müller & Zeckzer, 2015, S.172).

Abb. 3.2: *Recursive Disk Metaphor* (aus Müller & Zeckzer, 2015, S.173, Fig.1)

Der Kreis einer Klasse ist in einen oder mehrere innere Ringe unterteilt. Von innen nach außen werden innere Klassen, Attribute und Methoden diesen Elementen zugewiesen. Die Anordnung dieser

Elemente richtet sich nach der Größe der Elemente, wie der Anzahl der Statements einer Methode. Der äußerste Ring bildet eine Grenze, um in der Interaktion und in der Wahrnehmung eine klare Unterscheidung zu anderen Klassen zu treffen. Der äußerste graue Ring stellt das Paket dar. Da die RDM bisher für die Programmiersprache Java ausgelegt ist, beinhalten Pakete keine Methoden oder Attribute. Da in Java Namensräume durch das Paket definiert werden, sind diese durch die Paketkreise visualisiert. Die Farbgebung der Glypen richtet sich nach der *opponent process theory* nach Ware (2004): um Anwender mit einer Rot-Grün-Schwäche zu inkludieren, kommen blau, gelb und grau zum Einsatz, die gut zu unterscheiden sind.

4 Analyse der zu extrahierenden Strukturinformationen

Aufbauend auf den Grundlagen, werden in diesem Kapitel die für die Implementierung relevanten Vorüberlegungen getroffen. Dafür müssen die zu ermittelnden SAP-Strukturinformationen festgelegt werden. Aus diesem Grund wird ein Ansatz zur Visualisierung von SAP-Strukturinformationen in der *Recursive Disk Metaphor* untersucht, um einerseits die benötigten Informationen zu spezifizieren und andererseits die Ausmaße der Informationsermittlung abschätzen zu können.

4.1 Auswahl der ABAP-Objekte

In Müller (2015) wurde bereits eine Vorauswahl getroffen, in der eine FAMIX-Grammatik mit bestimmten Objekten festgelegt wurde. Dieser Auswahl wurden sämtliche SAP-Objekte, siehe Appendix, gegenübergestellt. Im ersten Schritt wurden die SAP-Objekte identifiziert, die bereits in der FAMIX-Grammatik von Müller (2015) vorhanden waren, beziehungsweise die auf ein ähnliches Objekt übertragen werden konnten. Zusätzlich zu diesen Objekten wurden im zweiten Schritt diejenigen SAP-Objekte ausgewählt, die häufig im ABAP-Quelltext verwendet werden und für das Strukturverständnis wichtig sind. Dazu zählen Reports, Funktionsbausteine, Datenbanken, Datenelemente und Strukturen.

Objekttyp	**Objekt**	**Typkürzel**
ABAP Dictionary	Datenbanktabellen	TABL
	Datenelemente	DTEL
	Strukturen	STRU
Programmbibliothek	Reports	REPS
	Funktionsgruppen	FUGR
	Funktionsbausteine	FUMO
Klassenbibliothek	Klassen	CLAS
	Methoden	METH
	Memberattribute	ATTR
Erweiterungen	Namespaces	NSPA
	Nachrichtenklassen	MSAG

Tab. 4.1: Selektierte Objekte aus SAP-System

Die Tabelle 4.1 enthält alle Objekte, die durch den Prototyp extrahiert werden sollen. Die Einteilung der Objekttypen in ABAP Dictionary, Programmbibliothek, Klassenbibliothek und Erweiterung richtet sich nach der Kategorisierung innerhalb des Repository Infosystems eines SAP-Systems.

4.2 Anforderungsanalyse für ABAP

Die Softwarevisualisierung muss als Werkzeug der Informationsvermittlung die Daten auf eine verständliche, übersichtliche und gleichzeitig umfassende Weise darstellen. Rohr (2007, S.16) definiert seine Anforderungen an die Softwarevisualisierung folgendermaßen:

- Wichtige Eigenschaften des Systems sollen durch eine übersichtliche Darstellung schnell und leicht zugänglich und zu verarbeiten sein.
- Dem Anwender soll es möglich sein, sich auf konkrete, für ihn interessante Teilaspekte in der Darstellung zu konzentrieren.
- Es soll vermieden werden, dass der Anwender durch eine Vielzahl gleichzeitig präsentierter Informationen überlastet wird und die dargestellten Informationen nicht mehr effektiv genug verarbeiten kann.
- Informationen sollen gemäß dem Interesse des Anwenders visualisiert werden, sodass dieser nur für seine aktuelle Aufgabe interessante Information präsentiert bekommt.

Die Erweiterung der *Recursive Disk Metaphor* muss neben der Beibehaltung des Konzepts aus Müller & Zeckzer (2015) die wichtigsten SAP-Strukturinformationen übersichtlich und detailreich abbilden. Die Anwendergruppe sind neben ABAP-Entwicklern auch die anderen Akteure, die an dem gesamten Softwareentwicklungsprozess teilhaben. Allen Akteuren soll es möglich sein, die für ihn interessanten Informationen in der Visualisierung dargestellt zu bekommen.

ABAP-Objekte	**RDM-Elemente**	**Darstellung**
Klasse	Klasse	Ring
Methode	Methode	Ringsegment
Memberattribut	Attribut	Äußerer Klassenring
Paket	Paket	Grauer Ring
Lokale Klasse	Innere Klasse	Klassenring innerhalb eines anderen Klassenrings
Datenbanktabelle	nicht zugewiesen	
Datenelement	nicht zugewiesen	
Struktur	nicht zugewiesen	
Report	nicht zugewiesen	
Formroutine	nicht zugewiesen	
Funktionsgruppe	nicht zugewiesen	
Funktionsbaustein	nicht zugewiesen	
Namespace	nicht eindeutig zugewiesen	

Tab. 4.2: Vergleich der Elemente der *Recursive Disk Metaphor* und SAP

Die FAMIX-Grammatik nach Müller (2015) beinhaltet Klassen, Methoden, Memberattribute und Pakete. Namensräume werden durch Pakete visualisiert. Daher können diese Entitäten aus ABAP einfach übernommen werden. Datenbanktabellen, Datenelemente, Strukturen, Reports, Formroutinen, Funktionsgruppen und Funktionsbausteine besitzen keine Darstellungsform in der *Recursive Disk*

Metaphor. Nachrichtenklassen werden vernachlässigt, da sie einerseits keine wichtige Eigenschaft eines Systems darstellen und andererseits die Gefahr besteht, dass der Anwender mit einer Vielzahl gleichzeitig präsentierter Informationen überfordert ist. Datenbanktabellen, Datentypen und Strukturen gehören dem Data Dictionary an. Sie besitzen keinen Quelltext, weshalb die Darstellungsgröße nicht durch ihre Lines Of Code (LOC) oder die Anzahl ihrer Statements visualisiert werden kann. Datenbanktabellen speichern Informationen in einem relationalen Datenformat. Die Größe wird üblicherweise durch die durchschnittliche Anzahl der Zeilen und die Anzahl der Spalten bestimmt. Ein Datenelement besitzt außer dem enthaltenen Datentypen keine weiteren Informationen. Daher können verschiedenen Datenelemente auf dieselbe Weise dargestellt werden. Strukturen beinhalten Datentypen, die den Spaltentyp festlegen, siehe Unterabschnitt 2.2.4. Die Anzahl der enthaltenen Spalten bestimmt die Größe einer Struktur. Reports, Formroutinen, Funktionsgruppen und Funktionsbausteine gehören zu den prozeduralen Programmen. Bis auf Funktionsgruppen enthalten diese Elemente Quelltext, wodurch die LOC und die Anzahl der Statements ermittelt werden kann.

4.3 Visualisierung in der Recursive Disk Metaphor

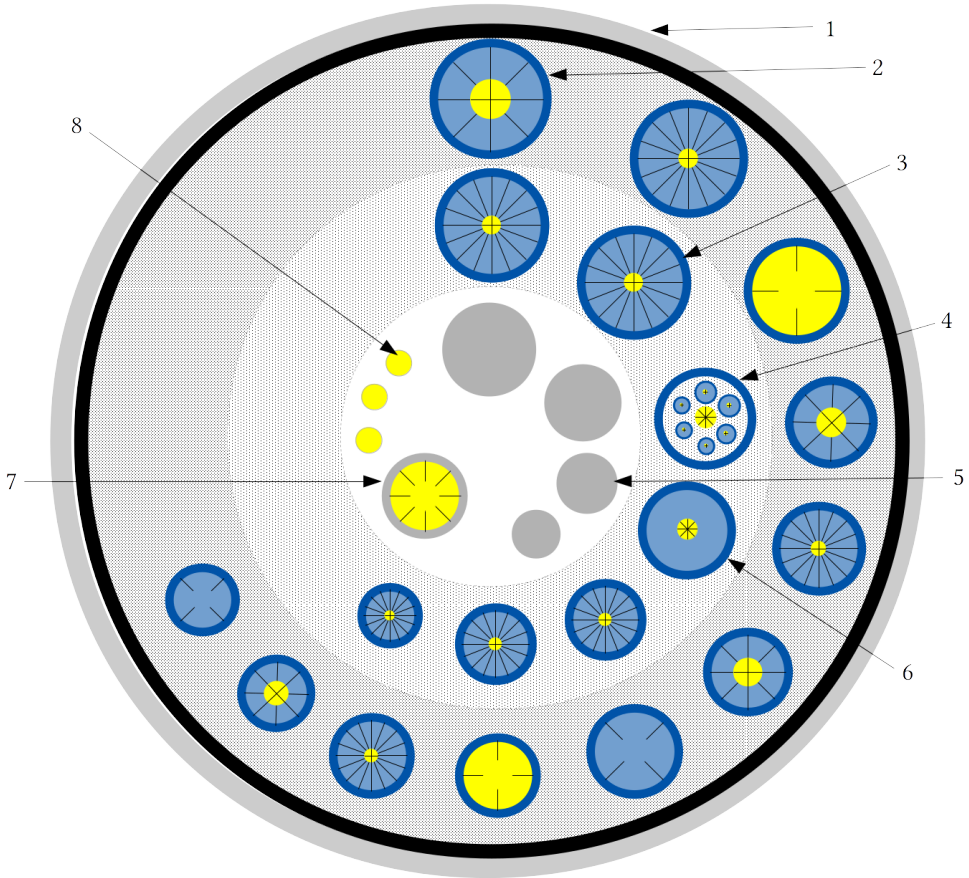

Abb. 4.1: *Recursive Disk Metaphor* mit SAP-Elementen (erweitert nach Müller & Zeckzer, 2015, S.173, Abb. 1)

Die Anforderungsanalyse im Abschnitt 4.2 ergab, dass bereits einige ABAP-Objekte mit in Elementen der *Recursive Disk Metaphor* visualisiert werden können, siehe Tabelle 4.2. In Abbildung 4.1 ist ein ein Visualisierungsansatz der SAP-Strukturinformationen innerhalb eines Pakets abgebildet. Die von

Ware (2004) empfohlene Farbgebung wurde beibehalten. Auch die Anordnung der Elemente nach ihrer jeweiligen Größe wurde aus Müller & Zeckzer (2015) übernommen.

