Gottfried Wilhelm Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Konzept und Umsetzung eines Refactorings für einen Parser und Generator für COBOL-Oberflächen

Diplomarbeit

im Studiengang Mathematik mit Studienrichtung Informatik

von
Iavor Fetvadjiev

Prüfer: Prof. Dr. Kurt Schneider
Zweitprüfer: Prof. Dr.-Ing. Helena Szczerbicka
Betreuer: Dipl. Wirt.-Inform. Daniel Lübke
Betreuer (FA frobese GmbH): Dipl. Ök. Roland Wünsch

Hannover, 14. Januar 2007

Erklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst habe und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe.

Hannover, den14.01.2007

Iavor Fetvadjiev

Danksagung

Ich danke Herrn Daniel Lübke und Herrn Roland Wünsch
für die hervoragende Betreuung dieser Arbeit
sowie meiner Frau Petia für die Unterstützung.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

COBOL *(Common Business Oriented Language = Allgemein kaufmännisch orientierte Programmiersprache)* ist bei den Großcomputern in den kaufmännischen Anwendungen, insbesondre im Bereich der Finanzindustrie, heute noch eine weit verbreitete Programmiersprache. Diese Sprache wurde entwickelt, um die im kaufmännischen Bereich anfallenden Aufgaben schnell und sicher zu verarbeiten. Daher werden auch Neu- und Weiterentwicklungen der Geschäftsanwendungen mit juristischen Dateninhalten weiterhin in der Programmiersprache COBOL durchgeführt.

Die Bedienung solcher Anwendungen wurde ausschließlich über Terminals mit formatiertem Text konzipiert. Diese Art Oberflächen und die ihnen zugrunde liegenden Bibliotheken wurden in den neunziger Jahren durch herstellerspezifische GUI-Emulationen auf Windows-Basis ersetzt. Einer der führenden Anbieter ist die Firma MicroFocus mit dem Produkt DialogSystem.

Auch diese Lösung erfüllt unter dem Blickwinkel von Mehrschichtarchitekturen

und Bedienerergonomie nicht mehr die heutigen Anforderungen. Deshalb wird sie auch nicht mehr weiterentwickelt. Daher hat Firma frobese GmbH Informatikservices einen Java-Ersatz für die alten COBOL-Oberflächenbibliotheken entwickelt, der die jetzigen Bedienbarkeitsanforderungen erfüllen soll.

Darüber hinaus, und um eine einfachere Migration von bestehenden MicroFocus DialogSystem-Oberflächen zu ermöglichen, entwickelt die Firma frobese einen Umsetzer (Compiler), mit dem alte Oberflächen in Java-Swing-Code maschinell überführt werden und sich so in die geschaffene Laufzeitumgebung integrieren lassen. Die Weiterentwicklung und die Pflege der Oberflächen sowie deren Logik sollen mit Standard-Java-Werkzeugen durchgeführt werden können.

Die Ersatzoberflächenbibliothek und der dazugehörige Maskencompiler können jedoch nicht allgemein verwendet und vom Software-Design her nicht weiterentwickelt werden. Komplexere Anwendungen führen zu zusätzlichen Aufwänden unterschiedlichen Umfangs in der Migration.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist, die innere Programmstruktur der Ersatzoberflächenbibliothek und des Maskenumsetzers zu verbessern *(Refactoring)*. Dazu sind passende Softwaremetriken und andere Software-Engeneerings-Verfahren zu verwenden, um den Zustand der Software vorher zu beschreiben, Ziele für das Refactoring zu definieren, und nach dem Refactoring die erreichten Verbesserungen festzuhalten. Der Schwerpunkt liegt hierbei auf der Codegeneratorkomponente des Systems. Daher soll die Grammatik der Oberflächensprache *DialogSystem* zunächst formal beschrieben werden.

1.3 Struktur der Arbeit

Im Kapitel 2 wird zunächst eine kurze Einführung in die für dieser Arbeit relevanten theoretischen Grundlagen gegeben. Im dritten Kapitel erfolgt eine Beschreibung der Komponenten des Systems. Im Kapitel vier werden die Anforderungen definiert sowie das Konzept des Refactorings dargestellt. Das Kapitel 5 präsentiert die Realisierung der Anforderungen. Die erzielten Ergebnisse werden im Kapitel 6 analysiert.

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen dieser Arbeit behandelt. Zu Beginn wird erklärt, was unter einem *Refactoring* zu verstehen ist. Diesbezüglich werde ich den Begriff *Softwaremetrik* als ein Grundwerkzeug des Refactorings sowie seine Grundlagen erläutern. Danach werde ich auf die theoretischen Grundbegriffe rund um die *Compilerkonstruktion* eingehen.

2.1 Refactoring Grundbegriffe

"Refactoring is a change made to the internal structure of a software component to make it easier to understand and cheaper to modify, without changing the observable behavior of that software component."

Martin Fowler
[MFOW]

Refactoring ist ein systematischer Ansatz zum Verbessern der inneren Struktur eines Programms. Der Ansatz besteht aus einer Serie von kleinen Modifikationen

des Designs. Dabei wird die vorhandene Software oder Teile davon verständlicher und leichter änderbar gemacht, ohne das beobachtbare Verhalten zu verändern. Die Modifizierung (die Umgestaltung des Quelletextes) des Programms erfolgt unter dem Gesichtspunkt folgender Qualitätsanforderungen:

- *Erweiterbarkeit*: Damit ist gemeint, dass die Programmierung einer Software ihre spätere unkomplizierte Erweiterbarkeit erlauben soll.
- *Verständlichkeit*: Dies bedeutet, der Code so zu gestalten, dass er für möglichst viele Programmierer verständlich ist. Die Verständlichkeit einer Software gilt als hoch, wenn ihre Struktur ihrem Verhalten angemessen ist, wenn es also eine enge Verbindung zwischen ihrer Struktur und ihrer Funktion gibt.
- *Wartbarkeit* ist ein Kriterium bei der Entwicklung von Software, das zeigt, mit welchem Erfolg Änderungen in einem Zusammenhang von Applikationen durchgeführt werden können.
- *Testbarkeit*: Darunter ist das Absichern der Möglichkeit zu verstehen, die korrekte Arbeitsweise des Programms möglichst einfach zu testen.
- *Vermeidung von Redundanz*: Es muss die Möglichkeit gewährleistet werden, konkrete Problemlösungen von anderen Stellen zu nutzen und diese nicht mehrfach implementieren zu müssen.

Da das Refactoring nur auf einen funktionierenden Code durchgeführt wird, besteht unter anderem ein erhöhtes Risiko, unerwünschte Änderungen sowie Fehler einzuführen. Um dieses Risiko zu vermeiden, wendet man Unit-Tests an. Dadurch wird sichergestellt, dass sich das Programm immer noch gleich verhält.

Ein Refactoring wird erleichtert und unterstützt durch den Einsatz von Werkzeugen. Insbesondere integrierte Entwicklungsumgebungen bieten eine Unterstützung

bei der Durchführung aller Refactoringsschritte. So eine IDE ist z.B. *Eclipse IDE*. *Eclipse* bietet automatisierte Refactoring-Funktionen, durch die sich der Aufwand deutlich minimieren lässt. Außerdem lassen sich damit Fehlerquellen effektiv ausschließen.

Um die Ziele des Refactorings zu definieren sowie die Ergebnisse quantitativ zu erfassen, werden Softwaremetriken herangezogen. Auf die Grundlagen der Softwaremetriken werde ich im nächsten Kapitel eingehen.

2.2 Softwarequalität messen

In diesem Abschnitt werde ich den Begriff *Softwaremetrik* erläutern. Um mithilfe von Softwaremetriken Aussagen über die Kostenaufwände und die Qualität fällen zu können, müssen ihre Eigenschaften festgestellt werden. Daher werde ich zunächst die theoretischen Grundlagen des Messens darlegen. Im Anschluss stelle ich einige für diese Arbeit wichtige Metriken vor und werde ihre Eigenschaften kommentieren.

2.2.1 Messtheoretische Grundbegriffe

Analog zur Messung von Volumina, Längen oder Massen lässt sich auch das Messen von Softwareobjekten definieren. Dabei wird zwischen empirischem und numerischem Bereich unterschieden. "Messen" bedeutet eine Abbildung der empirischen auf den numerischen Objekten. Diese Abbildung lässt sich mathematisch durch einen *Homomorphismus* darstellen.

Definition (Empirisches relationales System) *Sei A eine Menge von Objekten und sei $R \subseteq A^2$ eine Relation auf A und $\circ$ eine binäre Verknüpfungsoperation,*

so ist $\mathcal{A} = (A, R, \circ)$ *ein* ***empirisches relationales System.***[TFET]

Eine mögliche Relation auf Softwareobjekte[1] wäre z.B. die Verständlichkeit eines Programms. Es gilt also: Wenn $PROG1$ und $PROG2$ zwei Programme sind, die in einer Verständlichkeitsrelation zueinander stehen, dann würde dies bedeuten, dass $PROG1$ von einem Programmierer genau so gut zu verstehen ist wie $PROG2$.

Definition (Homomorpismus) *Es seien* $\mathcal{A} = (A, R, \circ)$ *ein empirisches relationales System und* $\mathcal{B} = (B, S, \bullet)$ *ein* ***numerisches relationales System***[2]*. Eine Abbildung* $\mu : A \to B$ *heißt genau dann Homomorpismus, wenn es gilt:*

$$\forall a, b \in A : aRb \Leftrightarrow \mu(a)S\mu(b)$$

und

$$\forall a, b \in A : \mu(a \circ b) = \mu(a) \bullet \mu(b)$$

[TFET]

Ein Homomorphisus bildet also empirische auf numerische Objekte ab. Die Menge der numerischen Objekte bestehet im Allgemeinen aus den reellen Zahlen. Sie bilden den Wertebereich der Maße. Jedes Maß mit der $\geq$ Relation auf reellen Zahlen induziert eine *Ordnung* auf den empirischen Objekten. Auf diese Weise lassen sich die Eigenschaften der Maße nachvollziehen.

Da der Einsatz der Meßtheorie als ein Werkzeug, mit dem Eigenschaften von Softwaremetriken abgeleitet werden, für dieser Arbeit nicht relevant ist, wurde sie an dieser Stelle nur in Kürze ausgeführt. Eine umfangreiche Analyse der Softwaremetriken mithilfe der Maßtheorie bieten [TFET], [HZUS] sowie [FROB].

[1] Softwareobjekte können als empirische Objekte erfasst werden.

[2] Numerisches relationales System lässt sich analog zum empirisches relationalen System definieren.

2.2.2 Klassifikation von Softwaremetriken

Um die Softwaremetriken zu klassifizieren, sollte zu jeder Metrik eine bestimmte Eigenschaft des Softwareproduktes zugeordnet werden. Da für diese Zuordnung im allgemeinen keine formale Definition gibt, werde ich im Folgenden einige Klassifikationen vorstellen, die in der Praxis am meisten verbreitet sind.

Um Metriken einerseits für Qualitätskontrolle und andererseits für die Planung der Ressourcen und Kosten einzusetzen, wurde ein weiteres Klassifikationsmerkmal eingeführt. Es wird unter *Prozess-* und *Produktmetriken* unterschieden.

- *Prozessmetriken* Mit Hilfe von Prozessmetriken können Dauer und Kosten des Softwareprojekts abgeschätzt werden.
- *Produktmetriken* Diese Metriken geben Aufschluss über Umfang, Komplexität, Verständlichkeit und Qualität der Software.

Im objekt-orientierten Bereich wird oft als Klassifikationsmerkmal die *Abstraktionsebene* angewandt. Somit lassen sich die Produktmetriken in folgende drei Gruppen einteilen:

- *Metriken auf Methodenebene*
- *Metriken auf Klassenebene*
- *Metriken auf Klassenstrukturenebene*

2.2.3 Auszug aus Produktmetriken

Lines of Code (LOC) *Lines of Code* dient zur Beschreibung der Größe (SIZE) von Programmen. Als Messwert wird die Anzahl von Programmzeilen betrachtet, die eventuell von Kommentarzeilen und leeren Zeilen bereinigt sind (Anzahl der Anweisungen im Programm). LOC lässt sich auf Methoden-, Klassen-, und Programmebene anwenden. Auf Klassenniveau wird LOC auch als "Anzahl Semikolons in der Klasse" definiert. Auf dem Kontrollfluss einer Methode M gibt $LOC(M)$ die Anzahl der Knoten n an.

Ein bekanntes Problem von LOC ist, dass es unmöglich ist, dieses Maß für Sprachen, die das Semikolon nicht als Anweisungs-Separator verwenden, zu benutzen.

McCabe's Cyclomatic Compexity (MCC) Die McCabe-Metrik (auch zyklomatische Komplexität gennant) ist auf dem Kontrollflussgraphen eines Programms basierend definiert. Damit lässt sich die Komplexität eines Software-Moduls messen. Die Idee dabei ist, dass eine zu hohe Komplexität vom Menschen nicht mehr zu verstehen ist.

Sei M ein Flussgraph mit e_M Kanten, m_M Knoten und p_M unabhängigen Teilgraphen. Dann ist $MCC(M) = e_M - n_M + 2p_M$.

Diese Metrik ist nur auf Methodenebene anwendbar, da nur dort ein Kontrollflussgraph in diese Sinne konstruierbar ist.

Halstead-Metric (HAM) HAM fasst eine Reihe von Kennzahlen zusammen. Damit lässt sich die Komplexität und die Größe des Codes ableiten. Diese Metrik basiert auf den im Quellcode verwendeten Operanden und Operatoren.

- *Operator*: Jedes Schlüsselwort, das eine Aktion kennzeichnet. Z.B. *for, if, +*
- *Operand*: Alle übrigen Symbole wie Parameter und Variablen. Operanden werden von Operatoren verändert.

Für jedes Programm werden zuerst folgende Maße ermittelt:

- n_1: Anzahl der verwendeten unterschiedlichen Operatoren.
- n_2: Anzahl der verwendeten unterschiedlichen Operanden.
- n: $n_1 + n_2$
- N_1: Anzahl der insgesamt verwendeten Operanden.
- N_2: Anzahl der insgesamt verwendeten Operanden.
- N: $N_1 + N_2$

Aus den Basisgrößen lassen sich das Halstead-Volumen HAM_V und die Halstead-Länge HAM_L des Programms P ermitteln:

$$HAM_L(P) = n_1 * log_2 n_1 - n_2 * log_2 n_2$$

und

$$HAM_V(P) = N * log_2 n$$

Darüber hinaus wird

$$S = n1\frac{1}{2}\frac{N_2}{n_2}$$

als die Schwierigkeit, ein Programms zu verstehen, interpretiert.

2.3 Compilerkonstruktion

In den weiteren Kapiteln werden Softwarekomponenten ausführlich beleuchtet, deren Aufgabe darin besteht, das in einer Programmiersprache geschriebene Programm in ein äquivalentes Programm in einer andere Programmiersprache umzusetzen. Diese Aufgabe ist im Allgemeinen nicht trivial.

Der nächste Abschnitt führt in die Grundbegriffe und Methoden des Compilerbaus und in die Theorie der formalen Sprachen ein, soweit diese für die Definition der Syntax und die Analyse von Programmiersprachen benötigt werden.

2.3.1 Formale Sprachen und Grammatiken

Historisch betrachtet stammt die Theorie der formalen Sprachen aus Arbeiten von Noam Chomsky[3]. Er hat versucht, mit Hilfe der formalen Logik den Aufbau natürlicher Sprachen zu beschreiben. Sein Ansatz basiert auf einem Satz von Regeln, die den syntaktischen Aufbau von Sätzen beschreiben(s. Bsp. 2.1). [HKOPP]

Beispiel 2.1:

Regel 1: $\underline{SATZ} \rightarrow \underline{SUBJEKT} \quad \underline{PRAEDIKAT} \quad \underline{OBJEKT}$

Regel 2: $\underline{OBJEKT} \rightarrow \underline{ARTIKEL} \quad \underline{SUBSTANTIV}$

Regel 3: $\underline{ARTIKEL} \rightarrow \underline{den} \quad \underline{der}$

Im Sinne solcher Regeln kann man verschiedene "korrekte" Sätze bilden, die jedoch semantisch nicht immer sinnvoll sein müssen. Dies zeigt die Unbrauchbarkeit dieser Methode zum Erfassen des Aufbaus natürlicher Sprachen. Zur syntaktischen

[3]Noam Chomsky (geb. 1928), amerikanischer Linguist, Begründer der generativen Transformationsgrammatiken.

Beschreibung und Analyse von Programmiersprachen zeigt sich der Ansatz jedoch als besonders nützlich.

Ich werde im Folgenden einige der wichtigsten Begriffe der Theorie der formalen Sprachen kurz erläutern:

Definition (Alphabet): *Ein* ***Alphabet*** Σ *ist eine endliche, nichtleere Menge von Zeichen oder Symbolen, die nicht weiter zerlegbar sind.*[HKOPP]

Definition (Wort): *Ein* ***Wort*** ω *über dem Alphabet* Σ *ist eine endliche Folge von Symbolen aus* Σ. $\omega = a_1...a_n, a_i \in \Sigma, 1 \leq i \leq n, n \geq 0$. *Die Länge von* ω *wird bezeichnet mit* $|\omega|$ *und gibt die Anzahl der Zeichen des Wortes. Für das* ***leere Wort*** ϵ *gilt* $|\epsilon| = 0$.[RPAR]

Eine **formale Sprache** ist jede Teilmenge $L \subset \Sigma^*$, wobei Σ^* die Menge aller Wörter über Σ bezeichnet.

Beispiel 2.2:
Sei $\Sigma_1 = \{a, b, c, d\}$. Dann ist $\omega_1 = ddcb$ ein Wort über Σ_1 und $|\omega_1| = 4$.

Definition (Produktion): ***Produktionen (Regeln)*** *sind Vorschriften in der Form* $\omega_1 \rightarrow \omega_2$ *zum Ersetzen von Wörtern durch andere. Man nennt eine endliche Menge von Produktionen* ***Produktionssystem*** *und bezeichnet es mit P (s. Bsp. 2.1).*

Mehrere Produktionen mit gleicher linker Seite lassen sich auch in der Form $\omega \rightarrow \omega_1|\omega_2|...|\omega_k$ zusammenfassen.

Das einem Produktionssystem P zugrunde liegende Alphabet Σ_p lässt sich in zwei disjunkte, nichtleere Mengen $T \subset \Sigma_p$ **terminales Alphabet** und $N \subset \Sigma_p$ **nichtterminales Alphabet** zerlegen. Dabei gilt: $N \cap T = \varnothing$ und $N \cup T \cup \epsilon = \Sigma_p$.

