

Universität Leipzig
Wirtschaftswissenschaftliche Fakultät
Institut für Softwareentwicklung
Professor Eisenecker

Thema

Visualisierung der Struktur von ABAP-Software mittels generativer Softwarevisualisierung

Bachelorarbeit zur Erlangung des akademischen Grades Bachelor of Science -
Wirtschaftsinformatik

vorgelegt von: Roth, Johannes
Matrikelnummer: 3714792
Email-Adresse: johannes@roth24.de
Telefonnummer: 017634012380
Anschrift: Landsberger Str. 96
04157 Leipzig

Leipzig, den 11. September 2017

Abstract

Der Forschungsbereich der generativen Softwarevisualisierung beschäftigt sich mit dem Problem, automatisch individuell konfigurierte Visualisierungen von Software zu erzeugen, um deren Struktur und Funktionsweise durch visuelle Mittel einfacher verständlich zu machen. Für die Darstellung werden sogenannte Metaphern verwendet, die es, anhand bekannter Anhaltspunkte, ermöglichen sich in der entsprechend visualisierten Software zurechtzufinden.

Dies ist bereits für verschiedene objektorientierte Programmiersprachen möglich, jedoch nicht für solche, die einige Spezialfälle beinhalten, wie beispielsweise ABAP. Das Ziel dieser Arbeit ist es, in einer methodischen Vorgehensweise einen prototypischen Generator, der eine entsprechend angepasste City-Metapher einer ABAP-Software erzeugen kann, zu entwickeln. Dabei wurde die Programmiersprache an sich, aber auch die typische Arbeitsumgebung des ABAP-Entwicklers analysiert, um den praktischen Nutzen der entstandenen Metapher zu optimieren.

Schlüsselwörter

Softwarevisualisierung, Generative Softwarevisualisierung, City-Metapher, ABAP, X3DOM, Xtext

Inhaltsverzeichnis

Abbildungsverzeichnis	IV
Tabellenverzeichnis	V
Listings	VI
Algorithmenverzeichnis	VII
Glossar	VIII
1 Einleitung	1
1.1 Motivation	1
1.2 Problem- und Zielstellung	2
1.3 Struktur der Arbeit	3
2 Generative Softwarevisualisierung	4
2.1 Softwarevisualisierung	4
2.2 Methoden der generativen Softwarevisualisierung	6
2.3 X3DOM	8
3 Visualisierung von ABAP	10
3.1 Besonderheiten von ABAP	10
3.2 Zu visualisierende Sprachelemente	12
3.3 Darstellungsmöglichkeiten in der City-Metapher	15
4 Entwicklung eines Prototypen zur Visualisierung	19
4.1 Ausgangspunkt	19
4.2 Erweiterung des Generators	20
4.3 FAMIX to City	27
4.4 City to City	28
4.5 City to X3DOM	31

Inhaltsverzeichnis

4.6 Evaluation	32
5 Fazit und Ausblick	34
A Anhang	IX
A.1 Tabellen	IX
Literaturverzeichnis	XVI

Abbildungsverzeichnis

1.1	CodeCity-Visualisierung des JDK v1.5 ¹	1
2.1	Visualisierungspipeline nach Card [CMS97]	5
2.2	Model zur generativen Softwarevisualisierung [MKSE11]	6
2.3	gerenderte Darstellung eines in x3d definierten Würfels	9
3.1	SAP Object Navigator	13
3.2	Überblick der Stadt-Metapher nach Wettel [Wet10]	16
4.1	Übersicht über die Bestandteile des erweiterten Generators	19
4.2	Darstellung eines Paketes mittels der erweiterten „City“-Metapher .	21
4.3	Abgrenzung durch Texturen	22
4.4	Kreisförmige Positionierung von Datenbanken	23
4.5	Positionierung von Attributen unter den zugehörigen Klassen	24
4.6	Verbindung mit projektfremden Paketen	24
4.7	Darstellung von Referenzen	25
4.8	Vergleich der Einstellungen zur Darstellung projektfremder Pakete .	25
4.9	mehrere Pakete in projekteigenen Elementen	26

Tabellenverzeichnis

3.1	Verbleibende ABAP-Sprachelemente	14
3.2	Visualisierungsmöglichkeiten der Advanced Business Application Programming (ABAP)-Sprachelemente	18
4.1	Mapping von FAMIX-Elementen auf City-Elemente	27
A.1	Verbleibende ABAP-Sprachelemente (vollständig)	X
A.2	Alle ABAP-Sprachelemente	XIII
A.3	Konfigurationsmöglichkeiten des Generators	XV

Listings

2.1	Template-Ausdruck in Xtend	7
2.2	Definition einer DSL in Xtext	7
2.3	x3d-Quelltext zur Erzeugung eines roten Würfels	9

Algorithmenverzeichnis

1	Funktion zur kreisförmigen Positionierung um ein City-Element in einem Kreis	30
2	Methode zur rechteckigen Positionierung auf ein City-Element	31

Glossar

ABAP Advanced Business Application Programming. II, V, 2–6, 10–21, 25, 26, 31, 32

DSL Domain Specific Language. IV, 7, 8, 20

Dynpro Dynamisches Programm. 11, 12, 14, 35

SQL Structured Query Language. 10

VM Virtual Machine. 10

1 Einleitung

1.1 Motivation

Die Softwarevisualisierung beschäftigt sich mit Methoden der graphischen Darstellung diverser Informationen über Software, wie statischer Struktur, dynamisches Verhalten oder auch historischer Evolution [Die03]. In der Professur Softwareentwicklung des Instituts für Wirtschaftsinformatik an der Uni Leipzig werden Softwarevisualisierungen erforscht und für einige Darstellungsweisen, sogenannten Metaphern, implementiert. Dabei wurde unter anderem für die Visualisierung der Struktur von Java-Programmen ein Generator entwickelt, der automatisch verschiedene Darstellungsweisen erzeugen kann, wie beispielsweise die "Recursive Disk"- oder City-Metapher (vgl. Abbildung 1.1) [Mü15].

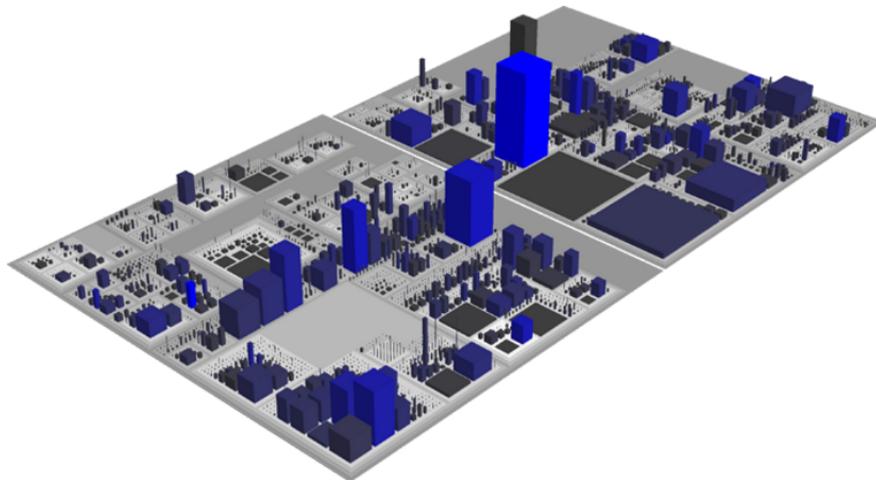


Abbildung 1.1: CodeCity-Visualisierung des JDK v1.5 ¹

Diese Metaphern stellen die strukturelle Information der Software in einer leicht verständlichen Form durch sich rekursiv aufbauende Kreise beziehungsweise Gebäudeähnlicher Bestandteile, visuell dar. Diese Visualisierungen können anschließend für

¹ <https://wettel.github.io/codcity.html> 31.07.217

1 Einleitung

Analyse, Präsentation und Qualitätsüberwachung der Software genutzt werden. Die Darstellung erfolgt mit Hilfe von X3D, einer auf XML basierenden Beschreibungssprache für 3D-Objekte im Browser. Für die Anzeige und interaktive Manipulation dieser 3D-Objekte wird X3DOM² verwendet, welches dies im Browser ermöglicht. Bisher existiert ein solcher Generator nur für Software, die mit der Programmiersprache Java geschrieben wurde. Dabei beschränkt sich dieser in der Darstellung auf Klassen, deren Methoden und Beziehungen. Um eine umfassendere praktische Anwendbarkeit der Softwarevisualisierung zu erreichen, müssen jedoch noch weitere Programmiersprachen unterstützt werden. Diese besitzen unter Umständen besondere Sprachelemente oder Datenstrukturen, die in den momentanen Lösungen zur Visualisierung noch nicht enthalten sind. Demzufolge müssen diese für jede Programmiersprache analysiert werden, damit auch der Generator entsprechend erweitert werden kann.

Bei der Wahl einer solchen zu unterstützenden Programmiersprache ist die Verbreitung in der Praxis ein signifikantes Auswahlkriterium: ABAP ist eine proprietäre Programmiersprache der Firma SAP für die Entwicklung von Anwendungen in SAP-Systemen und bietet sich durch die weltweite Nutzung von SAP-Produkten deshalb für den nächsten Schritt einer Analyse und Erweiterung an. Aus diesem Grund liegt der Fokus der hier vorliegenden Arbeit auf dieser Programmiersprache. Ein solches Vorhaben kann darüber hinaus den Grundstein für die Vorgehensweise bei der Integration neuer Sprachelemente in bestehende Visualisierungstechniken legen.

1.2 Problem- und Zielstellung

Das Ziel der Arbeit ist es, den bestehenden Generator für die Visualisierung von Java-Software um die Möglichkeit der Visualisierung von ABAP-Software als City-Metapher zu erweitern. Dafür wird die Programmiersprache ABAP auf ihre für die Visualisierung relevanten Sprachelemente analysiert und es wird geprüft, wie diese als Bestandteile der Stadt dargestellt werden können. ABAP bietet eine Vielzahl an Sprachelementen und Organisationselementen an, die spezifisch für diese Programmiersprache sind, wie beispielsweise Programme, Reports genannt, oder global definierte Datentypen- und Strukturen, das ABAP Dictionary. Jedoch sind nicht alle dieser Elemente für die Darstellung der Struktur der Software relevant. Eine Problematik besteht jedoch insbesondere in der Findung einer Darstellungsweise, die sich in die City-Metapher einfügen lässt.

² <https://www.x3dom.org/> 23.08.2017

Für die Analyse ergeben sich somit folgende Leitfragen:

1. Welche ABAP-Sprachelemente können in der City-Metapher dargestellt werden?
2. Wo und in welcher Art und Weise werden diese dargestellt?
3. Wie kann die City-Metapher erweitert werden, um die Visualisierung von ABAP-Software zu ermöglichen?

Dies involviert die Erstellung einer Literaturübersicht über den aktuellen Forschungsstand bezüglich der Visualisierung von ABAP-Software. Abschließend erfolgt die Entwicklung eines Prototypen auf Grundlage des bestehenden Generators. Da sich die hier vorliegende Arbeit in der Ausarbeitung auf die City-Metapher fokussiert werden andere Möglichkeiten der Visualisierung nicht im Detail betrachtet.

Die Zielstellung wird als erfüllt angesehen, sobald ein funktionstüchtiger Prototyp eines Generators existiert, der die vorher analysierten Anforderungen für die Visualisierung von ABAP-Software erfüllt.

1.3 Struktur der Arbeit

Das folgende Kapitel 2 befasst sich mit der generativen Softwarevisualisierung und betrachtet dabei zunächst den Begriff der Softwarevisualisierung im Allgemeinen. Anschließend folgt eine kurze Übersicht der verwendeten Hilfsmittel in der generativen Softwarevisualisierung bezüglich der Programmierung und der Darstellung von 3D-Grafiken im Web. In Kapitel 3 werden die Besonderheiten der Programmiersprache ABAP diskutiert und geklärt, worin sich ABAP hauptsächlich von den objektorientierten Programmiersprachen, für die der aktuelle Generator ausgelegt ist, unterscheidet. Außerdem wird betrachtet, welche Sprachelemente für die Visualisierung in Frage kommen und wie diese visualisiert werden können. Kapitel 4 erläutert zunächst die Funktionsweise des ursprünglichen Generators und beschreibt im Anschluss die Änderungen und Neuerungen, die vorgenommen wurden, um ABAP visualisieren zu können. Dabei wird auf jeden Teilschritt der Visualisierung gesondert eingegangen. Abschließend wird der entstandene Prototyp kritisch reflektiert und auf mögliche Schwachstellen oder fehlende Funktionalitäten überprüft. Den Abschluss bildet mit Kapitel 5 ein Ausblick über mögliche zukünftige Erweiterungen des entwickelten Prototypen.