Allgemein wird zwischen drei Bereichen für verschiedene Elemente unterschieden: objektorientierte Elemente, prozedurale Elemente und DDIC-Elemente. Der äußerste Abschnitt des Paketkreises (1) beinhaltet Klassen und die dazugehörigen Elementen wie Methoden und Memberattribute. Reports, Formroutinen, Funktionsgruppen und Funktionsbausteine finden sich im mittleren Kreisabschnitt. Im Kreiskern sind alle DDIC-Elemente wie Datenbanktabellen, Datenelemente und Strukturen dargestellt.

ABAP	**Darstellung**	**Zusätzliche Information**
Klasse	Ring, blau (2)	Anzahl der Statements
Methode	Ringsegment, blau	Anzahl der Statements
Memberattribut	Äußerer Klassenring	
Paket	Ring, grau (1)	
Lokale Klassen	Klassenringe innerhalb von anderen Klassenringen	
Datenbanktabellen	Ring, grau ausgefüllt (5)	Spaltenanzahl, durchschnittliche Zeilenanzahl
Datenelemente	Ring, gelb ausgefüllt (8)	
Strukturen	Ring, gelb (7)	Anzahl der Spaltenelemente
Reports	Ring, blau (4)	Anzahl der Statements
Formroutinen	Wie Reports	Anzahl der Statements
Funktionsgruppen	Ring, blau (3)	
Funktionsbausteine	Ringsegment, blau	Anzahl der Statements
Namespaces	Wie Paket	

Tab. 4.3: Visualisierung von SAP-Elementen in die *Recursive Disk Metaphor*

Die Visualisierung der Klassen (2), Methoden und Attribute wurde aus Müller & Zeckzer (2015) übernommen. Die Darstellung von Funktionsgruppen (3) ähnelt der der Klasse. Die einzelnen Funktionsbausteine sind als Kreissegmente des Funktionsgruppenkreises visualisiert. Da Funktionsgruppen selber Attribute enthalten können, auf die die zugehörigen Funktionsbausteine Zugriff haben, sind diese, ähnlich zu Memberattributen einer Klasse, als gelbe Kreissegmente in der Kreismitte dargestellt. Die Größe einer Funktionsgruppe ist durch die Anzahl der Statements der enthaltenen Funktionsbausteine definiert.

Reports sind mit einem blauen Kreis dargestellt, der wiederum Formroutinen als eigene Programmkreise enthält (4). Die global gültigen Attribute eines Reports sind in der Mitte des Kreises dargestellt, worauf neben dem Report auch alle Formroutinen zugreifen können. Die Kreisgröße der Reports und Formroutinen wird die Statementanzahl des Reports bestimmt. Da die Formroutine direkt im Report definiert und implementiert wird, kann die Größe einer Formroutine niemals die Größe des Elternreports überschreiten. Ein Report ohne Formroutinen wird als einfacher Kreis mit gelben Kreissegmenten für die globalen Attribute dargestellt (6).

Der Kreiskern beinhaltet alle DDIC-Elemente. Datenbanktabellen sind als graue Kreise dargestellt, deren Größe aus dem Produkt der durchschnittlichen Zeilenanzahl und der Spaltenanzahl gebildet wird (5). Eine Struktur ist als blauer Kreis mit grauen Kreissegmenten visualisiert (7). Die graue Visualisierung soll die Unterscheidung zu Datenklassen verdeutlichen. Eine Struktur enthält Spaltentypen und die dazugehörigen Datenelemente und -typen. Datenelemente, die einerseits im Paket angelegt sind und andererseits eine Referenz zu der Struktur besitzen, werden doppelt, einmal im DDIC-Bereich und einmal in der Struktur, abgebildet. Ein Datenelement wird, ähnlich wie ein Memberattribut einer Klasse, durch einen gelben Kreis visualisiert (8).

5 Prototyp zur Extraktion von Strukturinformationen

5.1 Konzeption

Die Konzeption ergibt sich aus den Anforderungen an den Prototypen. Die Anforderungen teilen sich in einen fachlichen und in einen technischen Bereich. Die fachlichen Anforderungen an den Prototypen, die im Abschnitt 1.3 beschrieben sind, stellen sich aus fünf Punkten zusammen:

- Auswahl der Entitäten,
- Visualisierung der Elemente,
- Informationen über die Entitäten,
- Erweiterung der FAMIX-Grammatik und
- Erzeugung des FAMIX-Dokuments.

Die Auswahl der Entitäten muss einerseits in der Visualisierung genug Informationen bereitstellen, sodass sich der jeweilige Anwender ein gutes Bild über das Programm machen kann. Andererseits steigt der Entwicklungsaufwand mit der Anzahl der Objekte. So muss der Aufwand im verhältnismäßigen Rahmen einer Bachelorarbeit bleiben. Die Anforderungen an die Visualisierung der Entitäten werden genauer in Abschnitt 4.2 für die *Recursive Disk Metaphor* und für die City Metaphor in Roth (2017) beschrieben. Die benötigten Informationen über Entitäten leiten sich aus den Vorgaben der Visualisierungen ab. Die *Recursive Disk Metaphor* verwendet bei Quelltextelementen die Anzahl der Statements, bei DDIC-Objekten wie Datenbanktabellen den Quotienten aus Spaltenanzahl und durchschnittlicher Zeilenanzahl als Referenzwert für die jeweilige Formgröße. Da die extrahierten SAP-Strukturinformationen nicht nur für die Visualisierung bereitgestellt werden, sondern auch zur Dokumentation von Programmen dienen sollen, werden noch weitere Informationen, wie zum Beispiel Beschreibung, Erstelldatum, Author und Änderungsdatum, extrahiert. Das für Softwarevisualisierung extrahierte FAMIX-Dokument baut auf der in Müller (2015) definierten Grammatik auf. Weiterhin sollen alle für die Softwarevisualisierung benötigten Informationen enthalten sein. Neben FAMIX sollen weitere Extraktionsformate zu Dokumentationszwecken anbindbar sein. Der in Müller et al. (2011) beschriebene Generator inklusive der Erweiterung nach Roth (2017) soll das FAMIX-Dokument ohne, beziehungsweise mit tolerierbaren Fehlern eingelesen werden. Das FAMIX-Dokument wird durch den Prototypen in einer korrekten, nach der oben beschriebenen Grammatik ausgegeben.

Die technischen Anforderungen leiten sich aus vier verschiedenen Punkten ab:

- Entwicklung auf einem SAP-System mit der Programmiersprache ABAP,
- Verwendung des SAP Code Inspector als Analysewerkzeug,
- Implementierung des Prototypen als eigene Prüfung und
- Ausgabe verschiedener Datenformate.

Da ein SAP-System in ABAP programmiert ist, bietet sich die Verwendung der SAP-eigenen Programmiersprache an. Zudem soll der Prototyp in ein bestehendes ABAP-Paket implementiert werden, welches bereits andere eigene Prüfungen enthält. In diesem Paket erbt der Prototyp von der Klasse /GSA/CL_CI_TEST_SCAN, welche wiederum ein Kind der SCI-Klasse CL_CI_TEST_SCAN ist (s. Jens Präkelt (GiSA GmbH), 2014, S.41). Auf der einen Seite kann durch die Vererbung eine bestehende Architektur für die Übergabe der SCI-Objekte genutzt werden. Auf der anderen Seite bietet die Vererbung Funktionen zur weiteren Analyse von Entitäten.

5.2 Funktionsbeschreibung

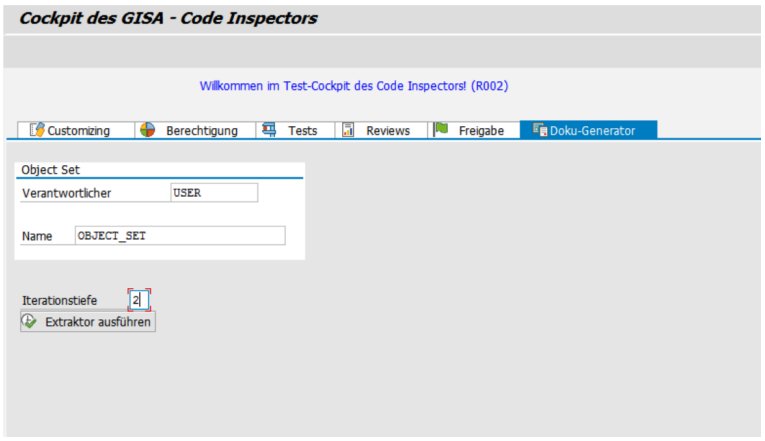

Abb. 5.1: Start des Prototypen mit Auswahl Objektmenge und Iterationstiefe (Screenshot 09.08.2017)

In Abbildung 5.1 ist das Start-DynPro, kurz für Dynamisches Programm (s. Mormann, 2014, S.9), des Prototypen gezeigt. Die Objektmenge OBJECT_SET enthält die Prototyp-Klasse CL_BWBCI_EXTRAC und die Iterationstiefe ist auf zwei gesetzt. Nach Start des Prototypen wird aus der Objektmenge und der Prüfvariante CL_BWBCI_EXTRAC eine SCI-Inspektion erstellt. Die Inspektion iteriert über die selektierte Menge und übergibt der Klasse CL_BWBCI_EXTRAC eine Tabelle mit Objektinformationen.

