Diplomarbeit

Generierung, Konfiguration und Auswertung von Eingabemasken
—
Ein Konzept

Ekkart Kleinod
Matrikelnummer: 111038
Berlin, 8. Mai 2001

Humboldt-Universität zu Berlin
Fraunhofer ISST Berlin

Betreuer:
Prof. Dr. Klaus Bothe
Andreas Billig

Thesen

1. Das Problem der Entwicklung stark veränderlicher Programme kann mit Abtrennung der veränderlichen von den unveränderlichen Programmteilen vereinfacht werden.

2. Im Fall einer sich häufig ändernden graphischen Benutzeroberfläche ist es möglich, diese in einem veränderlichen Programmteil zu beschreiben und in einen Struktur- und einen Dynamikbeschreibungsteil zu gliedern.

3. Eine gute Handhabbarkeit der Entwicklung und der Einbindung solcher veränderlicher Programmteile erfordert sauber definierte Schnittstellen.

4. Unter der Voraussetzung der häufigen Nutzung kann die Entwicklung und der Einsatz kleiner und spezialisierter Programmiersprachen für die Formulierung des veränderlichen Teils Vorteile gegenüber einer vorhandenen und universellen Programmiersprache bieten.

5. Eine spezialisierte Programmiersprache ergibt im vorliegenden Anwendungsfall Vorteile in der Programmentwicklung und Wartung. Die Abtrennung veränderlicher von unveränderlichen Programmteilen bringt Vorteile innerhalb des technologischen Ablaufs der Systemnutzung.

6. Das Konzept der Programmteilung und des Einsatzes einer spezialisierten Programmiersprache hat Auswirkungen auf alle Phasen der Softwareentwicklung.

7. Die Thesen lassen sich über den speziellen Einsatzfall hinaus verallgemeinern.

Danksagung

Ich möchte meinen Betreuern Andreas Billig und Ulrich Sacklowski sowie meinem Diplomvater Prof. Dr. Klaus Bothe herzlich für ihre Anregungen, Geduld und Mithilfe danken.

Ebenso viel Geduld und Unterstützung brachte meine Familie in Hannover, Berlin und Münster/Trier auf, die mir während des Studiums den Rücken moralisch (und nicht zu vergessen finanziell) stärkte und so entscheidend zum Gelingen des Studiums beitrug.

Sie mußte jedoch nicht die Entstehung dieser Diplomarbeit mit allen Höhen und Tiefen erleben, meinen Frust und meine Freude teilte ich vor allem mit meiner Verlobten Ilka, der an dieser Stelle für Geduld und Liebe gedankt werden soll.

Sie und viele andere Helfer, die aufzuzählen zu weit führen würde, korrigierten diese Arbeit mit viel Mühe und ärgerlicher Genauigkeit, die mir ebensoviel Mühe und Arbeit verschaffte. Dank an Euch.

Markus' Verdienste aufzuzählen – der Rahmen würde gesprengt.

Moralische Unterstützung während der gesamten Studienzeit (und hoffentlich darüber hinaus) trugen mir Jörg und Stephan an.

Ein spezieller Dank geht an Frau Renner. Jeder, der sie kennengelernt hat, wird verstehen, warum. Ihr Verdienst ist, daß ich dem Informatik-Studium in schwierigeren Zeiten treu geblieben bin.

Last but not least sei Thomas Röblitz gedankt, der mich vor vielen Jahren mit dem Satzsystem LaTeX bekannt machte, eine Bekanntschaft, die bis heute anhält und viele Word-Versuchungen überstanden hat.

Inhaltsverzeichnis

Abbildungsverzeichnis **ix**

Vorwort **xi**

Überblick **xiii**

1 Problemstellung und Analyse **1**

1.1 Inhalt des Kapitels 1

1.2 Das Problem 2

1.3 Analyse des Problems 3

1.3.1 Ziel und Methodik der Analyse 3

1.3.2 Die Zielsprache 4

1.3.3 Herstellungsprozeß von Software 4

1.3.4 Verwendung formaler Sprachen 6

1.3.5 Betrachtungen über Schnittstellen 7

1.3.6 Einordnung in den Softwareentwicklungsprozeß 8

1.4 Anforderungen und Kriterien 15

2 Programmiersprachen **17**

2.1 Inhalt des Kapitels 17

2.2 Literatur 17

2.3 Status Quo vorhandener Programmierkonzepte 18

2.3.1 Überblick 18

2.3.2 Java 19

2.3.3 C und C++ 20

2.3.4 Visual Basic 21

2.3.5 Delphi 22

2.3.6 GUI-Generatoren 23

2.3.7 Betriebssystem: Windows 24

2.4 Grundlagen von Programmiersprachen 25
2.4.1 Überblick 25
2.4.2 Allgemeine Anforderungen an Programmiersprachen . 25
2.4.3 Definitionen 26
2.4.4 Definition formaler Sprachen 26
2.4.5 Designkriterien für Programmiersprachen 37

3 Das Konzept 43
3.1 Inhalt des Kapitels 43
3.2 Zielsprache 44
3.3 Konzept des Implementationsprozesses 45
3.4 Schnittstellen 46
3.4.1 Standardverhalten und getroffene Annahmen 48
3.4.2 Datenaustausch 49
3.4.3 Interne Schnittstellen 49
3.4.4 Externe Schnittstellen 52
3.4.5 Schnittstellen der Prüfroutinen 53
3.5 Ausnahmen 54
3.6 Sprache 55
3.6.1 Vorüberlegungen 55
3.6.2 Konventionen 57
3.6.3 Definition der Sprache 58
3.7 Transfer der veränderlichen Programmteile 66
3.8 Zusammenfassung 66

4 Bewertung des Konzepts 69
4.1 Inhalt des Kapitels 69
4.2 Konzept und Problemstellung 70
4.3 Bewertung anhand Anforderungen und Kriterien 70
4.4 Bewertung der Sprache 73
4.4.1 Nutzung vorhandener Ansätze 74
4.4.2 Bewertung als Programmiersprache 75
4.4.3 Zusammenfassung 77
4.5 Nutzen des Konzepts 78
4.6 Zusammenfassung 79

5 Zusammenfassung und Fazit **81**
5.1 Zusammenfassung 81
5.2 Fazit 82

A Erweiterte Backus-Naur-Form **85**
A.1 Entwickelte Programmiersprache 85
A.2 Eine kleines Beispielprogramm 88

B Quellcode der Java-Definitionen **97**
B.1 Schnittstellen 97
B.1.1 EntryElement 97
B.1.2 EntryForm 98
B.1.3 FormChecker 99
B.2 Ausnahmen 100

C Abkürzungen und Glossar **101**

D Literaturangaben **105**

E Index **107**

Erklärungen **109**

Abbildungsverzeichnis

1.1 Herstellungsprozeß von Software 5
1.2 Geänderter Herstellungsprozeß von Software 6
1.3 Das Wasserfallmodell 9

2.1 Ableitungsbaum von Ableitung 2.7 31

3.1 Herstellung der veränderlichen Programmteile 45
3.2 Schnittstellen 47
3.3 Beispiel der Standardoberfläche 48

A.1 Im Beispiel dargestellte Oberfläche 88

Vorwort

Der Einsatz von Computern soll u. a. dabei helfen, Menschen langweilige, weil immer wiederkehrende Arbeitsschritte abzunehmen bzw. zu vereinfachen. Diese Arbeitsschritte können in den meisten Fällen so automatisiert werden, daß der Nutzen des Computereinsatzes den Aufwand zufriedenstellend übersteigt. Andererseits müssen die einzusetzenden Programme vor der Benutzung entworfen und implementiert werden.

Obwohl Implementierung als Kunst angesehen werden kann, ist sie jedoch in weiten Teilen mehr Handwerk als schöpferische Leistung. Nachdem der Entwurf steht, ist dessen Umsetzung meist keine spannende Arbeit mehr, es sei denn, der heutzutage übliche Zeitdruck intensiviert die Arbeit.

Während der Implementierung der Lösungen fallen oft auch Arbeiten an, die fast ebenso langweilig sind wie die Arbeit, die gespart werden soll. In alter Traditionspflege des Handwerks über Jahrhunderte hinweg, wird auch hier oft dazu übergegangen, solche Arbeiten an die billigsten Arbeitskräfte zu übergeben, im Fall der Implementierung sind das meist Studenten.

Diese müssen dann frustriert Routineaufgaben erledigen, die von einem Computer schneller und meist schöner bearbeitet werden könnten und wenn nicht, so könnte ein Computer doch die Routine etwas vereinfachen.

In dieser Situation sah ich mich, als ich beauftragt wurde, einen relativ einfachen Formulareditor innerhalb eines größeren Java-Programms zu implementieren. Zeitdruck ließ jedes größere Konzept hinfällig erscheinen, Schnellprogrammierungskünste waren gefragt. Interessanterweise nahm die Programmierung der graphischen Benutzeroberfläche den größten Teil der Zeit ein, da die Oberfläche mit ca. 120 Eingabefeldern und 200 Checkboxen umfangreich und langweilig zu implementieren war.

Als dann noch angekündigt wurde, daß sich die Oberflächengestaltung circa alle zwei Monate ändern würde, wurde mir klar, daß hier ein anderer Ausweg die auf lange Sicht bessere Lösung sei.

Ich spielte mit dem Gedanken, daß es eine Sprache geben müßte, die eine Java-Oberfläche so beschreibt, daß diese bei jedem Neustart des Programms neu erschaffen wird. Gleichzeitig wollte ich von den Routineaufgaben der ständigen Neuimplementierung der Oberfläche, ihrer Formatierung und der Anpassung der Prüfroutinen entlastet werden.

Aus dieser Idee entstand die vorliegende Diplomarbeit, die sich mit dem angesprochenen Thema beschäftigt.

Überblick

Im Verlauf eines Softwareprojekts wird es des öfteren zu der Situation kommen, daß langwierige, langweilige und damit fehlerbehaftete Aufgaben auszuführen sind. Im allgemeinen wird versucht, diese Aufgaben so automatisiert wie möglich zu erledigen.

Andererseits gibt es in der Praxis Spezialfälle, in denen der Einsatz einer speziell auf das Problem zugeschnittenen Lösung in seiner Realisierung aufwendiger ist als der Einsatz einer allgemeinen Lösung, die dafür allerdings komplizierter in der Anwendung auf das spezielle Problem ist.

Damit ergibt sich ein Konflikt zwischen dem Grad der angestrebten Automatisierbarkeit und der allgemeinen Verwendbarkeit eines Problemlösungsalgorithmus. Somit entstehen Nischen, in denen bisher die allgemeine Verwendbarkeit z. B. einer Programmiersprache höher eingestuft wurde als die Schnelligkeit und Einfachheit des Einsatzes einer speziellen Lösung.

Eine dieser Nischen ist die Programmierung von graphischen Benutzeroberflächen mit einer hohen Zahl von Einzelelementen, deren Funktionalität auf kleinere Eingabeprüfungen beschränkt ist. Diese Programmieraufgabe wird heutzutage zwar von integrierten Entwicklungsumgebungen unterstützt, die hohe Zahl von Eingabeelementen ist jedoch in vielen Fällen äußerst zeitraubend und schwierig zu implementieren.

Diese Diplomarbeit versucht, die Programmierlücke hinsichtlich jener Eingabemasken zu schließen, die hauptsächlich aus Eingabefeldern und Checkboxen bestehen und deren Nutzereingaben in geringem Umfang auf Korrektheit getestet werden sollen.

Dabei soll eine eigene Programmiersprache entwickelt werden, die die Implementierung solcher Eingabeoberflächen erleichtert und gleichzeitig einfache Mechanismen zur Verfügung stellt, um die eingegebenen Werte hinsichtlich vorgegebener Kriterien zu überprüfen.

Hierfür wird in Kapitel 1 das Problem umrissen und in den Gesamtzusammenhang der Softwareentwicklung eingeordnet. Dabei werden Anforderungen und Kriterien aufgestellt, anhand derer die Problemlösung auf Praxistauglichkeit sowie Nähe zum Problem geprüft werden kann.

In Kapitel 2 werden die Grundlagen der zu entwickelnden Programmiersprache gelegt. Dabei wird zunächst allgemein auf das theoretische Fundament von Programmiersprachen eingegangen und danach vorhandene Programmiersprachen oder Lösungsansätze auf ihre Verwendbarkeit im Rahmen der

Diplomarbeit hin untersucht.

Das darauffolgende Kapitel 3 beschäftigt sich mit der Darstellung des Konzepts und hierin hauptsächlich mit der Entwicklung einer eigenen Programmiersprache. Es fließen die Erkenntnisse der ersten beiden Kapitel ein, dargestellt werden wichtige Teile und Prinzipien der Programmiersprache sowie die zugrundeliegenden Ideen.

Das Kapitel 4 dient der Bewertung des Konzepts sowie der Programmiersprache. Hierbei sind die in den beiden ersten Kapiteln herausgearbeiteten Kriterien und Grundlagen von besonderer Wichtigkeit und dienen als Bewertungsmaßstab.

Das Kapitel 5 faßt die ermittelten Ergebnisse noch einmal in knapper Form zusammen.

Im Anhang A findet sich die Erweiterte Backus-Naur-Form (EBNF) der entwickelten Sprache. Um die entwickelte Programmiersprache etwas plastischer darzustellen, wird im Anhang B ihr praktischer Gebrauch anhand eines Beispiels dargestellt. Der Abschnitt C erläutert die im Text verwendeten Abkürzungen und Fachbegriffe. Weiterhin findet sich im Anhang D die zur Erstellung der Arbeit genutzte Literatur sowie im Anhang E der Index.

Im Verlauf der gesamten Diplomarbeit werden bestimmte Begriffe synonym verwendet werden. Dies sind im einzelnen Implementierung und Programmierung, die beide die Überführung eines Softwareentwurfs in Programmquellcode meinen sowie Graphical User Interface (GUI), Oberfläche und Benutzeroberfläche, die alle die graphische Schnittstelle zwischen Anwendung und Nutzer bezeichnen.

1 Problemstellung und Analyse

1.1 Inhalt des Kapitels **1**

1.2 Das Problem **2**

1.3 Analyse des Problems **3**

1.3.1 Ziel und Methodik der Analyse 3

1.3.2 Die Zielsprache 4

1.3.3 Herstellungsprozeß von Software 4

1.3.4 Verwendung formaler Sprachen 6

1.3.5 Betrachtungen über Schnittstellen 7

1.3.6 Einordnung in den Softwareentwicklungsprozeß . . 8

1.4 Anforderungen und Kriterien **15**

1.1 Inhalt des Kapitels

Dieses Kapitel dient dazu, das zugrundeliegende Problem zu untersuchen und zu gliedern, um so die Grundlage für das zu erarbeitende Konzept zu erstellen.

Zunächst wird das Problem überblicksartig dargestellt. Auf dieser Grundlage wird die Problemstellung untersucht. Dabei soll eine Verfeinerung der Darstellung stattfinden, die für die Analyse genutzt werden kann.

Die Analyse beginnt mit der Erläuterung von Ziel und Methodik, es folgen allgemeine Betrachtungen zur Zielsprache und zum Herstellungsprozeß von Software. Diesen schließen sich Betrachtungen über die Verwendung von formalen Sprachen und Schnittstellen an. Am Ende der Analyse steht die Einordnung in den Gesamtzusammenhang der Softwareentwicklung.

Die anschließende Zusammenfassung ordnet die gefundenen Ansätze, Teillösungen, Vorschläge und Kriterien und stellt sie in knapper Form als Liste von Anforderungen und Kriterien noch einmal dar.

1.2 Das Problem

Wie im Vorwort bereits angedeutet wurde, entstand das Problem aus einer realen Situation.

Es stellt sich wie folgt dar:

Ein Programm besitzt eine graphische Benutzeroberfläche. Neben den üblichen Elementen wie Menüs, Buttons usw. enthält das GUI auch eine oder mehrere Eingabemasken.

Diese Eingabemasken bestehen hauptsächlich aus Eingabefeldern, die mittels Tastatur oder Maus gefüllt werden sollen. Ein Beispiel hierfür sind Formulare, bei denen Texte eingegeben sowie Checkfelder angekreuzt werden können.

Die Eingaben, die in den Eingabemasken erfolgen, sollen vorgegebenen Prüfungen unterworfen werden. Die Prüfroutinen umfassen dabei Tests auf Vorhandensein eines Eintrags, Korrektheit von Einträgen und Konsistenzprüfungen der Felder untereinander.

Es besteht die Möglichkeit, daß eine Eingabemaske aus mehreren einzelnen Eingabeblättern besteht, um die Eingabe den Größenverhältnissen von Computern anzupassen, um die Eingabe ergonomischen Gesichtspunkten zu unterwerfen oder um die Eingabe in logischen Gruppen zusammenzufassen. Da die Eingaben dieser verschiedenen Blätter logisch zu derselben Eingabemaske gehören, also innere Beziehungen zwischen diesen Eingaben existieren können, muß die Möglichkeit zugelassen werden, auch diese Daten den Prüfroutinen zu unterwerfen.

Die Elemente einer Eingabemaske ändern sich im Laufe der Einsatzzeit des Programms häufiger. Es können z. B. neue Eingabefelder hinzukommen oder vorhandene obsolet werden. Es ist auch vorstellbar, daß sich die Beschriftung der Felder oder auch die Farbgestaltung des Formulars ändern.

Die Änderung der Oberfläche oder andere Gründe können dazu führen, daß die Prüfroutinen, die die eingegebenen Daten testen, ebenfalls geändert werden müssen. So ist vorstellbar, daß mit fortschreitender Entwicklung eines Programms neue Prüfbedingungen hinzukommen oder alte wegfallen. Damit müssen also die Prüfroutinen im Laufe des Lebenszyklus' des Programms angepaßt werden können.

All diese Änderungen sollen nun so erfolgen, daß der Nutzer nicht das gesamte bisherige Programm ersetzen muß, sondern nur die geänderte Nutzeroberfläche bzw. die veränderten Prüfroutinen. Mit dem nächsten Programmstart oder per speziellem Aufruf durch den Nutzer sollen dann die Änderungen im Programm sichtbar werden. Damit soll u. a. der Implementationsprozeß auf Programmiererseite kurz gehalten werden, auf Nutzerseite ist keine Neuinstallation nötig und die geänderten Programmteile können relativ klein gehalten werden.

Obwohl das Problem zunächst anhand einfacher Eingabeelemente erläutert und bearbeitet wird, darf eine spätere Erweiterung des Konzepts auf kom-

plexere Eingabefelder und -strukturen nicht verwehrt sein. Außerdem soll es möglich sein, einen GUI-Builder als Frontend einzusetzen, der die Beschreibung der Benutzeroberfläche und der notwendigen Prüfroutinen erzeugt.

Beispiel:
Firma A möchte eine Datenbank schaffen, die Produkte verschiedener Firmen beinhaltet. Diese Datenbank soll Nutzern per Internet zugänglich gemacht werden.
Die Datenerfassung soll von den teilnehmenden, produktanbietenden Firmen selbst vorgenommen werden. Dazu stellt Firma A ein entsprechendes Programm zur Verfügung, das Formulare beinhaltet, die von den Firmen ausgefüllt und dann vom Programm an die Datenbank gesandt werden.
Um die Konsistenz der Datenbank zu gewährleisten sollen die eingegebenen Daten innerhalb des Programms anhand vorgegebener Kriterien überprüft werden. Nur Daten, die die Prüfung bestanden haben, werden an die Datenbank weitergeleitet.
Die Beschreibung der Produkte erfolgt zum einen mit freien Eingabefeldern, in die Texte eingetragen werden können, zum anderen aber auch durch Ausfüllen von Checklisten. Die freien Eingabefelder ändern sich mit hoher Wahrscheinlichkeit nicht, die Checklisten sind jedoch einem ständigem Wandel unterworfen.
Es muß damit gerechnet werden, daß die Checklisten jeden Monat neu erstellt werden müssen. Damit ändern sich meist auch die Prüfroutinen, die für die Datenbankkonsistenz verantwortlich sind.
Es besteht also das Problem, daß das Abfrageprogramm jeden Monat neu ausgeliefert werden muß, was für die teilnehmenden Firmen eine große Belastung bezüglich der Koordinierung der Updates bedeutet.
Die Wunschvorstellung von Firma A ist, daß das Programm selbständig die Aktualisierung der Formulardaten vornimmt und nach einem Neustart die aktuellen Formulare anzeigt.

1.3 Analyse des Problems

1.3.1 Ziel und Methodik der Analyse

Die Analyse wird einzelne Teilaspekte definieren, innerhalb derer das Problem untersucht und abgegrenzt wird. Diese Vorgehensweise erklärt sich daraus, daß die verschiedenen Sichtweisen auf das Problem diverse Kriterien enthalten können, anhand derer das Problem analysiert und gelöst werden muß.

Das Ziel ist, Anforderungen an die Problemlösung zu definieren, gegenüber denen die Lösung abgeglichen werden kann.

Die Analyse wird innerhalb der verschiedenen Teilaspekte Redundanzen enthalten, die bewußt im Text belassen werden, da jeder Teilaspekt für sich vollständig analysiert werden sollte. Die abschließende Zusammenfassung wird diese Redundanzen auflösen und einheitliche Analyseergebnisse aufzeigen.

1.3.2 Die Zielsprache

Die Problemstellung gibt vor, daß ein Programm existiert, dessen Benutzeroberfläche zu beschreiben ist.

In der Problemstellung ist noch keine Aussage darüber getroffen worden, in welcher Sprache dieses Programm und damit die Benutzeroberfläche implementiert werden. Diese Sprache bezeichnen wir als *Zielsprache*.

Die Zielsprache sollte eine für Programmierer, Entwickler sowie Theoretiker „schöne" Programmiersprache sein. Dies gilt sowohl in Bezug auf mögliche Sprachkonstrukte als auch hinsichtlich der möglichst guten Implementierung verschiedener Algorithmen und Oberflächenelemente.

Zweitens sollte die Zielsprache die Möglichkeit bieten, klar objektorientiert zu programmieren. Dies ist notwendig, da die Implementation von Schnittstellen eine Grundvoraussetzung für das Konzept ist, wie weiter unten im Abschnitt über Schnittstellen (Kapitel 1.3.5) gezeigt wird.

Drittens wäre wünschenswert, daß das hier erarbeitete Konzept so breit gefächert wie möglich genutzt werden kann. Größtmögliche Plattformunabhängigkeit der Zielsprache erscheint daher als äußerst wünschenswert.

1.3.3 Herstellungsprozeß von Software

Software entsteht, wie in der vereinfachten Darstellung von Abbildung 1.1 zu sehen ist, aus einer Spezifikation. Diese wird vom Programmierer in Programmcode umgesetzt. Dabei nutzt dieser meist ein Integrated Development Environment (IDE), das ihn bei der Programmierung unterstützt.

Aus praktischen Erwägungen wird im allgemeinen zwischen Anwendungsprogrammierung und Herstellung der graphischen Benutzeroberfläche unterschieden. Der Anwendungsteil beinhaltet dabei die dem Programm zugrundeliegende Logik und alle Funktionen des Programms; das GUI realisiert die Verbindung zwischen der Anwendung und dem Nutzer. Neuere IDEs unterstützen diese Trennung und können dabei dennoch die Verbindung von Oberfläche und Anwendungsquellcode eng halten.

Nach der vollständigen Programmierung von Anwendungs- und GUI-Quellcode werden diese übersetzt und im Bedarfsfall gelinkt. Das entstehende Programm durchläuft eine oder mehrere Testphasen, die in der Abbildung nicht gezeigt werden, danach wird das fertige Programm an den Nutzer weitergegeben.

Etwaige Änderungen am Aussehen der Benutzeroberfläche bzw. den Prüfroutinen nach Weitergabe des Programms an den Nutzer bewirken Veränderungen im Entwurf des Programms. Das bedeutet, daß der Programmierer die Änderungen in den bestehenden Anwendungs- und GUI-Quellcode einarbeitet und diesen compiliert und linkt. Der Nutzer bekommt dann das komplette geänderte Programm geliefert.

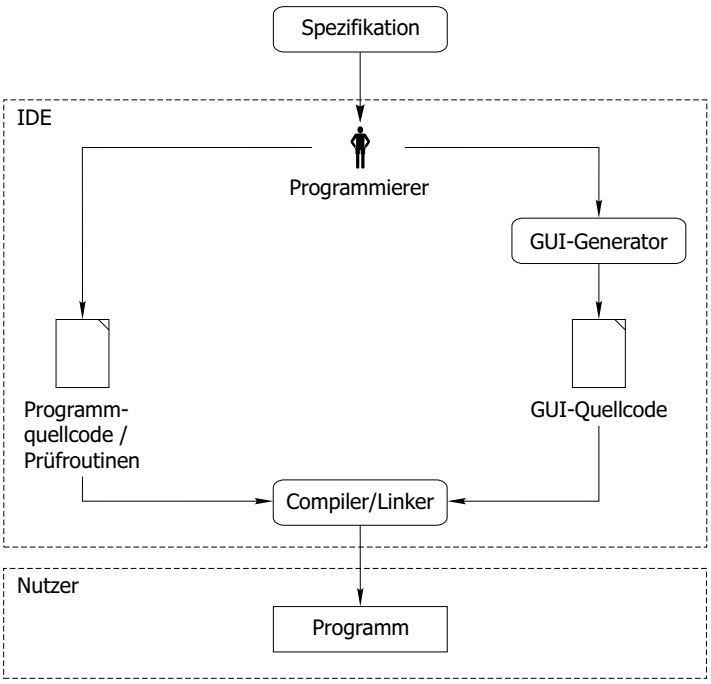

Abbildung 1.1: Herstellungsprozeß von Software

Quelle: Eigene Darstellung

Dieser Weg ist in den meisten Anwendungsfällen erprobt und als praktikabel übernommen worden, da Änderungen der Spezifikation nur selten erfolgen. Diese Änderungen umzusetzen, das entstandene Programm zu testen und vollständig an den Nutzer auszuliefern, ist daher im Verhältnis zur Häufigkeit der Änderung der einfachere Weg.

Unsere Problemstellung beinhaltet geplante, häufige Änderungen von Teilen der Oberfläche oder der zugehörigen Prüfroutinen. Es ist daher zweckmäßig, die Programmierung des Anwendungsquellcodes sowie der graphischen Benutzeroberfläche so aufzuteilen, daß die unveränderlichen Programmbestandteile von den oft zu verändernden getrennt werden. Wir erhalten also eine Situation nach Abbildung 1.2.

Der abgetrennte unveränderliche Anwendungsquellcode sowie die unveränderlichen Teile der Nutzeroberfläche werden unabhängig von den veränderlichen Teilen erstellt, compiliert und gelinkt. Wir erhalten den Programmrumpf, den unveränderlichen Teil des Programms.

Die GUI-Elemente und die Prüfroutinen, die in der Problemstellung als veränderlich beschrieben werden, durchlaufen einen anderen Prozeß. Sie werden in der im Konzept vorgestellten neuen Programmiersprache codiert und in separaten Dateien abgelegt. Schließlich werden der Programmrumpf und die externen Dateien an den Nutzer ausgeliefert.

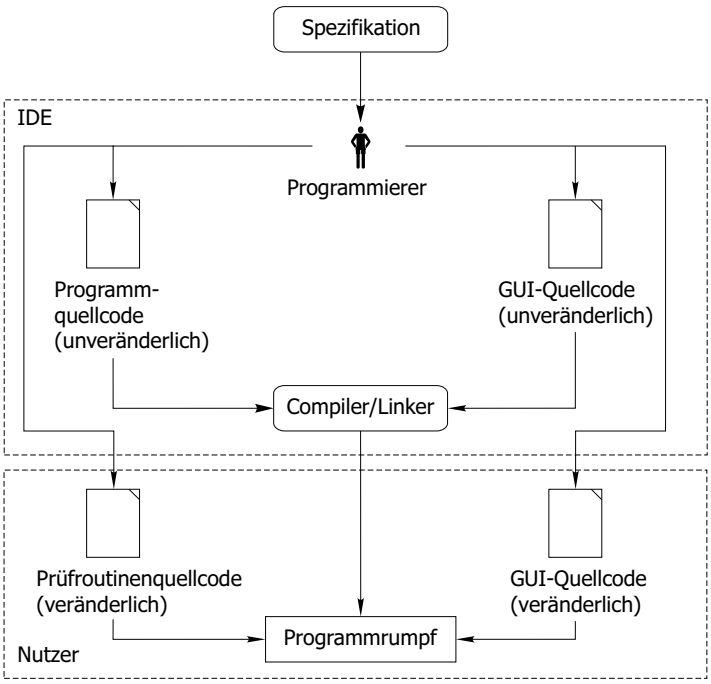

Abbildung 1.2: Geänderter Herstellungsprozeß von Software

Quelle: Eigene Darstellung

Bei jedem Start des Programmrumpfs werden die externen Programmteile von diesem eingelesen, in die graphische Benutzeroberfläche sowie Prüfroutinen übersetzt und im Dialog bereitgestellt.

Jegliche Änderungen, die zu einem späteren Zeitpunkt an den als veränderlich vorgesehenen Anwendungsteilen vorgenommen werden, brauchen daher nur neu als Quellcode programmiert zu werden. Der entstehende Quellcode wird an den Nutzer übermittelt, die bisherigen, nun veralteten Dateien werden durch die neuen ersetzt. Mit dem nächsten Programmaufruf oder durch explizites Einlesen der neuen Definitionen werden die geänderten Programmteile dann dynamisch in das Programm eingebunden.

Die Trennung der Implementierung des gesamten Programms in die Programmierung des veränderlichen und des unveränderlichen Quellcodes eröffnet die Möglichkeit, in den weiteren Betrachtungen nur noch die Programmierung der veränderlichen Teile zu berücksichtigen, da der Programmrumpf wie bisher üblich programmiert werden kann.

1.3.4 Verwendung formaler Sprachen

Die Problemstellung definiert zwei getrennte Aufgaben, die gelöst werden müssen:

Einerseits ist eine Benutzeroberfläche zu erstellen, andererseits soll diese Oberfläche mittels bestimmter Prüfroutinen getestet werden.

Außerdem werden Zusatzbedingungen genannt, die berücksichtigt werden sollen, so zum einen die Möglichkeit der Umgestaltung der Oberfläche, die nicht mit der Neuübersetzung des zugrundeliegenden Programmrumpfes einhergehen soll. Zum anderen darf die gefundene Lösung nicht in sich starr sein, spätere Erweiterungen hinsichtlich neuer Eingabefelder, eigener Objekte oder komplexerer Strukturen sollten nicht konzeptionell verwehrt werden.

Somit kann die Annahme getroffen werden, daß zwei Sprachen das Problem lösen können. Eine Sprache dient zur Darstellung der Eingabeoberfläche, die andere zur Darstellung der Prüfbedingungen. Diese Sprachen müssen, wie im folgenden nicht weiter explizit erwähnt wird, formale Sprachen im Sinne der Sprachtheorie sein. Das bedeutet, sie müssen Syntax und Semantik besitzen, die mittels eines Computerprogramms überprüft werden können.

Als Erweiterung dieser Annahme bleibt zu klären, inwieweit es möglich ist, beide Sprachen in eine gemeinsame Sprache zu überführen. Die Vorteile einer gemeinsamen Sprache müssen gegen die Nachteile abgewogen und der Nutzen bewertet werden.

Beide Sprachen sind dafür zuständig, die Prüfroutinen sowie den veränderlichen GUI-Quellcode (Abb. 1.2) zu erzeugen, die dem Programmrumpf zur Verfügung gestellt werden.

1.3.5 Betrachtungen über Schnittstellen

Die Trennung von Programmrumpf (unveränderlicher Programm- und GUI-Quellcode) und externem Quellcode (veränderliche GUI-Teile und zugehörige Prüfroutinen) kann mit den oben getroffenen Betrachtungen als Grundvoraussetzung zur Problemlösung gelten.

Bereits bei den Betrachtungen über die Zielsprache des Konzepts wurde darauf hingewiesen, daß Objektorientierung und damit die Idee der Schnittstellen für unser Konzept wichtig sind. Die Objektorientierung bietet nämlich die Möglichkeit, die angesprochene Trennung leicht und elegant durchzuführen.

Das mit der Trennung entstehende Problem ist, daß der Programmrumpf unabhängig vom konkreten Inhalt der externen variablen Quelltexte die Oberfläche und die Prüfroutinen einbinden und sinnvoll nutzen muß.

Wir betrachten den Programmrumpf als eigenständige Einheit, die den Code einzulesen, zu übersetzen und die variablen Programmteile zu generieren hat. Dabei benutzt der Programmrumpf fest vorgegebene Schnittstellen, die von den variablen Programmteilen implementiert werden müssen. Mit sinnvoller Definition dieser Schnittstellen haben wir die Möglichkeit, die Funktion des Gesamtprogramms einfach sicherzustellen.

Die Aufgabe während der Problemlösung muß nun sein, diese Schnittstellen sinnvoll zu definieren, d. h. die Definition muß alle noch zu findenden rele-

vanten Bedingungen beinhalten, andererseits aber auch beherrschbar bleiben, d. h. es dürfen nicht wahllos überflüssige Details in den Schnittstellendefinitionen enthalten sein.