2 Generative Softwarevisualisierung

2.1 Softwarevisualisierung

Um den Sinn der im Voraus definierten Zielstellung zu erläutern und den richtigen Denkansatz für eine geeignete Lösung zu finden, muss genauer auf das allgemeine Ziel der Visualisierung und die dafür notwendigen Schritte eingegangen werden. Laut Schumann et al. [SM00] ist das Ziel „[...] mit Hilfe der Visualisierung die Dinge so darzustellen, wie sie tatsächlich vorliegen und sich wirklich ereignet haben, und den Anwender in die Fähigkeit zu versetzen, nicht nur zu sehen, sondern auch zu erkennen, zu verstehen und zu bewerten.“ Die Visualisierung ist demnach ein Hilfsmittel für die explorative Analyse, die konfirmative Analyse und die Präsentation von Daten [SM00].

Price definiert den Begriff der Softwarevisualisierung als spezifische Unterkategorie der Visualisierung wie folgt: *„Software visualization is the use of the crafts of typography, graphic design, animation and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software.“* [PBS93] Dies erweitert die Definition durch Diehl [Die07] um die Verbesserung des menschlichen Verständnisses und der Nutzbarkeit von Software. Softwarevisualisierung kann dabei in mehreren Dimensionen betrachtet werden, die beispielsweise durch Price et. al [PBS93] und Roman [RC93] allgemein definiert wurden.

Maletic et al. setzen diesen Ansatz mit ihrer Arbeit *„A Task Oriented View of Software Visualization“* [MCM02] fort und richten ihn auf die Anwendung der Softwarevisualisierung für die Unterstützung der Entwicklung und Wartung von groß angelegter Software aus. Sie betrachten die Softwarevisualisierung also insbesondere im Hinblick auf ihre praktische Anwendbarkeit. Die Dimensionen von Maletic et al. sind:

- Aufgaben - Warum wird die Visualisierung benötigt?
- Audienz - Wer wird die Visualisierung nutzen?
- Ziel - Was repräsentiert die Datenquelle?

2 Generative Softwarevisualisierung

- Repräsentation - Wie wird die Datenquelle repräsentiert?
- Medium - Wo wird die Visualisierung repräsentiert?

Diese Dimensionen werden im Laufe der Arbeit implizit verwendet, um eine angemessene Form der Visualisierung für ABAP-Software zu finden.

Die Visualisierung folgt einer sogenannten Visualisierungspipeline, die die einzelnen Schritte des Visualisierungsprozesses vorschreibt (vgl. Abbildung 2.1).

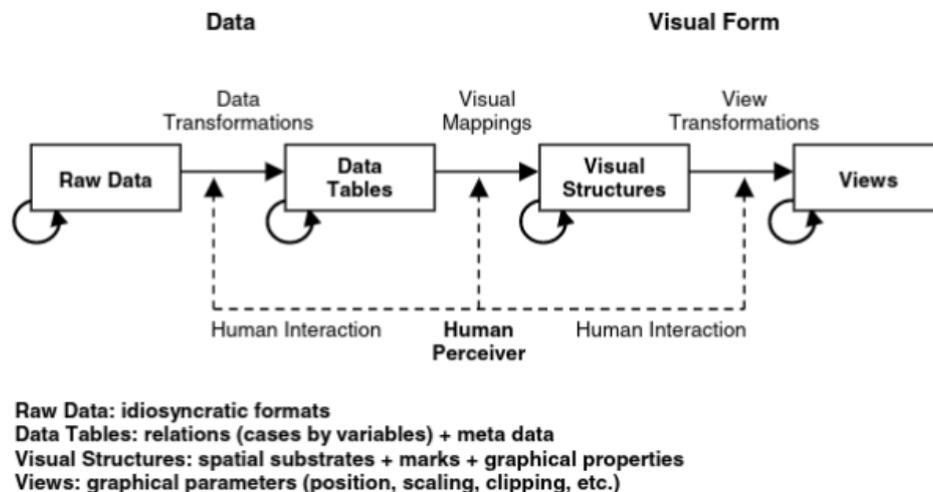


Abbildung 2.1: Visualisierungspipeline nach Card [CMS97]

Diese Visualisierungspipeline wurde zunächst in „Visualisierung: Grundlagen und allgemeine Methoden“ [SM00] beschrieben. Nach Card [CMS97] werden demzufolge zunächst Rohdaten in eine nutzbare Form, oft als domänenspezifische Sprachelemente, übertragen, um anschließend auf ihre entsprechende visuelle Struktur gemappt und gerendert zu werden. Grundlegende Arbeiten über die Softwarevisualisierung sind das Buch von Diehl [Die07] und die Arbeit von Gracanin [EM05]. In der Dissertation von Müller [Mü15] werden die Implementierung und Anwendbarkeit von 3D-Visualisierungen im Kontext von Java-Quelltext betrachtet und unterschiedliche Metaphern angewendet, darunter die Recursive Disk-Metapher. Der in seiner Arbeit entstandene und später weiter entwickelte Generator dient als Grundlage für die Entwicklung eines Prototypen zur Visualisierung von ABAP-Software.

2.2 Methoden der generativen Softwarevisualisierung

Für die Softwarevisualisierung eignet sich die generative Softwareentwicklung nach Eisenecker und Czarnecki [CE00], da sie es, auf der Grundlage von bestimmten Input-Parametern, ermöglicht, einen bestimmten Output, beispielsweise Quelltext, automatisch zu erzeugen. Dafür wird in der Regel ein Generator verwendet, der im Falle der generativen Softwarevisualisierung die im vorherigen Kapitel angesprochene Visualisierungspipeline realisiert.

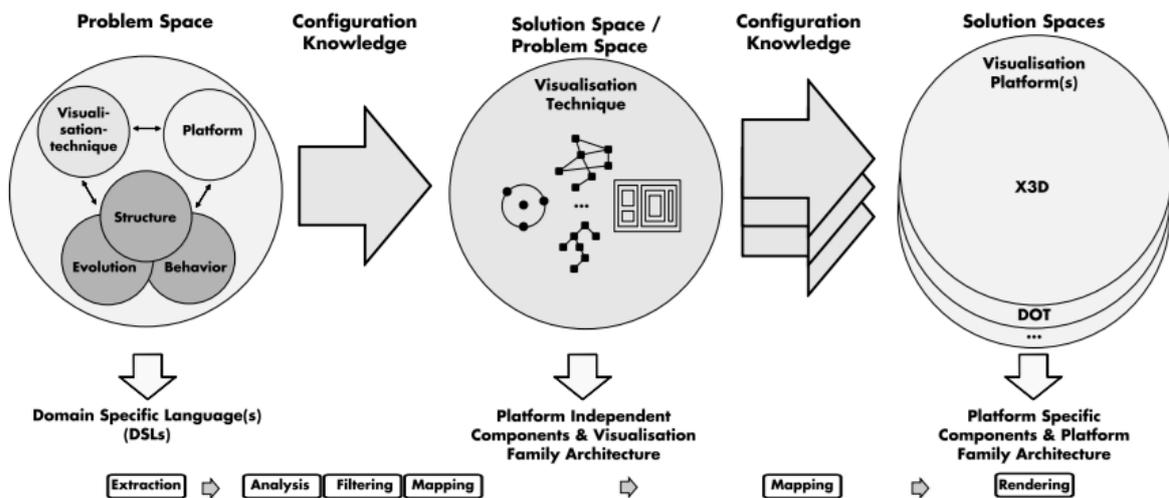


Abbildung 2.2: Model zur generativen Softwarevisualisierung [MKSE11]

Für die generative Softwarevisualisierung haben Eisenecker et. al das in Abbildung 2.2 abgebildete Domänenmodell, welches den groben Ablauf der Erzeugung einer Visualisierung beschreibt, definiert. Dafür werden Daten mit Hilfe von Konfigurationswissen aus einem Problem- in mehrere Lösungsräume überführt. In einem ersten Schritt werden diese Daten auf plattformunabhängige Komponenten gemappt, welche anschließend mithilfe diverser Visualisierungstechniken auf plattformspezifische Komponenten des Lösungsraums, sprich der Visualisierung, übertragen werden.

Die Werkzeuge, die für die Entwicklung eines solchen Generators verwendet werden, müssen alle diese Teilschritte abdecken können. Das involviert insbesondere die automatische Erzeugung von Ausgabewerten unter der Nutzung von Vorlagen. Außerdem ist es nötig, Domain Specific Languages (DSLs) definieren zu können. Erstere Anforderung wird durch XTend¹ erfüllt, einem Dialekt von Java, der in Java

¹ <http://www.eclipse.org/xtend/> 23.08.2017

5-Quelltext kompiliert wird. Diese Programmiersprache besitzt einige Template-Ausdrücke, die die Generierung von Quelltext sehr vereinfachen.

```
1 def someHTML(String content) '''
2     <html>
3         <body>
4             «content»
5         </body>
6     </html>
7 '''
```

Listing 2.1: Template-Ausdruck in Xtend

In Listing 2.1 wird beispielsweise eine Funktion definiert, die einen String-Parameter *content* übergeben bekommt und diesen anschließend in die folgende HTML-Vorlage einfügt. Der so generierte Quelltext wird anschließend zurückgegeben.

Für die Definition der DSL eignet sich Xtend², ein Framework, das speziell für diesen Zweck entwickelt wurde und nach Definition der Sprache eine komplette Infrastruktur mit Parser, Linker, Typechecker und Compiler für sie generiert. Die so definierten Sprachen lassen sich anschließend problemlos mit Xtend nutzen und werden somit verwendet, um die DSLs sowohl für Input-Daten, als auch für Visualisierungs-Daten zu erzeugen.

```
1 grammar org.example.domainmodel.Domainmodel with
2     org.eclipse.xtext.common.Terminals
3
4 generate domainmodel "http://www.example.org/domainmodel/Domainmodel"
5
6 DomainModel :
7     (elements+=Type)*;
8
9 Type:
10     DataType | Entity
11
12 DataType:
13     'datatype' name=ID;
14
15 Entity:
16     'entity' name=ID ('extends' superType=[Entity])? '{'
17         (features+=Feature)*
18     '}';
19
```

² <http://www.eclipse.org/Xtext/> 23.08.2017

```
20 Feature :  
21     (many?='many ')? name=ID ':' type=[Type];
```

Listing 2.2: Definition einer DSL in Xtext

Listing 2.2 zeigt ein Beispiel einer so definierte DSL. Hier wird das Modell „domainmodel“ definiert, das beliebig viele Elemente von Type haben kann, wobei Type entweder DataType-Elemente oder Entity-Elemente sein können. Für Entity und DataType wurden Eigenschaften angelegt; DataType besitzt beispielsweise das Schlüsselwort „datatype“, gefolgt von einem Identifikator „name“.

Xtext wird dafür verwendet, um unter anderem die DSL für die Metapher, aber auch die DSL für die eingehenden Strukturdaten zu erstellen. Diese Strukturdaten sind in einer von Sager [Sag17] erstellten Variante des FAMIX-Formats nach Ducasse et al. [DAB⁺11] formatiert.

2.3 X3DOM

Neben den Werkzeugen zur generativen Softwareentwicklung wird des weiteren ein Mittel zur dreidimensionalen Darstellung von generierten Daten benötigt. Eine Betrachtung der möglichen Werkzeuge der grafischen Darstellung im Web bietet die Arbeit „3D Graphics on the Web : a Survey“ [EAB⁺14]. In „3D Representations for Software Visualization“ [MFM03] werden zudem erste Untersuchungen über die dritte Dimension in der visuellen Darstellung von Software angestellt. Der von Müller [Mü15] entwickelte Generator verwendet hier X3DOM.

X3DOM ist sowohl ein open-source Framework, als auch eine Laufzeitumgebung für 3D-Graphiken im Web, die es ermöglicht eine X3D-Szene im HTML-DOM einzufügen, um somit, durch die Manipulation der DOM-Elemente, diese X3D-Szene zu verändern. Dabei werden im Browser keinerlei Plugins benötigt. Nach Evans et. al [EAB⁺14] wird es besonders in der akademischen Gemeinschaft genutzt. X3D steht dabei für Extensible 3D, eine auf XML basierende Beschreibungssprache für 3D-Modelle zur Darstellung in einem Browser. Dieses Format wurde 2001 vom W3C-Konsortium als offizieller Standard für 3D im Internet verabschiedet. Die Erstellung von 3D-Modellen ist dabei einfach und verbal.

2 Generative Softwarevisualisierung

```
1 <x3d width='500px' height='400px'>
2   <scene>
3     <shape>
4       <appearance>
5         <material diffuseColor='1 0 0'></material>
6       </appearance>
7       <box></box>
8     </shape>
9   </scene>
10 </x3d>
```

Listing 2.3: x3d-Quelltext zur Erzeugung eines roten Würfels

Die x3d-Szene in Listing 2.3 erzeugt beispielsweise das X3D-Element in Abbildung 2.3 im Browser.