Definition (Terminales Alphabet): *Ein* ***terminales Alphabet T*** *enthält nur terminale Symbole bzw. Terminalzeichen. Ein Terminalsymbol t kann nicht auf der linken Seite einer Produktion vorkommen. Es gilt also:* $\forall \omega_l, \omega_r \in \Sigma_p^*$, $P : \omega_l \rightarrow \omega_r$, $\omega_l \notin T$.

Definition (Nichtterminales Alphabet): *Ein* ***nichtterminales Alphabet N*** *enthält nur nichtterminale Symbole bzw. Nichtterminalzeichen. Die Nichtterminalsymbole sind alle übrigen Zeichen des Alphabets außer den terminalen Symbolen.*

Die Nichtterminalsymbole N wirken im Gegensatz zu den Terminalsymbolen T wie Platzhalter, die in den Produktionen durch andere Platzhalter oder Terminalsymbole oder eine beliebige Kombination von Platzhaltern und Terminalsymbolen ersetzt werden.

Definition (Grammatik): *Eine* ***formale Grammatik G*** *ist ein Tupel* $G = (N, T, P, S)$ *mit:*

- *N ist ein nichtterminales Alphabet,*
- *T ist ein terminales Alphabet,*
- *P ist eine endliche Menge von Produktionen (Produktionsmenge),*
- *S ist ein Startsymbol (auch Axiom) mit* $S \in N$.

Es gilt noch: N und T sind disjunkt, sowie $N, T, P \neq \varnothing$

Sei $G = (N, T, P, S)$ eine Grammatik, dann ist die Menge aller Wörter ***L(G)*** die von G erzeugte **Sprache**. Sie lässt sich vom Startsymbol S herleiten. d.h.

$$L(G) = \{\omega | \omega \in T^*, S \Rightarrow \omega\}^4 \text{ [CPOLZ]}$$

[4] $S \Rightarrow \omega$ bedeutet, dass ω sich direkt oder indirekt aus S herrleiten lässt.

Durch sinnvolle Einschränkungen bezüglich des Typs der Produktionen lassen sich Grammatiken nach dem Grad ihrer Komplexität klassifizieren. Noam Chomski beschreibt vier Grammatiktypen. Alle Typen entstehen aus einer uneingeschränkten Grundgrammatik *(Typ 0)* durch zunehmende Spezialisierung der Produktionen. Daraus ergibt sich folgende Hierarchie der Grammatik-Klassen:

Definition (Typ 0 Grammatiken): *G_0 ist eine Typ-0-Grammatik, wenn gilt $P : ((T \cup N)^+ - T^*) \rightarrow (T \cup N)^*$. Die Produktionen von G_0 unterliegen also keinerlei Einschränkungen.*

Grammatiken vom Typ 0 heißen **rekursiv-aufzählbar**. Die von G_0 erzeugten Sprachen heißen **rekursiv-aufzählbare Sprachen**.

Definition (Typ 1 Grammatiken): *G_1 ist eine Typ-1-Grammatik, wenn bei einer Ableitung entstehende Wörter (rechte Seite) immer länger oder gleichlang der linken Produktionsseite sind. Es gilt also: $P : ((T \cup N)^+ - T^*) \rightarrow (T \cup N)^*$ und $\forall p \in P$ mit $p : \alpha \rightarrow \beta$; $\alpha, \beta \in \Sigma^*$: $|\alpha| \leq |\beta|$.*

Grammatiken von Typ 1 werden auch **kontextsensitiv** gennant.

Typ 0 und Typ-1-Grammatiken eignen sich nicht besonders gut für die Analyse von Programmiersprachen, da die zur Verfügung stehenden Methoden sehr aufwendig zu benutzen sind. Dennoch ist die Beschreibung sehr komplexer Strukturen nur mithilfe dieser Grammatiktypen möglich.[HKOPP]

Definition (Typ 2 Grammatiken): *$G_2 = (N, T, P_2, S)$ ist eine Grammatik vom Typ 2, wenn gilt: $P_2 : A \rightarrow \omega$ mit $A \in N$ und $\omega \in \Sigma^*$.*

Bei den Typ 2 Grammatiken werden die nichtterminalen Symbole N in einem Wort, ohne Betrachtung des Kontextes ersetzt. Daher werden Grammatiken vom

Typ 2 auch **kontextfrei** gennant. Sie genügen im Wesentlichen, um die Struktur und die Syntax von Programmiersprachen zu beschreiben und zu analysieren.

Definition (Typ 3 Grammatiken): *Eine Grammatik* $G_3 = (N, T, P_3, S)$ *ist vom Typ 3, wenn sie kontextfrei ist, und deren Produktionen* P_3 *die Form* $A \to \alpha B | \alpha$ ***(linkslinear)*** *oder* $A \to B\alpha | \alpha$ ***(rechtslinear)*** *mit* $A, B \in N$ *und* $\alpha \in T^*$ *haben.*[RPAR]

Typ 3 Grammatiken heißen **regulär**. Reguläre Sprachen $L_3 = L(G_3)$ besitzen die einfachste Struktur. Zur Beschreibung von Programmiersprachen werden sie bei der lexikalischen Analyse gebraucht.

Die oben genannten Grammatikdefinitionen bauen die so genante *Chomsky-Hierarhie* auf, welche in Abbildung 2.1 illustriert ist:

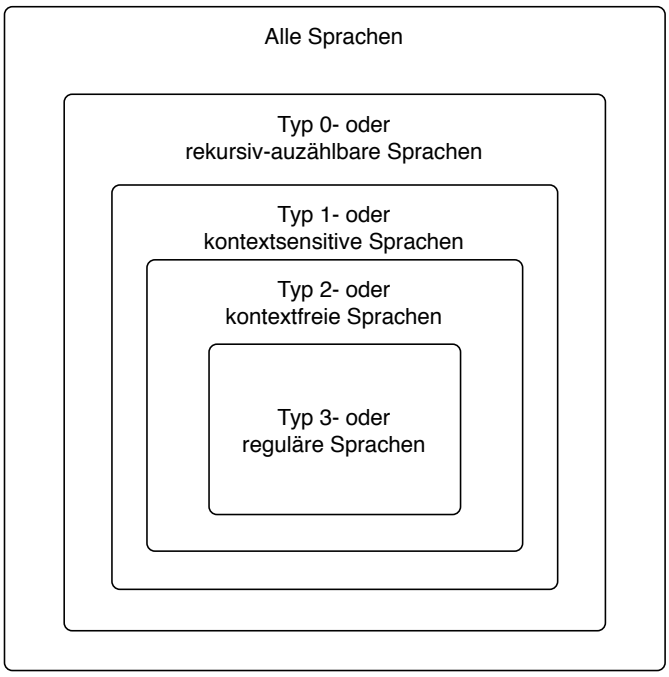

Abbildung 2.1: Die Chomsky-Hierarchie. *Quelle:* [GVKUW]

Formal lässt sich die Hierarchie auch so zum Ausdruck bringen:
$L(G_3) \subset L(G_2) \subset L(G_1) \subset L(G_0) \subset 2^{T^*}$, wobei 2^{T^*} die Menge aller Sprachen darstellt.

Die "klassische" Chomsky-Hierarhie lässt sich in viele Unterhierarchiestufen weiter verfeinern. Im Folgenden werde ich jedoch nur auf die Aspekte eingehen, die den Aufbau des in dieser Arbeit beschriebenen Compilers betreffen.

Die Klasse der ***deterministischen kontextfreien Sprachen*** $DPDA_T$ ist eine Sprachklasse von großer praktischer Bedeutung. Sie lässt sich in die Hierarchie wie folgt einordnen:

$$L(G_3) \subset DPDA_T \subset L(G_2) \subset ...$$

Um die Spezifika dieser Sprachklasse zu erläutern, ist es nötig, einige Begriffe von der Theorie der kontextfreien Sprachen darzulegen.

Definition (PDA[5])[GVKUW]: *Ein Kellerautomat (PDA) ist ein Automat* $K = (T, S, \Gamma, \delta, s_0, \perp, F)$ *mit:*

- T *ist ein Eingabealphabet,*
- S *ist eine endliche Zustandsmenge,*
- Γ *ist ein Kelleralphabet,*
- δ *ist eine Zustandsüberführung mit:* $\delta : S \times (T \cup \{\epsilon\}) \times \Gamma \rightarrow 2^{S \times \Gamma^*}$,
- s_0 *ist ein Startzustand mit* $s_0 \in S$,
- $\perp$ *ist ein Kellersymbol mit* $\perp \in \Gamma$,
- F *ist die Menge der Endzustände mit* $F \subseteq S$.

Mit $k = (s, \omega, \alpha) \in (S \times T^* \times \Gamma^*)$ wird die Konfiguration eines Kellerautomaten bezeichnet, wobei s der aktuelle Zustand, ω das noch zu verarbeitende Wort, und

[5]PDA: push down automation (nichtdeterministische Kellerautomat)

α der Kellerinhalt ist. Die Menge der Konfigurationsübergänge $\vdash$ lässt sich durch die Relation $\vdash \subseteq (S \times T^* \times \Gamma) \times (S \times T^* \times \Gamma)$ definieren. Dabei gilt:

$$(a, av, A\alpha) \vdash (s', v, \beta\alpha) \Rightarrow (s', \beta) \in \delta(s, a, A)$$

mit $s, s' \in S$, $a \in T \cup \{\epsilon\}$, $v \in T^*$, $\alpha, \beta \in \Gamma^*$ und $A \in \Gamma$.

Definition ($\mathbf{PDA_T}$): *Es sei $K = (T, S, \Gamma, \delta, s_0, \bot, F)$ ein PDA. Dann ist*

$$L(K) = \{\omega \in T^* | (s_0, \omega, \bot) \vdash^* (s_f, \epsilon, \gamma), s_f \in F, \gamma \in \Gamma^*\}$$

die von K über T akzeptierte Sprache, und mit PDA_T wird die Klasse der von nichtdeterministischen Kellerautomaten akzeptierten Sprachen über T bezeichnet. [GVKUW]

Die oben beschriebene Sprachenklasse PDA_T ist ein Äquivalent der Klasse der kontextfreien Sprachen L_{kf}. Es gilt also: $L(K) \equiv L(G_2)$. Damit können wir die Chomsky-Hierarhie um noch ein Glied erweitern:

$$L(G_3) \subset PDA_T = L(G_2) \subset ...$$

Die populärsten Parser implementieren einen abstrakten Automaten, der meist als Kellerautomat realisiert ist. In der Praxis wird jedoch nicht der nichtdeterministische Kellerautomat PDA realisiert, da beim PDA das Erkennen von Wörtern exponentiell lange "dauern" kann. Wenn z.B. ein Programm aus 2500 Zeichen besteht, kann der syntaktische Test darauf, ob das Programm akzeptiert wird oder nicht, mehr als 2^{2500} Schritte dauern. Auch bei diesem relativ kurzen Programm werden sogar die schnellsten Rechner überlastet. Daher wird ein *deterministischer Kellerautomat DPDA*[6] konstruiert, der in der Lage ist, eine lineare Parsezeit zu garantieren.

[6]DPDA: deterministic push down automaton

Definition (deterministische Kellerautomat): *Ein Kellerautomat* $K = (T, S, \Gamma, \delta, s_0, \perp, F)$ *heißt deterministisch oder DPDA, falls:*

$$\forall s \in S, a \in T, A \in \Gamma : |\delta(s, a, A)| + |\delta(s, \epsilon, A)| \leqslant 1.$$

[GVKUW]

Die Klasse der Sprachen über T, die von einem deterministischen Kellerautomaten akzeptiert werden, werden mit $DPDA_T$ bezeichnet. Dabei gilt $DPDA_T \subset PDA_T$. Nicht alle kontextfreien Sprachen sind also deterministisch. Es gilt aber, dass jede reguläre Sprache deterministisch kontextfrei ist.

Obwohl das $DPDA_T$ gegenüber PDA_T an Mächtigkeit verloren hat, erweist es sich in der Praxis als sehr nützlich. Die Klasse $DPDA_T$ stimmt mit der Klasse der ***LR(k)***[7]-Sprachen überein. Es gilt also $DPDA_T \equiv LR(k)$. Die Konzepte dieser Sprachklasse reichen weitgehend aus, um die Sprachdefinitionen in der Praxis abzuwickeln. Für sie gibt es sehr effiziente Syntaxanlyse- und Parse-Algorithmen. Im Vergleich zu den anderen Sprachen lässt sich $DPDA_T$ bzw. $LR(k)$ so darstellen:

$$L(G_3) \subset LR(k) = DPDA_T \subset PDA_T = L(G_2)$$

Die meisten in einem Compiler eingesetzten Verfahren zur syntaktischen Analyse behandeln die $LR(k)$-Grammatiken. In der Praxis werden die direkt auf $LR(k)$ beruhenden Parser jedoch zu groß. Man beschränkt sich daher auf die Teilklasse ***LALR(1)***[8] oder sogar auf ihre Unterklasse ***LL(1)***[9].

[7] LR(k): Lesen der Eingabe von **L**inks nach rechts, Erzeugen einer **R**echtsreduktion und **k** Zeichen Lookahead.

[8] LALR(1): Lookahead-LR

[9] LL(1): Lesen der Eingabe von **L**inks nach rechts. Erzeugen einer **L**inksableting und **ein** Zeichen Lookahead.

$$L(G_3) \subset LL(1) \subset LALR(1) \subset LR(k) = DPDA_T \subset PDA_T = L(G_2)$$

Zur formalen Festlegung der Charakteristika der letzten Grammatiktypen sollen zunächst zwei Funktionen, **first**() und **follow**(), sowie der Begriff **Handle** definiert werden.

Definition (*first()*): *Sei $\alpha \in (N \cup T)^*$. Dann ist*

$$first(\alpha) = \{\omega | \alpha \stackrel{*}{\Rightarrow} \alpha\beta\},\ \alpha \in T,\ \beta \in (N \cup T)^*\} \cup \{\epsilon | \alpha \stackrel{*}{\Rightarrow} \epsilon\}.$$

[RPAR]

Definition (*follow()*): *Sei $B \in N$. Dann ist*

$$follow(B) = \{\omega | S \stackrel{*}{\Rightarrow} \alpha B \omega \beta, \omega \in T, \alpha, \beta \in (N \cup T)^*\} \cup \{\$ | S \stackrel{*}{\Rightarrow} \alpha B, \alpha \in (N \cup T)^*\}$$

wobei \$ das Ende der Eingabe markiert.

[RPAR]

Definition (*Handle*): *Ein Teilwort β eines Wortes $\alpha\beta\gamma$ mit $\alpha\beta\gamma \in (N \cup T)^*$ heißt **Handle**, falls β die rechte Seite einer Produktion $A \rightarrow \beta$, $A \in N$ ist , und der Reduktionsschritt[10] $\alpha\beta\gamma \leftarrow \alpha A\gamma$ sich zu einer linkskanonischen Reduktion fortsetzen läßt.*

[HKOPP]

[10]Reduktion: Beim LR(k)-Parsingverfahren werden **Reduktionen** von der Eingabe zum Startsymbol berechnet.

Definition (*LR(k)* Grammatik): *Eine konextfreie Grammatik* $G_{lr(k)} = (N, T, P, S)$ *ist eine* $LR(k)$ *Grammatik, wenn jeder Reduktionsschritt eindeutig durch k Symbole der Eingabe (Lookahead) bestimmt ist.*

Eine Grammatik ist also vom Typ *LR(k)*, falls sich durch den aktuellen Kellerinhalt und die nächsten k Zeichen der Eingabe das *Handle* eindeutig bestimmen lässt.

Definition (*LL(1)* Grammatik): *Eine konextfreie Grammatik* $G_{ll(1)} = (N, T, P, S)$ *heißt* $LL(1)$*-Grammatik, falls für alle* $A \in N$ *mit* $A \rightarrow \alpha_1|...|\alpha_n \in P$ *gilt:*

-$First(\alpha_1)...First(\alpha_n)$ *sind paarweise disjunkt.*

-Ist $\epsilon \in First(\alpha_j)$*, dann ist Follow(A)*$\cap$*First(*$\alpha_i$*)=0 für* $1 \leq i \leq n, i \neq j$

[RPAR]

Die Charakteristika der *LALR(1)* Grammatiktyp werden aus der Spezifikation eines bestimmten Verfahrens für die Syntaxanalyse abgeleitet. Daher werde ich sie in diesem Abschnitt nicht ausführlich beleuchten.

2.3.2 Notation von Grammatiken

Da die Produktionsmenge einer Grammatik auch sehr umfangreich sein könnte, wurden unterschiedliche Notationen entwickelt, um die Produktionsdarstellung möglichst übersichtlich zu gestalten. In diesem Abschnitt werde ich einige unterschiedliche Darstellungsformen für kontextfreie Grammatiken betrachten, die bei der Lösung praktischer Problemstellungen häufig eine Anwendung finden.

Backus-Naur-Form (BNF) Die *Backus-Naur-Form* wurde nach *John Backus*[11] und *Peter Naur*[12] benannt. Diese Notation wird speziell zur Syntaxbeschreibung der Programmiersprachen angewandt. Dabei gelten folgende Regeln:

r1) Nichtterminale Symbole haben die Form: $< name >$.

r2) Terminale Symbole werden als Zeichen oder Tokennamen in der Form: *tokenname* bezeichnet.

r3) Das Zeichen | kennzeichnet Alternative.

r4) Die Zeichenfolge ::= kennzeichnet Definition.

Beispiel 2.3 (BNF):

$$<digit> ::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 0$$
$$<digitsequence> ::= <digit> \mid <digit><digitsequence>$$
$$<number> ::= - <digitsequence> \mid <digitsequence>$$

Erweiterte Backus-Naur-Form (EBNF) Die erweiterte *Backus-Naur-Form* erlaubt Iterationen von Zeichenketten in einer "Produktion".

r1) $M ::= \alpha\{\beta\}\gamma$ bedeutet: $M \Rightarrow^* \alpha\beta\gamma$, $i \geq 0$. Die Zeichenketten, die sich aus M herleiten können, bestehen aus einem α null oder mehreren β und einem γ.

r2) $M ::= \alpha[\beta]\gamma$ bedeutet: $M \Rightarrow \alpha\beta\gamma$ oder $M \Rightarrow \alpha\gamma$. β kann also nur ein oder kein Mal auftreten.

Beispiel 2.4 (EBNF):

$$<digit> ::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 0$$
$$<number> ::= [-]<digit>\{<digit>\}$$

[11] John Backus: Amerikanischer Mathematiker, geb. am 3. Dezember 1924 in Philadelphia.