Die angesprochenen relevanten Bedingungen an die Schnittstellen sollen hier grob umrissen werden:
Während des Programmlaufs müssen die Oberflächenelemente, soweit dies sinnvoll ist, die Manipulation von Farbe und Beschriftung zur Verfügung stellen. Weitere Manipulationsmöglichkeiten, z. B. Schriftart oder -größe, sind an dieser Stelle nicht vorgesehen, müssen jedoch als spätere Erweiterung im Konzept berücksichtigt werden. Die Eingabeelemente sollen darüber hinaus Werte annehmen und eingegebene Werte zurückgeben können.

Außerdem muß die Möglichkeit bestehen, Prüfroutinen aufzurufen, die alle eingegebenen Daten den im Prüfroutinenquellcode angebenen Test unterziehen. Dabei werden in einer statischen Auswertung alle in einem Formular eingegebenen Daten gleichzeitig getestet, die Fehlerauswertung und der Prüfzeitpunkt unterliegen dabei der Kontrolle des Programmrumpfs. Die dynamische Komponente der Prüfbedingungen umfaßt das Ein- und Ausblenden sowie automatisches Ausfüllen von GUI-Elementen. Die Dynamik soll dabei unabhängig vom Programm nur von den Nutzereingaben abhängen.

Das Verhalten der Benutzeroberfläche ist so festgelegt, daß Daten jederzeit eingegeben werden können. Der Programmrumpf hat darüber hinaus die Möglichkeit, die Oberflächenelemente in der mittels der Schnittstellen festgelegten Form zu manipulieren. Die variablen GUI-Bestandteile können also nur Daten aufnehmen, ausgeben und die Oberflächenelemente manipulieren.

Die Fehlerbehandlung wird zwischen Programmrumpf und generiertem GUI derart aufgeteilt, daß die GUI-Elemente die Möglichkeit bereitstellen, in einen Fehler- oder einen Warnzustand versetzt zu werden. Die Reaktion auf eine Zustandsänderung bleibt den einzelnen Elementen überlassen. Der Programmrumpf muß diese Möglichkeit nicht nutzen, da er die GUI-Elemente auch über die bisher angesprochenen Schnittstellen manipulieren kann.

Des weiteren müssen die Schnittstellen die Generierung der veränderlichen Programmteile aus dem externen Quellcode zur Verfügung stellen. Diese erfolgt durch Aufruf aus dem Programmrumpf, der über den Einlese- und Generierungszeitpunkt entscheidet.

1.3.6 Einordnung in den Softwareentwicklungsprozeß

In diesem Abschnitt sollen Erkenntnisse über den Umfang des zu entwickelnden Konzepts und die Relevanz einzelner Softwareentwicklungsphasen für das Konzept ermittelt werden.

Der Ablauf der Softwareentwicklung wird bei Balzert (vgl. [3], S. 99 ff. sowie [2] gesamtes Buch), der sich in diesen Büchern hauptsächlich mit Softwareentwicklung beschäftigt, unter anderem anhand des Wasserfallmodells (Abbildung 1.3) erklärt.

Das Wasserfallmodell ist ein sehr einfaches, allgemeingültiges Modell, das den prinzipiellen Ablauf der Softwareentwicklung darstellt, für die praktische Anwendung muß es beträchtlich erweitert werden. Für die hier getroffenen Überlegungen ist das Modell jedoch sehr gut geeignet, da die wichtigen Phasen der Softwareentwicklung enthalten sind und die Darstellung trotzdem sehr übersichtlich und verständlich bleibt.

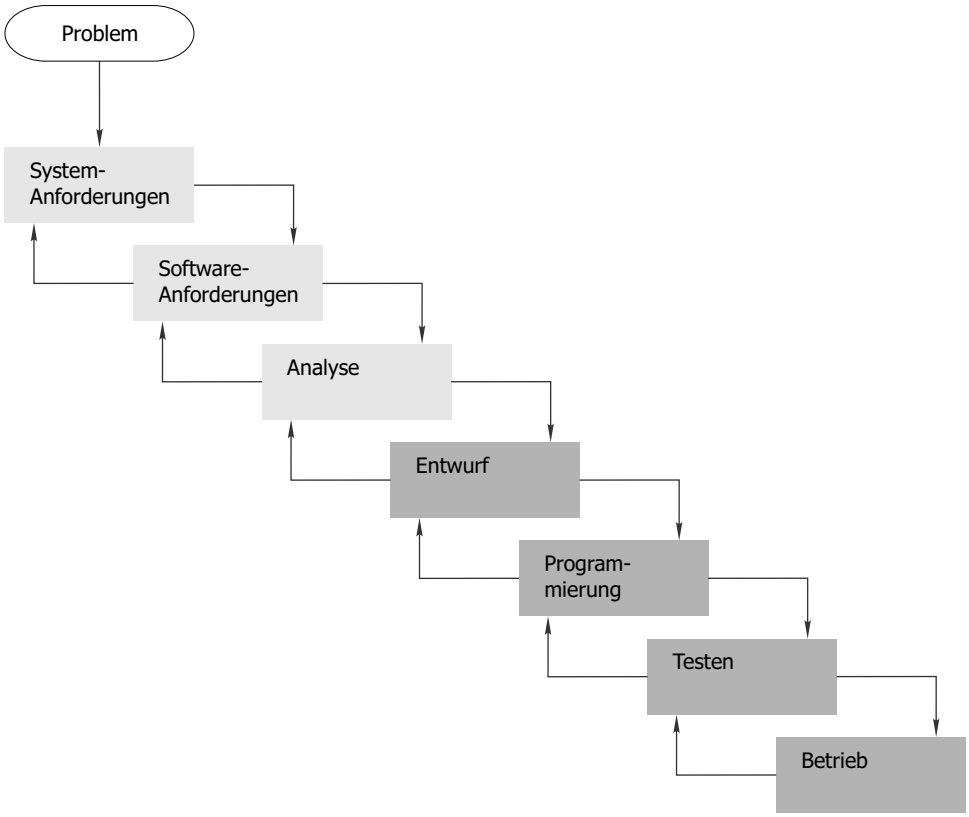

Abbildung 1.3: Das Wasserfallmodell

Quelle: Eigene Darstellung nach Balzert [3], S. 99

Das Wasserfallmodell erklärt den Softwareentwicklungsprozeß so, daß dessen einzelne Stufen (auch Phasen genannt) in einer Hierarchie angeordnet sind, die während der Softwareherstellung von links oben nach rechts unten durchlaufen werden. Dabei ist eine Rückkopplung zwischen jeweils benachbarten Stufen möglich.

Die Stufen *Systemanforderungen*, *Softwareanforderungen* und *Analyse* benennen die Anforderungen an das zu entwickelnde System und nehmen in Form eines Produktmodells eine erste Modellierung vor.

Die Stufe *Entwurf* umfaßt den gesamten Softwareentwurf ausgehend von den Resultaten der Analyse. Die Resultate beinhalten Softwaremodelle und Klassendefinitionen bis hin zu detaillierten Anwendungs- und Implementations-

modellen. Die Stufe sollte daher näher untersucht werden.

Die Stufe *Programmierung* sieht vor, die Ergebnisse der Entwurfsphase in konkreten Quellcode umzusetzen. Daher berührt die Problemstellung auch diese Entwurfsstufe. Das Ergebnis der Programmierung ist das fertige Softwareprodukt, das dem Test zugeführt wird.

Der *Test* dient dazu, das fertige Softwareprodukt gegen die Produktdefinition zu testen. Die Problemstellung verlangt nach speziellen Tests, um die veränderlichen Programmteile und den Programmrumpf ausreichend mit der Produktdefinition abzugleichen. Die Testphase muß daher ebenfalls untersucht werden. Die Testphase findet mit der Inbetriebnahme des fertigen Produkts ihren Abschluß.[1]

Der *Betrieb* der Software beinhaltet die Nutzung und Anwendung des Programms oder Programmpakets durch den Nutzer. Die Problemstellung berührt auch die Betriebsphase, da die geänderten Programmteile während der Betriebsphase integriert werden müssen.

Wir können also anhand des Wasserfallmodells die für unsere Problemstellung näher zu untersuchenden Entwicklungsphasen bestimmen. Dies sind die Phasen Entwurf, Programmierung, Testen und Betrieb, die in Abbildung 1.3 dunkelgrau hinterlegt sind. Im folgenden werden alle Phasen einzeln betrachtet und hinsichtlich ihrer Auswirkungen auf die Lösung der Problemstellung sowie auf Anforderungen an die Problemlösung untersucht.

Softwareentwicklungsphasen: Systemanforderungen, Softwareanforderungen und Analyse

Diese in Abbildung 1.3 hellgrau hinterlegten Phasen dienen der Festlegung der Anforderungen an das zu entwickelnde System und helfen, eine Problemstellung zu analysieren und erstmalig zu modellieren.

Insgesamt sind diese Entscheidungen vorbereitender Natur und werden daher im Kapitel 3 nicht weiter ausführlich untersucht.

Die für das hier vorgestellte Konzept wichtigen Punkte stellen sich wie folgt dar:

1. Im Konzept wird eine Zielsprache festgelegt werden müssen. Dies zieht Konsequenzen für System- und Softwareanforderungen nach sich. Auch die Analyse des Problems muß die festgelegte Zielsprache berücksichtigen.

2. Die Trennung der veränderlichen von den unveränderlichen Programmteilen ist eine Forderung des Konzepts, die schon in diesen frühen Phasen wichtig ist.

3. Es muß die Möglichkeit gegeben sein, die veränderlichen Programmteile

[1] Leider ist dies den meisten Softwareproduzenten nicht bewußt, die die Betriebsphase des Produkts als Testphase mißbrauchen.

an den Nutzer weiterzugeben, dieser muß die übertragenen Teile problemlos einbinden können.

4. Die Eingabefelder können zunächst einfach gehalten sein, es muß jedoch auch kombinierte bzw. komplexe Eingabefelder geben können. Die Prüfung der Eingabe muß für alle Eingabefelder ermöglicht werden.

5. Die Analysephase darf nicht durch Vorgaben des Konzepts unzulässig eingeschränkt werden. Erweiterungsmöglichkeiten müssen für alle festgelegten Einschränkungen vorgesehen werden, ein Beispiel sind veränderte Eingabefelder bzw. -strukturen.

6. Die Beschreibung der Benutzeroberfläche sollte unabhängig von der Implementierung des Programms sein, dies bedingt feste Schnittstellen, die bereits in der Analysephase berücksichtigt werden müssen.

Softwareentwicklungsphase: Entwurf

Die Entwurfsphase schließt sich an die Analysephase an. Als Ausgangspunkt dient das aus der Analyse entstandene Produktmodell, das die Anforderungen des Nutzers widerspiegelt. Ziel der Entwurfsphase ist es, eine Produktarchitektur für das gesamte Produkt zu entwickeln. Es wird also ein Produktmodell in eine Produktarchitektur umgewandelt.

Zu beachten ist dabei, daß das Produktmodell im allgemeinen noch die Anforderungen des Nutzers ohne technische Beschränkungen oder Implementationsdetails widerspiegelt. Erst die Produktarchitektur muß gewählte Implementationsvorgaben berücksichtigen. Dabei wird jedoch die eigentliche Programmierung über formalisierte Modellierungsmethoden gekapselt. Daher ist es normalerweise ausreichend, eine Modellierungsmethodik zu wählen und in dieser das Produktmodell zu beschreiben.

Die Aufgabe der Problemlösung ist es daher, Möglichkeiten zu schaffen, die es erlauben, spezielle Lösungsansätze in einer Modellierungsmethodik darstellen zu können.

Für unser Problem ergeben sich daraus die folgenden Kriterien, die in der Entwurfsphase berücksichtigt werden müssen:

1. Die zu erstellenden Eingabemasken müssen formalisiert dargestellt werden können, damit sie durch eine Modellierungsmethodik erfaßt werden können.

   Dabei ist natürlich besonders bei Eingabemasken zu bedenken, daß hier oft mit Prototypen gearbeitet wird, die meist mit einem GUI-Builder oder einem Graphikprogramm erstellt werden.

   Die Unterstützung externer GUI-Builder oder die Entwicklung eines eigenen ist dabei nicht unbedingt notwendig, kann aber als Wunschkriterium angesehen werden.

2. Ein weiterer Bestandteil des Problems ist die Prüfung von Eingabebedingungen für einzelne bzw. kombinierte Eingabefelder. Hierbei werden die im Produktmodell aufgestellten Bedingungen für die Eingabe in die Architektur eingebunden.

   Um systematisches Design zu ermöglichen, muß auch die Formulierung von Bedingungen in einer formalen Sprache erfolgen. Nur so kann sichergestellt werden, daß das Design mittels einer geeigneten Modellierungsmethode durchgeführt werden kann.

3. Der Nutzer darf in seiner Entwurfstätigkeit nicht von vornherein durch Restriktionen im Entwurf eingeschränkt werden. Daher ist es für die Problemlösung wichtig, so offen wie möglich zu bleiben.

   Dies bedeutet also, daß die Beschränkung auf einfache Eingabefelder und -strukturen nur vorläufig sein darf und komplexere Eingabeobjekte nicht von vornherein ausgeschlossen werden dürfen. Die Problemlösung muß also in diesem Bereich möglichst flexibel sein.

4. Die Forderung, daß die Beschreibung der Benutzeroberfläche unabhängig von der Implementierung des Programms sein soll, beinhaltet, daß beim Softwareentwurf nur gegen die Schnittstellen dieses Konzepts programmiert werden darf. Damit werden die Schnittstellen bereits an dieser Stelle berücksichtigt. Diese Berücksichtigung ist allerdings an dieser Stelle nur schwach, sie berührt in größerem Maße die Implementationsphase.

Die beiden ersten Punkte geben damit eine erste Begründung für die in Abschnitt 1.3.4 getroffene Annahme, daß formale Sprachen zur Problemlösung notwendig sind.

Außerdem erlaubt es der Einsatz formaler Sprachen, Werkzeuge zur Generierung von Quellcode, der in diesen Sprachen geschrieben ist, zu erstellen. Damit besteht die Möglichkeit, die Programmierung der Benutzeroberfläche und der Prüfbedingungen innerhalb eines einzigen Werkzeugs anzubieten.

Softwareentwicklungsphase: Programmierung

Die Programmierungsphase schließt sich direkt an die Entwurfsphase an und nutzt deren Ergebnisse, um aus diesen automatisiert oder manuell Programmcode zu erzeugen.

Dabei wird die automatisierte Programmierung, oft auch als Codegenerierung bezeichnet, der manuellen Programmierung, soweit dies möglich ist, vorgezogen. Damit können Fehler vermieden und die Testphase kurz und erfolgreich gehalten werden.

Die aus der Entwurfsphase stammende Produktarchitektur muß also in entsprechenden Programmcode umgewandelt werden. Für unser Problem ist die Sprache des Programmcodes zum einen die Zielsprache des Konzepts, soweit die unveränderlichen Programmteile betroffen sind. Die veränderlichen Teile

müssen in den zu entwickelnden formalen Sprachen programmiert werden, die vom Programmrumpf in die Zielsprache überführt werden.

Für die Programmierungsphase ergeben sich folgende Kriterien:

1. Es muß möglich sein, Code automatisch von entsprechenden Werkzeugen generieren zu lassen. Aus dieser Forderung folgt, daß der Quellcode der veränderlichen Programmteile in einer formalen Sprache geschrieben sein muß, damit eine Syntaxprüfung ermöglicht werden kann.

2. Dies ist auch bei der manuellen Programmierung wichtig, da hier ebenfalls die syntaktische Prüfung des geschriebenen Programmstücks möglichst vor den eigentlichen Programmtests erfolgen sollte.

3. Die entwickelten Sprachen müssen für den Programmierer nützlich sein. Dies gilt sowohl für die Syntax, die sich an bestehenden Sprachen und Sprachkonstrukten orientieren sollte als auch für die Semantik, die sinnvoll und klar definiert sein muß.

   Damit wird u. a. sichergestellt, daß die erarbeitete Lösung Praxisrelevanz erhält, da erfahrungsgemäß bekannte Sprachkonstrukte gerne weiterverwendet werden.

4. Die Schnittstellen der Oberflächenelemente müssen klar definiert sein, damit sie zum einen in Programmgeneratoren eingesetzt werden können und zum anderen bei manueller Programmierung schnell und einfach in den Programmcode integriert werden können.

   Gleiches gilt für die Schnittstellen der Prüfroutinen.

Auch die Analyse der Programmierungsphase unterstützt demnach die Forderung, daß formale Sprachen eingesetzt werden sollten. Außerdem werden erste Anforderungen an die zu erzeugenden Schnittstellen aufgestellt.

Softwareentwicklungsphase: Testen

Das Testen bildet den Abschluß der Programmentwicklung, es dient dazu, das aus der Programmierung stammende Produkt zu überprüfen, damit beim Betrieb des Programms keine Störungen mehr auftreten können.

Im allgemeinen wird beim Programmtest zwischen Verifikation und Validation unterschieden. Verifikation meint die Prüfung des Produkts gegen seine Spezifikation, beantwortet also die Frage: „Wird ein korrektes Produkt entwickelt?“[2] Die Validation hingegen prüft, ob das entwickelnde Produkt dem vorgesehenen Einsatzzweck genügt, hier lautet die Frage: „Wird das richtige Produkt entwickelt?“[3]

Für unser Problem ergeben sich folgende Kriterien:

[2] Balzert [3], S. 102.
[3] Ebenda.

1. Der Test der Benutzeroberfläche und der Prüfroutinen sollte darauf beschränkt werden zu testen, ob die Schnittstellen korrekt angesprochen werden. Die Funktion von Oberfläche und Prüfroutinen sollte bei syntaktischer Korrektheit des Programmcodes deterministisch nachvollziehbar sein, das Produkt wird also verifiziert.

   Damit ergibt sich die Forderung, daß korrekte Nutzung der Eingabeelemente und Prüfroutinen auf jeden Fall zu dem gewünschten Verhalten führt.[4] Dies ist eine Voraussetzung für die Validierung, die im Gesamttest des Programms durchgeführt wird und nicht speziell im Konzept berücksichtigt werden muß.

2. Die gefundene Problemlösung soll sowohl bei der Definition der Sprachen als auch der Schnittstellen die für Eingabeelemente üblichen Tests unterstützen.

   Dies bedeutet für den Programmierer, daß alle Werte aus Eingabeelementen ausgelesen, eingeschrieben und diese Werte durch die Prüfroutinen verifiziert werden können. Außerdem müssen alle Oberflächenmanipulationen funktionieren. Die eigentlichen Tests müssen also die zur Verfügung stehenden Schnittstellen nutzen und überprüfen können.

3. Als Wunschkriterium sei der Vollständigkeit halber noch genannt, daß bei einem möglichen Entwicklungswerkzeug Simulationen des Programmablaufs, des Verhaltens der Eingabemasken und des Verhaltens der Prüfroutinen möglich sein sollten, diese sind jedoch nicht mehr Bestandteil der Diplomarbeit.

   Ein weiteres Wunschkriterium ist die Möglichkeit, automatische Testmethoden zu schaffen, die selbständig Teile der Oberfläche oder der Prüfroutinen testen können.

Softwareentwicklungsphase: Betrieb

Die Betriebsphase sollte den größten Teil des Lebenszyklus' eines Programms bilden. Das Programm wird dem Nutzer übergeben, der es einsetzt und dadurch für ihn positive Resultate erzielt.

Wie in der Problemstellung festgelegt wurde, kommt es in unserem Fall während der Betriebsphase zu geplanten Veränderungen der Benutzeroberfläche bzw. der Prüfroutinen. Diese unterscheiden sich von nicht geplanten Änderungen dadurch, daß sie bereits beim Softwareentwicklungsprozeß berücksichtigt und in den Entwurf eingebunden wurden.

Die Aufgabe besteht also darin, die geänderten Programmteile so einfach, sicher und schnell wie möglich zu erstellen und dem Nutzer zu übermitteln. Des weiteren muß es dem Nutzer möglich sein, die geänderten Programmtei-

[4] Auch wenn diese Forderung selbstverständlich erscheint, ist sie doch bei vielen (auch kommerziellen) Produkten nicht erfüllt.

le entweder per Aufruf einer Programmethode oder per Programmneustart in das laufende Produkt einzubinden.

Das Konzept muß also berücksichtigen, daß die geänderten Programmteile dem Nutzer übermittelt werden, und dieser die Programmteile in das laufende Programm einbinden kann.

Ergebnisse

Wir erhalten aus der Einordnung in den Softwareentwicklungsprozeß mehrere Anforderungen und Kriterien, die zur Definition bzw. Überprüfung der Problemlösung nötig sind.

In fast allen Phasen ist die Forderung nach der Nutzung formaler Sprachen aufgetreten. Damit wird die in Abschnitt 1.3.4 getroffene Annahme bestätigt, daß der Einsatz formaler Sprachen notwendig ist, die die Aufgabe haben, sowohl die GUI-Elemente als auch die Prüfroutinen zu kapseln.

Bei der Untersuchung der Schnittstellen wurde bestätigt, daß das Konzept der Objektorientiertheit und darauf aufbauend das des Einsatzes von Schnittstellen notwendig ist.

Die Forderung nach Werkzeugen, die die Erstellung und den Test der Oberfläche möglich machen, erscheint zunächst als selbstverständlich. Die Realisierung dieser Forderung bleibt jedoch vorerst ein Wunschkriterium, da die Formalisierung der Lösung den Einsatz von Werkzeugen zwar unterstützt, diese aber noch geschaffen werden müssen. Dies ist nicht mehr Aufgabe dieser Diplomarbeit, stellt jedoch einen wichtigen Schritt zur weiteren Nutzung des Konzepts dar.

In allen Phasen tritt explizit oder implizit die Forderung auf, daß die Lösung sinnvoll und einfach in bestehende Arbeitsabläufe integriert werden muß. Diese Forderung wird implizit in allen Lösungen berücksichtigt werden müssen, da komplizierte oder unsinnige Lösungen selbstverständlich nicht entwickelt werden sollen.

1.4 Anforderungen und Kriterien

Als Ergebnis der Analyse liegen nun mehrere Anforderungen bzw. Kriterien vor, die als Basis Problemlösung verwendet werden können. Sie werden an dieser Stelle in knapper Form dargestellt, um einen schnellen Überblick und damit gute Vergleichsmöglichkeiten zu bieten.

1. Die Entscheidung, das vorgestellte Konzept zu berücksichtigen, muß in der Analysephase gefällt werden, berührt dann aber die vorhergehenden Stufen System- und Softwareanforderungen. Ein Beispiel dafür ist die notwendige Festlegung auf eine Zielsprache, deren Verwendung z. B. Einfluß auf sowohl Software- als auch Hardwareanforderungen hat.

2. Die Definition des Aussehens der Benutzeroberfläche erfolgt durch eine formale Sprache.

3. Die Definition der Prüfroutinen erfolgt ebenfalls durch eine formale Sprache.

4. Diese beiden Sprachen müssen nicht unbedingt getrennt sein, da wir später sehen werden, daß eine gemeinsame Sprache größere Vorteile bietet.

5. Die EBNF für jede (oder die gemeinsame) Sprache ist anzugeben.

6. Die Definition des Aussehens wird nach dem Programmstart oder per explizitem Aufruf vom Programmrumpf eingelesen und zur dynamischen GUI-Generierung genutzt. Die Definition der Prüfroutinen wird ebenfalls eingelesen und zur Überprüfung der eingegebenen Werte verwendet.

7. Die Schnittstellen der zu erstellenden Klassen sind fest definiert. Sie sind so festzulegen, daß sie auch bei Erweiterung des Objektumfangs ihre Gültigkeit nicht verlieren.

8. Die Schnittstellen bieten nur Zugriff auf die für das Konzept relevanten Attribute und Methoden.

9. Die Oberflächenelemente können während des Programmlaufs über Farbe und Beschriftung manipuliert werden. Weitere Kriterien werden nicht ausgeschlossen.

10. Den Eingabeelementen können durch das Programm Werte zugewiesen werden, ebenso können aus Eingabeelementen Werte ausgelesen werden.

2 Programmiersprachen – Bewertung und Grundlagen

2.1 Inhalt des Kapitels **17**
2.2 Literatur **17**
2.3 Status Quo vorhandener Programmierkonzepte **18**
2.3.1 Überblick 18
2.3.2 Java 19
2.3.3 C und C++ 20
2.3.4 Visual Basic 21
2.3.5 Delphi 22
2.3.6 GUI-Generatoren 23
2.3.7 Betriebssystem: Windows 24
2.4 Grundlagen von Programmiersprachen **25**
2.4.1 Überblick 25
2.4.2 Allgemeine Anforderungen an Programmiersprachen 25
2.4.3 Definitionen 26
2.4.4 Definition formaler Sprachen 26
2.4.5 Designkriterien für Programmiersprachen 37

2.1 Inhalt des Kapitels

Dieses Kapitel bietet im ersten Teil einen Überblick über den Status Quo der GUI-Programmierung mittels existierender Hochsprachen. Dabei werden spezielle GUI-Generierungssprachen aber auch Konzepte normaler Programmiersprachen untersucht.

Der zweite Teil des Kapitels beschäftigt sich mit den Grundlagen von Programmiersprachen. Es werden die theoretischen Grundlagen erläutert sowie praktische Kriterien zur Erstellung von Programmiersprachen betrachtet.

2.2 Literatur

Die zur Verfügung stehende Literatur zum Thema Zwischensprachen oder GUI-Generierung ist leider sehr selten und sehr schwer zu finden.

So werden bestehende Programmiersprachen entweder so ausführlich vorgestellt, daß dahinterstehende Konzepte nicht erkennbar sind oder die Konzepte werden so allgemein erläutert, daß die konkrete Umsetzung in einer Sprache nicht mehr deutlich zu erkennen ist.

Das Thema GUI-Generatoren scheint von Veröffentlichungen fast vollständig ausgenommen zu sein. Es konnte lediglich eine Arbeit (Hofmann [8]) dazu gefunden werden, die die zugrundeliegenden Konzepte erläutert. Die konkrete Umsetzung der Konzepte wird aber ebenfalls nicht angegeben, so daß aus Fehlern oder positiven Ansätzen nicht gelernt werden kann.

Das Thema der Entwicklung eigener Programmiersprachen scheint mit dem Erscheinen von allgemeinen Richtlinien und Büchern uninteressant geworden zu sein, so daß eine Einarbeitung in das Gebiet erschwert wird. Die vorliegende Literatur ist durchweg sehr alt, konkret vorgestellte Sprachen existieren meist gar nicht mehr.

Die Grundlagen formaler Sprachen sind das einzige Thema, zu dem sehr viel sehr gute Literatur existiert. Ein Vergleich zwischen älteren Quellen und neueren Veröffentlichungen ist ebenfalls sehr interessant, größere Qualitätsunterschiede konnten jedoch nicht festgestellt werden.

2.3 Status Quo vorhandener Programmierkonzepte

2.3.1 Überblick

Dieser Abschnitt untersucht bestehende Programmiersprachen und Konzepte auf ihre Eignung hinsichtlich unserer Problemstellung. Außerdem wird versucht, positive Ansätze aufzuzeigen, die in die Problemlösung einfließen können und negative Ansätze zu ermitteln, die in der Problemlösung vermieden werden sollten.

Jede Sprache wird daher zuerst in einer allgemeinen *Einführung* bewertet und vorgestellt. Anschließend erfolgt eine *Untersuchung* hinsichtlich der Eignung bezüglich der bereits ausgearbeiteten Kriterien von Kapitel 1.4. Diese Untersuchung arbeitet für die Lösung wichtige Entsprechungen bzw. Unterschiede heraus.

Abschließend faßt das *Fazit* die gefundenen Ergebnisse kurz zusammen und stellt nochmals die wichtigen Erkenntnisse vor.

Die Auswahl der Programmiersprachen orientierte sich an den wichtigsten objektorientierten Sprachen. Zusätzlich schafft die Untersuchung des Windows-Betriebssystems sowie allgemeiner Generatorkonzepte einen Überblick über vorhandene allgemeine Lösungsansätze.

2.3.2 Java

Java als stetig beliebter werdende, moderne, objektorientierte Sprache ist stark objektorientiert konzipiert und umgesetzt worden. Daher findet die Bewertung vorhandener Programmiersprachen mit Java ihren Anfang.

Einführung

Java ist eine relativ junge Sprache, die daher neuere Erkenntnisse der Programmierung, ohne den Zwang zu Abwärtskompatibilitäten, berücksichtigen konnte. Im Ergebnis des Versuchs, eine moderne Sprache zu kreieren, ist Java vollständig objektorientiert angelegt.

Die graphischen Elemente, die zu einer Nutzeroberfläche verbunden werden können, sind aus diesem Grund Klassen. Diese Klassen werden in größere Klassen eingebettet, dies wird soweit fortgeführt, bis die Benutzeroberfläche das gewünschte Aussehen erhält.

Die endgültige Oberfläche besteht aus einer oder mehreren Klassen, die im Normalfall einzeln angesprochen, plaziert und ausgewertet werden. Je nach Vererbungshierarchie weisen die Klassen ähnliche Schnittstellen auf, es gibt jedoch keine fest von Java vordefinierte Schnittstelle aller GUI-Klassen.

Die GUI-Klassen von Java werden im Java-Umfeld allgemein üblich als *swing*-Klassen bezeichnet, wir werden im folgenden beide Bezeichnungen synonym verwenden.

Untersuchung

Java ist eine formale Sprache, die es ermöglicht, graphische Nutzeroberflächen zu definieren und auch zu prüfen. Java selbst kann auf Korrektheit geprüft werden, es stehen genügend Syntaxchecker zur Verfügung.

Java ist eine compilierte Sprache, d. h. die Quelltextdateien müssen vor der Ausführung manuell oder automatisch compiliert werden. Die Compilierung erzeugt jedoch im Unterschied zu anderen compilierten Sprachen keinen ausführbaren Code sondern Bytecodedateien, die bei der Ausführung interpretiert werden.

Dies bedeutet dennoch, daß Änderungen am Programmcode die Neucompilierung einer oder mehrerer Quellcodedateien nach sich zieht.

Das heißt, daß eine dynamische Generierung der Oberfläche während des Programmlaufs von Java nicht explizit als Programmkonstrukt vorgesehen ist. Eine Änderung der Benutzeroberfläche zieht eine Neucompilierung mindestens der geänderten Programmteile nach sich, in deren Ergebnis der entstehende Bytecode an den Nutzer übertragen werden muß.

Die *swing*-Klassen haben, wie bereits erwähnt, keine für graphische Benutzeroberflächen einheitliche Schnittstelle, die es erlauben würde, die Elemen-

te wahllos untereinander auszutauschen. Dies gilt eingeschränkt auch für die Manipulationsmöglichkeiten, die in der Problemstellung gefordert werden.

Java verwendet in den *swing*-Klassen das *Model-View-Controller*-Konzept, das eine Trennung von Datenmodell, der Sicht auf die Daten und dem kontrollierenden Programmcode vorsieht.

Java bietet die Möglichkeit, Schnittstellen zu definieren, die Klassen implementieren müssen. Die Möglichkeit einer Vereinheitlichung ist also gegeben.

Jede Java-Klasse muß in einer eigenen Quellcodedatei gespeichert werden, die einzelnen Dateien können in sogenannten *packages* gruppiert werden.

Fazit

Es wird deutlich, daß Java als Zielsprache eine gute Wahl wäre, da Formalität, Schnittstellendefinition und strikte Objektorientierung unseren Kriterien zugute kommen.

Die Elemente der *swing*-Klassen können genutzt werden, müssen jedoch noch mit einheitlichen Schnittstellen versehen werden.

Das *Model-View-Controller*-Konzept ist in den *swing*-Klassen bereits enthalten und kann mit deren Nutzung verwendet werden. Es bietet für das vorgestellte Konzept jedoch keine nutzbaren Vorteile und wird daher nicht weiter behandelt.

Die Generierung und Änderung der GUIs zieht in jedem Fall eine Neucompilierung mindestens der geänderten Programmteile nach sich, was als Lösung für die Problemstellung inakzeptabel ist, da nicht der gesamte geänderte Programmcode sondern nur die variablen Programmteile ausgetauscht werden sollen.

Die Separierung aller Klassen in eigenen Dateien ist zwar für Java eine sehr praktikable Lösung, wird unserem Problem aber nicht gerecht, so daß diese Trennung nicht übernommen wird.

2.3.3 C und C++

Die Sprachen C und ihre spätere Erweiterung C++ sind zu zwei der meistverwendeten Sprachen bei professionellen Programmierern geworden.

Einführung

C und C++ werden vor allem wegen der Schnelligkeit der Programmausführung von Programmierern gerne eingesetzt. Demgegenüber steht relativ komplizierte und maschinennahe Programmierung. Auf schnelligkeitsmindernde Elemente wie graphische Benutzeroberflächen wurde bei der Erstellung von C bzw. C++ keine Rücksicht genommen.

Dies bedeutet, daß die Repräsentation von GUIs von den jeweiligen Anbietern der Compiler oder Entwicklungsumgebungen bereitgestellt werden muß. Es ist daher nicht möglich, einen bestimmten graphischen Ansatz hervorzuheben, da diese je nach Anbieter stark differieren.

C++ als objektorientierte Erweiterung von C besitzt alle Möglichkeiten der Objektorientierung.

Untersuchung

Die Generierung von GUIs kann aus den oben dargelegten Gründen nicht erfolgen, die Objektorientierungskonzepte von C++ können dennoch bewertet werden.