Abbildung 5.2 zeigt den Eintrag für die Klasse CL_BWBCI_EXTRAC in der Objekttabelle. Die Informationen werden aus der übergebenen Tabelle ausgelesen und um Metainformationen wie SUPERCLASS oder die Klassenart TYPE angereichert. Die Informationen über Methoden und Memberattribute der analysierten Klasse sind in Tabellen abgelegt, die als Referenz auf METHODS und MEMBERATTRIBUTES in der Objekttabelle eingetragen sind. Die letzte Zeile ITERATION zeigt an, in welcher Iteration die Klasse der Objekttabelle hinzugefügt wurde, in diesem Fall in der ersten

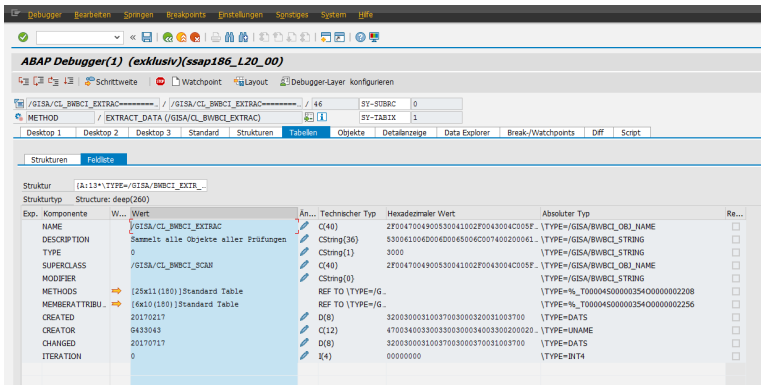

Abb. 5.2: CL_BWBCI_EXTRAC in Objekttabelle (Debugger-Ansicht, Screenshot 17.07.2017)

Iteration. Nach der Analyse der Daten werden die Informationen aus der Objekttabelle in FAMIX geschrieben und lokal abgespeichert.

```
1   (FAMIX.Devclass
2       (id: 2)
3       (name '/GISA/BWBCI')
4   )
5   (FAMIX.Class
6       (id: 57)
7       (name '/GISA/CL_BWBCI_EXTRAC')
8       (container (ref: 2))
9       (isInterface false)
10  )
11  (FAMIX.Method
12      (id: 59)
13      (name 'ADD_ATTR')
14      (numberOfStatements 46)
15      (parentType (ref: 57))
16      (signature 'ADD_ATTR()')
17  )
```

Code 5.1: Ausschnitt aus generiertem FAMIX-Format

Der Code 5.1 zeigt die extrahierten SAP-Strukturinformationen der Klasse CL_BWBCI_EXTRAC. Die Klasse ist im Paket `/GISA/BWBCI` abgelegt, was über die `container`-Referenz angezeigt wird. Das Attribut `isInterface` wird durch die Spalte TYPE der Objekttabelle bestimmt. ADD_ATTR ist eine Methode der analysierten Klasse, was durch das `parentType`-Attribut deutlich wird.

In Abbildung 5.3 wird die Visualisierung der analysierten Klasse in der City Metaphor nach Roth (2017) gezeigt. Der graue Untergrund repräsentiert ein Paket. Neben dem DDIC-Bereich ist die Klasse als Hochhaus dargestellt. In einem Kreis um das Paket der ursprünglich selektierten Objektmenge sind andere Pakete angeordnet, aus denen die Klasse CL_BWBCI_EXTRAC DDIC-Elemente verwendet. Gut sichtbar ist auch der Aufruf einer Datenbanktabelle eines anderen Pakets durch eine Methode der analysierten Klasse als blaue Verbindung.

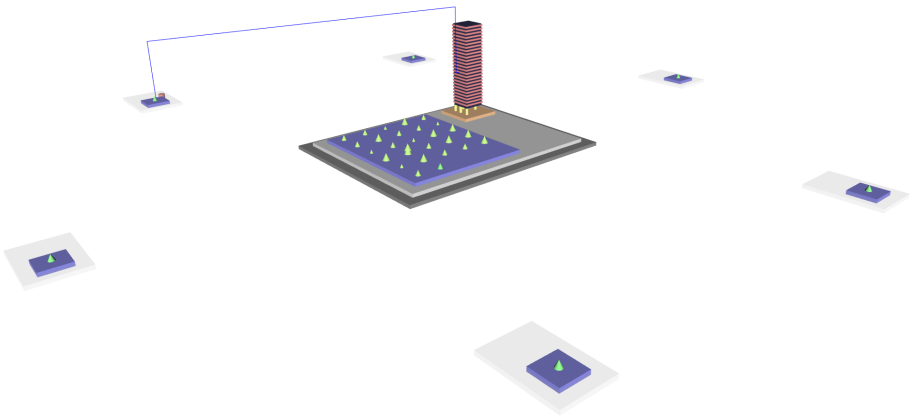

Abb. 5.3: City Metaphor nach Roth (2017) (Screenshot 23.07.2017)

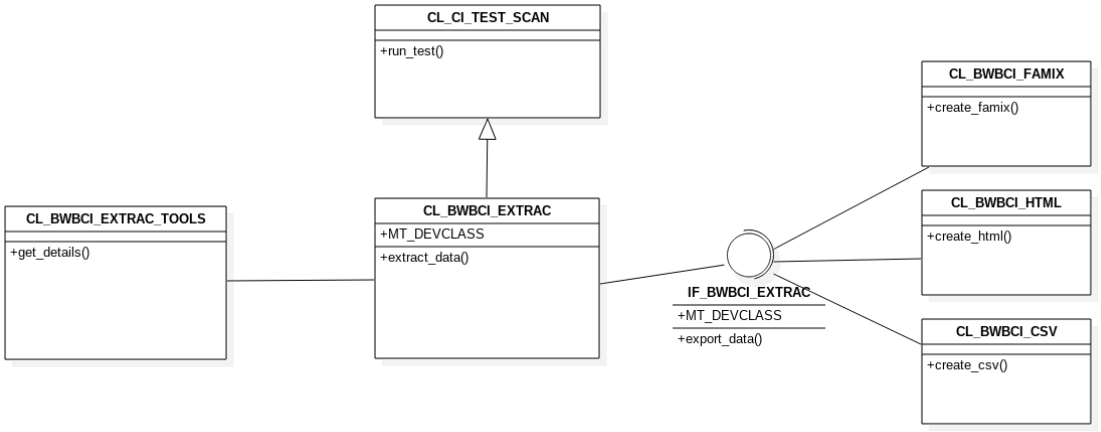

Abb. 5.4: Prototyparchitektur

5.3 Architektur

Objekt	Beschreibung
IF_BWBCI_EXTRAC	Interface zur Bereitstellung der Objekttabelle
CL_BWBCI_EXTRAC	Logische Steuerung und Hinzufügen der einzelnen Elemente an die Objekttabelle
CL_BWBCI_EXTRAC_TOOLS	Ergänzende Informationen ermitteln
CL_BWBCI_FAMIX	Erstellung der FAMIX-Beschreibung
CL_BWBCI_FAMIX_HELPER	Unterstützende Funktionen wie ID-Generierung und Verwaltung

Tab. 5.1: Übersicht über die Klassen des Prototyps

Die in Abbildung 5.4 dargestellte Architektur des Prototypen gliedert sich in drei Bereiche: logische Steuerung, Anreicherung und Extraktion.

Die logische Steuerung übernimmt die zentrale Klasse CL_BWBCI_EXTRAC. Sie ist als erbende Klasse von /GISA/CL_CI_TEST_SCAN eine eigene SCI-Prüfung. Über die vererbte Methode

RUN_TEST() werden die Informationen, die der SAP Code Inspector über die selektierte Objektmenge gesammelt hat, an die Methode EXTRACT_DATA() der Klasse CL_BWBCI_EXTRAC übergeben. EXTRACT_DATA() übernimmt die zentrale Steuerung der weiteren Verarbeitung. Hier werden die vom SAP Code Inspector übergebenen Informationen aufbereitet, dem richtigen Objekttyp zugeordnet und in die Objekttabelle MT_DEVCLASS eingetragen. Dafür werden im ersten Schritt alle relevanten DDIC-Objekte ermittelt. Da der SAP Code Inspector keine DDIC-Objekte analysiert, müssen diese Informationen aus verschiedenen Datenbanktabellen ausgelesen werden. Im zweiten Schritt werden die vom SCI übergebenen Informationen über Quelltextobjekte verarbeitet und angereichert.