[12] Peter Naur: Dänischer Informatiker geb. am 25. Oktober 1928 in Frederiksberg.

Manche Parser-Generatoren verwenden eine eigene Form der BNF als Eingabe und generieren hieraus einen Parser für die zugrundegelegte Programmiersprache. Ein der meist verbreiteten Parsergeneratoren in der UNIX Welt *yacc* generiert einen tabellengesteuerten Parser aus einer BNF-Definition, wobei nur Produktionen (: statt ::=) und Alternativen (|) zulässig sind.

Reguläre Ausdrücke (REGEXP) Die *regulären Ausdrücke* haben sich als ein Formalismus zur Beschreibung von regulären Sprachen eingebürgert. Ein regulärer Ausdruck wird rekursiv definiert und bezeichnet eine formale Sprache.

Reguläre Ausdrücke unterstützen genau drei Operationen: *Alternative, Aneinanderreihung und Wiederholung.* Die Syntax von *REGEXP* lässt sich formal wie folgt definieren:

1) Das leere Wort ϵ ist ein $REGEXP$.
2) $x, y \in REGEXP \Rightarrow (x \cup y)$, xy, x^+ und $x^* \in REGEXP$
3) Jedes Zeichen aus dem Alphabet Σ ist ein regulärer Ausdruck

2.3.3 Prinzipieller Aufbau eines Übersetzers

Unter dem Begriff *Übersetzung* ist generell die Umwandlung eines in einer Programmiersprache L_1 (Quellcode) geschriebenen Programms in eine andere Programmiersprache L_2 (Zielcode) aufzufassen. Dabei bleiben das Ziel- und Quellprogramm semantisch äquivalent. Das Programm, dass die Übersetzung durchführt, heißt Umsetzer oder Compiler.

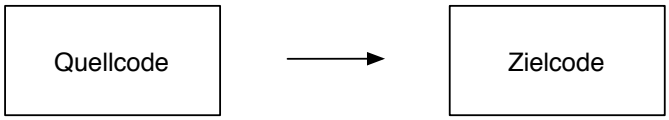

Abbildung 2.2: Übersetzung

Üblicherweise handelt es sich dabei um die Umsetzung (das Compilieren) eines in einer höheren Programmiersprache wie JAVA oder C++ geschriebenen Quelltextes in eine Maschinensprache oder einen Bytecode. Bei der in dieser Arbeit betrachteten Compiler-Komponente ist das jedoch nicht der Fall. Das Ziel- und Quellprogramm werden hier in einer jeweils höheren Programmiersprache realisiert.

Der Umsetzungsvorgang lässt sich in verschiedene Phasen unterteilen (s. Abb. 2.3), die jeweils verschiedene Teilaufgaben der Umsetzung übernehmen.

Dabei lassen sich die ersten drei Schritte (s. Abb. 2.3) unter dem Begriff der *Analysephase* und die letzten drei unter dem der *Synthesephase* zusammenfassen. Die Phasen der Compilierung werden sequentiell vom Compiler ausgeführt. Im Folgenden werde ich alle Compiler-Phasen beschreiben.

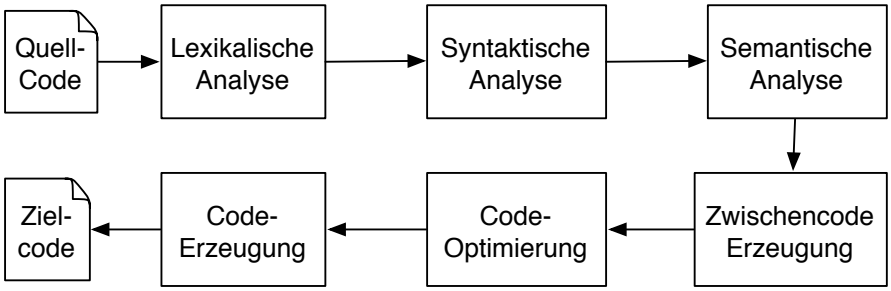

Abbildung 2.3: Compiler-Phasen

Lexikalische Analyse Die Aufgabe der lexikalischen Analyse besteht im Wesentlichen darin, Vorbereitungen für die eigentliche Umsetzung zu treffen. In diesem Schritt werden Eingabezeichen gelesen und zu größeren logischen Einheiten *(Tokens)* zusammengefasst. So wird z.B. die Zeichenfolge "10000" als eine Zahl aufgefasst. Bei der lexikalischen Analyse wird außerdem der Quellcode komprimiert, das heißt dieser wird z.B. von überflüssigen Leerzeichen sowie Kommentaren bereinigt. Fehler, soweit diese erkannt wurden, werden behandelt. Die Komponente des Compilers, die diese Aufgabe übernimmt, heißt *Lexical Scanner* oder einfach *Scanner*.

Das wesentliche Problem beim Konstruieren eines Scanners ist das Klassifizieren von Tokens. Zu diesem Zweck wird jede Tokenklasse[13] durch einen regulären Ausdruck definiert. Es gilt also, dass alle Worte, die zu einer Tokenklasse gehören, eine reguläre Sprache bilden (s. Kap. 2.3.1 - Typ 3 Grammatik).

Ein Scanner lässt sich auch durch einen sog. *Scanner-Generator* erzeugen. Dabei reicht es, die regulären Ausdrücke zu definieren und sie dem Scanner-Generator zu übergeben.

[13]Tokenklasse: Tokens werden abhängig von der beabsichtigen Anwendung zu **Tokenklassen** klassifiziert. Z.B. ZAHL, MINUSZEICHEN, IDENT usw.

Syntaktische Analyse Aufgabe der syntaktischen Analyse ist zu prüfen, ob die vom lexikalischen Scanner übergebene Tokenfolge der Grammatik der Quellprogrammiersprache entspricht. Es soll also kontrolliert werden, ob die Tokens in der richtigen Reihenfolge auftreten. Dabei werden Tokens zu größeren syntaktischen Baumstrukturen *(Syntaxbäumen)* zusammengefasst. Bei der syntaktischen Analyse werden durch Fehlerdiagnosen eventuelle syntaktische Fehler festgestellt. Diese Komponente des Compilers wird *Parser* gennant.

Je nach der Mächtigkeit der Grammatik setzt man verschiedene Parse-Verfahren ein, wobei alle Verfahren jeweils eine Teilklasse der kontextfreien Grammatik (s. Kap. 2.3.1 - *Typ 2 Grammatik*) behandeln. Weiter wird innerhalb der Parsertypen nach genereller Vorgehensweise zwischen *top-down* und *bottom-up* differenziert. Im Folgenden werde ich die Besonderheiten der unterschiedlichen Parse-Verfahren erläutern.

Top-Down-Parser

Bei dem Top-Down-Parsers wird versucht, den Syntaxbaum zum Eingabewort von oben nach unten und von links nach rechts zu rekonstruieren. Man erzeugt also eine Linksableitung zum Eingabewort. Nur eine stark begrenzte Auswahl von kontextfreien Grammatiken des Typs LL(1) (s. Kap. 2.3.1 - *LL(1)-Grammatik*) ermöglicht diese Art des Parsens.

Wenn man sich für dieses Parserverfahren entscheidet, muss zuerst geprüft werden, ob die Grammatik des Quellprogramms eine LL(1)-Grammatik ist. Falls nicht, könnten die Produktionsregeln durch passende Transformationen LL(1) konform gemacht werden. Solche Transformationen sind z.B. *Entfernen gemeinsamer Präfixe* oder *Entfernen linksrekursiver Produktionen.*

In der Praxis wird dieses Verfahren aufgrund seiner sehr einfachen Implementie-

rungstechnik oft eingesetzt. Wegen der Grammatik-Restriktionen von LL(1) wird es nur bei Sprachen mit sehr einfachen Grammatikregeln angewendet.

Bottom-Up-Parser

Bei dem Bottom-Up-Parsers wird versucht, im Gegensatz zu den Top-Down-Parsers, den Syntaxbaum zum Eingabewort von unten nach oben zu rekonstruieren. Im Vergleich zu dem Top-Down-Parser ermöglicht diese Methode das Erkennen einer viel mächtigeren Grammatikklasse, die Klasse der *deterministisch kontextfreien* Grammatiken (s. Kap. 2.3.1 - $DPDA_T$). Je nach spezifischer Vorgehensweise wird unter verschiedenen Bottom-Up-Parsers LR, LALR und SLR unterschieden. Die in der Praxis meist verbreiteten Bottom-Up-Parsers sind die tabellengesteuerten LALR(1) Parsers.

Da es kontextfreie Sprachen gibt, für die keine kontextfreie Grammatik existiert, die ein LALR-Parsing ermöglicht, muss die Grammatik des Quellprogramms auf potenzielle Konflikte aufgrund Mehrdeutigkeit untersucht werden. Solche Konflikte können auftreten, wenn z.B.:

k1) die rechte Seite einer Produktion gleichzeitig Präfix einer anderen Produktion mit einen gleichen Nichtterminal auf der linken Seite ist.

k2) die rechte Seite einer Produktion zugleich Suffix einer anderen Produktion mit gleichen Nichtterminal auf der linken Seite ist. [RPAR]

Ein Parser lässt sich auch durch einen sog. *Parser-Generator* erzeugen. Dabei reicht es, die Grammatikregeln in BNF-Syntax zu definieren und sie dem Parser-Generator zu übergeben. Die Bottom-Up-Parser-Generatoren sind in der Lage, die oben genannten Mehrdeutigkeiten zu erkennen und sie durch das Einführen von Prioritäten sogar zu lösen.

Semantische Analyse Bei der semantischen Analyse wird die statische Semantik überprüft. Dabei werden die Typen von Konstanten bezüglich der auf sie angewendeten Operationen überprüft. Die Ausgabe der semantischen Analyse wird *dekorierter Syntaxbaum* gennant.

Zwischencode-Erzeugung Viele Compiler, die mehrere Quellsprachen oder verschiedene Zielplattformen unterstützen, erzeugen aus dem Syntaxbaum einen Zwischencode, der eine möglichst einfache Struktur hat. Dieser Zwischenschritt ist für die Codeoptimierung von besonderer Bedeutung. Außerdem eignet sich der Zwischencode auch als Austauschformat.

Optimierung In dieser Phase wird die Struktur der Zielmaschine zum ersten Mal miteinbezogen. Der Zwischencode wird in Hinsicht auf Laufzeitverhalten und Speicherbedarf verbessert.

Codeerzeugung Bei der Codeerzeugung wird das Programm in der Zielsprache generiert. Der Programmcode wird entweder aus dem Zwischencode oder direkt aus dem Syntaxbaum erzeugt. Der erzeugte Code kann entweder in Maschinen- bzw. Assemblercode oder in einer höheren Programmiersprache sein. Falls die Zielsprache eine Maschinensprache ist, kann das Ergebnis direkt ein ausführbares Programm sein.

Kapitel 3

Systembeschreibung

3.1 Architekturüberblick

In diesem Abschnitt werde ich den allgemeinen Aufbau eines COBOL-Programms beschreiben, das *DialogSystem* bzw. *DialogSys2Java* als Oberflächenbiblithek einsetzt. Hierbei werde ich die Grundkomponenten mittels eines *Multi-Tier*[1]- Architekturmodels veranschaulichen. Damit lassen sich die Komponenten des Systems und die Technik, durch die sie realisiert sind, darstellen.

3.1.1 Mehrschichtiges Architekturmodel

Das mehrschichtige Architekturmodel setzt die Betonung auf die logische Aufteilung des Gesamtsystems in funktionale Bausteine und nicht auf die Aufteilung in Hardware-Layer.

In der Praxis ist das Drei-Schichten-Architekturmodel *(three tier architecture model)* sehr verbreitet. Dabei wird die Applikation in Präsentations-, Geschäftslogik-

[1]Multi-Tier: Mehrschichtiges Model

und Datenhaltungsschicht eingeteilt. Die Aufgabe der Präsentationsschicht besteht darin, die grafische Benutzeroberfläche (graphical user interface – GUI) zur Ansicht zu bringen und Interaktionen zu ermöglichen [FWER]. Die Geschäftslogikschicht übernimmt die Kernfunktionen des Systems, nämlich die Wiedergabe der fachlichen Vorgänge. Die Dritte Schicht, die Datenhaltung, ist für die dauerhafte Speicherung der Objekte zuständig.

Ich werde noch zwei weitere Schichten heranziehen, indem ich die Datenhaltung und die Präsentation weiter aufteile. Innerhalb der Datenhaltungsschicht werde ich eine Unterscheidung zwischen Datenverwaltung und Datenhaltung treffen. Innerhalb der Präsentationsschicht werden die Präsentationslogik (Prozesslogik) und Präsentation getrennt betrachtet (s. Abb. 3.1).

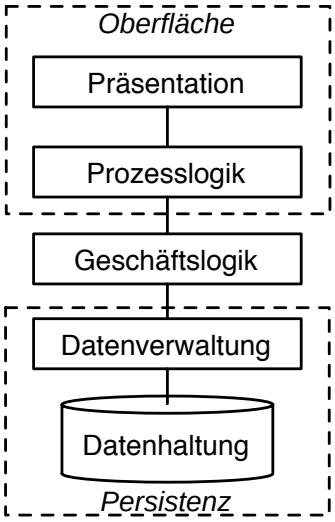

Abbildung 3.1: Allgemeine Darstellung einer Fünf-Schichten-Architektur

Dieses Architekturmodel eignet sich besonders gut zum Ausdrücken der Basiskomponenten einer auf DialogSystem basierenden COBOL-Anwendung. Im Folgenden wird diese Realisierung des Models verdeutlicht.

Da sich diese Arbeit im Wesentlichen mit der obersten Schicht des Models (der Präsentation) befasst, werde ich "die unteren" Schichten nicht in allen Einzelheiten ausführen.

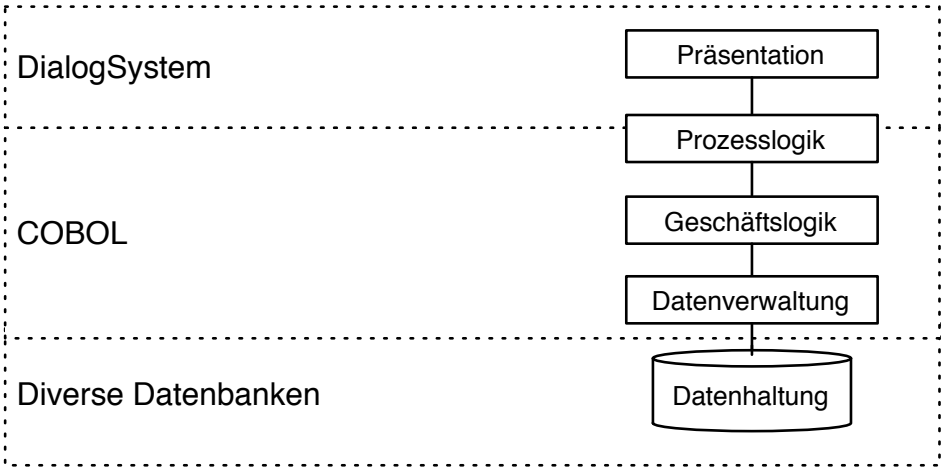

Abbildung 3.2: Architektur einer COBOL-Anwendung

Die Präsentationschicht wird mit DialogSystem vollständig abgebildet. DialogSystem ermöglicht eine klare Trennung zwischen der grafischen Benutzerschnittstelle und der Programm-Verarbeitung (Prozesslogik), wobei die Steuerung im Dialog über eine eigene Skriptsprache in der grafischen Oberfläche selbst erfolgen kann.

Die dialogübergreifende Anwendungslogik *(Workflow)*, die Geschäftslogik sowie die Datenverwaltung sind üblicherweise in der Programiersprache COBOL implementiert.

Die Vielfalt der COBOL-Datenbankschnittstellen verschiedener Hersteller bietet eine breite Auswahl von relationalen Datenbanken. Eine Beschreibung dieser Datenbanken würde jedoch den Rahmen der Arbeit sprengen. Daher gehe ich nicht darauf ein.

3.2 Das Ausgangssystem *DialogSystem*

Bei *DialogSystem* handelt es sich um eine API *(Application Programming Interface)* zum Programmieren von grafischen Benutzeroberflächen für COBOL-Anwendungen (s. Abbildung 3.3) und eine Laufzeitumgebung (GUI-Runtime), die die Oberflächen interpretiert. DialogSystem ist ein fester Bestandteil des Micro Focus COBOL-Werkzeuges NetExpress. Die graphischen Oberflächen lassen sich ausschließlich mit NetExpress entwerfen.

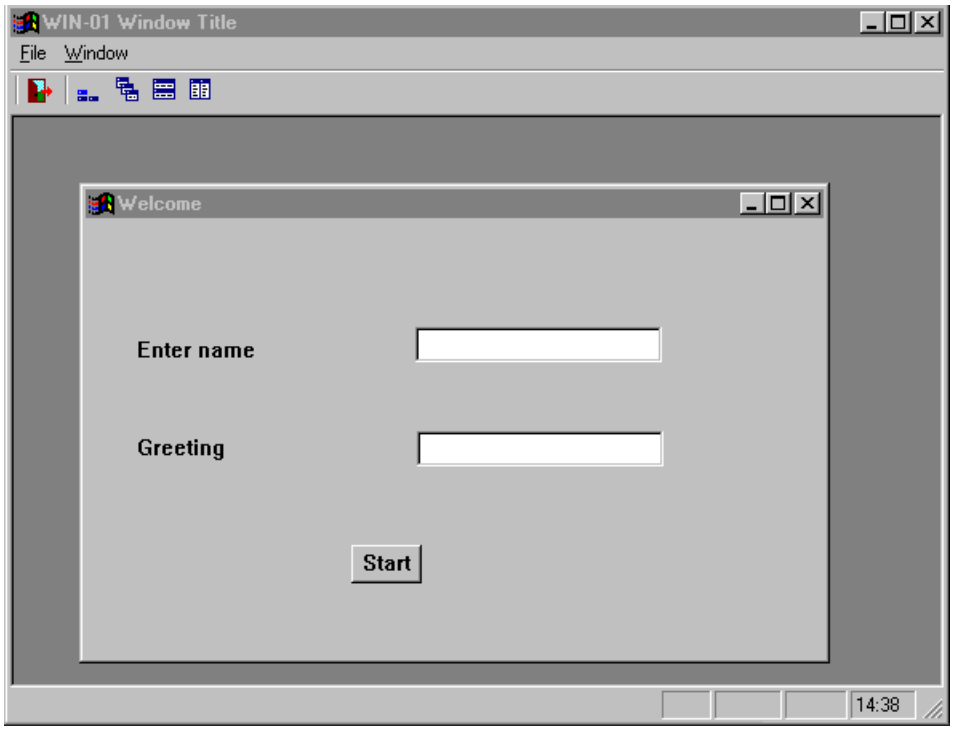

Abbildung 3.3: Beispiel einer in DialogSystem erstellten Benutzeroberfläche. *Quelle: www.microfocus.com*

3.2.1 Die Screensets

Ein *ScreenSet* ist eine Hierarchie von Formularen. Zu einem Screenset gehören:

- Ein oder mehrere Fenster: Unter den Fenstern des Screensets besteht eine hierarchische *Parent/Child*-Beziehung. Ein Screenset hat ein oder mehrere Einstiegsfenster (Hauptfenster), von denen aus alle andere Fenster und Dialogen im Screenset erreichbar sind. Das Öffnen eines Fensters impliziert das Öffnen seiner Parent-Fenster.