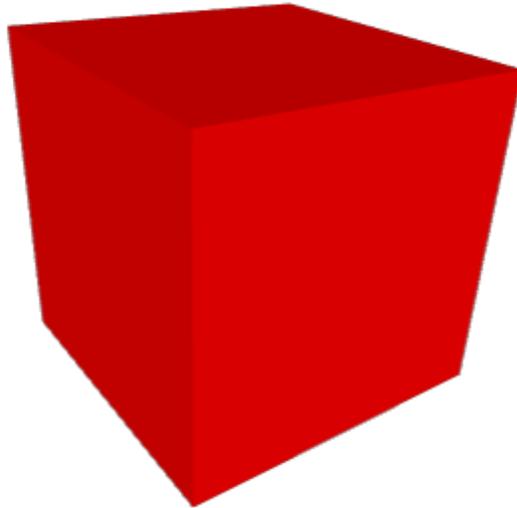


Abbildung 2.3: gerenderte Darstellung eines in x3d definierten Würfels

3 Visualisierung von ABAP

3.1 Besonderheiten von ABAP

ABAP ist eine proprietäre Programmiersprache der Firma SAP, die für die Programmierung von hauptsächlich SAP Application Servern genutzt wird. Sie ermöglicht es den Kunden von SAP deren Systeme um individuelle Anwendungen und Schnittstellen zu erweitern. Das ist oft nötig, da die SAP-Basiskomponenten, die für die grundlegenden Funktionalitäten der von SAP bereitgestellten Systeme verantwortlich sind, oft nicht ausreichen, um alle Anforderungen des Kunden abzudecken.

Anders als bei anderen Programmiersprachen, wie beispielsweise Java, werden ABAP-Programme nicht in externen Dateien, sondern in Datenbanken gespeichert. Dies stellt einen ersten großen Unterschied dar, da Teile des vorhandenen Generators auf dem Vorhandensein externer Dateien basieren und diese für einige Kernfunktionalitäten verwendet werden. Ähnlich zur Java Virtual Machine (VM) werden diese ABAP-Programme zunächst zu Binärcode kompiliert, der anschließend von einer Laufzeitumgebung des SAP-Kernels ausgeführt werden kann. Da viele der Aufgaben, die von ABAP-Programmen übernommen werden, mit dem Aufrufen und Ändern von Datenbanken zu tun haben, gibt es für solche Operationen die datenbank-unabhängige Abfragesprache Open SQL. Alle Open SQL-Statements werden von der Laufzeitumgebung bei der Ausführung in Structured Query Language (SQL) umgewandelt.

Ähnlich wie bei anderen Programmiersprachen sind ABAP-Programme entweder ausführbare Programme oder aber Bibliotheken, mit denen wiederverwendbarer Quelltext für andere Programme zur Verfügung gestellt werden kann. Ausführbare Programme sind entweder Reports oder Module-Pools. Reports nehmen eine Menge von Parametern an und nutzen diese Eingaben, um damit Daten zu verändern oder auszugeben. Module-Pools werden hingegen für die Definition von Mustern zur Nutzerinteraktion genutzt und stellen die Ablauflogik für die verschiedenen Bildschirme, die der Nutzer des Programms sieht, zur Verfügung. Sogenannte Dynamische Programme (Dynpros) werden genutzt, um diese Kombination aus Bildschirm und Ablauflogik zu beschreiben. Das übergeordnete Objekt für diese

3 Visualisierung von ABAP

beiden Sprachelemente ist die Programmbibliothek.

Neben diesen ausführbaren Programmen gibt es eine Reihe von Sprachelementen, die für die Modularisierung von Quelltext verantwortlich sind. Dies sind Includes, Elemente des ABAP Dictionary, Funktionsgruppen mit Funktionsbausteinen, Klassen mit Methoden und Formroutinen.

Includes werden genutzt, um Quelltext einzufügen. Sie werden ebenfalls in der Programmbibliothek abgelegt, die deren übergeordnetes Objekt darstellt.

Das ABAP Dictionary stellt eine große Besonderheit ABAPs dar. Es ermöglicht es, Datenelemente, Strukturen und Tabellentypen, aber auch viele weitere Sprachelemente wie beispielsweise Tabellen, Views für diese Tabellen, Domänen oder Sperrobjekte anzulegen. Dabei werden erstellte Typen durch das ABAP Dictionary auf die zugrundeliegenden Datenbanken gemappt, welche gewissermaßen die Grundpfeiler eines jeden ABAP-Programms darstellen.

Funktionsgruppen sind Container für Funktionsbausteine. Funktionsbausteine ermöglichen es, Funktionen oder Prozeduren global im System zu kapseln und somit überall wiederzuverwenden.

Klassen beschreiben, wie bei anderen objektorientierten Sprachen auch, den Aufbau von Objekten und deren Komponenten. Sie können entweder lokal in Programmen oder global deklariert werden, wodurch sie überall zur Verfügung stehen. Klassen können, mit Einstellungen über deren Sichtbarkeit, Attribute, Methoden, Konstanten und Events beinhalten. Attribute und Konstanten greifen für ihre Deklaration auf Typdefinitionen aus dem ABAP Dictionary zu. Das übergeordnete Objekt der Klassen ist die Klassenbibliothek.

Formroutinen werden in ABAP-Programmen definiert und kapseln Prozeduren für die Wiederverwendung. Sie sind auch in der Programmbibliothek abgelegt.

Neben diesen Sprachelementen gibt es eine Vielzahl weiterer Elemente, die für besondere Funktionalitäten von SAP-Systemen verantwortlich sind. Dazu gehören beispielsweise Web Dynpros, BSP-Applikationen, Business Add Ins und Customer Exits. Programme und Programmbestandteile können Paketen, Namensräumen und Transportaufträgen zugewiesen werden, wodurch sie für die Arbeit an verschiedenen Projekten strukturiert werden können. Pakete entsprechen bei ABAP den Namensräumen aus Java und sind somit eine der obersten Organisationsstrukturen, die für die Verwaltung von Entwicklungen genutzt wird.

Das nächste Kapitel beschäftigt sich mit der Problematik, welche Sprachelemente für die Visualisierung in Betracht gezogen werden sollten.

3.2 Zu visualisierende Sprachelemente

Wie bereits beschrieben verfolgt die Softwarevisualisierung das Ziel, Software für den Menschen verständlicher zu machen, um somit den Softwareentwicklungsprozess zu unterstützen und besser kontrollieren zu können. Die Visualisierung von ABAP-Quelltext wurde bisher jedoch noch nicht konkret untersucht. Ansätze für die dreidimensionale Visualisierung von serviceorientierten Architekturen, wie sie unter anderem mit ABAP entwickelt werden, werden jedoch bereits in „Softwarevisualisierung im Kontext serviceorientierter Architekturen“ [EKS07] betrachtet.

Zur Reduktion der Komplexität hinsichtlich der Zielstellung dieser Arbeit ist es sinnvoll, die primären Zielpersonen für die Nutzung der Visualisierung und deren Anforderungen zu analysieren. Im Falle der erleichterten Wartung und des besseren Verständnisses der Software ist dies insbesondere der ABAP-Entwickler. Die Visualisierung muss es diesem ermöglichen, eine unmittelbare und klare Übersicht über die wichtigsten Programmbestandteile, die visualisiert werden, und deren Zusammenhänge zu erhalten.

Zur Gewährleistung dessen sollte die Struktur, nach der die Sprachelemente visualisiert werden, der Struktur entsprechen, die der ABAP-Entwickler aus seinem Arbeitsumfeld kennt. ABAP-Programme werden entweder mit Hilfe der ABAP Workbench direkt im SAP-System oder mit den ABAP Development Tools für Eclipse entwickelt. Ein Großteil der Programmierarbeit in der ABAP Workbench spielt sich im Object Navigator ab, einer Schnittstelle für alle anderen Entwicklungswerkzeuge, wie etwa dem Class Builder, dem Function Builder oder dem ABAP Editor.

3 Visualisierung von ABAP

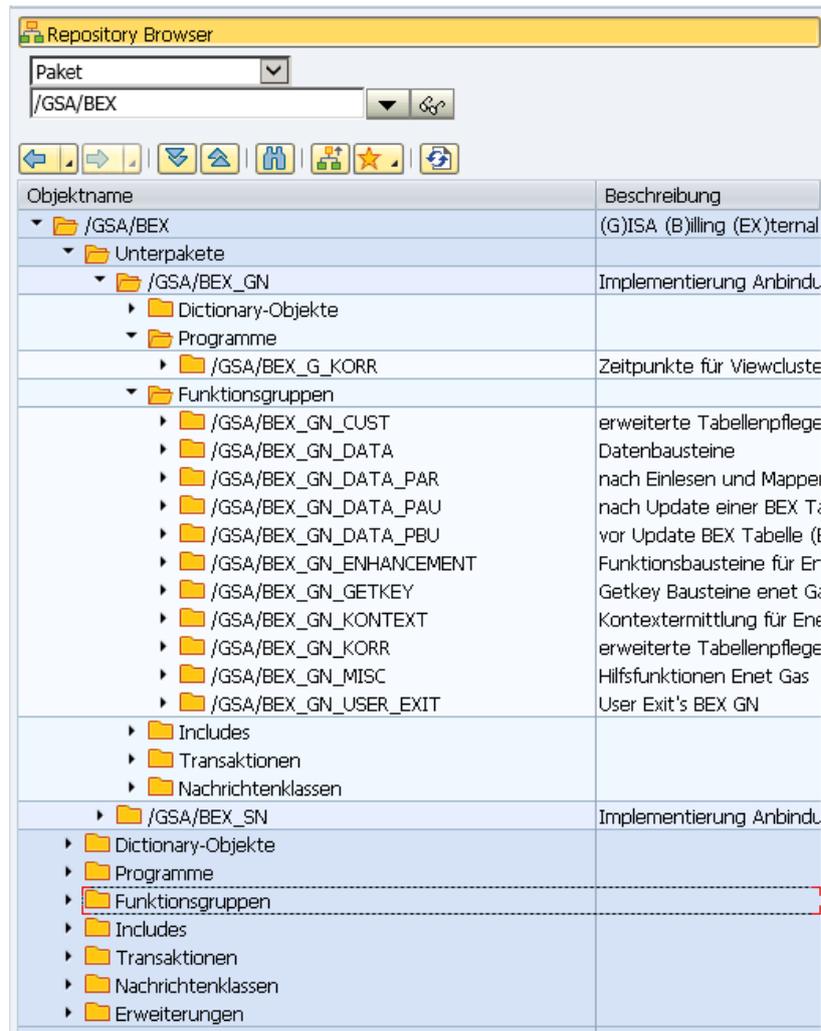


Abbildung 3.1: SAP Object Navigator

Wie in Abbildung 3.1 zu sehen stellt der Object Navigator eine Übersicht über alle Programmbestandteile im momentan ausgewählten Projekt bereit. Diese ist wiederum grob nach den in Abschnitt 3.1 beschriebenen übergeordneten Objekten gegliedert und teilt sich in ABAP Dictionary, im Object Navigator als Dictionary-Objekte bezeichnet, Programmbibliothek, Klassenbibliothek und weitere Bibliotheken auf. Im ABAP Dictionary werden neben den Typen zusätzlich die im Paket verwendeten Datenbanktabellen vermerkt. In der Programmbibliothek befinden sich darüber hinaus die Funktionsgruppen. Daraus leitet sich die Anforderung ab, dass die Visualisierung einer Strukturierung folgen muss, bei der alle Sprachelemente auch in der Visualisierung in ihren entsprechenden übergeordneten Strukturen enthalten sind. Diese Unterteilung in übergeordnete Objekte wird nicht in der Extraktion nach Sager [Sag17] mitgeliefert, sie ergibt sich jedoch implizit aus den jeweiligen Arten der Programmbestandteile. Es bietet sich zudem an, die Visualisierung nach den Paketen, die zu einem Projekt gehören, zu unterteilen.

3 Visualisierung von ABAP

Da es häufig vorkommt, dass aus Programmbestandteilen, die einer Organisationseinheit für eine spezifische Entwicklung angehören, auf Programmbestandteile aus anderen solchen Organisationseinheiten zugegriffen wird, muss dies als Sonderfall zusätzlich betrachtet und visuell dargestellt werden. Dies wird nicht in der Navigationsstruktur des Object Navigator angezeigt, stellt jedoch eine nützliche Zusatzinformation für den Programmierer dar.

Aus den vorangegangenen Überlegungen lassen sich folgende Anforderungen an die Struktur einer Visualisierung ableiten: Programmbestandteile werden im Zusammenhang mit ihren beinhaltenden Programmbestandteilen dargestellt, wobei die Gliederung nach den unterschiedlichen Bibliotheken in Betracht gezogen wird. Für Programmbestandteile, die sich nicht in der projekteigenen Objektmenge befinden, wird dieser Fakt entsprechend visuell gekennzeichnet.

Aufgrund dieser Fokussierung auf die Struktur verbleiben die ABAP-Sprachelemente in Tabelle 3.1 für die Visualisierung. Eine detaillierte Liste über die mitgegebenen Informationen zu den einzelnen Sprachelementen befindet sich im Anhang (vgl. Tabelle A.1)

Sprachelement	Famix-Name
Klassen	FAMIX.Class
Methoden	FAMIX.Method
Attribute	FAMIX.Attribute
Pakete	FAMIX.Devclass
Reports	FAMIX.Report
Formroutinen	FAMIX.Formroutine
Funktionsgruppen	FAMIX.FunctionGroup
Funktionsbaustein	FAMIX.FunctionModule
Datenelemente	FAMIX.DataElement
Strukturen	FAMIX.Struc
Strukturelemente	FAMIX.StrucElement
Datenbanken	FAMIX.Table
Referenzen	FAMIX.Reference

Tabelle 3.1: Verbleibende ABAP-Sprachelemente

Das nächste Kapitel befasst sich mit der Problematik, wie eben diese Strukturen, so-

wie deren Beziehungen untereinander, visuell mit Hilfe der City-Metapher dargestellt werden können.