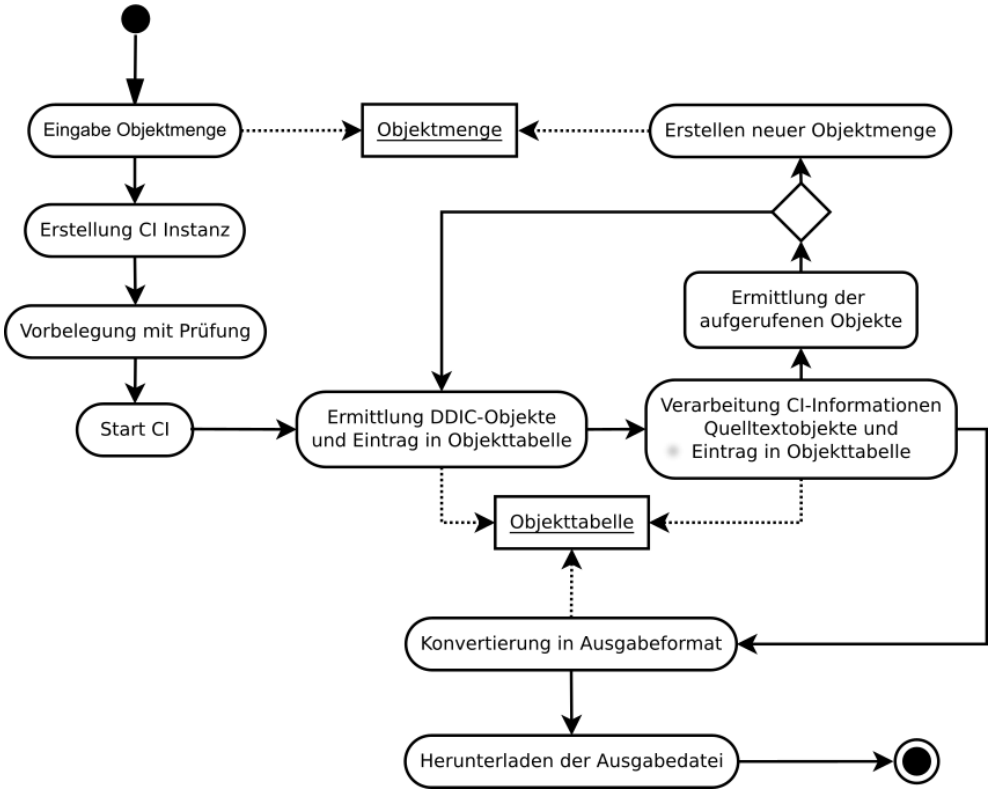

Abb. 5.5: Prozessverhalten des Prototypen

Die Anreicherung übernehmen Methoden der Hilfsklasse CL_BWBCI_EXTRAC_TOOLS, die von der zentralen Klasse CL_BWBCI_EXTRAC bei Bedarf aufgerufen werden. Zum einen liest die Klasse CL_BWBCI_EXTRAC_TOOLS spezifische Informationen aus Datenbanktabellen aus und bereiten diese für die Verwendung in der zentralen Klasse CL_BWBCI_EXTRAC auf. Zum anderen ermittelt die Hilfsklasse die Aufrufe oder die Verwendung anderer Objekte, sowie die Deklaration von verschiedenen Datenobjekten innerhalb des Quelltexts. Ein DDIC-Objekt, welches sich noch nicht in der Objekttabelle befindet, wird ihr direkt hinzugefügt. Ein aufgerufenes oder verwendetes Quelltextobjekt, welches sich nicht in der selektierten Objektmenge befindet, wird in einer neuen Objektmenge vermerkt. Diese Objektmenge wird in einem iterativen Aufruf des SAP Code Inspector verarbeitet.

Die Extraktion für das FAMIX-Format übernimmt die Klasse CL_BWBCI_FAMIX. Über das Interface IF_BWBCI_EXTRAC greift die Klasse auf die Strukturinformationen in der Objekttabelle MT_DEVCLASS zu. Die Methode CREATE_FAMIX() der Klasse CL_BWBCI_FAMIX verarbeitet in einer Schleife alle Objekte, die sich in der Objekttabelle befinden. Für die Aufbereitung

der Informationen und die Verwaltung der ID's, siehe Abschnitt 3.3 greift die Methode CREATE_FAMIX() auf Methoden der Klasse CL_BWBCI_FAMIX_HELPER zu.

5.4 Datenhaltung

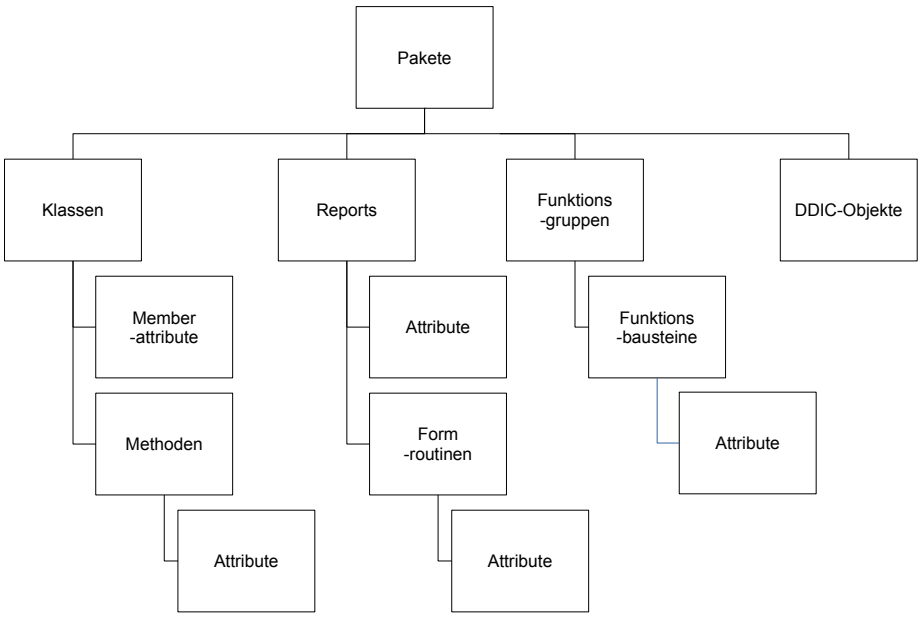

Abb. 5.6: Hierarchie der Objekttabelle für die analysierten Informationen

Die Objekttabelle dient als zentrale Informationshaltung der extrahierten Informationen für die spätere Verarbeitung. Der Aufbau der Objekttabelle orientiert sich dabei an dem Repository Browser der ABAP Workbench, der Entwicklungsumgebung eines SAP-Systems.

Pakete besitzen folgende Unterobjekte: Klassen, Reports, Funktionsgruppen, Datenbanktabellen, Datenelemente, Nachrichtenklassen und Strukturen. Die letzten vier sind in Abbildung 5.6 unter DDIC zusammengefasst. Klassen beinhalten wiederum neben ihren individuellen Eigenschaften Methoden und Memberattribute. Methoden besitzen Attribute als Unterobjekte. Reports können, ähnlich wie die Klasse, Attribute und Formroutinen enthalten. Letzteres enthält wiederum lokale Attribute. Wie in Unterabschnitt 2.2.2 dargestellt, sind Funktionsgruppen eine Zusammenfassung von mehreren Funktionsbausteinen. Diese besitzen wiederum Attribute als Unterobjekte.

Die Objekttabelle, siehe Abbildung 5.6, enthält zu jedem Objekttyp einen Tabellentyp. Die jeweilige Struktur zu einem Tabellentyp ist individuell auf den jeweiligen Objekttyp ausgerichtet. Zum Beispiel enthält die Struktur der Klasse die Spalte Superklasse, während die Attribut-Struktur eine Spalte für Datentypen besitzt. Diese verschiedenen Tabellentypen ermöglichen eine simple Erweiterung, falls eine weitere Information zu einem Objekttyp hinzugefügt werden soll. Diese Untertabellen zu einem Objekttyp sind als Referenz in einer Spalte eingehängt. Nur die Tabelle Pakete enthält eine Untertabelle Content, die elementare Informationen zu Klassen, Reports, Funktionsgruppen, Datenbanktabellen, Datenelemente, Nachrichtenklassen und Strukturen enthält. Methoden, Reports, Formroutinen und Funktionsbausteine enthalten zusätzlich eine Referenz auf eine Untertabelle mit Objekten, die in deren Quelltext aufgerufen werden.

5.5 Evaluation

Die in Abbildung 5.6 dargestellte Objekttabelle besitzt den Vorteil, dass allen Objekteinträgen individuelle Informationen zugeordnet werden können. Dies ist über die Verwendung verschiedener Strukturen, siehe Unterabschnitt 2.2.5, möglich. Zum Beispiel enthält die Struktur einer Klasse die Information über ihre Superklasse, was in der Struktur für ein Attribut nicht enthalten ist. Die Verwendung verschiedener Strukturen ergibt aber den Nachteil, dass die Suche nach einem spezifischen Objekt erschwert wird. Um die verschiedenen Teiltabellen zu durchsuchen, müssen die einzelnen Strukturen erst zugewiesen und dann durchsucht werden.

Die Architektur, siehe Abschnitt 5.3, ist darauf ausgerichtet, SAP-Strukturinformationen der SCI-Analyse zu extrahieren, mit weiteren Details zu ergänzen und in die Objekttabelle zu schreiben. Der SAP Code Inspector ist allerdings ein Werkzeug zur Analyse von Quelltext. Daher werden DDIC-Objekte wie Datenbanktabellen, Datenelemente und Strukturen nicht untersucht, mit Ausnahme von Datenbankaufrufen. Die Informationen über diese Elemente müssen durch den Prototypen selbstständig ermittelt werden.

Die Klasse `CL_BWBCI_FAMIX` benötigt in der aktuellen Version des Prototypen bei einer sehr großen Objektanzahl (schätzungsweise mehr als 60.000 Objekte in der Objekttabelle) mehr als zehn Minuten für die Generierung der String-Tabelle, die später als lokales FAMIX-Dokument abgespeichert wird. Das größte ausgegebene Dokument beinhaltete knapp 58.000 Objekte inklusive Referenzen. Da SAP-Standardcode oft in normalem Quelltext aufgerufen wird, werden viele SAP-Standardobjekte mit analysiert, wodurch schnell sehr viele Objekte, beispielsweise durch die komplette Analyse einer Funktionsgruppe anstatt eines aufgerufenen Funktionsbausteins, analysiert werden. Neben technischen Herausforderungen treten bei der Generierung der Visualisierung Fehler beim Einlesen der FAMIX-Referenzen auf. Diese konnten auch nach intensiver Suche und Refaktorisierung nicht behoben werden.

6 Ausblick

In Hinblick auf die Machbarkeit wurde bewiesen, dass eine Analyse und Extraktion von SAP- Strukturinformationen möglich ist. Weitere Elemente können in die Extraktion mitaufgenommen werden, um sie in Metaphern wie der *Recursive Disk Metaphor* oder der City Metaphor integrieren zu können. Auch können problemlos weitere Extraktoren für andere Formate als FAMIX problemlos hinzugefügt werden.