- Prozeduren zur Ereignisverarbeitung: Dabei handelt sich um anwendungsspezifische Methoden, die zum Abfangen und Verarbeiten der Benutzerereignisse dienen.

- Datenblock: Der Datenblock *(DS-DATA-BLOCK)* beinhaltet die Fachdaten, die die GUI-Elemente des Screensets benötigen (z.B. Inhalt eines Textfeldes). DS-DATA-BLOCK ist auch seitens der COBOL-Anwendung bekannt.

3.2.2 DSGRun-Interface

Aus der COBOL-Anwendung aus lässt sich auf die ScreenSets ausschließlich über der sog. DSGRun-Interface (kurz DSGRUN) zugreifen. Die DSGRun-Interface ist ein Teil der DialogSystem-Laufzeitungebung. Die Kommunikation zwischen COBOL und DialogSystem wird über das DS[2]-Protokoll abgewickelt. Das Protokoll besteht aus:

1. DS-CTRL-BLOCK: Enthält Informationen zur Dialogsteuerung (Steuerinformationen).

[2]DS: DialogSystem

2. DS-DATA-BLOCK: Beinhaltet die Fachdaten des aktuellen ScreenSets (s. Abschnitt 3.2.1)

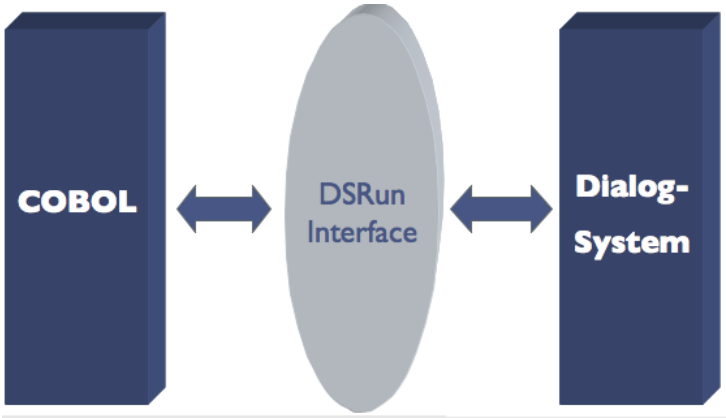

Abbildung 3.4: DSGRun-Interface *Quelle: [DSJ]*

3.3 Das Ersatzsystem *DialogSys2Java*

Bei der DialogSys2Java handelt es sich um einen vollwertigen Java-basierten DialogSystem-Ersatz für COBOL Anwendungen. DialogSys2Java verhält sich dabei an den Schnittstellen (DSGRUN) wie MicroFocus DialogSystem.

Ersatz des ScreenSet-Konzepts Die drei ScreenSet-Komponenten werden als Java-Klassen wie folg realisiert:

- Fenster: Jedes Fenster wird in einer eigenständigen, von *JFrame* oder *JDialog* abgeleiteten Klasse, dargestellt.
- Die Instanzierung und Referenzierung der Fenster sowie die Verarbeitung der globalen Benutzerereignisse erfolgt durch einen für jedes Set notwendigen ScreenSet-Kontroller (Oberklasse: *DSControler*).

- Fachdaten: Die Fachdaten eines ScreenSets werden in einer eigenen Datenklasse abgelegt.

Abbildung 3.5 veranschaulicht den Aufbau eines ScreenSets in DialogSys2Java.

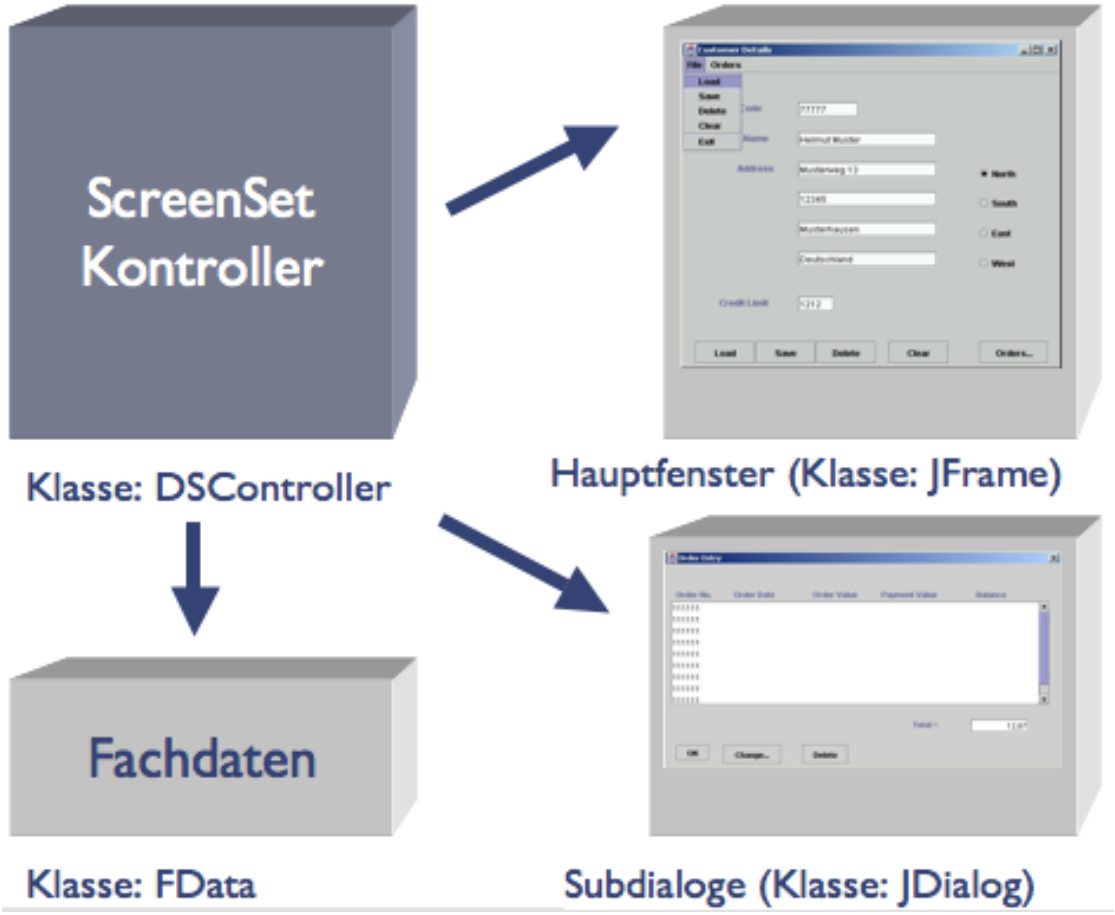

Abbildung 3.5: Die ScreenSet-Architektur in DialogSys2Java *Quelle: [DSJ]*

3.4 Der Screenset-Compiler

Die Aufgabe des Screenset-Compilers (Umsetzers) ist, DialogSystem-Screensets in DialogSys2Java-Screensets maschinell zu überführen (s. Abbildung 3.6).

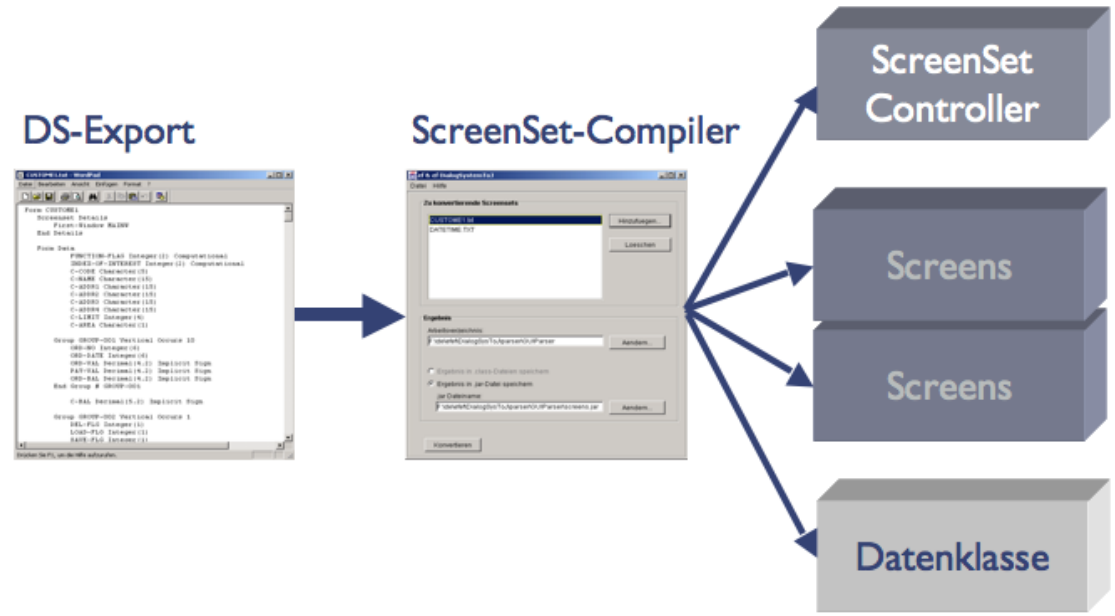

Abbildung 3.6: Der ScreenSet-Compiler *Quelle: [DSJ]*

Der Umsetzer verwendet das Exportformat von Dialog System als Input. Durch eine Single-Pass-Konvertierung wird sowohl der GUI als auch die Ablauflogik und der Datablock umgesetzt. Die umgesetzten Fenster können mit Standardwerkzeugen wie z.B. *VisualEditor* oder *Borland JBuilder* bearbeitet werden.

3.4.1 Aufbau

Im Abschnitt 2.3.3 *(Grundlagen)* habe ich den allgemeinen Aufbau eines Compilers beschrieben. An dieser Stelle werde ich die konkrete Realisierung der Compiler-

Bausteine darlegen.

Lexikale Analyse Der lexikale Scanner in DialogSys2Java wird mithilfe des Scanner-Generators *JFLEX*[3] erzeugt. Der Scanner *(lexer)* wird anhand einer .flex-Datei generiert (hier dialogsys2j.flex). In dieser Datei wird festgelegt, wie die generierte Scanner-Klasse heißen soll, wie die Zahlen in der Eingabedatei dargestellt sind usw. In dieser Implementierung wird die Scanner-Klasse durch die Klasse DSScanner vertreten. In DialogSys2Java wird der *lexer* zusammen mit einem Parser verwendet. In diesem Fall werden die Tokens weiter an den Parser gegeben. Für eine ausführliche Beschreibung der lexikalen Grammatik sehen Sie Abschnitt 3.4.2. (vgl. [DSJQ])

Syntaktische Analyse und Codeerzeugung Der Parser in DialogSy2Java wird mithilfe des Parser-Generators *Java-CUP*[4] *(auch CUP)* generiert. Dieser Parser-Generator erzeugt Parser für kontextfreie Grammatiken des Typs *LALR*.

Die Aufgabe des Parsers ist es zu prüfen, ob die Syntax des Eingabetextes der Grammatik des Exportformats *(GMDL*[5]*)* entspricht. Dabei wird nach jeder Produktion (oder Reduktion) eine Aktion eingebaut. Im Fall von DialogSys2J wird beim Parsen eine Java-Klassen-Hierarchie aufgebaut, aus der dann die Sourcen für ScreenSet-Klassen generiert werden *(Codeerzeugung)*.

Für das Parsen wird ein Scanner benötigt (s. oben), der das zu parsende Dokument in Form von Tokens an den Parser weiter gibt. Die Grammatik, die der Parser unterstützt, wird in einer .cap-Datei beschrieben (in diesem Fall heißt diese Datei dialogsys2j.cap). Anhand dieser Datei legt der CUP-Parsergenerator eini-

[3]JFLEX: The Fast Scanner Generator for Java; URL: http://www.jflex.de/

[4]CUP: LALR Parser Generator in Java; URL: http://www2.cs.tum.edu/projects/cup/

[5]GMDL: graphical mode definition language

ge Klassen an, die dann nur noch an die jeweilige Applikation angepasst werden sollen. Die bedeutendste davon ist die Klasse *parser*.

Die Klasse *parser* erweitert die abstrakte Klasse *lr_parser* des *Cup-Interfaces*. Eine Innereklasse *CUP.parser.actions* gibt die Möglichkeit, benutzerdefinierte Aktionen während des Parsens einzubauen. Dazu steht die callback-Methode *do_action(..)* zur Verfügung. Die Zusammenarbeit zwischen Parser und Scanner veranschaulicht Abbildung 3.7. (vgl. [DSJQ])

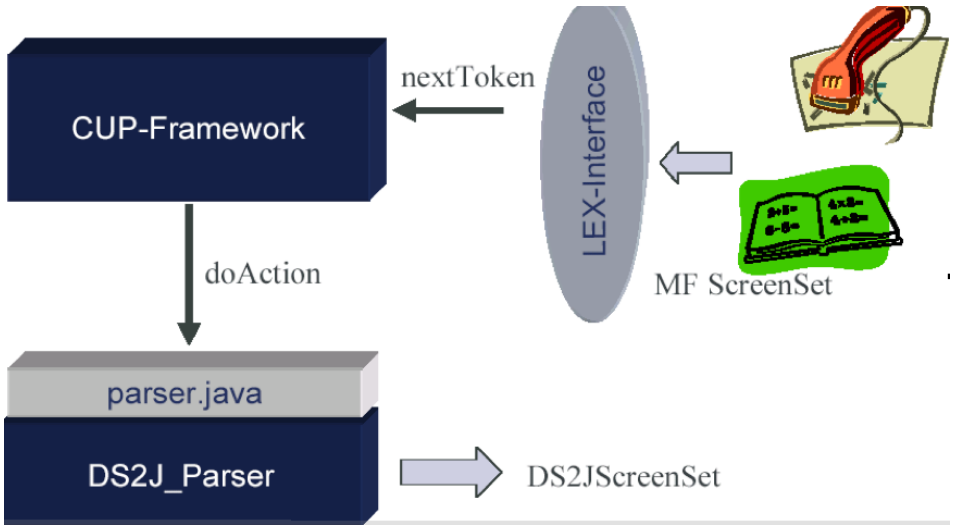

Abbildung 3.7: Zusammenarbeit zw. CUP-Parser und JFLEX-Scanner *Quelle: [DSJQ]*

Für eine ausführliche Beschreibung der GMDL-Grammatik sehen Sie Abschnitt 3.4.2.

3.4.2 Formale Beschreibung der Quellsprache

In diesem Abschnitt setze ich mich mit der Quellsprache des Screenset-Compilers und insbesondere mit ihrer Grammatik auseinander. Dabei handelt es sich um die Syntax des Import/Export-Formats des Dialog Systems *graphical mode definition language (GMDL)*. Zunächst werde ich die Sprache auf *syntaktischer*- und dann auf *lexikalischer* Ebene analysieren. Hierbei wird der Akzent auf die syntaktische Beschreibung gelegt.

GMDL-Syntax Die Grammatik der Sprache *GMDL* bezeichne ich mit G_{DS}. Es gilt dabei: $GMDL = L(G_{DS})$.

G_{DS} lässt sich formal wie folgt darstellen:

$$G_{DS} = (N_{DS}, T_{DS}, P_{DS}, S_{DS}),$$

wobei N_{DS} die Nichtterminalsymbole, T_{DS} die Trminalsymbole, P_{DS} die Menge aller Produktionen und S_{DS} die Startsymbolmenge über G_{DS} ist.

Die Produktionsregeln der Grammatik P_{DS} lassen sich aus den vorliegenden Syntaxgraphen (s. Abb. 3.8 und 3.9) ableiten.

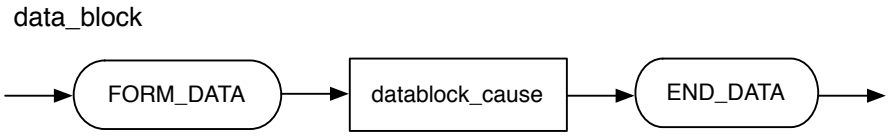

Abbildung 3.8: Syntaxgraph zu dem Nichtterminal *datablock*

Die Syntaxgraphen in den Abbildungen 3.8 und 3.9 stellen nur einen Beispielauszug dar. Den vollständigen Syntaxgraphen-Katalog bietet das *Dialog System Referenzhandbuch* [MFDS].

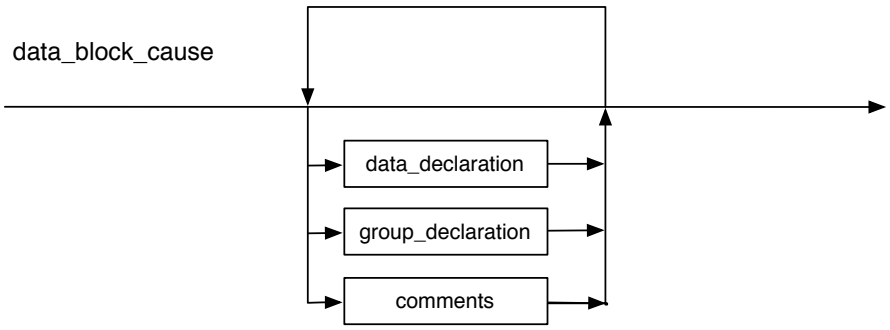

Abbildung 3.9: Syntaxgraph zu dem Nichtterminal *datablock-cause*

Als Syntaxform für die Produktionsregelnbeschreibung setze ich die erweiterte *Backus-Naur-Form (EBNF)* (s. Kapitel 2.3.2) ein. Aus Gründen der Übersichtlichkeit werden einige Modifikationen im Vergleich zu der im Kapitel 2.3.2 vorgestellten Notation eingeführt. Diese Abweichungen beeinflussen nicht die Ausdrucksmächtigkeit der *EBNF*.