Objektorientierung wird von C++ vollständig unterstützt, dies schließt sogar die umstrittene Mehrfachvererbung mit ein. Das Konzept der Schnittstellen ist in C++ bekannt, unterscheidet sich prinzipiell jedoch nicht vom entsprechenden Ansatz unter Java und ist in der Ausführung deutlich komplexer gehalten.

Fazit

Fehlende graphische Möglichkeiten und komplexere Syntax der objektorientierten Programmkonstrukte sprechen gegen eine Übernahme der Konstrukte aus C++, da in Java gleiche Möglichkeiten bei einfacherer Syntax und Semantik zur Verfügung stehen.

2.3.4 Visual Basic

Visual Basic ist eine vor allem bei Programmiereinsteigern gern eingesetzte Programmiersprache, da ihnen einfache Programmierung oftmals näher liegt als Programmgeschwindigkeit oder Programmierkonzepte. Außerdem wird Visual Basic von Programmierern oftmals für schnelles Prototyping verwendet.

Einführung

Die Erstellung von GUIs ist bei Visual Basic visuell möglich, die entstehenden Strukturen werden in ein Basic-Programm umgewandelt. Dieses Programm enthält dann die Beschreibung aller visuellen Elemente.

Visual Basic kann objektbasiert programmiert werden, die Objektbasierung ist bei der Sprachsyntax jedoch nicht so konsequent berücksichtigt worden wie z. B. die Objektorientierung bei Java. Außerdem fehlen einige Konzepte objektorientierter Sprachen bzw. sind abweichend von gängigen Konventionen implementiert worden. Ein Beispiel hierfür ist die fehlende Vererbung.

Untersuchung

Die GUI-Elemente von Visual Basic werden als Programmcode in einer Basic-Datei abgelegt. Dabei werden die graphischen Elemente in einem eigenen Abschnitt vom Programmcode getrennt und durch diesen mittels eindeutiger Namen angesprochen.

Die Definition der Elemente ist ausführlich und nicht sehr einfallsreich, alle gesetzten Properties werden den Elementen zugeordnet, diese entsprechend der visuellen Anordnung gruppiert und dann ausgegeben. Logische Gruppierung und Wiederverwendung bereits vorhandener Definitionen wird vom Sprachumfang her nicht explizit unterstützt.

Fazit

Die Trennung der graphischen Oberflächenbeschreibung vom restlichen Programmcode ist eine nützliche Eigenschaft, die im vorgestellten Konzept Berücksichtigung finden sollte. Die ausführliche Beschreibung der GUI-Elemente ist jedoch nicht zur Nachahmung geeignet, da hierarchische Gruppierungen oder Formatzuweisungen fehlen.

2.3.5 Delphi

Delphi, eine objektorientierte Erweiterung von Pascal, ist eine vor allem bei Datenbankanwendungen gerne eingesetzte Sprache. Visuelle GUI-Builder erlauben dabei schnelle und einfache Erstellung graphischer Benutzeroberflächen.

Einführung

Delphi ist eine vollständig objektorientierte Programmiersprache. Damit wird auch das GUI als Klasse aufgefaßt, die aus einzelnen Klassen zusammengesetzt wird. Diese Klassen können beliebig tief verschachtelt sein.

Die textuelle Repräsentation eines GUIs ist Quellcode, der die Klassendefinitionen enthält.

Untersuchung

Die Objektorientierung von Delphi ist, wie bei Java, eine positiv zu bewertende Eigenschaft.

Außerdem ist die Möglichkeit, mehrere GUI-Elemente in einer Quellcodedatei zu definieren, sehr gut für übersichtliche Oberflächenprogrammierung geeignet. Die Schachtelung der Oberflächenelemente sowie die Einfachheit der Delphi zugrundeliegenden Programmiersprache Pascal sind ebenfalls positive Attribute.

Fazit

Die Zusammenfassung des gesamten GUI-Quellcodes in einer Datei sollte von Delphi übernommen werden. Die Sprachkonstrukte von Pascal sind ebenfalls bei Bedarf zu übernehmen. Dabei sollte jedoch die Orientierung an Java nicht verloren gehen. Statt der lexikalischen Einheiten von Pascal sollten daher die lexikalischen Einheiten von Java benutzt werden.

2.3.6 GUI-Generatoren

GUI-Generatoren werden verwendet, um aus GUI-Beschreibungen, die in unterschiedlicher Form, visuell oder textuell, vorliegen, Quellcode zu erzeugen. Die Generatoren dienen als Ersatz für die aufwendige manuelle Programmierung von GUIs.

Einführung

GUI-Generatoren liegen konzeptuell sehr nahe an dem GUI-Generierungteil des vorliegenden Problems. Wie auch hier gefordert, können sie aus metasprachlichen Definitionen Quellcode erzeugen.

GUI-Generatoren kommen hauptsächlich in Computer Aided Softwareengineering (CASE)-Werkzeugen vor, bei denen komplette Programmgenerierung aus konzeptuellen Überlegungen angestrebt wird. Außerdem finden sich GUI-Generatoren in den meisten visuellen GUI-Buildern, bei denen aus visuell erstellten graphischen Benutzeroberflächen der entsprechende Programmcode erzeugt wird.

Untersuchung

Prinzipiell sind GUI-Generatoren sehr gut zur Lösung des vorliegenden Problems geeignet, wenn lediglich die Benutzeroberfläche betrachtet wird. Das zugrundeliegende Konzept entspricht dem in der Diplomarbeit angestrebten, aus metasprachlichen Konzepten schnell und möglichst einfach bzw. automatisiert Programmcode zu entwickeln.

Das vorliegende Konzept legt jedoch Wert auf die Eigenständigkeit des generierten Quellcodes, d. h. der generierte Quellcode ist nicht in der Zielsprache geschrieben, sondern in der zu entwickelnden Programmiersprache.

Fast jeder GUI-Generator besitzt ebenfalls eine eigene Zwischensprache, in der die interne Repräsentation der konzipierten GUI stattfindet. Diese Zwischensprache wird dann dazu genutzt, den eigentlichen Quellcode zu erzeugen. Leider sind die Anbieter von GUI-Generatoren nicht daran interessiert, die Zwischensprachen zu veröffentlichen, so daß diese selbst bei einem offen dargelegten Konzept (z. B. bei Hofmann [8]) nicht dokumentiert werden.

Fazit

Das GUI-Generatoren zugrundeliegende Konzept sollte in unseren Ausführungen berücksichtigt werden, da die Generierung von Quellcode aus metasprachlichen Konzepten oder visuell erstellten GUI-Beschreibungen das Teilproblem der Oberflächengenerierung löst.

Das zu entwickelnde Konzept beschäftigt sich dabei mit der Entwicklung der Zwischensprache, deren Nutzen in der einfachen Umsetzung des geforderten Aussehens liegt.

Da die Dokumentation der Zwischensprachen leider nicht veröffentlicht wird, können keine Designvergleiche vorgenommen werden, die es ermöglichen würden, vor- und nachteilige Programmkonstrukte zu verwenden bzw. zu vermeiden.

2.3.7 Betriebssystem: Windows

Windows steht als Beispiel für Betriebssysteme, die graphische Nutzeroberflächen zwischen Programmen und Nutzern ermöglichen. Da Betriebssysteme im allgemeinen keine speziellen Programmieransätze oder -konzepte bieten, wird Windows lediglich wegen der Verwendung von Ressourcendateien vorgestellt.

Einführung

Windows hat sich zu einem der am meisten eingesetzten Betriebssysteme entwickelt. Die starke Verbreitung führte vor allem über die graphische Nutzeroberfläche, die auch ungelernten Computerbenutzern einen Zugang zum Rechner ermöglichen soll. Jenseits aller Vorbehalte und Diskussionen um Microsoft und Windows interessiert für das vorgestellte Konzept die Möglichkeit, externe Ressourcendateien für Programme anzulegen.

Untersuchung

Die Untersuchung der Ressourcendateien gestaltet sich schwierig, da außer dem allgemeinen Ansatz der Separierung von GUI-Beschreibung und Programm keine weiteren Informationen über die externen Dateien zur Verfügung stehen. Programmierer, die nicht die C oder C++ Compiler von Borland benutzen, werden Ressourcendateien selten direkt bearbeiten, meist werden sie die Möglichkeit der Verwendung von Ressourcendateien nicht kennenlernen.

Der Ansatz sieht vor, daß die Ressourcen der GUI eines Programms extern bearbeitet und dem Programm beigegeben werden. Dies kann in einer extra Datei oder in einer Vermischung von compiliertem Programm und Ressourcendatei erfolgen. Die Ressourcen werden beim Programmstart eingelesen und in

Oberflächenelemente umgesetzt.

Dieses Konzept muß nicht von jedem Windowsprogramm benutzt werden, eigene Ansätze sind durchaus möglich.

Fazit

Das von Windows verfolgte Konzept vereinfacht die Anpassung von GUIs enorm, da im Idealfall nur die Ressourcendateien geändert werden müssen. Dies wird vor allem bei der Internationalisierung von Programmen genutzt.

Da der Ansatz nicht zwingend vorgeschrieben ist, sind die meisten Compileranbieter dazu übergegangen, graphische Oberflächen auf eigene Weise zu realisieren, ohne Ressourcendateien zu benötigen. Daher ist der Einsatz von Ressourcendateien zurückgegangen und der Nutzen des Ansatzes hat sich verringert.

Für das hier vorgelegte Konzept ist die Trennung von GUI und Programmcode ein vielversprechender Ansatz.

2.4 Grundlagen von Programmiersprachen

2.4.1 Überblick

Dieser Abschnitt beschäftigt sich mit den Grundlagen von Programmiersprachen. Dazu gehört zunächst eine Einführung in die theoretischen Grundlagen, bevor anschließend die praktischen Probleme und Hinweise zusammengetragen werden, die für die Erstellung einer Programmiersprache wichtig sind.

Zu den angesprochenen Themen existiert sehr viel Literatur, der Abschnitt faßt die Ergebnisse mehrerer Veröffentlichungen zusammen. Dabei handelt es sich um Arbeiten von Alber [1], Sethi [13], Doberkat [4] und Hopcroft [9].

Angaben zu speziellen Literaturstellen finden sich im Abschnitt an der relevanten Stelle.

2.4.2 Allgemeine Anforderungen an Programmiersprachen

Die Anforderungen an Programmiersprachen sind sehr vielfältig. Es hat sich jedoch bewährt, bei jeder Programmiersprache zwei wichtige Punkte zu berücksichtigen:

1. Standardisierung der Sprache
   Damit Programmierer mit einer Sprache arbeiten können, muß diese genau definiert sein. Dies betrifft sowohl Syntax als auch Semantik der Sprache.

2. Programmverifikation
   Es muß möglich sein, mittels einer Sprache geschriebene Programme veri-

fizieren zu können, d. h. den mathematischen Nachweis erbringen zu können, daß ein Programm eine vorgelegte Spezifikation erfüllt. Auch hier ist eine mathematisch präzise Definition der Sprache erforderlich.

In beiden Fällen ist die mathematische Definition der Programmiersprache gefordert worden, die zwei folgenden Abschnitte 2.4.3 und 2.4.4 werden daher die theoretischen Grundlagen der Definition von Programmiersprachen erläutern.

2.4.3 Definitionen

Eine Programmiersprache kann und muß, wie im vorigen Abschnitt festgestellt wurde, mathematisch genau definiert werden. Wir definieren daher eine Programmiersprache P als $P = (A, L, B, S)$, wobei A ein endliches Alphabet symbolisiert, L eine formale Sprache $L \subseteq A^*$, B eine Menge und $S : L \to B$ eine Abbildung.

Somit enthält A alle Zeichen, die in Programmen aus P auftreten können. Im allgemeinen definiert man das Alphabet durch Angabe aller seiner Elemente. Wichtig ist dabei, daß ein Alphabet meist nicht nur Buchstaben und Ziffern enthält sondern auch Sonderzeichen und reservierte Wörter (wie z. B. *if*).

Die Sprache L definiert alle syntaktisch korrekten Programme, wir bezeichnen L daher auch als Syntax von P. Unter den verschiedenen Möglichkeiten, die Syntax zu definieren, ist die am weitesten verbreitete Methode diejenige, eine kontextfreie Grammatik G anzugeben, die die entsprechende Sprache erzeugt.

B und S bilden zusammen die Semantik von P. Dabei ist B die Menge aller möglichen Bedeutungen von Programmen, S bildet dann jedes Programm $p \in L$ auf ein Element $S(p) \in B$ ab und ordnet dem Programm damit seine Bedeutung zu.

2.4.4 Definition formaler Sprachen

Die Compilierung einer Sprache erfolgt in vier Schritten:

1. lexikalische Analyse,
2. syntaktische Analyse,
3. semantische Analyse und
4. Codeerzeugung.

Dies bedeutet, daß für eine Sprache die lexikalischen Einheiten, die Syntax und die Semantik festgelegt werden müssen, bevor die Codeerzeugung starten kann. Für diese drei Bereiche existieren mehrere Möglichkeiten der Definition, die in den folgenden Abschnitten näher untersucht werden.

Definition der lexikalischen Einheiten

Jedes Programm, das von einem Compiler untersucht wird, besteht zunächst aus einer Folge von Zeichen. Die Syntaxdefinition eines Programms bezieht sich jedoch auf lexikalische Einheiten, die Tokens genannt werden.

Die Aufgabe der lexikalischen Analyse besteht daher darin, die lexikalischen Einheiten aus dem zu untersuchenden Programmtext zu extrahieren. Damit müssen die lexikalischen Einheiten dem Compiler bekannt gemacht werden, sie müssen definiert werden. Diese Definition wird auch als lexikalische Struktur der Sprache bezeichnet und erfolgt meist mittels regulärer Ausdrücke oder über endliche Automaten.

Dabei beschreiben reguläre Ausdrücke lexikalische Klassen in einer Folge von Zeichen sehr gut, sind jedoch für die Implementierung nicht gut geeignet, hier haben sich endliche Automaten durchgesetzt. Die Definitionen für beide Beschreibungsarten sind:

DEFINITION 1 (Reguläre Ausdrücke)

Es sei X ein endliches Alphabet und X^* die von X erzeugte freie Halbgruppe. Dann werden reguläre Ausdrücke über X und die von ihnen bezeichneten Mengen wie folgt definiert:

1. $\emptyset$ ist ein regulärer Ausdruck und bezeichnet die Menge $\{\}$,
2. ε ist ein regulärer Ausdruck und bezeichnet die Menge $\{\varepsilon\}$,
3. $\forall a \in X$ ist a ein regulärer Ausdruck, der die Menge $\{a\}$ bezeichnet
4. sind a und b reguläre Ausdrücke, die die Mengen A und B bezeichnen, so sind $a + b$, $a \cdot b$ und a^* reguläre Ausdrücke, welche die Mengen $A \cup B$, AB und A^* bezeichnen,
   wobei $AB = \{w \mid w = a \cdot b \wedge a \in A \wedge b \in B\}$ und
   $A^* = \bigcup_{i=0}^{\infty} A^i$, mit $A^0 = \{\varepsilon\}$; $A^i = AA^{i-1}$ sind,

DEFINITION 2 (Endliche Automaten)

Sei X ein endliches Alphabet und Z eine endliche Menge von Zuständen. Dann heißt das Quintupel (X, Z, z_0, δ, F) ein endlicher Automat, falls $z_0 \in Z$ der Anfangszustand ist, $F \subset Z$ eine Menge von Endzuständen ist und $\delta : X \times Z \to Z$ eine Abbildung, die Übergangsfunktion genannt wird, ist.

Dabei arbeitet der Automat so, daß er im Anfangszustand z_0 beginnt und je nach gelesenem Buchstaben und Zustand mittels der Überführungsfunktion δ in einen neuen Zustand überführt wird. Eine Zeichenfolge $v \in X^*$ wird dann vom Automaten *akzeptiert*, falls $\delta(v, z_0) \in F$ gilt, der Automat also vom Anfangszustand mittels der Zeichenfolge in einen Endzustand überführt werden kann.

Hiermit sind die theoretischen Voraussetzungen geschaffen, um zu zeigen, daß die regulären Ausdrücke über dem Eingabealphabet X genau die Mengen bezeichnen, die von endlichen Automaten akzeptiert werden. Dies wird hier nicht im Detail gezeigt, es sei auf Zima ([16], S. 47 ff.) verwiesen, der die Verbindung reguläre Ausdrücke $\rightarrow$ reguläre Mengen $\rightarrow$ endliche Automaten sehr übersichtlich und ausführlich angibt.

Es besteht nun die Möglichkeit, die lexikalischen Einheiten, die Token, durch reguläre Ausdrücke zu beschreiben. Da ein Algorithmus angegeben werden kann, der eine Verbindung zwischen regulären Ausdrücken und endlichen Automaten herstellt, kann ein die Tokenklasse erkennender Automat erzeugt werden. Es muß daher lediglich darauf geachtet werden, daß nach der Erkennung eines Tokens der entsprechende Automat angesprochen wird.

Diese Vorgehensweise wird in der Praxis z. B. durch das Programm *lex* realisiert. Wir müssen daher die Token so definieren, daß sie z. B. mittels *lex* aus der Eingabezeichenfolge extrahiert werden können.

Syntax

Der lexikalischen Analyse folgt die syntaktische Analyse, deren Aufgabe darin besteht, die syntaktische Struktur des beschriebenen Programms aus den Token zu erkennen.

Für die Erkennung ist es wichtig, die Syntax der Sprache so zu beschreiben, daß diese Beschreibung in der syntaktischen Analyse verwendet werden kann. Die Schwierigkeit besteht darin, so viele Sprachkonstrukte wie möglich zuzulassen, um dem Programmierer so viele Wege wie möglich zu geben, einen Algorithmus zu implementieren. Andererseits muß die Syntax maschinell überprüfbar sein, was die Vielfalt der Sprachelementkombinationen einschränkt.

Verallgemeinert können wir festhalten, daß zwischen der Menge A^* aller Worte über A, der Menge L der syntaktisch korrekten Programme und der Menge L' der semantisch korrekten Programme folgende Beziehung besteht:

$$L' \subseteq L \subseteq A^*$$

Für weitere Überlegungen betrachten wir zwei Extremfälle, die die Grenzen der obigen Aussage aufdecken:

1. $L = A^*$, d. h. alle Zeichenfolgen sind syntaktisch korrekte Programme.
   Wir erhalten eine Sprache ohne Syntax, damit ist größtmögliche Flexibilität für den Programmierer gegeben, da alle Wortkombinationen der Sprache als Programm syntaktisch erlaubt sind. Die Syntaxüberprüfung wäre damit überflüssig, die Bedeutung des Programms kann allein die Semantik festlegen.

2. $L = L'$, d. h. nur diejenigen Zeichenfolgen sind semantisch korrekt, die auch syntaktisch korrekt sind.

Wir erhalten eine Sprache mit sehr eingeschränkter Syntax, bei der semantisch nicht korrekte Programme bereits bei der Syntaxüberprüfung erkannt werden. Dies engt den Programmierer sehr stark ein, die Bedeutung eines Programms ist dafür aber bereits in der frühen Phase der Syntaxüberprüfung festgelegt.

Beide Möglichkeiten haben ihren Reiz, sind jedoch in der Praxis nicht vertretbar, die erste erfordert einen sehr hohen Aufwand der Semantikdefinition, die zweite schränkt die Möglichkeiten des Programmierers so stark ein, daß dieser Sprache sicherlich keine weite Verbreitung beschieden ist. In der Praxis wird daher ein Mittelweg zwischen den beiden Extremen gesucht.

Wir beschränken uns auf die Forderung, daß L eine entscheidbare Teilmenge von A^* ist, so daß wir einen Algorithmus angeben können, der die syntaktische Korrektheit eines Programms prüft. Außerdem wird versucht, die Syntax so festzulegen, daß so viele semantisch nicht korrekte Programme wie möglich und akzeptabel bereits bei der Syntaxprüfung ausgeschlossen werden können.

In der Sprachentheorie hat sich für die syntaktische Beschreibung von Sprachen das Konzept der Grammatiken durchgesetzt. Dabei wird die syntaktische Struktur des Programms durch eine Grammatik beschrieben, die aus einer endlichen Anzahl von Ableitungsregeln besteht und überprüfen kann, ob ein Programm zu einer Sprache gehört. Hierbei wird getestet, ob aus einem Startsymbol unter Benutzung der Ableitungsregeln das Programm abgeleitet werden kann.

Formal ausgedrückt bedeutet dies:

DEFINITION 3 (Grammatik)

Ein Quadrupel $G = (T, N, P, S)$ heißt eine Grammatik, wenn gilt:

1. T ist die Menge der terminalen Symbole, ein endliches Alphabet.
2. N ist die Menge der nichtterminalen Symbole, ebenfalls ein endliches Alphabet.
3. T und N sind disjunkt, d. h. $N \cap T = \emptyset$.
4. $P \subset (N \cup T)^* \times (N \cup T)^*$ ist eine Menge von Regeln, auch Produktionsregeln genannt, die die Form $\alpha \to \beta$ besitzen mit der Einschränkung, daß $\alpha \neq \varepsilon$ ist.
5. $S \in N$ ist das Startsymbol.

Um zur Sprache zu kommen, die diese Grammatik produziert, definieren wir noch die Relation „$\Rightarrow$“ eher informal mit:

$$\gamma\alpha\delta \Rightarrow \gamma\beta\delta \quad , \quad \text{wenn } \alpha \to \beta \text{ eine Produktion der Grammatik ist} \tag{2.1}$$

Außerdem bestimmen wir die reflexive und transitive Hülle $\Rightarrow^*$ der Relation durch:

$$\alpha_1 \Rightarrow^* \alpha_m \quad , \quad \text{wenn } \alpha_1 \Rightarrow \alpha_2, \alpha_2 \Rightarrow \alpha_3, \ldots, \alpha_{m-1} \Rightarrow \alpha_m \tag{2.2}$$

Jetzt besitzen wir das theoretische Rüstzeug, die durch die Grammatik G erzeugte Sprache $L(G)$ zu beschreiben:

$$L(G) \quad = \quad \{v \in T^* : S \Rightarrow^* v\} \tag{2.3}$$

Die Sprache ist demnach die Menge aller Wörter, die mittels der Grammatik aus dem Startsymbol S abgeleitet werden können.

Die oben angegebene Grammatik wird auch als nicht eingeschränkte Grammatik bezeichnet. Das bedeutet, daß die Form der Produktionsregeln keinerlei zusätzlichen Einschränkungen unterliegt. Diese Art von Grammatiken kann dazu benutzt werden, sehr komplexe Sprache zu schaffen. Der Nachteil liegt darin, daß die Bedingung $S \Rightarrow^* v$ nicht automatisiert geprüft werden kann.

Daher werden die Produktionsregeln Einschränkungen unterworfen und damit neue Grammatikklassen geschaffen. In der Sprachtheorie ist so eine Hierarchie von Grammatiken geschaffen worden, die sogenannte Chomski-Hierarchie. Die Grammatiken werden in vier Typen eingeteilt:

1. *Typ 0*- oder nicht eingeschränkte Grammatiken
2. *Typ 1*- oder kontextsensitive Grammatiken
3. *Typ 2*- oder kontextfreie Grammatiken
4. *Typ 3*- oder reguläre Grammatiken

Die von den Grammatiken erzeugten Sprachen werden dabei mit steigender Typzahl immer weiter eingeschränkt und so besser durch Automaten überprüfbar. Nachteilig wirkt sich aus, daß mit steigender Typzahl die Flexibilität der Programmiersprache abnimmt.

Für die Syntaxbeschreibung von Programmiersprachen haben sich kontextfreie Grammatiken als vorteilhaft erwiesen. Für Produktionsregeln kontextfreier Grammatiken gilt, daß jede Produktionsregel die Form:

$$A \rightarrow \alpha \qquad \text{mit } A \in N \text{ und } \alpha \in (N \cup T)^* \tag{2.4}$$

besitzt.

Die Vorteilhaftigkeit für Programmiersprachen ergibt sich hauptsächlich dadurch, daß es möglich ist, sie auf effiziente Weise durch (nichtdeterministische) Kellerautomaten erkennen zu lassen. Somit ist die maschinelle Entscheidung, ob ein Wort zu einer kontextfreien Sprache gehört, mittels des Kellerautomaten über die sprachdefinierende Grammatik möglich.

Bevor nun die Syntaxbeschreibung mittels kontextfreier Grammatiken erfolgen kann, werden noch drei Einschränkungen getroffen, deren Beachtung zum einen die Lesbarkeit der Syntaxbeschreibung erhöht, zum anderen die maschinelle Verarbeitung erleichtert.

1. Eine reine Schreiberleichterung ist folgende Konvention: wenn es Produktionen der Form $A \to \alpha$ und $A \to \beta$ gibt, wobei α und β Nichtterminale sowie Terminale sein können, so können wir diese Produktionen einfacher schreiben, indem wir eine Produktion schreiben, deren Alternativen durch das Zeichen „|“ getrennt werden. Wir schreiben: $A \to \alpha \mid \beta$.

2. Bei der zweiten Einschränkung sind sogenannte Ableitungsbäume von Interesse, die aus kontextfreien Grammatiken hergeleitet werden können. Ableitungsbäume sind graphische Repräsentationen einer Ableitung eines Wortes aus einer Grammatik. Sie dienen dazu, den Vorgang der Ableitung zu visualisieren und damit leichter verständlich zu machen.

   Es zeigt sich, daß zu jeder Grammatik ein Ableitungsbaum erstellt werden kann, der die Ableitung eines Wortes aus dieser Grammatik darstellt.

   Die formale Definition von Ableitungsbäumen wird z. B. bei Hopcroft ([9], S. 88 ff.) sehr gut beschrieben, daher wird an dieser Stelle nicht näher darauf eingegangen. Die Verwendung von Ableitungsbäumen wird anhand eines kleinen Beispiels demonstriert.

Beispiel:
Betrachten wir die Grammatik $G = (\{S, A\}, \{a, b\}, P, S)$, wobei P aus den Produktionen

$$S \to aAS \mid a \tag{2.5}$$
$$A \to SbA \mid SS \mid ba \tag{2.6}$$

besteht.
Wir leiten nun folgendes Wort ab:

$$S \Rightarrow aAS \Rightarrow aSbAS \Rightarrow aabAS \Rightarrow aabbaS \Rightarrow aabbaa \tag{2.7}$$

Der dazu äquivalente Ableitungsbaum ist in Abbildung 2.1 zu sehen.

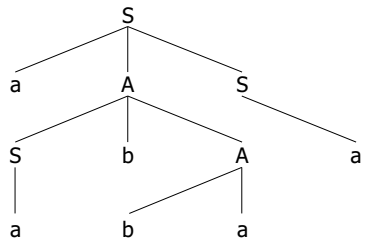

Abbildung 2.1: Ableitungsbaum von Ableitung 2.7

Quelle: Hopcroft [9], S. 88

Jetzt ist gut zu sehen, daß das abgeleitete Wort auf mehreren Wegen aus der Grammatik erzeugt werden kann, dies entspräche den unterschiedlichen

Möglichkeiten, den Ableitungsbaum von oben nach unten zu verfolgen. Wir legen fest, daß wir eine Ableitung, bei der in jedem Ableitungsschritt eine Produktion auf das am weitesten links stehende Nichtterminalsymbol angewendet wird, *Linksableitung* nennen. Entsprechend verfahren wir mit dem am weitesten rechts stehenden Nichtterminal und nennen die Ableitung *Rechtsableitung*.

Nun kann gezeigt werden, daß ein Wort zwar mehrere Ableitungsbäume besitzen kann, zu jedem Ableitungsbaum aber nur je eine Links- und Rechtsableitung existiert.

Wir haben damit die Möglichkeit, zur Überprüfung eines Worts entweder eine Links- oder Rechtsableitung zu konstruieren. Je nach gewählter Ableitungsart nennen wir diesen Vorgang für Linksableitungen LL-Analyse und für Rechtsableitungen LR-Analyse.[1] Im Ableitungsbaum bedeutet dies anschaulich, daß bei der LL-Analyse versucht wird, den Ableitungsbaum von der Wurzel her zu konstruieren, bei der LR-Analyse geht diese Konstruktion von den Blättern aus.

3. Wenden wir uns nun der dritten Einschränkung zu, der Normalisierung der Grammatiken. Wie bereits definiert, können Produktionen von kontextfreien Grammatiken die Form $A \rightarrow \alpha$ mit $A \in N$ und $\alpha \in (N \cup T)^*$ besitzen.

   Wir versuchen nun, die Grammatiken so zu vereinfachen, daß sowohl die Lesbarkeit als auch die maschinelle Verarbeitbarkeit der Sprache erhöht wird, ohne die Eigenschaft der Kontextfreiheit zu verlieren oder zu verschärfen.

   Dabei wird an dieser Stelle wieder lediglich eine informelle Beschreibung erscheinen, ausführliche Darstellungen mit Beweisen finden sich bei Hopcroft ([9], S. 94 ff.).

   Die gegebene kontextfreie Grammatik wird zunächst von nutzlosen Symbolen befreit, d. h. jedes Nichtterminal und jedes Terminal von G erscheint mindestens einmal in der Ableitung eines Wortes in L. Diese Einschränkung vereinfacht die Grammatik, da überflüssige Symbole entfallen, die spezifizierte Sprache bleibt jedoch offensichtlich erhalten, da die entfernten Symbole nicht in der Ableitung der Worte auftauchen.

   Als zweites erheben wir die Forderung, daß es keine Produktionen der Form $A \rightarrow B$ gibt, wenn A und B Nichtterminale sind. Auch diese Beschränkung erscheint sinnvoll, da Ableitungen der gegebenen Form keinen Fortschritt in der Ableitung von Worten erwirken und leicht ersetzt werden können.

   Als letztes streichen wir noch Produktionen der Form $A \rightarrow \varepsilon$ unter der Voraussetzung, daß ε nicht in L enthalten ist. Wir können festhalten, daß alle Produktionen eine der Formen $A \rightarrow BC$ oder $A \rightarrow a$ haben müssen, A, B und C sind dabei beliebige Nichtterminale, a ein beliebiges Terminal.

[1] Das erste L steht dabei für die Leserichtung von links nach rechts, der zweite Buchstabe kennzeichnet die Ableitungsart.

Damit erhalten wir die Möglichkeit, für jede kontextfreie Grammatik eine äquivalente Grammatik anzugeben, die der Chomski-Normalform genügt. Diese sogenannte CNF ist wie folgt angegeben:

SATZ 1 (Chomski-Normalform oder CNF)

> Jede kontextfreie Sprache ohne ε wird von einer Grammatik erzeugt, in der alle Produktionen von der Form $A \to BC$ oder $A \to a$ sind; dabei sind A, B und C beliebige Nichtterminale, und a ist ein beliebiges Terminalsymbol.

Mit diesen drei Einschränkungen haben wir nun die Möglichkeit, kontextfreie Grammatiken zum einen für den Menschen lesbar und zum anderen maschinell leicht bearbeitbar anzugeben. Wir bedienen uns dazu der sogenannten Backus-Naur-Form (BNF), die die oben genannten Eigenschaften in sich vereint.

Die BNF entspricht bis auf geringfügige Änderungen im Format der Notation für kontextfreie Grammatiken, die den oben genannten drei Einschränkungen unterworfen wurde. So hat der Sprachentwickler die Möglichkeit, die zu definierende Sprache auf „natürliche“ Art und Weise zu definieren.

Das Format der BNF sieht vor, daß eine Ersetzungsregel, die einer Produktionsregel einer kontextfreien Grammatik entspricht, auf der linken Seite ein Nichtterminalzeichen enthält. Dieses wird durch eine auf der rechten Seite stehende Zeichenkette definiert, die beiden Seiten werden durch „`::=`“ getrennt.

Nichtterminalzeichen werden in spitze Klammern „⟨“ und „⟩“ eingeschlossen. Alternative gültige rechte Seiten werden durch einen senkrechten Strich „`|`“ voneinander getrennt. Die Zeichen „⟨“, „⟩“ und „`|`“ gehören dabei nicht zum Alphabet der zu definierenden Sprache.

Beispiel:
Das syntaktische Objekt Bezeichner *der Programmiersprache PASCAL wird wie folgt definiert:*

```
⟨ziffer⟩      ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨buchstabe⟩   ::=  a | b | c | d | ... | x | y | z
⟨wort⟩        ::=  ⟨buchstabe⟩
                   | ⟨ziffer⟩
                   | ⟨wort⟩⟨buchstabe⟩
                   | ⟨wort⟩⟨ziffer⟩
⟨bezeichner⟩  ::=  ⟨buchstabe⟩
                   | ⟨buchstabe⟩⟨wort⟩
```

Die Auslassungspunkte sind nicht Bestandteil der Definition, an ihrer Stelle müssten alle fehlenden Buchstaben stehen, sie wurden aus Platzgründen hier weggelassen.

Um praktischen Anforderungen der Sprachdefinition gerecht zu werden, wurde die Backus-Naur-Form um drei Sprachelemente erweitert, die größere Flexibilität erlauben.

1. Zeichenfolgen können mit Hilfe von runden Klammern „(...)“ gruppiert werden.

2. Zeichenfolgen, die optional sind, werden in eckige Klammern „[...]“ eingeschlossen.

3. Zeichenfolgen, die beliebig oft (auch nullmal) wiederholt werden können, werden in geschweifte Klammern „{...}“ eingeschlossen.