Allerdings war der Entwicklungsaufwand der aktuellen Version viel höher als erwartet. Während der Entwicklung des Prototypen wurde festgestellt, dass der SAP Code Inspector keine DDIC-Objekte untersucht und übergibt. Daher konnte nur ein Teil der nötigen Informationen aus der SCI-Analyse bezogen werden. Die Ermittlung und Analyse des anderen Teil musste mit zusätzlichem Entwicklungsaufwand in den Prototyp implementiert werden. Wie in Abschnitt 5.5 beschrieben, besitzt der entwickelte Prototyp noch Schwachstellen in der Architektur, der Objektermittlung und in der FAMIX-Generierung. Nachfolgend wird ein praktischer Ansatz beschrieben, wie diese Schwachstellen in einer verbesserten Version des Prototypen behoben werden können. Die Verwendung einer internen Tabelle birgt für die temporäre Speicherung vieler Strukturinformationen einige Probleme. Einerseits sind Datenhaltung und -verarbeitung nicht voneinander getrennt, wodurch das Prinzip der *Separation of Concerns* (engl. Trennung von Belangen) verletzt wird (Dijkstra, 1972). Andererseits kann die Objekttabelle bei einer großen Anzahl von Objekten schnell unübersichtlich sein. Um die Datenhaltung einfacher zu gestalten, bietet sich die Verwendung einer Datenklasse an, in der alle Strukturinformationen zentral gehalten werden.

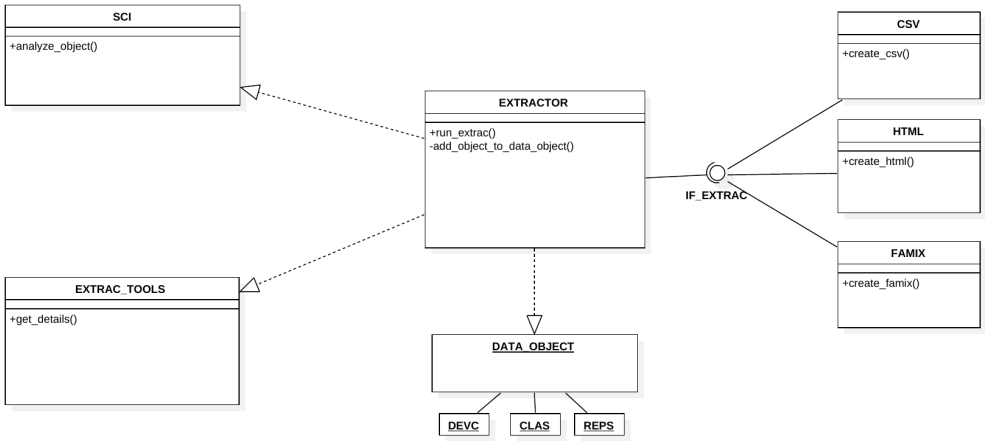

Abb. 6.1: Verbesserte Prototyparchitektur

Ein weiteres Problem ist die fehlende Analyse von DDIC-Objekten durch den SCI sowie die aufwendige Ermittlung aller Informationen zur Laufzeit. Letzteres ergibt sich aus der iterativen Verarbeitung des SCI. Die vollständigen Informationen zu einem aufgerufenen Programm, welches nicht in der ursprünglichen Objektmenge enthalten war, stehen erst nach der nächsten Iteration zu Verfügung.

Ein Lösungsansatz findet sich in Abbildung 6.1 in Form einer Standalone-Variante des Prototypen. Der Prototyp ist nicht mehr als eigene Prüfung an den SCI angebunden, sondern versteht sich als eigenständiges Programm. Die zu analysierenden Objekte entnimmt er, wie in der aktuellen Version, einer Objektmenge. Diese enthält im Unterschied zur aktuellen Version auch DDIC-Objekte, womit

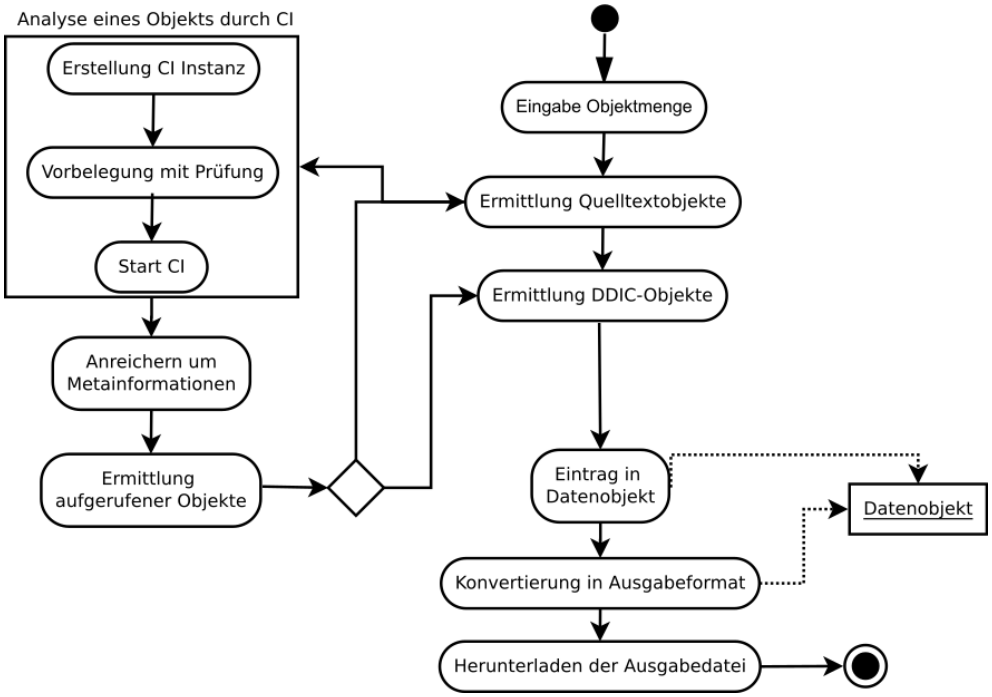

Abb. 6.2: Verbessertes Prozessverhalten des Prototypen

die Objekte viel spezifischer eingegrenzt werden können. Für die Analyse von Quelltextobjekten wie Klassen, Funktionsgruppen oder Reports wird der SAP Code Inspector direkt aufgerufen, siehe Abbildung 6.2. Dafür wird zur Laufzeit eine Objektmenge gebildet, welche nur dieses einzelne Objekt enthält, die dem SCI übergeben wird (vgl. Eilenberger et al., 2011, S.43). Außerdem werden ausschließlich Objekte analysiert, die von Interesse sind. Dies umgeht die Analyse ganzer Funktionsgruppen, obwohl aus dieser nur Informationen eines einzelnen Funktionsbausteins benötigt werden.

Dieser Lösungsansatz birgt auch Vorteile für die FAMIX-Generierung. Die kostenintensive Analyse der Verbindungen zwischen den analysierenden Objekten, zum Beispiel durch einen Aufruf, könnte in der verbesserten Architektur durch eine Referenz gelöst werden. In der FAMIX-Generierung stünden nach der Auflösung der Referenz ohne weitere Ermittlung alle Objektinformationen zur Verfügung. Außerdem ist eine reine Performanzoptimierung der Klasse `CL_BWBCI_FAMIX` möglich. Auf der einen Seite kann durch eine umfassendere Voranalyse und Aufbereitung der Daten in der Klasse `CL_BWBCI_EXTRAC` das aufgerufene Objekt direkt referenziert werden. Dadurch ist es nicht mehr nötig, die Objekttabelle in der Klasse `CL_BWBCI_FAMIX` nach zum Beispiel einem aufgerufenen Objekt zu durchsuchen. Auf der anderen Seite kann die erzeugte FAMIX-Dokument direkt auf den Applikationsserver geschrieben und anschließend heruntergeladen werden. Dadurch verringert sich einerseits der Bedarf an Arbeitsspeicher. So kann trotz eines Abbruchs die bis dahin erstellte Dokument heruntergeladen werden (vgl. Sascha Krüger & Jörg Seelmann-Eggebert, 2005, S.275).

Der Prototyp besitzt zwar noch Verbesserungsmöglichkeiten, erfüllt aber grundlegend die Anforderungen. Die SAP-Strukturinformationen werden in FAMIX extrahiert und können mit dem Softwarevisualisierungsgenerator nach Müller et al. (2011) mit der Erweiterung nach Roth (2017) visualisiert werden. In weiteren Arbeiten können die genannten Verbesserungen implementiert werden, um ei-

nerseits die Performanz zu erhöhen und andererseits mehr Informationen für die Visualisierung zu ermitteln.