- *A1:* Spitzenklammer-Zeichen entfallen bei der Nichtterminal-Notation.
- *A2:* Nichtterminalsymbole werden komplett mit Kleinbuchstaben notiert *(lowercase word)*.
- *A3:* Terminalsymbole, soweit sie keine Spezialzeichen sind, werden komplett mit Grossbuchstaben *(UPPER CASE WORD)* aufgeschrieben.
- *A4:* Spezialzeichen-Terminalsymbole werden durch Hochkomma angegeben (z.B. ';' , '*' oder '+').

Analyse der Grammatik G_{DS} Die Nichtterminalmenge N_{DS} besteht aus **58** Elementen. Zu jedem $n \in N_{DS}$ existiert ein Syntaxgraph wie in Abb. 3.8 bzw. 3.9 dargestellt. Aus jedem Syntaxgraphen lässt sich mindestens eine Produktionsregel $p \in P_{DS}$ ableiten. Es gilt also $|N_{DS}| \leq |P_{DS}|$.

Listing 3.1 Einige Produktionsregeln in *EBNF*-Notation

```
screenset ::= {form_def}
form_def ::=
        form_internals |
        FORM screenset_id screenset END FORM
form_internals ::=
        datablock | font_record |
        global_dialog | menu_bar | object |
        object_dialog | screenset_details | validation
datablock ::= FORM DATA datablock_clause END DATA
datablock_clause ::=
        {data_declaration | group_declaration | comment}
data_declaration ::= [ data_name data_dclr ]
data_dclr ::=
        CHARACTER [ERROR-FIELD] |
        ALPHABETIC |
        INTEGER [IMPLICIT SIGN | COMP | COMP-5]
        DDECIMAL [IMPLICIT SIGN]
group_declaration ::=
        GROUP group_name VERTICAL OCCURS n
        data_declaration {data_declaration} END GROUP
```

Die Produktionsmenge P_{DS}, in *EBNF*-Notation, besteht aus **64** Produktionsregeln. Dem oben dargestellten Listing 3.1 ist nur ein Ausschnitt zu entnehmen. Eine detaillierte Auflistung befindet sich im Anhang A.1.

Aus Anhang A.1 wird noch Folgendes deutlich:

Für alle Produktionsregeln $p \in P_{DS}$ gilt: $p : N \rightarrow \mu$, wobei $\mu \in (N \cup T)^*$ ist. Daraus folgt, dass G_{DS} eine *kontextfreie Grammatik* (Typ 2) ist (s. Kap. 2.3.1). Formal gilt:

$$GMDL \subset L(G_2) \text{ und } GMDL \not\subset L(G_3)$$

Das Nonterminalsymbol *screenset* ist das Startsymbol in G_{DS}. Es gilt also $S_{DS} = screenset$.

GMDL-Lexik Um die Sprache *GMDL* auf lexikaler Ebene zu beschreiben, reicht es ihre Tokenklassen festzulegen. Dabei werde ich jede Tokenklasse durch einen regulären Ausdruck definieren. Aus diesem Grund müssen alle Wörter, die zu einer Tokenklasse gehören, eine reguläre Sprache bilden.

Listing 3.2 (*GMDL*-Tokenklassen)

```
<LineTerminator> = \r|\n|\r\n
<InputCharacter> = [^\r\n]
<StringCharacter> = [^\r\n\""\\]
<SingleCharacter>  = [^\r\n\''\\]
<WhiteSpace> = {LineTerminator} | [ \t\f]
<Letter> = [A –Z] | [a – z]
<Digit> = [0 – 9]
<IntegerLiteral> = <Digit><Digit>*
<OctDigit> = [0–7]
<FloatLiteral> = <Digit>* \. <Digit>+
<Identifier> =
        <Letter> | ([\@] | [<Letter>])
```

```
        (([<Digit>|<Letter>] | [-])* [<Digit>|<Letter>])
<NumberedIdentifier> =
        ([\@] | [<Digit>|<Letter>])
        (([<Digit>|<Letter>] | [-])* [<Digit>|<Letter>])
<Comment> = \* {<InputCharacter>}*
<Alphabetic> = ALPHABETIC\(<IntegerLiteral>\)
<Character> = CHARACTER\(<IntegerLiteral>\)
<Integer> =  INTEGER\(<IntegerLiteral>\)
<Decimal> = DECIMAL\(<FloatLiteral>\)
<Size> = SIZE\(<IntegerLiteral>\,<IntegerLiteral>\)
```

Listing 3.2 bietet eine Darstellung der Tokenklassen zu $GMDL$, die sich mithilfe nichttrivialer regulärer Ausdrücke herleiten lassen. Zu der vollständigen Tokenklassenliste gehört eine Reihe von reservierten Wörtern wie $INTEGER$ und $DECIMAL$, die jeweils für sich eine Tokenklasse bilden. (s. Anhang A.3).

Bemerkungen zu der eingesetzten *REGEXP*-Notation: Die in dem Listing 3.2 bzw. im Anhang A.2 benutzte Notation bezieht sich auf einige Sonderoperatoren und Quantoren. Die Bedeutung dieser Sonderzeichen werde ich im Folgenden kurz erläutern.

- $\backslash x$ stellt ein zitiertes Zeichen x dar.
- x^* : Das Zeichen x kann null oder mehrmals auftreten.
- x^+ : Das Zeichen x kann ein oder mehrmals auftreten.
- $x?$: Das Zeichen x kann null oder einmal auftreten.
- [^x] stellt ein beliebiges Zeichen außer x dar. (Negation)

Kapitel 4

Anforderungen und Konzept

Ziel dieser Arbeit ist, die innere Programmstruktur des Systems zu verbessern *(Refactoring)*. Diese Verbesserung besteht vor allem aus einer Serie kleiner Modifikationen des Designs (Refactoring-Schritten). Um diese Schritte festzulegen und sie systematisch durchzuführen, gilt es zunächst, die Anforderungen zu erfassen.

4.1 Der Migrationsprozess

Die Integration der Ersatzlaufzeitumgebung in eine bestehende *Legacy*-Umgebung ist keine einfache Aufgabe. *DialogSys2Java* ist keine universelle Lösung, die sich für jede Kundensituation generisch einsetzen lässt. Daher definiert frobese GmbH einen Geschäftsprozess, den *DialogSystem-Migrationsprozess*, der die Schritte der Migration festlegt. Abbildung 4.1 veranschaulicht den Prozessverlauf. Im Folgenden werde ich die Prozessschritte kurz erläutern.

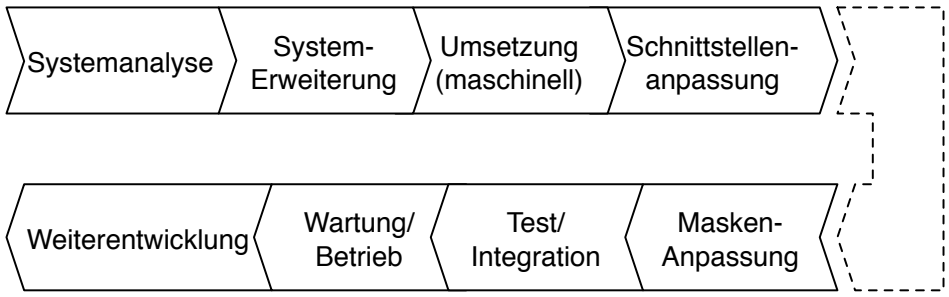

Abbildung 4.1: Migrationsablauf - Prozessschritte

Systemanalyse (ANL) Ziel der Migration ist
alte (legacy) DialogSystem-Anwendungen mit der neuen Laufzeitumgebung verträglich zu machen. Dabei soll die Altanwendung "am Leben erhalten" bleiben. Dies bedeutet, dass aus der Sicht des Anwenders keine Unterschiede zu "alt" feststellbar sein müssen, die eine Nacharbeit seitens des Auftraggebers verlangen würden. Um diese "eins zu eins"-Umstellung zu gewährleisten, ist es nötig, die interne Struktur der umzustellenden Anwendung detailliert zu analysieren. Das Augenmerk wird hierbei auf folgende Punkte gerichtet:

- Es wird geprüft, ob es Abhängigkeiten zu fremd GUI-Systeme bestehen, die die Umstellung erschweren könnten. So eine Abhängigkeit könnte z.B. die Mischnutzung zweier voneinander unabhängiger Oberflächenbibliotheken sein. Eine solche Kombination ist die Verwendung des *DialogSystems* zusammen mit *PANELS V2*[1]

- Die Umstellung erfordert eine Anpassung aller Schnittstellen der Altanwendung an die Oberflächenbibliothek. Es soll analysiert werden, wie gut diese Schnittstellen gekapselt sind. Keine oder schwache Kapselung erfordert zum

[1]PANELS V2: Alte GUI-Framework der Firma Merant.

einen einen höheren Programmieraufwand und zum anderen massive Angriffe in die Workflowschicht der Software, die das Fehlerrisiko erhöhen.

- Da der Maskenumsetzer nicht die volle DialogSystem-Spezifikation unterstützt, wird die genutzte Menge in der Anwendung analysiert. Darauf basierend wird der Aufwand für Systemerweiterungen geschätzt.

Systemerweiterung (ERW) Die volle *DialogSystem*-Spezifikation besteht aus mehreren hundert Funktionen und Befehlen. In den meisten kaufmännischen Anwendungen werden jedoch nur bestimmte Funktionen, die circa 50% der Gesamtfunktionalität ausmachen eingesetzt. Daher wurden vorerst nur diese Funktionen in die Ersatzoberflächenbibliotek implementiert, um unnötigen Entwicklungsaufwand zu sparen.

In der Phase der Systemanalyse wurde festgestellt, um welche anwendungsspezifische Funktionalität die Ersatzbibliothek erweitert werden soll. In der Phase der Systemerweiterung werden diese Anforderungen realisiert.

Maschinelle Umsetzung (UMS) Hier werden mit Hilfe des Maskenumsetzers alle DialogSystem-Masken der Anwendung in JAVA umgesetzt. Anschließend werden die ersten Vorab-Ergebnisse dem Auftraggeber vorgestellt.

Schnittstellenanpassung (SCH) Um die neue Laufzeitumgebung zu integrieren, sind die Schnittstellen der Altanwendung anzupassen. Dabei ist eine Durchführung (in der Regel leichter) Code-Angriffe in der COBOL-Anwendung nötig. Die Abweichungen der GUI-Schnittstellen gegenüber diesen von *DialogSystem* charakterisieren sich durch:

- Unterschiedliche Namenskonventionen
- Zusätzliche Aufrufparameter
- Zusätzliche Aufrufverarbeitungsroutinen

Manuelle Maskenanpassung (MAN) Nach der generischen Umsetzung erweisen die produzierten Oberflächen einige Unstimmigkeiten mit den Originalmasken. In diesem Fall sind manuelle Anpassungen erforderlich. Die Anpassungen werden mit einem Ist/Soll Vergleich begleitend durchgeführt.

Test/Integration (TES) In dieser Phase werden die neu erzeugten Masken (Screensets) im "echten" Betrieb getestet. Zuerst wird eine laufende Anwendung als Abnahmeumgebung seitens des Auftraggebers zur Verfügung gestellt. In dieser Umgebung wird die Laufzeitumgebung des DialogSystems durch DialogSys2Java und alle Anwendungsmasken durch die neu erzeugten JAVA-Masken ersetzt. Ferner stellt der Auftraggeber Testfälle zur Verfügung. Die Testfälle sollen eine ausreichende GUI-Überdeckung erreichen, um damit den Erfolg der Migration zu garantieren.

Die Testphase verläuft abhängig vom Überdeckungsgrad der Testfälle in drei Schritten.

1. *Maskenüberdeckung:* Während des Testens wird versucht, jede Maske anzuzeigen. Das ist die schwächste Überdeckung.
2. *GUI-Elementen-Überdeckung:* Dieser Schritt garantiert eine schärfere Überdeckung. Hier wird versucht, alle GUI-Ellemente zu aktivieren[2].

[2](z.B. Buttons: Alle Buttons müssen angeklickt werden; Textfelder: alle Textfelder müssen mit Inhalten befüllt werden)

3. *Überdeckung in der Struktur:* Dieser Test stellt die höchste Testschärfe dar. Hierbei muss jede Anweisung im Code angesprochen werden (Glass Box Test). Da das Erreichen dieser Überdeckung oft sehr aufwändig ist, wird sie zunächst nicht angestrebt.

Während des Testens wird versucht, eine möglichst vollständige Überdeckung in allen drei Schritten zu erreichen. Der Erfolg hängt allerdings von der Qualität der zur Verfügung gestellten Testfälle. Erfolgreich durchgeführten Tests setzen die Abnahmevoraussetzungen fest.

Betrieb und Wartung (BEW) Nach einer erfolgreichen Abnahme setzt der Auftraggeber die umgestellte Software in Betrieb. Ab diesem Zeitpunkt fängt die Gewährleistungsperiode seitens Firma frobese an. Dies bedeutet, dass während des Betriebs auftretende Fehler kurzfristig zu beheben sind. Darüber hinaus werden Wartungsarbeiten in Betrieb durchgeführt.

Weiterentwicklung (WEN) Bei der Weiterentwicklung handelt sich um den Prozess der Weiterentwicklung des umgestellten Programms. Zum einen werden in dieser Phase umgestellte Oberflächen modifiziert, und zum anderen für neu entwickelte Programmkomponenten wird die Präsentationschicht komplett mittels DialogSys2Java-GUI-Frameworks neu realisiert.

4.2 Erfassen der Rohanforderungen

Um die Rohanforderungen zu erfassen, benötigt man gute Anforderungsquellen. Diese zu finden, ist eine wichtige Aufgabe innerhalb der Anforderungsanalyse. Im folgenden Abschnitt werde ich die Anforderungsquellen bestimmen und die Anforderungen für den Refactoring festhalten.

Im vorherigen Abschnitt habe ich den Migrationsprozess ausführlich beschrieben. Ich betrachte hier die einzelnen Prozessschritte als gute Anforderungsquellen. Zu jedem Schritt werde ich Anforderungen festhalten, soweit es solche gibt, und sie anschließend mit einer Priorität versehen. Um die Anforderungen quantitativ zu betrachten, werde ich an dieser Stelle Prozessmetriken einsetzen. Nach der Abarbeitung der Anforderungen soll schließlich eine Prozessoptimierung erreicht werden.

Anforderungen aus der Systemanalyse Zu diesem Prozessschritt werden zuerst keine Anforderungen erfasst. Durch die eventuelle Optimierung einer der nächsten Prozessschritte ist jedoch möglich, dass einige Vorbereitungen in der Systemanalyse auf trivialer Art entfallen.

Anforderungen aus der Phase der Systemerweiterung Die Aufgabe dieses Schrittes war, das System um die erforderliche Funktionalität zu erweitern. Es stellt sich heraus, dass der an dieser Stelle benötigte Aufwand für den Einbau neuer Funktionen zu hoch ist. Es lässt sich sogar eine weiterhin steigende Tendenz beobachten.

	Neuentwicklung	Letzte Migration	jetziger Stand
PT/Erweiterung	$\frac{1}{10}$	$\frac{1}{3}$	$\frac{5}{4}$

Abbildung 4.2: Vergleich Aufwände in PT/Erweiterung

Dieses Erkenntnis lässt sich aus der Gegenüberstellung der Aufwände pro Funktionseinheit bei der Neuentwicklung, bei früheren Projekten und im jetzigen Stand ableiten. Der Tabelle 4.2 lässt sich der Vergleich in Personentage pro Erweiterung entnehmen.

Anforderung (A1): In dieser Phase soll eine konstante Entwicklungszeit pro Systemerweiterung im Durchschnitt garantiert werden.
Priorität: hoch

Anforderungen aus der maschinellen Umsetzung Aufgabe der maschinellen Umsetzung ist, alle DialogSystem-Masken in JAVA umzusetzen. Es ist dabei nicht klar, ob der eingesetzten Compiler durch sein Parsingsverfahren in der Lage ist, vollständig die Sprache des Exportformats des DialogSystems zu verarbeiten.

Anforderung (A2): Die Grammatik des Exportformats soll formal beschrieben werden. Der eingesetzte Parser soll bis zu einer vollständigen Abdeckung der Grammatik erweitert werden.
Priorität: mittel

Anforderungen aus der Schnittstellenanpassung Die Schnittstellenanpassung bedeutet oft einen Angriff in der Workflowschicht der Legacy-Anwendung. Diese Aktion wird als kritisch seitens des Auftraggebers empfunden, da Programmsourcen zur Verfügung gestellt werden müssen. Aus diesem Grund scheitern viele potenzielle Migrationsprojekte. Je nach Kapselungsgrad der Schnittstellen in der Legacy-Anwendung fällt der Aufwand in dieser Phase seitens Fa. Frobese unter-

schiedlich hoch aus. Die Gegenüberstellung "Aufwand in der Schnittstellenanpassung vs. Gesamtprojektaufwand" ($\frac{\mathbf{AUFW-SCHN(PT)}}{\mathbf{AUFW-GES(PT)}}$) ergibt Werte zwischen $\frac{\mathbf{1}}{\mathbf{20}}$ und $\frac{\mathbf{1}}{\mathbf{5}}$.

Anforderung (A3): Es soll eine Lösung gefunden werden, die die Schnittstellenanpassungen innerhalb des Kundencodes möglichst minimiert. Nach Möglichkeit soll die Schnittstellenanpassung völlig entfallen.
Priorität: hoch

Anforderungen aus der Maskenanpassung Bei der Maskenanpassung werden die Masken auf Unterschiede im Vergleich zum Original überprüft und manuell angepasst. Jedoch weisen bei komplexeren Masken die generierten zu viele Abweichungen auf.
Messung: Eingesetzte Metrik ist: $AQ = \frac{\mathbf{ABW(M)}}{\mathbf{GUI-C(M)}}$. Dabei bedeutet GUI-C(M) die Anzahl der GUI-Elemente in einer Maske (auch GUI-Komplexität), und ABW(M) die aufgetretenen Abweichungen in einer Maske.

Die Messungen habe ich statistisch durchgeführt. Als Stichprobenraum habe ich einen Altprojekt-Maskenbestand mit 730 Masken unterschiedlicher Komplexität benutzt. Die Masken habe ich in drei Komplexitätsklassen unterteilt:
GUI-C < 5, 5 <= GUI-C < 15 und GUI-C >= 15.