3.3 Darstellungsmöglichkeiten in der City-Metapher

Eine Metapher ist ein „[...] (besonders als Stilmittel gebrauchter) sprachlicher Ausdruck, bei dem ein Wort (eine Wortgruppe) aus seinem eigentlichen Bedeutungszusammenhang in einen anderen übertragen wird, ohne dass ein direkter Vergleich die Beziehung zwischen Bezeichnendem und Bezeichnetem verdeutlicht.“¹ Der eigentliche Ausdruck wird durch etwas ersetzt, das deutlicher, reicher oder sprachlich anschaulicher sein soll. Ebenso werden im Kontext der Softwarevisualisierung Metaphern genutzt, um gegebene Informationen über eine Software in einer expressiven und deutlichen Art und Weise darzustellen. Dabei wird von einer textlichen zu einer visuellen Darstellung übergegangen. Die Informationen können verschiedenster Art sein, jedoch beschäftigt sich diese Arbeit ausschließlich mit strukturellen Informationen.

Bisher wurden verschiedenste Arten der Metapher für die Visualisierung von Strukturinformationen gestaltet und getestet, darunter die Cody City-Metapher aus der Arbeit „Visualizing Software Systems as Cities“ von Wettel [Wet10]. Die Idee, Software als Stadt darzustellen, wird folgendermaßen begründet: „*Since a city, with its downtown and suburbs, is a familiar concept with a clear notion of orientation, our city representations enable the users to employ their natural wayfinding skills, reducing thus the risk of disorientation often associated with 3D visualization.*“ [Wet10].

¹ <http://www.duden.de/rechtschreibung/Metapher> 23.08.2017

3 Visualisierung von ABAP

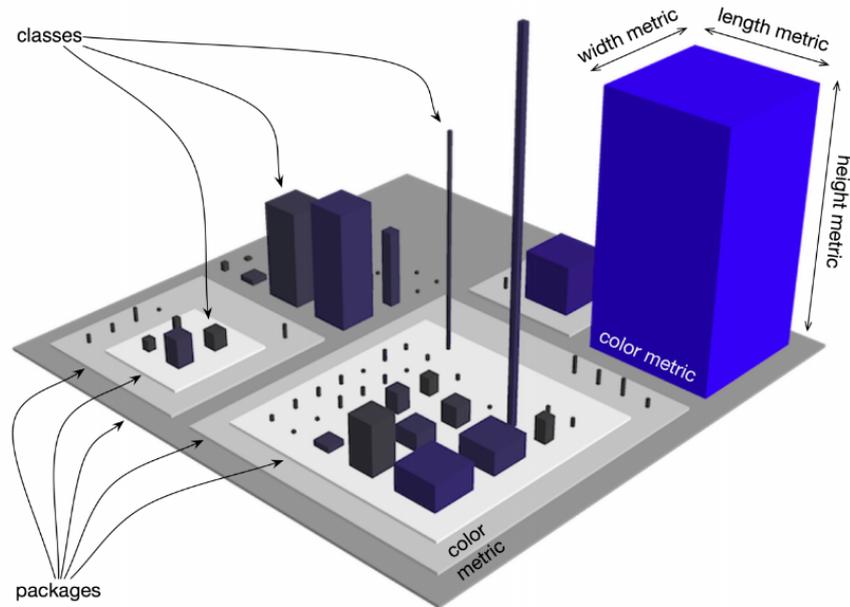


Abbildung 3.2: Überblick der Stadt-Metapher nach Wetzel [Wet10]

Wetzel beschreibt in seiner Arbeit zwei Schritte, die für eine adäquate Definition der Metapher relevant sind : Concept Mapping und Property Mapping [Wet10]: Dabei werden Konzepte oder Bestandteile einer Programmiersprache auf Bestandteile der Metapher, beziehungsweise ihre jeweiligen Eigenschaften, übertragen. Für das Concept Mapping im Fall der Code City werden, wie in Abbildung 3.2 zu sehen, Klassen auf Gebäude, und Pakete auf Distrikte der Stadt übertragen. Wetzel schreibt dazu als Begründung: „*This choice is rooted in the fact that classes are the cornerstone of the object-oriented paradigm and, together with the packages they reside in, the primary orientation point for developers.*“. [WL07] Die Pakethierarchien spiegeln sich dabei in den Distrikten der Stadt wider.

Diese Überlegungen ergeben im Rahmen einer rein objektorientierten Programmiersprache Sinn, jedoch reduziert sich ihr Wahrheitswert, wenn Programmiersprachen, wie beispielsweise ABAP, Sprachelemente und -konzepte enthalten, die den Rahmen der Objektorientierung sprengen. Im Falle von ABAP entsteht dieser Sachverhalt beispielsweise aus der Verwendung von ABAP Dictionary, Reports und Funktionsbausteinen. Der Ansatz, sich an die hauptsächlichen Orientierungspunkte des Programmierers zu halten, wird jedoch beibehalten. Der Programmierer orientiert sich bei ABAP insbesondere an der Navigationsstruktur des Object Navigators, weshalb dessen Aufteilung in Pakete, Bibliotheken innerhalb dieser Pakete und Programmbestandteile innerhalb der Bibliotheken so in die Metapher integriert werden sollte. Indem weiterhin davon ausgegangen wird, dass sich eine Stadt aus Distrikten, die wiederum aus Gebäude bestehen, zusammensetzt, kann die Metapher in dieser

Hinsicht beibehalten werden.

Die Abstraktion eines Gebäudes für Klassen kann für die meisten anderen Programmbestandteile in den Bibliotheken beibehalten werden. Haben Programmbestandteile in Bibliotheken hingegen eigene Programmbestandteile, wie beispielsweise Methoden von Klassen, so werden diese in den jeweiligen Gebäuden abgebildet. Dieser Ansatz wurde bereits von Müller [Mü15] verfolgt.

Für das Property Mapping der Code City entwickelt Wettel drei verschiedene Ansätze: Identity Mapping, Threshold based Mapping und Box plot based Mapping. In jedem Fall werden Metriken von Programmbestandteilen auf Dimensionen von Bestandteilen der Stadt übertragen [Wet10], jedoch mit teilweise unterschiedlichen Gewichtungen. Die Dimensionen der Bestandteile der Stadt sind, wie in Abbildung 3.2 zu sehen, Höhe, Breite, Länge und Farbe für Gebäude sowie Farbe für Distrikte. Da sich die Grundbestandteile der Metapher für ABAP nicht verändern, können diese mit der Ergänzung, dass Farben auch durch Texturen verschiedener Art ersetzt werden können, beibehalten werden.

Wie in Abschnitt 3.1 beschrieben, gibt es einige Sprachelemente, die gesondert behandelt werden müssen. Hierbei handelt es sich insbesondere um Datenbanken, da sie in SAP-Systemen als Grundpfeiler aller Programme und deren Verwendungen dienen. Dementsprechend müssen sie in der Stadt dargestellt werden. Des Weiteren können Pakete, die projektfremde Programmbestandteile beinhalten, nicht mit anderen, projekteigenen, Paketen dargestellt werden. Insofern müssen diese auch gesondert behandelt werden. Darüber hinaus sind die Definitionen von Datenelementen und Strukturen semantisch zu den anderen Sprachelementen verschieden, weil sie, im Gegensatz zu diesen, keinen eigenen Quelltext beinhalten.

Damit ergeben sich für die einzelnen Sprachelemente die Überlegungen in Tabelle 3.2, die bei der Programmierung eines prototypischen Generators für die Visualisierung in Betracht gezogen werden müssen.

Diese Überlegungen werden in den verschiedenen Varianten in den nächsten Kapiteln umgesetzt und anschließend kritisch reflektiert.

Sprachelement	Mögliche Visualisierung
Pakete	Distrikt, welcher Unterdistrikte für jeweilige Bibliotheken beinhaltet (diese werden nicht mitgeliefert und müssen programmatisch erzeugt werden). Bei projektfremden Programmbestandteilen sind klare Abtrennung der Pakete der projekteigenen Programmbestandteile vonnöten (räumliche / farbliche Abgrenzung).
Klassen	Gebäude in Distrikt der Klassenbibliothek, dabei zählt die Methodenanzahl als Metrik für die Höhe und die Attributanzahl als Metrik für Länge und Breite.
Methoden	Gebäudebestandteil, zum Beispiel als Etage.
Attribute	Gebäudebestandteil, zum Beispiel als Schornstein oder Stützpfeiler.
Reports	Gebäude in Distrikt der Programmbibliothek, dabei zählt die Formroutinenanzahl als Metrik für die Höhe
Formroutinen	Gebäudebestandteil, zum Beispiel als Etage.
Funktionsgruppen	Gebäude in Distrikt für Funktionsgruppen, dabei zählt die Funktionsbausteinanzahl als Metrik für die Höhe.
Funktionsbaustein	Gebäudebestandteil, zum Beispiel als Etage.
Nachrichtenklassen	Gebäude in Distrikt für Klassenbibliothek.
Datenelemente	Gebäude in Distrikt für ABAP Dictionary, dabei Abgrenzung zu anderen Gebäuden durch die Form.
Strukturen	Gebäude bestehend aus Strukturelementen, dabei zählt die Strukturelementanzahl als Metrik für die Höhe.
Strukturelement	Gebäudebestandteil, gleiche Form wie Datenelemente.
Datenbanktabellen	Gebäude in Distrikt als „stützender“ Bestandteil des beinhaltenden Paketdistrikts, oder kreisförmige Platzierung um den Stadtdistrikt herum. Dabei kann die Anzahl der Spalten oder die Größenkategorie als Metrik für die Höhe oder Breite verwendet werden.
Referenzen	Verbindung zwischen Gebäuden durch Linien

Tabelle 3.2: Visualisierungsmöglichkeiten der ABAP-Sprachelemente

4 Entwicklung eines Prototypen zur Visualisierung

4.1 Ausgangspunkt

Der bestehende Prototyp des Generators folgt der Visualisierungspipeline wie sie in Kapitel 2 beschrieben wurde und in Abbildung 2.1 zu sehen ist.

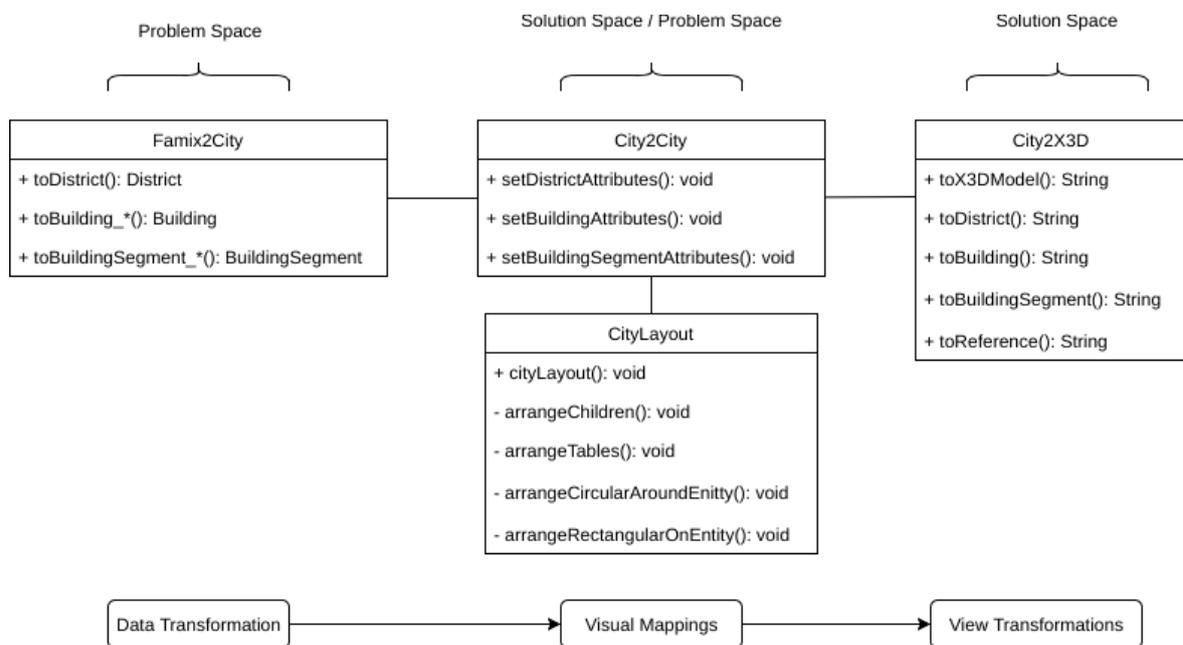


Abbildung 4.1: Übersicht über die Bestandteile des erweiterten Generators

Abbildung 4.1 zeigt eine Übersicht über die Architektur des Generators. Die drei Abschnitte „Data Transformation“, „Visual Mappings“ und „View Transformations“ der Pipeline sind grob in die Klassen „Famix2City“, „City2City“ und „City2X3D“ aufgeteilt. Die Logik für das Layout der Stadt ist in einer Klasse „CityLayout“ gekapselt. Wie zu sehen ist, folgt der Generator damit dem Domänenmodell zur generativen Softwarevisualisierung aus Abbildung 2.2.