Anhang

A.1 FAMIX Grammatik Definition mit Xtext

```
grammar org.svis.xtext.Famix with org.eclipse.xtext.common.Terminals

import "http://www.eclipse.org/emf/2002/Ecore" as ecore
generate famix "http://www.svis.org/famix"

Root:
    document=Document?;

Document:
    {Document}
    '(' elements+=FAMIXElement* ')';

FAMIXElement:
    FAMIXStructure | FAMIXFileAnchor | FAMIXInvocation | FAMIXAttribute |
    FAMIXInheritance | FAMIXAccess | FAMIXNamespace | FAMIXMethod |
    FAMIXPrimitiveType | FAMIXComment | FAMIXParameter | FAMIXReference |
    FAMIXParameterizedType | FAMIXAnnotationInstance |
    FAMIXAnnotationInstanceAttribute | FAMIXAnnotationTypeAttribute |
    FAMIXLocalVariable | FAMIXImplicitVariable | FAMIXType |
    FAMIXParameterType | FAMIXJavaSourceLanguage | FAMIXDeclaredException |
    FAMIXThrownException | FAMIXCaughtException | FAMIXEnumValue |
    FAMIXDevclass | FAMIXReport | FAMIXFormroutine | FAMIXFunctionModule |
    FAMIXTable | FAMIXDataElement | FAMIXABAPStruc | FAMIXStrucElement
;

FAMIXStructure:
    FAMIXClass | FAMIXParameterizableClass | FAMIXEnum |
    FAMIXAnnotationType | FAMIXFunctionGroup | FAMIXMessageClass
;

FAMIXNamespace:
    '(FAMIX.Namespace'
    '(' 'id: ' name=INT_ID ')'
    '(' 'name' value=MSESTRING ')'
    ('(' 'hash' id = STRING ')')?
    ('(' 'fqn' fqn = MSESTRING ')')?
    ('(' 'isStub' isStub=Boolean ')')?
    ('(' 'parentScope' parentScope=IntegerReference ')')?
    ('(' 'iteration' iteration=INT ')')?
    ')';

FAMIXFileAnchor:
    '(FAMIX.FileAnchor'
    '(' 'id: ' name=INT_ID ')'
    ('(' 'element' element=IntegerReference ')')?
    '(' 'endLine' endline=INT ')'
    '(' 'fileName' filename=MSESTRING ')'
```

```
48      '(' 'startLine' startline=INT ')'
49      ')';
50
51  FAMIXClass:
52      '(FAMIX.Class'
53      '(' 'id: ' name=INT_ID ')'
54      '(' 'name' value=MSESTRING ')'
55      ('(' 'hash' id = STRING ')')?
56      ('(' 'fqn' fqn = MSESTRING ')')?
57      '(' 'container' container=IntegerReference ')'
58      ('(' 'isInterface' isInterface=Boolean ')')?
59      ('(' 'isStub' isStub=Boolean ')')?
60      ('(' 'modifiers' modifiers+=MSESTRING* ')')?
61      ('(' 'sourceAnchor' type=IntegerReference ')')?
62      ('(' 'iteration' iteration=INT ')')?
63      ')';
64
65  FAMIXParameterizableClass:
66      '(FAMIX.ParameterizableClass'
67      '(' 'id: ' name=INT_ID ')'
68      '(' 'name' value=MSESTRING ')'
69      ('(' 'hash' id = STRING ')')?
70      ('(' 'fqn' fqn = MSESTRING ')')?
71      '(' 'container' container=IntegerReference ')'
72      ('(' 'isInterface' isInterface=Boolean ')')?
73      ('(' 'isStub' isStub=Boolean ')')?
74      ('(' 'modifiers' modifiers+=MSESTRING* ')')?
75      ('(' 'sourceAnchor' type=IntegerReference ')')?
76      ')';
77
78  FAMIXMethod:
79      '(FAMIX.Method'
80      '(' 'id: ' name=INT_ID ')'
81      '(' 'name' value=MSESTRING ')'
82      ('(' 'hash' id = STRING ')')?
83      ('(' 'fqn' fqn = MSESTRING ')')?
84      ('(' 'cyclomaticComplexity' cyclomaticComplexity=INT ')')?
85      ('(' 'declaredType' declaredType=IntegerReference ')')?
86      ('(' 'hasClassScope' hasClassScope=Boolean ')')?
87      ('(' 'isStub' isStub=Boolean ')')?
88      ('(' 'kind' kind=MSESTRING ')')?
89      ('(' 'modifiers' modifiers+=MSESTRING* ')')?
90      ('(' 'numberOfStatements' numberOfStatements=INT ')')?
91      '(' 'parentType' parentType=IntegerReference ')'
92      '(' 'signature' signature=MSESTRING ')'
93      ('(' 'sourceAnchor' sourceAnchor=IntegerReference ')')?
94      ('(' 'iteration' iteration=INT ')')?
95      ')';
96
97  FAMIXInvocation:
98      '(FAMIX.Invocation'
99      '(' 'id: ' name=INT_ID ')'
100     '(' 'candidates' candidates=IntegerReference ')'
101     ('(' 'previous' previous=IntegerReference ')')?
102     ('(' 'receiver' receiver=IntegerReference ')')?
```

```
103     '(' 'sender' sender=IntegerReference ')'
104     '(' 'signature' signature=MSESTRING ')'
105     ')';
106
107 FAMIXAttribute:
108     '(FAMIX.Attribute'
109     '(' 'id: ' name=INT_ID ')'
110     '(' 'name' value=MSESTRING ')'
111     ('(' 'hash' id = STRING ')')?
112     ('(' 'fqn' fqn = MSESTRING ')')?
113     ('(' 'declaredType' declaredType=IntegerReference ')')?
114     ('(' 'hasClassScope' hasClassScope=Boolean ')')?
115     ('(' 'isStub' isStub=Boolean ')')?
116     ('(' 'modifiers' modifiers+=MSESTRING* ')')?
117     '(' 'parentType' parentType=IntegerReference ')'
118     ('(' 'sourceAnchor' sourceAnchor=IntegerReference ')')?
119     ('(' 'dataType' dataType=MSESTRING ')')?
120     ('(' 'iteration' iteration=INT ')')?
121     ')';
122
123 FAMIXAccess:
124     '(FAMIX.Access'
125     '(' 'id: ' name=INT_ID ')'
126     '(' 'accessor' accessor=IntegerReference ')'
127     ('(' 'isWrite' isWrite=Boolean ')')?
128     ('(' 'previous' previous=IntegerReference ')')?
129     '(' 'variable' variable=IntegerReference ')'
130     ')';
131
132 FAMIXPrimitiveType:
133     '(FAMIX.PrimitiveType'
134     '(' 'id: ' name=INT_ID ')'
135     '(' 'name' value=MSESTRING ')'
136     '(' 'isStub' isStub=Boolean ')'
137     ')';
138
139 FAMIXComment:
140     '(FAMIX.Comment'
141     '(' 'id: ' name=INT_ID ')'
142     '(' 'container' container=IntegerReference ')'
143     '(' 'content' content=MSESTRING ')'
144     '(' 'sourceAnchor' sourceAnchor=IntegerReference ')'
145     ')';
146
147 FAMIXParameter:
148     '(FAMIX.Parameter'
149     '(' 'id: ' name=INT_ID ')'
150     '(' 'name' value=MSESTRING ')'
151     ('(' 'fqn' fqn = MSESTRING ')')?
152     '(' 'declaredType' declaredType=IntegerReference ')'
153     '(' 'parentBehaviouralEntity' parentBehaviouralEntity=IntegerReference ')'
154     ')';
155
156 FAMIXInheritance:
157     '(FAMIX.Inheritance'
```

```
158     '(' 'id: ' name=INT_ID ')'
159     ('(' 'previous' previous=IntegerReference ')')?
160     '(' 'subclass' subclass=IntegerReference ')'
161     '(' 'superclass' superclass=IntegerReference ')'
162     ')';
163
164 FAMIXReference:
165     '(FAMIX.Reference'
166     '(' 'id: ' name=INT_ID ')'
167     '(' 'source' source=IntegerReference ')'
168     '(' 'target' target=IntegerReference ')'
169     ')';
170
171 FAMIXParameterizedType:
172     '(FAMIX.ParameterizedType'
173     '(' 'id: ' name=INT_ID ')'
174     '(' 'name' value=MSESTRING ')'
175     ('(' 'hash' id = STRING ')')?
176     ('(' 'fqn' fqn = MSESTRING ')')?
177     ('(' 'arguments' arguments+=IntegerReference* ')')?
178     '(' 'container' container=IntegerReference ')'
179     ('(' 'isStub' isStub=Boolean ')')?
180     '(' 'parameterizableClass' parameterizableClass=IntegerReference ')'
181     ')';
182
183 FAMIXAnnotationInstance:
184     '(FAMIX.AnnotationInstance'
185     '(' 'id: ' name=INT_ID ')'
186     '(' 'annotatedEntity' annotatedEntity=IntegerReference ')'
187     '(' 'annotationType' annotationType=IntegerReference ')'
188     ')';
189
190 FAMIXAnnotationInstanceAttribute:
191     '(FAMIX.AnnotationInstanceAttribute'
192     '(' 'id: ' name=INT_ID ')'
193     ('(' 'annotationTypeAttribute' annotationTypeAttribute=IntegerReference ')')?
194     ('(' 'parentAnnotationInstance' parentAnnotationInstance=IntegerReference ')')?
195     '(' 'value' value=MSESTRING ')'
196     ')';
197
198 FAMIXAnnotationType:
199     '(FAMIX.AnnotationType'
200     '(' 'id: ' name=INT_ID ')'
201     '(' 'name' value=MSESTRING ')'
202     ('(' 'hash' id = STRING ')')?
203     ('(' 'fqn' fqn = MSESTRING ')')?
204     '(' 'container' container=IntegerReference ')'
205     ('(' 'isStub' isStub=Boolean ')')?
206     ('(' 'modifiers' modifiers+=MSESTRING* ')')?
207     ('(' 'sourceAnchor' sourceAnchor=IntegerReference ')')?
208     ')';
209
210 FAMIXAnnotationTypeAttribute:
211     '(FAMIX.AnnotationTypeAttribute'
212     '(' 'id: ' name=INT_ID ')'
```

```
213     '(' 'name' value=MSESTRING ')'
214     ('(' 'isStub' isStub=Boolean ')')?
215     ('(' 'modifiers' modifiers+=MSESTRING* ')')?
216     '(' 'parentType' parentType=IntegerReference ')'
217     ('(' 'sourceAnchor' sourceAnchor=IntegerReference ')')?
218     ')';
219
220 FAMIXLocalVariable:
221     '(FAMIX.LocalVariable'
222     '(' 'id: ' name=INT_ID ')'
223     '(' 'name' value=MSESTRING ')'
224     ('(' 'declaredType' declaredType=IntegerReference ')')?
225     ('(' 'isStub' isStub=Boolean ')')?
226     '(' 'parentBehaviouralEntity' parentBehaviouralEntity=IntegerReference ')'
227     ('(' 'sourceAnchor' sourceAnchor=IntegerReference ')')?
228     ')';
229
230 FAMIXImplicitVariable:
231     '(FAMIX.ImplicitVariable'
232     '(' 'id: ' name=INT_ID ')'
233     '(' 'name' value=MSESTRING ')'
234     ('(' 'parentBehaviouralEntity' parentBehaviouralEntity=IntegerReference ')')?
235     ')';
236
237 FAMIXType:
238     '(FAMIX.Type'
239     '(' 'id: ' name=INT_ID ')'
240     '(' 'name' value=MSESTRING ')'
241     '(' 'container' container=IntegerReference ')'
242     '(' 'isStub' isStub=Boolean ')'
243     ')';
244
245 FAMIXParameterType:
246     '(FAMIX.ParameterType'
247     '(' 'id: ' name=INT_ID ')'
248     '(' 'name' value=MSESTRING ')'
249     ('(' 'fqn' fqn = MSESTRING ')')?
250     '(' 'container' container=IntegerReference ')'
251     ('(' 'isStub' isStub=Boolean ')')?
252     ')';
253
254 FAMIXJavaSourceLanguage:
255     '(FAMIX.JavaSourceLanguage'
256     '(' 'id: ' name=INT_ID ')'
257     ')';
258
259 FAMIXDeclaredException:
260     '(FAMIX.DeclaredException'
261     '(' 'id: ' name=INT_ID ')'
262     '(' 'definingMethod' definingMethod=IntegerReference ')'
263     '(' 'exceptionClass' exceptionClass=IntegerReference ')'
264     ')';
265
266 FAMIXThrownException:
267     '(FAMIX.ThrownException'
```

```
268     '(' 'id: ' name=INT_ID ')'
269     '(' 'definingMethod' definingMethod=IntegerReference ')'
270     '(' 'exceptionClass' exceptionClass=IntegerReference ')'
271     ')';
272
273 FAMIXCaughtException:
274     '(FAMIX.CaughtException'
275     '(' 'id: ' name=INT_ID ')'
276     '(' 'definingMethod' definingMethod=IntegerReference ')'
277     '(' 'exceptionClass' exceptionClass=IntegerReference ')'
278     ')';
279
280 FAMIXEnum:
281     '(FAMIX.Enum'
282     '(' 'id: ' name=INT_ID ')'
283     '(' 'name' value=MSESTRING ')'
284     ('(' 'hash' id = STRING ')')?
285     ('(' 'fqn' fqn = MSESTRING ')')?
286     '(' 'container' container=IntegerReference ')'
287     ('(' 'isStub' isStub=Boolean ')')?
288     ('(' 'modifiers' modifiers+=MSESTRING* ')')?
289     ('(' 'sourceAnchor' sourceAnchor=IntegerReference ')')?
290     ')';
291
292 FAMIXEnumValue:
293     '(FAMIX.EnumValue'
294     '(' 'id: ' name=INT_ID ')'
295     '(' 'name' value=MSESTRING ')'
296     ('(' 'hash' id = STRING ')')?
297     ('(' 'fqn' fqn = MSESTRING ')')?
298     ('(' 'isStub' isStub=Boolean ')')?
299     '(' 'parentEnum' parentEnum=IntegerReference ')'
300     ')';
301
302 FAMIXDevclass:
303     '(FAMIX.Devclass'
304     '(' 'id: ' name=INT_ID ')'
305     '(' 'name' value=MSESTRING ')'
306     ('(' 'hash' id = STRING ')')?
307     ('(' 'fqn' fqn = MSESTRING ')')?
308     ('(' 'isStub' isStub=Boolean ')')?
309     ('(' 'parentScope' parentScope=IntegerReference ')')?
310     ('(' 'iteration' iteration=INT ')')?
311     ')';
312
313 FAMIXReport:
314     '(FAMIX.Report'
315     '(' 'id: ' name=INT_ID ')'
316     '(' 'name' value=MSESTRING ')'
317     ('(' 'hash' id = STRING ')')?
318     ('(' 'fqn' fqn = MSESTRING ')')?
319     '(' 'container' container=IntegerReference ')'
320     ('(' 'numberOfStatements' numberOfStatements=INT ')')?
321     ('(' 'iteration' iteration=INT ')')?
322     ')';
```

```
323
324 FAMIXFormroutine:
325     '(FAMIX.Formroutine'
326     '(' 'id: ' name=INT_ID ')'
327     '(' 'name' value=MSESTRING ')'
328     ('(' 'hash' id = STRING ')')?
329     ('(' 'fqn' fqn = MSESTRING ')')?
330     ('(' 'numberOfStatements' numberOfStatements=INT ')')?
331     '(' 'parentType' parentType=IntegerReference ')'
332     ('(' 'iteration' iteration=INT ')')?
333     ')';
334
335 FAMIXFunctionGroup:
336     '(FAMIX.FunctionGroup'
337     '(' 'id: ' name=INT_ID ')'
338     '(' 'name' value=MSESTRING ')'
339     ('(' 'hash' id = STRING ')')?
340     ('(' 'fqn' fqn = MSESTRING ')')?
341     '(' 'container' container=IntegerReference ')'
342     ('(' 'iteration' iteration=INT ')')?
343     ')';
344
345 FAMIXFunctionModule:
346     '(FAMIX.FunctionModule'
347     '(' 'id: ' name=INT_ID ')'
348     '(' 'name' value=MSESTRING ')'
349     ('(' 'hash' id = STRING ')')?
350     ('(' 'fqn' fqn = MSESTRING ')')?
351     '(' 'parentType' parentType=IntegerReference ')'
352     ('(' 'numberOfStatements' numberOfStatements=INT ')')?
353     ('(' 'iteration' iteration=INT ')')?
354     ')';
355
356 FAMIXMessageClass:
357     '(FAMIX.MessageClass'
358     '(' 'id: ' name=INT_ID ')'
359     '(' 'name' value=MSESTRING ')'
360     ('(' 'hash' id = STRING ')')?
361     ('(' 'fqn' fqn = MSESTRING ')')?
362     '(' 'container' container=IntegerReference ')'
363     ('(' 'numberOfMessages' numberOfMessages=INT ')')?
364     ('(' 'iteration' iteration=INT ')')?
365     ')';
366
367 FAMIXTable:
368     '(FAMIX.Table'
369     '(' 'id: ' name=INT_ID ')'
370     '(' 'name' value=MSESTRING ')'
371     ('(' 'hash' id = STRING ')')?
372     ('(' 'fqn' fqn = MSESTRING ')')?
373     '(' 'container' container=IntegerReference ')'
374     ('(' 'numberOfColumns' numberOfColumns=INT ')')?
375     ('(' 'rowCategory' rowCategory=INT ')')?
376     ('(' 'iteration' iteration=INT ')')?
377     ')';
```

```
378
379 FAMIXDataElement:
380     '(FAMIX.DataElement'
381     '(' 'id: ' name=INT_ID ')'
382     '(' 'name' value=MSESTRING ')'
383     ('(' 'hash' id = STRING ')')?
384     ('(' 'fqn' fqn = MSESTRING ')')?
385     '(' 'container' container=IntegerReference ')'
386     ('(' 'dataType' datatype=MSESTRING ')')?
387     ('(' 'iteration' iteration=INT ')')?
388     ')';
389
390 FAMIXABAPStruc:
391     '(FAMIX.ABAPStructure'
392     '(' 'id: ' name=INT_ID ')'
393     '(' 'name' value=MSESTRING ')'
394     ('(' 'hash' id = STRING ')')?
395     ('(' 'fqn' fqn = MSESTRING ')')?
396     '(' 'container' container=IntegerReference ')'
397     ('(' 'iteration' iteration=INT ')')?
398     ')';
399
400 FAMIXStrucElement:
401     '(FAMIX.StrucElement'
402     '(' 'id: ' name=INT_ID ')'
403     '(' 'name' value=MSESTRING ')'
404     ('(' 'hash' id = STRING ')')?
405     ('(' 'fqn' fqn = MSESTRING ')')?
406     '(' 'container' container=IntegerReference ')'
407     ('(' 'dataType' datatype=MSESTRING ')')?
408     ('(' 'dataElement' dataElement=IntegerReference ')')?
409     ('(' 'iteration' iteration=INT ')')?
410     ')';
411
412 Boolean:
413     'true' | 'false';
414
415 IntegerReference:
416     '(' 'ref: ' ref=[FAMIXElement|INT_ID] ')';
417
418 INT_ID returns ecore::EString:
419     '^'? INT;
420
421 terminal MSESTRING:
422     ("'"->"'")*;
```