Die Zeichen „(“, „)“, „[“, „]“, „{“ und „}“ gehören wiederum nicht zum Umfang der definierten Sprache. Diese Art der Notation wird als EBNF bezeichnet.

Neben der textuellen Darstellung der EBNF gibt es noch eine graphische Darstellungsart, die sogenannten Syntaxdiagramme. Sie dienen lediglich dazu, die Sprachdefinition in anderer Form darzustellen, inhaltlich unterscheiden sich die Darstellungsarten nicht. Die Entscheidung für die jeweilige Darstellungsart richtet sich daher hauptsächlich nach der zu erwartenden Klientel, die die Syntax der Sprache nachvollziehen soll.

Jetzt besitzen wir das nötige Rüstzeug, um die Syntax der von uns definierten Sprache mittels einer kontextfreien Grammatik angeben zu können. Wir werden dazu die EBNF verwenden, die auch von Compilererzeugungsprogrammen zur syntaktischen Analyse, wie z. B. *yacc*[2], verwendet wird.

Semantik

Der nach der Analyse als syntaktisch korrektes Programm erkannte Quellcode wird nun an die semantische Analyse weitergeleitet, die die Semantik des Textes untersucht, d. h. die Bedeutung des Programms festlegt.

Der von der Syntaxanalyse übergebene Quelltext ist syntaktisch aufgrund der angegebenen kontextfreien Grammatik überprüft worden. Ein Programm ist im Regelfall aber nicht völlig kontextfrei. Zum Beispiel hängt die Korrektheit eines Programms davon ab, ob die auftretenden Bezeichner konsistent mit ihren Deklarationen und Definitionen benutzt werden. Die Deklarationen und Definitionen können explizit oder auch implizit gegeben sein, diese Regeln werden als Kontextbedingungen bezeichnet und unter dem Begriff der *statischen Semantik* zusammengefaßt. Die erste Aufgabe der semantischen Analyse ist daher die Überprüfung der statischen Semantik.

Die zweite Aufgabe der semantischen Analyse ist die Bestimmung der Information für Optimierung, Übersetzung und Codegenerierung, dieser Teil soll uns jedoch nicht interessieren und ist nur der Vollständigkeit halber aufgeführt worden.

Als Ergänzung der statischen Semantik sei hier noch die *dynamische Semantik* genannt, die semantische Bedingungen beschreibt, die im Compiler nicht

[2] *yacc (yet another compiler compiler)* arbeitet nach der LR-Ableitungsmethode.

geprüft werden können, sich der statischen semantischen Analyse also entziehen. Ein Beispiel hierfür ist der Ausdruck a/b, der syntaktisch und statisch semantisch korrekt ist. Der Ausdruck kann jedoch dynamisch semantisch falsch sein, wenn b den Wert 0 enthält. Dieser Test ist meistens erst zur Laufzeit des Programms möglich.

Interessanterweise ist die Grenze zwischen statischer und dynamischer Semantik nicht festgelegt, sondern stark von der verwendeten Programmiersprache abhängig. Streng typisierte Sprachen haben z. B. den Vorteil, daß viele statische Semantikfehler erkannt werden können, die bei untypisierten Sprachen zur dynamischen Semantik zählen. Der Nachteil, der der frühen Fehlererkennung gegenübersteht, ist der Verlust von Programmierflexibilität, deren Wichtigkeit von jedem Programmierer persönlich festgelegt werden muß.

Um die Semantik einer Programmiersprache angeben zu können, stehen mehrere Möglichkeiten zur Verfügung, von denen im folgenden fünf kurz erläutert werden.

1. Algebraische Methode
   Diese Methode legt die Bedeutung der einzelnen Sprachkonstrukte durch Gleichungen fest, wobei auf der linken Seite das Sprachkonstrukt steht und die rechte Seite die Bedeutung oder die Herleitung der Bedeutung angibt. Die algebraische Semantik kann aus einer kontextfreien Grammatik hergeleitet werden und ist ebenso abstrakt wie Semantiken nach der axiomatischen Methode. Sie kann jedoch auf eine größere Klasse angewandt werden, da sie applikative Konzepte umfaßt.

2. Axiomatische Methode
   Hier sei ein Programm p gegeben sowie P und Q, die Aussagen über der Zustandsmenge Z seien. Nun wird der Fall beschrieben, daß ein Zustand z vor der Ausführung von p die Bedingung P erfüllt und nach der Terminierung von p die Aussage Q für den Folgezustand z' gilt. Dies beschreiben wir kurz durch $\{P\}p\{Q\}$. Wir beschreiben die Bedeutung des Programms durch alle Aussagenpaare (P, Q) mit $\{P\}p\{Q\}$.
   Ziel ist es nun, ein Axiomen- und Regelsystem so anzugeben, daß mit dessen Hilfe alle (und nur die) gültigen Ausdrücke $\{P\}p\{Q\}$ herleitbar sind.

3. Funktionale Methode
   Die funktionale Semantik beschreibt die Abhängigkeit der Ausgabe- von den Eingabedaten mit Hilfe von Abbildungen. Hierbei ist der semantische Bereich eine Halbordnung, die Bedeutung der Sprachelemente wird durch stetige Funktionen beschrieben, die durch Funktionalgleichungen definiert werden. Diese semantischen Funktionen ordnen jeder syntaktischen Struktur die entsprechenden Elemente im semantischen Bereich zu und beschreiben so die Bedeutung eines Programms.

4. Operationale Methode
   Dieser Ansatz der Interpretersemantik ordnet die syntaktischen Sprachelemente mittels Zustandstransformation ihrer Bedeutung zu. Diese Zustand-

stransformationen werden mit Hilfe der Operationen einer abstrakten Maschine ausgedrückt.
Formaler ausgedrückt bedeutet dies, daß ein abstrakter Automat $M = (Z, M, f, Z_a)$ angegeben wird.[3] Als Interpretation des Programms bezeichnen wir dann $z_0, z_1, \ldots, z_i, \ldots$ mit $z_i = M(z_{i-1})$. Die Bedeutung des Programms ist also eine Folge von Zuständen der abstrakten Maschine oder sie ist für die Eingabe nicht definiert.

5. Informelle Definitionen
   Diese Definitionen, die nicht als eigene Methode bezeichnet werden, umfassen vor allem natürlichsprachliche bzw. „halbformale" Definitionen der Semantik. Diese häufig bei älteren Sprachen oder in nichtwissenschaftlichen Werken angewandte Methode beschreibt eine Semantik auf eher operationelle Weise. Das kann z. B. durch Angabe der Operationen und durch Erläuterung der Bedeutung der einzelnen Konstrukte meist unter Zuhilfenahme von Umgangssprache erfolgen.
   Andere Ansätze sehen auch in der halbformalen Definition die Möglichkeit, Zustandsüberführungsbedingungen zu beschreiben und damit die Semantik zu definieren.

Jede dieser Methoden kann in bestimmten Bereichen besonders gut eingesetzt werden, so sind z. B. die axiomatische und algebraische Methode in der Programmverifikation sehr gut verwendbar.

Die Methoden nehmen von oben nach unten im Abstraktionsgrad ab. Läßt man die informelle Definition außer acht, so kann festgestellt werden, daß die Ergebnisse der verschiedenen Methoden konsistent sind, d. h. gilt ein Ergebnis innerhalb einer abstrakteren Methode, so gilt dieses Ergebnis auch in einer konkreteren Methode.
Daher ist es möglich, die Semantik in so vielen Methoden wie nötig zu definieren, um den verschiedenen Anforderungen der Semantikdefinition gerecht zu werden.

Die formale Überprüfung der Semantik durch Automaten benötigt eine formale Repräsentation der Semantik. Für eine dem Menschen verständliche Beschreibung der Semantik reicht es aus, die Konstrukte informell zu beschreiben. Diese Form der Beschreibung läßt sich allerdings nicht maschinell verifizieren.

Viele veröffentlichte Semantikbeschreibungen bestehen daher aus wenigstens zwei verschiedenen Dokumenten, einer formalen Beschreibung, die der Semantikprüfung durch z. B. einen Compiler zugrunde liegt und auch mehrere Semantikdefinitionen nach unterschiedlichen Methoden umfassen kann sowie einer eher informellen Beschreibung, die dem Programmierer oder interessierten Leser die Semantik der Sprache näher bringt.

---

[3]

Z	Zustandsmenge des Automaten
$M : Z \to Z$	Zustandsüberführungsfunktion
f	Zustandsinitialisierungsfunktion
$Z_a \subseteq Z$	Ausgabezustandsmenge

Im Rahmen dieses Konzeptes ist es nicht nötig, die Semantik formal so zu beschreiben, daß eine automatische Semantiküberprüfung ermöglicht wird. Dieser Entscheidung liegt der Gedanke zugrunde, daß die Diplomarbeit ein Konzept beschreiben soll, das zwar inhaltlich korrekt ist, zur Zeit jedoch noch nicht in einem real existierenden Compiler umgesetzt werden muß. Das bedeutet natürlich nicht, daß eine nicht zu realisierende Semantikdefinition gegeben werden kann.

Daher wird die Semantikbeschreibung nach der informellen Methode in textueller Form erfolgen, ohne jedoch die formalen Aspekte einer Semantikdefinition aus den Augen zu verlieren, d. h. unrealistische oder in sich nicht konsistente Definitionen zu verwenden.

Für den interessierten Leser ist eine sehr detaillierte und informative Übersicht der oben angegebenen Methoden mit Vergleich der Methoden und Umwandlung zwischen den Methoden z. B. bei Fehr ([5], S. 9 ff.) zu finden. Das Buch umfaßt auch ein durchgängiges Beispiel, anhand dessen die Methoden dargestellt werden.

2.4.5 Designkriterien für Programmiersprachen

Nach diesen Vorüberlegungen sind wir jetzt in der Lage, eine Programmiersprache formal korrekt angeben zu können. Nun fehlen noch inhaltliche Kriterien, die beachtet werden sollten, um praktikable und nützliche, kurz „schöne" Programmiersprachen zu entwickeln.

Bevor diese Kriterien aufgestellt werden können, ist es notwendig, Programmiersprachen und ihre Bedeutung in die allgemeine Entwicklung der Informatik und der Computer einzuordnen, um den zu findenden Kriterien inhaltlichen Rückhalt zu geben. In diesem Abschnitt werden daher die allgemeine Rolle und Entwicklung von Programmiersprachen dargelegt, danach wird versucht, allgemeingültige Kriterien für gute Programmiersprachen zu finden.

Sofern Literatur verwendet wurde, ist diese bei Pratt [10] und Fischer [6] zu finden, die sehr ausführliche und verständliche Werke zum Thema schrieben.

Rolle von Programmiersprachen

Zu Beginn der Computerära war die hauptsächlich eingesetzte Programmiersprache handkodierter Assemblercode. Da in dieser Zeit Hardware der wesentlich höhere Kostenfaktor als Software war, wurden höhere Programmiersprachen hauptsächlich als Entwicklungserleichterung für Programmierer angesehen, größtes Augenmerk lag auf der unveränderten Schnelligkeit der erzeugten Programme gegenüber Assemblercode.

Circa Mitte der sechziger Jahre wurde Hardware preiswerter, die Aufgaben der Programmierung änderten sich dahingehend, daß Programme auf mehreren Systemen eingesetzt werden sollten. Außerdem gewann der Aspekt der nachweisbar korrekten Ausführung von Programmen an Bedeutung.

Mit fortschreitender Entwicklung besserer Hardware und der damit eingehenden Verbreiterung der Anwendungsbasis von Computern entstanden Programmiersprachen für spezielle Anwendungsbereiche, wie z. B. genaue Berechnungen, wissenschaftliche Anwendungen, militärische Nutzung oder anfängliche Forschung im Bereich künstlicher Intelligenz.

Damit wurde die Konkurrenz von einfacherer Programmierung und schneller Abarbeitung zugunsten spezialisierter Programmstrukturen entschieden, Geschwindigkeit spielte eine, wenn auch große, so doch nicht mehr überragende Rolle.

Mitte der achtziger Jahre begann die zunehmende Verbilligung und rapide Leistungssteigerung der Hardware. Damit gab es nun z. B. die Möglichkeit, hochspezialisierte Programmiersprachen zu entwickeln; Sprachen zu bauen, die auf unterschiedlichsten Hardwareplattformen liefen und auch programmiertheoretische Neuerungen einzuführen, deren Ausführung erst jetzt in akzeptabler Geschwindigkeit möglich wurde.

Heutzutage ist die Diskussion hinsichtlich der grundlegenden Programmiermethoden, wie funktionale oder objektorientierte Programmierung, relativ abgeschlossen. Es sind vielmehr innerhalb der Programmiermethoden Verfeinerungen einzelner Bearbeitungsschritte zu verzeichnen, ein großes neues Programmierparadigma ist im Augenblick nicht zu sehen.

Vielmehr wird versucht, die Programmierung so zu vereinfachen, daß aus metasprachlichen Umschreibungen von Problemen, wie sie im Softwareentwicklungsprozeß in vielfältiger Form entstehen, automatisch Programmquellcode generiert werden kann, dessen konkrete Implementierung damit für den Programmierer nicht mehr interessant zu sein braucht.

Ein solcher Ansatz, den auch diese Diplomarbeit verfolgt, besteht darin, für bestimmte, klar eingegrenzte Probleme eigene Beschreibungsformen zu schaffen, in unserem Fall eine eigene Programmiersprache. Der Softwareentwicklungsprozeß von Abbildung 1.3 wird dabei so umgestaltet, daß die Programmierung in zwei getrennten Zweigen verläuft, der Programmierung eines Programmrumpfs und der Programmierung des speziellen Problemquellcodes in der von uns definierten Programmiersprache.

Der Vorteil dieser Vorgehensweise liegt darin, daß der speziellere Code sehr problemgenau angepaßt werden kann. Dadurch wird eine metasprachliche Beschreibung erleichtert, da die Programmiersprache bereits die Termini des Problems verwenden oder zumindestens berücksichtigen kann. Wir kommen also dem Ziel der Generierung von Code aus den Dokumenten der Entwurfsphase einen großen Schritt näher.

Als nachteilig erweist sich, daß für jedes Problem eine eigene Sprache definiert werden muß, da keine universelle Programmiersprache eingesetzt werden soll. Damit steigt zum einen der Lernaufwand für den Programmierer, zum anderen auch die Zeit, die die Programmierungsphase benötigt.

Diese Nachteile können zum Teil umgangen werden, indem allgemeine Probleme weiterhin in universellen Programmiersprachen kodiert werden und

nur für sehr spezielle Anwendungsgebiete eigene Sprachen entwickelt werden. Zum anderen begünstigt die Spezialisierung der Sprachen die automatische Generierung von Quellcode aus den metasprachlichen Definitionen, der entstehende Problemquellcode ist für den Programmierer also zunehmend weniger interessant, die dem Problemquellcode zugrundeliegende Programmiersprache muß nicht mehr vollständig beherrscht werden.

Die Vorteile einer eigenen Programmiersprache überwiegen daher gegenüber den Nachteilen, die relativ leicht egalisiert werden können, so daß in diesem Konzept die Definition einer eigenen problemorientierten Sprache erfolgt.

Attribute einer guten Sprache

Die Entwicklung der Programmiersprachen über die Jahre führte zur Entwicklung neuer Sprachen, Verbesserung bestehender Sprachen oder dem Aussterben nicht mehr benötigter Sprachen.

Der Mechanismus, der dem Erfolg verschiedener Sprachen zugrunde liegt, ist von vielen Faktoren abhängig: z. B. spielen zeitliche Einflüsse eine Rolle, das wurde bereits im vorigen Abschnitt dargelegt; manche Sprachen wurden staatlich zum Standard erklärt; manche Sprachen sind für ein bestimmtes Spezialgebiet geeignet; andere bestechen durch Schnelligkeit.

So vielfältig die Faktoren für den Erfolg sind, meist haben sie mit der Sprache an sich wenig zu tun, externe Faktoren sind wesentlich häufiger die bestimmenden Einflußgrößen. Trotzdem sollte die Sprache bestimmte Attribute berücksichtigen, die für die „Schönheit“ einer Programmiersprache wichtig sind. Die angesprochene Schönheit ist dabei eine Mischung aus Zweckmäßigkeit, Syntax, Semantik und persönlichen Vorlieben des Programmierers und wird, leider, von Programmierer zu Programmierer unterschiedlich empfunden.

An dieser Stelle werden einige Faktoren genannt, die sich für eine gute Sprache als wichtig erwiesen haben und dabei unabhängig von persönlichen Vorlieben sind. Die Wichtung der einzelnen Attribute ist dabei dem Sprachentwickler überlassen, im allgemeinen sollten alle Faktoren gleichermaßen berücksichtigt werden.

- Eignung für die Anwendung
  Eine gute Sprache sollte eine solche Syntax haben, daß die Programmstruktur die zugrundeliegende Struktur des Algorithmus widerspiegelt. Im von uns angestrebten Idealfall ist es also möglich, aus der Problembeschreibung einen äquivalenten Programmcode automatisch zu generieren.
  Die Eignung für eine Anwendung wird hauptsächlich durch angemessene Datenstrukturen, Operationen, Kontrollstrukturen und eine „natürliche“ Syntax ermöglicht.

- Einfachheit
  Die Einfachheit einer Sprache drückt sich zum einen darin aus, wie ein-

fach es ist, algorithmische Funktionalität zu programmieren, zum anderen darin, wie einfach der Programmquellcode wieder gelesen werden kann.

- Einheitlichkeit
  Einheitlichkeit ist insofern wichtig, als daß Programmelemente wie z. B. Operatoren im gleichen Kontext dieselbe Semantik besitzen sollten. Idealerweise unterscheidet sich die Semantik auch in unterschiedlichem Kontext nicht allzusehr voneinander, so daß Programmkonstrukte kontextunabhängig dieselbe Bedeutung symbolisieren.

- Klarheit
  Eine Programmiersprache bildet den Rahmen zur Programmierung von Algorithmen. Gute Programmiersprachen unterstützen den Entwurf dieser Algorithmen bereits vor der eigentlichen Programmierung. Dabei hilft Klarheit im syntaktischen und semantischen Sinn, so daß Algorithmen so direkt wie möglich aus Konzepten erarbeitet werden können.

- Leichte Programmverifikation
  Meist ist ein zu erstellendes Programm nicht Selbstzweck für den Programmierer sondern muß eine bestimmte Funktionalität erfüllen. Es gibt viele verschiedene Möglichkeiten zu prüfen, ob ein Programm diese Funktion richtig ausführt. Eine gute Sprache unterstützt dabei die Programmverifikation, wobei einfache Syntax und Semantik meist auf einfache Programmverifikation hindeuten.

- Lesbarkeit
  Lesbarkeit ist eine Kombination von Einfachheit, Eignung für die Anwendung, Einheitlichkeit und Klarheit. Sie bestimmt, wie schnell und einfach die Funktion eines Programms aus dem zugehörigen Quelltext bestimmt werden kann. Sprachen, die hohe Lesbarkeit besitzen, werden oft auch als selbstdokumentierend bezeichnet, da der Aufwand für eine zusätzliche Dokumentation des Quelltextes sehr gering ist.

- Nutzungskosten
  Die Nutzungskosten einer Sprache sind in der Praxis oft das entscheidende Kriterium. Wir unterscheiden zwischen unterschiedlichen Kostenarten:
  Programmausführungskosten, die heutzutage meist in Form der Ausführungszeit gemessen werden;
  Kosten der Programmübersetzung, ebenfalls eine Funktion der Zeit aber auch des Einsatzzweckes;
  Kosten für Entwicklung, Test und Nutzung des Programms, meist nur in der Codierungs- und Testzeit angegeben;
  Kosten der Programmpflege, die in letzter Zeit an besonderer Bedeutung gewonnen haben, da Programmänderungen zu bestimmten Zeitpunkten gehäuft auftreten und vorausschauende Entscheidungen hier große Kosten sparen.

- Orthogonalität
  Dieser Begriff bezeichnet die Eigenschaft einer Programmiersprache, ver-

schiedene Sprachkonstrukte in allen möglichen Kombinationen benutzen zu können, wobei jede dieser Kombinationen eine sinnvolle Bedeutung hat. Es ist also möglich, Programmkonstrukte in verschiedenen Kontexten zu benutzen und jedesmal eine sinnvolle Bedeutung zu erhalten.
Man hat so den Vorteil, daß bei der Codierung eines Programms weniger Sonderfälle und Ausnahmen auftreten, die bedacht werden müssen. Andererseits werden so auch Programme erfolgreich compiliert, die logische Fehlkonstruktionen enthalten, die bei weniger Orthogonalität bereits bei der Compilierung aufgefallen wären.
Daher ist Orthogonalität zwar ein Kriterium für eine gute Programmiersprache, jedoch kann eine gute Programmiersprache je nach Vorliebe des Sprachdesigners viel oder auch wenig Orthogonalität besitzen. Orthogonalität steht in engem Zusammenhang mit Einheitlichkeit.

- Portabilität von Programmen
  Vor allem bei größeren Programmierprojekten ist die Portabilität des Programmcodes von großer Bedeutung, da die Investitionen in das Programm mit weiter Verbreitung so schnell wie möglich amortisiert werden.

- Programmierumgebung
  Dieser Aspekt wurde in der Vergangenheit oft unterschätzt und hat zur Verbreitung vieler schlechter Programmiersprachen geführt. Eine gute Programmierumgebung beginnt bei ausführlicher und einfach zu bedienender Dokumentation einer Sprache, geht weiter bei der Verfügbarkeit von Editoren und Testpaketen und umfaßt auch Einrichtungen zur Pflege und Änderung großer Programmpakete.

- Schreibbarkeit
  Leicht schreibbare Programme gehen einem Programmierer sozusagen von der Hand, es wird weniger Zeit für das eigentliche Programmieren benötigt. Andererseits stehen leicht schreibbare Programme oftmals im Widerspruch zur Forderung nach lesbaren Programmen. Es ist schwer, hier einen geeigneten Kompromiß zu finden, da viele Programmierer lieber schnell zu schreibende als einfach zu lesende Programme produzieren. Hier ist wieder die Wichtung des Sprachdesigners entscheidend.

- Unterstützung für Abstraktion
  Selbst bei einer für ein Problem sehr gut geeigneten Programmiersprache ist ein gewisses Maß an Abstraktion meist nicht vermeidbar. Die Sprache sollte daher diese Abstraktion so gut wie möglich unterstützen. Darin eingeschlossen ist meist die Möglichkeit des Programmierers, eigene Abstraktionen von Daten- oder Kontrollstrukturen zu erzeugen und später wiederverwenden zu können.

3 Das Konzept

3.1 Inhalt des Kapitels **43**
3.2 Zielsprache **44**
3.3 Konzept des Implementationsprozesses **45**
3.4 Schnittstellen **46**
3.4.1 Standardverhalten und getroffene Annahmen 48
3.4.2 Datenaustausch 49
3.4.3 Interne Schnittstellen 49
3.4.4 Externe Schnittstellen 52
3.4.5 Schnittstellen der Prüfroutinen 53
3.5 Ausnahmen **54**
3.6 Sprache **55**
3.6.1 Vorüberlegungen 55
3.6.2 Konventionen 57
3.6.3 Definition der Sprache 58
3.7 Transfer der veränderlichen Programmteile **66**
3.8 Zusammenfassung **66**

3.1 Inhalt des Kapitels

Das Konzept der Problemlösung verlangt mehr als die bloße Definition der geforderten Sprachen. So müssen außerdem die Schnittstellen definiert werden, eine praktikable Lösung der Betriebsphase muß konzipiert werden und schließlich muß der Umfang der zum Konzept gehörenden Klassen festgelegt werden.

Es erscheint daher sinnvoll, die Problemstellungen einzeln zu betrachten und anhand der gefundenen Lösungen ein einheitliches Konzept zu entwickeln. Bei der Einteilung der Einzelprobleme ist zu berücksichtigen, daß sich die einzelnen Lösungen gegenseitig beeinflussen, daher sollten die grundlegenden Probleme zuerst geklärt werden, bevor darauf aufbauende Lösungen gesucht werden.

Wir werden deshalb mit Überlegungen darüber beginnen, welche Zielsprache für das Konzept festgelegt wird. Danach wird untersucht, wie der Implementationsprozeß umgestaltet werden muß, um der Problemlösung dienlich zu sein. Darauf aufbauend untersuchen wir die geforderten Schnittstellen und betrachten Ausnahmebedingungen, die behandelt werden müssen. Daran anschließend wird die Programmiersprache entwickelt, in der die veränderlichen Programmteile implementiert werden sollen. Das Ende der Analyse bilden Überlegungen zum Transfer der veränderlichen Programmteile vom Entwickler zum Nutzer.

Den Abschluß des Kapitels bildet eine Zusammenfassung des Konzepts.

3.2 Zielsprache

Die Zielsprache war in der ursprünglichen Aufgabenstellung fest vorgegeben. Das damalige Programm wurde mit Java entwickelt. Es bleibt nun zu untersuchen, ob diese Wahl auch für das allgemeine Konzept übernommen werden kann.

Die Anforderungen an die Zielsprache sind, kurz zusammengefaßt:

1. „schöne“ Sprache für Programmierer, Entwickler sowie Theoretiker.
2. Objektorientierung, speziell Schnittstellen.
3. größtmögliche Verbreitung.

Für Java gilt in Bezug auf diese Anforderungen:

1. Java besitzt viele Sprachkonstrukte, die die Programmierung komplexer Algorithmen unterstützen. Des weiteren ist Java eine so junge Sprache, daß aus Fehlern und Stärken bestehender Sprachen gelernt werden konnte. So entstand eine konsequent objektorientierte moderne Sprache, die bequeme Methoden auch für komplizierte Aufgabenstellungen wie Vererbung, Ausnahmenbehandlung oder nebenläufige Programmierung zur Verfügung stellt.

   Nachteilig wirkt sich für Java die geringe Geschwindigkeit der entstehenden Programme aus, die den Einsatz der Programme behindert. Dieser Nachteil wirkt sich hauptsächlich bei geschwindigkeitskritischen Anwendungen aus, zu denen die Problemstellung nicht unbedingt gezählt werden kann.

2. Wie bereits angesprochen ist Java objektorientiert konzipiert. Dies bezieht sich auf Konzepte wie Vererbung oder Schnittstellenprogrammierung und auf Programmkonstrukte, die die Implementierung dieser Konzepte unterstützen.

3. Die Plattformunabhängigkeit von Java wird mittels des zweistufigen Verarbeitungsprozesses von Javaprogrammen unterstützt. Die Compilierung des Javaquellcodes erzeugt Bytecode, der durch die sogenannte *Java-Virtual-Machine* interpretiert wird. Diese virtuelle Maschine ist für die verschiedensten Plattformen erhältlich. Damit sind Javaprogramme sehr leicht sehr weit zu verbreiten.

Die Untersuchung der für die Zielsprache vorgegebenen Kriterien zeigt, daß Java sehr gut als Zielsprache geeignet ist. Es spricht daher nichts dagegen, Java wie im Ursprungsprojekt geplant, einzusetzen.

Die folgenden Konzepte werden daher in der Syntax von Java angegeben.

3.3 Konzept des Implementationsprozesses

Das zu entwickelnde Konzept erfordert eine Modifikation des Implementationsprozesses innerhalb des Softwareentwicklungsprozesses.

Bereits in Abbildung 1.2 wurde dargestellt, daß unter Nutzung der Spezifikation vom Programmierer der unveränderliche Programm- und GUI-Quellcode in herkömmlicher Weise geschrieben wird. Lediglich die veränderlichen Programmteile werden separat programmiert und dem Programmrumpf zur Verfügung gestellt.

In Abbildung 3.1 ist noch einmal der Herstellungsprozeß der veränderlichen Programmteile separat dargestellt.

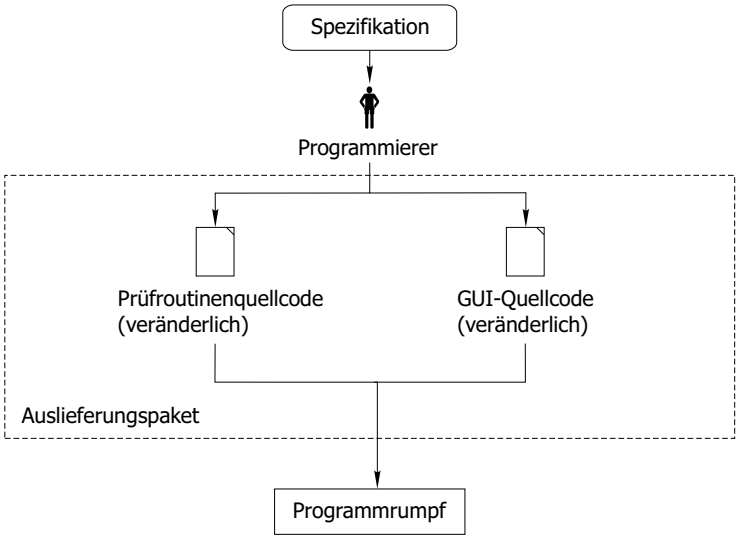

Abbildung 3.1: Herstellung der veränderlichen Programmteile

Quelle: Eigene Darstellung

Der Programmierer hat die Aufgabe, aus der Spezifikation den veränderlichen Quellcode für die Benutzeroberfläche und für die Prüfroutinen zu erstellen. Der entstehende Quellcode ist in der im Konzept zu entwickelnden Programmiersprache geschrieben.

Die in der Abbildung beschriebenen Quellcodes müssen nicht getrennt voneinander sein, es ist auch möglich, nur eine Programmiersprache und damit nur einen Quellcode zu erzeugen. Die in der Abbildung gezeigte Trennung verdeutlicht lediglich die logische Trennung der beiden Anwendungsebenen: graphische Oberfläche und Prüfroutinen.

Der entstandene Quellcode wird zur Übermittlung an den Nutzer in einem Auslieferungspaket zusammengefaßt. Dabei ist die Art der Übermittlung ein noch festzulegendes Detail, der Begriff des Pakets wird hier im Sinne einer Auslieferungseinheit und nicht im Sinn von Java-Packages gebraucht.

Da die Abbildung nur den eigentlichen Programmierprozeß abbildet, fehlen einige Details in der Darstellung. Dies sind sowohl die Schnittstellen als auch die Testroutinen. Wie bereits erwähnt, muß auch die Art der Übermittlung näher spezifiziert werden.

3.4 Schnittstellen

Innerhalb des Konzepts unterscheiden wir zwischen externen und internen Schnittstellen der einzubindenden Programmteile. Die Unterscheidung wird in Abbildung 3.2 verdeutlicht.

Es ist zu sehen, daß der zu generierende Programmteil, bestehend aus den Prüfroutinen sowie der Benutzeroberfläche, im Programmrumpf eingebettet wird.

So stehen dem Programmrumpf eine Schnittstelle für die Prüfroutinen sowie eine Schnittstelle für das GUI zur Verfügung. Diese Schnittstellen werden wir im folgenden als *externe* Schnittstellen bezeichnen, da sie nach außen hin dem Programmrumpf gegenüber als Verbindung fungieren. Sie sind in der Abbildung gestrichelt dargestellt.

Im Gegensatz dazu stehen die *internen* Schnittstellen, die die GUI-Einzelelemente gegenüber der zu generierenden Benutzeroberfläche verbergen. Die Bezeichnung wurde gewählt, da diese Schnittstellen für den Programmrumpf nicht sichtbar sind, sondern die interne Struktur der GUI-Einzelelemente vereinheitlichen. Die internen Schnittstellen sind in der Abbildung gepunktet.

Des weiteren wird deutlich, daß eine strikte Trennung zwischen dem Programm, in unserem Fall bestehend aus Programmrumpf und Prüfroutinen, sowie der Benutzeroberfläche bevorzugt wird.

Diese Trennung zwischen Programm und Benutzeroberfläche wird bereits bei Hübner ([7], S. 68 ff.) untersucht. In dieser schon etwas älteren Arbeit wird die Anzahl der Programme, die zwischen Programm und Benutzeroberfläche

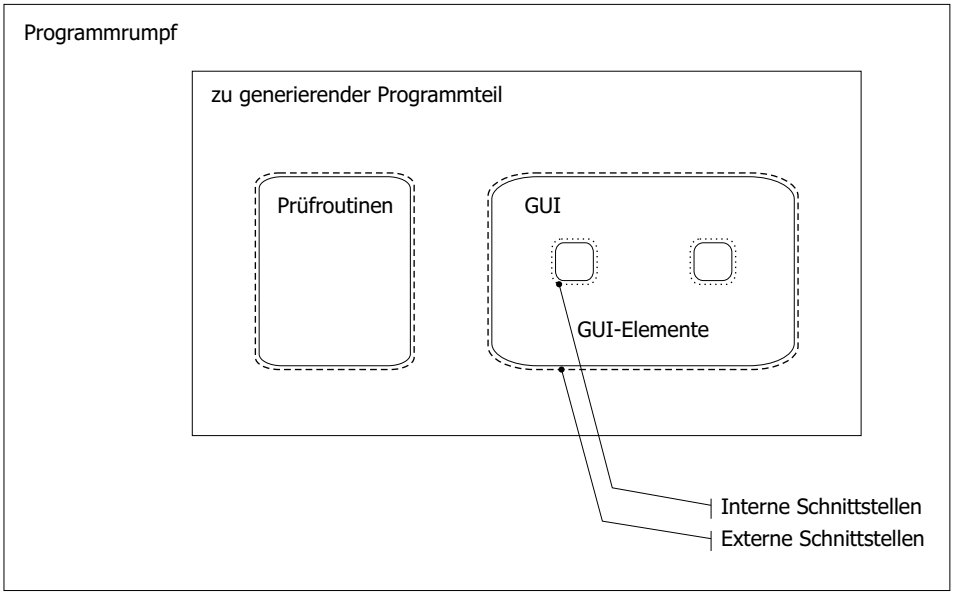

Abbildung 3.2: Schnittstellen

Quelle: Eigene Darstellung

trennen, mit deutlich über 50% angegeben. Mit zunehmender Einführung objektorientierter Programmiersprachen wie C++, Delphi oder Java kann davon ausgegangen werden, daß sich der Anteil der die Trennung vollziehenden Programme nicht verringert sondern erhöht hat. An angegebener Stelle wird auch der Vorteil der Trennung deutlich hervorgehoben, Programm und Benutzeroberfläche sind in der Entwicklung nicht mehr direkt voneinander abhängig, sondern entkoppelt, so daß Änderungen einer Komponente nicht zwingend in Änderungen an anderen Komponenten resultieren müssen.