Die Strukturinformationen, welche im ersten Schritt analysiert und den entsprechenden Bestandteilen der Stadt zugewiesen werden, entstammen einer als Input eingegebenen FAMIX-Datei. Anschließend werden diese Bestandteile mit Informationen bezüglich ihrer Metriken angereichert und ihre Position in der Stadt durch einen Layout-Algorithmus, der später beschrieben wird, bestimmt. Der letzte Schritt bildet das Stadtmodell auf eine valide X3D-Datei ab, die im Browser mittels X3DOM angezeigt werden kann. Jeder dieser Schritte bedarf für die Visualisierung von ABAP-Software einiger Änderungen.

Für die Positionierung der Stadtbestandteile benutzt Müller in seinem Generator einen abgewandelten Algorithmus zur Positionierung von Lightmaps¹, der auf einem hierarchischen Layout-Algorithmus nach Shneiderman [Shn92] basiert und durch Wettel weiterentwickelt wurde [Wet10]. Dieser berechnet die optimale Verteilung von Elementen verschiedener Größen auf eine rechteckige Fläche, sodass diese einem Quadrat ähnelt. Für die Fälle, in denen eine solche Anordnung erwünscht ist, kann der Algorithmus in seiner bestehenden Form weiterverwendet werden.

Müller definiert für seinen Prototypen des Generators, mit Hilfe von Xtext, eine DSL für die Bestandteile seiner Stadt. Jedoch ist diese für die Visualisierung von ABAP-Software nicht ausreichend, da die Informationen über Texturen, Zugehörigkeit zum Projekt und Referenzen des Bestandteils nicht übergeben werden können. Somit musste die DSL um diese Eigenschaften erweitert werden.

Im Folgenden werden die Erweiterungen des Generators erläutert und die getroffenen Änderungen an den einzelnen Schritten vom Visualisierungsprozess des Generators nachvollzogen.

4.2 Erweiterung des Generators

Es folgt eine Auswertung des entstandenen Prototypen anhand verschiedener Visualisierungen einiger Pakete des Extraktors von Sager [Sag17], sowie eines selbst zusammengestellten Fallbeispiels. Dabei werden die verschiedenen möglichen Einstellungen kurz beschrieben.

1 <http://blackpawn.com/texts/lightmaps/> 23.08.2017

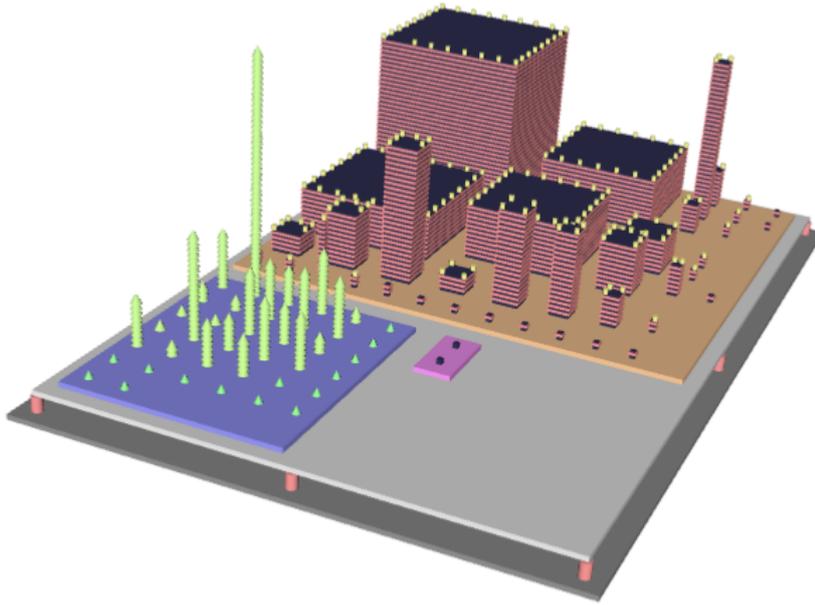


Abbildung 4.2: Darstellung eines Paketes mittels der erweiterten „City“-Metapher

Die Abbildung 4.2 zeigt die empfohlene Darstellung eines ABAP-Pakets mittels der erweiterten City-Metapher. Hier abgebildet ist ein Paket des Extraktors nach Sager, wie der ABAP-Entwickler es auch im Object Navigator vor sich hätte, ohne andere Elemente die gegebenenfalls von Elementen aus diesem Paket heraus aufgerufen werden. Dieses Paket wird im folgenden als projekteigenes Paket bezeichnet, die aufgerufenen Elemente und deren enthaltenden Pakete als projektfremd.

Hier ist bereits deutlich die farbliche Unterscheidung der verschiedenen Bibliotheken im Paket zu erkennen. Die Klassenbibliothek ist hier orange gefärbt, das ABAP Dictionary blau und die Programmbibliothek lila. Das Paket enthält in diesem Beispiel keine Funktionsgruppen, weshalb der Distrikt für diese auch nicht dargestellt wird, wobei dies durch den Nutzer des Generators frei entscheidbar ist. Die Klassen sind wie klassische Gebäude, mit ihren Methoden als rote Etagen, gestaltet, ähnlich wie die Reports, die hier jedoch nur einen geringen Umfang haben und deshalb nur kleine Kästen sind. Der semantische Unterschied zwischen diesen Elementen und den Elementen des ABAP Dictionary wird durch die Form verdeutlicht. Strukturen und Datenelemente sind als Kegel geformt, die bei Strukturen übereinander stehen und somit deren beinhaltete Datenelemente darstellen. Der entstehende Distrikt ähnelt dadurch einem Park mit Bäumen und Büschen und gliedert sich somit gut in die Metapher ein. Die Farben sind, da der Nutzer des Generators sie selbst einstellen kann, willkürlich ausgewählt und haben keine tiefere Bedeutung.

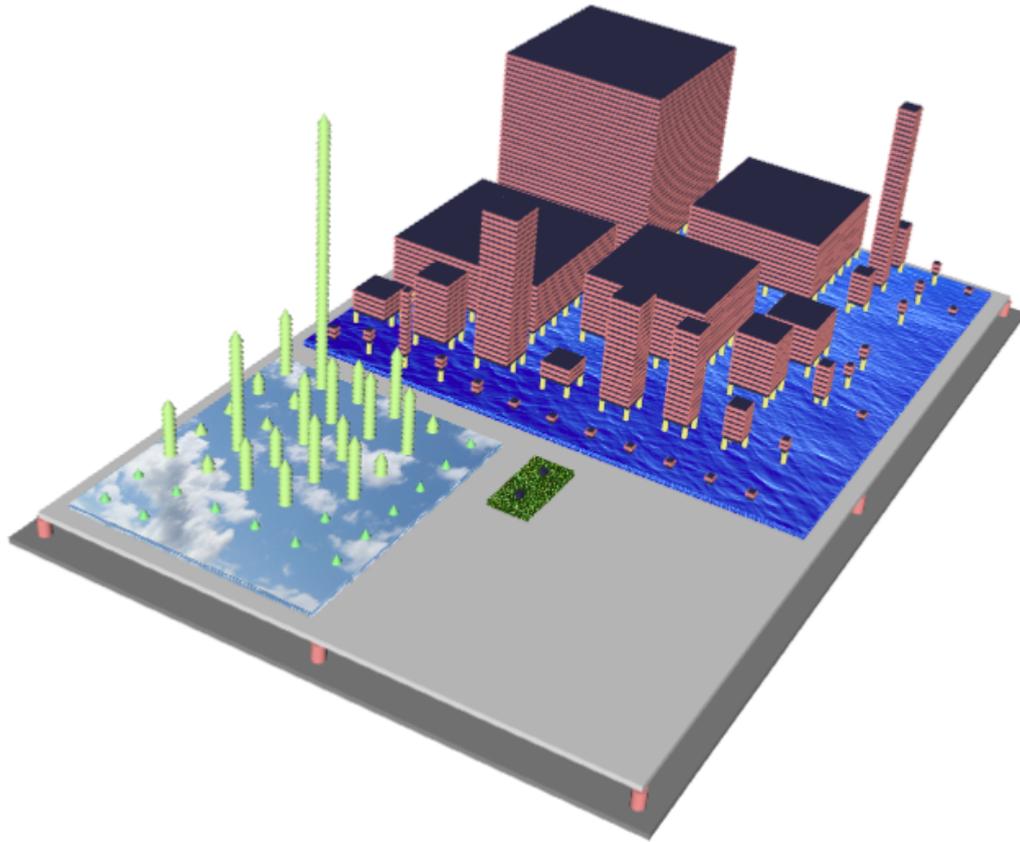


Abbildung 4.3: Abgrenzung durch Texturen

Neben der farblichen Abgrenzung der einzelnen Bibliothek-Distrikte ist, wie in Abbildung 4.3 zu sehen ist, auch eine Abgrenzung durch Texturen möglich. Dies ermöglicht es dem Nutzer für alle Distrikte individuelle Texturen einzustellen, die unter Umständen die Inhalte der Distrikte besser verdeutlichen können. Denkbar ist hier beispielsweise eine Baustein-Textur für den Distrikt der Funktionsgruppen, welcher Funktionsbausteine beinhalten. Eine schlechte Wahl der Texturen führt jedoch möglicherweise zu Problemen bei der Wahrnehmung, wie hier bei den kaum vom Untergrund zu unterscheidenden Reports im, durch eine Gras-Textur abgebildeten, Distrikt der Programmbibliothek.

Für die Darstellung der anderen Bestandteile der Stadt, wie die Datenbanken oder die Attribute, gibt es verschiedene Möglichkeiten, die im folgenden diskutiert werden. Abbildung 4.2 zeigt jedoch die empfohlenen Einstellungen für diese.

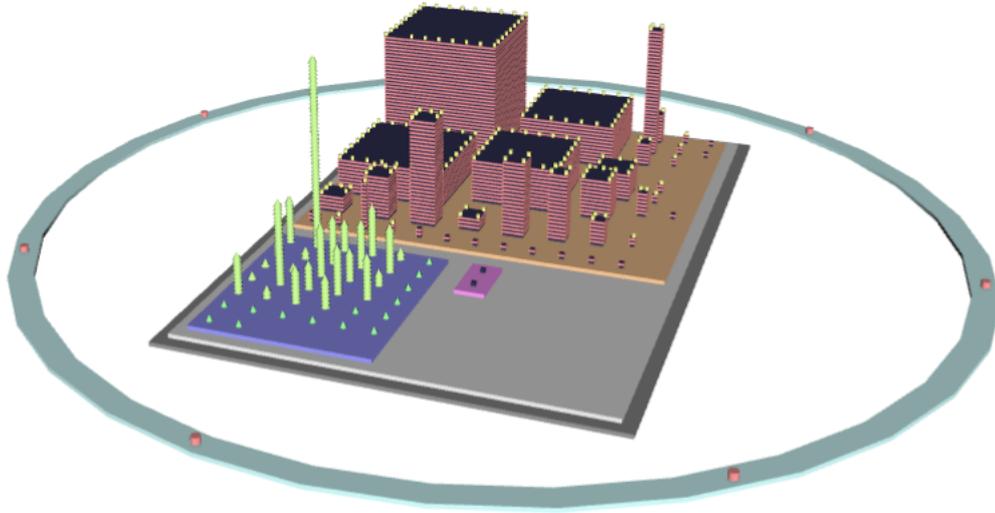


Abbildung 4.4: Kreisförmige Positionierung von Datenbanken

Für die Positionierung der Datenbanken gibt es zwei Möglichkeiten, entweder unter der Stadt als „Stützpfeiler“, wie in Abbildung 4.2 oder kreisförmig um die Stadt positioniert, wie in Abbildung 4.4 abgebildet. Diese zweite Darstellung bietet den Vorteil, dass Referenzen auf Datenbanken leichter verdeutlicht werden können und dass sich die Größe der Datenbanken je nach Spaltenzahl dynamisch verändert. Dagegen stellt die erste Variante den Umstand, dass Datenbanken tatsächlich die Grundlage einer jeden ABAP-Software sind, besser dar und sollte deshalb als Standard betrachtet werden.