A.2 SAP-Objekte

Überobjekte	**Objekt**	**Unterobjekte**
ABAP Dictionary	Datenbanktabellen	
	Datenelemente	
	Strukturen	
	Tabellentypen	
	Views	
	Domänen	
	Suchhilfen	
	Sperrobjekte	
	Typgruppen	
	Datenbankprozedur-Proxies	
	Matchcodeobjekte	
	Matchcode-Ids	
	Pool/Cluster-Tabellen	
	Tabellenindizes	
	Felder	
Programmbibliothek	Reports	
	Funktionsgruppen	
	Funktionsbausteine	
	Includes	
	Teilobjekte	Varianten
		Globale Daten
		Globale Typen
		Unterprogramme
		PBO-Module
		PAI-Module
		Makros
		Dynpros
		GUI-Status
		GUI-Titel
		GUI-Funktionen
		Klassen
		Interfaces
Klassenbibliothek	Klassen/Interfaces	
	Methoden	
	Attribute	
	Ereignisse	
	Typen	
Web Dynpro	Web Dynpro Components	

	Web Dynpro Anwendungen	
	Web Dynpro CHIPs	
	Component Konfigurationen	
	Anwendungs-Konfigurationen	
BSP-Bibliothek	BSP-Applikationen	
	BSP-Extension	
Enterprise Services	Service Definitionen	
Erweiterungen	Business Add Ins	Definitionen
		Implementierungen
	Customer Exits	Erweiterungen
		Projekte
	Erweiterungsimplementierungen	
	Zusammengesetzte Erweiterungsimplementierungen	
	Erweiterungsspots	
	Zusammengesetzte Erweiterungsspots	
Testobjekte	CATT Testfälle	
	eCATT	Testkonfigurationen
		Teskskripte
		Testdaten
		Systemdaten
Weitere Objekte	Web Objekte	ITS Service
		Themen
	Transaktionen	
	Dialogbausteine	
	Logische Datenbanken	
	SPA/GPA-Parameter	
	Nachrichtenklassen	
	Nachrichtennummern	
	Bereichsmenüs	
	Berechtigungsobjekte	
	Aktivierungs-Ids	
	XSLT-Programm	
	Ausnahmeklassen	
	Namespaces	

Literaturverzeichnis

Balogh, G., Gergely, T., Beszédes, Á. & Gyimóthy, T. (2016). "Using the City Metaphor for Visualizing Test-Related Metrics". In: *IEEE International Conference on Software Analysis, Evolution and Reengineering*. Bd. 2, S. 17–20. ISBN: 9781509018550. DOI: 10.1109/SANER.2016.48 (siehe S. 2).