	GUI-C < 5	5 <= GUI-C < 15	GUI-C >= 15
AQ	$\mathbf{0,1}$	$\mathbf{0,15}$	$\mathbf{0,26}$

Abbildung 4.3: GUI-Abweichungen in Masken unterschiedlicher Komplexitätsklassen

Anforderung (A4): Der Maskengenerator soll verbessert werden, um den manuellen Nacharbeitungsaufwand bei komplexeren Masken zu minimieren.
Priorität: niedrig

Anforderungen aus der Testphase Zu der Testphase sollten zuerst gute Testfälle erzeugt werden, die eine ausreichende Überdeckung garantieren. Es wurde festgestellt, dass allein der Aufwand für das Erzeugen der Testfälle gegenüber dem Gesamtprojektaufwand zu hoch ist. Als Referenz habe ich Projektdaten der zuletzt durchgeführten Migration mit einem Umfang von ca. 800000 LOC[3] in den Maskendefinitionen herangezogen. Daraus ergibt sich: $\frac{\textbf{Aufwand für Testfallerzeugen}}{\textbf{Gesamtaufwand}} \approx \frac{\mathbf{1}}{\mathbf{4}}$

Anforderung (A5): Der Aufwand für das Erzeugen von Testfällen, soll reduziert werden.
Priorität: mittel

Anforderungen aus dem Betrieb Die in der Produktion aufgetretenen Fehler sind schnell zu beheben. Hier dürfen allerdings keine Fehler auftreten, da das Produkt vorher vollständig getestet wurde. In der Praxis ist dies jedoch nicht der Fall. Als Referenz habe ich eine Altprojekt-Fehlerdatenbank mit ca. 320 Fehlern benutzt, die ausschließlich im Betrieb aufgetreten sind. Die resultierende Fehlerrate $\frac{\textbf{Fehleranzahl}}{KLOC}$ ergibt also $\frac{320}{800} = 0,475$.

Anforderung (A6): Die Fehlerrate sowie der Aufwand für Fehlerbeseitigung im Betrieb sollen minimiert werden.
Priorität: hoch

[3]LOC: lines of code

Anforderungen aus der Weiterentwicklung In dieser Phase werden vor allem bestehende Programmteile erweitert. Dabei müssen sich meist COBOL- und DialogSystem-Entwickler zunächst in die neue DialogSys2Java-Umgebung einarbeiten und dann die von ihnen erarbeiteten DialogSystem-Programmteile in JAVA weiterentwickeln. Es wurde allerdings festgestellt, dass das Wiederkennen und Verstehen des eigenen Codes in den umgesetzten Masken sehr aufwendig ist.

Anforderung (A7): Der Einarbeitungsaufwand in DialogSys2Java muss minimiert werden. Das Wiedererkennen und Verstehen von umgesetzten Programmabschnitte, muss ohne viel Aufwand möglich sein.
Priorität: hoch

4.3 Plan des Refactorings

In diesem Abschnitt werde ich die Schritte des Refactorings aus den oben definierten Rohanforderungen und ihrer Priorisierung ableiten. Um diese Schritte optimal zu definieren, sind zunächst folgende Fragen zu beantworten:

Welcher Zusammenhang besteht zwischen Anforderung und Systemkomponente?

Wie hängen Systemkomponenten von einander ab?

Zuordnung der Systemkomponenten Aus der Phasen des Geschäftsprozesses lassen sich Anforderungen ableiten. Um jeder Rohanforderung eine Systemkomponente zuzuordnen, reicht es die in jedem Prozessschritt betroffenen Systemkomponenten zu bestimmen. Der nachstehenden Tabelle 4.4 ist diese Zuordnung zu entnehmen.

	Runtime	Umsetzer	Gen. Code	Framework	Kundencode
Systemanalyse					•
Systemerweiterung	•	•			
Umsetzung (maschinell)		•	•		
Schnittstellenanpassung	•				•
Maskenanpassung			•		
Test und Integration	•		•		
Betrieb und Wartung	•		•		
Weiterentwicklung			•	•	•

Abbildung 4.4: Betroffene Systemkomponenten

Abhängigkeiten Die einzelnen Systemkomponenten in DialogSys2Java hängen programmiertechnisch von einander ab. Das *GUI-Framework* zum Beispiel stellt dem Anwendungsentwickler lediglich Funktionsmerkmale zur Verfügung, die die Laufzeitumgebung interpretieren kann. Dies bedeutet, dass eine Erweiterung der Funktionalität des Frameworks auch in die Runtime nachgezogen werden soll. Demzufolge spielen die Abhängigkeiten eine wesentliche Rolle bei der Festlegung der Reihenfolge der Refactoringsschritte. Die Abhängigkeiten unter den wichtigsten Systemkomponenten in DialogSys2Java veranschaulicht Abb. 4.5.

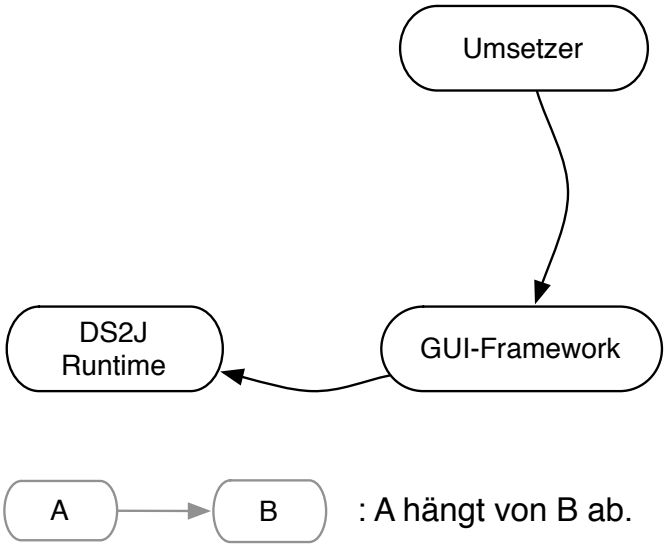

Abbildung 4.5: Abhängigkeiten unter den Systemkomponenten.

Festlegen der Refactoringsschritte Die Refactoringsschritte werde ich nach dem folgenden Prinzip festlegen:

- Hochpriorisierte Anforderungen werden zuerst abgearbeitet.
- Mehrere Anforderungen, die eine Systemkomponente betreffen, werden nach Möglichkeit zusammen abgearbeitet.
- Wenn Komponente A von Komponente B abhängt, wird zunächst die Anforderung, die B, und dann die Anforderung, die A betrifft, abgearbeitet.

Aus dieser Überlegung lege ich den folgenden Plan des Refactorings fest:

Schritt Nr.	Anforderung	Priorität	Komponente
1	A6	hoch	Runtime, Gen. Code
2	A1	hoch	Runtime, Umsetzer
3	A7	hoch	Gen. Code, Framework, Kunden-Code
4	A3	hoch	Runtime, Kunden-Code
5	A5	mittel	Runtime, Gen. Code
6	A2	mittel	Umsetzer, Gen. Code
7	A4	niedrig	Gen. Code

Abbildung 4.6: Plan des Refactorings

Kapitel 5

Realisierung

Im vorliegenden Kapitel werde ich die Anforderungen sukzessiv realisieren. Die Abarbeitung werde ich nach dem im Kapitel 5 festgelegten Plan *(s. Tabelle 4.6)* durchführen. Die im Kapitel 5 definierten Anforderungen beruhen auf einer Messung auf Prozessebene. Der Ursprung der meisten auf Prozessebene diskutierten Probleme liegt jedoch im Produkt selbst. Um die Uhrsache dieser Probleme zu finden, werde ich hier zu jeder Anforderung passende Produktmetriken einsetzen.

5.1 Realisieren der Anforderung aus dem Betrieb

Das Problem bestand in dem zu hohen Aufwand für Fehlerbeseitigung. Um die Ursache des Problems zu finden, habe ich zunächst die Altprojekt-Fehlerdatenbank mit 320 Fehlern und die dazugehörigen 600 Masken, vereinigt in 180 Screensets, benutzt. Darüber hinaus habe ich einige zusätzliche Messungen im Code durchgeführt. Aus dieser Analyse haben sich folgende Resultate ergeben:

Messungen im Produkt:

1. LOC (DS)[1] = ca. 200000.
2. LOC (umgesetzt) = ca. 800000.

Beobachtungen und Messungen aus der Fehlerdatenbank:

1. Der Aufwand für Beseitigung eines Fehlers ist relativ niedrig: ca. 30 Fehler pro PT.
2. Die Restfehlerrate ist dafür zu hoch: $\frac{\textbf{Fehleranzahl}}{KLOC} = \frac{320}{800} = 0,475$.
3. Es ist eine Fehlerredundanz zu beobachten. Ein Fehler tritt also fast nie allein. Der Aufwand für Beseitigung des Erstfehlers beträgt ca. $\frac{1}{4}$ PT und für die Folgefehler etwa jeweils $\frac{1}{30}$ PT.

Aus der Analyse der Messungen und den detaillierten Codeuntersuchungen lässt sich schlussfolgern, dass den Maskendefinitionen eine Code-Redundanz zugrunde liegt. Im Folgenden werde ich solche Stellen an einigen Beispielen aus Quellentextabschnitten veranschaulichen.

Listing 5.1 (Auszug aus Originalmaskendefinitionen (DialogSystem))

```
1 ...
2 GET–SELECTED–LIST–ITEM LB–DATUM $REGISTER DATUM;
3 REFRESH–OBJECT LB–DATUM;
4 SET–TOP–LIST–ITEM LB–DATUM $REGISTER;
5 GET–SELECTED–LIST–ITEM LB–AUSWERT $REGISTER AUSWPOS;
6 ...
```

[1]LOC (DS): LOC in den original Masken.

Listing 5.2 (Das Äquivalent des Listings 5.1 in Java)

```
1  ...
2  // GetSelectedListItem LbDatum Register Datumpos
3  for(int i = datumpos.toInt(); i < jLbDatum.getSize(); i++){
4          if(jLbDatum.getModel().isSelectedIndex(i)){
5                  this.getRegister().setValue(i+1);
6                  break;
7          }
8  }
9  //RefreshObject LbDatum
10 jLbDatumGlue.update();
11 // SetTopListItem LbDatum Register
12 jLbDatum.ensureIndexIsVisible(Register.toInt()-1);
13 //GetSelectedListItem LbAuswert Register Auswertpos
14 for(int i = auswpos.toInt(); i < jLbDatum.getSize(); i++){
15         if(jLbDatum.getModel().isSelectedIndex(i)){
16                 this.getRegister().setValue(i+1);
17                 break;
18         }
19 }
20 ...
```

Im Listing 5.2 ist eine Code-Redundanz klar zu erkennen. In den Zeilen 2 und 5 wird die DialogSystem-Funktion "GET-SELECTED-LIST-ITEM" zwei mal aufgerufen. In dem umgesetzten Java-Code (Listing 5.2) dagegen wird die gesamte Implementierung dieser Methode jedes Mal abgbildet *(s. Zeilen 3 - 8 und 14 - 19)*. Das Problem tritt bei der Umsetzung aller DialogSystem-Funktionen auf.

Das produziert zum einen Code-Redundanz und zum anderen LOC im Überfluss. Die Messungen im Produkt ergeben eine vierfache Erhöhung der Anzahl der Programmcodezeilen (LOC) in den produzierten Masken gegenüber dem Original. In dem Beispielcode werden 4 in 18 Zeilen umgesetzt.

Eine Lösung dieser Problematik stellt die Auslagerung dieser Funktionalität in eine zentrale Stelle in der Runtime. Um das zu realisieren, müssen folgende Bedingungen erfüllt werden:

1. Die Kontextunabhängigkeit aller Funktionen muss gewährleistet werden.
2. Die Funktionen müssen synchron im ScreenManager-Thread ausgeführt werden. So z.B. ist eine Ausführung aus dem AWT-Event-Thread unzulässig.

Realisierung: Vorhandene DialogSystem-Funktionen habe ich mittels *command design pattern* [GHJV] in der Runtime abgebildet. (s. Abbildung 5.1). Für Weiterentwicklungen habe ich eine API, die *system enhancement API*, zur Verfügung gestellt.

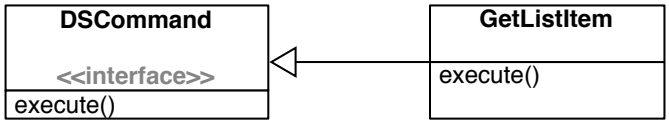

Abbildung 5.1: CommandPattern für DS-Funktionen. Bsp. GetListItem

Commands werden in einem GUI-Context erzeugt. Das tatsächliche Ausführen des Commands findet in dem *ScreenManager* statt. Die Klasse *ScreenManager* spielt dabei die Rolle des *Invokers* (vgl. [GHJV]). Die *execute()* Methode wird innerhalb des *ScreenManager*-Threads ausgeführt. Somit ist die Bedingung *2* erfüllt (s. Abbildung 5.2).

Die Kontextunabhängigkeit (Bedingung 2) wird implizit gewährleistet. Konkrete Kommandos "kennen" ihren Kontext nicht. Betroffene GUI-Elemente *(Receiver)* werden als Parameter übergeben.

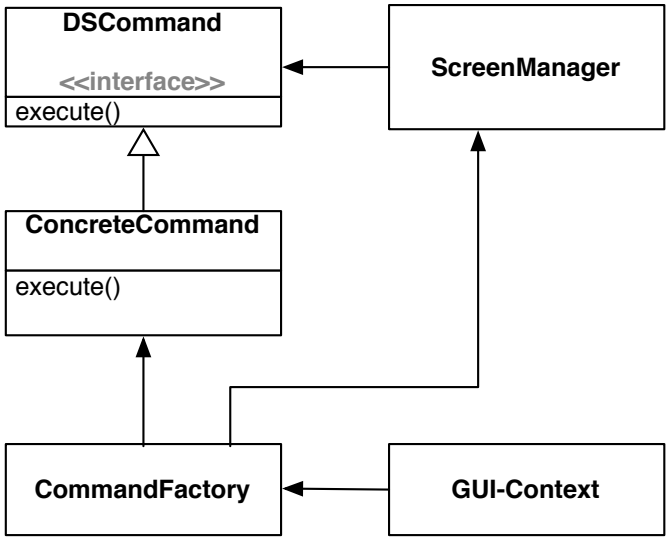

Abbildung 5.2: DialogSys2Java und CommandPattern für DS-Funktionen.

Das Framework bietet dem Anwendungsentwickler keine Möglichkeit, direkt auf die Methoden des *ScreenManagers* zuzugreifen. Daher habe ich die *CommandFactory* eingeführt (s. Abbildung 5.2). Aus einem GUI-Kontext heraus wird ausschließlich über *CommandFactory* auf die *Command*-Funktionalität zugegriffen.

Das Beispiel aus dem Listing 5.2 wird nach dem Refactoring wie folgt aussehen:

Listing 5.2 (Das Beispiel aus dem Listing 5.1 nach dem Refactoring)

```
1 ...
2 GetSelectedListItem.execute(jLbDatum, getRegister());
3 RefreshObject.execute(jLbDatum);
4 SetTopListItem.execute(jLbDatum, getRegister().toInt());
5 GetSelectedListItem.execute(jLbAuswert, getRegister());
6 ...
```

5.2 Realisieren der Anforderung aus der Systemerweiterung

Das Problem bei der Systementwicklung war, dass der Aufwand für die Implementierung einer neuen DialogSystem-Finktionalität zu hoch ist.

Die Entwicklung betrifft zwei Systemkomponenten, die *Runtime* und den *Umsetzer*. Dabei werden folgende Schritte durchgeführt:

1. Die Runtime wird um die gewünschte Funktion-Unterstützung erweitert. Eine DialogSystem-Funktion ist entweder ein *DSCommand* oder ein *DSEvent*.

2. Die neue Funktionalität wird in den Maskendefinitionen angewendet. Deswegen wird der Maskenumsetzer auch erweitert.

Refactoring der DSEvent-Verarbeitung Ein *DSEvent* repräsentiert eine Benutzeraktion auf der GUI-Oberfläche *(vgl. AWTEvent [JAVAAPI])*. Die Aufga-

be der DSEvent-Verarbeitung ist, diese Events abzufangen und sie entsprechend threadsicher zu verarbeiten.

Die Verarbeitung des Events wird mittels *command design pattern* realisiert *(s. Abbildung 5.3)*. In der *GUIListener* werden *AWTEvents* abgefangen und zu *DSEvents* gemapt. Anschließend werden sie dem *ScreenManager* übergeben, der sie sequenziell ausführt. Eine Erweiterung der Runtime um einen neuen DSEvent-Typ verlangt eine Modifizierung der Klasse *DSEvent*. Im Folgenden werde ich die Elemente der Eventverarbeitung analysieren. Dabei sind einige Messungen im Code durchzuführen.

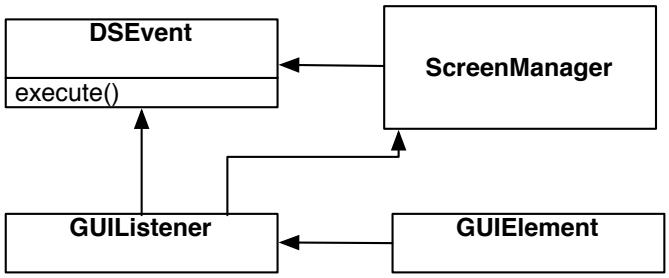

Abbildung 5.3: DSEvent Verarbeitung (Klassendiagramm).

Messungen in der Klasse *DSEvent* (Ausgangszustand mit 16 unterstützten DSEvent-Typen) :

- Total LOC (lines of code): 120
- Maximal MCC (McCabe's Cyclomatic Compexity): 23

Zum Testzweck habe ich die Runtime um noch 16 Dummy-DSEvents auf die vorgeschriebene Art erweitert. Ich habe also die Mächtigkeit verdoppelt und nochmals die gleichen Messungen durchgeführt.

Messungen in der Klasse *DSEvent* (Runtime mit 32 unterstützten DSEvent-Typen) :

- Total LOC (lines of code): 180
- Maximal MCC (McCabe's Cyclomatic Compexity): **39 !**

Es ist eindeutig, dass die Metrik MMC proportional zur Mächtigkeit $M_{dsevent}$[2] steigt. Es gilt also $MMC(DSEvent) \sim M_{dsevent}$. Dies ist vermutlich eine Ursache für den steigenden Entwicklungsaufwand in der Systemerweiterungsphase. Aufgabe des Refactorings an dieser Stelle ist, diese Abhängigkeit abzulösen.

Realisiering: Die Uhrsache der steigenden MMC ist eine *if else statenment* Verschachtelung in der *execute()*-Methode in *DSEvent*. Eine Standardlösung bietet hier [GHJV] durch die *strategy design pattern*. Durch den Einsatz des strategy patterns gewinnt man ein zusätzliches Nutzen bezüglich des Entwicklungsaufwands. Die Kapselung mit strategy pattern gewährleistet eine bessere Ausführung. Diesbezüglich habe ich eine Veränderung des Designs durchgeführt. Diese wird in der Abbildung 5.4 veranschaulicht.