Hübner [7] macht darüber hinaus deutlich, daß zwischen vier verschiedenen Möglichkeiten zur Kontrolle der Interaktion zwischen Programm und Benutzeroberfläche unterschieden werden kann:

- das Programm ruft die Benutzeroberfläche auf (externe Kontrolle)
- die Benutzeroberfläche ruft das Programm auf (interne Kontrolle)
- die Aufrufe erfolgen wechselseitig (wechselnde Kontrolle)
- die Kommunikation wird über eine eigene Verwaltung realisiert (gemischte Kontrolle)

Es wird weiter ausgeführt, daß wechselnde Kontrolle und gemischte Kontrolle die am häufigsten verwendeten Kommunikationsmodelle sind. Da sich unsere Voraussetzungen von denen der herkömmlichen Anwendungsprogrammierung unterscheiden, wird in unserem Konzept das Modell der externen

Kontrolle bevorzugt. Damit kann die gesamte Kontrolle dem Programm übergeben werden, die Oberfläche dient lediglich der Datenein- und -ausgabe. Dies wurde in der Problemstellung so gefordert. Die zweite, vorerst abgelehnte Möglichkeit bestünde darin, gemischte Kontrolle zu realisieren, d. h. die Kommunikation mittels dafür vorgesehener Kommunikationsklassen zu realisieren. Dies erfordert höheren Aufwand und ist für unser relativ einfaches Problem nicht nötig. Wir verwerfen jedoch eine solche Lösung nicht explizit, da die Schnittstellen der GUI-Elemente und Prüfroutinen gemischte Kontrolle durchaus zulassen, so daß in späteren Entwicklungsphasen von dieser Möglichkeit Gebrauch gemacht werden kann.

Im folgenden werden die Schnittstellen per Java-Quellcode definiert und ausführlich erläutert. Der vollständige Java-Quellcode ist im Anhang B.1 zu finden.

3.4.1 Standardverhalten und getroffene Annahmen

Bevor die Schnittstellen definiert werden können, muß geklärt werden, wie sich die Eingabeformulare standardmäßig verhalten sollen. Dabei ist die Abbildung 3.3 hilfreich, die zeigt, in welchen Rahmen die variablen Programmteile typischerweise eingebunden werden.

Abbildung 3.3: Beispiel der Standardoberfläche

Quelle: Eigene Darstellung

Der abgebildete Programmausschnitt stellt ein Beispiel für eine in der Problemstellung beschriebene Standardoberfläche dar. Der Programmrumpf, der am hellgrauen Hintergrund zu erkennen ist, umschließt die generierte Benutzeroberfläche. Dabei übernimmt der Programmrumpf alle Funktionen, die zur Generierung und Ansteuerung des GUIs notwendig sind. Der generierte Programmteil ist in einem Scrollfeld eingeschlossen und am helleren Hintergrund erkennbar.

Die Funktionalität ist derart, daß die Oberfläche vollständig vom Programm-

rumpf generiert wird. Die Interaktion zwischen Programmrumpf und generierter Benutzeroberfläche wird über die drei unteren Buttons vollzogen, mittels „OK“ sollen die eingegebenen Werte ausgelesen werden, „Zurücksetzen“ setzt alle Eingabefelder auf einen vorgegebenen Wert zurück, „Überprüfen“ ruft die Prüfroutinen auf, die die Eingaben überprüfen.

Für die variablen GUI-Elemente und die daraus zusammengesetzte veränderliche Benutzeroberfläche gilt zudem, daß zum einen die veränderliche Benutzeroberfläche aus einem oder mehreren GUI-Elementen zusammengesetzt werden kann, die entweder Basiselemente sind oder wiederum aus anderen GUI-Elementen gebildet werden können. Die Schachtelung von GUI-Elementen ist damit explizit erlaubt.

Zweitens kann ein GUI-Element Eingaben erlauben, z. B. Texteingabefelder, oder diese nicht gestatten, z. B. Beschriftungen. Es gibt auch GUI-Elemente, die Eingaben gestatten und eine Beschriftung enthalten können.

3.4.2 Datenaustausch

Die eingegebenen Daten müssen zwischen den GUI-Elementen untereinander und mit dem Programmrumpf ausgetauscht werden. Außerdem müssen der generierten Benutzeroberfläche Beschriftungen oder Farbwerte übergeben werden können.

Diese Daten sind vor der Generierung noch nicht in Umfang und Inhalt absehbar. Der Datenaustausch muß daher in möglichst allgemeiner Form definiert werden, um größtmögliche Flexibilität zu garantieren.

Die ausgetauschten Daten beziehen sich auf ein bestimmtes GUI-Element und enthalten meist einen bestimmten Wert, der diesem Element zugeordnet werden soll. Für die Übertragung so strukturierter Daten bietet sich eine sogenannte Hashtabelle an. Diese enthält einen eindeutigen Schlüssel und ordnet diesem einen bestimmten Wert zu.

Die Verwendung einer Hashtabelle bedeutet also, daß jedem GUI-Element ein eindeutiger Name zugeordnet werden muß. Dies ist eine leicht realisierbare Forderung, zusätzlich bedeuten eindeutige Namen auch für die GUI-Generierung Vorteile.

Die einem Schlüssel zugeordneten Daten können in der Java-Implementierung einer Hashtabelle, der Klasse *java.util.Hashtable*, alle in Java verfügbaren Objekte sein. Es muß daher zu jedem GUI-Element bekannt sein, welcher Objekttyp in der Hashtabelle übergeben wird und wie dieser innerhalb des GUI-Elements eingesetzt wird.

3.4.3 Interne Schnittstellen

Als erstes werden die internen Schnittstellen des Konzepts betrachtet. Es muß festgelegt werden, wie die GUI-Elemente angesprochen werden können.

Zur Erleichterung der Schnittstellendefinition listet die folgende Übersicht die für die internen Schnittstellen relevanten Punkte noch einmal auf.

- Die Schnittstellen müssen die tatsächliche Struktur der GUI-Elemente verbergen und einheitlichen Zugriff elementunabhängig ermöglichen.
- Die eingegebenen Werte müssen auf Anfrage an die aufrufende Klasse ausgegeben werden.
- Es muß die Möglichkeit bestehen, GUI-Elemente mit vordefinierten Werten zu füllen.
- Die Beschriftung der GUI-Elemente muß veränderbar sein.
- Gleiches gilt für die Farbe der GUI-Elemente.
- Ein GUI-Element muß in einen Fehler oder einen Warnzustand versetzt werden können. Dieser muß rücksetzbar sein.

Die Umsetzung dieser Forderungen ist auf zweierlei Art möglich, zum einen kann die Methodenauswahl so stattfinden, daß zu einem GUI-Element gehörende Manipulationen gemeinsam durchgeführt werden, es würde eine Methode zur Änderung von Werten und Farben benötigt. Die andere Möglichkeit besteht darin, die Manipulation der Werte bzw. Beschriftungen von der der Farben zu trennen.

Da sich die Werte der Eingabefelder in der Regel während eines Programmlaufs im Vergleich zu der Farbgebung häufiger ändern, bevorzugen wir die zweite Möglichkeit der Trennung der manipulierenden Methoden, um nicht ständig unnötig viele Daten übermitteln zu müssen.

Damit erhalten wir präzise Anforderungen und Lösungsansätze, die im folgenden umgesetzt werden. Die Schnittstellen werden in Java-Quellcode angegeben und ausführlich erläutert. Ein kompletter Überblick sowie der vollständige Quellcode der Schnittstellen findet sich im Anhang B.

Die erste Festlegung, die getroffen werden muß, ist die Wahl eines eindeutigen Namens. Wir wollen GUI-Elemente generieren und wählen daher als Name `EntryElement`. Da Java die Möglichkeit bietet, zusammengehörige Klassen in *Packages* zu ordnen, nutzen wir diese Möglichkeit und teilen die Schnittstelle dem Package `de.ekkart.diplomarbeit.interfaces` zu.

Das Package `java.util.*` wird importiert, da für die Datenübergabe die Hashtabellen-Klasse benötigt wird, die sich im genannten Package befindet. Das Package `de.ekkart.diplomarbeit.exceptions.*` enthält die benutzte Exception und wird ebenfalls importiert. Wir erhalten:

```
package de.ekkart.diplomarbeit.interfaces;
import java.util.*;
import de.ekkart.diplomarbeit.exceptions.*;
public interface EntryElement {
```

Die nächsten Methoden realisieren den Werteaustausch zwischen GUI-Element und der das Element einbindenden Klasse. Der Austausch erfolgt in beide Richtungen, daher sind zwei Methoden vorzusehen, eine für die Werteübergabe vom aufrufenden Objekt an das GUI-Element (`putValues`). Die andere Methode dient der Werteübergabe vom GUI-Element zum aufrufenden Objekt (`getValues`).

Der Rückgabetyp der `putValues`-Methode ist `void`, d. h. es wird kein Wert zurückgegeben. Der Parameter `hshNewValues` ist die Wertehashtabelle, die die für das GUI-Element neuen Werte enthält.

Bei diesen und allen folgenden Methoden, die Hashtabellen zum Datenaustausch nutzen, wird von der Prämisse ausgegangen, daß GUI-Elemente, die nicht als Schlüssel in der Hashtabelle auftreten, bedeuten, daß das entsprechende Element unverändert bleibt.

Die `getValues`-Methode gibt die Hashtabelle zurück, die die eingegebenen Werte des GUI-Elements enthält. Die Methode benötigt keine Parameter.

Beide Methoden können eine sogenannte `MethodNotSupportedException` auslösen. Diese Ausnahme ist im Abschnitt 3.5 näher erläutert. Der Ausnahmefall tritt ein, wenn versucht wird, einem reinen Beschriftungsfeld Werte zuzuweisen bzw. Werte aus diesem auszulesen.

```
Hashtable getValues()
  throws MethodNotSupportedException;
void putValues(Hashtable hshNewValues)
  throws MethodNotSupportedException;
```

Es fehlt jetzt noch die Möglichkeit, Beschriftung (in Text und Aussehen) sowie die Optik der GUI-Elemente zu verändern. Da wir uns für die Trennung in verschiedene Methoden entschieden haben, benötigen wir dafür drei einzelne Methoden.

Die `changeTextColor`-Funktion dient dazu, den Beschriftungen andere Farben zuzuweisen. Der Zugriff auf die Beschriftungen erfolgt wiederum über eindeutige Namen in einer Hashtabelle. Hier besteht der zu einem Schlüssel gehörende Wert allerdings aus einem Wertefeld mit zwei Werten, dem ersten für die Vordergrund-, dem zweiten für die Hintergrundfarbe.

Analog wird die `changeFieldColor`-Methode implementiert, die die Farbe der Eingaben verändert.

Mittels `changeTextTexts` besteht letztendlich die Möglichkeit, die Beschriftungstexte zu ändern. Der Zugriff erfolgt ebenfalls über eindeutige Namen einer Hashtabelle, als Wert wird ein String-Wert gespeichert.

Alle drei Methoden können ebenfalls eine `MethodNotSupportedException` auslösen. Diese tritt dann auf, wenn reinen Beschriftungsfeldern eine Eingabefeldfarbe oder reinen Eingabefeldern Beschriftungsattribute zugewiesen werden sollen.

```
void changeFieldColor(Hashtable hshNewValues)
```

```
    throws MethodNotSupportedException;
  void changeTextColor(Hashtable hshNewValues)
    throws MethodNotSupportedException;
  void changeTextTexts(Hashtable hshNewValues)
    throws MethodNotSupportedException;
```

Zuletzt wird die Möglichkeit gegeben, das Element in einen Fehler- oder einen Warnzustand zu versetzen. Dazu dient die Methode `setErrorState`, der eine Hashtabelle übergeben wird, die die Namen der zu setzenden Elemente sowie den jeweils zugehörigen Fehlerstatus als Zeichenkette enthält.

Ein Fehlerstatus kann dabei die Werte *error*, *warning* oder *normal* enthalten, die den Zuständen Fehler, Warnung oder Normalzustand entsprechen.

```
  void setErrorState(Hashtable hshNewStates);
}
```

Damit ist die interne Schnittstelle der GUI-Elemente komplett beschrieben.

3.4.4 Externe Schnittstellen

Es bleibt nun noch die Aufgabe, die externen Schnittstellen festzulegen, die zwischen dem Programmrumpf und der generierten Benutzeroberfläche bestehen.

Der Programmrumpf muß dem generierten GUI Werte zuweisen und diese wieder auslesen können, außerdem sollen die Farb- und Textattribute manipuliert werden können. Weiterhin muß der GUI-Quellcode aus einer Datei eingelesen und generiert werden.

Wir können feststellen, daß die Funktionalität größtenteils der bereits bekannten Funktionalität der GUI-Elemente entspricht. Als neue, nur an das generierte GUI zu stellende Forderung erscheint lediglich:

- Es muß eine Methode geben, die die graphischen Definitionen aus einer externen Datei einliest und die dazugehörige Benutzeroberfläche generiert.

Wir beginnen die Definition wieder mit der Festlegung des Namens, die neue Schnittstelle wird `EntryForm` heißen. Sie liegt ebenfalls im Package `de.ekkart.diplomarbeit.interfaces`.

Der Import beschränkt sich auf das Paket `java.io.*`, das Möglichkeiten zur Ein- und Ausgabe zur Verfügung stellt.

```
package de.ekkart.diplomarbeit.interfaces;
import java.io.*;
public interface EntryForm extends EntryElement {
```

An dieser Stelle können wir von der strikten Objektorientiertheit Javas profitieren und definieren, daß die generierte Oberfläche (`EntryForm`) die Schnittstelle von GUI-Elementen (`EntryElement`) erweitert und können so die bereits definierte Funktionalität der GUI-Elementschnittstelle weiter benutzen.

Damit benötigt unsere Schnittstelle lediglich eine zusätzliche Methode zum Einlesen und zur Generierung der Oberflächenbeschreibung, da die Zugriffsmethoden auf Werte und Aussehen bereits in der Schnittstelle für GUI-Elemente enthalten sind.

Wir nennen die Methode `generateMask`. Sie besitzt einen Parameter `fleDescription`, in dem die Datei übergeben wird, die die Beschreibung der Oberfläche beinhaltet. Der Rückgabewert der Methode ist vom Typ *boolean* und wird auf *true* gesetzt, wenn Einlesen und Generierung ohne Fehler vonstatten gingen, der Rückgabewert wird auf *false* gesetzt, wenn ein Input/Output-Fehler auftrat oder die Generierung der Oberfläche Probleme verursachte.

```
  boolean generateMask(File fleDescription)
}
```

Damit ist die externe Schnittstelle der generierten Benutzeroberfläche vollständig beschrieben.

3.4.5 Schnittstellen der Prüfroutinen

Die Programmierung der Prüfroutinen kann durch eine einzige Schnittstelle abgedeckt werden. Diese muß folgende Methoden zur Verfügung stellen:

- Der gelieferte Prüfroutinenquellcode muß aus einer externen Datei eingelesen und verarbeitet werden.
- Es muß möglich sein, die generierte Benutzeroberfläche auf Richtigkeit der Eingaben zu prüfen.

Diese Forderungen gilt es nun umzusetzen. Wie bereits bei den zuvor definierten Schnittstellen muß auch diese zunächst benannt werden. Es wurde der Name `FormChecker` gewählt. Auch diese Schnittstelle bekommt einen Platz im Package `de.ekkart.diplomarbeit.interfaces`.

Es werden die Pakete `java.io.*` und `java.util.*` importiert, da wir Dateizugriff und zur Datenübertragung eine Hashtabelle benötigen. Der Quellcode sieht dann wie folgt aus:

```
package de.ekkart.diplomarbeit.interfaces;
import java.io.*;
import java.util.*;
public interface FormChecker {
```

Als erstes wird eine Methode zum Einlesen des Prüfroutinenquellcodes vorgesehen, analog zur `EntryForm`-Klasse wird der Methode als Parameter eine Datei übergeben, der Rückgabewert ist ein boolesches Flag mit *true*, wenn der Einlesevorgang problemlos beendet wurde und *false*, wenn Probleme auftraten.

```
boolean readCheckData(File fleCheckdata);
```

Die zweite Methode dient zur Ausführung der eigentlichen Prüffunktionalität. Als Parameter wird zum einen die Quelle der eingegebenen Daten, die generierte Benutzeroberfläche, also die Schnittstelle `EntryForm` übergeben. Zum anderen benötigen wir eine Speicherstruktur, die im Fehlerfall die Fehlermeldungen zur weiteren Verarbeitung zur Verfügung stellt. Hier wird wieder eine Hashtabelle benutzt, deren Schlüsselwerte die fehlerverursachenden GUI-Elementnamen und deren Werte die generierten Fehlermeldungen darstellen.

Der Rückgabewert ist ein boolesches Flag, daß mit *true* anzeigt, daß keine Fehler gefunden wurden und mit *false* signalisiert, daß Fehler bei der Überprüfung auftraten.

```
  boolean checkForm(EntryForm frmDatasource,
                    Hashtable hshErrors);
}
```

Diese zwei Methoden genügen, um die Funktionalität der Prüfroutinen abzusichern.

3.5 Ausnahmen

Ausnahmen dienen dazu, während des Programmlaufs definiert auf Ausnahmebedingungen zu reagieren. Diese Ausnahmebedingungen können in bekannten Bereichen auftreten, sie können aber auch unvermittelt eintreten. Für den Programmierer ist wichtig, eine Möglichkeit zu besitzen, Laufzeitfehler, und als solche treten Ausnahmebedingungen meist auf, gezielt abzufangen und per Programmcode darauf zu reagieren.

In Java werden Ausnahmen durch das Konzept der Exceptions realisiert. Es besteht die Möglichkeit, die verfügbaren Ausnahmen mit eigenen Definitionen zu erweitern. Dies kann vielfältige Gründe haben, oft ist die klare Identifizierung der Ausnahmesituation die Ursache dafür, eine eigene Ausnahme zu kreieren.

Dieses Kapitel stellt die im Konzept benutzte Ausnahme kurz vor, der komplette Quellcode ist im Anhang B.2 zu finden.

Die einzige zu definierende Ausnahme ist die `MethodNotSupportedException`. Sie dient dazu, anzuzeigen, daß eine Methode nicht von der jeweiligen Klasse unterstützt wird.

Alle eine Schnittstelle implementierenden Klassen müssen alle in der Schnittstelle definierten Methoden implementieren. Dies kann zu Problemen führen, wenn sich Schnittstellen nur in wenigen Methoden unterscheiden. Dann können entweder mehrere Schnittstellen definiert werden oder es wird eine einzige Schnittstelle angegeben, die allgemeiner gehalten ist.

Dieser Weg führt dazu, daß einige implementierende Klassen Methoden beinhalten, die nicht unbedingt benötigt werden. Dieser Fall wird in unserem Konzept durch die `MethodNotSupportedException` abgefangen.

So sollen z. B. in unserem Konzept alle GUI-Elemente eine einheitliche Schnittstelle bekommen. Die `getValues`-Methode ist aber für GUI-Elemente, die Eingaben nicht erlauben, z. B. Beschriftungen, nicht notwendig. Es wird eine allgemeine Schnittstelle definiert, und der Aufruf der `getValues`-Methode durch die Ausnahmebedingung innerhalb der implementierenden Klassen abgefangen.

Damit bleibt die Schnittstelle klar und übersichtlich und die aufrufende Klasse kann auf die verschiedenen GUI-Elemente angemessen reagieren.

3.6 Sprache

Da mit den Schnittstellen und Ausnahmen die wichtigsten Vorbereitungen getroffen sind, kann nun die eigentliche Beschreibungssprache entwickelt werden.

In Vorbereitung dazu werden einige Vorüberlegungen angestellt, bevor die Definition der Sprache beginnt. Diese wird verbal und formal gemischt stattfinden. Die rein formale Beschreibung der Sprache findet sich im Anhang A.

3.6.1 Vorüberlegungen

Gemeinsame oder getrennte Sprachen

Die Kriterien des zu entwickelnden Konzepts legen noch nicht fest, ob eine gemeinsame oder zwei getrennte Sprachen für den GUI- und Prüfroutinenquellcode entwickelt werden sollen. Zur Entscheidung dieses Problems werden die Vor- und Nachteile der Ansätze gegenübergestellt und gegeneinander abgewogen.

Die Vorteile einer gemeinsamen Sprache sind:

- Es existiert eine gemeinsame Syntax für beide Beschreibungen, die dafür sorgt, daß der Programmierer schnell beide Aspekte bearbeiten kann, ohne sich an unterschiedliche Schreibweisen gewöhnen zu müssen.

- Es existiert eine gemeinsame Semantik, soweit dies möglich ist. Dies führt ebenfalls zu Erleichterungen im Entwicklungsprozeß.

- Sowohl die GUI-Beschreibung als auch die Prüfroutinen können im selben Sprachraum arbeiten, z. B. wenn dieselben Bezeichner in beiden Teilen verwendet werden.
- Es besteht die Möglichkeit, die gesamte geänderte Beschreibung in einer einzigen Datei auszuliefern. Damit werden Unbequemlichkeiten und potentielle Fehlermöglichkeiten seitens des Nutzers möglichst gering gehalten.
- Es ist lediglich ein einziger Parser für beide Beschreibungen notwendig.
- Die logische Zuordnung von GUI-Elementen und deren Prüfroutinen, die der Fehlervermeidung dient, wird betont.

Die Vorteile getrennter Sprachen zeigen sich wie folgt:

- Eine getrennte Syntax schafft die Möglichkeit, die unterschiedlichen Anforderungen und Aspekte der Beschreibungen deutlich hervorzuheben.
- Getrennte Semantik schafft die gleichen Vorteile, maßgeschneiderte Lösungen können so realisiert werden.
- Die Betonung der logischen Trennung der einzelnen Beschreibungsziele ermöglicht es, eventuell die Trennung auch personell besser zu vollziehen und damit spezialisierte Fähigkeiten zu unterstützen.

Es existieren also gute Gründe sowohl für eine Trennung der Sprachen als auch dagegen. Unser Konzept verfolgt einen Mittelweg, der versucht, die Stärken der einzelnen Ansätze zu betonen, ohne die Schwächen zu stark hervortreten zu lassen.

Es wird eine gemeinsame Sprache entwickelt, die beide Bereiche der Beschreibung abdeckt. Dies hat hauptsächlich praktische Gründe, zum einen wird so nur ein Parser benötigt, der für beide Teilbereiche eingesetzt werden kann. Zum anderen braucht vom Programmierer nur eine Sprache erlernt zu werden, schnellerer praktischer Nutzen steht im Vordergrund.

Allerdings wird die Sprache deutlich getrennte syntaktische und semantische Strukturen schaffen, die den jeweiligen Einsatzzweck im Bereich GUI- oder Prüfbeschreibung deutlich erkennbar machen. Es wird möglich sein, beide Bereiche einzeln zu beschreiben und damit auch ausliefern zu können.

Syntaktische Nähe zu bestehenden Sprachen

Die Syntax der zu entwickelnden Sprache orientiert sich an den Möglichkeiten von Java. So wird z. B. eine Gruppierung ebenfalls durch geschweifte Klammern vorgenommen.

Dies hat den Vorteil, daß bekannte syntaktische Elemente sofort übernommen werden können. Wichtig ist allerdings in diesem Zusammenhang, daß unterschiedliche semantische Konstrukte nicht syntaktisch zweideutig dargestellt werden können.

So böte es sich beispielsweise an, die GUI-Elemente der generierten Benutzeroberfläche oder die Oberfläche selbst als Klasse (`class`) zu bezeichnen. Dies würde sofort zu Problemen führen, da Klassen in Java eine andere Bedeutung besitzen, die in der zu entwickelnden Beschreibungssprache keine Entsprechung fänden. Daher muß darauf geachtet werden, Begrifflichkeiten strikt zu trennen.

Dynamik der Programmiersprache

Die zu entwickelnde Programmiersprache beschreibt hauptsächlich statische Eigenschaften des veränderlichen GUIs. Dies bedeutet, daß der GUI-Quellcode lediglich dazu dient, die Oberfläche zu erzeugen und dann unverändert zu lassen. Auch die Prüfroutinen müssen vom Programmrumpf aufgerufen werden und liefern die Fehlermeldungen an den Programmrumpf zurück. Dieser entscheidet, wie auf Fehlersituationen reagiert wird.

Während eines Programmlaufs können aber auch Situationen auftreten, die eine dynamische Reaktion auf Eingaben des Nutzers sinnvoll erscheinen lassen. So kann z. B. das Anklicken einer Option dazu führen, daß eine andere Eingabe nicht mehr sinnvoll ist, die verborgen oder unzugänglich gemacht werden soll. Dies soll natürlich sofort nach einer Eingabe durch den Nutzer erfolgen.

Die Programmiersprache muß daher erstens eine statische Repräsentation der Benutzeroberfläche enthalten. Zweitens sollen statische Prüfbedingungen formuliert werden, die vom Programmrumpf in Prüfroutinen umgewandelt werden. Diese Prüfroutinen können dann durch den Programmrumpf aufgerufen werden. Drittens muß die Möglichkeit bestehen, dynamische Prüfbedingungen zu definieren, die die Eingaben des Nutzers abwarten, um sofort darauf reagieren zu können.

3.6.2 Konventionen

In diesem Abschnitt wie auch im Anhang A wird die EBNF zur Beschreibung der Sprachsyntax eingesetzt.

Dabei gelten folgende Annahmen, die durch kurze nachgestellte Beispiele visualisiert werden:

(1) Nichtterminale werden mit spitzen Klammern gekennzeichnet.
„⟨`nonterminal`⟩“

(2) Terminale werden fett gedruckt.
„**terminal**“

(3) Eine Produktion besitzt auf der linken Seite ein Nichtterminal, es folgt das `::=`-Zeichen, die rechte Seite enthält die Definition des Nichtterminals.
„⟨nonterminal⟩ ::= ⟨nonterminal⟩ **terminal**“

(4) Mehrere Möglichkeiten der Definition eines Nichtterminals werden durch einen senkrechten Strich getrennt.
„⟨nonterminal⟩ ::= **terminal1 | terminal2**“

(5) Die leere Ersetzung wird durch ein Nichtterminal mit Namen *empty* erreicht.
„⟨nonterminal⟩ ::= ⟨empty⟩“

(6) Die Gruppierung von Definitionen wird mit runden Klammern realisiert.
„⟨nonterminal⟩ ::= **(terminal1 | terminal2) terminal3**“

(7) Optionale Zeichenfolgen werden in eckige Klammern eingeschlossen.
„⟨nonterminal⟩ ::= ⟨empty⟩ | [⟨nonterminal⟩] **terminal**“

(8) Zeichenfolgen, die beliebig oft (auch nullmal) wiederholt werden können, werden in geschweifte Klammern eingeschlossen.
„⟨nonterminal⟩ ::= {**terminal**}“

(9) Die oben aufgeführten Zeichen mit Sonderbedeutung können als Terminalzeichen verwendet werden, sie müssen dann in Hochkommata eingeschlossen werden.
„⟨nonterminal⟩ ::= {'{'**terminal**'}'}“

(10) Aufzählungen von Zeichen können bei eindeutiger Fortsetzung der Folge durch Angabe des Anfangs- und Endzeichens und dazwischengeschalteten Auslassungspunkten angegeben werden.
„⟨digit⟩ ::= **0|...|9**“

Einschränkungen, die nur schwierig in EBNF ausgedrückt werden können, werden im Text als Bedingungen formuliert. Diese Formulierungen müssen bei unveränderter Syntax in die Semantik der Programmiersprache aufgenommen werden. Sie können in den meisten Fällen aber auch durch eine umfangreichere Syntaxdefinition ersetzt werden.

Während der Sprachdefinition wird der Begriff *Programm* durchgehend als Synonym für einen Quelltext, der in der entwickelten Programmiersprache geschrieben ist, verwendet. Es handelt sich also nur um den Quelltext der veränderlichen Programmteile, nicht des Programmrumpfs oder des gesamten Programms.

3.6.3 Definition der Sprache

In diesem Kapitel wird die Programmiersprache entwickelt. Die vorhergehenden Abschnitte versuchten, die Aufgaben der Programmiersprache sowie die Restriktionen, denen sie unterworfen sein wird, so genau und differenziert wie möglich zu untersuchen.

Von diesen Daten ausgehend wird die Programmiersprache definiert und die Syntax in EBNF-Notation dargestellt. Es werden alle Konstrukte vorgestellt, je nach Wichtigkeit des Konstrukts sind die verbalen Ausführungen mehr oder weniger ausführlich. Im Fließtext werden zusätzliche Kontextbedingungen beschrieben, die der Semantik der Sprache zuzuordnen sind. Der Fließtext ergibt im Zusammenhang mit den notierten syntaktischen Konstrukten die informelle Semantikbeschreibung.

Die vollständige zusammenhängende EBNF-Notation der entwickelten Programmiersprache ist im Anhang A.1 zu finden.

Lexikalische Einheiten

Die lexikalischen Einheiten, die Token der Programmiersprache, sind alle Zeichenkombinationen, die durch ein Leerzeichen getrennt werden können.

Weiterhin trennen Klammern, Kommata, Sonderzeichen und das Semikolon lexikalische Einheiten, im Gegensatz zu Leerzeichen sind sie ebenfalls lexikalische Einheiten.

Alle Nichtbuchstaben und Nichtzahlen werden als Sonderzeichen bezeichnet.

Alle Token können durch beliebig viele Leerzeichen und Zeilenumbrüche getrennt werden, so daß durch Einrückung erhöhte Lesbarkeit geschaffen werden kann.

Syntax und Semantik

Kommentare

Kommentare können überall im Quelltext stehen, sie werden durch `//` eingeleitet, womit die gesamte restliche Zeile als Kommentartext angesehen und bei der Übersetzung ignoriert wird.

Anweisungen und Anweisungsblöcke

Alle Anweisungen müssen mit einem Semikolon abgeschlossen werden. Dies gilt nicht, wenn die Anweisung mit einer schließenden geschweiften Klammer endet.

Alle Anweisungsblöcke sind zwingend in geschweifte Klammern zu setzen. Dies wird bereits bei der Syntaxdefinition festgeschrieben. Die Vollklammerung erhöht den Schreibaufwand minimal, liefert dafür aber sehr gut lesbaren und gegenüber optionaler Klammerung meist fehlerfreieren Programmcode.

Gross- und Kleinschreibung

In der gesamten Sprache und insbesondere bei den Bezeichnern wird zwischen Groß- und Kleinschreibung unterschieden.

Alle reservierten Wörter sind kleingeschrieben.

Programm

Ein Programm besteht aus drei Hauptteilen, der GUI-Definition, der Definition der statischen Prüfbedingungen sowie der Definition der dynamischen Prüfbedingungen. Die Hauptabschnitte dürfen nur einmal im Programm vorkommen, sie können auch entfallen, um z. B. getrennte Dateien für GUI- und Prüfquellcode zu erstellen.

Da die Prüfbedingungen im wesentlichen das Verhalten des Programms im Hinblick auf den Aufruf der Prüfroutinen und die Reaktion auf fehlerhafte Eingaben bestimmen, wurden die Teile, die den zugehörigen Programmcode enthalten, mit `behaviour` bezeichnet.

Jeder Hauptabschnitt besteht aus einer *define*-Anweisung, deren Anweisungsblock die jeweilige Definition des Abschnitts enthält. Diese Definition setzt sich aus beliebig vielen Anweisungen zusammen, die dem jeweiligen Kontext entsprechen müssen.

```
⟨program⟩          ::=  [define view as '{'
                         {⟨guiStatement⟩}
                        '}']
                        [define static behaviour as '{'
                         {⟨staticStatement⟩}
                        '}']
                        [define dynamic behaviour as '{'
                         {⟨dynamicStatement⟩}
                        '}']
```

GUI-Definitionen

Die Benutzeroberfläche besteht aus den einzelnen GUI-Elementen, die mit Hilfe der *create*-Anweisung erzeugt werden können. Dabei müssen ein eindeutiger Name gewählt und dem Element ein Elementtyp zugewiesen werden. In geschweiften Klammern folgen die Definitionen des erzeugten Elements.

Innerhalb der GUI-Beschreibung muß genau einmal das sogenannte *main*-Element erzeugt werden. Dieses Element ist die Verbindung zwischen dem Programmrumpf und der generierten Benutzeroberfläche. Der ihm zugewiesene Elementtyp muß die `EntryForm`-Schnittstelle implementieren.