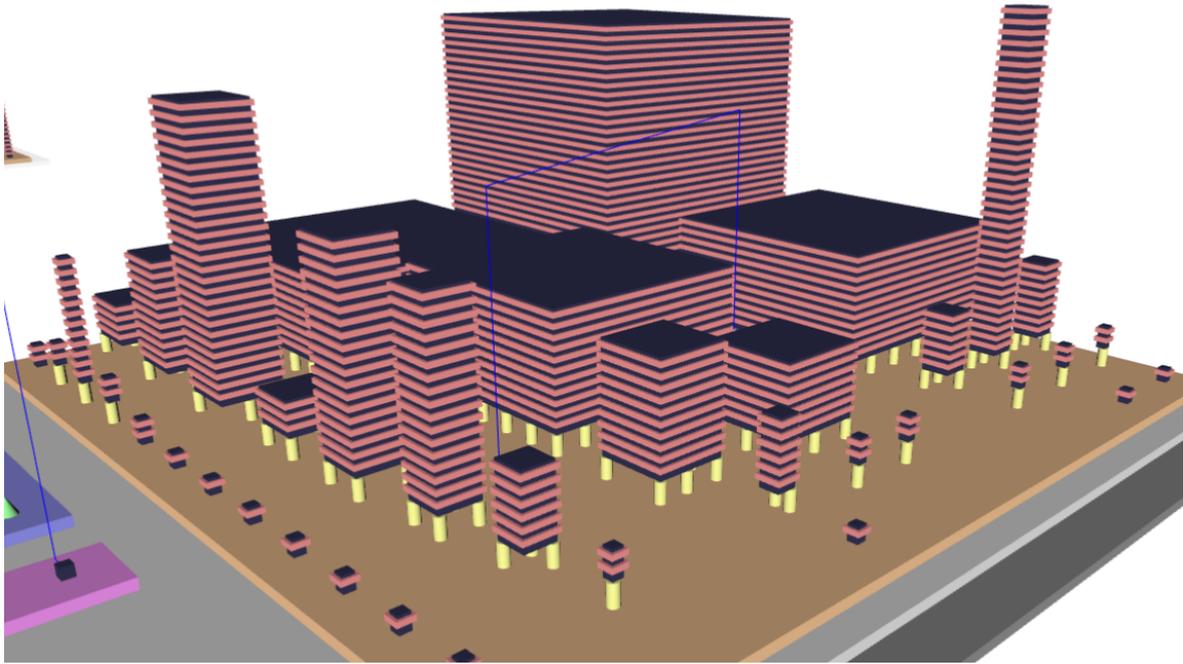


Abbildung 4.5: Positionierung von Attributen unter den zugehörigen Klassen

Abbildung 4.5 zeigt die zweite Variante der Darstellung von Attributen, hier unter den zugehörigen Gebäuden der Klassen positioniert, im Gegensatz zu der alternativen Darstellung auf den Gebäuden (vgl. Abbildung 4.2). Dies folgt dem Gedanken, dass Attribute, ähnlich wie Datenbanken bei ABAP-Software, die Grundpfeiler einer Klasse darstellen. Leider schränkt diese Einstellung die praktische Nutzbarkeit insofern ein, dass die mögliche Interaktion mit den Attributen durch die eng beieinander stehenden Gebäude erheblich erschwert werden würde.

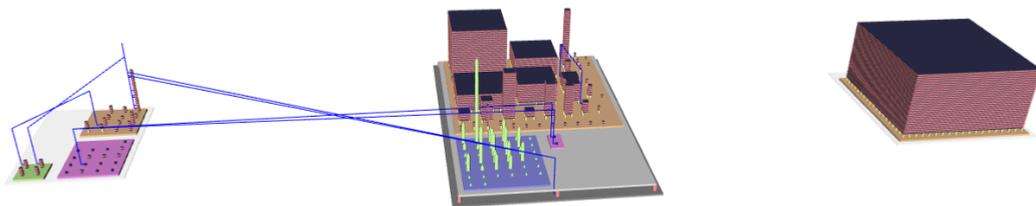


Abbildung 4.6: Verbindung mit projektfremden Paketen

Im Fall, dass es projektfremde Pakete gibt, werden diese kreisförmig um das projekteigene Paket positioniert, das mittig auf dunkelgrauem Grund dargestellt wird

4 Entwicklung eines Prototypen zur Visualisierung

(vgl. Abbildung 4.6). Die Unterscheidbarkeit zum projekteigenen Paket wird durch die Transparenz der zugrunde liegenden Distrikte gewährleistet. Hier sind auch die Verbindungen zwischen aufrufenden und aufgerufenen Elementen durch blaue Verbindungslinien zu sehen.

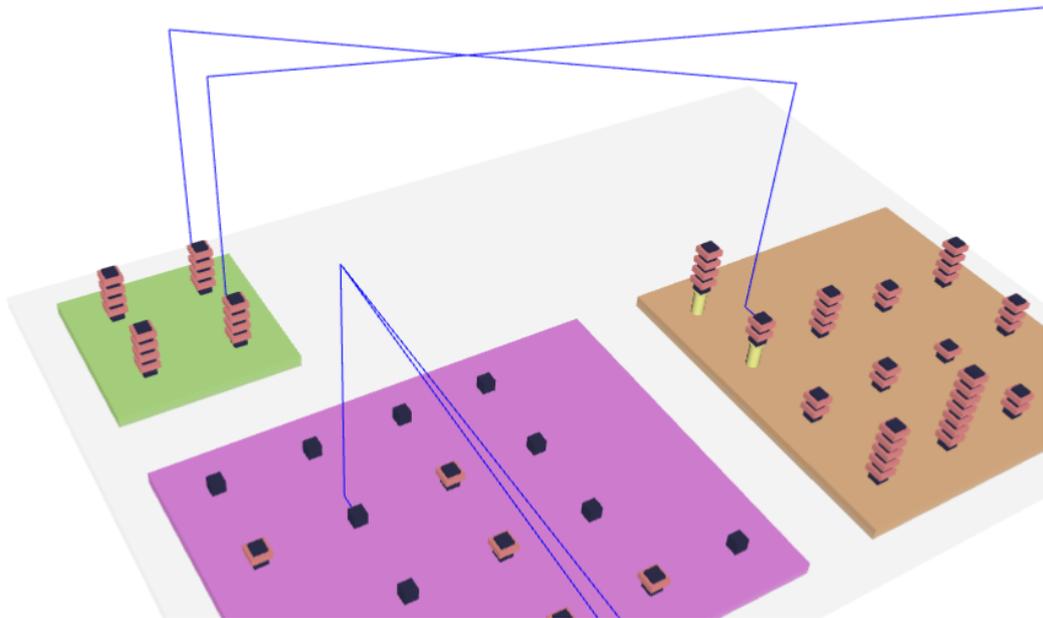
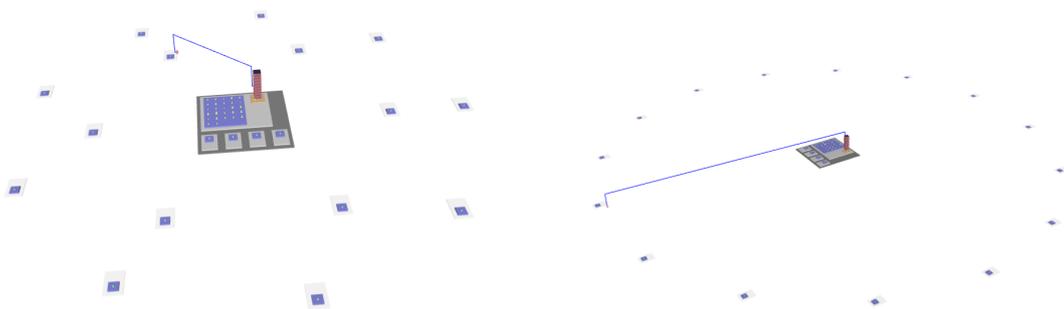


Abbildung 4.7: Darstellung von Referenzen

Diese Verbindungslinien sind, wie in Abbildung 4.7 zu sehen, mehrteilig und führen über die Stadt hinweg zu den jeweilig referenzierten Elementen. Dies dient der Erhöhung der Übersichtlichkeit und der Vermeidung möglicher Kollisionsprobleme.



(a) mehrere Kreise (6 mehr pro Kreis)

(b) ein dynamischer Kreis

Abbildung 4.8: Vergleich der Einstellungen zur Darstellung projektfremder Pakete

Für die Positionierung der projektfremden Pakete existieren außerdem mehrere

4 Entwicklung eines Prototypen zur Visualisierung

Varianten, die in Abbildung 4.8 abgebildet sind. Es ist also somit möglich, diese entweder in einem dynamischen Kreis zu platzieren, der sich bei Erhöhung der Anzahl der Pakete vergrößert (vgl. Abbildung 4.8b), oder aber mehrere Kreise zu verwenden, bei denen sich die Pakete pro Kreis jeweils um eine frei wählbare Anzahl vergrößern (vgl. Abbildung 4.8a). Bei sehr vielen projektfremden Paketen ist die zweite Einstellungsmöglichkeit wahrscheinlich besser geeignet, da sich sonst der eine Kreis extrem ausweiten würde.

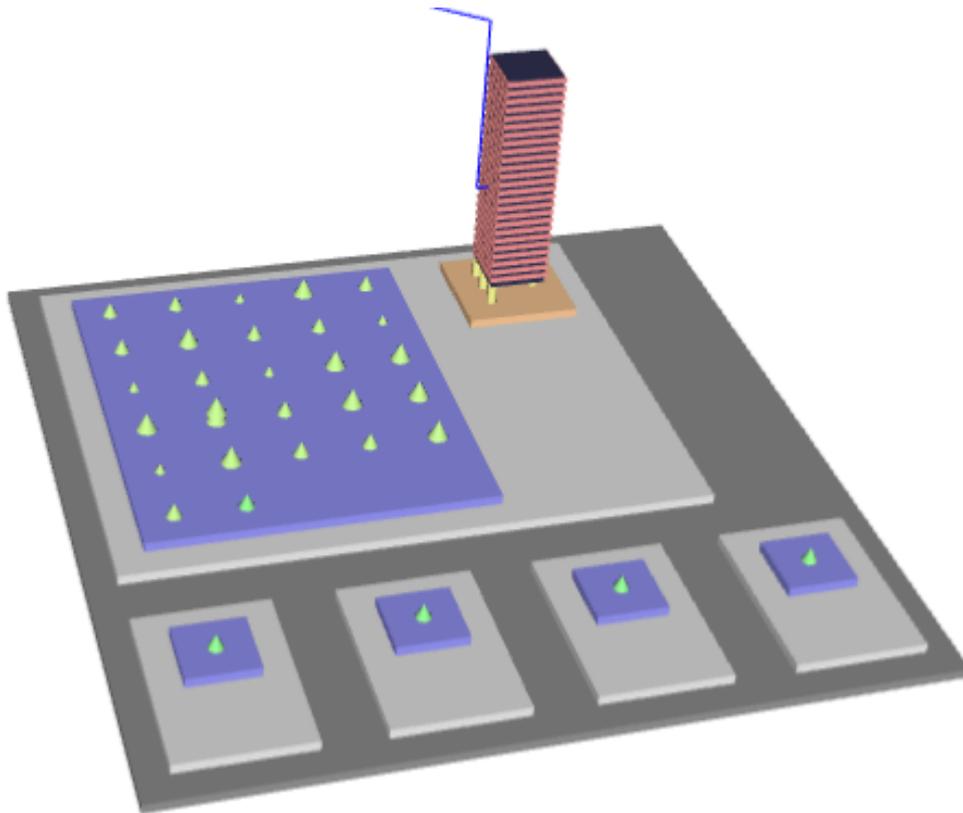


Abbildung 4.9: mehrere Pakete in projekteigenen Elementen

Das erzeugte FAMIX-Modell nach Sager [Sag17] bietet außerdem die Möglichkeit, dass mehrere Pakete als projekteigen für die Extraktion angegeben werden können. Die Visualisierung deckt, wie in Abbildung 4.9 zu sehen, auch diesen Fall ab. Sie werden, ähnlich wie die anderen Elemente der Stadt, mit dem Lightmap-Algorithmus nach Wettel [Wet10] positioniert.

Alle Einstellungen über die eben vorgestellten Varianten werden durch Konstanten getätigt, die in der Enumeration „CitySettings“, bzw. „X3DSettings“ zu finden sind.

Die folgenden Kapitel gehen genauer auf die einzelnen Einstellungsmöglichkeiten ein, eine vollständige Liste über diese befindet sich im Anhang (vgl. Tabelle A.3).

4.3 FAMIX to City

Im ersten Schritt wird die als Input angegebene FAMIX-Datei ausgelesen und für deren Elemente die äquivalenten Stadtbestandteile erzeugt. Dafür ist die Klasse „Famix2City“ verantwortlich: Zunächst werden alle möglichen FAMIX-Elemente aus der FAMIX-Datei gefiltert und in eigene Listen abgelegt, um anschließend Methoden zum Erstellen neuer entsprechender City-Elemente auf diese anwenden zu können (vgl. Tabelle 4.1).