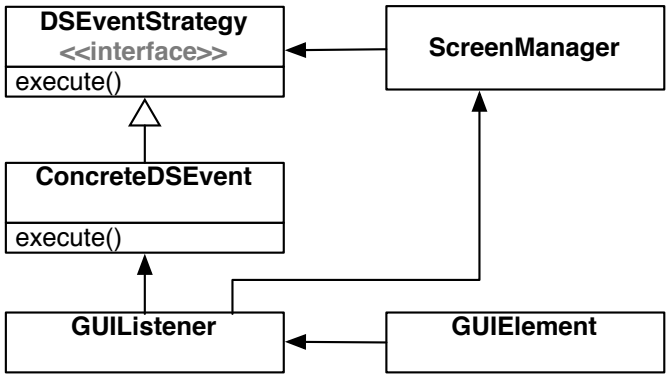

Abbildung 5.4: Strategy pattern in der DSEvent-Verarbeitung.

Nach dem Redesign habe ich LOC und MCC noch einmal mit 16 bzw. 32 unterstützten DSEvent-Typen gemessen. Die Klasse *DSEvent* existierte in der

[2] $M_{dsevent}$: Anzahl unterstützer DSEvent-Typen

herkömmlichen Form nicht mehr. Sie wurde durch die Klassen *DSEventStrategy* und je eine Implementierung *(ConcreteDSEvent)* für jedes Event ersetzt.

Messungen in der Klasse *DSEventStrategy* und 16 *DSEventStrategy* Implementierungen (Ausgangszustand mit 16 unterstützten DSEvent-Typen) :

- Total LOC (lines of code): 280
- Maximal MCC (McCabe's Cyclomatic Compexity): **6**

Messungen in der Klasse *DSEventStrategy* und 32 *DSEventStrategy* Implementierungen (Ausgangszustand mit 32 unterstützten DSEvent-Typen) :

- Total LOC (lines of code): 410
- Maximal MCC (McCabe's Cyclomatic Compexity): **6**

Es ist offensichtlich, dass die Abhängigkeit $MMC(DSEvent) \sim M_{dsevent}$ abgelöst wurde. Der maximale MMC-Wert beträgt 6, unabhängig von der Anzahl der Implementierungen. Das Code-Volumen (LOC) hat massiv zugenommen (vorher: 180, nacher 410). Das liegt daran, dass jede Event-Implementierung in eine eigene Klasse stattfindet.

Auswirkungen des DSCommand-Konzepts in der Systemerweiterung Im Kapitel 6.1 habe ich das DSCommand-Konzept eingeführt. Dabei wurden unter anderem einzelne DialogSystem-Funktionen in eigenen Klassen gekapselt *(encapsulation)*. Dies verhindert das Umgehen von Invarianten des Programms. Somit besteht auch kein Risiko mehr, durch eine Systemerweiterung alte Funktionalität zu behindern, was früher viel Entwicklungsaufwand gekostet hat. Das erleichtert erheblich die Weiterentwicklung und wird zu einer Optimierung dieses Pozessschritts führen.

5.3 Realisieren der Anforderung aus der Weiterentwicklung

Das Problem bei der Weiterentwicklung des umgesetzten Programms war, dass der Einarbeitungsaufwand in DialogSys2Java zu hoch liegt.

Als Quelltext-Referenz benutze ich eine Altprojekt-Quellentextdatenbank mit ca. 600 Masken und die dazugehörige Dokumentation. Um die Ursachen des Problems zu ermitteln, reicht es jedoch nicht, nur der Quellentext einer Analyse zu unterziehen. Ich habe deswegen die damals beteiligten Anwendungsentwickler interviewiert. Aus den erhobenen Meinungen konkretisierten sich folgende zwei Problempunkte:

1. Alte DialogSystem-Definitionen sind nach der Umsetzung sehr schwer wiederzuerkennen. Somit ist die Prüfung der Coderichtigkeit auch sehr aufwändig. Namenskonventionen sind verloren gegangen.

2. Das Einbetten neuer Event-Verarbeitungsroutinen ist langwierig. Die Wartung erfordert einen hohen Aufwand.

Realisieren vom Punkt 1: Wenn wir Listing 5.1 mit dem Listing 5.2 aus Abschnitt 5.1 vergleichen, werden wir kaum eine Ähnlichkeit erkennen, obwohl die beiden Programmabschnitte dasselbe leisten. Das spielt sicherlich keine Rolle bezüglich der Richtigkeit des Programms, hat aber eine grosse Bedeutung in der Phase der Weiterentwicklung und der Fehlerbehebung.

Die *DSCommand*-Lösung aus Abschnitt 5.1 bietet eine implizite Lösung dieses Problems. Dabei wurden alte DialogSystem-Funktionen in einer gleichnamigen Klasse gekapselt. z.B. die DialogSystem-Funktion **GET-SELECTED-LIST-ITEM** wurde in die Klasse **GetSelectdedListItem** umgesetzt. Im Abschnitt 5.1

habe ich die Namen der Klassen intuitiv so vergeben. An dieser Stelle lege ich eine Namenskonvention fest:

Namenskonvention 5.1: DialogSystem-Funktionen werden als *DSCommand*- Implementierungen in der DialogSys2Java-Runtime dargestellt. Dabei behalten alle Funktionen ihre Namen als Klassennamen. Eine Namensveränderung, soweit die Java-Syntax-Konformität gewährleistet werden muss, ist erlaubt.

Somit wird der generierte Code aus Listing 5.1 wie folgt aussehen:

Listing 5.3

```
1 ...
2 GetSelectedListItem.execute(jLbDatum, getRegister());
3 RefreshObject.execute(jLbDatum);
4 SetTopListItem.execute(jLbDatum, getRegister().toInt());
5 GetSelectedListItem.execute(jLbAuswert, getRegister());
6 ...
```

Realisieren des Punkts 2: Um das Problem zu untersuchen, musste ich zunächst einige Messungen in dem generierten Code durchführen:

- Total KLOC (DS): 200
- Total KLOC (JAVA): 801
- Maximal MCC (McCabe's Cyclomatic Compexity): **191 !**

Problematisch ist der MCC-Wert. Er liegt weit über dem zulässigen[3]. Problemstellen sind die Event-Handling-Routinen (s. Listing 5.4). Die Fallunterscheidung wird durch verschachtelte *if-else* Anweisungen realisiert.

Listing 5.4 (Auszug au den Event-Handling-Routinen in dem gen. Code)

```
1  ...
2  else if (src == EfWi02R1Stat && code == KeyEvent.VK_A)
3  {
4          this.jEfWi02R1StatCrEvent();
5  }
6  else if (src == jEfWi02UebR4 && code == KeyEvent.VK_B)
7  {
8          this.jEfWi02UebR4CrEvent();
9  }
10 else if (src == jEfWi02UebR3 && code == KeyEvent.VK_ENTER)
11 {
12         this.jEfWi02UebR3CrEvent();
13 }
14 else if (src == jEfWi02UebR2 && code == KeyEvent.VK_ENTER)
15 ...
```

Als Lösung dieses Problems ersetze ich die *if-else* Verarbeitung durch eine *strategy pattern*-Realisierung (s. Abbildung 5.5).

[3]MCC > 50 gilt als sehr schlecht wartbar.

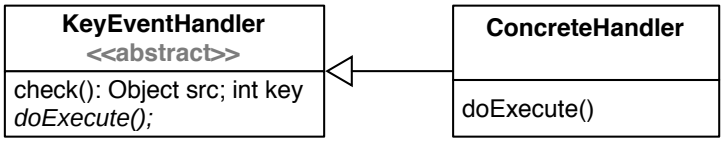

Abbildung 5.5: Strategy Pattern bei der KeyEvent-Handling.

Im Listing 5.5 sind die KeyEvent-Verarbeitungsroutinen nach dem Refactoring dargestellt.

Listing 5.5 (Event-Handling Routinen nach dem Refactoring)

```
1  ...
2  private static final KeyEventHandler HANDLER_WI0220_A
3          = new KeyEventHandler(jMfeld0220, KeyEvent.VK_A){
4
5          public abstract void doExecute(){
6                  jEfWi0220MfeldCrEvent();
7          }
8  };
9
10 private static final KeyEventHandler HANDLER_WI0221_A
11         = new KeyEventHandler(jMfeld0221, KeyEvent.VK_A){
12
13         public abstract void doExecute(){
14                 jEfWi0221MfeldCrEvent();
15         }
16 };
17 ...
```

Nach einer Neuumsetzung der Referenzmasken, diesmal mit der oberen Modifizierung, erhalte ich einen maximalen MCC-Wert von **19**, der jedoch nicht in den KeyEvent-Handling-Routinen auftritt. Da diese Lösung ähnlich wie die im Abschnitt 5.1 präsentierte ist, erwarte ich auch hier einen Anstieg des LOC-Wertes. Die LOC Messung ergibt den Wert **860 KLOC**.

5.4 Realisieren der Anforderung aus der Schnittstellenanpassung

Die Anforderung aus der Prozessebene war, die Phase der Schnittstellenanpassung zu überspringen. Um das zu realisieren, werde ich mich zunächst mit der Spezifikation der Originalschnittstelle auseinandersetzen.

Die Originalschnittstelle (DSGRUN - Interface) Die originale DialogSystem-Schnittstelle befindet sich in einer dynamischen Bibliothek *(DLL)*[4] namens "DSGRUN.dll". Diese Bibliothek stellt nur eine Einstiegsprozedur *(entry point)* mit dem gleichen Namen (DSGRUN) zur Verfügung. Die Einstiegsprozedur *DSGRUN* hat keinen Rückgabewert *(void)* und erwartet zwei Parameter: $CTRL$ und $DATA$.

- Parameter $CTRL$: Zeiger auf der Speicheradresse der *DS-CTRL-DATA*-Block. Die Länge dieses Speicherbereichs beträgt **400 Bytes**.
- Parameter $DATA$: Zeiger auf der Speicheradresse des *DS-DATA*-Blocks. Die Länge dieses Speicherbereichs ist variabel. Sie hängt von der Komplexität der aktuellen Maske ab.

[4]DLL: dynamic link library

Die *DSGRUN*-Schnittstelle lässt sich aus jedem Kontext in der Anwendung verwenden. In COBOL-Syntax sieht den Aufruf so aus:

```
...
CALL DSGRUN USING DS–CTRL–DATA, DS–DATA.
```

Bei der Neuentwicklung der Ersatzschnittstelle sind folgende Anforderungen zu erfüllen:

1. Namenskonventionen der Schnittstelle müssen erfüllt werden (DLL-Name und entry point-Name).
2. Die Schnittstellenspezifikation muss "stabil" bleiben.
3. Die Ermittlung der Größe der DS-DATA muss implizit erfolgen.
4. Das Umwandeln der nativen Speicherbereiche ins Java-Datentypten muss implizit erfolgen.

Realisierung: Die neue Schnittstelle habe ich in der Programmiersprache *C* entwickelt. Um die Anbindung *Native-to-JAVA* zu realisieren, habe ich *Java Native Interface (JNI)* Version 1.4 verwendet. Die Speichergröße des DS-DATA-Blocks lässt sich javaseitig aus den umgesetzten Fachdataobjekten ermitteln.

Anhand des in Abbildung 5.6 veranschaulichten Sequenzdiagramms stelle ich das Interobjektverhalten in der von mir entwickelten Ersatzschnittstelle dar. Alle Interaktionen verlaufen synchron. Der Aufruf der Methode *getDataSize* liefert zur Laufzeit die Grösse des DS-DATA-Blocks. Anschließend werden Kopien der beiden Speicherbereiche formatiert als *byte[]* an die DialogSys2Java-Runtime übergeben.

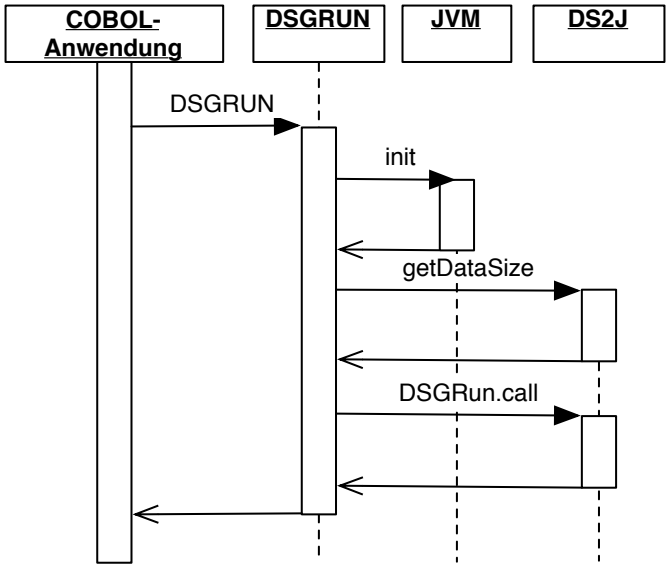

Abbildung 5.6: DSGRUN-Call (Sequenzdiagramm).

In dieser Ausarbeitung werde ich nicht auf weitere Implementierungsdetails eingehen. Mit dem Ersatz der DSGRUN-Schnittstelle habe ich ein zukünftiges Überspringen dieses Prozessschrittes erreicht.

5.5 Realisieren der Anforderung aus der Testphase

Als Anforderung in der Testphase hat sich eine Aufwandsreduktion hinsichtlich des Testfallerzeugens ergeben. Das konventionelle Verfahren (manuelles Beschreiben eines Testfalls) erwies sich als sehr mühsam.

Eine Lösung dieses Problems kann ein Tool zur Testautomatisierung *(Capture/Replay)* anbieten. Damit kann der Benutzer bzw. der Tester alle Aktivitäten auf der Bildschirmoberfläche (Mausklicks, Tastatureingaben etc.) automatisiert aufzeichnen (Capture) und sie später wiedergeben (Replay).

In der Laufzeitumgebung habe ich einen Testfallgenerator eingebettet, der die

Capture-Funktionalität anbietet. Die Benutzeraktivitäten werden damit in CSV[5]-Format aufgezeichnet. Abbildung 5.7 veranschaulicht den prinzipiellen Aufbau des Testfallgenerators.

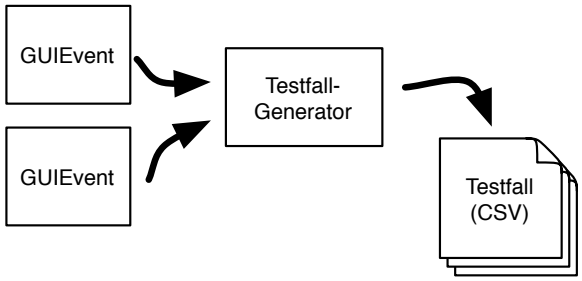

Abbildung 5.7: Testfallgenerator in der DS2J-Runtime

5.6 Realisieren der Anforderungen aus der maschinellen Umsetzung

Die Anforderungen aus der maschinellen Umsetzung waren, die Grammatik des Exportformats zunächst formal zu beschreiben, und anschließend den Parser zu erweitern. Die formale Beschreibung des Exportformats habe ich bereits im Abschnitt 3.4.3 *(Formale Beschreibung der Quellsprache)* geschildert. In den Anhängen A.1 und A.2 sind die syntaktischen bzw. lexikalischen Regeln vollständig abgebildet. Im Folgenden werde ich die Schritte vorstellen, die ich durchgeführt habe, um den eingesetzten Parser zu erweitern.

1.) Wie ich im Kapitel 3 *(Systembeschreibung)* schon erwähnt habe, wird der Parser mithilfe eines Parsergenerators (CUP) erzeugt. Als Input des Parsergenerators fungieren die Grammatikregeln der Sprache GMDL (s. Anhang A.1) in BNF-Notation. Diese Grammatikregeln habe ich zunächst aus $EBNF$ in BNF überführt. Die Listings 5.6 und 5.7 stellen einen Auszug dieser Überführung dar.

[5]CSV: Comma Separated Values

Listing 5.6 (EBNF)

```
datablock_clause ::=
        {data_declaration | group_declaration | comment}
```

Listing 5.7 (BNF)

```
datablock_clause ::=
        data_declaration datablock_clause|
        group_declaration datablock_clause |
        comments datablock_clause |
        <EPSILON>
```

2.) Die in *(1)* abgeleiteten Regeln habe ich in die Eingabedatei *dialogsys2j.cap* des Parsergenerators CUP eingetragen (s. Listing 5.8). Dabei verzichte ich darauf, Generatoranweisungen einzugeben, da das Erweitern des Generators ein Teil der Systemerweiterungsphase ist. Anstelle einer Generatoranweisung gebe ich den Namen des Terminals bzw. Nonterminals in einer log-Datei aus (s. Listing 5.8, Zeile 8).

Listing 5.8 (BNF-Regeln in CUP-Notation)

```
1  form_data_declaration ::=
2          FORM DATA
3          {:
4                  logger.log("found FORM_DATA");
5          :}
6          data_elements
7          {:
8                  logger.warn("found not implemented item");
9          :}
10         END DATA
11         {:
12                 logger.log("found END_DATA");
13         :};
```

3.) Ich habe die eingebetteten Grammatik-Regeln auf eventuelle *shift/reduce* bzw. *reduce/reduce* Konflikte hingeprüft (s. Kapitel 2.3.2 -*Bottom-Up-Parser*). Anschließend ließ ich den Parsergenerator mit dem Befehl *java java_cup.Main dialogsys2j.cap* laufen.

Der Parser wurde damit fehlerfrei (konfliktfrei) erzeugt. Das garantiert die vollständige Abbildung der Grammatikregeln der Exportformats von DialogSystem. In Zukunft müssen bei der Systemerweiterung die leeren Codegenerator-Anweisungen implementiert werden.

Kapitel 6

Ergebnisanalyse

6.1 Allgemeines

Die Ziele des Refactorings wurden aus Problemen auf der Prozessebene heraus definiert. Die betroffenen Systemkomponenten sind in der Realisierung auf Ursachen für diese Probleme hin untersucht und anschließend zielgerecht verbessert worden. In diesem Kapitel werde ich die Verbesserungen hinsichtlich der gesetzten Ziele auf der Prozessebene überprüfen.

6.2 Ergebnisanalyse auf Prozessebene

Die Realisierung der Anforderung *A3* hat den kompletten Prozessschritt *Schnittstellenanpassung* aus dem Prozessverlauf eliminiert. Abbildung 6.1 stellt die veränderte Prozessstruktur dar.