Bereits definierte Elemente können mit Hilfe der *group*-Anweisung temporär gruppiert werden. Auf alle in der Gruppe enthaltenen Elemente werden dann die im Anweisungsblock angegebenen Definitionen angewandt. Dadurch können ähnliche Elemente schnell und einfach in Aussehen und Verhalten angepaßt werden.

```
⟨guiStatement⟩     ::=  create [main] ⟨identifier⟩ as ⟨elementType⟩ '{'
                         {⟨elementDefinition⟩}
                        '}'
                      |  group ⟨elementGroup⟩ for '{'
                         {⟨elementManipulation⟩}
                        '}'
```

Zur Erzeugung der GUI-Elemente stehen fünf verschiedene Methoden zur Verfügung.

Die *set data*-Anweisung dient dazu, dem Element einen Defaultwert zuzuwei-

sen. Das reservierte Wort `data` unterstreicht dabei, daß der zu setzende Wert je nach GUI-Element unterschiedliche Datentypen besitzen kann. Der angegebene Wert muß also zum im Element benutzten Datentyp passen.

Die *format*-Anweisung kann dazu benutzt werden, der Ein- und Ausgabe des Elements einen Datentyp zuzuweisen. Dies ist meist nur nötig, wenn sich der zuzuweisende Datentyp vom Defaultdatentyp des Elements unterscheidet oder dieser durch einen Formatierungsstring eingeschränkt werden soll.

Wir haben weiterhin die Möglichkeit, mit der *set*-Anweisung allgemein den vorhandenen Eigenschaften des Elements Werte zuzuweisen. Auch hier muß auf passende Datentypen geachtet werden.

Mit Hilfe der *add*-Anweisung können GUI-Elemente zu dem aktuell bearbeiteten Element addiert werden. Sie erscheinen an der angegebenen Position, wobei der erste Wert die Zeile und der zweite Wert die Spalte angibt. Dabei wird von der Position `(0, 0)` ausgegangen, die sich oben links im Element befindet. Diese Element-Addition kann nur bei Datentypen erfolgen, die das Java-Standardlayout `GridBagLayout` oder ein entsprechendes Pendant vorweisen können, bei dem Elemente auf die vorgestellte Art addiert werden können.

Die *add*-Anweisung bietet außerdem die Möglichkeit, ein Element temporär zu erzeugen und sofort zu addieren. Dann werden nur der Datentyp und die Elementdefinitionen angegeben und das Element addiert.

Die *connect*-Anweisung dient dazu, das definierte Element an ein anderes zu binden. Meist wird ein Eingabeelement an ein oder mehrere Anzeigeelemente gebunden. Bei einem auftretenden Fehler in einem Element kann dieses Element in einen Fehlerzustand versetzt werden. Dann werden alle mit ihm verbundenen Elemente ebenfalls in den Fehlerzustand versetzt. Diese können entsprechend darauf reagieren.

```
⟨elementDefinition⟩      ::=  set data '=' ⟨dataValue⟩';'
                           |  format data as ⟨dataType⟩';'
                           |  ⟨valueAssignment⟩
                           |  add (⟨identifier⟩ | ⟨elementType⟩)
                                [with '{'⟨elementDefinition⟩'}']
                                at position⟨coordinates⟩';'
                           |  connect to ⟨identifier⟩';'
⟨elementGroup⟩           ::=  '{'⟨identifier⟩{, ⟨identifier⟩}'}'
⟨elementManipulation⟩    ::=  ⟨valueAssignment⟩
⟨valueAssignment⟩        ::=  set ⟨identifier⟩ '=' ⟨dataValue⟩';'
```

Statisches Verhalten

Für das statische Verhalten steht zunächst eine Anweisung zur Verfügung, die die Eingabe in ein Element als zwingend erforderlich deklariert. Das bedeutet, daß eine Eingabe in das Element erfolgen muß, anderenfalls werden die angegebenen Anweisungen ausgeführt, wobei normalerweise ein Fehler- oder Warnzustand gesetzt wird.

Sollte bei der Definition in Klammern eine Zahl angegeben werden, so bedeu-

tet dies, daß das Eingabeelement die angegebene Anzahl an Werten zurückliefern muß. Eine solche Zahl kann natürlich nur in sinnvollem Zusammenhang eingesetzt werden, wenn das Eingabeelement mehrere Werte liefern kann. Bei falscher Nutzung tritt ein Laufzeitfehler auf.

Des weiteren kann überprüft werden, ob die Eingabe eines Elements eine bestimmte Bedingung erfüllt. Falls nicht, können wiederum weitere Anweisungen angegeben werden, auch hier wird meist der Fehlerzustand des Elements verändert.

Das statische Verhalten umfaßt eine *if*-Anweisung, die zum Aufbau komplexerer Tests der Eingaben eines oder mehrerer Elemente dienen kann. Innerhalb der Anweisung oder im optionalen *else*-Zweig können wiederum statische Anweisungen stehen.

Die Fehleranweisung ermöglicht es, einen Fehler- oder Warnzustand mit dazugehöriger Meldung zu setzen.

```
⟨staticStatement⟩        ::=  ⟨requiredAssignment⟩
                           |  ⟨conditionCheck⟩
                           |  ⟨staticIfStatement⟩
                           |  ⟨errorStatement⟩
⟨requiredAssignment⟩     ::=  set ⟨identifier⟩ to required['('⟨integer⟩')']
                               otherwise '{'
                                 {⟨staticStatement⟩}
                              '}'
⟨conditionCheck⟩         ::=  ⟨identifier⟩ hasToBe ⟨condition⟩
                               otherwise '{'
                                 {⟨staticStatement⟩}
                              '}'
⟨staticIfStatement⟩      ::=  if ⟨booleanExpression⟩ '{'
                                {⟨staticStatement⟩}
                              '}' [else '{'
                                {⟨staticStatement⟩}
                              '}']
⟨errorStatement⟩         ::=  (error | warning)'('⟨string⟩')'';'
```

DYNAMISCHES VERHALTEN

Das dynamische Verhalten ist konzeptbedingt in der Funktionalität stark eingeschränkt, es umfaßt eine *if*-Anweisung zur Definition von Testbedingungen. Die Einschränkung rührt daher, daß in der Problemstellung statische Eingabeprüfungen gefordert werden und die bereitgestellte Dynamik lediglich implizit enthalten ist.

Weiterhin wird eine *set*-Anweisung zur Verfügung gestellt, die einer Eigenschaft eines anzugebenden Elements einen Wert zuweisen kann.

```
⟨dynamicStatement⟩       ::=  ⟨propertyAssignment⟩
                           |  ⟨dynamicIfStatement⟩
⟨propertyAssignment⟩     ::=  set ⟨property⟩ '=' ⟨dataValue⟩';'
⟨dynamicIfStatement⟩     ::=  if ⟨booleanExpression⟩ '{'
                                {⟨dynamicStatement⟩}
                              '}' [else '{'
                                {⟨dynamicStatement⟩}
                              '}']
```

DATENTYPEN

Die im Programm verwendeten Datentypen sind zum einen Datentypen, die Elementen zugewiesen werden können, d. h. die als GUI-Element fungieren können. Diese Datentypen umfassen einerseits die als primitive Typen bezeichneten Grunddatentypen, können andererseits auch selbst definiert werden. In diesem Fall muß der Name des Datentyps angegeben werden.

Die zweite Datentypart sind die den Eingabedaten zuweisbaren Typen. Sie umfassen die in Java üblichen Datentypen sowie die Möglichkeit, jedem Datentyp einen Formatierungsstring zuzuweisen, der die Ein- und Ausgabe im GUI-Element steuert.

```
⟨dataType⟩              ::=  ⟨primitiveDataType⟩
                          |  ⟨primitiveDataType⟩'('⟨formatString⟩')')
⟨primitiveDataType⟩     ::=  Boolean
                          |  Double
                          |  Date
                          |  Integer
                          |  String
                          |  ⟨identifier⟩
⟨elementType⟩           ::=  ⟨primitiveElementType⟩
                          |  ⟨typeIdentifier⟩
⟨primitiveElementType⟩  ::=  Checkbox
                          |  Combobox
                          |  Label
                          |  Panel
                          |  Radiobutton
                          |  Textfield
                          |  Textarea
```

BOOLESCHE KONSTRUKTE

Neben den in allen Programmiersprachen üblichen booleschen Konstrukten wird in der entwickelten Sprache auch eine Bedingung zugelassen, die nur aus einem Konditionaloperator und dem zugehörigen booleschen Ausdruck besteht. Diese Konstruktion wird im Zusammenhang mit `hasToBe` eingesetzt und erhöht die Les- und Schreibbarkeit des statischen Verhaltens enorm.

Alle booleschen Ausdrücke müssen voll durchgeklammert werden. Dies erhöht wie bei der Vollklammerung der Anweisungsblöcke den Schreibaufwand minimal, führt aber zu verständlichen und unzweideutigen Konstruktionen, bei denen Vorrangregeln unnötig sind.

```
⟨condition⟩             ::=  ⟨relationalOperator⟩ ⟨booleanExpression⟩
⟨booleanExpression⟩     ::=  '('⟨simpleBoolExpression⟩
                              [⟨conditionalOperator⟩ ⟨booleanExpression⟩]')'
                          |  '!'⟨booleanExpression⟩
⟨simpleBoolExpression⟩  ::=  ⟨dataValue⟩ [⟨relationalOperator⟩ ⟨dataValue⟩]
```

DATENWERTE

Als Datenwerte stehen üblicherweise boolesche Werte, Farbwerte, numerische Werte, Zeichenketten (Strings) sowie Werte der Eingabeelemente, die mit Namen angesprochen werden, zur Verfügung. Die Farbwerte müssen dabei in

hexadezimaler Form als RGB-Werte angegeben werden.

Zusätzlich werden Aufzählungen erlaubt, die Werte für Auswahlfelder definieren. Sie werden als Wertegruppe in geschweiften Klammern angegeben, die einzelnen Werte werden durch Kommata voneinander getrennt.

```
⟨dataValue⟩          ::=  ⟨booleanValue⟩
                       |  ⟨colorValue⟩
                       |  ⟨comboValue⟩
                       |  ⟨numericValue⟩
                       |  ⟨stringValue⟩
                       |  ⟨property⟩
⟨booleanValue⟩       ::=  true
                       |  false
⟨colorValue⟩         ::=  '#'⟨rgbColor⟩⟨rgbColor⟩⟨rgbColor⟩
⟨comboValue⟩         ::=  '{'⟨dataValue⟩{',' ⟨dataValue⟩}'}'
⟨numericValue⟩       ::=  [⟨sign⟩]⟨integer⟩
                       |  [⟨sign⟩]⟨floatingValue⟩
                       |  ⟨property⟩
                       |  '('⟨numericValue⟩
                           ⟨arithmeticOperator⟩ ⟨numericValue⟩')'
⟨stringValue⟩        ::=  '"'{⟨character⟩}'"'
                       |  ⟨property⟩
                       |  '('⟨stringValue⟩
                           ⟨stringOperator⟩ ⟨stringValue⟩')'
```

Spezielle Datenkonstrukte

Die Sprache sieht zwei spezielle Datenkonstrukte vor, zum einen Koordinaten, die zur Addition von Elementen zu anderen Elementen benötigt werden. Eine Koordinatenangabe besteht aus zwei positiven ganzen Zahlen. Die erste Zahl gibt die Zeile, die zweite die Spalte an, an der das Element addiert werden soll. Die beiden Zahlen werden durch Komma voneinander getrennt und in runde Klammern gruppiert.

Das zweite Konstrukt ist der sogenannte Formatierungsstring, der die Ein- und Ausgabe von Werten eines GUI-Elements steuert. Er kann die Ziffern **'#'** und **'9'** enthalten, die für optional einzugebende und zwingend einzugebende Ziffern stehen. Die Ziffer **'0'** steht für optional einzugebende Ziffern, deren Nichteingabe zur Ausgabe von Nullen führt. Die Zeichen **'-'**, **'.'** und **':'** werden so ausgegeben, wie sie eingegeben werden, sie dienen zur Formatierung der Ein- bzw. Ausgabe. Die Zeichen **'t'**, **'m'** und **'j'** dienen zur Formatierung von Daten und dienen als Platzhalter für Tag, Monat und Jahr.

```
⟨coordinates⟩        ::=  '('⟨integer⟩',' ⟨integer⟩')'
⟨formatString⟩       ::=  '"'{⟨formatCharacter⟩}'"'
⟨formatCharacter⟩    ::=  0|9|#|-|.|:|t|m|j
```

Einfache Datentypen

Die einfachen Datentypen entsprechen den in den meisten Programmiersprachen üblichen Datentypen.

Lediglich der Datentyp *rgbColor*, der aus zwei aufeinanderfolgenden hexadezimalen Zahlen besteht, wurde neu eingeführt.

Die Sonderzeichen sind nicht näher bestimmt, sie umfassen alle Zeichen, die keine Ziffer und kein Buchstabe sind. Diese Exklusion läßt sich in Automaten gut erkennen, jedoch in EBNF nicht sehr gut beschreiben.

```
⟨rgbColor⟩          ::=  ⟨hexDigit⟩⟨hexDigit⟩
⟨hexDigit⟩          ::=  ⟨digit⟩|a|...|f|A|...|F
⟨sign⟩              ::=  '+'
                      |  '-'
⟨floatingValue⟩     ::=  {⟨integer⟩}'.'{⟨digit⟩}
⟨integer⟩           ::=  ⟨digit⟩{⟨digit⟩}
⟨character⟩         ::=  ⟨empty⟩
                      |  ⟨digit⟩
                      |  ⟨letter⟩
                      |  ⟨specialChar⟩
⟨digit⟩             ::=  0|...|9
⟨letter⟩            ::=  _|a|...|z|A|...|Z
⟨specialChar⟩       ::=  '#'|'{'|...|ß|ä|...
```

BEZEICHNER

Wir unterscheiden drei Bezeichnertypen, normale Bezeichner, Eigenschaftsbezeichner und Typbezeichner.

Normale und Typbezeichner sind gleich aufgebaut und entsprechen den in den meisten Sprachen üblichen Standards, sie müssen aus Buchstaben und Ziffern aufgebaut sein und dürfen nicht mit einer Ziffer beginnen. Das einzige zugelassene Sonderzeichen ist der Unterstrich.

Eigenschaftsbezeichner bestehen aus zwei Bezeichnern, die mit einem Punkt verbunden werden. Der erste Bezeichner steht für das anzusprechende Element, der zweite Bezeichner für die ausgewählte Eigenschaft. Die Notation folgt damit der Schreibweise, die für Eigenschaften auch in Java üblich ist.

```
⟨identifier⟩        ::=  ⟨letter⟩{⟨letter⟩|⟨digit⟩}
⟨property⟩          ::=  ⟨identifier⟩'.'⟨identifier⟩
⟨typeIdentifier⟩    ::=  ⟨letter⟩{⟨letter⟩|⟨digit⟩}
```

OPERATOREN

Die zur Verfügung stehenden Operatoren sind Java entlehnt, für boolesche Verknüpfungen stehen logische *und*- und *oder*-Operatoren bereit. Vergleiche können mit vielfältigen relationalen Operatoren ausgeführt werden, mathematisch stehen die vier Grundrechenarten zur Verfügung, für Strings ist die Konkatenation erlaubt.

```
⟨conditionalOperator⟩   ::=  '||'|'&&'
⟨relationalOperator⟩    ::=  '=='|'<'|'>'|'<='|'>='|'!='
⟨arithmeticOperator⟩    ::=  '+'|'-'|'*'|'/'
⟨stringOperator⟩        ::=  '+'
```

3.7 Transfer der veränderlichen Programmteile

Der Transfer der geänderten Programmteile gliedert sich zum einen in das Problem, welche Daten zum Nutzer übertragen werden, zum anderen in das Problem, wie genau die Übertragung vonstatten gehen soll.

Die veränderlichen Programmteile können als Textdatei(en) übertragen werden, die den Quellcode enthalten. Dies hat den Vorteil, daß die Beschreibung im Klartext vorliegt, d. h. der Quellcode kann ohne Probleme vom Programmrumpf gelesen oder mit einem American Standard Code for Information Interchange (ASCII)-Editor gelesen und verändert werden. Dies ist gleichzeitig ein Nachteil, da die Veränderung durch den Nutzer nicht in jedem Fall sinnvoll ist. Außerdem können die entstehenden Dokumente relativ umfangreich werden und damit langsamer übertragbar sein.

Eine andere Möglichkeit besteht darin, die Quellcodedateien in einer *jar*-Datei zu übertragen. Eine *jar*-Datei ist eine von Java genutzte Möglichkeit, mehrere Dateien in einer größeren Datei zusammenzufassen und bei Bedarf auch zu packen. So bekommt der Nutzer nur eine einzige Datei geliefert. Auch hier können die Dateien geändert werden, sie müssen aber vorher entpackt, dann geändert und danach wieder in die *jar*-Datei gepackt werden, eine unabsichtliche Veränderung ist damit erheblich schwerer geworden.

In unserem Konzept sollen beide Möglichkeiten genutzt werden können, da es dem jeweiligen Programmierer bzw. Nutzer vorbehalten sein soll, diese Entscheidung zu treffen.

Die eigentliche Übertragung wird ebenfalls nicht genau festgelegt, wichtig ist, daß ein Übertragungsweg gewählt wird, der zum einen die sichere Übertragung der Dateien gewährleistet, zum anderen schnell genug für die Anforderungen des Nutzers ist. Dies kann von der ftp-Übertragung bis zur Versendung einer Diskette oder CD-ROM reichen. Die Möglichkeit, nach dem Programmstart die geänderten Dateien vom Programmrumpf übertragen zu lassen, ist natürlich auch erlaubt, allerdings trägt hierfür der Programmrumpf die Verantwortung, das Konzept wird davon nicht berührt.

Natürlich muß sichergestellt werden, daß die übertragenen Daten an der richtigen Stelle im System des Nutzers gespeichert werden. Wie dies zu realisieren ist, ist ebenfalls zwischen Programmierer und Nutzer festzulegen.

3.8 Zusammenfassung

Wir haben jedes Teilproblem des Konzepts betrachtet und eine Lösung gefunden. Die Ergebnisse dieses Prozesses sollen noch einmal kurz zusammengefaßt werden.

Die Konzeption sieht vor, daß das gestellte Problem dadurch gelöst werden kann, daß die veränderlich geplanten Programmteile von den statischen Programmteilen getrennt werden. Die statischen Programmteile werden als Pro-

grammrumpf bezeichnet.

Der Programmrumpf wird auf die übliche Weise hergestellt, d. h. konzeptioniert, programmiert und getestet. Die veränderlichen Programmteile werden in der beschriebenen Programmiersprache implementiert. Die neue Programmiersprache ist dabei streng an den vorgegebenen Anforderungen ausgerichtet worden, sie kann nicht oder nur aufwendig für andere Zwecke eingesetzt werden.

Beide Programmbestandteile werden an den Nutzer übergeben, der sie am Einsatzort abspeichert und den Programmrumpf ausführt. Dieser liest bei Bedarf den veränderlichen Quellcode ein, übersetzt ihn und generiert damit die Benutzeroberfläche sowie die Prüfroutinen.

Falls die veränderlichen Programmteile geändert werden müssen, wird nur für diese der Quellcode neu geschrieben und an den Nutzer ausgeliefert. Dieser ersetzt den alten Quellcode durch den neuen und startet den Programmrumpf neu oder veranlaßt den laufenden Programmrumpf, den neuen Quellcode einzulesen. Der Programmrumpf generiert dann wieder Benutzeroberfläche und Prüfroutinen.

Das Konzept trifft keine Aussagen darüber, wie der Austausch zwischen Programmierer und Nutzer stattzufinden hat, da dieser Austausch den jeweiligen Bedingungen angepaßt sein muß. Es wird lediglich festgelegt, daß der Austausch sicher, vollständig und korrekt erfolgen muß, so daß gewährleistet ist, daß nach der Übertragung die geänderten Daten an der richtigen Stelle im System des Nutzers zu finden sind.

Die geänderten Programmteile werden im Konzept mit festen Schnittstellen versehen, so daß zum einen gegenüber dem Programmrumpf feststeht, wie dieser die generierte Benutzeroberfläche sowie die Prüfroutinen ansprechen muß. Andererseits wird ebenfalls festgelegt, wie sich ein einzelnes GUI-Element verhält, so daß neue GUI-Elemente ohne Probleme verwendet werden können, wenn sie die dafür vorgesehene Schnittstelle implementieren.

4 Bewertung des Konzepts

4.1 Inhalt des Kapitels **69**
4.2 Konzept und Problemstellung **70**
4.3 Bewertung anhand Anforderungen und Kriterien **70**
4.4 Bewertung der Sprache **73**
4.4.1 Nutzung vorhandener Ansätze 74
4.4.2 Bewertung als Programmiersprache 75
4.4.3 Zusammenfassung 77
4.5 Nutzen des Konzepts **78**
4.6 Zusammenfassung **79**

4.1 Inhalt des Kapitels

Dieses Kapitel beschäftigt sich mit der Bewertung des Konzepts unter Berücksichtigung der in Kapitel 1.4 herausgearbeiteten Anforderungen und Kriterien. Dabei soll insbesondere ermittelt werden, inwieweit sich das Konzept zur Lösung der Problemstellung eignet und welchen Nutzen es für potentielle Anwender bringt.

Zunächst wird das Konzept der Aufgabenstellung gegenübergestellt. So wird ein grober Überblick darüber erhalten, ob die Problemstellung unter Verwendung des Konzepts bearbeitet werden kann.

Danach wird das Konzept anhand der Anforderungen und Kriterien untersucht und bewertet, die die Eignung des Konzepts hinsichtlich der Problemstellung aufzeigen.

Weiterhin wird untersucht, ob die Ergebnisse des zweiten Kapitels im vorgestellten Konzept berücksichtigt wurden, das heißt, ob die entwickelte Programmiersprache den Anforderungen an formale und Programmiersprachen genügt.

Anschließend soll untersucht werden, an welchen Stellen des Entwicklungs- und Einsatzzyklus' eines Programms bei Einsatz des vorliegenden Konzepts Nutzen erzielt wird.

Den Abschluß der Bewertung bildet eine kurze Zusammenfassung der erarbeiteten Ergebnisse.

4.2 Konzept und Problemstellung

Die Problemstellung bestand darin, daß ein Programm stark veränderliche Programmteile besitzt. Es handelt sich dabei um Eingabemasken sowie zugehörige Prüfroutinen, die Eingaben in diesen Masken überprüfen. Die Implementierung eines solchen Programms soll so optimiert werden, daß es möglich wird, diese stark veränderlichen Programmteile schnell und unkompliziert bei jeder Änderung an den Nutzer weiterzugeben. Auf Nutzerseite sollen die geänderten Programmteile nach einem erneuten Aufruf des Programms zur Verfügung stehen oder per Knopfdruck integriert werden.

Das daraufhin entwickelte Konzept sieht vor, die angesprochenen veränderlichen Programmteile von den unveränderlichen Programmteilen zu trennen. Damit wird erreicht, daß dem Nutzer ein Programmrumpf zur Verfügung steht, der bei Änderung der veränderlichen Programmteile nicht zusätzlich zu übertragen werden braucht. Es bietet sich dem Nutzer eine Rumpfanwendung, die unabhängig von den veränderlichen Programmteilen weniger oft aktualisiert wird.

Die Abtrennung der veränderlichen Programmteile bewirkt außerdem, daß bei einer Änderung nur noch der veränderte Quellcode übertragen werden muß. Damit werden Zeit, Geld und oft auch Nerven gespart. Außerdem können für die Übertragung der geänderten Teile, aufgrund der drastisch reduzierten Größe der zu übertragenden Daten, einfache Wege wie ftp oder Mail genutzt werden, die bei kompletter Programmübertragung nicht praktikabel wären.

Außerdem bietet die Möglichkeit, daß die veränderlichen Programmteile in der eigens dafür entwickelten Programmiersprache geschrieben werden können, den Vorteil, daß das GUI sowie die Prüfroutinen schneller und weniger fehleranfällig zu programmieren sind.

Es ist festzustellen, daß der Einsatz des entwickelten Konzepts eine Lösung für die Problemstellung darstellt und auf elegante Weise über die Problemstellung hinaus die Fehlerwahrscheinlichkeit innerhalb der veränderlichen Programmteile reduziert.

4.3 Bewertung anhand Anforderungen und Kriterien

Die Anforderungen und Kriterien, die bereits im Kapitel 1.4 aufgestellt wurden, werden nun dafür verwendet, das Konzept zu überprüfen. Dabei werden jede Anforderung und jedes Kriterium einzeln behandelt und die Ergebnisse danach zusammengefaßt.

- Veränderungen durch die Zielsprache Java
  Der Einsatz des Konzepts erfordert, daß als Zielsprache des Programmrumpfs Java eingesetzt wird. Dies ergibt sich daraus, daß die Schnittstellen und Ausnahmen in Java definiert wurden. Die entwickelte Programmiersprache orientiert sich in der Syntax ebenfalls an Java. Es ist zwar theoretisch möglich, den veränderlichen Quellcode innerhalb eines in einer anderen Zielsprache geschriebenen Programmrumpfs einzusetzen, dies verkompliziert die Generierung der veränderlichen Programmteile jedoch außerordentlich.

  Daraus ergibt sich, daß die Anforderungen an Hard- und Software sowie die Einsatzbedingungen des Programms gegenüber dem Einsatz anderer Zielsprachen verändert werden müssen. Außerdem muß der Einsatz des Konzepts in der Analysephase angemessen bei der Konzeption berücksichtigt werden.

  Der Einsatz des Konzepts verändert damit die Anforderungen und die Konzeption des gesamten Projekts. Dies muß sich jedoch nicht zwangsläufig nachteilig auswirken.

- GUI-Definition
  Es wurde herausgearbeitet, daß die Definition der veränderlichen GUI-Teile in einer formalen Sprache zu erfolgen hat.

  Dies ist im Konzept berücksichtigt worden, es wurde eine Programmiersprache angegeben, die dafür verwendet wird, die veränderlichen GUI-Teile zu programmieren.

- Prüfroutinendefinition
  Es wurde ebenfalls gefordert, daß die Prüfroutinen in einer formalen Sprache programmiert werden sollen.

  Auch dies wird mit der angegebenen Programmiersprache erreicht, deren zweiter Teil für die Definition von Prüfroutinen verantwortlich ist.

- Gemeinsame Sprache
  Die Entscheidung für oder gegen getrennte Sprachen ist positiv für eine gemeinsame Sprache getroffen worden.

  Die Anforderung besagte, daß dies eine erlaubte Möglichkeit ist. Die Analyse ergab, daß eine gemeinsame Sprache mit syntaktisch getrennten Teilen für die GUI- und Prüfroutinendefinition die Nachteile getrennter Sprachen vermeidet, ohne deren Vorteile aufzugeben.

- Erweiterte Backus-Naur-Form
  Die Forderung, eine EBNF der entwickelten Programmiersprache anzugeben, konnte erfüllt werden.

  Die EBNF wurde sowohl ausführlich erläutert als auch in kompakter Form im Anhang angegeben.

- Einlesen der veränderlichen Programmteile
  Die Anforderung, die veränderlichen Programmteile nach dem Programmstart oder während des Programmlaufs einlesen zu können wurde dadurch erfüllt, daß zum einen die veränderlichen von den festen Programmteilen getrennt wurden. Damit konnten die veränderlichen Programmteile unabhängig von den festen Programmteilen gehalten werden.

  Zweitens bietet die entwickelte Programmiersprache die Möglichkeit, die veränderlichen Programmteile in dieser Sprache zu programmieren. Der so entstehende Programmcode kann vom Programmrumpf eingelesen und zur Generierung genutzt werden.

  Drittens wurde durch Schnittstellendefinition die Möglichkeit geschaffen, die zu generierenden Programmteile innerhalb fester Schnittstellen zu betrachten und damit unabhängig vom konkreten Inhalt anzusprechen.

  Alle drei Punkte spielen zusammen, um die variablen Programmteile unabhängig halten zu können und so die Anforderung nach späterer Generierung zu erfüllen.

- Schnittstellen
  Es bestand die Forderung, die Schnittstellen festzulegen und dabei spätere Erweiterungen nicht zu verhindern.

  Die Schnittstellen für die GUI-Elemente wurden so festgelegt, daß nur die Manipulationen erlaubt sind, die Eingabeelemente benötigen. Sie umfassen Wert- und Aussehensmanipulationen. Damit können auch komplexere GUI-Elemente eingesetzt werden, sie müssen lediglich die Schnittstellen implementieren.

  Der Datenaustausch wurde über Hashtabellen geregelt, bei denen zu einem eindeutigen Schlüssel Objekte als Werte gespeichert werden. Der Objekttyp der Werte ist nicht festgelegt, so daß alle denkbaren GUI-Elemente ihre Werte übertragen können, da jede Klasse von Java ein Objekt ist.

  Somit erlauben die Schnittstellen Wert- und Aussehenmanipulationen in umfangreichem Maß, da die übertragenen Objekte beliebig komplex sein dürfen, lediglich die angesprochenen GUI-Elemente müssen die Werte verarbeiten können.

  Die Schnittstellen der Prüfroutinen umfassen Methoden zur Überprüfung der Nutzereingaben sowie die Möglichkeit, die Prüfroutinen zu generieren. Damit ist alle Funktionalität berücksichtigt.

- Schnittstellenrelevanz
  Laut Anforderungen sollten die Schnittstellen nur relevante Methoden und Attribute freigeben.

  Die Analyse ergab, daß alle GUI-Elemente Wert- und Aussehenmanipulationen zur Verfügung stellen müssen. Diese Manipulationen wurden bei der Schnittstellendefinition berücksichtigt.

  Nur das oberste GUI-Element besitzt eine weitere Methode, die dazu genutzt wird, das veränderliche GUI zu generieren. Damit wird sicherge-

stellt, daß normale GUI-Elemente diese Möglichkeit nicht besitzen, da diese Methode für sie nicht relevant ist.

Gleiches gilt für die Schnittstellen der Prüfroutinen, die die Überprüfung der Nutzereingaben sowie die Generierung der Prüfroutinen kapseln und damit überflüssige Manipulationen ausschließen.

- GUI-Manipulation
  Die Manipulation von Farbe und Text der GUI-Elemente ist explizit in den entsprechenden Methoden der Schnittstellen berücksichtigt worden.

  Die Forderung, weitere Manipulationen nicht auszuschließen wurde derart berücksichtigt, daß es möglich ist, die Schnittstellen manuell zu erweitern, ohne andere Methoden zu behindern. Es wurde darauf verzichtet, eine generische Lösung für die Manipulation beliebiger Eigenschaften in das Konzept zu übernehmen, da dadurch der enge Bezug zu den geforderten Manipulationsmöglichkeiten verlorengegangen wäre.

  Dies ist sicherlich die weniger universelle Lösung, da zusätzliche Manipulationsmöglichkeiten schwer zu realisieren sind. Es wurde bewußt der enge Problembezug vor der Universalität gewählt. Dabei spielte der Gedanke eine Rolle, daß andere Manipulationsmöglichkeiten eher selten gewünscht werden und die universelle Behandlung dieser Manipulationen zu unverhältnismäßig erhöhtem Aufwand seitens des Programmierers führt.

- GUI-Wertzuweisung
  Wie bereits mehrfach beschrieben, ist die Forderung, daß die GUI-Elemente Wertemanipulationen zur Verfügung stellen sollten, erfüllt worden.

  Der Werttyp ist nicht festgelegt worden, hier wurde die universellere Möglichkeit gewählt, um spätere Erweiterung der Eingabeelemente nicht zu verhindern. Der Einsatz erweiterter Eingabeelemente wurde im Konzept als höchst wahrscheinlich angesehen.

Die Anforderungen und Kriterien, die sich aus der Analyse des Problems ergaben, sind erfüllt worden.

Wie zu sehen ist, wurden die Anforderungen teilweise stärker eingeschränkt oder erweitert, falls dies geschah, ist die Begründung dafür oben angegeben worden.

Die untersuchten Anforderungen stellen den Kern der für die Problemlösung allgemein notwendigen Merkmale dar, die benötigt werden, um das aufgestellte Problem zufriedenstellend zu bearbeiten.

4.4 Bewertung der Sprache

Die Programmiersprache als zentraler Bestandteil des Konzepts muß gesondert bewertet werden.

Zunächst soll dargestellt werden, inwieweit bereits existierende Ansätze und Sprachen verwendet werden konnten. Dann wird interessieren, ob die allgemeinen Grundlagen bei der Definition von Programmiersprachen beachtet wurden und ob eine „schöne“ Programmiersprache geschaffen werden konnte.

Den Abschluß bildet wiederum eine kurze Zusammenfassung der gewonnenen Erkenntnisse.

4.4.1 Nutzung vorhandener Ansätze

Syntaktisch und in der Definition der Token orientiert sich die entwickelte Programmiersprache an Java. Damit gehen die Vorteile einher, daß die syntaktischen Strukturen dem Programmierer vertraut erscheinen. Nachteilig ist, daß in vielen Fällen die Semantik der Programmiersprache nicht mit der von Java übereinstimmt. Diesem Problem wurde versucht zu begegnen, indem nur semantisch ähnliche syntaktische Strukturen übernommen wurden. Semantisch unterschiedliche Strukturen wurden mittels differenter Syntax dargestellt.