Brockhaus (2006). *Brockhaus. Enzyklopädie in 30 Bänden*. Hrsg. von Annette Zwahr. 21., neu b. Leipzig; Mannheim: Brockhaus (siehe S. 18).

Carr, Mahil & Verner, June (2004). "Prototyping and Software Development Approaches". In: *Prototyping and Software Development Approaches* 3, S. 1–16 (siehe S. 4).

Diehl, S. (2003). "Softwarevisualisierung". In: *Informatik-Spektrum* 26.4, S. 257–260. ISSN: 0170-6012. DOI: 10.1007/s00287-003-0314-4. URL: http://link.springer.com/10.1007/s00287-003-0314-4 (siehe S. 1).

Diehl, Stephan (2007). *Software visualization : Visualizing the structure, behaviour, and evolution of software; with ... 5 tables*. Springer, XII, 187 S. ISBN: 3540465049 (siehe S. 2, 18).

Dijkstra, Edsger W. (1972). "Notes on structured programming". In: *Structured programming*. Structured Programming. London: Academic Press Ltd., S. 1–82. ISBN: 0-12-200550-3. URL: http://www.informatik.uni-bremen.de/agbkb/lehre/programmiersprachen/artikel/EWD-notes-structured.pdf%7B%5C%%7D5Cnhttp://www.narcis.nl/publication/RecordID/oai:library.tue.nl:252825 (siehe S. 35).

Ducasse, Stéphane, Anquetil, Nicolas, Bhatti, Muhammad Usman, Hora, Andre Cavalcante, Laval, Jannik & Girba, Tudor (2011). "MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family". In: November, S. 40–40. URL: https://hal.inria.fr/hal-00646884/ (siehe S. 19).

Efftinge, Sven (2014). *Xtext Documentation*. URL: https://eclipse.org/Xtext/documentation/2.7.0/Xtext%20Documentation.pdf (siehe S. 19).

Eilenberger, Randolf, Ruggaber, Frank & Schilcher, Reinhard (2011). *Praxishandbuch SAP Code Inspector*. Galileo Press, S. 466. ISBN: 9783836212861. URL: http://91.216.243.21/fachbuch/leseprobe/9783836217064%7B%5C_%7DExcerpt%7B%5C_%7D001.pdf (siehe S. 13 f., 16 f., 36).

Gupta, Tanmaya (2011). *ABAP Data Dictionary*. SAP Press, S. 403. ISBN: 9781592293797 1592293794. URL: https://www.sap-press.com/abap-data-dictionary%7B%5C_%7D2540/ (siehe S. 12 f.).

Igd, Fraunhofer (2014). *X3DOM Documentation*. URL: https://doc.x3dom.org/, Zugriff am 11. 07. 2017 (siehe S. 18).

Jens Präkelt (GiSA GmbH) (2014). "Entwicklung eines objektorientierten Programmgerüsts zur Sicherstellung der GISA-Entwicklungsrichtlinien in SAP- Systemlandschaften 16.09.2014". Bachelorarbeit. Hochschule Merseburg (siehe S. 7–10, 15 ff., 28).

Kappauf, Jens, Lauterbach, Bernd & Koch, Matthias (2012). *Logistic Core Operations with SAP - Warehouse Logistics and Inventory Management*. 1st. 1. Springer Publishing Company, Incorporated, S. 1–5. ISBN: 9780874216561. DOI: 10.1007/s13398-014-0173-7.2. arXiv: arXiv:1011.1669v3 (siehe S. 6 f., 9).

Knight, Claire & Munro, Malcolm (1999). *Comprehension with[in] Virtual Environment Visualisations*. DOI: http://doi.ieeecomputersociety.org/10.1109/WPC.1999.777733. URL: http://ieeexplore.ieee.org/xpls/abs%7B%5C%%7D7B%7B%5C_%7D%7B%5C%%7D7Dall.jsp?arnumber=777733 (siehe S. 1).

Kovacs, Pascal (2010). "Ansatz zur Interaktion mit dreidimensional visualisierten Softwaremodellen". Diplomarbeit. Universität Leipzig. URL: http://nbn-resolving.de/urn:nbn:de:bsz:15-qucosa-71070 (siehe S. 2).

Krüger, Horst Keller; Sascha (2009). *ABAP Objects; Korrigierter Nachdruck*. SAP Press (siehe S. 11 f.).

Mormann, Hannah (2014). "Das Projekt SAP". Dissertation. Universität Bielefeld. ISBN: 9783837633764 (siehe S. 6, 8, 28).

Müller, Richard (2015). "Software Visualization in 3D - Implementation, Evaluation, and Applicability". Dissertation. Universität Leipzig, S. 126. URL: http://nbn-resolving.de/urn:nbn:de:bsz:15-qucosa-164699 (siehe S. 2 f., 19, 22 f., 27).

Müller, Richard, Kovacs, Pascal, Schilbach, Jan & Eisenecker, Ulrich W. (2011). "Generative Software Visualizaion: Automatic Generation of User-Specific Visualisations". In: *Proceedings of the International Workshop on Digital Engineering*, S. 45–49 (siehe S. 2, 18 f., 27, 36).

Müller, Richard & Zeckzer, Dirk (2015). "The Recursive Disk Metaphor: A Glyph-based Approach for Software Visualization". In: *6th International Conference on Information Visualization Theory and Applications (IVAPP 2015)*, S. 171–176. ISBN: 9789897580888. DOI: 10.5220/0005342701710176. URL: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0005342701710176 (siehe S. 1, 3, 18, 20, 23 ff.).

Naumann, Justus D. & Jenkins, A. Milton (1982). "Prototyping: The New Paradigm for Systems Development." In: *MIS Quarterly* 6.3, S. 29–44. ISSN: 02767783. DOI: 10.2307/248654. URL: http://dx.doi.org/10.2307/248654 (siehe S. 3 f.).

Rohr, Oliver (2007). *Softwarevisualisierung in mehreren Dimensionen*. Saarbrücken: VDM Verlag, S. 140. ISBN: 9783836417259 (siehe S. 23).

Roth, Johannes (2017). "Visualisierung von ABAP-Software mittels generativer Softwarevisualisierung". Bachelorarbeit. Universität Leipzig (siehe S. 2 f., 27, 29 f., 36).

Sascha Krüger & Jörg Seelmann-Eggebert (2005). *ABAP Best Practises - Lösungen für die täglichen Aufgaben der ABAP-Programmierung*. SAP Press, S. 275–290. ISBN: 3898423549. URL: https://buch-findr.de/buecher/abap-best-practices/ (siehe S. 36).

Statista (2013). *Marktanteile der führenden Anbieter am Umsatz mit ERP-Software in Deutschland bis 2013*. URL: https://de.statista.com/statistik/daten/studie/262275/umfrage/marktanteile-der-anbieter-von-erp-software-in-deutschland/, Zugriff am 22. 07. 2017 (siehe S. 6).

Steinberg, David, Budinsky, Frank, Paternostro, Marcelo & Merks, Ed (2009). *EMF: Eclipse Modeling Framework 2.0*. 2nd. Addison-Wesley Professional. ISBN: 0321331885 (siehe S. 18).

Theobald, Patrick (2007). *Profikurs ABAP ®*. 2. Aufl. Vieweg+Teubner Verlag, S. 316. ISBN: 9783834801432 (siehe S. 10, 12).

VirtualForge (2017). *CODEPROFILER for ABAP*. URL: https://www.virtualforge.com/de/leistungen/codeprofiler.html, Zugriff am 24. 04. 2017 (siehe S. 8).

Ware, Collin (2004). *Information Vizualization: Perception for Design*. 3. Aufl. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., S. xxvi, 27. ISBN: 1558608192 (siehe S. 21, 25).

Wettel, Richard, Lanza, Michele & Robbes, Romain (2011). "Software systems as cities: a controlled experiment". In: *2011 33rd International Conference on Software Engineering (ICSE)*. ICSE '11. New York, NY, USA: ACM, S. 551–560. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985868. arXiv: arXiv:1011.1669v3. URL: https://wettel.github.io/download/Wettel11a-icse.pdf (siehe S. 2).

Zilch, Denise (2015). "Generative und modellgetriebene Softwarevisualisierung am Beispiel der Stadtmetapher". Bachelorarbeit. Universität Leipzig, S. 32 (siehe S. 1 f.).

❧ Kolophon

Diese Bachelorarbeit wurde mit LaTeX 2ε und BibLaTeX erstellt und in TeXstudio geschrieben. Kapitel- und Abschnittsüberschriften wurden in Helvetica Neue gesetzt. UML-Diagramme wurden mit StarUML und Dia Diagram Editor erstellt, Schemata mit LibreOffice Draw. Die Definition der FAMIX-Grammatik wurde mit IntelliJ IDE von *JetBrains* geschrieben.

Selbstständigkeitserklärung

Ich versichere, dass ich diese Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Darüber hinaus versichere ich, dass die elektronische Version der Bachelorarbeit mit der gedruckten Version übereinstimmt.

Ort, Datum	Unterschrift