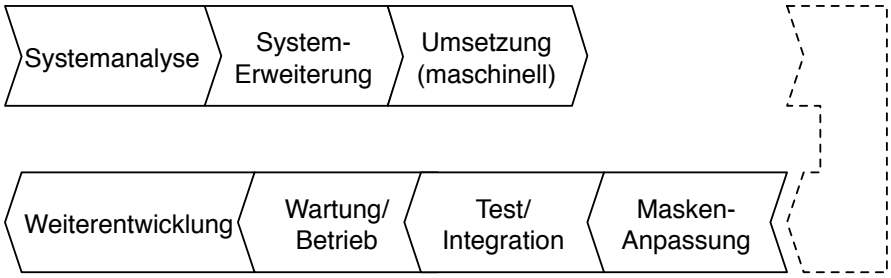

Abbildung 6.1: Prozessphasen nach dem Refactoring

Vor dem Refactoring hat dieser Prozessschritt bis zu $\frac{1}{5}$ des Gesamtprozessaufwandes verbraucht. Bei einem Mirgationsrojekt der mittleren Größe mit ca. 700 zu umsetzenden Masken wird der Gesamtaufwand auf ca. 70 PT geschätzt. Davon wurden ca. $70 * \frac{1}{5} = 14$ PT nur in der *Schnittstellenanpassung* aufgebraucht. Nach dem Refactoring wird also der Gesamtprojektaufwand dank dieser Verbesserung um ca. $\frac{1}{5}$ reduziert.

Als Lösung der Anforderung aus der Systemerweiterung *(A2)* habe ich eine *API*, die *system enhancement API*, im Kapitel 5 entwickelt. Damit lassen sich neue DialogSystem-Funktionen schneller und effektiver ins System integrieren. Auf Prozessebene werde ich versuchen, diese Behauptung zu verifizieren.

Da ich im Rahmen dieser Arbeit leider nicht über ausreichend Zeit verfüge, Tests in einer "echten" Umgebung durchzuführen, habe ich stichprobenweise zwei DialogSystem-Funktionen selbst implementiert und die Aufwandsmessungen dabei

durchgeführt:

- Die DialogSystem-Funktionen SET-COLOR und SET-OBJECT-LABEL wurden testweise integriert.
- LOC(SET-COLOR) = 118
- LOC(SET-OBJECT-LABEL) = 176
- Aufwand für die Entwcklung inklusive Test: 1,5 PT $\Rightarrow \frac{3}{4} \frac{PT}{Erweiterung}$

Nach dem Refactoring lässt sich also ein mittlerer Entwicklungsaufwand pro neue Funktion auf $\frac{3}{4}$ abschätzen, was gegenüber dem alten Wert von $\frac{5}{4}$ als effizienter gilt.

Im fünften Refactoringsschritt (s. Abschnitt 5.5) habe ich ein Tool zur Testautomatisierung in die Runtime eingebettet. Um zu prüfen, ob der Einsatz dieses Tools Nutzen auf den Prozessebene bringt, habe ich zum Messzweck Testfälle damit erzeugt. Ich habe dabei folgende Anforderung definiert: Testfälle für 20 vorgegebene Masken mittlerer Komplexität zu erzeugen. Hierbei soll eine volle Maskenüberdeckung, GUI-Elementen-Überdeckung sowie Überdeckung in der Struktur erreicht werden.

Ergebnisse:

- Testmenge: 20 Masken; ca. 6 KLOC
- Insgesamt 36 erzeugte Testfälle
- Aufwand: ca. 1,5 PT
- Erreichte Überdeckung: Masken: 100%; GUI-Elemente: 100%; Strukturüberdeckung: 62% lines-coverage und 77% methods-coverage

Der Aufwand von 1,5 PT gegenüber dem Aufwand für manuelle Testfallerzeugung (ca. 5 Testfälle pro PT) macht das Nutzen des Tools offensichtlich. Der Testfallgenerator bietet leider keine Unterstützung im Hinblick auf die Strukturüberdeckung. Um die erreichte Strukturüberdeckung zu ermitteln, habe ich das Tool *Koalog Code Coverage*[1] herangezogen.

Ich kann die Vorteile aus dem Refactoring im Betrieb nicht bewerten, da das System in dieser Form noch nicht produktiv ist. Die unten aufgelisteten Ergebnisse werden vermutlich zur Reduktion der Restfehlerrate führen.

1. Das Code-Volumen (LOC) des generierten Codes wurde um Faktor $\frac{1}{2}$ reduziert.

2. Die Code-Redundanz im generierten Code wurde durch die Auslagerung der DialogSystem-Funktionen in die Runtime deutlich minimiert.

[1]Koalog Code Coverage ist ein Tool für Messung der Code-Überdeckung der Firma Koalog. URL:http://www.koalog.com

Kapitel 7

Zusammenfassung und Ausblick

DialogSys2Java ist ein vollwertiger Java basierter *DialogSystem*-Ersatz für COBOL-Anwendungen. Es handelt sich dabei um eine GUI-Laufzeitumgebung und um einen Umsetzer, der DialogSystem-Oberflächen maschinell in DialogSys2Java überführt. Die Migration einer Legacy-Anwendung (der Migrationsprozess) verläuft nicht immer völlig generisch. Das Volumen und die Komplexität der Altannwendungen führen zu zusätzlichen Aufwänden unterschiedlichen Umfangs im Migrationsprozess.

Ziel dieser Arbeit war, die Zusatzaufwände und die Risiken auf Prozessebene zu analysieren, die Uhrsachen dafür zu finden und diese durch das Verbessern der inneren Programmstruktur der DialogSys2Java *(refactoring)* zu eliminieren.

Im Kapitel 2 wurden die benötigten Grundlagen vorgestellt. Es wurde eine Definition des Refactorings herausgearbeitet. Danach habe ich auch eine kurze Einführung in die Messung von Softwarequalität gegeben, weil die Softwaremetriken eins der wichtigsten Werkzeuge des Refactorings darstellen. Bei dem Maskenumsetzer handelt es sich um einen Compiler. Wegen der Komplexität des Themas "Compilerbau" habe ich einen Überblick über dessen Grundbegriffe gegeben.

Im dritten Kapitel wurden die Komponenten des Systems beschrieben. Unter dem Gesichtspunkt der Mehrschichtenarchitektur wurde zuerst ein Komponentenüberblick geschaffen. Dann wurden die Konzepte und die Grundbausteine des Legacy-Systems *(DialogSystem)* und des Ersatzsystems *(DialogSys2Java)* betrachtet. Anschießend erfolgt eine formale Beschreibung der Sprache des Exportformats von DialogSystem (der Quellsprache des Umsetzers).

Das Kapitel vier befasst sich mit dem Konzept des Refactorings. Daher wurde zuerst der Migrationsprozess erläutert. Jeder Migrationschritt wurde als eine Quelle der Rohanforderungen betrachtet. Ausgehend von der Priorität der Rohanforderung und der betroffenen Systemkomponenten wurde ein Plan des Refactorings zusammengestellt.

Im Kapitel 5 wurden die Anforderungen plangemäß sukzessiv abgearbeitet. Dabei wurde zuerst der Ist-Zustand des Systems durch passende Softwaremetriken untersucht.

Das Kapitel sechs stellt eine Analyse der Ergebnisse dar. Dabei wurde die Auswirkung des Refactorings auf den Migrationsprozess betrachtet.

Resümierend lässt sich sagen, dass das in dieser Arbeit entwickelte Konzept und die durchgeführten Refactoringsarbeiten durchaus erfolgreich waren. Alle Anforderungen wurden realisiert. Eine Ausnahme macht jedoch die Anforderung aus der manuellen Maskenanpassung, die wegen ihrer niedrigen Priorität im Rahmen dieser Arbeit nicht behandelt wurde.

Ob das Refactoring die gewünschten Ergebnisse geliefert hat, insbesondere bei der Systemerweiterung und bei dem Betrieb, wird sich erst dann in der Praxis bestätigen, wenn neue Migrationsprojekte durchgeführt werden. Entscheidend wird dabei sein, ob die beteiligten Softwareentwickler (Systementwickler) die von mir vorgeschlagenen Richtlinien befolgen.

Anhang A

Listings

A.1 GMDL-Syntax

```
screenset ::= {form_def}
form_def ::=
        form_internals | FORM screenset_id screenset END FORM
form_internals ::=
        datablock | font_record |
        global_dialog | menu_bar | object |
        object_dialog | screenset_details | validation
datablock ::= FORM DATA datablock_clause END DATA
datablock_clause ::=
        {data_declaration | group_declaration | comment}
data_declaration ::= [ data_name data_dclr ]
data_dclr ::=
        CHARACTER [ERROR-FIELD] |
        ALPHABETIC |
```

```
        INTEGER [IMPLICIT SIGN | COMP | COMP-5]
        DDECIMAL [IMPLICIT SIGN]
group_declaration ::=
        GROUP group_name VERTICAL OCCURS n
        data_declaration {data_declaration} END GROUP
comment ::= '*' text
font_record ::=
        [FONT-RECORD style_name font details END-RECORD]
style_name ::= STYLENAME CHARSEQUENCE
font_details ::= [typeface pointsize attributes]
pointsize ::= POINTSIZE INTEGER(5)
attributes ::= {ATTRIBUTES attribute {attribute_list}}
attribute_list ::= ',' attribute
attribute ::= BITMAPPED | BOLD | ITALIC | MONOSPACED | ...
global_dialog ::= GLOBAL DIALOG dialog_defn END DIALOG
dialog_defn ::= {dialog}
dialog ::=
        '*' comment |
        EVENT event {dialog function ';'} END EVENT |
        PROCEDURE name {dialog function ';'} END PROCEDURE
menu_bar ::= MENU menu_description END MENU
menu_description ::=
        PARENT window_name menu_choice {menu_choice}
menu_choice ::= ITEM [choice_name] item_info    END ITEM
item_info ::=
        [DDISPLAY string] [KEY string] [LEVEL n] [style]
style ::= CHECKABLE | CHECKMARKED | DISABLED | ...
```

```
object ::= OBJECT string object_descr END OBJECT
object_descr ::= TYPE type PARENT parent_name object_info
type ::= BITMAP | CHECK_BOX | ENTRY_FIELD | ..
object_info ::=
        bitmap | conrol_group | field | STYLENAME string
        icon | initial_text | local_dialog | masterfield
        msg_text | picture | start_size | style | title
bitmap ::= string | string string | string string string
control_group ::= CONTROL GROUP integer
display ::= DDISPLAY string
field ::= {FIELD field_info END FIELD}
field_info ::= masterfield picture PADDING n
icon ::= ICON string
inittial_text ::= INITIAL TEXT string {string} END TEXT
local_dialog ::= DIALOG dialog_defn END DIALOG
masterfield ::= MASTERFIELD dataname
msg_text ::= MESSAGE string
picture ::= PICTURE string
start_size ::= START [SIZE (n,m)]
style ::= STYLE {style_name [,]}
object_dialog ::= OBJECT DIALOG name dialog_defn END DIALOG
screenset_deteils ::= SCREENSET DETAILS details END DETAILS
details ::= first_window ss_symbols STYLE style ss_files
first_window ::= [FIRST_WINDOW window_name]
ss_symbols ::=
        {[DECIMAL_CHAR|COMMA_CHAR|CURRENCY] 'character'}
ss_files ::= {[ERROR_FILE|ICON_FILE|FONT_FILE] string}
```

```
validation ::=
        DATA VALIDATION data_name null_val range_val
        date_val_user_val END VALIDATION
null_val ::= [NULL DISALLOWED [err]]
range_val ::=
        [RANGE_TABLE TRUE|FALSE [err] {range} END RANGE_TABLE]
date_val ::= [DATE date_format [err]]
date_format ::= DDMM|MMDD|MMYY|...
user_val ::= [EXTERNAL program_name [CANCEL] [err]]
err ::= ERRMSGNO msg_no
range ::= lower_string_def | lower_number_def
lower_string_def ::= lower_string [THRU upper_string]
lower_string_number ::= [THRU upper_number]
```

A.2 GMDL-Tokenklassen (regulär)

```
<LineTerminator> = \r|\n|\r\n
<InputCharacter> = [^\r\n]
<StringCharacter> = [^\r\n""\\]
<SingleCharacter>  = [^\r\n''\\]
<WhiteSpace> = {LineTerminator} | [ \t\f]
<Letter> = [A -Z] | [a - z]
<Digit> = [0 - 9]
<IntegerLiteral> = <Digit><Digit>*
<OctDigit> = [0-7]
<FloatLiteral> = <Digit>* \. <Digit>+
```

$<Identifier> =$

$<Letter> \mid ([\backslash @] \mid [<Letter>])$

$(([<Digit>|<Letter>] \mid [-])* [<Digit>|<Letter>])$

$<NumberedIdentifier> =$

$([\backslash @] \mid [<Digit>|<Letter>])$

$(([<Digit>|<Letter>] \mid [-])* [<Digit>|<Letter>])$

$<Comment> = \backslash* \{<InputCharacter>\}*$

$<Alphabetic> =$ ALPHABETIC$\backslash(<IntegerLiteral>\backslash)$

$<Character> =$ CHARACTER$\backslash(<IntegerLiteral>\backslash)$

$<Integer> =$ INTEGER$\backslash(<IntegerLiteral>\backslash)$

$<Decimal> =$ DECIMAL$\backslash(<FloatLiteral>\backslash)$

$<Size> =$ SIZE$\backslash(<IntegerLiteral>\backslash,<IntegerLiteral>\backslash)$

A.3 GMDL-Reservierte Wörter

FORM, FIRST–WINDOW, SCREENSET, SEARCH, KEYTRANS, ON,
OFF, LOCAL, GLOBAL, DETAILS, DECIMAL–CHAR, COMMA–CHAR,
CURRENCY–SIGN, ERROR–FILE, ICON–FILE, NLS–FILE, FONT–FILE,
OLE–BASENAME, STYLE, STYLENAME, END, DATA, VALIDATION,
OBJECT, REFERENCE, DIALOG, FIELD, MENU, ITEM, GROUP, LEVEL,
PRIVILEGE, KEY, VERTICAL, OCCURS, CHARACTER, SIZE,
ALPHABETIC, COMPUTATIONAL, COMP, COMPUTATIONAL–3, DISALLOWED,
COMPUTATIONAL–5, COMPUTATIONAL–X, IMPLICIT SIGN,
ERROR–FIELD, SIGNED, ERROR ,MESSAGES ,FONT–RECORD,
STYLENAME, TYPEFACE, POINTSIZE, END–RECORD, ATTRIBUTES,
ERRMSGNO ,RANGE–TABLE, THRU, CHECKDIGIT, DATE, TYPE,
PARENT, DISPLAY, MASTERFIELD, PROGRAM–NAME, EVENT,

PROCEDURE, BITMAP–NAME, INITIAL, STATUS–TEXT, TAB–TEXT, TEXT, MESSAGE, PADDING, ICON, $REGISTER, $EVENT–DATA, $NULL, $WINDOW, $INSTANCE, $CONTROL, XIF<, *XIF*>, XIF<=, *XIF*>=, XIF=, XIFNOT=, IF<, *IF*>, IF<=, *IF*=, *IF*>=, IFNOT=, NULL, CONTROL–GROUP, CASE, RESIZE–CHAR, #, @, INTEGER, DECIMAL

Literaturverzeichnis

[BVIEL] Vielsack, Bertram: *Project Compiler Generation - The Parser Generators Lalr and Ell*; Gesellschaft für Mathematik und Datenverarbeitung MBH; Forschungsstelle für Programmstrukturen an der Universität Karlsruhe; April 1998

[CPOLZ] Polze, Christoph: *Formale Grammatiken und Compilerbau - Literatur, Einführung, LL- und LR- Verfahren*; Humboldt Universität zu Berlin; Dezember 2000

[DKNU] Knuth, Donald. E *Syntactic Algorithms, The Art of Computer Programming*; Band 5; Addison Wesley

[DSJ] frobese GmbH Informatikservices : *DialogSys2Java reference manual*; frobese GmbH; 2003

[DSJQ] frobese GmbH Informatikservices : *DialogSys2Java quick start*; frobese GmbH; 2005

[FROB] Roberts, Fred S.: *Measurement Theory with Applications to Decisionmaking*; Encyclopedia of Mathematics and its Applications, Addision Wesley Publisching Company, 1979

[FWER] Werner, Florian : *Von der Analyse zum Entwurf*; Universität Tübingen SS 2006, S. 3f.

[GHJV] Gamma ,Erich;Richard ,Helm;Johnson, Ralph; Vissides, John : *Design Patterns - Elements of Reusable Object-Orientet Software*; Addison-Wesly Professional Computing Series; Addison-Wesley Publisching Company, Inc.; 1995

[GTHA] Thaller, Georg Erwin: *Software Qualität - Der Weg zu Spitzenleistungen in der Softwareentwicklung*; VDE Verlag Berlin und Offenbach, 2000

[GVKUW] Vossen, Gottfried und Witt Kurt-Ulrich: *Grundlagen der Theoretischen Informatik mit Anwendungen*; Vieweg Verlagsgesellschaft mbH; Braunschweig/Wiesbaden; 2000

[HKOPP] Kopp, Herbert: *Compilerbau : Grundlagen, Methoden, Werkzeuge*; München, Wien; Hanser, 1988

[HSCH] Schwanke, Helmut : *COBOL: aktuelles Lehrbuch und Nachschlagwerk, Entwicklung von Windows- und Internet- Applikationen* ; München: Markt und Technik, Buch und Software-Verl., 1999

[HZUS] Zuse, Horst: *Software Complexity; Measures und Methods*; Berlin 1991.

[JAVAAPI] Sun Microsystems, Inc.: *JavaTM 2 Platform, Standard Edition, v 1.4.2 API Specification*; URL: http://java.sun.com/j2se/1.4.2/docs/api/

[JNI] Sun Microsystems, Inc. : *Java Native Interface*; URL: http://java.sun.com/j2se/1.4.2/docs/guide/jni/

[MFDS] Micro Focus: *Micro Focus Dialog System - Dialog System Reference*; Micro Focus; Issue 5; September 1994

[MFOW] Fowler, Martin: *Refactoring Improving The Design Of Existing Code*; The Addison-Wesley Object Technology Series

[RPAR] Parchmann, Reiner: *Programmiersprachen und Übersetzer, Skript zur Vorlesung*; Universität Hannover, Sommersemester 2002

[TFET] Fetcke, Thomas: *Softwaremetriken bei objektorientierter Programmierung*; Geselschaft für Mathematik und Datenverarbeitung, April 1995

[WLRK] wikipedia: *Kategorie Compilerbau, LR(k)-Grammatik*; URL: http://de.wikipedia.org/wiki/LR(k)-Grammatik