Die Trennung von GUI- und Programmquellcode innerhalb einer Quellcodedatei wird bereits von Visual Basic verwendet. Eine solche Trennung kombiniert die Vorteile von Strukturbeschreibungssprachen mit denen von Prozeßbeschreibungssprachen, besitzt jedoch den Nachteil, daß vom Programmierer zwischen Struktur- und Prozeßbeschreibung unterschieden werden muß.

Delphi besitzt von allen untersuchten Sprachen in vielen Bereichen die einfachste und präziseste Syntax. Leider konnte dies innerhalb des Konzepts nur marginal berücksichtigt werden, da aufgrund der Wahl von Java als Zielsprache die Orientierung an den syntaktischen und semantischen Strukturen von Java im Vordergrund stand. Nichtsdestotrotz wurden syntaktische Elemente an den Stellen übernommen, an denen sie geeigneter als die äquivalenten Konstrukte von Java erschienen.

Die Programmierung von Zwischencode, der dem Programmrumpf zur Verfügung gestellt wird, ähnelt dem Konzept von GUI-Generatoren, deren Zwischenspeicherung der GUI-Beschreibung meist ebenfalls in einer eigenen Programmiersprache erfolgt. Hier ist es besonders nachteilig, daß die entsprechende Dokumentation nicht freigegeben wird, da ein Vergleich gerade hier positive und negative Ansätze ausarbeiten könnte.

Die Möglichkeit, externe Dateien für die veränderlichen Programmteile zu verwenden, ähnelt dem Konzept der externen Ressourcendateien von Windows. Der Ansatz wurde im vorliegenden Konzept verwendet, die konkrete Realisierung konnte jedoch mangels geeigneter Dokumentation nicht untersucht werden.

4.4.2 Bewertung als Programmiersprache

Die Definition der Token und der Semantik erfolgte im Konzept informell. Dies entspricht nicht den Anforderungen an formale Sprachen, da diese sowohl lexikalisch als auch semantisch formal definiert sein müssen. Es ist jedoch darauf hinzuweisen, daß beide Beschreibungen formal erfolgen können, dies jedoch nicht als Aufgabe der Diplomarbeit angesehen wurde.

Bei dieser Arbeit stand die syntaktische Definition der Sprache im Vordergrund, die mit Angabe der EBNF erfolgte. Die EBNF wurde innerhalb des Konzepts ausführlich erläutert, wobei die semantischen Zusatzbedingungen im beschreibenden Text definiert wurden.

Damit erfüllt die entwickelte Sprache syntaktisch die Anforderungen an formale Sprachen. Die Semantik muß dafür noch formal beschrieben werden, die benutzte informelle Darstellungsweise reicht für unsere Zwecke jedoch aus.

Es bleibt zu prüfen, ob die entwickelte Sprache die Attribute einer guten Sprache aus Abschnitt 2.4.5 erfüllt. Dazu werden analog zu den Anforderungen und Kriterien die Attribute einzeln untersucht:

- Eignung für die Anwendung
  Die entwickelte Sprache unterteilt den Quellcode in zwei Teile, einen für die GUI-Beschreibung und einen für die Beschreibung des Verhaltens, d. h. der Prüfroutinen. Außerdem werden syntaktische Konstrukte verwendet, die aus dem Anwendungsbereich des jeweiligen Teils entlehnt wurden.

  Damit besitzt die Sprache eine natürliche Eignung für die Lösung des vorgestellten Problems, da sowohl die Struktur des Problems als auch seine Terminologie berücksichtigt wurden.

- Einfachheit
  Die entwickelte Sprache ist syntaktisch an Java angelehnt. Außerdem wird versucht, semantisch zweideutige Situationen durch syntaktische Vorgaben zu beseitigen, ein Beispiel hierfür ist die zwingende Klammerung von Anweisungsblöcken.

  Es werden relativ wenig reservierte Wörter verwendet, diese stammen jedoch aus dem Sprachgebrauch des Anwendungsgebietes.

  Damit ist die Einfachheit in den Fällen gegeben, in denen der Programmierer keine Probleme mit der syntaktischen Struktur von Java besitzt. Diese ist jedoch leicht und schnell erlernbar.

- Einheitlichkeit
  Die Einheitlichkeit wurde versucht dadurch zu erreichen, daß unterschiedlichen semantischen Bedeutungen auch unterschiedliche syntaktische Konstrukte zugeordnet wurden.

  Dies bedeutet, daß syntaktische Strukturen unabhängig vom Kontext die gleiche oder ähnliche Bedeutung besitzen, so daß Programmierfehler aufgrund uneinheitlicher syntaktischer Verwendung von Anweisungen minimiert werden.

- Klarheit
  Die starke Orientierung der entwickelten Programmiersprache an der vorgegebenen Problemstellung sowie die oben bereits angesprochene Entlehnung der syntaktischen Konstrukte aus der Terminologie der Problemstellung führen dazu, daß Algorithmen und Beschreibungen sehr leicht aus der abstrakten Problembeschreibung heraus implementiert werden können.

- Leichte Programmverifikation
  Die bisher angesprochenen Faktoren bedingen, daß ein Programm, das in der entwickelten Programmiersprache geschrieben wurde, einfach und klar die Problemstellung wiedergibt. Damit ist die Programmverifikation ebenfalls einfach und klar. Durch die strikt geforderte Klammerung von booleschen Ausdrücken und Anweisungsblöcken wird die Programmverifikation nochmals vereinfacht.

- Lesbarkeit
  Wiederum mit dem Verweis auf die bisher angegebenen Punkte kann davon ausgegangen werden, daß die Lesbarkeit eines in der Sprache geschriebenen Programms sehr hoch ist. Je nach Komplexität und Umfang des geforderten Programms kann die Lesbarkeit durch den Umfang des Quellcodes leiden, dieses Problem läßt sich jedoch leider nicht vermeiden.

- Nutzungskosten
  Die Verringerung der Nutzungskosten ist nicht einfach zu bewerten, da je nach Qualifikation und Vertrautheit des Programmierers mit dem Konzept und der entwickelten Sprache die Zeiten und damit die Kosten stark variieren können.

  Bei starker Nutzung des Konzepts kann davon ausgegangen werden, daß zumindestens die Kosten der Programmpflege reduziert werden, da genau das Problem der Programmpflege mit dem Einsatz des Konzepts vereinfacht und damit kostengünstiger gestaltet werden sollte.

- Orthogonalität
  Die Orthogonalität der Sprache ist relativ niedrig, da Einheitlichkeit und Einfachheit der Sprache im Vordergrund standen. Die Verwendung der gleichen Programmkonstrukte in unterschiedlichen Kontexten verringert die Einheitlichkeit und Einfachheit in dem Sinne, daß die semantische Bedeutung der Konstrukte je nach Kontext sehr stark variieren kann.

- Portabilität von Programmen
  Da Quellcode in der entwickelten Programmiersprache lediglich innerhalb des Programmrumpfs übersetzt wird, kann er auf alle Systeme portiert werden, auf denen der Programmrumpf läuft. Damit ist die entwickelte Programmiersprache unabhängig von speziellen Betriebssystemen.

- Programmierumgebung
  Das Konzept der entwickelten Sprache läßt die Programmierung von integrierten Programmierumgebungen zu, diese existieren zur Zeit jedoch

noch nicht. Die Entwicklung von Programmierumgebungen wird von der Sprache nicht behindert. Die zunehmende Verbreitung der Sprache könnte durch den Einsatz von Programmierumgebungen gesteigert werden.

- Schreibbarkeit
  Für Programmierer, die mit Java vertraut sind, ist ein Programm in der Programmiersprache leicht zu schreiben. Auch wenig geübte Programmierer dürften keine Probleme mit der Einarbeitung in die Sprache und ihrer Anwendung besitzen. Die vollständige Klammerung von booleschen Ausdrücken und Anweisungsblöcken verlangsamt etwas das Schreiben eines Programms, birgt jedoch bei Test und Fehlersuche so viele Vorteile, daß die allgemeine Schreibbarkeit nicht beeinträchtigt wird.

- Unterstützung für Abstraktion
  Die Abstraktion von Daten- oder Programmstrukturen in eigenen Konstrukten wird in der Programmiersprache wenig unterstützt, eigene Prozeduren oder Funktionen können nicht definiert werden. Lediglich GUI-Elemente können logisch und physisch gruppiert und so komplexere Strukturen geschaffen werden. Diese gelten jedoch nur für den aktuellen Quellcode und können, außer durch Kopieren und Einfügen, nicht weiterverwendet werden.

  Der Grund für diese Entscheidung liegt darin, daß bei der angestrebten Prüffunktionalität kaum größere Komplexität zu erwarten ist und GUI-Elemente eher gruppiert als neu erschaffen werden sollten. Diese Neuerschaffung sollte außerhalb des Quellcodes durch Programmierung einer neuen GUI-Klasse stattfinden. Damit kann die Komplexität und daraus resultierend die Fehleranfälligkeit der Sprache drastisch reduziert werden.

Die meisten der angeprochenen Punkte können selbst überprüft werden, indem das Beispielprogramm im Anhang A.2 betrachtet wird.

4.4.3 Zusammenfassung

Die entwickelte Programmiersprache erfüllt teilweise die Bedingungen, die an eine formale Sprache gestellt werden. Die lexikalischen Einheiten sowie die Semantik der Sprache wurden nicht formal definiert. Dies reicht für den gewählten Rahmen aus, da die formale Darstellung der Token und der Semantik möglich und im Rahmen der Diplomarbeit nicht notwendig ist.

Die entwickelte Programmiersprache erfüllt auch die Kriterien für gute Programmiersprachen, die zu einem großen Teil über den Erfolg und den Fortbestand von Programmiersprachen entscheiden.

Die Programmiersprache ist damit sowohl formal als auch inhaltlich den gestellten Anforderungen gewachsen und kann zur Problemlösung eingesetzt werden.

4.5 Nutzen des Konzepts

Der Nutzen des Konzepts zeigt sich in mehreren Stufen der Softwareentwicklung. Voraussetzung dafür, daß der Einsatz des Konzepts überhaupt einen Nutzen zeigt, ist, daß die das Konzept einsetzenden Personen damit vertraut sind, d. h. es tritt keine Verzögerung aufgrund von Lernprozessen ein. Diese Annahme erscheint plausibel, da auch normale Lösungsansätze unter diesen Annahmen bewertet werden.

Die Phasen Systemanforderung sowie Softwareanforderung werden hauptsächlich durch die Wahl der Zielsprache, in unserem Fall Java, beeinflußt. Wichtig ist, daß sowohl die System- als auch die Softwareanforderungen auf den Einsatz der Zielsprache und im allgemeinen auf den Einsatz des Konzepts abgestimmt werden. Der Nutzen der gewählten Zielsprache zeigt sich bei den System- und Softwareanforderungen vor allem darin, daß die Plattformunabhängigkeit von Java genutzt werden kann. Im allgemeinen bedeutet dies, daß das entwickelte Programm eine größere Verbreitung finden kann.

Der Einsatz des Konzepts in der Analysephase bedeutet, schon bei der Grobmodellierung die Trennung der Anwendungsteile zu berücksichtigen. Außerdem muß der Nutzer mit den veränderten Auslieferungsmechanismen vertraut gemacht werden. Die Planung kann durch den Einsatz des Konzepts erleichtert werden, da Designentscheidungen bereits innerhalb des Konzepts getroffen werden.

Die Entwurfsphase muß den Einsatz des Konzepts stark berücksichtigen, da die Modellierungswerkzeuge gegenüber herkömmlichem Vorgehen geändert werden können. Außerdem müssen alle Modelle das Konzept einbeziehen sowie die betroffenen Programmteile bereits getrennt geplant werden. Hierbei reduziert sich normalerweise der Aufwand, da auf bereits getroffene Annahmen des Konzepts zurückgegriffen werden kann und GUI-Modelle sowie Prüfroutinen in einfacherer Form beschrieben werden können. Es ergibt sich positiver zeitlicher wie modellorganisatorischer Nutzen. Die Trennung der variablen von den festen Programmteilen erfordert auf der anderen Seite erhöhten Modellierungsaufwand, da die getrennten Teile zu identifizieren sowie extra zu modellieren sind. Insgesamt dürfte der Nutzen des Einsatzes den erhöhten Aufwand überwiegen.

Die Programmierungsphase ist ebenfalls stark vom Einsatz des Konzepts betroffen, hier müssen die einzelnen Programmteile unabhängig voneinander implementiert werden. Allerdings läßt sich der vom Programmrumpf getrennte variable Code leichter programmieren als vergleichbarer herkömmlicher Code, da die eingesetzte Programmiersprache extra für diesen Zweck entworfen wurde. Die Entwicklung des Programmrumpfs wird ebenfalls erleichtert, da nur zwei feste Schnittstellen der variablen Programmteile zu betrachten sind. Dies bedeutet eine erhebliche Komplexitätseinsparung. Die fest definierten Schnittstellen dienen außerdem dazu, die Fehlermöglichkeiten des entstehenden Programmcodes zu veringern.

Für die Programmierer ergeben sich weiterhin Vorteile bei der Versionsverwaltung des entstehenden Codes. Die stark veränderlichen Programmteile können unabhängig vom Programmrumpf versioniert werden. Damit kann der Programmrumpf unabhängig von den veränderlichen Programmteilen weiterentwickelt werden, ohne daß Probleme bei der Auslieferung der veränderlichen Teile entstehen.

In dieser Phase kommen die Vorteile des Einsatzes des Konzepts erstmalig voll zum Tragen.

Die Testphase wird mit dem Einsatz des Konzepts nicht erheblich verändert. Die vorgesehenen Tests müssen auch bei Trennung der Programmteile durchgeführt werden, der zusätzliche Testaufwand des variablen Programmcodes ist unerheblich. Diese Phase profitiert hauptsächlich von der Standardisierung der Schnittstellen und der damit verbundenen Erleichterung bei Systemtests.

Die Betriebsphase ist für den Nutzer leichter geworden, da bei Änderung der variablen Programmteile diese dynamisch eingebunden werden und damit nicht das gesamte Programm ausgetauscht werden muß. Der Nutzer kann somit von aktuellsten Programmdaten sowie einfacher Integration neuer Daten ausgehen. Für den Programmierer ergeben sich ebenfalls Vorteile, da die oft zu ändernden Programmteile einfach, schnell und problemlos angepaßt und ausgeliefert werden können.

4.6 Zusammenfassung

Die Bewertung des Konzepts hat ergeben, daß das Konzept die angegebene Problemstellung zu lösen vermag.

Dabei wurde ein Konzept entwickelt, das eine Problemlösung präsentiert, die durch die Trennung der festen von den variablen Programmteilen bisher übliche Programmstrukturen durchbricht. Dieser Bruch wird mit erhöhtem Nutzen nicht nur für den Programmierer sondern auch für den Nutzer gerechtfertigt.

Die Trennung wird anhand fester Schnittstellen gekapselt, die variablen Programmteile sind jedoch nicht in der Entwicklung eingeengt, sondern für neue Teilelemente offen. Der zu programmierende Zwischencode der variablen Programmteile verringert die Fehlerwahrscheinlichkeit und erhöht die Realisierungsgeschwindigkeit, da die entwickelte Programmiersprache problemnah entworfen wurde und damit bestens zur Lösung der vorgegebenen Problemstruktur geeignet ist.

Die Sprache genügt den Anforderungen an formale Sprachen und an allgemeine Programmiersprachen, lediglich die noch fehlende Werkzeugunterstützung könnte einer weiteren Verbreitung entgegenstehen.

Insgesamt ist das Konzept positiv zu bewerten, da ein Einsatz in der Regel Vorteile für Programmierer und Nutzer bringt, die Fehlerwahrscheinlichkeit verringert und den Austausch geänderter Programmteile erleichtert.

5 Zusammenfassung und Fazit

Diese Diplomarbeit hat versucht, anhand einer konkreten Aufgabenstellung ein Konzept zu entwickeln, das das in der Aufgabenstellung aufgeworfene Problem einfach und elegant zu lösen vermag.

5.1 Zusammenfassung

Die Darstellung des Problems erfolgte im ersten Kapitel. Eine anschließende Untersuchung versuchte, das Problem aufzugliedern und die einzelnen Teilprobleme zu analysieren. Es wurden die Bedeutung der Zielsprache herausgearbeitet und der Herstellungsprozeß von Software untersucht. Weiterhin wurden die Verwendung formaler Sprachen und die Bedeutung von Schnittstellen diskutiert, um anschließend eine Einordnung in den Softwareentwicklungsprozeß vornehmen zu können. Die Analyse ergab eine Liste von Anforderungen und Kriterien, anhand derer das entwickelte Konzept mit der Problemstellung verglichen werden sollte.

Der Analyse folgte die Untersuchung existierender Programmiersprachen und Konzepte hinsichtlich geeigneter Lösungen und Konstrukte, die in die Problembearbeitung übernommen werden konnten. Nach dieser Untersuchung wurden die Grundlagen von Programmiersprachen dargestellt. Dies beinhaltete eine Einführung in die Theorie formaler Sprachen sowie die Untersuchung existierender Programmiersprachen mit dem Ziel, Designkriterien für gute Programmiersprachen zu finden.

Damit waren die Voraussetzungen geschaffen, um das Konzept zu entwickeln. Dazu wurden die Zielsprache festgelegt und der Implementationsprozeß untersucht. Diese Untersuchung führte zur Trennung der veränderlichen von den unveränderlichen Programmteilen. Um diese Trennung zu unterstützen, wurden nun die notwendigen Schnittstellen definiert und benötigte Ausnahmebedingungen festgelegt.

Dann konnte dazu übergegangen werden, eine neue Sprache für die variablen Programmteile zu entwickeln. Es wurde eine gemeinsame Sprache für die Oberflächenelemente und die Prüfroutinen definiert und deren Syntax und Semantik angegeben.

Abschließend wurde noch untersucht, wie der Transfer des Programmcodes

bzw. der Programmteile vonstatten zu gehen hat. Damit war das Konzept fertig definiert und konnte mit den Anforderungen und Kriterien des ersten Kapitels verglichen werden.

Dieser Vergleich fand im vierten Kapitel statt. Zunächst erfolgte eine Gegenüberstellung von Konzept und Problemstellung, anschließend konnte mit einer Bewertung des Konzepts anhand der Anforderungen und Kriterien fortgefahren werden. Darauf folgte die Bewertung der entwickelten Sprache als Programmiersprache. Den Abschluß bildete eine Untersuchung des Nutzens, der beim Einsatz des Konzepts entsteht.

5.2 Fazit

Im Ergebnis der gesamten Arbeit konnte ein Konzept entwickelt werden, dessen Grundlage die Trennung eines Programms in veränderliche und unveränderliche Programmteile ist. Diese Programmteile sollten in jeweils verschiedenen Sprachen implementiert werden. Somit wird die Zielsprache, in unserem Fall Java, für die Programmierung des unveränderlichen Programmrumpfs benutzt. Die zweite, neu definierte Sprache dient dazu, die veränderlichen Programmteile zu implementieren.

Die Trennung der Programmteile bei der Implementierung ist keine grundlegend neue Idee. Das Konzept geht jedoch noch einen Schritt weiter und behält die Trennung bis zur Betriebsphase des Programms bei. Damit bekommt der Nutzer zwei getrennte Programmteile geliefert, von denen einer, der veränderliche, jederzeit ausgetauscht werden kann. Das bedeutet, daß mit dem Einsatz des Konzepts der Herstellungsprozeß von Software bis hin zur Betriebsphase umgestellt wird.

Damit ist es möglich, die jeweiligen Programmteile von den dafür am besten geeigneten Softwareentwicklern bearbeiten zu lassen. Dies führt zu Geschwindigkeitsvorteilen bei der Implementierung und zu stabileren Programmen. Außerdem brauchen die jeweiligen Programmierer nur gegen fest definierte Schnittstellen zu programmieren, dies hilft ebenfalls, fehlerreduzierten Code auszuliefern. Weiterhin wird die Versionsverwaltung des programmierten Codes erleichtert, da die stark veränderlichen Programmteile unabhängig vom Programmrumpf versioniert werden können.

Der Nachteil der Trennung besteht darin, daß die Programmierer gegeneinander arbeiten könnten, wenn die Spezifikation nicht genau genug erfolgt. Es wird also eine möglichst fehlervermeidende Synchronisation der Programmteile benötigt. Diese Synchronisation zwischen den einzelnen Programmteilen wird mittels der erwähnten Schnittstellen vorgenommen, die alle relevanten Methoden und Attribute kapseln, die beim Einsatz des Konzepts vorkommen können.

Das entwickelte Konzept sieht vor, die am häufigsten geänderten Programmteile auszulagern, damit die Stabilität der statischen Programmteile zu wahren

und gleichzeitig die häufigen Änderungen einfach, schnell und sicher zu gestalten.

Die Änderung des Herstellungsprozesses betrifft, wie erwähnt, auch die Nutzer, da diese nur die aktuellen Programmteile einfach und schnell geliefert bekommen. Zur Integration dieser Teilstücke müssen sie nur das alte Programm neu starten oder einen entsprechenden Menüpunkt aufrufen.

Das Einsatzgebiet des vorliegenden Konzepts ist relativ stark eingeengt, da die vorgegebene Situation mit Eingabemasken und darauf operierenden Prüfroutinen nicht sehr häufig auftritt. Die zugrundeliegende Idee ist aber von hohem Nutzen, da Programme sehr oft Teile beinhalten, die starken Veränderungen ausgesetzt sind. Nimmt man nun das vorliegende Konzept, entwickelt eine eigene Programmiersprache und definiert die entsprechenden Schnittstellen, so können die in der Arbeit gewonnenen Ergebnisse nutzbringend verwendet werden.

Das Konzept zeigt außerdem, daß sich der Einsatz kleiner und spezialisierter Programmiersprachen gegenüber universellen Programmiersprachen auch dann lohnen kann, wenn keine umfangreichen Programme erstellt werden müssen. Die neue Programmiersprache birgt auch bei geringem Einsatzfeld große Vorteile gegenüber universellen Lösungen.

A Erweiterte Backus-Naur-Form

A.1 Entwickelte Programmiersprache

Die im Abschnitt 3.6.2 festgelegten Konventionen gelten auch in diesem Kapitel.

Zur leichteren Lesbarkeit ist die vorliegende EBNF in größere Abschnitte analog denen der ausführlichen Beschreibung von Abschnitt 3.6.3 unterteilt. Die Abschnittsbezeichnung erfolgt in Kapitälchen und ist kein formaler Bestandteil der EBNF.

PROGRAMM

```
⟨program⟩                ::=  [define view as '{'
                               {⟨guiStatement⟩}
                              '}']
                              [define static behaviour as '{'
                               {⟨staticStatement⟩}
                              '}']
                              [define dynamic behaviour as '{'
                               {⟨dynamicStatement⟩}
                              '}']
```

GUI-DEFINITIONEN

```
⟨guiStatement⟩           ::=  create [main] ⟨identifier⟩ as ⟨elementType⟩ '{'
                               {⟨elementDefinition⟩}
                              '}'
                           |  group ⟨elementGroup⟩ for '{'
                               {⟨elementManipulation⟩}
                              '}'
⟨elementDefinition⟩      ::=  set data '=' ⟨dataValue⟩';'
                           |  format data as ⟨dataType⟩';'
                           |  ⟨valueAssignment⟩
                           |  add (⟨identifier⟩ | ⟨elementType⟩)
                               [with '{'⟨elementDefinition⟩'}']
                               at position⟨coordinates⟩';'
                           |  connect to ⟨identifier⟩';'
⟨elementGroup⟩           ::=  '{'⟨identifier⟩{, ⟨identifier⟩}'}'
⟨elementManipulation⟩    ::=  ⟨valueAssignment⟩
⟨valueAssignment⟩        ::=  set ⟨identifier⟩ '=' ⟨dataValue⟩';'
```

STATISCHES VERHALTEN

```
⟨staticStatement⟩        ::=  ⟨requiredAssignment⟩
                           |  ⟨conditionCheck⟩
```

```
                          |  ⟨staticIfStatement⟩
                          |  ⟨errorStatement⟩
⟨requiredAssignment⟩    ::=  set ⟨identifier⟩ to required['('⟨integer⟩')']
                              otherwise '{'
                                {⟨staticStatement⟩}
                             '}'
⟨conditionCheck⟩        ::=  ⟨identifier⟩ hasToBe ⟨condition⟩
                              otherwise '{'
                                {⟨staticStatement⟩}
                             '}'
⟨staticIfStatement⟩     ::=  if ⟨booleanExpression⟩ '{'
                              {⟨staticStatement⟩}
                             '}' [else '{'
                              {⟨staticStatement⟩}
                             '}']
⟨errorStatement⟩        ::=  (error | warning)'('⟨string⟩')'';'
```

DYNAMISCHES VERHALTEN

```
⟨dynamicStatement⟩      ::=  ⟨propertyAssignment⟩
                          |  ⟨dynamicIfStatement⟩
⟨propertyAssignment⟩    ::=  set ⟨property⟩ '=' ⟨dataValue⟩';'
⟨dynamicIfStatement⟩    ::=  if ⟨booleanExpression⟩ '{'
                              {⟨dynamicStatement⟩}
                             '}' [else '{'
                              {⟨dynamicStatement⟩}
                             '}']
```

DATENTYPEN

```
⟨dataType⟩              ::=  ⟨primitiveDataType⟩
                          |  ⟨primitiveDataType⟩'('⟨formatString⟩')')
⟨primitiveDataType⟩     ::=  Boolean
                          |  Double
                          |  Date
                          |  Integer
                          |  String
⟨elementType⟩           ::=  ⟨primitiveElementType⟩
                          |  ⟨typeIdentifier⟩
⟨primitiveElementType⟩  ::=  Checkbox
                          |  Combobox
                          |  Label
                          |  Panel
                          |  Radiobutton
                          |  Textfield
                          |  Textarea
```

BOOLESCHE KONSTRUKTE

```
⟨condition⟩             ::=  ⟨relationalOperator⟩ ⟨booleanExpression⟩
⟨booleanExpression⟩     ::=  '('⟨simpleBoolExpression⟩
                              [⟨conditionalOperator⟩ ⟨booleanExpression⟩]')'
                          |  '!'⟨booleanExpression⟩
⟨simpleBoolExpression⟩  ::=  ⟨dataValue⟩ [⟨relationalOperator⟩ ⟨dataValue⟩]
```

DATENWERTE

```
⟨dataValue⟩             ::=  ⟨booleanValue⟩
                          |  ⟨colorValue⟩
```

```
                          |  ⟨comboValue⟩
                          |  ⟨numericValue⟩
                          |  ⟨stringValue⟩
                          |  ⟨property⟩
⟨booleanValue⟩          ::=  true
                          |  false
⟨colorValue⟩            ::=  '#'⟨rgbColor⟩⟨rgbColor⟩⟨rgbColor⟩
⟨comboValue⟩            ::=  '{'⟨dataValue⟩{',' ⟨dataValue⟩}'}'
⟨numericValue⟩          ::=  [⟨sign⟩]⟨integer⟩
                          |  [⟨sign⟩]⟨floatingValue⟩
                          |  ⟨property⟩
                          |  '('⟨numericValue⟩
                               ⟨arithmeticOperator⟩ ⟨numericValue⟩')'
⟨stringValue⟩           ::=  '"'{⟨character⟩}'"'
                          |  ⟨property⟩
                          |  '('⟨stringValue⟩
                               ⟨stringOperator⟩ ⟨stringValue⟩')'
```

Spezielle Datenkonstrukte

```
⟨coordinates⟩           ::=  '('⟨integer⟩',' ⟨integer⟩')'
⟨formatString⟩          ::=  '"'{⟨formatCharacter⟩}'"'
⟨formatCharacter⟩       ::=  0|9|#|-|.|:|t|m|j
```

Einfache Datentypen

```
⟨rgbColor⟩              ::=  ⟨hexDigit⟩⟨hexDigit⟩
⟨hexDigit⟩              ::=  ⟨digit⟩|a|...|f|A|...|F
⟨sign⟩                  ::=  '+'
                          |  '-'
⟨floatingValue⟩         ::=  {⟨integer⟩}'.'{⟨digit⟩}
⟨integer⟩               ::=  ⟨digit⟩{⟨digit⟩}
⟨character⟩             ::=  ⟨empty⟩
                          |  ⟨digit⟩
                          |  ⟨letter⟩
                          |  ⟨specialChar⟩
⟨digit⟩                 ::=  0|...|9
⟨letter⟩                ::=  _|a|...|z|A|...|Z
⟨specialChar⟩           ::=  '#'|'{'|...|ß|ä|...
```

Bezeichner

```
⟨identifier⟩            ::=  ⟨letter⟩{⟨letter⟩|⟨digit⟩}
⟨property⟩              ::=  ⟨identifier⟩'.'⟨identifier⟩
⟨typeIdentifier⟩        ::=  ⟨letter⟩{⟨letter⟩|⟨digit⟩}
```

Operatoren

```
⟨conditionalOperator⟩   ::=  '||'|'&&'
⟨relationalOperator⟩    ::=  '=='|'<'|'>'|'<='|'>='|'!='
⟨arithmeticOperator⟩    ::=  '+'|'-'|'*'|'/'
⟨stringOperator⟩        ::=  '+'
```

A.2 Eine kleines Beispielprogramm

Eine Beispieloberfläche soll mittels unserer Sprache dargestellt werden. Sie ist in Abbildung A.1 zu sehen.

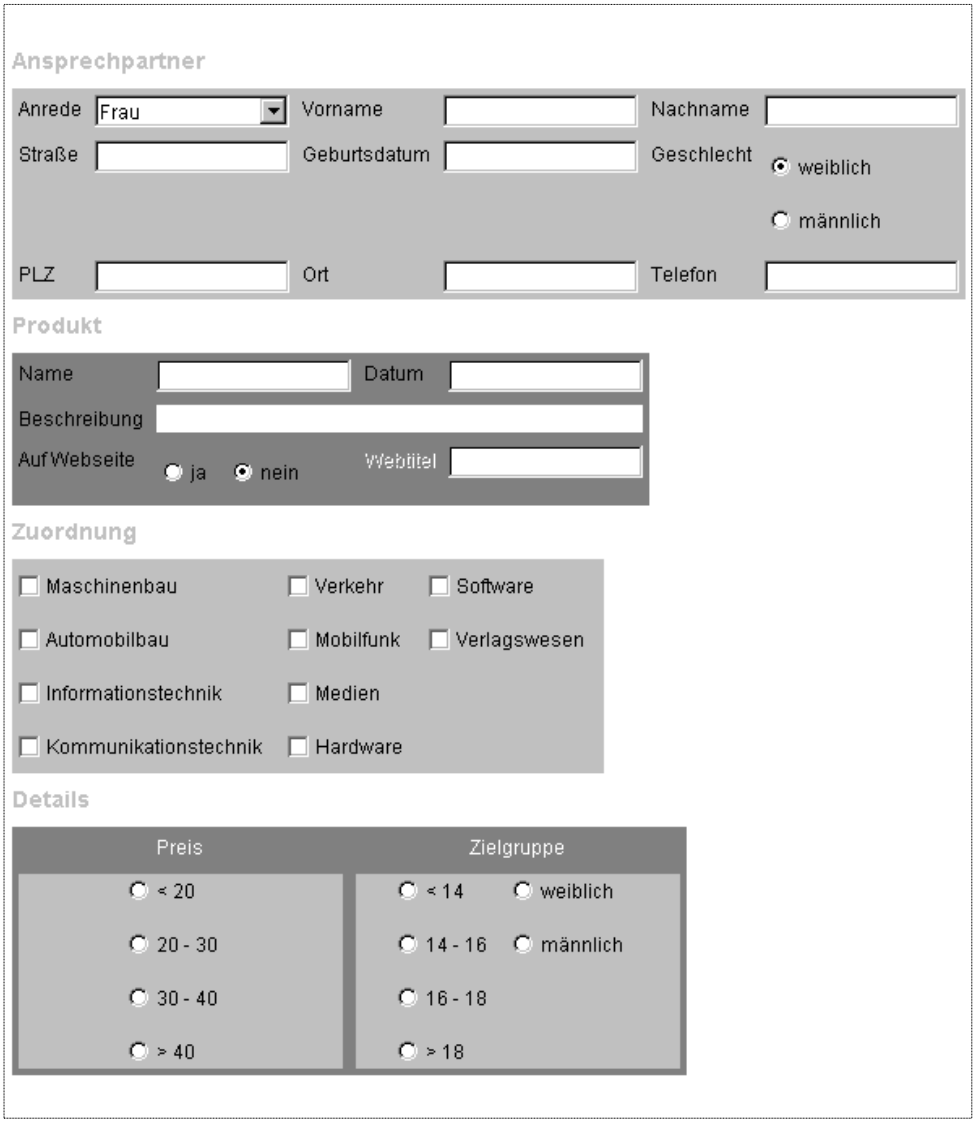

Abbildung A.1: Im Beispiel dargestellte Oberfläche

Quelle: Eigene Darstellung

Außerdem sollen folgende Prüfbedingungen realisiert werden:

STATISCHE PRÜFBEDINGUNGEN

- Ist das Geschlecht weiblich, so muß „Frau“ als Anrede gewählt sein, bei männlichem Geschlecht „Herr“.