FAMIX-Element	City-Element
FAMIX.Devclass →	District
FAMIX.Class →	Building
FAMIX.Report →	Building
FAMIX.FunctionGroup →	Building
FAMIX.MessageClass →	Building
FAMIX.DataElement →	Building
FAMIX.Struc →	Building
FAMIX.Table →	Building
FAMIX.Method →	BuildingSegment
FAMIX.Formroutine →	BuildingSegment
FAMIX.FunctionModule →	BuildingSegment
FAMIX.StrucElement →	BuildingSegment
FAMIX.Reference →	Attribut von City-Element

Tabelle 4.1: Mapping von FAMIX-Elementen auf City-Elemente

Dies erfolgt durch die Funktion „toDistrict“, die auf alle verfügbaren FAMIX-Pakete angewandt wird. Dadurch wird für jedes Paket ein neuer City-Distrikt erstellt. Aufgrund der weiter oben besprochenen Anforderungen müssen an dieser Stelle außerdem für jeden Distrikt, der aus einem Paket folgt, zusätzliche Unterdistrikte für die Bibliotheken in dem Paket angelegt werden. Hier bietet die Konstante

SHOW_EMPTY_DISTRICTS aus „CitySettings“ die Einstellungsmöglichkeit, ob leere Bibliotheken ebenfalls als Distrikt erscheinen sollen oder nicht. Des Weiterem wird jedes Paket, das projekteigen ist, was durch Sager [Sag17] durch das Fehlen des FAMIX-Elementattributs „iteration“ ausgezeichnet ist, einem übergeordneten Distrikt namens „originDistrict“ zugewiesen, um eine spätere Abgrenzung in der Visualisierung zu ermöglichen.

In der Methode „toDistrict“ werden nach Anlegen der jeweiligen Unterdistrikte für die Bibliotheken, deren entsprechenden FAMIX-Elemente als City-Gebäude zugewiesen. In Abhängigkeit von dem momentan hinzuzufügenden Element werden dafür verschiedene „toBuilding_*“-Methoden verwendet. Diese Aufteilung nach verschiedenen Methoden ist nötig, weil die Attribute der entsprechenden FAMIX-Elemente teilweise verschieden sind.

Ähnlich wird mit den Bestandteilen der City-Gebäude verfahren, wofür es weiterhin diverse „toBuildingSegment_*“-Methoden gibt. Abschließend werden alle FAMIX-Referenzen den entsprechenden City-Elementen zugeordnet. Das entsprechende FAMIX-Element „FAMIXReference“ besitzt neben der eigenen ID nur die source-, beziehungsweise target-ID des jeweiligen FAMIX-Elements. Diese werden genutzt, um mit der Funktion „toABAPReference“ dem des source-Elements entsprechenden City-Gebäudes eine neue ABAP-Referenz zuzuweisen, die auf das target-Gebäude zeigt.

Das infolgedessen entstandene City-Dokument wird als Workflow-Kontext gesetzt, um im nächsten Schritt weiterverarbeitet werden zu können.

4.4 City to City

Das Layout der Stadt wird durch die Klasse „City2City“ veranlasst. Dieser Vorgang lässt sich in fünf Schritte aufteilen, in denen zunächst die einzelnen Stadtbestandteile mit deren Eigenschaften angereichert werden, um anschließend mithilfe mehrerer Methoden der Klasse „CityLayout“ positioniert werden.

Schritt 1 vor der eigentlichen Platzierung in der Stadt ist die Zuweisung von Werten für die Beschaffenheit der Dimensionen einiger City-Elemente. Dies sind Größe (Höhe, Breite, Länge) und Farbe bzw. Textur. Für Distrikte werden an dieser Stelle nur die jeweilige Farbe bzw. Textur zugewiesen, da ihre Größe später während der eigentlichen Positionierung der beinhalteten Elemente bestimmt wird.

Für alle Gebäude wird in diesem Schritt neben der Farbe außerdem die Größe festgelegt. Dies sind die in „CitySettings“ festgelegten Minimalwerte, außer die jeweiligen Gebäude besitzen ihrerseits Gebäudebestandteile. Bei Methoden, Funktionsbau-

steinen und Formroutinen wird die Höhe der beinhaltenden Gebäude auf deren jeweilige Anzahl gesetzt. Die Anzahl der Attribute bestimmt bei Klassen die Breite und Länge des Gebäudes. Für Datenbanken werden deren Spaltenzahl als Metrik für die Höhe verwendet, wenn sie kreisförmig um die Stadt herum positioniert werden. Bei der Positionierung unter der Stadt ist ihre Höhe über die Konstante `TABLE_HEIGHT` in „CitySettings“ einstellbar.

Als nächstes erfolgt das Layout der City-Elemente mit Hilfe der Klasse „CityLayout“. Als **Schritt 2** arrangiert deren Methode „cityLayout“ zunächst fast alle projekteigenen City-Elemente mit dem LightMap-Algorithmus und passt anschließend die Position dieser Elemente noch an. An diesem Punkt sind nun sämtliche projekteigenen City-Elemente, abgesehen von Datenbanken, ähnlich der Stadt von Wettel [Wet10] positioniert. Außerdem wurden eine Liste mit allen projektfremden Paketen und eine Liste mit allen Gebäuden, die Gebäudebestandteile besitzen, erstellt.

Für die Visualisierung von ABAP-Software müssen infolgedessen in einem dritten Schritt alle projektfremden Pakete von den projekteigenen Paketen getrennt werden. Des Weiteren müssen die Tabellen, damit sie ihrer Funktion als „Grundpfeiler“ der Stadt auch visuell gerecht werden, entsprechend positioniert werden. Als abschließender Schritt werden die einzelnen Gebäudebestandteile, wie beispielsweise Methoden oder Datenelemente von Strukturen, gesetzt.

Für die Positionierung der projektfremden Pakete in **Schritt 3** gibt es zwei Optionen. In beiden Fällen wird zunächst der `originDistrict` herausgesucht und anschließend die Hilfsmethode „`arrangeCircularAroundEntity`“ (vgl. algorithm 1) aufgerufen, die den `originDistrict` sowie die Liste aller projektfremden Pakete als Parameter entgegennimmt. Diese Methode arrangiert die Pakete kreisförmig um den `originDistrict` herum. Über die Konstante `NOTINORIGIN_SINGLE_CIRCLE` in „CitySettings“ kann ausgewählt werden, ob alle Pakete in einem Kreis oder in mehreren Kreisebenen positioniert werden sollen. Die Positionen der Pakete werden mit Hilfe von Polarkoordinaten berechnet: Um die Anzahl der Pakete pro Kreis, bzw. um den nötigen Umfang des Kreises, der allen Paketen genügend Platz für die Darstellung gewährleistet, zu berechnen, wird ein einfacher Algorithmus genutzt. Die gewünschte Anzahl Pakete, die pro Kreis zusätzlich angezeigt werden soll, kann frei über die Konstante `PACKAGES_PER_CIRCLE` in „CitySettings“ gewählt werden.

Algorithm 1: Funktion zur kreisförmigen Positionierung um ein City-Element in einem Kreis

```
1 function arrangeCircularAroundEntity (entities, entity);  
   Input : entities to be arranged around entity  
   Output: void  
2 calculate circle that fits all elements;  
3 calculate angle  $\alpha$  between elements;  
4 for entity  $\in$  entities do  
5 |   set position (x, y) of entity using polar coordinates;  
6 end
```

Die genutzten Polarkoordinaten werden dabei wie folgt berechnet:

$$x = r * \cos(\alpha)$$

$$y = r * \sin(\alpha)$$

Da hierdurch nur die Position der Distrikte, nicht aber die Position der enthaltenen Gebäude verändert wird, müssen diese anschließend noch relativ zu ihren enthaltenen Distrikten angeordnet werden.

In **Schritt 4** werden als nächstes die Datenbanken, je nach Einstellung von TABLES_AROUND_CITY in „CitySettings“, entweder unter dem enthaltenden Paket, oder kreisförmig um das Paket herum positioniert. Im ersten Fall wird die Hilfsmethode „arrangeRectangularOnEntity“ (vgl. algorithm 2) verwendet, die die optimale Anzahl von Elementen pro Seite des Elements, auf denen die Elemente platziert werden sollen, berechnet und diese entsprechend positioniert.

Algorithm 2: Methode zur rechteckigen Positionierung auf ein City-Element

```

1 function arrangeRectangularOnEntity (entities, entity);
   Input : entities to be arranged on entity
   Output: void
2 calculate elements per side;
3 calculate gap between elements;
4 initialize counter variable;
5 for element ∈ side do
6     place element on side with a distance of counter * gap to corresponding
       corner;
7     counter++;
8     if at last element then
9         set counter to 0;
10    end
11 end

```

Im zweiten Fall wird die Hilfsmethode „arrangeCircularAroundEntity“ wiederverwendet.

In **Schritt 5** erfolgt abschließend die Positionierung der Gebäudebestandteile. Dafür wird die zuvor angelegte Liste aller Gebäude mit Gebäudebestandteilen verwendet. Methoden werden dabei schlicht als „Etagen“ des enthaltenden Gebäudes platziert. Attribute können unter dem enthaltenden Gebäude, ähnlich wie die Datenbanken, oder auf dem Gebäude, als eine Art Schornstein, platziert werden. Dies wird wieder durch die Hilfsmethode „arrangeRectangularOnEntity“ realisiert. Die in Strukturen enthaltenden Elemente werden ähnlich wie Methoden übereinander platziert, wobei sie leicht ineinander übergehen.

Die Referenzen müssen hier nicht platziert werden, da ihre Position durch die Position ihrer source-, beziehungsweise target-Elemente bereits bekannt ist.

Die so positionierten City-Elemente können nun mithilfe von x3d visuell dargestellt werden.

4.5 City to X3DOM

Die Klasse „City2X3D“ generiert aus dem City-Dokument im Workflow-Kontext ein gültiges x3d-Modell, das anschließend im Browser angezeigt werden kann. Dafür wird dynamisch für jedes City-Element der entsprechende x3d-Quelltext erzeugt. „X3DSettings“ dient dazu, Einstellungen für die Quelltext-Erzeugung treffen zu

können. Hier wurden die vorhandenen Methoden zur Quelltexterzeugung um die entsprechenden X3D-Modelle der neuen City-Elemente erweitert.

Die Funktion „toX3DModel“ geht schrittweise durch die City-Elemente und führt für jedes Element die Methode zur Erzeugung des entsprechenden x3d-Quelltextes aus. Hier wurden die entsprechenden Methodenaufrufe für neue City-Elemente eingefügt.

„toDistrict“ erzeugt den Quelltext für die Distrikte der Stadt, wobei diese als Boxen niedriger Höhe, auf denen anschließend die Gebäude platziert werden, dargestellt werden. Ist der momentane Distrikt projektfremd, so wird seine Transparenz reduziert, um ein weiteres Unterscheidungsmerkmal zu den projekteigenen Distrikten zu schaffen. Die Transparenz kann dabei über die Konstante NOTINORIGIN_TRANSPARENCY in „CitySettings“ individuell angepasst werden. Für den Fall, dass die Datenbanken kreisförmig um die Stadt platziert werden, wird an dieser Stelle außerdem ein Kreis um die Stadt herum platziert, auf dem die Datenbanken später „stehen“.

Der Quelltext für die Gebäude der Stadt wird durch die Methode „toBuilding“ erzeugt. Klassen, Reports und Funktionsgruppen sind Boxen, also konventionelle Häuser; Datenbanken sind Zylinder. Datenelemente sowie Strukturen werden als Kegel dargestellt, um eine ausreichende Unterscheidung zu gewährleisten. „toBuildingSegment“ erzeugt den Quelltext für die Gebäudebestandteile, dabei werden Methoden als schmale Boxen, die etwas breiter als ihre beinhaltenden Gebäude sind, dargestellt und Attribute als kleine Zylinder.

Abschließend erzeugt „toReference“ den Quelltext für die Referenzen. Dafür wird vom source-Element ausgehend eine horizontale Linie aus dem jeweiligen Element heraus gezeichnet, anschließend eine vertikale Linie um über die anderen City-Elemente zu gelangen, sowie eine weitere horizontale Linie bis zur Position kurz vor dem target-Element. Abschließend verbindet eine vertikale und eine horizontale Linie das target-Element mit der Verbindungslinie vom source-Element. Die Berechnungen für diese Positionierung erfolgt dabei auch in dieser Methode, wobei sich die Länge der Linien aus dem Abstand der zu verbindenden Stadtbestandteile ergibt und die Höhe, je nach Umstand, frei eingestellt werden kann.

4.6 Evaluation

Der eben vorgestellte Generator erfüllt alle vorher definierten Anforderungen an einen Prototypen, hat jedoch einige Einschränkungen, die nicht außen vor gelassen werden dürfen.

4 Entwicklung eines Prototypen zur Visualisierung

Zunächst einmal stellt diese Arbeit den entwickelten Generator nur anhand einiger weniger Fallstudien vor. Dies hat den Grund, dass die Extraktion von Strukturinformationen nach Sager [Sag17] noch nicht vollständig funktionstüchtig ist, wodurch es bei größeren Objektmengen zu Fehlern kommt, kleine bis mittlere Objektmengen jedoch zum Teil fehlerfrei extrahiert werden können. Dadurch ist es sehr aufwendig, die bereitgestellten FAMIX-Dateien so aufzubereiten, dass sie durch den Generator fehlerfrei genutzt werden können. Es ist somit durchaus möglich, dass es bei größeren Paketen zu nicht vorhersehbaren Problemen kommen kann.