- Der Vorname, der Nachname, der Produktname, die Produktbeschreibung, das Produktdatum, der Preis sowie die Zielgruppe sind Felder, die ausgefüllt werden müssen.
- Die Angabe einer Telefonnummer ist optional, fehlt sie, soll jedoch eine Warnung ausgegeben werden.
- Die Zeitangaben dürfen nicht in der Zukunft liegen, das Produktdatum muß zwischen Geburts- und aktuellem Datum liegen.
- Es müssen mindestens drei Zuordnungen angeklickt werden.

DYNAMISCHE PRÜFBEDINGUNGEN

- Wird angeklickt, daß ein Produkt auf einer Webseite präsentiert wird, so soll das zugehörige Webtitel-Eingabefeld freigegeben und mit dem Produktnamen gefüllt werden. Wird „nein“ angeklickt, so soll das Webtitel-Eingabefeld deaktiviert und geleert werden.

Dieses Beispiel soll die Anwendung der Sprache zeigen und kurz demonstrieren, inwiefern Lesbarkeit und Verständlichkeit der Sprache gegeben sind.

Den Bezeichnern für die einzelnen Elemente sind dreibuchstabige Präfixe vorangesetzt. Dies ist nicht zwingend erforderlich, sondern dient der besseren Klarheit des Quelltextes in Bezug auf die verwendeten Typen und ist eine Namenskonvention des Autors. Bei konsequentem Einsatz dieser Konvention erhöht sich die Lesbarkeit durch Erkennung der verwendeten Datentypen stark.

```
// Beispiel für eine Oberfläche in der Konzeptsprache
// Autor: Ekkart Kleinod
// ***************************************************
// Aussehen
define view as {
  // ***********************************
  // Überschriften
  create lblHeader1 as Label {
    set data = "Ansprechpartner";
  }
  create lblHeader2 as Label {
    set data = "Produkt";
  }
  create lblHeader3 as Label {
    set data = "Zuordnung";
  }
  create lblHeader4 as Label {
    set data = "Details";
  }
  // Überschriften gruppieren und formatieren
  group {lblHeader1,
         lblHeader2,
         lblHeader3,
```

```
        lblHeader4} for {
  set foreground = #CCCCCC;
  set fontsize = 14;
}
// Geschlechtswähler
create pnlGeschlecht as Panel {
  add Radiobutton with {
    set data = "weiblich";
    set selected = true;
    set group = grpGeschlecht;
  } at position(0, 0);
  add Radiobutton with {
    set data = "männlich";
    set group = grpGeschlecht;
  } at position(1, 0);
}
// ***********************************
// Ansprechpartner
// Beschriftungen
create lblAnrede as Label {
  set data = "Anrede";
}
create lblVorname as Label {
  set data = "Vorname";
}
create lblNachname as Label {
  set data = "Nachname";
}
create lblStrasse as Label {
  set data = "Straße";
}
create lblGeburtsdatum as Label {
  set data = "Geburtsdatum";
}
create lblGeschlecht as Label {
  set data = "Geschlecht";
}
create lblPLZ as Label {
  set data = "";
}
create lblOrt as Label {
  set data = "";
}
create lblTelefon as Label {
  set data = "";
}
// Textfelder
create txtVorname as Textfield {
  connect to lblVorname;
}
create txtNachname as Textfield {
  connect to lblNachname;
}
create txtStrasse as Textfield {
```

```
    connect to lblStrasse;
  }
  create txtGeburtsdatum as Textfield {
    set errorcolor = #FFFF00;
    format data as Date("tt.mm.jjjj");
    connect to lblGeburtsdatum;
  }
  create txtPLZ as Textfield {
    format data as Integer("00000");
    connect to lblPLZ;
  }
  create txtOrt as Textfield {
    connect to lblOrt;
  }
  create txtTelefon as Textfield {
    format data as String("0000-0000000");
    connect to lblTelefon;
  }
  // Combobox
  create cboAnrede as Combobox {
    format data as String;
    set data = {"Frau", "Herr"};
    connect to lblAnrede;
  }
  // Geschlechtswähler
  create pnlAGeschlecht as pnlGeschlecht {
    connect to lblGeschlecht;
  }
  // Ansprechpartnerpanel
  create pnlAnsprechpartner as Panel {
    set background = #CCCCCC;
    add lblAnrede at position(0, 0);
    add lblStrasse at position(1, 0);
    add lblPLZ at position(2, 0);
    add lblVorname at position(0, 2);
    add lblGeburtsdatum at position(1, 2);
    add lblOrt at position(2, 2);
    add lblNachname at position(0, 4);
    add lblGeschlecht at position(1, 4);
    add lblTelefon at position(2, 4);
    add cboAnrede at position(0, 1);
    add txtStrasse at position(1, 1);
    add txtPLZ at position(2, 1);
    add txtVorname at position(0, 3);
    add txtGeburtsdatum at position(1, 3);
    add txtOrt at position(2, 3);
    add txtNachname at position(0, 5);
    add pnlAGeschlecht at position(1, 5);
    add txtTelefon at position(2, 5);
  }
  // ***********************************
  // Produkt
  // Beschriftungen
  create lblPName as Label {
    set data = "Name";
```

```
}
create lblPDatum as Label {
  set data = "Datum";
}
create lblPBeschreibung as Label {
  set data = "Beschreibung";
}
create lblPWebseite as Label {
  set data = "Auf Webseite";
}
create lblPWebtitel as Label {
  set data = "Webtitel";
}
// Textfelder
create txtPName as Textfield {
  connect to lblPName;
}
create txtPDatum as Textfield {
  set errorcolor = #FFFF00;
  format data as Date("tt.mm.jjjj");
  connect to lblPDatum;
}
create txtPBeschreibung as Textarea {
  connect to lblPBeschreibung;
}
create txtPWebtitel as Textfield {
  set enabled = false;
  connect to lblPWebtitel;
}
// Ja/Nein Wähler
create radJa as Radiobutton {
  set data = "ja";
  set group = grpJaNein;
}
create radNein as Radiobutton {
  set data = "nein";
  set selected = true;
  set group = grpJaNein;
}
create pnlJaNein as Panel {
  add radJa at position(0, 0);
  add radNein at position(0, 1);
}
// Produktpanel
create pnlProdukt as Panel {
  set background = #808080;
  add lblPName at position(0, 0);
  add lblPDatum at position(0, 2);
  add lblPBeschreibung at position(1, 0);
  add lblPWebseite at position(2, 0);
  add lblPWebtitel at position(2, 2);
  add txtPName at position(0, 1);
  add txtPDatum at position(0, 3);
  add txtPBeschreibung at position(1, 1..3);
```

```
    add pnlJaNein with {connect to lblPWebseite;} at position(2, 1);
    add txtPWebtitel at position(2, 3);
  }
  // ***********************************
  // Zuordnung
  create pnlAnsprechpartner as Panel {
    set background = #CCCCCC;
    add Checkbox with {
      set data = "Maschinenbau";
    } at position(0, 0);
    add Checkbox with {
      set data = "Automobilbau";
    } at position(1, 0);
    add Checkbox with {
      set data = "Informationstechnik";
    } at position(2, 0);
    add Checkbox with {
      set data = "Kommunikationstechnik";
    } at position(3, 0);
    add Checkbox with {
      set data = "Verkehr";
    } at position(0, 1);
    add Checkbox with {
      set data = "Mobilfunk";
    } at position(1, 1);
    add Checkbox with {
      set data = "Medien";
    } at position(2, 1);
    add Checkbox with {
      set data = "Hardware";
    } at position(3, 1);
    add Checkbox with {
      set data = "Software";
    } at position(0, 2);
    add Checkbox with {
      set data = "Verlagswesen";
    } at position(1, 2);
  }
  // ***********************************
  // Details
  // Beschriftungen
  create lblPreis as Label {
    set foreground = #FFFFFF;
    set data = "Preis";
  }
  create lblZielgruppe as Label {
    set foreground = #FFFFFF;
    set data = "Zielgruppe";
  }
  // Preis
  create pnlPreis as Panel {
    set background = #CCCCCC;
    add Radiobutton with {
      set data = "< 20";
      set group = grpPreis;
```

```
    } at position(0, 0);
    add Radiobutton with {
      set data = "20 - 30";
      set group = grpPreis;
    } at position(1, 0);
    add Radiobutton with {
      set data = "30 - 40";
      set group = grpPreis;
    } at position(2, 0);
    add Radiobutton with {
      set data = "> 40";
      set group = grpPreis;
    } at position(3, 0);
    connect to lblPreis;
  }
  // Geschlechtswähler
  create pnlZGeschlecht as pnlGeschlecht {}
  // Zielgruppe
  create pnlZielgruppe as Panel {
    set background = #CCCCCC;
    add Radiobutton with {
      set data = "< 14";
      set group = grpZielgruppe;
    } at position(0, 0);
    add Radiobutton with {
      set data = "14 - 16";
      set group = grpZielgruppe;
    } at position(1, 0);
    add Radiobutton with {
      set data = "16 - 18";
      set group = grpZielgruppe;
    } at position(2, 0);
    add Radiobutton with {
      set data = "> 18";
      set group = grpZielgruppe;
    } at position(3, 0);
    add pnlZGeschlecht at position(0..1, 1);
    connect to lblZielgruppe;
  }
  // Detailpanel
  create pnlDetails as Panel {
    set background = #808080;
    set aligning = center;
    add lblPreis at position(0, 0);
    add lblZielgruppe at position(0, 1);
    add pnlPreis at position(1, 0);
    add pnlZielgruppe at position(1, 1);
  }
  // ***********************************
  // Gesamtpanel
  create main pnlGUI as Panel {
    set aligning = northwest;
    set foreground = #000000;
    set foreground = #000000;
    set background = #FFFFFF;
    set errorcolor = #FF0000;
```

```
    set inputbackground = #FFFFFF;
    set inputforeground = #000000;
    add lblHeader1 at position(0, 0);
    add pnlAnsprechpartner at position(1, 0);
    add lblHeader2 at position(2, 0);
    add pnlProdukt at position(3, 0);
    add lblHeader3 at position(4, 0);
    add pnlZuordnung at position(5, 0);
    add lblHeader4 at position(6, 0);
    add pnlDetails at position(7, 0);
  }
} // end of: define view as
// ***************************************************
// Statisches Verhalten
define static behaviour as {
  // Geschlechtswähler
  if (pnlAGeschlecht == "weiblich") {
    cboAnrede hasToBe = "Frau" otherwise {
      error("Anrede und Geschlecht passen nicht");
    }
  } else {
    cboAnrede hasToBe = "Herr" otherwise {
      error("Anrede und Geschlecht passen nicht");
    }
  }
  // required-Felder
  set txtNachname to required otherwise {
    error("Der Nachname muß eingegeben werden.");
  }
  set txtVorname to required otherwise {
    error("Der Vorname muß eingegeben werden.");
  }
  set txtPName to required otherwise {
    error("Der Produktname muß eingegeben werden.");
  }
  set txtPDatum to required otherwise {
    error("Das Produktdatum muß eingegeben werden.");
  }
  set txtPBeschreibung to required otherwise {
    error("Die Produktbeschreibung muß eingegeben werden.");
  }
  set pnlPreis to required otherwise {
    error("Ein Preis muß angegeben werden.");
  }
  set pnlZielgruppe to required otherwise {
    error("Die Zielgruppe muß angegeben werden.");
  }
  // Telefon
  if (isEmpty(txtTelefon)) {
    warning("Die Angabe einer Telefonnummer wäre hilfreich");
  }
  // Geburtsdatum
  txtGeburtsdatum hasToBe < Date("now") otherwise {
    error("Das Geburtsdatum sollte in der Vergangenheit liegen.");
  }
```

```
  // Produktdatum
  txtPDatum hasToBe <= Date("now") otherwise {
    error("Das Produktdatum liegt in der Zukunft.");
  }
  if ((txtPDatum > Date("now")) && (txtPDatum < txtGeburtsdatum)) {
    error("Das Produktdatum muß zwischen Geburtsdatum
            und heute liegen");
  }
  // Zuordnung
  set pnlZuordnung to required(3) otherwise {
    error("Es müssen mindestens 3 Zuordnungen angegeben werden.");
  }
}
// ***************************************************
// Dynamisches Verhalten
define dynamic behaviour as {
  // Webseite
  if (pnlJaNein == "ja") {
    set txtWebtitel.enabled = true;
    set txtWebtitel.text = txtPName.text;
  } else {
    set txtWebtitel.enabled = false;
    set txtWebtitel.text = "";
  }
}
```

B Quellcode der Java-Definitionen

B.1 Schnittstellen

Die Erläuterung und ausführliche Beschreibung der Schnittstellen findet sich in Kapitel 3.4.

B.1.1 EntryElement

```
package de.ekkart.diplomarbeit.interfaces;
import java.util.*;
import de.ekkart.diplomarbeit.exceptions.*;
/**
* Interface für ein Eingabeelement, das in die Eingabemaske eingebettet wird.
*
* @author Ekkart Kleinod
* @version 1.0
* @since 1.0
*/
public interface EntryElement {
  /**
  * Weist den Eingabefeldern neue Farbwerte zu.
  * <br>
  * Der Wert besteht aus einem Color-Array mit zwei Einträgen,
  * Nr. 1 für Vordergrund-, Nr. 2 für Hintergrundfarbe.
  *
  * @param hshNewValues java.util.Hashtable neue Farbwerte
  * @throws MethodNotSupportedException wenn die Methode nicht unterstützt
  * wird (z. B. reines Beschriftungsfeld)
  */
  void changeFieldColor(Hashtable hshNewValues)
    throws MethodNotSupportedException;
  /**
  * Weist den Eingabebeschriftungen neue Farbwerte zu.
  * <br>
  * Der Wert besteht aus einem Color-Array mit zwei Einträgen,
  * Nr. 1 für Vordergrund-, Nr. 2 für Hintergrundfarbe.
  *
  * @param hshNewValues java.util.Hashtable neue Farbwerte
  * @throws MethodNotSupportedException wenn die Methode nicht unterstützt
  * wird (z. B. reines Eingabefeld)
  */
  void changeTextColor(Hashtable hshNewValues)
    throws MethodNotSupportedException;
  /**
  * Weist den Eingabebeschriftungen neue Texte zu.
  *
  * @param hshNewValues java.util.Hashtable neue Texte
```

```
  * @throws MethodNotSupportedException wenn die Methode nicht unterstützt
  * wird (z. B. reines Eingabefeld)
  */
  void changeTextTexts(Hashtable hshNewValues)
    throws MethodNotSupportedException;
  /**
  * Holt die eingegebenen Werte aus dem Eingabefeld und
  * gibt sie in einer Hashtable zurück.
  * <br>
  * Die Werte sind als Werte in der Hashtable gespeichert, die
  * entsprechenden Schlüssel sind die Eingabefeldbezeichnungen.
  * <br>
  * Komplexere Eingabefelder können auch komplexere Objekte zurückgeben.
  *
  * @return java.util.Hashtable Wertehashtable mit den aktuell
  * eingegebenen Werten
  * @throws MethodNotSupportedException wenn die Methode nicht unterstützt
  * wird (z. B. reines Beschriftungsfeld)
  */
  Hashtable getValues() throws MethodNotSupportedException;
  /**
  * Schreibt die in der Parameterhashtable übergebenen
  * Werte in die Eingabefelder.
  * <br>
  * Auch hier können komplexere Eingabestrukturen komplexere
  * Datentypen erfordern.
  * <br>
  * Nicht vorhandene Schlüssel werden als Nichtveränderung
  * des Wertes angesehen.
  *
  * @param hshNewValues java.util.Hashtable neue Werte der Eingabefelder
  * @throws MethodNotSupportedException wenn die Methode nicht unterstützt
  * wird (z. B. reines Beschriftungsfeld)
  */
  void putValues(Hashtable hshNewValues) throws MethodNotSupportedException;
  /**
  * Setzt die in der Parameterhashtable übergebenen Fehlerstati.
  * <br>
  * Mögliche Werte:
  * <ul>
  *   <li>"error" == Fehlerzustand</li>
  *   <li>"warning" == Warnungszustand</li>
  *   <li>"normal" == Normalzustand</li>
  * <ul>
  * @param hshNewStates java.util.Hashtable neue Fehlerstati
  */
  void setErrorState(Hashtable hshNewStates);
}
```

B.1.2 EntryForm

```
package de.ekkart.diplomarbeit.interfaces;
import java.io.*;
/**
* Interface für die Eingabemaske, die in die Anwendung eingebettet wird.
*
* @author Ekkart Kleinod
* @version 1.0
* @since 1.0
*/
public interface EntryForm extends EntryElement {
  /**
  * Liest die GUI-Beschreibung aus der Datei
```

```
  * <code>fleDescription</code> ein und generiert
  * die Eingabemaske.
  *
  * @return true == alles ok;
  * false == I/O-Fehler oder Problem bei der GUI-Generierung
  * @param fleDescription Datei mit der GUI-Beschreibung
  */
  boolean generateMask(File fleDescription);
}
```

B.1.3 FormChecker

```
package de.ekkart.diplomarbeit.interfaces;
import java.io.*;
import java.util.*;
/**
* Interface für die Bereitstellung der Prüffunktionalität.
*
* @author Ekkart Kleinod
* @version 1.0
* @since 1.0
*/
public interface FormChecker {
  /**
  * Prüft die eingelesenen Prüfbedingungen für das
  * angegebene Eingabeformular.
  *
  * @return true == alles ok; false == Fehler bei der Prüfung
  * @param frmDatasource Eingabeformular, dessen Daten geprüft
  * werden sollen
  * @param hshErrors Fehlermeldungen des Prüfvorgangs
  */
  boolean checkForm(EntryForm frmDatasource, Hashtable hshErrors);
  /**
  * Liest die Prüfbedingungen aus der Datei
  * <code>fleCheckdata</code> ein.
  *
  * @return true == alles ok;
  * false == I/O-Fehler
  * @param fleDescription Datei mit den Prüfbedingungen
  */
  boolean readCheckData(File fleCheckdata);
}
```

B.2 Ausnahmen

Die Erläuterung und ausführliche Beschreibung der Ausnahme findet sich in Kapitel 3.5.

METHODNOTSUPPORTEDEXCEPTION

```
package de.ekkart.diplomarbeit.exceptions;
/**
* Ausnahme, die benutzt wird, wenn eine Klasse eine
* Methode nicht unterstützt, diese jedoch aus wichtigen
* Gründen zur Verfügung stellen muß.
*
* @author Ekkart Kleinod
* @version 1.0
* @since 1.0
*/
public class MethodNotSupportedException extends Exception {
  /**
  * Default Konstruktor.
  */
  public MethodNotSupportedException() {
  super();
  }
  /**
  * Konstruktor, der <code>s</code> als Fehlernachricht speichert.
  * @param s Fehlernachricht
  */
  public MethodNotSupportedException(String s) {
  super(s);
  }
}
```

C Abkürzungen und Glossar

ASCII American Standard Code for Information Interchange
Der ASCII (auch ASCII-Code genannt) ist eine Codierungsvorschrift, die 128 Zeichen je eine Zahl von 0 bis 127 zuordnet. Dies entspricht binär der Nutzung von 7 Bit. Damit ist es möglich, auf allen Computern der Welt 128 Zeichen in gleicher Weise anhand des Codes zu identifizieren. Das 8. Bit, das Paritätsbit, dient zur Überprüfung des Byte-Werts.

Leider ist der Standard hauptsächlich für englischsprachige Rechner gültig, andere Sprachen und die Erweiterung auf 8 Bit Daten und der damit verbundene Wegfall des Paritätsbits ergeben Differenzen zwischen verschiedenen Codierungen. Das normale Alphabet, die Zahlen und die wichtigsten Sonderzeichen sind jedoch einheitlich kodiert.

Ausnahme
siehe *Exception*

BNF Backus-Naur-Form
Die BNF ist eine Darstellungsart für die Syntax formaler Sprachen, die durch Normierung der zur Verfügung stehenden Zeichen und Ausdrucksmöglichkeiten eine formale Darstellung der Sprache ermöglicht. Kontextfreie Grammatiken werden bei Benutzung der BNF durch Angabe von Ersetzungsregeln beschrieben. Diese Ersetzungsregeln sind in einer Metasprache kodiert, deren Aufbau den kontextfreien Chomski-Grammatiken entspricht.

Die Praktikabilität und die einfache Lesbarkeit (es ist möglich, BNF-Grammatiken sowohl textuell als auch graphisch mittels Syntaxdiagrammen anzugeben) der Backus-Naur-Form erlaubte ihr eine starke Verbreitung bei der Definition formaler Sprachen, denn durch diese einfache Möglichkeit der Beschreibung kann die Korrektheit von Programmen mittels Automaten überprüft werden.

CASE Computer Aided Softwareengineering
CASE bedeutet soviel wie computergestützte Softwareentwicklung und steht für Werkzeuge und Techniken, die die Softwareentwicklung in allen Phasen (siehe Abbildung 1.3) unterstützen sollen.

Dabei werden die entstehenden Dokumente verwaltet, versioniert, angefordert und ihre Erstellung unterstützt.

Compilierer, Compiler
Ein Compilierer hat die Aufgabe, Quelltexte einer Programmiersprache, die für Menschen lesbar sind, in Code umzuwandeln, der entweder direkt ausgeführt oder von einem Interpreter gelesen werden kann. Den Vorgang der Umwandlung nennt man Compilierung.

Der erhaltene compilierte Code wird zunächst zwischengespeichert und danach ausgeführt.

siehe auch *Interpreter*

Deterministischer Programmablauf
Mit Determinismus wird die Eigenschaft eines Programms bezeichnet, in derselben Situation gleich zu reagieren. Diese Forderung bezieht sich sowohl auf die Eingaben des Nutzers als auch auf die situative Umgebung des Programmablaufs.
Damit wird gewährleistet, daß sich dem Nutzer oder Tester bei gleicher Eingabe gleiches Programmverhalten bietet.

EBNF Erweiterte Backus-Naur-Form
Die EBNF erweitert die BNF um mehrere Sprachkonstrukte, die praxisnahere Definitionen von kontextfreien Grammatiken ermöglichen.

siehe auch *BNF*

Exception, Ausnahme
Exceptions bieten dem Programmierer die Möglichkeit, unvorhergesehene oder bewußte Störungen im Programmablauf abzufangen und gezielt darauf zu reagieren.

Oft treten im Programmlauf Situationen auf, die vom Programmierer nicht explizit berücksichtigt wurden oder die zu einer allgemeinen Klasse von Problemen gehören, auf die ähnlich reagiert werden kann. Für diesen Fall sind Exceptions vorgesehen, die vom Programmierer behandelt werden müssen, das heißt, es obliegt ihm, in angemessener Weise darauf zu reagieren. Unterläßt er dies, wird die Ausnahmebedingung zum Programmabbruch führen.

Frontend
Mit Frontend wird der Teil eines Systems bezeichnet, der direkt mit dem Benutzer interagiert. Bei Programmen ist dies meist die Benutzeroberfläche (siehe *GUI*).

Zu jedem Frontend existiert auch ein Backend, das die Funktionalität des Systems realisiert und dessen Ergebnisse dem Frontend wieder zur Verfügung stellt.

GUI Graphical User Interface
Das GUI wird auch als Nutzeroberfläche, Nutzerschnittstelle, Benutzeroberfläche oder Frontend bezeichnet.

Dabei umfaßt das eigentliche GUI lediglich den Teil eines Programms, der die Verbindung zwischen Nutzer und System auf graphische Weise realisiert. Die meisten modernen GUIs bestehen aus Fenstern, Buttons und Eingabefeldern, die mittels Maus oder Tastatur bedient werden.

GUI-Builder
Ein GUI-Builder ist ein Werkzeug der Softwareentwicklung, das es ermöglicht, graphische Benutzeroberflächen (siehe *GUI*) auf graphischem Weg zu entwickeln. Neuere Builder realisieren bereits Verknüpfungen der Oberfläche mit Funktionen des Systems.

IDE Integrated Development Environment
Eine IDE wird auch als integrierte Entwicklungsumgebung bezeichnet und ist ein Werkzeug der Softwareentwicklung, das bestimmte Aspekte des Programmierens (Schnittstellenprogrammierung, allgemeine Programmierung, Debugging usw.) in einem universellen Werkzeug zusammenfaßt.

Damit wird dem Programmierer eine einheitliche Möglichkeit gegeben, die verschiedenen Aspekte des Programmierens umzusetzen.

Interface, Schnittstelle
Ein Interface bezeichnet die Verbindung eines Objekts zu seiner Außenwelt. Bei Programmen ist dies z. B. das GUI, das die Verbindung zwischen Benutzer und Programm darstellt. Bei Klassen ist die äußere Sicht einer Klasse gemeint, die festlegt, was die Klasse leisten kann (Methoden) und welche Daten die Klasse zur Verfügung stellt (Attribute).

Interpreter
Ein Interpreter hat die Aufgabe, Quelltexte oder Bytecode so umzuwandeln, daß der entstehende Code sofort ausgeführt werden kann. Die Umwandlung findet dabei im Arbeitsspeicher statt, die interpretierte Datei wird also nicht verändert oder abgespeichert.

siehe auch *Compilierer*

Java
Java ist eine objektorientierte Programmiersprache, die in den neunziger Jahren von SUN entworfen wurde. Java kombiniert die knappen und effizienten Ausdrucksmöglichkeiten von C bzw. C++ mit modernen objektorientierten Möglichkeiten.

Java ist als Zielsprache für das Konzept besonders interessant, da mit demselben Programmcode ohne Neuübersetzung des Programms unterschiedliche Zielplattformen angesprochen werden können. Nachteil von Java ist u. a. die noch fehlende Schnelligkeit.

Parser
Ein Parser hat die Aufgabe, einen Quelltext in die einzelnen syntaktischen Elemente einer Programmiersprache zu zerlegen. Danach werden sowohl der Quelltext gegen die der Programmiersprache zugrundeliegende Grammatik überprüft, als auch eventuell aufgetretene Syntaxfehler angezeigt. Dieser Vorgang wird als syntaktische Analyse bezeichnet.

Prototyp / Prototyping
Als Prototyp wird in der Softwaretechnik ein schnell hergestelltes Produkt bezeichnet, das keine oder nur wenig Funktionalität enthält und zur Demonstration bzw. Anforderungsverfeinerung gedacht ist.

Oft ist es auch üblich, nur die Oberfläche und Bedienung eines Programms zu entwickeln und als Prototyp dem Auftraggeber zu übergeben.

Schnittstelle
siehe *Interface*

XML Extended Markup Language
XML ist eine Sprache, die es ermöglicht, Dokumente strukturiert darzustellen. Dabei geht XML (als Untermenge von SGML) davon aus, daß Dokumente eine innere Struktur besitzen. Diese Struktur wird extrahiert und eine sogenannte DTD (*document type definition*) gebildet.

Nun können alle Dokumente, die vom Typ der DTD sein sollen, nach dieser strukturiert werden. Damit ist es möglich, die Dokumente computerisiert zu bearbeiten.

D Literaturangaben

[1] ALBER, Werner: *Einführung in die Semantik von Programmiersprachen*. Mannheim; Wien; Zürich: BI-Wissenschafts-Verlag, 1988 (Reihe Informatik; Bd. 59)

[2] BALZERT, Helmut: *Lehrbuch der Software-Technik: Software-Entwicklung*. Heidelberg; Berlin; Oxford: Spektrum Akademischer Verlag, 1996

[3] BALZERT, Helmut: *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Heidelberg; Berlin; Oxford: Spektrum Akademischer Verlag, 1998

[4] DOBERKAT, Dietmar: *Praktischer Übersetzerbau*. Stuttgart: B. G. Teubner, 1990

[5] FEHR, Elfriede: *Semantik von Programmiersprachen*. Berlin; Heidelberg; New York; London; Paris; Tokyo: Springer, 1989 (Studienreihe Informatik)

[6] FISCHER, Frances S.: *The anatomy of programming languages*. Englewood Cliffs: Prentice Hall, 1993

[7] HÜBNER, Wolfgang: *Entwurf graphischer Benutzerschnittstellen: ein objektorientiertes Interaktionsmodell zur Spezifikation graphischer Dialoge*. Berlin; Heidelberg; New York; London; Paris; Tokyo; Hong Kong; Barcelona: Springer, 1990 (Beiträge zur graphischen Datenverarbeitung)

[8] HOFMANN, Frank: *Grafische Benutzeroberflächen: Generierung aus OOA-Modellen*. Heidelberg; Berlin; Oxford: Spektrum Akademischer Verlag, 1998 (Forschung in der Softwaretechnik)

[9] HOPCROFT, John E.: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. 3. korrigierte Auflage. Bonn; Paris; Mass.: Addison-Wesley, 1994

[10] PRATT, Marvin: *Programmiersprachen: Design und Implementierung*. München [i. e.] Haar; London; Mexiko; New York; Singapur; Sydney; Toronto: Prentice Hall, 1997

[11] ROZENBERG, Arto (Hrsg.): *Handbook of formal languages: Vol. 1 word, language, grammar*. Berlin; Heidelberg; New York: Springer, 1997

[12] SCHNEIDER, Hans-Jochen (Hrsg.): *Lexikon der Informatik und Datenverarbeitung*. 4. Auflage. München; Wien; Oldenbourg: R. Oldenbourg Verlag, 1998

[13] SETHI, Ravi: *Programming languages: Concepts and constructs*. Addison-Wesley, 1989

[14] WERDER, Lutz von: *Kreatives Schreiben von Diplom- und Doktorarbeiten*. 2. Auflage. Berlin, Milow: Schibri-Verlag, 1998

[15] ZIMA, Hans: *Compilerbau II: Synthese und Optimierung*. Mannheim; Wien; Zürich: Bibliographisches Institut, 1983 (Informatik Band 37)

[16] ZIMA, Hans: *Compilerbau I: Analyse*. 2. Auflage. Mannheim; Wien; Zürich: BI Wissenschaftsverlag, 1989 (Informatik Band 36)

E Index

A

Ableitungsbäume 31
Auslieferungspaket 46
Ausnahmen 54

B

Backus-Naur-Form . . . . . *siehe* BNF
Benutzeroberfläche . . . . . *siehe* GUI
Bewertung 69
  Anforderungen 70
  Nutzen 78
  Problemstellung 70
  Sprache 73
BNF 33

C

C 20
C++ 20
Chomski Hierarchie 30
Chomski-Normalform 33

D

Datenaustausch 49, 66
Delphi 22, 74

E

EBNF 34
  Konventionen 57
Endliche Automaten 27
Erweiterte BNF . . . . . *siehe* EBNF
Exceptions . . . . . *siehe* Ausnahmen

F

Frontend 3

G

Grammatik 29
  eingeschränkte (Typ 0) 30
  kontextfreie (Typ 2) 30
  kontextsensitive (Typ 1) 30
  reguläre (Typ 3) 30
GUI 2, 5, 8, 46, 67
GUI-Builder 3, 11
GUI-Generator 23, 74

I

Implementationsprozeß 45

J

jar 66
Java 19, 74

K

Kellerautomaten 30

L

Linksableitung 32
LL-Analyse 32
LR-Analyse 32

M

Modellierungsmethodik 11

O

Objektorientierung 7

P

Prüfroutinen 2, 5, 8, 46, 67
Problem 2
Problemstellung 5
Produktarchitektur 12
Produktmodell 11
Programmiersprache 4
Programmrumpf 7, 46, 67

R

Rechtsableitung 32
Reguläre Ausdrücke 27

S

Schnittstellen 7, 8, 15, 46

externe 46, 52
interne 46, 49
Prüfroutinen 53
Semantik . *siehe* Sprachen, Semantik
Softwareentwicklung
Ablauf 8
Phasen 9, 78
Analyse 10
Betrieb 14
Entwurf 11
Programmierung 12
Softwareanforderungen . . 10
Systemanforderungen 10
Testen 13
Prozeß 8
Stufen . . *siehe* Softwareentwicklung, Phasen
Softwareentwurf . . . . . *siehe* Softwareentwicklung
Softwareherstellungsprozeß 4
Sprachen
Attribute 39
Designkriterien 37
entwickelte 55
Definition 58
Semantik 59
Syntax 59
Token 59
formale 7, 15
Definition 26
Verwendung 6
Grundlagen 25
Anforderungen 25
Rolle von 37
Semantik 34
Definitionsmethoden 35
dynamische 34
statische 34
Syntax 28
Token 27
Standardverhalten 48
Status Quo 18
syntaktische Analyse 28
Syntax . . . . . *siehe* Sprachen, Syntax
Syntaxdiagramme 34

T

Token . . . . . *siehe* Sprachen, Token
Typ 0 . . . . . *siehe* Grammatik
Typ 1 . . . . . *siehe* Grammatik
Typ 2 . . . . . *siehe* Grammatik
Typ 3 . . . . . *siehe* Grammatik

V

Validation 13
Verifikation 13
Visual Basic 21, 74

W

Wasserfallmodell 8–10
Windows 24, 74

Z

Zielsprache 4, 12, 44, 74

Erklärungen

Eigenständigkeitserklärung

Ich versichere durch eigenhändige Unterschrift, daß die Arbeit von mir selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde.

Berlin, den 8. Mai 2001

Ekkart Kleinod

Einverständniserklärung

Ich erkläre mich damit einverstanden, daß ein Exemplar meiner Diplomarbeit in der Institutsbibliothek der Humboldt-Universität zu Berlin aufgestellt wird.

Berlin, den 8. Mai 2001

Ekkart Kleinod