Die vielen vorhandenen Konfigurationsmöglichkeiten verdeutlichen außerdem, dass es schwierig ist, eine allgemein gültige Visualisierung für eine Programmiersprache zu finden, die besonders zu behandelnde Sprachelemente besitzt. Viele der Varianten haben ihre Vor- und Nachteile und eine ausführliche Evaluation ist im Rahmen dieser Arbeit nicht möglich gewesen. Aus diesem Grund werden alle Konfigurationsmöglichkeiten uneingeschränkt für den Nutzer freigegeben. Dadurch würde auch vermieden werden, dass beispielsweise einige Textur- oder Farbkombinationen von Distrikten für den einen Nutzer klar deutlich, für den anderen Nutzer aber verwirrend sein können.

Des Weiteren bietet die Art und Weise der Erweiterung des Generators einige Verbesserungsmöglichkeiten. Methoden und Funktionen sind teilweise redundant und liefern manchmal nur ein approximatives Ergebnis, das für die Zwecke dieser Arbeit jedoch ausreicht. Für neue Funktionalitäten wurden alte Codebestandteile oft zweckentfremdet, wodurch der erweiterte Generator nicht mehr in der Lage ist andere Sprachen neben ABAP zu visualisieren. Die Überführung der neuen Funktionalitäten in den alten Generator wird dadurch auch um einiges erschwert. Hier bietet sich eine genauere Analyse im Voraus darüber an, welche Funktionalitäten hinzukommen und wie diese integriert werden können. Allgemein wurde an einigen Stellen die Entwicklung durch eine mangelnde Anforderungsanalyse erschwert, beispielsweise als erst im Laufe der Arbeit klar wurde, dass auch mehrere projekteigene Pakete möglich sind. Hier ist also teilweise ein umfangreiches Refactoring nötig.

5 Fazit und Ausblick

Die Softwarevisualisierung befasst sich mit der visuellen Darstellung von Informationen über Software. Dies wurde vom Institut für Softwareentwicklung der Universität aufgegriffen, wodurch ein prototypischer Generator entstand. Dieser ist in der Lage, die Strukturinformationen von, mit objektorientierten Sprachen entwickelter, Software visuell mit Hilfe mehrerer Metapher darzustellen. Um die praktische Anwendbarkeit dieses Generators zu erweitern, war es das Ziel dieser Arbeit, die Programmiersprache ABAP für die Visualisierung durch die City-Metapher zu analysieren und anschließend die entsprechende Erweiterung des bestehenden Generators zu implementieren.

Die vorangegangenen Kapitel beschreiben anhand visueller Beispiele die Ergebnisse einer, durch den in dieser Arbeit implementierten Generator erzeugten, Visualisierung eines kleineren Pakets aus einer ABAP-Software. An ihnen ist zu erkennen, dass die in Kapitel 3 analysierten Anforderungen an einen möglichen Generator erfüllt wurden. Somit ist das Ziel dieser Arbeit erfüllt und bietet einen Ausgangspunkt für weitere wissenschaftliche Arbeiten.

Der entstandene Prototyp hat zudem eine Vielzahl an Möglichkeiten für die spätere Erweiterung. Beispielsweise beinhalten die von Sager [Sag17] generierten Daten neben den Informationen zu projekteigenen und projektfremden Paketen außerdem Informationen darüber, in welcher Iteration sich diese ermittelten Daten befinden. Das bedeutet, dass nicht nur extrahiert wird, in welchem Paket eine projektfremde Funktion liegt, die aus einem projekteigenen Element aufgerufen wird, sondern auch anschließend, welche weiteren Funktionen oder Elemente durch diese aufgerufen werden. Dadurch wäre es möglich, alle komplexen Zusammenhänge zwischen verschiedenen Paketen auf eine einfache Art und Weise zu visualisieren. Problematisch ist hier, dass bei mehreren Iterationsstufen die Anzahl der zu visualisierende Pakete stark ansteigt und somit die momentane Art der Darstellung für vergleichsweise wenig Pakete wahrscheinlich nicht mehr ausreicht. Eventuell wäre es an dieser Stelle sinnvoll, nur die Aufrufe von einem spezifischen Programmbestandteil darzustellen und nicht alle Aufrufe des gesamten Programms.

Außerdem wäre es denkbar, dass zukünftig durch eine Weiterentwicklung des Extrak-

tors von Sager [Sag17] weitere Metriken und Elemente hinzukommen. Beispielsweise wäre es möglich, aus Datenbankabfragen Informationen darüber auszulesen, welche Daten aus welchen Datenbanken abgefragt werden und diese mit den abfragenden Sprachelementen zu verknüpfen. Dies würde einen weiteren Fokus auf die Datenbanken setzen, die bei der Entwicklung von ABAP-Software eine zentrale Rolle spielen.

Die Interaktivität mit den entstandenen Modellen wird durch die vorliegende Arbeit nicht beachtet und muss deshalb noch vollständig implementiert werden. Hier gibt es Ansätze nach Kovacs [Kov11], die für die erste Anforderungsanalyse und Prototypentwicklung verwendet werden können.

A Anhang

A.1 Tabellen

Sprachelement	Famix-Name	Vorhandene Informationen
Klassen	FAMIX.Class	ID, Name, beinhaltendes Objekt (container), projekteigene Objektmenge (optional)
Methoden	FAMIX.Method	ID, Name, beinhaltendes Objekt (parentType), projekteigene Objektmenge (optional)
Attribute	FAMIX.Attribute	ID, Name, beinhaltendes Objekt (parentType), projekteigene Objektmenge (optional)
Pakete	FAMIX.Devclass	ID, Name, beinhaltendes Objekt (container), projekteigene Objektmenge (optional)
Reports	FAMIX.Report	ID, Name, beinhaltendes Objekt (parentScope), projekteigene Objektmenge (optional)
Formroutinen	FAMIX.Formroutine	ID, Name, beinhaltendes Objekt (parentType), projekteigene Objektmenge (optional)
Funktionsgruppen	FAMIX.FunctionGroup	ID, Name, beinhaltendes Objekt (container), projekteigene Objektmenge (optional)
Funktionsbaustein	FAMIX.FunctionModule	ID, Name, beinhaltendes Objekt (parentType), projekteigene Objektmenge (optional)

A Anhang

Nachrichtenklassen	FAMIX.MessageClass	ID, Name, beinhaltendes Objekt (container), Anzahl Nachrichten, projekteigene Objektmenge (optional)
Datenelemente	FAMIX.DataElement	ID, Name, beinhaltendes Objekt (container), projekteigene Objektmenge (optional)
Strukturen	FAMIX.Struc	ID, Name, beinhaltendes Objekt (container), projekteigene Objektmenge (optional)
Strukturelement	FAMIX.StrucElement	ID, Name, beinhaltendes Objekt (container)
Datenbanken	FAMIX.Table	ID, Name, beinhaltendes Objekt (container), Anzahl Spalten, Größenkategorie, projekteigene Objektmenge (optional)
Referenzen	FAMIX.Reference	ID, referenzierendes Objekt, referenziertes Objekt

Tabelle A.1: Verbleibende ABAP-Sprachelemente (vollständig)

A Anhang

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	

A Anhang

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

A Anhang

Namespace

Tabelle A.2: Alle ABAP-Sprachelemente

A Anhang

Konstante	Mögliche Werte	Erläuterung
SHOW_EMPTY_DISTRICTS	boolean	Anzeige von leeren Bibliotheken als Distrikt
TABLES_AROUND_CITY	boolean	Positionierung von Datenbanken kreisförmig um Stadt oder als Stützpfeiler
TABLE_HEIGHT	float	Höhe von Datenbanken bei Positionierung als Stützpfeiler
ATTRIBUTES_BELOW_BUILDINGS	boolean	Positionierung von Attributen unter oder auf Gebäuden
ATTRIBUTES_BELOW_BUILDINGS_HEIGHT	float	Höhe von Attributen bei Positionierung unter Gebäuden
ATTRIBUTES_ON_BUILDINGS_HEIGHT	float	Höhe von Attributen bei Positionierung auf Gebäuden
NOTINORIGIN_TRANSPARENCY	float	Transparenz der Distrikte von projektfremden Paketen zwischen 0 und 1
NOTINORIGIN_SINGLE_CIRCLE	boolean	Positionierung von projektfremden Paketen in einem Kreis oder mehreren Kreisen
PACKAGES_PER_CIRCLE	int	Anzahl an Paketen, die pro Kreis mehr angezeigt werden
CLASS_DISTRICT_color	RGBColor	Farbe des Distrikts der Klassenbibliothek

A Anhang

REPORT_DISTRICT_color	RGBColor	Farbe des Distrikts der Programmbibliothek
FUNCTIONGROUP_DISTRICT_color	RGBColor	Farbe des Distrikts der Funktionsgruppen
DICTIONARY_DISTRICT_color	RGBColor	Farbe des Distrikts des ABAP Dictionary
TABLE_color	RGBColor	Farbe von Datenbanken
ATTRIBUTE_color	RGBColor	Farbe von Attributen
METHOD_color	RGBColor	Farbe von Methoden
DATAELEMENT_color	RGBColor	Farbe von Datenelementen
STRUC_color	RGBColor	Farbe von Strukturen
STRUCELEMENT_color	RGBColor	Farbe von Strukturelementen
CLASS_DISTRICT_texture	String	Textur des Distrikts der Klassenbibliothek
REPORT_DISTRICT_texture	String	Textur des Distrikts der Programmbibliothek
FUNCTIONGROUP_DISTRICT_texture	String	Textur des Distrikts der Funktionsgruppen
DICTIONARY_DISTRICT_texture	String	Textur des Distrikts des ABAP Dictionary
BUILDING_texture	String	Textur von Gebäuden

Tabelle A.3: Konfigurationsmöglichkeiten des Generators

Literaturverzeichnis

- [CE00] Czarnecki, K.; Eisenecker, U.: *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2000
- [CMS97] Card, S.; Machinlay, J.; Shneiderman, B.: *Readings in Information Visualization Using Vision to Think*. Morgan Kaufmann, 1997
- [DAB⁺11] Ducasse, S.; Anquetil, N.; Bhatti, U.; Girba, T.; Hora, A.; Laval, J.: MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family. (2011)
- [Die03] Diehl, S.: *Softwarevisualisierung*. <https://www.gi.de/service/informatiklexikon/detailansicht/article/softwarevisualisierung.html>, 2003. – letzter Zugriff: 11.03.2017
- [Die07] Diehl, S.: *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer Verlag, Berlin Heidelberg, 2007
- [EAB⁺14] Evans, A.; Agenjo, J.; Bahrehmand, A.; Blat, J.; Romeo, M.: 3D Graphics on the Web: a Survey. In: *Interactive Technologies Group* (2014)
- [EKS07] Eicker, S.; Kahl, C.; Spies, T.: Softwarevisualisierung im Kontext service-orientierter Architekturen. In: *ICB Research Reports* (2007)
- [EM05] Eltoweissy, M.; Matkovic, K.: Software visualization. In: *Innovations in Systems and Software Engineering* 1 (2005), Nr. 2
- [Kov11] Kovacs, P.: *Ansatz zur Interaktion mit dreidimensionalen visualisierten Softwaredremodellen*. 2011. – Qucosa Publikationsserver
- [MCM02] Maletic, J.; Collard, M.; Marcus, A.: A Task Oriented View of Software Visualization. In: *Proceedings of The First International Workshop on Visualizing Software for Understanding and Analysis* (2002)
- [MFM03] Marcus, A.; Feng, L.; Maletic, J.: 3D Representations for Software Visualization. In: *Proceedings of the 2003 ACM Symposium On Software Visualization* (2003)

Literaturverzeichnis

- [MKSE11] Müller, R.; Kovacs, P.; Schilbach, J.; Eisenecker, U.: Generative Software Visualization: Automatic Generation of User-Specific Visualizations. In: *Proceedings of 2nd International Workshop on Digital Engineering* (2011)
- [Mü15] Müller, R.: *Software Visualization in 3D - Implementation, Evaluation and Applicability*, Universität Leipzig, Diss., 2015
- [PBS93] Price, B.; Baecker, R.; Small, I.: A Principled Taxonomy of Software Visualization. In: *Journal of Visual Languages and Computing* 4 (1993), Nr. 2
- [RC93] Roman, G.-C.; Cox, C.: A Taxonomy of Program Visualization Systems. In: *IEEE Computer* 26 (1993), December, Nr. 12
- [Sag17] Sager, T.: *Entwicklung eines Prototypen zur Extraktion statischer ABAP-Quelltextinformationen in FAMIX für die 3D-Softwarevisualisierung*. 2017. – Universität Leipzig, Leipzig, Germany
- [Shn92] Shneiderman, B.: Tree Visualizations with tree-maps: 2-d space filling approach. In: *ACM Trans. Graph* (1992)
- [SM00] Schumann, H.; Müller, W.: *Visualisierung: Grundlagen und allgemeine Methoden*. Springer-Verlag, Berlin Heidelberg, 2000
- [Wet10] Wettel, R.: *Software Systems as Cities*, Università della Svizzera Italiana, Diss., 2010
- [WL07] Wettel, R.; Lanza, M.: *Visualizing Software Systems*. (2007)

Erklärung

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